Protocol Labs

Drand Security Assessment

Version: 1.1

August, 2020

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 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 the following commits:

- `drand/drand` : 698cb89 (Release v0.9.1)
- `drand/bls12-381` : 15b1036 (Release v0.3.2)
- `drand/kyber` : daa30f0

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:

- `blsdiffg1add`
- `blsdiffg1mul`
- `blsdiffg1multiexp`
- `blsdiffg2add`
- `blsdiffg2mul`
- `blsdiffg2multiexp`
- `blsdiffmapg1`
- `blsdiffpairing`
- `ecieskey`
- `eciesmsg`
- `privaterandreqserver`
- `tblsandblsverify`
- `unmarshalg1key`

The targets beginning with “blsdiff” perform differential fuzz testing, comparing between the `drand/bls12-381` implementations of finite field arithmentic, optimized for different architectures.

These fuzzing targets have been shared with the development team.

`drand/bls12-381` is forked from kilic/bls12-381, which will potentially be introduced in the go-ethereum implementation of the Ethereum Virtual Machine as part of EIP2537. The testing team adapted test vectors created for EIP2537 and applied them to `drand/bls12-381`.

Along with fuzzing activities, the testing team deployed a customised testnet environment on a dedicated cloud infrastructure to demonstrate and illustrate some of the findings of this review.

The testing team identified a total of twenty-one (21) issues during this assessment, of which:

- Two (2) are classified as high risk,
- Four (4) are classified as medium risk,
- Eight (8) are classified as low risk,
- Seven (7) are classified as informational.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within Drand. 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 the affected code as been updated, or relevant controls implemented, 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
RND-01	Valid Partial Signatures can be Rebroadcasted to Perform DoS During Randomness Generation	High	Resolved
RND-02	Replay Attack in DKG Can Lead to Arbitrary Node Eviction	High	Resolved
RND-03	Rogue Key Attack on DKG	Medium	Resolved
RND-04	Incorrect *NoADX* Field Arithmetic `Mul` Result	Medium	Resolved
RND-05	Non-constant Time Cryptographic Primitives	Medium	Resolved
RND-06	Fields Not Included in `Group.Hash()`	Low	Resolved
RND-07	Clients Do Not Automatically Verify That Randomness is Unbiased	Low	Resolved
RND-08	Local, Unprivileged Users can Control Drand	Low	Resolved
RND-09	Broadcast assumption does not Guarantee Packet Order	Low	Resolved
RND-10	Lack of Complexity Requirements on `secret` Parameter	Low	Resolved
RND-11	Preshared DKG `secret` Can Theoretically be Exposed via a Timing Attack	Low	Resolved
RND-12	Pre-shared DKG Secret is Passed Sub-Optimally as a CLI Parameter	Low	Resolved
RND-13	Incorrect SWU Constant	Low	Resolved
RND-14	Insufficient Check on Number of Complaints	Low	Resolved
RND-15	Deserialisation of Points is not Strict on Bytes Length	Informational	Open
RND-16	Unused Constants in BLS Library	Informational	Resolved
RND-17	Usage of out-of-date Golang Packages	Informational	Resolved
RND-18	Locally Triggered Deadlock of Leader's Drand Daemon	Informational	Resolved
RND-19	Unused/Dead Code	Informational	Resolved
RND-20	Miscellaneous Observations on BLS Library	Informational	Closed
RND-21	Miscellaneous Observations on Drand Primary Codebase	Informational	Closed

RND-01	Valid Partial Signatures can be Rebroadcasted to Perform DoS During Randomness Generation		
Asset	`drand/beacon/chain.go` , `drand/beacon/handler.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

During randomness generation, Drand nodes exchange `PartialBeaconPacket` messages (a.k.a *"partials"*) that are combined to produce distributed randomness beacons. By flooding a node with correctly signed `PartialBeaconPacket` messages for the current round, a malicious actor can consume all available memory and CPU time. By modifying the number of packets sent per round, an attacker can choose to either extend processing time such that the node often fails to produce new partial beacons within the round period, or crash the Drand daemon entirely due to memory overconsumption.

When a sufficient number of nodes are attacked (no. targets $x = n - t + 1$), the network is rendered inoperable; unable to produce new beacons.

To perform this attack, the malicious actor must have timely access to correctly signed `PartialBeaconPacket` messages containing the current or next round number. As such, the most likely scenario would involve the attacker having control over a corrupted, malicious node or having obtained that node's private share. However, because the packets do not have to be sent by the node to be accepted as valid (i.e. the peer address does not have to match the address of the group member), it is also sufficient for the attacker to eavesdrop packets e.g. if TLS is disabled at some stage of transmission or via MITM with a valid certificate from a compromised CA.

In addition, because the sending peer does not have to be the originating Drand node (or any node), an attacker can leverage multiple devices to perform a DDoS (distributed denial-of-service) attack.

The actual vulnerability is primarily associated with:

- `roundCache.tryAppend()` not differentiating between "not appending because an exact duplicate entry is present" and "not appending because the partial is not relevant for that `roundCache` ".

  As such, processing a partial that is the exact duplicate of one already received results in a new `roundCache` being created.
- accepting an unlimited number of distinct `PartialBeaconPacket` s per node, per round (valid signatures with different `PreviousSig` values).
- adding to the `roundCache` s partials from a round before the current finalized Beacon.

See Appendix A for a more detailed explanation of the technical issues and relevant evidence.

Recommendations

Consider actioning the following:

- Drop immediately (preferably before even signature verification) any `PartialBeaconPacket` messages with a `Round <= lastBeacon.Round()`, where `lastBeacon` is the `Beacon` most recently added to the `chainStore` (or approximate).

  Note: it is likely preferable to perform this check twice—during `(*Handler).ProcessPartialBeacon` and `(*chainStore).runAggregator`.

- Do not accept/store partials that are exactly the same as others already seen (i.e. same `Round` and `PreviousSig`).

  Possible solutions include modifying `roundCache.tryAppend` to return a different value if that exact partial is already present. Alternatively, the caches could be based on a map instead of a slice, keyed by `concat(round, previousSig)`.

- Ensure each valid partial is included in, at most, a single relevant `roundCache` entry.

  i.e. within the caches, `roundCache` entries should be uniquely identifiable by `round + previousSig`

- Consider dropping protocol messages from peers who are not part of the group or, if reasonable, require partials to be delivered only by the peer with the address associated with that `Sig index`.

- Be careful before storing partials for future usage e.g. if relaxing the "invalid_future_round" check in `beacon/handler.go`, as the attacker can then add more long-lived partials to the `roundCache`s

- Consider enabling a configuration where Protocol messages have a dedicated port/listener.

  This could more easily facilitate firewall rules that whitelist access to the Protocol port while allowing public access to the public GRPC API.

  We note an alternative approach is currently possible. The same effect can be achieved by restricting access to only other nodes and trusted relayers, which then expose the public API. *Consider documenting this as a desirable configuration.*

- To protect against a malicious node populating the caches with multiple distinct partials (created by signing a `beacon.Message` composed of the relevant round and some arbitrary `prevSig`), consider keeping some limited number of partials per round per node/index, or only the most recently created one (if such ordering is tracked).

  In certain circumstances, a correct node can reasonably broadcast more than 1 partial for a single round, for example when reconnecting after a network partition.

  - By associating a monotonically increasing nonce or timestamp with each partial signature, nodes can indicate which signatures they want kept when more than 1 partial is received for a round. However, to stop this being forged, it would likely require an additional signature around the whole `PartialBeaconPacket + nonce`.
  - It appears less complicated to limit the number of partials that are accepted per node, per round. The testing team have not identified a scenario where a non-malicious node would ever sign 10 distinct partials for a single round.

- This malicious behavior can be readily identified and attributed to a corrupted node, provided relevant metrics or logging. Consider adding a metric tracking partial submissions per node or index.

- Be careful before emitting high-severity, *"enabled by default"* logs that a malicious actor is able to trigger on demand.

  A flood of these logs may introduce IO bottle-necks, particularly when the `SyncWriter` used by default in `(drand/drand/log).NewLogger` can cause logging calls to block the goroutine.

  In this case, large-scale rebroadcasts emit similar numbers of `"ignoring_partial"` logs (defined at `chain.go:145`). See Appendix A for an example.

  Other resolutions may mitigate this particular example, so no action may be required in this case.

Resolution

While issues identified in the initial review had been fixed, subsequent testing found Drand nodes to still be susceptible to denial-of-service from a malicious peer flooding it with duplicate partials. Further troubleshooting identified additional issues, which were promptly and effectively rectified by the development team.

With regards to remediation of initially identified issues:

The `roundCache` implementation has been largely reworked in PR #543, resolving previously identified issues.

- Partials are now uniquely associated with a single cache entry via its `Round` and `PreviousSig` fields. As such, duplicate partials no longer result in the creation of additional `roundCache` entries.
- The cache will only store a limited number of unique partials for each group member (`MaxPartialsPerNode = 100`). Because of this, individual malicious nodes can no longer cause the cache to expand indefinitely and exhaust available resources.
- The cache now stores entries in a map rather than a list. Because of this, lookup and removal operations are no longer performed in linear time relative to the size of the cache.
- High severity logs are no longer emitted when a duplicate partial is processed. This protects against an attacker exploiting logging overheads to exhaust available resources.

Retesting was performed using the same experimental methodology, and confirmed that CPU and memory consumption now remains stable when a node is subject to a flood of duplicate `PartialBeaconPacket` messages.

Risks from a DDoS-style attack are effectively mitigated by the standard operator guidelines (provided to node operators), which recommend external access to the Drand node be restricted to only other Drand nodes and trusted relayers (e.g. via IP-based whitelisting). Because the node is not exposed to the internet, there is a limited number of devices that an attacker can use to flood nodes with traffic.

Remaining Vulnerability

During retesting, the experimental network was found to still be susceptible to DoS by a single, malicious node rebroadcasting duplicate partials. Either the cause of the DoS had shifted, or the increased memory and CPU consumption was previously masking the source of the vulnerability.

Further troubleshooting identified that the global state lock (`(*Drand).state`) was held for the entirety of initial partial verification in `(*Drand).PartialBeacon` (defined in `drand/core/drand_public.go:39`)). As such, all signature validation is performed synchronously and, should `(*Handler).ProcessPartialBeacon()` [1] block because the `chainStore.newPartials` channel buffer is full[2], everything requiring the state lock will also block. When the flood of partials exceeds manageable limits, GRPC calls time-out and the network fails to aggregate new beacons. Because `SyncChain()` and `PublicRand()` also require the state lock, targeted nodes cannot adequately assist with network recovery or respond to client requests for randomness.

See Appendix C for more details.

[1] Defined at `drand/chain/beacon/node.go:92`.

[2] As part of the call to `(*chainStore).NewValidPartial()` defined at `drand/chain/beacon/chain.go:72`.

Mitigation of Remaining Vulnerability

Contention on the state lock was fixed by PR #732, so a flooded node is no longer unresponsive to `$ drand stop` and other commands requiring the state lock.

Although duplicate partials are still passed to the `chainStore.newPartials` channel, causing the buffer to quickly fill and block, further testing found this to have minimal impact and the network could still generate beacons while under very heavy attack. This indicates the majority of time waiting on the state lock was attributed to the (now concurrent) signature verification in `ProcessPartialBeacon()`, not the single-threaded processing in `runAggregator()`.

While high rates of flooded partials result in noticeably increased memory consumption, the effect is small enough that an extremely high rate would be required to exhaust available memory (bandwidth appeared to become the limiting factor during retesting). As such, the testing team deems it highly unlikely that, in a *mainnet* scenario, a single node operator would have sufficient egress bandwidth to successfully attack a relevant subset of the network.[3]

[3] Because the standard operator guidelines recommend a protected internal network, where firewalls allow only connections from whitelisted IPs to reach the Drand node, it is protected from a DDoS-style attack

RND-02	Replay Attack in DKG Can Lead to Arbitrary Node Eviction		
Asset	`drand/drand`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

The testing team identified the possible exploitation scenario against the Drand network (assuming `FastSync` is enabled, which is currently the case as per line [118] in `drand_control.go`):

Assuming a node ends up in the same index more than once there is the potential to replay messages between different DKG/resharing iterations:

- If the index being repeated is in the old set of nodes, then `DealBundle` and `JustifBundle` messages may be replayed:
  - A `DealBundle` that has been replayed will cause `delete(deals, newDeal.Bundle.DealerIndex)` to be triggered. The result is that a node's deal will not be processed and thus cannot be included in `QUAL` .
  - If a `JustifBundle` is replayed the existing `JustifBundle` will be overwritten, resulting in the replayed bundle being processed. The result is likely that the node will be evicted for providing an invalid share or index. The `JustifBundle` can be replayed even if no complaints were made about this node, resulting in its eviction. Thus, complaining about a node in a previous round will allow for the collection of a `JustifBundle` and arbitrarily evicting the node in future rounds (under the assumption that their index does not change).
- If the index being repeated is in the new set of nodes, then `ResponseBundle` messages may be replayed:
  - A `ResponseBundle` that is replayed will overwrite any existing `ResponseBundle` from a node. If `len(oldNodes) == len(newNodes)` and the previous round did not include any complaints, a malicious user could resend all the successful `ResponseBundle` from the previous round.

On a side note, when running the DKG with `FastSync` set to false, if a message with a valid signature is sent twice, it will be appended to `deals` in `Protocol.Start()` , resulting in `ProcessDeals()` being processed twice. As such, the following code path will be reached, allowing a malicious actor to arbitrarily evict any node:

```go
if seenIndex[bundle.DealerIndex] {
    // already saw a bundle from the same dealer - clear sign of
    // cheating so we evict him from the list
    d.evicted = append(d.evicted, bundle.DealerIndex)
    continue
}
```

dkg.go

Recommendations

For the case when `FastSync` is enabled, we recommend either:

- Including a timestamp after the DKG start time for all messages exchanged by nodes;
- Using a nonce in all messages for each DKG/reshare iteration.

For the case when `FastSync` is disabled, we recommend using a `Set` for `deals`, `resps` and `justifs` to ensure uniqueness of messages in addition to the recommendation above to prevent replay over multiple iterations.

Resolution

The data types have been updated to use a `Set` rather than slice in commit abb4a98.

A nonce has been added to ensure that messages are unique between rounds in this pull request. The nonce is a hash of the transition time which must be in the future, thereby preventing reply from previous rounds.

RND-03	Rogue Key Attack on DKG		
Asset	`drand/drand`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

During the initial Joint-Feldman DKG of threshold t, each player is required to select a polynomial of degree $t-1$. It is possible to manipulate the final public and private key to any arbitrary key pair, given there are m malicious nodes satisfying $m \geq n-t+2$.

The following is using additive group notation for elliptic curve operations and G as the generator of the group.

During the `Deal` phase one of the malicious nodes, index j, will wait until all other deals have been received. They will then calculate the final public key and set their public key, $a_{j,0}$ to be the negation plus a secret offset, u. That is,

$$a_{j,0} = u - \sum_{i \neq j} a_{i,0}$$

Thus, the final public key y will be,

$$y = \sum[a_{i.0}]G = [a_{j,0}]G + \sum_{i \neq j}[a_{i,0}]G = [u - \sum_{i \neq j} a_{i,0}]G + \sum_{i \neq j}[a_{i,0}]G = [u]G$$

To create the malicious nodes final polynomial $f_j(z)$ they will use two helper polynomials, $h(z)$ and randomly generated $g(z)$ such that,

$$f_j(z) = (z - \sum_{i \neq j} a_{i,0})h(z) + g(z)$$

We define the group of non-malicious polynomials $GOOD$. To produce valid secrets for each of the nodes in $GOOD$ we set,

$$h(k) = 0 \text{ for } k \in GOOD$$

Thus,

$$f_j(k) = (z - \sum_{i \neq j} a_{i,0}) * 0 + g(k) = g(k) \text{ for } k \in GOOD.$$

Since we have selected $g(z)$ the secrets are all known. Additionally, we have the condition $h(0) = 1$ and $g(0) = u$ such that,

$$f_j(0) = (0 - \sum_{i \neq j} a_{i,0}) * 1 + u = a_{j,0}$$

Defining $h(z)$ as,

$$h(z) = b_0 + b_1 * z + b_2 * z^2 + ... + b_{t-2} * z^{t-2}$$

Since we must have $h(k) = 0$ for $k \in GOOD$ and $h(0) = 1$, we have a system of linear equations in b_i. The system of equations has depth $|GOOD| + 1$ and width $t-1$. Hence, there will be a solution if,

$$|GOOD| + 1 \leq t - 1$$

Now by definition,

$$|GOOD| = n - m$$

Thus we have solutions if,

$$m \geq n - t + 2$$

Finally, we may calculate of the public polynomials for node j with coefficients $A_{j,i}$. Let c_i be the co-efficients of $g(z)$. Then we calculate $A_{j,i}$ as,

$$A_{j,i} = [b_{i-1} - \sum_{i \neq j}(a_{i,0}) * b_i]G + [c_i]G$$

Since $[\sum_{i \neq j}(a_{i,0})]G$ can be summed from each $[a_{i,0}]G$, additionally b_i and c_i are known.

Recommendations

DKG should be run with $t < n/2 + 1$. One solution for odd t is to select $t = floor(n/2) + 1$. The number of nodes require to generate a signature will then be t which is greater than 50%. The number of nodes required to break DKG would then be $n - floor(n/2) + 2$.

Resolution

The development team is aware of this issue and have mitigated the risk by adding a warning for users when a high threshold is used.

RND-04	Incorrect *NoADX* Field Arithmetic `Mul` Result		
Asset	`kilic/bls12-381/arithmetic_x86.s`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

The file `drand/bls12-381` library provides 3 finite field arithmetic implementations, which are selected based on CPU architecture:

i) *ADX* - assembly for X86 CPUs with ADX and BMI2 support.

ii) *NoADX* - assembly for X86 CPUs without ADX.

iii) *Generic* - pure Go for other architectures.

While performing differential fuzzing between the implementations, the testing team identified differences between the results of the *NoADX* `G1` `MapToCurve` and the other implementations.

The team confirmed that this was the same issue found during independent testing of the upstream `kilic/bls12-381` library[4], for which the upstream maintainer identified a bug in the multiplication operations.

In a worst-case scenario, inconsistent arithmetic can result in disagreement amongst nodes as to whether a signature is valid. If a threshold number of nodes validate an incorrect signature, the minority can "fork" but fail to generate partial beacons that are accepted as current by the rest of the network, effectively removing their ability to participate and increasing the network's susceptibility to DOS attacks and malicious actors.

Recommendations

We recommend updating the `mulNoADX` and `mulAssignNoADX` implementations to resolve the differences found during testing.

If performance benchmarks indicate an acceptable impact, consider disabling the assembly arithmetic implementations.

Consider introducing regular/automated differential fuzzing between implementations in order to identify regressions.[5]

Resolution

This issue has been addressed in the upstream BLS library (`kilic/bls12-381`) in PR #14.

[4] See `TestFromGethFuzzBlsDiffMapG1Panic` in `fuzz/bls-eip2537-diff_test.go` and https://github.com/ethereum/go-ethereum/pull/21018#issuecomment-630788929

[5] It looks like the upstream maintainer has performed fuzzing here https://github.com/kilic/go-ethereum/blob/fuzz/crypto/bls12381/arithmetic_fuzz.go, but does not appear that fuzzing is yet performed as part of regular maintenance of the upstream library.

RND-05	Non-constant Time Cryptographic Primitives		
Asset	`kilic/bls12-381`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

The cryptographic primitives implemented in the BLS library used by Drand appear to be non-constant time (specifically the scalar multiplication and signing functions), which might result in timing attacks (side-channels), allowing an adversary to attempt retrieving information about exchanged messages and secret keys.

Note: Side-channel exploitation and timing attacks are out-of-the scope of this review.

Recommendations

Consider revamping the cryptographic primitives implemented in the BLS library to ensure and mitigate timing/side-channel attacks.

Resolution

The development team has confirmed that there are no signatures that are generated on demand. Thus, information about the secret key will not be leaked.

RND-06	Fields Not Included in `Group.Hash()`		
Asset	`drand/drand/key/group.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

In the `(*Group).Hash` function, the following `Group` fields are not included in the calculation:

- `Period`
- `GenesisSeed` (when the current group is the result of a Resharing procedure)

This means that when a hash is used to verify the contents of a `Group` (e.g. by the client in `(drand/client/*httpClient).FetchGroupInfo` or during a post-setup verification step), those fields can be modified without altering the verification result.

With regards to the `Period` field and a client initialized with a trusted `groupHash` root (e.g. created with `(drand/client).NewHTTPClient`), a malicious relayer can misreport the `Period` as some integer multiple of the real `Period` (so the client thinks the current round is r but the real round is $\approx k \times r$). This may be exploitable in cases where there is benefit in knowing some amount of "future" results, like a lottery.

Recommendations

Consider including the `Period` and `GenesisSeed` fields in the hash calculation. The latter is only applicable when the Group is a result of a resharing.

Resolution

This issue is partially resolved for `(*httpClient).FetchGroupInfo` as part of PR #507. Renamed `(*httpClient).FetchChainInfo`, this responds with the contents of a dedicated `chain.Info` struct, whose hash appropriately covers all fields.

However, `(*Group).Hash()` still omits the `Period` and `GenesisSeed` fields, which is sub-optimal with regards to verifying group configurations post DKG. As a fix would invalidate all existing group hashes in the current *mainnet* deployment (or otherwise require a complicated hard fork where group hashes are calculated differently after some round), the minuscule residual risk has been deemed acceptable.

RND-07	Clients Do Not Automatically Verify That Randomness is Unbiased		
Asset	`drand/client/http.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

In the event of compromise of more than t (threshold) secret shares or nodes, the following claim is made:

> *The attacker is now able to derive the whole chain, i.e. it can derive any given random beacon of the chain. The drand randomness is not unpredictable anymore from the point of view of the attacker. However, the drand randomness stays* unbiasable: *attacker is not able to change the randomness in any way.* (*https://github.com/drand/drand/blob/master/docs/SECURITY_MODEL.md#corruption-of-the-drand-network* Scenario 2)

While it is true that the attacker cannot modify a beacon (and therefore its randomness) without "disconnecting" it from the chain, clients do not verify that the beacon is part of the chain. When receiving a `PublicRandResponse`, the client verifies that the signature is correct via `(drand/beacon).VerifyBeacon` but not that it is part of the chain (that the `PreviousSignature` is correct). As such, a malicious actor who has control of t shares can generate correctly signed Beacons containing an arbitrary, non-zero[6] `PreviousSignature` (which can be chosen to produce a desirable randomness result) that clients accept.

An independent auditor (given the full chain) could identify that the client's randomness output is incorrect but would only realize this in hindsight, too late to rectify the abuse.

Because an exploit already requires access to t shares, the impact here depends primarily on the use-case and the importance of *unpredictability* versus *unbiasablility*:

- For some, like lotteries, unpredictable results are crucial so this issue does not constitute any additional threat.
- For others, including election auditing and selection from a population, the result's *fairness* can be arguably more important than its unpredictability. Here, this issue introduces sub-optimal security guarantees.

Though not implemented in the current client, it is possible to use the current public API to perform a full, historical validation of a chain without resharings. However, this is not currently possible when a group-member–modifying reshare operation has occurred during generation of the chain. To request and validate the whole chain, the client must know both the `GenesisSeed` (hash of the original group) and the current `GroupHash` (to request the current group config from the network).

[6] This is checked at `http.go:137`

```
if len(randResp.Signature) == 0 || len(randResp.PreviousSignature) == 0 {
  return nil, fmt.Errorf("insufficent response")
}
```

Recommendations

Consider actioning the following:

- Visibly document current client threat model, where biased results can be accepted upon compromise of t nodes. This is likely an acceptable threat model for many, but is preferable to provide users a conscious choice.
- Implement an alternative (higher level?) verification mode where, when enabled, the client verifies the chain. This would necessarily be a stateful client that would be initialized with the `GenesisSeed` and, as a prerequisite before validating the latest round, would validate all previous rounds.

  Several possible optimizations could be beneficial:

  - After a result is fully validated, subsequent rounds can start validation from a previously verified result.
  - After an initial sync from genesis (assuming the majority of requests are for "recent" rounds), the majority of old beacons could be pruned to reduce resource utilization. To provide reasonable bounds when validating requests for older rounds, an iterative *checkpoint-like* pruning method could be used (e.g. All beacons are stored for the last day, then one per hour for the last 3 days, then 1 per day for the past week etc).
  - To avoid a full resync on startup, consider persisting some results to disk.
- Expose an additional public API endpoint (perhaps called `GenesisGroup`) that is similar to the `Group` endpoint but returns info associated with the original group. Because a verifying client only needs to use the network's distributed public key, round period and genesis time—all of which persist across resharings—it is sufficient to provide the genesis group's info.

  This also allows an intializing client to verify the group configuration via `(*httpClient).FetchGroupInfo` without needing to update the `groupHash` after each resharing.

Resolution

This issue has been addressed in PR #600 via the implementation of a `verifyingClient` client type, that verifies the chain from genesis or some trusted beacon checkpoint.

Although this functionality is yet to be explicitly exposed to the reference client CLI (`drand/cmd/client/main.go`), the functionality is easily enabled by client applications.

RND-08	Local, Unprivileged Users can Control Drand		
Asset	`drand/core/drand_control.go` , `drand/protobuf/drand/control.*` , `drand/net/control.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

Apart from the localhost listener, there are no restrictions to the Drand "Control" API. This means any local, unprivileged user (or process with network access) can retrieve the Drand node's secrets, stop it, initiate resharing etc.

It may be acceptable that local users and processes are not part of Drand's threat model. However, this is not addressed in the documentation.

A Drand operator may easily assume that the Drand node is more protected from internal operations than it is.

Recommendations

If part of the acceptable threat model, document clearly that this is the case. Provide best practice documentation recommending that the Drand node be installed on a restricted system, without non-essential users and services.

Consider providing support for binding the control API to a unix socket, which (on linux) can be more easily configured for restrictive permissions (e.g. so only the drand daemon user and the admin user can access the socket)

Alternatively, protect the control API by requiring credential authentication (across a tunnelled connection).

Resolution

Technical improvements have been addressed in PR #582, which allows the control listener to bind to a Unix socket that can be restricted to a subset of local users.

While this functionality appears yet to be documented (e.g. at `https://beta.drand.love/operator`), standard operating guidelines (communicated directly to node-operators) include recommendations that Drand be run on a dedicated machine/VM with relevant access controls. This (and other best-practice guidelines included) appropriately mitigates the risk, minimizing likelihood that that node operators run Drand in an inappropriate environment.

The testing team also notes some public documentation that warns about the sensitive nature of the control API at `https://beta.drand.love/operator/drand-cli/#drand-start`.

RND-09	Broadcast assumption does not Guarantee Packet Order		
Asset	`drand/drand`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Medium

Description

During DKG it is possible for a packet to be received in one phase by one group of nodes and the next phase by other nodes. Packets travelling on the network may be delayed and arrive after the phaser has incremented a round.

For example if certain users receive a `DealBundle` in the `Response` phase after `ProcessDeals()`, it will not be processed by these users. While other nodes on the network may have received it during the `Deal` phase and therefore processed this particular deal. As a result, this will create a situation where some nodes may have evicted (if it was a malicious deal) a player and some have not, creating inconsistencies across the network.

Recommendations

Consider using complaints in addition to evicting malicious actors. This approach requires different types of verifiable complaints (e.g. a complaint `DoubleMessage` which sends both signed messages).

Alternatively, similar to the reliable broadcast assumption, consider manually verifying the results of the DKG / resharing instances.

Resolution

The short term solution is to use a full broadcast channel by rebroadcasting every packet. Attackers are still able to split the network by sending a packet right on an epoch boundary. However, pulling off the attack is unlikely due to the requirement of intimate knowledge of the network topography.

See commit a6e4c31.

The proposed long term solution is to swap to a DKG protocol that is asynchronous. The asynchronous potocol is described in this paper. This solution will overcome the shortfalls described above from using a synchronous protocol.

RND-10	Lack of Complexity Requirements on `secret` Parameter		
Asset	`drand/drand`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Medium

Description

The setup phase leading to the distributed key generation relies on a common secret string shared amongst all participants. Used as a CLI flag (`-secret <value>`), this parameter allows the authentication of nodes authorised to join the network. It is expected to be shared with participants via a separate secure channel, prior to network initialisation.

The Drand node does not enforce any complexity requirements on the `secret` parameter, allowing it to be set to a weak, insecure and/or predictable string, which could be guessed/brute forced by a malicious actor to join the setup phase. A short secret string also increases the likelihood that RND-11 is successfully exploited. This issue is raised with a *low* severity: as mentioned in the Drand documentation, *"nodes can detect if there are some unwanted nodes after the setup and in that case, setup a new network again"*.

Recommendations

We recommend implementing and enforcing strong complexity requirements for the `secret` parameter on the Drand node:

- String must be longer that 12 characters;
- String must comprise at least three of:
  - Lower case characters;
  - Higher case characters;
  - Numbers;
  - Special character.
- String must not comprise:
  - The `Drand` keyword;
  - More than three identical consecutive characters.

Resolution

This issue has been addressed in PR #601, which enforces a minimum secret size of 32 bytes.

RND-11	Preshared DKG `secret` Can Theoretically be Exposed via a Timing Attack		
Asset	`drand/core/group_setup.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The setup phase leading to the distributed key generation relies on a common secret string shared amongst all participants. The setup coordinator authenticates participating nodes by verifying that the `secret_proof`, submitted via a `SignalDKGPacket`, matches its own secret.

In `group_setup.go:68`, the Drand leader node performs this verification using a non-constant-time comparison. As an authentication failure response is returned, a malicious actor can theoretically perform a timing attack to obtain the secret and participate in the DKG.

This issue is raised with a *low* severity:

- The likelihood of a successful exploit is deemed low due to a requirement for consistent network conditions between the leader and attacker, and the relatively short timing window during which the vulnerability exists (whilst the leader is accepting `SignalDKGParticipant` calls as part of the setup phase).
- The impact of a successful exploit (undesirable nodes participate in DKG) is equivalent to RND-10 and can be mitigated by reviewing group membership upon completion of DKG. As mentioned in the Drand documentation, *"nodes can detect if there are some unwanted nodes after the setup and in that case, setup a new network again"*.

Recommendations

We recommend replacing the current `verifySecret` implementation with one that performs a constant-time comparison. If using `(crypto/subtle).ConstantTimeCompare` or a similar comparison, it is important to note that the implementation will immediately return if the length of its arguments differ. To protect against an attacker using this to learn the length of the secret, we recommend comparing a cryptographic hash of the secrets, which should always have equal length.

Also consider introducing a short, random delay prior to returning a "shared secret is incorrect" response. If of an appropriate duration, this introduced "noise" should make any such timing attacks infeasible.

Resolution

This issue has been addressed in PR #528, which performs a constant-time comparison of a SHA256 hash of the secret.

RND-12	Pre-shared DKG Secret is Passed Sub-Optimally as a CLI Parameter		
Asset	`drand/main.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The pre-shared secret used for DKG setup is passed to the Drand sharing process as a CLI flag. This risks exposure via the following:

- Recovery from bash history (including long after the command has completed).
- It is visible to other users and processes while running (e.g. via `ps aux`)
- The command is often added to syslogs and similar, and can often be transmitted externally as part of a log aggregation solution like a SIEM.

Like other issues associated with the secret, the severity of this threat is reduced because of the secret's short lifespan (assuming that it is not reused during resharing operations) and the ability for node operators to review the group members upon completion of DKG.

Recommendations

Consider implementing the following:

- For interactive use, prompt for the secret separately and read it from STDIN.
- For automated use (potentially useful for regular, automated resharing), best practice is to load from a secrets manager like Hashicorp Vault or AWS Secrets Manager.

  Note: a naive implementation for automated resharing would involve reusing the secret, which should be avoided.
- Also preferable to the current solution is to pass a CLI parameter that is a path to a keystore file with appropriate permissions, or reading it from an environment variable.

Resolution

This issue has been resolved in PR #601, which allows the secret to be provided via an environment variable, and PR #740, which disables the ability to provide the secret via a CLI flag and allows alternative loading from a file.

The testing team also acknowledges that risks associated with malicious local user accounts are largely mitigated for node operators that follow the standard operator guidelines, where the restrictive and minimal environment predominantly prevents untrusted local users.

RND-13	Incorrect SWU Constant		
Asset	`kilic/bls12-381`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The file `swu.go` provides an implementation of the *Shallue-van de Woestijne-Ulas* mapping. The testing team identified the following non-compliances:

- The constant `zInv`, used in the struct `swuParamsForG2` is set to z^{-1} when it should be set to `minusZInv` (i.e. $(-z)^{-1}$).
- In `swuParamsForG1`, the constant `zInv` is set to the correct value of $(-z)^{-1}$, but named misleadingly.

The testing team notes that the probability of reaching this particular scenario/code path is close to zero ($\frac{1}{2^{(381)}}$) for `HashToCurve()` and `EncodeToCurve()` as it would require the output of a `SHA256` function to be zero.

However, this is easily reached in `MapToCurve(0)` in `G2` (not used in Drand).

Recommendations

We recommend updating the *Shallue-van de Woestijne-Ulas* implementation to address the observations above and align the constants used in `swu.go` with the official specification.

Resolution

This issue has been addressed in the upstream BLS library (`kilic/bls12-381`) in PR #14.

RND-14	Insufficient Check on Number of Complaints		
Asset	`kyber/share/dkg`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

During the Distributed Key Generation (DKG) a node is able to issue a complaint against another node if they have not provided them with a valid secret. The accused node will then publicly release that secret as a justification that the complaint was invalid.

In the case where t complaints have been made against a node there will be t publicly available secrets. This will allow the recovery of the original polynomial and thus that user's private shares. As a result, any observer may now sign on behalf of the accused node.

Additionally, if $t - 1$ complaints are made against a single node then the remaining nodes who have not made a complaint will be able to reconstruct the accused nodes private share.

A worst case scenario would be that if t nodes all have t complaints, in which case the final private key may be recovered by any observer. Thus, all future signatures can be calculated. However, for a complaint to be made, the `DealBundle` must not have been received which implies no public polynomials were received hence the node will be evicted. Alternatively, the node distributed invalid shares in which case they are malicious and we assume there is less than t malicious nodes.

Recommendations

Any node who receives t unique complaints should be evicted during the `Response` phase.

Resolution

The recommendation has been implemented in the PR #17.

RND-15	Deserialisation of Points is not Strict on Bytes Length
Asset	`kilic/bls12-381`
Status	Open
Rating	Informational

Description

The `g1.go` and `g2.go` files define the logic for the two cyclic groups of prime order used within the BLS library powering the Drand network.

Each of these files provide two functions (`FromUncompressed()` and `FromCompressed`) that expect byte slices and returns a point in G1 or G2.

These functions are not enforcing strict validation on the byte slices, meaning that given a valid byte slice, it is possible to append any number of bytes and it will deserialise to the same point. Therefore the functions `G1.FromCompressed()` , `G1.FromUncompressed()` , `G2.FromCompressed()` and `G2.FromUncompressed()` are not bijective (one-to-one mapping) when considering the domain to be only valid points.

Similarly, the `fp6.fromBytes()` function accepts inputs greater than 288 bytes (while the equivalent `fp2.fromBytes()` function strictly enforces an input length of 288 bytes). This observation also applies to the `G1.fromBytes()` function. However, in these cases the recursive nature will catch this error when deserialising smaller field elements.

Recommendations

Consider enforcing strict byte lengths for the functions mentioned above for consistency purposes and to prevent possible mutation of signatures.

RND-16	Unused Constants in BLS Library
Asset	`kilic/bls12-381`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The following constants are defined in the `bls12_381.go` file but are never used within the BLS library:

- line [55]: `h2`
- line [56]: `h1`
- line [140] `frobeniusCoeffs2`

Recommendations

Consider removing these unused constants from the code base.

Resolution

This issue has been addressed in the upstream BLS library (`kilic/bls12-381`) in PR #14.

RND-17	Usage of out-of-date Golang Packages
Asset	`drand/drand`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The Drand codebase relies on various external dependencies for which newer versions are available. The table below summarises the versions used in the Drand codebase, and the available updates at the time of writing (direct dependencies only):

Module	Used Version	Available Version
`github.com/go-kit/kit`	v0.9.0	v1.0.2
`github.com/golang/protobuf`	v1.3.5	v1.4.2
`github.com/grpc-ecosystem/grpc-gateway`	v1.14.3	v1.14.6
`github.com/prometheus/client_golang`	v1.5.1	v1.6.0
`github.com/stretchr/testify`	v1.5.1	v1.6.1
`golang.org/x/crypto`	729f1e841bcc	70a84ac30bf9
`google.golang.org/genproto`	33397c535dc2	12044bf5ea91
`google.golang.org/grpc`	v1.27.0	v1.29.1

This finding is raised as *informational* as the updates available to the packages listed above do not seem to include security fixes.

Recommendations

Where possible, consider upgrading the packages listed above to their latest available versions.

Resolution

The development team have clarified their workflow to mitigate risk associated with insecure, outdated dependencies:

- Dependabot alerts notify the development team and trigger a new release when security issues are identified for dependencies.

In addition, the development team has introduced regular dependency updates to their release procedure via PR #729.

RND-18	Locally Triggered Deadlock of Leader's Drand Daemon
Asset	`drand/core/drand_control.go`
Status	**Resolved:** See Resolution
Rating	Informational

Description

It is possible to execute a series of CLI commands that put the Drand daemon into a state similar to a *deadlock*, where each thread that attempts to acquire the `(drand/core/Drand).state` lock blocks indefinitely.

This is due to a bug in `leaderRunSetup()` where it is possible to return without releasing the `state` mutex (see lines [`73,77`] below).

```
   func (d *Drand) leaderRunSetup(in *control.SetupInfoPacket, newSetup func() (*setupManager,
       error)) (*key.Group, error) {
70   // setup the manager
     d.state.Lock()
72   if d.manager != nil {
       return nil, errors.New("drand: setup dkg already in progress")
74   }
     manager, err := newSetup()
76   if err != nil {
       return nil, fmt.Errorf("drand: invalid setup configuration: %s", err)
78   }
```

This issue is deemed informational because, although the impact of an unresponsive daemon is significant, such a state can only be triggered by those who already have access to the daemon's control API and thus full control over the node. This bug, whilst potentially annoying and can be readily triggered by accident, imparts no additional risk.

See Appendix B for the exact commands used to trigger the deadlock.

Recommendations

Ensure that `(*Drand).leaderRunSetup` always releases the `(*Drand).state` lock.

The recommended coding practice to avoid these issues is to `defer d.state.Unlock()` immediately after obtaining the lock. To implement equivalent behavior using this best practice would involve splitting [`71-82`] out into their own function.

An example mitigation could look like the following snippet:

```
func (d *Drand) leaderRunSetup(in *control.SetupInfoPacket, newSetup func() (*setupManager,
    error)) (*key.Group, error) {
  if err := d.checkAndSetupManager(newSetup); err != nil {
    return err
  }
  // ...
}

// setup the manager
```

```go
func (d *Drand) checkAndSetupManager(newSetup func() (*setupManager, error)) (error) {
  d.state.Lock()
  defer d.state.Unlock()

  if d.manager != nil {
    return errors.New("drand: setup dkg already in progress")
  }
  manager, err := newSetup()
  if err != nil {
    return fmt.Errorf("drand: invalid setup configuration: %s", err)
  }
  go manager.run()

  d.manager = manager
  return nil
}
```

Resolution

This issue has been addressed in PR #594.

RND-19	Unused/Dead Code
Asset	`drand/drand`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The testing team identified that the following code sections (functions, types and variables) are unused within Drand:

- `beacon/chain.go:261:6` : struct `beaconInfo`
- `beacon/chain.go:268:2` : struct field `roundCache.previous`
- `beacon/handler.go:359:5` : variable `errOutdatedRound`
- `beacon/store.go:274:6` : function `printStore`
- `demo/main.go:27:5` : variable `tls`
- `demo/main.go:136:6` : function `findTransitionTime`
- `core/config.go:251:6` : function `withClock`
- `core/drand.go:353:17` : function `(*Drand).beaconCallback`
- `core/drand.go:318:17` : function `(*Drand).isDKGDone`
- `core/group_setup.go` : struct `groupReceiver`
- `control.go:120:6` : function `getShare`
- `control.go:219:6` : function `fileExists`
- `key/group.go:105:17` : function `(*Group).identities`
- `main.go:40:7` : constant `gname`
- `main.go:41:7` : constant `dpublic`
- `main.go:110:5` : variable `fromGroupFlag`
- `main.go:202:5` : variable `startInFlag`
- `main.go:580:6` : function `getThreshold`
- `main.go:592:6` : function `getPublicKeys`
- `main.go:706:6` : function `keyIDFromAddr`

- `net/client_grpc.go:329:6` : type `proxyClient`
- `net/client_grpc.go:22:5` : variable `defaultJSONMarshaller`
- `net/client_grpc.go:333:6` : function `newProxyClient`

The dead code referenced above unnecessarily increases the complexity of the Drand code base.

Recommendations

- Consider removing the functions, types, variables and constants listed above.
- Consider incorporating dead–code checkers into the CI pipeline.

Resolution

This issue has been resolved in PRs #421, #534, #558 & #577.

As well as removing the listed dead code, linting has been introduced to identify and prevent this in the future.

RND-20	Miscellaneous Observations on BLS Library
Asset	`kilic/bls12-381`
Status	**Closed:** See Resolution
Rating	Informational

Description

This section details miscellaneous findings discovered by the testing team on the BLS library used by Drand that do not have a direct security implication:

- References to academic papers and actual algorithms implemented, especially for field functions (which are mostly automatically generated) could be added;
- Further details on constants, especially if they are in *Montgomery* form could be included;
- `bls12_381.go` constants refer to `p` and `q` both as the field modulus and `q` as the curve order, which could lead to confusion. Consider using `p` to represent the field modulus and `q` to represent the curve order.
  ✓ Resolved in commit [PR #14]
- The code base includes multiple references to `hash-to-curve-06`, which should be updated to `hash-to-curve-08` or at least to `hash-to-curve-07`):
- *Frobenius* is sometimes spelt incorrectly, see:
  - line [137] in `bls12_381.go` (comment *Frobenious Coeffs*)
  - in `fp2.go` and `fp6.go` (functions `frobeniousMap` and `frobeniousMapAssign`)
- For consistency, in `bls12_381.go` on line [64], `cofactorG2` should be declared with the `0x` prefix;
- In `field_element.go`, consider renaming `fe2` in `Set(fe2 *fe)`, `Cmp(fe2 *fe)` and `Equals(fe2 *fe)` to avoid confusion with type `fe2`.
  ✓ Resolved in commit [PR #14]
- Potential optimization in `func (fe *fe) IsOne()` (in `field_element.go`): `return 1 == fe[0] && 0 == (fe[1] | fe[2] | fe[3] | fe[4] | fe[5])`
- In `g1.go:110` there is a docstring copy-paste error. The docstring for `FromCompressed` states the bytes slice should be at least 96 bytes long, but the requirement is actually 48 bytes.

Recommendations

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

Resolution

The comments have been provided to the upstream maintainer, and have been understood and acknowledged.

RND-21	Miscellaneous Observations on Drand Primary Codebase
Asset	`drand/drand`
Status	**Closed:** See Resolution
Rating	Informational

Description

This section details miscellaneous findings discovered by the testing team on the main Drand codebase that do not have direct security implications:

- In `core/drand_control.go` the following closure is defined:

  ```
  // setup the manager
  newSetup := func() (*setupManager, error) {
      return newDKGSetup(d.log, d.opts.clock, d.priv.Public, in.GetBeaconPeriod(), in.GetInfo())
  }
  ```

  From the function signature alone, it is not obvious that `newSetup` references the `Drand` struct fields and should only be called when the `Drand.state` lock is held. Taking the `Drand` state as a parameter instead of a closure may make this more clear: `func(d *Drand) (*setupManager, error)`

- in `client/http.go`, the `groupResp` and `randResponse` http response bodies (defined at lines [`75,128`]) are not closed, causing resource leakage. See `https://stackoverflow.com/questions/33238518/what-could-happen-if-i-dont-close-response-body` for more info.

- At `core/group_setup.go:130` the docstring for `ReceivedKeys` is out of date, describing return values that are no longer present.

- At `client/client.go:78` the `clientConfig.groupHash` docstring appears to refer to an old struct name: `‘key.GroupInfo.Hash()‘` should likely be `‘key.Group.Hash()‘`.

- In `core/group_setup.go:validInitPacket()`, there is no check for `threshold <= expected`, only that `threshold > expected/2`. There is no expected security impact but such a check could be helpful to catch typos. Though quite unlikely that all participating nodes provide matching, incorrect values, this protects against generating a group that cannot produce a signature.

- In `key/keys.go:19-20`, the docstring is no longer accurate. It claims that the public key is in `G2`, but it is currently in `G1` (the `key.KeyGroup`).

  Similarly, “Currently, drand only supports bn256” (lines [`63-64`]) should be changed to `bls12-381`.

- The directory specified by the `-certs-dir` CLI option apparently needs to only contain valid PEM certificates or a panic is emitted. Consider documenting that the provided certs should be in a PEM format (e.g. in the help text for that CLI argument).

- Invalid values passed to the API can return a 500 response code. e.g. passing an incorrect key to the `/api/private` endpoint can return `500 Internal Server Error` with a message “input string should be equal or larger than 48”, which is a correct rejection of invalid input. This is not a big problem but can cause noisy false-positive alerts in monitoring systems, which generally interpret 500 errors as a potential issue.

- in `key/group.go:175-177` the struct field tag `toml:omitempty` is an incorrect syntax. It should be in the form `toml:"omitempty"`. The tag looks to perhaps work currently this case but is unspecified behavior.[7]
- There are unnecessary locking overheads in `log.NewKitLogger()`.

  `(go-kit/kit/log).NewSyncWriter` and `(go-kit/kit/log).NewSyncLogger` have the same effect, and it is unnecessary to use both at once.[8] The current implementation requires acquiring 2 `sync.Mutex` locks when writing a log (though the performance impact should be minimal with the inner lock never expected to have contention).

  Consider removing the `NewKitLoggerFrom()` call at `log/log.go:83`
- In `main.go:791:2` the returned error should be checked before deferring `file.Close()`

Recommendations

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

Consider incorporating static analysis tools into the CI pipeline, like golangci-lint (we recommend also including the `bodyclose` and `gosec` linters, though the latter necessitates annotations to mask false positives).

Resolution

The comments have been understood and acknowledged, and changes have been actioned where appropriate.

Linting via golangci-lint has been introduced to the CI workflow.

Actions that were taken include:

- Closing http response bodies (http.go:249).
- `ReceivedKey()` docstring updated (group_setup.go:139).
- `NewKeyPair()` docstring updated (keys.go:86-87).
- CLI `-certs-dir` flag updated to note the required PEM format (cli.go:115).
- `/api/private` HTTP endpoint no longer exists to return `500` errors (server.go:52).
- TOML struct tags updated to correct syntax (group.go:172-174).

[7] "If the tag does not have the conventional format, the value returned by Get is unspecified." (https://golang.org/pkg/reflect/#StructTag.Get)

[8] A SyncWriter alone should be sufficient - https://github.com/go-kit/kit/blob/81a2d1f550bfc91aceb4d0d5e7a7709d76548e7e/log/logfmt_logger.go#L55-L57

Appendix A Evidence for DoS via Partial Signature Flood

This section provides a technical description of RND-01 and relevant evidence.

Analysis

RND-01 is caused by a combination of factors associated with how a Drand node processes incoming `PartialBeaconPacket` messages ("partials"). The relevant code is primarily located in `(*Handler).ProcessPartialBeacon` in `drand/beacon/handler.go` and processing of `(*chainStore).newPartials` messages in `(*chainStore).runAggregator` in `drand/beacon/chain.go`. Also important, is the handling of pending beacons, built in the `var caches []*roundCache` variable (which we will refer to as `caches`, to differentiate between it and a single `roundCache` struct).

- A duplicate partial is added as a new `roundCache` entry.

  When processing a new partial in `runAggregator`, each `caches` entry is iterated through and `tryAppend` executed.

  Because `tryAppend` returns `false` when passed a duplicate partial (that has already been seen), a new `roundCache` will be created upon completion of the iteration. Instead, this partial should be dropped and not processed further.

  With this, an attacker can arbitrarily increase the length of `caches` and cause multiple `caches` to contain multiple `roundCache` structs for the same `round` and `previousSig` (where the design appears to intend otherwise). By sending k valid, but duplicate partials per round, an attacker can expect `caches` to consistently contain on the order of $k+1$ entries (because each new Beacon will clear entries from previous rounds).

- When `caches` contains more than one `roundCache` entry with the same `round` & `previousSig`, applicable partials will be copied and added to all entries.

  With this, partials from friendly nodes can amplify duplicate partials sent by a malicious actor. For a network consisting of n friendly nodes and 1 malicious node, when the malicious node sends k duplicate partial signatures per round, each targeted node is expected to have a `caches` that contains roughly l partials, where $k+n<l\leq kn$.

- Several operations are performed in time linearly proportional to the length of `caches`. These include:

  - Adding a new partial
  - Filtering the `caches` upon completion of a Beacon

  These operations occur quite regularly, and become expensive when `len(caches)` is large.

- When a partial is submitted where its round which already has a complete Beacon (i.e. in the context of `runAggregator`, `pRound <= lastBeacon.Round`) an "ignoring_partial" log is emitted but the partial has already been added to `caches`. This should be dropped and not processed further.

- When a partial is passed to `(*Handler).ProcessPartialBeacon`, but the partial's round already has a complete Beacon, there is no need to validate the partial's signature or send it to the `chainStore`. It can be immediately dropped. The handler could track the round of the most recent complete Beacon via callback or reading from the store.

- If duplicate partials are ignored and dropped, a malicious node can still arbitrarily increase `len(caches)` by sending correctly signed partials containing the current round but some random, invalid `previousSig`.

- The `(*chainStore).newPartials` channel is a buffered channel of size 10. If this is full (occurs when `runAggregator` spends too much time iterating through large `caches`), the handler will block on `NewValidPartial` in `ProcessPartialBeacon`.

Small demo and `roundCache` entry size

The log excerpt below is from a Drand docker demo with the following modifications:

- `drand2` broadcasts all partials generated by it 5 times to each of its peers (so sends $(n-1) \times 5 = 20$ `PartialBeaconPacket` messages per round).
- Additional logging has been added to `drand/beacon/chain.go`.

  Upon receiving from the `c.lastInserted` channel (when a new beacon has been added to the chain), the `cache_len_before_filter` log emits the number of currently active `roundCache` entries.

  When subsequently "filter[ing] all caches inferior to this beacon" a log is emitted for each `roundCache` entry processed that contains more than 1 partial signature (`len(roundCache.sigs) > 1`). When the entry is removed, a `clearing=large_cache_entry` log is emitted with `num_sigs` denoting `len(roundCache.sigs)`. Similarly, `keeping=large_cache_entry` is output when the entry is retained.

From this, we can make the following observations:

- Here `drand2` is the malicious node, so its cache remains small (doesn't receive any additional partials).
- The number of cache entries roughly corresponds to the number of partial duplicates broadcast by the malicious node.
- Most cache entries contain more than the 1 `sig` sent by the malicious node, therefore partials from friendly nodes are being copied.
- Most cache entries are being removed every round.

  This is expected, as the clocks of each node are in sync and all duplicate partials are associated with the current round. If malicious partials were sent for `currentRound + 1` (allowed through via `(drand/beacon/*Handler).ProcessPartialBeacon`), we would expect the cache length to roughly double with each malicious entry persisting for 2 rounds.

```
drand4    | ts="Thu, 04 Jun 2020 09:05:50 UTC" call=asm_amd64.s:1373 level=info
    NEW_BEACON_STORED="{ round: 2, sig: 87842a, prevSig: b8add2 }"
drand4    | ts="Thu, 04 Jun 2020 09:05:50 UTC" call=:0 level=info cache_len_before_filter=4
drand4    | ts="Thu, 04 Jun 2020 09:05:50 UTC" call=:0 level=info clearing=large_cache_entry
    num_sigs=5
drand2    | ts="Thu, 04 Jun 2020 09:05:50 UTC" call=asm_amd64.s:1373 level=info
    NEW_BEACON_STORED="{ round: 2, sig: 87842a, prevSig: b8add2 }"
drand2    | ts="Thu, 04 Jun 2020 09:05:50 UTC" call=:0 level=info cache_len_before_filter=1
drand2    | ts="Thu, 04 Jun 2020 09:05:50 UTC" call=:0 level=info clearing=large_cache_entry
    num_sigs=5
drand1    | ts="Thu, 04 Jun 2020 09:05:50 UTC" call=asm_amd64.s:1373 level=info
    NEW_BEACON_STORED="{ round: 2, sig: 87842a, prevSig: b8add2 }"
drand1    | ts="Thu, 04 Jun 2020 09:05:50 UTC" call=:0 level=info cache_len_before_filter=6
drand1    | ts="Thu, 04 Jun 2020 09:05:50 UTC" call=:0 level=info clearing=large_cache_entry
    num_sigs=5
drand1    | ts="Thu, 04 Jun 2020 09:05:50 UTC" call=:0 level=info clearing=large_cache_entry
    num_sigs=2
drand1    | ts="Thu, 04 Jun 2020 09:05:50 UTC" call=:0 level=info clearing=large_cache_entry
    num_sigs=2
```

```
drand1    | ts="Thu, 04 Jun 2020 09:05:50 UTC" call=:0 level=info clearing=large_cache_entry
    num_sigs=2
drand1    | ts="Thu, 04 Jun 2020 09:05:50 UTC" call=:0 level=info clearing=large_cache_entry
    num_sigs=2
drand4    | ts="Thu, 04 Jun 2020 09:05:50 UTC" call=:0 level=error ignoring_partial=2
    last_beacon_stored=2
drand3    | ts="Thu, 04 Jun 2020 09:05:50 UTC" call=:0 level=info aggregated_beacon=2
drand3    | ts="Thu, 04 Jun 2020 09:05:50 UTC" call=asm_amd64.s:1373 level=info
    NEW_BEACON_STORED="{ round: 2, sig: 87842a, prevSig: b8add2 }"
drand5    | ts="Thu, 04 Jun 2020 09:05:50 UTC" call=:0 level=info aggregated_beacon=2
drand5    | ts="Thu, 04 Jun 2020 09:05:50 UTC" call=asm_amd64.s:1373 level=info
    NEW_BEACON_STORED="{ round: 2, sig: 87842a, prevSig: b8add2 }"
drand5    | ts="Thu, 04 Jun 2020 09:05:50 UTC" call=:0 level=info cache_len_before_filter=9
drand5    | ts="Thu, 04 Jun 2020 09:05:50 UTC" call=:0 level=info clearing=large_cache_entry
    num_sigs=5
drand5    | ts="Thu, 04 Jun 2020 09:05:50 UTC" call=:0 level=info clearing=large_cache_entry
    num_sigs=2
drand5    | ts="Thu, 04 Jun 2020 09:05:50 UTC" call=:0 level=info clearing=large_cache_entry
    num_sigs=2
drand5    | ts="Thu, 04 Jun 2020 09:05:50 UTC" call=:0 level=info clearing=large_cache_entry
    num_sigs=2
drand5    | ts="Thu, 04 Jun 2020 09:05:50 UTC" call=:0 level=info clearing=large_cache_entry
    num_sigs=2
drand3    | ts="Thu, 04 Jun 2020 09:05:50 UTC" call=:0 level=info aggregated_beacon=2
drand3    | ts="Thu, 04 Jun 2020 09:05:50 UTC" call=:0 level=info cache_len_before_filter=8
drand3    | ts="Thu, 04 Jun 2020 09:05:50 UTC" call=:0 level=info clearing=large_cache_entry
    num_sigs=2
drand3    | ts="Thu, 04 Jun 2020 09:05:50 UTC" call=:0 level=info clearing=large_cache_entry
    num_sigs=2
drand3    | ts="Thu, 04 Jun 2020 09:05:50 UTC" call=:0 level=info clearing=large_cache_entry
    num_sigs=2
drand3    | ts="Thu, 04 Jun 2020 09:05:50 UTC" call=:0 level=info clearing=large_cache_entry
    num_sigs=5
drand3    | ts="Thu, 04 Jun 2020 09:05:50 UTC" call=:0 level=info clearing=large_cache_entry
    num_sigs=3
drand3    | ts="Thu, 04 Jun 2020 09:05:50 UTC" call=:0 level=info clearing=large_cache_entry
    num_sigs=3
drand3    | ts="Thu, 04 Jun 2020 09:05:50 UTC" call=:0 level=info clearing=large_cache_entry
    num_sigs=3
drand3    | ts="Thu, 04 Jun 2020 09:05:50 UTC" call=:0 level=info clearing=large_cache_entry
    num_sigs=3
drand1    | ts="Thu, 04 Jun 2020 09:06:00 UTC" call=:0 level=info aggregated_beacon=3
drand2    | ts="Thu, 04 Jun 2020 09:06:00 UTC" call=:0 level=info aggregated_beacon=3
drand4    | ts="Thu, 04 Jun 2020 09:06:00 UTC" call=:0 level=info aggregated_beacon=3
drand3    | ts="Thu, 04 Jun 2020 09:06:00 UTC" call=:0 level=info aggregated_beacon=3
drand5    | ts="Thu, 04 Jun 2020 09:06:00 UTC" call=:0 level=info aggregated_beacon=3
drand1    | ts="Thu, 04 Jun 2020 09:06:00 UTC" call=asm_amd64.s:1373 level=info
    NEW_BEACON_STORED="{ round: 3, sig: a2289b, prevSig: 87842a }"
drand1    | ts="Thu, 04 Jun 2020 09:06:00 UTC" call=:0 level=info cache_len_before_filter=5
drand1    | ts="Thu, 04 Jun 2020 09:06:00 UTC" call=:0 level=info clearing=large_cache_entry
    num_sigs=5
drand1    | ts="Thu, 04 Jun 2020 09:06:00 UTC" call=:0 level=info clearing=large_cache_entry
    num_sigs=2
drand1    | ts="Thu, 04 Jun 2020 09:06:00 UTC" call=:0 level=info clearing=large_cache_entry
    num_sigs=2
drand1    | ts="Thu, 04 Jun 2020 09:06:00 UTC" call=:0 level=info clearing=large_cache_entry
    num_sigs=2
drand1    | ts="Thu, 04 Jun 2020 09:06:00 UTC" call=:0 level=info clearing=large_cache_entry
    num_sigs=2
drand2    | ts="Thu, 04 Jun 2020 09:06:00 UTC" call=asm_amd64.s:1373 level=info
    NEW_BEACON_STORED="{ round: 3, sig: a2289b, prevSig: 87842a }"
```

DoS PoC and Number of `roundCache` entries

The following logs are an excerpt from a friendly daemon running during a test where a network of 4 nodes (threshold 3, no TLS) were run across separate AWS EC2 instances (4GB Memory, 2 Core).

Additional logging was added where, upon receiving from the `c.lastInserted` channel (when a new beacon has been added to the chain), the `cache_len_before_filter` log emits the number of currently active `roundCache` entries.

A single "malicious" node was modified to broadcast a configurable number of duplicate `PartialBeaconPacket` messages every round.

From the below log excerpts we can observe the following:

- At "manageable" rates of flooding, the size of `caches` is remains similar to the rate of partials per round.
- As the rate increases, the partials are still trickling into `(beacon/chainStore).newPartials`, such that they are increasingly associated with previous Beacons.

```
(1000 broadcasts per round)
ts="Thu, 04 Jun 2020 09:53:40 UTC" call=:0 level=info aggregated_beacon=15
ts="Thu, 04 Jun 2020 09:53:40 UTC" call=asm_amd64.s:1357 level=info NEW_BEACON_STORED="{ round
    : 15, sig: a0d095, prevSig: 8be376 }"
ts="Thu, 04 Jun 2020 09:53:40 UTC" call=:0 level=info cache_len_before_filter=1001
ts="Thu, 04 Jun 2020 09:53:40 UTC" call=:0 level=error ignoring_partial=15 last_beacon_stored
    =15
ts="Thu, 04 Jun 2020 09:53:41 UTC" call=:0 level=error ignoring_partial=15 last_beacon_stored
    =15
ts="Thu, 04 Jun 2020 09:53:41 UTC" call=:0 level=error ignoring_partial=15 last_beacon_stored
    =15
...
ts="Thu, 04 Jun 2020 09:53:50 UTC" call=:0 level=info aggregated_beacon=16
ts="Thu, 04 Jun 2020 09:53:50 UTC" call=asm_amd64.s:1357 level=info NEW_BEACON_STORED="{ round
    : 16, sig: b559c8, prevSig: a0d095 }"
ts="Thu, 04 Jun 2020 09:53:50 UTC" call=:0 level=info cache_len_before_filter=1001
... (2000 broadcasts per round)
 --- STREAM EOF
ts="Thu, 04 Jun 2020 09:56:10 UTC" call=asm_amd64.s:1357 level=info NEW_BEACON_STORED="{ round
    : 29, sig: b2bf3b, prevSig: 9876c0 }"
ts="Thu, 04 Jun 2020 09:56:10 UTC" call=:0 level=info cache_len_before_filter=2297
ts="Thu, 04 Jun 2020 09:56:10 UTC" call=:0 level=error ignoring_partial=28 last_beacon_stored
    =29
ts="Thu, 04 Jun 2020 09:56:10 UTC" call=:0 level=error ignoring_partial=28 last_beacon_stored
    =29
ts="Thu, 04 Jun 2020 09:56:10 UTC" call=:0 level=error ignoring_partial=28 last_beacon_stored
    =29
...
ts="Thu, 04 Jun 2020 09:56:21 UTC" call=asm_amd64.s:1357 level=info NEW_BEACON_STORED="{ round
    : 30, sig: 908ae9, prevSig: b2bf3b }"
ts="Thu, 04 Jun 2020 09:56:21 UTC" call=:0 level=info cache_len_before_filter=1322
ts="Thu, 04 Jun 2020 09:56:21 UTC" call=:0 level=error ignoring_partial=28 last_beacon_stored
    =30
ts="Thu, 04 Jun 2020 09:56:21 UTC" call=:0 level=error ignoring_partial=28 last_beacon_stored
    =30
ts="Thu, 04 Jun 2020 09:56:21 UTC" call=:0 level=error ignoring_partial=28 last_beacon_stored
    =30
ts="Thu, 04 Jun 2020 09:56:21 UTC" call=:0 level=error ignoring_partial=28 last_beacon_stored
    =30
ts="Thu, 04 Jun 2020 09:56:21 UTC" call=:0 level=error ignoring_partial=28 last_beacon_stored
    =30
...
ts="Thu, 04 Jun 2020 09:56:34 UTC" call=:0 level=error ignoring_partial=28 last_beacon_stored
    =30
 --- STREAM ERR: rpc error: code = Canceled desc = context canceled
ts="Thu, 04 Jun 2020 09:56:34 UTC" call=:0 level=error ignoring_partial=28 last_beacon_stored
    =30
 --- STREAM EOF
ts="Thu, 04 Jun 2020 09:56:34 UTC" call=asm_amd64.s:1357 level=info NEW_BEACON_STORED="{ round
    : 31, sig: 87282b, prevSig: 908ae9 }"
ts="Thu, 04 Jun 2020 09:56:34 UTC" call=:0 level=error ignoring_partial=28 last_beacon_stored
    =30
```

```
ts="Thu, 04 Jun 2020 09:56:34 UTC" call=:0 level=error ignoring_partial=28 last_beacon_stored
    =30
ts="Thu, 04 Jun 2020 09:56:34 UTC" call=asm_amd64.s:1357 level=info NEW_BEACON_STORED="{ round
    : 32, sig: 95fc6e, prevSig: 87282b }"
ts="Thu, 04 Jun 2020 09:56:34 UTC" call=:0 level=info cache_len_before_filter=1547
ts="Thu, 04 Jun 2020 09:56:34 UTC" call=:0 level=info cache_len_before_filter=1
ts="Thu, 04 Jun 2020 09:56:34 UTC" call=:0 level=error ignoring_partial=28 last_beacon_stored
    =32
ts="Thu, 04 Jun 2020 09:56:34 UTC" call=:0 level=error ignoring_partial=28 last_beacon_stored
    =32
ts="Thu, 04 Jun 2020 09:56:34 UTC" call=:0 level=error ignoring_partial=28 last_beacon_stored
    =32
ts="Thu, 04 Jun 2020 09:56:34 UTC" call=:0 level=error ignoring_partial=28 last_beacon_stored
    =32
ts="Thu, 04 Jun 2020 09:56:34 UTC" call=:0 level=error ignoring_partial=28 last_beacon_stored
    =32
```

The following excerpt is from a node that fell behind far enough that it attempted to sync (while nodes are experiencing more than 10 000 partials per round).

```
ts="Thu, 04 Jun 2020 09:59:35 UTC" call=:0 level=error sync_from=34 error="rpc error: code =
    Unavailable desc = connection error: desc = \"transport: Error while dialing dial tcp
    3.235.223.246:8080: connect: connection refused\"" from=3.235.223.246:8080
ts="Thu, 04 Jun 2020 09:59:36 UTC" call=:0 level=error sync_from=34 error="rpc error: code =
    Unavailable desc = connection error: desc = \"transport: Error while dialing dial tcp
    3.235.223.246:8080: connect: connection refused\"" from=3.235.223.246:8080
ts="Thu, 04 Jun 2020 09:59:38 UTC" call=:0 level=error sync_from=34 error="rpc error: code =
    Unavailable desc = connection error: desc = \"transport: Error while dialing dial tcp
    34.236.36.165:8088: connect: connection refused\"" from=34.236.36.165:8088
ts="Thu, 04 Jun 2020 09:59:41 UTC" call=:0 level=error beacon_round=34 err_request="rpc error:
     code = DeadlineExceeded desc = context deadline exceeded" from=3.231.224.34:8080
ts="Thu, 04 Jun 2020 09:59:43 UTC" call=:0 level=error ignoring_partial=30 last_beacon_stored
    =33
ts="Thu, 04 Jun 2020 09:59:44 UTC" call=:0 level=error beacon_round=34 err_request="rpc error:
     code = Unavailable desc = connection error: desc = \"transport: Error while dialing dial
    tcp 34.236.36.165:8088: connect: connection refused\"" from=34.236.36.165:8088
ts="Thu, 04 Jun 2020 09:59:47 UTC" call=:0 level=error sync_from=34 error="rpc error: code =
    Unavailable desc = connection error: desc = \"transport: Error while dialing dial tcp
    34.236.36.165:8088: connect: connection refused\"" from=34.236.36.165:8088
 --- STREAM ERR: rpc error: code = Canceled desc = context canceled
ts="Thu, 04 Jun 2020 09:59:48 UTC" call=:0 level=error beacon_round=34 err_request="rpc error:
     code = Unavailable desc = connection error: desc = \"transport: Error while dialing dial
    tcp 3.235.223.246:8080: connect: connection refused\"" from=3.235.223.246:8080
 --- STREAM ERR: rpc error: code = Unavailable desc = transport is closing
ts="Thu, 04 Jun 2020 09:59:52 UTC" call=:0 level=error beacon_round=34 err_request="rpc error:
     code = DeadlineExceeded desc = context deadline exceeded" from=3.231.224.34:8080
ts="Thu, 04 Jun 2020 09:59:56 UTC" call=:0 level=error ignoring_partial=30 last_beacon_stored
    =33
ts="Thu, 04 Jun 2020 09:59:57 UTC" call=:0 level=error beacon_round=34 err_request="rpc error:
     code = Unavailable desc = transport is closing" from=3.231.224.34:8080
ts="Thu, 04 Jun 2020 10:00:00 UTC" call=:0 level=error beacon_round=34 err_request="rpc error:
     code = Unavailable desc = transport is closing" from=3.231.224.34:8080
ts="Thu, 04 Jun 2020 10:00:07 UTC" call=:0 level=error beacon_round=34 err_request="rpc error:
     code = Unavailable desc = transport is closing" from=3.231.224.34:8080
ts="Thu, 04 Jun 2020 10:00:11 UTC" call=:0 level=error beacon_round=34 err_request="rpc error:
     code = Unavailable desc = transport is closing" from=3.231.224.34:8080
ts="Thu, 04 Jun 2020 10:00:13 UTC" call=:0 level=error beacon_round=34 err_request="rpc error:
     code = Unavailable desc = transport is closing" from=3.231.224.34:8080
ts="Thu, 04 Jun 2020 10:00:15 UTC" call=:0 level=error beacon_round=34 err_request="rpc error:
     code = Unavailable desc = connection error: desc = \"transport: Error while dialing dial
    tcp 3.235.223.246:8080: connect: connection refused\"" from=3.235.223.246:8080
ts="Thu, 04 Jun 2020 10:00:18 UTC" call=:0 level=error sync_from=34 error="rpc error: code =
    Unavailable desc = connection error: desc = \"transport: Error while dialing dial tcp
    3.235.223.246:8080: connect: connection refused\"" from=3.235.223.246:8080
ts="Thu, 04 Jun 2020 10:00:23 UTC" call=:0 level=error sync_from=34 error="rpc error: code =
    Unavailable desc = connection error: desc = \"transport: Error while dialing dial tcp
    3.235.223.246:8080: connect: connection refused\"" from=3.235.223.246:8080
ts="Thu, 04 Jun 2020 10:00:26 UTC" call=:0 level=error ignoring_partial=30 last_beacon_stored
    =33
 --- STREAM ERR: rpc error: code = Canceled desc = context canceled
```

```
ts="Thu, 04 Jun 2020 10:00:32 UTC" call=:0 level=error beacon_round=34 err_request="rpc error:
     code = Unavailable desc = connection error: desc = \"transport: Error while dialing dial
    tcp 3.231.224.34:8080: connect: connection refused\"" from=3.231.224.34:8080
 --- STREAM ERR: rpc error: code = Canceled desc = context canceled
ts="Thu, 04 Jun 2020 10:00:40 UTC" call=:0 level=error beacon_round=34 err_request="rpc error:
     code = Unavailable desc = connection error: desc = \"transport: Error while dialing dial
    tcp 3.235.223.246:8080: connect: connection refused\"" from=3.235.223.246:8080
ts="Thu, 04 Jun 2020 10:00:44 UTC" call=:0 level=error sync_from=34 error="rpc error: code =
    Unavailable desc = connection error: desc = \"transport: Error while dialing dial tcp
    3.231.224.34:8080: connect: connection refused\"" from=3.231.224.34:8080
ts="Thu, 04 Jun 2020 10:00:51 UTC" call=:0 level=error beacon_round=34 err_request="rpc error:
     code = Unavailable desc = connection error: desc = \"transport: Error while dialing dial
    tcp 3.231.224.34:8080: connect: connection refused\"" from=3.231.224.34:8080
ts="Thu, 04 Jun 2020 10:00:54 UTC" call=:0 level=error sync_from=34 error="rpc error: code =
    Unavailable desc = connection error: desc = \"transport: Error while dialing dial tcp
    3.231.224.34:8080: connect: connection refused\"" from=3.231.224.34:8080
ts="Thu, 04 Jun 2020 10:00:59 UTC" call=:0 level=error sync_from=34 error="rpc error: code =
    Unavailable desc = connection error: desc = \"transport: Error while dialing dial tcp
    3.231.224.34:8080: connect: connection refused\"" from=3.231.224.34:8080
ts="Thu, 04 Jun 2020 10:01:18 UTC" call=:0 level=error ignoring_partial=30 last_beacon_stored
    =33
ts="Thu, 04 Jun 2020 10:01:20 UTC" call=:0 level=error beacon_round=34 err_request="rpc error:
     code = Unavailable desc = connection error: desc = \"transport: Error while dialing dial
    tcp 3.235.223.246:8080: connect: connection refused\"" from=3.235.223.246:8080
```

After increasing the number of `rebroadcasts` to 50 000 per round, all (synced?) nodes experienced similar runtime errors, causing a complete halt of the network:

```
fatal error: runtime: cannot allocate memory

runtime stack:
runtime.throw(0xbbaef2, 0x1f)
  /usr/lib/golang/src/runtime/panic.go:774 +0x72
runtime.persistentalloc1(0x4000, 0x0, 0x117ed18, 0x0)
  /usr/lib/golang/src/runtime/malloc.go:1323 +0x2c7
runtime.persistentalloc.func1()
  /usr/lib/golang/src/runtime/malloc.go:1277 +0x45
runtime.persistentalloc(0x4000, 0x0, 0x117ed18, 0x0)
  /usr/lib/golang/src/runtime/malloc.go:1276 +0x82
runtime.(*fixalloc).alloc(0x11687b8, 0x1)
  /usr/lib/golang/src/runtime/mfixalloc.go:80 +0xf5
runtime.(*mheap).allocSpanLocked(0x1166280, 0x1, 0x117ece8, 0x7f6d05689f78)
  /usr/lib/golang/src/runtime/mheap.go:1195 +0xfa
runtime.(*mheap).alloc_m(0x1166280, 0x1, 0x7ffccc9b0025, 0x7f6d05689f78)
  /usr/lib/golang/src/runtime/mheap.go:1022 +0xc2
runtime.(*mheap).alloc.func1()
  /usr/lib/golang/src/runtime/mheap.go:1093 +0x4c
runtime.systemstack(0x45e104)
  /usr/lib/golang/src/runtime/asm_amd64.s:370 +0x66
runtime.mstart()
  /usr/lib/golang/src/runtime/proc.go:1146

goroutine 178 [running]:
runtime.systemstack_switch()
  /usr/lib/golang/src/runtime/asm_amd64.s:330 fp=0xc0012c69e8 sp=0xc0012c69e0 pc=0x45e200
runtime.(*mheap).alloc(0x1166280, 0x1, 0x10025, 0xa83220)
  /usr/lib/golang/src/runtime/mheap.go:1092 +0x8a fp=0xc0012c6a38 sp=0xc0012c69e8 pc=0x427a4a
runtime.(*mcentral).grow(0x1166f78, 0x0)
  /usr/lib/golang/src/runtime/mcentral.go:255 +0x7b fp=0xc0012c6a78 sp=0xc0012c6a38 pc=0
    x41999b
runtime.(*mcentral).cacheSpan(0x1166f78, 0x203037)
  /usr/lib/golang/src/runtime/mcentral.go:106 +0x2fe fp=0xc0012c6ad8 sp=0xc0012c6a78 pc=0
    x4194be
runtime.(*mcache).refill(0x7f6d349306d0, 0x25)
  /usr/lib/golang/src/runtime/mcache.go:138 +0x85 fp=0xc0012c6af8 sp=0xc0012c6ad8 pc=0x418f65
runtime.(*mcache).nextFree(0x7f6d349306d0, 0xb0f425, 0x8, 0xadd4a0, 0x8)
  /usr/lib/golang/src/runtime/malloc.go:854 +0x87 fp=0xc0012c6b30 sp=0xc0012c6af8 pc=0x40d8b7
runtime.mallocgc(0x120, 0xabf0a0, 0xc0012c6c01, 0x58799b)
  /usr/lib/golang/src/runtime/malloc.go:1022 +0x793 fp=0xc0012c6bd0 sp=0xc0012c6b30 pc=0
    x40e1f3
```

```
runtime.newobject(0xabf0a0, 0x117cb70)
  /usr/lib/golang/src/runtime/malloc.go:1151 +0x38 fp=0xc0012c6c00 sp=0xc0012c6bd0 pc=0x40e5e8
github.com/drand/bls12-381.nullKyberG2(...)
  /home/ec2-user/go/pkg/mod/github.com/drand/bls12-381@v0.3.2/kyber_g2.go:23
github.com/drand/bls12-381.NewGroupG2.func1(0x0, 0xc0dca5ba72)
  /home/ec2-user/go/pkg/mod/github.com/drand/bls12-381@v0.3.2/kyber_group.go:72 +0x2d fp=0
    xc0012c6c28 sp=0xc0012c6c00 pc=0x57fdcd
github.com/drand/bls12-381.(*groupBls).Point(...)
  /home/ec2-user/go/pkg/mod/github.com/drand/bls12-381@v0.3.2/kyber_group.go:39
github.com/drand/bls12-381.(*groupBls).PointLen(0xc0000a6d00, 0x62)
  /home/ec2-user/go/pkg/mod/github.com/drand/bls12-381@v0.3.2/kyber_group.go:35 +0x2b fp=0
    xc0012c6c48 sp=0xc0012c6c28 pc=0x57b49b
github.com/drand/kyber/sign/tbls.(*scheme).IndexOf(0xc000122750, 0xc00af1f490, 0x62, 0x70, 0x1
    , 0x0, 0x0)
  /home/ec2-user/go/pkg/mod/github.com/drand/kyber@v1.0.1-0.20200502215402-daa30f0ec4f8/sign/
    tbls/tbls.go:90 +0x38 fp=0xc0012c6c88 sp=0xc0012c6c48 pc=0x587ee8
github.com/drand/drand/beacon.(*roundCache).tryAppend(0xc0790e39f0, 0xc0004fd500, 0xc00036cb00
    )
  /home/ec2-user/drand-review/code/drand/beacon/chain.go:308 +0x99 fp=0xc0012c6d18 sp=0
    xc0012c6c88 pc=0x99fca9
github.com/drand/drand/beacon.(*chainStore).runAggregator(0xc000130c40)
  /home/ec2-user/drand-review/code/drand/beacon/chain.go:124 +0x513 fp=0xc0012c6fd8 sp=0
    xc0012c6d18 pc=0x99e163
runtime.goexit()
  /usr/lib/golang/src/runtime/asm_amd64.s:1357 +0x1 fp=0xc0012c6fe0 sp=0xc0012c6fd8 pc=0
    x4602d1
created by github.com/drand/drand/beacon.newChainStore
  /home/ec2-user/drand-review/code/drand/beacon/chain.go:46 +0x1fc

...(repeated per goroutine)
```

Appendix B Local Deadlock Evidence

This section denotes CLI commands and replication steps used to trigger the deadlock-like bug described in RND-18.

The following commands assume a precondition that the persistent keys have been generated and the Drand daemon is currently running.

DKG already in progress

The following commands trigger the bug by exercising the error result “drand: setup dkg already in progress” (defined at `drand/core/drand_control.go:73`).

In one terminal:

```
$ ./drand share --tls-disable --secret asdf --period 10s --nodes 5 --threshold 3 --leader
drand master (date unknown, commit none) by nikkolasg
WARNING: this software has NOT received a full audit and must be used with caution and
    probably NOT in a production environment.
Initiating the DKG as a leader
You can stop the command at any point. If so, the group file will not be written out to the
    specified output. To get thegroup file once the setup phase is done, you can run the ‘
    drand showgroup‘ command
```

In another terminal:

```
$ ./drand share --tls-disable --secret asdf --period 10s --nodes 5 --threshold 3 --leader
drand master (date unknown, commit none) by nikkolasg
WARNING: this software has NOT received a full audit and must be used with caution and
    probably NOT in a production environment.
Initiating the DKG as a leader
You can stop the command at any point. If so, the group file will not be written out to the
    specified output. To get thegroup file once the setup phase is done, you can run the ‘
    drand showgroup‘ command
 --- got err rpc error: code = Unknown desc = drand: invalid setup configuration: drand: setup
     dkg already in progress group <nil>
error setting up the network: <nil>
```

The lock has now been lost (the previous terminals can now be closed), and any subsequent instruction attempting to acquire the lock will cause the goroutine to block indefinitely. e.g. This is not actually performing the share setup:

```
$ ./drand share --tls-disable --secret asdf --period 10s --nodes 5 --threshold 1 --leader
drand master (date unknown, commit none) by nikkolasg
WARNING: this software has NOT received a full audit and must be used with caution and
    probably NOT in a production environment.
Initiating the DKG as a leader
You can stop the command at any point. If so, the group file will not be written out to the
    specified output. To get thegroup file once the setup phase is done, you can run the ‘
    drand showgroup‘ command
^C
```

Once hung, the daemon is completely unresponsive:

```
$ ./drand util check 127.0.0.1:8088
drand master (date unknown, commit none) by nikkolasg
WARNING: this software has NOT received a full audit and must be used with caution and
    probably NOT in a production environment.
drand: error checking id 127.0.0.1:8088
drand: error running app: Following nodes don’t answer: 127.0.0.1:8088
```

It can only be killed via signal, not stopped cleanly:

```
$ ./drand stop
drand master (date unknown, commit none) by nikkolasg
WARNING: this software has NOT received a full audit and must be used with caution and
    probably NOT in a production environment.
^C
# this hangs
```

Below are the relevant daemon logs:

```
drand master (date unknown, commit none) by nikkolasg
WARNING: this software has NOT received a full audit and must be used with caution and
    probably NOT in a production environment.
drand: will run as fresh install -> expect to run DKG.
ts="Fri, 05 Jun 2020 07:16:58 UTC" call=daemon.go:29 level=info network=tls-disable
ts="Fri, 05 Jun 2020 07:16:58 UTC" call=daemon.go:29 level=info private_listen=127.0.0.1:8088
    control_port=8888 public_listen=
ts="Fri, 05 Jun 2020 07:17:35 UTC" call=server.go:1024 level=info init_dkg=begin time
    =1591341455 leader=true
ts="Fri, 05 Jun 2020 07:17:35 UTC" call=control.pb.go:1145 level=info leader_pid=4324
ts="Fri, 05 Jun 2020 07:18:15 UTC" call=server.go:1024 level=info init_dkg=begin time
    =1591341495 leader=true
ts="Fri, 05 Jun 2020 07:18:15 UTC" call=control.pb.go:1145 level=info leader_pid=4324
```

Invalid Setup Configuration

The following commands trigger the bug by exercising the error result “drand: invalid setup configuration: %s” (defined at `drand/core/drand_control.go:77`)

```
$ ./drand share --tls-disable --secret asdf --period 10s --nodes 5 --threshold 1 --leader
drand master (date unknown, commit none) by nikkolasg
WARNING: this software has NOT received a full audit and must be used with caution and
    probably NOT in a production environment.
Initiating the DKG as a leader
You can stop the command at any point. If so, the group file will not be written out to the
    specified output. To get thegroup file once the setup phase is done, you can run the ‘
    drand showgroup‘ command
ts="Thu, 11 Jun 2020 09:47:16 UTC" call=server.go:1024 level=info init_dkg=begin time
    =1591868836 leader=true
ts="Thu, 11 Jun 2020 09:47:16 UTC" call=control.pb.go:1145 level=info leader_pid=32287
 --- got err rpc error: code = Unknown desc = drand: invalid setup configuration: drand:
    invalid setup configuration: invalid thr: 5 nodes, need thr 1 got 3 group <nil>
error setting up the network: <nil>
```

The daemon is now similarly hung and does not respond.

```
$ ./drand show private
^C
```

Appendix C DoS via Partial Signature Flood - Additional Testing Details

This section provides a technical description of additional issues found during retesting and associated RND-01, along with relevant evidence and analysis.

Methodology and Test Setup

The test network and methodology used here is largely the same as for initial testing of RND-01 (though updated to test more recent code changes):

- 4 AWS EC2 instances, running Drand nodes, each with at least 4GB RAM and 2 cores
- Network size: 4
- Threshold: 3
- No TLS
- 1 node modified to act as an attacker and broadcast a configurable number of duplicate `PartialBeaconPacket` messages each round, to every other group member.

As well as the modified logging used during the previous tests, additional logging was added to track the time spent executing relevant code sections.

Only one friendly node ran with this increased logging, so the team could identify if the logging overheads introduced their own DoS vulnerabilities.

Logging in `PartialBeacon` helps track both the amount of time spent waiting on the state lock, and time spent processing the packet in the incoming GRPC goroutine while holding the lock. (This includes signature verification, and any time spent waiting to pass to the `chainStore.newPartials` channel).

Logging in `ProcessPartialBeacon()` measures time spent adding to `chainStore.newPartials`, which could block.

The relevant code sections are listed below, and a relevant snapshot of the codebase has been provided to the development team.

```go
// PartialBeacon receives a beacon generation request and answers
// with the partial signature from this drand node.
func (d *Drand) PartialBeacon(c context.Context, in *drand.PartialBeaconPacket) (*drand.Empty,
    error) {
  before := time.Now()
  d.state.Lock()
  defer d.state.Unlock()
  lock_wait := time.Since(before)
  // could also log right now, but then would be pretty spammy with the logs

  defer func() {
    full_process := time.Since(before)
    not_waiting_on_state := full_process - lock_wait
    d.log.Error("sigp_partial_beacon", "elapsed",
      "time_waiting_for_lock", lock_wait.String(),
      "time_not_waiting_on_state", not_waiting_on_state.String(),
      "total_time", full_process.String(),
      "time_waiting_for_lock_ns", lock_wait.Nanoseconds(),
      "time_not_waiting_on_state_ns", not_waiting_on_state.Nanoseconds(),
      "total_time_ns", full_process.Nanoseconds(),
    )
  }()

  if d.beacon == nil {
    return nil, errors.New("drand: beacon not setup yet")
  }
  return d.beacon.ProcessPartialBeacon(c, in)
}
```

drand_public.go

```
// ProcessPartialBeacon receives a request for a beacon partial signature. It
// forwards it to the round manager if it is a valid beacon.
func (h *Handler) ProcessPartialBeacon(c context.Context, p *proto.PartialBeaconPacket) (*
    proto.Empty, error) {
  // ...

  before := time.Now()
  h.chain.NewValidPartial(addr, p)
  elapsed := time.Since(before)
  h.l.Error("sigp_process_partial", "add to internal chain", "time_elapsed", elapsed.String(),
      "time_elapsed_ns", elapsed.Nanoseconds())
  return new(proto.Empty), nil
}
```

core/beacon/node.go

During final retesting, logging was modified to include changes from PR #732, `PartialBeacon()`, as follows: (*NOTE*: The state lock is no longer held for the duration of `time_not_waiting_on_state`.)

```
// PartialBeacon receives a beacon generation request and answers
// with the partial signature from this drand node.
func (d *Drand) PartialBeacon(c context.Context, in *drand.PartialBeaconPacket) (*drand.Empty,
     error) {
  before := time.Now()
  d.state.Lock()
  lock_wait := time.Since(before)
  if d.beacon == nil {
    d.state.Unlock()
    return nil, errors.New("drand: beacon not setup yet")
  }
  inst := d.beacon
  d.state.Unlock()
  defer func() {
    full_process := time.Since(before)
    not_waiting_on_state := full_process - lock_wait
    d.log.Error("sigp_partial_beacon", "elapsed",
      "time_waiting_for_lock", lock_wait.String(),
      "time_not_waiting_on_state", not_waiting_on_state.String(),
      "total_time", full_process.String(),
      "time_waiting_for_lock_ns", lock_wait.Nanoseconds(),
      "time_not_waiting_on_state_ns", not_waiting_on_state.Nanoseconds(),
      "total_time_ns", full_process.Nanoseconds(),
    )
  }()
  return inst.ProcessPartialBeacon(c, in)
}
```

drand_public.go

Findings and Analysis Illustrating Remaining Issue

The test network was exercised under normal conditions, and with the malicious node broadcasting 1000 and 10000 partials per round. Log analysis indicates the following:

- A rebroadcast rate of $k = 10000$ was sufficient to halt beacon generation on the test network.

Figure 1: Time waiting on state lock

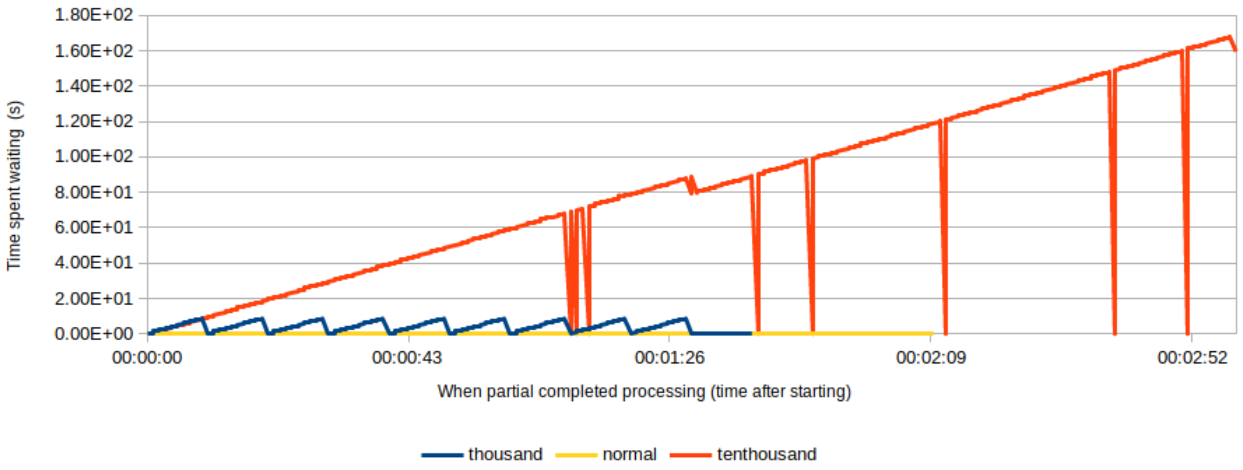
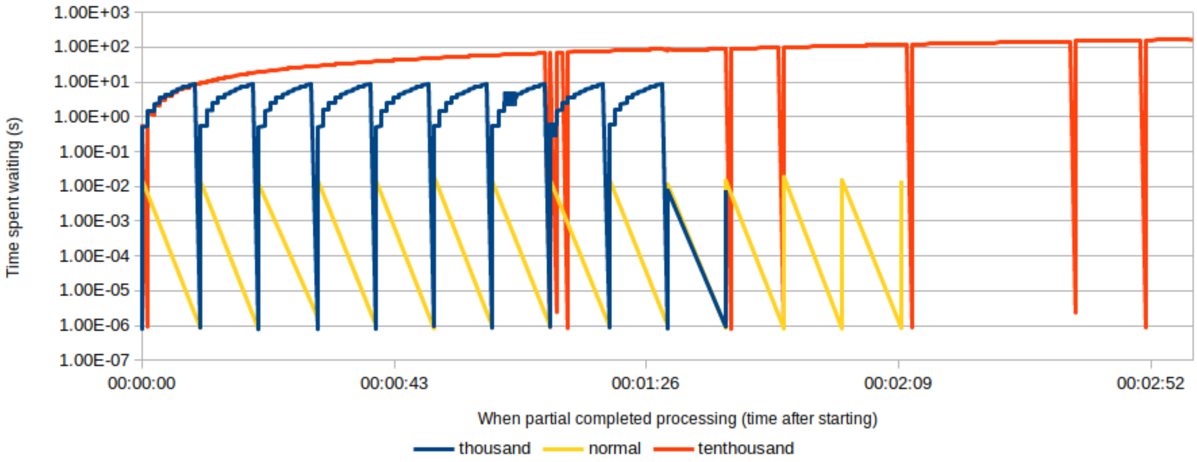


Figure 2: Time waiting on state lock (log scale)



When the rate of rebroadcasts exceeds manageable levels, each partial spends a linearly increasing amount of time waiting on the state lock (See Figure 1 and Figure 2).

- The time to process each partial is fairly stable across k and time, when excluding the time spent waiting for the state lock (See Figure 3).

  As such, an average time to process is meaningful and, from Figure C, we can see this does not differ greatly when under attack.

- From Figure 4, we can see that, although there may be some increased time spent sending to the `newPartials` channel when under attack, this is bounded. It is likely that this bound is enforced by the state lock, which ensures partials are sent synchronously.

From this, we can confirm that the state lock is the chief bottleneck, though it may mask additional ones.

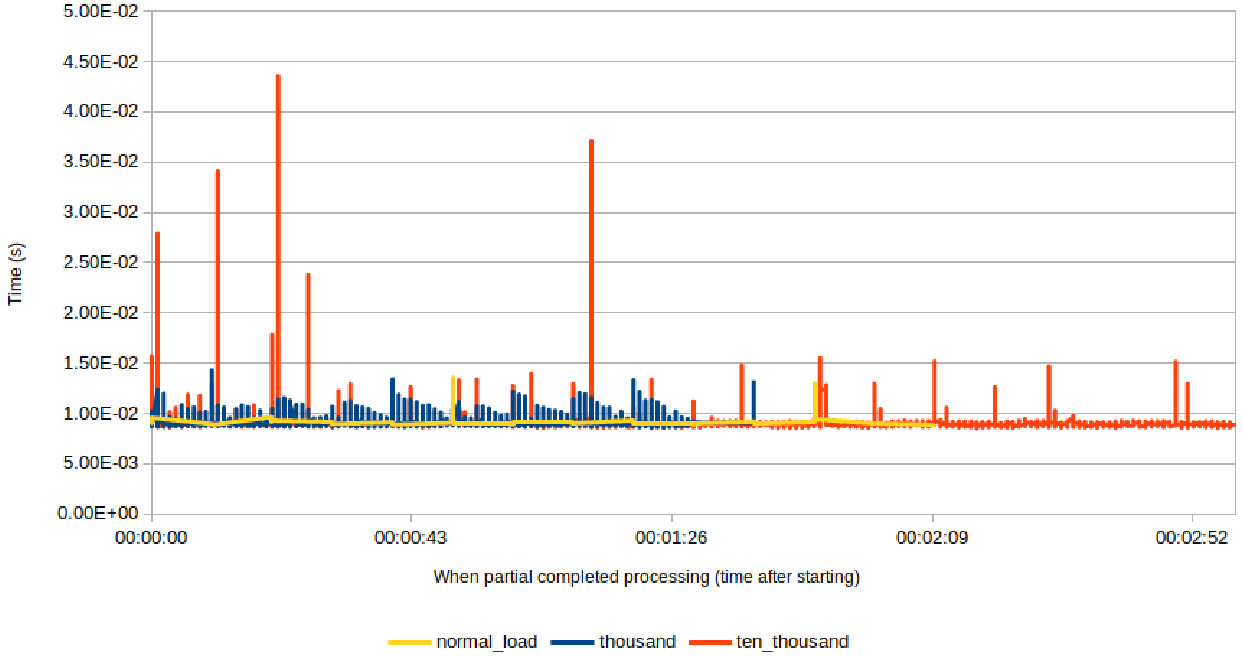
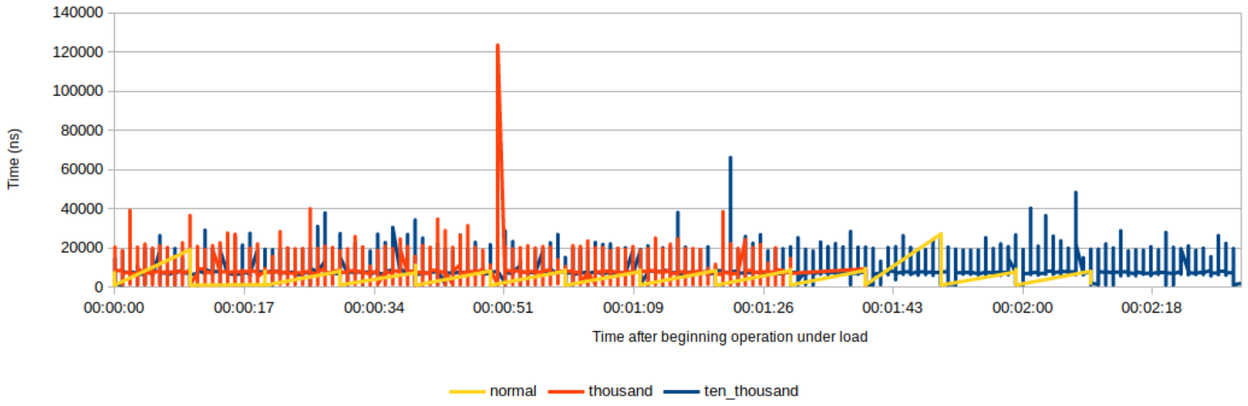


Figure 3: Time to process — excluding lock wait

	Normal load	1000 Rebroadcasts/rnd	10000 Rebroadcasts/round
Time (ms)	9.03	8.94	8.91

Table 1: Median time to process (excluding waiting on state lock)

Figure 4: Time spent sending to `newPartials`



Results Illustrating Resolution

The tests were repeated with PR #732 incorporated, to determine whether this was sufficient to resolve the susceptibility to DoS. With PR #732, the state lock is now released before performing signature verification and further processing, and was indeed found to be sufficient to adequately handle a flood of duplicate partials.

The network was still able to aggregate beacons in a timely manner while under attack. At higher rates of flooding, the additional logging had an unacceptable performance impact and could not be used as part of representative testing.

Although unable to contribute to the results displayed below, a network running unmodified code was able to continue functioning when the attacking node was configured to broadcast even 100000 and 500000 partials per round. However, as minimal difference was observed in the impact of these extreme rates, it is likely that the attacker's egress bandwidth became a bottleneck and targeted nodes received fewer partials than configured.

From the figures below, we can infer the following:

- Contention for the state lock spikes when under attack, but delays no longer expand indefinitely i.e. exhaustion does not occur (observable in Figure 5 and Figure 6).
- Figure 7 and Figure 8 illustrate that time spent processing the partials in the GRPC goroutine is no longer stable under stress.

  It should be noted, however, that these delays no longer affect state lock contention and signature verification can now be parallelised.
- Figure 9 shows that this spiking can can be attributed to sends to the `chainStore.newPartials` channel blocking.

  It should also be noted that the logging used to record these details may have a noticeable impact at these higher rates, but we can see that, because these delays do not continuously increase, the `runAggregator` is able to consume and process duplicate partials at a sufficient rate.
- Because these values spike in an unpredictable manner, comparison between averages is no longer meaningful.

Figure 5: Post PR #732: Time waiting on state lock

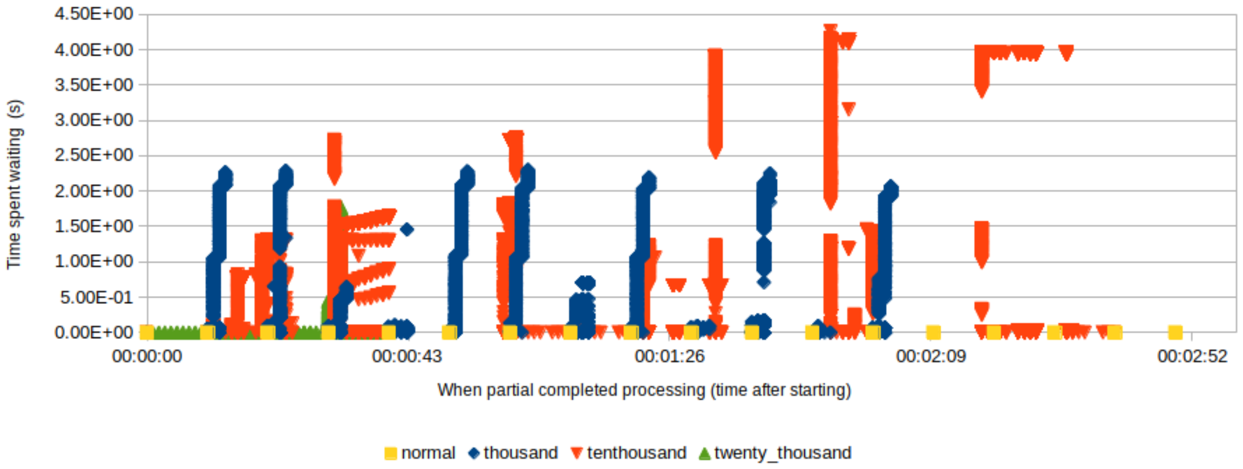


Figure 6: Post PR #732: Time waiting on state lock (log scale)

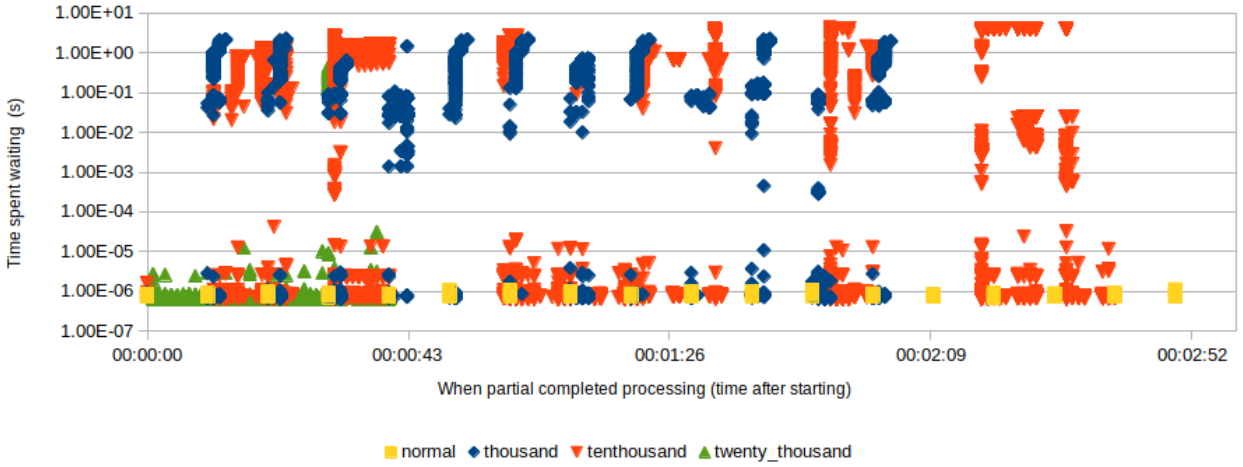


Figure 7: Post PR #732: Time to process — excluding lock wait

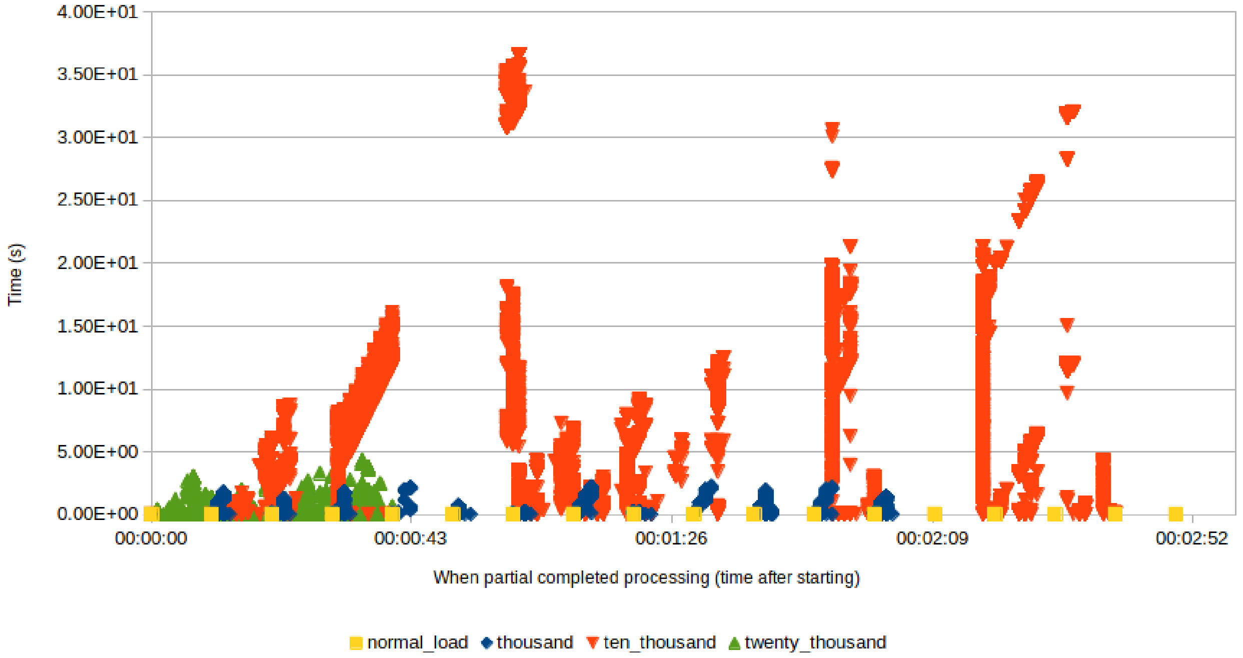


Figure 8: Post PR #732: Time to process — excluding lock wait (log scale)

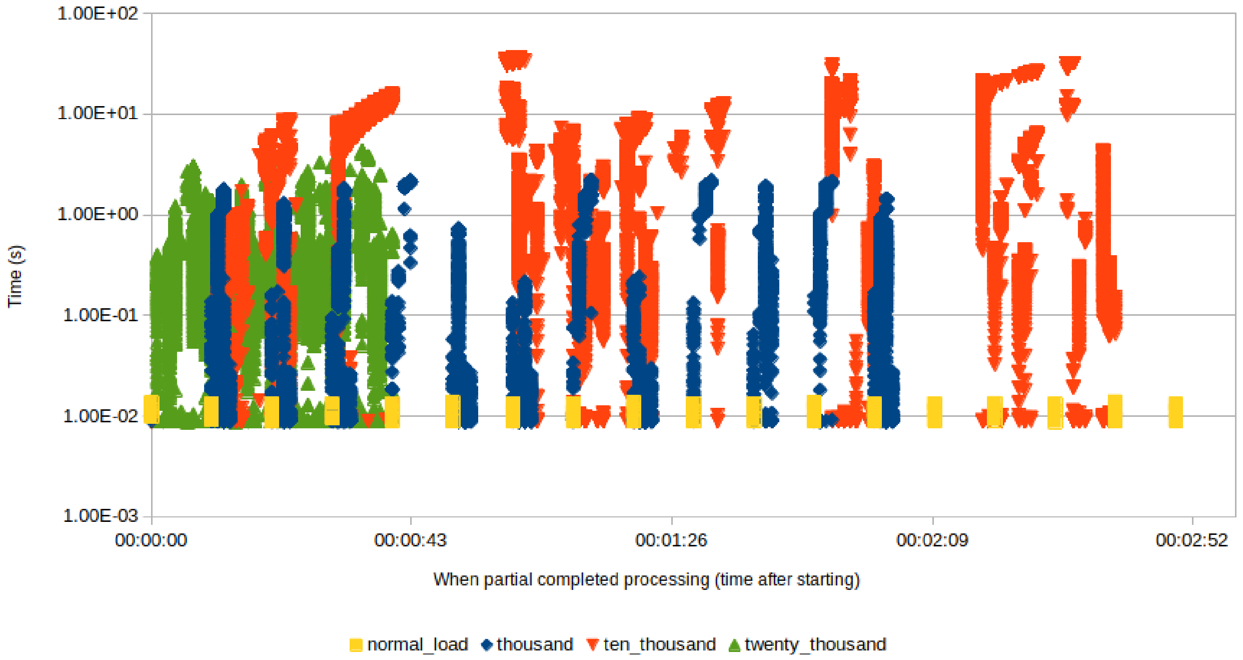
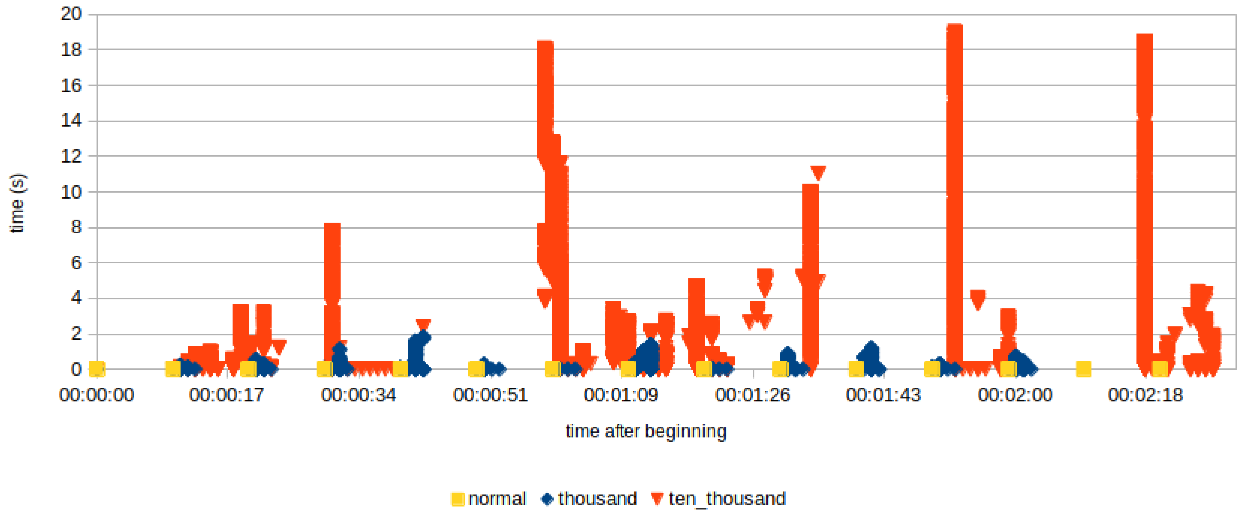


Figure 9: Post PR #732: Time spent sending to `newPartials`



Appendix D 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.

Impact	High	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Low	Low	Medium
		Low	Medium	High
			Likelihood	

Table 2: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.