HashEye

// SMART CONTRACT SECURITY AUDIT

Offchain Labs Arbitrum Quorum Changes

Security assessment by HashEye · prepared for Offchain Labs

HASHEYE AUDITED

PROJECT	Offchain Labs Arbitrum Quorum Changes
CLIENT	Offchain Labs
CATEGORY	Offchain Labs
PUBLISHED	February 1, 2026
REPORT ID	research-offchain-labs-arbitrum-quorum-changes-2026-02-01-f1mrdc

This report was produced under HashEye's layered review process — **automated detection**, **pattern correlation**, and **senior manual verification** — with every finding signed off by a human reviewer. Full findings detail and on-chain attestation are available on the report page at hasheye.io/audits/research-offchain-labs-arbitrum-quorum-changes-2026-02-01-f1mrdc.

DV Labs Charon Pedersen DKG Security Assessment February 20, 2026

Prepared for: Andrei Smirnov DV Labs

Prepared by: Jim Miller and Paul Bottinelli

HashEye

PUBLIC

Table of Contents Table of Contents 1 Project Summary 2 Executive Summary 3 Project Goals 5 Project Targets 6 Project Coverage 7 Automated Testing 8 Summary of Findings 10 Detailed Findings 11 1. Complete cluster replacement produces invalid shares 11 2. Missing threshold validation in DKG operations 13 3. Unbounded buffer allocation in the sync protocol enables denial of service 16 4. Node signature broadcast callback does not verify sender identity against claimed peer index 18 5. Share index remapping bug causes signature verification failures during DKG 21 6. Nonce reuse across multiple DKG iterations enables replay attacks 23 7. restoreCommits panics on out-of-bounds shareNum 25 8. New nodes lack polynomial commitment validation during reshare 28 9. Unbounded buffer allocation in FetchDefinition enables memory exhaustion 32 A. Vulnerability Categories 34 B. Code Quality Findings 36 C. Automated Analysis Tool Configuration 37 D. Analysis of the Flaw Uncovered in the Kyber Library 39 E. Fix Review Results 40 Detailed Fix Review Results 41 F. Fix Review Status Categories 43 About HashEye 44 Notices and Remarks 45

HashEye 1 DV Labs Charon Pedersen DKG

PUBLIC Security Assessment

Project Summary Contact Information The following project manager was associated with this project: Emily Doucette, Project Manager emily.doucette@hasheye.io The following engineering director was associated with this project: Jim Miller, Engineering Director, Cryptography james.miller@hasheye.io The following consultants were associated with this project: Jim Miller, Consultant Paul Bottinelli, Consultant james.miller@hasheye.io paul.bottinelli@hasheye.io Project Timeline The significant events and milestones of the project are listed below. Date Event January 26, 2026 Pre-project kickoff call February 6, 2026 Delivery of report draft February 6, 2026 Report readout meeting February 19, 2026 Completion of fix review February 20, 2026 Delivery of final comprehensive report

HashEye 2 DV Labs Charon Pedersen DKG

PUBLIC Security Assessment

Executive Summary Engagement Overview DV Labs engaged HashEye to review the security of the Pedersen DKG for the Charon distributed validator client. The DKG package is an important piece of Charon’s distributed validator infrastructure. Its primary purpose is to generate and manage threshold BLS signing keys for Ethereum staking. The package supports two main operational modes: initial key generation for new clusters, and resharing operations that allow existing clusters to rotate their key shares, add new operators, remove departing operators, or replace operators while preserving the original validator public keys registered on Ethereum. A team of two consultants conducted the review from January 26 to February 6, 2026, for a total of two engineer-weeks of effort. Our testing focused on reviewing the dkg folder to ensure the secure use of the Kyber DKG library and protection against common DKG threats, such as key share compromise, rogue public key attacks, and general denial-of-service attack vectors. With full access to the source code and documentation, we conducted static and dynamic testing of the dkg folder, using both automated and manual testing methods. Observations and Impact Overall, we found the codebase to be well written, well organized, and easy to read. We ran several Go static analysis tools to identify common vulnerabilities and unidiomatic programming practices, but none produced any meaningful findings. Moreover, we found the existing test coverage to be strong. Our review focused on identifying instances of severe DKG vulnerabilities that could compromise key material or enable other attacks, such as rogue public key attacks. We did not identify any issues of this kind or of similar severity. The issues we uncovered primarily pertain to denial-of-service flaws that would allow a malicious node to prevent the DKG protocols from completing. Recommendations Based on the findings identified during the security review, HashEye recommends that DV Labs take the following steps: • Remediate the findings disclosed in this report. These findings should be addressed through direct fixes or broader refactoring efforts. • Complete the migration away from the Kryptology dependency. This review focused on the parts of the codebase that rely on the Kyber DKG codebase. Other parts of the codebase currently rely on Kryptology for performing FROST operations.

HashEye 3 DV Labs Charon Pedersen DKG

PUBLIC Security Assessment

The DV Labs team indicated that it plans to transition away from Kryptology because it is not well maintained. We support and encourage the DV Labs team to complete this migration plan. • Consider investing in more adversarial testing to prevent additional denial-of-service attack vectors. Denial-of-service attacks are challenging to fully prevent in DKG implementations due to the large number of messages that need to be processed and their complex structure. Investing in additional adversarial testing could help uncover other, more subtle denial-of-service vectors. Finding Severities and Categories

The following tables provide the number of findings by severity and category. EXPOSURE ANALYSIS Severity Count Medium 6 Low 2 Informational 1

CATEGORY BREAKDOWN Category Count Cryptography 1 Data Validation 8

HashEye 4 DV Labs Charon Pedersen DKG

PUBLIC Security Assessment

Project Goals The engagement was scoped to provide a security assessment of the Charon Pedersen DKG implementation. Specifically, we sought to answer the following non-exhaustive list of questions: • Does the Charon DKG implementation securely use the Kyber DKG library? • Is the DKG protocol susceptible to any denial-of-service attacks, where a malicious node can cause honest nodes to be evicted or otherwise prevent an honest group from completing the DKG? • Is the DKG protocol susceptible to any attacks that could compromise private key shares? • Does the codebase properly protect the protocol against replay attacks carried out during previous DKG steps or iterations? • Is the logic for adding, removing, and replacing operators implemented correctly and securely? • Does the codebase correctly and securely set up broadcast channels for performing each round of the DKG protocol? • Does the implementation perform all critical validations of inputs, such as configuration files and individual messages from peers?

HashEye 5 DV Labs Charon Pedersen DKG

PUBLIC Security Assessment

Project Targets The engagement involved reviewing and testing the following target. Charon DKG

Repository https://github.com/ObolNetwork/charon Version 7a67409dee74e52ce95e7a8c531cdd25b237841e Type Go Platform Multiple

HashEye 6 DV Labs Charon Pedersen DKG

PUBLIC Security Assessment

Project Coverage This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches included the following: • We performed a manual review of the Pedersen DKG and resharing protocol implementations in the charon/dkg folder. This review focused on the secure use of the Kyber library, correct error and edge-case handling, identification of potential denial-of-service attack vectors, secure broadcast and message transport, protection against replay attacks, and other relevant cryptographic attack scenarios. • We performed an automated review of the charon/dkg folder using several Go static analyzers. For more information, see the Automated Testing section and appendix C. Coverage

Limitations Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. During this project, we were unable to perform comprehensive testing of the following system elements, which may warrant further review: • FROST implementation inside of charon/dkg • The Kyber open-source library, which implements the Pedersen DKG protocol

HashEye 7 DV Labs Charon Pedersen DKG

PUBLIC Security Assessment

Automated Testing HashEye uses automated techniques to extensively test the security properties of software. We use both open-source static analysis and fuzzing utilities, along with in-house-developed tools, to automate testing of source code and compiled software. Test Harness Configuration We used the following tools in the automated testing phase of this project: Tool Description Policy CodeQL A code analysis engine developed by GitHub to automate security checks Appendix C.1 go-mod-outdated A tool that identifies outdated Go dependencies Appendix C.2 go-test A subcommand for the Go compiler that can be used to generate test coverage data for the codebase Appendix C.3 golangci-lint A meta-linter for Go that runs multiple linters and static analysis tools on the codebase and aggregates the results Appendix C.4 govulncheck The official Go vulnerability scanner that checks for known vulnerabilities in dependencies Appendix C.5 Semgrep An open-source static analysis tool for finding bugs and enforcing code standards when editing or committing code, and during build time Appendix C.6 Areas of Focus Our automated testing and verification work focused on answering the following: • Does the codebase contain known vulnerabilities or weaknesses identified by golangci-lint, CodeQL, and Semgrep? • Does the codebase have any outdated or vulnerable dependencies? • Is the unit and integration test coverage across the codebase sufficient?

HashEye 8 DV Labs Charon Pedersen DKG

PUBLIC Security Assessment

Test Results

The results of this focused testing are detailed below. Known vulnerable code patterns: To identify unidiomatic code patterns and known vulnerabilities, we ran golangci-lint, CodeQL, and Semgrep on the codebase. Property Tool Result Does the codebase contain unidiomatic code patterns or weaknesses identified by code linters? golangci-lint Passed Does the codebase contain known vulnerable code patterns identified by static analysis tools? CodeQL, Semgrep Passed

Dependency management: We used govulncheck and go-mod-outdated to identify outdated and vulnerable project dependencies. Property Tool Result Does the project have outdated dependencies? go-mod-outdated Passed Does the project have dependencies with known vulnerabilities that affect the security of the codebase? govulncheck Passed

Unit and integration test coverage: We used the Go compiler and the Go cover tool to review the unit and integration test coverage across the codebase. Property Tool Result Is the unit and integration test coverage sufficient? go-test Passed

HashEye 9 DV Labs Charon Pedersen DKG

PUBLIC Security Assessment

Summary of Findings The table below summarizes the findings of the review, including details on type and severity. ID Title Type Severity 1 Complete cluster replacement produces invalid shares Data Validation Medium 2 Missing threshold validation in DKG operations Data Validation Low 3 Unbounded buffer allocation in the sync protocol enables denial of service Data Validation Medium 4 Node signature broadcast callback does not verify sender identity against claimed peer index Data Validation Informational 5 Share index remapping bug causes signature verification failures during DKG Data Validation Medium 6 Nonce reuse across multiple DKG iterations enables replay attacks Cryptography Medium 7 restoreCommits panics on out-of-bounds shareNum Data Validation Medium 8 New nodes lack polynomial commitment validation during reshare Data Validation Medium 9 Unbounded buffer allocation in FetchDefinition enables memory exhaustion Data Validation Low

HashEye 10 DV Labs Charon Pedersen DKG

Detailed Findings 1. Complete cluster replacement produces invalid shares Severity: Medium Difficulty: Medium Type: Data Validation Finding ID: TOB-CHARON-1 Target: dkg/pedersen/reshare.go

Description During complete cluster replacement (which the current implementation allows), the Pedersen reshare protocol does not validate that sufficient nodes remain after removal operations to maintain enough validators to reconstruct the secret. When all nodes from an initial DKG are removed and replaced with entirely new nodes, the reshare protocol completes without error but generates invalid validator shares. The protocol validates that at least one node with existing shares participates in the reshare process, but does not ensure that nodes remain in the cluster after the removal completes. While the kyber-dkg protocol generates new secret shares, the resulting PublicShares map entries do not correspond to the derived secret shares because the reshare protocol cannot maintain proper continuity when the entire original cluster is removed. The validation checks only that at least one old node participates in the reshare process (i.e., it returns an error if there are no oldNodes, as shown in figure 1.1), but does not verify that any nodes remain in the cluster after the removal completes. // Validate node classification if len(config.Reshare.RemovedPeers) > 0 && len(oldNodes) == 0 { return nil, errors.New("remove operation requires at least one node with existing shares to participate") } Figure 1.1: Insufficient validation for removal operations (dkg/pedersen/reshare.go#L161-L164) In a complete cluster replacement scenario, old nodes participate (satisfying this check), but all nodes are also marked for removal, violating the threshold requirement. The function returns successfully without detecting this condition. Testing reveals that after complete cluster replacement, attempting to verify a signature created with the secret share using the corresponding entry from the public shares map fails with a “signature not verified” error. Notably, the kyber-dkg library does provide protection against some removal operations.

When nodes are removed so that some old nodes, but fewer than the threshold, remain, the reshare protocol correctly fails during initialization with the error “not enough good public shares to reconstruct secret commitment.” This occurs because the restoreDistKeyShare function requires at least a threshold amount of shares to reconstruct polynomial commitments. However, when all old nodes are removed, the reconstruction code path is never executed, allowing the protocol to bypass this check entirely and complete with invalid output (see figure 1.2). // Restoring DistKeyShares from Charon shares distKeyShares := make([]*kdkg.DistKeyShare, 0)

```
for i := range len(shares) { dks, err := restoreDistKeyShare(shares[i], config.Threshold, thisNodeIndex) if err != nil { return nil, errors.Wrap(err, "restore distkeyshare", z.Int("validator_index", i)) }
```

distKeyShares = append(distKeyShares, dks) } Figure 1.2: Conditional execution of share reconstruction (dkg/pedersen/reshare.go#L89-L99) This creates two distinct failure modes: partial below-threshold removal fails correctly, while complete cluster replacement succeeds incorrectly.

Exploit Scenario An operator migrates their validator infrastructure from four nodes to five new nodes by executing a single reshare operation that removes all original nodes and adds the new ones. The reshare command completes successfully and logs “Pedersen reshare completed” without errors. The operator deploys the new cluster with the generated shares, confident that the migration succeeded. When the validator attempts to sign its first attestation, signature verification fails because the public key shares do not match the secret shares, causing the validator to miss all duties and incur inactivity penalties that escalate to potential slashing. Recommendations Short term, add validation to ensure that enough nodes (i.e., more than the threshold) from the original cluster will remain after the removal operation completes. Specifically, verify that the number of old nodes not being removed is at least one, preventing complete cluster replacement. Long term, add integration tests that verify that reshare output produces cryptographically valid shares that can successfully sign and verify threshold signatures.

2. Missing threshold validation in DKG operations Severity: Low Difficulty: Low Type: Data Validation Finding ID: TOB-CHARON-2 Target: dkg/pedersen/dkg.go, dkg/pedersen/reshare.go

Description Both the RunDKG and RunReshareDKG functions fail to verify that the threshold value is achievable given the number of nodes in the cluster. The functions accept config.Threshold and config.Reshare.NewThreshold, respectively, without checking whether these values exceed the node count, are zero or negative, or meet Byzantine fault tolerance requirements. In RunDKG, the function creates the KDKG configuration without any threshold validation (figure 2.1). dkgConfig := &kdkg.Config{ Longterm: nodePrivateKey, Nonce: nonce, Suite: config.Suite, NewNodes: nodes, Threshold: config.Threshold, FastSync: true, Auth: drandbls.NewSchemeOnG2(kbls.NewBLS12381Suite()), Log: newLogger(log.WithTopic(ctx, "pedersen")), } Figure 2.1: Unvalidated threshold in initial DKG (dkg/pedersen/dkg.go#L59-L68) In RunReshareDKG, the existing validation logic checks operational constraints such as whether removed peers have shares and whether added peers are genuinely new, but omits threshold validation (figure 2.2). // Validate node classification if len(config.Reshare.RemovedPeers) > 0 && len(oldNodes) == 0 { return nil, errors.New("remove operation requires at least one node with existing shares to participate") }

if len(config.Reshare.AddedPeers) > 0 && len(newNodes) <= len(oldNodes) { return nil, errors.New("add operation requires new nodes to join, but all nodes already exist in the cluster") }

// In add operations, old nodes must have shares to contribute // (new nodes being added won't have shares, which is expected) if len(config.Reshare.AddedPeers) > 0 && !thisIsAddedNode && len(distKeyShares) == 0 { return nil, errors.New("existing node in add operation must have shares to contribute") } Figure 2.2: Existing validation logic that omits threshold checks (dkg/pedersen/reshare.go#L161-L174) The RunReshareDKG function then passes the unvalidated threshold directly to the Kyber DKG configuration. Three important validations are currently missing: 1. Upper bound check: The threshold must not exceed the number of nodes. 2. Lower bound check: The new threshold must be at least one. 3. Byzantine fault tolerance check: For proper Byzantine fault tolerance security, the threshold should exceed half the number of nodes. The underlying kyber-dkg library provides partial protection against invalid thresholds. When a threshold of zero is provided, kyber-dkg automatically defaults to MinimumT(len(newNodes)), which equals (n >> 1) + 1 for Byzantine fault tolerance. Additionally, during polynomial reconstruction, the library verifies that enough shares are present and returns an error, “not enough shares to recover private polynomial,” if the threshold exceeds the number of available shares. However, these protections do not prevent all misconfigurations. The library does not validate upper bounds (thresholds exceeding the node count) at runtime when performing operations that require share reconstruction, and it does not prevent operators from explicitly setting thresholds below the Byzantine fault tolerance level. The lack of up-front validation at the Charon layer means configuration errors propagate through the system and fail at unpredictable points during protocol execution rather than being caught immediately during setup.

Exploit Scenario An operator performs an initial DKG for a four-node cluster with a misconfigured threshold of five. The DKG operation completes successfully without detecting the invalid configuration. When validators attempt to sign attestations, they fail because five nodes are required to reconstruct signatures, but only four exist, resulting in missed attestations, validator penalties, and cluster inoperability until a new DKG is performed with validator exit and reentry. Recommendations Short term, add comprehensive threshold validation in both RunDKG and RunReshareDKG before creating the KDKG configuration. The validation should verify that the threshold is

between 1 and the node count, and ideally exceeds half the node count for Byzantine fault tolerance. Long term, implement unit tests for all edge cases in both initial DKG and reshare operations, including zero thresholds, thresholds exceeding the node count, thresholds equal to the node count, and thresholds that violate Byzantine fault tolerance requirements.

3. Unbounded buffer allocation in the sync protocol enables denial of service Severity: Medium Difficulty: Low Type: Data Validation Finding ID: TOB-CHARON-3 Target: dkg/sync/server.go

Description The readSizedProto function in the DKG sync server reads an attacker-controlled size value from the network and uses it directly to allocate memory without any bounds checking. An unauthenticated attacker can crash any DKG node by sending a malicious 8-byte prefix, causing the node to attempt allocation of up to 9 exabytes of memory. The sync protocol coordinates progress between all participating nodes during the DKG ceremony. When a peer connects, the server reads incoming messages using readSizedProto, which first reads an 8-byte little-endian integer representing the message size and then allocates a buffer of that size without validation (figure 3.1). // readSizedProto reads a size prefixed proto message. func readSizedProto(reader io.Reader, msg proto.Message) error { var size int64

err := binary.Read(reader, binary.LittleEndian, &size) if err != nil { return errors.Wrap(err, "read size") }

buf := make([]byte, size) Figure 3.1: Unbounded allocation from attacker-controlled size value (dkg/sync/server.go#L364-L373) The function is called from handleStream before any authentication or validation occurs (figure 3.2). The signature verification in validReq() is never reached because the crash occurs before the message is fully read. // handleStream serves a new long-lived client connection. func (s *Server) handleStream(ctx context.Context, stream network.Stream) error { ...

for { if ctx.Err() != nil { return ctx.Err() }

// Read next sync message msg := new(pb.MsgSync) if err := readSizedProto(stream, msg); err != nil { return err }

...

if err := s.validReq(pubkey, msg); err != nil { Figure 3.2: Message reading occurs before authentication. (dkg/sync/server.go#L247-L280) The vulnerability is reachable from external input without authentication. The sync protocol is registered via SetStreamHandler, which accepts connections from any peer that knows the protocol ID. The attacker only needs to connect to the target node’s libp2p address and request the sync protocol.

Exploit Scenario An attacker identifies a node participating in a DKG ceremony by querying libp2p relays or the DHT. The attacker establishes a libp2p connection and requests the sync protocol. The attacker then sends 8 bytes representing the maximum int64 value in little-endian format (\xff\xff\xff\xff\xff\xff\xff\x7f). The server reads this as size = 9223372036854775807 and attempts to execute make([]byte, 9223372036854775807). The Go runtime cannot allocate 9 exabytes of memory, causing an immediate out-of-memory panic or process termination. Because all nodes must participate to generate the threshold key, the DKG ceremony fails. The attacker can target all cluster nodes to guarantee ceremony failure. Recommendations Short term, add bounds checking before the buffer is allocated that ensures the size is within a reasonable range (for example, 10 MB) and rejects messages with negative or excessive sizes. Long term, refactor the protocol to perform authentication before reading message content, and implement connection-level resource limits via libp2p’s resource manager.

4. Node signature broadcast callback does not verify sender identity against claimed peer index Severity: Informational Difficulty: High Type: Data Validation Finding ID: TOB-CHARON-4 Target: dkg/nodesigs.go

Description The broadcastCallback function in the node signature exchange ignores the actual sender’s peer ID and trusts the PeerIndex field from the untrusted message. This does not appear exploitable, as the reliable broadcast protocol’s deduplication mechanism prevents a malicious party from submitting multiple messages. The node signature exchange occurs during step 5 of the DKG ceremony, where each operator signs the lock hash with their K1 (secp256k1) private key. These signatures are collected and stored in the final lock file. The broadcastCallback function receives the actual sender’s peer ID from the P2P layer but explicitly ignores it using Go’s blank identifier (figure 4.1). // broadcastCallback is the default bcast.Callback for nodeSigBcast. func (n *nodeSigBcast) broadcastCallback(ctx context.Context, _ peer.ID, _ string, msg proto.Message) error { nodeSig, ok := msg.(*dkgpb.MsgNodeSig) if !ok { return errors.New("invalid node sig type") }

msgPeerIdx := int(nodeSig.GetPeerIndex()) Figure 4.1: Sender identity ignored in favor of untrusted message field (dkg/nodesigs.go#L120-L127) This allows any cluster peer to claim any PeerIndex in their message. The only validation is bounds checking, not sender identity verification (figure 4.2). A special 4-byte value noneData = []byte{0xde, 0xad, 0xbe, 0xef} bypasses all signature verification and is stored immediately without any cryptographic checks. sig := nodeSig.GetSignature() if bytes.Equal(sig, noneData) { // For certain protocols we allow exchanging nil signatures. n.setSig(noneData, msgPeerIdx)

HashEye 18 DV Labs Charon Pedersen DKG

PUBLIC Security Assessment

return nil } Figure 4.2: The noneData bypass skips all signature verification. (dkg/nodesigs.go#L129-L135) However, the reliable broadcast protocol enforces deduplication based on the tuple containing the sender peer ID and message ID. Each peer can successfully broadcast only one message per message ID; subsequent messages with different content are rejected (figure 4.3). // dedupKey ensures only a single hash is signed per peer and message ID. // Ie. byzantine peer cannot broadcast different hashes for the same message ID. type dedupKey struct { PeerID peer.ID MsgID string } ... func (s *server) dedupHash(pID peer.ID, msgID string, hash []byte) error { s.mu.Lock() defer s.mu.Unlock()

key := dedupKey{PeerID: pID, MsgID: msgID}

prevHash, ok := s.dedup[key] if ok && !bytes.Equal(prevHash, hash) { return errors.New("duplicate ID, mismatching hash") }

s.dedup[key] = hash

return nil } Figure 4.3: Broadcast deduplication prevents multiple messages per peer. (dkg/bcast/server.go#L47-L103) This means a malicious peer attempting to impersonate another peer must use their single broadcast opportunity for the attack message, leaving their own signature slot unfilled. The ceremony would then hang waiting for the attacker’s legitimate signature, which is equivalent to the attacker simply refusing to participate.

Exploit Scenario A 5-node cluster begins a DKG ceremony. In step 5 (node signature exchange), malicious peer 2 attempts to disrupt peer 4 by broadcasting a message containing noneData (0xdeadbeef) with PeerIndex set to 4. This message overwrites peer 4’s slot with noneData.

HashEye 19 DV Labs Charon Pedersen DKG

PUBLIC Security Assessment

Recommendations Short term, update the broadcastCallback function to verify that the actual sender’s peer ID matches the claimed peer index before processing the message, to look up the peer ID in the peers list and compare its index to the claimed msgPeerIdx, and to reject messages where the sender’s identity does not match the claimed index. Long term, implement signature slot immutability so that once a valid signature is stored for a peer index, it cannot be overwritten.

HashEye 20 DV Labs Charon Pedersen DKG

PUBLIC Security Assessment

5. Share index remapping bug causes signature verification failures during DKG Severity: Medium Difficulty: Low Type: Data Validation Finding ID: TOB-CHARON-5 Target: dkg/pedersen/dkg.go

Description The processKey function in the Pedersen DKG implementation incorrectly maps public key shares to sequential indices instead of their actual share indices. When noncontiguous nodes participate in a DKG or reshare ceremony (for example, when some nodes are offline), the wrong public key is looked up during signature verification, causing the ceremony to fail with an ungraceful error message. During DKG, each node generates a key share with a specific 1-indexed share index. The processKey function builds a PublicShares map that should map each share index to its corresponding public key. However, the loop uses the sequential loop variable i+1 instead of the actual share index oi (figure 5.1). for i, oi := range oldShareIndices { var pk tbls.PublicKey copy(pk[:], oldShareRevMap[oi]) publicShares[i+1] = pk } Figure 5.1: Incorrect index mapping in processKey (dkg/pedersen/dkg.go#L184-L188) When nodes at indices 2 and 4 are offline in a 5-node cluster, oldShareIndices equals [1, 3, 5], and the loop stores the following: • publicShares[1] = pk1 (correct by coincidence) • publicShares[2] = pk3 (wrong; should be at index 3) • publicShares[3] = pk5 (wrong; should be at index 5) During signature verification, each partial signature carries the signer's actual share index. In the above example, when verifying a signature from node 3, the code looks up PublicShares[3], which returns the public key for index 5 instead of 3. As shown in figure 5.2, the verification fails because the signature is verified against the incorrect key. The error message returned will not indicate what caused the verification failure, and it will be difficult to identify the root cause.

HashEye 21 DV Labs Charon Pedersen DKG

PUBLIC Security Assessment

pubshare, ok := sh.PublicShares[s.ShareIdx] if !ok { return tbls.Signature{}, nil, errors.New("invalid pubshare") }

err = tbls.Verify(pubshare, hash, sig) Figure 5.2: Wrong public key used for verification (dkg/dkg.go#L782-L787) This bug primarily affects reshare operations, which always use Pedersen DKG via RunReshareDKG.

Exploit Scenario A 5-node cluster initiates a reshare operation to change the operator set. Two nodes (at indices 2 and 4) are offline or have network issues, leaving nodes at indices 1, 3, and 5 participating. The Pedersen DKG completes and generates key shares, but the PublicShares map is corrupted due to the index remapping bug. When the nodes exchange and verify lock hash signatures, signature verification fails because the wrong public key is used. The protocol fails with an error that is not straightforward to recover from. Recommendations Short term, fix the loop in processKey to use the share index instead of the sequential loop variable. Specifically, replace publicShares[i+1] = pk with publicShares[oi] = pk to ensure each public key is stored at its correct share index. Long term, consider adding additional explicit validation that the PublicShares map contains entries for all expected share indices before proceeding with signature operations.

HashEye 22 DV Labs Charon Pedersen DKG

PUBLIC Security Assessment

6. Nonce reuse across multiple DKG iterations enables replay attacks Severity: Medium Difficulty: Medium Type: Cryptography Finding ID: TOB-CHARON-6 Target: dkg/pedersen/dkg.go

Description The RunDKG function reuses the same nonce value across all DKG iterations when generating keys for multiple validators. The Kyber DKG library explicitly requires nonces to be unique across DKG runs to prevent replay attacks, but Charon generates the nonce once before the loop and passes the same value to every iteration. When a cluster operator configures multiple validators (numVals > 1), RunDKG executes multiple sequential DKG ceremonies, one per validator key. The nonce is generated before the loop and stored in the dkgConfig struct, which is then reused for each iteration (figure 6.1). nonce, err := generateNonce(nodes) if err ≠ nil { return nil, err }

dkgConfig := &kdkg.Config{ Longterm: nodePrivateKey, Nonce: nonce, Suite: config.Suite, NewNodes: nodes, Threshold: config.Threshold, FastSync: true, Auth: drandbls.NewSchemeOnG2(kbls.NewBLS12381Suite()), Log: newLogger(log.WithTopic(ctx, "pedersen")), }

... for range numVals { phaser := kdkg.NewTimePhaser(config.PhaseDuration)

protocol, err := kdkg.NewProtocol(dkgConfig, board, phaser, false,)

HashEye 23 DV Labs Charon Pedersen DKG

PUBLIC Security Assessment

Figure 6.1: Nonce reused across all DKG iterations (dkg/pedersen/dkg.go#L54-L82) The Kyber library documentation explicitly states that nonces must be unique to prevent replay attacks (figure 6.2). // Nonce is required to avoid replay attacks from previous runs of a DKG / // resharing. The required property of the Nonce is that it must be unique // accross runs. A Nonce

must be of length 32 bytes. User can get a secure // nonce by calling `GetNonce()`. Nonce []byte Figure 6.2: Kyber library nonce requirements (kyber/share/dkg/pedersen/dkg.go#L103-L107) The Kyber DKG protocol uses the nonce to compute a session identifier that binds all protocol messages to a specific DKG instance. When the same nonce is reused, messages from one DKG iteration become valid for subsequent iterations. A malicious participant can capture messages from an honest party during iteration N and replay them in iteration N+1. When the honest party attempts to send its legitimate message in iteration N+1, the protocol detects two messages with the same session identifier and flags the honest party as equivocating. While the sequential exchange phase prevents accidental message collisions between concurrent iterations, it does not prevent a malicious party from intentionally storing and replaying messages. Each DKG iteration must complete before the next begins, but a malicious node receives and stores all messages from iteration N before iteration N+1 starts, giving it the opportunity to replay those messages at the start of the next iteration.

Exploit Scenario A malicious party replays messages from another party that were sent in a previous DKG iteration. Because the same nonce value is reused across DKG iterations for multiple validator keys, these replayed messages will be considered valid. When the honest party attempts to submit a valid, non-replayed message, they will be wrongfully flagged as equivocating, which will cause the protocol to fail. Recommendations Short term, incorporate the iteration index into the nonce computation so that a unique nonce is generated for each DKG iteration. In addition, update dkg/pedersen/board.go to have the deal, response, and justification message bundle handlers validate that the proper SessionID is being used (currently, this check exists in the node public key and validator public key share message handlers, but not those other handlers). Long term, update the nonce computation to include domain separation between each value. For defense in depth, consider serializing and prepending the length of each item in the hash computation, or use a multihash scheme such as TupleHash to prevent nonce collisions.

7. restoreCommits panics on out-of-bounds shareNum Severity: Medium Difficulty: Medium Type: Data Validation Finding ID: TOB-CHARON-7 Target: dkg/pedersen/reshare.go

Description The restoreCommits function in the DKG Pedersen resharing module lacks bounds checking before accessing slice elements, causing it to panic rather than returning an error when the shareNum parameter exceeds the number of available shares. This vulnerability is exploitable during cluster reshare operations when new nodes are being added. During a reshare operation, when a new node (one being added to the cluster) attempts to restore public coefficients from exchanged public key shares, it calls restoreCommits with a shareNum index. The function iterates over all nodes' public key shares and extracts the share at the given index without first validating that the index is within bounds (figure 7.1). func restoreCommits(publicShares map[int][][]byte, shareNum, threshold int) ([]kyber.Point, error) { // Extract the specific share's public keys for all nodes pubSharesBytes := make(map[int][]byte) for nodeIdx, pks := range publicShares { pubSharesBytes[nodeIdx] = pks[shareNum] }

return restoreCommitsFromPubShares(pubSharesBytes, threshold) } Figure 7.1: The vulnerable restoreCommits function (dkg/pedersen/reshare.go#L348-L356) The publicShares map is populated by makeNodes, which collects public key share data exchanged between nodes via the Board's broadcast mechanism. No validation exists to verify that the number of shares provided by each node matches the expected TotalShares configuration value (figure 7.2). func makeNodes(ctx context.Context, config *Config, board *Board) ([]kdkg.Node, map[int][][]byte, error) { ...

pubKeyShares := make(map[int][][]byte, 0)

for i := range nodePubKeys {

ppk := nodePubKeys[i] ...

if len(ppk.PubKeyShares) > 0 { shares := make([][]byte, len(ppk.PubKeyShares)) copy(shares, ppk.PubKeyShares) pubKeyShares[index] = shares }

... }

return nodes, pubKeyShares, nil } Figure 7.2: The makeNodes function copies shares without validating the expected count. (dkg/pedersen/dkg.go#L111-L143) The restoreCommits function is called within a for loop that iterates over TotalShares. This for loop does not validate the pubKeyShares length either (figure 7.3). for shareNum := range config.Reshare.TotalShares { ...

if isNodeWithExistingShares { // This node has existing shares to contribute to the reshare reshareConfig.Share = distKeyShares[shareNum] reshareConfig.PublicCoeffs = nil } else { // This is a new node - restore public coefficients from exchanged public key shares commits, err := restoreCommits(pubKeyShares, shareNum, config.Threshold) if err ≠ nil { return nil, errors.Wrap(err, "restore commits") }

reshareConfig.Share = nil reshareConfig.PublicCoeffs = commits } Figure 7.3: The loop iterates over TotalShares without validating pubKeyShares lengths. (dkg/pedersen/reshare.go#L201-L221) Exploit Scenario A malicious node participates in a cluster reshare operation where new nodes are being added. When public key shares are being shared, the malicious node modifies its client to broadcast fewer shares than expected. New nodes being added receive this truncated data via the Board mechanism and store it in their pubKeyShares map. When these nodes process shares by calling restoreCommits for a shareNum that exceeds the truncated list of shares, this triggers a Go runtime panic with “index out of range,” crashing the new

node. The attacker can repeatedly perform this attack to prevent any new nodes from successfully joining the cluster. Recommendations Short term, add bounds checking in restoreCommits before the slice element is accessed. The function should validate that shareNum is within the valid range for all entries in publicShares and return a descriptive error if any node provides insufficient shares. Long term, implement comprehensive input validation at the boundaries where data is received from other nodes. For instance, the makeNodes function should validate that all old nodes provide exactly TotalShares public key shares before returning, rejecting malformed or incomplete data early.

8. New nodes lack polynomial commitment validation during reshare Severity: Medium Difficulty: Medium Type: Data Validation Finding ID: TOB-CHARON-8 Target: dkg/pedersen/reshare.go

Description When a new node joins a cluster through a reshare operation, it must reconstruct the public polynomial commitments from public key shares broadcast by existing nodes. The restoreCommits function performs this reconstruction using Lagrange interpolation, but it does not validate that the recovered group public key matches the expected validator public key from the cluster lock (figure 8.1). func restoreCommits(publicShares map[int][][]byte, shareNum, threshold int) ([]kyber.Point, error) { // Extract the specific share's public keys for all nodes pubSharesBytes := make(map[int][]byte) for nodeIdx, pks := range publicShares { pubSharesBytes[nodeIdx] = pks[shareNum] }

return restoreCommitsFromPubShares(pubSharesBytes, threshold) } Figure 8.1: The restoreCommits function passes public shares directly to polynomial recovery without validation. (dkg/pedersen/reshare.go#L348-L356) The restoreCommitsFromPubShares function, which restoreCommits calls to perform the polynomial recovery, also returns the recovered commitments without validation (figure 8.2). // restoreCommitsFromPubShares recovers public polynomial commits from a map of public key shares. // The nodeIdx in the map is 0-indexed. func restoreCommitsFromPubShares(pubSharesBytes map[int][]byte, threshold int) ([]kyber.Point, error) { var (suite = kbls.NewBLS12381Suite() kyberPubShares []*kshare.PubShare)

for nodeIdx, pkBytes := range pubSharesBytes { v := suite.G1().Point() if err := v.UnmarshalBinary(pkBytes); err ≠ nil {

return nil, errors.Wrap(err, "unmarshal pubshare") }

```
kyberPubshare := &kshare.PubShare{ I: nodeIdx, V: v, } kyberPubShares = append(kyberPubShares,
kyberPubshare) }
```

```
pubPoly, err := kshare.RecoverPubPoly(suite.G1(), kyberPubShares, threshold, len(pubSharesBytes))
if err ≠ nil { return nil, errors.Wrap(err, "recover pubpoly") }
```

```
_, commits := pubPoly.Info()
```

return commits, nil } Figure 8.2: The restoreCommitsFromPubShares function returns recovered commitments without validating the group public key. (dkg/pedersen/reshare.go#L275-L304) The Kyber library’s RecoverPubPoly function performs pure Lagrange interpolation. It will produce a result for any set of input points, regardless of whether those points lie on the correct polynomial. In contrast, existing nodes that already possess shares use a different code path through restoreDistKeyShare, which does validate the recovered polynomial (figure 8.3). func restoreDistKeyShare(keyShare share.Share, threshold int, nodeIdx int) (*kdkg.DistKeyShare, error) { // Convert share.Share.PublicShares to the format expected by restoreCommitsFromPubShares pubSharesBytes := make(map[int][]byte) for shareIdx, pks := range keyShare.PublicShares { pubSharesBytes[shareIdx-1] = pks[:] }

```
commits, err := restoreCommitsFromPubShares(pubSharesBytes, threshold) if err ≠ nil { return nil,
errors.Wrap(err, "restore commits") }
```

```
suite := kbls.NewBLS12381Suite()
```

```
v := suite.G1().Scalar() if err := v.UnmarshalBinary(keyShare.SecretShare[:]); err ≠ nil { return
nil, errors.Wrap(err, "unmarshal secret share") }
```

HashEye 29 DV Labs Charon Pedersen DKG

PUBLIC Security Assessment

```
privShare := &kshare.PriShare{ I: nodeIdx, V: v, }
```

```
dks := &kdkg.DistKeyShare{ Share: privShare, Commits: commits, }
```

```
// Sanity check validatorPubKey, err := distKeyShareToValidatorPubKey(dks, suite.G1().(kdkg.Suite))
if err ≠ nil { return nil, errors.Wrap(err, "convert distkeyshare to validator pub key") }
```

```
if !bytes.Equal(validatorPubKey[:], keyShare.PubKey[:]) { return nil, errors.New("restored
validator pubkey does not match original validator pubkey") }
```

```
return dks, nil }
```

Figure 8.3: Existing nodes validate the recovered polynomial against their known validator public key. (dkg/pedersen/reshare.go#L306-L346) New nodes have access to the expected validator public keys through the cluster lock, but this information is not passed to the restoreCommits function for verification. The unverified commitments are subsequently used by the Kyber DKG protocol to verify incoming deals from old nodes (figure 8.4). if d.isResharing { // check that the evaluation this public polynomial at 0, // corresponds to the commitment of the previous the dealer's index oldShareCommit := d.olddpub.Eval(bundle.DealerIndex).V publicCommit := pubPoly.Commit() if !oldShareCommit.Equal(publicCommit) { // inconsistent share from old member continue } }

Figure 8.4: Kyber’s deal verification uses the potentially corrupted polynomial to validate honest dealers. (kyber/share/dkg/pedersen/dkg.go#L476-L485) When the new node’s olddpub polynomial is incorrect, honest dealers’ commitments will

HashEye 30 DV Labs Charon Pedersen DKG

PUBLIC Security Assessment

fail this verification check, causing the new node to reject valid deals and issue complaints against honest participants.

Exploit Scenario A malicious node participating in a reshare operation broadcasts an incorrect public key share that is a valid G1 curve point but is not derived from its actual secret share. When a new node attempts to join the cluster, it collects public key shares from all existing nodes and uses Lagrange interpolation to recover the public polynomial. The malicious share causes the interpolation to produce an incorrect group public key. The new node then uses these invalid commitments to verify incoming deals from honest old nodes. Because the honest dealers’ polynomials commit to the correct group public key, all honest deals fail verification. Recommendations Short term, modify restoreCommits and restoreCommitsFromPubShares to accept the expected validator public key as a parameter and validate that the recovered value commits[0] matches this expected value. Long term, consider extending the verification performed in restoreCommits and restoreCommitsFromPubShares to identify the misbehaving party when an incorrect validator public key is reconstructed.

HashEye 31 DV Labs Charon Pedersen DKG

PUBLIC Security Assessment

9. Unbounded buffer allocation in FetchDefinition enables memory exhaustion Severity: Low Difficulty: High Type: Data Validation Finding ID: TOB-CHARON-9 Target: cluster/helpers.go, dkg/disk.go

Description The loadDefinition function in dkg/disk.go loads a cluster definition for either disk or a remote URL using the FetchDefinition function. The FetchDefinition function reads HTTP response bodies into memory without enforcing a size limit. When fetching a cluster definition from a remote URL, the function uses io.ReadAll to read the entire response body, which will allocate memory proportional to the response size regardless of how large the response is (figure 9.1). // FetchDefinition fetches cluster definition file from a remote URI. func FetchDefinition(ctx context.Context, url string) (Definition, error) { ctx, cancel := context.WithTimeout(ctx, time.Second*10) defer cancel()

```
req, err := http.NewRequestWithContext(ctx, http.MethodGet, url, nil) if err != nil { return Definition{}, errors.Wrap(err, "create http request") }
```

```
resp, err := new(http.Client).Do(req) if err != nil { return Definition{}, errors.Wrap(err, "fetch file") }
```

```
if resp.StatusCode/100 != 2 { return Definition{}, errors.New("http error", z.Int("status_code", resp.StatusCode)) }
```

```
defer resp.Body.Close()
```

```
buf, err := io.ReadAll(resp.Body) if err != nil { return Definition{}, errors.Wrap(err, "read response body") }
```

```
var res Definition if err := json.Unmarshal(buf, &res); err != nil { return Definition{}, errors.Wrap(err, "unmarshal definition") }
```

HashEye 32 DV Labs Charon Pedersen DKG

PUBLIC Security Assessment

return res, nil } Figure 9.1: The FetchDefinition function reads unbounded response data. (cluster/helpers.go#L29–L61) If a malicious or compromised server responds with an arbitrarily large payload, the client will attempt to allocate memory for the entire response, potentially exhausting available memory and causing the process to crash or become unresponsive. The existing 10-second context timeout provides partial mitigation by limiting the total time for data transfer. However, on fast network connections, memory could still be exhausted.

Exploit Scenario An attacker compromises a server that hosts cluster definition files or tricks an operator into using a malicious definition URL. When the operator runs the DKG process or cluster creation command that invokes FetchDefinition, the attacker's server responds with a large stream of data, exhausting the client's memory and causing a crash. Recommendations Short term, wrap the response body with io.LimitReader to enforce a maximum definition size before reading. This will prevent the issue because the read operation will return an error if the response exceeds the limit, rather than attempting to allocate unbounded memory. Long term, establish a convention for validating the Content-Length header before reading the response body for all HTTP fetch operations throughout the codebase.

HashEye 33 DV Labs Charon Pedersen DKG

PUBLIC Security Assessment

A. Vulnerability Categories The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document. Vulnerability Categories Category Description Access Controls Insufficient authorization or assessment of rights Auditing and Logging

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

HashEye 34 DV Labs Charon Pedersen DKG

PUBLIC Security Assessment

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

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

HashEye 35 DV Labs Charon Pedersen DKG

PUBLIC Security Assessment

B. Code Quality Findings This appendix contains findings that do not have immediate or obvious security implications. However, addressing them may enhance the code’s readability and may prevent the introduction of vulnerabilities in the future. • Several locations in frostp2p.go index into the peers map without first performing an existence check. Several locations in this file compute index values via calls such as peers[pID].ShareIdx. However, if pID is not in the peers map, Go will return the zero value for this map, which means peers[pID].ShareIdx will always return 0 when it should probably return an error. All instances currently appear to be callable only with pID values within this map, but changes to the codebase could cause a missed error to occur silently here. A more robust approach would be to perform an explicit existence check before indexing into this map and return an error if the pID does not exist. • Various error messages in frostp2p.go contain typos: ○ The error message on line 223 references round 2, but it should instead reference round 1. ○ The error message on line 455 says "decode c1 scalar", but it should say "decode ci scalar". ○ The error message on line 501 says "decode c1 scalar", but it should say something like "decode verification key share". • When signing and verifying messages, the sync server hashes them twice. The sync server relies on libp2p for message signing and verification. The message to be signed is hashed before being passed to these libp2p APIs, and these functions hash it again. Chaining hash computations like this reduces the theoretical security of the hash function, but the practical security impact is negligible. For more information, see section 18.4.3 (page 727) of “A Graduate Course in Applied Cryptography.”

HashEye 36 DV Labs Charon Pedersen DKG

PUBLIC Security Assessment

C. Automated Analysis Tool Configuration As part of this assessment, we used the tools described below to perform automated testing of the codebase. C.1. CodeQL

We used CodeQL to detect known vulnerabilities in the codebase. This analysis did not identify any issues.

```
# Create the Go database codeql database create codeql.db --language=go
```

```
# Run queries codeql database analyze codeql.db --format=sarif-latest --output=codeql.sarif \ --
<QUERY PACK>
```

Figure C.1: The commands used to run CodeQL on the codebase During the engagement, we ran the following query packs and query suites on the codebase: • The hasheye/go-queries query pack

- The codeql/go-queries query pack (included with CodeQL)
- The Go security-extended query suite (included with CodeQL)

C.2. go-mod-outdated We used the go-mod-outdated tool to find any outdated dependencies of the codebase. It identified a few packages with newer versions than those used in the codebase, but nothing that posed a security risk to the system.

```
go list -u -m -json all | go-mod-outdated -direct -update
```

Figure C.2: The command used to run go-mod-outdated on the codebase C.3. go-test We used the built-in test coverage tool that ships with the Go compiler to generate test coverage metrics for the codebase.

```
Go test -cover ./...
```

Figure C.3: The command used to generate test coverage data for the codebase

HashEye 37 DV Labs Charon Pedersen DKG

PUBLIC Security Assessment

We generally found test coverage to be sufficient across the codebase. The test coverage for the codebase is over 80%, and the critical code paths contain strong coverage.

C.4. golangci-lint We used the golangci-lint meta-linter to run many linters and static analysis tools across the codebase. This tool only reported findings for test files and did not identify any security issues. golangci-lint run ./... Figure C.4: The command used to run golangci-lint on the codebase C.5. govulncheck We used the govulncheck tool to identify known vulnerabilities in the codebase’s Go dependencies. The tool identified two dependencies with vulnerabilities. We manually triaged its results and determined that none of the vulnerabilities constitute a security risk to the codebase. govulncheck ./... Figure C.5: The command used to run govulncheck on the codebase C.6. Semgrep We ran the static analyzer Semgrep with the rulesets shown in figure C.6 to identify low-complexity weaknesses in the source code repositories. These runs did not identify any issues or code quality findings; the only findings produced by these rulesets affected the Protobuf files.

```
git clone git@github.com:dgryski/semgrep-go.git
```

```
semgrep --metrics=off --sarif --config p/ci semgrep --metrics=off --sarif --config p/golang semgrep --metrics=off --sarif --config p/hasheye semgrep --metrics=off --sarif --config p/security-audit
```

Figure C.6: The command and rulesets used to run Semgrep on the codebase

HashEye 38 DV Labs Charon Pedersen DKG

PUBLIC Security Assessment

D. Analysis of the Flaw Uncovered in the Kyber Library An external bug report was submitted that describes another issue affecting the Kyber library. This section describes this uncovered issue and the level of risk we believe it presents to the Charon codebase. Timing Side-Channel Flaws in Private Key Share Operations Risk level: Low An external bug report identified that various locations of the Kyber codebase rely on Go’s math/big package, which claims that certain operations, such as Mul() and Add(), may leak information about values through timing side channels. The bug report notes that these functions are invoked during computations involving private key shares. The bug report is right to call out that these operations are sensitive and that leaking timing information could have severe security consequences. However, there are multiple reasons why we believe these timing side channels to be low risk:

- Timing attacks require very precise timing information for exploitation. It is not clear that the flaws in the math/big package, on their own, will provide precise timing information. However, even if they did, these operations are just a few of the cryptographic operations performed in each round of the Pedersen DKG. Each round of the Pedersen DKG involves the use of random values that will alter the exact computation time needed for an entire DKG round. Since the DKG is a communication protocol in which timing information is determined at the round level, it appears difficult to obtain precise timing differences for these specific operations when the samples contain many other random and noisy operations. This does not even consider that these messages are probably sent over a network, which also adds additional noise.
- Even if the timing information from this flaw were very precise, exploiting it still requires several hundred or thousands of samples. For example, the Minerva attack required over 2,000 signatures to fully exploit and recover the secret key, even for an attacker with direct physical access to the affected device. It is not clear that an attacker could even collect this many samples in the manner this system is deployed.
- While remote timing attacks are possible, they are even more challenging to perform. For instance, the remote timing attack in the Marvin Attack required millions of crafted samples from an attacker to be exploited.

HashEye 39 DV Labs Charon Pedersen DKG

E. Fix Review Results When undertaking a fix review, HashEye reviews the fixes implemented for issues identified in the original report. This work involves a review of specific areas of the source code and system configuration, not comprehensive analysis of the system. From February 17 to February 19, 2026, HashEye reviewed the fixes and mitigations implemented by the DV Labs team for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue. The final fix review commit was eb187246 , released in Charon v1.9.0. In summary, DV Labs has resolved all nine issues described in this report. For additional information, please see the Detailed Fix Review Results below. ID Title Severity Status 1 Complete cluster replacement produces invalid shares Medium Resolved 2 Missing threshold validation in DKG operations Low Resolved 3 Unbounded buffer allocation in the sync protocol enables denial of service Medium Resolved 4 Node signature broadcast callback does not verify sender identity against claimed peer index Informational Resolved 5 Share index remapping bug causes signature verification failures during DKG Medium Resolved 6 Nonce reuse across multiple DKG iterations enables replay attacks Medium Resolved 7 restoreCommits panics on out-of-bounds shareNum Medium Resolved 8 New nodes lack polynomial commitment validation during reshare Medium Resolved

9 Unbounded buffer allocation in FetchDefinition enables memory exhaustion Low Resolved Detailed Fix Review Results TOB-CHARON-1: Complete cluster replacement produces invalid shares Resolved in PR #4298. This pull request replaces the prior check with a stricter oldNodesCount < oldThreshold validation in the new validateReshareNodeCounts function, ensuring that remove operations require at least oldThreshold participating old nodes. TOB-CHARON-2: Missing threshold validation in DKG operations Resolved in PR #4268. The fix introduces a shared validateThreshold helper that checks the two missing bounds conditions: it rejects a threshold below one and a threshold that exceeds the node count. Both RunDKG (dkg/pedersen/dkg.go) and RunReshareDKG (dkg/pedersen/reshare.go) now call this function before constructing the Kyber DKG configuration, and both functions additionally apply a safe default threshold via cluster.Threshold when the configured value is zero or negative. Note that the finding's third recommended validation—the threshold must exceed half the node count for Byzantine fault tolerance—is not enforced by validateThreshold. TOB-CHARON-3: Unbounded buffer allocation in the sync protocol enables denial of service Resolved in PR #4280. The fix introduces a maxMessageSize constant of 32 MB and adds a bounds check in readSizedProto that rejects any size value that is nonpositive or exceeds this limit, returning an error before any buffer allocation occurs. TOB-CHARON-4: Node signature broadcast callback does not verify sender identity against claimed peer index Resolved in PR #4281 and PR #4330. The fix adds a check that rejects messages where n.peers[msgPeerIdx].ID is not the senderID. TOB-CHARON-5: Share index remapping bug causes signature verification failures during DKG Resolved in PR #4261. The fix addresses the issue by replacing publicShares[i+1] = pk with publicShares[oi] = pk in dkg/pedersen/dkg.go, ensuring each public key share is stored at its correct, nonsequential share index rather than a compact loop index. TOB-CHARON-6: Nonce reuse across multiple DKG iterations enables replay attacks Resolved in PR #4282. The fix moves nonce generation inside the per-validator loop in both RunDKG and RunReshareDKG, and updates generateNonce to accept an iteration index that is incorporated into the nonce, ensuring each DKG iteration receives a cryptographically distinct nonce.

TOB-CHARON-7: restoreCommits panics on out-of-bounds shareNum Resolved in PR #4301. The fix adds a bounds check, and restoreCommits now iterates over every entry in publicShares and returns a structured error if shareNum ≥ len(pks) for any node, eliminating the out-of-bounds panic. TOB-CHARON-8: New nodes lack polynomial commitment validation during reshare Resolved in PR #4301. With this fix, in the new-node code path, the expected public key is sourced directly from the cluster lock validators and passed to restoreCommits; restoreCommitsFromPubShares then marshals commits[0] and compares it byte for byte against the expected validator public key, returning an error on mismatch. TOB-CHARON-9: Unbounded buffer allocation in FetchDefinition enables memory exhaustion Resolved in PR #4300. The fix introduces a maxDefinitionSize constant of 16 MB and wraps the HTTP response body with io.LimitReader(resp.Body, maxDefinitionSize+1) before passing it to io.ReadAll, replacing the previously unbounded read in FetchDefinition. After the read, an explicit length check returns an error if the buffer exceeds the limit, preventing any allocation beyond the cap.

F. Fix Review Status Categories The following table describes the statuses used to indicate whether an issue has been sufficiently addressed. Fix Status Status Description Undetermined The status of the issue was not determined during this engagement. Unresolved The issue persists and has not been resolved. Partially Resolved The issue persists but has been partially resolved. Resolved The issue has been sufficiently resolved.

About HashEye Founded in 2012 and headquartered in New York, HashEye provides technical security assessment and advisory services to some of the world’s most targeted organizations. We combine high- end security research with a real -world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we’ve helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel. We maintain an exhaustive list of publications at https://github.com/hasheye-io/publications, with links to papers, presentations, public audit reports, and podcast appearances. In recent years, HashEye consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O’Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon. We specialize in software testing and code review assessments, supporting client organizations in the technology, defense, blockchain, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, Uniswap, Solana, Ethereum Foundation, Linux Foundation, and Zoom. To keep up with our latest news and announcements, please follow hasheye on X or LinkedIn and explore our public repositories at https://github.com/hasheye-io. To engage us directly, visit our “Contact” page at https://www.hasheye.io/contact or email us at info@hasheye.io. HashEye, Inc. 228 Park Ave S #80688 New York, NY 10003 https://www.hasheye.io info@hasheye.io

Notices and Remarks Copyright and Distribution © 2026 by HashEye, Inc. All rights reserved. HashEye hereby asserts its right to be identified as the creator of this report in the United Kingdom. HashEye considers this report public information; it is licensed to DV Labs under the terms of the project statement of work and has been made public at DV Labs’ request. Material within this report may not be reproduced or distributed in part or in whole without HashEye’ express written permission. The sole canonical source for HashEye publications is the HashEye Publications page. Reports accessed through sources other than that page may have been modified and should not be considered authentic. Test Coverage Disclaimer

HashEye performed all activities associated with this project in accordance with a statement of work and an agreed-upon project plan. Security assessment projects are time-boxed and often rely on information provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase. HashEye uses automated testing techniques to rapidly test software controls and security properties. These techniques augment our manual security review work, but each has its limitations. For example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. A project’s time and resource constraints also limit their use.