Security Audit - Quantus Network's Dilithium and HDWallet

conducted by Neodyme AG

Lead Auditor:	Ruben Gonzalez
Second Auditor:	Jasper Slusallek

December 15, 2025

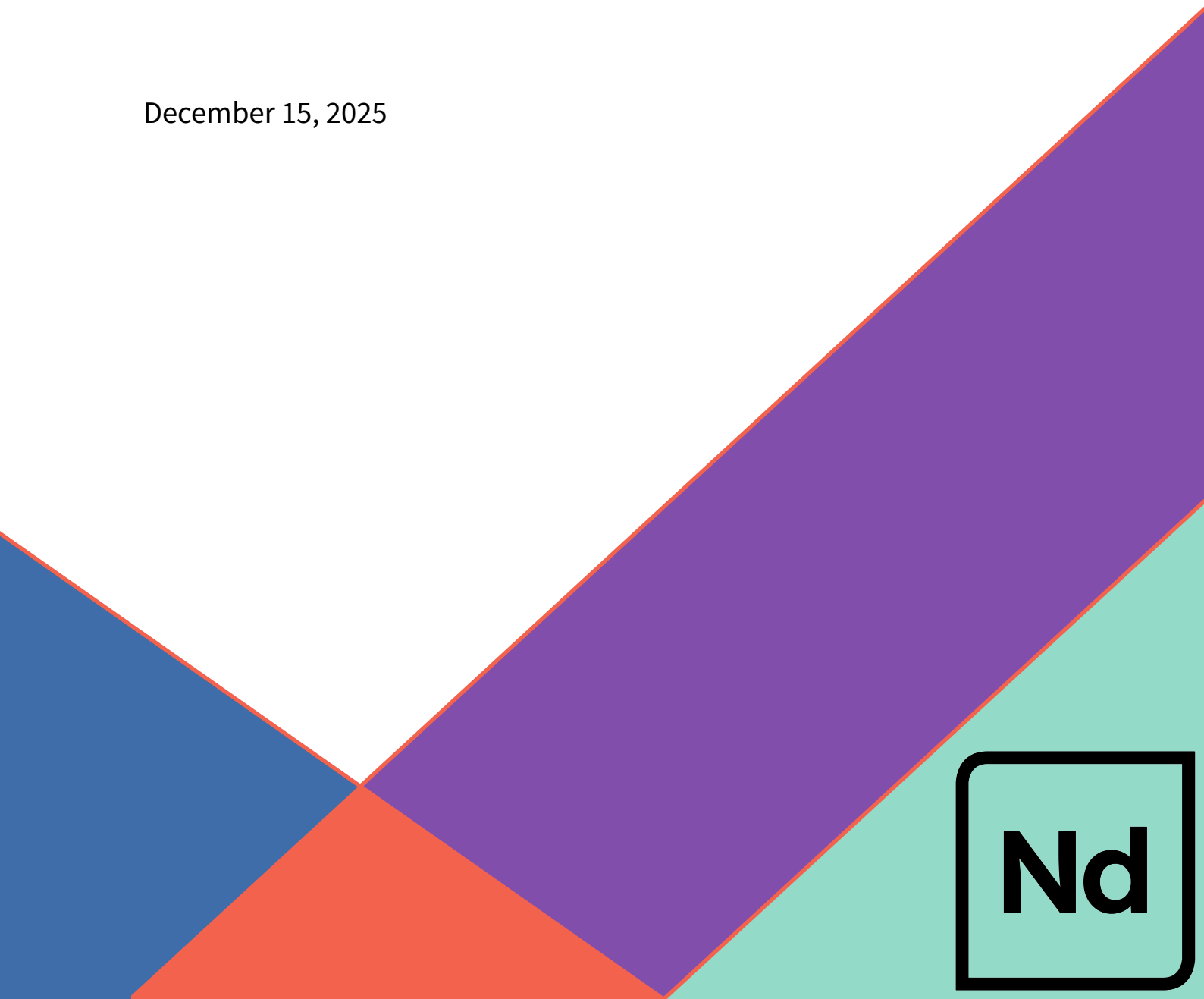

Table of Contents

Appendices

1 | Executive Summary

Neodyme audited **Quantus Network's** Dilithium and HDWallet implementation from November 2025 until December 2025.

The implementations form the cryptographic core of the entire Quantus Network. Due to the foundational relevance to overall security, both components were audited in-depth.

The audit was conducted in an early stage of Quantus Network's implementation efforts. Some components were indeed not production-ready at the start of the audit, but made great strides towards maturity over its course.

During the audit, **13 security relevant** and **2 informational** findings were identified. Figure 1 shows all identified findings grouped by severity.

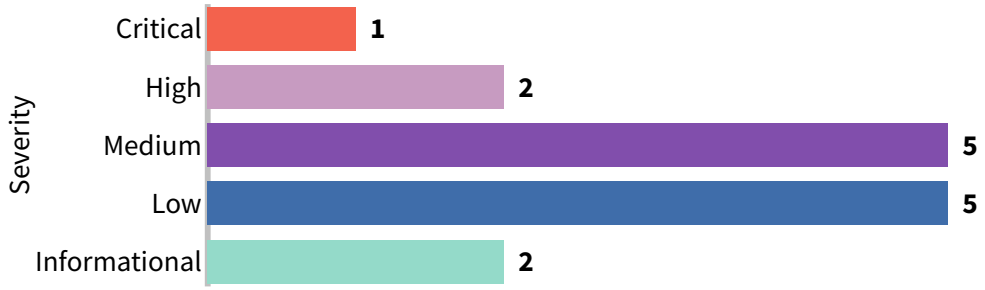

Figure 1: Overview of Findings

The auditors reported all findings to the Quantus Network developers, who addressed them promptly. Security fixes for the findings were verified for completeness by Neodyme. In addition to these findings, Neodyme delivered the Quantus Network team further nit-picks and implementation guidance that are not part of this report.

2 | Introduction

Quantus Network engaged Neodyme to do a detailed security analysis of their Dilithium and HDWallet implementations from November 2025 until December 2025. Both components are described in Section 4. Two senior security researchers from Neodyme conducted the audit of both components between the 1st of November 2025 and the 15th of December 2025. Combined, both auditors have a long track record of finding critical vulnerabilities in various blockchain ecosystems and extensive knowledge of implementing cryptographic primitives. The lead auditor pursued a PhD in post-quantum cryptography and has published multiple peer-reviewed papers, online publications, and code on the topic. He has further consulted and trained developers and security professionals on the subject at the leading cybersecurity conferences. The second auditor has found dozens of critical vulnerabilities in various blockchain ecosystems, thereby protecting at least $3B of funds at risk.

The audit focused on implementation security of the components' software written in Rust.

In the following sections, we detail the audit's scope, introduce context by summarizing implementation-relevant background and discuss findings.

Summary of Findings

All identified and reported issues were **quickly remediated** by the Quantus Network team.

In total, the audit revealed:

1 critical • **2** high-severity • **5** medium-severity • **5** low-severity • **2** informational

issues.

3 | Scope

The audit's scope comprised of two components written in the Rust programming language:

- A Dilithium post-quantum signature algorithm
- A hierarchical, deterministic wallet based on the Dilithium implementation

The audit focused on implementation security of the software. Explicitly not included in the audit's threat model are hardware-based attacks. These include, but are not limited to, fault injection attacks, speculative execution attacks, power analysis or similar side channels. Their mitigation requires knowledge of the specific targeted hardware architecture, which was not available at the time of writing. The audit's scope therefore only includes the software security on source code level.

Neodyme considers the source code, located at https://github.com/Quantus-Network/qp-rusty-crystals/, in scope for this audit. Third-party dependencies are not in scope. During the audit, changes and fixes were made by Quantus Network, which the auditors also reviewed in-depth.

Relevant source code revisions are:

- `784b3b65081e1624dc74d8ccb01986e306fe6b7c` · Start of the audit
- `8bbe920dc5bfe88fa40028eb1ea9bae4e39a600c` · Last reviewed revision

4 | Project Overview

This section briefly outlines aspects of Dilithium and HDWallet's background, functionality, design, and architecture necessary to comprehend the findings presented in the next section.

Dilithium

Dilithium is a structured-lattice-based signature algorithm that was recently standardized by the National Institute of Standards and Technology (NIST) under the name Module-Lattice-Based Digital Signature Algorithm (ML-DSA). From here on out it is therefore referred to as ML-DSA.

ML-DSA is a post-quantum algorithm, aiming for security against attackers in possession of both classical and quantum computers. Traditional signature algorithms such as Ed25519 or RSA are known to be entirely broken upon the arrival of a large quantum computer. As development towards large quantum computers has accelerated significantly in past years, quantum security has become a major concern.

Quantus Network has implemented ML-DSA in its highest, standardized parameters, referred to as ML-DSA-87. ML-DSA-87 has a claimed security strength of NIST Level 5, meaning an attack would require at least as many computational resources as an exhaustive key search on a 256 bit key block cipher. Informally, this is referred to as 256 bits of security and forms the minimum security bound for the audit.

In its structured lattice, ML-DSA-87 employs the polynomial ring $\mathbb{Z}_p[X]/(X^{256}+1)$ with $p = 8380417$. That is, it operates on vectors of polynomials of degree at most 255 with coefficients in $\mathbb{Z}_p$. A core component of ML-DSA implementations is the Number Theoretic Transform (NTT), vastly accelerating operations within the lattice. Further specifics can be found in the original publication [pq-crystals].

Quantus Network's implementation is written in pure Rust and does therefore not include assembly code commonly found in other Dilithium implementations. Implementation in assembly allows for the use of vector instructions on modern desktop CPUs, resulting in performance gains. However, as stated goals of the audited implementation include portability and memory safety, such optimizations were not included. The audited implementation therefore has a conservative coding style that — in regards to NTT and ring operations — closely resembles the official Dilithium reference implementation and the ML-DSA standard.

Constant Time Requirements

An opaque aspect of Dilithium implementation is the necessary relaxation of the usually mandated constant time property of code. Ideally, traditional signature algorithms have a permanently constant runtime. That means, the algorithms runtime can by no means be influenced by sensitive values, such as key material or nonce values. This serves to protect the implementation from leaking sensitive values through runtime measurements (i.e. timing-based side channels).

Dilithium internally uses rejection sampling to derive values within publicly known bounds from key material. This makes the code inherently variable in runtime. Seemingly, the runtime is even directly influenced by the secret key material.

However, as the algorithms runtime only reflects how long it took to sample suitable values that lie within publicly known bounds, this does not impact security. Hence, strict constant time for the overall algorithm is not required. In parts of the implementation that directly handle key material however, it is required.

HDWallet

A hierarchical deterministic (HD) wallet is a cryptocurrency wallet that uses a single master seed to deterministically generate a structured tree of key pairs, allowing subaddresses to be derived reproducibly. This allows an organization to manage and securely backup many addresses with just one seed. Typically, subtrees of the overall structured trees are delegated to members of the organization, which in turn can then manage and organize addresses derived from their subtree.

Quantus Network's implementation is largely inspired by the Bitcoin Improvement Proposals (BIPs) *Hierarchical Deterministic Wallets* [BIP-0032], *Multi-Account Hierarchy for Deterministic Wallets* [BIP-0044] and *Mnemonic Code for Generating Deterministic Keys* [BIP-0039].

As these BIPs make heavy use of the distributive property of groups of elliptic curve points, they can not be directly applied to a Wallet managing Dilithium-based addresses. For example, non-hardened keys can't be used for computing parent public keys from child keys. This forces Quantus Network's HDWallet to function different from the aforementioned BIPs. Specifically, only hardened keys are supported making upwards tree traversal impractical. Other BIP aspects however, such as path level definitions, are adhered to.

5 | Findings

This section details all findings encountered during the audit. They are classified into one of five severity levels, detailed in Appendix C. In addition to these findings, Neodyme delivered the Quantus Network team further nit-picks and implementation guidance that are not part of this report.

All findings are listed in Table 1 and further described in the following subsections.

Identifier	Name	Severity	Status
ND-QNPQC-CR-01	Domain Separation Bypass	CRITICAL	Resolved
ND-QNPQC-HI-01	Insecure Hash Function	HIGH	Resolved
ND-QNPQC-HI-02	Permitting Low Entropy	HIGH	Resolved
ND-QNPQC-MD-01	Missing Drop on Sensitive Values	MEDIUM	Resolved
ND-QNPQC-MD-02	Missing Entropy Check	MEDIUM	Resolved
ND-QNPQC-MD-03	Mass Use of Heap Allocation	MEDIUM	Resolved
ND-QNPQC-MD-04	Dependencies in Cryptographic Core	MEDIUM	Resolved
ND-QNPQC-MD-05	DOS Vector in Library	MEDIUM	Resolved
ND-QNPQC-LO-01	Reliance on BlackBox for Constant Time	LOW	Resolved
ND-QNPQC-LO-02	Misleading Documentation	LOW	Resolved
ND-QNPQC-LO-03	Incorrect Documentation	LOW	Resolved
ND-QNPQC-LO-04	CT dependence on unprotected variable	LOW	Acknowledged
ND-QNPQC-LO-05	Non-Constant Runtime	LOW	Acknowledged
ND-QNPQC-IN-01	ChaCha20 Arithmetic on Sensitive Seeds	INFORMATIONAL	Resolved
ND-QNPQC-IN-02	Elliptic Curve Arithemtic on ML-DSA Seeds	INFORMATIONAL	Resolved

Table 1: Findings

[ND-QNPQC-CR-01] Domain Separation Bypass

Severity	Impact	Affected Component	Status
CRITICAL	Domain Separation Break	ML-DSA	Resolved

The implementation always applies the zero context, even if context is given. This occurs as the signing context always prefixes two null bytes, stating the message is signed under no context, even if the message was already formatted under a context:

```
/// Compute message hash and signing randomness
 fn prepare_signing_context(
  unpacked_sk: &UnpackedSecretKey,
  message: &[u8],
  hedge_randomness: Option<[u8; params::SEEDBYTES]>,
 ) -> SigningContext {
  // Compute message hash μ = H(tr || pre || msg) where pre = (0, 0) for pure signatures
  let mut keccak_state = fips202::KeccakState::default();
  fips202::shake256_absorb(&mut keccak_state, &unpacked_sk.public_key_hash_tr, params::TR_BYTES);
  let context_prefix = [0u8, 0u8]; // (domain_sep=0, context_len=0) for pure signatures
  fips202::shake256_absorb(&mut keccak_state, &context_prefix, 2);
```

Due to this behavior, the supplied context is then interpreted as part of the message. This breaks domain separation introduced through contexts, which can be leveraged for signature forging.

Proof of Concept

Poof of concept code to illustrate this problem:

```
let entropy = get_random_bytes();
let ml87_keypair = ml_dsa_87::Keypair::generate(&entropy);

// Message to be signed under context X
let test_msg = b"stack usage test message";
// Signing context
let test_ctx = b"X";
// Malicious message without context
let test_msg_fake_ctx = b"\x00\x01Xstack usage test message";

let ml87_sig = ml87_keypair.sign(test_msg, Some(test_ctx), None);

// This should work
assert!(ml87_keypair.verify(test_msg, &ml87_sig, Some(test_ctx)));
```

```
15
16 // This should NOT work
17 assert!(ml87_keypair.verify(test_msg_fake_ctx, &ml87_sig, None));
```

Resolution

The Quantus Network team resolved the issue by removing the falsely-applied zero context.

This finding was fixed by commit bef994d292f976c2e711a4f39cf56527ceecbc40.

[ND-QNPQC-HI-01] Insecure Hash Function

Severity	Impact	Affected Component	Status
HIGH	Reduced Security	ML-DSA	Resolved

The implementation offers a HashML-DSA-compatible interface for pre-hashing inputs. This interface is described in [ML-DSA].

In the code base, the SHA256 hash function for this purpose:

```
pub fn prehash_sign(
    &self,
    msg: &[u8],
    ctx: Option<&[u8]>,
    hedge: Option<[u8; params::SEEDBYTES]>,
    ph: crate::PH,
  ) -> Option<Signature> {
    let mut oid = [0u8; 11];
    let mut phm: Vec<u8> = Vec::new();
    match ph {
      crate::PH::SHA256 => {
        oid.copy_from_slice(&[
          0x06, 0x09, 0x60, 0x86, 0x48, 0x01, 0x65, 0x03, 0x04, 0x02, 0x01,
        ]);
        phm.extend_from_slice(Sha256::digest(msg).as_slice());
      },
```

This is problematic as the implementation specifically targets NIST security level 5 (ML-DSA-87). The HashML-DSA security, however, is strictly dependent (a.o.) on the collision resistance of the underlying hash function used (see [ML-DSA] section 5.4). SHA256 has a collision resistance of approximately 128 bit and therefore reduces the algorithms security to that of NIST level 2. It is noteworthy that SHA256 is not insecure or broken, but offers less security than the targeted security level assuming a quantum attacker.

Resolution

The Quantus Network team removed the HashML-DSA compatibility layer from their code base.

This finding was fixed by commit 666c4967bfc5ac04c1f9205f11c60916895e5e79.

[ND-QNPQC-HI-02] Permitting Low Entropy

Severity	Impact	Affected Component	Status
HIGH	Weak Keys	HDWallet	Resolved

The `generate_mnemonic` function within HDWallet, despite requiring 32 bytes of entropy as input, allows shorter output mnemonics:

```
pub fn generate_mnemonic(word_count: usize, seed: [u8; 32]) -> Result<String, HDLatticeError> {
  // Calculate entropy bytes needed (12 words = 16 bytes, 24 words = 32 bytes)
  let bits = match word_count {
    12 => 128,
    15 => 160,
    18 => 192,
    21 => 224,
    24 => 256,
    _ => return Err(HDLatticeError::BadEntropyBitCount(word_count)),
  };
```

This makes it possible to generate mnemonics with 128 bits of entropy, whereas 256 bits are required for securely generating ML-DSA-87 keys.

Resolution

The Quantus Network team resolved the issue by removing support for too short mnemonics.

This finding was fixed by commit d0ee4aa627dbccaeaf100b6254ac33.

[ND-QNPQC-MD-01] Missing Drop on Sensitive Values

Severity	Impact	Affected Component	Status
MEDIUM	Possible Key Leakage	ML-DSA	Resolved

Within ML-DSA as well as HDWallet, sensitive values are placed on heap and stack throughout the program runtime, without being wiped after usage. The standard explicitly mandates clearing memory and intermediate values after use.

Resolution

The Quantus Network team mitigated the issue by employing the locked [zeroize] Rust crate for overwriting sensitive values as soon as they run out of scope.

This finding was mitigated by commits 5de7d0f5b27748f3fce704b3e85a0759019c0b3d and 8bbe920dc5bfe88fa40028eb1ea9bae4e39a600c.

[ND-QNPQC-MD-02] Missing Entropy Check

Severity	Impact	Affected Component	Status
MEDIUM	Possible Weak Keys	ML-DSA	Resolved

The ML-DSA library does not to gather entropy for the user. It relies solely on the user to supply strong key material:

```
pub fn generate(entropy: &[u8]) -> Keypair {
    let mut pk = [0u8; PUBLICKEYBYTES];
    let mut sk = [0u8; SECRETKEYBYTES];
    crate::sign::keypair(&mut pk, &mut sk, entropy);
    Keypair {
      secret: SecretKey::from_bytes(&sk).expect("Should never fail"),
      public: PublicKey::from_bytes(&pk).expect("Should never fail"),
    }
  }
```

This is an unconventional, although acceptable, choice for a cryptography primitive. However, there are no safety measures in place.

Recommended Resolution

Either:

- gather entropy within library, e.g. by employing a fork of the [getrandom] crate

Or alternatively:

- as a bare minimum, check that the supplied slice has a length of at least 32. This length constitutes a necessary but not a sufficient criterion. The documentation should reflect his fact and urge users to supply 32 bytes of high-quality entropy.

Resolution

The Quantus Network team mitigated the issue by performing a length check on the supplied entropy parameter.

This finding was fixed by commit 925c9cffd53bfe5cb470a77613cab2626ab83519.

[ND-QNPQC-MD-03] Mass Use of Heap Allocation

Severity	Impact	Affected Component	Status
MEDIUM	Possible Key Leakage and DOS vectors	ML-DSA	Resolved

The ML-DSA code relies heavily on Rust's `Box` type for storing sensitive values on the heap. Customarily, cryptography primitives should not be using the heap. While unavoidable in the implementation of higher protocols (such as TLS), the primitives themselves do not need to allocate memory dynamically. Using the heap introduces additional failure modes and timing variance.

Examples:

- Some developers overwrite a worst-case amount of bytes on the stack after using a crypto primitive - wiping all sensitive values from RAM. For heap-allocated values this is not possible as they don't form a contingent block of memory.
- Using this library within a C code base leads to the risk of exposing sensitive values after memory is freed, but not cleared, and then re-allocated.

Comments within the implementation imply that the heap was used to save on stack space for resource-constrained devices. In these devices, using `Box` does not have an advantage over using the stack, as the heap consumes even more memory (due to heap meta data) and requires a heap allocator to be available for the targeted platform.

Recommended Resolution

Drop use of heap allocation within ML-DSA altogether.

Resolution

The Quantus Network team resolved the issue by removing the usage of `Box`'ed values.

This finding was fixed by commit c7b118f04512c95bd7e8ae25b2ebe7eb0e58c918.

[ND-QNPQC-MD-04] Dependencies in Cryptographic Core

Severity	Impact	Affected Component	Status
MEDIUM	Loss of Funds	ML-DSA	Resolved

The ML-DSA production build relies on dependencies from external authors.

Namely:

```
sha2 = { version = "0.10.8", default-features = false }
subtle = { version = "2.4.1", default-features = false }
```

Ideally, the implementation of a primitive that is supposed to safeguard large funds comes without external dependencies. This is to avoid potential supply chain attacks. The same is, though less severe, true for HDWallet dependencies.

Of course this risk can also be accepted, but one should consider removing those dependencies or forking the underlying source into a repository under control of the libraries authors. Relying on the authors of these dependencies for the cryptographic core effectively gives them full access to funds.

Recommended Resolution

Remove subtle crate, together with constant time claims, altogether. Remove SHA2 altogether (see also Section 5.2).

Alternatively, accept the risk, but mandate the locked versions of the dependencies for production builds using cargo's `--locked` flag.

Resolution

The Quantus Network team resolved the issue by removing the two dependencies. The newly added `zeroize` (see Section 5.4) dependency was locked (via `Cargo.lock`) and CI builds use the `--locked` flag. HDWallet dependencies underwent the same treatment.

This finding was fixed by commit 073ca390f524d70d68f171f99d56bca487d4f07d.

[ND-QNPQC-MD-05] DOS Vector in Library

Severity	Impact	Affected Component	Status
MEDIUM	Denial of Service	ML-DSA	Resolved

The Dilithium implementation panics if a context larger than 255 bytes is supplied:

```
Some(x) => {
 if x.len() > 255 {
   panic!("ctx length must not be larger than 255");
 }
 let x_len = x.len();
```

This length check is necessary. However, the library should return an error value instead of raising a panic. This could lead to Denial of Service attack vectors when the signature code is used in critical code paths.

Resolution

The Quantus Network team resolved the issue by replacing the panic with returning an error value.

This finding was fixed by commit 93ed6c3f9278b4466c90ae58b913b66c483117f6.

[ND-QNPQC-LO-01] Reliance on BlackBox for Constant Time

Severity	Impact	Affected Component	Status
LOW	Timing Leakage	ML-DSA	Resolved

The implementation tries to hide timing introduced through store operations with dummy writes to a dummy variable. This variable is "protected" from removal due to compiler optimization (i.e. dead store elimination) with the black_box Rust intrinsic:

```
// Prevent compiler from optimizing away dummy_value
core::hint::black_box(dummy_value);
```

However, this type might be used for benchmarking but does not guarantee protection from the compiler. The Rust documentation states in this regard: "As such, it must not be relied upon to control critical program behavior. This also means that this function does not offer any guarantees for cryptographic or security purposes.".

This is not a critical vulnerability simply because ML-DSA does not require a constant time implementation in that regard (see Section 4.1.1). However, the Quantus Network team set out to introduce constant time execution and hence breaks its own guarantees.

Resolution

The Quantus Network team resolved the issue by rewriting the code and removing `black_box`.

This finding was fixed by commit cc15d7804d2ad870b1bdccd036fb324d2bbad1bf.

[ND-QNPQC-LO-02] Misleading Documentation

Severity	Impact	Affected Component	Status
LOW	Weak Keys	ML-DSA	Resolved

The ML-DSA implementation's README specifies how the library should be used:

```
use qp_rusty_crystals_dilithium::ml_dsa_87;

// Generate a keypair with entropy
let entropy = b"my_random_seed_exactly_32_bytes!";
let keypair = ml_dsa_87::Keypair::generate(entropy);

// Sign a message
let message = b"Hello, post-quantum world!";
let signature = keypair.sign(message, None, None);

// Verify the signature
let is_valid = keypair.verify(message, &signature, None);
assert!(is_valid);
```

Experience shows that these kind of examples lead to library misuse. The examples imply that an arbitrary string can be fed into the library to receive a secure key, as long as the input string is at least 32 bytes of length. However, this is of course not true, as the implementation does add entropy at all. This finding is related to the finding presented in Section 5.5.

Recommended Resolution

The example code should show a more secure use. For example with the [getrandom] crate.

Resolution

The Quantus Network team resolved the issue by updating the example code.

This finding was fixed by commit 5f0c36c8dd98de13ba35dc014ee73dcaacf35c74.

[ND-QNPQC-LO-03] Incorrect Documentation

Severity	Impact	Affected Component	Status
LOW	Library Misuse	ML-DSA	Resolved

Both the ML-DSA and HDWallet implementation feature line docs. Rust line docs are used to automatically generate documentation based on comment string.

On multiple occasions in the code base, these line docs contain invalid claims. This could lead to the library being used under wrong assumption, leading to vulnerabilities.

An example of this within the HDWallet claims:

```
1   /// Generates a new `WormholePair` using secure system entropy (only
    available with `std`).
```

Even though this is not the case, as the code does not add entropy at all.

Further examples scattered around the code base include:

- `polyveck_is_norm_within_bound` docstring states that 0 is returned when the bound is adhered. The (almost) opposite is true. A bool (true) is returned in that case.
- `reduce32` states that output is in range −6283009 <= r <= 6283007. whereas that bound should be −6283008 <= r <= 6283008 (2^31-2^22-1 - 255 * Q)
- `k_reduce` states the output coefficients will be in [0, 2Q], when they are in [−6283008, 6283008]
- `uniform_gamma1` says it samples uniformly in [-(GAMMA1 - 1), GAMMA1 - 1], when really it does so in [-(GAMMA1-1), GAMMA1]

Recommended Resolution

Align code's documentation and code.

Resolution

The Quantus Network team mitigated the issue by updating line documentation.

This finding was mitigated by commit d2480d278d087384231b4d3703d7463d42331598.

[ND-QNPQC-LO-04] CT dependence on unprotected variable

Severity	Impact	Affected Component	Status
LOW	Timing Leakage	ML-DSA	Acknowledged

The ML-DSA ball sampling implementation uses a dummy buffer `dummy_buf` to mask timing in regards to the sampling:

```
let mut dummy_buf = [0u8; fips202::SHAKE256_RATE];
  let mut dummy_pos = 0;

  let mut pos: usize = 8;
  c.coeffs.fill(0);
  for i in (N - params::TAU)..N {
    let mut b: usize = 0;
    let mut found = false;

    // in vast majority of cases this outer loop will run exactly once
    while !found {
      // do 16 iterations no matter what for constant time
      for _ in 0..16 {
        if !found {
          if pos >= fips202::SHAKE256_RATE {
            fips202::shake256_squeezeblocks(&mut buf, 1, &mut state);
            pos = 0;
          }
          b = buf[pos] as usize;
          pos += 1;
          if b <= i {
            found = true;
          }
        } else {
          // Dummy operations when already found to maintain constant timing
          if dummy_pos >= fips202::SHAKE256_RATE {
            fips202::shake256_squeezeblocks(&mut dummy_buf, 1, &mut
dummy_state);
            dummy_pos = 0;
          }
          let _dummy = dummy_buf[dummy_pos] as usize;
          dummy_pos += 1;
        }
      }
    }

    c.coeffs[i] = c.coeffs[b];
    c.coeffs[b] = 1 - 2 * ((signs & 1) as i32);
```

```
38      signs >>= 1;
39    }
40  }
```

However, this variable is quickly going out of scope and is not used in any other way than stores. It could therefore be eliminated by the compiler using dead store elimination. Timing can therefore still leak. Additionally, the outer loop within that function could also execute more than once, with a low but non-negligible probability.

This is not a critical vulnerability simply because ML-DSA does not require a constant time implementation in that regard (see Section 4.1.1). However, the Quantus Network team set out to introduce constant time execution and hence breaks its own guarantees.

Resolution

The Quantus Network team acknowledge the finding and has declared the code as acceptable within their threat model.

[ND-QNPQC-LO-05] Non-Constant Runtime

Severity	Impact	Affected Component	Status
LOW	Timing Leakage	ML-DSA	Acknowledged

The rejection sampling loop in the signature function is supposed to be constant time:

```
// this outer loop should run exactly once in the vast majority of cases
loop {
 for _ in 0..MIN_SIGNING_ATTEMPTS {
   // Generate masking vector and compute commitment
```

This property is introduced by the authors choice and not mandated by the standard (see Section 4.1.1). However, in around 1% of executions (assuming random inputs), this is not true as the code loops more than once.

This is not a critical vulnerability simply because ML-DSA does not require a constant time implementation in that regard (see Section 4.1). However, the Quantus Network team set out to introduce constant time execution and hence breaks its own guarantees.

Resolution

The Quantus Network team acknowledge the finding and has declared the code as acceptable within their threat model.

[ND-QNPQC-IN-01] ChaCha20 Arithmetic on Sensitive Seeds

Severity	Impact	Affected Component	Status
INFORMATIONAL	Increased Attack Surface	ML-DSA	Resolved

The code within `generate_mnemonic` employs a random number generator based on ChaCha20:

```
// Use seed to initiate chacha stream and fill it
 // NOTE: chacha will "whiten" the entropy provided by the os
 // if an attacker does not 100% control the os entropy, chacha
 // will provide full entropy, due to avalanche effects
 let mut chacha_rng = ChaCha20Rng::from_seed(seed);
```

This usage, when done properly (see Section 5.3), is not in itself problematic. However, as the entropy of the generated mnemonic depends entirely on the input `seed`, this call to ChaCha20 does not improve security. To avoid dependence on external crates and a false sense of added security, this usage could be avoided.

Resolution

The Quantus Network team resolved the issue by removing the ChaCha20 crate usage.

This finding was fixed by commit f368013bc099d1b457cd1058c77b4810e668c505.

[ND-QNPQC-IN-02] Elliptic Curve Arithemtic on ML-DSA Seeds

Severity	Impact	Affected Component	Status
INFORMATIONAL	Domain Seperation Break	ML-DSA	Resolved

The HDWallet code allows for non-hardened keys after layer 3:

```
for (index, element) in p.iter().enumerate() {
  // Enforce hardened for the first three indices (purpose, coin_type,
account) as per
  // BIP44 standard. The reason being, we do not have derivable public keys
anyway, it
  // does not work for dilithium key pairs.
  if index < 3 && !element.is_hardened() {
    return Err(HDLatticeError::HardenedPathsOnly());
  }
```

In principle, this does not break security in a quantum-attacker scenario. This is due to the fact, that the computed (ECC) public keys are never outputted or used by the Wallet. The key derivation relies only on the collision/preimage resistance of the SHA512 hashing performed.

However, if non-hardened keys are used, the employed (intermediate) seeds are used for elliptic curve scalar multiplication. This opens the code up to unnecessary side channels and inefficiencies.

If only hardened keys would be allowed, no code path would apply elliptic curve arithmetic on these sensitive values. As Dilithium offers no distributivity and the ECC public keys are never actually used, enforcing hardened keys on all layers comes with no downsides.

Recommended Resolution

Exclusively use hardened keys within the Wallet.

Resolution

The Quantus Network team resolved the issue by removing the support for non-hardened keys.

This finding was fixed by commit 30e9ba21fc21edb8ec86758abf92a0987133de66.

A | About Neodyme

Security is difficult.

To understand and break complex things, you need a certain type of people. People who thrive in complexity, who love to play around with code, and who don’t stop exploring until they fully understand every aspect of it. That’s us.

Our team never outsources audits. Having found bugs that could have lead to the loss of more than $50B in TVLs, we believe that Neodyme hosts the most qualified auditors in this ecosystem.

All of our team members have a background in competitive hacking. During such hacking competitions, called CTFs, we competed and collaborated while finding vulnerabilities, breaking encryption, reverse engineering complicated algorithms, and much more. Through the years, many of our team members have won national and international hacking competitions, and keep ranking highly among some of the hardest CTF events worldwide. In 2020, some of our members started experimenting with validators and became active members of the crypto community. With the prospect of an interesting technical challenge and bug bounties, they quickly encouraged others from our CTF team to look for security issues. And so, Neodyme was born.

B | Methodology

We are not checklist auditors.

In fact, we pride ourselves on that. We adapt our approach to each audit, investing considerable time into understanding the code upfront and exploring its expected behavior, edge cases, invariants, and ways in which the latter could be violated.

We use our uniquely deep knowledge of internals, and our years-long experience in auditing code bases to find bugs that others miss. We often extend our audit to cover off-chain components in order to see how users could be tricked or the code affected by bugs in those components.

Over the years, we have found hundreds of high and critical severity findings, many of which are highly nontrivial and do not fall into any strict category. This is why our approach has always gone way beyond simply checking for common vulnerabilities. We believe that the only way to truly secure a code base is a deep and tailored exploration that covers all its aspects, from small low-level bugs to complex vulnerabilities.

C | Vulnerability Severity Rating

We use the following guideline to classify the severity of vulnerabilities. Note that we assess each vulnerability on an individual basis and may deviate from these guidelines in cases where it is well-founded. In such cases, we always provide an explanation.

Severity	Description
CRITICAL	Vulnerabilities that will likely cause loss of funds. An attacker can trigger them with little or no preparation, or they are expected to happen accidentally. Effects are difficult to undo after they are detected.
HIGH	Bugs that can be used to set up loss of funds in a more limited capacity, or to render the program unusable.
MEDIUM	Bugs that do not cause direct loss of funds but that may lead to other exploitable mechanisms, or that could be exploited to render the program partially unusable.
LOW	Bugs that do not have a significant immediate impact and could be fixed easily after detection.
INFORMATIONAL	Bugs or inconsistencies that have little to no security impact, but are still noteworthy.

Additionally, we often provide the client with a list of nit-picks, i.e. findings whose severity lies below Informational. In general, these findings are not part of the report.

Neodyme AG

Dirnismaning 55
Halle 13
85748 Garching
Germany

E-Mail: contact@neodyme.io

https://neodyme.io