Nym

Security Review

JP Aumasson

20210913

Contents

1

Summary

We reviewed Nym's code base and design for security defects in terms of

- Cryptography
- Code security
- Protocol security

The repositories covered are

- `https://github.com/nymtech/nym`, the main repository of the Nym project, implementing mixnode, gateway, and validator services.
- `https://github.com/nymtech/sphinx`, the SPHINX mixnet packet format, used by mixnodes.
- `https://github.com/nymtech/coconut`, the Coconut threshold blind signature protocol, used for credential issuance.

As agreed with Nym, we reviewed the code in the main branch (develop) in its latest version at the time of review, refreshing the repository regularly. Because of the code rapid evolution, working on the latest version appeared to be a better approach than picking a commit at the start of the audit and reviewing only this version.

When evaluating the correctness of implementations, we used as "ground truth" the corresponding research papers and specifications, and related papers when relevant. In the case of Coconut, Nym created multiple implementations, which helped in reviewing correctness and interoperability.

Note that, given time constraints, we could not cover the whole Nym code base and attack surface. We notably recommend further scrutiny regarding:

- Adversarial scenarios involving malicious gateways and validators
- Contracts and transactions safety (reward system abuse, failure cases)
- Risks from unreliable network connectivity (potential DoS)
- Wasm-specific security risks (esp. regarding CosmWasm)
- Peer authentication and revocation aspects
- Custom messaging protocol's security
- Services proper configuration (API exposure, admin features, etc.)
- The token contracts

Also note that, in our list of issues and observations, we did not duplicate limitations known to or documented by the Nym team (for example as TODOs in the code, or already reported in the GitHub Issue tracker), unless those for which we could propose additional useful information or recommendation.

We would like to thank Nym for their trust, and for their responsiveness to our inquiries on the team's chat platform.

1.1 System overview

This section presents our understanding of the Nym Protocol at a high level. This is for our own benefit as auditors, but also serves to confirm our understanding with Nym. We present these notes as part of our report for completeness.

The Nym network

The Nym Network is a mixnet-based system where data traffic through the system is funded by its users using an associated cryptocurrency called Nym tokens.

The objective is that users will pay for their access to the network, to give node operators an incentive to run their network. At the same time these tokens or credentials cannot be linked to user activity, otherwise the system loses the privacy guarantees of a mixnet.

The project therefore comprises several components from the point of view of a security audit:

- Bandwidth and service credentials
- The mixnet protocol
- The Nym blockchain

Anonymous credentials

The anonymous credential system is based on Coconut. The properties that are desired of the anonymous credential system are as follows:

1. *Selective disclosure*: a set of attributes attested by an issuer may be disclosed to a verifier. Holders determine this. In Coconut, zero-knowledge attestations of credential properties are possible, meaning third parties learn nothing except the truthhood of logical statements combining the attributes.
2. *Unlinkability*: it should be impossible except for the holder to link actions made with the same credential. This property should also hold for issuers (other than knowing they have issued the credential) and verifiers (other than learning in zero knowledge the truthiness of an asserted statement).

Coconut is a threshold-issuance selective-disclosure scheme providing these properties. In Coconut, authorities issue credentials in response to requests by users, who then aggregate and randomize these credentials prior to showing them to verifiers,

Concretely, the user can aggregate and randomize credentials from multiple authorities and present a single credential to verifiers, who can then only learn that a) the credential was issued by a suitable threshold of issuers and that b) the selectively disclosed attribute matches some logical statement.

In Nym, therefore, the user is the holder of the aggregate credential. The Nym blockchain is run by *validators*, who in checking the consistency of the blockchain are also *issuers* of credentials to users. So when a user deposits funds as Nym tokens, they may request from as many validators as there exist that this deposit is attested to, and possibly for what purpose it may serve (bandwidth, access to a specific service).

The *verifier* in Coconut terms can be multiple entities in the Nym network: anyone who needs to verify a credential. According to the whitepaper this can be validators, but also gateways and service providers.

The mixnet and credential interaction

According to the whitepaper, only gateways verify bandwidth credentials. If a credential is attested, the gateway will then route the traffic through the mixnet.
Mixnet nodes receive their rewards from the Nympool (the blockchain) but not directly from validators. Services on the network do.

In terms of anonymous credentials, this means that the traffic to a gateway must carry a credential the gateway can read, and the gateway learns the truthiness of a property, presumably 'this is a paid up bandwidth credential'. The traffic is then forwarded to the mixnet.

If a paid for service is being used, this credential is not visible directly to the mixnet.

1.2 Detailed scope

In this section we provide details on the security issues covered.

Code security

When reviewing Rust code, we notably looked for:

- Unsafe coding patterns
- Unhandled/mishandled errors (including panics)
- Insecure dependencies
- Superfluous use of 'unsafe' blocks
- Unsafe use of `unwrap()`
- Integer overflows

We notably used the standard Rust utilities `cargo audit`, `cargo geiger`, `cargo outdated` to automate basic verification tasks.

Crypto primitives

When reviewing core crypto primitives, we covered:

- AES-128-CTR
- BLAKE2b (as used in LIONESS)
- BLAKE3
- ChaCha (as used in LIONESS)
- Ed25519
- LIONESS (as used in SPHINX)
- HMAC
- HKDF
- SHA-256
- X25519

We notably looked for exploitable

- Misimplementation of the schemes
- Side channels (timing leaks, oracles)
- Randomness generation or usage flaws
- Unsafe parameters support
- Degenerate cases and weak keys support

SPHINX

When reviewing SPHINX, we notably looked for exploitable

- Differences with the paper
- Unsafe mix message construction and processing
- Leaks compromising sender/receiver anonymity
- Leaks making packets linkable or distinguishable
- Forward/reply distinguishing features
- Insecurity with variable size payloads
- MAC verification bypass
- Small subgroup attacks
- Hashing domain separation failures
- Inconsistencies with the specifications

We did not review in detail the security proofs and formal arguments from the paper.

Coconut

When reviewing Coconut, we notably looked for exploitable

- Misimplementation of the specification
- BLS12-381 arithmetic and subgroups flaws
- Hash-to-curve implementation flaws
- Zero-knowledge proofs implementation flaws

Furthermore, we checked in detail the implementations' correctness, with respect to the paper's specification (Rust, Go, as well as Python reference). We did not review in detail the security proofs and formal arguments from the paper.

Nym protocols

When reviewing the Nym architecture and sub-protocols, we looked for potential abuse by its parties, or insecure states triggered by edge case of malicious inputs. Given the multitude of protocols, the constant evolution of their implementation details, the complexity of their interaction, and the variety of attack vectors, we would describe our security review as "scratching the surface" of all the possible security risks related to a real deployment of Nym. Greater security assurance will require more rigorous threat modeling, formal reasoning, and automated analysis (for example via fuzzing).

2

Security issues

We reference security issues as I–ID, where is one in:

- SPHX, for SPHINX-related issues
- COCO, for Coconut-related issues
- CRYP, for other crypto-related issues
- PROT, for general protocol-level issues
- NYM, for issues that don't fit in any of the above

2.1 S-SPHX-01: Missing validations of keys

Severity: low

Description

In the nymtech/sphinx repository, crypto/keys.rs does not sufficiently validate private keys and public keys, as we reported in https://github.com/nymtech/sphinx/issues/77. Degenerate keys could use to insecure states, or reduce the cryptographic security properties of the system.

Recommendation

`PrivateKey` and `PublicKey` objects should be checked to be valid as early as possible, namely at deserialization or conversion from another (non-validated) format, as described in the GitHub issue.

2.2 S-COCO-01: Hash-to-scalar biased distribution

Severity: low

Description

When "hashing to a scalar", where a scalar is a BLS12-381 field element, the following code comments were noted:

```
let mut h = D::new();
for point_representation in iter {
    h.update(point_representation);
}
let digest = h.finalize();
```

```
// TODO: I don't like the 0 padding here (though it's what we've been using before,
// but we never had a security audit anyway...)
// instead we could maybe use the `from_bytes` variant and adding some suffix
// when computing the digest until we produce a valid scalar.
let mut bytes = [0u8; 64];

let pad_size = 64usize
    .checked_sub(D::OutputSize::to_usize())
    .unwrap_or_default();

//
bytes[pad_size..].copy_from_slice(&digest);

Scalar::from_bytes_wide(&bytes)
```

The method `Scalar::from_bytes_wide` performs modular reduction in the field $\mathbb{F}_q$ of BLS12-381. There are two potential issues with this code as it stands:

1. mod p reduction introduces bias, except where $H(t)$ may emit integers in the range $[0, np)$ i.e. some multiple of p.

   You may calculate the range of potential outcomes $[0, k)$ from the Hash function (taken as possible integers that may be encoded from the direct output, which will be a power of 2) compare this to the prime modulus. If you are only slightly larger than the modulus, there will be significant bias.

2. `checked_sub()` is redundant here if the size of the digest is known. It is not constant time in its calculation, but this is not important in this case.

Recommendations

1. For the mod p case, you have two choices:

   a) Attempt to hash to a value in a deterministic way, using a 384 bit hash. If the outcome exceeds the value of the 381-bit prime, reject it, increment some counter and feed that into the hash to try again.

   b) With a 512-bit hash, you have 140 bits more than you need. The recommendation for DSA-like algorithms is to have 64 'surplus' bits, so this well exceeds your needs.

2. Remove the padding. Either decode directly from a 48-byte buffer with the aforementioned rejection (solution a) or into a 64-byte buffer with `from_bytes_wide` and accept the negligible bias level.

2.3 S-COCO-02: keygen-cli key files permissions

Severity: low

Description

The files were keys are written have default permissions (typically, 644), whereas they should be more limited (600). testing.)

Recommendation

Set key file permissions to 600 (or equivalent).

2.4 S-CRYP-01: Potential stream cipher IV reuse

Severity: high

Description

In gateway/gateway-requests/src/registration/handshake/shared_key.rs, the following code is used to encrypt content after the handshake between a gateway and a client:

```
/// Encrypts the provided data using the optionally provided initialisation vector,
/// or a 0 value if nothing was given. Then it computes an integrity mac and concatenates it
/// with the previously produced ciphertext.
pub fn encrypt_and_tag(
    &self,
    data: &[u8],
    iv: Option<&IV<GatewayEncryptionAlgorithm>>,
) -> Vec<u8> {
    let encrypted_data = match iv {
        Some(iv) => stream_cipher::encrypt::<GatewayEncryptionAlgorithm>(
            self.encryption_key(),
            iv,
            data,
        ),
        None => {
            let zero_iv = stream_cipher::zero_iv::<GatewayEncryptionAlgorithm>();
            stream_cipher::encrypt::<GatewayEncryptionAlgorithm>(
                self.encryption_key(),
                &zero_iv,
                data,
            )
        }
    };
```

If no IV is provided, zero is used by default, potentially leading to IV reuse, and thus insecure encryption.

The following comment in gateway/gateway-requests/src/types.rs suggests that key may only be used once, but we are not sure of it so prefer to report this issue nonetheless:

```
// Right now the only valid `BinaryRequest` is a request to forward a sphinx packet.
// It is encrypted using the derived shared key between client and the gateway. Thanks to
// randomness inside the sphinx packet themselves (even via the same route), the 0s IV can be used
// HOWEVER, NOTE: If we introduced another 'BinaryRequest', we must carefully examine if a 0s IV
// would work there.
```

Recommendation

If this issue is indeed a security risk, ensure that distinct IVs are used for each use of a given key.

2.5 S-PROT-01: Potential double spend of credentials

Severity: high

Description

In the current version of the protocol, we did not find a way to prevent two (or more) gateways from suffering from concurrency issues, and validating a credential as unused multiple times.

More generally, colluding malicious gateways may perform a number of attacks on the network. A rigorous security analysis seems necessary to understand the attack model and communication model assumed (such as, honest majority, static attacker, reliable broadcast, etc.).

Recommendation

A way to address this problem would be to integrate this validation in the Cosmos consensus, and only granting access after the credential have been accepted by a new round of consensus.

2.6 S-NYM-01: Dependencies with known vulnerabilities

Severity: medium

Description

A number of components pull dependencies in older versions that include known security vulnerabilities.

For example, at the time of writing, sphinx/ relied on an insecure generic-array, nym/ on an insecure tokio, while nym/clients/native had 10 crates with knowns issues (some of which concern code and functionalities used by Nym and which seem exploitable in the context of Nym).

Recommendation

Update to latest versions, or investigate impact of the update in terms of risk and integration. Such an update doesn't need to be done (say) every week, but not too close to the planned release date, in case newer versions have specific changes that require changes in the Nym code (upgrade to a new major version may be delayed because of API changes, if it does not create security risks).

2.7 S-NYM-02: Decryption failure not handled

Severity: low

Description

In nym/common/nymsphinx/acknowledgments/identifier.rs:

```
pub fn recover_identifier(
    key: &AckKey,
    iv_id_ciphertext: &[u8],
) -> Option<SerializedFragmentIdentifier> {

(...)

    let id = stream_cipher::decrypt::<AckEncryptionAlgorithm>(
        key.inner(),
        iv,
        &iv_id_ciphertext[iv_size..],
    );

    let mut id_arr = [0u8; FRAG_ID_LEN];
    id_arr.copy_from_slice(&id);
```

```
    Some(id_arr)
}
```

Here decryption failure cases (for example in case of invalid parameters) is not handled, and might lead to a panic or other insecure state.

A similar issue can be found in common/nymsphinx/src/receiver.rs, in `recover_plaintext()`.

Recommendation

Identify failure cases (for example by receiving a `Result` object. Furthermore, it might be safer to check that `id` is not longer than `FRAG_ID_LEN` (unless guaranteed as a pre-condition of this function by its callers).

If applicable, consider using authenticated encryption (which would increase the ciphertext size).

2.8 S-NYM-03: Panic in fragment IDs generation

Severity: low

Description

New fragment IDs are randomly chosen, using

```
    let potential_id = rng.gen::<i32>().abs();
```

and retries if it's zero. However, because of the way 2-complement encoding works, this will panic with 'attempt to negate with overflow' if `i32::MIN` is the value picked. Indeed, consider the following program:

```
fn main() {
    println!("MAX: {}", i32::MAX.abs());
    println!("MIN: {}", i32::MIN);
}
```

This will print

```
MAX: 2147483647
MIN: -2147483648
```

If the second line is changed to `println!("MIN: {}", i32::MIN);`, the program will thus panic because the absolute value of -2147483648 is not a valid `i32`. There's only a chance in roughly 4 billion that this case happens, but fixing it would avoid the inevitable panics and mysterious crashes when millions of users generate thousands of fragment IDs.

Recommendation

To obtain a random 31-bit value, we recommend to instead pick a random 4-byte value, and clear the MSB (or pick an `u32`). A quick fix would be to check for -2147483648 and discard it.

2.9 S-NYM-04: Mnemonic not zeroized

Severity: low

Description

A BIP39 mnemonic is read from the filesystem's configuration to connect the nymd service and issue a new signature, via

```
let nymd_client = NymdClient::connect_with_mnemonic(
    config.nymd_url.as_str(),
    config.mixnet_contract_address.clone(),
    mnemonic.clone(),
)?;
```

The `mnemonic` is a `bip39::Mnemonic` (from the bip39 crate), but this value is not zeroized after the object goes out of scope, and is cloned into another copy than the initial one. Copies of the memory will thus be written in memory for every new call to `new_signing()`, increasing the risk of exposure of the seed (easily identified in memory, unlike a raw cryptography key).

Recommendation

Since the mnemonic is arguably the most sensitive value in the system, we recommend to reduce its exposure.

3

Observations and suggestions

Here we list observations and suggestions not directly about security risks, but potential improvements, “defense-in-depth”, quality assurance, and performance.

We reference these observations as O–ID, with the same COMPONENT codes as for security issues.

3.1 O-SPHX-01: Insufficient PRNG testing

As reported in https://github.com/nymtech/sphinx/issues/78.

3.2 O-SPHX-02: Deprecated crates

The sphinx repository uses the deprecated crates `aes-ctr` (merged into the `aes` crate) and `stream_cipher` (merged into the `cipher` crate).

(We noted that a PR fixes the `aes` case.)

Also, the feature `wasm-bindgen` of `rand` has been obsolete in version 0.8.4, so will not be available when `rand` is updated.

3.3 O-SPHX-03: Potential leaks from routing information

The version number in the routing information might be exploited to partially deanonymize a client, if sets of clients are known to use (or not to use) a given version. There is probably little that Nym can do to prevent this, except warn users of the potential risk.

3.4 O-COCO-01: keygen-cli panic upon invalid arguments

The command-line utility `keygen-cli` panics instead of gracefully failing when receiving invalid validators/threshold combinations (threshold zero or greater than the number of validators).

3.5 O-COCO-02: Hash-to-curve and point deserialization

When hashing to curve, the following logic is used:

```
let option: Option<G1Affine> = G1Affine::from_compressed_unchecked(&digest_array).into();
if let Some(point) = option {
    let point_projective: G1Projective = point.into();
    return point_projective.clear_cofactor();
}
```

Whilst there is no security concern with this code as presented, we suggest using the `from_compressed()` API for deserialization. Alternatively we recommend code comments to document that the use of `from_compressed_unchecked()` is safe, but alternative uses may not be.

The code in the `from_compressed()` function performs correct logic to check for both curve group membership and q-subgroup membership. The point can then optionally be transformed to projective coordinates (the crate recommends against this for mixed use cases).

If `from_uncompressed_unchecked()` had been used, the `G1Affine` point would accept input that is not a valid point on the curve. The conversion to projective coordinates would not produce a valid curve point, nor would cofactor clearing.

This stems from the fact that the unchecked functions do not check that the curve equation is satisfied. In the "compressed" case, this is fine since the `y` coordinate is "recovered" to satisfy the equation for the given `x` coordinate, so all points from this function should be valid curve points (and so in the overall group but not the desired q-order subgroup).

Note also that clearing the cofactor could produce a point at infinity, which may give rise to "non-contributory" behaviour for some decoded input values.

3.6 O-COCO-04: Projective and affine coordinate use

The `bls12_381` crate suggests using the affine coordinate representation for most efficient use cases, saying of the affine representations:

> It is ideal to keep elements in this representation to reduce memory usage and improve performance through the use of mixed curve model arithmetic.

In the coconut code, however, in multiple places parameters are converted into projective coordinates and then converted back into affine coordinates for use in functions such as `check_bilinear_pairing()`.

We suggest reviewing if this is necessary, and if no benefit is realized, converting to use affine representation everywhere as suggested by the crate.

3.7 O-COCO-05: "Magic constants" code smell

In various places throughout the coconut Rust library, logic like this exists:

```
pub(crate) fn try_deserialize_g2_projective(
    bytes: &[u8; 96],
    err: CoconutError,
) -> Result<G2Projective> {
```

While we understand `96` represents three coordinate values in the BLS12 finite field, the usual problems with magic numbers exist:

- Not all developers will understand this.
- Future proofing the code, for example refactoring to a different curve size, becomes much harder.

We recommend defining these constants in a single place and using them from there. An alternative is the Rust "newtype" pattern, `struct ProjectiveBytes([u8;96])`.

3.8 O-CRYP-01: Use Ristretto points encoding

The Ristretto encoding provides indistinguishable-from-random public key encoding, which might be beneficial to Nym in preventing traffic analysis. However, we have not investigated in detail how it could be integrated and the exact benefits in the context of Nym.

3.9 O-CRYP-02: Suboptimal KDF and MAC interfaces

In nym/common/nymsphinx/params/src/lib.rs:

```
pub type SerializedFragmentIdentifier = [u8; FRAG_ID_LEN];

// TODO: ask @AP about the choice of below algorithms

/// Hashing algorithm used during hkdf for ephemeral shared key generation per
/// sphinx packet payload.
pub type PacketHkdfAlgorithm = blake3::Hasher;

/// Hashing algorithm used during hkdf while establishing long-term shared key
/// between client and gateway.
pub type GatewaySharedKeyHkdfAlgorithm = blake3::Hasher;

/// Hashing algorithm used when computing digest of a reply SURB encryption key.
pub type ReplySurbKeyDigestAlgorithm = blake3::Hasher;

/// Hashing algorithm used when computing integrity (H)Mac for message exchanged
/// between client and gateway.
// TODO: if updated, the pem type defined in
/// gateway\gateway-requests\src\registration\handshake\shared_key
// needs updating!
pub type GatewayIntegrityHmacAlgorithm = blake3::Hasher;
```

Here BLAKE3 is used for various (keyed) hashing operations, via the `blake3` crate. We note that:

- As `PacketHkdfAlgorithm`, BLAKE3 can act as a key derivation function (KDF), which is different from the specific HKDF API (a type of KDF which uses the extract-and-expand construction). To use BLAKE3 as a KDF, one must use the `Hasher::new_derive_key(context: &str)` method.
- The different instances of BLAKE3 as a KDF, namely `GatewaySharedKeyHkdfAlgorithm` and `PacketHkdfAlgorithm`, should use a different context string, for domain separation (for example, strings “Gateway” and “Packet”).
- To use BLAKE3 as a MAC (which is different from the specific HMAC construction), one must use the `Hasher::new_keyed(key: &[u8; 32])` method.

3.10 O-PROT-01: Use Noise-based P2P secure channels

Connections between peer appear to use a custom key agreement and secure transport, whereas an established protocol such as libp2p-noise could provide higher security guarantees, and be integrated via its reference Rust implementation. Security guarantees of libp2p-noise notably include non-repudiation, non-replayability, forward secrecy, and identity hiding.

For peer discovery, if applicable, a protocol such as discv5 may be suitable.

3.11 O-PROT-02: Make random path of mixnodes verifiable

To prevent biases in the selection of random mixing nodes, or the voluntary selection of certain nodes by senders, distributed verifiable randomness might be used, for example via a randomness beacon and/or a verifiable randomness functions.

3.12 O-PROT-03: Use crypto randomness for routes selection

The following comment in topology/src/lib.rs suggests that non-crypto RNG may be acceptable for picking a sequence of mixers:

```
    pub fn random_route_to_gateway<R>(
        &self,
        rng: &mut R,
        num_mix_hops: u8,
        gateway_identity: &NodeIdentity,
    ) -> Result<Vec<SphinxNode>, NymTopologyError>
    where
        // I don't think there's a need for this RNG to be crypto-secure
        R: Rng + ?Sized,
    {
```

A non-crypto RNG may make routes predictable to an attacker, or just reduce the number of possible sequences. Callers of this function however appear to use `RngCore + CryptoRNG`, which is cryptographic.

3.13 O-PROT-04: Prevent use of mixers from a same entity

Future versions of validations and nodes may implement heuristics to identify mixers run by a same party or organization, to ensure that mixes from different layers belong to different parties. One person or organization may for example wish to run different mixers, but should not be trusted in operating multiples layers of a same route. Such a logic may for example be added to `can_construct_path_through()`.

This is not against adversarial collusion, however, for any serious attacker would be careful to operate nodes that cannot be easily linked to one another (different locations, infrastructure, etc.).

3.14 O-NYM-01: Avoid risky `unwrap()` instances

The use of `unwrap()` is potentially unsafe as a panic can happen when an error occurs and is not handled. The developers appear to know what they're doing, as shown by comments such as

```
impl FramedSphinxPacket {
    pub fn new(packet: SphinxPacket, packet_mode: PacketMode) -> Self {
        // If this fails somebody is using the library in a super incorrect way, because
        // they already managed to somehow create a sphinx packet
        let packet_size = PacketSize::get_type(packet.len()).unwrap();
```

or

```
        if let Some((pending_ack_data, queue_key)) = self.pending_acks_data.remove(&frag_id) {
            // this Action is triggered by `RetransmissionRequestListener` which held
            // the other potential reference to this Arc.
            // HOWEVER, before the Action was pushed onto the queue, the reference
            // was dropped hence this unwrap is safe.
            let mut inner_data = Arc::try_unwrap(pending_ack_data).unwrap();
```

That said, in some cases it's less clear to us whether the `unwrap()` is safe. In particular, is it the case that the right response is to crash the program via a `panic()`, for example in

```
impl MixnetResponseListener {
    pub(crate) fn new(
        buffer_requester: ReceivedBufferRequestSender,
        controller_sender: ControllerSender,
    ) -> Self {
        let (mix_response_sender, mix_response_receiver) = mpsc::unbounded();
        buffer_requester
            .unbounded_send(ReceivedBufferMessage::ReceiverAnnounce(mix_response_sender))
            .unwrap();
```

In other cases it is clearer that the use of unwrap presents insecurities. For example in coconut-interface in the main nym crate, the following code:

```
    pub fn public_attributes(&self) -> Vec<Attribute> {
        self.public_attributes
            .iter()
            .map(|x| Attribute::try_from_bs58(x).unwrap())
            .collect()
    }
```

This will panic on any input that is not base-58. Thus, placing a character such as `!` in the attribute description will crash any nym node relying on this code path. Although a signature should be verified on the attributes, the signature is verified on decoded attribute types reported by this function, not on the encoded originals.

For this case, we would recommend simply rejecting any attributes that fail to decode. Signature verification could not be attempted. Alternatively,. `unwrap_or_default` with an empty attribute will cause signature verification to later fail.

In general We recommend that developers review whether such `unwrap()`s are acceptable. Consider for example the following example:

```
fn some_function(input: Option<&str>) {
    let str = input.unwrap();
    println!("Input was: {}", str);
}

fn main() {
    some_function(Some("Message"));
    some_function(None);
}
```

This may be improved with

```
fn some_function(input: Option<&str>) {
    let str = input.expect("some_function was called with None");
    println!("Input was: {}", str);
}

fn main() {
    some_function(Some("Message"));
    some_function(None);
}
```

This will still panic, but will give details. The panic can be eliminated with

```
fn some_function(input: Option<&str>) {
    let str = input.unrap_or("No message Provided");
    println!("Input was: {}", str);
}

fn main() {
    some_function(Some("Message"));
    some_function(None);
}
```

Alternatively a `match` statement could be used to handle the `None` case differently.

Another place where `unwrap()` (and `panic!()`) are used is the `ProxyRunner` implementation, with the following code in `run()`:

```
        if inbound_result.is_err() || outbound_result.is_err() {
            panic!("TODO: some future error?")
        }

        let read_half = inbound_result.unwrap();
        let (write_half, mix_receiver) = outbound_result.unwrap();

        self.socket = Some(write_half.reunite(read_half).unwrap());
```

A final note on Rust errors: the case above handles Option-type matches. The same applies to `Results`, except that a convenience operator exists. If you have a function:

```
fn some_func(args) -> Result<A, B> {
    let x = some_operation()?;
}
```

Then the `?` operator will allow you to propagate the result type from `some_operation` if the error types match. The `or_else()` function on result types may be used to transform error types if needed, and are called only when the result is an `Err`, otherwise the `Ok` value is unwrapped.

Carefully propagating errors throughout the code will make production issues much easier to debug, and reduce user complaints of instability.

The use of `unwrap()` in unit tests is of course not a problem.

The following one-liner may be used to quickly get an overview of `unwrap()` occurrences in a Rust tree, while excluding the unit tests code:

```
fd "\.rs" . -x sed '/cfg(test)/q' {} | rg "unwrap\(\)" -C 2
```

3.15 O-NYM-02: Support safer key storage

Currently the keys used to run a node or validator (including private keys) are stored in clear on the file system. If the server is compromised or part of its stored data leaks, keys could be exposed. For high-value servers (such as validators and staker nodes), this problem is typically addressed by interfacing with a separate key storage or management system, such as KMS, local or remote HSM, via authenticated APIs or PKCS11 interfaces, either to fetch keys or just to perform cryptographic operations with private keys. A “remote signer” API may be defined for this purpose, using for example a design similar to that in EIP-3030.

3.16 O-NYM-03: Use module structure to enforce guarantees

As an example in Coconut-rs, The following function is present:

```
pub(crate) fn construct(
        params: &Parameters,
        pub_key: &elgamal::PublicKey,
        ephemeral_keys: &[elgamal::EphemeralKey],
        commitment: &G1Projective,
        blinding_factor: &Scalar,
        private_attributes: &[Attribute],
        public_attributes: &[Attribute],
    ) -> Self {
        // note: this is only called from `prepare_blind_sign` that already checks
        // whether private attributes are non-empty and whether we don't have too many
        // attributes in total to sign.
        // we also know, due to the single call place, that
        // ephemeral_keys.len() == private_attributes.len()

        // witness creation
```

The comments make it clear that the expected caller will pre-validate certain conditions of the function, but this function is visible to the entire crate. This may potentially introduce bugs in the future, should a refactor call these functions directly and ignore the verification.

Our recommendations:

1. Document the function correctly in a rust-style doc-comment, e.g. //!, so that developers are aware of the function constraints.
2. Rust supports differing levels of visibility. Consider whether `pub(crate)` can be further restricted to the current module to protect against potential misuse.
3. Comment the call to this function with the same requirements, so programmers reading the call to see how it works may also note the comments. This can be as simple as // checks already performed above for properties, see function doc.

3.17 O-NYM-04: Too short fragment IDs? (`FRAG_ID_LEN`)

We noted the following comment in nymsphinx/params/src/packet_sizes.rs:

> // TODO: even though we have 16B IV, is having just 5B (FRAG_ID_LEN) of the ID possibly insecure?

A fragment ID appears to be composed of 4 bytes for the identifier (actually encoded over 31 bits, and the bit left is used for signaling), plus one byte for the fragment position. Fragment IDs are picked randomly (see S-NYM-03), therefore the first repetition of a previously generated ID is expected after roughly $\sqrt{(\pi/2)2^{31}} \approx 58000$ fragment IDs have been generated.

We recommend that the Nym designers assess the related risk, with respect to the number of fragment IDs and impact of a collision, and review the fragment ID format accordingly.