Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Audit-Report noble-ed25519 TypeScript Library 02.2022

Cure53, Dr.-Ing. M. Heiderich, Dr. A. Pirker, Dipl.-Ing. David Gstir

Index

Introduction

"Fastest JS implementation of ed25519 / ristretto255. High-security, auditable, 0-dependency EDDSA signatures and ECDH key agreement"

From https://github.com/paulmillr/noble-ed25519

This report describes the results of a security assessment of the *noble-ed25519* elliptic curve TypeScript implementation. Carried out by Cure53 in January and February of 2022, the project included a dedicated audit of the source code and a comprehensive review of the cryptographic premise.

Registered as *NBL-03,* the project was requested by Paul Miller, the library maintainer, in January 2022. Given the scope, it could be scheduled for upcoming weeks and was ultimately carried out in CW05. A total of five days were invested to reach the coverage expected for this assignment, whereas a team of three testers has been composed and tasked with this project's preparation, execution and finalization.

The work was contained into a single Work Package (WP):

- **WP1**: Crypto reviews and code audits against noble-ed25519 TypeScript implementation

Cure53 was given access to all relevant sources, material supporting documentation to review. Additionally, chat access to the maintainer team for Q&A during the audits and reviews was set up. White-box approaches were favored and deployed in this examination. The project progressed effectively on the whole. All preparations were done in CW04 to foster a smooth transition into the testing phase. As the software is available as open-source, Cure53 just needed to verify which is the correct release to look at.

Over the course of the engagement, the communications were done using a private, dedicated and shared Slack channel with Paul Miller and the involved testers from the Cure53's side. The discussions throughout the test were very good and productive and not many questions had to be asked. The scope was well-prepared and clear, greatly contributing to the fact that no noteworthy roadblocks were encountered during the test.

Cure53 gave frequent status updates about the test and the related findings, live-reporting was not explicitly requested by Paul Miller but several of the findings were discussed in depth in the mentioned Slack channel, so the library maintainer could already start working on fixes and benefit from having the results of repairs reviewed by the auditors.

The Cure53 team managed to get very good coverage over the scope. Among eight security-relevant discoveries, not a single item was classified to be a security vulnerability, hence indicating that all issues represent general weaknesses with lower exploitation potential. In summation, the number of findings is moderate and the spotted items have limited - if any - security impact. This is clearly an outstanding result, testifying to the good security posture of the project. The spotted general weaknesses were not only ranked as *Low* or *Info* but, in fact, most of them have been eradicated before the submission of this final report.

In the following sections, the report will first shed light on the scope and key test parameters, as well as the structure and content of the WPs. A dedicated chapter on test methodology and coverage then clarifies what the Cure53 did in terms of attack-attempts, coverage and other test-relevant tasks. Next, all findings will be discussed in the miscellaneous category in a chronological order. Alongside technical descriptions, PoC and mitigation advice are supplied when applicable. Finally, the report will close with broader conclusions about this early 2022 project. Cure53 elaborates on the general impressions and reiterates the verdict based on the testing team's observations and collected evidence. Tailored hardening recommendations for the noble-ed26619 TypeScript library complex are also incorporated into the final section.

Scope

- **Reviews & Code audits against noble-ed25519 TypeScript implementation**
    - **WP1**: Crypto reviews & code audits against noble-ed25519 TypeScript implementation
        - https://github.com/paulmillr/noble-ed25519/releases/tag/1.6.0-pre-audit
        - Commit:
            - fa14496908cf286da53d17b739accd8f7c3790be
    - **Key focus areas**
        - Possible timing attacks targeting algorithmic resistance
        - Functional correctness of elliptic curve operations in use
        - Safety in the face of known side-channels
        - Elliptic curve validation errors and elliptic-curve-specific attacks
        - Checks against constant-time operations
        - Side-channels and information leaks, secure storage and data processing
        - DoS vectors, information leakage and logic bugs as applicable
        - Secure random number usage and generation as applicable
        - Secure handling of numeric values and floating point numbers
        - Tests against the existing third-party integrations & dependencies
    - **Test-supporting material was shared with Cure53**
    - **All relevant sources were shared with Cure53**

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Vulnerability Summary

Project name	**Number of findings by type**
noble-ed25519	Total: 8 Medium: 1 Low: 5 Informational: 2

Table 1: Overview of the findings under the given scope and threat model

Testing Methodology

This section describes the testing methodology and the resulting coverage of the security audit against the *noble-ed25519-1.6.0-pre-audit* repository. The repository contains an implementation for generating and verifying signatures for the elliptic twisted Edwards curve Ed25519, an implementation of ECDH over curve25519 (X25519), as well as an implementation of Ristretto (ristretto255).

Overall, Cure53 was able to achieve good coverage of the areas in scope and the review focused primarily on the following areas and tasks:

- Possible timing attacks targeting algorithmic resistance
- Functional correctness of elliptic curve operations in use
- Safety in the face of known side-channels
- Elliptic curve validation errors and elliptic-curve-specific attacks
- Checks against constant-time operations
- Side-channels and information leaks, secure storage and data processing
- DoS vectors, information leakage and logic bugs as applicable
- Secure random number usage and generation as applicable
- Secure handling of numeric values and floating point numbers
- Tests against the existing third-party integrations & dependencies
- Tests against known implementation vulnerabilities

The implementation of *ristretto255* was checked for standards-compliance[1]. It was quickly noticed that the implementation of Ristretto used in the tested realm is fully standard-compliant, revealing no deviations whatsoever.

[1] https://ristretto.group/ristretto.html

The *Ed25519* implementation of the *verify* and *sign* functions appears to support two standards: RFC 8032[2] and ZIP215[3]. The former standard requires more stringent checks of input parameters. In particular, for a signature (R,S) it should be validated that the parameter R is a valid curve point and S is within the range 0 <= s < L, wherein L is the order of the curve. Furthermore, the public key A needs to be a valid curve point as well. Here, a valid curve point means that also the y coordinate of the point is < p, where p is again a constant specific to *Ed25519*, specifically 2^{255} - 19. The ZIP215 standard relaxes these conditions for R and A slightly, thereby allowing the y coordinate to be unreduced 2^{255} - 19.

It was observed that the *Ed25519* implementation is not fully standards-compliant with regard to the validation criteria, neither RFC 8032 nor ZIP215. This is further explained in NBL-03-001 and NBL-03-003. Since RFC 8032 and ZIP215 actually contradict each other in terms of verification, another item is required. This could either concern two dedicated functions or a parameter where the caller can select the standard to verify against.

Besides the observations noted above both implementations appear cryptographically sound and correct. No issues related to the implementation of the *verify* and *sign* functions were identified.

Timing attacks represent a common issue within elliptic curve cryptography. One popular target is the scalar point multiplication during signature generation. The noble-ed25519 codebase attempts to rely on constant-time, however, it was discovered that it fails to fully achieve this. One reason is that the code omits usage of constant-time operations (e.g., the data-dependent branching operation in the *mod* function), and the selection of JavaScript as a programming language. The latter does not directly support constant-time operation, namely JavaScript is commonly just-in-time compiled, hence making the aforementioned impossible to guarantee.

As noble-ed25519 is written in JavaScript and JavaScript is commonly just-in-time-compiled, it is not possible to guarantee constant-time operations. This is a fact that is known and the *README.md* file of noble-ed22519 explains this subject matter. As a result of this, Cure53 refrained from reporting any findings related to non-constant time operations. Furthermore, the utility implementations *Point*, *RistrettoPoint* and *ExtendedPoint* were reviewed. All of them appear sound, however, for addition and subtraction in *Point*, it was identified that - in principle - points which are not on the curve could be provided. This issue is summarized in NBL-03-002. Similar observations apply to the Ristretto point's addition and subtraction, as explained in NBL-03-004. Finally, the

[2] https://datatracker.ietf.org/doc/html/rfc8032
[3] https://zips.z.cash/zip-0215

transformation from an extended point back to an affine point accepts a parameter for the inverse of the *z* component for performance speed-ups. However, negative values, invalid inverse elements or even zero are allowed, as shown in NBL-03-005. Cure53 also noted that the comparison of two extended points for equality was not done properly, as captured by NBL-03-007.

Apart from the issues indicated above, the algorithms in place for addition and multiplication appear correct and sound. The multiplication of a point with a scalar was also investigated, given that it uses the wNAF method for performance-speed-up. For that purpose, it keeps a weak map of points associated with pre-computed values, potentially opening a side-channel for timing-attacks due to shorter look-up times. However, the auditors were unable to deliver a Proof-of-Concept which would demonstrate subject matter.

To support X25519, noble-ed25519 offers two functions. The first is *getSharedSecret,* which accepts the same format for private and public keys as the Ed25519 code, converting them to X25519 format and then calls *curve25519.scalarMult.* The second function performs an ECDH in principle. Contrary to Ed25519, the scalar multiplication is implemented here using the Montgomery Ladder algorithm[4], as suggested in the original X25519 paper[5]. The principle further rejects non-contributory scalars. This is not strictly required for ECDH but needed for other use-cases[6]. During the audit of the code, no security-relevant vulnerabilities were uncovered in the allotted time-frame. The *verify* operation of Ed25519 and the *curve25519.scalarMult* operation used for X25519 were also tested against a ported version of Google's Project Wycheproof[7]. This confirmed the signature malleability issues documented in NBL-03-003. For the *curve25519.scalarMult*, no issues were identified using this tool.

The generation of random bytes is done through *crypto.node* or *crypto.web*, which allows for the secure generation of random values for cryptographic use. Therefore, no issues related to the generation of random private keys were spotted. However, the method *hashToPrivateKey*, which according to the developers should not be used for the generation of a private key for Ed25519, has a misleading name. In fact, it should be essentially used for the generation of a private scalar rather than for a private key, which is elaborated on under NBL-03-006.

[4] https://en.wikipedia.org/wiki/Elliptic_curve_point_multiplication#Montgomery_ladder
[5] https://cr.yp.to/ecdh/curve25519-20060209.pdf
[6] https://cr.yp.to/ecdh.html#validate
[7] https://github.com/google/wycheproof

Lastly, the external dependencies with regard to other libraries and packages were verified and it was found that the library uses a deprecated hash library, as summarized in NBL-03-008.

Miscellaneous Issues

This section covers those noteworthy findings that did not lead to an exploit but might aid an attacker in achieving their malicious goals in the future. Most of these results are vulnerable code snippets that did not provide an easy way to be called. Conclusively, while a vulnerability is present, an exploit might not always be possible.

NBL-03-001 WP1: *verify* method non-compliant with RFC 8032 (*Info*)

While investigating the code repository of the *noble-ed25519-1.6.0-pre-audit*, it was observed that the *verify* method relaxes conditions for the input parameters, normally imposed by RFC 8032[8]. Specifically, section 5.1.7 of RFC 8032 requires the following validations:

- Decode the point *R* from the first half of the signature bytes as a point on the curve; fail if results correspond to an invalid point.
- Decode the value *S* from the second half of the signature bytes, and make sure it is within the range *0 <= s < L* where *L* is a constant specific to Ed25519; fail otherwise.
- Decode the public key *A* as a point on the Ed25519 curve, and fail if *A* is not a valid point on the Ed25519 curve.

The current implementation is not compliant with RFC 8032. In case the caller provides parameters different from the *Signature* type for the signature and different from *Point* for the public key, it always applies the fromHex methods with *strict* set to *false*. In effect, this allows public keys as points, in accordance with ZIP215 rather than RFC 8032.

Furthermore, the code in *Point.fromHex* fails to be fully RFC 8032-compliant even with the strict mode enabled. It does not adhere to the following requirement from section 5.1.3.:

> *Use the x_0 bit to select the right square root. If*
> *x = 0, and x_0 = 1, decoding fails. Otherwise, if x_0 != x mod*
> *2, set x <-- p - x, return the decoded point (x,y).*

Whenever the conditions of x = 0 and x_0 = 1 are met, the decoding logic of *Point* does not fail in strict mode.

[8] https://datatracker.ietf.org/doc/html/rfc8032#section-5.1.7

Affected file:
index.ts

Affected code:
```
export async function verify(sig: SigType, message: Hex, publicKey: PubKey):
Promise<boolean> {
        [...]
        message = ensureBytes(message);
        if (!(publicKey instanceof Point)) publicKey = Point.fromHex(publicKey,
        false);
        if (!(sig instanceof Signature)) sig = Signature.fromHex(sig, false);
        const SB = ExtendedPoint.BASE.multiplyUnsafe(sig.s);
        const k = await sha512ModqLE(sig.r.toRawBytes(), publicKey.toRawBytes(),
        message);
        [...]
}

// Converts hash string or Uint8Array to Point.
// Uses algo from RFC8032 5.1.3.
static fromHex(hex: Hex, strict = true) {
  [...]
  // 4.  Finally, use the x_0 bit to select the right square root.  If
  // x = 0, and x_0 = 1, decoding fails.  Otherwise, if x_0 != x mod
  // 2, set x <-- p - x.  Return the decoded point (x,y).
  const isXOdd = (x & _1n) === _1n;
  const isLastByteOdd = (hex[31] & 0x80) !== 0;
  if (isLastByteOdd !== isXOdd) {
    x = mod(-x);
  }
  return new Point(x, y);
}
```

It is recommended to differentiate which parameter validation the *verify* method applies. This can be done by a boolean parameter to the *verify* function which could be passed on further to the *Point.fromHex* and *Signature.fromHex* methods, similar to the handling of the *strict* parameter. In order to be compliant with RFC 8032, the strict parameters need to be set to *true*.

It is further recommended to adapt the decoding logic in *Point.fromHex* to exactly match the RFC when used in strict mode, so as to avoid implementation fingerprinting.

NBL-03-002 WP1: Missing validation on point/extended point addition *(Low)*

During a source code review of the *noble-ed25519-1.6.0-pre-audit* repository, it was observed that the point addition and extended point addition (hence also the subtraction), skip the validation of the provided point. They do not check whether the point to add is a valid point on the curve. Such a check would involve the verification of the equation $x^2 = (y^2 - 1) / (d\ y^2 + 1)\ (mod\ p)$, where d and p denote curve constants. Therefore, it is possible to perform point additions and subtractions with points and extended points which are not on the curve.

Affected file:
index.ts

Affected code:
```
class ExtendedPoint {
	[...]
	add(other: ExtendedPoint) {
		const X1 = this.x, Y1 = this.y, Z1 = this.z, T1 = this.t; //
		prettier-ignore
		const X2 = other.x, Y2 = other.y, Z2 = other.z, T2 = other.t; //
		prettier-ignore
		const A = mod((Y1 - X1) * (Y2 + X2));
		const B = mod((Y1 + X1) * (Y2 - X2));
		const F = mod(B - A);
		if (F === _0n) return this.double(); // Same point.
		const C = mod(Z1 * _2n * T2);
		const D = mod(T1 * _2n * Z2);
		const E = mod(D + C);
		const G = mod(B + A);
		const H = mod(D - C);
		const X3 = mod(E * F);
		const Y3 = mod(G * H);
		const T3 = mod(E * H);
		const Z3 = mod(F * G);
		return new ExtendedPoint(X3, Y3, Z3, T3);
	}
	[...]
}
[...]
class Point {
	[...]
	add(other: Point) {
		return
		ExtendedPoint.fromAffine(this).add(ExtendedPoint.fromAffine(other)
		).toAffine();
	}
	[...]
}
```

It is recommended to provide two methods for point and extended point addition, one with point validation and one without it. The latter could be used in case of performance-critical operations. Furthermore, the documentation should clearly state that the functions without point validation do not validate the points in any form.

NBL-03-003 WP1: Signature malleability due to insufficient validation (*Medium*)

***Note**: This issue was resolved[9] during the security assessment and no longer exists, as verified by Cure53.*

During a source code review of the *noble-ed25519-1.6.0-pre-audit* repository, it was observed that the *verify* method is not fully compliant with ZIP215[10] and RFC 8032. The ZIP215 standard relaxes the conditions on the *y* coordinate of points from the Ed25519 curve in the sense that it allows for *y* coordinates larger than or equal to *p*, wherein *p* is a constant. The commonality between the standards is that the value *s* of a signature is still required to be within the range *0* <= *s* < *L*, where *L* is a constant.

Affected file:
index.ts

Affected code:
```
function normalizeScalar(num: number | bigint, max: bigint, strict = true):
bigint {
        if (!max) throw new TypeError('Specify max value');
        if (typeof num === 'bigint') {
                if (strict) {
                        if (_0n < num && num < max) return num;
                } else {
                        if (_0n <= num && num < MAX_256B) return num;
                }
        }
        if (typeof num === 'number' && Number.isSafeInteger(num)) {
                if (strict) {
                        if (0 < num) return BigInt(num);
                 } else {
                        if (0 <= num) return BigInt(num);
                }
        }
        throw new TypeError('Expected valid scalar: 0 < scalar < max');
}
[...]
```

[9] https://github.com/paulmillr/noble-ed25519/commit/8c08ff6750cf0604b3089891c397061cf9464724
[10] https://zips.z.cash/zip-0215

```
export async function verify(sig: SigType, message: Hex, publicKey: PubKey):
Promise<boolean> {
        [...]
        message = ensureBytes(message);
        if (!(publicKey instanceof Point)) publicKey = Point.fromHex(publicKey,
        false);
        if (!(sig instanceof Signature)) sig = Signature.fromHex(sig, false);
        [...]
}
```

The consequence of this is that two signatures for the same message will be considered valid by the *verify* method, as the following code snippet illustrates:

```
import * as ed from '@noble/ed25519';

(async () => {
        const privateKey = ed.utils.randomPrivateKey();
        const message = Uint8Array.from([0xab, 0xbc, 0xcd, 0xde]);
        const publicKey = await ed.getPublicKey(privateKey);
        const signature = await ed.sign(message, privateKey);

        const sig_valid = await ed.Signature.fromHex(signature);
        const sig_invalid = await new ed.Signature(sig_valid.r, sig_valid.s +
        ed.CURVE.l, false);

        const sig_invalid_bytes = await sig_invalid.toHex();

        const isValid = await ed.verify(signature, message, publicKey);
        const isValid2 = await ed.verify(sig_invalid_bytes, message, publicKey);

        console.log('Done');
})();
```

Both variables *isValid* and *isValid2* evaluate to *true*. This was also confirmed using a ported version of Google Project Wycheproof with test vectors 63-66 from the EdDSA set being accepted as valid signatures when they should not be seen as such[11]. As signature malleability could be problematic in the blockchain context[12], this finding was given a *Medium* severity level.

It is recommended to check that the provided value *s* of the signature is within the range of $0 <= s < L$, as specified in ZIP215 and RFC 8032.

[11] https://github.com/google/wycheproof/blob/master/testvectors/eddsa_test.json#L529-L568
[12] https://en.bitcoin.it/wiki/Transaction_malleability

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

NBL-03-004 WP1: Missing validations for RistrettoPoint operations *(Low)*

During a source code review of the *noble-ed25519-1.6.0-pre-audit* repository, it was observed that the *RistrettoPoint* class implements methods for addition and subtraction. For that purpose, these methods accept RistrettoPoint as parameters, transform them into extended coordinates, perform the operation and transform them back.

However, for addition and subtraction, the implementation fails to verify whether the provided points are indeed valid Ristretto points. In the worst case, someone could add a valid Ristretto point with a non-RistrettoPoint, potentially leading to a non-Ristretto point being accepted.

Affected file:
index.ts

Affected code:
```
add(other: RistrettoPoint): RistrettoPoint {
       return new RistrettoPoint(this.ep.add(other.ep));
}

subtract(other: RistrettoPoint): RistrettoPoint {
       return new RistrettoPoint(this.ep.subtract(other.ep));
}
```

It is recommended to perform validity checks inside the *add* and *subtract* methods.

NBL-03-005 WP1: Missing validations on extended point conversion *(Low)*

***Note**: This issue was resolved[13] during the security assessment and no longer exists, as verified by Cure53.*

During a source code review of the *noble-ed25519-1.6.0-pre-audit* repository, it was observed that the *ExtendedPoint* class provides a method for transforming the extended point to an affine point.

For that purpose, the method *toAffine* accepts a parameter named *invZ*, which the function uses to calculate the *x* and *y* component of the affine point. However, the implementation does not check whether the provided *invZ* value corresponds to the inverse of the *z* component of the extended point. This essentially permits invalid inverses of the *z* component, but also provides, for example, the possibility of value *0*, which should not be allowed for this transformation.

[13] https://github.com/paulmillr/noble-ed25519/commit/e51f79bc23d7fdb5dfdec56f1d26639f23b5158c

Affected file:
index.ts

Affected code:
```
toAffine(invZ: bigint = invert(this.z)): Point {
       const x = mod(this.x * invZ);
       const y = mod(this.y * invZ);
       return new Point(x, y);
}
```

It is recommended to verify the provided *invZ* parameter with regard to its validity, including a check for being greater than zero, as well as checking that it is indeed the inverse of *z* (for example by verifying *mod(this.z * invZ) == 1n*).

NBL-03-006 WP1: Improper naming of the private key generation function *(Info)*

***Note**: This issue was resolved[14] during the security assessment and no longer exists, as verified by Cure53.*

During a source code review of the *noble-ed25519-1.6.0-pre-audit* repository, it was observed that the library supports the generation of Ed25519 private keys by using either the function *randomPrivateKey* or *hashToPrivateKey*.

The *randomPrivateKey* randomly samples 32 bytes using a secure random function, whereas *hashToPrivateKey* takes the provided parameter module *L* and excludes the 1-element and the 0-element. This is not in accordance with the key generation process outlined in RFC 8032.[15]

Affected file:
index.ts

Affected code:
```
hashToPrivateKey: (hash: Hex): Uint8Array => {
       hash = ensureBytes(hash);
       if (hash.length < 40 || hash.length > 1024)
              throw new Error('Expected 40-1024 bytes of private key as per FIPS
              186');
       const num = mod(bytesToNumberLE(hash), CURVE.l);
       // This should never happen
```

[14] https://github.com/paulmillr/noble-ed25519/commit/00fc580d53eacc9e4844a2789336219e51028b44
[15] https://datatracker.ietf.org/doc/html/rfc8032

```
        if (num === _0n || num === _1n) throw new Error('Invalid private key');
        return numberToBytesLEPadded(num, 32);
    }
```

It is recommended to rename the function in order to better reflect its actual purpose and avoid possible confusion.

NBL-03-007 WP1: Insufficient equality comparison for extended points *(Low)*

***Note**: This issue was resolved[16] during the security assessment and no longer exists, as verified by Cure53.*

The *noble-ed25519* implementation uses extended homogeneous coordinates[17] (X, Y, Z, T) with x = X/Z, y = Y/Z, x * y = T/Z for its Ed25519 signature creation and verification. To compare two instances of such points, the *ExtendedPoint* class implements a *equals()* method. It works by comparing the T element of both points. However, this is insufficient due to not guaranteeing that both points are in fact identical.

To exemplify, consider p1 = (0, 10, 1, 0) and p2 = (42, 0, 1, 0). These points have different values for X and Y, but identical values for T. Hence, they would be treated as equal by this method. This also works with the non-zero values, e.g., p3 = (3, 14, 1, 42) and p4 = (2, 24, 1, 42) which would be considered equal.

This will result in points being treated as equal even if they are not. Similarly, a comparison with the *ZERO* point would return *true* when it should not do so.

Affected file:
index.ts

Affected code:
```
equals(other: ExtendedPoint): boolean {
  const a = this;
  const b = other;
  return mod(a.t * b.z) === mod(b.t * a.z);
}
```

To compare two Edwards Points without converting the points into affine coordinates, it is recommended to follow a similar approach as in the *curve25519_dalek* Rust library[18],

[16] https://github.com/paulmillr/noble-ed25519/commit/e51f79bc23d7fdb5dfdec56f1d26639f23b5158c
[17] https://www.rfc-editor.org/rfc/rfc8032#section-5.1.4
[18] https://doc.dalek.rs/src/curve25519_dalek/edwards.rs.html#341

which essentially compares *x* and *y* coordinates of two points without performing a transformation into affine coordinates.

NBL-03-008 WP1: Use of deprecated hash library when run in Deno *(Low)*

***Note**: This issue was resolved[19] during the security assessment and no longer exists, as verified by Cure53.*

The *noble-ed25519* can be used in the browser, Node.js or Deno. Deno is a more modern alternative to Node.js, supporting JavaScript as well as TypeScript. The Deno version of *noble-ed25519* is exposed via the file *mod.ts* which is just a wrapper around the main *index.ts* file.

As Deno does not have the same API as Node.js, the SHA512 implementation exposed via *utils.sha512* is replaced with the one from Deno's standard library, using the deprecated Deno *std/hash* namespace and its deprecated SHA512 implementation[20]. The newer and still maintained namespace is *std/crypto,* which implements the Web Crypto API[21] and uses a WASM-compiled version of the Rust crate *sha2*[22].

Affected file:
mod.ts

Affected code:
```
import { getPublicKey, sign, verify, utils, CURVE, Point, ExtendedPoint } from
'./index.ts';
import { Sha512 } from 'https://deno.land/std@0.119.0/hash/sha512.ts';

utils.sha512 = async (message: Uint8Array): Promise<Uint8Array> => {
  return new Uint8Array(new Sha512().update(message).arrayBuffer());
};

export { getPublicKey, sign, verify, utils, CURVE, Point, ExtendedPoint };
```

It is recommended to switch to the maintained version of SHA512 provided by Deno.

[19] https://github.com/paulmillr/noble-ed25519/commit/15defe186870a493025777410189db293c6ba35d
[20] https://deno.land/std@0.119.0/hash/sha512.ts
[21] https://deno.land/std@0.119.0/crypto/mod.ts
[22] https://docs.rs/crate/sha2/latest

Conclusions

The assessment featured one repository, namely *noble-ed25519-1.6.0-pre-audit*. The main focus of this Cure53 investigation was to identify possible timing attacks and judge the repository's safety against known side-channels in general. Furthermore, the involved testers were to ensure the functional correctness of the elliptic curve operations in use, looking also at the potential for curve validation errors or curve-specific attacks. Checks for information leaks were also performed, while evaluations of storage and data processing in general served as a backdrop. Secure generation of random and the secure handling of numeric and floating point values were also verified.

The repository contained two source files, namely *index.ts* and *mod.ts.* They are used for implementing signature generation and verification for the elliptic curve *Curve25519 (Ed25519)*, ECDH using the same curve (X25519), as well as the implementation of Ristretto (ristretto255). The code is very clean and robust. It is evident that the developer(s) are fully aware of secure coding principles.

The Ristretto implementation is in accordance with the Ristretto proposal and the signature generation complies with Ed25519 from RFC 8032 as well. The signature verification implementation, however, neither fully complies with ZIP215 nor with RFC 8032, as described in NBL-03-001 and NBL-03-003. Further dynamic testing of the library with a ported version of Google's Project Wycheproof confirmed NBL-03-003 but did not reveal any other issues.

In total, the assessment revealed eight miscellaneous issues. The majority pertain to non-standard compliance, like NBL-03-001 and NBL-03-003. Alternatively, they relate to missing validations. Even though not directly leading to a security issue at the moment, the missing validation may still lead to invalid points or operations which should not be executable. Lastly, one use of a deprecated, but not insecure SHA512 library, was found when used with the Deno runtime (see NBL-03-008).

As already mentioned within the testing methodology section, issues related to timing attacks due to non-constant time operations were not reported and treated as findings, since the author is fully aware of this subject matter in connection to using JavaScript. Confirmation can be seen in the *README.md* of the project.

Cure53 was in constant communication with the customer and frequently sent status updates and raised questions or concerns. The communication with the client on Slack was excellent and help was provided whenever requested. The testing team could actually perform fix verification towards the end of this review.

The outcome of this source code review demonstrates that the repository is in a good shape from a security perspective. In light of the findings gathered during this Cure53 examination, only minor adjustments are required to harden the already good security posture of the library.

Cure53 would like to thank Paul Miller for his excellent project coordination, support and assistance, both before and during this assignment.