Fine penetration tests for fine websites

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

# Review-Report noble-secp256k1 Library 04.2021

Cure53, Dr.-Ing. M. Heiderich, Dr. N. Kobeissi

## Index



## Introduction

*“Fastest JS implementation of secp256k1, an elliptic curve that could be used for asymmetric encryption, ECDH key agreement protocol and signature schemes.”*

From https://github.com/paulmillr/noble-secp256k1

This report describes the results of a cryptography review and source code audit against the noble-secp256k1 JavaScript crypto library. The work was requested by Paul Miller, the maintainer of the library in early April 2021. It was quickly scheduled and carried out by Cure53 in mid-April, namely in CW15 and CW16.

In terms of resources, a total of five days were invested to reach the coverage expected for this project, whereas the testing team consisted of two senior testers assigned to this project’s preparation, execution and finalization.

All preparatory tasks were done in early April 2021, namely in CW14, so as to ensure an efficient cooperation during the core period of this assignment. Cure53 was given access to all relevant sources which are available publicly as Open Source Software anyway. What is more, all necessary material and review-supporting documentation were also furnished to the Cure53 testing team. In summation, the methodology chosen here was

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

white-box. The areas chosen and prioritized for the cryptography reviews and audit were delineated and listed as follows:

- Timing attacks targeting noble's algorithmic resistance against those
- Functional correctness of elliptic curve operations in use
- Safety against known side channels
- Checks against elliptic curve validation errors
- Checks against elliptic-curve-specific attacks
- General checks against constant-time operations
- Misuse prevention of high-level cryptographic API

The project moved forward as scheduled. Communications during the test were done using a dedicated, shared Slack channel which was used to connect the workspaces of the relevant entities partaking in the project. Once the audits and reviews got started, communications were very smooth and fruitful. Not many questions had to be asked, the scope was well-prepared and no noteworthy roadblocks were encountered during the test. Cure53 offered frequent status updates about the test and the emerging findings, so as to let the maintainer react accordingly.

More broadly, the Cure53 team managed to get very good coverage over the given scope items and made four security-relevant discoveries. Three items were classified to be security vulnerabilities and one should be seen as general weakness with lower exploitation potential. Note that the one finding with *High* severity ratings was classified as such given the large damage potential in case a developer uses the library wrongly. Additionally, this risk could be combined with the weird type-comparison properties of JavaScript and justifies the elevated marker. This was further discussed with the library maintainer prior to the report's finalization.

In the following sections, the report will first shed light on the scope and key test parameters, as well as the areas selected for closer inspection. Next, all findings will be discussed in grouped vulnerability and miscellaneous categories, then following a chronological order in the latter. Alongside technical descriptions, PoC and mitigation advice are supplied when applicable, together with notes on the status of fixes. Finally, the report will close with broader conclusions about this April 2021 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-secp256k1 JavaScript crypto library are also incorporated into the final section.

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

## Scope

- **Cryptography Reviews & Code Audits against noble-secp256k1 JavaScript Library**
  - **WP1**: Cryptography Reviews & Audits against noble-secp256k1 JS Library
    - https://github.com/paulmillr/noble-secp256k1
  - **Focus areas for this audit**
    - Timing attacks targeting noble's algorithmic resistance against those
    - Functional correctness of elliptic curve operations in use
    - Safety against known side-channels
    - Checks against elliptic curve validation errors
    - Checks against elliptic-curve-specific attacks
    - General checks against constant-time operations
    - Misuse prevention of high-level cryptographic API
  - **Test-supporting Material**
    - https://paulmillr.com/posts/noble-secp256k1-fast-ecc/
  - **Sources were available as OSS, see above**

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

## Identified Vulnerabilities

The following sections list both vulnerabilities and implementation issues spotted during the testing period. Note that findings are listed in chronological order rather than by their degree of severity and impact. The aforementioned severity rank is simply given in brackets following the title heading for each vulnerability. Each vulnerability is additionally given a unique identifier (e.g. *NBL-01-001*) for the purpose of facilitating any future follow-up correspondence.

### NBL-01-001 Crypto: Boolean value accepted as key pair basis *(High)*

***Note**: commit 9e7f4c610ad0d799a7cb7ba9cbfabc8d60b37a3f was confirmed to address this issue.*

It was observed that the noble top-level API accepted Boolean values as the basis (i.e. private key values) for key pairs. Those signify output in response to *secp.getPublicKey(true),* a value that is equivalent to *secp.getPublicKey(1)*:

```
> secp.getPublicKey(1)
public key: Uint8Array(65) [
    4, 198,   4, 127, 148,  65, 237, 125, 109,  48,  69,
   64, 110, 149, 192, 124, 216,  92, 119, 142,  75, 140,
  239,  60, 167, 171, 172,   9, 185,  92, 112, 158, 229,
   26, 225, 104, 254, 166,  61, 195,  57, 163, 197, 132,
   25,  70, 108, 234, 238, 247, 246,  50, 101,  50, 102,
  208, 225,  35, 100,  49, 169,  80, 207, 229,  42
]

> secp.getPublicKey(true)
public key: Uint8Array(65) [
    4, 198,   4, 127, 148,  65, 237, 125, 109,  48,  69,
   64, 110, 149, 192, 124, 216,  92, 119, 142,  75, 140,
  239,  60, 167, 171, 172,   9, 185,  92, 112, 158, 229,
   26, 225, 104, 254, 166,  61, 195,  57, 163, 197, 132,
   25,  70, 108, 234, 238, 247, 246,  50, 101,  50, 102,
  208, 225,  35, 100,  49, 169,  80, 207, 229,  42
]
```

The issue likely arises from the fact that *Point.fromPrivateKey* does not sufficiently type-check input values:

```
static fromPrivateKey(privateKey: PrivKey) {
    return Point.BASE.multiply(normalizePrivateKey(privateKey));}
```

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

*TypeScript* may also be to blame, as the *PrivKey* type does not seem to rule out a value of *true* despite the syntax suggesting otherwise:

```
type PrivKey = Hex | bigint | number;
```

JavaScript is a notoriously type-unsafe language, making it likely for all of the noble cryptographic deployments to target JavaScript code environments. This could lead to situations where a private key input is mangled into a Boolean, with the resulting input still treated as valid by the public key generation API.

It is recommended to impose more stringent checks on private key inputs, as well as other potential inputs such as public keys, message payloads and signature payloads.

### NBL-01-003 Crypto: Mangled hex inputs accepted as payloads *(Medium)*

***Note**: commit 8f7fa1ae8f8e4ec13a087b6c48fdb62425592d98 was confirmed to address this issue.*

It was found that the noble API accepted mangled hexadecimal string inputs. Those could, for example, include half-byte values, thus possibly resulting in processing ambiguity. To clarify, one could expect a hex input value of *“aabbc”* to either map to *“0aabbc”* as a correct and the established norm for hexadecimal, or to a malformed one. Specifically, it could potentially be the result of a malformed *“aabbc0”* value, which is especially possible if the hex string was constructed based on a JavaScript parser processing a binary format.

Currently, noble will correct odd hex strings by manually prepending a *0*:

```
function hexToBytes(hex: string): Uint8Array {
 hex = hex.length & 1 ? `0${hex}` : hex;
 const array = new Uint8Array(hex.length / 2);
 for (let i = 0; i < array.length; i++) {
   let j = i * 2;
   array[i] = Number.parseInt(hex.slice(j, j + 2), 16);
 }
 return array;
}
```

However, the resulting ambiguity could lead to incorrect outputs being obtained on input that is presumed correct. Therefore, it is recommended to enforce that all input hex strings have an even number of characters.

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

### NBL-01-004 Crypto: Arbitrary reversal of key share input order *(Medium)*

***Note**: commit 9e7f4c610ad0d799a7cb7ba9cbfabc8d60b37a3f was confirmed to address this issue.*

It was observed that the noble API will attempt to detect an inverse order of public and private key inputs to the *getSharedSecret()* function. If an inverted order is found (i.e. the private key is in fact a public key and vice versa), the API will reverse the inputs before continuing with the function's logic:

```
export function getSharedSecret(privateA: PrivKey, publicB: PubKey, isCompressed
= false): Hex {
 if (isPub(privateA) && !isPub(publicB)) {
   [privateA, publicB] = [publicB as PrivKey, privateA as PubKey];
 } else if (!isPub(publicB)) {
   throw new Error('Received invalid keys');
 }
 const b = publicB instanceof Point ? publicB : Point.fromHex(publicB);
 b.assertValidity();
 const shared = b.multiply(normalizePrivateKey(privateA));
 return typeof privateA === 'string'
   ? shared.toHex(isCompressed)
   : shared.toRawBytes(isCompressed);
}
```

It is more likely in the security-context of critical key share inputs that no such arbitrary reversal should occur. In such marginal cases, the library should rather return an error in case incorrect key share inputs are provided.

In order to avoid the currently incorrect adherence to key share input order due to human error, it is recommended to simply accept an object that clearly labels the key inputs, thereby turning the function's signature to *getSharedSecret(keyShares: {private: [...], public: [...]})*. This would ensure that the application layer will have to label key shares according to their role before passing them down to the low-level cryptographic API.

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

## 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-01-002 Crypto: Scalar multiplication control flow considerations *(Info)*

***Note**: This issue was confirmed as out-of-scope with the noble crypto team prior to publication. Furthermore, commit 25909c4c4a6f9fe47647ebf5bc56fedf493c6cc2 was introduced with additional improvements to endomorphism calculations conducted as part of the noble crypto's scalar multiplication function.*

It was observed that the Jacobian scalar multiplication step included a minor branching condition in the event that endomorphisms are used:

```
multiply(scalar: number | bigint, affinePoint?: Point): JacobianPoint {
   if (typeof scalar !== 'number' && typeof scalar !== 'bigint') {
     throw new TypeError('Point#multiply: expected number or bigint');
   }
   let n = mod(BigInt(scalar), CURVE.n);
   if (n <= 0) {
     throw new Error('Point#multiply: invalid scalar, expected positive
integer');
   }
   // Real point.
   let point: JacobianPoint;
   // Fake point, we use it to achieve constant-time multiplication.
   let fake: JacobianPoint;
   if (USE_ENDOMORPHISM) {
     const [k1neg, k1, k2neg, k2] = splitScalarEndo(n);
     let k1p, k2p, f1p, f2p;
     [k1p, f1p] = this.wNAF(k1, affinePoint);
     [k2p, f2p] = this.wNAF(k2, affinePoint);
     if (k1neg) k1p = k1p.negate();
     if (k2neg) k2p = k2p.negate();
     k2p = new JacobianPoint(mod(k2p.x * CURVE.beta), k2p.y, k2p.z);
     [point, fake] = [k1p.add(k2p), f1p.add(f2p)];
   } else {
     [point, fake] = this.wNAF(n, affinePoint);
   }
   // Normalize `z` for both points, but return only real one
   return JacobianPoint.normalizeZ([point, fake])[0];
 }
```

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

As discussed in the noble crypto *README* file - namely its *"Security"* subsection, it is unlikely that keeping or removing code similar to the branching conditions described above would result in a tangible impact on the practical side-channel resistance of the noble crypto library. However, given that scalar multiplication tends to be where elliptic curve side-channel attacks are most commonly exploited, this branching condition is noted here for completeness. It is unclear if a practical fix is possible.

## Conclusions

This Cure53 examination of the noble crypto library has not led to overly numerous security-relevant discoveries. After spending five days on this assignment in April 2021, two members of the Cure53 team pointed out four items. With the exception of one *High*-scored flaw, the testers only noted *Medium* and lower risks on the noble-secp256k1's scope.

It should be specified that the noble crypto's low-level algorithms for elliptic curve operations - such as scalar multiplication across Weierstrass curves - are implemented correctly. Similarly, other low-level cryptographic and mathematical operations seem to be deployed properly, with attention to code readability and the functional programming of discrete mathematical functions. As such, the library survives scrutiny at the lowest level.

However, as is common with JavaScript-based low-level libraries, the API itself still introduces potential security issues that are due to human error. NBL-01-001, NBL-01-003 and NBL-01-004 all document how an overly permissive noble crypto API could allow for a variety of potentially greatly misleading cryptographic operations to take place. This could include generating public keys based on the clearly inadequate private key share values and the quiet substitution of public and private key values.

Finally, NBL-01-002 mentions the inclusion of some potentially non-constant time code in a core functionality for scalar multiplication. As noted in the noble crypto project's *README*, any claims made towards side channel resistance cannot be truly generalized into practical claims given the JavaScript stack's inability to reliably produce constant-time or side-channel-resistant code. Those caveats should be kept in mind when moving forward with the noble-secp256k1 project.

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