Fine penetration tests for fine websites

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

Audit-Report Paul Miller Noble Crypto Libraries 08.2024

Cure53, Dr.-Ing. M. Heiderich, Dr. D. Bleichenbacher, Dr. M. Conde, Dr. N. Kobeissi

Index

Introduction

This report presents the findings of a Q3 2024 security evaluation conducted on the noble-ciphers and noble-curves cryptographic libraries. Commissioned by Paul Miller, the library maintainer, in June 2024, the assessment was undertaken by Cure53 during July and early August 2024 (CW30 and CW31), with a total resource allocation of twenty days.

The scope of the engagement encompassed three Work Packages (WPs), focusing on white-box penetration testing and code auditing of the following features:

- **WP1**: Cryptography reviews & code audits against noble-ciphers implementation
- **WP2**: Cryptography reviews & code audits against noble-curves implementation
- **WP3**: Crypto. reviews & code audits against noble-curves bn254 implementation

Cure53 was granted unfettered access to the system, including source code, test-user credentials, and any additional resources necessary for a thorough evaluation. A dedicated team of four senior security specialists oversaw the project from start to finish, with preparatory activities commencing in July 2024 (CW32).

Effective communication was maintained through a dedicated and private Slack channel, facilitating collaboration among all relevant stakeholders. The well-defined project scope and open conversational platform ensured a seamless testing process. Cure53 provided regular status updates concerning the assessment's progress and findings, while live reporting was deemed a valuable tool and thus implemented using the aforementioned medium.

The assessment identified a total of eight findings, categorized as six security vulnerabilities and two general weaknesses with lower exploitation potential. While the number of findings is moderate, the identification of a single *High*-severity vulnerability (see NBL-04-004) highlights the need for immediate attention. This vulnerability pertains to a variable nonce size within AES-GCM, which could potentially lead to misuse.

While JavaScript offers limitations for implementing low-level cryptographic primitives, the cryptographic libraries provide a generally resilient security posture. The inspected libraries adhere to best practices and include diverse, high-quality test vectors, with only a select few areas that would benefit from further improvements. To further enhance the security of these cryptographic libraries, it is recommended to address the outlined flaws promptly.

Moving forward, the *Scope*, test setup, and available materials are enumerated below. Subsequently, all *Identified Vulnerabilities* and *Miscellaneous Issues* are provided in chronological order of detection alongside a technical overview, a Proof-of-Concept (PoC) if required, and high-level fix advice. Lastly, the *Conclusions* chapter elaborates on the general impressions gained for the in-scope features and verifies the perceived security posture.

Scope

- **Cryptography reviews & code audits against Paul Miller's noble-ciphers & noble-curves**
    - **WP1:** Cryptography reviews & code audits against noble-ciphers implementation
        - **Sources:**
            - https://github.com/paulmillr/noble-ciphers/releases/tag/0.6.0
        - **In scope:**
            - *all modules*
    - **WP2:** Cryptography reviews & code audits against noble-curves implementation
        - **Sources:**
            - https://github.com/paulmillr/noble-curves/releases/tag/1.5.0
        - **In scope:**
            - ed25519, ed448, their add-ons, bls12-381, bn254, hash-to-curve, low-level primitives "bls", "tower", "edwards", "montgomery", etc.
            - bn254 aka alt_bn128 was added in curves for EVM
    - **WP3:** Crypto. reviews & code audits against noble-curves bn254 implementation
        - See above
    - **Test-supporting material was shared with Cure53**
    - **All relevant sources were shared with Cure53**

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., *NBL-04-001*) to facilitate any future follow-up correspondence.

NBL-04-001 WP1: Timing side channels in AES implementation *(Medium)*

Cure53 observed that the AES implementation provided in *noble-ciphers* is vulnerable to cache-timing attacks, since it relies on lookup tables (S-boxes, T-boxes) for the AES algorithm, which can leak timing information based on cache access patterns. An attacker can exploit these timing variations to extract the secret AES key. Table lookups are employed with input-dependent indices, which can result in variable execution times based on whether the required data is in the cache.[1]

An attacker can exploit these vulnerabilities by:

- Sending carefully crafted input to the AES encryption function.
- Measuring the time taken for encryption operations.
- Using statistical analysis to correlate timing information with potential key values.
- Gradually narrowing down the possible key space and eventually recovering the full AES key.

The potential impact of this vulnerability is severe and could entail complete key recovery. However, fully neutralizing timing side channels in the specific context of a JavaScript runtime (as *noble-ciphers* is constrained to), is extremely difficult. Albeit, BearSSL's S-box implementation, written by Thomas Pornin, leverages a technique described by Boyar and Peralta[2] to translate the S-box into a circuit. The implementation can be reviewed and potentially adapted into *noble-ciphers*:

BearSSL's S-box implementation:
https://www.bearssl.org/gitweb/?p=BearSSL;a=blob;f=src/symcipher/aes_ct.c;h=66776d9e206c92cbbf3fa799c651a3d3652bd75a;hb=HEAD#l27

To mitigate this vulnerability, Cure53 suggests exploring the aforementioned approach for implementation in *noble-ciphers*. If the performance overhead of BearSSL's implementation strategy proves excessive, the constant-time implementation may be offered as an optional variant by the *noble-ciphers* library.

[1] https://cr.yp.to/antiforgery/cachetiming-20050414.pdf
[2] https://eprint.iacr.org/2009/191.pdf

NBL-04-002 WP1: AES error messages may leak information *(Low)*

The *noble-ciphers* library contains multiple instances whereby error messages may provide detailed information regarding the nature of failures during cryptographic operations, in the event that these error messages are allowed to surface to the application layer. While error messages of this nature are helpful for debugging, they can potentially leak sensitive information concerning the structure and content of encrypted data, aiding potential attackers in their efforts to break the encryption.

The error messages reveal specific insights, such as the data length, padding structure, and nature of integrity check failures. This information could potentially be exploited by an attacker to infer details about the plaintext or the encryption process, with the intention of performing malicious activities that could involve the following:

- **Padding oracle attacks:** In the PKCS padding validation, the different error messages for invalid padding byte and incorrect padding could allow an attacker to distinguish between these cases, potentially leading to padding oracle attacks.
- **Length-based attacks:** Error messages revealing the exact length requirements of plaintexts or ciphertexts could assist an attacker in collecting insights into the encrypted data or process.

Affected file:
noble-ciphers/src/aes.ts

Affected code (non-exhaustive):
```
function validatePCKS(data: Uint8Array, pcks5: boolean) {
  // ...
  if (lastByte <= 0 || lastByte > 16) throw new Error(`aes/pcks5: wrong 
padding byte: ${lastByte}`);
  // ...
  if (data[len - i - 1] !== lastByte) throw new Error(`aes/pcks5: wrong 
padding`);
  // ...
}

[...]

export const aeskw = wrapCipher(
  { blockSize: 8 },
  (kek: Uint8Array): Cipher => ({
    encrypt(plaintext: Uint8Array) {
      // ...
      if (!plaintext.length || plaintext.length % 8 !== 0)
        throw new Error('plaintext length must be non-empty and a multiple 
of 8 bytes');
      if (plaintext.length === 8)
```

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

```
        throw new Error('8-byte keys not allowed in AESKW, use AESKWP
instead');
      // ...
    },
    decrypt(ciphertext: Uint8Array) {
      // ...
      if (ciphertext.length % 8 !== 0 || ciphertext.length < 3 * 8)
        throw new Error('ciphertext must be at least 24 bytes and a
multiple of 8 bytes');
      // ...
      if (!equalBytes(out.subarray(0, 8), AESKW_IV)) throw new
Error('integrity check failed');
      // ...
    },
  })
);

[...]

export const aeskwp = wrapCipher(
  { blockSize: 8 },
  (kek: Uint8Array): Cipher => ({
    encrypt(plaintext: Uint8Array) {
      // ...
      if (!plaintext.length) throw new Error('plaintext length must be non-
empty');
      // ...
    },
    decrypt(ciphertext: Uint8Array) {
      // ...
      if (ciphertext.length < 16)
        throw new Error('ciphertext must be at least 16 bytes and a
multiple of 8 bytes');
      // ...
      if (o32[0] !== AESKWP_IV || out.length - 8 !== padded)
        throw new Error('integrity check failed');
      for (let i = len; i < padded; i++)
        if (out[8 + i] !== 0) throw new Error('integrity check failed');
      // ...
    },
  })
);
```

To mitigate these issues, Cure53 advises replacing specific error messages with generic information in order to obscure the nature of the failure. For example, a generic message such as *Decryption failed* could be presented for all types of decryption errors. In addition, the dev team should ensure that all cryptographic operations, including error checking, are implemented in constant time to prevent timing attacks based on different error conditions.

NBL-04-004 WP1: Variable nonce size in AES-GCM could lead to misuse *(High)*

Cure53 identified that the implementation of AES-GCM in the *noble-ciphers* library allows for variable nonce sizes. While this flexibility is permitted by the GCM specification, it introduces potential security risks if handled suboptimally at the application level or by users of the library.

Specifically, the implementation allows for nonces of any length greater than 0 bytes, with a comment indicating that *smaller nonces [are] less secure*. This flexibility, while adhering to the GCM specification, can lead to several security issues including (but not limited to):

- **Nonce reuse:** Shorter nonces increase the probability of nonce reuse, which is catastrophic for GCM's security.
- **Reduced security margin:** Nonces smaller than the recommended 12 bytes provide a reduced security margin.
- **Inconsistent behavior:** The code handles 12-byte nonces varyingly to other lengths, which might lead to unexpected behaviors.

Affected file:
noble-ciphers/src/aes.ts

Affected code:
```
export const gcm = wrapCipher(
  { blockSize: 16, nonceLength: 12, tagLength: 16 },
  function gcm(key: Uint8Array, nonce: Uint8Array, AAD?: Uint8Array):
Cipher {
    abytes(key);
    abytes(nonce);
    if (AAD !== undefined) abytes(AAD);
    // Nonce can be pretty much anything (even 1 byte). But smaller nonces
less secure.
    if (nonce.length === 0) throw new Error('aes/gcm: empty nonce');
    const tagLength = 16;
    function _computeTag(authKey: Uint8Array, tagMask: Uint8Array, data:
Uint8Array) {
      const tag = computeTag(ghash, false, authKey, data, AAD);
      for (let i = 0; i < tagMask.length; i++) tag[i] ^= tagMask[i];
      return tag;
    }
    function deriveKeys() {
      const xk = expandKeyLE(key);
      const authKey = EMPTY_BLOCK.slice();
      const counter = EMPTY_BLOCK.slice();
      ctr32(xk, false, counter, counter, authKey);
      if (nonce.length === 12) {
        counter.set(nonce);
      } else {
        // Spec (NIST 800-38d) supports variable size nonce.
        // Not supported for now, but can be useful.
        const nonceLen = EMPTY_BLOCK.slice();
        const view = createView(nonceLen);
        setBigUint64(view, 8, BigInt(nonce.length * 8), false);
        // ghash(nonce || u64be(0) || u64be(nonceLen*8))
        const g = ghash.create(authKey).update(nonce).update(nonceLen);
        g.digestInto(counter); // digestInto doesn't trigger '.destroy'
        g.destroy();
      }
      const tagMask = ctr32(xk, false, counter, EMPTY_BLOCK);
      return { xk, authKey, counter, tagMask };
    }
```

To mitigate this issue, Cure53 suggests integrating a number of solutions. Firstly, one should modify the *gcm* function to enforce a 12-byte nonce size, similar to the SIV implementation in the same library. This aligns with NIST recommendations and reduces the risk of misuse.

Secondly, if variable nonce sizes must be supported, clear documentation should be provided on the security implications of different nonce sizes, while 12-byte nonces are strongly recommended.

Next, one should implement a secure nonce generation function that always produces 12-byte nonces and encourage its use in the library documentation.

Lastly, if variable nonce sizes are to be supported, the developer team should integrate a warning system that alerts users when utilizing non-standard nonce sizes.

NBL-04-005 WP2: Absent length check in *expand_message_xmd* (*Low*)

While reviewing the implementation of several algorithms for encoding or hashing arbitrary strings to points on elliptic curves according to RFC 9380[3], Cure53 found that the *expand_message_xmd* function is not fully compliant with the specification.

In particular, the RFC indicates that the function should return with an error if the length of the requested output (namely *lenInBytes*) is greater than 65535, which is absent from the function as highlighted in the following code excerpt.

Affected file:
noble-curves-1.5.0/src/abstract/hash-to-curve.ts

[3] https://www.rfc-editor.org/rfc/rfc9380.pdf

Affected code:
```
export function expand_message_xmd(
  msg: Uint8Array,
  DST: Uint8Array,
  lenInBytes: number,
  H: CHash
): Uint8Array {
  abytes(msg);
  abytes(DST);
  anum(lenInBytes);
  // https://www.rfc-editor.org/rfc/rfc9380#section-5.3.3
  if (DST.length > 255) DST = H(concatBytes(utf8ToBytes('H2C-OVERSIZE-
DST-'), DST));
  const { outputLen: b_in_bytes, blockLen: r_in_bytes } = H;
  const ell = Math.ceil(lenInBytes / b_in_bytes);
  if (ell > 255) throw new Error('Invalid xmd length');
  const DST_prime = concatBytes(DST, i2osp(DST.length, 1));
  const Z_pad = i2osp(0, r_in_bytes);
  const l_i_b_str = i2osp(lenInBytes, 2); // len_in_bytes_str
  const b = new Array<Uint8Array>(ell);
  const b_0 = H(concatBytes(Z_pad, msg, l_i_b_str, i2osp(0, 1),
DST_prime));
  b[0] = H(concatBytes(b_0, i2osp(1, 1), DST_prime));
  for (let i = 1; i <= ell; i++) {
    const args = [strxor(b_0, b[i - 1]), i2osp(i + 1, 1), DST_prime];
    b[i] = H(concatBytes(...args));
  }
  const pseudo_random_bytes = concatBytes(...b);
  return pseudo_random_bytes.slice(0, lenInBytes);
}
```

To mitigate this issue, Cure53 suggests including a condition to verify that the length of the requested output (i.e., *lenInBytes*) is less than 65535 and return with an error if this is not the case, as already actioned in the *expand_message_xor* function.

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

NBL-04-007 WP2: Timing leak from BigInt operations *(Low)*

While reviewing the elliptic curve code for timing side channels, Cure53 acknowledged that smaller timing channels still remain, despite the incorporation of effective countermeasures. The library generally ensures that critical primitives use a constant number of arithmetic operations. However, the arithmetic is implemented using Javascript's BigInt type. The runtime of operations with this data type depends on the bit-length of its values.

For example, if the timing leaks depend on the bits of a private key, then the timing can ultimately leak it. Hence, one must prevent timing leaks that offer information regarding the internal state during scalar multiplications.

Affected file:
noble-curves-1.5.0/src/abstract/weierstrass.ts

Affected code (example):
```
// Renes-Costello-Batina exception-free doubling formula.
// There is 30% faster Jacobian formula, but it is not complete.
// https://eprint.iacr.org/2015/1060, algorithm 3
// Cost: 8M + 3S + 3*a + 2*b3 + 15add.
double() {
const { a, b } = CURVE;
const b3 = Fp.mul(b, _3n);
const { px: X1, py: Y1, pz: Z1 } = this;
let X3 = Fp.ZERO, Y3 = Fp.ZERO, Z3 = Fp.ZERO; // prettier-ignore
let t0 = Fp.mul(X1, X1); // step 1
let t1 = Fp.mul(Y1, Y1);
let t2 = Fp.mul(Z1, Z1);
let t3 = Fp.mul(X1, Y1);
t3 = Fp.add(t3, t3); // step 5
Z3 = Fp.mul(X1, Z1);
Z3 = Fp.add(Z3, Z3);
X3 = Fp.mul(a, Z3);
Y3 = Fp.mul(b3, t2);
Y3 = Fp.add(X3, Y3); // step 10
X3 = Fp.sub(t1, Y3);
Y3 = Fp.add(t1, Y3);
Y3 = Fp.mul(X3, Y3);
X3 = Fp.mul(t3, X3);
Z3 = Fp.mul(b3, Z3); // step 15
t2 = Fp.mul(a, t2);
t3 = Fp.sub(t0, t2);
t3 = Fp.mul(a, t3);
t3 = Fp.add(t3, Z3);
Z3 = Fp.add(t0, t0); // step 20
t0 = Fp.add(Z3, t0);
t0 = Fp.add(t0, t2);
```

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

```
t0 = Fp.mul(t0, t3);
Y3 = Fp.add(Y3, t0);
t2 = Fp.mul(Y1, Z1); // step 25
t2 = Fp.add(t2, t2);
t0 = Fp.mul(t2, t3);
X3 = Fp.sub(X3, t0);
Z3 = Fp.mul(t2, t1);
Z3 = Fp.add(Z3, Z3); // step 30
Z3 = Fp.add(Z3, Z3);
return new Point(X3, Y3, Z3);
}
```

A timing leak can occur whenever one of the variables in the code above is small. An attacker that can select the base point of a scalar multiplication can search for extreme values in a way that intermediate values become small positive integers or even 0.

To mitigate this issue, Cure53 recommends introducing additional countermeasures against timing leaks, which can be evaluated by their effectiveness and implementation complexity. Discussions with the author verified that the library is not intended for situations such as smart cards, whereby highly precise measurements are feasible and hence rewriting BigInt arithmetic in a constant time manner is not planned. Rather than investing extensive efforts into enforcing constant time for all operations, it is also possible to randomize the computation to ensure that timing differences do not correlate with keys and confidential data.

In this regard, one rather simple countermeasure that has been proposed is to randomize projective coordinates. For instance, when an affine point (x,y) is converted into its projective equivalent (x,y,1), it is possible to use the coordinates (x*r, y*r, r) with a random non-zero value r instead.[4] Similarly, Jacobian points (x, y, z) can be randomized as (x*(r^2), y*(r^3), z*r). While this is relatively simple to implement and significantly minimizes potential timing leaks, some cases are still not covered by this method. Akishita and Takagi detected an attack that exploits the fact that the 0 coordinate is not randomized in the proposal above.[5] The strategy requires locating points leading to a computation with the 0 value and the attacker selecting them, which is achievable by selecting a fake public key in an ECDH exchange.

[4] http://www.crypto-uni.lu/jscoron/publications/dpaecc.pdf
[5] http://download.mmag.hrz.tu-darmstadt.de/media/FB20/Dekanat/Publikationen/CDC/TI-03-01.zvp.pdf

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

NBL-04-008 WP2: *findGroupHash* function not constant-time *(Low)*

While reviewing the implementation of the Jubjub curve in the *jubjub.ts* file, Cure53 located a function (*findGroupHash*) that constructs a tag from an input *message* and an input *personalization* byte array. Subsequently, the *findGroupHash* function calls the function *groupHash* (which succeeds if a point of large order is found after certain operations are performed) for at most 256 iterations. If a point of large order is not found, an error is thrown.

However, as soon as a point with the intended properties is found, the loop will be exited. This can result in timing differences that can eventually be exploited by an attacker.

Affected file:
noble-curves-1.5.0/src/jubjub.ts

Affected code:
```
export function findGroupHash(m: Uint8Array, personalization: Uint8Array) {
  const tag = concatBytes(m, new Uint8Array([0]));
  for (let i = 0; i < 256; i++) {
    tag[tag.length - 1] = i;
    try {
      return groupHash(tag, personalization);
    } catch (e) {}
  }
  throw new Error('findGroupHash tag overflow');
}

export function groupHash(tag: Uint8Array, personalization: Uint8Array) {
  const h = blake2s.create({ personalization, dkLen: 32 });
  h.update(GH_FIRST_BLOCK);
  h.update(tag);
  // NOTE: returns ExtendedPoint, in case it will be multiplied later
  let p = jubjub.ExtendedPoint.fromHex(h.digest());
  // NOTE: cannot replace with isSmallOrder, returns Point*8
  p = p.multiply(jubjub.CURVE.h);
  if (p.equals(jubjub.ExtendedPoint.ZERO)) throw new Error('Point has small
order');
  return p;
}
```

To mitigate this issue, Cure53 advises implementing the function in constant time by enforcing that the loop runs a fixed number of occasions, irrespective of whether a large order value is identified.

Miscellaneous Issues

This section covers any and all noteworthy findings that did not incur an exploit but 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.

NBL-04-003 WP1: *Clean* function memory effects not guaranteed (*Info*)

Cure53 noted that the *clean* function in the symmetric cipher library *noble-ciphers*, which purports to erase sensitive data from memory, may not reliably achieve this activity. The current implementation uses the *fill* method to overwrite array contents with zeroes, which is not guaranteed to securely erase data from memory.

Affected file:
noble-ciphers/src/utils.ts

Affected code:
```
export function clean(...arrays: TypedArray[]) {
  for (let i = 0; i < arrays.length; i++) {
    arrays[i].fill(0);
  }
}
```

While this function attempts to clear the contents of the provided TypedArrays, a number of shortcomings are presented:

- **Optimizations:** JavaScript engines may optimize out operations that appear to offer no visible effect, potentially avoiding the fill operation entirely.
- **Garbage collection:** Even if the array is zeroed out, the original data may persist in memory until garbage collection occurs, which is non-deterministic.
- **Memory reallocation:** The underlying ArrayBuffer may be reallocated by the JavaScript runtime, retaining copies of the sensitive data in previously used memory locations.
- **Lack of memory barriers:** There are no guarantees regarding when the zeroing operation will be executed relative to other memory operations.
- **Compiler optimizations:** In optimized builds, compilers might eliminate code that writes to memory locations that are not subsequently read.

The failure to securely erase sensitive cryptographic material from memory can lead to data exposure. An adversary with access to the system's memory (through various attack vectors such as cold boot attacks, malware, or core dumps) could potentially recover cryptographic keys or other sensitive information, compromising the security of the entire cryptographic system.

However, given that *noble-cipher*'s stated purpose is to implement cryptographic primitives in TypeScript or JavaScript, and considering that neither offer any known solution to the problem of secure memory erasure, Cure53 unfortunately cannot currently offer remedial guidance for this issue within the context of *noble-ciphers*. As such, the ticket has been included here for completist reasons only.

NBL-04-006 WP2: Defensive coding enhancements *(Info)*

While reviewing the *noble-curves* repository, Cure53 observed that a code-defensive style is followed throughout the codebase, which effectively avoids weaknesses related to malformed or malicious inputs.

The testing team noted the *i2osp* function while reviewing the hash-to-curve algorithms, which converts an integer value given as an input into an unsigned integer string representation of a length also given as an input. Although the function correctly checks that the value is not negative and verifies that it can be represented as a string of the indicated length, the implementation neglects to enforce that the input value is a safe integer.

Affected file:
noble-curves-1.5.0/src/abstract/hash-to-curve.ts

Affected code:
```
function i2osp(value: number, length: number): Uint8Array {
  if (value < 0 || value >= 1 << (8 * length)) {
    throw new Error(`bad I2OSP call: value=${value} length=${length}`);
  }
  const res = Array.from({ length }).fill(0) as number[];
  for (let i = length - 1; i >= 0; i--) {
    res[i] = value & 0xff;
    value >>>= 8;
  }
  return new Uint8Array(res);
}
```

Notably, a function in the same file (namely *anum*) explicitly performs the aforementioned action and is used as a consistency check elsewhere. This could be incorporated into the function in question for improved resilience and consistency with the remaining code. Accordingly, this ticket has been compiled for informational purposes only.

Conclusions

Cure53 was tasked with evaluating two distinct cryptographic libraries for this summer 2024 project, *noble-ciphers* and *noble-curves*, in order to identify any implementation vulnerabilities and errors in the symmetric and asymmetric primitives.

In context, four experienced senior testers handled the various stages of the project, initiating line-by-line code analysis and automated testing to ascertain the wider security posture and any plausible exploitation pathways.

Firstly, Cure53 would like to comment on the scope's positive attributes, including strengths and best practices:

- Defensive coding techniques are employed throughout the codebase, including input validation and point validation in elliptic curve operations.
- Extensive unit testing provides ample test coverage.
- Optimizations are implemented without compromising robustness, in most cases.
- The use of BigInt data types mitigates certain common JavaScript flaws.
- Critical methods are mostly written to utilize a constant number of arithmetic operations, reducing potential timing leaks.
- Cryptographically secure pseudo-random number generators (CSPRNGs) are used when randomness is required.
- The implementation of elliptic curves and hash-to-curve algorithms generally adheres closely to specifications (e.g., RFC 9380).

Next, Cure53 would like to outline the most pertinent areas of concern observed during the investigative procedures:

Timing-related issues:

- Firstly, the AES implementation is vulnerable to cache-timing attacks due to the use of lookup tables with input-dependent indices (see NBL-04-001).
- Moreover, a non-constant-time function was identified in the Jubjub curve implementation (see NBL-04-008).
- Lastly, the use of BigInt for underlying arithmetic operations introduces a smaller timing side channel (see NBL-04-007).

Error handling and information leakage:

- Here, some error messages may inadvertently leak sensitive information regarding cryptographic operations (see NBL-04-002).
- In addition, verbose error messages, while useful for debugging, could potentially aid attackers if exposed.

Memory management:

- The *clean* function serving to erase sensitive data from memory may not reliably accomplish its purpose due to JavaScript runtime limitations (see NBL-04-003).

Implementation specifics:

- In this respect, the AES-GCM implementation allows for variable nonce sizes, which could facilitate security risks if improperly managed (see NBL-04-004).
- Elsewhere, a minor deviation from specifications was detected regarding the validation of input parameters in the implementation of RFC 9380 (see NBL-04-005).
- Finally, a minor recommendation to bolster the resilience of a particular function in one of the hash-to-curve algorithms has been provided for informative purposes (see NBL-04-006).

In light of the various subpar circumstances outlined above, Cure53 would like to offer a number of recommended hardening measures that would benefit the in-scope constructs if installed. Please note that the advice below is not presented in order of priority:

- Firstly, Cure53 suggests investigating alternative AES implementations, such as BearSSL's circuit-based S-box approach, to mitigate timing attacks.
- The dev team could also replace specific error messages with generic counterparts to prevent information leakage.
- A standard 12-byte nonce size for AES-GCM could be enforced. Alternatively, clear documentation could be provided on the security implications of different nonce sizes.
- The non-constant-time function in the Jubjub curve implementation should be addressed.
- Moreover, randomization techniques to mitigate timing channels in BigInt operations should be considered.
- As previously mentioned, the resilience of a particular function in one of the hash-to-curve algorithms should be enhanced.
- Finally, one could ensure complete adherence to specifications, particularly regarding input parameter validation for hash-to-curve algorithms.

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

To conclude, Cure53 is pleased to confirm that the overall security posture of the libraries is robust, with most pitfalls merely incurring low-risk impact if successfully exploited by a threat actor. However, the limitations of implementing low-level cryptographic primitives in JavaScript are evident and should be carefully considered. Despite these challenges, the libraries generally adhere to best practices and include diverse, high-quality test vectors.

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