Fine penetration tests for fine websites

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

Audits-Report TypeScript Hashing Libraries 12.2021

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

Index

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

Introduction

"This package contains all pure-js cryptographic primitives normally used when developing Javascript / TypeScript applications and tools for Ethereum."

From https://github.com/ethereum/js-ethereum-cryptography

This report describes the results of a security assessment of the newly written TypeScript (TS) hashing library, also referred to as Paul Miller's cryptographic library complex. Carried out by Cure53 in late December 2021, the project included a comprehensive review of the cryptographic premise and a dedicated audit of the source code. The project targeted not only the library itself but also several related components and dependencies, also written in TS.

Registered as *NBL-02,* the project was requested by Paul Miller, who is the maintainer of the software, back in early September 2021. It was then scheduled for the final weeks of 2021 to allow ample time for preparations on both sides.

Cure53 completed the assessment in CW50 and CW51 of 2021. A total of seven days were invested to reach the coverage expected for this assignment, whereas a team of three senior testers has been composed and tasked with this project's preparation, execution and finalization.

For optimal structuring and tracking of tasks, the work was contained into a single work package (WP):

- **WP1**: Cryptography Reviews & Audits against newly written TypeScript Hashing Libraries

White-box methodology was utilized as the preferred manner of assessment for items available publicly on GitHub as open source software. Beyond the available OSS source code, Cure53 was additionally given access to documentation and supplementary material that facilitated testing.

The project progressed effectively on the whole. All preparations were done in CW49 to foster a smooth transition into the testing phase. Over the course of the engagement, the communications were done using a private, dedicated and shared Slack channel, set up to connect the respective workspaces of Cure53 and the library maintainer. The discussions throughout the test were very good and productive. Quite a lot of questions were raised and addressed during different stages of the project.

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

The scope was generally well-prepared and clear, with a minor exception of the initial lack of clarity about which exact items would constitute the scope. Once this was resolved, the project went well and no noteworthy roadblocks were encountered during the test. Cure53 offered frequent status updates about the test and the emerging findings. Live-reporting and extensive discussions were held on Slack on a regular basis. The involved personnel discussed the software, project expectations, assumed threats and findings. Thanks to live-reporting, some issues could be fixed and verified as correctly tackled when the tests were still ongoing.

The Cure53 team managed to get very good coverage over the WP1 scope items. Among thirteen security-relevant discoveries, six were classified to be security vulnerabilities and seven to be general weaknesses with lower exploitation potential. It needs to be noted that two findings were given elevated severity scores, with one determined to be a *High*-level risk (see NBL-02-001) and one even seen as a *Critical* bug (see NBL-02-010). The remaining findings reside in the realm of discoveries without that significant threat potential.

What is more, three of the miscellaneous issues were later identified as false alerts. A note has been added to each affected ticket to communicate this. While the number of findings is not overly high, it needs to be analyzed in the specific context of the library. Thus, given the purpose of the software, it is recommended to address all findings quickly and prioritize the ones with elevated risk factors.

In the following sections, the report will first offer a table summarizing the vulnerabilities opposite the threat model. Cure53 then sheds light on the scope and key test parameters, as well as the structure and content of the WP.

Next, all findings will be discussed in grouped vulnerability and miscellaneous categories, then following a chronological order in each group. Alongside technical descriptions, PoC and mitigation advice are supplied when applicable. Finally, the report will close with broader conclusions about this December 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 TypeScript hashing library and related components are also incorporated into the final section.

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

Vulnerability Summary

Project Name	Number of Findings
js-ethereum-cryptography	1 Critical, 1 High, 3 Medium, 2 Low
noble-hashes	0 Vulnerabilities found
micro-base	1 Medium
micro-bip32	1 Critical, 1 High, 1 Medium
micro-bip39	0 Vulnerabilities found

Table: Overview of findings with the given scope and threat model

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

Scope

- **Reviews & Code Audits against several TypeScript Crypto libraries & addons**
  - The following libraries and sources were examined:
    - *ethereum-cryptography*
      - https://github.com/ethereum/js-ethereum-cryptography/tree/v0.2.1
      - https://github.com/ethereum/js-ethereum-cryptography/pull/15
      - Dependents were in scope as well,
      - *noble-secp* was already audited, hence it was out-of-scope
    - *noble-hashes*
      - https://github.com/paulmillr/noble-hashes
      - Out-of-scope:
        - *blake3.ts*
        - *sha3-addons.ts*
    - *micro-base*
      - https://github.com/paulmillr/micro-base
    - *micro-bip32* & *micro-bip39*
      - https://github.com/paulmillr/micro-bip32
      - https://github.com/paulmillr/micro-bip39
      - Note that the relevant code from *ethereum-cryptography* was copied into *micro-bip32* & *micro-bip39* projects for convenience
  - Reviews & audits focused on the following areas, as requested by the maintainer:
    - Possible timing attacks targeting algorithmic resistance against them
    - Functional correctness of elliptic curve operations in use
    - Safety checks for the known side-channels
    - Checks against curve validation errors & elliptic-curve-specific attacks
    - General checks against constant-time operations
    - Misuse prevention related to the high-level cryptographic API
  - **Test supporting material was shared with Cure53**
  - **All relevant sources were made available to Cure53**

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-02-001*) for the purpose of facilitating any future follow-up correspondence.

NBL-02-001: BIP32 HD key from seed not compliant with standards *(High)*

***Note**: This issue was addressed by the maintainers and Cure53 was able to verify the fix by inspecting a diff. The issue no longer exists.*

While reviewing the *js-ethereum-cryptography* repository, it was noticed that the library contains an implementation for HD key generation. For that purpose, the library creates a new HD key from a master seed. The master key derivation from a master seed is specified in the GitHub repository of Bitcoin[1] and states that the seed's length should be between 128 bit and 512 bit, with 256 advised. The current implementation lacks a check that ensures the seed meets the official specification.

Affected file:
js-ethereum-cryptography/src/hdkey.ts

Affected code:

```
public static fromMasterSeed(seed: Uint8Array, versions?: Versions): HDKey {
       const I = hmac(sha512, MASTER_SECRET, seed);
       const hdkey = new HDKey(versions);
       hdkey.chainCode = I.slice(32);
       hdkey.privateKey = I.slice(0, 32);
       return hdkey;
}
```

It is recommended to comply with the standards proposed in the official repository. This will help prevent generation of weak keys.

[1] https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki#master-key-generation

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

NBL-02-002: Private key allows zero addition *(Low)*

***Note**: It was confirmed that the issue was addressed successfully by PR18[2].*

While reviewing the *js-ethereum-cryptography* repository, it was noticed that the file *secp256k1-compat.ts* implements a function for adding two points on an elliptic curve. It is supposed to return the sum of these two points, however, the implementation does not check whether the tweak point added to the private key is an all-zero vector. In case that an all-zero value is provided as a tweak, the function returns the private key from the input.

Affected file:
js-ethereum-cryptography/src/secp256k1-compat.ts

Affected code:

```
export function privateKeyTweakAdd(
	privateKey: Uint8Array,
	tweak: Uint8Array
): Uint8Array {
	assertBytes(privateKey, 32);
	assertBytes(tweak, 32);
	let bn = bytesToNumber(tweak);
	if (bn >= ORDER) {
		throw new Error("Tweak bigger than curve order");
	}
	bn += bytesToNumber(privateKey);
	if (bn >= ORDER) {
		bn -= ORDER;
	}
	if (bn === 0n) {
		throw new Error(
		"The tweak was out of range or the resulted private key is
		invalid"
		);
	}
	privateKey.set(hexToBytes(numberToHex(bn)));
	return privateKey;
}
```

It is recommended to check the tweak parameter for the all-zero vector.

[2] https://github.com/ethereum/js-ethereum-cryptography/pull/18

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

NBL-02-003: Private key multiplication allows identity multiplication *(Low)*

***Note**: It was confirmed that the issue was addressed successfully by PR21[3].*

While reviewing the *js-ethereum-cryptography* repository, it was noticed that the file *secp256k1-compat.ts* implements a function for multiplying two points on an elliptic curve. The function operates on byte arrays which are converted into big numbers. However, the implementation does not check whether the provided tweak for multiplication is equal to the *identity* element. Multiplying the private key by the *identity* results in the private key means that the private key is not getting changed.

Affected file:
js-ethereum-cryptography/src/secp256k1-compat.ts

Affected code:

```
export function privateKeyTweakMul(
	privateKey: Uint8Array,
	tweak: Uint8Array
): Uint8Array {
	assertBytes(privateKey, 32);
	assertBytes(tweak, 32);
	let bn = bytesToNumber(tweak);
	if (bn >= ORDER) {
		throw new Error("Tweak bigger than curve order");
	}
	bn = mod(bn * bytesToNumber(privateKey), ORDER);
	if (bn >= ORDER) {
		bn -= ORDER;
	}
	if (bn === 0n) {
		throw new Error(
		"The tweak was out of range or the resulted private key is
		invalid"
		);
	}
	privateKey.set(hexToBytes(numberToHex(bn)));
	return privateKey;
}
```

It is recommended to check the *tweak* parameter for *identity* to avoid accidentally retrieving the same private key as the one provided as input. This could result in a compromise of the origin key and have security-relevant consequences.

[3] https://github.com/ethereum/js-ethereum-cryptography/pull/21

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

NBL-02-009: BIP32 HD key code fails to reject invalid keys *(Medium)*

***Note**: This issue was addressed by the maintainers and Cure53 was able to verify the fix by inspecting a diff. The issue no longer exists.*

While reviewing the *js-ethereum-cryptography* repository, it was noticed that the BIP32 implementation fails to reject keys with invalid index or parent fingerprint. Specifically, some invalid test vectors from the official specification[4] are accepted by the library. The following test vectors are accepted instead of being rejected:

```
// zero depth with non-zero parent fingerprint
xprv9s2SPatNQ9Vc6GTbVMFPFo7jsaZySyzk7L8n2uqKXJen3KUmvQNTuLh3fhZMBoG3G4ZW1N2kZuHE
PY53qmbZzCHshoQnNf4GvELZfqTUrcv
xpub661no6RGEX3uJkY4bNnPcw4URcQTrSibUZ4NqJEw5eBkv7ovTwgiT91XX27VbEXGENhYRCf7hyEb
WrR3FewATdCEebj6znwMfQkhRYHRLpJ

// zero depth with non-zero index
xprv9s21ZrQH4r4TsiLvyLXqM9P7k1K3EYhA1kkD6xuquB5i39AU8KF42acDyL3qsDbU9NmZn6MsGSUY
ZEsuoePmjzsB3eFKSUEh3Gu1N3cqVUN
xpub661MyMwAuDcm6CRQ5N4qiHKrJ39Xe1R1NyfouMKTTWcguwVcfrZJaNvhpebzGerh7gucBvzEQWRu
gZDuDXjNDRmXzSZe4c7mnTK97pTvGS8
```

These keys are valid keys but their properties (*index* or *parent fingerprint*) do not match the *depth* property.

The result of accepting such invalid keys depends on how the BIP32 code is used by an application. None of these key properties are used in the *derive()* nor *deriveChild()* calls, so they cannot influence the generated child keys. It is nevertheless highly recommended to reject such keys, as they might lead to vulnerabilities where a key might be used as a key of a different type (i.e. a wallet key vs an account key). This depends on how the objects of the given library are used by the application.

Affected file:
js-ethereum-cryptography/src/hdkey.ts

Affected code:

```
public static fromExtendedKey(base58key: string, versions?: Versions): HDKey {
      // => version(4) || depth(1) || fingerprint(4) || index(4) || chain(32)
      || key(33)
      const hdkey = new HDKey(versions);
      const keyBuffer: Uint8Array = base58c.decode(base58key);
      const keyView = createView(keyBuffer);
      const version = keyView.getUint32(0, false);
      hdkey.depth = keyBuffer[4];
```

[4] https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki

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

```
        hdkey.parentFingerprint = keyView.getUint32(5, false);
        hdkey.index = keyView.getUint32(9, false);
        [...]
        return hdkey;
}
```

It is recommended to fix the logic in *HDKey.fromExtendedKey()* as well as to add a similar check to *deriveChild(),* so as to detect any changes in *index*, *parent fingerprint* and *depth*. They are all publicly accessible properties of the object and may have changed after loading of a key.

NBL-02-010: BIP32 HD key accepts invalid *index* for *deriveChild()* (*Critical*)

***Note**: This issue was addressed by the maintainers and Cure53 was able to verify the fix by inspecting a diff. The issue no longer exists.*

While reviewing the *js-ethereum-cryptography* repository, it was noticed that the BIP32 implementation allows the *index* parameter of the *deriveChild()* function to be higher than what is allowed by the specification. The specification states that the valid range is *0* <= *index* < 2^{32}, while the code accepts values up to 2^{33}-1.

The value of *index* is used by this function as an unsigned 32-bit integer in the generation of the child-key. This results in a different than expected child-key being derived by the caller. For example, an *index* of 2^{33}-2 will yield the same (hardened) child-key as the (specification-compliant) *index* of 2^{32}-1. An attacker can abuse this to potentially steal keys from another chain.

Attack scenario:

Assumed here is a Bitcoin wallet with multiple accounts, each controlled by a different entity. The paths for the existing accounts are $m/0_H$ (*index* = 2^{31}) and $m/1_H$ (*index* = 2^{31}+1). An attacker who is able to influence the value of *index* during account key creation can choose the value of *index* larger than 2^{32}-1, such that it looks like an unused account, yet, in fact, results in a key for an existing account. Specifically, the attacker could choose the index 2^{32} + 2^{31} to get the same account key as the existing account $m/0_H$.

This attack works when a hardened child-private-key is generated because the code in *deriveChild()* will check that the *index* is larger than the minimum hardened key index. This is fatal because the private key gives the attacker full control over the account and means they can transfer funds out of it.

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

Affected file:
js-ethereum-cryptography/src/hdkey.ts

Affected code:

```
public deriveChild(index: number): HDKey {
	if (!Number.isSafeInteger(index) || index < 0 || index >= 2 ** 33) {
		throw new Error(`Child index should be positive 32-bit integer,
		not ${index}`);
	}
	if (!this.pubKey || !this.chainCode) {
		throw new Error("No publicKey or chainCode set");
	}
	if (!this.pubKey || !this.chainCode) {
		throw new Error("No publicKey or chainCode set");
	}
	let data = new Uint8Array(4);
	createView(data).setUint32(0, index, false);
	[...]
}
```

It is essential to fix this bounds-check to prevent faulty key derivation and potential loss of funds among users.

NBL-02-011: Insufficient array type checks in multiple repositories *(Medium)*

***Note**: This issue was addressed by the maintainers and Cure53 was able to verify the fix by inspecting a diff. The issue no longer exists.*

While reviewing the *micro-base* repository, it was noticed that some of the *encode()* and *decode()* functions do not fully validate the types of input array elements. As the compiled code will be JavaScript, a single array can hold multiple and different types. For example, the following is a valid array:

```
var arr = [42, 'foo', 12, 'bar];
```

The code in *micro-base* and other repositories misses this fact when checking the type of input arrays. This can be seen in the *decode()* function of *radix2()*:

```
if (!Array.isArray(digits) || (digits.length && typeof digits[0] !== 'number'))
	throw new Error('radix2.decode input should be array of strings');
```

This code verifies only the type of the first element in the array. All other elements can represent any other type. It was found that at least the *radix2.decode()* and *bech32.encode()* functions suffer from this flaw with regard to the *number* type. Additionally, the same flaw persists with arrays that are expected to contain strings. For

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

example, in *alphabet.encode()* and in the *js-ethereum-cryptography* repository, this is evident in the *getCoder()* call of the BIP39 implementation.

Affected files:

- *micro-base/index.ts*
- *js-ethereum-cryptography/src/bip39/index.ts*

It is recommended to revisit all array type checks in the codebase and modify them in order to validate the type of all array elements instead of just the first element.

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-02-004: Incomplete *destroy* implementation in Blake2b *(Low)*

***Note**: This issue was reviewed by the maintainer after the audit and classified as a false alert. Cure53 agrees with this assessment.*

While reviewing the *noble-hashes* repository, it was noticed that the hash classes implement a *destroy* function in order to wipe the internal state of the instance. The class within the *blake2b.ts* file implements the *destroy* function which wipes the *buffer32* member, but not the *buffer* member of the instance.

Affected file:
noble-hashes/src/blake2b.ts

Affected code:

```
destroy() {
	this.destroyed = true;
	this.buffer32.fill(0);
	this.set(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0);
}
```

It is recommended to wipe all buffers used during computation of the hash.

NBL-02-005: Incomplete *destroy* implementation in Blake2s *(Low)*

***Note**: This issue was reviewed by the maintainer after the audit and classified as a false alert. Cure53 agrees with this assessment.*

While reviewing the *noble-hashes* repository it was noticed that the hash classes implement a *destroy* function in order to wipe the internal state of the instance. The class within the *blake2s.ts* file implements the *destroy* function which wipes the *buffer32* member, but not the *buffer* member of the instance.

Affected file:
noble-hashes/src/blake2s.ts

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

Affected code:

```
destroy() {
        this.destroyed = true;
        this.buffer32.fill(0);
        this.set(0, 0, 0, 0, 0, 0, 0, 0);
}
```

It is recommended to wipe all buffers used during computation of the hash.

NBL-02-006: Missing *destroy* call for HMAC *(Low)*

***Note**: This issue was reviewed by the maintainer after the audit and classified as a false alert. Cure53 agrees with this assessment.*

While reviewing the *noble-hashes* repository, it was noticed that the HMAC implementation uses several instances of hashes during computation. In case that the key length exceeds the iPad's length, the HMAC implementation applies a hash function to the key before XORing it with the iPad. The implementation does not call the *destroy f*unction of this hash method.

Additionally, *hmac.ts* exports a one-shot function named *hmac()*, making it possible for the users to calculate the hash in one go:

```
export const hmac = (hash: CHash, key: Input, message: Input): Uint8Array
        => new HMAC<any>(hash, key).update(message).digest();
```

It was discovered that this method also fails to call *destroy()* on the HMAC object.

Affected file:
noble-hashes/src/hmac.ts

Affected code:

```
constructor(hash: CHash, _key: Input) {
        [...]
        const pad = new Uint8Array(blockLen);
        // blockLen can be bigger than outputLen
        pad.set(key.length > this.iHash.blockLen ?
        hash.create().update(key).digest() : key);
        [...]
}
```

It is recommended to invoke the *destroy* function on all hashes involved in the derivation of the HMAC. Furthermore, it may be useful to document this behavior for the users of the library within its official documentation.

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

NBL-02-007: Insufficient private key wipe for HD keys *(Low)*

***Note**: It was confirmed that the issue was addressed successfully by PR17[5].*

While reviewing the *js-ethereum-cryptography* repository, it was noticed that the HD key implementation allows setting public keys by invoking the respective *setter* function. The *setter* function correctly clears the private member *privKey* of the *HDKey* class, but it fails to wipe the data in the *privKeyBytes* member of the class.

Affected file:
js-ethereum-cryptography/src/hdkey.ts

Affected code:

```
set publicKey(value: Uint8Array | null) {
	let hex;
	try {
		hex = secp.Point.fromHex(value!);
	} catch (error) {
		throw new Error("Invalid public key");
	}
	this.pubKey = hex.toRawBytes(true); // force compressed point
	this.pubHash = hash160(this.pubKey);
	this.privKey = undefined;
}
```

It is recommended to wipe all private key material by invoking the *wipePrivateData* function of the *HDKey* class.

NBL-02-008: Recommendations on improving seed generation in ESKDF *(Info)*

While reviewing the *ESKDF* implementation in the *noble-hashes* repository, it was noticed that the main seed is generated by feeding the username and password into *pbkdf2*[6] and *scrypt*[7]. Each function consumes a slightly different input and the output of both is then XOR'ed and used as a seed.

Consultation with the authors revealed that the idea behind this is to make it harder for attackers during cracking, increasing the time needed for attacks. Additionally, it is intended as a safety net in case one of the algorithms is broken in the future. The current design, however, does not take into account that the operations for *scrypt* and *pbkdf2* can be paralleled and will, therefore, not cause much increase in the efforts needed from an attacker.

[5] https://github.com/ethereum/js-ethereum-cryptography/pull/17/commits
[6] https://www.ietf.org/rfc/rfc2898.txt
[7] https://datatracker.ietf.org/doc/html/rfc7914

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

Affected file:
noble-hashes/src/eskdf.ts

Affected code:

```
export function deriveMainSeed(username: string, password: string): Uint8Array {
	if (!strHasLength(username, 8, 255)) throw new Error('invalid username');
	if (!strHasLength(password, 8, 255)) throw new Error('invalid password');
	const scr = scrypt(password + '\u{1}', username + '\u{1}');
	const pbk = pbkdf2(password + '\u{2}', username + '\u{2}');
	const res = xor32(scr, pbk);
	scr.fill(0);
	pbk.fill(0);
	return res;
}
```

Secure design of cryptographic primitives is a complex task. It requires careful and thorough consideration of the threat model and use case. As time has shown, primitives that were considered secure, were discovered to be insecure many years later. We recommend performing a thorough cryptographic analysis of the ESKDF design. As the time frame of this audit did not permit going into full detail, this should be a dedicated task.

NBL-02-012: Sensitivity of chain code for BIP32 HD keys *(Medium)*

***Note**: It was confirmed that the issue was addressed successfully by PR17[8].*

While reviewing the *js-ethereum-cryptography* repository, it was noticed that the BIP32 standard uses so-called *chain codes* to derive new child-keys. The standard describes that the *chain code* of a child-key corresponds to 32-bytes from a HMAC-SHA512 computation over the child's *index* and the parent's public key (for non-hardened child items), where the *chain code* of the parent is entered as key. The other 32 bytes of this output get used as a *tweak* to derive the new child-key.

Specifically, the private key of a non-hardened child is computed as $key_{child}=key_{parent}$ + *tweak,* wherein keys refer to private keys. When the parent's public key and the *index* of the child become publicly known, an attacker who managed to get a hold of the *chain code* of the parent can reconstruct the *tweak*. In turn, via the *tweak,* they would be able to reconstruct the parent's private key via $key_{parent} = key_{child}$ - tweak[9]. The *hdkey* class stores the *chain code* as a public field, thereby it is accessible to a library user. What is

[8] https://github.com/ethereum/js-ethereum-cryptography/pull/17/commits
[9] https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki#implications

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

more, the documentation of the library does not emphasize sensitivity of the *chain code* for HD keys.

It is recommended to add a paragraph to the documentation of the library to emphasize that the *chain code* of HD keys needs to be protected from unauthorized access.

NBL-02-013: Type ambiguity leads to duplicate keys in ESKDF *(Medium)*

While reviewing the *noble-hashes* repository, it was noticed that the ESKDF implementation contains a function to derive new keys from a seed. The function *deriveChildKey* takes a seed, a protocol string, an account ID and a key length as arguments. The account ID parameter can be either of the type *string*, or of the type *number*. For *string* account IDs, the implementation converts the provided account ID into a byte-array, which is consequently used as the salt for the HKDF function.

Alternatively, when a numeric account ID is provided, the code puts the account ID into a byte-array of the length four, and again consequently uses it as the salt for the HKDF function. Therefore, in case the numeric byte-array representation coincides with the string byte array representation, the ESKDF function derives identical keys for the two distinct account IDs. This results in a security issue for a library caller in case there are both numeric and string IDs used as input to the *deriveChildKey* function.

Affected file:
noble-hashes/src/eskdf.ts

Affected code:

```
export function deriveChildKey(
	seed: Uint8Array,
	protocol: string,
	accountId: number | string = 0,
	keyLength = 32
): Uint8Array {
	[...]
	const allowsStr = PROTOCOLS_ALLOWING_STR.includes(protocol);
	let salt: Uint8Array; // Extract salt. Default is undefined.
	if (typeof accountId === 'string') {
		if (!allowsStr) throw new Error('accountId must be a number');
		if (!strHasLength(accountId, 1, 255)) throw new Error('accountId
		must be valid string');
		salt = toBytes(accountId);
	} else if (Number.isSafeInteger(accountId)) {
		if (accountId < 0 || accountId > 2 ** 32 - 1) throw new
		Error('invalid accountId');
		// Convert to Big Endian Uint32
		salt = new Uint8Array(4);
```

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

```
            createView(salt).setUint32(0, accountId, false);
        } else {
            throw new Error(`accountId must be a number${allowsStr ? ' or
            string' : ''}`);
        }
        const info = toBytes(protocol);
        return hkdf(sha256, seed, salt, info, keyLength);
}
```

It is recommended to only accept byte-arrays as account IDs for the *deriveChildKey* function in order to avoid ambiguity for the caller.

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

Conclusions

During this December 2021 assessment of the new hashing library written in TypeScript and maintained by Paul Miller, three members of the Cure53 team found evidence of both strengths and weaknesses in the complex. This is evident from the list of findings which, on the one hand, contains a manageable number of flaws largely characterized by low-risk levels and, on the other hand, includes two major problems in the form of *High* and *Critical* bugs.

To give some details, the assessment featured three repositories, namely *js-ethereum-cryptography, noble-hashes* and *micro-base*. The main focus was set on the possibilities to identify timing attacks, especially against noble's algorithms. Cure53 explicitly honed in on the functional correctness of the elliptic curve operations in use, the validation of elliptic curve specific validation errors and elliptic curve-specific attacks. The library's safety against other known side-channel approaches was also in focus. Furthermore, special attention was paid to the possible misuse of the high-level cryptographic API.

Cure53 was in constant communication with the customer and frequently sent status updates, alongside raising questions and concerns. The communication with the maintainer - executed via Slack - was excellent. Assistance was provided whenever requested.

At the meta-level, Cure53 can testify to the fact that the code of all three repositories is well-structured and easy to understand. This meant the auditors could become familiar with the codebase quite fast. It is evident that the developer is skilled in regard to implementing cryptographic primitives and secure coding principles.

The code is written in TypeScript and provides the developers with the ability to specify types for variables. This is, however, just a compile-time check, hence information is lost afterwards. The resulting plain JavaScript code does not enforce these types and permits invalid input. The code takes measures against this and contains type checks. In terms of arrays, these checks are not sufficient, as demonstrated in finding NBL-02-011.

In total, the assessment initially revealed six vulnerabilities and seven miscellaneous issues, albeit three items from the latter category were later noted as false alerts. Crucially, the most severe vulnerabilities have been identified in the *js-ethereum-cryptography* repository, especially in the *hdkey.ts* file. This file implements the BIP32 standard for crypto-wallets and the security issue is related to missing input data validation. In particular, the implementation accepts arbitrary seeds for the derivation of the master key, which is neither standards-compliant nor safe against brute-force.

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

Furthermore, *hdkey.ts* also enables deserialization of an invalid key at depth 0 and generation of two identical keys with different numeric indices due to a faulty input check.

The other vulnerabilities spotted during this project relate to missing checks for operations on elliptic curves which involve tweaks. The purpose of tweaks is to modify an existing point on an elliptic curve and derive new public or private keys from an existing key. As BIP32 is a critical part of the Bitcoin software and there have been severe issues with it in other implementations, special care has to be taken to highlight various invalid input instances. Cure53 reports that the BIP32 (HD key) implementation in *js-ethereum-cryptography* contained multiple issues, including one *Critical*-level risk which is now resolved.

The codebase across all three repositories contains a lot of tests for the individual implementations. These tests ensure that the implementations fulfill the specification in terms of the official test vectors.

The developers took care to have critical parts of the algorithms implemented as constant-time operations. With elliptic curve operations, JavaScript's *BigInt* types are used, which do not provide constant-time operations. However, JS code cannot be made constant-time, because of garbage collection and just-in-time compilation. Libraries are tested for algorithmic constant-timeness, which is the best they can aim at.

In sum, the outcome of this source code review demonstrates that the repositories are in a moderate state from a security perspective. Overall a good coverage over all repositories was achieved. Given the list of findings from this Cure53 December 2021 project, further work on validation logic would help to harden the existing security measures of the library.

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