Fine penetration tests for fine websites

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

Pentest-Report CasperDash Crypto-Wallet 09.-10.2022

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

Index

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

Introduction

"A non-custodial wallet built in Casper blockchain Store, send and receive tokens on Casper ecosystem with CasperDash. Also, CasperDash is available for storing all digital assets as NFTs and tokens."

From https://casperdash.io/

This report describes the results of a security assessment of the CasperDash complex, featuring the CasperDash *casper-storage* crypto wallet library written in TypeScript. Carried out by Cure53 in September 2022, the project included a cryptographic review and a dedicated audit of the source code.

To give some details, this project marks the first security-centered cooperation between Cure53 and CasperDash. Registered as *CAS-01,* the project was requested by CasperDash in September 2022 and then immediately scheduled for the following weeks of the same month.

As for the precise timeline and specific resources, Cure53 completed the examination in CW38 and CW39, as scheduled. A total of nine days were invested to reach the coverage expected for this assignment, whereas a team of four senior testers has been composed and tasked with this project's preparation, execution and finalization.

For optimal organization and tracking of tasks, the work was contained into one work package (WP):

- **WP1**: Crypto review and audit of CasperDash casper-storage Wallet TS library

Cure53 was provided with sources and all other means of access required to complete the tests. Additionally, relevant documentation was shared to make sure the project can be executed in line with the agreed-upon framework. Hence, it can be deduced that the methodology chosen here was a white-box approach.

The project progressed effectively on the whole. All preparations were done in September 2022, namely CW37, 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, which could be joined by all relevant personnel from both Cure53 and CasperDash.

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

The discussions throughout the test were very good and productive and not many questions had to be asked. Ongoing interactions positively contributed to the overall outcomes of this project. The scope was well-prepared and clear, greatly contributing to the fact that no noteworthy roadblocks were encountered during the test. Cure53 offered frequent status updates about the test and the emerging findings. Live-reporting was not used.

The Cure53 team managed to get very good coverage over the targets featured in the WP1 scope. Among thirteen security-relevant discoveries, seven were classified to be security vulnerabilities and six to be general weaknesses with lower exploitation potential.

It needs to be noted that the number of findings is rather high, especially given the size of the scope. On its own, it already points to certain problems within the overall security posture of the CasperDash crypto-wallet library. Furthermore, this conclusion is supported by the fact that Cure53 documented multiple issues with elevated severity scores. Besides *High*-ranking risks, one issue was deemed to be *Critical* for the integrity of the CasperDash service provisions. The problem needs to be treated with the utmost priority and be fixed swiftly, as it demonstrates a broken AES-CTR encryption (see CAS-01-005).

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. 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 pertinent to this September 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 CasperDash complex - specifically the CasperDash *casper-storage* crypto wallet library written in TypeScript - 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 of the following components and aspects:**
  - **WP1**: Crypto review & Audit of CasperDash *casper-storage* wallet TS library
    - **Sources:**
      - https://github.com/CasperDash/casper-storage.git
    - **Branch:**
      - *master*
    - **Commit:**
      - *bd00fe5fc222603914ff557941a81457e1b4eb3f*
    - **Documentation:**
      - https://casperdash.github.io/casper-storage/
  - **Test-supporting material was shared with Cure53**
    - https://casperdash.github.io/casper-storage/
  - **All relevant sources were shared with Cure53:**
    - https://github.com/CasperDash/casper-storage

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. *CAS-01-001*) for the purpose of facilitating any future follow-up correspondence.

CAS-01-001 WP1: Hardcoded salt in *CryptoUtils.scrypt* implementation *(High)*

During the review of the *CryptoUtils* class in the *casper-storage* repository, it was noticed that the method *CryptoUtils.scrypt* relied on a hardcoded *salt* value. This effectively defeats the purpose of the use of a salt and makes all (password) hashes, as long as they are generated using this method, vulnerable to attacks with rainbow tables.

Affected file:
casper-storage-master/src/cryptography/utils/crypto-utils.ts

Affected code:

```
export class CryptoUtils {
[...]
  static scrypt(input: string): Uint8Array {
    if (!input) {
      throw new Error("Input is required");
    }
    return scrypt(input, 'salt', { N: 2 ** 16, r: 8, p: 1, dkLen: 32 });
  }
}
```

It is recommended to generate a random salt value for every invocation of *CryptoUtils.scrypt*. This salt value must be returned to the caller together with the generated hash in order to allow verification.

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

CAS-01-002 WP1: AES implementation vulnerable to timing attacks *(Medium)*

While reviewing the dependencies of the *AESUtils* class in *casper-storage,* it was noticed that it used an AES implementation vulnerable to timing attacks. The *AESUtils* class wraps the AES implementation of the *aes-js*[1] library.

While looking at this library, it was noticed that lookup tables for the AES S-box operations were in use. Notably, the usage of lookup tables enables cache-timing-attacks due to the fact that each lookup time depends on the (plaintext) input (during encryption).

While this timing leak appears minimal, it is enough to successfully - under certain conditions - extract the cipher-key[2]. It is important to note that this attack might be harder to carry out, especially when compared to an implementation written in C. This is because JavaScript is Just-in-Time (JIT) compiled.

Affected file:
aes-js/index.js

Affected code:

```
AES.prototype.encrypt = function(plaintext) {
[...]
        // apply round transforms
        for (var r = 1; r < rounds; r++) {
            for (var i = 0; i < 4; i++) {
                a[i] = (T1[(t[ i          ] >> 24) & 0xff] ^
                        T2[(t[(i + 1) % 4] >> 16) & 0xff] ^
                        T3[(t[(i + 2) % 4] >>  8) & 0xff] ^
                        T4[ t[(i + 3) % 4]        & 0xff] ^
                        this._Ke[r][i]);
            }
            t = a.slice();
        }

        // the last round is special
        var result = createArray(16), tt;
        for (var i = 0; i < 4; i++) {
            tt = this._Ke[rounds][i];
            result[4 * i    ] = (S[(t[ i          ] >> 24) & 0xff] ^ (tt >> 24))
& 0xff;
            result[4 * i + 1] = (S[(t[(i + 1) % 4] >> 16) & 0xff] ^ (tt >> 16))
& 0xff;
```

[1] https://www.npmjs.com/package/aes-js
[2] http://cr.yp.to/antiforgery/cachetiming-20050414.pdf

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

```
            result[4 * i + 2] = (S[(t[(i + 2) % 4] >>  8) & 0xff] ^ (tt >>  8))
& 0xff;
            result[4 * i + 3] = (S[ t[(i + 3) % 4]        & 0xff] ^  tt       )
& 0xff;
        }

        return result;
    }
```

It is recommended to use the WebCrypto or NodeJS crypto API for cryptographic primitives like AES. This is because these APIs are backed with the common, well-known libraries like OpenSSL (dependent on the actual OS), which are written in C or similar languages and ensure constant time execution whenever possible.

CAS-01-003 WP1: AES-CTR leaks plaintext info due to hardcoded nonce *(High)*

Auditing of the *AESUtils* class of *casper-storage* revealed that the *encrypt* method did not set the *counter* argument when instantiating the CTR mode of the operation[3]. As a result, the same *counter* value is used for all encryption operations. This translates to leaks of plaintext information, rendering the value vulnerable to certain plaintext attacks whenever the same key is used.

The CTR mode of operation works by turning a block-cipher (here AES) into a stream-cipher to generate a key stream. This key stream is then used to encrypt the plaintext using a simple XOR operation. To generate the key stream, a counter is encrypted with the key supplied by the caller. In case of AES, this counter is 16-bytes long and will generate 16 bytes of key stream. For the next 16 bytes of the key stream, the counter is incremented by 1 and encrypted again using the same key. This is done as long as there is plaintext to encrypt.

For this to be secure, the generated key stream must never repeat. Otherwise, an attacker can take two encrypted ciphertexts (XOR) and, using the same key stream, XOR them to get a XOR of the original plaintext message. If an attacker can launch a chosen plaintext attack, it is possible to fully recover the other plaintext. The key stream will repeat whenever the same input is supplied, for example the same keys and same counter values are in use. Thus, the CTR mode must always be supplied with a different starting value for the counter (called *nonce* or *IV*).

Affected file:
casper-storage-master/src/cryptography/utils/aes-utils.ts

[3] https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation#CTR

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

Affected code:

```
export class AESUtils {

  /**
   * Encrypt a string using AES in CTR mode
   * @param {string} key - The key to use for encryption.
   * @param {string} value - The value to encrypt.
   * @returns The encrypted value.
   */
  static encrypt(key: string, value: string): Promise<string> {
    if (!key) {
      throw new Error("Key is required")
    }
    if (!value) {
      throw new Error("Value is required")
    }

    // Ensure to have a strong private key
    const keyBytes = CryptoUtils.scrypt(key);

    // Convert the value into a byte array
    const textBytes = aes.utils.utf8.toBytes(value);

    // use CTR - Counter mode (recommended method)
    const aesCtr = new aes.ModeOfOperation.ctr(keyBytes);
    const encryptedBytes = aesCtr.encrypt(textBytes);
```

It is recommended to supply a different *nonce* for the CTR mode on every use with the same key. This *nonce* can be generated at random, or by incrementing a counter value outside. However, as *aes-js* uses the supplied counter as-is, the caller has to ensure that the supplied *nonce* values never cause the same CTR mode's key stream. This can happen if an 80-byte plaintext is encrypted with *nonce* = *15,* followed by an 20-byte plaintext with *nonce* = *18*, and using the same key for both operations.

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

CAS-01-004 WP1: No authenticity for ciphertexts of user-class *(High)*

During a source code review of the *casper-storage-master* repository, it was found that the library provided encryption and decryption functions in terms of AES in CTR mode. In contrast to AE (Authenticated Encryption) or AEAD (Authenticated Encryption with Associated Data) cipher modes (for instance AES in GCM mode), the used AES in CTR mode does not authenticate the ciphertext. Authenticity, and more general integrity, constitute vital properties of ciphertexts, since they make it possible for software to detect changes made to a ciphertext.

In case that an attacker is able to modify ciphertexts produced through the *aes-utils.ts* file, the *casper-storage* library would not notice this and might at best produce random errors to the user of the library.

Affected file:
casper-storage-master/src/cryptography/utils/aes-utils.ts

Affected code:

```
static encrypt(key: string, value: string): Promise<string> {
       [...]
       // use CTR - Counter mode (recommended method)
       const aesCtr = new aes.ModeOfOperation.ctr(keyBytes);
       const encryptedBytes = aesCtr.encrypt(textBytes);

       // Convert the encrypted bytes to a hex string
       const text = TypeUtils.convertArrayToHexString(encryptedBytes);

       return Promise.resolve(text);
}
[...]
static decrypt(key: string, encryptedValue: string): Promise<string>  {
       [...]
       // use CTR - Counter mode (recommended method)
       const aesCtr = new aes.ModeOfOperation.ctr(keyBytes);
       const textBytes = aesCtr.decrypt(encryptedTextBytes);

       // Convert back the decrypted bytes to the original string
       const text = aes.utils.utf8.fromBytes(textBytes);

       return Promise.resolve(text);
}
```

To mitigate this issue, Cure53 recommends using an AEAD ciphermode, for example AES-GCM, AES-GCM-SIV[4], XSalsa20/Poly1305 or ChaCha20/Poly1305.

[4] https://datatracker.ietf.org/doc/html/rfc8452

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

CAS-01-005 WP1: Broken AES-CTR encryption for user-class (*Critical*)

While reviewing the source code of the *casper-storage-master* repository, it was identified that the *user*-class relied on the *AESUtils* class for encryption and decryption. The *user* class provides *encrypt* and *decrypt* functions to the *caller* of the library. Furthermore, other publicly available functions like - for example - the *serialize* and *deserialize* functions - also use the *encrypt* and *decrypt* function of the *user* class.

Cure53 found that the aforementioned functions used a constant key, namely the output of a *PBKDF2* function applied to the user's password. As described in CAS-01-003, this enables plaintext recovery via a chosen or known plaintext attack for all ciphertexts encrypted with the same key (user password).

A concrete attack works as follows:

1. Suppose that an attacker gets a hold of one plaintext (*pt1*) together with the corresponding ciphertext (*ct1)*.
2. Observe that *ct1 = pt1 + ks*, where *ks(k,n)* denotes the key stream generated by key *k* and nonce *n* and + is a bitwise XOR operation.
3. The library uses the same nonce and key for all encryption operations in *AESUtils*/
4. Based on the above, an attacker can obtain an arbitrary plaintext *pt2* by capturing the ciphertext *ct2* through the following two steps:
    1. Computing *ct1 + ct2 = pt1 + ks(k,n) + pt2 + ks(k,n) = pt1 + pt2*. Note here that *ks(k,n)* cancels out since all encryption operations utilize the same key and nonce.
    2. XOR'ing *pt1* to this result
5. Ultimately, the attacker has reached *pt2*.

Affected file:
casper-storage-master/src/user/user.ts

Affected code:

```
public encrypt(value: string): Promise<string> {
       return AESUtils.encrypt(this.password.getPassword(), value);
}

public decrypt(value: string): Promise<string> {
       return AESUtils.decrypt(this.password.getPassword(), value);
}
```

To mitigate this issue, Cure53 strongly advises to use an AEAD cipher as recommended in CAS-01-004. It has to be ensured that a fresh nonce is generated for every *encrypt* operation the library performs. For nonce generation, there are generally two options:

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

- A cryptographically secure pseudo-random number generator (CSPRNG)
- A counter which is persisted (e.g., in the password hashing options of *user*)

For AES-GCM and ChaCha20/Poly1305, the second option is usually required[5]. For AES-GCM-SIV and XSalsa20/Poly1305, both options can be considered. Overall, it is recommended to avoid using the same cipher-key (user-password) endlessly, because it will certainly lead to a *nonce* reuse at some point. It must be noted, however, that - depending on the *nonce* size - this requires a large number of encryption operations. Hence, the latter is unlikely in this setting.

CAS-01-006 WP1: Plaintext storage in browsers' local storage *(Medium)*

During a source code review of the *casper-storage-master* repository, Cure53 confirmed that the library provides a *storage* functionality, as also described on the official GitHub page[6]. The implementation uses - by default - the local storage of the browser which runs the library.

Local storage within browsers is not encrypted and can be accessed by exploiting XSS or similar vulnerabilities. Furthermore, the browser does not clear the local storage on closing and reopening. These circumstances render the local storage of the browser not suitable for persisting security-sensitive information. In other words, any plaintext information the user persists within the storage of the library is put at risk. Since the *user*-class additionally allows to *serialize/deserialize* plaintext information (see CAS-01-009) this could put the user's wallet at risk of being accessible to third parties.

Affected file:
casper-storage-master/src/storage/providers/default-storage.ts

Affected code:

```
async set(key: string, value: string): Promise<void> {
        window.localStorage.setItem(key, value);
}
```

To mitigate this issue, Cure53 recommends to either remove this functionality entirely or to always encrypt information before persisting it in the browser storage. This could be achieved by using a derivation of the user password as the encryption key.

[5] https://www.rfc-editor.org/rfc/rfc7539#section-4
[6] https://github.com/CasperDash/casper-storage#storage

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

CAS-01-007 WP1: No validation of key-seed in *CasperHDWallet* (*Medium*)

Dynamic testing of the *casper-storage* library revealed that the *CasperHDWallet* class offered a constructor which relied on a seed and encryption type. The seed to the constructor is then used to derive a new HD key from the given seed. Official standards provide recommendations of such seed with respect to lengths[7]. Having no restrictions regarding seeds could result in enumerations of master keys for the HD wallet, additionally fostering weak keys.

Affected file:
casper-storage-master/src/wallet/common/casper/casper-hd-wallet.ts

Affected code:

```
export class CasperHDWallet extends HDWallet<CasperWallet> {

	private static PATH = new CoinPath(Purpose.BIP44, CoinType.Casper);

	constructor(masterSeed: Hex, encryptionType: EncryptionType =
	EncryptionType.Ed25519) {
		super(CasperWallet, CasperHDWallet.PATH, encryptionType,
		masterSeed);
	}

}
```

As demonstrated above, the *CaspterHDWallet* class extends the *HDWallet* class. In the constructor, the class calls the constructor of *HDWallet* passing on the *masterSeed* without validation. Furthermore, the constructor of *HDWallet* does not validate the provided seed either.

Affected file:
casper-storage-master/src/wallet/hdwallet/hd-wallet.ts

Affected code:

```
public async getWalletFromPath(path: string): Promise<TWallet> {
	const hdKey = await
	this.getHDKeyManager().fromMasterSeed(this.masterSeed).derive(path);
	return Promise.resolve(new this.walletConstructor(hdKey,
	this.encryptionType));
}
```

The *getWalletFromPath* function uses the *masterSeed* without further validation.

[7] https://en.bitcoin.it/wiki/BIP_0032#Master_key_generation

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

Affected file:
casper-storage-master/src/bips/bip32/hd-key-manager-base.ts

Affected code:

```
public fromMasterSeed(seed: Hex, versions?: Versions) {
	if (!seed || !seed.length) {
		throw new Error("Master seed is required");
	}

	const { key, chainCode } = CryptoUtils.digestSHA512(seed,
	this.GetMasterSecret());
	return this.createNewHDKey(key, chainCode, versions || BITCOIN_VERSIONS);
}
```

To mitigate this issue, Cure53 advises to check the length of the provided seed before using it within the process of deriving a new master key for the HD wallet.

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.

CAS-01-008 WP1: Salt reuse in *update password* function *(Low)*

A source code review of the *casper-storage-master* repository corroborated that the *user*-class provides an *updatePassword* function to the developers who use this library. This function means that one can pass an instance of *PasswordOptions*, which also contains a salt for deriving an encryption key from the user's password. The option to let the user reuse the same salt for a password change is dangerous. With identical passwords, this essentially yields identical encryption keys for the *user*-class.

Two conditions have to be met before an attacker could perform decryption operations, First, the attacker would need to get a hold of a user's past encryption key derived from a password. Second, the user would need to reset the password to this particular password.

Affected file:
casper-storage-master/src/user/user.ts

Affected code:

```
public updatePassword(
	newPassword: string,
	options?: Partial<PasswordOptions>
) {
	this.password = new Password(newPassword, options);
}
```

Affected file:
casper-storage-master/src/user/core.ts

Affected code:

```
export interface PasswordOptions {
	[...]
	/* Salt is a random value that is used to make the hash more secure. */
	salt: Uint8Array;
	[...]
}
```

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

To mitigate this issue, Cure53 advises to always generate a new random salt on each action aimed at changing a user's password.

CAS-01-009 WP1: Configurable encryption on user's *serialize/deserialize* actions *(Info)*

While reviewing the source code of the *casper-storage-master* repository, it was identified that the *user*-class implements *serialize* and *deserialize* functions to import or export the wallet of a user. Those functions contain an *encrypt* parameter which defaults to *true*. In that case, the *serialize* function encrypts the *JSON* representation of the wallet of a user. However, the parameter could be adjusted by the developer, thereby turning off encryption. It should be noted that turning off encryption on this function leaks the private key of the user's wallet.

Even though this does not constitute an immediate security risk, it was decided to file this observation as an issue. This is because it essentially permits an insecure use of the library which could result in security breaches.

Affected file:
casper-storage-master/src/user/user.ts

Affected code:

```
public async serialize(encrypt = true): Promise<string> {
	[...]
	let result = JSON.stringify(obj);
	if (encrypt) {
		result = await this.encrypt(result);
	}
	return result;
}
[...]
public async deserialize(value: string, encrypted = true): Promise<void> {
	let text = value;
	try {
		if (encrypted) {
			text = await this.decrypt(value);
		}
	} catch (err) {
		throw new Error(`Unable to decrypt user information. Error: $
		{err}`);
	}
	[...]
```

To mitigate this issue, Cure53 advises to remove the *encrypt* parameter from the *serialize* and *deserialize* functions of the *user*-class and always encrypt the JSON representation of a wallet.

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

CAS-01-010 WP1: Low default entropy for mnemonic generator *(Info)*

During a review of the *KeyFactory* class in *casper-storage-master,* it was noticed that the default *MnemonicKey* generator defaults to creating mnemonics with 128 bits of entropy. As the primary use of this class is for seed generation of the Casper HD wallet, it is recommended to increase the default entropy in this context.

The Casper HD wallet implements BIP32[8] hierarchical deterministic wallets. At the top of this hierarchy is a master key which is generated from a random seed (master seed), characterized by 128 to 512 bits of entropy. Per specification, the recommended size would be 256 bits of entropy. Currently, *MnemonicKey* defaults to generating 128 bits of entropy which results in a 12-words mnemonic. This is the lower bound of the BIP32 master seed.

Affected file:
casper-storage-master/src/key/mnemonic/mnemonic-key.ts

Affected code:

```
/**
 * Available options
 * https://github.com/bitcoin/bips/blob/master/bip-0039.mediawiki
 */
const WORDS_LENGTH_STRENGTH_MAP = new Map<number, number>([
  [12, 128],
  [15, 160],
  [18, 192],
  [21, 224],
  [24, 256]
]);

/**
 * Let's use the longest available length
 */
const DEFAULT_WORDS_LENGTH = 12;

/**
 * Wrapper to work with mnemonic
 */
export class MnemonicKey implements IKeyManager {

  generate(wordsLength?: number): string {
    if (wordsLength == null) {
      wordsLength = DEFAULT_WORDS_LENGTH;
    }
    if (!WORDS_LENGTH_STRENGTH_MAP.has(wordsLength)) {
```

[8] 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

```
      throw new Error(`Length of words must be in allowed list: $
{Array.from(WORDS_LENGTH_STRENGTH_MAP.keys()).join(", ")}`)
    }
    const byteLength = WORDS_LENGTH_STRENGTH_MAP.get(wordsLength);
    const entropy = CryptoUtils.randomBytes(byteLength / 8);
    return bip39.entropyToMnemonic(entropy, wordlist);
  }
```

It is suggested to increase the default entropy of *MnemonicKey* to 256 bits, such that it matches the recommended master seed size in BIP32. This will result in a 24-words mnemonic.

CAS-01-011 WP1: Non-standard BIP32 compliance in master key derivation *(Info)*

While reviewing the source code of the *casper-storage-master* repository, it was identified that the HD Key Manager implementation employs the BIP32 standard for HD wallets[9]. This standard states that implementations derive a new master secret key by applying a HMAC-SHA512 to a seed phrase with the *"Bitcoin seed"* key.

The result of this operation contains both the master secret key and the master chain code in the lower and upper 32 bytes, respectively. However, the standard also explicitly states that the derived master secret key should not be *zero* and that the derived master chain code shall not exceed *n*. Note that *n* here refers to the order of the *Secp256k1* curve. In case any of the conditions is not met, the master key is seen as invalid.

Affected file:
casper-storage-master/src/bips/bip32/hd-key-manager-secp256k1.ts

Affected code:

```
protected createNewHDKey(privateKey: Uint8Array, chainCode: Uint8Array,
versions: Versions) : IHDKey {
        return new HDKeySecp256k1(new HDKeyConfig(this.encryptionType, versions,
        this.GetMasterSecret()), privateKey, chainCode, null);
}
```

To mitigate this issue, Cure53 advises to fully comply with the BIP32 standards. This will further prevent generation of weak keys.

[9] https://en.bitcoin.it/wiki/BIP_0032#Master_key_generation

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

CAS-01-012 WP1: PBKDF2 implementation allows weak parameters *(Info)*

While auditing the *CryptoUtils* class, it was noticed that both methods exposed to calculate *PBKDF2-HMAC-SHA512* allow for weak input parameters. This can result in weak, crackable hashes.

The PBKDF2[10] algorithm is a key derivation function which transforms an arbitrary input message into a password hash, usually combining a user-password with a salt. It is designed to require more computational resources than a regular SHA512 hash would need. The main reason is to increase the effort for brute-force and similar attacks. To make this effort tunable, especially with hardware getting faster and more affordable, the algorithm also takes an *iterations* parameter as input. This parameter specifically controls the computational effort. In addition, it accepts a *keyLength* input, which controls the length of the algorithm output. The latter can be used to reduce the output's length.

The implementation in *casper-storage-master* allows to specify arbitrary values for *iterations* and *keyLength* parameters, allowing for weak values like *iterations* = *1* or *keyLength* = *1*. All in all, this will result in weak password hashes.

Affected file:
casper-storage-master/src/cryptography/utils/crypto-utils.ts

Affected code:

```
/**
   * It hashes the password using the PBKDF2 algorithm.
   * @param {Uint8Array} password - The password to use for the key derivation.
   * @param {Uint8Array} salt - A salt to use for the key derivation function.
   * @param {number} iterations - The number of iterations to perform.
   * @param {number} keySize - The size of the derived key in bytes.
   * @returns The PBKDF2 function returns a Uint8Array.
   */
  static pbkdf2Sync(password: Uint8Array, salt: Uint8Array, iterations: number,
keySize: number): Uint8Array {
    return pbkdf2(sha512, password, salt, { c: iterations, dkLen: keySize });
  }

  /**
   * It uses the SHA-512 hash function to generate a key from the password and
salt.
   * @param {Uint8Array} password - The password to use for the key derivation.
   * @param {Uint8Array} salt - A salt to use for the key derivation function.
   * @param {number} iterations - The number of iterations to perform.
   * @param {number} keySize - The size of the derived key in bytes.
```

[10] https://www.ietf.org/rfc/rfc2898.html

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

```
   * @returns The PBKDF2 function returns a Promise that resolves to a
Uint8Array containing the
   * derived key.
   */
  static pbkdf2(password: Uint8Array, salt: Uint8Array, iterations: number,
keySize: number): Promise<Uint8Array> {
    return pbkdf2Async(sha512, password, salt, { c: iterations, dkLen:
      keySize });
  }
```

It is suggested to enforce a minimum value for *iterations* and *keyLength* to ensure that the output of PBKDF2 is secure. OWASP recommends 120.000 iterations for PBKDF2-HMAC-SHA512[11]. It is also worth-considering to require a minimum length for the *salt* value as it increases the *search* space for dictionary attacks. RFC2898 recommends a minimum of 8 bytes[12], but as this RFC is more than 20 years old, higher values may be more suitable.

CAS-01-013 WP1: ed25519 HDKey derivation does not follow SLIP-0010 *(Low)*

Cure53 audited the *HDKeyED25519* class, which implements a key derivation algorithm similar to BIP32 specified as SLIP-0010[13]. Here it was noticed that one could derive non-hardened private keys from the implementation. According to the specification, only hardened keys can be derived for *ed25519*, whilst trying to derive a non-hardened key must return a failure.

The SLIP-0010 specification follows BIP32 very closely in the sense that it defines the same procedure to derive private and public keys from a master seed. The main difference is that SLIP-0010 extends the BIP32 definition of other ECC curves besides *Secp256k1*. One of these is *ed25519*, which is implemented by *casper-storage* within the *HDKeyED25519* class. As per specification, implementations of SLIP-0010 for *ed25519* must only allow users to derive hardened keys from a private key. This means that trying to derive a key for a non-hardened index ($0 \leq \text{index} < 2^{32}$) must result in a failure returned by the library. This is not the case for *HDKeyED25519.deriveChild*.

Affected file:
casper-storage-master/src/bips/bip32/hdkey/hd-key-ed25519.ts

[11] https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html#pbkdf2
[12] https://www.ietf.org/rfc/rfc2898.html#section-4.1
[13] https://github.com/satoshilabs/slips/blob/master/slip-0010.md

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

Affected code:

```
export class HDKeyED25519 extends HDKey {

  /**
   * It creates a new HDKeyED25519 object.
   * @param {Uint8Array} privateKey - The private key.
   * @param {Uint8Array} chainCode - The chain code is a 32-byte sequence that
is used to derive child keys.
   * @param {Uint8Array} publicKey - The public key of the new HDKey.
   * @returns An HDKeyED25519 object.
   */
  protected createNewHDKey(privateKey: Uint8Array, chainCode: Uint8Array,
publicKey: Uint8Array) {
    return new HDKeyED25519(this.config, privateKey, chainCode, publicKey);
  }

  /**
   * Create a new HDKey from the current HDKey and the provided index
   * @param {number} index - The index of the child key to derive.
   * @returns An HDKey object.
   */
  protected deriveChild(index: number): Promise<HDKey> {
    const privateKey = this.getPrivateKey();
    if (!privateKey) {
      throw new Error("Cannot derive a hardened child without private key")
    }

    // data = 0x00 || ser256(kpar) || ser32(index)
    const data = TypeUtils.concatBytes(TypeUtils.getBytesOfZero(), privateKey,
      TypeUtils.convertU32ToBytes(index));

    const { key, chainCode } = CryptoUtils.digestSHA512(data,
this.getChainCode());
    return this.createChildHDKey(index, key, chainCode, null);
  }
```

As can be seen from the code listing above, there is no check for the *index* being in the required number range for hardened keys (≥ 2^{31}).

While this does not constitute a security issue by itself, it allows for non-standard use of ed25519 master keys. These could potentially cause further, yet unspecified harm. It is suggested to add the required check, meaning that only *index* values ≥ 2^{31} should be allowed.

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

Conclusions

As noted in the *Introduction,* Cure53 concludes that there is still a lot of room of improvement as far as the security posture of CasperDash is concerned. The findings of this cryptographic review and source code audit - especially in regard to the presence of *Critical* and *High* problems - clearly indicate that the CasperDash team needs to invest more efforts into strengthening their wallet library against various attacks. Thirteen risks unveiled during this September 2022 assessment should be investigated and addressed as stepping stones towards a better security outcome.

It should be reiterated that this security assessment mostly encompassed an audit focused on the cryptography of the *casper-storage-master* repository. As such, the repository was reviewed with special focus on cryptographic flaws, including - but not limited to - broken ciphers, inappropriate cryptographic primitives for sensitive tasks, authentication failures, key generation, key storage but also side-channel attacks and insecure defaults.

The repository in scope contains the *casper-storage* implementation for crypto wallets, and it has been written in TypeScript. The source code is well-organized and its structure can be characterized as clear. Furthermore, the repository also contained a detailed *README.md* file which is explicitly dedicated to the usage of the library.

In summary, the audit revealed the presence of seven vulnerabilities and six miscellaneous issues in CasperDash. The highest-rated vulnerability pertained to a broken AES CTR implementation, which explains its *Critical* severity ranking (see CAS-01-005). Specifically, the library reuses the same *nonce* together with the same encryption key for multiple *encrypt* operations. In case an attacker reaches just one plaintext, all ciphertexts created by the same key can be decrypted.

Besides the major flaw described above, the audit also revealed several other vulnerabilities, for instance in regard to missing authentication of ciphertexts and hard-coded salts. The miscellaneous issues relate to diverse weaknesses of the library, including low default entropy or lack of standards-compliance. Possible improvements for defaults and the public APIs were also suggested by Cure53.

As the main goal of *casper-storage* includes furnishing a secure way to store wallet secrets, providing the caller with the option to disable the storage encryption (see CAS-01-009) seems counter intuitive for such a library and should be remedied. This takes place when the default storage implementation uses the Local Storage API, which is accessible to all code running in the context of the web page (see CAS-01-006).

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

It should also be noted that the library is implemented in TypeScript, which allows specifying types for variables. This does not mean, however, that these types are enforced at runtime. Instead, using this library from a JavaScript context makes it possible to hand over different types for variables than *casper-storage* classes expect. This will mostly go unnoticed at runtime and can have unwanted side effects. Should the goal of *casper-storage* be to also be extended to JavaScript applications, it is highly recommended to add runtime type-checks (*typeof* and similar) to enforce correct types.

One example where this can be an issue is the method *CoinPath.createPath(accountIndex, internal, walletIndex)*, which expects *accountIndex* and *walletIndex* to be numbers. This is mostly because they are used to construct a BIP32-compliant path. These types are not enforced at runtime, thus handing down a string for any of the arguments will work. Essentially, it equips an attacker the possibility to influence the internally generated path string in unwanted ways, as can be considered for *walletIndex='0/100/12'*.

The impression that Cure53 gained about the *casper-storage* is mixed. Besides the vulnerabilities surrounding *nonce*-reuse for AES and scrypt, the library appears to be in good shape and no major issues were found in the ECC ciphers. This is mainly due to the use of the *@noble/** libraries which implement the underlying ciphers and include appropriate error checks. The issues surrounding *nonce*-reuse conversely point to the fact that some fundamental concepts of secure cryptographic code have not been respected.

In the end, the audit achieved a good coverage of the repository in scope. Cure53 was in constant communication with the customer through a dedicated Slack channel. The communication was excellent and help was provided whenever requested. As the main outcome of this Cure53 September 2022 audit, addressing the vulnerabilities of *Critical* and *High* severity should have the highest priority. This will help to strengthen the security offered to users who rely on this library. Furthermore, mitigating all other vulnerabilities can narrow down the attack surface, while fixing the miscellaneous issues would provide defense-in-depth.

Moving forward, the library would certainly benefit from recurring cryptography audits. This is needed to make sure that the discovered and existing vulnerabilities have been mitigated correctly. Furthermore, as development progresses, it is the best way to guarantee that new features are unlikely to introduce new vulnerabilities.

Cure53 would like to thank Kien Nguyen, Hoang Dang and Thinh Nguyen from the CasperDash team for their excellent project coordination, support and assistance, both before and during this assignment.