Fine penetration tests for fine websites

Dr.-Ing. Mario Heiderich, Cure53
Rudolf Reusch Str. 33
D 10367 Berlin
cure53.de · mario@cure53.de

Audit-Report MetaMask key-tree Interface 02.2023

Cure53, Dr.-Ing. M. Heiderich, Dr. M. Conde

Index

Dr.-Ing. Mario Heiderich, Cure53
Rudolf Reusch Str. 33
D 10367 Berlin
cure53.de · mario@cure53.de

Introduction

"MetaMask provides the simplest yet most secure way to connect to blockchain-based applications. You are always in control when interacting on the new decentralized web."

From https://metamask.io/

This report details the scope, results, and conclusory summaries of a cryptography review and source code audit against the MetaMask key-tree interface. The assessment was requested by ConsenSys Software Inc. in February 2023 and initiated by Cure53 in the same month, namely in CW08. A total of six days were allocated to fulfill this project's coverage expectations. All evaluation actions for this review were condensed into a single work package (WP) for execution efficiency, as follows:

- **WP1**: Crypto review & code audit against the MetaMask key-tree interface

Cure53 was granted access to the repository, the corresponding commit, and any alternative means of access required to ensure a smooth audit completion. For this purpose, the selected methodology was white-box and a team comprising two skillmatched senior testers was assigned to the project's preparation, execution, and finalization. All preparatory actions were completed in February 2023, namely in CW07, to ensure testing could proceed without hindrance or delay.

Communications were facilitated via a dedicated, shared Slack channel deployed to combine the workspaces of ConsenSys Software and Cure53, thereby creating an optimal collaborative working environment. All participatory personnel from both parties were invited to partake throughout the test preparations and discussions. In light of this, communications proceeded smoothly on the whole. The scope was well-prepared and transparent, no noteworthy roadblocks were encountered throughout testing, and cross-team queries remained minimal as a result.

Cure53 gave frequent status updates concerning the test and any related findings, whilst simultaneously offering prompt queries and receiving efficient, effective answers from the maintainers. Live reporting was offered by Cure53 and implemented via the aforementioned Slack channel.

Concerning the findings specifically, the testing team achieved widespread coverage over the sole scope item, detecting a total of seven. Five of the findings were categorized as security vulnerabilities, whilst the remaining two were deemed general weaknesses with lower exploitation potential.

Dr.-Ing. Mario Heiderich, Cure53
Rudolf Reusch Str. 33
D 10367 Berlin
cure53.de · mario@cure53.de

The total yield of findings is relatively moderate, which garners a positive impression concerning the security foundation exhibited by the MetaMask key-tree interface. Additionally, the fact that none of the identified weaknesses exceeded a severity rating of *Medium* attests to the relatively low exploitation potential persisted by the components in scope.

All in all, Cure53 is pleased to confirm that the MetaMask team has made commendable progress toward providing a sufficiently secured interface. Nevertheless, one can strongly recommend addressing all issues documented in this report at the earliest possible convenience to elevate the security foundation to an exemplary standard.

The report will now shed more light on the scope and testing setup as well as provide a comprehensive breakdown of the available materials. This section will be followed by a chapter detailing the cryptography review, which serves to provide information regarding the scope and findings, as well as highlight this audit's key goals and threat and attacker model. Furthermore, the coverage achieved and assessment executed pertaining to the in-scope interface areas are extrapolated. Finally, Cure53 highlights potential focus areas and considerations for future MetaMask key-tree interface enhancements.

Subsequently, the report will list all findings identified in chronological order, starting with the detected vulnerabilities and followed by the general weaknesses unearthed. Each finding will be accompanied by a technical description and Proof of Concepts (PoCs) where applicable, plus any relevant mitigatory or preventative advice to action.

In summation, the report will finalize with a conclusion in which the Cure53 team will elaborate on the impressions gained toward the general security posture of the MetaMask key-tree interface, giving high-level hardening advice where applicable.

Dr.-Ing. Mario Heiderich, Cure53
Rudolf Reusch Str. 33
D 10367 Berlin
cure53.de · mario@cure53.de

Scope

- **Crypto review & code audit against the MetaMask key-tree interface**
  - **WP1**: Crypto review & code audit against the MetaMask key-tree interface
    - **Package:**
      - https://www.npmjs.com/package/@metamask/key-tree
    - **All sources were shared with Cure53 and are available as OSS:**
      - https://github.com/MetaMask/key-tree
    - **Version in scope:**
      - 6.2.1
  - **Key focus areas:**
    - Functional correctness of elliptic curve operations in use.
    - Elliptic curve validation errors.
    - Functional correctness of master key and child keys derivation function.
    - Implementation logic to validate key derivation paths.
  - **Test-supporting material was shared with Cure53**
  - **All relevant sources were shared with Cure53**

Dr.-Ing. Mario Heiderich, Cure53
Rudolf Reusch Str. 33
D 10367 Berlin
cure53.de · mario@cure53.de

Cryptography review

In blockchain, the creation of a unique private/public keypair for each transaction is a crucial element toward ensuring comprehensive security and privacy. Subsequently, every private key must be backed up, for which a popular notion for wallet construction simplification constitutes a *hierarchical deterministic* wallet (or HD wallet for short). At a very high level, an HD wallet allows one to derive child keys from a single seed, thus *deterministic*, and each child key can in turn be used to derive other child keys, thus *hierarchical*, since keys can be organized into a tree structure or hierarchy.

The logic behind an HD wallet pertains to the following:

The seed is used to generate a master private extended key as follows:

> *private master extended key = private master key (32 bytes), chain code (32 bytes) = HMAC-SHA512("Bitcoin seed", seed)*

The master public key can be computed from the master private key as usual, and the master public key together with the exact same chain code above represents the master public extended key. The chain code (32 random bytes) is required to derive child keys, which is why the concept of extended key is introduced.

Any extended key can be used to derive a child key, and as such the extended key is referred to as its parent. The number of child keys that can be derived from any extended key constitutes 2^{32} maximum, so each child key has an associated index. There are two types of child keys that can be derived: hardened (only the private extended key can be used to derive the child public key) or unhardened (both the extended private key or extended public key can be used to derive the child public key). The algorithm to derive a hardened child private key (stated only over secp256k1 for simplicity) constitutes the following:

- *key material (32 bytes), child chain code (32 bytes) = HMAC-SHA512(parent chain code, 0x00 || parent private key || index)*
- *child private key (32 bytes) = key material + parent private key (mod curve order)*

The method by which to derive a non-hardened child key (stated only over secp256k1 for simplicity) represents the following:

- *key material (32 bytes), child chain code (32 bytes) = HMAC-SHA512(parent chain code, parent public key || index)*
- Non-hardened private child key:
  child private key (32 bytes) = key material + parent private key (mod curve order)

Dr.-Ing. Mario Heiderich, Cure53
Rudolf Reusch Str. 33
D 10367 Berlin
cure53.de · mario@cure53.de

- Non-hardened public child key:
  child public key (32 bytes) = key material + parent public key (point addition)

Several BIP and SLIP specifications related to HD wallets are relevant for this review, since MetaMask's *key-tree* interface is based upon them. All of these are extrapolated below:

- **BIP32:** This specification describes an algorithm to generate a master key from a sufficiently random seed, and child keys from a given parent extended key. This algorithm is based on HMAC-SHA512 and elliptic curve operations performed over the curve secp256k1. Furthermore, BIP32 introduces a notation to refer to a given child key by its derivation path. For example, m/0 denotes the child key with index 0 that is generated from the master key, with a depth 1 since it's one derivation away from the master key. m/0/1 denotes the child key with index 1 derived from the child key, an index 0 derived from the master key (with depth 2), and so on. In this notation, a hardened child is indicated by an apostrophe ('); for instance, m/0' denotes the hardened child key with index 0 derived from the master key.
- **BIP39:** This specification details a user-friendly method by which to create a random seed from a mnemonic code (12-24 words), which is easier for users to remember or safely store.
- **SLIP10:** This specification is a generalization of BIP32 to include support for other curves, namely secp256r1 (NIST P-256) and ed25519. SLIP10 reduces exactly to BIP32 if secp256k1 is considered.
- **BIP44:** BIP32 imposes no constraints on hierarchy levels or use of keys of each level, which may lead to interoperability issues amongst wallets following BIP32. This specification defines a standard logical hierarchy for HD wallets, establishing at most five hierarchical levels, as follows:

  m/purpose'/coin type'/account'/change/address index

MetaMask's *key-tree* package offers an interface that allows the creation of child keys for any depth of a SLIP10 or BIP44 path. Notably, this relies on a third-party implementation of the curve secp256k1 and includes support for ed25519 (though not for NIST-256).

Dr.-Ing. Mario Heiderich, Cure53
Rudolf Reusch Str. 33
D 10367 Berlin
cure53.de · mario@cure53.de

Scope & findings

Due to the relatively simple cryptographic primitives underlying the specifications and in light of the above, Cure53 focused on a cryptographic review of the following functionalities deemed sensitive in nature:

- *createBip39KeyFromSeed*: used to generate the master key from a BIP39 seed.
- *publicAdd*: elliptic curve logic used to generate unhardened child public keys.
- *privateAdd*: elliptic curve logic used to generate private child keys (both hardened and unhardened).
- *fromExtendedKey*: method to instantiate extended keys with certain properties (depth, index, parent fingerprint, etc.).
- *validatePathSegment*: logic to validate a derivation path.
- *deriveChildKey*: logic to generate a child key either from a master key, from a parent extended key, or from a derivation path.

Here, Cure53 would like to underline that the majority of detected issues pertain to an improper check concerning whether the validity of the master key or the child keys derived from a parent extended key. Nevertheless, this is likely risk-averse in practice since the probability of occurrence is extremely low, though still remains formally incorrect and supposes a deviation from the SLIP10 standard.

Key goals

The primary goal of this audit was to verify that the *key-tree* package permits secure key creation for any valid SLIP10 or BIP44 path.

From a cryptographic perspective, this involves ensuring that the SLIP10 and BIP44 specifications are followed so that the master key and child key derivation are correctly realized. From an implementation point of view, one must validate the correctness of both the logic utilized to establish key hierarchy and the logic used to validate key derivation paths.

Threat & attacker models

If the BIP32/SLIP10 strong specification is followed, the creation of keys is secure - if correctly implemented - under the assumption that the seed, the private master key, and every other extended private key are not leaked at any point.

MetaMask's key-tree repository was analyzed to verify that it does not persist any additional attack surface to that stipulated by the specifications.

Dr.-Ing. Mario Heiderich, Cure53
Rudolf Reusch Str. 33
D 10367 Berlin
cure53.de · mario@cure53.de

Test methodology

The assessment was initiated by reviewing the correctness of the elliptic curve operations involved in the algorithm for child key derivation. The integration of these elliptic curve operations within the child key derivation function was also reviewed. Cure53 paid particular attention to the master key derivation and any potentially invalid keys that could arise during child key derivation.

Since MetaMask's *key-tree* constitutes an implementation for key creation for SLIP10 or BIP44 paths, full compliance with these specifications was closely reviewed, particularly regarding child key derivation.

Finally, the logic used to validate extended keys and key derivation paths was subjected to stringent review in order to locate any potential weaknesses that may impact child key derivation security, though no severe associated issues were identified.

Future work & considerations

All in all, Cure53 is pleased to confirm that MetaMask's *key-tree* interface facilitates secure key creation for SLIP10 or BIPP44 paths. Nevertheless, the implementation would certainly benefit from ensuring the expected rejection of invalid extended keys, which would help to negate any undesirable attack avenues.

Dr.-Ing. Mario Heiderich, Cure53
Rudolf Reusch Str. 33
D 10367 Berlin
cure53.de · mario@cure53.de

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., *MM-02-001*) to facilitate any future follow-up correspondence.

MM-02-001 WP1: Public key addition allows all-zero vector as input *(Low)*

Whilst reviewing the *key-tree* repository, the observation was made that the *deriveChildKey* function ultimately relies on a call to the *publicAdd* function whenever an unhardened public key child is due for derivation from a parent public key. However, the *publicAdd* function accepts an input that results in the generation of a public key identical to the parent public key.

Specifically, the *publicAdd* function inserts two points of the elliptic curve given as an input to the function. When called inside the function *deriveChildKey*, the addition of the parent public key is triggered with an arbitrary point *tweak*G*, whereby *tweak* represents an input to the function and *G* represents the curve generator. However, in the event *tweak* equals the all-zero vector, the child public key equals the addition of the parent public key and 0*G. This addition yields precisely the parent public key, since 0*G equals the neutral element of the elliptic curve. All in all, this means that the derived public key child equals the parent public key.

Public key reuse - and thus the reuse of an address - is considered an undesirable behavior in the blockchain context for privacy reasons. Transactions are publicly recorded on the blockchain, whilst address reuse may facilitate easier tracking, which is particularly suboptimal if the address is linkable to an identity.

Affected file:
key-tree/src/curves/secp256k1.ts

Affected code:

```
export const publicAdd = (
  publicKey: Uint8Array,
  tweak: Uint8Array,
): Uint8Array => {
  const point = Point.fromHex(publicKey);
  [...]
  const newPoint = point.add(Point.fromPrivateKey(tweak));

  newPoint.assertValidity();
  return newPoint.toRawBytes(false);
};
```

Dr.-Ing. Mario Heiderich, Cure53
Rudolf Reusch Str. 33
D 10367 Berlin
cure53.de · mario@cure53.de

To mitigate this issue, Cure53 advises enforcing a check to ensure that *tweak* differs from the all-zero vector, though the probability of this condition holding is negligible.

MM-02-002 WP1: Private key addition allows all-zero vector as input *(Low)*

Whilst reviewing the *key-tree* repository, the observation was made that the *deriveChildKey* function ultimately relies on a call to the *privateAdd* function whenever a private key child is due for derivation from a parent private key. However, the *privateAdd* function accepts an input that results in the generation of a private key identical to the parent private key.

Specifically, the *privateAdd* function inserts two scalars given as an input to the function. When called inside the *deriveChildKey* function, the addition of the parent private key is triggered with an arbitrary scalar *tweak*, whereby *tweak* represents an input to the function. However, in the eventuality *tweak* equals the all-zero vector, this addition will precisely yield the parent private key.

All in all, this behavior means that the derived child private key equals the parent private key, which constitutes an undesirable behavior as previously mentioned in ticket MM-02-001.

Affected file:
key-tree/src/derivers/bip32.ts

Affected code:

```
export function privateAdd(
  privateKeyBytes: Uint8Array,
  tweakBytes: Uint8Array,
  curve: Curve,
): Uint8Array {
  const privateKey = bytesToBigInt(privateKeyBytes);
  const tweak = bytesToBigInt(tweakBytes);

  if (tweak >= curve.curve.n) {
    throw new Error('Invalid tweak: Tweak is larger than the curve order.');
  }
  const added = mod(privateKey + tweak, curve.curve.n);
  const bytes = hexToBytes(added.toString(16).padStart(64, '0'));
  if (!curve.isValidPrivateKey(bytes)) {
    throw new Error(
      'Invalid private key or tweak: The resulting private key is invalid.',
    );
  }
  return bytes;
}
```

Dr.-Ing. Mario Heiderich, Cure53
Rudolf Reusch Str. 33
D 10367 Berlin
cure53.de · mario@cure53.de

To mitigate this issue, Cure53 advises enforcing that the *tweak* differs from the all-zero vector, though the probability of this condition holding is negligible.

MM-02-004 WP1: Potentially invalid master private key *(Low)*

Whilst assessing the *key-tree* repository, the discovery was made that the derivation of the master private key is not fully compliant with BIP32 specification, as invalid private master keys are not discarded if the curve secp256k1 is utilized. This is not relevant for the ed25519 curve, since all possible private master keys are valid in this scenario.

For the secp256k1 in particular, one must ensure that the resulting 32-bytes private master key (interpreted as an integer) computed in *createBip39KeyFromSeed* is greater than zero and strictly less than the curve order. The check against an all-zero private master key is achieved when an SLIP10 node intends to be instantiated with a private key of this nature, resulting in an error. However, the check to guarantee that the 32-bytes private master key is strictly less than the curve order is absent.

Affected file:
key-tree/src/derivers/bip39.ts

Affected code:

```
export async function createBip39KeyFromSeed(
  seed: Uint8Array,
  curve: Curve = secp256k1,
): Promise<SLIP10Node> {
  const key = hmac(sha512, curve.secret, seed);
  const privateKey = key.slice(0, 32);
  const chainCode = key.slice(32);

  const masterFingerprint = getFingerprint(
    await curve.getPublicKey(privateKey, true),
  );

  return SLIP10Node.fromExtendedKey({
    privateKey,
    chainCode,
    masterFingerprint,
    depth: 0,
    parentFingerprint: 0,
    index: 0,
    curve: curve.name,
  });
}
```

Dr.-Ing. Mario Heiderich, Cure53
Rudolf Reusch Str. 33
D 10367 Berlin
cure53.de · mario@cure53.de

To mitigate this issue and fully comply with BIP32, Cure53 advises ensuring that the master private key (as an integer) is strictly less than the curve order. In this case, an error should be thrown to allow an application to manage the generation of another mnemonic.

MM-02-005 WP1: Child key derivation function lacks error handling *(Low)*

Whilst evaluating the *key-tree* repository, the observation was made that the function that derives child keys does not handle the case in which the derived key is invalid. In this case, the function *deriveChildKey* would simply fail, which does not comply with the SLIP10 specification.

In particular, the SLIP10 specifications include error handling in this case. SLIP10 establishes that the child key should be re-derived if an invalid key is encountered when deriving a child key with index i, as follows:

key material (32 bytes), child chain code (32 bytes) = HMAC-SHA512(parent chain code, 0x01 || chain code from invalid key || index)

Again, this process exhibits a negligible probability of occurrence, though should still be addressed to ensure compliance with SLIP10. Furthermore, this would prevent an application from consuming the library via error management when this can simply be made transparently.

Affected file:
key-tree/src/derivers/bip32.ts

Affected code:

```
export async function deriveChildKey({
  path,
  node,
  curve = secp256k1,
}: DeriveChildKeyArgs): Promise<SLIP10Node> {
  assert(typeof path === 'string', 'Invalid path: Must be a string.');

  if (node.privateKeyBytes) {
    [...]
    const { privateKey, chainCode } = await generateKey({
      privateKey: node.privateKeyBytes,
      chainCode: node.chainCodeBytes,
      secretExtension,
      curve,
    });
```

Dr.-Ing. Mario Heiderich, Cure53
Rudolf Reusch Str. 33
D 10367 Berlin
cure53.de · mario@cure53.de

```
    [...]
  }

  [...]
  const { publicKey, chainCode } = generatePublicKey({
    publicKey: node.compressedPublicKeyBytes,
    chainCode: node.chainCodeBytes,
    publicExtension,
    curve,
  });

  [...]
}
```

To mitigate this issue, Cure53 recommends deriving a new child key (as established in SLIP10) in the event an error is encountered due to the derivation of an invalid key. This should be easily resolved with a try/catch block.

MM-02-006 WP1: Code fails to reject some invalid extended keys *(Medium)*

Whilst examining the *key-tree* repository, the observation was made that the *fromExtendedKey* method of the *SLIP10Node* class fails to reject certain types of keys with invalid properties. In fact, some invalid keys are accepted from the BIP32 specification[1] test vectors.

As deducible in the code snippet, the method only checks for non-negative index, depth, and parent fingerprint. However, these checks are insufficiently exhaustive since impossible relations between depth and index, depth and parent fingerprint, and depth and master fingerprint are not investigated.

In particular, valid keys with the following invalid properties (hence invalid extended keys) would be accepted when they should be rejected:

- A valid key with zero depth and non-zero parent fingerprint.
- A valid key with zero depth and non-zero index.
- A valid key with non-zero depth and zero parent fingerprint.
- A valid key with depth >=2 and parentFingerprint = masterFingerprint.

Notably, the *BIP44Node* classes' *fromExtendedKey* method (which calls *decodeFromExtendedKey*) and *BIP44CoinTypeNode* share this same issue.

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

Dr.-Ing. Mario Heiderich, Cure53
Rudolf Reusch Str. 33
D 10367 Berlin
cure53.de · mario@cure53.de

The impact of accepting these invalid key families (and others) entirely depends on how the library is leveraged by the application in question. Nevertheless, Cure53 recommends ensuring that invalid keys of this nature are rejected to avoid any associated vulnerabilities, such as a key usage of a different type.

Affected file:
key-tree/src/SLIP10Node.ts

Affected code:

```
  static async fromExtendedKey({
    depth,
    masterFingerprint,
    parentFingerprint,
    index,
    privateKey,
    publicKey,
    chainCode,
    curve,
  }: SLIP10ExtendedKeyOptions) {
    const chainCodeBytes = getBytes(chainCode, BYTES_KEY_LENGTH);

    validateCurve(curve);
    validateBIP32Depth(depth);
    validateBIP32Index(index);
    validateParentFingerprint(parentFingerprint);

    if (privateKey) {
      const privateKeyBytes = getBytes(privateKey, BYTES_KEY_LENGTH);

      return new SLIP10Node({
        depth,
        masterFingerprint,
        parentFingerprint,
        index,
        chainCode: chainCodeBytes,
        privateKey: privateKeyBytes,
        publicKey: await getCurveByName(curve).getPublicKey(privateKeyBytes),
        curve,
      });
    }

    if (publicKey) {
      const publicKeyBytes = getBytes(
        publicKey,
        getCurveByName(curve).publicKeyLength,
      );

      return new SLIP10Node({
```

Dr.-Ing. Mario Heiderich, Cure53
Rudolf Reusch Str. 33
D 10367 Berlin
cure53.de · mario@cure53.de

```
        depth,
        masterFingerprint,
        parentFingerprint,
        index,
        chainCode: chainCodeBytes,
        publicKey: publicKeyBytes,
        curve,
      });
    }

    throw new Error(
      'Invalid options: Must provide either a private key or a public key.',
    );
  }
```

To mitigate this issue, Cure53 recommends including exhaustive checks to reject these invalid key families. In addition, the developer team should thoroughly check that all invalid extended keys from the reference test vectors are indeed rejected.

Dr.-Ing. Mario Heiderich, Cure53
Rudolf Reusch Str. 33
D 10367 Berlin
cure53.de · mario@cure53.de

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, whilst a vulnerability is present, an exploit may not always be possible.

MM-02-003 WP1: Master key derivation lacks BIP32 compliance *(Low)*

Whilst reviewing the *key-tree* repository, the observation was made that *createBip39KeyFromSeed* allows generating the master key from a seed that is not guaranteed to meet the length requirements from the BIP32 specification, which establishes that the seed length should comprise between 128 and 512 bits.

Cure53 would like to underline that this function is leveraged correctly at all times in the *key-tree* repository with an input seed that is generated according to BIP39 via the call to *mnemonicToSeed*, hence satisfying the length requirements. However, since consumers can still access this method by importing *metamask/key-tree/derivation* (although its use is strongly discouraged in the documentation), one can recommend including a check for the seed length, as this could lead to the generation of a weak master key (and thus all keys) if used incorrectly.

Affected file:
key-tree/src/derivers/bip39.ts

Affected code:

```
export async function createBip39KeyFromSeed(
  seed: Uint8Array,
  curve: Curve = secp256k1,
): Promise<SLIP10Node> {
  const key = hmac(sha512, curve.secret, seed);
  const privateKey = key.slice(0, 32);
  const chainCode = key.slice(32);

  const masterFingerprint = getFingerprint(
    await curve.getPublicKey(privateKey, true),
  );

  return SLIP10Node.fromExtendedKey({
    privateKey,
    chainCode,
    masterFingerprint,
    depth: 0,
    parentFingerprint: 0,
    index: 0,
```

Dr.-Ing. Mario Heiderich, Cure53
Rudolf Reusch Str. 33
D 10367 Berlin
cure53.de · mario@cure53.de

```
    curve: curve.name,
  });
}
```

To mitigate this issue, Cure53 advises integrating a check to ensure that the length of the seed resides in the expected range (128 to 512 bits) and ensuring that the master private key is valid.

MM-02-007 WP1: All-zero private key treated as invalid for ed25519 *(Info)*

Whilst reviewing the *key-tree* repository, the observation was made that the private key consisting of the all-zero vector is considered invalid, irrespective of the curve choice. However, a private key of this nature remains valid for the ed25519 curve.

This behavior occurs with extremely low probability and does not negatively influence security. As such, this ticket merely serves for greater transparency and full compliance with SLIP10.

Dr.-Ing. Mario Heiderich, Cure53
Rudolf Reusch Str. 33
D 10367 Berlin
cure53.de · mario@cure53.de

Conclusions

The impressions gained during this report - which details and extrapolates on all findings identified during the CW08 2023 testing against the MetaMask key-tree interface by the Cure53 team - will now be discussed at length. To summarize, the confirmation can be made that the components under scrutiny have garnered a relatively strong impression; evident security strengths were observed, though some minor vulnerabilities were detected, as follows.

In context, a hierarchical deterministic wallet eases the use of a unique private/public keypair for each transaction, which benefits privacy and security in the blockchain realm.

The hierarchical deterministic wallet exhibits a twofold raison d'etre: firstly, the existence of a seed from which all keys are derivable; and secondly, the fact that any given key can be used to derive other keys from it, allowing for a defined key hierarchy.

Specifications related to the realization of hierarchical deterministic wallets are BIP32, SLIP10 (which represents a generalization of BIP32 to add support for more elliptic curves), BIP39, and BIP44. BIP32 and SLIP10 describe the algorithm to derive the master key from a seed and child keys from a parent, and introduce a notation to identify a child key from their derivation path (this can be seen as the path from the master to the child key). BIP39 purports a user-friendly method by which to derive a seed from a mnemonic code, whilst BIP44 describes a standard logical hierarchy that can be leveraged.

MetaMask's key-tree interface implements key creation for any level of a valid SLIP10 or BIP44 derivation path, including support for the ed25519 curve only (though not for NIST P-256).

In summation, Cure53 observed that the MetaMask interface offers relatively solid security measures on the whole. The majority of vulnerabilities detected pertain to the lack of insufficient checks in order to verify the validity of the master key derived from the seed or the child keys derived from a parent. Moreover, the all-zero vector private key is treated as invalid for the ed25519 curve, which is an incorrect procedure. All of these issues are formally incorrect but do not incur any tangible risk in practice due to the extremely low probability of occurrence. However, they still constitute formal issues that should be addressed to ensure adherence to the aforementioned specifications.

The most relevant issue is documented under ticket MM-02-006, since a method accepts certain families of invalid extended keys when they should be rejected. This is persisted because correlations between depth, index, and parent fingerprint - which are all properties of extended keys - have not been taken into consideration.

Dr.-Ing. Mario Heiderich, Cure53
Rudolf Reusch Str. 33
D 10367 Berlin
cure53.de · mario@cure53.de

Cure53 would also like to underline that the function responsible for child key generation relies on the aforementioned invalid extended key instantiation method. As such, one can strongly recommend resolving this issue as soon as possible to avoid unexpected attacks.

All in all, following the completion of this review, Cure53 is pleased to confirm that the MetaMask key-tree interface adheres to relatively robust security standards. This viewpoint is corroborated by the lack of any serious cryptography-related issues raised during this audit, as well as the absence of any suboptimal implementation choices that may facilitate generation of child keys with unexpected derivation paths or invalid properties, or abuse of the child keys derivation function for malicious purposes.

Cure53 would like to thank Christian Montoya, Shuyao Kong, Maarten Zuidhoorn, Olaf Tomalka, and Erik Marks from the ConsenSys Software Inc. team for their excellent project coordination, support, and assistance, both before and during this assignment.