Keyfork Security Assessment

Distrust

Version 1.0 – June 4, 2024

©2024 – NCC Group

Prepared by NCC Group Security Services, Inc. for Distrust. Portions of this document and the templates used in its production are the property of NCC Group and cannot be copied (in full or in part) without NCC Group's permission.

While precautions have been taken in the preparation of this document, NCC Group the publisher, and the author(s) assume no responsibility for errors, omissions, or for damages resulting from the use of the information contained herein. Use of NCC Group's services does not guarantee the security of a system, or that computer intrusions will not occur.

Prepared By
Parnian Alimi
Elena Bakos Lang
Kevin Henry

Prepared For
Ryan Heywood
Lance Vick

1 Executive Summary

Synopsis

In April 2024, Distrust engaged NCC Group's Cryptography Services team to perform a cryptographic security assessment of *keyfork*, described as "*an opinionated and modular toolchain for generating and managing a wide range of cryptographic keys offline and on smartcards from a shared BIP-0039 mnemonic phrase*". The tool is intended to be run on an air-gapped system and allows a user to split or recover a cryptographic key using Shamir Secret Sharing, with shares imported and exported using mechanisms such as mnemonics or QR codes. These shares can be managed by one or more users, with a defined threshold of shares required to recover the original secret. The review was performed by 3 consultants over 2 calendar weeks with a total effort of 20 person-days. A retest was conducted in May 2024, which resulted in all findings and notes being marked *Fixed*.

Scope

The review targeted the *keyfork* repository at https://git.distrust.co/public/keyfork. The formal target is tagged release `keyfork-v0.1.0`, however the review also included commits up to the current (at the time of review) `main` branch at commit `089021a` as they did not meaningfully impact the scope. The review was further guided by the security model documented in *security.md*. Distrust also indicated that memory-related (e.g., zeroization) and timing-related attacks were not a concern due to the trusted nature of the hardware and its environment, and as such were not investigated. Retesting did not include any new functionality added that was not in direct response to a finding or note in this report.

Limitations

Overall good coverage of the in-scope code was achieved. However, the reviewed version of the library has some core features currently left as `todo!()` items in the *CLI*, including OpenPGP `discover()` and `provision()` functions, handling of shards *not* using OpenPGP (e.g., P256), and mnemonics using entropy from sources other than the system (e.g., cards, dice).

Key Findings

The assessment uncovered a number of low impact findings along with notes and comments captured in the section Engagement Notes. These include:

- Finding "Encrypting Shards for Transport Leaks Secret Length" and some related notes in the Engagement Notes summarize an information leak and potential optimizations for shard encryption during transport.
- Finding "Non-Standard BIP-0032 Derivation" identifies both missing and extraneous checks that are mandated by BIP-0032, but which only pose problems with negligible probability.
- Finding "Manipulating System Time Allows Unlimited QR Scanning Retries" proposes a small change to prevent certain clock-based attacks from circumventing a timeout mechanism.

After retesting, NCC Group found that Distrust had addressed all findings and notes in this report, with each now marked as *Fixed*.

Strategic Recommendations

- Consider catching and handling errors gracefully in all cases instead of defaulting to panics, thereby allowing more appropriate feedback to be provided to the user when error occurs.
- Ensure that any `todo!()` macros and annotations are addressed or reflected in the documentation such that a user of the library is not misled.

2 Dashboard

Target Data

Name	Keyfork
Type	Standalone Application
Platforms	Rust
Environment	Local

Engagement Data

Type	Cryptographic Security Assessment
Method	Code-assisted
Dates	2024-04-01 to 2024-04-12
Consultants	3
Level of Effort	20 person-days

Targets

Keyfork https://git.distrust.co/public/keyfork/src/tag/keyfork-v0.1.0

"an opinionated and modular toolchain for generating and managing a wide range of cryptographic keys offline and on smartcards from a shared BIP-0039 mnemonic phrase"

Finding Breakdown

Critical issues	0
High issues	0
Medium issues	0
Low issues	5
Informational issues	1
Total issues	**6**

Category Breakdown

Cryptography	4
Data Validation	1
Patching	1

Component Breakdown

keyfork	2
keyfork-derive-util	1
keyfork-qrcode	1
keyfork-shard	1
keyforkd	1

Critical High Medium Low Informational

3 Table of Findings

For each finding, NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors.

Title	Status	ID	Risk
Encrypting Shards for Transport Leaks Secret Length	Fixed	UGV	Low
Non-Hardened Derivation at Top Level of Hierarchical Wallet	Fixed	9YA	Low
Manipulating System Time Allows Unlimited QR Scanning Retries	Fixed	KJG	Low
Incorrect Path Used In Hierarchical Key Derivation	Fixed	PLK	Low
Non-Standard BIP-0032 Derivation	Fixed	6FU	Low
Vulnerable and Outdated Dependencies	Fixed	V94	Info

4 Finding Details

Low

Encrypting Shards for Transport Leaks Secret Length

Overall Risk	Low	**Finding ID**	NCC-E010467-UGV
Impact	Low	**Component**	keyfork-shard
Exploitability	Low	**Category**	Cryptography
		Status	Fixed

Impact

Padding and length fields are not part of the encrypted/authenticated payload. An attacker may arbitrarily modify padding bytes without detection and may cause confusion by triggering unexpected errors on the receiving end.

Description

The function `decrypt_one_shard_for_transport()` decrypts a shard, establishes a Diffie-Hellman-derived key with the recipient, and then encrypts the shard using AES-GCM and the derived key such that the shard can be sent to the recipient. The encrypted payload is padded to 64 bytes such that it may be represented as a standard mnemonic. This represents both an upper bound and the target length for the encoded payload:

```
// 256 bit share encrypted is 49 bytes, couple more bytes before we reach max size
const ENC_LEN: u8 = 4 * 16;
```

Figure 1: keyfork-shard/src/lib.rs

The to-be-encrypted payload consists of the following information:

- 1 byte `version`
- 1 byte `threshold`
- 33 byte `Share` (1 byte for the x coordinate, 32 bytes for its evaluation).

When including the authentication tag after encryption, this represents an expected total of 51 bytes. To prepare the final output, the `out_bytes` vector is initialized such that the last byte contains the length of the encrypted payload, and the first bytes are populated with the encrypted payload. This vector is then padded using the following:

```
        out_bytes[..payload_bytes.len()].clone_from_slice(&payload_bytes);

        // NOTE: This previously used a single repeated value as the padding byte, but
        ↳ resulted in
        // difficulty when entering in prompts manually, as one's place could be lost due
        ↳ to
        // repeated keywords. This is resolved below by having sequentially increasing
        ↳ numbers up to
        // but not including the last byte.
        #[allow(clippy::cast_possible_truncation)]
        for (i, byte) in (out_bytes[payload_bytes.len()..(ENC_LEN as usize - 1)])
            .iter_mut()
```

```
            .enumerate()
        {
            *byte = (i % u8::MAX as usize) as u8;
        }
```

Figure 2: keyfork-shard/src/lib.rs

Thus, the final 64-byte message consists of:

- 35+16 = 51 byte encrypted payload
- 12 byte padding
- 1 byte length of encrypted payload.

The key observation here is that *the padding and length fields are not included in the authenticated ciphertext* and are therefore not confidential nor guaranteed to be authentic. This does not appear to introduce any meaningful attack, but it does result in potentially unwanted or unintuitive behavior:

- The padding is malleable. An attacker may alter the padding without affecting the correctness of decryption.
- The length field can be modified, which can cause the program to panic due to out of bounds errors rather than decryption errors.

It is understood that the above messages are assumed to be tamperproof, and that under the expected use cases the size of a secret share is a fixed constant. However, small modifications to the approach would constrain the above behavior and hide the length of the secret if it is ever changed from the current value of 32 bytes.

Instead of padding the *ciphertext* to 63 bytes (plus 1 length byte), one could instead pad the *plaintext* to 47 bytes (plus 1 length byte). This would result in an AES-GCM ciphertext that is exactly `ENC_LEN` (64 bytes), such that the length field is encrypted and *any* modification to the encoded ciphertext will always result in a decryption failure, instead of other deserialization errors. Such an approach could be viewed as being strictly stronger than the existing approach, and does not introduce any overhead.

Alternatively, it appears as though the expected padding length is precisely the length of the AES-GCM nonce. One could instead prepend the nonce to the ciphertext, as is commonly done, and achieve a ciphertext that is exactly the desired length. The Engagement Notes section provides additional commentary on the usage of nonces within this process.

Recommendation

Consider padding the *plaintext* bytes to `ENC_LEN - TAG_SIZE` bytes such that the padding and length bytes are included in the authenticated ciphertext.

Location

keyfork-shard/src/lib.rs

Retest Results

2024-05-06 – Fixed

Commit *6fa434e* updated the encryption process to pad the *plaintext* up to a pre-defined fixed length of 36 bytes. Subsequently, commit *1a036a0* revised some code comments and error messages to provide additional clarity.

The revised approach will pad a plaintext value of *up to* 32 bytes with a value representing the length of the `version | threshold | index | secret` bytes input. The chosen length of

36 bytes ensures that there is always at least one byte available to store the length. This change results in a 52-byte AES-GCM ciphertext for which any modification will be detectable. As such, this finding is considered *Fixed*.

Low

Non-Hardened Derivation at Top Level of Hierarchical Wallet

Overall Risk	Low	**Finding ID**	NCC-E010467-9YA
Impact	High	**Component**	keyforkd
Exploitability	Low	**Category**	Cryptography
		Status	Fixed

Impact

If non-hardened derivation is used, an attacker may be able to recover the "general" or "master" key for a hierarchical wallet stored in *keyforkd*.

Description

The *keyforkd* backend can be used to derive hierarchical wallet keys from a master secret, according to the process defined in the BIP-0032 standard. This derivation function accepts a wide range of inputs, with only minimal validation:

```
By default, the only validation provided for the request is to ensure the
request contains two indices. By requiring this, `keyforkd` can ensure the
master key is not leaked, and "general" keys (such as `m/44'`, see [BIP-0044])
are not leaked. In the future, `keyforkd` could implement GUI or TTY approval
for users to approve the path requested by the client, such as `m/44'/0'` being
"Bitcoin", or `m/7366512'` being "OpenPGP".
```

Figure 3: *docs/src/bin/keyforkd.md*

This validation is implemented in the `call()` function:

```
fn call(&mut self, req: Request) -> Self::Future {
    let seed = self.seed.clone();
    match req {
        Request::Derivation(req) => Box::pin(async move {
            let len = req.path().len();
            if len < 2 {
                return Err(DerivationError::InvalidDerivationLength(len).into());
            }
```

Figure 4: *daemon/keyforkd/src/service.rs*

However, note that this validation does not restrict the first two levels to use hardened derivation. The use of non-hardened derivation can in some cases lead to the recovery of higher-level private keys in a hierarchical wallet[1]. In particular, knowledge of a private child key and of the public parent key can be used by a potential adversary to recover the private parent key.

Indeed, given an extended private parent key (k_{par}, c_{par}), the corresponding public parent key is given by $K_{par} = (k_{par} \cdot G, c_{par})$, where G is the standard generator for the secp256k1 curve. The non-hardened child keys for index i can then be computed as follows:

- Compute $i_L || i_R$ = HMAC-SHA512(Key=c_{par}, Data = $0x00 || k_{par} \cdot G || i$) = HMAC-SHA512(Key=$c_{par}$, Data = $0x00 || K_{par} || i$)

1. https://medium.com/@blainemalone01/hd-wallets-why-hardened-derivation-matters-89efcdc71671

- The extended private child key is then $(k_i, c_i) = (i_L + k_{par} \mod n, i_R)$
- The extended public child key is then $(K_i, c_i) = (k_i \cdot G, i_R)$

An adversary that learns the extended parent public key and the extended private child key can thus re-compute $i_L \| i_R$ = HMAC-SHA512(Key=c_{par}, Data = $0x00 \| K_{par} \| i$), and recover $k_{par} = k_i - i_L \mod n = (i_L + k_{par}) - i_L \mod n$.

As such, if non-hardened derivation is used at the first two levels, an adversary may be able to recover one of the "general" keys or the "master" key, by recovering a less-protected private child key and a public key corresponding to the "general" key or the "master" key. Mandating hardened derivation at the first two levels of the hierarchical wallet, as is done in other hierarchical wallet standards such as BIP-0044, would prevent this attack, and would provide stronger security guarantees for the "general" and "master" keys of the wallet.

Recommendation

Determine whether the system anticipates use-cases that require non-hardened derivation at the top two levels of the hierarchical wallet. If such use-cases are not expected, consider requiring two levels of *hardened* derivation for all key derivation requests in *keyforkd*.

Location

daemon/keyforkd/src/service.rs

Retest Results

2024-05-03 – Fixed

Commit *40551a5* added a check to ensure the first two levels of derivation are hardened, along with tests to validate this behavior. As such, this finding is considered *Fixed*.

Low

Manipulating System Time Allows Unlimited QR Scanning Retries

Overall Risk	Low	**Finding ID**	NCC-E010467-KJG
Impact	Medium	**Component**	keyfork-qrcode
Exploitability	Medium	**Category**	Data Validation
		Status	Fixed

Impact

An on-path attacker that is able to manipulate system time can disable the QR scanner's timeout mechanism.

Description

The *keyfork-qrcode* crate's `scan_camera()` implementations (enabled by `decode-backend-rqrr` or `decode-backend-zbar` features) rely on Rust's *SystemTime* crate to enforce a timeout on the time that is spent scanning camera images for a valid QR code:

```
let start = SystemTime::now();

while SystemTime::now()
    .duration_since(start)
    .unwrap_or(Duration::from_secs(0))
    < timeout
{
...
}
```

Figure 5: qrcode/keyfork-qrcode/src/lib.rs

The `duration_since()` API will fail when the second call to the *SystemTime*'s `now()` yields an earlier time than `start`. In this case, the `unwrap_or()` call defaults to 0, which makes the loop condition true. Anomalies such as adjusting the system time backwards accidentally or by an on-path attacker will disable the timeout mechanism temporarily or permanently. This will give an attacker unlimited retries.

Recommendation

The *SystemTime* documentation recommends using `Instant` to measure elapsed time without the risk of failure.

Alternatively, a default larger than `timeout` can be used to exit the loop in case of a *SystemTime* failure.

Location

qrcode/keyfork-qrcode/src/lib.rs

Retest Results

2024-05-03 – Fixed

Commit *fa125e7* implements the recommended change of using `Instant` over `SystemTime`. As such, this finding is considered *Fixed*.

Low

Incorrect Path Used In Hierarchical Key Derivation

Overall Risk	Low	**Finding ID**	NCC-E010467-PLK
Impact	Low	**Component**	keyfork
Exploitability	High	**Category**	Cryptography
		Status	Fixed

Impact

The *keyfork* crate will generate a key using a path different from the one expected by users, which may lead to confusion or misplaced funds.

Description

The *keyfork* crate describes a process to generate a 256-bit seed, derive OpenPGP keys using the seed, provision smart cards using the derived keys, and export a Shard file. As part of this process, a subkey is derived from the master entropy using the BIP-0032 hierarchical wallet derivation methods:

```
3. The seed is then derived using BIP-0032 along the path `m / pgp' / shrd' /
   index'`, where the values "pgp" and "shrd" converted to bytes and cast to a
   32 bit integer, and the "index" is a numeric iterator `0..max`. BIP-0032
   uses HmacSha512 with the "chain code" of the previous depth, the private-key
   bytes of the current extended private key, and the index, to derive a new
   extended private key and a new chain code.
```

Figure 6: docs/src/bin/keyfork/wizard/index.md

This portion of the derivation is implemented in the `derive_key()` function in the *keyfork* crate, excerpted below:

```
    let mut pgp_u32 = [0u8; 4];
    pgp_u32[1..].copy_from_slice(&"pgp".bytes().collect::<Vec<u8>>());
    let chain = DerivationIndex::new(u32::from_be_bytes(pgp_u32), true)?;
    let mut shrd_u32 = [0u8; 4];
    shrd_u32[..].copy_from_slice(&"shrd".bytes().collect::<Vec<u8>>());
    let account = DerivationIndex::new(u32::from_be_bytes(pgp_u32), true)?;
    let subkey = DerivationIndex::new(u32::from(index), true)?;
    let path = DerivationPath::default()
        .chain_push(chain)
        .chain_push(account)
        .chain_push(subkey);
    let xprv = XPrv::new(seed).derive_path(&path)?;
```

Figure 7: keyfork/src/cli/wizard.rs

However, note that the actual path used for the derivation will be `m / pgp' / pgp' / index'`, not `m / pgp' / shrd' / index'` as expected. This may lead to users of the library to use keys located at a different index than they were expecting, potentially leading to confusion or misplaced funds.

Recommendation

Update the `derive_key()` function to use the derivation path described in the documentation.

Client Confidential

Location

- *keyfork/src/cli/wizard.rs*

Retest Results

2024-05-03 – Fixed

Commit *cdf4015* updated the derivation to use the correct parameter. As such, this finding is considered *Fixed*.

Client Confidential

Low

Non-Standard BIP-0032 Derivation

Overall Risk	Low	**Finding ID**	NCC-E010467-6FU
Impact	Low	**Component**	keyfork-derive-util
Exploitability	None	**Category**	Cryptography
		Status	Fixed

Impact

Deviating from the BIP-0032 standard during hierarchical key derivation may result in insecure keys, or incompatible behavior with other libraries implementing BIP-0032.

Description

The BIP-0032 standard defines methods for deriving a collection of secp256k1 private and public key pairs as part of a hierarchical deterministic wallet based on a single master secret. In particular, this standard defines a method for deriving a master key from a mnemonic seed by generating two 32-byte sequences, I_L and I_R from the mnemonic seed, and interpreting them as the master secret key and master chain code respectively. This method documents the following invalid sequence values:

> In case $parse_{256}(I_L)$ is 0 or $parse_{256}(I_L) \geq n$, the master key is invalid.

However, in the implementation of this functionality in the `new()` function for the `ExtendedPrivateKey` class, the range of the `private_key` variable, corresponding to the sequence I_L, is not checked:

```
    pub fn new(seed: impl as_private_key::AsPrivateKey) -> Self {
        Self::new_internal(seed.as_private_key())
    }

    fn new_internal(seed: &[u8]) -> Self {
        let hash = HmacSha512::new_from_slice(&K::key().bytes().collect::<Vec<_>>())
            .expect(bug!("HmacSha512 InvalidLength should be infallible"))
            .chain_update(seed)
            .finalize()
            .into_bytes();
        let (private_key, chain_code) = hash.split_at(KEY_SIZE / 8);

        Self::new_from_parts(
            private_key
                .try_into()
                .expect(bug!("KEY_SIZE / 8 did not give a 32 byte slice")),
            0,
            // Checked: chain_code is always the same length, hash is static size
            chain_code.try_into().expect(bug!("Invalid chain code length")),
        )
    }
```

Figure 8: derive/keyfork-derive-util/src/extended_key/private_key.rs

Note that this function is also used within the *keyfork-derive-util* crate by the ed25519 master key generation as per the SLIP-0010 standard, which does not have any invalid sequence values. This function is thus correctly implemented for that use-case.

As a related issue, the BIP-0032 standard also defines a method for deriving child private keys from parent private keys. As part of this process, two 32-byte sequences, I_L and I_R are computed from the parent key, which are then used to compute the child key. The following invalid sequence values are documented for this method:

> In case $parse_{256}(I_L) \geq n$ or $k_i = 0$, the resulting key is invalid, and one should proceed with the next value for i. (Note: this has probability lower than 1 in 2^{127}.)

These checks are implemented for the curve secp256k1 in the function `derive_child()`:

```
    fn derive_child(&self, other: &PrivateKeyBytes) -> Result<Self, Self::Err> {
        if other.iter().all(|n| n == &0) {
            return Err(PrivateKeyError::NonZero);
        }
        let other = *other;
        // Checked: See above nonzero check
        let scalar = Option::<NonZeroScalar>::from(NonZeroScalar::from_repr(other.into()))
            .expect(bug!("Should have been able to get a NonZeroScalar"));

        let derived_scalar = self.to_nonzero_scalar().as_ref() + scalar.as_ref();
        Ok(
            Option::<NonZeroScalar>::from(NonZeroScalar::new(derived_scalar))
                .map(Into::into)
                .expect(bug!("Should be able to make Key")),
        )
    }
```

Figure 9: derive/keyfork-derive-util/src/private_key.rs

However, the function `derive_child()` additionally rejects the input $I_L == 0$, which is a valid input according to BIP-0032. A similar issue occurs in the `derive_child()` implementation for the derivation of public child keys from public parent keys in *derive/keyfork-derive-util/src/public_key.rs*.

Note that all of these events occur with negligible probability and will likely never be encountered in practice.

Recommendation

Ensure that invalid values are properly handled in BIP-0032 compatible hierarchical key derivation functions.

Location

- *derive/keyfork-derive-util/src/extended_key/private_key.rs*
- *derive/keyfork-derive-util/src/private_key.rs*
- *derive/keyfork-derive-util/src/public_key.rs*

Retest Results

2024-05-06 – Fixed

Commit *1de466c* adds a check that throws an error if the private key is equal to the zero-vector but introduces a potential timing side channel. Commit *de4e98a* revises this check to run in constant time, thereby mitigating the side channel attack.

The above commits also refactored `derive_child()` such that $I_L == 0$ will be correctly accepted. As such, this finding is considered *Fixed*.

Info

Vulnerable and Outdated Dependencies

Overall Risk	Informational	**Finding ID**	NCC-E010467-V94
Impact	None	**Component**	keyfork
Exploitability	None	**Category**	Patching
		Status	Fixed

Impact

Stale dependencies or dependencies with public RUSTSEC advisories may introduce or represent vulnerabilities within an application. Even if RUSTSEC advisories do not apply, failure to respond to advisories may affect the perceived security posture of an application or organization.

Description

This finding is purely informational. No affecting vulnerabilities within the dependency tree were identified.

Rust provides several utilities for managing dependencies, such as `cargo audit`, `cargo outdated`, and `cargo deny`. This informational finding briefly summarizes the output of these tools on the reviewed code.

`cargo audit` results in 3 ***vulnerable*** dependencies and 2 *warnings*. Two of these vulnerabilities are for the same crate, which has been reviewed and added as an exception in *deny.toml*. The remaining vulnerability was posted *after* the `v0.1.0` release targeted in this review.

- `mio 0.8.10`: Tokens for named pipes may be delivered after deregistration (RUSTSEC-2024-0019)
  - This vulnerability is specific to Windows and ***does not*** affect *keyfork*.
- `rsa 0.8.2`, `rsa 0.9.6`: Marvin Attack: potential key recovery through timing sidechannels (RUSTSEC-2023-0071)
  - The RSA algorithm is included due to smart card dependencies but is not used within *keyfork*. Therefore, this vulnerability ***does not*** affect *keyfork*.
- `yaml-rust 0.4.5` (Unmaintained) and `iana-time-zone 0.1.59` (yanked).

It appears as though RUSTSEC advisories are actively monitored and reviewed, and the only unaddressed vulnerable dependency was published after the release of the reviewed code. It is recommended to update dependencies or add RUSTSEC-2024-0019 to the ignore list with justification.

`cargo outdated` returned several crates with minor revisions, but all direct dependencies were found to be at an otherwise current major revision.

Recommendation

Ensure dependencies are updated and that RUSTSEC-2024-0019 is added to the ignore list, if still applicable, before the next release of *keyfork*.

Location

deny.toml

Retest Results

2024-05-03 – Fixed

Commit *68f07f6* updated the affected package versions such that the recent `cargo audit` hits are cleared. As such, this finding is considered *Fixed*.

5 Finding Field Definitions

The following sections describe the risk rating and category assigned to issues NCC Group identified.

Risk Scale

NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors. The risk rating is NCC Group's recommended prioritization for addressing findings. Every organization has a different risk sensitivity, so to some extent these recommendations are more relative than absolute guidelines.

Overall Risk

Overall risk reflects NCC Group's estimation of the risk that a finding poses to the target system or systems. It takes into account the impact of the finding, the difficulty of exploitation, and any other relevant factors.

Rating	Description
Critical	Implies an immediate, easily accessible threat of total compromise.
High	Implies an immediate threat of system compromise, or an easily accessible threat of large-scale breach.
Medium	A difficult to exploit threat of large-scale breach, or easy compromise of a small portion of the application.
Low	Implies a relatively minor threat to the application.
Informational	No immediate threat to the application. May provide suggestions for application improvement, functional issues with the application, or conditions that could later lead to an exploitable finding.

Impact

Impact reflects the effects that successful exploitation has upon the target system or systems. It takes into account potential losses of confidentiality, integrity and availability, as well as potential reputational losses.

Rating	Description
High	Attackers can read or modify all data in a system, execute arbitrary code on the system, or escalate their privileges to superuser level.
Medium	Attackers can read or modify some unauthorized data on a system, deny access to that system, or gain significant internal technical information.
Low	Attackers can gain small amounts of unauthorized information or slightly degrade system performance. May have a negative public perception of security.

Exploitability

Exploitability reflects the ease with which attackers may exploit a finding. It takes into account the level of access required, availability of exploitation information, requirements relating to social engineering, race conditions, brute forcing, etc, and other impediments to exploitation.

Rating	Description
High	Attackers can unilaterally exploit the finding without special permissions or significant roadblocks.

Rating	Description
Medium	Attackers would need to leverage a third party, gain non-public information, exploit a race condition, already have privileged access, or otherwise overcome moderate hurdles in order to exploit the finding.
Low	Exploitation requires implausible social engineering, a difficult race condition, guessing difficult-to-guess data, or is otherwise unlikely.

Category

NCC Group categorizes findings based on the security area to which those findings belong. This can help organizations identify gaps in secure development, deployment, patching, etc.

Category Name	Description
Access Controls	Related to authorization of users, and assessment of rights.
Auditing and Logging	Related to auditing of actions, or logging of problems.
Authentication	Related to the identification of users.
Configuration	Related to security configurations of servers, devices, or software.
Cryptography	Related to mathematical protections for data.
Data Exposure	Related to unintended exposure of sensitive information.
Data Validation	Related to improper reliance on the structure or values of data.
Denial of Service	Related to causing system failure.
Error Reporting	Related to the reporting of error conditions in a secure fashion.
Patching	Related to keeping software up to date.
Session Management	Related to the identification of authenticated users.
Timing	Related to race conditions, locking, or order of operations.

6 Engagement Notes

This section consists of notes and observations from the review that did not warrant standalone security findings, or that are not security related, but may nevertheless be of interest to the team at Distrust.

Inconsistent Test Infrastructure Visibility

The test structure `TestPrivateKey` has its visibility reduced by the `#[doc(hidden)]` directive, which omits the `TestPrivateKey` structure from the documentation. However, it is extensively used in documentation, for instance in the *keyforkd* client documentation in *daemon/keyforkd-client/src/lib.rs*, which may limit the usefulness of the `#[doc(hidden)]` attribute. Additionally, the public enum element `DerivationAlgorithm.Internal` is exclusively used for tests and is also marked as `#[doc(hidden)]`. As it is still exposed to the public, and only hidden from the documentation, it may be clearer to rename this element to `InternalTestAlgorithm` to align with the nomenclature used for the rest of the test-only functionalities.

Retest Results

Commit *2bca0a1* renamed the `Internal` element to `TestAlgorithm`, as recommended. As such, this note is considered *Fixed*.

Incorrect Documentation

- The `HardenedIndex` error is returned if a hardened derivation index is provided in the parent public key to child public key derivation. However, the comment describing it states that `/// BIP-0032 does not support deriving public keys from hardened private keys`, which is inaccurate, as public keys can be derived from hardened private keys via the standard derivation method for secp256k1 public keys. It may be more accurate to note that BIP-0032 does not support hardened public child key derivation from public parent keys.
- The `derive_child()` functions defined in *derive/keyfork-derive-util/src/private_key.rs* and *derive/keyfork-derive-util/src/public_key.rs* document the following error conditions:

```
/// # Errors
///
/// An error may be returned if:
/// * A nonzero `other` is provided.
/// * An error specific to the given algorithm was encountered.
fn derive_child(&self, other: &PrivateKeyBytes) -> Result<Self, Self::Err>;
```

Figure 10: derive/keyfork-derive-util/src/private_key.rs

However, note that an error is actually returned if an *all-zero* `other` is provided, as documented in the comment `/// For the given algorithm, the private key must be nonzero` describing the corresponding `NonZero` error.

Retest Results

Commit *2bca0a1* corrected the comments identified above. As such, this note is considered *Fixed*.

Potentially Inconsistent Guidance on RNG Generation

It was observed that the `main()` function in the *keyfork-entropy* crate returns an error on input > 256 bits:

```
    assert!(
        bit_size <= 256,
        "Maximum supported bit size is 256, got: {bit_size}"
    );

```

```
    let entropy = keyfork_entropy::generate_entropy_of_size(bit_size / 8)?;
    println!("{}", smex::encode(entropy));

    Ok(())
}
```

Figure 11: util/keyfork-entropy/src/main.rs

Similarly, it was observed that the internal documentation examples call the same underlying function with 64 bytes (512 bits) as the sample parameter:

```
/// Read system entropy of a given size.
///
/// # Errors
/// An error may be returned if an error occurred while reading from the random source.
///
/// # Examples
/// ```rust,no_run
/// # fn main() -> Result<(), Box<dyn std::error::Error>> {
/// # std::env::set_var("SHOOT_SELF_IN_FOOT", "1");
/// let entropy = keyfork_entropy::generate_entropy_of_size(64)?;
/// assert_eq!(entropy.len(), 64);
/// # Ok(())
/// # }
/// ```
pub fn generate_entropy_of_size(byte_count: usize) -> Result<Vec<u8>, std::io::Error> {
```

Figure 12: util/keyfork-entropy/src/lib.rs

There is no error or unsafe behavior exhibited here, but it could be seen as misleading for the example use case of this function to not match the public interface of *key-fork-entropy*. In general, the `generate_entropy_of_size()` function will safely return OS-provided entropy for sizes larger than 32 or 64 bytes. The `main()` function is presumably meant to cap the size of the entropy to the expected current use case of *keyfork*. However, if this assumption is true, it is not clear why values smaller than 32 bytes are supported as well.

Retest Results

Commit *5438f4e* updated the `main()` function to accept a value of 128, 256, or 512 bits, which is consistent with the identified use cases and documentation. As such, this note is considered *Fixed*.

Lack of Input Validation Leads to Unexpected Behavior

The `decrypt_one_shard_for_transport()` function, in *keyfork-shard* crate, uses the recipient's public key with the Diffie-Hellman protocol to generate an AES-GCM key, which encrypts the decrypted shard. The function attempts to read the recipient's public key from a QR code and in case of failure gracefully falls back to reading it as mnemonics via command prompt. It is assumed that the QR code is a valid encoding of a 12 bytes nonce followed by a 32 bytes public key. A shorter QR code will result in a panic when the `decoded_data` slice is parsed with ranges:

```
prompt
    .lock()
    .expect(bug!(POISONED_MUTEX))
    .prompt_message(PromptMessage::Text(QRCODE_PROMPT.to_string()))?;
if let Ok(Some(hex)) =
    keyfork_qrcode::scan_camera(std::time::Duration::from_secs(30), 0)
```

```rust
{
    let decoded_data = smex::decode(&hex)?;
    nonce_data = Some(decoded_data[..12] .try_into().map_err(|_| InvalidData)?);
    pubkey_data = Some(decoded_data[12..] .try_into().map_err(|_| InvalidData)?)
```

Figure 13: keyfork-shard/src/lib.rs

In case of a corrupt QR code, the intent seems to have been to fallback to reading from the command prompt, however lack of validation will result in a panic. Since the `decrypt_one_shard_for_transport()` is used as part of a tool, the workaround is to not use the corrupt QR code in the next attempt. As such this observation is left as a note. Similarly, the `remote_decrypt()` function, in *keyfork-shard* crate, exhibits the same lack of input validation on line 477.

Retest Results

Commit *0fe5301* updated the error handling around failed QR code scans to make the causes of failure more explicit. Furthermore, the identified nonce-parsing panic was removed due to changes in the protocol, as detailed below. As such, this note is considered *Fixed*.

Comments on Nonce Generation and Usage

Keyfork supports mnemonic- or QR-based transportation of encrypted shards. This is triggered by the Receiver of such a shard in the function `remote_decrypt()` and handled by the Sender in function `decrypt_one_shard_for_transport()`. The process is summarized as:

1. Receiver generates an x25519 ephemeral secret key and nonce.
2. Receiver encodes the public key and nonce as a mnemonic / QR code for transport.
3. Sender receives and decodes the public key and nonce.
4. Sender generates an x25519 ephemeral secret key.
5. Sender uses ECDH and HKDF to derive an AES key.
6. Sender re-encrypts shard using newly derived AES key and received nonce.
7. Sender pads the resulting ciphertext and encodes as a mnemonic / QR code.
8. Sender encodes public key as a mnemonic / QR code for transport.
9. Receiver receives and decodes the ciphertext and public key.
10. Receiver uses ECDH and HKDF to derive the AES key.
11. Receiver unpads and decrypts received ciphertext using derived key and stored nonce.

The above protocol differs from most conventional protocols in that the Receiver is responsible for generating the nonce and communicating it to the encrypting party, rather than the nonce being generated by the encrypting party. This does not compromise the security of AES-GCM, as nonces are considered public information. However, it can be seen as increasing the attack surface of the protocol, as malicious modification of the nonce in transit to the encrypting party could result in accidental nonce reuse. However, it should be emphasized that in *keyfork*'s implemented use case, each ephemeral key is only used for a single encryption. A malicious party that can influence the nonce in transit should not be able to influence the Sender's ECDH key derivation, making the chance of nonce-reuse negligible. Furthermore, *keyfork* assumes that QR codes or mnemonics in transit are tamperproof.

Although the current approach to nonce management should be safe under the current threat model, it should be noted that alternate approaches are possible, which may be seen as restricting the attack surface further:

1. The Receiver could generate the nonce at random and transmit it alongside the ciphertext. This matches the usual usage of AES-GCM, where a nonce is freshly generated at random by the encrypting party at the time of encryption. Furthermore, the encrypted response is padded to 64 bytes, and under the current parameter set could accommodate the nonce simply by replacing some of the padding bytes with the nonce. In practice, the nonce is usually prepended to the ciphertext, as it is a fixed length and easily parsed in that manner.
2. The Sender and Receiver could leverage HKDF to derive a deterministic nonce based on the shared key. In short, an additional 12 bytes for the nonce can be derived from HKDF after the key and used as the nonce. In this manner the nonce does not need to be transmitted at all, which slightly simplifies the process. There are two practical approaches that could be considered:
   a. Expand a total of 32+12 = 44 bytes in a single call to `hkdf.expand()` and use disjoint slices of the result as the key and the nonce.
   b. Leverage the `info` parameter to make two different calls to `hkdf.expand()` for the key and the nonce. The `info` parameter must be a distinct value for each call, and is typically chosen to be a descriptor, such as `info = b"key bytes"` and `info = b"nonce bytes"`.

   It remains imperative that a given key + nonce pair are never reused. Therefore, the above recommendations must be reconsidered if this key is ever used for more than a single encryption operation.

Retest Results

Commit *9394500* implements the second recommendation above where the sender and receiver each derive the nonce deterministically using HKDF:

```
        let mut shared_key_data = [0u8; 256 / 8];
        hkdf.expand(b"key", &mut shared_key_data)?;
        let shared_key = Aes256Gcm::new_from_slice(&shared_key_data)?;

        let mut nonce_data = [0u8; 12];
        hkdf.expand(b"nonce", &mut nonce_data)?;
        let nonce = Nonce::<U12>::from_slice(&nonce_data);
```

Figure 14: keyfork-shard/src/lib.rs

The above changes are aligned with the recommendations, and the resulting key and nonce are only ever used once for encryption. As such, this note is considered *Fixed*.

Public Key Validation

It was observed that the ECDH key derivation used during transport encryption (described in the previous section) does not consider maliciously generated public keys, and there is no validation that a provided public key is not the identity element. The *x25519_dalek* crate provides the following function that can be used to ensure a received public key contributed to the ECDH derivation:

```
    #[must_use]
    pub fn was_contributory(&self) -> bool {
        !self.0.is_identity()
    }
```

Figure 15: x25519_dalek/x25519.rs

The Receiver is implicitly trusted, which means their input to the ECDH process should always be contributory. In transit, the public key is assumed to be tamperproof. The Sender has knowledge of the shard, and malicious behavior on their part can always leak the secret. Therefore, it is only with negligible probability that honest shares could be non-contributory, thereby leading to a leak of the derived AES key. In practice, this risk can effectively be ignored.

Nevertheless, it could be seen as a defense-in-depth measure to explicitly validate received public keys to ensure they are not the identity point and are therefore providing a contribution to the ECDH derivation.

Retest Results

Commit *c0b19e2* updated the ECDH process to fail if `was_contributory()` returns false. As such, this note is considered *Fixed*.

Potential Integer Overflow in User Prompt

Keyfork supports placing each individual shard onto multiple smart cards, presumably for the purposes of backup/redundancy. The prompt displayed to a user chooses to translate from the internal 0-based index to a 1-based index system:

```
            pm.prompt_message(Message::Text(format!(
                "Please remove all keys and insert key #{} for user #{}",
                i + 1,
                index + 1,
            )))?;
```

Figure 16: keyfork/src/cli/wizard.rs

However, because `index` and `i` are both `u8` values, if either is equal to `u8::MAX` (255) then an overflow will occur. In a debug build, this will cause the program to panic. In a release build, this will cause the value wrap back around to 0. This may be unintuitive or unexpected by the user but does not represent a security issue in practice. However, one could ensure the outputs of these operations are represented as a larger type to ensure the user prompts are consistent, such as using `i as u16 + 1` and `index as u16 + 1`.

Retest Results

Commit *289cec3* adds the recommended cast to `u16`. As such, this note is considered *Fixed*.

Inaccurate Code Comment

The default PGP certificate expiration is one day, but this can be overridden by an environment variable.

```
    // Set certificate expiration to one day
    let mut keypair = primary_key.clone().into_keypair()?;
    let signatures =
        cert.set_expiration_time(&policy, None, &mut keypair, Some(expiration_date))?;
    let cert = cert.insert_packets(signatures)?;
```

Figure 17: derive/keyfork-derive-openpgp/src/lib.rs

Therefore, the comment here is only correct when the environment variable is not set or is set to exactly one day. A similar comment appears elsewhere:

```
pub enum DeriveSubcommands {
    /// Derive an OpenPGP Transferable Secret Key (private key). The key is encoded using
    ↪ OpenPGP
    /// ASCII Armor, a format usable by most programs using OpenPGP.
    ///
    /// The key is generated with a 24-hour expiration time. The operation to set the
    ↪ expiration
    /// time to a higher value is left to the user to ensure the key is usable by the user.
    #[command(name = "openpgp")]
    OpenPGP {
        /// Default User ID for the certificate, using the OpenPGP User ID format.
        user_id: String,
    },
}
```

Figure 18: keyfork/src/cli/derive.rs

These comments could be updated to specify this as a default time, or to reference the environment variable as an override.

Retest Results

Commit *f0e5ae9* revised the documentation to reference the `KEYFORK_OPENPGP_EXPIRE` environment variable. As such, this note is considered *Fixed*.

Error Handling During Mnemonic Generation

The *keyfork-mnemonic-util* crate implements the mnemonic sentence generation approach defined in BIP-0039. Additionally, this crate provides custom methods to extend this functionality to a much wider range of entropy values:

```
    /// Create a Mnemonic using an arbitrary length of given data. The length does not
    ↪ need to
    /// conform to BIP-0039 standards, but should be a multiple of 32 bits or 4 bytes.
```

Figure 19: util/keyfork-mnemonic-util/src/lib.rs

An example of an incorrect use of this functionality is provided further in the documentation:

```
    /// If given an invalid length, undefined behavior may follow, or code may panic.
    ///
    /// ```rust,should_panic
    /// use keyfork_mnemonic_util::Mnemonic;
    /// use std::str::FromStr;
    ///
    /// // NOTE: Data is of invalid length, 31
    /// let data = b"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA";
```

```
    /// let mnemonic = unsafe { Mnemonic::from_raw_bytes(data.as_slice()) };
    /// let mnemonic_text = mnemonic.to_string();
    /// // NOTE: panic happens here
    /// let new_mnemonic = Mnemonic::from_str(&mnemonic_text).unwrap();
    /// ```
```

Figure 20: *util/keyfork-mnemonic-util/src/lib.rs*

However, as shown in this example, malformed data will not be detected during mnemonic generation, and will simply result in a mnemonic with a malformed checksum. In particular, the `mnemonic_text` variable will be generated successfully, and only parsing it using `from_str()` will return an `InvalidChecksum` error, which results in a panic during the attempt to `unwrap()`.

As users of this library may not attempt to parse a newly generated mnemonic immediately after creation, consider detecting invalid entropy lengths during mnemonic generation and returning an error instead of an invalid mnemonic.

Retest Results

Commit *6a265ad* added `AssertValidMnemonicSize::<N>::OK_CHUNKS()` to validate that the mnemonic length is a multiple of 32 bits for mnemonics created via `from_nonstandard_bytes()` and added explicit checks for the same in `from_raw_bytes()`, along with tests to ensure that this is enforced. As such, this note is considered *Fixed*.

Undocumented Upper Limit For Custom Mnemonic Implementation

The mnemonic sentence generation for non-standard entropy values is implemented in the `words()` function:

```
395     /// Encode the mnemonic into a list of integers 11 bits in length, matching the length
        ↪ of a
396     /// BIP-0039 wordlist.
397     pub fn words(&self) -> Vec<usize> {
398         let bit_count = self.data.len() * 8;
399         println!("{:?}", bit_count);
400         let mut bits = vec![false; bit_count + bit_count / 32];
401
402         for byte_index in 0..bit_count / 8 {
403             for bit_index in 0..8 {
404                 bits[byte_index * 8 + bit_index] =
405                     (self.data[byte_index] & (1 << (7 - bit_index))) > 0;
406             }
407         }
408
409         let mut hasher = Sha256::new();
410         hasher.update(&self.data);
411         let hash = hasher.finalize().to_vec();
412         for check_bit in 0..bit_count / 32 {
413             bits[bit_count + check_bit] = (hash[check_bit / 8] & (1 << (7 - (check_bit % 8))
            ↪ )) > 0;
414         }
```

Figure 21: *util/keyfork-mnemonic-util/src/lib.rs*

However, calling this function on entropy that is longer than 1024 bytes (8192 bits) will cause a panic on line 412, as the function will attempt to include a checksum that is longer than the 256 bits of the SHA-256 hash. Consider either documenting and ensuring that the data provided is shorter than this limit, or designing an alternate checksum approach if the checksum length is above 256 bits.

Retest Results

Commit *6a265ad* added `AssertValidMnemonicSize::<N>::OK_SIZE()` to validate that the mnemonic length is less than or equal to 1024 bytes for mnemonics created via `from_nonstandard_bytes()` and added explicit checks for the same in `from_raw_bytes()`, along with tests to ensure that this is enforced. As such, this note is considered *Fixed*.