Fine penetration tests for fine websites

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

# Audit-Report Distrust Keyfork Toolkit & Library 04.2024

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

## Index

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

## Introduction

This report describes the results of a cryptography review and source code audit against the Distrust Keyfork Toolkit, which was performed by Cure53 in April 2024.

In context, the customer invested a total of twenty-five days for extended coverage over the targets. A team comprising five experienced pentesters was assembled to fulfill all stages of the examination, from preparation through to execution and finalization. A single Work Package (WP) was created entitled *WP1: Cryptography reviews & code audits against Distrust Keyfork Toolkit*. Materials such as sources, a list of key focus areas, and other access points were handed over in advance. The provision of sources meant that the pentesting methodology conformed with a white-box approach.

A number of preliminary actions were undertaken during the week prior to the engagement (CW14 2024) to foster a hindrance-free environment for the testers to operate in.

Communications between the maintainers and Cure53 were facilitated through the creation of a dedicated and shared Slack channel. All personnel that played an active role in this exercise were invited to join the channel. Both parties contributed to clear and consistent communication. The well-defined scope of the assessment minimized the need for clarification, further streamlining the testing process. Cure53 kept all stakeholders informed by providing frequent status updates on testing progress and any associated findings. Live reporting was offered as an additional service and conducted via the aforementioned medium.

Regarding the findings, after achieving satisfactory coverage during the prescribed time frame, seven findings were encountered and documented. Fortunately, only one of those was categorized as a security vulnerability, while the other six were tagged with an impact score of *Low* or *Info*, denoting manageable threats or general weaknesses.

The overall number of findings is relatively moderate for a scope of this magnitude, which certainly reflects favorably on the security premise offered by the Distrust Keyfork library. It is additionally positive to note that this review identified no critical cryptographic or general security vulnerabilities.

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

The overall positive final verdict stands as a testament to the proactive approach taken towards secure cryptographic engineering practices within the Keyfork codebase. Nonetheless, Cure53 recommends committing to ongoing security improvements as well as performing regular external pentests in order to maintain this foundational layer of defense.

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 will be followed by a chapter outlining the *Test Methodology*, which serves to provide greater clarity on the techniques applied and coverage achieved throughout this audit. Subsequently, the report will list all findings identified in chronological order, starting with the *Identified Vulnerabilities* and followed by the *Miscellaneous Issues* 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 Distrust Keyfork Toolkit, giving high-level hardening advice where applicable.

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

## Scope

- **Cryptography reviews & code audits against Distrust Keyfork Toolkit & library**
  - **WP1**: Cryptography reviews & code audits against Distrust Keyfork Toolkit
    - **Source code:**
      - https://git.distrust.co/public/keyfork
    - **Key focus areas:**
      - Does Keyfork generate secure entropy?
      - Does Keyfork derive new keys from this seed securely?
      - Does Keyfork provision derived keys to Yubikeys securely?
      - Does Keyfork support securely splitting the key as an M-of-N set encrypted to each Yubikey?
      - Does Keyfork support the secure transmission of key shards back to a central offline machine, with only offline systems granted access to secrets?
  - **Test-supporting material was shared with Cure53**
  - **All relevant sources were shared with Cure53**

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

## Test Methodology

This section documents the testing methodology applied by Cure53 during this project and discusses the resulting coverage, shedding light on how various components were examined. Further clarification concerning areas of investigation subjected to deep-dive assessment is offered, especially in the absence of significant security vulnerabilities detected.

The assignment consisted of a single work package that was dedicated to cryptography reviews and a source code audit against the Distrust Keyfork Toolkit. The customer provided the source code and supporting documentation, as well as pinpointed several key focus areas for investigation, including the entropy generation, key derivation, smartcard provisioning (Yubikeys in particular), the process of splitting a key into shares and encrypting them, and the secure transmission of the encrypted shares for remote recovery of the secret in the use case of a disaster recovery.

The first aspect explored by Cure53 was the generation of initial entropy and a corresponding mnemonic, as mandated by BIP39. The generation of the initial entropy, which is directly gathered from */dev/urandom,* was found to be secure. However, the protocol limits code portability across different operating systems (see DIS-01-001). Abstracting this functionality into a dedicated module or using a library could improve maintainability, reduce error handling repetition, and streamline device management. While auditing the process of entropy generation, the test team observed the presence of initial safety checks for network connection and kernel version. However, these checks lack a clear security model to effectively gauge their relevance or the security proficiency they provide. In this regard, DIS-01-003 recommends establishing a formal threat model that clearly defines the security objectives and threats these checks aim to mitigate. This model would contextualize the security measures, acknowledge limitations, and help refine or justify future security enhancements based on a structured framework. Besides this, Cure53 was unable to detect any flaws in the process of generating a mnemonic from the entropy.

Next, the key derivation process was extensively studied. Since Keyfork follows BIP32 to derive deterministic keys, the implementation was audited for common vulnerabilities such as incorrect handling of the indices of hardened and non-hardened child keys, malformed paths, and unbounded depth. Positively, the testing team found that these undesirable situations were avoided. The only noteworthy observation in this area is that the implementation is not fully compliant with BIP32, as some derived keys are declared as invalid, which BIP39 accepts. However, the developer team is already aware[1] of this weakness and thus it was not re-documented in this report, since it occurs with a negligible probability and hence will likely never be encountered in practice.

[1] This issue is mentioned in a report from NCC Group shared by the customer at the beginning of the audit.

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

The process of splitting a secret into shares via Shamir's secret sharing scheme, as well as the encryption of the resulting shares, was systematically analyzed. The team's undertakings here confirmed the presence of a flaw in the library used for sharding, as reported in ticket DIS-01-002. This issue cannot be exploited in Keyfork under the current implementation, since a secret is randomly generated and split, hence re-splitting a fixed secret is not supported. Nevertheless, the flaw in the underlying library exists and may result in a leakage that allows reconstructing secrets in future evolutions of the software. Moreover, while reviewing the shard encryption procedure, Cure53 found that the OpenPGP location public keys that the shards are encrypted to are retrieved from within a file or directory. However, the uniqueness of these public keys is not guaranteed (see DIS-01-005), which can result in two different shards being encrypted to the same public key.

Subsequently, the assessment team vetted the disaster recovery mechanism, which was considered the most complex use case. Regarding Yubikey provisioning, Cure53 noticed that easily-guessable numeric PINs are permitted for configuration, as reported in DIS-01-006. This constitutes insufficient protection of the private keys contained in the smartcard should an attacker obtain physical access to the smartcards, even temporarily. Furthermore, the confirmation was made that the implementation of signature verification iterates through all signatures without ensuring that only one valid signature exists, potentially allowing multiple valid signatures and introducing timing side channels, as discussed in ticket DIS-01-004. This behavior could be exploited to glean insight into signature processing, particularly as the software evolves. As such, a major proportion of the secret reconstruction is the protocol utilized to transport the shares to the operator reconstructing the secret. To the testing team's knowledge, the protocol employs well-established cryptographic primitives (XDH, HKDF, AES-GCM). However, since it is custom-made, Cure53 deemed it appropriate to scrutinize this aspect extensively. The protocol was compared with similar established protocols, such as NIST SP 800-56a[2] and the ISO proposal for ECIES by Shoup[3]. The testers were unable to detect an attack strategy against the shard transport protocol during the assignment time frame, though ticket DIS-01-007 provides a recommendation for improving the robustness of the shard transport mirroring ECIES.

Finally, the Rust implementation was audited from a wider perspective, abstracting it from the use case. This review process commenced by checking all application dependencies, whereby the team verified that the RSA crate exhibits an advisory that is unpatched and ignored on the source's configuration (namely RUSTSEC-2023-0071). However, the vulnerability does not directly impact the application and is already known by the developer team. Rust's unsafe keyword usage was thoroughly analyzed since it remains a critical feature of the source code. Most references were related to terminal handling input/output and QR code parsing that relies on low-level libraries. An unsafe *from_raw_bytes* function is defined on *keyfork-mnemonic-util*, though no memory-related flaw was identified. The unsafe keyword is only used to express that non-compliant results may be produced, which is a sound development practice.

[2] https://csrc.nist.gov/pubs/sp/800/56/a/r3/final
[3] https://www.shoup.net/papers/iso-2_1.pdf

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

The possibility of command and argument injection vulnerabilities was considered, since some application features rely on calling external commands/binaries, such as *qrencode* and *tput*. Cure53 confirmed the inability to inject or modify either the arguments or the binary called from user-controlled input. All possible inputs originating from external sources were tested to ensure that the application was not vulnerable to any form of injection. Here, the application demonstrated commendable reliability to malformed, large, and malicious inputs in various areas, exiting successfully and avoiding undefined behaviors.

**Dr.-Ing. Mario Heiderich, Cure53**
Wilmersdorfer Str. 106
D 10629 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., *DIS-01-001*) to facilitate any future follow-up correspondence.

### DIS-01-006 WP1: Easily guessable numeric sequences accepted as PINs *(Low)*

***Fix note:*** *This issue has been mitigated by the maintainer team and subsequently fix-verified by Cure53 by inspecting a PR/diff.*

When provisioning smartcards handed out to shareholders, a user PIN and an administrator PIN are configured. Each shareholder's smartcard contains the OpenPGP location private key required to decrypt the share. Thus, the PIN is the only layer that protects the private key in the event an attacker is able to obtain physical access to the smartcard.

When auditing the process of configuring smartcard PINs, the test team found that these PINs are actually validated, as indicated by the following code excerpt.

**Affected file:**
*crates/keyfork/src/cli/wizard.rs*

**Affected code:**

```
fn generate_shard_secret(
    threshold: u8,
    max: u8,
    keys_per_shard: u8,
    output_file: &Option<PathBuf>,
) -> Result<()> {
    let seed = keyfork_entropy::generate_entropy_of_const_size::<{256 /
8}>()?;
    let mut pm = default_terminal()?;
    let mut certs = vec![];
    let mut seen_cards: HashSet<String> = HashSet::new();
    let stdout = std::io::stdout();
    if output_file.is_none() {
        assert!(
            !stdout.is_terminal(),
            "not printing shard to terminal, redirect output"
        );
    }

    let user_pin_validator = PinValidator {
        min_length: Some(6),
```

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

```
            ..Default::default()
        }
        .to_fn();
        let admin_pin_validator = PinValidator {
            min_length: Some(8),
            ..Default::default()
        }
        .to_fn();

        for index in 0..max {
            let cert = derive_key(seed, index)?;
            for i in 0..keys_per_shard {
                pm.prompt_message(Message::Text(format!(
                    "Please remove all keys and insert key #{} for user #{}",
                    i + 1,
                    index + 1,
                )))?;
                let card_backend = loop {
                    [...]
                };
                let user_pin = pm.prompt_validated_passphrase(
                    "Please enter the new smartcard User PIN: ",
                    3,
                    &user_pin_validator,
                )?;
                let admin_pin = pm.prompt_validated_passphrase(
                    "Please enter the new smartcard Admin PIN: ",
                    3,
                    &admin_pin_validator,
                )?;
                factory_reset_current_card(
                    &mut seen_cards,
                    user_pin.trim(),
                    admin_pin.trim(),
                    &cert,
                    card_backend,
                )?;
            }
            certs.push(cert);
        }

        [...]
    }
```

This validator ensures that the PIN's length lies within the range of a hardcoded minimum (variable depending on whether it constitutes a user or admin PIN) and a maximum. It also ensures that the PIN consists only of numeric characters, which is essential toward preventing injection attacks.

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

**Affected file:**
*crates/util/keyfork-prompt/src/validators.rs*

**Affected code:**
```
impl Validator for PinValidator {
    type Output = String;
    type Error = PinError;

    fn to_fn(&self) -> Box<dyn Fn(String) -> Result<String, Box<dyn
std::error::Error>>> {
        let min_len = self.min_length.unwrap_or(usize::MIN);
        let max_len = self.max_length.unwrap_or(usize::MAX);
        let range = self.range.clone().unwrap_or('0'..='9');
        Box::new(move |mut s: String| {
            s.truncate(s.trim_end().len());
            let len = s.len();
            if len < min_len {
                return Err(Box::new(PinError::TooShort(len, min_len)));
            }
            if len > max_len {
                return Err(Box::new(PinError::TooLong(len, max_len)));
            }
            for (index, ch) in s.chars().enumerate() {
                if !range.contains(&ch) {
                    return Err(Box::new(PinError::InvalidCharacters(ch,
index)));
                }
            }
            Ok(s)
        })
    }
}
```

However, the implementation does not provide additional logic that prevents easily-guessable numeric sequences (such as *00…00*) from being configured as PINs for the smartcards. Limiting the number of attempts generally contributes to smartcard security, though this should be complemented with the rejection of trivially-guessable PINs, which would provide another safeguard layer if a smartcard is stolen.

To mitigate this issue, Cure53 recommends preventing easily-guessable patterns from being configured as the user/admin PIN, as implemented for other correlatory systems[4].

[4] https://learn.microsoft.com/en-us/windows/security/identity-protection/hello-for-business/faq#[...]-pins

**Dr.-Ing. Mario Heiderich, Cure53**
Wilmersdorfer Str. 106
D 10629 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, while a vulnerability is present, an exploit may not always be possible.

### DIS-01-001 WP1: Mnemonic system seed source platform-dependent *(Info)*

The codebase's *generate_entropy_of_size* and *generate_entropy_of_const_size* Rust functions directly access the */dev/urandom* device to retrieve random data. This direct access poses negligible security concern and remains viable from a security perspective. However, potential areas of improvement regarding eventual portability and maintainability are introduced as a result:

- **Portability:** The direct use of */dev/urandom* limits the code's portability across different operating systems. While */dev/urandom* is commonly available on Unix-like systems, this approach does not inherently support other platforms such as Windows, which uses different mechanisms for secure randomness.
- **Error handling:** The current implementation opens and reads from the device directly within each function call. This approach can lead to repeated error handling and device management overhead, which could be streamlined through abstraction.

**Affected file:**
*crates/util/keyfork-entropy/src/lib.rs*

**Affected code:**

```
pub fn generate_entropy_of_size(byte_count: usize) -> Result<Vec<u8>,
std::io::Error> {
    ensure_safe();
    let mut vec = vec![0u8; byte_count];
    let mut entropy_file = File::open("/dev/urandom")?;
    entropy_file.read_exact(&mut vec[..])?;
    Ok(vec)
}

pub fn generate_entropy_of_const_size<const N: usize>() -> Result<[u8; N],
std::io::Error> {
    let mut output = [0u8; N];
    let mut entropy_file = File::open("/dev/urandom")?;
    entropy_file.read_exact(&mut output[..])?;
    Ok(output)
}
```

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

To enhance the code's robustness, portability, and maintainability, it is recommended to abstract the randomness source via a library or a dedicated module. This abstraction can provide several advantages:

- **Cross-platform support:** Utilizing a library such as *rand* or *ring* in Rust, which abstracts over the OS-specific mechanisms to generate random numbers, can facilitate code functionality across multiple platforms without any additional burden on the end developer.
- **Centralized error handling:** By abstracting the randomness generation, error handling can be centralized in a single location, improving the reliability and readability of the code. This will also reduce redundancy and facilitate easier updates or fixes related to random number generation.

## DIS-01-002 WP1: Bias of polynomial coefficients in secret sharing *(Low)*

While investigating the process of splitting a secret into shares via Shamir's secret sharing scheme (SSS), Cure53 observed that an underlying library named *sharks* handles this operation. Since secret sharing is crucial to the overall security of the system, Cure53 deemed it apt to inspect this library. Here, the team determined that the coefficients of the polynomial involved in the SSS are not fully selected at random; specifically, the coefficients cannot constitute zero. This behavior introduces a slight bias.

**Affected file:**
*sharks/src/math.rs*

**Affected code:**

```
pub fn random_polynomial<R: rand::Rng>(s: GF256, k: u8, rng: &mut R) ->
Vec<GF256> {
	let k = k as usize;
	let mut poly = Vec::with_capacity(k);
	let between = Uniform::new_inclusive(1, 255);

	for _ in 1..k {
	poly.push(GF256(between.sample(rng)));
	}
	poly.push(s);

	poly
}
```

The correct method to select a random polynomial would be to select all coefficients (including the most significant coefficient) uniformly in the range 0..255 (inclusive). Otherwise, knowledge that a coefficient in a polynomial cannot be 0 permits the exclusion of single byte values for the shared secret given one share less than required.

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

Notably, the information leaked via the biased generation is minimal and does not pose a threat for the current implementation, due to the fact that each seed is generated randomly and shared only once. However, exploiting this weakness necessitates sharing the same secret multiple times. In this scenario, an attacker could exclude an exponential number of values for each of the shared bytes until sufficiently few values remain for brute forcing. Cure53 estimates that under ideal circumstances (e.g., a 2-out-of-N scheme) a shared secret can be reconstructed if the same secret has been distributed 500-1500 times.

### DIS-01-003 WP1: Unclear security model for kernel & network safety *(Info)*

Keyfork implements two safety checks on initialization that are designed to verify the baseline of security for the endpoint it is operating on. The first involves the network connection, which validates that the endpoint is not connected to a network, and the second entails the kernel version, which ensures that the system is running a kernel version considered relatively modern.

While the intentions behind these safety checks are commendable and their respective implementations are correct, it is unclear how their overall effectiveness and relevance could be evaluated within a structured security model.

Firstly, the mere absence of an active network connection does not inherently secure a device. Malicious actions can occur offline and other network interfaces (e.g., virtual network interfaces not considered in the check) may still compromise the system. Relying solely on network disconnection can lead to overlooking other critical security aspects such as physical access security, local data storage vulnerabilities, and the integrity of the running applications.

Secondly, the kernel version alone is a poor proxy for security. Newer kernels can be affected by unpatched vulnerabilities, while older kernels may offer sufficient security if properly patched and configured.

**Affected file:**
*crates/util/keyfork-entropy/src/lib.rs*

The aforementioned safety checks implemented in the smartcard API library are fundamentally sound in their intent to enhance the system's robustness. However, the effectiveness and clarity of these checks can be substantially improved by formally defining a threat model. This threat model should outline the specific security goals that these checks aim to accomplish, as well as the threats they are designed to mitigate.

The definition of a clear threat model can introduce a number of benefits, including:

- **Check contextualization**, i.e., clarifying the necessity of each check and determining how they contribute to the system's security posture. This helps in

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

understanding the role of each security measure within the broader security framework.

- **Limitations and assumption identification**, i.e., acknowledging the limitations of each check under the defined threat model and specifying the assumptions (for instance, the absence of physical tampering or the level of user privilege required).
- **Guidance for future security measures**, i.e. providing a baseline for evaluating whether additional security checks are needed or if the existing checks should be modified in response to evolving threats.

Cure53 must reiterate that amending or expanding the existing checks at this stage is not strictly necessary. Alternatively, the developer team could define a detailed threat model in order to provide the necessary clarity and justification for these checks, ensuring that they are both adequate and appropriate for the intended security objectives.

### DIS-01-004 WP1: Minor safety improvement for signature parsing logic *(Info)*

***Fix note:*** *This issue has been mitigated by the maintainer team and subsequently fix-verified by Cure53 by inspecting a PR/diff.*

The current implementation for the signature verification interface, namely *VerificationHelper* across both *Keyring* and *SmartCartManager* types, iterates through all signatures within a *SignatureGroup* and checks them one by one. However, only one valid signature was expected here, as confirmed in discussions with the maintainer team. As such, this method raises the following issues:

- **Multiple valid signatures:** The loop allows for multiple signatures to be checked without validating if more than one valid signature should be acceptable. For fortified security, particularly in environments where strict signature policies are enforced, the system should verify that exactly one valid signature is present and all others are either absent or explicitly flagged as invalid.
- **Potential for timing side channels:** The loop through all signatures introduces a potential timing side channel, as the time taken to verify signatures may correlate with the number of signatures or their validity states. An attacker could potentially leverage this information to infer aspects of the signature processing, such as the number of valid or invalid signatures. A timing side channel is likely inexploitable in the current software use case; however, fixing the code early could avoid this code pattern generalizing across different novel use cases that may be introduced in the future.

**Affected files:**

- *keyfork-shard/src/openpgp/keyring.rs*
- *src/openpgp/smartcard.rs*

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

**Affected code:**

```
impl<P: PromptHandler> VerificationHelper for &mut Keyring<P> {
    fn get_certs(&mut self, ids: &[KeyHandle]) ->
openpgp::Result<Vec<Cert>> {
        Ok(ids
            .iter()
            .flat_map(|kh| {
                self.root
                    .iter()
                    .filter(move |cert| &cert.key_handle() == kh)
            })
            .cloned()
            .collect())
    }
    fn check(&mut self, structure: MessageStructure) -> openpgp::Result<()>
{
        for layer in structure {
            #[allow(unused_variables)]
            match layer {
                MessageLayer::Compression { algo } => {}
                MessageLayer::Encryption {
                    sym_algo,
                    aead_algo,
                } => {}
                MessageLayer::SignatureGroup { results } => {
                    for result in results {
                        if let Err(e) = result {
                            // FIXME: anyhow leak: VerificationError impl
                                 std::error::Error
                            // return Err(e.context("Invalid signature"));
                            return Err(anyhow::anyhow!("Invalid signature:
                                 {e}"));
                        }
                    }
                }
            }
        }
        Ok(())
    }
}

impl<P: PromptHandler> VerificationHelper for &mut SmartcardManager<P> {
    fn get_certs(&mut self, ids: &[openpgp::KeyHandle]) ->
openpgp::Result<Vec<Cert>> {
        #[allow(clippy::flat_map_option)]
        Ok(ids
            .iter()
```

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

```
            .flat_map(|kh| self.root.as_ref().filter(|cert|
cert.key_handle() == *kh))
            .cloned()
            .collect())
    }

    fn check(&mut self, structure: MessageStructure) -> openpgp::Result<()>
{
        for layer in structure {
            #[allow(unused_variables)]
            match layer {
                MessageLayer::Compression { algo } => {}
                MessageLayer::Encryption {
                    sym_algo,
                    aead_algo,
                } => {}
                MessageLayer::SignatureGroup { results } => {
                    for result in results {
                        if let Err(e) = result {
                            // FIXME: anyhow leak
                            return Err(anyhow::anyhow!("Verification error:
                                {}", e.to_string()));
                        }
                    }
                }
            }
        }
        Ok(())
    }
}
```

To mitigate these issues, Cure53 advises incorporating the following improvements. Firstly, the dev team should enforce single signatures by modifying the signature verification logic to ensure that exactly one valid signature is required and processed. This can be achieved by counting the number of valid signatures and returning an error if the count does not equal one. This check should be performed after all signatures have been processed to avoid early exits that could affect timing.

Secondly, constant time operations can be adopted, which requires refactoring the signature checking to execute in constant time with respect to the number of signatures. This can typically be managed by using fixed-time comparison functions and ensuring that the code path (including time taken for memory accesses, CPU cycles, and so on) is identical regardless of the signature count or validity.

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

## DIS-01-005 WP1: Public key discovery fails to ensure key distinction *(Low)*

***Fix note:*** *This issue has been mitigated by the maintainer team and subsequently fix-verified by Cure53 by inspecting a PR/diff.*

Cure53 noted that the *shard_and_encrypt* function splits a secret into shares and subsequently locates OpenPGP public keys within a file or directory. Once it is ensured that the number of identified public keys is equal to the number of shares, the function iterates the tuples *(pk, share)* and encrypts each share to the corresponding public key. This logic can be observed in the following code excerpt.

**Affected file:**
*crates/keyfork-shard/src/lib.rs*

**Affected code:**

```
fn shard_and_encrypt(
    &self,
    threshold: u8,
    max: u8,
    secret: &[u8],
    public_key_discovery: impl KeyDiscovery<Self>,
    writer: impl Write + Send + Sync,
) -> Result<(), Box<dyn std::error::Error>> {
    let mut signing_key = self.derive_signing_key(secret);

    let sharks = Sharks(threshold);
    let dealer = sharks.dealer(secret);

    let public_keys = public_key_discovery.discover_public_keys()?;
    assert!(
        public_keys.len() < u8::MAX as usize,
        "must have less than u8::MAX public keys"
    );
    assert_eq!(
        max,
        public_keys.len() as u8,
        "max must be equal to amount of public keys"
    );
    let max = public_keys.len() as u8;
    assert!(max >= threshold, "threshold must not exceed max keys");

    [...]
    for (pk, share) in public_keys.iter().zip(dealer) {
        let shard = Vec::from(&share);
        messages.push(self.encrypt_shard(&shard, pk, &mut signing_key)?);
    }
```

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

```
    self.format_shard_file(&messages, writer)?;

    Ok(())
}
```

While auditing the function that handles public key discovery, the test team confirmed it simply collects every OpenPGP public key that can be correctly parsed. However, this process does not verify that all public keys are distinct. Therefore, any configuration errors whereby the OpenPGP keys are not correctly loaded into the directory or file would remain unnoticed, causing two different shards to be encrypted to the same public key.

**Affected file:**
*crates/keyfork-shard/src/openpgp.rs*

**Affected code:**

```
impl<P: PromptHandler> KeyDiscovery<OpenPGP<P>> for &Path {
    fn discover_public_keys(&self) -> Result<Vec<<OpenPGP<P> as
Format>::PublicKey>> {
        OpenPGP::<P>::discover_certs(self)
    }
    [...]
}

impl<P: PromptHandler> OpenPGP<P> {
    pub fn discover_certs(path: impl AsRef<Path>) -> Result<Vec<Cert>> {
        let path = path.as_ref();

        if path.is_file() {
            let mut vec = vec![];
            for cert in
CertParser::from_file(path).map_err(Error::Sequoia)? {
                vec.push(cert.map_err(Error::Sequoia)?);
            }
            Ok(vec)
        } else {
            let mut vec = vec![];
            for entry in path
                .read_dir()
                .map_err(Error::Io)?
                .filter_map(Result::ok)
                .filter(|p| p.path().is_file())
            {

vec.push(Cert::from_file(entry.path()).map_err(Error::Sequoia)?);
            }
            Ok(vec)
```

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

```
        }
    }
}
```

To mitigate this issue, Cure53 recommends ensuring that the public keys discovered within the given file or directory during the sharding and encryption process are all distinct.

### DIS-01-007 WP1: Robustness improvement for shard transport shared key *(Info)*

***Note:*** *Following discussions with the customer, it became apparent that the user IDs of the involved parties are unknown to the combiner of the shares. As such, the proposal (which assumes known user IDs) unnecessarily complicates the protocol and can be disregarded in the current setup.*

The shard transport protocol utilizes an encryption mode that resembles ECIES[5]. ECIES includes additional steps that safeguard against certain attack strategies; however, these steps are not present in Keyfork's implementation.

In particular, ECIES includes the ephemeral key as a parameter of the key derivation function, which prevents a rather benign form of malleability. Moreover, ECIES facilitates integrating additional context data in the integrity check of the symmetric encryption, which prevents replaying ciphertexts out of context.

**Affected file:**
*crates/keyfork-shard/src/lib.rs*

**Affected code:**

```
pub fn remote_decrypt(w: &mut impl Write) -> Result<(), Box<dyn
std::error::Error>> {
    [...]
    while iter_count.is_none() || iter_count.is_some_and(|i| i > 0) {
        let shared_secret =
our_key.diffie_hellman(&PublicKey::from(pubkey)).to_bytes();
        let hkdf = Hkdf::<Sha256>::new(None, &shared_secret);
        let mut hkdf_output = [0u8; 256 / 8];
        hkdf.expand(&[], &mut hkdf_output)?;
        let shared_key = Aes256Gcm::new_from_slice(&hkdf_output)?;
        [...]
    }
    [...]
}
```

Even though the key shard transport did not yield any vulnerabilities during the time frame of the assignment and is generally considered secure, the maintainer team can add context to cryptographic protocols and improve their robustness by mirroring the ECIES proposal with

[5] https://www.shoup.net/papers/iso-2_1.pdf

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

minimal expense required. To achieve this, firstly one could include the ephemeral key in the *info* string of the HKDF. Secondly, the index *i* of the shareholder could be included as additional authenticated data (AAD) in the AES-GCM encryption.

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

## Conclusions

This project's test initiatives comprised a single work package consisting of a cryptography review and code audit of the Distrust Keyfork Toolkit. The customer provided the source code and several key targets for the audit team to prioritize, which were validated as integral components by Cure53 following further reviews of the codebase, while effective cross-organization communication was enabled through the use of a shared Slack channel.

The generation of the initial entropy and associated mnemonic (as dictated by BIP39) was subjected to stringent analysis. Simultaneously, the team audited the deterministic derivation of keys as in BIP32. The codebase was found to avoid pitfalls that are commonly encountered in other implementations of these standards, including (but not limited to) out-of-bounds indices or depth of the child keys, malformed paths, and seeds generated from a weak source of entropy. The discoveries reported in these areas merely constitute hardening recommendations and should be relatively straightforward to administer (see DIS-01-001 and DIS-01-003).

Next, Cure53's assessment of the process of splitting a secret into shards and the ensuing encryption revealed the presence of a flaw in the library that implements Shamir's secret sharing scheme leveraged by Keyfork, as described in ticket DIS-01-002. However, since a secret is not re-split in the current implementation except with negligible probability, the flaw remains unexploitable at present. Nevertheless, the recommended guidance should be considered to avoid potential erroneous behaviors in future use cases.

Additionally, the assessors found that the shards are encrypted to public keys retrieved from a file or directory, though their distinction is not ensured (see DIS-01-005). As a consequence, a shareholder or attacker could ultimately decrypt two distinct shares. Another correlating limitation in this area pertains to the fact that signature verification iterates through all signatures without ensuring that only one valid signature exists, potentially allowing multiple valid signatures and introducing timing side channels (see DIS-01-004).

Concerning the disaster recovery use case, the review of smartcard provisioning uncovered that extremely weak numeric patterns are configurable as PINs (see DIS-01-006), which is wholly insufficient from a security perspective should an attacker gain physical access to any of the smartcards. The protocol that transports the encrypted shards to an operator, which performs the remote decryption and reconstruction of the secret, was verified to be custom-made. Cure53 therefore honed in on this specific facet for thorough evaluation, though no prevalent weaknesses were acknowledged except for a minor hardening improvement (see DIS-01-007).

As discussed in the Test Methodology chapter, the Rust codebase was also reviewed from a general perspective, which remained unaffected by common drawbacks. The implementation's adherence to optimal coding practices when dealing with untrusted data serves to both reduce the attack surface and foster a sound overarching security posture.

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

All in all, Cure53 is pleased to confirm that the Keyfork codebase successfully avoids all major cryptographic and general security shortcomings. The codebase exhibited a proactive approach towards secure cryptographic engineering, as substantiated by the scant volume of findings and minor median severity rating on the whole.

Cure53 would like to thank Lance Vick, and Ryan Heywood as well as the Distrust team for their excellent project coordination, support, and assistance, both before and during this assignment.