Turtl Security Audit

Johannes Hald

May 5, 2021

Introduction

> *"The secure, collaborative notebook"*[1]

> *"Turtl lets you take notes, bookmark websites, and store documents for sensitive projects. From sharing passwords with your coworkers to tracking research on an article you're writing, Turtl keeps it all safe from everyone but you and those you share with."*[2]

This report describes the results of an audit of Turtl's core Rust library "core-rs" (referred to as "Turtl" throughout the rest of the report), based on the commit fd3146aa09377e0a11d5a8873c2da5983a9832a6.

The audit focused primarily on the cryptographic components and their usage throughout the library.

Audit summary

It was found that Turtl currently lacks security-related maintenance updates. A number of dependencies are affected by reported security issues, which may put user-data at risk.

Even though TLS between server and client is optional, the security of communication between the two, relies heavily upon the use of it. Turtl should not be used without TLS and, according to the developers of Turtl, TLS is also highly recommended. Further, the Turtl server should not currently be used with self-signed certificates (see TURTL-002).

[1] https://turtlapp.com/

[2] https://privacytools.io/software/notebooks/

Changelog

- 02-05-2021: Release of report
- 05-05-2021: Expanded suggestion for additional Key protections based on feedback from Ionut Mihalcea.

Identified issues

TURTL-001: Missing check for correct initialization of libsodium

The libsodium[3] wrapper sodiumoxide[4], provides an initialization function that ensures, among other things, thread-safety when accessing the PRNG. This PRNG is used to generate cryptographic key-material used to secure data.

sodiumoxide is used in Turtl, but the initialization check is never performed. As such, the use of sodiumoxide poses a thread-unsafety risk. Turtl must include a check for correct initialization, whenever Turtl is launched.

```
if !sodiumoxide::init() {
    // panic! the library couldn't be initialized, it is not safe to use
}
```

Listing 1: Example check for correct initialization

TURTL-002: Possibility to trust any certificate

It is possible for users to host their own Turtl servers. For this purpose, Turtl has a configuration option intended to allow self-signed certificates:

```
match config::get::<Option<bool>>(&["api", "allow_invalid_ssl"]) {
    Ok(x) => {
        if let Some(allow_invalid_ssl) = x {
            if allow_invalid_ssl {
                debug!("api::call() -- req: allow invalid ssl");
                cachekey.push(String::from("allow-invalid-ssl"));
                client_builder =
↪   client_builder.danger_accept_invalid_certs(true);
            }
        }
    }
    Err(_) => {}
}
```

Listing 2: core-rs/src/api.rs, L120-131

However, *danger_accept_invalid_certs(true)* does not only allow self-signed certificates. According to the documentation of reqwest (the library used to per-

[3] libsodium - init()

[4] sodiumoxide - init()

form HTTP requests), this allows the use of any certificate for any site, including expired certificates[5]. A user deploying a server with this config-option enabled, is vulnerable to attacks such as MITM. Turtl should instead correctly validate self-signed certificates. Alternatively, Turtl could require valid certificates signed by a trusted root and not accept self-signed certificates at all. This has become very accessible with CAs such as Let's Encrypt[6] that offer free, valid, and automatically renewable certificates.

TURTL-003: Vulnerable dependencies

Running cargo-audit[7] on the Turtl project reports 10 dependencies with security issues and other problems. These dependencies must be updated and a new Turtl version must be released.

TURTL-004: Randomly generating 12-byte nonces

Turtl randomly generates 12-byte nonces to be used with ChaCha20-Poly1305. This is considered unsafe, due to the risk of collision for a 12-byte nonce. If a collision occurs, a nonce-reuse scenario will break confidentiality and authenticity of data encrypted with the corresponding key.

```rust
pub fn encrypt(key: &Key, plaintext: Vec<u8>, op: CryptoOp) -> CResult<Vec<u8>> {
    let version = CRYPTO_VERSION;
    match op.algorithm {
        "chacha20poly1305" => {
            let nonce = match op.nonce {
                Some(x) => x,
                None => low::chacha20poly1305::random_nonce()?,
            };
```

Listing 3: core-rs/src/crypto/mod.rs, L237-244

This can be avoided by either using an incremental nonce, or switching to the XChaCha20-Poly1305 construction, for which it is save to randomly generate nonces.

TURTL-005: Replay-attack on authentication tokens

When a user logs in to Turtl, their email and password will be used to derive a master key. This master key is, in part, used to construct the authentication token that authenticates the user to the Turtl server.

[5] reqwest - danger_accept_invalid_certs
[6] Let's Encrypt
[7] crates.io - cargo-audit

```rust
/// Generate a user's auth token given some variables or something
pub fn generate_auth(username: &String, password: &String, version: u16) -> TResult<(Key, String)> {
    info!("user::generate_auth() -- generating v{} auth", version);
    let key_auth = match version {
        0 => {
            let key = generate_key(username, password, version)?;
            let nonce_len = crypto::noncelen();
            let nonce = (crypto::sha512(username.as_bytes())?)[0..nonce_len].to_vec();
            let pw_hash = crypto::to_hex(&crypto::sha512(&password.as_bytes())?)?;
            let user_record = String::from(&pw_hash[..]);
            let op = crypto::CryptoOp::new_with_nonce("chacha20poly1305", nonce)?;
            let auth_bin = crypto::encrypt(&key, Vec::from(user_record.as_bytes()), op)?;
            let auth = crypto::to_hex(&auth_bin)?;
            (key, auth)
        }
        _ => return TErr!(TError::NotImplemented),
    };
    Ok(key_auth)
}
```

Listing 4: core-rs/src/models/user.rs, L142-160

The above function defines the generation of aforementioned authentication tokens. When sent over an insecure channel, these authentication tokens may leak whether or not a user has changed their password, by comparing two different authentication tokens, because the nonce is a hash of the username. The authentication tokens are also not session-dependent. This means, an attacker may record an authentication token for a given user at one point, and use it to impersonate the user at a later time.

The above is possible, only if the server and client do not communicate securely using TLS.

In general, the design of these tokens seem unnecessarily complex, which is why the recommendation is to move to another authentication token format altogether (such as PASETO or Branca) or simply authenticate the user based on password hashes instead.

Suggested improvements

The following are suggestions for aspects that weren't found to have exploitable security issues, but will help strengthen the applications security stance.

Set size-limit for files or encrypt them in chunks

The Turtl app allows users to attach files, images, etc to notes. These files are encrypted in-memory before they're written to disk:

```
// encrypt the file using the turtl standard serialization format
let enc = turtl.work.run(move || {
    crypto::encrypt(&note_key, data, crypto::CryptoOp::new("chacha20poly1305")?)
        .map_err(|e| From::from(e))
})?;
```

Listing 5: core-rs/src/models/file.rs, L248-252

```
// decrypt the file using the turtl standard serialization format
let data = turtl.work.run(move || {
    crypto::decrypt(&note_key, enc)
        .map_err(|e| From::from(e))
})?;
```

Listing 6: core-rs/src/models/file.rs, L218-222

There is however no file-size limit when saving such files. This means the entirety of a file's content is encrypted in memory. Having no limit makes it easier for the application to crash during encryption or decryption of the files. This can be avoided by setting a file-size limit, which is checked before encrypting & decrypting files. Otherwise, the files must be encrypted in chunks, which will require a secure format for chunked encryption, such as secretstream[8] from libsodium.

Document that backups are not encrypted

If a user exports data for backup-purposes or migration between servers, this data is not encrypted. All notes, passwords, etc are saved to disk in plaintext. A suggestion would be to make it clear to the user through the documentation and UI, that any exported data is not encrypted.

[8] libsodium - secretstream

Make the salt used for generating invite keys random

When a user sets a passphrase for an invite key, this passphrase is stretched with a KDF. The salt used here is a constant - the string ”invite salt” hashed with SHA512. There is no need for the salt to be constant, so the salt should be generated randomly instead.

```
fn gen_invite_key(&mut self, passphrase: Option<String>) -> TResult<()> {
        let passphrase = match passphrase {
            Some(pass) => pass,
            None => String::from(DEFAULT_INVITE_PASSPHRASE),
        };
        let hash = crypto::sha512("invite salt".as_bytes())?;
        let key = crypto::gen_key(passphrase.as_bytes(),
↪   &hash[0..crypto::KEYGEN_SALT_LEN], crypto::KEYGEN_OPS_DEFAULT,
↪   crypto::KEYGEN_MEM_DEFAULT)?;
        self.set_key(Some(key));
        Ok(())
}
```

Listing 7: core-rs/src/models/invite.rs, L125-134

Add additional protections for the key type

The Key typed used throughout Turtl, is a wrapper for sensitive key material used to secure data. However, this type lacks additional protections that may be implemented without much added complexity.

Key automatically derives the ‘Debug‘ trait, which opens the possibility for the key to be leaked in debug logs. Instead, if the ‘Debug‘ trait is manually implemented, the sensitive values in the underlying byte-array may be omitted from the output.

```
#[derive(Debug, Default)]
pub struct Key {
    /// Holds the actual bytes for our key
    data: Vec<u8>,
}
```

Listing 8: core-rs/src/crypto/key.rs, L8-12

Although not in the scope of Turtls threat-model, zeroization of the underlying bytes can be implemented relatively cheaply through the ‘Drop‘ trait. This would be a ”best-practice improvement” for which the zeroize[9] crate may be

[9] crates.io - zeroize

used. Alternatively, the Key newtype may be completely replaceable by the types provided in the secrecy library[10], which is built on top of zeroize.

Lastly, the ‘PartialEq‘ trait is implemented manually for this type. However, the implementation does not perform comparison on the underlying bytes in constant-time. By changing the ‘PartialEq‘ implementation, this could also be added relatively cheaply e.g. using the subtle[11] crate.

```rust
impl PartialEq for Key {
    fn eq(&self, other: &Key) -> bool {
        self.data() == other.data()
    }
}
```

Listing 9: core-rs/src/crypto/key.rs, L51-55

Weak minimum length requirement for password

When creating an account or changing the password for an existing one, the user is required to provide a password of at least four characters. This is a very low requirement and suggested to be raised to eight or higher.

[10] crates.io - secrecy
[11] crates.io - subtle