LUCENE CEFF 1.0

LUCENE CHUNKED ENCRYPTED FILE FORMAT

BY ELIATRA

28.03.2021

FOR THE IMPATIENT

- Ceff for Lucene is a library for encryption at rest of Lucene data
- All files written to (and read from) disk by Lucene are encrypted and authenticated
- Support for AES-GCM and ChaCha20-Poly1305 is built-in but encryption schemes are pluggable and extensible
- Works on Lucene 8.0.0+ and Java 8+ (Java 11+ recommended)
- No 3rd party dependencies
- Easy to use

FOR THE IMPATIENT

- Wraps NIOFS or MMAP Directory

```
Path path = ...;
byte[] key = ...; // 256 bit key
int chunkSize = 64 * 1024;// chunkSize in bytes
CeffMode encryptionMode = CeffMode.AES_GCM_MODE;// ChaCha20-Poly1305 is also available

Directory encryptedIndex =
new CeffDirectory(
new MMapDirectory(path), key, chunkLength, encryptionMode);
IndexWriter w = new IndexWriter(encryptedIndex, config);
DirectoryReader r = DirectoryReader.open(encryptedIndex);
```

IMPLEMENTATION DETAILS

- Because Lucene needs to have random access to files (when rading them) we can notbuse a simple stream chipher approach
- The solution is to cut up the plaintext into chunks
- Each chunk is encrypted and authenticated and, because all chunks are of the same length, randomly addressable
- A seek to a specific position determine the respective chunk and decrypt them
- This approach also supports „slicing“ with reasonable performance (given that the chunks are not too big)

IMPLEMENTATION DETAILS

- To ensure that the file was not tampered with every chunk contains metadata which will be validated
- In particular every chunk contains a UUID and the (zero-based) chunk number
- At the end of every encrypted file a digest over the UUID‘s as well as the total number of chunks is attached
- Validating this signature makes sure that the chunks were not reordered, truncated or otherwise amnipulated

FILE FORMAT

- The chunked encrypted file format consists of
  - A 9 byte header
    - 8 bytes: „magic bytes“ used to detect the file format and to encode versions
    - 1 byte: encryption mode
  - Abtritray number of chunks where every chunk consists of
    - 12 bytes: Nonce/IV
    - 16 bytes: UUID
    - 8 bytes: chunk number
    - Encrypted data with the length of „chunklen“ (the last chunk can be smaller)
    - 16 bytes: authentication tag
  - A signature at the end of the file after the last chunk
    - 12 bytes: Nonce/IV
    - 8 bytes: chunklen
    - 8 bytes: last chunk number
    - 8 bytes: length of the original plaintext
    - 64 bytes : SHA-512 digest over the UUID‘s
    - 16 bytes: authentication tag

WRITE THE FORMAT

- Write the header
- Read the plaintext in portions of „chunklen“
- For every chunk
  - Create a new random nonce and write it to the file
  - Get the current chunk number and write it to the file (zero-based)
  - Create a new random UUID, update the digest, and write it to the file
  - Encrypt the plaintext portion (with the chunk number and UUID as AAD) using the nonce and write it to the file
- Write the signature
  - Create a new random nonce and write it to the file
  - Write the chunklength, the number of the last chunk and the length of the plaintext to the file
  - Calculate the final SHA-512 digest, encrypt it together with chunklength, the number of the last chunk and the length of the plaintext as AAD using the nonce and write it to the file

READ THE FORMAT

- Read the header and make sure the magic bytes are matching. Determine the encryption mode.
- Scan all UUID‘s and chunk numbers and validate that the chunk number is strictly monotonically increasing. For every UUID update the digest.
- Seek to the signature and decrypt it. Make sure the decrypted signature match your previously calculated digest. You also need to check that the chunklenght and last chunk number stored along with signature match.

SEEKING

- Given the fact that the length of plaintext and ciphertext is always equal and that we know the chunklength, we can easily calculate plaintext offsets from the ciphertext offsets and vice versa.
- See `void seek(long pos)` in `CeffIndexInput`

SLICING

- Slicing is an operation which provides a relative view on a portion of data defined by an offset and a length
- To enable slicing in CEFF we need to determine the start and end block of the slice, seek to the start block, decrypt it and treat every read operation with an offset
- See `slice(String sliceDescription, long offset, long length)` in `CeffIndexInput`

THE END

- More information can be found here: https://eliatra.com/ceff
- info@eliatra.com

SLICE