Dexios: A Format for Encrypting and Storing Sensitive Data

brxken128

September 16, 2022

Abstract

Dexios is an open-source, symmetric-key-based file encryption program which uses audited encryption libraries to provide the highest level of security. It uses a custom header format in order to keep the deserialization and decryption of files performant.

1 Introduction

Over the past few years, there has been a great push for users to encrypt their files and other personal data - whether this be to blind oppressive government's prying eyes, "big tech", or even just for their peace of mind. Dexios was designed and created in spite of a seemingly oversaturated market for file encryption software. There was a large lack of user-friendly, fast, and safe tools to get the job done - and Dexios aims to fulfill this role.

2 Modes of Operation

This section describes all of Dexios' modes of operation, including everything from encrypting a file to unpacking an encrypted archive.

2.1 Encryption

Encryption uses the LE31 Stream object provided by the aead crate in order to encrypt data in "blocks". This block size is 1MiB. A 32-byte master key, two nonces, and one salt are generated.

The user's password is hashed, and this hashed password is used to encrypt the master key (with one of the generated nonces). The master key is encrypted using the same encryption algorithm that was selected to encrypt the data.

The header is serialised and written to the file. Next, the source file is read (in blocks of 1MiB), encrypted, and then written to the output file. With each encrypted block, the first 32 bytes of the header are authenticated as AAD. After each iteration, the source data is destroyed with the use of our protected wrapper, and the zeroize crate.

2.1.1 File-size Edge Cases

Edge cases regardigng very specific file sizes have been checked and confirmed to be working.

The major concern was when a file contains bytes exactly divisible by 1048576, but this is not an issue.

To test this, a file of exactly 3145728 bytes (3 blocks) was created. We edited Dexios to show us the read count when encrypting, and it read the file a total of 4 times. 3 times for each block, and then a final 0 byte read at the end. The 0 byte read was encrypted, adding an additional 16 bytes to the end of the file.

The final size of the encrypted file was 3145856 bytes:

- 64 bytes for the Dexios header (header version 3)
- (16×3) bytes for the AEAD tag (16 additional bytes per block)
- 16 extra bytes for the final encrypt

Another 16 bytes in file size is a fair trade-off to prevent any nasty edge cases such as this, so this functionality still remains.

2.1.2 Detached Mode

Encrypt mode allows for the user to encrypt in "detached" mode. This is where the header is written to a different location from the encrypted data, and no space is left at the start of the file. Due to the lack of empty space at the start of the file, the header can not be re-attached with the use of the header restoration command (8.4.2).

This provides a few advantages, the main one being that the data is both unrecognisable and undecryptable without the header file. A user could very easily store the header files on their own computers (ensuring they make a few backups), and upload just the encrypted data to a cloud storage provider.

2.2 Decryption

Decryption uses the LE31 Stream object provided by the aead crate, in order to decrypt data in "blocks".

Once the header has been read and deserialized, Dexios must obtain the encrypted master key, which is done by iterating through all available keyslots. With each iteration, the user's password is hashed with the salt provided in the keyslot, before attempting to decrypt the master key. If no iteration is successful, Dexios alerts the user that the provided key must be incorrect.

If the master key decryption is successful, the source file is read in blocks of 1MiB. Each block is decrypted, written to the output file, and securely erased from memory. With each block, the first 32 bytes of the header are authenticated with the use of AAD. If the header has been changed at all, decryption will fail and the user will be alerted. This continues until the end of the file.

2.3 Packing

Packing is an extension of encryption (2.1).

The source directory is "walked", and an index of all files within is created. Dexios then creates a temporary zip container, and copies all of the source files into it. Once this has been created, encrypt mode (2.1) is used to encrypt the file, and then erase mode (2.5) is called to remove our temporary file. One erasure pass is used here to prevent unnecessary wear on solid state drives.

ZSTD compression is available, but the user must opt-in for this. We chose to allow compression here, despite compression-with-encryption being discouraged in most cases. In the case of offline, cold file storage, there are almost no attacks that an attacker use to gain information over the file's contents. It is very unlikely that an attacker has control over the source information, which is often the key for compression-with-encryption attacks.

The file created with pack mode is just an encrypted zip archive. Users are free to use decrypt mode (2.2) to retrieve the zip archive contained within. They are then able to extract this file with the use of external tools.

2.4 Unpacking

Unpacking is an extension of decryption (2.2).

The source file is first decrypted into a temporary zip archive. Dexios then extracts this zip archive to the target directory provided by the user. Once this has successfully completed, erase mode is used (2.5) to remove the temporary file. One erasure pass is used here to prevent unnecessary wear on solid state drives.

Zip-slip protections have been put in place in order to prevent potential exploits against this mode.

2.5 Erasing

Erasure requires the number of "passes" to overwrite a file. It first opens the file in rw mode, and overwrites the entirety of the file's contents with random data. It does this n times, where n is the number of passes provided. As a final measure, it then goes over the entirety of the file's contents with zeroes, before removing the file with the operating system's default method.

It should be noted that this mode will NOT be effective on solid state drives, due to their wear-levelling features.

This mode is used for a few functions internally within Dexios - namely for erasing temporary archives created during the pack and unpack processes.

2.6 Hashing

BLAKE3 is the only supported hashing algorithm for this mode. The source file is read in "blocks" (1MiB in size), and on each iteration it is passed to a BLAKE3 hash object. This continues until the end of the file. Once the end has been reached, the BLAKE3 hash is finalized, and displayed in hexadecimal format. This format is consistent throughout all modes within Dexios.

3 Key Operations

Dexios is able to modify the keylots attached to a file due to the implemented header format. These keyslots are not included within the AAD, and can be edited freely.

3.1 Key Changing

Changing a key is done by decrypting the master key, and re-encrypting it with the user's new desired key. A new salt and nonce are used. The old keyslot is removed, and the new one is added to the header, and the header is written to the start of the file.

3.2 Key Addition

Adding a new key to a file is done by decrypting the master key, and re-encrypting it with the user's new desired key. A new salt and nonce are used. The keyslot is then appended to the header, and the header is written to the start of the file.

3.3 Key Deletion

Key deletion requires the key that the user would like to remove - this is so Dexios can identify the target keyslot, and remove it from the header accordingly. Key deletion works by retrieving the keyslot index that matches the user's old key, deleting it from the header, and writing this newly-modified header to the start of the file.

4 GitHub and Pull Requests

On the main GitHub repository, there are only two collaborators. We keep this number low in order to reduce the chance that a malicious actor can tamper with the codebase.

Before a pull request is accepted, it undergoes a rigorous series of tests. Each line of code is inspected, to ensure that nothing malicious, insecure or undefined has been added. We use GitHub Actions to ensure that pull requests do not break the core functionality of Dexios.

4.1 Automatically Compiled Binaries

With the use of GitHub actions, we are able to distribute a set of compiled binaries for many operating systems. These are built automatically, and are uploaded to the releases page. Each release contains a SHA256 hash of the binary, which users can cross-reference to ensure their file is correct.

5 Cryptography

5.1 Password Hashing

Dexios opts to use memory-hard password hashing algorithms, due to their resistance to brute-force attacks. With these algorithms, even a weak password will provide a rather high level of security.

5.1.1 Hashing Algorithms

Dexios offers both argon2id, and BLAKE3-Balloon for password hashing. This section refers to the latest provided parameters by Dexios v8.8.1.

argon2id uses 256MB of memory, and is using the parameters $m = 262114, t = 10, p = 4$.

BLAKE3-Balloon uses 64MB of memory, and is using the parameters $s = 278528, t = 1, p = 1$.

Both of the selected password hashing algorithms are deemed secure, and are more than adequate for this task. The memory cost was kept moderately high to protect against brute-force attacks with the use of ASICs. The higher memory cost greatly reduces the amount of concurrent attacks an attacker can run, due to the nature of memory-hard password hashing algorithms.

BLAKE3-Balloon was selected as the default password hashing algorithm for a few reasons. It is provably memory-hard[5], and the Balloon specification itself brings up a few flaws with argon2 (particularly argon2i). Argon2id is still assumed to be safe, which is why Dexios still offers it, but Balloon hashing seemed like the better choice in this case.

5.1.2 Performance

Both argon2id and BLAKE3-Balloon using the latest provided parameters are able to hash a password in roughly one second (with a Ryzen 7 3700x). A one second duration was chosen as it is a fair balance between performance and security.

5.2 Encryption Algorithms

Dexios has support for two encryption algorithms: XChaCha20-Poly1305 and AES-256-GCM. This information is stored within the header, so that Dexios can choose the appropriate algorithm for decryption. More information can be found in chapter 8.1. The implementations for both of these algorithms have been audited by the NCC group[3].

XChaCha20-Poly1305 is the default encryption algorithm, as it offers both good performance, and great security (even on devices that lack encryption hardware extensions). XChaCha20-Poly1305 also allows for extremely large message sizes before security decreases.

AES-256-GCM was chosen to be added as a secondary choice, for users who don't feel comfortable with XChaCha20-Poly1305. NIST recommend AES-GCM[1], so it's understandable that some users would like the option.

Deoxys-II-256 was also initially selected, but due to performance issues upstream, it was decided to deprecate encrypting files with it. Decryption is still possible for all files using Deoxys-II-256.

5.3 Threat Model

Dexios aims to protected against even the largest of threats, including state-sponsored actors. It does this with the use of modern AEADs and strong password hashing algorithms. No information regarding the contents of the file is available to prying eyes, making Dexios great for cloud storage applications.

5.4 Key Inputs

Dexios has support for 3 different key types: a password, a keyfile, and passphrase-autogeneration. All of these are wrapped in the protected wrapper type (5.5.1), and are only ever read when required. They are securely erased from memory once no longer in use. All keys are passed directly to the password hashing function, and undergo no additional "treatment" beforehand.

5.4.1 Passphrase Autogeneration

Passphrase autogeneration offers an easy way for a user to create a secure, and memorable passphrase to protect their files. The construction is as follows:

```
{word1}-{word2}-{word3}-iiiiii
```

Each word is capitalised, and *iiiiii* refers to 6 random digits. This feature uses the "Large Wordlist" provided by the EFF[4].

5.5 Cryptographic Hygiene

Dexios ensures that all sensitive information is securely erased from memory, with the use of the zeroize crate and the protected wrapper type (5.5.1). This wrapper is used for passwords, encryption keys, and even data that has yet to be encrypted.

5.5.1 Protected Wrapper

Dexios uses a protected wrapper, in order to prevent the leakage or accidental copying of sensitive data. The protected wrapper implements "zeroize-on-drop", meaning once it is no longer in use by Dexios, it is securely erased from memory in order to maintain cryptographic hygiene.

Copying (implicit clones) are fully forbidden with the use of the protected wrapper, and only explicit clones are allowed. This is to keep things auditable, maintainable, and prevent unwanted behaviour within Dexios.

The only way to access the information held within this wrapper is to call .expose() on the value, which further keeps things auditable. A quick search through the codebase can show where expose is called on secret data.

The protected wrapper type also ensures that information will be redacted in any logs.

5.6 RNGs

ThreadRng (provided by the rand crate) was chosen for all random value generation. This RNG is used for: nonce generation, salt generation, master password generation, and anything that you'd expect securely random values to be used. ThreadRng offers some additional protection over the standard StdRng[2], and obtains "randomness" from the system's entropy sources.

6 Dependencies

Dexios aims to keep dependencies at a minimum, in order to reduce both the compiled binary size, and the attack surface of the application.

For any cryptographic functions, RustCrypto's "crates" were chosen due to their reliability, attention to detail and audits[3]. The BLAKE3 crate is not provided by RustCrypto, but it does implement the same traits which allows easy integration within the ecosystem. It is clear that these libraries were developed with security in mind.

A definitive list of all dependencies, and some reasoning as to why they were chosen, can be found below. Dependencies marked with a * are provided by the RustCrypto team.

aes-gcm*	provides the AES-256-GCM AEAD
chacha20poly1305*	provides the XChaCha20-Poly1305 AEAD
argon2*	provides the argon2id password hashing algorithm
balloon-hash*	provides the Balloon password hashing algorithm
zeroize*	to erase sensitive information from memory
aead*	contains traits related to the AEADs
deoxys*	it provides Deoxys-II-256, for decryption only
blake3	for Balloon hashing, and file checksums
zip	for pack and unpack modes
rand	for random value generation
indicatif	used by the core crate for a progress spinner
anyhow	allows for fast, customisable error messages
rpassword	allows for hidden terminal password entry
clap	command-line argument parsing
walkdir	allows for directory walking, used by pack mode

7 Supported Hardware and Operating Systems

As Dexios is written in Rust, it is able to run on a wide variety of both hardware and software configurations. As of v8.8.1, this includes: Linux (both glibc and musl), MacOS, Windows (10 and 11), FreeBSD and even Android (with the use of Termux, a terminal emulator and Linux environment app). These are the only supported operating systems, but there have been reports of Dexios running in more obscure environments.

8 Headers

The total length of the header is 416 bytes, and this includes all information required for decryption, and 4 keyslots.

8.1 Header Structure

The main header structure is as follows:

```
{Identifier | Version | Encryption Algorithm |
Encryption Mode | Nonce | Padding | Keyslot Area}
```

The Identifier is 0xDE (in hexadecimal format).

The (header) Version, as of Dexios v8.8.1, is 0x05 (in hexadecimal format).

The Encryption Algorithm can be either [0x0E, 0x01] (for XChaCha20-Poly1305), [0x0E, 0x02] (for AES-256-GCM), or [0x0E, 0x03] (for Deoxys-II-256).

The Encryption Mode can be either [0x0C, 0x01] for STREAM mode, or [0x0C, 0x02] for "memory" mode. Memory mode has been deprecated for file encryption, but files using it may still be decrypted.

The Nonce is a cryptographically-secure random value, and its length varies based on the selected encryption algorithm.

The Padding consists of zeroes, up until the header reaches 32 bytes in total length. The amount of padding varies on the length of the nonce.

It was decided to not include any indentifier related to whether or not the file was packed. This is to prevent exploitation against users of Dexios - allowing for automatic unpacking could be a very bad idea, and it's recommended that users only directly unpack files that they are sure haven't been tampered with (this can be done with Dexios' hashing functionality).

8.2 Keyslots

After the initial 32 bytes of the header have been constructed, we then contstruct the keyslots. Each header has the possibility to hold 4 keyslots, at 96 bytes each.

The structure of each keyslot is as follows:

```
{Identifier | Master Key | Nonce |
Padding1 | Salt | Padding2}
```

The Identifier is 0xDF (in hexadecimal format).

The Master Key is 48 bytes in length, and is encrypted.

The Nonce is a cryptographically-secure random value, and its length varies based on the selected encryption algorithm.

The Padding1 consists of zeroes, up until the keyslot reaches 74 bytes in total length.

The Salt is 16 bytes in length, and is a cryptographically-secure random value.

The Padding2 consists of zeroes, up until the keyslot reaches 96 bytes in total length.

8.3 AAD

The first 32 bytes of the header are authenticated with each encrypted block of data, and are authenticated during decryption. If this check fails, it's safe to assume the header has been tampered with or corrupted, and the user is alerted of this. The file will not be decrypted. Only the first 32 bytes are authenticated, as the rest of the header contains keyslots for the file (and these may change at any time).

8.4 Header Operations

As Dexios uses a standard header format, it is able to apply operations that wouldn't have been possible otherwise. These operations are detailed in this section.

Please note, this section refers to Dexios v8.8.1, and header version 5.

8.4.1 Header Stripping

The source file is opened in *rw* mode, and the header is deserialized. If deserialization fails, the user is alerted and stripping will not continue. On successful deserialization, the header length is then calculated. The start of the file, up until length n, is then replaced with zeroes, with n being the length of the header.

8.4.2 Header Restoration

The source file containing the header is opened, and the header is deserialized. If deserialization fails, the user is alerted and restoration will not continue. On successful deserialization, the header length is then calculated. The target file is opened in *rw* mode, and Dexios checks that there is enough empty space at the start of the file for the header. If this check fails, restoration will not continue. If it is successful, the header is then written to the start of the target file.

Files encrypted with "detached" mode (2.1.2) can not have their header restored with this command. Users are able to manually re-attach the header, but this is not something Dexios supports.

8.4.3 Header Dumping

The source file containing the header is opened, and the header is deserialized. If deserialization fails, the user will be alerted and dumping will not continue. In the event that it is successful, Dexios will write the header to the target (output) file.

9 Conclusion

It is more crucial than ever that people have access to encryption software - whether that's for their files, messages, or even email. The internet is under ever-increasing surveillance, by both corporate and state actors. Intelligence agencies are always on the hunt for information. Using encryption software will slow this down, or stop it completely.

Through the use of modern, authenticated and audited encryption libraries (plus the help of a standardised header format), Dexios is able to provide users with confidence that their data will be safe from prying eyes, no matter their budget.

References

[1] https://csrc.nist.gov/projects/block-cipher-techniques/bcm

[2] https://rust-random.github.io/rand/rand/rngs/struct.ThreadRng.html

[3] https://research.nccgroup.com/2020/02/26/public-report-rustcrypto-aes-gcm-and-chacha20poly1305-implementation-review/

[4] https://www.eff.org/document/passphrase-wordlists

[5] https://eprint.iacr.org/2016/027.pdf