SMOL

A Dictionary Compression and Symmetric-Key Encryption Protocol

Most compression schemes use a form of repetition compression, which relies on the prevalence of certain sequences of text, and then converts them to multibyte characters for compactness. A lot of these compression mechanisms will lose their compression if you try to straightforwardly convert the compressed text to a simpler format (like Base-64 or Hexadecimal), and the compression rates can vary largely from nothing at all or a compression that is underwhelming, to anything else. This is a problem SMOL seeks to solve, for language-based information. Because language has a lot of internal structure, we can massively compress information using a dictionary-style compression, which associates binary codes to words. SMOL can compress English text around 60%, shrinking it to 40% of its original size, while the end-result of the compression is in Base-64, allowing for potentially greater compression if the binary data is encoded using Base-64 instead of UTF-8.

The target userbase of SMOL is anyone working with data where storage is a constraint, such as saving text in a distributed database or a blockchain. With 60% compression rates, fast compression times, and optional encryption, SMOL offers a free and open-source product that can massively boost the operating efficiency of these kinds of services.

SMOL Compression

The compression algorithm used in SMOL is a relatively straightforward dictionary compression algorithm, which makes certain tradeoffs to achieve the best compression rates possible. In the circumstance that a compression is unsuccessful or yields more data in the form of bloat, the compression algorithm will not be used, and a substitute Base-64 conversion algorithm is used to convert the text into an encryption-friendly format. If no key is given then no encryption occurs, and if no encryption occurs then strings will not be coerced into a Base-64 format. The worst possible compression rate any message could get is around -15%, and more if compressing multibyte characters. If encryption isn’t used then its impossible for compression to make your message any larger, excluding the three bytes added to every message as a form of version control.

The underlying compression mechanism involves taking the most common ~8000 english words, and assigning each one a 13-bit binary sequence. Some space is also delegated to suffixes and general “pieces” of words, so that unfamiliar English text bloats the message significantly less, and high compression rates can be maintained. A sequence is also delegated to accessing a second 13 bit list, where emojis and multibyte characters are stored, so that they can be compressed and encrypted too. Multibyte characters and emojis are always going to bloat a message.

Certain assumptions are made about each message that allows for significantly better compression, without making any assumptions that would break someone’s message. For example, its assumed all words are preceded by a space, the beginning of every sentence denoted by a closing punctuation mark and a space is capitalized, the beginning of every message is capitalized, and every other word is uncapitalized. To write a message that doesn’t follow these rules just means that the compression algorithm has to specify otherwise, which will bloat your text. In the rare circumstance that a message is bugged and doesn’t perfectly decompress, the compression is not utilized, preserving the integrity of the original message. Most bugs of this sort would require willful attempts by a user to escape the compression algorithm using different command codes utilized by it.

SMOL ENCRYPTION

The encryption algorithm in SMOL is a four-step process:

1) A cipherwheel representing all Base-64 characters is shuffled using a password-based shuffle. The shuffle process is described in Step 4.

2) The message is split in two, each side is used to encrypt the other side using the described process in steps 3-4, it is then shuffled with a password-based shuffle (including the nonce), and then this step is repeated once.

3) The message is “rotated”, which means that each substring of the string is rotated along the cipherwheel seeded with the password (and nonce) and the location of the substring. A nonce, if used, is appended to the front.

4) The message is shuffled using a password-based shuffle. This is done by swapping each substring of a string with a seemingly random substring, seeded with the password.

The rotation of the text will completely and pseudorandomly change the frequency of the characters in the text, and the two shuffling steps make it impossible to identify the relative rotations of each character in the message. Keep in mind, this is not like a Caesar cipher, because Caesar ciphers have uniform structure where every character in the message is rotated in the same way. This is also not like a Vignere cipher where each character is rotated the same way for every other character of that type, but rather, every substring of a message is rotated regardless of what character it is, which is a far more secure model.

There is an optional parameter for nonces, for those who want messages to be extra secure. It is recommended to use a nonce if you use the same password every time, or especially if you are also encrypting the same message multiple times. Nonces add random data to the password in exchange for some space being used up in the message.