VisualCrypt AG

October 4, 2015

Announcement of the VisualCrypt 2 Encryption Standard (Draft)

 At the time of writing, only the VisualCrypt 2 software library and the VisualCrypt implement the VisualCrypt 2 Encryption Standard. The purpose of the publication is to allow for the documentation and public evaluation of the methods used within the VisualCrypt 2 software library and the VisualCrypt 2 message data format. The document is published under the GNU GPL v3 and additional provisions that explicitly disallow criminal use of our products.

1. Name of Standard. VisualCrypt 2
2. Category of Standard. Computer Security Standard, Cryptography
3. Description. VisualCrypt 2 specifies a method of employing several well-known cryptographic algorithms to protect electronic data by encryption. VisualCrypt 2 is a symmetric encryption scheme that uses the Advanced Encryption Standard (AES) cryptographic algorithm in its core to encrypt and decrypt Unicode Plaintext messages using a Unicode password. The cipher text is further encoded into the ASCII-based VisualCrypt 2 Message Format to allow for user-friendly and software-independent storage and network transmission of the encrypted messages or notes.
4. Applicability. The standard is used by the VisualCrypt line of applications and may be used by implementers who want to create their own compatible applications, especially when based on the publicly available source code of the VisualCrypt libraries and VisualCrypt applications that have been published under the GNU GPL v3. Any derivative works that refer to the VisualCrypt or to the original source code brand MUST conform to the standard.
5. Specifications. VisualCrypt 2 Specification, included in this document.
6. Implementations. The current implementation is provided by VisualCrypt in source code form on Github and in object form on the VisualCrypt website.
7. Schedule. The standard is currently in draft status and under review.
8. Patents. VisualCrypt 2 itself does not invent or implement functionality that would be deemed patentable under European patent law. However, the third-party algorithms employed in VisualCrypt 2, such as AES, may be covered by U.S. and international patents.
9. Export control. Depending on the country, export and/or import control regulations may apply.

VisualCrypt AG

October 4, 2015

Specification for the VisualCrypt 2 Encryption Standard (Draft)

Contents

1 Introduction

This standard specifies the VisualCrypt 2 algorithm and the VisualCrypt 2 message format.

The VisualCrypt 2 algorithm is used to transform a plaintext message into a ciphertext message using a password. VisualCrypt 2 is an application of well-known cryptographic algorithms. It uses the Advanced Encryption Standard (AES) (see [1]) algorithm with a key length of 256 bits to perform symmetric encryption. The hash algorithms BCrypt (see [2]) SHA-512 (see [3] chapter 6.4) and SHA-256 (see [3] chapter 6.2) are used for slow and entropy-preserving key derivation. Deflate (see [4]) is used for plaintext compression. Ciphertext is encoded into encoded into a custom base-64-based message format.

A BCrypt hash of the password is used to encrypt the 256-Bit random encryption key, which uses 256-Bit AES to encipher the plaintext message into the ciphertext message. The ciphertext message is a byte stream that is always serialized into the VisualCrypt 2 message format, which consists solely of ASCII characters, providing a 'visible' representation of the ciphertext that is easy to handle with common methods like copy/paste, viewable with any text editor and transmittable via text messaging services.

The VisualCrypt 2 encryption algorithm tries to work around some known problems of password-based encryption schemes such as weak passwords, dictionary-based brute force attacks and known plaintext. It also includes an encrypted 128-Bit message hash to protect against message forgery.

The VisualCrypt 2 algorithm provides a user-configurable slowness to reduce the efficiency of brute-force attacks. An exponential factor 2^n is used to configure how much 'Rounds' both the BCrypt hash algorithm and AES should take enciphering the message. While this is a common feature of BCrypt, the same work factor is also applied on AES, so that every plaintext message is encrypted 2^n times.

Plaintext is compressed before encryption using the Deflate algorithm. The compressed plaintext message is padded before encryption with 1 to 15 random bytes. AES-256 Bit CBC with padding mode 'none', a 128-Bit initialization vector (IV), and a 256-Bit key from the system's random number generator is then used to encrypt the message 2n times using the same parameters. The IV and the padding amount, the exponent in 2^n, the encrypted hash and the algorithm version is stored together with the ciphertext message's serialized form, the Base64-based VisualCrypt 2 message format.

This specification provides the details to the algorithm and message format so that an implementation of VisualCrypt 2 can be made without considering existing implementations.

2 Definitions

2.1 Terms

Term	Description
Unicode Character	A character defined by the Unicode® Standard (see [9])
Code Unit (Unicode)	The minimal bit combination that can represent a unit of encoded text (see [10] chapter 3.9, D77).
UTF-8	The Unicode encoding form that assigns each Unicode scalar value to an unsigned byte sequence of one to four bytes in length (see [10] chapter 3.9, D92).
UTF-16	The Unicode encoding form that assigns each Unicode scalar value in the ranges U+0000..U+D7FF and U+E000..U+FFFF to a single unsigned 16-bit code unit

	with the same numeric value as the Unicode scalar value, and that assigns each Unicode scalar value in the range U+10000..U+10FFFF to a surrogate pair (see [10] chapter 3.9, D 91).
AES	Advanced Encryption Standard (see [1])
BCrypt	BCrypt Hash Algorithm (see [2])
Initialization Vector, IV	Synonyms: Salt, nonce. A random start value e.g. used to protect block ciphers from chosen plaintext attacks.
Block (Block Cipher)	A data block of a certain length (16 bytes for AES) that constitutes a suitable input length for the algorithm.
Datatype	A classification, definition and semantic description of a type of data.
Ciphertext	The encrypted plaintext.
Plaintext	Input of the encryption algorithm, also called Cleartext when referring to unprotected data.
Round	A repetition of the application of an algorithm on input data.
Rounds Exponent	Refers to the exponent n in 2^n when notating the total amount of rounds in terms of 2 to the power of n.
object	An object as defined in object-oriented programming.
string	A sequence of 2-byte UTF-16 code units. Length is measured in code units.
bit	A binary digit with a value of 0 or 1.
byte	A value of eight bits.
byte[]	byte array, enumerable sequence of bytes.
byte[n]	A byte array with the length of n.

2.2 VisualCrypt 2 Data Types

For conciseness, this standard uses named data types to describe more general underlying data types such as byte, byte array, string or aggregates.

VisualCrypt 2 Data Type	Description	
BCrypt24	A 24-byte BCrypt hash.	byte[24][1]
CipherV2	Object containing the enciphered message body and all metadata required for decryption.	object with data fields, see 3.3
Cleartext	The human-readable plaintext consisting of Unicode characters encoded in UTF-16 LE format.	string[2]
Compressed	Deflate-compressed *Cleartext*.	byte[][1]
IV16	The 16-byte initialization vector for AES and BCrypt.	byte[16] [1]
MAC16	BCrypt hash of *MessageCipher*, *PlaintextPadding* and *CipherV2.Version*.	byte[16] [1]
MACCipher16	*MAC16* encrypted with *RandomKey32*.	byte[16] [1]
MessageCipher	The ciphertext body without metadata.	byte[][1]
NormalizedPassword	Normalized form of the user-entered password.	string, length 0..10^6

PaddedData	*Compressed* expanded with *PlaintextPadding*.	byte[][1]
PasswordDerivedKey32	The 256-Bit AES encryption key for the encryption of *RandomKey32*. Derived from *SHA512PW64*, *IV16* with BCrypt using *RoundsExponent*.	byte[32] [1]
PlaintextPadding	Amount of n random padding bytes added to *Compressed.*	byte/ushort [0..15]
RandomKey32	The 256-Bit AES encryption key for the encryption of *PaddedData* and *MAC16*. Generated by the computer's random number generator.	byte[32] [1]
RandomKeyCipher32	*RandomKey32* encrypted with *PasswordDerivedKey32*.	byte[32] [1]
RoundsExponent	The base-2 exponent for the number of BCrypt and AES rounds.	byte/ushort [4..31]
SHA512PW64	SHA-512 hash of *NormalizedPassword*.	byte[64] [1]
VisualCryptText	Encoded VisualCrypt 2 ciphertext message.	string[2]
[1)] implementations may disallow byte arrays with all-zero byte values [2)] implementations may restrict the maximum allowed string length		

2.3 Notation and conventions

The following notation and conventions are used throughout this document.

- When referring to VisualCrypt 2 data types (2.2), italic text is used, e.g. *RandomKey32*.
- The description of the encryption and decryption procedures uses pseudo-code notation, e.g. *RandomKey32* = aesDecrypt(*RandomKeyCipher32*, *PasswordDerivedKey32*, *IV16*, *RoundsExponent*). This means that the data type RandomKey32 is the result of a function that takes decrypts data with AES, using *RandomKeyCipher32*, *PasswordDerivedKey32*, *IV16* and *RoundsExponent)* as parameters.

3 Message format specification

3.1 Encoding Format

A serialized VisualCrypt 2 message has the following encoded format:

VisualCrypt/[Base64]

'VisualCrypt/' is a mandatory case-sensitive string.

Base64 is composed of base-64 digits, a trailing padding character and line break characters. The base-64 digits in ascending order from zero are the uppercase characters "A" to "Z", lowercase characters "a" to "z", numerals "0" to "9", and the symbols "+" and "$". The character "=" is used for trailing padding and can appear zero to two times. The line break characters are newline (0x0A) and carriage return (0x0D). When reading, all characters other than the base-64 digits, the padding character and the 'VisualCrypt/'-prefix must be ignored.

To facilitate interoperability of implementations, when written to a file, either UTF-8 without byte order mark or ASCII encoding must be used. The filename may contain Unicode characters and must end with '.visualcrypt'.

3.2 Serialization format

The base-64 part of the encoded VisualCrypt 2 message is base-64-decoded into a single byte array, starting with the first base-64 after the 'VisualCrypt/'-prefix and ending with the last base-64 digit or padding character supplied. The resulting byte array is comprised of seven ordered parts where each part can be located according to the following schema:

Start Index	Length	Part Name	Data Type	Description
0	1	Version	byte	Version, always 2
1	1	RoundsExponent	byte	Rounds exponent, 4-31
2	1	PlaintextPadding	byte	Plaintext padding bytes, 0-15
3	16	IV16	byte[]	AES/BCrypt Initialization Vector
19	16	MACCipher16	byte[]	Encrypted hash of securables
35	32	RandomKeyCiper32	byte[]	Encrypted random AES key
67	total-67	MessageCipher	byte[]	Message cipher

3.3 Object format

A VisualCrypt 2 message is represented in memory as an object, named CipherV2.

CipherV2
Version: byte RoundsExponent: RoundsExponent PlaintextPadding: PlaintextPadding IV16: IV16 MACCipher16: MACCipher16 RandomKeyCipher32: RandomKeyCipher32 MessageCipher: MessageCipher

Properties of CipherV2 other than Version are semantically named data types that encapsulate byte arrays.

4 Algorithm specification

The VisualCrypt 2 algorithm for encryption, decryption and key derivation is defined in terms of the transformations of named data types.

4.1 Encryption

Encryption is the complete process of transforming human-readable cleartext to encoded ciphertext (3.1).

4.1.1 Encryption inputs

The inputs to the cipher are *Cleartext*, *SHA512PW64* and *RoundsExponent*. Cleartext is the plaintext contents of the message as string with an assumed UTF-16 Unicode encoding. *SHA512PW64* is a SHA-512 hash of the preprocessed user password (see 4.3 Password preprocessing). RoundsExponent is the user-chosen 'work factor' as known from BCrypt with possible ranges from 4 to 31, meaning $2^{\text{RoundsExponent}}$ repetitions will be applied to the cryptographic operation, with the main goal to slow it down by scaling the up the computational work to perform it. In VisualCrypt 2, the idea is reused and

applied as multiple rounds of AES encryption, using the very same value for the both number of BCrypt and AES rounds.

4.1.2 Encryption procedure

The high-level sequence of the encryption procedure in terms of the datatype transformations is:

1. *Compressed* = compress(*Cleartext*)
2. *PaddedData*, *PlaintextPadding* = applyRandomPadding(*Compressed*)
3. *IV16* = generateRandomBytes(16)
4. *PasswordDerivedKey32* = createPasswordDerivedKey(*SHA512PW64*, *IV16*, *RoundsExponent*)
5. *RandomKey32* = generateRandomBytes(32)
6. *RandomKeyCipher32* = aesEncrypt(*RandomKey32*, *PasswordDerivedKey32*, *IV16*, *RoundsExponent*)
7. *MessageCipher* = aesEncrypt(*PaddedData*, *RandomKey32*, *IV16*, *RoundsExponent*)
8. *MAC16* = computeBCrpyt((*MessageCipher* + *PlaintextPadding* + *Version*), *IV16*, *RoundsExponent*)
9. *MACCipher16* = aesEncrypt(*MAC16*, *RandomKey32*, *IV16*, *RoundsExponent*)
10. *CipherV2* = *Version* + *RoundsExponent* + *PlaintextPadding* + *IV16* + *MACCipher16* + *RandomKeyCipher32* + *MessageCipher*
11. *VisualCryptText* = serializeAndEncode(*CipherV2*)

4.1.2.1 Compression

Compression is performed using the DEFLATE algorithm developed by Phil Katz for PKZIP 2 (see [4]). Deflate has been specified in RFC 1951 with implementations available for many platforms. Deflate is an attractive choice for VisualCrypt because of its relatively small header size, thus saving space and avoiding a large characteristic fingerprint in the plaintext message.

For compression, the UTF-16 *Cleartext* string is converted into an UTF-8 byte array, allowing for space-efficient encoding of Latin character sets. The UTF-8 bytes are then processed by the DEFLATE algorithm. The resulting compressed byte array is stored in the data type named *Compressed*.

4.1.2.2 Padding

Padding the plaintext is required when an encryption algorithm assumes the input length to be a multiple of k bytes as is the case with AES, which has a block size of 16 bytes. Then, the input must be expanded with k - (l modulo k) bytes, where l is the length of the input. This is normally done at the trailing end. While the AES specification excludes padding, there are a number of established methods how it can be performed, such as PKCS#7 [5], ANSI X.923 [6], ISO 10126 [7] or zeros. VisualCrypt uses random bytes for padding and the number of bytes added is stored in the CipherV2 object. The rationale is to avoid known-plaintext-attacks as good as possible. As the padding is appended to the compressed plaintext, which already has a high degree randomness, adding another 1 to 15 random bytes should not leak much information to the attacker, even if the amount of padding is known. In contrast to this, PKCS#7 padding is highly discoverable when appended to compressed data but it has the practical advantage that the amount is detectable from its pattern. However, in the context of encryption this is also a disadvantage because it makes it easy to *exclude* a given key in brute-force attacks because the padding will be invalid for the major part of wrong keys.

To perform the padding, the required amount is calculated by 16 – (*Compressed*.Lenght modulo 16) and stored in PlaintextPadding. If the amount is larger than zero, the required amount of random bytes is generated and appended to *Compressed*. The concatenated byte array is then stored in

PaddedData. To remove the padding, the amount of bytes stored in PlaintextPadding is removed from the byte array.

4.1.2.3 Initialization vector

AES and BCrypt are both used with a 16-byte random identical initialization vector *IV16* that is unique to each subsequent run of the encryption while the same password derived key *SHA512PW64* may be used. The *IV16* must be preserved for the decryption and is therefore stored in the *CipherV2* object. During the encryption, the *IV16* is obtained from the computer's random number generator. When decrypting, the *IV16* is read from the *CipherV2* object. Due to the multiple uses of IV16 in VisualCrypt 2, an incorrectly transmitted or forged IV16 will *always* lead to the rejection of the decryption result because both the calculation of *MAC16* and the decryption result of *MacCipher16* depends on it. A bit error in IV16 will also lead to a bit error in the first block of *RandomKey32*, thus leading to errors in *all* blocks of the decrypted message (see also [8] Appendix D) which will be reliably detected by the comparison of the actual message hash and the decrypted message hash.

4.1.2.4 Creating the password-derived encryption key

The preprocessed password stored in *SHA512PW64* (see 4.3 Password preprocessing) is the starting point to create the password-derived key *PasswordDerivedKey32* during each encryption process. The core of the password-derivation process is to hash the key material using BCrypt with the *IV16* and the BCrypt work factor *RoundsExponent*. Thus, the actual password-derived key (that will be used only for the encryption of *RandomKey32*) will differ for each encryption by using *IV16* as the salt, even if the password stays the same.

The goal during key derivation is to preserve as much cryptographic entropy as possible from the password, given that typical passwords already have a very low entropy compared to true-random keys anyway. The key material in *SHA512PW64* is the result of applying a SHA-512 hash to the preprocessed password chosen by the user. The 64-byte byte array is then split in its two halves. Each halve is then hashed with BCrypt and the common *IV16*, using the given *RoundsExponent*. BCrypt returns hashed with the length of 24 bytes each, so this operation results in 48 bytes of key material that must be condensed to 32 bytes to be useable for AES-256. To preserve entropy well, both 24-bit BCrypt hashes and the original SHA-512 hash are concatenated into one 112-byte byte array which is then hashed with SHA-256, resulting in the final 32-byte *PasswordDerivedKey32* for use with AES, that is slow to generate by the use of BCrypt with its configurable work factor.

4.1.2.5 Generating the random key

The AES specification [1], chapter 6.2, states that "no weak or semi-weak keys have been identified for the AES algorithm, and there is no restriction on key selection" as such. There are nevertheless other reasons why it can be advantageous to use random data as key material for the actual plaintext encryption and to encrypt the random key with the password-derived key. When running a brute-force attack on the encrypted key and all that is known about the encrypted key is that it is supposed to contain random data, no known-plaintext attack is possible in this step. Secondly, rainbows table with known encrypted plaintext, commonly used or short passwords and all possible initialization vectors (if that is feasible at all) will not be helpful because the relationship between the commonly used passwords and known plaintext is not present in any given ciphertext. Thus, using a secondary random key adds to the strength of the encryption.

However, the entire burden is shifted to the quality and integrity of the computer's random number generator (RNG). If the random numbers can be predicted or spoofed, any of the other security features in VisualCrypt 2 would be bypassed and useless because the random key leads directly to the plaintext. Therefore, implementations should test the RNG with statistical methods. Moreover, user awareness should be built for the problem and a checklist should be provided on what the user

can do to cover the many possible attacks, such as verifying the source code, the signature of the package and encrypting only on permanently offline systems – in the cases where an extraordinary degree of secrecy is deemed required.

RandomKey32 is created by calling the computer's 'true' random number generator with the amount of bytes needed, in this case 32 bytes to obtain a 256-bit encryption key for use with AES. *IV16* and the random padding bytes are generated in the same way, implying the same risks.

4.1.2.6 Encrypting the message

The compressed and padded message *PaddedData* is encrypted using the 256-bit random key *RandomKey32* and the common initialization vector *IV16*. Encryption is performed with AES algorithm in AES Cipher Block Chaining (CBC) mode (see [8] chapter 6.2). AES is used with 'PaddingMode.None' because a custom padding has been applied before (see 4.1.2.2). The block size of AES is 16 bytes. The AES algorithm is run two power *RoundsExponent* times where *PaddedData* is the plaintext input for the first round. The output bytes of one round is then used as 'plaintext' input for the subsequent round, all other parameters staying equal until the number of rounds reaches $2^{RoundsExponent}$. This increases the computational work for the encryption and, more importantly, the decryption significantly, as long as no way has been found to shortcut the AES algorithm in a way that the result of the $2^{RoundsExponent}$ decryption can be reached more directly than decrypting the ciphertext $2^{RoundsExponent}$ times. This technique will make brute-force attacks less feasible. After all rounds are completed, the multiple-enciphered message bytes are stored in *MessageCipher*, which is part of the CipherV2 message object.

4.1.2.7 Encrypting the random key

The random key *RandomKey32* is used in VisualCrypt 2 to encrypt the plaintext message *PaddedData* and the MAC *MAC16*. It is encrypted using PasswordDerivedKey32 and the common initialization vector IV16 using the same technique and AES configuration described in 4.1.2.6, including the multiple rounds of encryption. The result is stored in *RandomKeyCipher32*, which is part of the CipherV2 message object.

4.1.2.8 Computing MAC data

A 16-byte message authentication code *MAC16* (that is encrypted later) is computed in the form of a BCrypt hash over the concatenation of *MessageCipher*, *PlaintextPadding* and *CipherV2*.Version. *IV16* and *RoundsExponent* are the other inputs to the BCrypt algorithm. The 24-byte BCrypt hash is then truncated after 16 bytes. Hash truncation is a practice that has been used elsewhere (see e.g. [3] chapter 7) and 16 bytes are assumed sufficient to provide the desired degree of message integrity in VisualCrypt 2. Moreover, a less accurate hash tells the attacker less about the validity of the key used to decrypt the fitting hash, which makes brute-force more complicated because matching hashes do not sufficiently uncover the key used to encrypt the hash. In the VisualCrypt 2 algorithm, if it is found that the encrypted hash does not match the recalculated hash, the decryption is stopped with the conclusion "wrong password or corrupted/forged message". For the encryption of MAC16, see 4.1.2.9 below.

4.1.2.9 Encrypting the MAC

The MAC16 is encrypted with RandowKey32 and the AES algorithm similar to the encryption of the message (see 4.1.2.6) and the encryption of the random key (see 4.1.2.7). The 16-byte result is stored in *MACCipher16* and stored in the CipherV2 message object.

4.1.2.10 Parts of the encrypted message

The result of the encryption process is the CipherV2 object (see 3.3), which contains all the parts of the enciphered message that are needed for a subsequent decryption: *CipherV2*.Version, *PlaintextPadding*, *RoundsExponent*, *IV16*, *MACCipher16*, *RandomKeyCipher32* and *MessageCipher*.

4.1.2.11 Serialization and encoding the encrypted message

In the final step of the encryption process, the CipherV2 object is encoded to VisualCryptText that can be saved as an encrypted note or transferred as a message over the wire. See 3.2 for the serialization format and 3.1 for the encoding format of the CipherV2 object.

4.2 Decryption

Decryption is the complete process of transforming encoded ciphertext into human-readable cleartext.

4.2.1 Decryption inputs

The inputs of the decryption procedure are the hash *SHA512PW64* of the user-supplied password and *VisualCryptText*. *VisualCryptText* is the encoded ciphertext message as specified in 3.1 that has been loaded from the file system or received as a message. *VisualCryptText* is deserialized to obtain the *CipherV2* object (see 4.3) that contains the ciphertext body *MessageCipher* and the necessary metadata to allow for the decryption: *CipherV2*.Version, *PlaintextPadding*, *RoundsExponent*, *IV16*, *MACCipher16* and *RandomKeyCipher32*.

4.2.2 Decryption procedure

The high-level sequence of the decryption procedure in terms of the datatype transformations is:

1. *CipherV2* = decode(*VisualCryptText*)
2. *PasswordDerivedKey32* = createPasswordDerivedKey(*SHA512PW64*, *IV16*, *RoundsExponent*)
3. RandomKey32 = aesDecrypt(*RandomKeyCipher32*, *PasswordDerivedKey32*, *IV16*, *RoundsExponent*)
4. *MAC16* decryptedMAC = aesDecrypt(*RandomKey32*, *MACCipher16*, *IV16*, *RoundsExponent*)
5. *MAC16* actualMAC = computeBCrpyt((*MessageCipher* + *PlaintextPadding* + *Version*), *IV16*, *RoundsExponent*)
6. Continue processing = decryptedMAC equals actualMAC
7. PaddedData = aesDecrypt(*RandomKey32*, *MessageCipher*, *IV16*, *RoundsExponent*)
8. *Compressed* = removePadding(*PlaintextPadding*, *PaddedData*)
9. *Cleartext* = decompress(*Compressed*)

4.2.2.1 Decoding the enciphered message

The input of the decoding procedure is a string that may visually appear to be in the format specified in 3.1 but that does not mean it is valid *VisualCryptText*. If the formatter cannot construct a CipherV2 during the deserialization attempt, a format error is returned and the decryption procedure is aborted. As specified in 3.1, all characters of the input string that do not belong to the white list of characters of the message format are removed, including all line breaks added in the encoding process. After whitelisting, the input string must start with 'VisualCrypt/' and the following characters must be base-64-decodable into a byte array. The byte array must be decomposable to the parts specified in 3.2. If successful, a CipherV2 object is obtained and the decryption can proceed.

4.2.2.2 Creating the password-derived decryption key

The password-derived decryption key *PasswordDerivedKey32* must equal the password-derived encryption key, which is therefore obtained with the same procedure (see 4.1.2.4).

4.2.2.3 Decrypting the random key

The next step is to obtain the actual decryption key *RandomKeyCipher32* for the enciphered message *MessageCipher* and the enciphered MAC *MACCipher16*. To do this, RandomKeyCipher32 is AES-decrypted using PasswordDerivedKey32, IV16 and RoundsExponent, reversing the procedure of the encryption (see 4.1.2.7 Encrypting the random key), repeating the number of rounds.

4.2.2.4 Decrypting the original MAC

The encrypted MAC *MACCipher16* (see 4.1.2.9) is decrypted using *RandomKeyCipher32* to obtain the plaintext version of the original *MAC16* that was created during the encryption process.

4.2.2.5 Computing the MAC data for comparison

The *MAC16* of the message being decrypted is now computed in the same way as during the encryption in step 4.1.2.8, now using the parts from the CipherV2 object received.

4.2.2.6 Authenticating and ensuring integrity

The decrypted MAC16 and the actual MAC16 are now compared for equality. In case they are *not* equal, the decryption is aborted and the following interpretation is given to the user: "The password is incorrect or the message is corrupted or forged". Otherwise, the decryption proceeds and it is as likely that the password is correct, the message is authentic and data integrity is given as can be ensured by checking against a 16-byte message authentication code. The algorithm implementation must ensure that errors that might occur due to a fallacious decision to continue do not jeopardize system stability or leak data.

4.2.2.7 Decrypting the message

After successful authentication and integrity verification the enciphered message body *MessageCipher* is decrypted with *RandomKey32* using the AES configuration and reversed procedure from the corresponding encryption step specified in 4.1.2.6. The result is *PaddedData*.

4.2.2.8 Removing padding

The amount of random byte padding *PlaintextPadding* added during the encryption (see 4.1.2.2) that is stored in the *CipherV2* object is now being removed from the end of *PaddedData*. The result is the compressed plaintext message *Compressed*.

4.2.2.9 Decompression

In the final step of the decryption, *Compressed* is now DEFLATE-decompressed to obtain the human-readable plaintext string *Cleartext* of the message.

4.3 Password preprocessing

Essential for the success of a password-based encryption is the ability to decrypt an enciphered message with the same password across different versions of the programs and on different platforms. VisualCrypt 2 is designed to allow for more flexibility in the choice of passwords than traditional solutions:

- Passwords can contain Unicode characters (like in 'password', 'пароль' or '密碼').
- The allowed password length is zero to 1.000.000 UTF-16 chars.
- Passwords can be entered in a multi-line way and/or be formatted with whitespace.
- Formatted passwords that have been written down or printed should be reproducible by the user.
- As the choice of words is never random and even short passwords are allowed, VisualCrypt 2 makes an unseen effort to slow down brute-force password attacks or password dictionary attacks.
- VisualCrypt 2 features a routine that suggests random passwords that make use of the full 256 bits of cryptographic key entropy that AES-256-Bit enables.

This flexibility is a stark contrast to the traditional whitelisting of allowed US-ASCII password chars. In our opinion, this approach fits better into a globalized world and it assumes a mature, informed user who can decide on a case-by-case basis whether a very strong or a very weak and convenient password is required. However, it also requires additional effort and thoroughness in preprocessing

the password. And because of the complexity and ongoing specification of Unicode and differing implementations of the Unicode standard across platforms it cannot be ruled out that there are password choices possible that are irreproducible on a system other than the one where the encryption password was entered. Nevertheless, the offering of password entry in the user's native script seems to outweigh the risks for the purpose of VisualCrypt.

4.3.1.1 Password entry

VisualCrypt 2 expects a password string entered by the user or that has been pasted into a multi-line text field. The password string is expected to have Unicode encoding. The accepted string length in characters from zero to 10^6 is a deliberate choice. The contents of the password text field is called the *raw password*.

4.3.1.2 Password preprocessing

The raw password is converted into a normalized form, called *normalized password*, possibly reducing the effective length. The steps are the following:

1. Remove the Unicode characters that are Unicode control characters AND NOT Unicode whitespace characters.
2. Then condense and normalize the Unicode whitespace characters so that
    a. all Unicode whitespace characters are converted into the Space (0x20) character
    b. and all adjacent Space characters are replaced by a single Space character

This normalization of the raw password makes printing and re-entering the password easier because it removes control characters or whitespace that is not displayable or recognizable such as adjacent spaces.

If the normalized password is an empty string, the user should be asked for permission if he really wants to use an empty password.

4.3.1.3 Intermediate non-salted password hash

The normalized password is hashed immediately upon entry for secrecy. However, the password should be available without re-entry for multiple encryptions and decryptions, and the actual password-derived encryption key *PasswordDerivedKey32* is obtained from the BCrypt algorithm, which, according to this specification, must use a different initialization vector *IV16* for each encryption. Therefore an intermediate password hash SHA512PW64 is needed that can be stored until the password is cleared or changed. SHA512PW64 is a 64-byte SHA-512 hash of the normalized password that is then used as input for encryption step 4.1.2.4 and decryption step 4.2.2.2.

5 Test Cases

The following tables can be used as normative test cases to verify that an implementation conforms to the specification.

5.1 Sample 1

The given *Cleartext* has been encrypted with the password using a *RoundsExponent* of 13. The result is the ciphertext *VisualCryptText*.

Cleartext	Hello Secrets! ☺
Password	$Qxyg nEA4s FTCGu AloGv Cey5Z CmZIJ I2hk2 igq4P DVw
RoundsExponent	13

VisualCryptText	VisualCrypt/Ag0M0yuCs4s0oalNcBl$SQTafO$DV3b6qyk73RCQvYNdAIfMTsent CpsNhRK1K+KpyNYMDo5Qqlr5CRhW12+3VhuU92Gn9z4TFCt4Ey4lyaME4n1m kYuFNCpOi4iAeuF1k7y

The following table shows the internal intermediate values that occurred during the encryption process of the above example.

NormalizedPassword	$Qxyg nEA4s FTCGu AloGv Cey5Z CmZIJ I2hk2 igq4P DVw
SHA512PW64	4B 77 5B D0 FE BC F2 D3 DF 90 01 83 DF BF 37 0C 1F B0 7D BD D0 91 56 8F 01 CC A7 9E 33 E6 AC 1A CA 86 6A 66 48 FA D8 6B D1 30 FA 57 17 F9 96 DF B6 EA 30 9C 16 B0 77 31 99 08 24 62 C3 D2 16 A5
Compressed	F3 48 CD C9 C9 57 08 4E 4D 2E 4A 2D 29 56 54 78 34 63 17 00
PaddedData	F3 48 CD C9 C9 57 08 4E 4D 2E 4A 2D 29 56 54 78 34 63 17 00 FE C7 AC 17 04 C6 0A 10 95 6C 08 4F
PlaintextPadding	0C
IV16	D3 2B 82 B3 8B 34 A1 A9 4D 70 19 7F 49 04 DA 7C
PasswordDerivedKey32	AC E3 A0 BC A7 40 60 4A 47 8A 21 87 3F 4C 85 2C C7 A0 98 D9 1F 52 DC 23 5B 8E 91 70 05 52 87 BE
RandomKey32	73 A3 42 F5 D5 A6 66 47 3B 6C 84 99 72 76 B0 2E AF 0A E1 E9 BD 9B 38 7C C9 C7 2B 22 5A F4 22 B8
RandomKeyCipher32	CC 4E C7 A7 B4 2A 6C 36 14 4A D4 AF 8A A7 23 58 30 3A 39 42 A9 6B E4 24 61 5B 5D BE DD 58 6E 53
MessageCipher	DD 86 9F DC F8 4C 50 AD E0 4C B8 97 26 8C 13 89 F5 9A 46 2E 14 D0 A9 3A 2E 22 01 EB 85 D6 4E F2
MAC16	C2 77 20 C0 97 63 BD 16 7E 25 AA F4 30 6C AE 32
MACCipher16	EF C3 57 76 FA AB 29 3B DD 10 90 BD 83 5D 00 87

5.2 Sample 2

This sample uses the same *Cleartext*, password and *RoundsExponent* and shows another run, where a different *IV16* and different random padding bytes were generated. The tables include a remark whether the values are same or different from sample 1.

Cleartext	Hello Secrets! ☺ (same)
Password	$Qxyg nEA4s FTCGu AloGv Cey5Z CmZIJ I2hk2 igq4P DVw (same)
RoundsExponent	13 (same)
VisualCryptText	VisualCrypt/Ag0MxpllgaAlx0ITkmzY1TssVq1UZ4Rkom1GEEjPFn31AVp2QfdsfH mN5OjfviiMZNb2gI14NLtIyUfjURVqy4XT4qWxLyPjsogk6FqYbx+xpZs7YysyV0D GUL+SU8$nMFx7 (different)

The intermediale values of sample 2.

NormalizedPassword	$Qxyg nEA4s FTCGu AloGv Cey5Z CmZIJ I2hk2 igq4P DVw (same)
SHA512PW64	4B 77 5B D0 FE BC F2 D3 DF 90 01 83 DF BF 37 0C 1F B0 7D BD D0 91 56 8F 01 CC A7 9E 33 E6 AC 1A CA 86 6A 66 48 FA D8 6B D1 30 FA 57 17 F9 96 DF B6 EA 30 9C 16 B0 77 31 99 08 24 62 C3 D2 16 A5

	(same)
Compressed	F3 48 CD C9 C9 57 08 4E 4D 2E 4A 2D 29 56 54 78 34 63 17 00 (same)
PaddedData	F3 48 CD C9 C9 57 08 4E 4D 2E 4A 2D 29 56 54 78 34 63 17 00 62 3F 23 CC 9B F9 99 EE DD C2 89 EB (last 13 bytes different)
PlaintextPadding	0C (same)
IV16	C6 99 65 81 A0 25 C7 42 13 92 6C D8 D5 3B 2C 56 (different)
PasswordDerivedKey32	93 51 26 B2 BC 54 27 D8 3F 65 DE 8C 4E 89 A0 94 D7 E4 A3 DB 93 8C AB DD D3 AB 5D E8 AE 31 26 B0 (different)
RandomKey32	7D 57 77 87 C8 8C 36 31 88 39 99 50 B1 32 5F E7 43 7B 86 33 1A 77 93 0F 41 9A AF B1 07 F3 66 A7 (different)
RandomKeyCipher32	76 41 F7 6C 7C 79 8D E4 E8 DF BE 28 8C 64 D6 F6 80 8D 78 34 BB 48 C9 47 E3 51 15 6A CB 85 D3 E2 (different)
MessageCipher	A5 B1 2F 23 E3 B2 88 24 E8 5A 98 6F 1F B1 A5 9B 3B 63 2B 32 57 40 C6 50 BF 92 53 CF E7 30 5C 7B (different)
MAC16	08 EE 87 34 22 28 E6 64 1C 28 26 E9 FF F9 6B CF (different)
MACCipher16	AD 54 67 84 64 A2 6D 46 10 48 CF 16 7D F5 01 5A (different)

Samples generated with VisualCrypt Pro 2.0.6 [11].

6 References

Source links as of October 2015.

[1] Specification for the ADVANCED ENCRYPTION STANDARD (AES), NIST 2001, http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

[2] A Future-Adaptable Password Scheme, Niels Provos and David Mazieres 1999, https://www.usenix.org/legacy/events/usenix99/provos/provos_html/node1.html

[3] FIPS 180-4 Secure Hash Standard (SHS), NIST 2015, http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf

[4] DEFLATE Compressed Data Format Specification version 1.3, IETF 1996, https://tools.ietf.org/html/rfc1951

[5] PKCS #7: Cryptographic Message Syntax Standard, Version 1.5, RSA Security 1993, ftp://ftp.rsasecurity.com/pub/pkcs/ascii/pkcs-7.asc

[6] ANSI X.923, in: Wikipedia 2015, https://en.wikipedia.org/wiki/Padding_(cryptography)

[7] ISO 10126-2:1991 (withdrawn), International Organization for Standardization 1991, http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=18114

[8] Recommendation for Block Cipher Modes of Operation – Methods and Techniques, NIST 2001, http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf

[9] The Unicode® Standard: A Technical Introduction, Unicode Inc. 2015 http://www.unicode.org/standard/principles.html

[10] The Unicode Standard Version 7.0 – Core Specification, Unicode Consortium 2014, http://www.unicode.org/versions/Unicode7.0.0/ch03.pdf#G7404

[11] VisualCrypt Source Code, Claus Ehrenberg, VisualCrypt AG 2015, https://github.com/VisualCrypt/VisualCrypt/commit/5beb3c5392da40a815d2ac7cd1098dab162e420a