Encryption and authentication in PHP 7[1]

by Enrico Zimuel

Encryption is an important aspect of many software applications. Today, most of the web sites store sensitive data like credit card numbers and user's passwords.
In this article I will explain how to encrypt and authenticate sensitive data using PHP 7, the last release of PHP that offers new security features.

Encryption is not enough

An encryption algorithm works with a *secret key* that is used to decrypt a message. Encryption provides *confidentiality*, that means only the authorized persons (the owners of the *secret key*) have access to the information.

Encryption does not provide *integrity* and *authenticity* assurance on the data. Anyone can falsify an encrypted message without any evidence of the alteration. For instance, Bob encrypts a message using the Advanced Encryption Algorithm (AES) with a 256 bit key. Bob wants to send the message to Alice that has the decryption key. Mallory, a malicious attacker, intercept the message and alter the content. Alice tries to decrypt the message, obtaining another message; she cannot prove that the message has been altered and that it comes from Bob.

Encryption is not enough. We need to provide authentication. We have to protect the encrypted message from tampering and we want to be sure that the message has been encrypted by authorized users.

Moreover, there are many cryptographic attacks on encryption algorithms without authentication. For instance, the padding oracle attack performed using the padding of a cryptographic message. This attack is critical in many use cases, for instance in the CBC encryption mode it can recover the encryption key in a few seconds! The original attack was published in 2002 by Serge Vaudenay. The attack was applied to several web frameworks, including JavaServer Faces, Ruby on Rails and ASP.NET. In order to prevent this attack we need to add an authentication layer.

To provide authentication we have two options: use the encrypt-then-authenticate approach, that adds the authentication after the encryption, or use an authenticated encryption algorithm that offers authentication built-in.

Authenticated encryption is available from PHP 7.1 using the OpenSSL extension of PHP. If you are using PHP 7.0 or less, you can use the *encrypt-then-authenticate* approach, presented in the next section.

[1] Article published in *Software Developer's Journal*, Vol 6. No 2, Issue Feb 2017. ISSN 1734-3933

Encrypt-then-authenticate

In *encrypt-then-authenticate* we apply the encryption first and than the authentication. For authenticate a message, we can use an HMAC function.

HMAC is a *keyed-Hash Message Authentication Code* used to generate the hash of a message providing a secret key. The function is defined by the following formula:

```
HMAC(K,m) = H((K' ⊕ opad) || H((K' ⊕ ipad) || m))
```

where **H** is a cryptographic hash function, **K** is the secret key, **m** is the message to be authenticated, **K'** is another secret key, derived from the original key K (by padding K to the right with extra zeros to the input block size of the hash function, or by hashing K if it is longer than that block size), **||** denotes concatenation, ⊕ is the XOR operator, **opad** is the outer padding (0x5c5c5c…5c5c, one-block-long hexadecimal constant), and **ipad** is the inner padding (0x363636…3636, one-block-long hexadecimal constant).

The secret key **K** should be different from the encryption key. This can improve the security of the system. Typically, we can generate the encryption and the authentication keys starting from a user's password using a Key Derivation Function (KDF). One of the most used algorithm is PBKDF2 a PKCS #5 v2.0 standard and RFC 2898.

PHP offers PBKDF2 a built-in function hash_pbkdf2(). Here is reported an example:

```
$password = 'supersecretpassword';
$salt = random_bytes(16);
$hash = hash_pbkdf2("sha256", $password, $salt, 20000);
var_dump($hash);
```

In this example, the hash_pbkdf2() function generates the output using the hash function SHA-256 iterated 20,000 times. The function requires a random *salt* value that is very important for the security of the algorithm. We used the random_bytes() function of PHP 7 that generates cryptographically secure pseudo-random bytes.

Note: for cryptographic purposes never use rand() or mt_rand() functions of PHP.

The output of the PBKDF2 is a string of 32 bytes in hex format (64 characters). If you want a binary string with a different size you have to specify two additional parameters. For instance, if you need an hash value of 128 bytes in binary format you can use the following syntax:

```
$hash = hash_pbkdf2("sha256", $password, $salt, 20000, 128, true);
```

The number of rounds of the hash function (20,000 in our example) is a very important parameter for the security of the algorithm. We used the value 20,000 that can be considered enough for many web applications. You should use always the maximum number of rounds which is tolerable, performance-wise in your application. Here you can have more information about this parameter. For instance, LastPass uses 100,000 rounds to generate a server-side password.

Now that we know how to authenticate a message and how to generate keys, we can use the HMAC and the PBKDF2 functions. Below is a PHP example of *encrypt-then-authenticate* using AES-256 for encryption and HMAC-SHA256 for authentication. We used OpenSSL for the encryption part.

```
function encrypt(string $text, string $key): string {
  $iv      = random_bytes(16); // iv size for aes-256-cbc
  $keys    = hash_pbkdf2('sha256', $key, $iv, 20000, 64, true);
  $encKey  = mb_substr($keys, 0, 32, '8bit');
  $hmacKey = mb_substr($keys, 32, null, '8bit');

  $ciphertext = openssl_encrypt(
      $text,
      'aes-256-cbc',
      $encKey,
      OPENSSL_RAW_DATA,
      $iv
  );

  $hmac = hash_hmac('sha256', $iv . $ciphertext, $hmacKey);
  return $hmac . $iv . $ciphertext;
}
```

The output of this function is the concatenation of the HMAC hash (**$hmac**), the random IV (**$iv**) and the encrypted message (**$ciphertext**). All these are needed for the decryption.

```
function decrypt(string $text, string $key): string {
  $hmac       = mb_substr($text, 0, 64, '8bit');
  $iv         = mb_substr($text, 64, 16, '8bit');
  $ciphertext = mb_substr($text, 80, null, '8bit');

  $keys    = hash_pbkdf2('sha256', $key, $iv, 20000, 64, true);
  $encKey  = mb_substr($keys, 0, 32, '8bit');
  $hmacKey = mb_substr($keys, 32, null, '8bit');
  $hmacNow = hash_hmac('sha256', $iv . $ciphertext, $hmacKey);
  if (! hash_equals($hmac, $hmacNow)) {
      throw new Exception('Authentication error!');
  }
  return openssl_decrypt($ciphertext, 'aes-256-cbc', $encKey,
      OPENSSL_RAW_DATA, $iv
  );
}
```

Before decrypting the data, we test the authentication using the hash_equals() function of PHP. This function can be used to prevent timing attacks.

There are many PHP libraries offering *encrypt-then-authenticate*, for instance:

- paragonie/halite
- defuse/php-encryption
- zendframework/zend-crypt

Full disclosure: I'm the author of zendframework/zend-crypt library.

Authenticated encryption

Starting from PHP 7.1 we can use the authenticated encryption modes included in the OpenSSL extension. We can use two encryption modes: GCM and CCM.

The OpenSSL extension provides two functions to encrypt and decrypt a message. These functions are openssl_encrypt() and openssl_decrypt().

PHP 7.1 introduced some additional parameters to these functions for the authenticated encryption usage.

```
string openssl_encrypt(
      string $data,
      string $method,
      string $password,
      [ int $options = 0 ],
      [ string $iv = "" ],
      [ string &$tag = NULL ],
      [ string $aad = "" ],
      [ int $tag_length = 16 ]
)
string openssl_decrypt(
      string $data,
      string $method,
      string $password,
      [ int $options = 0 ],
      [ string $iv = "" ],
      [ string $tag = "" ],
      [ string $aad = "" ]
)
```

The authentication hash is stored in the **$tag** variable. This value is passed by reference to the **openssl_encrypt()** function.
The other optional parameter **$aad** represents additional authentication data that you could use to protect the message against alterations, without encrypting it. For instance, if you need to encrypt an email leaving the header information in plaintext, like the sender and the receiver, you can pass the header in **$aad**.

The last optional parameter **$tag_length** is the length in bytes of the hash value, that is 16 by default. In GCM mode, the tag length can be between 4 and 16 bytes. CCM has no limits of tag's length and also the resulting tag is different for each length.

To decrypt an authenticated message, you need to pass the **$tag** value to **openssl_decrypt()** and optionally the additional authenticated data (**$aad**).

Galois/Counter Mode (GCM)

The Galois/Counter Mode (GCM) is a mode of operation for symmetric key cryptographic block ciphers that provides encryption and authentication.

The algorithm works as follow:

- the message is divided into blocks numbered sequentially;
- each block number is encrypted using a cipher, usually AES;
- the result of the encryption is xored with the block message;
- the encrypted block is xored with a Galois Mult function for authentication;
- the authentication tag is then generated with a couple of last xor, using:
    - the size of the ciphertext concatenated with the size of authentication;
    - a Galois Mult function xored with the first encrypted block number.

In Figure 1 is reported a diagram that explains the single steps of GCM mode.

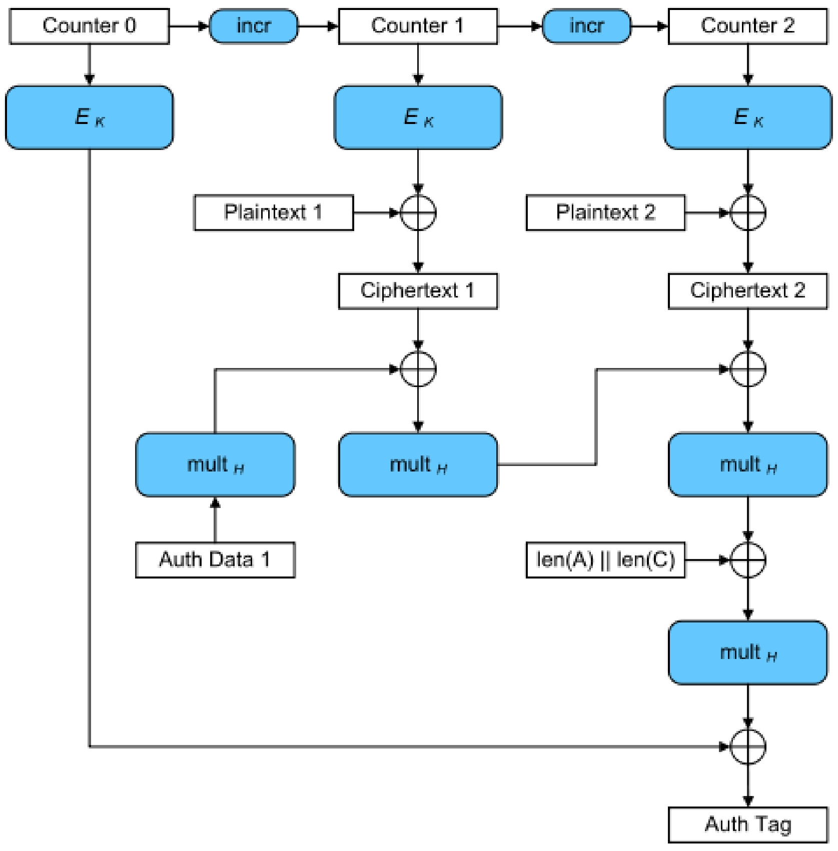

Figure 1: GCM encryption mode (Source: Wikipedia)

This algorithm is used in many applications like IPsec, SSH and TLS 1.2. Used together with AES (AES-GCM) is included in the NSA Suite B Cryptography. GCM is very fast because the execution can be parallelized. Moreover, the algorithm does not have any patents and can be used without restrictions.

Starting from PHP 7.1 the GCM mode is supported by OpenSSL extension. You can check if the mode is available on your system using the openssl_get_cipher_methods() function. The GCM mode is reported as "-gcm" or "-GCM" string at the end of a cipher name. You need to have at least OpenSSL 1.1 to support this algorithm.

Below is reported an example to encrypt and decrypt a message using 'aes-256-gcm' algorithm (i.e. AES with 256 bit key in GCM mode):

```
$algo = 'aes-256-gcm';
$iv   = random_bytes(openssl_cipher_iv_length($algo));
$key  = random_bytes(32); // 256 bit
$msg  = random_bytes(1024); // random message, 1 Kb

$ciphertext = openssl_encrypt(
      $msg,
      $algo,
      $key,
      OPENSSL_RAW_DATA,
      $iv,
      $tag
);

$decrypt = openssl_decrypt(
      $ciphertext,
      $algo,
      $key,
      OPENSSL_RAW_DATA,
      $iv,
      $tag
);

if (false === $decrypt) {
      throw new Exception(sprintf(
      "OpenSSL error: %s", openssl_error_string()
      ));
}
printf ("Decryption %s\n", $msg === $decrypt ? 'Ok' : 'Failed');
```

The authentication tag generated by **openssl_encrypt()** is stored in **$tag** passed by reference in the function. This value and the *Initialization Vector* (**$iv**) should be stored together with the encrypted part (**$ciphertext**). In fact, in order to decrypt we need to pass **$iv** and **$tag** to openssl_decrypt() function.

If the ciphertext has been altered, the decrypt function is able to recognize it because the authentication fails. This will generate a false result that can be intercepted, obtaining more information using openssl_error_string() function.

The GCM mode offers also *additional authenticated data* that you can pass to the encrypt function. This data will not be encrypted and used only for authentication. For instance, if we need to send an encrypted email, we can use the sender and recipient as additional authenticated data. These data should be sent in plaintext, to allow the routing of the email, but the rest of the information can be encrypted.

We can pass *additional authenticated data* using the **$aad** parameter of **openssl_encrypt()** function. Below is reported an example:

```
$algo  = 'aes-256-gcm';
$iv     = random_bytes(openssl_cipher_iv_length($algo));
$key   = random_bytes(32); // 256 bit
$email = 'This is the secret message!';
$aad   = 'From: foo@domain.com, To: bar@domain.com';
$ciphertext = openssl_encrypt(
      $email,
      $algo,
      $key,
      OPENSSL_RAW_DATA,
      $iv,
      $tag,
      $aad
);

$decrypt = openssl_decrypt(
      $ciphertext,
      $algo,
      $key,
      OPENSSL_RAW_DATA,
      $iv,
      $tag,
      $aad
);
if (false === $decrypt) {
      throw new Exception(sprintf(
      "OpenSSL error: %s", openssl_error_string()
      ));
}
printf ("Decryption %s\n", $email === $decrypt ? 'Ok' : 'Failed');
```

The encrypted message, using *additional authenticated data,* is composed by **$tag . $iv . $aad . $ciphertext**. Basically, you need to store also the **$aad** in plaintext, to be able to perform the authentication during the decryption of the message.

Counter with CBC-MAC (CCM)

Counter with CBC-MAC (CCM) is another authenticated encryption mode for symmetric block ciphers with a block length of 128 bits.
CCM mode combines the CBC-MAC for authentication and the Counter CTR mode for encryption.

The CCM mode is used in many applications like IPsec and TLS 1.2 and is part of the IEEE 802.11i standard. CCM is an alternative implementation of the OCB mode that was originally covered by patents. The CCM can be used without any restriction.

This encryption mode can be used in PHP 7.1 if the OpenSSL extension installed shows the "-ccm" or "-CCM" string in the name of the algorithm.

The previous example code for GCM works also for CCM, you need only to replace the first line with :

```
$algo = 'aes-256-ccm';
```

GCM vs. CCM

If we compare GCM vs. CCM from a security point of view they are equivalent. Until now, no critical attack has been found on these encryption modes.
From a performance point of view after some benchmark tests that I did on PHP 7.1, I can say that **GCM is 3x faster than CCM**.

If you are using PHP 7.1, I suggest using GCM mode as an encryption algorithm.

Conclusion

In this article, I showed the importance to guarantee authentication and not just encryption to sensitive data. I showed some examples in PHP 7. If you are using PHP 7.0 (or PHP 5.5+) you can use the *encrypt-then-authenticate* approach with some of the libraries such as paragonie/halite, defuse/php-encryption, zendframework/zend-crypt.
If you can upgrade to PHP 7.1 you can benefit from the usage of *authenticated encryption* using GCM or CCM mode of OpenSSL. For performance reasons, I suggest using the GCM mode that is 3x faster than CCM.