PHP data encryption primer

A short guide to help to avoid the common mistakes and pitfalls with symmetric data encryption using PHP.

This primer assumes a “client-server” situation, which is probably a typical case with PHP applications.

Naturally the recommendations given here are not the “only possible way” to handle data encryption in PHP, but this primer aims to be straightforward and tries to leave less room for mistakes and (possibly confusing) choices.

18 Jun 2014 Post title was revised from "PHP data encryption cheatsheet" to "PHP data encryption primer".

Encryption functions available in PHP

Use either Mcrypt extension (http://php.net/mcrypt) or OpenSSL extension (http://php.net/openssl).

Encryption algorithm / mode of operation / nonce (initializing vector)

Use **AES-256** in **CTR** mode with random nonce. AES is the standard and can be used with both the Mcrypt and OpenSSL extensions.

Make sure to **always** generate a new random nonce when encrypting data. This **must** be done using cryptographically secure randomness source. See more about random number generation here. The nonce can be concatenated with the ciphertext to allow decryption.

The nonce length must be 128 bits (16 bytes) and must contain raw bytes, *not* encoded in any way.

With Mcrypt, AES is known as `rijndael-128`. With OpenSSL, it is respectively `AES-256-CTR`.

```
<?php
// $key length must be exactly 256 bits (32 bytes).
// $nonce length must be exactly 128 bits (16 bytes).
$ciphertext = mcrypt_encrypt(MCRYPT_RIJNDAEL_128, $key, $plaintext, 'ctr', $nonce)
; // Mcrypt
$ciphertext = openssl_encrypt($plaintext, 'AES-256-CTR', $key, true, $nonce); // O
penSSL
```

Verify your encryption and decryption routines against AES test vectors (http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf).

There are some data length limits with AES in CTR mode. While not probably in practical manner, but keep in mind that you should encrypt less than 2^64 bytes of data with a single key (no matter if it is a “few” smaller messages or just only one big message).

Also, CTR mode is only safe when you **do not** reuse nonces under a single key. That is why it is important to create the nonces with cryptographically secure random number generator. At the same time it means you must not encrypt more than 2^64 different messages with a single key (as the nonce space with AES is 128 bits, it is important to limit the number of messages (nonces) to 2^128/2 because of the birthday paradox).

And remember that encrypting the data will not hide, most importantly, the fact how much data you are sending. As a drastic example, if you only encrypt messages containing “yes” / “no”, the plain encryption do not hide the confidential details.

Data authentication

Always authenticate the encrypted data.

Use Encrypt-then-MAC construction (http://www.daemonology.net/blog/2009-06-24-encrypt-then-mac.html). That is, first encrypt the data and finally take an HMAC-SHA-256 of the resulted ciphertext, and include all the relevant pieces under the HMAC (namely ciphertext and nonce).

When decrypting, first check the HMAC using a constant-time string comparison (do not directly compare `$user_submitted_mac` and `$calculated_mac` with === string comparison). Or better yet, compare the strings using “double HMAC verification (https://www.isecpartners.com/blog/2011/february/double-hmac-verification.aspx)”. This is to avoid leaking exploitable timing information that occurs on the === string comparison.

If the HMAC matches, the ciphertext is safe to feed to decrypt process. If the HMAC does not match, exit immediately.

Encryption and authentication keys

Ideally, use keys generated using cryptographically secure random number generator (see more about random number generation here). With AES-256 you need 32 bytes of random data (raw bytes, *not* encoded).

If you have to rely on user typed keys (ie. a config parameter), it needs to be derived to be suitable to use as an encryption key. Use PBKDF2 algorithm to turn a human supplied key into an encryption key. See http://php.net/hash_pbkdf2 (http://php.net/hash_pbkdf2) (and make sure to use raw output).

If you are not on PHP 5.5 or higher, you have to use an userland PHP PBKDF2 implementation. One such implementation can be found here: https://github.com/defuse/password-hashing/blob/master/PasswordHash.php#L87 (https://github.com/defuse/password-hashing/blob/master/PasswordHash.php#L87).

Note that when relying on userland implementations, you can not stretch the key as much as you could with more efficient PHP’s native `hash_pbkdf2()` function, which means you can not squeeze as much security out of the user supplied key.

Do not use same key for encryption and authentication. As seen above, you need 32 bytes for an encryption key. Use also 32 bytes for an authentication (HMAC) key.

With PBKDF2 you can derive 64 bytes from a single password/master key and use, say, the first 32 bytes for encryption and the last 32 bytes for authentication.

If you have the keys stored in a file, say, hex encoded, do not decode them prior to feeding to the encryption routines. Instead, as earlier mentioned, use PBKDF2 to turn the hex encoded keys into proper encryption/authentication keys. Or use SHA-256 (with raw output) to hash the hex encoded keys and turn them into proper raw bytes (the use of "plain" hashing assumes the initial keys has enough guessing entropy, as explained in the next paragraphs).

Key stretching

Low entropy keys should be avoided in the first place. But if you need to rely on, say, user's passwords, you need to use as high PBKDF2 iteration count as possible to squeeze as much security as possible out of the passwords.

PBKDF2 algorithm can be adjusted for specific iteration count. The higher the iteration count the higher the security of the derived key. If your code runs on 64-bit platform, use `sha512` as the underlying PBKDF2 hashing algorithm. If you are on 32-bit platform, use `sha256` as the underlying hashing algorithm.

In general, it is not possible to use relatively high iteration count in online applications (which face the public internet). And thus the added security to the key will not be as high as in more ideal situation (i.e. an offline application could use higher iteration count without the fear of a DoS attack). As a rule of thumb, for online applications, adjust the PBKDF2 iteration count to take less than 100 ms.

If you can use high entropy passwords (or config parameter etc.), you don't need to stretch them as much as low entropy passwords. For example if you created "master_encryption_key" and "master_authentication_key" using `/dev/urandom`, you don't need PBKDF2 necessarily at all. This is because the initial keys contains already enough guessing entropy. Just make sure you input raw bytes to the encryption/authentication routines, as earlier mentioned.

However, it is easy to derive both the encryption and authentication keys with PBKDF2 from the *single* master password (just use low iteration count, i.e. 1). This is useful if you have only one "master key" which should be derived for both the encryption and authentication use.

Key storage and management

Ideally, use a separate hardware to store keys (i.e. HSM).

If this is not possible, one method to mitigate the attack surface is by encrypting your key file or config file (which holds the actual encryption/authentication keys) with a key stored in a separate location from the actual key file (separate from the home/www folder). For example, you can use an Apache environment variable via httpd.conf to store the key needed to unlock the actual key file:

```
<VirtualHost *:80>
SetEnv keyfile_key crypto_strong_high_entropy_key
# You can access this variable in PHP using $_SERVER['keyfile_key']
# Rest of the config
</VirtualHost>
```

Now, if your www-root (including your key/config file) leaks i.e. via backup tape, the encrypted data is still safe as long as the `keyfile_key` environment variable stays secret. Remember to have a separate backup of the httpd.conf file (e.g. in a safe) and make sure you do not leak the `keyfile_key` via `phpinfo()`.

If you use a specific key file (instead of a config parameter), it is feasible to rotate keys. In the worst-case scenario where an adversary has got your encryption/authentication keys and nobody knows that, rotating the keys by some period of time might cut her access (assuming she can not get the new keys). This may make the damage smaller because the adversary can not abuse the leaked keys endlessly.

Data compression

As a rule of thumb, do not compress plaintext prior to encryption. This might leak information about the plaintext to an adversary.

For example, if you store session data into an encrypted browser cookie, and you store some user submitted data along with some secret data, the adversary can learn about the secret data by sending specially crafted payloads (the user submitted data) and measuring how the ciphertext sizes vary.

This is because the underlying compression is more effective if there are similarities in the user submitted data and the secret data, and thus leaks information via the payload size. CRIME (http://en.wikipedia.org/wiki/CRIME) is based on this information leak (side-channel).

If you are in doubt, do not use data compression.

Server environment

As a rule of thumb, do not host your security critical application on a shared hardware (i.e. a VM on a web host where an adversary could host her VM as well on the same physical hardware).

There are different side-channels (among others problems) which makes shared hardware a questionable place to host security critical code. For example, cross-VM attacks has been recently demonstrated: http://eprint.iacr.org/2014/248.pdf (http://eprint.iacr.org/2014/248.pdf). This is a good reminder that attacks never get worse, instead they get better and better over time. Such pitfalls should be always acknowledged.

If in doubt, do not deploy your security critical application to a shared hardware.

Consult an expert

Last but not least, consult an expert to review your security critical code.

@rootlabs (https://twitter.com/rootlabs/status/474741767607033856), 5. June 2014:

> *@plo* @veorq I have been working in crypto since 1997, and I still get every design or implementation I do reviewed by a 3rd party.

Appendix

Cryptographically safe random numbers

Use operating system provided randomness. In PHP, use `mcrypt_create_iv($count, MCRYPT_DEV_URANDOM)` or manually read from `/dev/urandom`.

Make sure you get the needed amount of bytes. If not, exit immediately (do not try to recover from an error by falling back to home-made randomness construction).

---

timoh6@gmail.com (mailto:timoh6@gmail.com)