

 **Christopher Eskow** Update the tutorial with reference to new key-generating script

4dfdd8d on Apr 26, 2017

5 contributors     

299 lines (235 sloc) | 12 KB

# Tutorial

Hello! If you're reading this file, it's because you want to add encryption to one of your PHP projects. My job, as the person writing this documentation, is to help you make sure you're doing the right thing and then show you how to use this library to do it. To help me help you, please read the documentation *carefully* and *deliberately*.

## A Word of Caution

Encryption is not magic dust you can sprinkle on a system to make it more secure. The way encryption is integrated into a system's design needs to be carefully thought out. Sometimes, encryption is the wrong thing to use. Other times, encryption needs to be used in a very specific way in order for it to work as intended. Even if you are sure of what you are doing, we strongly recommend seeking advice from an expert.

The first step is to think about your application's threat model. Ask yourself the following questions. Who will want to attack my application, and what will they get out of it? Are they trying to steal some information? Trying to alter or destroy some information? Or just trying to make the system go down so people can't access it? Then ask yourself how encryption can help combat those threats. If you're going to add encryption to your application, you should have a very clear idea of exactly which kinds of attacks it's helping to secure your application against. Once you have your threat model, think about what kinds of attacks it *does not* cover, and whether or not you should improve your threat model to include those attacks.

**This isn't for storing user login passwords:** The most common use of cryptography in web applications is to protect the users' login passwords. If you're trying to use this library to "encrypt" your users' passwords, you're in the wrong place. Passwords shouldn't be *encrypted*, they should be *hashed* with a slow computation-heavy function that makes password guessing attacks more expensive. See [How to Safely Store Your Users' Passwords in 2016](#).

**This isn't for encrypting network communication:** Likewise, if you're trying to encrypt messages sent between two parties over the Internet, you don't want to be using this library. For that, set up a TLS connection between the two points, or, if it's a chat app, use the [Signal Protocol](#).

What this library provides is symmetric encryption for "data at rest." This means it is not suitable for use in building protocols where "data is in motion" (i.e. moving over a network) except in limited set of cases.

## Getting the Code

There are several different ways to obtain this library's code and to add it to your project. Even if you've already cloned the code from GitHub, you should take steps to verify the cryptographic signatures to make sure the code you got was not intercepted and modified by an attacker.

Please head over to the [Installing and Verifying](#) documentation to get the code, and then come back here to continue the tutorial.

## Using the Library

I'm going to assume you know what symmetric encryption is, and the difference between symmetric and asymmetric encryption. If you don't, I recommend taking [Dan Boneh's Cryptography I course](#) on Coursera.

To give you a quick introduction to the library, I'm going to explain how it would be used in two stereotypical scenarios. Hopefully, one of these stereotypes is close enough to what you want to do that you'll be able to figure out what needs to be different on your own.

## Formal Documentation

While this tutorial should get you up and running fast, it's important to understand how this library behaves. Please make sure to read the formal documentation of all of the functions you're using, since there are some important security warnings there.

The following classes are available for you to use:

- **Crypto**: Encrypting and decrypting strings.
- **File**: Encrypting and decrypting files.
- **Key**: Represents a secret encryption key.
- **KeyProtectedByPassword**: Represents a secret encryption key that needs to be "unlocked" by a password before it can be used.

### Scenario #1: Keep data secret from the database administrator

In this scenario, our threat model is as follows. Alice is a server administrator responsible for managing a trusted web server. Eve is a database administrator responsible for managing a database server. Dave is a web developer working on code that will eventually run on the trusted web server.

Let's say Alice and Dave trust each other, and Alice is going to host Dave's application on her server. But both Alice and Dave don't trust Eve. They know Eve is a good database administrator, but she might have incentive to steal the data from the database. They want to keep some of the web application's data secret from Eve.

In order to do that, Alice will use the included `generate-defuse-key` script which generates a random encryption key and prints it to standard output:

```
$ composer require defuse/php-encryption
$ vendor/bin/generate-defuse-key
```

Alice will run this script once and save the output to a configuration file, say in `/etc/daveapp-secret-key.txt` and set the file permissions so that only the user that the website PHP scripts run as can access it.

Dave will write his code to load the key from the configuration file:

```
<?php
use Defuse\Crypto\Key;

function loadEncryptionKeyFromConfig()
{
    $keyAscii = // ... load the contents of /etc/daveapp-secret-key.txt
    return Key::loadFromAsciiSafeString($keyAscii);
}
```

Then, whenever Dave wants to save a secret value to the database, he will first encrypt it:

```
<?php
use Defuse\Crypto\Crypto;

// ...
$key = loadEncryptionKeyFromConfig();
// ...
$ciphertext = Crypto::encrypt($secret_data, $key);
// ... save $ciphertext into the database ...
```

Whenever Dave wants to get the value back from the database, he must decrypt it using the same key:

```
<?php
use Defuse\Crypto\Crypto;

// ...
$key = loadEncryptionKeyFromConfig();
// ...
$ciphertext = // ... load $ciphertext from the database
try {
    $secret_data = Crypto::decrypt($ciphertext, $key);
} catch (\Defuse\Crypto\Exception\WrongKeyOrModifiedCiphertextException $ex) {
    // An attack! Either the wrong key was loaded, or the ciphertext has
```

```

    // changed since it was created -- either corrupted in the database or
    // intentionally modified by Eve trying to carry out an attack.

    // ... handle this case in a way that's suitable to your application ...
}

```

Note that if anyone ever steals the key from Alice's server, they can decrypt all of the ciphertexts that are stored in the database. As part of our threat model, we are assuming Alice's server administration skills and Dave's secure coding skills are good enough to stop Eve from being able to steal the key. Under those assumptions, this solution will prevent Eve from seeing data that's stored in the database.

However, notice that our threat model says nothing about what could happen if Eve wants to *modify* the data. With this solution, Eve will not be able to alter any individual ciphertext (because each ciphertext has its own cryptographic integrity check), but Eve *will* be able to swap ciphertexts for one another, and revert ciphertexts to what they used to be at previous times. If we needed to defend against such attacks, we would have to re-design our threat model and come up with a different solution.

## Scenario #2: Encrypting account data with the user's login password

This scenario is like Scenario 1, but subtly different. The threat model is as follows. We have Alice, a server administrator, and Dave, the developer. Alice and Dave trust each other, and Alice wants to host Dave's web application, including its database, on her server. Alice is worried about her server getting hacked. The application will store the users' credit card numbers, and Alice wants to protect them in case the server gets hacked.

We can model the situation like this: after the server gets hacked, the attacker will have read and write access to all data on it until the attack is detected and Alice rebuilds the server. We'll call the time the attacker has access to the server the *exposure window*. One idea to minimize loss is to encrypt the users' credit card numbers using a key made from their login password. Then, as long as the users all have strong passwords, and they are never logged in during the exposure window, their credit cards will be protected from the attacker.

To implement this, Dave will use the `KeyProtectedByPassword` class. When a new user account is created, Dave will save a new key to their account, one that's protected by their login password:

```

<?php
use Defuse\Crypto\KeyProtectedByPassword;

function CreateUserAccount($username, $password)
{
    // ... other user account creation stuff, including password hashing

    $protected_key = KeyProtectedByPassword::createRandomPasswordProtectedKey($password);
    $protected_key_encoded = $protected_key->saveToAsciiSafeString();
    // ... save $protected_key_encoded into the user's account record
}

```

Then, when the user logs in, Dave's code will load the protected key from the user's account record, unlock it to get a `Key` object, and save the `key` object somewhere safe (like temporary memory-backed session storage). Note that wherever Dave's code saves the key, it must be destroyed once the user logs out, or else the attacker might be able to find users' keys even if they were never logged in during the attack.

```

<?php
use Defuse\Crypto\KeyProtectedByPassword;

// ... authenticate the user using a good password hashing scheme
// keep the user's password in $password

$protected_key_encoded = // ... load it from the user's account record
$protected_key = KeyProtectedByPassword::loadFromAsciiSafeString($protected_key_encoded);
$user_key = $protected_key->unlockKey($password);
$user_key_encoded = $user_key->saveToAsciiSafeString();
// ... save $user_key_encoded in the session

```

```

<?php
// ... when the user is logging out ...
// ... securely wipe the saved $user_key_encoded from the system ...

```

When a user adds their credit card number, Dave's code will get the key from the session and use it to encrypt the credit card number:

```
<?php
use Defuse\Crypto\Crypto;
use Defuse\Crypto\Key;

// ...

$user_key_encoded = // ... get it out of the session ...
$user_key = Key::loadFromAsciiSafeString($user_key_encoded);

// ...

$credit_card_number = // ... get credit card number from the user
$encrypted_card_number = Crypto::encrypt($credit_card_number, $user_key);
// ... save $encrypted_card_number in the database
```

When the application needs to use the credit card number, it will decrypt it:

```
<?php
use Defuse\Crypto\Crypto;
use Defuse\Crypto\Key;

// ...

$user_key_encoded = // ... get it out of the session
$user_key = Key::loadFromAsciiSafeString($user_key_encoded);

// ...

$encrypted_card_number = // ... load it from the database ...
try {
    $credit_card_number = Crypto::decrypt($encrypted_card_number, $user_key);
} catch (Defuse\Crypto\Exception\WrongKeyOrModifiedCiphertextException $ex) {
    // Either there's a bug in our code, we're trying to decrypt with the
    // wrong key, or the encrypted credit card number was corrupted in the
    // database.

    // ... handle this case ...
}
```

With all caveats carefully heeded, this solution limits credit card number exposure in the case where Alice's server gets hacked for a short amount of time. Remember to think about the attacks that *aren't* included in our threat model. The attacker is still free to do all sorts of harmful things like modifying the server's data which may go undetected if Alice doesn't have secure backups to compare against.

## Getting Help

---

If you're having difficulty using the library, see if your problem is already solved by an answer in the [FAQ](#).