Cryptography made easy with Zend Framework 2

by Enrico Zimuel (enrico@zend.com)

Senior Software Engineer
Zend Framework Core Team
Zend Technologies Ltd

About me

@ezimuel

enrico@zend.com

- Enrico Zimuel
- Software Engineer since 1996
- Senior PHP Engineer at Zend Technologies, in the Zend Framework Team
- Author of articles and books on cryptography, PHP, and secure software
- International speaker of PHP conferences
- B.Sc. (Hons) in Computer Science and Economics from the University “G'Annunzio” of Pescara (Italy)

Cryptography in Zend Framework

- In **2.0.0beta4** we released **Zend\Crypt** to help developers to use cryptography in PHP projects
- In PHP we have built-in functions and extensions for cryptography purposes:
  - crypt()
  - Mcrypt
  - OpenSSL
  - Hash (by default in PHP 5.1.2)
  - Mhash (emulated by Hash from PHP 5.3)

Cryptography in not so easy to use

- To implement cryptography in PHP we need a solid background in **cryptography engineering**
- The Mcrypt, OpenSSL and the others PHP libraries are good primitive but you need to know how to use it
- This can be a barrier that discouraged PHP developers
- We decided to offer a **simplified API for cryptography** with security best practices built-in
- The goal is to support **strong cryptography** in ZF2

Cryptography in Zend Framework

- **Zend\Crypt** components:
  - Zend\Crypt\Password
  - Zend\Crypt\Key\Derivation
  - Zend\Crypt\Symmetic
  - Zend\Crypt\PublicKey
  - Zend\Crypt\Hash
  - Zend\Crypt\Hmac
  - Zend\Crypt\BlockCipher

Zend\Crypt\BlockCipher

- **Zend\Crypt\BlockCipher** can be used to encrypt/decrypt sensitive data
- Provides **encryption + authentication (HMAC)**
- API simplified:
  - **setKey($key)**
  - **encrypt($data)**
  - **decrypt($data)**
- It uses the **Mcrypt** adapter (Zend\Crypt\Symmetric\Mcrypt)

Zend\Crypt\BlockCipher (2)

- Default values used by **BlockCipher**:
  - AES algorithm (key of 256 bits)
  - CBC mode + HMAC (SHA-256)
  - PKCS7 padding mode (RFC 5652)
  - PBKDF2 to generate encryption key + authentication key for HMAC
  - Random IV for each encryption

Example: encrypt

```
use Zend\Crypt\BlockCipher;

$cipher = BlockCipher::factory('mcrypt',
   array('algorithm' => 'aes')
);
$cipher->setKey('this is the encryption key');
$text      = 'This is the message to encrypt';
$encrypted = $cipher->encrypt($text);

printf("Encrypted text: %s\n", $encrypted);
```

- The encrypted text is encoded in Base64, you can get binary output using **setBinaryOutput(true)**

Example: decrypt

```
use Zend\Crypt\BlockCipher;

$cipher = BlockCipher::factory('mcrypt',
   array('algorithm' => 'aes')
);
$cipher->setKey('this is the encryption key');
$ciphertext = 'c093e6d...';
$encrypted  = $cipher->decrypt($text);

printf("Decrypted text: %s\n", $encrypted);
```

Parameters

- **factory($adapter, $parameters)**, where $parameters can be an array with the following keys:
  - **algorithm** (or **algo**), the name of the block cipher to use (supported algorithms are: aes (rijndael-128), rijndael-192, rijndael-256, blowfish, twofish, des, 3des, cast-128, cast-256, saferplus, serpent);
  - **mode**, the encryption mode of the block cipher (the supported modes are: cbc, cfb, ctr, ofb, nofb, ncfb);
  - **key**, the encryption key;
  - **iv** (or **salt**), the Initialization Vector (IV) also known as salt;
  - **padding**, the padding mode (right now we support only the PKCS7 standard);

Zend\Crypt\Symmetric

- Implements symmetric ciphers (single *key* to encrypt/decrypt)
- We support the **Mcrypt** extensions
- **Zend\Crypt\Symmetric\Mcrypt** is a wrapper of **Mcrypt** extension with a simplified API and security best practices built-in
- Don't use Zend\Crypt\Symmetric\Mcrypt to encrypt sensitive data (you need also authentication, use BlockCipher)

Zend\Crypt\PublicKey

- Implements public key algorithms
- We support:
  - **RSA** (Zend\Crypt\PublicKey\Rsa)
  - **Diffie-Hellman** (Zend\Crypt\PublicKey\DiffieHellman), for key exchange
- We use the **OpenSSL** extension

Example: digital signature of a file using RSA

```
use Zend\Crypt\PublicKey\Rsa,
    Zend\Crypt\PublicKey\RsaOptions;

$rsa = new Rsa(new RsaOptions(array(
   'passPhrase' => 'insert the passphrase here',
   'pemPath'    => 'name of the private key file .pem'
)));
$filename = 'name of the file to sign';
$file = file_get_contents($filename);

$signature = $rsa->sign($file, $rsa->getOptions()->getPrivateKey(), Rsa::FORMAT_BASE64);
$verify    = $rsa->verify($file, $signature, $rsa->getOptions()->getPublicKey(),
Rsa::FORMAT_BASE64);

if ($verify) {
   echo "The signature is OK\n";
   file_put_contents($filename . '.sig', $signature);
   echo "Signature saved in $filename.sig\n";
} else {
   echo "The signature is not valid!\n";
}
```

Zend\Crypt\Password

- How do you safely store a password?
  - MD5() + salt is not secure anymore, dictionary attacks can be performed much faster with modern CPU + cloud environments
  - A secure alternative is the **bcrypt** algorithm
- **Bcrypt** uses Blowfish cipher + iterations to generate secure hash values
- Bcrypt is secure against brute force or dictionary attacks because is slow, very slow (that means attacks need huge amount of time to be completed)

Work factor parameter of bcrypt

- The algorithm needs a *salt* value and a work factor parameter (*cost*), which allows you to determine how expensive the bcrypt function will be
- We used the **crypt()** function of PHP to implement the bcrypt algorithm
- The *cost* is an integer value from 4 to 31
- The default value for Zend\Crypt\Password\Bcrypt is 14 (that is equivalent to 1 second of computation using an Intel Core i5 CPU at 3.3 Ghz).
- The cost value depends on the CPU speed, check on your system! I suggest to set **at least 1 second**.

Example: bcrypt

```
use Zend\Crypt\Password\Bcrypt;

$bcrypt = new Bcrypt();
$start  = microtime(true);
$hash   = $bcrypt->create('password');
$end    = microtime(true);

printf ("Hash  : %s\n", $hash);
printf ("Exec. time: %.2f\n", $end-$start);
```

- The output of bcrypt ($hash) is a string of 60 bytes

How to verify a password

- In order to check if a password is valid against an hash value we can use the method:
    - **verify($password, $hash)**

  where **$password** is the value to check and **$hash** is the hash value generated by bcrypt
- This method returns true if the password is valid and false otherwise.

Zend\Crypt\Key\Derivation

- **Never use a user's password as cryptographic key**
- User's password are not secure because:
  1) **they are not random;**
  2) **they generate a small space of keys (low entropy).**
- We should always use a **Key Derivation Function** (or KDF)
- KDF are special algorithms that generate cryptographic keys, of any size, from a user's password
- One of the most used KDF is the PBKDF2 algorithm (RFC 2898).

PBKDF2

- “PBKDF2 applies a pseudorandom function, such as a cryptographic hash, cipher, or HMAC to the input password or passphrase along with a salt value and repeats the process many times to produce a derived key, which can then be used as a cryptographic key in subsequent operations. The added computational work makes password cracking much more difficult, and is known as key stretching” (from Wikipedia)
- The PBKDF2 algorithm is implemented in **Zend\Crypt\Key\Derivation\Pbkdf2**

Example: Pbkdf2

```
use Zend\Crypt\Key\Derivation\Pbkdf2,
    Zend\Math\Math;

$salt = Math::randBytes(32);
$pass = 'this is the password of the user';
$key  = Pbkdf2::calc('sha256',$pass, $salt, 100000, 32);
```

- We generated a cryptographic key of 32 bytes
- We used a random salt value
- We used 100'000 iterations for the algorithm (1 second of computation on Intel Core i5 CPU at 3.3 Ghz)

Zend\Crypt\Hash

- Implements the hash algorithms
- We used the Hash extension included in PHP 5.1.2
- **Zend\Crypt\Hash** provides static methods
- The usage is very simple:
    - **Zend\Crypt\Hash::compute($hash, $data, $output = Zend\Crypt\Hash::STRING)**

where **$hash** is the hash algorithm to be used (i.e. sha256), **$data** is the data to hash and **$output** specify if the output is a *string* or a *binary*.

Zend\Crypt\Hash (2)

- We can retrieve the list of all the supported algorithms using the method:
  - **Zend\Crypt\Hash::getSupportedAlgorithms()**

  this is a wrapper to the **hash_algos()** function of PHP.

- We can use retrieve the output size of a specific hash algorithm using the method:
  - **Zend\Crypt\Hash::getOutputSize($hash, $output = Zend\Crypt\Hash::STRING)**

  where **$hash** is the name of the algorithm and **$output** specify *string* or *binary* as result

Zend\Crypt\Hmac

- Implements the Hash-based Message Authentication Code (HMAC) algorithm supported by **Mhash** extension of PHP (emulated by Hash from PHP 5.3)
- **Zend\Crypt\Hmac** provides static methods
- The usage is very simple:
  - **Zend\Crypt\Hmac::compute($key, $hash, $data, $output = Zend\Crypt\Hmac::STRING)**

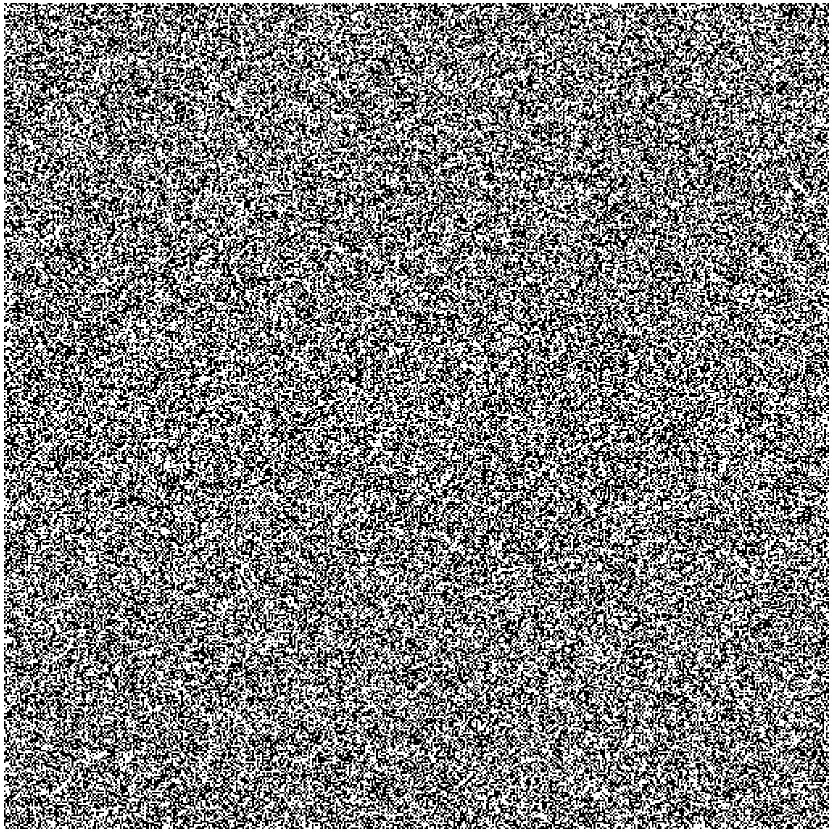
where **$key** is the key of HMAC, **$hash** is the name of the hash algorithm to be use, **$data** is the input data, and **$output** specify the output format, *string* or *binary*

PHP vs. randomness

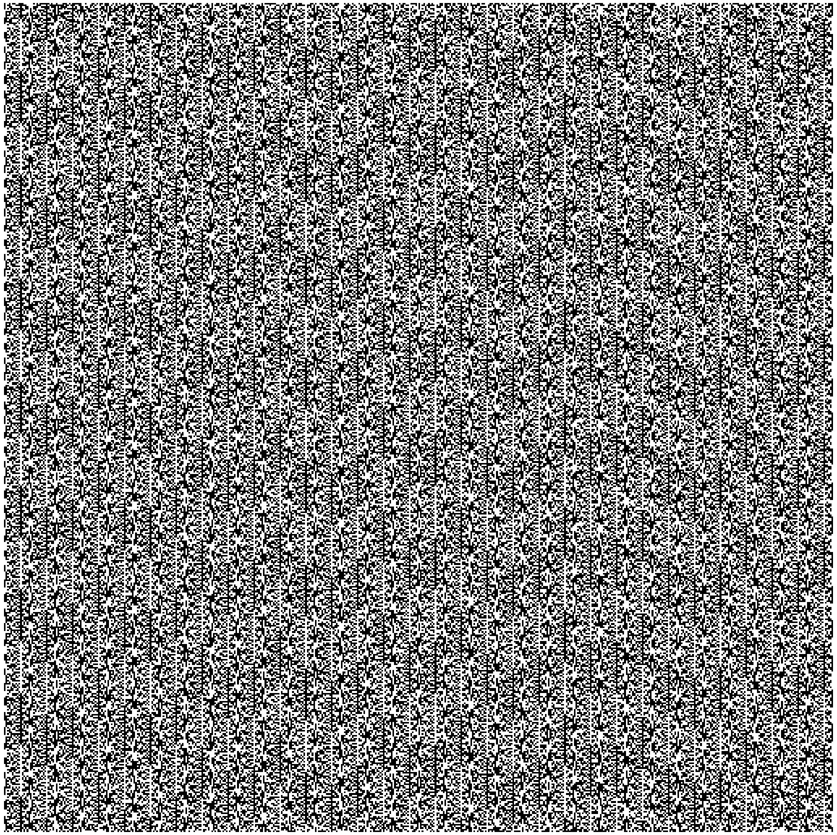
- How generate a pseudo-random value in PHP?
- Not good for cryptography purpose:
  - rand()
  - mt_rand()
- Good for cryptography (PHP 5.3+):
  - openssl_random_pseudo_bytes()

rand() is not so random

Pseudo-random bits



rand() of PHP on Windows



Source: random.org website

Random Number Generator in ZF

- We refactored the random number generator in ZF2 to use (in order):
    1) **openssl_random_pseudo_bytes()**
    2) **mcrypt_create_iv(), with MCRYPT_DEV_URANDOM**
    3) **mt_rand(), not used for cryptography!**
- OpenSSL provides secure random numbers
- Mcrypt with /dev/urandom provides medium security
- mt_rand() has low security (for crypto purposes)

/dev/urandom used by MCRYPT_DEV_URANDOM

- **/dev/urandom** is the "unlocked"/non-blocking version of **/dev/random**, it reuses the internal pool to produce more pseudo-random bits
- **/dev/urandom** is considered “less secure” of **/dev/random** because contains less entropy
- **/dev/urandom** is much faster than **/dev/random** (milliseconds compared with seconds)
- There are some environments where are the same, for instance **OpenBSD** and **FreeBSD**

/dev/urandom is considered secure?

- There are **some attacks** that can affect the security of /dev/urandom (forcing re-initialization of the pool)
- In general, even if is less secure than **/dev/random** is used in many cryptographic projects
- We used in ZF2 only as second option

Random number in Zend\Math\Math

- In 2.0.0beta4 we moved Zend\Crypt\Math in the new **Zend\Math**
- We added a couple of methods for RNG:
  - Zend\Math\Math::**randBytes(**$length, $strong = false)
  - Zend\Math\Math::**rand(**$min, $max, $strong = false)
- **randBytes()** generates *$length* random bytes
- **rand()** generates a random number between $min and $max
- If **$strong === true**, the functions use only OpenSSL or Mcrypt (if PHP doesn't support these extensions throw an Exception)

Future works

- More key derivation algorithms (we just merged the **SaltedS2k** in the ZF2 github repository)
- More padding methods for the block ciphers
- More password algorithms (we would like to offer adapters for specific systems)
- Supports encryption/decryption of **streams**
- A new **Zend\Math\Rand** (already in review) component to improve the RNG of ZF2 based on RFC 4086
- Supports authenticated encryption algorithm, like CCM, EAX, etc

References

- N. Ferguson, B. Schneier, T. Kohno, “Cryptography Engineering”, Wiley Publishing, 2010
- D. Boneh "Cryptography course" Stanford University, Coursera - free online courses
- C. Hale, “How to safely store a password”
- S. Vaudenay, “Security Flaws Induced by CBC Padding Applications to SSL, IPSEC, WTLS”, EuroCrypt 2002
- T. Biege, “Analysis of a strong Pseudo Random Number Generator”, 2006
- PHP-CryptLib, all-inclusive cryptographic library for PHP
- Random.org, true random numbers to anyone on the Internet
- stackexchange.com, Recommended numbers of iterations when using PKBDF2
- E.Zimuel, “Cryptography in PHP” Web & PHP Magazine, issue 2/2012
- E.Zimuel, “Cryptography made easy with Zend Framework”

Thank you!

- Email: enrico@zend.com
- Twitter: @ezimuel
- Blog: http://www.zimuel.it
- GitHub: https://github.com/ezimuel

zend The PHP Company

ZEND FRAMEWORK

php·2012
zendcon
October 22-25, 2012 • Santa Clara, CA

Call for Papers is now open!
Submit your talks by
May 21, 2012

Join us at ZendCon

The premier PHP conference!

October 22-25, 2012 – Santa Clara, CA

Conference Themes

PHP in 2012 - *The latest PHP technologies and tools*

Learn how to leverage the latest mobile, HTML 5, testing and PHP best practices

Zend Framework 2 - *Hit the ground running*

Learn how to build faster, more modular and more expandable applications

Development & The Cloud - *A love story*

Learn how the latest developments in cloud-based services, infrastructure and best practices can benefit you

Conference Highlights

- Sessions focused on how to best develop and deploy PHP
- Sessions designed for all knowledge levels
- Intensive tutorials for accelerated learning
- PHP Certification crash courses and testing
- Exhibit hall showcasing the latest products
- Special networking opportunities during meals and events

www.zendcon.com