cryptal

Release latest-e1b9d37

Sep 29, 2018

Contents

Cryptal is a Cryptography Abstraction Layer for PHP.

Warning: This documentation was automatically built from the latest changes in GitHub. It does not necessarily reflect features from any current or upcoming release. Check out https://readthedocs.org/projects/cryptal/ for documentation on supported versions.

This project is comprised of a unified API, which serves as an abstraction layer for various cryptographic libraries, and plugins which add support for the various cryptography primitives.

You can find additional information about this project on the following websites:

- Source repository
- Continuous Integration
- API documentation

Contents:

CHAPTER 1

Introduction

There are several extensions & libraries that provide cryptography primitives for PHP:

- the legacy mcrypt extension
- the OpenSSL extension
- the libsodium extension
- the tomcrypt extension
- probably others I don't know about...

Although these extensions all provide roughtly the same features, the programmatic interface they expose is very different.

Also, very few of those extensions support on-the-fly encryption/decryption.

Cryptal was created to work around these issues by providing a unified interface & transparent support for on-the-fly encryption/decryption using stream filters.

CHAPTER 2

Installation

Cryptal relies on Composer for its installation. It also uses plugins to provide implementations for the various algorithms.

Cryptal can be installed by itself using the following command:

```
$ php composer.php require fpoirotte/cryptal
```

However, the core package only provides a few algorithms using mostly PHP code. Therefore, you will usually want to install additional plugins to get access to more algorithms.

Which plugin to install depends on the algorithms you need to use and whether you're willing to sacrifice a bit of speed and security to get additionnal algorithms.

Cryptal supports 3 types of implementations:

- Assembly code, which provides maximum speed and is usually secure.
- Compiled code, which can be a tiny bit slower, but is often more secure.
- PHP code, which is slower and less secure, but provides support for some niche algorithms.

2.1 Choosing the plugin(s) to install

The following tables list the algorithms provided by each plugin, with their implementation type. "Core" means the algorithm is provided by the Cryptal package itself and does not require any additional plugin to work.

Please note that these lists are only given as an indication of what the underlying library supports. The actual supported algorithms may vary due to differing compilation options or differing versions being used.

2.1.1 Modes of operation

Algorithm	Core	Mcrypt	OpenSSL	LibTomCrypt	LibSodium	Hash	PHP-Crypto
CBC	PHP code	compiled	compiled	compiled	n/a	n/a	compiled
CCM	PHP code	n/a	compiled[1]	compiled	n/a	n/a	compiled
CFB	PHP code	compiled	compiled	compiled	n/a	n/a	compiled
CTR	PHP code	compiled	compiled[1]	compiled	n/a	n/a	compiled
EAX	PHP code	n/a	n/a	compiled	n/a	n/a	n/a
ECB	PHP code	compiled	compiled	compiled	n/a	n/a	compiled
GCM	PHP code	n/a	compiled[1]	compiled	compiled[2]	n/a	compiled
OCB	PHP code	n/a	compiled[1]	compiled	n/a	n/a	n/a
OFB	PHP code	compiled	compiled	compiled	n/a	n/a	compiled

[1] Availability is highly dependent on OpenSSL version, hardware, compilation options and selected cipher. Your mileage may vary.

[2] libsodium only supports AES-256 in GCM mode. Also, this cipher/mode combination is not available unless the processor of the machine running the code has support for the AES-NI instruction set.

2.1.2 Cipher algorithms

Algorithm	Core	Mcrypt	OpenSSL	LibTomCrypt	LibSodium	Hash	PHPCrypto
TripleDES (3DES)	n/a	compiled	compiled	compiled	n/a	n/a	compiled
AES-128	n/a	compiled	compiled	compiled	n/a	n/a	compiled
AES-192	n/a	compiled	compiled	compiled	n/a	n/a	compiled
AES-256	n/a	compiled	compiled	compiled	compiled[2]	n/a	compiled
Blowfish	n/a	compiled	compiled	compiled	n/a	n/a	compiled
Camellia-128	PHP code	n/a	compiled	compiled[3][4]	n/a	n/a	compiled
Camellia-192	PHP code	n/a	compiled	compiled[3][4]	n/a	n/a	compiled
Camellia-256	PHP code	n/a	compiled	compiled[3][4]	n/a	n/a	compiled
CAST5	n/a	compiled	compiled	compiled	n/a	n/a	compiled
ChaCha20 (IETF variant)	PHP code	n/a	n/a	compiled[3][4]	compiled	n/a	n/a
ChaCha20 (OpenSSH variant)	PHP code	n/a	n/a	n/a	n/a	n/a	n/a
DES	n/a	compiled	compiled	compiled	n/a	n/a	compiled
RC2	n/a	compiled	compiled	compiled	n/a	n/a	compiled
RC4	n/a	compiled	compiled	compiled[3][4]	n/a	n/a	compiled
SEED	n/a	n/a	compiled	compiled	n/a	n/a	compiled
Twofish	n/a	compiled	n/a	compiled	n/a	n/a	compiled

[3] Requires version 1.18 or later of LibTomCrypt.
[4] Requires version 0.3.0 or later of the PHP tomcrypt extension.

2.1.3 Hashing algorithms

Algorithm	Core	Mcrypt	OpenSSL	LibTomCrypt	LibSodium	Hash	PHP-Crypto
MD2	n/a	n/a	compiled	compiled	n/a	compiled	compiled
MD4	n/a	n/a	compiled	compiled	n/a	compiled	compiled
MD5	compiled	n/a	compiled	compiled	n/a	compiled	compiled
RIPEMD160	n/a	n/a	compiled	compiled	n/a	compiled	compiled
SHA1	compiled	n/a	compiled	compiled	n/a	compiled	compiled
SHA224	n/a	n/a	compiled	compiled	n/a	compiled	compiled
SHA256	n/a	n/a	compiled	compiled	n/a	compiled	compiled
SHA384	n/a	n/a	compiled	compiled	n/a	compiled	compiled
SHA512	n/a	n/a	compiled	compiled	n/a	compiled	compiled

2.1.4 Message authentication algorithms

Algorithm	Core	Mcrypt	OpenSSL	LibTomCrypt	LibSodium	Hash	PHP-Crypto
CMAC	PHP code	n/a	n/a	compiled	n/a	n/a	compiled
HMAC	n/a	n/a	n/a	compiled	n/a	compiled	compiled
Poly1305	PHP code	n/a	n/a	compiled[34]	n/a	n/a	n/a
UMAC-32	PHP code	n/a	n/a	n/a	n/a	n/a	n/a
UMAC-64	PHP code	n/a	n/a	n/a	n/a	n/a	n/a
UMAC-92	PHP code	n/a	n/a	n/a	n/a	n/a	n/a
UMAC-128	PHP code	n/a	n/a	compiled	n/a	n/a	n/a

2.2 Installing the plugins

Once you have determined the algorithms you are going to use and the plugins providing these algorithms that you want to use, execute the following commands to install the appropriate plugins:

```
$ # Plugin based on the old Mcrypt PHP extension (PHP <= 7.1)
$ php composer.php require fpoirotte/cryptal-mcrypt
$
$ # Plugin based on the OpenSSL PHP extension
$ php composer.php require fpoirotte/cryptal-openssl
$
$ # Plugin based on the LibTomCrypt PHP extension
$ php composer.php require fpoirotte/cryptal-tomcrypt
$
$ # Plugin based on the new LibSodium PHP extension (PHP >= 7.2)
$ php composer.php require fpoirotte/cryptal-sodium
$
$ # Plugin based on the Hash PHP extension
$ php composer.php require fpoirotte/cryptal-hash
$
$ # Plugin based on the PHP-Crypto extension
$ php composer.php require fpoirotte/cryptal-php-crypto
```

CHAPTER 3

Usage

Cryptal provides support for the following main features:

- Encryption/decryption
- Hashes (also known as message digests)
- Message Authentication Codes

For each feature, two sets of interfaces are provided:

- PHP stream filters, which hide the complexity of the operations and provide transparent support for the features.

  This mode of operation is usually adequate for network protocols or when manipulating large files that would not fit into memory.

- Regular PHP interfaces that describe available operations, as well as a central `Registry` to help look up for an actual implementation of some algorithm.

  This mode is usually adequate when working with in-memory strings and small files.

The rest of this document describes the interfaces available for each feature.

3.1 Encryption/decryption

3.1.1 Using stream filters

Warning: When using the stream filters, the library relies mostly on PHP code to handle encryption/decryption. The underlying library is only used to provide the cryptographic primitives for the selected cipher in ECB mode.

This hurts performance a bit, but more importantly, this may diminish your application's security, because some values (keys, IVs, etc.) cannot be safely erased from memory and may linger there even after you are done processing the data.

If you are concerned about these issues, do not use the stream filters.

Encryption

Encrypting data is easy:

```
// Initialize the library
\fpoirotte\Cryptal::init();

// Open a new stream
$stream = stream_socket_client('tcp://localhost:12345');

// Create an encryption context (see below)
$ctx = stream_context_create(
    array(
        'cryptal' => array(
            // Secret key.
            // Size must be compatible with the cipher's expectations.
            'key'   => '0123456789abcdef',

            // Initialization Vector.
            // Size must be compatible with the cipher's expectations.
            'IV'    => 'abcdef0123456789',
        )
    )
);
```

(continues on next page)

(continued from previous page)

```
// Add an encryption layer to the stream.
$filter = stream_filter_append(
    $stream,
    'cryptal.encrypt',
    // We want the data to be encrypted as we write it.
    STREAM_FILTER_WRITE,
    array(
        // Encrypt the data using AES-128 in CTR mode.
        'algorithm' => CipherEnum::CIPHER_AES_128(),
        'mode'      => ModeEnum::MODE_CTR(),

        // Secret key.
        // Size must be compatible with the cipher's expectations.
        'key'       => '0123456789abcdef',

        // Initialization Vector.
        // Size must be compatible with the cipher's expectations.
        'iv'        => 'abcdef0123456789',
    )
);

// We make sure the filter was successfully applied.
if (false === $filter) {
    throw new \Exception('Could not add the encryption layer');
}

// Now that the encryption layer is in place, we can write
// to the stream just like we would normally do.
// Any data written to the stream will be encrypted on the fly.
fwrite($stream, "Some secret message we want to transmit securely");
```

Warning: When adding the filter, the 3rd argument to `stream_filter_append()` (`$read_write`) should be set to either `STREAM_FILTER_WRITE` if the encryption should happen during writes (eg. via `fwrite()`), or `STREAM_FILTER_READ` if it should happen during reads (eg. via `fread()` or `fgets()`).

Using the default value (`STREAM_FILTER_ALL`) means the same filter is applied to both operations, which is not supported and may produce unexpected results.

Here's another example, this time using Authenticated Encryption with Associated Data (AEAD):

```
@TODO
```

Decryption

Decryption works the same way. Just substitute `cryptal.decrypt` in place of `cryptal.encrypt` when adding the filter.

When using Authenticated Encryption, @TODO

Filter parameters for `cryptal.encrypt`/`cryptal.decrypt`

When using streams, the following options may be used when adding the filter to control the way encryption/decryption is performed:

Table 1: Parameters for cryptal.encrypt/cryptal.decrypt

Name	Optional	Expected type	Description
`mode`	yes	`\fpoirotte\Cryptal\ModeEnum`	The cipher's mode of operations to use. This parameter is important as the various modes offer different security garantees. Make sure you have read documentation on the various modes and their implications before setting this value.
`algorithm`	yes	`\fpoirotte\Cryptal\CipherEnum`	The cipher algorithm to use to encrypt/decrypt the data. This parameter is important as the various ciphers offer different security garantees. Make sure you have read documentation on the various ciphers and their limitations before setting this value.
`allowUnsafe`	no	boolean	Whether userland PHP implementations may be used or not. Defaults to `false`. While those implementations add support for some rarely used algorithms, they are usually way slower than implementations based on PHP extensions. Also, those implementations are considered unsafe because they cannot protect the application from certain classes of attacks like PHP extensions usually do (eg. side-channel attacks). Last but not least, when using those implementations, secret values may reside in memory for longer than is actually necessary (possibly even longer than the program's actual execution time), making them vulnerable to memory forensic techniques and such.
`data`	no	string	Additional Data to authenticate when using Authenticated Encryption
`iv`	yes/no	string	Initialization Vector for the cipher. Whether this parameter is optional or not depends of the encryption/decryption mode used.
`key`	yes	string	Symmetric key to use for encryption/decryption
`padding`	no	`\fpoirotte\Cryptal\PaddingInterface`	Padding scheme to use. Defaults to no padding.
`tag`	no	string	Authentication tag for the current block. This value is set by the filter during encryption of a block. It should be set manually when decrypting, before passing a block to decrypt to the stream.
`tagLength`	no	integer	Desired tag length (in bytes) when using Authenticated Encryption. Defaults to 16 bytes (128 bits). This parameters is only used during encryption, as it can be deduced from the `tag`'s actual length when decrypting.

Padding

By default, no padding is applied to streams (ie. the padding scheme is set to an instance of `fpoirotte\Cryptal\Padding\None`).

If you need to use another padding scheme, you can easily swap the default for an alternate implementation. Just set the `padding` filter parameter to an instance of the padding scheme to use when adding the filter:

```
use fpoirotte\Cryptal\Padding\AnsiX923;

// Open the stream
$stream = fopen(..., 'wb');

stream_filter_append(
    $stream,
    'cryptal.encrypt',
    STREAM_FILTER_WRITE,
    array(
            'key'       => '0123456789abcdef',
            'IV'        => 'abcdef0123456789',
            'algorithm' => CipherEnum::CIPHER_AES_128(),
            'mode'      => ModeEnum::MODE_CTR(),

            // Use the ANSI X.923 padding scheme.
            'padding'   => new AnsiX923,
    )
);

// Do something with the stream...
```

3.1.2 Using the registry

The following snippet shows how to retrieve an implementation of the AES cipher in ECB mode for encryption/decryption:

```
use \fpoirotte\Cryptal\Registry;
use \fpoirotte\Cryptal\Padding\None;
use \fpoirotte\Cryptal\CipherEnum;
use \fpoirotte\Cryptal\ModeEnum;

// Initialize the library
\fpoirotte\Cryptal::init();

// Retrieve an implementation for the chosen cipher & mode.
// See fpoirotte\Cryptal\CipherEnum and fpoirotte\Cryptal\ModeEnum
// for a list of valid ciphers/modes.
$impl = Registry::buildCipher(
    CipherEnum::CIPHER_AES_128(),   // Cipher to use
    ModeEnum::MODE_ECB(),           // Mode of operations
    new None(),                     // Padding scheme
    '0123456789abcdef'              // Secret key
    0,                              // Desired tag length (AEAD-only)
    true                            // Whether using plain PHP code
                                    // is okay (less secure/slower)
);
```

(continues on next page)

(continued from previous page)

```
// Generate an appropriate Initialization Vector
$iv = 'abcdef0123456789';

// Since no padding was used in this example, the plaintext's length
// must be a multiple of the cipher's block size. That's 16 bytes for AES.
// Use $impl->getBlockSize() if necessary to retrieve the block size.
$plaintext = "Some secret text";
var_dump(bin2hex($plaintext));

// Encrypt the data
$ciphertext = $impl->encrypt($iv, $plaintext);
var_dump(bin2hex($ciphertext));

// Decryption is just as easy
$decoded = $impl->decrypt($iv, $ciphertext);
var_dump(bin2hex($decoded));
```

Here's another example, this time using Authenticated Encryption with Associated Data (AEAD):

```
@TODO
```

3.2 Hashes (message digests)

3.2.1 Using stream filters

Replicating `md5_file()` using Cryptal

Hashing data using streams is really easy. For example, to obtain an MD5 message digest for a file (similar to what the PHP `md5_file()` function returns), the following snippet can be used:

```
// Initialize the library
\fpoirotte\Cryptal::init();

// Open the binary file for reading.
$fp = fopen("/path/to/some.data", "rb");

// Add the hashing filter to the stream.
stream_filter_append(
    $fp,
    'cryptal.hash',
    // We want to compute the hash based on data read from the file.
    STREAM_FILTER_READ,
    array(
        'algorithm' => HashEnum::HASH_MD5()
    )
);

// Read the resulting message digest (returned in raw form).
// The MD5 algorithm produces a 128-bit hash (16 bytes).
$hash = stream_get_contents($fp);
```

Warning: When adding the filter, the 3rd argument to `stream_filter_append()` (`$read_write`) should be set to either `STREAM_FILTER_WRITE` if the hashing should happen during writes (eg. via `fwrite()`), or `STREAM_FILTER_READ` if it should happen during reads (eg. via `fread()` or `fgets()`).

Using the default value (`STREAM_FILTER_ALL`) means the same filter is applied to both operations, which is not supported and may produce unexpected results.

Filter parameters for `cryptal.hash`

When using streams, the following options may be used when adding the filter to control the way the message digest is computed:

Table 2: Parameters for cryptal.hash

Name	Optional	Expected type	Description
`algorithm`	yes	`\fpoirotte\Cryptal\HashEnum`	The algorithm to use to hash the data. This parameter is important as the various algorithms offer different security garantees. Make sure you have read documentation on the various algorithms and their limitations before setting this value.
`allowUnsafe`	yes	boolean	Whether userland PHP implementations may be used or not. Defaults to `false`. While those implementations add support for some rarely used algorithms, they are usually way slower than implementations based on PHP extensions. Also, those implementations are considered unsafe because they cannot protect the application from certain classes of attacks like PHP extensions usually do (eg. side-channel attacks). Last but not least, when using those implementations, secret values may reside in memory for longer than is actually necessary (possibly even longer than the program's actual execution time), making them vulnerable to memory forensic techniques and such.

3.2.2 Using the registry

Hashing data using the registry is easy too:

```
use \fpoirotte\Cryptal\Registry;
use \fpoirotte\Cryptal\HashEnum;

// Initialize the library
\fpoirotte\Cryptal::init();

// Grab an instance of the hash implementation.
// The last argument indicates whether implementations based on
// userland PHP code can be returned too.
// By default, they are not because they are usually slower and
// more prone to timing attacks.
$hasher = Registry;::buildHash(HashEnum::HASH_MD5(), true);
```

(continues on next page)

(continued from previous page)

```
// Pass the data to hash to the implementation.
$hasher->update(file_get_contents("/path/to/some.data"));

// Retrieve the resulting hash.
// The argument given to finish() decides whether the hash
// should be returned in raw binary form (true) or not (false).
$hash = $hasher->finish(true);
```

3.3 Message Authentication Codes (MAC)

Compared to the previous features, message authentication codes can be a bit tricky to deal with. First, they actually require 2 algorithms to work:

- One algorithm to process the input data (to compute intermediate values), called the "inner algorithm" hereafter.
- One algorithm to compute the final output (a message authentication code, also know as a tag), called the "outer algorithm" in the rest of this section.

The algorithms' names are usually combined to obtain a more descriptive (and unique) name for the whole construct. So for example, "HMAC-MD5" is often used to refer to the HMAC outer algorithm applied to the MD5 hashing algorithm.

But it gets trickier: the type of the first algorithm depends on the second one. Some "outer algorithms" (eg. HMAC) expect a hashing algorithm as their "inner algorithm". Some (eg. CMAC & UMAC) expect a cipher algorithm as their "inner algorithm". And finally, some (eg. Poly1305) do not use an inner algorithm at all. Some "outer algorithms" also impose further limitations on the "inner algorithm" such as restrictions on the cipher's block size for cipher-based message authentication codes.

Last but not least, every combination of algorithms requires a secret key, known only by the two parties trying to prevent any message tampering. A few algorithms also require what's known as a "nonce", to make the output less predictable.

Before computing any MAC, we suggest that you first read some documentation about whatever algorithm you plan on using and then learn about its specific requirements and limitations.

3.3.1 Using stream filters

Quick example: HMAC-MD5 on a file

To compute a MAC using the stream interface, just use code similar to this one:

```
// Initialize the library
\fpoirotte\Cryptal::init();

// Open the binary file for reading.
$macGiver = fopen("/path/to/some.data", "rb");

// Add the MAC filter to the stream.
stream_filter_append(
    $macGiver,
    'cryptal.mac',
    // We want to compute the MAC based on data read from the file.
    STREAM_FILTER_READ,
```

(continues on next page)

(continued from previous page)

```
    array(
        'algorithm'         => MacEnum::MAC_HMAC(),
        'innerAlgorithm'    => HashEnum::HASH_MD5(),

        // Size must be compatible with the algorithms in use.
        'key'               => '0123456789abcdef',
    )
);

// Retrieve the Message Authentication Code in raw binary form.
// The HMAC-MD5 algorithm produces a 128-bit hash (16 bytes).
$mac = stream_get_contents($macGiver);
```

Warning: When adding the filter, the 3rd argument to `stream_filter_append()` (`$read_write`) should be set to either `STREAM_FILTER_WRITE` if the tag computation should happen during writes (eg. via `fwrite()`), or `STREAM_FILTER_READ` if it should happen during reads (eg. via `fread()` or `fgets()`).

Using the default value (`STREAM_FILTER_ALL`) means the same filter is applied to both operations, which is not supported and may produce unexpected results.

Filter parameters for `cryptal.mac`

When using streams, the following options may be used when adding the filter to control the way the message authentication code is computed:

Table 3: Parameters for cryptal.mac

Name	Optional	Expected type	Description
`algorithm`	yes	`\fpoirotte\Cryptal\MacEnum`	Outer algorithm to use to perform the computation. This parameter is important as the various algorithms offer different security garantees. Make sure you have read documentation on the various algorithms and their limitations before setting this value.
`innerAlgorithm`	yes	`\fpoirotte\Cryptal\SubAlgorithmEnum`	Inner algorithm to use to perform the computation. Depending on the selected `algorithm`, this parameter should be set to either an instance of `\fpoirotte\Cryptal\CipherEnum` or `\fpoirotte\Cryptal\HashEnum`. This parameter is important as the various algorithms offer different security garantees. Make sure you have read documentation on the various algorithms and their limitations before setting this value.
`allowUnsafe`	no	boolean	Whether userland PHP implementations may be used or not. Defaults to `false`. While those implementations add support for some rarely used algorithms, they are usually way slower than implementations based on PHP extensions. Also, those implementations are considered unsafe because they cannot protect the application from certain classes of attacks like PHP extensions usually do (eg. side-channel attacks). Last but not least, when using those implementations, secret values may reside in memory for longer than is actually necessary (possibly even longer than the program's actual execution time), making them vulnerable to memory forensic techniques and such.
`nonce`	yes/no	string	Nonce to make the output less predictable. Whether this parameter is optional or not depends on the selected `algorithm`/`innerAlgorithm`.
`key`	yes	string	Symmetric key to use for the computation

3.3.2 Using the registry

Computing a MAC using the registry is very similar to hashing:

```
use \fpoirotte\Cryptal\Registry;
use \fpoirotte\Cryptal\MacEnum;
use \fpoirotte\Cryptal\HashEnum;

// Initialize the library
\fpoirotte\Cryptal::init();

// Grab an instance of the MAC implementation.
// The last argument indicates whether implementations based on
// userland PHP code can be returned too.
// By default, they are not because they are usually slower and
// more prone to timing attacks.
$macGiver = Registry;::buildMac(
```

(continues on next page)

(continued from previous page)

```
    MacEnum::MAC_HMAC(),
    HashEnum::HASH_MD5(),
    '0123456789abcdef',     // Secret key
    '',                     // Nonce, for algorithms that require one
    true
);

// Pass the data to process to the implementation.
$macGiver->update(file_get_contents("/path/to/some.data"));

// Retrieve the resulting tag/MAC.
// The argument given to finish() decides whether the tag
// should be returned in raw binary form (true) or not (false).
$tag = $macGiver->finish(true);
```

3.4 Miscelleanous features

In addition to the ones listed above, Cryptal also provides the following filters:

- `cryptal.binify` can be used to convert an hexadecimal-encoded string into its binary counterpart on the fly (eg. `4372797074616c` → `Cryptal`).
- `cryptal.hexify` does the reverse operation and can be used to convert a string into its hexadecimal representation (eg. `Cryptal` → `4372797074616c`). It accepts a single option named `uppercase`. When set to `true`, the filter will generate its output using uppercase characters instead of the (default) lowercase characters.

CHAPTER 4

Implementers

This page contains guidelines for implementers.

The key words **MUST**, **MUST NOT**, **REQUIRED**, **SHALL**, **SHALL NOT**, **SHOULD**, **SHOULD NOT**, **RECOMMENDED**, **MAY**, and **OPTIONAL** in this document are to be interpreted as described in **RFC 2119**.

4.1 Guidelines

New implementations **MUST** be delivered as Composer packages. Such a package **MUST** provide an implementation for one or several of the interfaces defined in the `\fpoirotte\Cryptal\Implementers` namespace.

It is **RECOMMENDED** that implementers always support as many algorithms recognized by the Cryptography Abstraction Layer as the underlying library and Cryptal permit when adding support for a feature.

The following sections describes how to turn a regular Composer package into a Cryptal plugin.

4.2 Creating a new plugin

4.2.1 Update your `composer.json` file

The basic skeleton for a plugin's `composer.json` looks like this:

```
{
    "name": "fpoirotte/cryptal-tomcrypt",
    "type": "cryptal-plugin",
    "description": "Plugin for Cryptal based on LibTomcrypt",
    "license": "MIT",
    "require": {
        "php": ">=5.4",
        "fpoirotte/cryptal": "*"
    },
```

(continues on next page)

(continued from previous page)

```
    "provide": {
        "fpoirotte/cryptal-implementation": "*"
    },
    "autoload": {
        "psr-4": {
            "fpoirotte\\Cryptal\\Plugins\\Tomcrypt": "src/"
        }
    },
    "extra": {
        "cryptal.entrypoint": "fpoirotte\\Cryptal\\Plugins\\Tomcrypt\\Entrypoint"
    }
}
```

There are four important things to note:

- The package's type **MUST** be set to `cryptal-plugin` in order for the plugin to be properly recognized as such.
- The package **MUST** contain a requirement on `fpoirotte/cryptal` as part of the `require` section, so that the core files needed to load and use the plugin are available at runtime.
- To make it easier to find compatible plugins for Cryptal on Packagist, an implementation **SHOULD** `provide` the `fpoirotte/cryptal-implementation` virtual package in its `composer.json` file.

  The version number associated with the provided virtual package **SHOULD** be set to a sensible value.
- The package **MUST** declare a key named `cryptal.entrypoint` in the `extra` section of their `composer.json` file, pointing to a class that implements the `fpoirotte\Cryptal\Implementers\PluginInterface` interface.

  If your plugin provides implementations for several features and you would like each feature to use its own entry point, you may also use an array of entry points here in place of a string.

4.2.2 Write the code for the entry point

The entry point is responsible for registering any algorithm implemented by the plugin into Cryptal's registry.

Assuming the plugin adds support for the AES cipher using 128 bit keys (AES-128) in Electronic Codebook (ECB) mode, the MD5 hash algorithm and the HMAC message authentication code, an entry point may look like this:

```
namespace fpoirotte\Cryptal\Plugins\Tomcrypt;

use fpoirotte\Cryptal\Implementers\PluginInterface;
use fpoirotte\Cryptal\ImplementationTypeEnum;
use fpoirotte\Cryptal\CipherEnum;
use fpoirotte\Cryptal\ModeEnum;
use fpoirotte\Cryptal\HashEnum;
use fpoirotte\Cryptal\MacEnum;

class Entrypoint implements PluginInterface
{
    public function registerAlgorithms(RegistryWrapper $registry)
    {
        // Declare support for AES-128 in ECB mode
        $registry->addCipher(
            '\\fpoirotte\\cryptal\\Plugins\\Tomcrypt\\Aes',
            CipherEnum::CIPHER_AES_128(),
```

(continues on next page)

(continued from previous page)

```
            ModeEnum::MODE_ECB(),
            ImplementationTypeEnum::TYPE_COMPILED()
        );

        // Declare support for the MD5 message digest algorithm
        $registry->addHash(
            '\\fpoirotte\\cryptal\\Plugins\\Tomcrypt\\Md5',
            HashEnum::HASH_MD5(),
            ImplementationTypeEnum::TYPE_COMPILED()
        );

        // Declare support for the HMAC message authenticator algorithm
        $registry->addMac(
            '\\fpoirotte\\cryptal\\Plugins\\Tomcrypt\\Hmac',
            MacEnum::MAC_HMAC(),
            ImplementationTypeEnum::TYPE_COMPILED()
        );
}
```

The `RegistryWrapper` provides 3 methods, meant to declare support for new ciphers (addCipher), hash algorithms (addHash) and message authentication codes (addMac).

Each of these methods expects the full path to a class providing the algorithm as their first argument, followed by Cryptal's identifier for that algorithm and an identifier for the implementation type.

For ciphers, the algorithm identifier is made of two arguments:

- The cipher's identifier itself (one of the values declared in the `CipherEnum` enumeration)
- The mode of operations which can be applied to this cipher (one of the values declared in the `ModeEnum` enumeration)

For hash and MAC algorithms, just pass the algorithm's identifier defined in `HashEnum` or `MacEnum`, respectively.

The implementation type **SHOULD** match the actual nature of the algorithm's implementation:

- `TYPE_ASSEMBLY()` **SHOULD** be used when the underlying code is known to be optimized for speed/uses assembly code.
- `TYPE_COMPILED()` **SHOULD** be used for other forms of compiled code, such as code from a PHP extension coded in C or C++.
- `TYPE_USERLAND()` **SHOULD** be used for algorithms implemented using regular (userland) PHP code, as opposed to code from a PHP extension.

Cryptal uses this information at runtime to determine the fastest/most secure implementation it can use.

4.3 Available plugins

You can browse the list of existing plugins for Cryptal on this page

Badges: 

Index

R