Chapter 9. Security and Encryption

PHP is a remarkably easy-to-use language because of the forgiving nature of the runtime. Even when you make a mistake, PHP will try to infer what you *meant* to do and, often, keep executing your program just fine. Unfortunately, this strength is also seen by some developers as a key weakness. By being forgiving, PHP allows for a sheer amount of "bad" code to continue functioning as if it were correct.

Worse still, much of this "bad" code finds its way into tutorials, leading developers to copy and paste it into their own projects and perpetuate the cycle. This forgiving runtime and long history of PHP have led to the perception that PHP itself is insecure. In reality, it's easy to use *any* programming language insecurely.

Natively, PHP supports the ability to quickly and easily filter malicious input and sanitize user data. In a web context, this utility is critical to keeping user information safe from malicious inputs or attacks. PHP also exposes well-defined functions for both securely hashing and securely validating passwords during authentication.

NOTE

PHP's default password hashing and validation functions both leverage secure hashing algorithms and constant-time, secure implementations. This protects your application against niche attacks like those using timing information in an attempt to extract information. Attempting to implement hashing (or validation) yourself would likely expose your application to risks for which PHP has already accounted.

The language also makes cryptography—for both encryption and signatures—simple for developers. Native, high-level interfaces protect you from the kinds of mistakes easily made elsewhere.[1] In fact, PHP is one of the *easiest* languages in which developers can leverage strong, modern cryptography without relying on a third-party extension or implementation!

Legacy Encryption

Earlier versions of PHP shipped with an extension called mcrypt. This extension exposed the lower-level mcrypt library, allowing developers to easily leverage a variety of block ciphers and hash algorithms. It was removed in PHP 7.2 in favor of the newer sodium extension, but it can still be manually installed via the PHP Extension Community Library (PECL).[2]

WARNING

While still available, the mcrypt library has not been updated in over a decade and should not be used in new projects. For any new encryption needs, use either PHP's bindings against OpenSSL or the native sodium extension.

PHP also supports direct use of the OpenSSL library through an extension. This is helpful when building a system that must interoperate with legacy cryptographic libraries. However, the extension does not expose the full-feature offering of OpenSSL to PHP; it would be useful to review the functions and features exposed to identify whether PHP's implementation would be useful to your application.

In any case, the newer sodium interfaces support a wide range of cryptographic operations in PHP and should be preferred over either OpenSSL or mcrypt.

Sodium

PHP formally added the sodium extension (also known as *Libsodium*) as a core extension in version 7.2, released in late 2017.[3] This library supports high-level abstractions for encryption, cryptographic signatures, password hashing, and more. It is itself an open source fork of an earlier project, the Networking and Cryptography Library (NaCl) by Daniel J. Bernstein.

Both projects add easy-to-use, high-speed tools for developers who need to work with encryption. The opinionated nature of their exposed interfaces aims to make cryptography *safe* and proactively avoid the pitfalls presented by other low-level tools. Shipping with well-defined opinionated interfaces helps developers make the right choices about algorithm implementations and defaults because those very choices (and potential mistakes) are entirely

abstracted away and presented in safe, straightforward functions for everyday use.

NOTE

The only problem with sodium and its exposed interfaces is a lack of brevity. Every function is prefixed with `sodium_`, and every constant with `SODIUM_`. The high level of descriptiveness in both function and constant names makes it easy to understand what is happening in code. However, this also leads to incredibly long and potentially distracting function names, like `sodium_crypto_sign_keypair_from_secretkey_and_publickey()`.

While sodium is bundled as a core extension for PHP, it also exposes bindings for several other languages as well. It is fully interoperable with everything from .NET to Go to Java to Python.

Unlike many other cryptographic libraries, sodium focuses primarily on *authenticated* encryption. Every piece of data is automatically paired with an authentication tag that the library can later use to validate the integrity of the underlying plaintext. If this tag is missing or invalid, the library throws an error to alert the developer that the associated plaintext is unreliable.

This use of authentication isn't unique—the Galois/Counter Mode (GCM) of the Advanced Encryption Standard (AES) does effectively the same thing. However, other libraries often leave authentication and validation of an authentication tag as an exercise for the developer. There are a number of tutorials, books, and Stack Overflow discussions that point to a proper implementation of AES but omit the validation of the GCM tag affixed to a message! The sodium extension abstracts both authentication and validation away and provides a clear, concise implementation, as illustrated in Example 9-1.

Example 9-1. Authenticated encryption and decryption in sodium

```
$nonce = random_bytes(SODIUM_CRYPTO_SECRETBOX_NONCEBYTE
                                                    ❶

$key = random_bytes(SODIUM_CRYPTO_SECRETBOX_KEYBYTES);
                                                    ❷
```

```
$message = 'This is a super secret communication!';

$ciphertext = sodium_crypto_secretbox($message, $nonce,
                                                    ❸


$output = bin2hex($nonce . $ciphertext);
                                                    ❹


// Decoding and decryption reverses the preceding steps
$bytes = hex2bin($input);
                                                    ❺

$nonce = substr($bytes, 0, SODIUM_CRYPTO_SECRETBOX_NONC
$ciphertext = substr($bytes, SODIUM_CRYPTO_SECRETBOX_NC

$plaintext = sodium_crypto_secretbox_open($ciphertext,
                                                    ❻


if ($plaintext === false) {
                                                    ❼

    throw new Exception('Unable to decrypt!');
}
```

❶ Encryption algorithms are deterministic—the same input will always produce the same output. To ensure that encrypting the same data with the same key returns *different* outputs, you need to use a sufficiently random *nonce* (number used once) to initialize the algorithm each time.

❷ For symmetric encryption, you leverage a single, shared key used to both encrypt and decrypt data. While the key in this example is random, you would likely store this encryption key somewhere outside the application for safekeeping.

❸ Encrypting is incredibly straightforward. Sodium chooses the algorithm and cipher mode for you—all you provide is the message, the random nonce, and the symmetric key. The underlying library does the rest!

❹ When exporting the encrypted value (either to send to another party or to store on disk), you need to keep track of both the random nonce and the subsequent ciphertext. The nonce itself is not secret, so storing it in

plaintext alongside the encrypted value is safe (and encouraged). Converting the raw bytes from binary to hexadecimal is an effective way to prepare data for an API request or to store in a database field.

❺ Since the output of encryption is encoded in hexadecimal, you must first decode things back to raw bytes and then separate the nonce and ciphertext components before proceeding with decryption.

❻ To extract the plaintext value back out of an encrypted field, provide the ciphertext with its associated nonce and the original encryption key. The library pulls the plaintext bytes back out and returns them to you.

❼ Internally, the encryption library also adds (and validates) an authentication tag on every encrypted message. If the authentication tag fails to validate during decryption, sodium will return a literal `false` rather than the original plaintext. This is a sign to you that the message has been tampered with (either intentionally or accidentally) and should not be trusted.

Sodium also introduces an efficient means to handle public-key cryptography. In this paradigm, encryption uses one key (a known or publicly exposed key), while decryption uses an entirely different key known only to the recipient of the message. This two-part key system is ideal when exchanging data between two separate parties over a potentially untrusted medium (like exchanging banking information between a user and their bank over the public internet). In fact, the HTTPS connections used for most websites on the modern internet leverage public-key cryptography under the hood within the browser.

In legacy systems like RSA, you need to keep track of relatively large cryptographic keys in order to exchange information safely. In 2022, the minimum recommended key size for RSA was 3,072 bits; in many situations, developers will default to 4,096 bits to retain the keys' safety against future computing enhancements. Juggling keys of this size can be difficult in some situations. In addition, traditional RSA can only encrypt 256 bytes of data. If you want to encrypt a larger message, you are forced to do the following:

1. Create a 256-bit random key.
2. Use that 256-bit key to *symmetrically* encrypt a message.
3. Use RSA to encrypt the symmetric key.
4. Share both the encrypted message and the encrypted key that protects it.

This is a workable solution, but the steps involved can easily become complicated and introduce unnecessary complexity for a development team building a project that *just happens* to include encryption. Thankfully, sodium fixes this almost entirely!

Sodium's public-key interfaces leverage elliptic-curve cryptography (ECC) rather than RSA. RSA uses prime numbers and exponentiation to create the known (public) and unknown (private) components of the two-key system used for encryption. ECC instead uses the geometry and particular forms of arithmetic against a well-defined elliptic curve. Whereas RSA's public and private components are numbers used for exponentiation, ECC's public and private components are literal *x* and *y* coordinates on a geometric curve.

With ECC, a 256-bit key has strength equivalent to that of a 3,072-bit RSA key. Further, sodium's choice of cryptographic primitives means that its keys are just numbers (rather than *x* and *y* coordinates as with most other ECC implementations)—a 256-bit ECC key for sodium is simply a 32-byte integer!

In addition, sodium entirely abstracts the "create a random symmetric key and separately encrypt it" workflow from developers, making asymmetric encryption just as simple in PHP as Example 9-1 demonstrated for symmetric encryption. Example 9-2 illustrates exactly how this form of encryption works, along with the key exchange required between participants.

Example 9-2. Asymmetric encryption and decryption in sodium

```
$bobKeypair = sodium_crypto_box_keypair();
                                          ❶

$bobPublic = sodium_crypto_box_publickey($bobKeypair);
                                          ❷

$bobSecret = sodium_crypto_box_secretkey($bobKeypair);

$nonce = random_bytes(SODIUM_CRYPTO_SECRETBOX_NONCEBYTE
                                          ❸


$message = 'Attack at dawn.';

$alicePublic = '...';
                                          ❹
```

```
$keyExchange = sodium_crypto_box_keypair_from_secretkey
                                ❺

    $bobSecret,
    $alicePublic
);

$ciphertext = sodium_crypto_box($message, $nonce, $keyE
                                ❻


$output = bin2hex($nonce . $ciphertext);
                                ❼


// Decrypting the message reverses the key exchange prc
$keyExchange2 = sodium_crypto_box_keypair_from_secretke
                                ❽

    $aliceSecret,
    $bobPublic
);

$plaintext = sodium_crypto_box_open($ciphertext, $nonce
                                ❾


if ($plaintext === false) {
                                ❿

    throw new Exception('Unable to decrypt!');
}
```

❶ In practice, both parties would generate their public/private key pairs locally and distribute their public keys directly. The `sodium_crypto_box_keypair()` function creates a random key pair each time so, conceivably, you would only need to do this once, so long as the secret key remains private.

❷ Both the public and secret components of the key pair can be extracted separately. This makes it easy to extract and communicate only the public key to a third party but also makes the secret key separately available for use in the later key-exchange operation.

❸ As with symmetric encryption, you need a random nonce for each asymmetric encryption operation.

❹

Alice's public key was potentially distributed via a direct communication channel or is otherwise already known.

5. The key exchange here isn't being used to agree upon a new key; it's merely combining Bob's secret key with Alice's public key to prepare for Bob to encrypt a message that can only be read by Alice.

6. Again, sodium chooses the algorithms and cipher modes involved. All you need to do is provide the data, random nonce, and keys, and the library does the rest.

7. When sending the message, it's useful to concatenate the nonce and ciphertext, then encode the raw bytes as something more readily sent over an HTTP channel. Hexadecimal is a common choice, but Base64 encoding is equally valid.

8. On the receiving end, Alice needs to combine her own secret key with Bob's public key in order to decrypt a message that could have only been encrypted by Bob.

9. Extracting the plaintext is as straightforward as encryption was in step 6!

10. As with symmetric encryption, this operation is authenticated. If the encryption fails for any reason (e.g., Bob's public key was invalid) or the authentication tag fails to validate, sodium returns a literal `false` to indicate the untrustworthiness of the message.

Randomness

In the world of encryption, leveraging a proper source of randomness is critical to protecting any sort of data. Older tutorials heavily reference PHP's `mt_rand()` function, which is a pseudorandom number generator based on the Mersenne Twister algorithm.

Unfortunately, while the output of this function appears random to a casual observer, it is not a cryptographically safe source of randomness. Instead, leverage PHP's `random_bytes()` and `random_int()` functions for anything critical. Both of these functions leverage the cryptographically secure source of randomness built into your local operating system.

WARNING

A *cryptographically secure pseudorandom number generator* (CSPRNG) is one with an output that is indistinguishable from random noise. Algorithms like the Mersenne Twister are "random enough" to fool a human into thinking they're safe. In reality, they're easy for a computer to predict or even crack, given a series of previous outputs. If an attacker can reliably predict the output of your random number generator, they can conceivably decrypt anything you try to protect based on that generator!

The following recipes cover some of the most important security- and encryption-related concepts in PHP. You'll learn about input validation, proper password storage, and the use of PHP's sodium interface.

9.1 Filtering, Validating, and Sanitizing User Input

Problem

You want to validate a specific value provided by an otherwise untrusted user prior to using it elsewhere in your application.

Solution

Use the `filter_var()` function to validate that the value matches a specific expectation, as follows:

```
$email = $_GET['email'];

$filtered = filter_var($email, FILTER_VALIDATE_EMAIL);
```

Discussion

PHP's filtering extension empowers you to either validate that data matches a specific format or type or sanitize any data that fails that validation. The subtle difference between the two options—validation versus sanitization—is that sanitization removes invalid characters from a value, whereas validation explicitly returns `false` if the final, sanitized input is not of a valid type.

In the Solution example, untrusted user input is explicitly validated as a valid email address. Example 9-3 demonstrates the behavior of this form of validation across multiple potential inputs.

Example 9-3. Testing email validation

```
function validate(string $data): mixed
{
    return filter_var($data, FILTER_VALIDATE_EMAIL);
}

validate('blah@example.com');
                            ❶

validate('1234');
                            ❷

validate('1234@example.com<test>');
                            ❸
```

Returns `blah@example.com` ❶

Returns `false` ❷

Returns `false` ❸

The alternative to the preceding example is to *sanitize* user input such that invalid characters are stripped from the entry. The result of this sanitization is guaranteed to match a specific character set, but there is no guarantee that the result is a valid input. For example, Example 9-4 properly sanitizes every possible input string even though two results are invalid email addresses.

Example 9-4. Testing email sanitization

```
function sanitize(string $data): mixed
{
    return filter_var($data, FILTER_SANITIZE_EMAIL);
}

sanitize('blah@example.com');
                            ❶

sanitize('1234');
                            ❷

sanitize('1234@example.com<test>');
```

Returns `blah@example.com` ❶

Returns `1234` ❷

Returns `1234@example.comtest` ❸

Whether you want to sanitize or validate your input data depends highly on what you intend to use the resulting data for. If you merely want to keep invalid characters out of a data storage engine, sanitization might be the right approach. If you want to ensure that data is both within the expected character set *and* a valid entry, data validation is a safer tool.

Both approaches are supported equally well by `filter_var()` based on various types of filters in PHP. Specifically, PHP supports validation filters (enumerated in Table 9-1), sanitization filters (enumerated in Table 9-2), and filters falling under neither category (see Table 9-3). The `filter_var()` function also supports an optional third parameter for flags that enable more granular control of the overall output of a filter operation.

Table 9-1. Validation filters supported by PHP

ID	Options	Flags
`FILTER_VALIDATE_BOOLEAN`	`default`	`FILTER_NULL_ON_FAILURE`
`FILTER_VALIDATE_DOMAIN`	`default`	`FILTER_FLAG_HOSTNAME`, `FILTER_NULL_ON_FAILURE`

ID	Options	Flags
FILTER_VALIDATE_EMAIL	default	FILTER_FLAG_EMAIL_UNICODE, FILTER_NULL_ON_FAILURE
FILTER_VALIDATE_FLOAT	default, decimal, min_range, max_range	FILTER_FLAG_ALLOW_THOUSANDS, FILTER_NULL_ON_FAILURE
FILTER_VALIDATE_INT	default, max_range, min_range	FILTER_FLAG_ALLOW_OCTAL, FILTER_FLAG_ALLOW_HEX, FILTER_NULL_ON_FAILURE
FILTER_VALIDATE_IP	default	FILTER_FLAG_IPV4, FILTER_FLAG_IPV6, FILTER_FLAG_NO_PRIV_RANGE,

ID	Options	Flags
		FILTER_FLAG_NO_RES_RANGE, FILTER_NULL_ON_FAILURE
FILTER_VALIDATE_MAC	default	FILTER_NULL_ON_FAILURE
FILTER_VALIDATE_REGEXP	default, regexp	FILTER_NULL_ON_FAILURE
FILTER_VALIDATE_URL	default	FILTER_FLAG_SCHEME_REQUIRED, FILTER_FLAG_HOST_REQUIRED, FILTER_FLAG_PATH_REQUIRED, FILTER_FLAG_QUERY_REQUIRED, FILTER_NULL_ON_FAILURE

Table 9-2. Sanitization filters supported by PHP

ID	Flags
FILTER_SANITIZE_EMAIL	
FILTER_SANITIZE_ENCODED	FILTER_FLAG_STRIP_LOW, FILTER_FLAG_STRIP_HIGH, FILTER_FLAG_STRIP_BACKTICK, FILTER_FLAG_ENCODE_HIGH, FILTER_FLAG_ENCODE_LOW

ID	Flags
FILTER_SANITIZE_ADD_SLASHES	
FILTER_SANITIZE_NUMBER_FLOAT	FILTER_FLAG_ALLOW_FRACTION, FILTER_FLAG_ALLOW_THOUSANDS, FILTER_FLAG_ALLOW_SCIENTIFIC
FILTER_SANITIZE_NUMBER_INT	
FILTER_SANITIZE_SPECIAL_CHARS	FILTER_FLAG_STRIP_LOW, FILTER_FLAG_STRIP_HIGH, FILTER_FLAG_STRIP_BACKTICK, FILTER_FLAG_ENCODE_HIGH
FILTER_SANITIZE_FULL_SPECIAL_CHARS	FILTER_FLAG_NO_ENCODE_QUOTES
FILTER_SANITIZE_URL	

Table 9-3. Miscellaneous filters supported by PHP

ID	Options	Flags	Description

ID	Options	Flags	Description
`FILTER_CALLBACK`	`callable` function or method	All flags are ignored.	Calls a user-defined function to filter data

The validation filters also accept an array of options at runtime. This gives you the ability to code specific ranges (for numeric checks) and even fallback default values should a particular user input not pass validation.

For example, say you are building a shopping cart that allows the user to specify the number of items they want to purchase. Clearly, this must be a value greater than zero and less than the total inventory you have available. An approach like that illustrated in Example 9-5 will force the value to be an integer between certain bounds or the value will fall back to 1. In this way, a user cannot accidentally order more items than you have, a negative number of items, a partial number of items, or some non-numeric amount.

Example 9-5. Validating an integer value with bounds and a default

```
function sanitizeQuantity(mixed $orderSize): int
{
    return filter_var(
        $orderSize,
        FILTER_VALIDATE_INT,
        [
            'options' => [
                'min_range' => 1,
                'max_range' => 25,
                'default'   => 1,
            ]
        ]
    );
}

echo sanitizeQuantity(12) . PHP_EOL;
                          ❶

echo sanitizeQuantity(-5) . PHP_EOL;
                          ❷
```

```
echo sanitizeQuantity(100) . PHP_EOL;
❸

echo sanitizeQuantity('banana') . PHP_EOL;
❹
```

❶ The quantity checks out and returns `12`.

❷ Negative integers fail to validate, so this returns the default of `1`.

❸ The input is above the max range, so this returns the default of `1`.

❹ Non-numeric inputs will always return the default of `1`.

See Also

Documentation on PHP's data filtering extension.

9.2 Keeping Sensitive Credentials Out of Application Code

Problem

Your application needs to leverage a password or API key, and you want to avoid having that sensitive credential written in code or committed to version control.

Solution

Store the credential in an environment variable exposed by the server running the application. Then reference that environment variable in code. For example:

```
$db = new PDO($database_connection, getenv('DB_USER'),
```

Discussion

A common mistake made in many developers' early careers is to hardcode credentials for sensitive systems into constants or other places in application

code. While this makes those credentials readily available to application logic, it also introduces severe risk to your application.

You could accidentally use production credentials from a development account. An attacker might find credentials indexed in an accidentally public repository. An employee might abuse their knowledge of credentials beyond their intended use.

In production, the best credentials are those unknown to and untouched by humans. It's a good idea to keep those credentials only in the production environment and use *separate accounts* for development and testing. Leveraging environment variables within your code makes your application flexible enough to run anywhere, as it uses not hardcoded credentials but those bound to the environment itself.

WARNING

PHP's built-in information system, `phpinfo()`, will automatically enumerate all environment variables for debugging purposes. Once you begin leveraging the system environment to house sensitive credentials, take extra care to avoid using detailed diagnostic tools like `phpinfo()` in publicly accessible parts of your application!

The method of *populating* environment variables will differ from one system to another. In Apache-powered systems, you can set environment variables by using the `SetEnv` keyword within the `<VirtualHost>` directive as follows:

```
<VirtualHost myhost>
...
SetEnv DB_USER "database"
SetEnv DB_PASS "password1234"
...
</VirtualHost>
```

In NGINX-powered systems, you can set environment variables for PHP only if it's running as a FastCGI process. Similar to Apache's `SetEnv`, this is done with a keyword within a `location` directive in the NGINX configuration as follows:

```
location / {
    ...
    fastcgi_param DB_USER database
    fastcgi_param DB_PASS password1234
    ...
}
```

Separately, Docker-powered systems set environment variables in either their Compose files (for Docker Swarm) or the system deployment configuration (for Kubernetes). In all of these situations, you are defining a credential within the environment itself rather than within your application.

An additional option is to use PHP dotenv.[4] This third-party package allows you to define your environment configuration in a flat file named *.env* and it automatically populates both environment variables and the `$_SERVER` superglobal. The biggest advantage of this approach is that dotfiles (files prefixed with a `.`) are easy to exclude from version control and are typically hidden on a server to begin with. You can use a *.env* locally to define development credentials and keep a separate *.env* on the server to define production credentials.

In both cases, you never have to directly manage an Apache or NGINX configuration file at all!

A *.env* file defining the database credentials used in the Solution example would look something like the following:

```
DB_USER=database
DB_PASS=password1234
```

In your application code, you would then load the library dependency and invoke its loader as follows:

```
$dotenv = Dotenv\Dotenv::createImmutable(__DIR__);
$dotenv->load();
```

Once the library is loaded, you can leverage `getenv()` to reference the environment variables wherever you need access.

See Also

Documentation on `getenv()`.

9.3 Hashing and Validating Passwords

Problem

You want to authenticate users leveraging passwords only they know and prevent your application from storing sensitive data.

Solution

Use `password_hash()` to store secure hashes of passwords:

```
$hash = password_hash($password, PASSWORD_DEFAULT);
```

Use `password_verify()` to verify that a plaintext password produces a given, stored hash as follows:

```
if (password_verify($password, $hash)) {
    // Create a valid session for the user ...
}
```

Discussion

Storing passwords in plaintext is always a bad idea. If your application or data store is ever breached, those plaintext passwords can and will be abused by attackers. To keep your users safe and protect them from potential abuse in the case of a breach, you must always hash those passwords when storing them in your database.

Conveniently, PHP ships with a native function to do just that—`password_hash()`. This function takes a plaintext password and automatically generates a deterministic but seemingly random hash from that data. Rather than storing the plaintext password, you store this hash. Later, when the user chooses to log into your application, you can compare the

plaintext password against the stored hash (using a safe comparison function like `hash_equals()`) and assert whether they match.

PHP generally supports three hashing algorithms, enumerated in Table 9-4. At the time of this writing, the default algorithm is bcrypt (which is based on the Blowfish cipher), but you can choose a particular algorithm at runtime by passing a second parameter into `password_hash()`.

Table 9-4. Password hashing algorithms

Constant	Description
`PASSWORD_DEFAULT`	Use the default bcrypt algorithm. The default algorithm could change in a future release.
`PASSWORD_BCRYPT`	Use the `CRYPT_BLOWFISH` algorithm.
`PASSWORD_ARGON2I`	Use the Argon2i hashing algorithm. (Only available if PHP has been compiled with Argon2 support.)
`PASSWORD_ARGON2ID`	Use the Argon2id hashing algorithm. (Only available if PHP has been compiled with Argon2 support.)

Each hashing algorithm supports a set of options that can determine the difficulty of calculating a hash on the server. The default (or bcrypt) algorithm supports an integer "cost" factor—the higher the number, the more computationally expensive the operation will be. The Argon2 family of algorithms supports two cost factors—one for the memory cost and one for the amount of time it will take to compute a hash.

NOTE

Increasing the cost for computing a hash is a means of protecting the application from brute-force authentication attacks. If it takes 1 second to calculate a hash, it will take *at least* 1 second for a legitimate party to authenticate (this is trivial). However, an attacker can attempt to authenticate *at most* once per second. This renders brute-force attacks relatively costly both in terms of time and computing capacity.

When you first build your application, it is a good idea to test the server environment that will run it and set your cost factors appropriately. Identifying cost factors requires testing the performance of `password_hash()` on the live environment, as demonstrated in Example 9-6. This script will test the hashing performance of the system with increasingly large cost factors and identify a cost factor that achieves the desired time target.

Example 9-6. Testing the cost factors for `password_hash()`

```
$timeTarget = 0.5; // 500 milliseconds

$cost = 8;
do {
    $cost++;
    $start = microtime(true);
    password_hash('test', PASSWORD_BCRYPT, ['cost' => $
    $end = microtime(true);
} while(($end - $start) < $timeTarget);

echo "Appropriate cost factor: {$cost}" . PHP_EOL;
```

The output of `password_hash()` is intended to be fully forward compatible. Rather than merely generate a hash, the function will also internally generate a salt to make the hash unique. Then the function returns a string representing the following:

- The algorithm used
- The cost factors or options
- The generated random salt
- The resulting hash

Figure 9-1 shows an example of this string output.

NOTE

A unique, random salt is generated internally by PHP every time you call `password_hash()`. This has the effect of producing distinct hashes for identical plaintext passwords. In this way, hashed values can't be used to identify which accounts are using the same passwords.

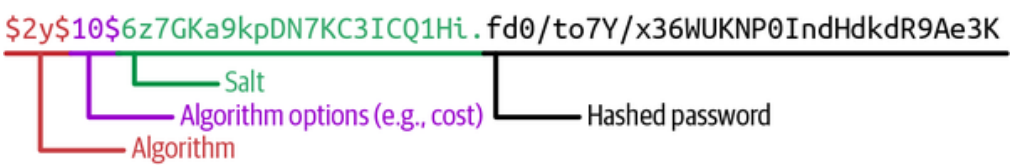

Figure 9-1. Example output of `password_hash()`

The advantage of encoding all of this information into the output of `password_hash()` is that you don't need to maintain this data in the application. At a future date, you might change the hashing algorithm or modify the cost factors used for hashing. By encoding the settings originally used to generate a hash, PHP can reliably re-create a hash for comparison.

When a user logs in, they provide only their plaintext password. The application needs to recompute a hash of this password and compare the newly computed value with the one stored in the database. Given that the information you need to calculate a hash is stored alongside the hash, this becomes relatively straightforward.

Rather than implement the comparison yourself, though, you can leverage PHP's `password_verify()` function that does all this in a safe and secure way.

See Also

Documentation on `password_hash()` and `password_verify()`.

9.4 Encrypting and Decrypting Data

Problem

You want to protect sensitive data leveraging encryption and reliably decrypt that information at a later time.

Solution

Use sodium's `sodium_crypto_secretbox()` to encrypt the data with a known symmetric (aka shared) key, as shown in Example 9-7.

Example 9-7. Sodium symmetric encryption example

```
$key = hex2bin('faae9fa60060e32b3bbe5861c2ff290f' .
               '2cd4008409aeb7c59cb3bad8a8e89512');
                                                    ❶


$message = 'Look to my coming on the first light of ' .
           'the fifth day, at dawn look to the east.';

$nonce = random_bytes(SODIUM_CRYPTO_SECRETBOX_NONCEBYTE
                                                    ❷


$ciphertext = sodium_crypto_secretbox($message, $nonce,
                                                    ❸

$output = bin2hex($nonce . $ciphertext);
                                                    ❹
```

❶ The key must be a random value of length `SODIUM_CRYPTO_SECRETBOX_KEYBYTES` (32 bytes). If you are creating a new, random string, you can leverage `sodium_crypto_secretbox_keygen()` to create one. Just be sure to store it somewhere so you can decrypt the message later.

❷ Every encryption operation should use a unique, random nonce. PHP's `random_bytes()` can reliably create one of the appropriate length by using built-in constants.

❸ The encryption operation leverages both the random nonce and the fixed secret key to protect the message and returns raw bytes as a result.

❹ Often you want to exchange encrypted information over a network protocol, so encoding the raw bytes as hexadecimal can make it more portable. You'll also need the nonce to decrypt the exchanged data, so you store it alongside the ciphertext.

When you need to decrypt the data, use `sodium_crypto_secretbox_open()` to both extract and validate the data, as shown in Example 9-8.

Example 9-8. Sodium symmetric decryption

```
$key = hex2bin('faae9fa60060e32b3bbe5861c2ff290f' .
               '2cd4008409aeb7c59cb3bad8a8e89512');
                                                    ❶

$encrypted = '8b9225c935592a5e95a9204add5d09db' .
             'b7b6473a0aa59c107b65f7d5961b720e' .
             '7fc285bd94de531e05497143aee854e2' .
             '918ba941140b70c324efb27c86313806' .
             'e04f8e79da037df9e7cb24aa4bc0550c' .
             'd7b2723cbb560088f972a408ffc973a6' .
             '2be668e1ba1313e555ef4a95f0c1abd6' .
             'f3d73921fafdd372';
                                                    ❷

$raw = hex2bin($encrypted);

$nonce = substr($raw, 0, SODIUM_CRYPTO_SECRETBOX_NONCEE
                                                    ❸

$ciphertext = substr($raw, SODIUM_CRYPTO_SECRETBOX_NONC

$plaintext = sodium_crypto_secretbox_open($ciphertext,
if ($plaintext === false) {
                                                    ❹

    echo 'Error decrypting message!' . PHP_EOL;
} else {
    echo $plaintext . PHP_EOL;
}
```

❶ Use the same key for decryption that you used for encryption.

❷ The resulting ciphertext from Example 9-7 is reused here and is extracted from a hexadecimal-encoded string.

❸ Because you concatenated the ciphertext and nonce, you must split the two components apart for use with `sodium_crypto_secretbox_open()`.

❹ This entire encryption/decryption operation is authenticated. If anything changed in the underlying data, the authentication step will fail and the function will return a literal `false` to flag the manipulation. If it returns anything else, the decryption succeeded and you can trust the output!

Discussion

The `secretbox` family of functions exposed by sodium implements authenticated encryption/decryption by using a fixed symmetric key. Every time you encrypt a message, you should do so with a random nonce to fully protect the privacy of the encrypted message.

WARNING

The nonce used in encryption itself isn't a secret or sensitive value. However, you should take care never to reuse a nonce with the same symmetric encryption key. Nonces are "numbers used once" and are intended to add randomness to the encryption algorithm such that the same value encrypted twice with the same key can produce different ciphertexts. Reusing a nonce with a specific key compromises the security of the data you're aiming to protect.

The symmetry here is that the same key is used both to encrypt and decrypt a message. This is most valuable when the same system is responsible for both operations—for example, when a PHP application needs to encrypt data to be stored in a database and then decrypt that data when reading it back out.

The encryption leverages the XSalsa20 stream cipher to protect data. This cipher uses a 32-byte (256-bit) key and a 24-byte (192-bit) nonce. None of this information is necessary for developers to keep track of, though, as it's safely abstracted behind the `secretbox` functions and constants. Rather than keep track of key sizes and encryption modes, you merely need to create or *open* a box with the matched key and nonce for the message.

Another advantage of this approach is authentication. Every encryption operation will also generate a message authentication tag leveraging the Poly1305 algorithm. Upon decryption, sodium will validate that the authentication tag matches the protected data. If there is not a match, it's possible that the message was either accidentally corrupted or intentionally manipulated. In either case, the ciphertext is unreliable (as would be any decrypted plaintext), and the open function will return a Boolean `false` .

Asymmetric encryption

Symmetric encryption is easiest when the same party is both encrypting and decrypting data. In many modern tech environments, these parties might be

independent and communicating over less-than-trustworthy media. This necessitates a different form of encryption, as the two parties cannot directly share a symmetric encryption key. Instead, each can create pairs of public and private keys and leverage key exchange to agree upon an encryption key.

Key exchange is a complicated topic. Luckily, sodium exposes simple interfaces for performing the operation in PHP. In the examples that follow, two parties will do the following:

1. Create public/private key pairs
2. Exchange their public keys directly
3. Use these asymmetric keys to agree upon a symmetric key
4. Exchange encrypted data

Example 9-9 illustrates how each party will create their own public/private key pairs. Though the code example is in a single block, each party would create their keys independently and only ever exchange their public keys with one another.

Example 9-9. Asymmetric key creation

```
$aliceKeypair = sodium_crypto_box_keypair();
$alicePublic = sodium_crypto_box_publickey($aliceKeypai
                                           ❶

$alicePrivate = sodium_crypto_box_secretkey($aliceKeypa
                                           ❷


$bethKeypair = sodium_crypto_box_keypair();
                                           ❸

$bethPublic = sodium_crypto_box_publickey($bethKeypair)
$bethPrivate = sodium_crypto_box_secretkey($bethKeypair
```

❶ Alice creates a key pair and extracts her public and private key from it separately. She shares her public key with Beth.

❷ Alice keeps her private key secret. This is the key she will use to encrypt data *for* Beth and decrypt data *from* her. Similarly, Alice would use her private key to encrypt data for anyone else who has shared their public key with her.

❸

Beth independently does the same thing as Alice and shares her own public key.

Once Alice and Beth have shared their public keys, they are free to communicate privately. The `cryptobox` family of functions in sodium leverages these asymmetric keys to compute a symmetric key that can be used for confidential communication. This symmetric key is not exposed directly to either party, but allows the parties to easily communicate with one another.

Note that anything Alice encrypts for Beth can *only* be decrypted by Beth. Even Alice cannot decrypt these messages, as she does not have Beth's private key! Example 9-10 illustrates how Alice can encrypt a simple message leveraging both her own private key and Beth's advertised public key.

Example 9-10. Asymmetric encryption

```
$message = 'Follow the white rabbit';
$nonce = random_bytes(SODIUM_CRYPTO_BOX_NONCEBYTES);
$encryptionKey = sodium_crypto_box_keypair_from_secretk
    $alicePrivate,
    $bethPublic
);
                                    ❶


$ciphertext = sodium_crypto_box($message, $nonce, $encr
                                    ❷


$toBeth = bin2hex($nonce . $ciphertext);
                                    ❸
```

❶ The encryption key used here is actually a key pair composed of information from both Alice's and Beth's individual key pairs.

❷ As with symmetric encryption, you leverage a nonce in order to introduce randomness into the encrypted output.

❸ You concatenate the random nonce and the encrypted ciphertext so Alice can send both to Beth at the same time.

The key pairs involved are points on an elliptic curve, specifically Curve25519. The initial operation sodium uses for asymmetric encryption is a *key exchange* between two of these points to define a fixed but secret number.

This operation uses the X25519 key exchange algorithm and produces a number based on Alice's private key and Beth's public key.

That number is then used as a key with the XSalsa20 stream cipher (the same one used for symmetric encryption) to encrypt the message. As with symmetric encryption discussed for the `secret box` family of functions, a `cryptobox` will leverage a Poly1305 message authentication tag to protect the message against tampering or corruption.

At this point, Beth, by leveraging her own private key, Alice's public key, and the message nonce, can reproduce all of these steps on her own to decrypt the message. She performs a similar X25519 key exchange to derive the same shared key and then uses that for decryption.

Thankfully, sodium abstracts the key exchange and derivation for us, making asymmetric decryption relatively straightforward. See Example 9-11.

Example 9-11. Asymmetric decryption

```
$fromAlice = hex2bin($toBeth);
$nonce = substr($fromAlice, 0, SODIUM_CRYPTO_BOX_NONCEE
                            ❶

$ciphertext = substr($fromAlice, SODIUM_CRYPTO_BOX_NONC

$decryptionKey = sodium_crypto_box_keypair_from_secretk
    $bethPrivate,
    $alicePublic
);
                            ❷


$decrypted = sodium_crypto_box_open($ciphertext, $nonce
                            ❸

if ($decrypted === false) {
                            ❹

    echo 'Error decrypting message!' . PHP_EOL;
} else {
    echo $decrypted . PHP_EOL;
}
```

❶

Similar to the `secretbox` operation, you first extract the nonce and ciphertext from the hexadecimal-encoded payload provided by Alice.

❷ Beth uses her own private key along with Alice's public key to create a key pair appropriate for decrypting the message.

❸ The key exchange and decryption operations are abstracted away with a simple "open" interface to read the message.

❹ As with symmetric encryption, sodium's asymmetric interfaces will validate the Poly1305 authentication tag on the message before decrypting and returning the plaintext.

Either mechanism—symmetric or asymmetric encryption—is well supported and cleanly abstracted by sodium's functional interfaces. This helps avoid errors commonly encountered when implementing older mechanisms (like AES or RSA). Libsodium (the C-level library powering the sodium extension) is widely supported in other languages as well, providing solid interoperability between PHP, Ruby, Python, JavaScript, and even lower-level languages like C and Go.

See Also

Documentation on `sodium_crypto_secretbox()`, `sodium_crypto_secretbox_open()`, `sodium_crypto_box()`, and `sodium_crypto_box_open()`.

9.5 Storing Encrypted Data in a File

Problem

You want to encrypt (or decrypt) a file that is too large to fit in memory.

Solution

Use the *push* streaming interfaces exposed by sodium to encrypt one chunk of the file at a time, as shown in Example 9-12.

Example 9-12. Sodium stream encryption

```
define('CHUNK_SIZE', 4096);

$key = hex2bin('67794ec75c56ba386f944634203d4e86' .
               '37e43c97857e3fa482bb9dfec1e44e70');

[$state, $header] = sodium_crypto_secretstream_xchacha2
                                    ❶


$input = fopen('plaintext.txt', 'rb');
                                    ❷

$output = fopen('encrypted.txt', 'wb');

fwrite($output, $header);
                                    ❸


$fileSize = fstat($input)['size'];
                                    ❹


for ($i = 0; $i < $fileSize; $i += (CHUNK_SIZE - 17)) {
                                    ❺

    $plain = fread($input, (CHUNK_SIZE - 17));
    $cipher = sodium_crypto_secretstream_xchacha20poly1
                                    ❻


    fwrite($output, $cipher);
}

sodium_memzero($state);
                                    ❼


fclose($input);
                                    ❽

fclose($output);
```

❶ Initializing the stream operation yields two values—a header and the current state of the stream. The header itself contains a random nonce and is required in order to decrypt anything encrypted with the stream.

❷

Open both the input and output files as binary streams. Working with files is one of the few times you want to emit raw bytes rather than leverage hexadecimal or Base64 encoding to wrap an encrypted output.

To ensure that you can decrypt the file, first store the fixed-length header for later retrieval. ❸

Before you begin iterating over chunks of bytes in the file, you need to determine how large the input file really is. ❹

As with other sodium operations, the stream encryption cipher is built atop Poly1305 authentication tags (which are 17 bytes in length). As a result, you read 17 bytes less than the standard 4,096-byte block, so the output will be a total of 4,081 bytes written to the file. ❺

The sodium API encrypts the plaintext and automatically updates the state variable (`$state` is passed by reference and updated in place). ❻

After you're done with encryption, explicitly zero out the memory of the state variable. PHP's garbage collector will clean up the references, but you want to ensure that no coding errors elsewhere in the system can inadvertently leak this value. ❼

Finally, close the file handles since the encryption is complete. ❽

To decrypt the file, use sodium's *pull* streaming interfaces, as shown in Example 9-13.

Example 9-13. Sodium stream decryption

```
define('CHUNK_SIZE', 4096);

$key = hex2bin('67794ec75c56ba386f944634203d4e86' .
               '37e43c97857e3fa482bb9dfec1e44e70');

$input = fopen('encrypted.txt', 'rb');
$output = fopen('decrypted.txt', 'wb');

$header = fread($input, SODIUM_CRYPTO_SECRETSTREAM_XCHA
                                                    ❶


$state = sodium_crypto_secretstream_xchacha20poly1305_i
                                                    ❷
```

```
$fileSize = fstat($input)['size'];
try {
    for (
        $i = SODIUM_CRYPTO_SECRETSTREAM_XCHACHA20POLY13
        $i < $fileSize;
        $i += CHUNK_SIZE
    ) {
                            ❸

        $cipher = fread($input, CHUNK_SIZE);

        [$plain, ] = sodium_crypto_secretstream_xchacha
            $state,
            $cipher
        );
                            ❹


        if ($plain === false) {
                            ❺

            throw new Exception('Error decrypting file!
        }
        fwrite($output, $plain);
    }
} finally {
    sodium_memzero($state);
                            ❻


    fclose($input);
    fclose($output);
}
```

❶ Before you can begin decryption, you must explicitly retrieve the header value from the file you encrypted in the first place.

❷ Armed with the encryption header and key, you can initialize the stream state as desired.

❸ Keep in mind that the file is prefixed with a header, so you skip those bytes and then pull chunks of 4,096 bytes at a time. This will include 4,079 bytes of ciphertext and 17 bytes of authentication tag.

❹ The actual stream decryption operation returns a tuple of the plaintext and an optional status tag (e.g., to identify that a key needs to be rotated).

If the authentication tag on the encrypted message fails to validate

If the authentication tag on the encrypted message fails to validate, however, this function will return `false` to indicate the authentication failure. Should this occur, halt decryption immediately.

Again, once the operation is complete, you should zero out the memory storing the stream state, as well as close out the file handles.

Discussion

The stream cipher interfaces exposed by sodium are not actual streams to PHP.[5] Specifically, they are stream *ciphers* that work like block ciphers with an internal counter. XChaCha20 is the cipher used by the `sodium_crypto_secretstream_xchacha20poly1305_*()` family of functions leveraged to push data into and pull data from an encrypted stream in this recipe. The implementation in PHP explicitly breaks a long message (a file) into a series of related messages. Each of these messages is encrypted with the underlying cipher and tagged individually but in a specific order.

These messages cannot be truncated, removed, reordered, or manipulated in any way without the decryption operation detecting that tampering. There is also no practical limit to the total number of messages that can be encrypted as part of this stream, which means there is no limit to the size of a file that can be passed through the Solution examples.

The Solution uses a chunk size of 4,096 bytes (4 KB), but other examples could use 1,024, 8,096, or any other number of bytes. The only limitation here is the amount of memory available to PHP—iterating over smaller chunks of a file will use less memory during encryption and decryption. Example 9-12 illustrates how `sodium_crypto_secretstream_xchacha20poly1305_push()` encrypts one chunk of data at a time, "pushing" that data through the encryption algorithm and updating the algorithm's internal state. The paired `sodium_crypto_secretstream_xchacha20poly1305_pull()` does the same thing in reverse, pulling corresponding plaintext back out of the stream and updating the algorithm's state.

Another way to see this in action is with the low-level primitive `sodium_crypto_stream_xchacha20_xor()` function. This function leverages the XChaCha20 encryption algorithm directly to generate a stream of seemingly random bytes based on a given key and random nonce. It then performs an XOR operation between that stream of bytes and a given message to produce

a ciphertext.[6] Example 9-14 illustrates one way in which this function might be used—encrypting phone numbers in a database.

Example 9-14. Simple stream encryption for data protection

```
function savePhoneNumber(int $userId, string $phone): v
{
    $db = getDatabase();

    $statement = $db->prepare(
        'INSERT INTO phones (user, number, nonce) VALUE
    );

    $key = hex2bin(getenv('ENCRYPTION_KEY'));
    $nonce = random_bytes(SODIUM_CRYPTO_STREAM_XCHACHA2

    $encrypted = sodium_crypto_stream_xchacha20_xor($ph

    $statement->execute([$userId, bin2hex($encrypted),
}
```

The advantage of using a cryptographic stream in this way is that the ciphertext exactly matches the length of the plaintext. However, this also means there is no authentication tag available (meaning the ciphertext could be corrupted or manipulated by a third party and jeopardize the reliability of any decrypted ciphertext).

As a result, Example 9-14 is not likely something you will ever use directly. However, it does illustrate how the more verbose `sodium_crypto_secretstream_xchacha20poly1305_push()` works under the hood. Both functions use the same algorithm, but the "secret stream" variant generates its own nonce and keeps track of its internal state over repeated uses (in order to encrypt *multiple* chunks of data). When using the simpler XOR version you would need to manage that state and repeated calls manually!

See Also

Documentation on `sodium_crypto_secretstream_xchacha20poly1305_init_push()`, `sodium_crypto_secretstream_xchacha20poly1305_init_pull()`, and `sodium_crypto_stream_xchacha20_xor()`.

9.6 Cryptographically Signing a Message to Be Sent to Another Application

Problem

You want to sign a message or piece of data before sending it to another application such that the other application can validate your signature on the data.

Solution

Use `sodium_crypto_sign()` to attach a cryptographic signature to a plaintext message, as shown in Example 9-15.

Example 9-15. Cryptographic signatures on messages

```
$signSeed = hex2bin('eb656c282f46b45a814fcc887977675d'
                    'c627a5b1507ae2a68faecee147b77621');

$signKeys = sodium_crypto_sign_seed_keypair($signSeed);

$signSecret = sodium_crypto_sign_secretkey($signKeys);
$signPublic = sodium_crypto_sign_publickey($signKeys);

$message = 'Hello world!';
$signed = sodium_crypto_sign($message, $signSecret);
```

❶ In practice, your signing seed should be a random value kept secret by you. It could also be a secure hash derived from a known password.

Discussion

Cryptographic signatures are a way to verify that a particular message (or string of data) originated from a given source. So long as the private key used to sign a message is kept secret, anyone with access to the publicly known key can validate that the information came from the owner of the key.

Likewise, only the custodian of that key can sign the data. This helps verify that the custodian signed off on the message. It also forms the basis for nonrepudiation: the owner of the key cannot claim that someone else used their key without rendering their key (and any signatures it created) invalid.

In the Solution example, a signature is calculated based on the secret key and the contents of the message. The bytes of the signature are then prepended to the message itself, and both elements together are passed along to any party who wishes to verify the signature.

It is also possible to generate a *detached* signature, effectively producing the raw bytes of the signature without concatenating it to the message. This is useful if the message and signature are meant to be sent independently to the third-party verifier—for example, as different elements in an API request.

NOTE

While raw bytes are fantastic for information stored on disk or in a database, they can cause problems with remote APIs. It would make sense to Base64-encode the entire payload (the signature and message) when sending it to a remote party. Otherwise, you might want to encode the signature separately (e.g., as hexadecimal) when sending the two components together.

Rather than using `sodium_crypto_sign()` as in the Solution example, you can use `sodium_crypto_sign_detached()` as in Example 9-16.

Example 9-16. Creating a detached message signature

```
$signSeed = hex2bin('eb656c282f46b45a814fcc887977675d'
                    'c627a5b1507ae2a68faecee147b77621');
$signKeys = sodium_crypto_sign_seed_keypair($signSeed);

$signSecret = sodium_crypto_sign_secretkey($signKeys);
$signPublic = sodium_crypto_sign_publickey($signKeys);

$message = 'Hello world!';
$signature = sodium_crypto_sign_detached($message, $sig
```

Signatures will always be 64 bytes long, regardless of whether they're attached to the plaintext they sign.

See Also

Documentation on `sodium_crypto_sign()` .

9.7 Verifying a Cryptographic Signature

Problem

You want to verify the signature on a piece of data sent to you by a third party.

Solution

Use `sodium_crypto_sign_open()` to validate the signature on a message, as shown in Example 9-17.

Example 9-17. Cryptographic signature verification

```
$signPublic = hex2bin('d58c47ddb986dcb2632aa5395e8962d3
                      'e636ee236b38a8dc880e409c19374a5f

$message = sodium_crypto_sign_open($signed, $signPublic
                                   ❶


if ($message === false) {
                                   ❷
    throw new Exception('Invalid signature on message!'
}
```

❶ The data in `$signed` is a concatenated raw signature and plaintext message, as in the return of `sodium_crypto_sign()` .

❷ If the signature is invalid, the function returns `false` as an error. If the signature is valid, the function instead returns the plaintext message.

Discussion

Signature verification is straightforward when the literal returns from `sodium_crypto_sign()` are involved. Merely pass the data and the public

key of the signing party into `sodium_crypto_sign_open()`, and you will have either a Boolean error or the original plaintext back as a result.

If you're working with a web API, there's a good chance that the message and signature were passed to you separately (e.g., if someone were using `sodium_crypto_sign_detached()`). In that case, you need to concatenate the signature and the message together before passing them into `sodium_crypto_sign_open()`, as in Example 9-18.

Example 9-18. Detached signature verification

```
$signPublic = hex2bin('d58c47ddb986dcb2632aa5395e8962d3
                      'e636ee236b38a8dc880e409c19374a5f

$signature = hex2bin($_POST['signature']);
$payload = $signature . $_POST['message'];

$message = sodium_crypto_sign_open($payload, $signPubli

if ($message === false) {
    throw new Exception('Invalid signature on message!'
}
```

See Also

Documentation on `sodium_crypto_sign_open()`.

[1] The OpenJDK Psychic Signatures bug of 2022 illustrated how an error in cryptographic authentication could expose not just applications *but an entire language implementation* to potential abuse by malicious actors. This bug was due to an implementation error, further underscoring how critical it is to rely on solid, proven, well-tested primitives when using cryptographic systems.

[2] For more on native extensions, see Chapter 15.

[3] For a full breakdown of the process through which this extension was added to PHP core, reference the original RFC.

[4] Loading PHP packages via Composer is discussed at length in Recipe 15.3.

[5]

PHP streams, covered extensively in Chapter 11, expose effective ways to work with large chunks of data without exhausting available system memory.

[6] For more on operators and XOR in particular, review Chapter 2.