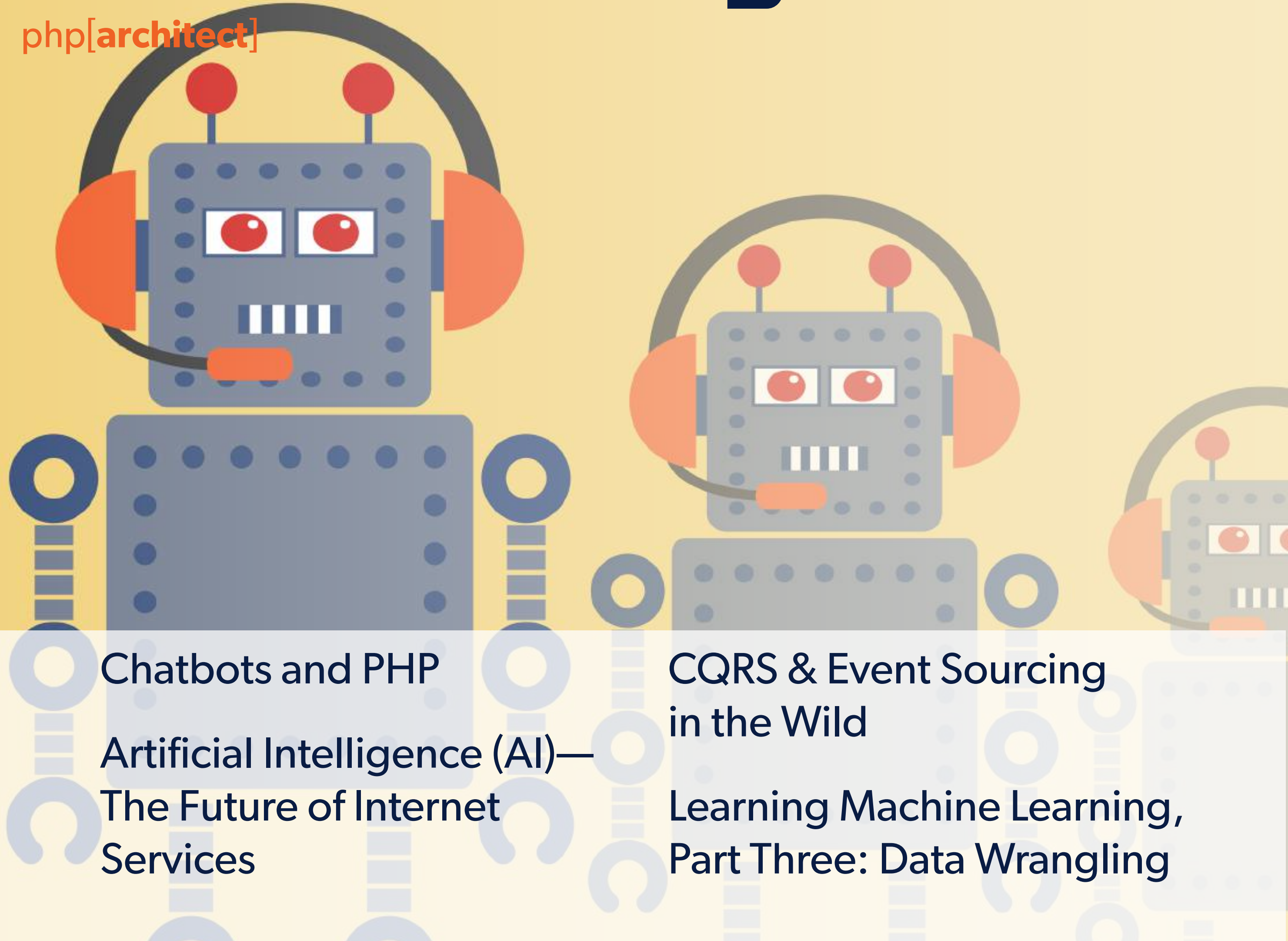
www.phparch.com

December 2017
VOLUME 16 - Issue 12

php[architect]

Talking Code



Free Sample Article

nexcess™

beyond hosting.

We’re hiring
PHP developers

15 years of experience with

PHP Application Hosting

SUPPORT FOR *php7* SINCE DAY ONE

Contact **careers@nexcess.net** for more information.

www.nexcess.net | US +1-866-639-2377 / UK +0-808-120-7609 / AU 1-800-765-472

PHP[TEK] 2018

The Premier PHP Conference

13th Annual Edition

MAY 31ST – JUNE 1ST

Downtown Atlanta

tek.phparch.com

PHP, meet Libsodium

Eric Mann

By the time you read this, the PHP community should have introduced the world to the newest version of our favorite language. This latest version adds better support for type annotations, allows trailing commas in lists (just like JavaScript and other dynamic languages) and introduced several security improvements. The most notable security addition, however, is the introduction of the Sodium cryptographic library as a core extension.

Introducing Sodium

Sodium[1] is a cryptographic library which supports high-level abstractions for encryption, decryption, signing, password hashing, and more. It is a fork of an earlier project, NaCl[2], Networking and Cryptography library.

The aim of both projects is to provide an easy-to-use, high-speed tool for programmers to work with encryption safely and with which they can build even higher-level tools for end users.

Authenticated Encryption

Unlike many other cryptographic libraries, Sodium focuses on *authenticated* encryption schemes. This means every piece of encrypted data automatically carries a message authentication code (MAC) which can validate the integrity of the data itself. If the MAC is found to be invalid, Sodium will immediately error.

Using a MAC to validate an encrypted message isn't itself a unique trait. However, many other libraries will leave it to the end developer to implement MAC validation. Sodium builds this primitive into the library itself to better enforce best practices with message integrity.

Elliptic Curves

One of the ways Sodium truly shines is public key cryptography. In this paradigm, every user has a pair of keys—one key is kept secret while the other is shared with the world. Anyone can encrypt a message for a particular user with their public key; it can only be read with their private key. Likewise, a user can sign a piece of data with their private key; a third party can use the already-distributed public key to verify the signature.

Many people are familiar with RSA[3], which is an older style of public key cryptography which uses large prime numbers, exponentiation, and modulo arithmetic to build security. The keys involved need to be rather large to guarantee privacy; the National Institute of Standards and Technology recommends RSA keys[4] of at least 3072 bits.

The strength of an RSA private key is tied both to its length and to the computing power available to an attacker. Breaking RSA generally requires guessing to break the factorization of an encrypted message; a longer key requires more computing power to decrypt and thus makes each "guess" from an attacker somewhat costly. As computers gain in speed and overall performance, keys once thought to be secure become weak.

Sodium uses a different kind of mathematics for cryptography. Rather than leveraging prime numbers and factorization, Sodium uses mathematical calculations over a discrete field defined by an elliptic curve[5]. The math itself is a bit more complex but yields a similar public/private key relationship to traditional RSA. Due to the math involved, however, a 256-bit elliptic key is as strong as a 3072-bit RSA key.

Legacy Support

While Sodium is supported natively as of PHP 7.2[6], some projects might want to leverage the same cryptographic interfaces on platforms running older builds of PHP. Thankfully, it's fully possible thanks to two projects.

Before Sodium was in PHP natively, it was available as a PECL extension[7]. Anyone running at least PHP 7.0 can install the PECL module and will have the same level of functionality and support as those using the native builds in 7.2. Both the PECL module and the core PHP extension are written by

1 *Sodium:* *https://download.libsodium.org/doc/*
2 *NaCl:* *http://nacl.cr.yp.to*
3 *RSA:* *http://phpa.me/wikip-rsa-crypto*
4 *RSA keys:* *https://www.keylength.com/en/4/*
5 *defined by an elliptic curve:* *http://phpa.me/wikip-elliptic-crypto*
6 *as of PHP 7.2:* *https://wiki.php.net/rfc/libsodium*
7 *PECL extension:* *https://pecl.php.net/package/libsodium*

the same authors, so there is zero trade-off on older environments.

Some developers have yet to update to PHP7, though. For those teams, the `sodium_compat`[8] module by Paragon Initiative[9] is the way forward. This module implements Sodium in vanilla PHP if there isn't a native extension available to expose the API. It's significantly slower to encrypt and decrypt this way but means older servers can still leverage Sodium even without a binary distribution.

In fact, `sodium_compat` is a solid approach for anyone working with Sodium who wants to maintain backward compatibility with PHP. The module will attempt to use the native PHP 7.2 features if they're available. It will automatically look for the PECL extension on older systems and use it if it's supported. Finally, on older systems with no PECL module for Sodium, the polyfill will load a vanilla PHP implementation of the cryptographic primitives. Using `sodium_compat` means you can write your code once then defer to the library to pick the best implementation for you.

8 `sodium_compat`: *https://github.com/paragonie/sodium_compat*

9 *Paragon Initiative: https://paragonie.com*

Secret-Key Crypto

Libsodium makes *symmetric* key cryptography (where a single encryption/decryption key is shared by both parties involved) simple with the `sodium_crypto_secretbox()` and `sodium_crypto_secretbox_open()` functions. The first function is used to *encrypt* a string message given a random nonce and a specific symmetric key. See Listing 1.

The key is a shared secret; our nonce needs to be unique for every encryption operation but does not need to be kept secret so long as it continues to be randomly generated each time. Decrypting our message is just like encryption, only in reverse as in Listing 2.

Sodium uses authenticated encryption for every transaction. The message is both encrypted and affixed with a message authentication code (MAC) to verify the message hasn't been tampered with. When decrypting the message, Sodium will verify no one has tampered with the message and automatically error if it's been changed.

Listing 1

```
// Create a random nonce
$nonce = random_bytes(SODIUM_CRYPTO_SECRETBOX_NONCEBYTES);

// Create a random key
$key = random_bytes(SODIUM_CRYPTO_SECRETBOX_KEYBYTES);

// Our plaintext message
$message = 'This is a super secret communication!';

// Encrypted
$ciphertext = bin2hex(
    sodium_crypto_secretbox($message, $nonce, $key)
);
```

Listing 2

```
<?php
// Our known symmetric key
$key = '...';

// The original message nonce
$nonce = '...';

// Decrypt our message
$plaintext = sodium_crypto_secretbox_open(
    hex2bin($ciphertext), $nonce, $key
);
```

Listing 3

```
// Create our public/private keypair
$keypair = sodium_crypto_box_keypair();
$private_key = substr($keypair, 0, 32);

// Create a random nonce
$nonce = random_bytes(SODIUM_CRYPTO_SECRETBOX_NONCEBYTES);

// Our plaintext message
$message = 'This is a super secret communication!';

// Use the recipient's known public key
$public_key = '...';


// Encrypt our message for a specific recipient's public key
$ciphertext = bin2hex(
    sodium_crypto_box(
        $message, $nonce, $private_key . $public_key
    )
);
```

Listing 4

```
// The sender's public key
$public_key = '...';

// Our known private key
$private_key = '...';

// The original message nonce
$nonce = '...';

// Decrypt our message
$plaintext = sodium_crypto_box_open(
    hex2bin($ciphertext), $nonce, $public_key . $private_key
);
```

Public-Key Crypto

Similarly, Sodium introduces simple methods to power cryptograph with *asymmetric* keys (where a public key is distributed for encryption, and a private key is used for decryption). These functions are simply named `sodium_crypto_box()` and `sodium_crypto_box_open()`. As with the secret key model above, encryption requires a unique nonce for every operation (and decryption requires the same nonce). Refer to Listing 3.

When sending a message to a third party, you send the ciphertext, the nonce that helped generate it, and your public key as well. When Sodium begins decrypting the message, it will check a message authentication code to authenticate the message and will use your public key to both help authenticate and decrypt the message. Not only can the recipient verify *you* sent the message, but they can also verify one else has manipulated it.

As with the symmetric decryption above, this operation is *authenticated*. If the MAC affixed to the message fails to validate, the message has been manipulated in transit, and the decryption operation will abort.

Keeping Data Secure

As of November, PHP is the first language to ship with a modern cryptographic library available by default and without the need for third-party extensions. This new feature introduces both secret and public key cryptography, without requiring you to install anything else on the server. It's a fantastic way to leverage encryption which will run on any system using PHP 7.2.

The choice of elliptic curves and specific algorithms makes it easy for developers everywhere to utilize fast, secure crypto without needing to take a graduate-level course in cryptography. The higher-level abstractions provided by the library also make it harder to make a mistake when using the exposed cryptographic primitives.

As of PHP 7.2, we have the tools available to easily and concretely keep our customers' data secure. Let's wield this power responsibly and use it wherever possible!

Eric is a seasoned web developer experienced with multiple languages and platforms. He's been working with PHP for more than a decade and focuses his time on helping developers get started and learn new skills with their tech of choice. Eric works as a Tekton for Tozny, a privacy and security-focused startup in the Portland area. You can reach out to him directly via Twitter: @EricMann