www.phparch.com

php[architect]

September 2019
Volume 18 - Issue 9

Master of Puppets

End-to-End Testing Automation With PuPHPeteer

Load Testing Your App with K6

Reformat, Refactor, Replace

ALSO INSIDE

Education Station:
Visual Studio Code for PHP Developers

Community Corner:
On Diversity in Conference Speakers

Pragmatic PHP:
Studying Singletons

Security Corner:
Twist and Shout

The Workshop:
Real World PDF Generation

finally{}:
The State of PHP User Groups

PHP[WORLD] 2019

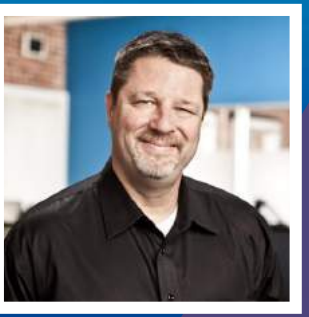
CELEBRATING 25 years of PHP programming

WASHINGTON DC
OCT 23-24

Join us this fall for our annual php[world] conference. This year marks the 25th anniversary of PHP and we plan to celebrate it in a big way!

KEYNOTE SPEAKERS

Samantha Quinoñes



Cal Evans

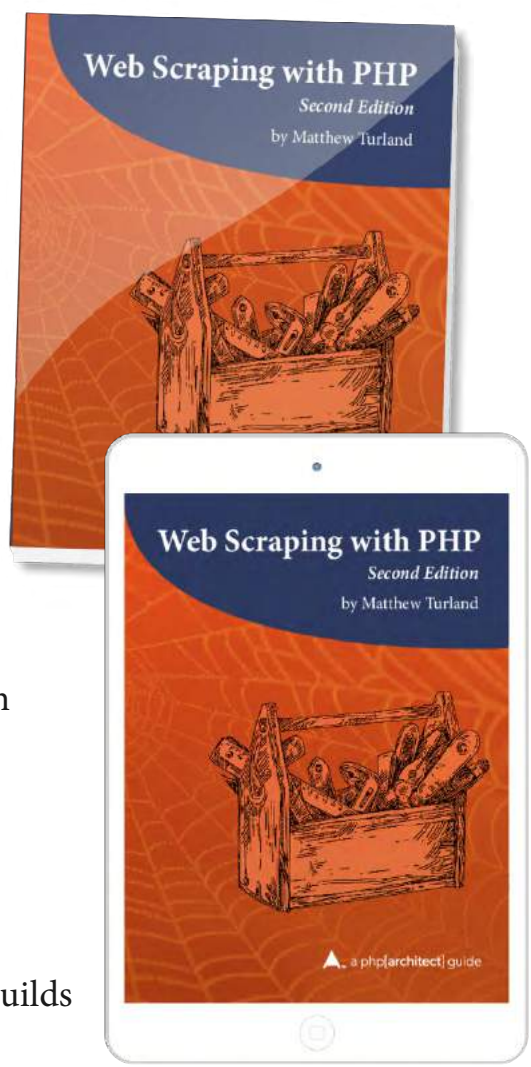
world.phparch.com

a php[architect] guide

Read a Sample Online

Integrating with another web site but an API is not available?

Web scraping is a time-honored technique for collecting the information you need from a web page. In this book, you'll learn the various tools and libraries available in PHP to **retrieve, parse, and extract data** from HTML.



Web Scraping with PHP, 2nd Edition includes updates to the techniques of the first edition to account for modern PHP 7 based libraries written to more easily interact with web markup and data.

- HTTP requests and responses
- PHP's HTTP Stream wrapper
- Using the cURL extension
- Working with the pecl_http extension
- Parsing responses with Guzzle
- Zend Framework's HTTP classes
- An overview of Symfony's libraries for web automation
- Writing a client from scratch
- Extensions for parsing and tidying XML and HTML
- Using regular expressions
- … and more.

Written by PHP professional Matthew Turland, this book builds on his expertise in creating custom web clients.

Available in Print, PDF, EPUB, and Mobi.

Order Your Copy

http://phpa.me/web-scraping-2ed

Twist and Shout

Eric Mann

Most self-taught developers in our industry learn to leverage an API long before they spend time learning lower-level coding patterns. This experience isn't necessarily a bad thing. All the same, it's important to take some time to dig deeper and better understand the tools and technologies at the core of our trade.

Computers and digital technology, in general, are highly deterministic. Provide a specific set of inputs to an algorithm, interface, or system, and you'll always get a reliable, predictable output in return. It's this determinism that makes computers so well suited for rote tasks and routine operations. However, this same determinism can also be a weakness—particularly in the realm of security.

A calculator that can't reliably or predictably solve arithmetic is useless. A token or identity verification service that issues predictable session identifiers to authenticating parties, however, is also useless. Said another way, we don't want PHP session IDs to be automatically incrementing integer values; otherwise, an attacker could easily predict and potentially hijack your users' otherwise secure data. Computers are deterministic by nature, so we need to leverage purpose-built random number generators to introduce unpredictability into the system.

Psuedo-Random Number Generators (PRNG)

When programming, most random number generators aren't truly random. They're based on specific algorithms which produce patterns that are very difficult, but not impossible to reproduce. These generators are "pseudo-random" and help provide a level of non-determinism our applications can leverage.

There's an entire field in computer science focused on developing genuinely random systems and tests to verify the randomness of anything intended to be random. For most purposes, pseudo-random systems are usually random enough.

> *Truly secure applications should leverage a cryptographically secure pseudo-random number generator (CSPRNG). A cryptographically secure system is one that passes rigorous statistical tests and is highly resistant to common forms of statistical attack. Both the* `random_bytes()` *and* `random_int()` *functions introduced in PHP 7*[1] *are considered cryptographically secure. You should use one of these functions by default anywhere you need randomness in your application. Older functions like* `mt_rand()` *(discussed in this article) served a purpose in older versions of PHP, but you should not use them unless you have a legitimate reason to do so in a legacy application. Even if, for some reason, you are still running PHP 5 and need a CSPRNG, the paragonie/random_compat*[2] *package provides the same functionality by way of a polyfill.*

A PRNG algorithm typically starts with a single seed, then generates an infinite (or at least very long) series of seemingly random numbers from that seed. The advantages of such an algorithm are that they're fast, easy to implement, and generate a deterministic sequence of numbers each time. Start with the same seed, and you'll always get the same pseudo-random sequence back. The most significant disadvantage of these algorithms is that it's easy to get the implementation wrong. A second disadvantage is, given a seed, you'll always get the same sequence of numbers back each time.

The algorithm's advantage—deterministic sequence generation—is also a drawback; we'll come back to this in a bit.

One of the most widely-used PRNGs in computer science is the Mersenne Twister[3], developed in 1997. It's the "mt" in PHP's `mt_rand()` function.

Merssene Twister

In a nutshell, PRNGs follow a pretty specific pattern[4]:

1. Initialize the state of the system from a specific seed.
2. Use a one-way function `f` to output a random number from that state.
3. Use a one-way function `g` to mutate the state.
4. Repeat steps 2-3 for every subsequent request of a random number.

The Mersenne Twister is a bit different. It still uses a seed, keeps track of state, and outputs random numbers but doesn't use one-way functions and the state is larger than a single number. The outline for the Merssene Twister is:

1. Initialize the state of the system from a specific seed.

1 *introduced in PHP 7:* https://php.net/book.csprng.php

2 *paragonie/random_compat:* https://phpa.me/random_compat

3 *the Mersenne Twister:* https://phpa.me/wikip-mersenne-twister

4 *specific pattern:* https://phpa.me/cryptologie-mersennes

2. *Twist* the state (this would be the `f` above).
3. *Temper* the state (this would be `g`) to return a random number.
4. Repeat steps 2-3 for every subsequent request.

One key difference between the Merssene Twist and other PRNGs is the state isn't a single number; it's an array of 624 numbers. Each iteration of the algorithm could produce 624 distinct random numbers rather than a single one!

Let's work through each step, in turn, to see how the algorithm looks in userland PHP.

Step 1. Initialize

Given a seed, we need to initialize an array of 624 integers. We can pick any seed—PHP defaults to the current Unix timestamp multiplied by the process ID in an attempt to provide a somewhat "random" seed which differs between environments. We need to filter our seed to ensure it's only 32 bits, then use a magic initialization constant[5] and some well-known bit arithmetic[6] to initialize our state container as in Listing 1.

Listing 1

```
<?php

class MT_Rand
{
    private $state = [];

    public function __construct(?int $seed = null) {
        $seed = $seed ?? time();

        $this->state = [$seed & 0xffffffff];

        foreach (range(1, 624) as $i) {
            $this->state[$i] = (
                (
                    (0x6c078965
                        * ($this->state[$i - 1]
                            ^ ($this->state[$i - 1] >> 30))
                    )
                    + $i)
            ) & 0xffffffff;
        }
    }

    // ...
```

5 *magic initialization constant:* https://phpa.me/cpp-mersenne-psuedo

6 *well-known bit arithmetic:* https://phpa.me/wikip-mersene-engine

Step 2. Twist

At this point, we have a 624-element array containing our initial state. Given the same seed, this initializes the same array every time. Before we can use it, though, we need to mutate the array with the `twist()` algorithm (See Listing 2).

Step 3. Temper

The almost-final step, shown in Listing 3, is to "temper"[7] the transformation created by the twist operation above. This tempering helps to "compensate for the reduced dimensionality of"[8] the distribution of numbers in our state array and render each subsequent iteration unpredictable.

Listing 2

```
1.    private function twist() {
2.       foreach (range(0, 624) as $i) {
3.          $y = (($this->state[$i] & 0x80000000)
4.                + ($this->state[($i + 1) % 624] & 0x7fffffff))
5.                & 0xffffffff;
6.
7.          $this->state[$i] = (
8.                $this->state[($i + 397) % 624] ^ ($y >> 1)
9.             ) & 0xffffffff;
10.
11.         if (($y % 2) == 1) {
12.            $this->state[$i] = ($this->state[$i] ^ 0x9908b0df)
13.                               & 0xffffffff;
14.         }
15.      }
16.   }
```

Listing 3

```
1.    function temper() {
2.       static $index = 0;
3.
4.       if ($index == 0) {
5.          $this->twist();
6.       }
7.
8.       $y = $this->state[$index];
9.       $y = ($y ^ ($y >> 11)) & 0xffffffff;
10.      $y = ($y ^ (($y << 7) & 0x9d2c5680)) & 0xffffffff;
11.      $y = ($y ^ (($y << 15) & 0xefc60000)) & 0xffffffff;
12.      $y = ($y ^ ($y >> 18)) & 0xffffffff;
13.
14.      $index = ($index + 1) % 624;
15.
16.      return $y;
17.   }
```

7 *"temper":* https://phpa.me/wikip-tempered-rep

8 *"compensate for the reduced dimensionality of":* https://phpa.me/wikip-mersenne-algo

Step 4. Output

Once we've initialized, twisted, and tempered, we can return the final output as the next random number in our series. The PHP implementation[9] explicitly confirms with the reference implementation in `mt19937ar.c` and performs a right-shift to return a 31-bit integer. If we want to match the same, then our entire operation resembles Listing 4.

To verify our implementation, we can wire all of the above scripts together and compare directly with PHP's `mt_rand()`. The trick is to set the same seed value for each run as in Listing 5.

If you execute the above at the command line, you'll see the contents of Output 1.

Listing 4

```
1.    public function __invoke() {
2.       $random = $this->temper();
3.       return ($random >> 1) & 0x7fffffff;
4.    }
5.
6. } // end of class
7.
8. $seed = 12345;
9. $rand = new MT_Rand($seed);
10. $random = $rand();
```

Listing 5

```
1. <?php
2. // Our series
3. $rand = new MT_Rand(42);
4. echo '---- Our implementation ----' . PHP_EOL;
5. echo $rand() . PHP_EOL;
6. echo $rand() . PHP_EOL;
7. echo $rand() . PHP_EOL;
8.
9. echo '---- PHP mt_rand() ----' . PHP_EOL;
10. mt_srand(42); // sets the seed
11. echo mt_rand() . PHP_EOL;
12. echo mt_rand() . PHP_EOL;
13. echo mt_rand() . PHP_EOL;
```

Output 1

```
1. $ php mt_rand.php
2. ---- Our implementation ----
3. 804318771
4. 1710563033
5. 2041643438
6. ---- PHP mt_rand() ----
7. 804318771
8. 1710563033
9. 2041643438
```

9 *PHP implementation:* https://phpa.me/php-src-mt-rand

Both our userland implementation above and the default PHP implementation of the Mersenne Twister generate the same pseudo-random series given the same seed. This property is useful for testing purposes or when you need a predictable but seemingly random series. Unfortunately, this consistency is one of the indicators of this system's inherent weakness.

Weaknesses In Implementation

Neither the "twist" nor "temper" operations are one-way functions. If you know the output of the system, you can "untemper" it to return to the raw state, then "untwist" the state to get to a previous step. Go back far enough, and you can even recover the initial seed. Since this Merssene Twist implementation leverages a 624-element internal state, capturing 624 subsequent outputs of the PRNG gives an attacker sufficient information to walk back through the algorithm and effectively break the system!

> *Yet all is not perfect in terms of non-predictability. The MT19937 algorithm keeps track of its state in 624 32-bit values. If an attacker were able to gather 624 sequential values, then the entire sequence—forward and backward—could be reverse-engineered. source10*Similarly, knowing exactly how the PRNG is seeded gives an attacker a good idea of where to start with an attack by guessing the seed and checking the RNG's output. If you fail to provide a seed, the algorithm will seed itself[11] with an integer based on the PHP process' ID and the current UNIX time in seconds.

Again, the deterministic nature of the computer makes it reasonably straight forward for an attacker to brute force their way to a root value. Once an attacker discovers a seed, they can predict every "random" number your system uses moving forward.

The Mersenne Twister is a handy algorithm for some things internal to your application that need to seem random but for which true randomness is not a strict requirement. If you need true randomness, instead use a *cryptographically secure* PRNG like `random_bytes()`[12] or `random_int()`[13].

Never Roll Your Own Crypto!

Now we're to the part that includes shouting. It's a critical point, so lean in and ensure you're listening.

Never. Roll. Your. Own. Cryptography.

It is *incredibly* important you fully understand the risk associated with designing or building your cryptographic system. It's vital you work with the experts who understand the proper implementations of the system you need to use, and even experts make mistakes. The original PHP implementation of `mt_rand()` actually had a typo[14] that rendered it incompatible with the canonical mt19937 reference implementation[15] of the Mersenne Twister. This bug has since been fixed, but the point that even the experts can get it wrong should be enough.

Your team should fully understand the inner workings of the libraries, tools, and technologies upon which your project stands. Knowing the strengths and limitations of these more primitive implementations helps your team write better, more stable software. Leverage expertise when and where possible—let the cryptography experts write the crypto while your in-house application experts write the application. This domain-specific focus will undeniably lead to better deliverables.

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. You can reach out to him directly via Twitter: @EricMann

Related Reading

- *Security Corner: Adventures in Hashing*
  by Eric Mann, December 2018.
  https://phpa.me/security-corner-dec-2018
- *Cryptography Best Practices in PHP*
  by Enrico Zimuel, May 2017.
  http://phparch.com/magazine/2017-2/may
- *Implementing Cryptography*
  by Edward Barnard, July 2016.
  http://phparch.com/magazine/2016-2/july

10 *source:* https://phpa.me/science-direct-mt

11 *seed itself:* https://phpa.me/php-src-php-rand

12 `random_bytes()`: https://php.net/random-bytes

13 `random_int()`: https://php.net/random-int

14 *actually had a typo:* https://bugs.php.net/bug.php?id=71152

15 *mt19937 reference implementation:* https://phpa.me/cpp-mt19937

Borrowed this magazine?

Get php[**architect**] delivered to your doorstep or digitally every month!

Each issue of php[**architect**] magazine focuses on an important topic that PHP developers face every day.

We cover topics such as frameworks, security, ecommerce, databases, scalability, migration, API integration, devops, cloud services, business development, content management systems, and the PHP community.

Digital and Print+Digital Subscriptions Starting at $49/Year

http://phpa.me/mag_subscribe

php[**architect**]