Poradnik o budowaniu bezpiecznego oprogramowania PHP w 2018r.

[https://paragonie.com/blog/2017/12/2018-guide-building-secure-php-softwarehttps://paragonie.com/blog/2017/12/2018-guide-building-secure-php-software]

Wersje PHP

W skrócie: Jeśli masz taką możliwość to powinieneś używać PHP 7.2 w 2018, i planować przejście na 7.3 na początku 2019.

PHP 7.2 zostało wydane 30 października 2017. W momencie pisania tego artykułu, tylko PHP 7.1 i 7.2 mają aktywne wsparcie deweloperów, a PHP 5.6 i 7.0 będą otrzymywać łatki bezpieczeństwa jeszcze przez rok.

Niektóre systemy operacyjne zapewniają długoterminowe wsparcie dla nieobsługiwanych wersji PHP, ale ta praktyka jest ogólnie uważana za szkodliwą. W szczególności, ich zły nawyk przenoszenia poprawek bezpieczeństwa bez zwiększania numerów wersji sprawia, że trudno jest weryfikować bezpieczeństwo systemu, na podstawie tylko wersji PHP.

W związku z tym, niezależnie od tego, co obiecują inni dostawcy, zawsze staraj się używać tylko aktywnie wspierane wersje PHP (**actively supported**) w danym momencie, jeśli jest taka możliwość. W ten sposób, nawet jeśli będziesz przez jakiś czas używał wersji security-only, konsekwentne dążenie do aktualizacji pozwoli utrzyma Cię z dala od nieprzyjemnych niespodzianek.

Zarządzanie zależnościami

W skrócie: Używaj Composer'a.

Najnowocześniejsze rozwiązanie do zarządzania zależnościami dla ekosystemu PHP to **Composer**. *PHP: The Right Way* ma całą sekcję poświęconą rozpoczynaniu pracy z Composer (**getting started with Composer**), którą gorąco polecamy.

Jeśli nie używasz Composer'a do zarządzania swoimi zależnościami, w pewnym momencie (miejmy nadzieję, że *później*, ale najprawdopodobniej *wcześniej*) znajdziesz się w sytuacji, w której jedna z bibliotek oprogramowania, od której jesteś zależny, staje się mocno przestarzała, a następnie wśród przestępców komputerowych zacznie krążyć exploit dla luki w starszych wersjach.

Ważne: Zawsze pamiętaj o aktualizacji zależności (**keep your dependencies updated**) podczas rozwijania oprogramowania. Na nasze szczęście wymaga to tylko 1 wiersza:

```
composer update
```

Jeśli robisz coś szczególnie wyspecjalizowanego wymagającego używania rozszerzeń PHP (które są napisane w C), nie możesz ich zainstalować za pomocą Composer'a. Będziesz także potrzebował PECL.

Rekomendowane Paczki

Bez względu na to co budujesz, najprawdopodobniej odniesiesz korzyść dzięki tym paczkom. To jest dodatek do tego co rekomenduje większość deweloperów PHP (PHPUnit, PHP-CS-Fixer, itp)

roave/security-advisories

Paczka **Roave's security-advisories** używa repozytorium **Friends of PHP repository** by zapewnić że Twój projekt nie jest zależny od żadnych paczek, które zawierają znane podatności.

```
composer require roave/security-advisories:dev-master
```

Możesz również przesłać plik `composer.lock` do Sensio Labs w ramach rutynowego automatycznego workflow ocen podatności, aby poinformować Cię o wszelkich nieaktualnych paczkach.

vimeo/psalm

Psalm jest narzędziem analizy statycznej które pomaga identyfikować możliwe bugi w kodzie. Mimo, że istnieją inne dobre narzędzia analizy statycznej (np **Phan** i **PHPStan**) to jeśli znajdziesz się w sytuacji gdy potrzebujesz obsługiwać PHP 5, Psalm jest idealnym narzędziem analizy statycznej dla PHP 5.4+

Używanie Psalm jest dość proste:

```
# Version 1 doesn't exist yet, but it will one day:
composer require --dev vimeo/psalm:^0


# Only do this once:
vendor/bin/psalm --init
```

```
# Do this as often as you need:
vendor/bin/psalm
```

Jeśli używasz tego w istniejącej bazie kodu po raz pierwszy, to może pojawić się dużo czerwonego koloru. Jeśli nie budujesz aplikacji tak dużej, jak WordPress, to jest to mało prawdopodobne, że będziesz musiał włożyć w to tyle wysiłku co Herkules, by wszystkie testy były pozytywne.

Niezależnie od tego, którego narzędzia do analizy statycznej używasz, zalecamy włączenie go do twojego workflow “Continuous Integration” (jeśli dotyczy), aby było uruchamiane po każdej zmianie kodu.

HTTPS i Bezpieczeństwo Przeglądarki

W skrócie: HTTPS, które powinno być testowane (**tested**), i nagłówki bezpieczeństwa (**security headers**).

W 2018 dostęp do stron przez niezabezpieczone HTTP będzie już nie do zaakceptowania. Na szczęście, uzyskanie certyfikatów TLS za darmo było do tej pory możliwe, jak również automatycznie odnowienie tych certyfikatów dzięki protokołowi ACME i **Let's Encrypt certificate authority**.

Integracja ACME z twoim serwerem web jest prosta.

- **Caddy**: wbudowane automatycznie.
- **Apache**: Niedługo dostępne jako mod_md. Do tego czasu możesz użyć tutoriali (**tutorials**).
- **Nginx**: W miarę proste.

Możesz sobie pomyśleć “mam certyfikat TLS. Teraz muszę godzinami grzebać w konfiguracji żeby to było bezpieczne i szybkie”.

Nie! **Mozilla** ci pomoże. Możesz użyć generatora konfiguracji by zbudować zalecane **ciphersuites** oparte na twoich odbiorcach.

*HTTPS (HTTP przez TLS) jest absolutnie bezdyskusyjne (**absolutely non-negotiable**)* jeśli chcesz by twoja strona była bezpieczna. Używanie HTTPS od razu eliminuje kilka klas ataków na Twoich użytkowników (man-in-the-middle content injection, wiretapping,

replay attacks, i kilka form manipulacji sesją który mogłyby umożliwić podanie fałszywej tożsamości)

Nagłówki bezpieczeństwa (Security Headers)

Użycie HTTPS na serwerze oferuje różnorodne korzyści bezpieczeństwa wydajności dla użytkowników, jednak możesz pójść o krok dalej i użyć innych przeglądarkowych funkcji bezpieczeństwa. Większość z nich wymaga wysłania nagłówka odpowiedzi HTTP (HTTP response header) wraz z twoją treścią.

- `Content-Security-Policy`
  - Potrzebujesz tego nagłówka bo daje ci kontrolę nad wewnętrznymi i zewnętrznymi zasobami które przeglądarka może ładować, co pozwala na silną warstwę ochrony przeciwko podatnościom cross sir scripting
  - Zobacz **CSP-Builder** jeśli szukasz łatwego i szybkiego sposobu na wdrożenie/zarządzanie polityką bezpieczeństwa treści (Content Security Policies).
  - Dla głębszej analizy ten wstęp do Nagłówków polityki bezpieczeństwa treści (**Content-Security-Policy headers**) jest świetnym punktem wyjścia.
- `Expect-CT`
  - Potrzebujesz tego nagłówka bo on dodaje warstwę ochrony przeciwko nieuczciwymi/szkodliwymi certyfikatami poprzez zmuszanie sprawców do publikowania dowodów błędnie wydanych certyfikatów w publicznie weryfikowalnej strukturze danych typu “append-only”. Dowiedz się więcej o `Expect-CT`.
  - Ustaw `enforce,max-age=30` na początek i zwiększ `max-age` wraz z tym jak zwiększa się twoja pewność, że ten nagłówek nie będzie powodował zakłóceń usług.
- `Referrer-Policy`
  - Potrzebujesz tego nagłówka bo pozwala ci kontrolować czy pozwalasz na “wyciek” informacji do stron trzecich o zachowaniu twoich użytków.
  - Po raz kolejny **Scott Helme** oferuje świetne materiały na temat nagłówków `Referrer-Policy`
  - Ustaw `same-origin` lub `no-referrer` chyba że masz powód żeby ustawić mniej ograniczające ustawienia.
- `Strict-Transport-Security`

    - Potrzebujesz tego nagłówka bo on mówi przeglądarkom by wymuszały wszystkie przyszłe żądania na to samo źródło (origin) przez HTTPS zamiast niezabezpieczonego HTTP.
    - Ustaw `max-age=30` przy pierwszym wdrożeniu, potem zwiększ wartość (np. `31536000`) gdy jesteś pewien że nic się nie popsuje.
- `X-Content-Type-Options`
    - Potrzebujesz tego nagłówka bo nieporozumienia typu MIME (MIME type confusion) może prowadzić do nieprzewidywalnych rezultatów, w skrajnych przypadkach nawet do podatności XSS. Najlepiej zastosować to razem ze standardowym nagłówkiem `Content-Type`.
    - Ustaw `nosniff` chyba że potrzebujesz domyślnego zachowania (np. by pobrać plik).
- `X-Frame-Options`
    - Potrzebujesz tego nagłówka bo pozwala ci zapobiegać przechwyceniu kliknięć (**clickjacking**).
    - Ustaw `DENY` (lub `SAMEORIGIN`, ale tylko jeśli używasz elementów `<frame>`)
- `X-XSS-Protection`
    - Potrzebujesz tego nagłówka bo włącza w niektórych przeglądarkach funkcje przeciwdziałające XSS, które nie są domyślnie włączone.
    - Ustaw `1; mode=block`

Podobnie, jeśli używasz wbudowanych w PHP funkcji zarządzania sesją (które są zalecane), powinieneś wywołać `session_start()`:

```
session_start([
    'cookie_httponly' => true,
    'cookie_secure' => true
]);
```

To wymusza by twoja używała flag “HTTP-Only” i “Secure” gdy wysyła ciasteczko identyfikacji sesji, co nie pozwala udanemu atakowi XSS na wykradnięcie ciasteczek użytkownika, i wymusza by były one wysyłane tylko przez HTTPS. Mówiliśmy już o bezpiecznych sesjach PHP (**secure PHP sessions**) w poście z roku 2015.

Integralność pod-zasobów

W przyszłości pewnie będziesz pracował nad projektem który używa CDN by odładować zwykłe frameworki Javascript/CSS i biblioteki na centralną lokalizację.

Nie powinno być to zaskoczeniem że eksperci bezpieczeństwa już przewidzieli oczywiste pułapki: Jeśli wiele stron używa CDN by obsługiwać część ich treści to shackowanie CDN i podmienienie treści pozwoli ci wstrzyknąć dowolny kod do tysięcy (jeśli nie nawet milionów) stron.

Wprowadź integralność pod-zasobów (subresource integrity).

Integralność pod-zasobów (SRI) pozwala na użycie hash'a treści pliku który ma obsługiwać CDN. SRI w obecnej formie pozwala na użycie funkcji kryptograficznych hash'a, co oznacza że atakujący nie jest w stanie wygenerować szkodliwych wersji tych samych zasobów, które wytwarzają ten sam hash co oryginalny plik.

Przykład z życia: **Bootstrap v4-alpha uses SRI in their CDN example snippet**.

```
<link
    rel="stylesheet"

href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-alpha.6/css/bootstrap.min.css"

integrity="sha384-rwoIResjU2yc3z8GV/NPeZWAv56rSmLldC3R/AZzGRnGxQQKnKkoFVhFQhNUwEyJ"
    crossorigin="anonymous"
/>
<script
    src="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-alpha.6/js/bootstrap.min.js"

integrity="sha384-vBWWzlZJ8ea9aCX4pEW3rVHjgjt7zpkNpZk+02D9phzyeVkE+jo0ieGizqPLForn"
    crossorigin="anonymous"
></script>
```

Powiązania dokumentów

Deweloperzy web często ustawiają atrybut `target` na hyperlinkach (np. `target="_blank"` by link otworzył się w nowym oknie). Jeśli jednak nie przekazujesz też tagu `rel="noopener"`, to możesz pozwolić stronie docelowej przejąć kontrolę nad oryginalną stroną (**allow the target page to take control of the original page**).

Nie rób tego

```
<a href="http://example.com" target="_blank">Click here</a>
```

To pozwala `example.com` przejąć kontrolę nad obecną stroną.

Rób tak

```
<a href="https://example.com" target="_blank" rel="noopener noreferrer">Click here</a>
```

To otwiera `example.com` w nowym oknie ale nie poddaje kontroli nad obecnym oknem potencjalnie złośliwej stronie trzecie. Przeczytaj więcej: **Further reading**.

Tworzenie bezpiecznego oprogramowania PHP

Jeśli bezpieczeństwo aplikacji to nowe zagadnienie dla Ciebie to zacznij od tego: ***A Gentle Introduction to Application Security***.

Większość ekspertów bezpieczeństwa kieruje deweloperów od razu do źródeł typu **OWASP Top 10**.

Jednakże, większość zwykłych podatności może być wyrażona jako specyficzne instancje tego samego problemu wysokiej rangi (kod/dane niewystarczająco

odseparowane, słaba logika, niezabezpieczone środowisko operacyjne, naruszone protokoły kryptograficzne)

Nasza hipoteza jest taka, że uczenie nowicjuszy prostszego, bardziej fundamentalnego zrozumienia tych problemów bezpieczeństwa i jak je rozwiązać doprowadzi do lepszej inżynierii bezpieczeństwa na dłuższą metę.

Zatem, unikamy polecania list kontrolnych typu top10 lub top 25.

Interakcja z bazą danych

Szczegółowo: Zapobieganie **SQL Injection** w aplikacjach PHP

Jeśli sam piszesz zapytania SQL upewnij się że używasz przygotowanych wyrażeń, i że jakiekolwiek informacje dostarczane przez sieć lub system plików są przekazane jako parametr, a nie są włączane do ciągu zapytania. Dodatkowo, upewnij się że nie używasz emulowanych przygotowanych wyrażeń(***emulated* prepared statements**).

Dla uzyskania najlepszych rezultatów użyj **EasyDB**.

NIE RÓB TEGO:

```
/* Insecure code: */
$query = $pdo->query("SELECT * FROM users WHERE username = '" . $_GET['username'] .
"'");
```

RÓB TAK:

```
/* Secure against SQL injection: */
$results = $easydb->row("SELECT * FROM users WHERE username = ?", $_GET['username']);
```

Istnieją inne warstwy abstrakcji bazy danych które oferują podobny poziom bezpieczeństwa (EasyDB używa PDO w tle, ale robi wszystko by wyłączyć emulację przygotowanych wyrażeń na rzecz właściwych przygotowanych wyrażeń, by nie strzelić sobie w stopę jeśli chodzi o bezpieczeństwo). Dopóki dane wprowadzanie przez użytkownika nie mogą wpływać na strukturę zapytań będziesz bezpieczny. (To obejmuje przechowywane procedury).

Wgrywanie pliku

Szczegółowo: Jak bezpiecznie pozwolić użytkownikom wgrywać pliki (**Upload Files**).

Akceptowanie wgrywanych plików jest ryzykowne ale da się to zrobić w bezpieczny sposób, jeśli pewne podstawowe środki bezpieczeństwa zostaną zachowane. Chodzi o: Zapobieganie bezpośredniemu dostępowi wgranych plików w taki sposób który przypadkowo mógłby pozwolić na uruchomienie lub interpretację tych plików.

Wgrywane pliki powinny być tylko do odczytu albo odczyt/zapis, nigdy wykonywalne.

Jeśli rootem twojego dokumentu strony internetowej jest `/var/www/example.com`, nie przechowuj plików w `/var/www/example.com/uploaded_files`.

Zamiast tego, przechowuj pliki w odrębnym katalogu który nie jest bezpośrednio dostępny (np. `/var/www/example.com-uploaded/`), by nie zostały przypadkowo uruchomione jako skrypt po stronie serwera i pozwoliły na zdalne wykonanie kodu.

Bardziej czyste rozwiązanie to przeniesienie roota dokumentu o jeden poziom w dół (np.do `/var/www/example.com/public`).

Innym problemem z wgrywaniem plików jest ich bezpieczne pobieranie.

- Obrazki SVG, w przypadku bezpośredniego dostępu uruchomią kod JavaScript w przeglądarce użytkownika. To prawda, mimo mylącego prefiksu `image/` w MIME type.
- MIME type sniffing może prowadzić do ataków nieporozumienia (confusion attacks) jak wspominaliśmy poprzednio. Zobacz `X-Content-Type-Options` w sekcji “nagłówki bezpieczeństwa”.
- Jeśli zrezygnujesz z poprzedniej porady na temat bezpiecznego przechowywania przesłanych plików, atakujący, któremu uda się przesłać plik `.php` lub `.phtml` może być w stanie uruchomić dowolny kod poprzez uzyskanie dostępu bezpośrednio w przeglądarce, tym samym dając mu całkowitą kontrolę nad serwerem. Bądź ostrożny.

Cross-Site Scripting (XSS)

Szczegółowo: Wszystko co musisz wiedzieć o przeciwdziałaniu podatnościom na XSS w PHP (**Preventing Cross-Site Scripting**)

W idealnym świecie XSS byłoby tak łatwe do zapobiegnięcia jak SQL injection. Mielibyśmy proste, łatwe w użyciu API do separowania struktury dokumentu od danych które go zapełniają.

Niestety doświadczenia z prawdziwego życia większości deweloperów web wiążą się z generowaniem długich blobów HTML i wysyłaniem ich w odpowiedzi HTTP. To nie dotyczy tylko rzeczywistości PHP, tak po prostu działa programowanie.

Ograniczanie podatności na XSS nie jest jest sprawą z góry przegraną. Jednak wszystko w sekcji o bezpieczeństwie przeglądarek nagle staje się bardzo istotne. W skrócie:

1. **Zawsze uciekaj[?] (escape) na danych wyjściowych, nigdy na wejściowych.** Jeśli przechowujesz oczyszczone dane w bazie, a podatność na SQL injection jest potem znaleziona w innym miejscu, to atakujący może całkowicie ominąć twoje zabezpieczenia XSS poprzez zanieczyszczenie tego niby oczyszczonego rekordu złośliwym oprogramowaniem.
2. Jeśli twój framework ma silnik szablonowy który oferuje automatyczne kontekstowe filtrowanie to używaj tego. To zadanie twojego framework'u by zrobić to bezpiecznie.
3. `echo htmlentities($string, ENT_QUOTES | ENT_HTML5, 'UTF-8');` jest bezpiecznym i efektywnym sposobem by zatrzymać wszystkie ataki XSS na stronę zakodowaną w UTF-8, ale nie pozwala na żadne HTML.
4. Jeśli twoje wymagania pozwalają ci na użycie Markdown zamiast HTML to nie używaj HTML.
5. Jeśli musisz zezwolić na trochę HTML i nie używasz silnika szablonowego (punkt 1) to użyj **HTML Purifier**. HTML Purifier nie jest odpowiedni na uciekanie (escape) do kontekstu atrybutu HTML (HTML attribute context).
6. Dla URL przesłanych przez użytkowników powinieneś dodatkowo zezwalać na `http:` i `https:`; nigdy `javascript:`. Dodatkowo koduj za pomocą URL wszystkie dane wejściowe użytkownika.

Cross-Site Request Forgery (CSRF)

Cross-Site Request Forgery jest rodzajem zdezorientowanego ataku zastępczego, gdzie oszukujesz przeglądarkę użytkownika by wykonała złośliwe żądanie HTTP dla atakującego, z podwyższonymi uprawnieniami użytkownika.

Sposób na rozwiązanie tego jest dziecinnie prosty w łatwych 2 krokach:

1. Używaj HTTPS. To jest warunek początkowy. Bez HTTPS każda ochrona staje się krucha. Jednakże HTTPS samo nie zapobiegnie CSRF.
2. Dodaj podstawowe zabezpieczenie typu challenge-response authentication.
    - Dodaj ukryty atrybut formularza do każdego formularza
    - Zapełnij kryptograficznie bezpieczną losową wartością (tzw “token”)
    - Zweryfikuj że ukryty atrybut formularza został przesłany i zgadza się z tym czego oczekujesz.

Napisaliśmy bibliotekę **Anti-CSRF** która idzie o krok dalej:

- Każdy token może być użyty tylko raz, by zapobiec atakom typu “replay”
    - Wiele tokenów jest przechowywanych w backend.
    - Rotacja tokenów następuje gdy ich pojemność jest osiągnięta, zaczynając od najstarszych.
- Każdy token może być powiązany z konkretnym URI
    - Jeśli jeden token wycieknie, to nie można go użyć w innym kontekście.
- Tokeny mogą opcjonalnie być powiązane z konkretnym IP.
- Od v2.1 tokeny mogą być ponownie użyte (np.dla AJAX calls).

Jeśli nie używasz frameworku który zajmuje się podatnościami na CSRF to wypróbuj Anti-CSRF.

W najbliższej przyszłości, ciasteczka **SameSite** pozwolą nam pozbyć się ataków CSRF w mniej złożony sposób.

Ataki XML (XXE, XPath Injection)

Istnieją 2 główne podatności które napotkasz w aplikacjach które przetwarzają dużo XML:

1. XML External Entities (XXE)
2. XPath Injection

Ataki XXE mogą być wykorzystane jako platforma startowa dla lokalnych/zdalnych exploitów inkluzji plików (file inclusion exploits).

Wcześniejsza wersja Google Docs padła ofiarą XXE ale ale są one w większości przypadków niespotykane poza aplikacjami biznesowymi, które wykonują dużo ciężkiej roboty związanej z XML.

Najlepszym sposobem łagodzenie ataków XXE jest:

```
libxml_disable_entity_loader(true);
```

XPath Injection jest bardzo podobne do SQL Injection, tylko ma zastosowanie w dokumentach XML.

Na szczęście, sytuacje gdzie przekazujesz dane wejściowe użytkownika do zapytania XPath nie zdarzają się często w ekosystemie PHP.

Na nieszczęście, to też oznacza że najlepszy dostępny sposób na łagodzenie (wstępnie skompilowane i sparametryzowane zapytania XPath) nie występuje w ekosystemie PHP.

Najlepiej jest użyć białej listy dozwolonych znaków na jakichkolwiek danych, które dotykają zapytania XPath.

```
<?php
declare(strict_types=1);

class SafeXPathEscaper
{
    /**
     * @param string $input
     * @return string
     */
    public static function allowAlphaNumeric(string $input): string
    {
```

```
        return \preg_replace('#[^A-Za-z0-9]#', '', $input);
    }

    /**
     * @param string $input
     * @return string
     */
    public static function allowNumeric(string $input): string
    {
        return \preg_replace('#[^0-9]#', '', $input);
    }
}

// Usage:
$selected = $xml->xpath(
    "/user/username/" . SafeXPathEscaper::allowAlphaNumeric(
        $_GET['username']
    )
);
```

Białe listy są bezpieczniejsze niż czarne listy.

Deserializacja i PHP Object Injection

Szczegółowo: Bezpieczne wdrażanie (De)Serializacji w PHP([(De)Serialization in PHP]())

Jeśli przekażesz niezaufane dane do `unserialize()`, to prosisz się o jeden z dwóch wyników:

1. PHP Object Injection, które może być użyte do rozpoczęcia łańcucha POP i innych podatności związanych z nieodpowiednio użytymi obiektami.
2. Uszkodzenie pamięci w samym PHP interpreter.

Większość deweloperów woli używać serializacji JSON co stanowi znaczącą poprawę w bezpieczeństwie ich oprogramowania, ale pamiętaj że `json_decode()` jest podany na

ataki **hash-collision denial-of-service (Hash-DoS)**. Niestety, całkowity fix dla Hash-Dos dla PHP nie został jeszcze znaleziony

Migracja z djb33 do Siphash z najwyższym bitem danych wyjściowych hash'a ustawionym na 1 dla danych wejściowych ciągu, i ustawionym na 0 dla danych wejściowych liczb całkowitych, z kluczem na żądanie dostarczonym przez CSPRNG na pewno rozwiązałaby te ataki.

Niestety, zespół PHP nie jest jeszcze gotowy, aby pozbyć się przyrostów wydajności, które osiągają w serii PHP 7, więc trudno jest ich przekonać by zrezygnowali z djb33 (które jest bardzo szybkie, ale też niebezpieczne) na korzyść SipHash (który jest szybki, ale nie tak szybki jak djb33, jednak o wiele bezpieczniejszy). Znaczący spadek wydajności może nawet zniechęcić do przyjęcia przyszłych wersji, co z kolei będzie szkodliwe dla bezpieczeństwa.

Najlepiej zatem zrobić tak:

- Używaj JSON, bo jest bezpieczniejszy niż `unserialize()`.
- Gdzie to możliwe upewnij się że dane wejściowe są uwierzytelnione zanim je zdeserializujesz.
    - Dla danych które dostarczasz użytkownikowi końcowemu używaj `sodium_crypto_auth()` i `sodium_crypto_auth_verify()` z sekretnym kluczem znanym tylko serwerowi web.
    - Dla danych dostarczanych przez inne strony trzecie zaplanuj by podpisywały one ich wiadomości JSON przez `sodium_crypto_sign()` i potem zweryfikuj je za pomocą `sodium_crypto_sign_open()` i kluczem publicznym tej strony trzeciej.
        - Możesz też użyć odrębnego API podpisu (detached signing API) jeśli musisz zakodować te podpisy za pomocą hex lub base64 w celu transportu.
- Gdzie nie możesz uwierzytelnić ciągów JSON zastosuj ścisłe ograniczanie częstotliwości (strict rate-limiting) i blokuj adresy IP by zapobiec wielokrtonym atakom przez tę samą osobę.

Hashowanie haseł

Szczegółowo: Jak bezpiecznie przechowywać hasła użytkowników w roku 2016. (**How to Safely Store Passwords**)

Bezpieczne przechowywanie haseł było kiedyś tematem gorącej debaty ale w obecnych czasach jest dość łatwe do zaimplementowania, zwłaszcza w PHP:

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

if (\password_verify($password, $hash)) {
    // Authenticated.
    if (\password_needs_rehash($hash, PASSWORD_DEFAULT)) {
        // Rehash, update database.
    }
}
```

Nie musisz nawet wiedzieć, jaki algorytm jest używany w tle, ponieważ jeśli korzystasz z najnowszej wersji PHP, będziesz również korzystać z najnowocześniejszych technologii, a hasła użytkowników będą automatycznie aktualizowane, gdy tylko nowy domyślny algorytm będzie dostępny.

Cokolwiek zrobisz, nie zrób tego co robi **WordPress**.

Jeśli jednak jesteś ciekaw: od PHP 5.5 do 7.2 domyślnym algorytmem jest bcrypt. W przyszłości może się to zmienić na Argon2, czyli zwycięzcę konkursu **Password Hashing Competition**.

Jeśli do tej pory *nie używałeś* `password_*` API i masz starsze hashe które muszą być zmigrowane, upewnij się że robisz to w ten sposób - ***this* way**. Wiele firm popełniło ten błąd, najsłynniejszym przypadkiem jest **Yahoo**. Ostatnio niepoprawna implementacja aktualizacji starszych hashy najprawdopodobniej spowodowała bug **Apple `iamroot`**.

Kryptografia ogólnego przeznaczenia

To temat o którym pisaliśmy więcej tutaj:

- **Using Encryption and Authentication Correctly** (2015) [Poprawne używanie szyfrowania i uwierzytelniania]
- **Rekomendowane**: **Choosing the Right Cryptography Library for your PHP Project: A Guide** (2015) [Wybór odpowiedniej biblioteki kryptograficznej dla twojego projektu PHP: Przewodnik]
- **Rekomendowane**: **You Wouldn't Base64 a Password - Cryptography Decoded** (2015) [Nie użyłbyś Base64 na Haśle - Kryptografia odkodowana]
- **Cryptographically Secure PHP Development** (2017) [Kryptograficznie bezpieczne rozwijanie PHP]
- **Rekomendowane**: **Libsodium Quick Reference: Similarly-Named Functions and Their Use-Cases**(2017) [Krótki Opis Libsodium: podobnie nazwane funkcje i ich przypadki użycia]

Ogólnie rzecz biorąc, zawsze powinieneś używać biblioteki kryptograficznej Sodium (libsodium) do kryptografii warstwy aplikacji. Jeśli musisz obsługiwać wersji PHP wcześniejsze niż 7.2 (tak wczesne jak 5.2.4), możesz użyć **sodium_compat** i w zasadzie udawać, że twoi użytkownicy również korzystają z wersji 7.2.

W szczególnych przypadkach może być potrzebna inna biblioteka ze względu na restrykcyjne wybory algorytmów i interoperacyjność. W razie wątpliwości należy skonsultować się z kryptografem w sprawie wyboru szyfrów i z inżynierem kryptografii, czy wdrożenie jest bezpieczne. (Jest to jedna z usług, które my również świadczymy - **services**).

Jeśli pracujesz z szyframi/trybami bezpośrednio, to sprawdź ten krótki przewodnik o najlepszych praktykach kryptograficznych - **guide on cryptography**.

Losowość

Szczegółowo: Jak bezpiecznie generować losowe ciągi i liczby całkowite w PHP- **Generate Random Strings and Integers in PHP**

Jeśli potrzebujesz losowych liczb to użyj **random_int()**. Jeśli potrzebujesz losowych ciągów bajtów użyj **random_bytes()**. **Nie używaj `mt_rand()`, `rand()`, lub `uniqid()` w tym celu.**

Jeśli potrzebujesz wygenerować pseudolosowe liczby z secret seed, zamiast `srand()` lub `mt_srand()`, sprawdź **SeedSpring**.

```php
<?php
use ParagonIE\SeedSpring\SeedSpring;

$seed = random_bytes(16);
$rng = new SeedSpring($seed);

$data = $rng->getBytes(1024);
$int = $rng->getInt(1, 100);
```

Żądania HTTPS po stronie serwera

W skrócie: Upewnij się że walidacja certyfikatu nie jest wyłączona.

Używaj jakiegokolwiek klienta HTTP kompatybilnego z PSR-7, który już znasz. My lubimy Guzzle. Niektórzy lubią pracować bezpośrednio z cURL.

Czegokolwiek będziesz używał, upewnij się że używasz **Certainty**, by zawsze mieć najnowsze pakiety CACert, co pozwoli ci włączyć najbardziej rygorystyczne ustawienia walidacji certyfikatów TLS i zabezpieczy żądania wychodzące twojego serwera.

Instalacja Certainty jest łatwa:

```
composer require paragonie/certainty:^1
```

Używanie Certainty też jest łatwe:

```php
<?php
use ParagonIE\Certainty\RemoteFetch;

$latestCACertBundle = (new RemoteFetch())->getLatestBundle();

# cURL users:
    $ch = curl_init();
    curl_setopt($ch, CURLOPT_SSL_VERIFYHOST, 2);
    curl_setopt($ch, CURLOPT_SSL_VERIFYPEER, true);
    curl_setopt($ch, CURLOPT_CAINFO, $latestCACertBundle->getFilePath());

# Guzzle users:
    /** @var \GuzzleHttp\Client $http */
    $repsonse = $http->get(
        'https://example.com',
        [
            'verify' => $latestCACertBundle->getFilePath()
        ]
    );
```

To ochroni cię przed atakami typu man-in-the-middle między Twoim serwerem i API zewnętrznych aplikacji z którymi się integrujesz.

Czy *naprawdę* potrzebujesz Certainty?

Certainty nie jest koniecznie niezbędne by chronić twoje systemy. Jego brak nie powoduje podatności.

Ale bez Certainty oprogramowanie open source musi zgadywać gdzie jest pakiet CACert, i jeśli się pomyli, to często kończy się to niepowodzeniem i powoduje problemy z użytecznością.
Historycznie, zachęciło to wielu programistów do wyłączenia sprawdzania poprawności certyfikatów, aby ich kod "po prostu zadziałał", jednak nie zdawali oni sobie sprawy, jak bardzo narazili swoje aplikacje na aktywne ataki.

Certainty usuwa tę zachętę poprzez aktualizację pakietów CACert i przechowywanie ich w przewidywalnej lokalizacji. Certainty też zapewnia również wiele narzędzi dla przedsiębiorstw, które chcą uruchomić własny system CA.

Kto wyłącza walidację certyfikatów?

Robią to deweloperzy wtyczek / rozszerzeń dla popularnych systemów zarządzania treścią (WordPress, Magento itp.)! To ogromny problem, który próbujemy rozwiązać na poziomie ekosystemu. Ten problem nie dotyczy konkretnego systemu CMS, znajdziesz wtyczki / itp. dla wszystkich tych CMS, które są na to podatne.

Jeśli korzystasz z takiego CMS, znajdź w swoich wtyczkach `CURLOPT_SSL_VERIFYPEER` i `CURLOPT_SSL_VERIFYHOST` i prawdopodobnie znajdziesz kilka których wartości będą ustawione na `FALSE`.

Czego unikać

Nie używaj **mcrypt**, biblioteki kryptograficznej która nie jest rozwijana od dekady. Jeśli śledzisz nasze rekomendacje wersji **PHP**, to powinno to być łatwe do uniknięcia, ponieważ mcrypt nie jest dostarczany z PHP 7.2 i nowszymi.

Zalecenia bezpieczeństwa oparte na konfiguracji powinny być w większości przypadków lekceważone. Jeśli czytasz przewodnik bezpieczeństwa PHP i mówią Ci, aby zmienić ustawienia php.ini zamiast pisać lepszy kod, to prawdopodobnie czytasz bardzo przestarzały przewodnik. Zamknij okno i przejdź do czegoś, co nie rozwodzi się nad `register_globals`.

Nie używaj **JOSE (JWT, JWS, JWE)**, zestawu internetowych standardów które kodyfikują serię projektów kryptograficznych podatnych na błędy, które z jakiegoś powodu przyciągają wielu ewangelistów pomimo tego że przysłowiowe “strzały w stopę” są w nie wpisane.

Szyfrowanie parametrów adresu **URL** jest antywzorcem projektowym, którego firmy często używają do zaciemniania metadanych (np. Ilu użytkowników mamy?). Niesie to ze sobą duże ryzyko błędu implementacji, tworząc fałszywe poczucie bezpieczeństwa. Proponujemy bezpieczniejszą alternatywę w podlinkowanym artykule.

Nie wdrażaj funkcjonalności “Zapomniałem hasła” chyba że koniecznie musisz. By nie przebierać w słowach - **Funkcje resetowania hasła to backdoor**. Istnieją sposoby ich wdrożenia, które zabezpieczają przeciwko rozsądnemu modelowi zagrożenia, ale użytkownicy wysokiego ryzyka powinni mieć możliwość całkowitego wycofania się z tej opcji.

Unikaj używania **RSA** jeśli możesz. Używaj libsodium zamiast tego. Jeśli musisz używać RSA upewnij się że zdefiniujesz OAEP padding.

```
<?php

openssl_private_decrypt(
    $ciphertext,
    $decrypted, // Plaintext gets written to this variable upon success,
    $privateKey,
    OPENSSL_PKCS1_OAEP_PADDING // Important: DO NOT OMIT THIS!
);
```

Jeśli jesteś zmuszony używać PKCS #1 v1.5 padding, to cokolwiek z czym się integrujesz, jest prawie na pewno podatne na **ROBOT**, więc zgłoś go do odpowiedniego dostawcy (lub US-CERT) jako lukę umożliwiającą ujawnienie jawnego tekstu i fałszowanie podpisu .

Specjalistyczne przypadki użycia

Teraz, gdy już znasz podstawy budowy bezpiecznych aplikacji PHP w 2018 roku i później, przeanalizujmy niektóre z bardziej specjalistycznych przypadków użycia.

Szyfrowanie możliwe do wyszukania (Searchable Encryption)

Szczegółowo: Budowanie przeszukiwalnych zaszyfrowanych baz danych z PHP i SQL (**Building Searchable Encrypted Databases with PHP and SQL**)

Przeszukiwalne zaszyfrowane bazy danych są pożądane, ale powszechnie uważane za trudne do wdrożenia. W powyższym wpisie na blogu próbuje się przeprowadzić czytelnika przez rozwój naszego rozwiązania stopniowo, ale w istocie:

1. Zaprojektuj architekturę tak, aby naruszenie bazy danych nie zapewniało intruzom dostępu do kluczy kryptograficznych.
2. Szyfruj dane pod jednym tajnym kluczem.
3. Utwórz wiele indeksów (z ich własnymi odrębnymi tajnymi kluczami), w oparciu o HMAC lub bezpieczne KDF ze static salt (np. Argon2).
4. Opcjonalnie: obetnij dane wyjściowe z kroku 3, użyj ich jako filtru Bloom.
5. Użyj danych wyjściowych z kroku 3 lub 4 w zapytaniach SELECT.

Na każdym etapie procesu możesz dokonać różnych kompromisów wg tego, co ma sens w twoim przypadku użycia.

Uwierzytelnianie oparte na tokenach bez kanałów bocznych

Szczegółowo: **Split Tokens:** protokoły uwierzytelniania oparte na tokenach bez kanałów bocznych

Mówiąc o bazach danych (poprzednia sekcja), czy wiesz, że zapytania SELECT mogą teoretycznie być źródłem wycieków informacji o timingu (timing information)?

Proste rozwiązanie:

1. Obetnij tokeny autoryzacji o połowę.
2. Użyj jedną część w zapytaniach SELECT.

3. Waliduj drugą połowę w czasie stałym.
    - Możesz opcjonalnie przechowywać hash drugiej połowy w bazie danych zamiast samego tokena. Ma to sens w przypadku tokenów, które zostaną użyte tylko raz; np. tokenów resetu hasła lub "zapamiętaj mnie na tym komputerze".

Nawet jeśli uda Ci się użyć wycieku timingu by wykraść pół tokena, to jego reszta będzie wymagała ataku brute force by atak się powiódł.

Tworzenie bezpiecznych API

In-depth: **Hardening Your PHP-Powered APIs with Sapient**

Napisaliśmy **SAPIENT**, **S**ecure **API EN**gineering **T**oolkit, by przesyłanie uwierzytelnionych wiadomości między serwerami było oczywiste.

Sapient umożliwia szyfrowanie i / lub uwierzytelnianie wiadomości za pomocą kryptografii typu shared-key lub public-key, oprócz zabezpieczeń zapewnianych przez protokół HTTPS.

Umożliwia to uwierzytelnianie żądań i odpowiedzi API za pomocą Ed25519 lub szyfrowanie wiadomości na serwerze docelowym, który może zostać odszyfrowany tylko przez tajny klucz serwera odbiorcy, nawet jeśli atakujący typu man-in-the-middle posiada naruszony Certificate Authority.

Ponieważ każda treść komunikatu HTTP jest uwierzytelniana przez bezpieczną kryptografię, można jej bezpiecznie używać zamiast protokołów tokenów (np. OAuth). Jednakże, jeśli chodzi o kryptografię, zawsze upewniaj się, że ich implementacja jest badana przez ekspertów zanim zrobisz coś niestandardowego.

Cała kryptografia używana przez Sapient jest dostarczana przez bibliotekę kryptograficzną Sodium.

Więcej:

- **Sapient Documentation**
- **Sapient Tutorial**
- **Sapient Specification**

Paragon Initiative Enterprises już używa Sapient w wielu swoich produktach (w tym wielu projektach oprogramowania open source) i będzie nadal dodawać projekty oprogramowania do portfolio użytkowników Sapient.

Logowanie zdarzeń bezpieczeństwa za pomocą Chronicle

Szczegółowo: Chronicle sprawi że zaczniesz kwestionować potrzebę technologii **Blockchain**

Chronicle jest dziennikiem kryptograficznym pozwalającym tylko na zapis, opartym na strukturze danych łańcucha hash (hash-chain), który ma wiele właściwości które przyciągają firmy do technologii “blockchain”, bez bycia zbyt przesadnym.

Poza tymi bardziej kreatywnymi przypadkami użycia dziennika kryptograficznego który pozwala tylko na zapis, Chronicle pokazuje swoje możliwości gdy jest zintegrowane z SIEM, ponieważ możesz wysyłać zdarzenia krytyczne dla bezpieczeństwa do prywatnego Chronicle i nie będzie można ich zmienić.

Jeśli twoje Chronicle jest ustawione tak, by podpisywać swój hash podsumowania do innych instancji Chronicle, i / lub jeśli inne instancje są skonfigurowane tak by replikować zawartość Twojego Chronicle, to utrudni to atakującym manipulację Twoimi logami bezpieczeństwa.

Dzięki Chronicle możesz uzyskać pełną elastyczność, którą obiecują blockchains, bez licznych problemów związanych z prywatnością, wydajnością i skalowalnością.

Aby opublikować dane w lokalnym Chronicle możesz użyć dowolnego API zgodnego z **Sapient**, ale najprostszym rozwiązaniem jest **Quill**.