CEZARY G. CEREKWICKI

Bezpieczne aplikacje internetowe

Stopień trudności

Dla wielu mniej zaawansowanych twórców aplikacji webowych projekt kończy się wraz z wyklikaniem w swoim tworze podstawowych przypadków użycia. W efekcie sieć pełna jest witryn dziurawych jak ser szwajcarski. Znaczne zwiększenie bezpieczeństwa aplikacji webowych jest proste. Trzeba tylko wiedzieć, jak to zrobić.

Większość moich tekstów na łamach hakin9 mieściła się gdzieś na pograniczu pomiędzy teorią i praktyką, najczęściej jednak z lekkim wskazaniem w stronę teorii. Tym razem postanowiłem napisać coś silnie praktycznego. W dalszej części tekstu opisuję kilka prostych sposobów na wzmocnienie bezpieczeństwa aplikacji webowych.

W mojej praktyce zawodowej i hobbystycznej spotkałem się z wieloma aplikacjami, często napisanymi przez duże firmy, w których tematyka bezpieczeństwa była traktowana zdecydowanie raczej mniej niż bardziej poważnie.

Przekonałem się, że nawet najprostsze i – wydawałoby się – najbardziej oczywiste zasady są bardzo powszechnie ignorowane.

Techniki opisane w tym tekście mają charakter dość uniwersalny, mają sens zarówno w klasycznej konfiguracji LAMP (Linux, Apache, MySQL, PHP), jak i we wszystkich innych.

Tekst jest przeznaczony dla autorów i administratorów aplikacji webowych, założyłem więc podstawową znajomość odpowiedniej terminologii.

Minimalna ekspozycja kodu

Każdy serwer WWW (w dalszej części tekstu będziemy się odwoływali do Apache, jako najpopularniejszego narzędzia tego typu) ma określony główny katalog dla plików witryny i to w nim będzie poszukiwał pliku startowego (nazywającego się zwykle *index* lub *default*). Dla naszych potrzeb nazwijmy go *public_html*. Załóżmy, że nasz serwer jest wskazywany przez domenę *www.przyklad.pl* oraz że Apache na tej maszynie ma skonfigurowaną obsługę tej domeny do katalogu */home/user1/public_html*. Zatem zapytanie *www.przyklad.pl/jeden/dwa/plik.html* spowoduje, że Apache będzie próbował otworzyć plik */home/user1/public_html/jeden/dwa/plik.html*.

Wszystko, co leży w katalogu */home/user1/public_html* jest teoretycznie osiągalne przez każdego użytkownika witryny. Teoretycznie, ponieważ jest to zależne od konfiguracji Apache.

Możemy do danych katalogów zabezpieczyć dostęp hasłem, możemy też zakazać dostępu do danego katalogu lub pliku wszystkim użytkownikom łączącym się przez protokół HTTP.

Możemy też dla danych rozszerzeń plików tak skonfigurować Apache, aby ten nie wysyłał ich od razu do przeglądarki, tylko przekazał je do określonego programu i dopiero zwrócony przez niego wynik przesłał do przeglądarki. Według tej koncepcji działają wszystkie techniki stron dynamicznych.

Może się jednak zdarzyć, że, na przykład, Apache nie wyświetli wyniku wykonania skryptu *strona.php*, tylko jego kod źródłowy. Możemy

Z ARTYKUŁU DOWIESZ SIĘ

na jakie sposoby można dość łatwo wydatnie zwiększyć bezpieczeństwo aplikacji webowej?

jak należy modelować uprawnienia w bazach danych?

CO POWINIENEŚ WIEDZIEĆ

wskazana jest praktyczna znajomość choćby podstaw programowania webowego, relacyjnych baz danych oraz administracji serwera webowego.

sobie wyobrazić wiele scenariuszy, w których tak właśnie się dzieje – może to być nieudana próba łatania czy upgrade oprogramowania, przywrócenie plików konfiguracyjnych z backupu, błąd konfiguracyjny, naruszenie integralności konfiguracji przez włamywacza itd. Jest to sytuacja dalece niepożądana, osoba nieupoważniona może w ten sposób wiele się dowiedzieć o witrynie. Może poznać i skopiować jej kod źródłowy, zatem jeśli źródłem przewagi danego serwisu internetowego nad konkurencją jest jakiś oryginalny algorytm, to może on zostać w ten sposób wykradziony. Dostęp do kodu źródłowego aplikacji może być bardzo pomocny w wyszukiwaniu kolejnych podatności serwisu.

Z takiej lektury możemy się dużo dowiedzieć o umiejętnościach jego autora oraz o jego wrażliwości na problematykę bezpieczeństwa. Jeśli na przykład odkryjemy, że oszczędzał sobie czas na precyzyjnym filtrowaniu wejścia, to dalszy kierunek działań jest dość jasny.

Atakujący może dotrzeć do zaszytych w aplikacji komend SQL i na ich podstawie odtworzyć sobie strukturę bazy danych (co może być przydatne w kolejnych atakach). Wreszcie, jeśli uda się w ten sposób podejrzeć plik zawierający nazwę użytkownika i hasło do bazy danych, będzie można się do niej dostać i wykraść lub zniszczyć całą jej zawartość.

Problem ochrony hasła do bazy danych jest generalnie dość trudny. Aplikacja musi się tam dostać, więc hasło musi być dla niej jakoś dostępne. W przypadku aplikacji desktopowej praktycznie zawsze będzie możliwość wykradzenia przechowywanych przez nią haseł, bo nawet jeśli będą przechowywane w postaci zaszyfrowanej, to i tak aplikacja musi gdzieś mieć klucz do tego szyfru.

Zatem zaledwie kwestią czasu jest wykrycie klucza oraz użytego algorytmu szyfrowania, a w konsekwencji – wszystkich chronionych przezeń tajemnic.

W przypadku aplikacji webowej jest odrobinę lepiej, możemy możliwie restrykcyjnie ograniczyć dostęp do aplikacji, dzięki czemu dotarcie do jej kodu będzie wymagało pokonania jakiegoś solidnego mechanizmu kontroli dostępu.

Zalecenie jest więc następujące: określamy, które pliki koniecznie muszą się znaleźć w katalogu *public_html*, a wszystkie inne przenosimy gdzieś poza zasięg Apache. Na pewno do *public_html* muszą trafić pliki z definicji jawne, a więc wszelkie CSS, skrypty JavaScript, wszystkie grafiki, pliki przeznaczone do ściągnięcia itd.

Jeśli chodzi o kod, to w *public_html* wystarczy umieścić plik *index.php* zawierający jedną dyrektywę `include` i nic więcej. W ten sposób nawet jeśli dojdzie do złamania Apache, atakujący najwyżej dowie się, pod jaką ścieżką znajduje się właściwy kod źródłowy, ale za pośrednictwem Apache nie dostanie się do jego treści.

Zasadę możemy streścić tak: do katalogów publicznych wrzucamy tylko to, co bezwzględnie musi się tam znaleźć, wszystkie inne dane umieszczamy w katalogach możliwie prywatnych i w miarę możliwości dostępnych zdalnie jedynie za pomocą protokołów zapewniających solidne uwierzytelnienie i połączenie szyfrowane.

Ta zasada, pomimo prostoty jej realizacji, nie jest powszechnie stosowana, choć zalecają ją liczne książki i publikacje z zakresu bezpieczeństwa informatycznego. Wiele popularnych aplikacji PHP realizujących CMSy, fora, sklepy internetowe, blogi wszystkie swoje pliki trzyma w jednym miejscu, bez podziału na pliki publiczne i poufne.

Kod tym łatwiej jest schować, im mniej ma on punktów wejścia. Przez punkt wejścia rozumiem każdy skrypt podawany jawnie w URLu. W przypadku aplikacji zbudowanych w oparciu o architekturalny wzorzec projektowy Model View Controller (MVC), sprawa jest prosta. Tu sterowaniem zajmuje się kontroler, będący właśnie punktem wejścia (zazwyczaj jedynym). Jeśli w używanej przez nas aplikacji URLe zawsze mają końcówkę typu *index.php?strona=coś tam&atrybut=coś tam*, to zapewne mamy do czynienia z takim właśnie przypadkiem.

W takiej sytuacji możemy także wzmocnić bezpieczeństwo programu, którego sami nie napisaliśmy.

Przenosimy wszystkie jego niepubliczne pliki do jakiegoś bezpieczniejszego katalogu, po czym przekierowujemy tam PHP odpowiednią dyrektywą `include`.

Projektując nową witrynę, warto przewidzieć tylko jeden punkt wejścia, nawet jeśli nie zamierzamy korzystać ze wzorca MVC.

Oprócz już opisanej korzyści, będziemy ich mieli dużo więcej, m.in. łatwość zagwarantowania, że dany kod inicjalizujący na pewno zawsze się wykona (zakładając właściwą obsługę wyjątków), czy też ogólnie większy porządek w kodzie.

Nadpisywanie danych wrażliwych

Jak długo aplikacja webowa musi pamiętać hasło do bazy danych? Zaraz po stworzeniu połączenia bazodanowego przestaje być ono potrzebne. Połączenie takie będziemy raczej robić tyko raz, bo przechowa je odpowiedni singleton (albo, w przypadku nieobiektowych aplikacji, jakiś inny twór, np. zmienna globalna). Zatem czy dalsze przechowywanie treści hasła ma jakiekolwiek pozytywy?

Gdybyśmy naprawdę go potrzebowali – co jednak wydaje się być sytuacją mało prawdopodobną i możliwą do uniknięcia – to można je ponownie pozyskać (poprzez ponowne wykonanie skryptu inicjującego odpowiednią zmienną, czy tworzącą odpowiedni obiekt).

Pozytywów nie ma, jakie są zatem negatywy? Każde miejsce występowania hasła to miejsce, z którego można je wykraść. Obecność hasła w pamięci daje napastnikowi szereg potencjalnych możliwości jego odczytania, np. na skutek udanego code injection.

Zatem zaraz po użyciu każda wrażliwa zmienna powinna być nadpisywana jakąś wartością bezpieczną (np. ciągiem pseudolosowym). Ważne jest, żeby nadpisywać zmienne, a nie tylko je zwalniać, ponieważ zwolnienie nie gwarantuje, że treści zmiennej nie uda się już odczytać. Warto zapoznać się z mechanizmem przydziału pamięci, z

którego korzystamy. Jeśli jest to *garbage collector*, to jakim algorytmem alokacji się posługuje?

A naprawdę kluczowym pytaniem jest: czy nadanie nowej wartości zmiennej gwarantuje, że stara wartość nigdzie nie pozostanie i nie będzie można jej odczytać?

Listing 1. *Przykładowa klasa PHP służąca do pierwszego etapu ogólnej walidacji ciągów znaków i liczb*

```
class Validator
{
   function ValidString($str, $minlen, $maxlen=null, $ereg=null, $casesensitive=false)
   {
      if(!is_string($str))
         return false;

      if(($maxlen != null) && (ln_strlength($str) > $maxlen))
         return false;

      if(ln_strlength($str) < $minlen)
         return false;

      if($ereg != null)    // check if $str matches regular expression
         if($casesensitive)
         {
            if(!ereg($ereg, $str))
               return false;
         }else
            if(!eregi($ereg, $str))
               return false;

      return true;
   }

   function ValidInt($i, $minint=null, $maxint=null, $nonzero=null,
                     $naturalnumber=null)
   {
      $int = (int) $i;

      if(!ctype_digit($i))
         return false;

      if(!is_int($int))
         return false;

      if(($minint != null) && ($int < $minint))
         return false;

      if(($maxint != null) && ($int > $maxint))
         return false;

      if(($nonzero != null) && ($int == 0))
         return false;

      if(($naturalnumber != null) && ($int < 1))
         return false;

      return true;
   }

   function ValidEMail($email)
   {
      if(!eregi("^[_a-z0-9-]+(\.[_a-z0-9-]+)*@[a-z0-9-]+(\.[a-z0-9-]+)*(\.[a-
                     z]{2,3})$", $email))
         return false;

      return true;
   }

}
```

W przypadku zapisu plików na dysku nie jest to takie proste i podobnej gwarancji mieć nie możemy. Dlatego programy realizujące bezpieczne kasowanie zapisują dane pseudolosowe wielokrotnie, aby mieć pewność, że pomimo kaprysów algorytmu alokacji doprowadzi to do rzeczywistego nadpisania wszystkich starych danych. Tego typu programy mają też często opcję nadpisania tej części dysku, która nie jest obecnie wykorzystywana przez żadne pliki. Pozwala to na zniszczenie danych plików skasowanych w sposób klasyczny, jak również jest naturalnym uzupełnieniem kasowania kilkukrotnego – zachowane części plików powinny być na skutek takiej operacji całkowicie zniszczone.

Użycie ciągu pseudolosowego może mieć znaczenie w przypadku, gdy próba odzyskania danych doprowadziła do uzyskania fragmentarycznych danych oryginalnych, częściowo nadpisanych danymi pseudolosowymi. Odfiltrowanie ich będzie trudniejsze niż w przypadku ciągu zer lub innego, łatwo przewidywalnego, wzorca.

Zasada brzmi następująco: wszelkie dane wrażliwe należy skutecznie zniszczyć, kiedy tylko przestaną być potrzebne, aby nikt nie był w stanie już ich odczytać.

Centralizacja danych wrażliwych

Aplikacja webowa może zawierać wiele różnych danych wrażliwych. Są to loginy i hasła wykorzystywanych kont bazodanowych, login i hasło użytkownika SMTP, jeśli aplikacja rozsyła maile, klucze prywatne, jeśli aplikacja używa kryptografii klucza publicznego itd.

Zasada jest następująca: wszystkie wrażliwe dane należy trzymać w jednym miejscu, np. w wydzielonym katalogu i nigdzie więcej. Każdą wykonaną chwilowo kopię należy stworzyć jak najpóźniej przed wykorzystaniem i zniszczyć jak najwcześniej po wykorzystaniu.

Dzięki takiej organizacji łatwiej będzie zarządzać dostępem do tych danych. Na przykład jeśli nazwiemy sobie ten

katalog *sejf*, to odnalezienie wszystkich plików kodu źródłowego odwołujących się do tego katalogu sprowadzi się do odpowiedniego ich *przegrepowania*. Taką czynność trzeba będzie wykonać podczas audytu bezpieczeństwa kodu – warto przyjrzeć się, w których miejscach nasze tajne pliki są wywoływane. Naturalnie liczba tego typu odwołań powinna być minimalna. Jeśli dane wrażliwe sensownie podzielimy na pliki, to będziemy mogli je dość rozsądnie wczytywać tuż przed tym, gdy są potrzebne i nadpisywać je tuż po tym. W ten sposób czas ich duplikacji w pamięci serwera będzie minimalny, co ograniczy możliwości dobrania się do nich przez osoby nieupoważnione.

Kilkukrotnie widziałem już aplikacje, w których dane wrażliwe były dość przypadkowo i niekonsekwentnie rozrzucone po kodzie źródłowym, przez co zarządzanie nimi było bardzo trudne. Nawet tak banalna i pożądana operacja, jak zmiana hasła, prowadziła do gorączkowego szukania wszystkich miejsc, w których hasło zapisano, a potem testowania z drżącym sercem, czy aby na pewno gdzieś się jakiś zaszyfrowany *connection string* nie zachował.

Zdecydowanie łatwiej jest utrzymać poufność i integralność cennych danych w jednym miejscu, a operacje zmiany wszystkich haseł bazodanowych, kluczy prywatnych czy jakichkolwiek innych kluczowych danych wrażliwych powinny być typowym przypadkiem użycia, testowanym na wszystkich etapach rozwoju oprogramowania.

Minimalne przywileje użytkowników bazodanowych

Wielu mniej zaawansowanych programistów tworzy jednego użytkownika bazodanowego, który ma wszystkie uprawnienia włącznie z DDL (może więc praktycznie wszystko, z DROP DATABASE włącznie). Nie zawsze jest to wina mało świadomego autora. Instalując własną aplikację webową, jesteśmy często mocno ograniczeni przez wykorzystywaną przez nas usługę hostingową.

W przypadku najgorszych paneli dostępu do baz danych mamy możliwość zaledwie stworzenia użytkownika, któremu automatycznie nadawane są wszystkie przywileje dla konkretnej bazy danych. W takiej sytuacji, poza poszukaniem na rynku lepszej oferty, nasze możliwości są bardzo ograniczone.

Jeśli mamy w nadmiarze dostępnych baz danych, możemy to wykorzystać do wprowadzenia jakiegoś mechanizmu *obrony wgłąb*. W zależności od tego, z jaką aplikacją mamy do czynienia, możemy stworzyć dla niej kilka baz danych i jakoś sensownie podzielić pomiędzy nie dane.

Na przykład, jeśli mamy aplikację, w której niewiele da się zrobić bez zalogowania się, to tabelę z kontami użytkowników oraz ich hasłami można wyodrębnić do osobnej bazy. Wówczas kod uwierzytelniający miałby swojego użytkownika bazodanowego z odpowiednim hasłem. Ten kod podlegałby bardzo rygorystycznym testom bezpieczeństwa, aby zminimalizować prawdopodobieństwo, że jest w nim błąd pozwalający na odczytanie kryptogramów haseł, zniszczenie kont itd. Kod uwierzytelniający raczej rzadko podlega zmianom, więc ryzyko wprowadzenia do niego błędu na skutek jakiejś nieprzemyślanej poprawki również byłoby niewielkie.

Takie wyodrębnienie spowodowałoby, że błędy w innych częściach kodu (korzystających już z innego użytkownika bazodanowego), nawet gdyby doprowadziły do kompromitacji loginu i hasła, to dawałyby dostęp napastnikowi tylko do bazy zawierającej treść witryny, a loginy, hasła, e-maile i inne dane użytkowników witryny byłyby ciągle bezpieczne.

W przypadku usług hostingowych o lepszej funkcjonalności zarządzania uprawnieniami mamy dużo szerszą gamę możliwości realizacji zasady minimalnego dostępu. Zarysowany wyżej pomysł można wprowadzić w życie bez tworzenia nadmiarowych baz. Wystarczy stworzyć dedykowanego użytkownika, dać mu prawa SELECT oraz wszystkie DML (INSERT, UPDATE, DELETE) na tabeli z użytkownikami oraz dopilnować, aby żaden inny użytkownik nie miał jakichkolwiek uprawnień do tej samej tabeli. Naturalnie, jeśli aplikacja nie przewiduje – dajmy na to – usuwania kont użytkowników, to nikt nie powinien mieć uprawnienia DELETE na tej tabeli. Tak również ograniczamy potencjalne skutki przejęcia loginu oraz hasła.

Jeżeli nasza aplikacja loguje dane, to żaden użytkownik dostępny dla aplikacji nie powinien mieć szerszych uprawnień niż INSERT i SELECT na tabeli przechowującej logi. W ten sposób nikt logów nie zmieni ani nie skasuje, choćby nawet przejął wszystkie loginy i hasła znane aplikacji.

Jeśli aplikacja jest jakiegoś rodzaju CMSem, np. blogiem, to mamy niewielu użytkowników tworzących treść i wielu tylko czytających. Aż się prosi, aby w takiej sytuacji wyodrębnić dwóch użytkowników bazodanowych: jednego dla czytelników, mającego uprawnienia SELECT do wszystkich potrzebnych sobie tabel i np. INSERT do tabeli z komentarzami oraz drugiego dla redaktorów serwisu. Dzięki takiej konfiguracji potencjalne SQL Injection czy wykradzenie loginu i hasła przy użyciu stron dostępnych dla wszystkich będzie miało minimalne skutki dla bazy danych.

Włamywacz co najwyżej będzie mógł przeczytać artykuł oraz dodać nowy komentarz (co i tak może). Nie będzie w stanie zmienić cudzego komentarza czy artykułu, nie doda ani nie skasuje artykułu, nie usunie też komentarza. Użytkownik przeznaczony dla redaktorów mógłby już więcej, ale też nie miałby zbędnych uprawnień, w szczególności DDL (CREATE, ALTER, DROP, TRUNCATE) ani DCL (GRANT, REVOKE), ale też i nadmiarowych DML (np. modyfikacji komentarzy, jeśli logika biznesowa ich nie przewiduje).

Zasady konfiguracji uprawnień w bazie danych możemy opisać następującymi regułami:

- Przeczytaj dokumentację używanej bazy danych i zastosuj wszystkie zawarte tam rekomendacje.
- Stwórz użytkownika administracyjnego dla własnych potrzeb (mogącego wszystko) i pod żadnym pozorem jego danych nie

umieszczaj gdziekolwiek w aplikacji! Ten użytkownik ma służyć wyłącznie do czynności wykonywanych ręcznie, w szczególności czynności developerskich (np. zmiany struktury bazy danych) oraz administracyjnych (np. przywrócenie stanu bazy z dumpa). Zmieniaj mu systematycznie hasło.

- Dla każdej tabeli określ, jakie operacje są na niej w ogóle potrzebne. Jeśli logika biznesowa nie przewiduje, powiedzmy, dodawania nowych wpisów słownikowych, to żaden z przewidzianych dla aplikacji użytkowników bazodanowych nie powinien mieć uprawnienia INSERT na tej tabeli.
- Zaplanuj użytkowników bazodanowych i przypisz im minimalne niezbędne uprawnienia. Suma uprawnień tych użytkowników powinna być równa sumie uprawnień z punktu 3.

Walidacja wejścia

Wszystkie poprzednie reguły mówiły raczej o metodach minimalizacji skutków potencjalnego ataku niż o bezpośrednich metodach obrony.

Walidacja wejścia to bodaj najbardziej bezpośrednia z możliwych form obrony.

Wejście należy rozumieć bardzo szeroko. Dla każdego jest oczywiste, że wejściem jest to, co użytkownik wpisze w formularzu. Już mniej oczywiste jest, że wejściem jest też np. plik konfiguracyjny albo zmienne sesyjne. Najbardziej defensywnie napisane programy weryfikują nawet integralność swojego kodu (choćby przez policzenie sumy kontrolnej CRC i porównanie go z wcześniej zapisaną wielkością). Analogicznie można badać integralność innych danych, których zmienianie nie jest dopuszczalne w trakcie eksploatacji aplikacji. Pełną ścieżkę walidacyjną powinny przechodzić zmienne systemowe oraz zmienne sesyjne (i te z cookies, i te PHP-owe).

W kluczowych algorytmach warto wyróżnić niezmienniki i kontrolować ich spełnienie asercjami. Niezmiennik to reguła opisująca określoną relację między zmiennymi, która zawsze musi być spełniona.

Jeśli na skutek jakiegoś błędu bądź ingerencji napastnika dojdzie do naruszenia niezmiennika, aplikacja zachowa się w sposób niepoprawny, np. dany warunek wyjścia z pętli może nigdy nie zostać spełniony. Naruszenie niezmiennika jest błędem krytycznym, uniemożliwiającym poprawne działanie aplikacji, zatem wykrycie takiej sytuacji powinno prowadzić do wykonania jakiejś procedury awaryjnej, ograniczającej w miarę możliwości skutki awarii.

Najbardziej ogólne minimalne reguły walidacji można streścić następująco:

- wszystkie ciągi znaków i liczby muszą być kontrolowane pod kątem maksymalnej długości,
- wszystkie ciągi znaków, które będą doklejone do renderowanego kodu XHTML, muszą być wolne od tagów (zwłaszcza tych pozwalających wykonywać wskazane lub podane explicite skrypty),
- wszystkie ciągi znaków, które będą doklejane do komend SQL, muszą być wolne od znaków specjalnych (zwłaszcza wszelkich cudzysłowów),
- wszystkie ciągi znaków, które będą doklejane do komend powłoki, muszą być wolne od znaków specjalnych.

Listing 2. *Funkcja zabezpieczająca string mający trafić do zapytania do MySQL*

```
function QuoteSmart($value, $conn)
{
   if (get_magic_quotes_gpc()) {

      $value = stripslashes($value);
   }

   if (!is_numeric($value)) {

      $value = "'" . mysql_real_escape_string($value, $conn) . "'";
   }

   return $value;
}
```

Na Listingu 1. przedstawiono przykładową klasę PHP przeznaczoną do przeprowadzenia pierwszego etapu walidacji, a więc sprawdzenia minimalnych i maksymalnych długości, zgodności z zadanym wyrażeniem regularnym, a dla liczb – testu, czy w ogóle jest to liczba, zgodności jej wielkości z zadanymi ekstremami, zerowości i naturalności.

W drugim etapie, w zależności od przyjętych reguł, można niepoprawne ciągi znaków skonwertować do poprawnych (np. znak „<” na `<`, znak cudzysłowu na cudzysłów zacytowany itd.) albo odrzucić jako nieprawidłowe.

Jeśli tworzymy aplikację o dużych potrzebach z zakresu bezpieczeństwa, próby wprowadzenia podejrzanych wartości możemy logować (kto, gdzie, kiedy, z jakiego IP i co próbował wpisać). Kolejnym krokiem może być wprowadzenie mechanizmu zliczającego takie próby i podejmującego określoną akcję w przypadku przekroczenia zadanego maksimum, np. wysłanie e-maila do administratora czy zablokowanie dostępu z tego numeru IP.

Podsumowanie

W tekście przedstawiono kilka dość elementarnych reguł hardeningu aplikacji webowych, z uwzględnieniem najpopularniejszych narzędzi wykorzystywanych w tej dziedzinie. Opisane tu techniki, gdyby je poprawnie zaimplementować, neutralizują liczne metody ataku na aplikacje webowe na kilku poziomach: prewencji i profilaktyki oraz minimalizacji skutków udanych ataków cząstkowych.

Cezary G. Cerekwicki
Autor jest z wykształcenia informatykiem i politologiem. Pracował jako projektant, programista, administrator, konsultant, tłumacz, koordynator międzynarodowych projektów, dziennikarz i publicysta.
Pisał programy w dziesięciu językach programowania (od asemblerów po języki skryptowe), w czterech systemach operacyjnych, na dwóch platformach sprzętowych.
Kontakt z autorem: *cerekwicki@tlen.pl*