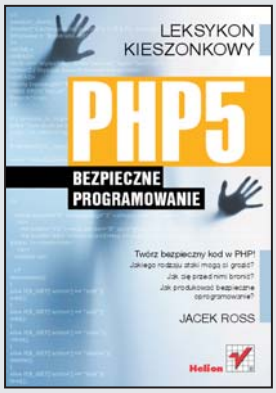
PHP5. Bezpieczne programowanie. Leksykon kieszonkowy



Autor: Jacek Ross
ISBN: 83-246-0635-1
Format: 115x170, stron: 160

Twórz bezpieczny kod w PHP!

- Jakie rodzaje ataków mogą Ci zagrozić?
- Jak się przed nimi bronić?
- Jak produkować bezpieczne oprogramowanie?

PHP jest z pewnością jednym z najbardziej popularnych języków programowania, pozwalających na tworzenie dynamicznych aplikacji WWW. Swoją popularność zdobył dzięki prostej składni, łatwej konfiguracji oraz przejrzystym zasadom działania. PHP jest świetnym przykładem na to, że prostota i elegancja bywają lepsze niż nadmierne zaawansowanie i niepotrzebna komplikacja. Pomimo swej prostoty język PHP jest bardzo wymagający w sprawach związanych z bezpieczeństwem. Zmusza on programistę do poświęcenia niezwykłej uwagi kwestii wyboru bezpiecznych rozwiązań.

Z pewnością brakowało Ci książki, która w jednym miejscu gromadziłaby wszelkie informacje związane z bezpieczeństwem w PHP. Dzięki pozycji „PHP5. Bezpieczne programowanie. Leksykon kieszonkowy ” poznasz podstawy bezpiecznego programowania, sposoby obsługi danych pobranych z zewnątrz oraz przekazywania ich pomiędzy skryptami. Autor przedstawi Ci rodzaje ataków na aplikacje PHP oraz najlepsze metody obrony przed nimi. Ponadto nauczysz się we właściwy sposób konfigurować PHP oraz zdobędziesz wiedzę na temat zasad bezpiecznej produkcji oprogramowania. Jeżeli chcesz tworzyć bezpieczne rozwiązania w PHP, koniecznie zapoznaj się z tą książką!

- Obsługa danych zewnętrznych
- Wstrzykiwanie kodu
- Dobór odpowiednich uprawnień
- Sposoby uwierzytelniania użytkownika
- Bezpieczne obsługiwanie błędów
- Rodzaje ataków na aplikacje napisane w PHP
- Obrona przed atakami XSS
- Zagrożenie wstrzyknięciem kodu SQL
- Ataki DOS i DDOS
- Bezpieczna konfiguracja PHP
- Sposoby tworzenia bezpiecznego oprogramowania

Wykorzystaj możliwości PHP w pełni i bezpiecznie!

Helion SA
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl

Spis treści

5. Metody produkcji bezpiecznego oprogramowania

5.1. Architektura programu a bezpieczeństwo

Architektura programu może mieć istotny wpływ na poziom jego bezpieczeństwa. Nie ma zbyt wielu ogólnych reguł dotyczących tego, jak prawidłowo powinna być zaprojektowana aplikacja sieciowa, wiele zależy bowiem od: użytych technologii, przyjętej metodologii projektowej, rozmiaru projektu i zespołu, oprogramowania używanego podczas tworzenia aplikacji, a także od samego jej rodzaju i wszystkich szczegółów jej działania. Istnieje jednak kilka zasad, o których powinien pamiętać programista i projektant:

- Prostota. Trawestując Einsteina, można by powiedzieć, że kod powinien być tak prosty, jak to możliwe, ale nie bardziej. W prostym, eleganckim i przejrzystym kodzie znajdzie się prawdopodobnie znacznie mniej błędów niż w zawiłym, pełnym nadmiarowości i tricków programistycznych. Kod krótszy nie zawsze jest prostszy i bardziej przejrzysty. Warto czasem napisać go więcej, lecz czytelniej — w sposób bardziej zrozumiały. Może on zostać potem łatwiej przeanalizowany przez innego programistę, który, jeśli znajdzie w nim błędy, będzie mógł zasugerować poprawki. Jest to szczególnie istotne przy programach typu open source.
- Kontrola jakości. W przypadku żadnej większej aplikacji nie jest dobrym rozwiązaniem przerzucanie kontroli jakości na programistów czy użytkowników. Błędy wykryte przez tych ostatnich są kosztowne w naprawie, pogarszają wizerunek programu, a w przypadku gdy dotyczą zabezpieczeń, mogą stanowić przyczynę dużej liczby udanych ataków na

aplikację (nie każdy użytkownik zgłosi błąd producentowi — niektórzy mogą postanowić wykorzystać go do własnych celów). Programiści z kolei nie są zazwyczaj w stanie wykryć wielu nieprawidłowości ze względu na brak obiektywizmu względem własnego kodu i problem z dostrzeżeniem tych błędów, które umknęły ich uwadze na etapie implementacji. Warto więc podzielić testowanie na etapy, uściślić jego procedury i scenariusze oraz skorzystać z takich technik, jak testy jednostkowe (tworzone przez programistów), automatyczne, funkcjonalne (ręczne), bezpieczeństwa czy też testy obciążenia (które mogą także mieć wpływ na bezpieczeństwo, minimalizując ryzyko ataków typu DOS i DDOS).

- Skupienie kluczowych elementów aplikacji. Jeśli nasz program może mieć następujące wywołania: `login.php?user` `↪=54`, `basket.php?what=add&pid=10`, `product.php?cat=17&` `↪prod=12` i `details.php?uid=3&mode=1`, to poprawne chronienie go może stać się sporym wyzwaniem. Dlaczego? Ponieważ istnieje wiele punktów wejścia do niego i każdy z nich musi zostać niezależnie zabezpieczony. Wprawdzie możemy procedury zabezpieczeń wydzielić do osobnego pliku czy klasy i wywoływać je w skryptach, lecz będziemy wtedy musieli pamiętać o tym, aby robić to prawidłowo w każdym z tych miejsc, a gdy dodamy nowy plik, jego także będziemy musieli zabezpieczyć. W dodatku każdy ze skryptów przyjmuje zupełnie inne parametry. W takim gąszczu łatwo o błąd, a jeden źle zabezpieczony skrypt może wystarczyć do tego, aby cała aplikacja przestała być bezpieczna. Lepiej zrobić jeden punkt wejścia do aplikacji, sterując jej przebiegiem poprzez parametr, i zminimalizować liczbę pozostałych parametrów. Powyższe odwołania mogą przyjąć postać: `index.php?what=login&uid=54`, `index.php?` `↪what=basket_add&pid=10`, `index.php?what=product&cat=` `↪17&prod=12` i `index.php?what=details&uid=3&mode=1`.

- Zaprojektowanie rozmieszczenia elementów składowych aplikacji, takich jak baza danych, system prezentacji itp. Zastanówmy się, na jakich maszynach będą one umieszczone oraz jakie będą tego konsekwencje. Zaplanujmy też rozmieszczenie plików na serwerze. Rozdzielmy koniecznie różne warstwy i podsystemy aplikacji. Niech prezentacja nie będzie zmieszana z warstwą logiki czy z danymi. Planując takie rzeczy, nie tylko unikniemy chaosu i uzyskamy przejrzyste wewnętrznie oprogramowanie, ale też zwiększymy poziom jego bezpieczeństwa. Dla przykładu, umieszczenie skryptów w tym samym katalogu, w którym znajdują się dane przesyłane na serwer w wyniku akcji użytkownika, niesie ze sobą potencjalne zagrożenie.

5.2. Ochrona przez ukrycie informacji (security by obscurity)

Nie możemy liczyć na to, że użytkownik nie wie, jak działa nasz skrypt, nie zna parametrów wywołania, nazw pól formularzy (w tym pól ukrytych), wartości identyfikatorów itp. Tego typu informacje są bardzo łatwe do odkrycia. Często wystarczy kilkanaście minut eksperymentów z działającą aplikacją, aby dowiedzieć się wszystkiego, co jest potrzebne do ataku. W ostateczności mający cierpliwość włamywacz dokona na te informacje ataku siłowego, czyli zgadnie je metodą prób i błędów. Podobno pierwsze prawo Murphy'ego dotyczące ochrony programów głosi, że ilość czasu, jaką włamywacz może poświęcić na łamanie zabezpieczeń, jest zawsze dostatecznie duża i w razie potrzeby dąży do nieskończoności. Stąd też wywołanie typu `index.php?o=sjka&` `↪i=8271&t=981&a=aabf1a`, nawet jeśli trudno w to uwierzyć, nie musi być wynikiem pracy aplikacji, lecz może zostać wprowadzone do przeglądarki celowo i niewykluczone, że w złych intencjach. Nawet jeśli źródła programu są dobrze ukryte, to włamywacz

może zgadnąć czy też w inny sposób odkryć, że przez wywołanie `o=sjka` przeprowadzamy zmianę hasła administratora lub wykonujemy inną istotną funkcję, a wtedy grozi nam katastrofa.

Nie chciałbym zostać źle zrozumiany. Ukrywanie przed użytkownikiem informacji, które go nie dotyczą, jest dobrą praktyką. Jeśli algorytmy, błędy wewnętrzne, parametry, użyte funkcje systemowe itp. będą niedostępne dla niepowołanych oczu, to zmniejszy się nieco ryzyko odkrycia przez włamywacza dziur w zabezpieczeniach. W końcu każda dodatkowa ochrona jest dobra — nawet taka. Jest to jednak zabezpieczenie słabe i lepiej, żeby tych dziur w kodzie nie było, niż żebyśmy musieli polegać na ich dobrym maskowaniu. Szczególnie jeśli tworzymy oprogramowanie typu open source, musimy być pewni na 100%, że ten punkt nas nie dotyczy. W otwartych źródłach nie ma sensu niczego ukrywać, kodować czy zaciemniać. Błędy takiego oprogramowania prędzej czy później i tak wyjdą na jaw. Każdy może swobodnie analizować jego kod, a gdy jeden użytkownik po wykryciu w nim dziury wykorzysta tę wiedzę by nam zaszkodzić, to drugi prześle nam informację o znalezionych nieprawidłowościach, abyśmy mogli je naprawić (a może nawet od razu dostarczy nam gotową poprawkę). Przy otwartych źródłach rozsądna jest więc strategia zupełnie przeciwna niż security by obscurity: należy pisać program jak najbardziej przejrzysty, czytelny i udokumentowany, tak aby kod źródłowy był doskonale zrozumiały nawet dla początkującego programisty. Dzięki takiemu podejściu więcej osób ma szansę zapoznać się z nim i, co za tym idzie, istnieje większa szansa na to, że ktoś odkryje błąd i podzieli się tym z autorem. Warto jednak pamiętać, że nawet w przypadku publikacji źródeł jako open source jawność nie dotyczy konfiguracji serwera. Ukrycie informacji o nim, o wersji zainstalowanego na nim oprogramowania, komunikatów o błędach czy innych tego typu istotnych danych jest zawsze korzystne. Swobodny dostęp do nich prowokuje bowiem włamywaczy i zwiększa szansę na udany atak.

5.3. Pozostawianie „tylnych wejść" i kodu tymczasowego

Programiści dość często zostawiają w swoim kodzie rozmaite „tylne wejścia": funkcje diagnostyczne, narzędziowe, testowe, tymczasowe (te w informatyce mają nieraz zaskakująco długie życie) i wiele innych fragmentów oprogramowania, które nie powinny być używane i udostępniane publicznie. Najczęściej liczą na to, że nikt nie zgadnie, gdzie znajduje się takie „tylne wejście" i jak należy go użyć. Włamywacze dysponują jednak zazwyczaj dużą ilością czasu, a coraz częściej także dużymi zasobami finansowymi (szczególnie ci działający na zlecenie grup przestępczych) i mają wiele sposobów na odkrycie słabych punktów kodu:

- metody siłowe, czyli odgadnięcie dogodnego sposobu włamania lub włamanie poprzez odgadnięcie hasła czy innej tajnej informacji;
- przekupienie pracowników firmy i zdobycie tą drogą informacji;
- włamanie do sieci firmowej lub na prywatne bądź służbowe komputery pracowników i odszukanie istotnych, a źle zabezpieczonych informacji (wewnątrz intranetów firmowych są one zazwyczaj słabo chronione);
- wykorzystanie robotów do automatyzacji prób włamań;
- wykorzystanie znanych dziur w bibliotekach używanych przez oprogramowanie;
- rozpoznanie metod działania programu poprzez analizę jego zachowania.

Programiści często dodają tylne wejścia do aplikacji po to, aby ułatwić sobie ich testowanie i następujące po nim usuwanie błędów. Tego typu działanie świadczy jednak o złej organizacji

procesu produkcji oprogramowania, o braku przemyślanych procedur czy też o niewłaściwym lub nieistniejącym przepływie zgłoszeń nieprawidłowości. Warto przemyśleć, czy opłaca się iść na skróty i zostawiać otwartą tylną furtkę, gdy poświęca się długie godziny na zabezpieczenie głównych drzwi.

5.4. Aktualizowanie wersji PHP i używanych bibliotek

Jedną z częstszych metod włamań do aplikacji sieciowych jest wykorzystanie istniejących dziur bądź to w oprogramowaniu serwera lub interpretera, bądź to w samej konstrukcji języka. Wiele starszych wersji PHP miało istotne luki, w tym zarówno takie, które umożliwiały programiście stworzenie złego, niebezpiecznego kodu, jak i takie, które same w sobie mogły zostać wykorzystane przez włamywacza. Kolejne wydania zawierają wiele poprawek eliminujących możliwości wykonywania takich ataków, dlatego bardzo ważne jest używanie jak najnowszej wersji języka. Jeśli sam go nie aktualizujesz — sprawdzaj co pewien czas, czy robi to administrator. Co prawda nie ma nigdy pewności, że aktualizacje te nie wprowadzają nowych dziur (cóż, nikt nie jest doskonały, twórcy PHP również), jednak korzystanie z najnowszej, stabilnej wersji języka PHP zazwyczaj per saldo i tak się opłaca.

- Dziury istniejące w starszych wersjach są zazwyczaj dobrze znane, a im więcej osób zna słabe punkty Twojego oprogramowania, tym większa jest szansa na skuteczny atak. Co więcej: istnieją specjalne programy, które aktywnie przeszukują Internet pod kątem stron działających na przestarzałych wersjach oprogramowania serwerowego i tym samym podatnych na atak. Po znalezieniu takiej strony przesyłają one raport do swojego właściciela, który może tę informację wykorzystać do najróżniejszych złych celów. Dlatego można uznać za prawdziwe stwierdzenie, że im dziura w oprogra-

mowaniu jest starsza, tym groźniejsza (jej istnienie przynosi też większy wstyd właścicielowi oprogramowania, który przez tak długi okres czasu nie zdołał jej usunąć).

- Nowsze wersje PHP często eliminują niebezpieczne konstrukcje samego języka, które mogą bardzo łatwo skutkować włamaniem. Co prawda ma to swoje wady: starsze programy mogą wymagać, i często wymagają, zmian, lecz podjęty wysiłek opłaca się — otrzymamy w wyniku bezpieczniejszą aplikację. Niektóre funkcje są w nowych wydaniach oznaczane jako wycofywane (ang. *deprecated*). Rozumie się przez to, że mogą one zostać usunięte w kolejnych wersjach oprogramowania. Warto wcześniej pomyśleć o pozbyciu się ich, bo później może być na to mniej czasu, co zmusi nas do niepotrzebnego pośpiechu i w efekcie do popełnienia błędów. Status `deprecated` posiada obecnie np. opcja konfiguracyjna `register_global`. Twórcy PHP planują usunięcie jej w wersji 6. Jeśli z niej korzystasz, wyeliminuj ją już teraz!

Pamiętaj, że odświeżanie oprogramowania dotyczy także bibliotek i gotowego kodu, z którego korzystasz. Sprawdzaj regularnie, czy ich autor wykonał aktualizację. Jeśli projekt jest martwy, a autor (autorzy) nie poprawiają kodu, to nie masz wyjścia: albo będziesz regularnie przeglądał źródła i wprowadzał samodzielnie poprawki, albo powinieneś poszukać innej biblioteki. Pozostawianie w swoim programie przestarzałego kodu jest zbyt niebezpieczne i grozi poważnymi konsekwencjami.

5.5. Użycie gotowych bibliotek i frameworków

Używanie gotowych frameworków i bibliotek ma zarówno wady, jak i zalety, w tym takie, które w znaczący sposób wpływają na bezpieczeństwo aplikacji sieciowej. Programista powinien sam rozważyć, co jest dla niego bardziej opłacalne. Oto krótkie zestawienie plusów i minusów korzystania z gotowego kodu z punktu

widzenia ochrony programu (pomijam więc kwestie takie jak oszczędność czasu, użycie sprawdzonych standardów kodowania itp.).

ZA:

- Najpopularniejsze biblioteki zostały stworzone przez najlepszych programistów, mających duże doświadczenie w programowaniu w języku PHP, dzięki czemu prawdopodobieństwo, że zawierają istotne, grube błędy, jest znacznie mniejsze niż w przypadku własnoręcznie napisanego kodu. Nawet jeśli jesteśmy geniuszami, to nie mamy pewności, że posiadamy wszystkie informacje, które były znane autorom podczas pisania bibliotek (np. zgłoszone przez użytkowników błędy w starszych wersjach czy specjalistyczna dokumentacja). Bez takiej wiedzy nawet najlepszy programista może popełnić błąd.

- Popularny kod najczęściej jest testowany przez tysiące użytkowników, więc istnieje spora szansa, że wiele błędów, które my możemy dopiero popełnić, zostało już popełnionych, zauważonych i poprawionych. Szczególnie znacząca powinna być informacja o istnieniu silnej i licznej społeczności skupionej wokół oprogramowania. Jeśli wiele osób używa biblioteki, dyskutuje o niej, zgłasza nieprawidłowości i przesyła autorom poprawki lub modyfikacje, jest to znak, że pojawienie się ewentualnej dziury może zostać szybko zauważone i natychmiast powstanie łatka zażegnująca niebezpieczeństwo. Bogata i aktywna społeczność to najczęściej gwarancja częstych aktualizacji i wysokiego standardu.

- Wiele z bibliotek zostało sprawdzonych przez twórców PHP i zaakceptowanych jako pewna podstawa. Niektóre z nich w kolejnych wersjach języka stanowią standardowy moduł,

a ich stosowanie jest zalecane przez podręcznik użytkownika (*http://php.net*). Do takich bibliotek należy mieć większe zaufanie niż do kodu zewnętrznego i raczej nie powinniśmy mieć oporów przed korzystaniem z nich. Przykładem może być tu np. biblioteka PDO.

PRZECIW:

- Im popularniejsza biblioteka, tym większa liczba potencjalnych włamywaczy będzie zaznajomiona z nią i z jej dziurami. W pierwszej kolejności próbują oni znaleźć metody włamań do typowego kodu, korzystającego z typowych bibliotek, ponieważ mają największą szansę na znalezienie takich aplikacji w sieci. Oryginalny, napisany po raz pierwszy kod może ich zaskoczyć. Nie będą mogli zastosować wyćwiczonych technik, lecz zostaną zmuszeni do poświęcenia sporej ilości czasu na jego badanie, co utrudni atak lub nawet zniechęci ich.

- Programiści to też ludzie i popełniają oni błędy. Przed użyciem zewnętrznego frameworka czy biblioteki sprawdźmy, kim są jej autorzy, jaka jest opinia środowiska o nich, a także o samej bibliotece. Zobaczmy, co twórcy piszą o swoim kodzie, czy mają sprawny system obsługi błędów, czy wspierają społeczność użytkowników swojego oprogramowania (i czy taka społeczność w ogóle istnieje), a także czy reagują odpowiednio na doniesienia o błędach i regularnie wypuszczają aktualizacje (a przynajmniej po zgłoszeniu nieprawidłowości). W miarę możliwości przejrzyjmy chociaż z grubsza kod i stosowane w nim konstrukcje. Sprawdźmy, czy nie ma w nim najbardziej podejrzanych i alarmujących błędów oraz niebezpiecznych technik, takich jak np. korzystanie z dyrektywy `register_globals`. Dowiedzmy się też, na jakiej wersji PHP się on opiera. Jeśli biblioteka ma zostać użyta w jakimś istotnym miejscu naszej aplikacji, nie bójmy

się zadawać pytań, gdy cokolwiek jest niejasne. Jeśli w rażący sposób nie przejdzie ona któregoś z powyższych testów — odrzućmy ją.

- To, że kod jest dostępny publicznie, może spowodować, że wszystkie błędy będą dla potencjalnego włamywacza widoczne jak na dłoni. Wprawdzie dobry program powinien być napisany tak, aby jawność źródeł nie była osłabieniem jego bezpieczeństwa, jednak w praktyce dość często tak nie jest. Najpopularniejsze aplikacje PHP, takie jak: phpMyAdmin, PHP-Nuke, phpBB, jPortal, myPHPCalendar, PHP-Ticket czy PHP-Fusion, zawierały w przeszłości (a być może zawierają nadal i będą miały w przyszłości) istotne dziury. Niektóre z nich nie otrzymały łat i poprawek przez dość długi okres czasu po opublikowaniu problemów, co niestety dało szansę włamywaczom na przeprowadzenie wielu udanych włamań, a nawet na stworzenie wirusów, robaków i automatów przeczesujących sieć w ich poszukiwaniu.

Kiedy lepiej stosować gotowe biblioteki:

- Do wszelkich funkcji niskopoziomowych, bliskich systemowi, a także takich jak komunikacja z bazą danych i przez FTP czy wysyłanie e-maili.
- Gdy biblioteki te zostały włączone do PHP jako oficjalne rozszerzenia (należy używać wszystkich takich bibliotek).
- Do implementacji skomplikowanych algorytmów wymagających dużej wiedzy algorytmicznej, matematycznej i (lub) specjalistycznej z innej dziedziny nauki. Po co wyważać otwarte drzwi, ryzykując popełnienie błędów, gdy ktoś już wcześniej poświęcił mnóstwo pracy na odkrycie najlepszych rozwiązań? Przykładem mogą być algorytmy szyfrowania czy kompresji danych — jeśli nie jesteś geniuszem matematycznym, lepiej skorzystaj w tym zakresie z gotowej biblioteki.

- Gdy istnieje bardzo małe prawdopodobieństwo, że będziemy chcieli modyfikować kod biblioteki.

Kiedy lepiej jednak napisać własny kod, a przynajmniej poważnie zastanowić się przed użyciem gotowych bibliotek:

- Dobrze jest samemu zaimplementować kod związany z podstawową logiką działania aplikacji (logiką biznesową), nawet jeśli istnieją gotowe komponenty. Gdy tworzymy np. sklep internetowy dla klienta, to albo zdecydujmy się na zaproponowanie mu gotowego, istniejącego kodu (ewentualnie po niewielkich modyfikacjach), albo napiszmy własny, korzystając jedynie z najbardziej bazowych rozwiązań. Zdecydowanie odradzałbym jednak tworzenie go w oparciu o wykrojone kawałki kodu (komponenty, klasy lub wręcz jego fragmenty) z jednego lub kilku gotowych sklepów i sklejanie ich własnymi wstawkami. Jest mało prawdopodobne, że w przyszłości uda się zapanować nad dalszym rozwojem i aktualizacją takiego programu, jak również że powstały kod-zombie będzie bezpieczny. Zgodnie z regułą, aby samemu implementować logikę biznesową, natomiast do niskopoziomowych funkcji korzystać z bibliotek, dobrą praktyką jest na przykład użycie jednej z nich do komunikacji z bazą danych, wysyłania poczty elektronicznej czy uwierzytelniania i autoryzacji użytkowników. Można też skorzystać z jakiegoś prostego frameworka panelu administracyjnego. Natomiast już, dajmy na to, koszyk sklepowy powinien być raczej samodzielnie napisanym fragmentem kodu.
- Zawsze, gdy ważna jest inwencja i nieszablonowość, a jednocześnie napisanie dobrego kodu nie jest trudne (nie trzeba być geniuszem matematycznym). Przykładem może być opisana w rozdziale 3.14 weryfikacja, czy użytkownik jest człowiekiem, czy programem. Walcząc z automatem, warto być twórczym i stworzyć własny, niebanalny kod. Roboty zdecydowanie wolą kod standardowych, dostępnych w sieci

bibliotek (a raczej preferują go ich twórcy, bo mogą wtedy łatwiej nauczyć swoje programy odpowiednich reakcji). Być może nikt nigdy nie napisze automatu oszukującego Twój własny kod, natomiast gdy użyjesz gotowego komponentu, może się okazać, że taki robot już istnieje.

- Bezpieczeństwa nigdy za wiele. Możemy korzystać z gotowych systemów zabezpieczeń, dobrze jest jednak nawet wtedy wpleść w kod od czasu do czasu pewną własną, niestandardową procedurę.
- Gdy zamierzamy często modyfikować kod. Użycie gotowego, zwłaszcza często aktualizowanego, może w takim przypadku zmusić nas do poświęcania dużej ilości czasu na łączenie zmian czy eliminowanie konfliktów.

Warto pamiętać, że język PHP jest bardzo rozbudowany i czasem to, co chcielibyśmy sami zaimplementować, jest już dostępne w jego podstawowej wersji. Dlatego gdy stajemy przed nowym problemem, warto jest rozpocząć pracę od przejrzenia podręcznika użytkownika PHP — może to zaoszczędzić nam sporo czasu i zmniejszyć liczbę potencjalnych błędów.

5.6. Zaciemnianie kodu PHP

Zaciemnianie kodu programu nigdy nie powinno być traktowane jako podstawowy sposób jego ochrony. Zabezpieczenie przez zaciemnienie (ang. *security by obscurity*) nie daje żadnej gwarancji bezpieczeństwa. Może być ono traktowane jedynie jako dodatek zmniejszający liczbę ataków wykonywanych przez „niedzielnych" włamywaczy bądź napastników uzbrojonych w ogólnodostępne narzędzia służące do automatycznych włamań (w języku angielskim funkcjonuje trudne do przełożenia, lecz dobrze określające ten typ włamywaczy określenie *script kiddies*). Zaciemnianie kodu może także mieć na celu jego ochronę przed konkurencją. Jeśli

chcemy uzyskać taki efekt i jest on dla nas ważny, najlepiej będzie jednak zrezygnować z otwartości aplikacji i użyć jednego z profesjonalnych narzędzi kodujących. Tworzą one pliki binarne zawierające kod pośredni, dekompilowany przed właściwą interpretacją przez specjalny program (więcej na ten temat w rozdziale 5.7). Jeśli z różnych przyczyn nie jesteśmy w stanie z nich skorzystać, to zaciemnienie kodu może okazać się dobrym, kompromisowym wyjściem.

Security by obscurity może mieć jeszcze jeden pozytywny efekt uboczny. Jawny i dostępny kod może prowokować klienta, który go zakupił, lub innego niedoświadczonego użytkownika do „grzebania" w nim. Osoba znająca jedynie podstawy PHP może próbować modyfikować go na własną rękę, najczęściej psując go. Zaciemnienie utrudnia takie zabawy. Trzeba jednak pamiętać, że nie jest to rozwiązanie do końca uczciwe wobec klienta czy użytkownika i nie każdy z nich jest w stanie je zaakceptować. Warto więc, zanim zdecydujemy się dokonać zaciemnienia naszego kodu, zawsze indywidualnie rozważyć wszystkie za i przeciw.

Decydując się na zaciemnienie kodu PHP, powinniśmy pamiętać o następujących kwestiach:

- Kod staje się wtedy nieczytelny i niedebugowalny (usuwanie błędów i śledzenie wykonania takiego programu jest ekstremalnie trudne), dlatego nigdy nie zaciemniajmy go przed wypuszczeniem ostatecznej, stabilnej wersji.
- Zaciemnienie kodu może zmienić sposób jego działania. Nawet jeśli ono samo wykonane zostanie poprawnie, to mogą wystąpić takie problemy, jak zmiana czasu działania poszczególnych elementów programu czy wyścig. Jeśli kod jest wrażliwy na zmiany zależności czasowych, może nawet przestać działać. Z tej cechy zaciemniania wynika postulat ponownego testowania całej aplikacji po jego wykonaniu.

- Najlepiej jest napisać program, który będzie w stanie zaciemniać kod w sposób zautomatyzowany. Dzięki temu na serwerze deweloperskim będziemy mogli pracować z otwartymi źródłami, a przed publikacją dokonywać szybkiego zaciemnienia ich. Jako że proces ten będzie wykonywany automatycznie, będziemy mogli powtarzać go wielokrotnie (np. po wykryciu i poprawieniu kolejnych błędów).

Istnieje wiele sposobów zmniejszania czytelności kodu i czynienia go niezrozumiałym dla człowieka, na przykład:

- Zamiana identyfikatorów z czytelnych dla niego na nic nieznaczące. Nazwa zmiennej `'lNumerSeryjny'` może sporo powiedzieć potencjalnemu włamywaczowi, ale jeśli zmienimy ją na `'ab45hc98a_9skj'`, nie będzie on wiedział, o co chodzi. Dla komputera jest to natomiast bez znaczenia — nie interpretuje on nazw zmiennych, funkcji itp. pod kątem znaczenia. Dla niego każda nazwa jest równie dobra.

- Usunięcie znaków końca linii oraz spacji. Odczyt treści programu będzie dość trudny dla człowieka, gdy całość kodu zapiszemy w jednym wierszu, podczas gdy komputerowi nie zrobi to oczywiście żadnej różnicy.

- Stałych występujących w programie nie można zamienić tak łatwo jak identyfikatorów — jeśli to zrobimy, przestanie on działać prawidłowo. Możemy jednak zapisać je w innej formie. Już podział tekstu na poszczególne znaki i napisanie: `'Q' + 'W' + 'E' + 'R' + 'T' + 'Y'` zamiast `'QWERTY'` utrudni życie włamywaczowi. Nie będzie on mógł wyszukać tego ciągu przy pomocy automatu i podmienić go. Jednak przeglądając kod samodzielnie, nadal będzie w stanie go zauważyć. Jeśli ciąg ten jest kluczowy z punktu widzenia bezpieczeństwa, warto pójść dalej. Można zamienić znaki na odpowiadające im kody ASCII, a je z kolei zapisywać nie wprost, lecz np. jako działanie matematyczne. Można też

użyć mniej czytelnego systemu liczbowego, jak np. ósemkowy (aczkolwiek warto pamiętać, że dla włamywacza system szesnastkowy może być czytelniejszy niż dziesiętny).

- Usunięcie komentarzy. Jest to banalna metoda, ale łatwo o niej zapomnieć. Niektórzy programiści modyfikują ją i zamiast usuwać komentarze, zmieniają je na mylące, tj. sugerujące, że dany fragment kodu służy czemuś innemu niż w rzeczywistości. Osobiście nie polecam tego sposobu — łatwo może się on obrócić przeciwko nam.

Dla osób pragnących zaciemnić swój kod stworzyłem narzędzie wykonujące tę pracę. Jest to rozwiązanie bardzo proste, które nie stosuje skomplikowanych i wymyślnych technik. Daleko mu także do wielu dostępnych na rynku, darmowych czy komercyjnych programów tego typu, jest ono jednak interesujące z kilku powodów:

- Jego kod jest niedługi i prosty w zrozumieniu, tak więc czytelnik może go łatwo przeanalizować, aby zobaczyć, jak wygląda korzystanie z różnych technik zaciemniających źródła PHP w praktyce. Można go również użyć jako bazy dla własnego rozwiązania.

- To proste narzędzie jest bardzo użyteczne i sprawdza się co najmniej w 90% sytuacji, w których może zajść potrzeba zaciemnienia kodu.

- Nie uniemożliwi ono zdolnemu włamywaczowi modyfikacji kodu, ale z pewnością ułatwi ochronę własności intelektualnej przed osobami nieznającymi dobrze języka PHP. Dzięki niemu z większym zaufaniem można np. umieścić swój kod na serwerze PHP wynajętym od mało znanej firmy hostingowej, do której nie mamy pełnego zaufania (być może jednak nie powinniśmy nigdy go tam zamieszczać).

Cały kod znajduje się pod adresem: *ftp://ftp.helion.pl/przyklady/php5lk.zip* — poniżej omówię jedynie kilka jego najciekawszych fragmentów i zastosowanych w nim technik.

- Podmiana nazw zmiennych. Zmienna o nazwie `'identi ↪fier'` zostaje zamieniona na ciąg: `'_' . $los . md5($iden ↪tifier . $license_number)`, gdzie `$los` ma wartość losową z zakresu od 0 do 10000, `$identifier` to stara nazwa, a `$license_number` to stała wartość zadana jako parametr. Nazwa zmiennej podmieniana jest na nową we wszystkich plikach we wskazanej lokalizacji (łącznie z podfolderami). Ze względu na zastosowanie dość prostego algorytmu identyfikacji zmiennych w kodzie nie wolno stosować technik takich jak:
  - użycie podwójnego operatora `$$`,
  - dostęp do prywatnych pól klasy spoza niej, np. `$zmien ↪na->moje_pole`. Zamiast tego należy skorzystać z funkcji dostępowej `$zmienna->GetMojePole()` i w niej użyć dozwolonej konstrukcji `$this->moje_pole`. Inaczej mówiąc, wszystkie pola klasy traktujmy jak prywatne. Publiczne powinny być jedynie metody.
- Narzędzie ma zdefiniowane dwie tablice: `DONT_TOUCH` — należy w niej umieścić nazwy zmiennych, które mają zostać pominięte (nie zostaną one zmienione), oraz `CONST_TOKEN`. Zmienne o identyfikatorach z tej drugiej nie będą posiadały części losowej — ich nowe nazwy będą zawsze takie same.
- Narzędzie nie modyfikuje nazw zmiennych rozpoczynających się od znaku `_`.
- Usuwanie znaków przejścia do nowej linii. Po ich wyeliminowaniu kod programu zostanie zapisany w jednym wierszu.

- Usuwanie komentarzy. Dotyczy to zarówno tych zaczynających się od `//`, jak również zawartych pomiędzy znakami `/*` i `*/`.

Użycie narzędzia polega na wywołaniu jednej z dwóch funkcji:

- `ExecuteAllDirectory($license_number, $dir, $verbose)` — dokonuje ona zaciemnienia wszystkich plików znajdujących się w folderze `$dir` oraz w jego podfolderach. `$license_number` to parametr, który zostanie zakodowany w nazwie zmiennej. `$verbose` przyjmuje wartości `0` lub `1`, gdzie `1` oznacza wypisanie na wyjściu informacji o przebiegu zaciemniania, m.in. o każdej modyfikowanej zmiennej, a `0` — cichy tryb pracy.
- `ExecuteAllFile($license_number, $file, $verbose)` — działa ona tak samo jak `ExecuteAllDirectory`, lecz tylko dla pliku `$file`.

Główną funkcją, która zamienia identyfikatory zmiennych, jest `GetVarsFromFile`. Zostaje ona wywołana raz dla każdego pliku, a jej działanie składa się z dwóch etapów:

- Wyszukania ciągu znaków alfanumerycznych, rozpoczynającego się symbolem dolara (identyfikuje on w języku PHP zmienne) i, jeśli nazwa zmiennej była już modyfikowana, podmienienia jej, a jeśli nie miało to jeszcze miejsca — wygenerowania nowej, zapamiętania jej i dokonania zamiany.
- W drugim etapie robimy dokładnie to samo, lecz zamiast znaku `$` szukamy `$this->`.

Kod jest bardzo prosty i z pewnością można go usprawnić i zoptymalizować. Jest napisany w taki sposób, aby był przejrzysty nawet dla osób słabiej znających język PHP. Warto jeszcze wspomnieć o parametrze `$license_number`. Jego wartość jest zakodowana w nowej nazwie zmiennej. Nie jest tam użyta wprost, lecz

razem ze starą nazwą poddana zostaje działaniu funkcji `md5`. Dzięki temu prostemu zabiegowi uzyskujemy następujące cechy:

- Nowa nazwa zmiennej wygląda na losową, lecz jej fragment (zawsze ten sam) zawiera ciąg, który możemy interpretować. Inaczej mówiąc, jedna jej część jest losowa, a druga stała i w dodatku zawiera zakodowaną przez nas tajną informację.
- Powyższą cechę można wykorzystać w celu zabezpieczenia programu. Wszystkie nazwy zmiennych zawierają klucz seryjny, dzięki czemu kod uzyskuje jakby indywidualny „stempel" — można zweryfikować, czy dany jego egzemplarz jest używany przez właściwego klienta. Daje to także możliwość stosowania różnych technik ochronnych (kod programu może na przykład weryfikować, czy pewna konkretna zmienna zawiera w nazwie tajną informację w postaci, której się spodziewamy).
- Jeśli ktoś zmodyfikuje kod poprzez zmianę nazwy zmiennej lub doda nową — jesteśmy w stanie to wykryć. Można też napisać procedurę, która automatycznie sprawdzi poprawność nazw zmiennych w całym programie.

5.7. Kodowanie źródeł PHP

Zakodowanie źródeł programu to coś więcej niż tylko zaciemnienie (rozdział 5.6). Dostępne na rynku narzędzia, takie jak ionCube czy Zend Guard, dokonują kompilacji źródeł PHP do tzw. kodu pośredniego, zapisanego w postaci binarnej i nieczytelnego dla człowieka. Dodatkowo mogą one szyfrować kod, chronić go przed nieuprawnionym użyciem poprzez najróżniejsze formy licencjonowania (ograniczenia czasowe, liczby użyć, limit równolegle korzystających użytkowników itp.) oraz tworzyć jego cyfrowe sygnatury. Narzędzia takie wymagają instalacji na serwerze rozszerzenia dekodującego kod pośredni przed interpretacją przez

serwer PHP kodu właściwego. Ta cecha powoduje, że są one najczęściej niedostępne dla osób korzystających z usług firm hostingowych (choć nieraz firmy takie udostępniają narzędzia dekodujące źródła, tym bardziej że moduły dekodujące najczęściej są darmowe). Jeśli zależy nam na dużym stopniu poufności źródeł, to warto skorzystać z takich narzędzi. Pomimo silnej ochrony, jaką one dają, nie należy jednak opierać systemu bezpieczeństwa aplikacji jedynie na nich!

5.8. Psychologiczne aspekty bezpieczeństwa aplikacji sieciowych

Psychologiczne aspekty bezpieczeństwa aplikacji sieciowych to bardzo szeroka tematyka. Poruszę tu jedynie kilka istotnych aspektów. Postulaty zawarte w tym rozdziale nie powinny stanowić gotowych rozwiązań, a jedynie być „pożywką intelektualną", wspomagającą Czytelnika w dalszych rozmyślaniach, mających na celu stworzenie własnego systemu zabezpieczeń. „Miękkie" sposoby ochrony, tj. metody psychologiczne, w żadnym wypadku nie powinny zastępować „twardych" technik programistycznych. Program powinien być po prostu dobrze zabezpieczony, a pewne psychologiczne techniki, zniechęcające potencjalnego włamywacza bądź skłaniające użytkownika do zwiększenia poziomu swojego bezpieczeństwa, stanowić mogą dodatkowy czynnik zmniejszający prawdopodobieństwo udanego ataku na naszą aplikację.

5.8.1. Dancing pigs vs security — tańczące świnki kontra bezpieczeństwo: zmuś użytkownika do wybrania rozwiązań bezpiecznych

Określenie dancing pigs (tańczące świnki) wzięło się od uwagi Edwarda Feltena i Gary'ego McGraw: *Given a choice between dancing pigs and security, users will pick dancing pigs every time*, co

można przetłumaczyć: „Jeśli dasz użytkownikowi wybór między tańczącymi świnkami a większym bezpieczeństwem, to zawsze wybierze on tańczące świnki". Inaczej mówiąc, przeciętny użytkownik zawsze skłonny jest ignorować komunikaty o zagrożeniach, a mając wybór — wybierać rozwiązanie mniej bezpieczne, ale za to efektowniejsze, modniejsze czy ładniejsze. Użytkownicy często nie czytają ostrzeżeń, klikając *Tak* niezależnie od wielkości użytej w nich czcionki, ilości wykrzykników czy powagi ich tonu. Po prostu ignorują kwestie bezpieczeństwa, uważając, że skoro tyle razy nic się nie stało, to nie stanie się i teraz. Niestety — jeśli użytkownik dozna negatywnych skutków swojego (złego) wyboru, to najczęściej winą obarczy twórcę oprogramowania, a nie swoją lekkomyślność. Wszystko to sprawia, że programista nie powinien w sprawach bezpieczeństwa pytać go o zdanie ani iść na ustępstwa czy kompromisy. Każdy program czy strona WWW powinny domyślnie być zabezpieczone w maksymalnym możliwym stopniu, a dopiero potem można rozważyć, czy nie umożliwić zmniejszenia stopnia ochrony najbardziej zaawansowanym użytkownikom. Taka możliwość powinna być jednak ukryta wewnątrz zaawansowanych opcji i trudna do znalezienia dla początkujących. Najistotniejszych zasad, a w szczególności tych, które mogą wpłynąć na bezpieczeństwo innych użytkowników, nie powinno się nigdy dać wyłączyć lub obejść, chyba że za zgodą i pod kontrolą administratora systemu.

5.8.2. Security by obscurity — prezentuj jak najmniej informacji o aplikacji

Wielokrotnie już podkreślone zostało w tej książce, że zaciemnianie kodu i ukrywanie informacji o wewnętrznych szczegółach aplikacji nie jest mocnym zabezpieczeniem. Warto jednak zapewnić sobie tę dodatkową barierę zmniejszającą liczbę potencjalnie groźnych ataków. Ukrywając informacje o sposobie komunikacji

z bazą danych, działaniu algorytmów, parametrach czy wręcz o tym, że korzystamy z PHP, możemy wyeliminować bądź zniechęcić część włamywaczy, w tym tzw. *script kiddies*, czyli początkujących — posługujących się gotowymi narzędziami, których zasad działania nie rozumieją. Takie podejście może także ograniczyć liczbę ataków wykonywanych przez roboty i — co jest dodatkową zaletą — zmniejszyć nieco niepożądane obciążenie programu (brak danych może zniechęcić część włamywaczy i nie dać robotom punktów zaczepienia do dalszych ataków. W ten sposób bezużyteczny ruch do naszej aplikacji zostanie zmniejszony).

Istotne jest również to, że nie ma ludzi nieomylnych. Każdy z nas popełnia błędy, nawet jeśli nie zawsze zdajemy sobie z tego sprawę. Nawet bardzo dobrze przetestowany program wciąż może zawierać nieprawidłowości i luki w systemie bezpieczeństwa. Nigdy nie będziemy pewni na 100%, że tak nie jest. Dobrze jest więc przynajmniej podjąć próbę zmniejszenia ryzyka wykrycia naszych pomyłek i dziur przez innych. Więcej na temat paradygmatu security by obscurity w rozdziale 5.2.

5.8.3. Czarne listy kontra białe listy

Obrona przed pewnymi zabronionymi elementami, np. odwiedzinami z niektórych adresów IP, określonymi frazami w danych zewnętrznych, niechcianą korespondencją e-mail itp., może zostać przeprowadzona na dwa sposoby:

- Przy użyciu białej listy. Polega ona na dopuszczeniu wyłącznie zamkniętego zbioru akceptowalnych elementów i zablokowaniu wszystkich innych.
- Przy użyciu czarnej listy. Polega ona na zablokowaniu wyłącznie zamkniętego zbioru elementów i dopuszczeniu wszystkich innych.

Elementy zaliczane do czarnej listy mogą być wyznaczane na podstawie pewnych reguł. Podejście takie nazywane jest heurystycznym. Ułatwia ono stworzenie listy i zwiększa szansę na zablokowanie elementów nowych, tj. nieznajdujących się na niej, ale zachowujących się podobnie do znanych już „czarnych elementów". Heurystyczna budowa czarnej listy może wiązać się z blokadą pewnych elementów niezasłużenie, tylko z powodu ich podobieństwa do niebezpiecznych. Analogicznie można tworzyć także białą listę, ale jej skuteczność może być wówczas mniejsza i, podobnie jak w przypadku czarnej, pewne elementy mogą zostać zakwalifikowane do niej nieprawidłowo, co zmniejszy bezpieczeństwo systemu. Generalnie uważa się, że białe listy są bezpieczniejsze od czarnych, gdyż problemem tych drugich jest to, że nie można przewidzieć wszystkich zagrożeń, jakie mogą pojawić się w przyszłości. Wadą białych list może być z kolei ich nadmierna restrykcyjność dla użytkownika, który musi zatwierdzić każdy nowy element. Przykładem programów korzystających z nich są zapory sieciowe, posiadające spis dopuszczalnych portów i adresów, które mogą łączyć się z chronionym przez nie komputerem. Z kolei programy antywirusowe, które do identyfikowania zagrożeń używają zazwyczaj znanej sobie bazy wirusów, są przykładem użycia czarnych list.

Do różnych celów należy stosować różne rozwiązania. Oto przykład:

- Jaś prowadzi małą firmę. Jego głównym sposobem korespondencji z klientami jest poczta elektroniczna. Wielu z nich po raz pierwszy zgłasza zapotrzebowanie na sprzedawane przez niego produkty, właśnie wysyłając e-mail. Jaś otrzymuje także dużo niechcianej korespondencji, w tym wiele reklam. Rozwiązaniem odpowiednim dla niego jest więc użycie czarnej listy z pewnymi elementami heurystyki. Jego program pocztowy powinien blokować korespondencję zawierającą słowa, o których Jaś wie, że nie użyją ich nigdy

jego klienci, a które często stosowane są w reklamach. Ponieważ Jaś prowadzi działalność wyłącznie na terenie Polski, program powinien blokować także wszystkie e-maile z innych krajów, a w szczególności z grupy państw wysyłających najwięcej spamu. W ten sposób Jaś wyeliminuje większość niechcianej poczty, lecz musi liczyć się z tym, że program przepuści niewielką ilość spamu pochodzącego z Polski i niezawierającego zabronionych słów. Użycie białej listy nie jest dla niego dobrym rozwiązaniem, ponieważ nie wie on, kto może zostać jego klientem, i nie jest w stanie stworzyć skończonej listy osób, których wiadomości chce czytać. Sposobem noszącym cechy białej listy byłoby akceptowanie wyłącznie wiadomości z Polski, sprawdziłby się on jednak gorzej niż czarna lista.

- Małgosia używa poczty elektronicznej wyłącznie do komunikacji z rodziną i znajomymi. Ona też otrzymuje dużo niechcianej korespondencji i chciałaby to ograniczyć. Użycie białej listy jest dla niej idealnym rozwiązaniem, ponieważ umożliwia dopuszczenie wiadomości TYLKO od ograniczonej grupy nadawców. Jeśli Małgosia pozna nowych znajomych, to doda ich do niej ręcznie. Nie powinno sprawić jej to kłopotu, bo taka sytuacja ma miejsce rzadziej niż raz w tygodniu. Natomiast użycie czarnej listy, choć także możliwe — nie daje jej gwarancji pozbycia się wszystkich niechcianych e-maili.

Jeżeli zbiór prawidłowych, dopuszczalnych kombinacji jest z góry znany, to lepiej jest zastosować białą listę. Przykładem może być ochrona przed atakami typu cross site scripting. Można wymyślić świetne metody filtrowania i blokady złych fragmentów kodu wstrzykiwanych do danych, ale nie mamy pewności, że ktoś w przyszłości nie wymyśli nowego ich zestawu, obchodzącego nasze zabezpieczenia. Lepiej jest weryfikować informacje z zewnątrz, sprawdzając, czy pasują do przygotowanego przez nas

wzorca, o którym wiadomo, że jest zawsze poprawny — czyli sprawdzić, czy znajdują się one na naszej białej liście. Jeśli nie da się stworzyć wzorca w 100% poprawnego i obawiamy się, że na naszej białej liście nadal mogą znajdować się elementy niechciane, to zastosowanie czarnej listy jako dodatkowej ochrony jest sensowne.

5.8.4. Karanie włamywacza

Co zrobić po wykryciu próby włamania? Czy karać włamywacza, np. usunięciem konta w systemie, a może nawet powiadomieniem organów ścigania? Niekiedy może mieć to sens, jednak zawsze należy przemyśleć następujące problemy:

- Czy naprawdę mamy 100% pewności, że jest to włamanie? A może to legalny użytkownik przez przypadek wygenerował zapytanie do serwera, wyglądające jak próba ataku? Ponieważ w prawie stosuje się domniemanie niewinności, to i my nie możemy zakładać złych intencji, nie mając pewnych dowodów. Atakowi trzeba zapobiec — to oczywiste, karanie włamywacza powinno być jednak zarezerwowane tylko dla specyficznych, najbardziej ewidentnych przypadków.
- Ludźmi kierują różne motywy: ciekawość, chęć sprawdzenia się, uzyskania sławy. Ale może być to także chęć szkodzenia lub uzyskania korzyści cudzym kosztem. Czy napastnik tylko przełamał zabezpieczenia i nic ponadto, czy też dokonał realnych zniszczeń i (lub) w efekcie się wzbogacił? Włamanie nie zawsze cechuje się taką samą szkodliwością i nie zawsze domaga się zastosowania kary. Być może nawet atakujący zgłosi błąd w systemie i pomoże w jego rozwiązaniu? Karaj włamywaczy, którzy dokonali realnych zniszczeń. Pamiętaj jednak, że nie muszą one być fizyczne — kradzież poufnych informacji, np. przez firmę konkurencyjną, może spowodować spore straty.

- Próba automatycznego ukarania włamywacza przez program może umożliwić mu uzyskanie danych cennych przy kolejnych włamaniach (np. gdy program „chwali" się, że zablokował mu konto — nigdy nie wiesz, czy komunikat taki nie da włamywaczowi jakichś informacji, których szukał, podczas gdy samo konto nie jest mu do niczego potrzebne). Może być też tak, że pierwszy atak jest fałszywy i ma na celu odwrócenie uwagi od właściwego lub że mamy do czynienia z atakiem pośrednim i odpowiedź programu (kara) może być w jakiś sposób wykorzystana przez włamywacza w kolejnym jego etapie. Z tego punktu widzenia lepiej jest skupić się na odcięciu możliwości wykonania kolejnych ataków (np. poprzez niewielką blokadę czasową aplikacji, wyłączenie pewnych funkcji administracyjnych do czasu zakończenia śledztwa przez administratora itp.) niż na karaniu, które i tak może być nieskuteczne.

Podsumowując, aplikacja wykrywająca próbę włamania powinna zneutralizować ją i odmówić dalszej współpracy, w miarę możliwości przygotowując dla administratora plik z logiem, zawierający ślady pomocne w namierzeniu przyczyny ataku i samego włamywacza. Nie jest natomiast priorytetem automatyczne karanie go, chyba że specyfika przypadku naprawdę tego wymaga. W razie dokonania poważnego przestępstwa o sporej szkodliwości należy zebrać dowody i zgłosić ten fakt organom ścigania.

5.8.5. Brzytwa Ockhama

Stosując brzytwę Ockhama, nie należy mnożyć bytów ponad potrzebę. Każdy dodatkowy moduł programu, każda nadmiarowa linia kodu, zawiłość czy komplikacja jest niepotrzebna, bo może zawierać błąd wpływający na bezpieczeństwo. Podczas programowania warto mieć w pamięci metaforę przedstawiającą system zabezpieczeń jako łańcuch — jest on tak silny, jak jego najsłabsze ogniwo. W dobrze zabezpieczonym systemie wystarczy jeden

źle chroniony moduł lub funkcja, by był on podatny na ataki. Wszystkie inne środki ostrożności stają się wówczas mało istotne, bo włamywacz prawie na pewno uderzy tam, gdzie opór będzie najmniejszy.

Utrzymywanie kodu w postaci czytelnej i samodokumentującej się również spełnia postulaty brzytwy Ockhama. Im jest on prostszy i łatwiejszy w zrozumieniu, tym mniejsza jest szansa na to, że będzie zawierał błąd obniżający poziom bezpieczeństwa, i tym łatwiej będzie ewentualną nieprawidłowość poprawić. Często lepiej jest napisać kilka linii więcej, uzyskując bardziej czytelny kod, niż skracać go maksymalnie, ryzykując powstanie błędów.

5.8.6. Socjotechnika

Najczęstszą przyczyną udanych włamań jest uzyskanie przez włamywacza informacji znacząco ułatwiających mu działanie (w skrajnym przypadku może on po prostu zdobyć hasło ofiary i podszyć się pod nią — przy takich atakach nie jest ważna wiedza informatyczna). Dane takie często zdobywane są przy użyciu socjotechniki. Włamywacz może przed próbą ataku przeprowadzić rozpoznanie, używając do tego celu telefonu bądź wypytując osoby korzystające z aplikacji. Może na przykład:

- podszyć się pod firmę administrującą serwerem lub inną infrastrukturą IT organizacji korzystającej z programu;
- podszyć się pod dział księgowy, kontrahentów, dostawców, a nawet pod zarząd organizacji;
- udawać zagubionego użytkownika i wyciągnąć w ten sposób ważne informacje od działu obsługi klienta lub informatycznego;
- znając osobiście pracownika, wyłudzić od niego przydatne dane podczas prywatnej rozmowy;

- podsłuchać lub podejrzeć informacje podczas „przypadkowej" wizyty w siedzibie organizacji w roli sprzedawcy, montera, potencjalnego klienta itp. (pracownicy często przyczepiają karteczki z hasłami do monitorów lub na blat biurka);
- przekonać rozmówcę, aby pobrał i zainstalował oprogramowanie przesłane przez Internet (odmianą tego działania może być phishing);
- przy pomocy podstępu zarazić wirusem bądź koniem trojańskim jedną lub więcej maszyn w sieci wewnętrznej organizacji itd.

Metod jest wiele i zależą one głównie od wyobraźni i zdolności psychologicznych oraz aktorskich włamywacza. Zazwyczaj uderza on w osobę najsłabszą z jego punktu widzenia, np. najmniej znającą się na informatyce — czyli taką, której najtrudniej będzie zweryfikować techniczne niuanse, lub np. najkrócej pracującą w organizacji, która nie zna pracowników czy kontrahentów i nie jest w stanie odróżnić ich od włamywacza.

Nie ma łatwych sposobów zapobiegania zdobyciu informacji przez sprytnego włamywacza stosującego socjotechnikę. W grę wchodzi tutaj zawodny „czynnik ludzki". Można jedynie przyjąć pewne zasady i spróbować narzucić je organizacji użytkującej aplikację:

- Informacje istotne z punktu widzenia bezpieczeństwa powinny być znane jak najmniejszej grupie osób.
- Ponieważ czasem ciężko jest przewidzieć, jakie dane mogą być istotne, a jakie nie, użytkownicy powinni udzielać informacji na temat programu czy infrastruktury informatycznej tylko uprawnionym osobom i tylko poprzez zdefiniowany przez nie kanał (np. jeśli administrator wyznaczył specjalną stronę WWW z panelem do zgłaszania uwag, to pracownicy nie powinni wysyłać ich poprzez e-mail ani odpowiadać na jakiekolwiek pytania zadane tą drogą).

- W organizacji powinna istnieć przemyślana polityka haseł. Mogą one na przykład wygasać co pewien czas. Nie powinny być zbyt proste do odgadnięcia ani zapisywane w sposób trwały, szczególnie w miejscu dostępnym dla osób z zewnątrz.
- Infrastruktura informatyczna organizacji powinna być aktualizowana na bieżąco. Dotyczy to w szczególności uaktualniania systemów operacyjnych, programów serwerowych i baz wirusów oprogramowania antywirusowego.
- Użytkownicy powinni stale korzystać z programu antywirusowego oraz zapory systemowej. Instalowanie prywatnych programów powinno być zabronione.
- Co pewien czas powinna być przeprowadzana inspekcja infrastruktury informatycznej pod kątem jej bezpieczeństwa.
- Kluczowe dla organizacji dane powinny być stale archiwizowane. Idealnym rozwiązaniem byłoby przechowywanie archiwów poza siedzibą firmy (na wypadek powodzi, pożaru itp.).
- Pracownicy powinni być regularnie szkoleni w kwestiach bezpieczeństwa systemów informatycznych.

5.8.7. Nie poprawiaj włamywacza

W sytuacji gdy wykryjemy, że dane wejściowe nie są zgodne z dopuszczalnym wzorcem, np. zawierają litery, gdy dozwolone są tylko cyfry — możemy zareagować dwojako:

- spróbować poprawić je, usuwając nieprawidłowe znaki bądź podmieniając je na poprawne (wielu programistów zastępuje na przykład nielegalne symbole znakiem podkreślenia);

- przerwać dalsze ich przetwarzanie i zwrócić informację o błędzie.

Drugi ze sposobów jest zazwyczaj bezpieczniejszy. Lepiej jest nie poprawiać danych potencjalnie pochodzących od włamywacza, ponieważ:

- Nigdy nie mamy 100% pewności, że potrafimy znaleźć i poprawić wszystkie nieprawidłowe znaki. Możemy zapomnieć o jednym z nich i narazić program na włamanie, mimo że poprawimy ich **część**. Gdy będziemy zgłaszać błąd po napotkaniu choćby jednego złego znaku, włamywacz straci szansę na udany atak, nawet jeśli poza nim przemyca też inne symbole, których nie rozpoznajemy prawidłowo. Aby atak się udał, musiałby on dodać do danych wyłącznie znaki, których nie potrafimy odrzucić.
- Nigdy nie wiemy, czy nasza interwencja nie jest na rękę włamywaczowi. Być może atak jest pośredni i liczy on właśnie na naszą procedurę naprawczą. Przypuśćmy, że mamy dwustopniową weryfikację — najpierw sprawdzamy, czy ciąg nie zawiera zabronionych słów, wśród których jest „javascript", a następnie usuwamy znaki spacji. Wprowadzenie przez włamywacza ciągu „javascript" zostanie zablokowane przez pierwszy test. Jeśli jednak poda on go w postaci „java script", to dane te przejdą test pomyślnie, a w drugim etapie „naprawimy" je do postaci „javascript", eliminując spację.