JAKUB KAŁUŻNY

Robactwo w kodzie

Stopień trudności

Zastanawiałeś się kiedyś, czy kod, który napisałeś, jest nieskazitelnie czysty? Czy możesz użyć go na własnej stronie bez obawy, że ktoś wykryje lukę i zaatakuje system? Bałeś się , że sprzedasz klientowi skrypt, a jakaś osoba włamie się na stronę?

Napisałem ten tekst, ponieważ często zdarza mi się oglądać sklepy internetowe bądź strony tematyczne, które stały się ofiarami ata ków hakerów. I to wcale nie są ataki tzw. *black hats* (patrz Ramka 1) czy profesjonalnych złodziei nastawionych na zysk z ukradzionych danych. Setki witryn padają z powodu młodych ludzi, którzy z nudy bawią się zabezpieczeniami stron. Czasem powodują oni tylko małe zakłócenie w pracy serwisu, jednak bywa i tak, że stronę trzeba zamknąć na jakiś czas. Dodają oni jakiś zabawny komunikat, a czasem nawet usuwają pewne dane z dysku. Dlaczego te strony stają się tak łatwym łupem? Moim zdaniem ze skąpstwa, bądź po prostu z lenistwa. Lenistwa tych, którym nie chce się jeszcze raz dokładnie przejrzeć napisanego kodu pod kątem zabezpieczeń. A ze skąpstwa tych, którzy woleli zaoszczędzić i umieścić ogłoszenie na jednej z bezpłatnych stron typu *http://zlecenia.przez.net* (oczywiście nie mówię, że wszyscy, którzy odpisują tam na ogłoszenia wykonują pracę nieporządnie – aczkolwiek dość często stawkę obniżają ludzie, którzy jednak nie znają się na rzeczy), niż zapłacić – co prawda 10 razy więcej, ale legalnej firmie, która porządnie wykona robotę. Bowiem ostatnio na rynku usług internetowych (konkretnie stron WWW) nastąpiła wielka ekspansja małych firm oraz *freelancerów*, których motto brzmi *szybko i tanio*, co – jak wiadomo – rzadko idzie w parze z jakością. Często zdarza się, że wykonawcy stron są licealistami bądź studentami, którzy dorabiają nielegalnie w Internecie. Tak więc docelowi właściciele dość często wolą zaufać jakiejś osobie, nawet bez większego portfolio, zlecić jej wykonanie strony całkowicie na czarno, bez umowy o dzieło.

Boom na e-shopping

Za przykład posłuży nam fikcyjny sklep internetowy, od jakich ostatnio aż roi się w polskim Internecie. Według mnie to już ostatnia chwila na wejście w biznes. Niedługo będzie ich po prostu za dużo, w związku z czym wielki popyt na e-shopping rozłoży się na wiele małych serwisików, które w końcu zaczną plajtować. Sprawa wygląda dokładnie tak samo, jak z rzeczywistymi sklepami. Mniejsi handlowcy z trudem walczą o swoje parę groszy, podczas gdy supermarkety zbierają kokosy. Być może właśnie w tym problem z nowymi, dziurawymi e-sklepami – każdy chce się wbić w rynek, ale nie ma odpowiedniego kapitału, więc np. zleca wykonanie strony za marne pieniądze. Wracając do naszego przykładu – strona funkcjonuje podobnie, jak Allegro – użytkownik wystawia produkt, a inni mogą go kupić. Ponadto właściciel poprosił o zaimplementowanie systemu banerów – za odpowiednią ilość złotówek można wykupić pakiet reklam na stronie (odnoszących się do produktów znajdujących

Z TEGO ARTYKUŁU DOWIESZ SIĘ

jakie są podstawowe luki w zabezpieczeniach aplikacji webowych,

jak się obronić przed atakami hakerów.

CO POWINIENEŚ WIEDZIEĆ

podstawy PHP,

podstawy HTML,

podstawy składni shella linuksowego.

się w bazie danych). Cały portal opiera się na *PHP* i bazie *MySQL*. W kolejnej części artykułu przedstawione zostaną podstawowe błędy popełniane przez niedoświadczonych programistów.

Tricki w URLu

Po pierwsze, koderzy mają tendencję do nadużywania formularzy *GET*, co prowadzi do stworzenia możliwości prostego ataku poprzez modyfikację adresu *URL*. Powiedzmy, że mamy konto w sklepie i chcemy wykupić pakiet 1000 banerów. Możliwa jest opcja uploadu własnego obrazka. Najpierw dodajemy nasz baner, formularz wygląda tak, jak pokazuje Rysunek 1.

Oczywiście, mamy możliwość usunięcia tych, a dodania innych obrazków. Gdy chcemy usunąć baner, klikamy w odpowiedni link. W pasku adresu widać *http://adres.strony.pl/banner.php?del=4333*. A więc teraz dodajemy kolejny obrazek. O, ale co to? Tym razem, gdy chcemy usuwać nasz baner, jego ID to już `4335` – widocznie ktoś międzyczasie dodał swój baner i został on zapisany w bazie jako `4334`. A gdyby tak wpisać w pasek adresu *http://adres.strony.pl/banner.php?del=4334*. Komunikat *Twój baner* (`ID: 4334`) *został usunięty* potwierdza domysły – programista nie wziął pod uwagę faktu, że należy przed usunięciem baneru sprawdzić, czy ID jego właściciela zgadza się z identyfikatorem zalogowanego użytkownika, przechowywanym w superglobalnej zmiennej `$_SESSION['login']`. `1:0` dla nas. Może gdyby użyto formularza *POST*, nikt nie pomyślałby, że należy tylko podmienić ID w URLu? Bardzo złośliwym posunięciem byłoby teraz usunięcie wszystkich banerów. Przysporzy to sporo pracy administratorom, żeby odtworzyć stronę, zdenerwuje klientów, którzy – bądź co bądź – zapłacili za reklamę, no i oczywiście popsuje opinię sklepu oraz właściciela. Wymagane narzędzia ataku? Dowolna przeglądarka internetowa, a jeśli banerów jest dużo, możemy napisać krótki program, który zautomatyzuje nasza pracę, np. w bashu linuksowym. Spójrzmy na Listing 1.

Oczywiście obrona przed tego typu *atakiem* jest trywialna – w kodzie PHP strony wystarczy dodać tylko jeden warunek, przedstawiony na Listingu 2.

Zadaniem pokazanej na listingu funkcji user(ID) jest zwrócenie loginu użytkownika, który jest właścicielem banera o danym numerze ID.

Modyfikacja formularza

Drugim częstym błędem jest przekazywanie niepotrzebnych argumentów w formularzach. I tu już nie ma różnicy, czy formularz jest typu GET, czy POST (patrz Ramka 2). Tak więc

Listing 1. *Zmiana URLa strony w celu usunięcia banerów*

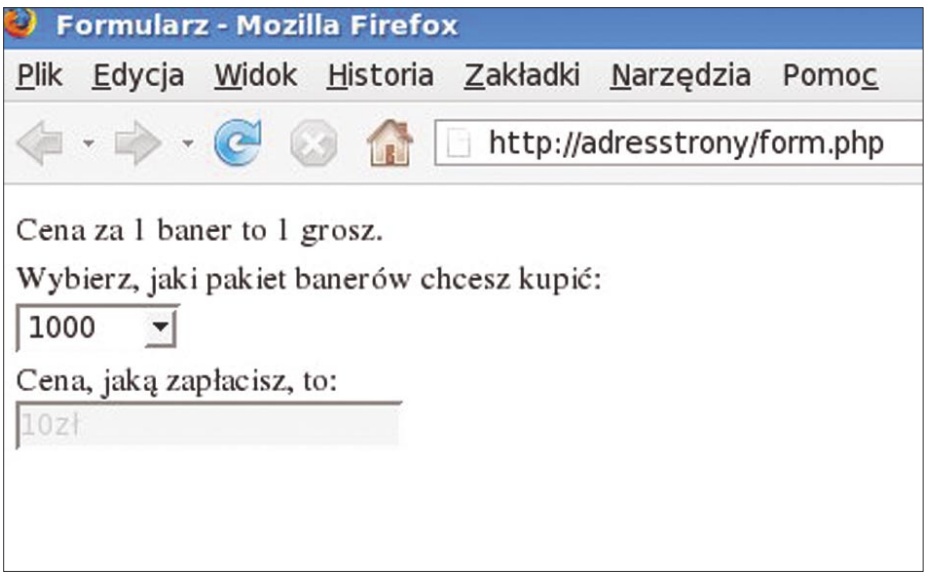
```
#!/bin/bash
for i in `seq 1 100000`
do
touch $i.tmp
links http://adresstrony/banner.php\?del\=$i > $i.tmp
echo „Baner $i usuniety”
done
exit 0
```

Listing 2. *Warunek sprawdzający, chroniący przed nieautoryzowanym usuwaniem banerów*

```
if($_SESSION['login']==user($_POST['del'])
{
 // tu kod, który usuwa baner
}
```

Listing 3. *Kod źródłowy formularza zamawiania banerów*

```
<form action=”banner.php” method=”post”>
Cena za 1 baner to 1 grosz. <br>
Wybierz, jaki pakiet banerów chcesz kupić: <br>
<select name=”ile”>
<option> 100 </option>
<option> 1000 </option>
<option> 5000 </option>
<option> 10000 </option>
<option> 20000 </option>
<option> 100000 </option>
</select>
<br>
 Cena, jaką zapłacisz, to:
<br> <input type=”text” name=”cena” disabled=”true”>
</form>
```



Rysunek 1. *Wygląd formularza w przeglądarce.*

– dodaliśmy już nasz obrazek, teraz czas na wykupienie (oczywiście drogiego, a jakżeby inaczej) pakietu reklam. W tabeli MySQL prawdopodobnie przechowywane są następujące dane – ID pakietu, ilość wykupionych banerów, ilość już wyświetlonych oraz cena. Kupując banery, wypełniamy prosty formularz z jednym polem, oznaczającym ilość wyświetleń banerów, które chcemy umieścić na stronie.

Spoglądamy w kod źródłowy pliku (Listing 3.).

Z listy wybieramy ilość pakietów, automatycznie poniżej w polu *cena* wyświetla się koszt, jaki poniesiemy – do jego wyświetlenia posłuży prosty kod JavaScript (`document.forms[0].cena.value=document.forms[0].ile.value *cena_za_pakiet;`). Oczywiście, tego pola nie można edytować, bez atrybutu `disabled=true` byłoby to zbyt proste. Zastosujemy teraz trochę sprytniejszą technikę. Kopiujemy źródło formularza, wstawiamy na swój serwer i zmieniamy kod w sposób przedstawiony na Listingu 4.

Oczywiście, zmodyfikowaliśmy wartość atrybutu `action` na *http://adresstrony/banner.php*. W polu, którego wartość jest oryginalnie wyznaczana przy pomocy skryptu JS, zmieniamy wartość na symboliczną złotówkę. Warto usunąć także sam skrypt JS, mógłby on zakłócić pracę tak zmodyfikowanego kodu. Wypełniamy formularz na naszym serwerze tak, jak byśmy to zrobili poprzez stronę sklepu i zaglądamy później na swoje wirtualne konto, już w sklepie. Ponownie punkt dla nas, programista przekazywał cenę w tablicy `$_POST`, zamiast obliczać ją już na serwerze, bazując na wartościach pozostałych pól.

Obrona w najprostszym wypadku polegałaby na dodaniu w skrypcie akceptującym zamówienie banerów warunku sprawdzającego, czy ilość banerów pomnożona przez cenę za pakiet daje wynik równy wartości przekazanej w zmiennej cena (pokazuje to Listing 5).

Listing 4. *Modyfikacja formularza POST*

```
<form action="http://adresstrony/banner.php" method="post">
Cena za 1 baner to 1 grosz. <br>
Wybierz, jaki pakiet banerów chcesz kupić: <br>
<select name="ile">
<option> 100 </option>
<option> 1000 </option>
<option> 5000 </option>
<option> 10000 </option>
<option> 20000 </option>
<option> 100000 </option>
</select>
<br>
 Cena, jaką zapłacisz, to:
<br>
<input type="text" name="cena" disabled="true" value="1">
</form>
```

Listing 5. *Warunek sprawdzający poprawność zmiennej cena*

```
if($_POST['cena']==($_POST['ile'])*$cenazapakiet)
{
// tu kod odpowiedzialny za dodanie baneru
}
```

Hakerów ze względu na etykę dzielimy na 3 grupy:

- *black hats*: ludzie niedzielący się zdobytą wiedzą z innymi, nie publikują odkrytych luk, działają najczęściej poza granicami prawa,
- *white hats*: ludzie, którzy chętnie pomagają innym, o odkrytych lukach informują administratorów, nie udostępniają bezpośrednio sposobu na włamanie, wśród nich często można znaleźć informatyków, którzy zajmują się bezpieczeństwem IT,
- *grey hats*: ludzie, których trudno jednoznacznie przypisać do którejś z powyższych grup.

Javascript – dodatek, nie podstawa!

Trzecia istotna kwestia – wykonywanie ważnych operacji za pośrednictwem kodu JavaScript może ułatwić atak. W naszym sklepie dodajemy produkty, rejestrujemy się bądź kupujemy banery. Często w polach formularza należy wpisać jakieś dane, zwykle określonego typu. Załóżmy, że musimy podać numer telefonu stacjonarnego w Polsce. Wpisujemy go w postaci `KKXXXXXXX`, gdzie `K` – to numer kierunkowy, a `X` – cyfra. Po wpisaniu ostatniej cyfry, gdy pole zawiera 9 znaków, wykonywana jest pewna funkcja JS, np. `CheckNumber(document.forms[0].telefon.value);`. Jeżeli pole nie zawiera samych cyfr, funkcja ustawia zmienną `documents.forms[0].ok.value (hidden)` na `0` (nie pozwoli ona przejść dalej w formularzu), w przeciwnym razie pozostawia kontrolki bez zmian i wyświetla dla wiadomości użytkownika obok pola zielonego ptaszka (OK) bądź czerwony krzyżyk (błąd). Co można tutaj zrobić? Patrzymy w kod źródłowy i analizujemy linijki pokazane na Listingu 6.

Wpisujemy w formularzu błędny numer. Kiedy pojawi się krzyżyk – możemy po prostu wpisać w okno przeglądarki (najlepiej Firefox, tam na pewno działa) poprawny numer:

```
javascript:CheckNumber(123456789);
```

Innym sposobem jest obejście sprawdzania poprzez wpisanie:

```
javascript:document.forms[0].ok.value=1;
```

Co to spowoduje? W obu przypadkach efektem będzie ustawienie zmiennej *ok* na 1, ponadto w drugim przypadku nie pojawi się zielony ptaszek. Całość pozwoli na wysłanie formularza z fałszywym numerem telefonu. Z pewnością bardziej interesujący będzie praktyczny przykład. Załóżmy, że chcemy zagrać w brydża na serwisie *kurnik.pl* – niestety, akurat pokój, w którym gra nasz przyjaciel, jest już

przepełniony i jego nazwa przestała być aktywnym hiperłączem. Jaki problem stanowi najechanie myszką na inny pokój, skopiowanie do pasku adresu kodu JavaScript i zmiana parametru – nazwy pokoju? Nie zarzucam niczego autorowi skryptu – jeżeli chcemy zagrać z przyjacielem, możemy kulturalnie otworzyć jego profil i wejść do pokoju, w którym akurat jest. Bądź co bądź, jest to jednak pewien sposób na obejście blokady i przeciążenie serwera.

Jak się przed tym obronić? Unikajmy pisania takich funkcji w JS. Podstawową wadą skryptów JS jest oczywiście jawność kodu, której pozbywamy się używając PHP. Ponadto funkcji PHP nie wywołamy z okna przeglądarki. Kod pisany w JS nazywamy `client-side` – wykonywany po stronie klienta (czyli przeglądarki), natomiast kod PHP – `server-side` – wykonywany jest na serwerze, co znacznie utrudnia znajdywanie luk.

Jeden problem to nie problem

Problemy podczas rejestracji to dość częste zjawisko. Bardzo łatwo można uprzykrzyć życie administratorowi. Jeżeli witryna nie wykorzystuje np. obrazka z kodem weryfikującym, możemy napisać bota (czasem wystarczy odpowiednią ilość razy odpalić spreparowany formularz), który zarejestruje fikcyjnych użytkowników z nieprawidłowymi adresami e-mail. Wynik jest łatwy do przewidzenia: baza MySQL zapchana tysiącami rekordów. Spowolnienie działania skryptów. Na skrzynkę roota trafiają setki maili od mailer-demona, że listy do zarejestrowanych użytkowników nie dochodzą. Co więcej, trudno takie coś nazwać spamem – w końcu serwer sam wysyła pocztę. Jak temu zapobiec? Stworzenie obrazka zawierającego kod weryfikujący, który trzeba podać w formularzu oraz blokowanie kilku rejestracji pod rząd z jednego adresu IP powinny wystarczyć. Jeżeli haker użyje programu OCR (ang. *Optical Character Recognition*), rozpoznającego tekst zapisany w formie graficznej, albo będzie zmieniał szybko IP, to już będzie poważniejsza sprawa i obrona okaże się znacznie trudniejsza.

SQL injection – standard

Często w portalach czy sklepach udostępniamy funkcję *Szukaj na stronie*. Oczywiście, skrypt ten przekazuje zapytanie do bazy danych MySQL i zwraca odpowiednie wyniki. Przy połączeniach z bazą należy zwracać szczególną uwagę na bezpieczeństwo. Podstawowy błąd to umożliwienie przeprowadzenia ataku tzw. SQL Injection. Jeżeli pozwolimy na używanie znaków typu '!@#$%^&*()¯\/|=›‹.,;: , bardzo prawdopodobne, że haker użyje ich do ataku. Powiedzmy, że haker domyśla się sposobu przerabiania pola *Szukaj* na zapytanie do bazy i zamiast nazwa_użytkownika wpisze:

```
nazwa_użytkownika"; SELECT * FROM
 'users';
```

Skrypt działa prawdopodobnie tak: wysyła zapytanie do bazy `SELECT 'users' WHERE nazwa LIKE „[tu skrypt kopiuje zawartość pola szukaj];` , zapisuje je w jakiejś tablicy i drukuje w odpowiedniej tabeli. Jeżeli hakerowi się poszczęści, uzyskuje on dane użytkowników i ich hasła. Dlaczego? Ponieważ wymusił wykonanie innego zapytania, które wypisało zawartość tabeli users z bazy danych. Sytuacja wygląda podobnie, gdy mamy zapytanie postaci:

```
SELECT * FROM 'users' WHERE
login=$login AND pass=$pass;
```

w polu login wpisujemy `admin; ~`. Tylda (~) jest w języku SQL oznaczeniem komentarza – wszystkie występujące po nim komendy nie będą wykonywane. W tym przypadku zostanie pominięta druga część zapytania związana z hasłem. Bardzo popularne jest również wstrzyknięcie kodu `' OR 1=1;~~`. Tak zmodyfikowane zapytanie wypisuje z konkretnej tabeli wszystkie rekordy – który bowiem który z nich jest niezgodny z warunkiem `1=1`? Jak się obronić przed tym atakiem? Przede wszystkim należy stosować funkcję `addslashes()`, która doda do potencjalnie niebezpiecznych znaków slash (\ ). Możemy też stworzyć klasę czy funkcję, która np. pobierając na wejściu trzy argumenty sprawdzi,

Listing 6. *Fragment kodu strony odpowiedzialny za analizę numeru telefonu*

```
CheckNumber(number)
{
if(length(number)!=9) document.forms[0].ok.value=0;
if(!is_numeric(number)) document.forms[0].ok.value=0;
WyswietlPtaszkaBadzKrzyzyk(numer);
}
```

Listing 7. *Sprawdzenie, czy zmienna liczbowa ma poprawny format*

```
if(is_numeric($id))
{
// tu kod odpowiadający za wybranie produktu
}
```

Listing 8. *Usprawnienie algorytmu MD5 – zastosowanie soli*

```
moje_md5($string)
{
$string1=$string."sol123!@#";
return md5($string1)
}
```

Typy formularzy

GET umieszcza zmienne w pasku adresu, co ładnie prezentuje linki i pozwala na łatwą i bezpośrednią zmianę tych danych. POST przesyła informacje przez przeglądarkę internetową i uniemożliwia przesłanie linku z konkretnymi danymi, można dopiero je zmienić w formularzu.

czy nie zawierają one niebezpiecznych znaków i przerobi na zapytanie do bazy danych (np. `Sql::cmd('select', 'produkty', 'id="'.$id.'"');`). Jeżeli mamy wykonać zapytanie typu `SELECT * FROM PRODUKTY WHERE id="$id"`, warto wcześniej sprawdzić, czy parametr funkcji jest rzeczywiście liczbą – przykładowy kod weryfikujący znajduje się na *Listingu 7*. Analogicznie postąpić można z loginem (np. sprawdzając, czy zawiera tylko litery i cyfry, ewentualnie dopuszczalne znaki specjalne).

Passy bezpieczeństwa

Jeszcze jeden niezwykle istotny element to hasła. W żadnym wypadku nie można przechowywać ich w bazie w czystej postaci (ang. *plaintext*). Dzisiejszym standardem jest – mimo znanych słabości – algorytm haszujący MD5. Polecam jednak zastosowanie dodatkowego mechanizmu tzw. soli (ang. *salt*) – losowego ciągu znaków dodawanego na początku lub końcu danego wyrazu przed haszowaniem – utrudnia to ataki bruteforce i rainbowcrack, z predefiniowaną tablicą deszyfrującą). Przykład zastosowania soli zawarto w kodzie znajdującym się na Listingu 8.

Dlaczego powinno się stosować takie dodatkowe zabezpieczenia? Sporo słyszeliśmy o projektach typu rainbowcrack. Jak dla mnie, hasła poniżej 13 znaków są w MD5 po prostu zbyt łatwe do złamania. Dobrym pomysłem jest również zamiana na 'porządniejszy' algorytm – choćby SHA2, czy Blowfish. Także do nich można dołożyć własną modyfikację – na przykład sól. Pamiętajmy, że podstawowym obowiązkiem administratora jest dbanie o bezpieczeństwo klientów. Jeśli umożliwimy przejęcie haseł przez kogoś obcego – na pewno wpłynie to negatywnie na statystyki oglądalności strony, nie mówiąc o możliwych kłopotach innego rodzaju.

Listing 9. *Zabezpieczenie przed zdalnym wykonaniem kodu*

```
if(is_numeric($sekundy))
{

// tu funkcja
}
```

Listing 10. *Skrypt służący do automatycznego tworzenia zapytań*

```
#!/bin/bash
TAB=(a b c d e f g h i j k l m n o p q r s t u v w x y z 0 1 2 3 4 5 6 7 8 9)

for j in `seq 0 35`
do
for i in `seq 0 35`
do

touch ${TAB[$j]}${TAB[$i]}.tmp
links http://adresstrony/szukaj.php?query=${TAB[$j]}${TAB[$i]} >
                      ${TAB[$j]}${TAB[$i]}.tmp

done
done
exit 0
```

Projekt rainbowrack

Projekt rainbowrack – tworzenie tablic (tzw. Tęczowych), które pozwalają na odwrócenie jednostronnej funkcji MD5 i podobnych, bazując na algorytmie przyśpieszającym ten proces. Tablica dotyczy konkretnego zestawu znaków do ograniczonej długości hasza. Jeżeli wiele osób wygeneruje małe cząstki tablicy, można je zebrać razem i stworzyć wielką tablice odwracającą algorytm haszujący. Projekt rainbowcrack można znaleźć w Sieci pod adresem *http://www.antsight.com/zsl/rainbowcrack*, a popularne zestawy tablic – *http://www.milw0rm.com/cracker* oraz *http://rainbowcrack.com*.

System zostawmy w spokoju

W żadnym wypadku nie wywołujmy poleceń systemowych poprzez PHP. Słyszałem kiedyś o pomyśle napisania demona w PHP, który miałby wykonywać określone działania co zdefiniowaną liczbę sekund. Powiedzmy, że wyglądało to tak:

```
$sekundy=$_POST['sek']; while(true)
{ system(„sleep „.$sekundy.”); coś(); }.
```

Zmienne w PHP najczęściej nie mają zdefiniowanego typu. Napastnik zawsze mógłby zrobić coś, co nazwałbym *PHP Injection*, i wywołać stronę z parametrem w URL podobnym do *?sekundy=5;cat%20/etc/shadow%20›%20/var/www/html/index.html*. Jeżeli już skrypt potrzebuje dostępu do polecenia systemowego, należy to odpowiednio zabezpieczyć. W tym przypadku proponuję zrobić coś, co znajduje się na Listingu 9. Warto zapoznać się również z funkcjami języka PHP, takimi jak `escapeshellarg()` czy `escapeshellcmd()`.

Łatwo wyobrazić sobie następującą sytuację – skrypt PHP odpowiada za wyświetlanie newsów, które na serwerze są umieszczane w postaci plików o nazwach np. *20010504.txt*. Linki do artykułów bazują na formularzu GET w postaci *http://adresstrony/show.php?file=20010504.txt*. Czy wywołanie newsa o takiej nazwie jest trudne: *http://adresstrony/show.php?file=/etc/shadow*? Oczywiście należy ograniczyć dostęp funkcjom czytającym pliki jedynie do wskazanego katalogu.

Mówiąc o systemie, należy wspomnieć o połączeniu aplikacji internetowej napisanej w PHP z bazą danych, np. MySQL. Zazwyczaj piszemy wtedy plik config.php, który musi być włączony do każdego pliku w naszej aplikacji otwierającego połączenie z bazą. W tym pliku znajdują się bardzo ważne dane, jak nazwa użytkownika, nazwa bazy danych i hasło. Jeżeli na naszym serwerze udostępniamy miejsce komuś innemu (a często tak jest na serwerach uczelnianych,

szkolnych, firmowych), należy zadbać o odpowiednie prawa dostępu do pliku. Najlepiej CHMOD 640 – zapis i odczyt dla właściciela, odczyt dla serwera oraz żadnych praw dla innych użytkowników!

Bazy danych

Abstrahując od sklepu – załóżmy, że prowadzimy pewien katalog. Może być to dowolny zbiór – np. firm w Polsce zajmujących się wyrobem cukierków, lista nazwisk, lista linków do stron. Cokolwiek to nie jest – publikując, zadbajmy o to, żebyśmy nie łamali prawa. A teraz sprawa techniczna. Jak uniknąć sytuacji, w której ktoś bezczelnie skopiował nasze dane i opublikował je na swojej stronie? Oglądalność naszej witryny najprawdopodobniej by zmalała, co spowodowałoby np. zmniejszenie zysków z reklam na stronie. Podzielenie listy na parę/set/tysięcy stron jest pewnym pomysłem – aczkolwiek broni jedynie przed laikami, którzy korzystają z metod Copy'ego-Paste'a. Obrona w tym przypadku jest bardzo skomplikowana – można dodawać w źródle ukryte znaki, które wyjdą na jaw dopiero przy kopiowaniu i zepsują listę. Można ograniczać ilość zapytań z jednego adresu IP, czy nawet blokować podobne zapytania z kilku IP. Przejście do kolejnej strony można zrealizować w JavaScript. Sprawa z pewnością nie jest prosta. Jeżeli udostępniamy funkcję *Szukaj* i przekazujemy parametr przez formularz GET, przygotujmy się na sytuację, w której haker napisze krótki skrypt, wyglądający np. tak, jak na Listingu 10.

Wtedy blokujmy jego IP np. po 100 zapytaniach albo po 50 zapytaniach, które różnią się tylko o jeden znak.

Co dzieje się, jeśli szukamy danych w *Google*? Strona *google.pl* blokuje IP po średnio 300 zapytaniach na jedna godzinę. Daje to ok. 300 000 wyników (z jednego zapytania wyciągniemy tylko 1000 wyników). Jednak serwery *google.com* są niezależne od tych z Polski – tam również możemy wykonać 300 zapytań. Podobnie *google.de*, *google.it* i tak dalej – a za godzinę wracamy i rozpoczynamy wszystko od początku. Domen krajowych *Google* ma ok. 150 (nie wszystkie są na różnych serwerach), ich adresy znajdziemy na: *http://www.google.pl/language_tools?hl=pl* . W tym przypadku wystarczy wiec napisać kod, który różni się od przedstawionego na *Listingu 10* tym, że co 300 zapytań zmienia końcówkę adresu, czyli links *http://google.${DOMENA[$z]}/?q=.....* gdzie tablica DOMENA zawiera wpisy np. *pl us it fr de*.

Jeżeli już o przeciążeniach mowa. Z całym szacunkiem – ale co to za firma, która swoją stronę internetową umieszcza na którymś z darmowych serwerów? Wykonanie paru tysięcy odświeżeń nie wymaga nawet pisania programu, wystarczy w Operze bądź Firefoksie otworzyć kilkanaście zakładek i ustawić odświeżanie np. co 30 sekund. *Bandwidth* – czyli limit (np. miesięczny) danych przesyłanych przez serwer – od razu zostaje przekroczony, a strona zawieszona. Nie jest to zmora tylko darmowych serwerów – nawet na płatnych, jeżeli nie zablokujemy konkretnego IP po np. tysiącu napływających z niego żądań wyświetlenia strony, parę GB transferu nietrudno zużyć przez jedną noc.

Socjotechniki

Akapit ten nie dotyczy już bezpośrednio kodu stron internetowych, ale haseł, które chronią dostępu do witryn. Nie chcę tu mówić o zasadach tworzenia haseł – 3 klasy znaków, długość co najmniej 10 znaków itp. – a raczej o częstotliwości ich zmieniania oraz samej treści. Powiedzmy, że interesujemy się łodziami podwodnymi oraz II wojną światową. Wymyślając parę lat temu hasło akurat wpadliśmy na *orpdzik39*. Udzielając się na forach internetowych, wiele razy wspominaliśmy o naszych zainteresowaniach. Taktyki hakerskie nie polegają tylko na włamywaniu się na komputery czy strony internetowe. To również poznawanie ofiary. Osoba, która zna tylko nasz adres email, może szybko znaleźć wiele innych informacji. Google nie pokazuje tylko aktywnych stron, ale posiada również archiwum dostępne przy wynikach jako *kopia*. Potężne archiwum Internetu znajduję się na *http://web.archive.org*. Haker poznaje naszą psychikę i zainteresowania, tworzy listę wyrazów związanych z nami w jakiś sposób i dokonuje ataku słownikowego. Statystyki mówią, że hasła obecnie są najczęściej składają się z 6-9 małych liter, na końcu znajdują się zwykle dwie, rzadziej jedna cyfra. Bardzo rzadko hasła są dłuższe niż 12 znaków. Jeżeli haker odgadnie hasło, znajdziemy się w bardzo niemiłej sytuacji – może on np. przeglądać naszą korespondencję właściwie bez naszej wiedzy. Jak się obronić? Nie jest to proste w zastosowaniu – należy często zmieniać hasła i w żadnym wypadku nie używać tego samego na dwóch kontach. A samo hasło nie powinno być jakkolwiek z nami kojarzone – najlepiej, gdyby było przypadkowym ciągiem znaków. Wtedy metoda słownikowa na nic się nie przyda, a jeżeli hasło jest dostatecznie długie, metody brute force staną się nieoptymalne czasowo.

Podsumowanie

Rozwój usług internetowych w Polsce jest bardzo dynamiczny, ponieważ każdy produkt, czy usługa potrzebuje reklamy. Pomimo, że na promocję w Internecie wydaje się nieznaczny procent budżetu firmy (w porównaniu do spotów w telewizji, banerów na ulicach), jest to dość łatwy sposób na wstępne 'rozkręcenie' biznesu. Obecnie powstaje ogromna ilość małych firm wykonujących strony internetowe, jednak nie należy ufać całkowicie komuś jedynie dlatego, że wystawia faktury VAT. Również takie osoby mogą wykonać stronę nieprofesjonalnie i umożliwić atak hakerowi. Jeżeli zlecamy wykonanie strony – zapytajmy programistę o to, czy końcowy skrypt jest dostatecznie bezpieczny. Natomiast jeśli to my jesteśmy twórcami aplikacji – zawsze warto kolejny raz przeczytać kod, sprawdzając dogłębnie każdą funkcję, niż później tłumaczyć się z *dziurawca*. W tej branży ceni się solidność, a nie czas wykonania – w końcu strona internetowa ma w zamierzeniu funkcjonować jak najdłużej.

Jakub Kałużny
Od kilku lat interesuje się bezpieczeństwem IT, w szczególności aplikacji internetowych. Zaangażowany w rozwój systemów zarządzania treścią (ang. *CMS*). Fascynuje go kryptografia, jest zapalonym miłośnikiem Linuksa.
Kontakt z autorem: gadulix@gmail.com