BIULETYN INSTYTUTU AUTOMATYKI I ROBOTYKI
NR 23, 2006

# Metody usuwania podatności URL na ataki na aplikacje internetowe

**Janusz FURTAK, Łukasz STRZELECKI, Kamil RENCZEWSKI**

Instytut Teleinformatyki i Automatyki WAT
ul. Kaliskiego 2, 00-908 Warszawa

STRESZCZENIE: W artykule przedstawione zostały wybrane problemy bezpieczeństwa aplikacji internetowych dotyczące przechowywania danych i pobierania danych od użytkownika. Przedstawiono różne przykładowe rozwiązania zmniejszające możliwość przeprowadzenia skutecznego ataku na system portalowy.

SŁOWA KLUCZOWE: podatność URL, szyfrowanie URL, bezpieczeństwo aplikacji internetowych

## 1. Bezpieczeństwo systemu portalowego

Czas, w którym witryny WWW były traktowane jako dodatek, czy też urozmaicenie wachlarza usług poszczególnych instytucji dawno przeminął. Obecnie są one podstawowym elementem stale wykorzystywanym w strategii marketingowej firm. Kreują wizerunek firmy, przyczyniają się do zwiększenia zysków, ale również stanowią potencjalny cel ataków i mogą spowodować poważne straty. Główne powody, dla których ważnym jest utrzymywanie wysokiego poziomu bezpieczeństwa aplikacji internetowych są następujące [1]:

- w sieci Internet witryna WWW jest ważnym elementem wpływającym na wizerunek publiczny firmy,
- w zasobach serwera WWW mogą znajdować się dane zawierające strategiczne dane o firmie,
- intruz, któremu udało się przypuścić skuteczny atak na serwer WWW firmy, może pozyskać dane, które mogą być wykorzystane podczas ataku na serwery produkcyjne firmy,
- brak dostępu do usługi WWW może powodować straty materialne,

- usuwanie szkód wyrządzonych przez intruza jest kosztowne i wymaga czasu,
- jeśli serwer firmy został przejęty i wykorzystany do ataku na inny obiekt, to taka firma może zostać pociągnięta do odpowiedzialności.

W różnych źródłach można znaleźć różniące się od siebie definicje pojęcia system portalowy. W dalszej części opracowania pojęcie to będzie rozumiane następująco.

> ***System portalowy*** *jest to zbiór połączonych funkcjonalnie, bądź też współpracujących na innej płaszczyźnie, aplikacji internetowych realizujących zadania związane z udostępnianiem danych.*

W większości przypadków wnioski z dyskusji na temat bezpieczeństwa danych udostępnianych w sieci Internet poprzez systemy portalowe stwierdzają wyższość produktów komercyjnych nad rozwiązaniami otwartymi - tzw. open source. Głównymi argumentami są zazwyczaj: wsparcie ze strony producenta oraz niedostępność kodu źródłowego dla osób postronnych. W rzeczywistości bezpieczeństwo danych dużo bardziej zależne jest od sposobu implementacji poszczególnych elementów aplikacji i rozwiązania problematycznych kwestii związanych z pobieraniem danych od użytkownika (ich poprawności, wiarygodności, itd.). W chwili obecnej ataki na systemy portalowe wykonywane zarówno przez osoby doświadczone, nowicjuszy (na podstawie dostępnych w zasobach sieci Internet przewodników!) oraz tzw. robaki sprowadzają się zazwyczaj do wykorzystania jednej z poniższych metod:

- modyfikacja danych w URI[1],
- modyfikacja danych przesyłanych w formularzu,
- modyfikacja danych w bazie danych.

Należy zwrócić uwagę, iż dobór metody ataku na aplikację internetową zależy głównie od celu, który postawił sobie agresor. Zarówno w przypadku prób kompromitacji ofiary (np. przez modyfikację treści), jak i spowodowania strat poprzez zablokowanie określonej usługi, napastnik wykorzysta odmienne metody ataku. W dalszej części artykułu przedstawione zostaną wybrane rozwiązania zmniejszające ryzyko skutecznego zastosowania wymienionych metod ataku.

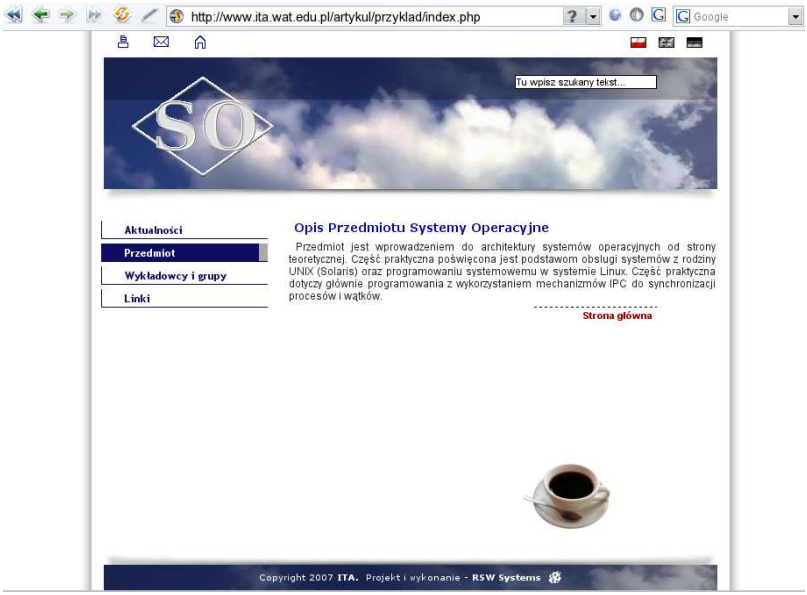
[1] URI (ang. Uniform Resource Identifier) jest standardem internetowym umożliwiającym identyfikację zasobów w sieci. Zdefiniowany jest w dokumencie RFC 2396. URI składa się z URL (ang. Uniform Resource Locator) i URN (ang. Uniform Resource Name)."[4]

## 2. Modyfikacja adresu URL

Podstawowym atakiem występującym w przypadku aplikacji internetowych jest modyfikacja adresu URL[2], czyli adresu strony WWW. Technika ta opiera się na tym, że zawartość URL jest jawna (np. po wybraniu łącza hipertekstowego aktualna postać URL jest widoczna w pasku adresu). Przykładowa postać adresu URL została przedstawiona poniżej:

```
http://www.ita.wat.edu.pl?sid=ASDFA&des=false
|_____________________________|_________________|
   lokalizacja zasobów    |  QUERY_STRING
```

Dla zobrazowania sytuacji rozważmy przykład witryny znajdującej się pod adresem
`http://www.ita.wat.edu.pl/artykul/przyklad/index.php`
Zawartość tej witryny jest pokazana na rysunku 1.



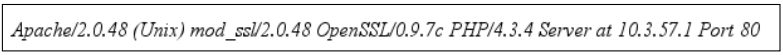

**Rys. 1. Zawartość przykładowej strony *WWW***

[2] URL (ang. Uniform Resource Locator) - komponent URI wskazujący położenie zasobów w sieci i sposób dostępu do tych zasobów.

Po wprowadzeniu drobnej modyfikacji do adresu URL polegającej na usunięciu członu `/przyklad/index.php` tak, aby adres miał postać: `http://www.ita.wat.edu.pl/artykul/` otrzymujemy dostęp do danych niepublicznych. Zasoby, do których w ten sposób można uzyskać dostęp zostały pokazane na rysunku 2.


**Rys. 2. Zawartość przykładowej strony *WWW* po wprowadzeniu zmodyfikowanego adresu *URL***

Szkody, które może spowodować nieuprawniony dostęp do danych przechowywanych w zasobach serwera WWW są oczywiste. Jednak nie są to wszystkie konsekwencje. Intruz, który uzyskał dostęp do danych w pokazany sposób, otrzymuje często jeszcze inne cenne informacje, np. informacje o konfiguracji wykorzystywanego serwera WWW. Dane te dostępne są w dolnej części strony wyświetlanej po modyfikacji adresu URL (przykład pokazany jest w dolnej części rysunku 2 i na rysunku 3).




**Rys. 3. Informacje o serwerze *WWW* po wprowadzeniu zmodyfikowanego adresu *URL***

Warto także zwrócić uwagę na to, że przy nieprawidłowej konfiguracji serwera WWW mogą zostać udostępnione informacje o strukturze wewnętrznej sieci instytucji, w której zasobach ten serwer się znajduje (w tym przypadku

można dodatkowo odczytać adres IP serwera i port, na którym działa usługa, oraz wersje zainstalowanego oprogramowania).

W celu uniknięcia problemów związanych z modyfikacją URL można zastosować następujące procedury:

- poprawna konfiguracja serwera,
- kodowanie adresu URL,
- szyfrowanie adresu URL.

Istnieje kilka sposobów poprawnego skonfigurowania serwera WWW. Skutek tych działań powinien być taki, jak zastosowanie pokazanej na rysunku 4 dyrektywy określającej opcje dla wykorzystywanego katalogu przez serwer WWW. Dyrektywa ta blokuje możliwość listowania zawartości wskazanego katalogu i jego podkatalogów w przypadku braku pliku, którego nazwa zaczyna się od słowa „index" np. **`index.htm`**

```
<Directory /web/public_html/old>
     :
   Options –Indexes
     :
</Directory>
```

**Rys. 4. Przykładowa postać dyrektywy blokującej listowanie zawartości katalogu**

W dalszej części opracowania będą przedstawione pozostałe z wymienionych wcześniej procedur.

## 2.1. Kodowanie adresu URL

Najprostszym rozwiązaniem problemu modyfikacji przez użytkowników lub potencjalnych agresorów postaci adresu URL jest zakodowanie go, tzn. zastąpienie tekstu w kodzie ASCII jego postacią szesnastkową. W tym przypadku aplikacja internetowa w trakcie generowania kodu strony HTML przy tworzeniu łącza hipertekstowego powinna oryginalną postać adresu URL wskazującą zasób serwera zamienić na jej postać szesnastkową. Przykładowa funkcja realizująca to zadanie zapisana w języku Perl [[2],[3]]została przedstawiona poniżej:

```
sub get_index {
    # pobranie parametrów: podstawowy url
    # oraz docelowa lokalizacja
    my ($base_index, $location) = @_;
```

```
# zapisanie miejsca docelowego pod
# zmienna "location" w sekwencji GET
$location = 'location='.$location;

# zakodowanie sekwencji GET
$location = unpack "h*", $location;

# zwrocenie wartosci
return $base_index."?SID=$location";  }
```

Wynikiem działania powyżej procedury jest następująca postać adresu URL:

```
http://www.ita.wat.edu.pl?SID=AEF23E34F10A...
```

Dodatkowym zabezpieczeniem jest wykorzystanie ciągu znaków *"SID="* sugerujące, iż dalej następujący ciąg znaków jest identyfikatorem sesji. Przykład fragmentu strony, w której zastosowano takie kodowanie jest pokazany na rysunku 5. Zakodowane adresy URL w kolejnym kroku będą wysyłane przez użytkownika do serwera WWW.


**Rys. 5. Strona WWW ze zmodyfikowanym adresem URL**

Aplikacja internetowa po otrzymaniu adresu URL powinna być w stanie odtworzyć właściwe dane i wyświetlić właściwą stronę. Zadanie to może realizować poniższa procedura.

```
sub get_location {
    # pobranie parametru - adresu URL
    my ($url) = @_;

    # wyłuskanie ciągu heksadecymalnego
    # za pomocą wyrazenia regularnego
    $url =~ /SID=([0-9A-F]+)$/;
    my $get = $1;

    # zwrocenie bledu w przypadku blednego wyluskania
```

```
return undef unless $get;

# przekodowanie do postaci ASCII
$get = pack "h*", $get;

# zwrocenie bledu w przypadku blednego odkodowania
return undef unless $get;

# usuniecie czlonu "location="
$get =~ s/^location=//;

# zwrocenie wartosci zmiennej "location"
return $get;  }
```

## 2.2. Szyfrowanie adresu URL

Kodowanie adresu URL jest rozwiązaniem bardzo prostym i zdecydowanie wystarczającym w przypadku tzw. nowicjuszy (zwłaszcza, jeśli wykorzystamy inne metody kodowania, nie tylko na postać heksadecymalną). Niestety, jeśli atak przeprowadza osoba z dużym doświadczeniem, to metoda ta jest zupełnie nieskuteczna. Logicznym rozwinięciem idei kodowania adresów URL jest ich szyfrowanie. Ilość bibliotek umożliwiających szyfrowanie jest imponująca, zatem wybór nie powinien stanowić problemu zwłaszcza, że duża część z nich jest udostępniana na zasadach *Open Source*.

Rozszerzenie wcześniej zaprezentowanych procedur o możliwość szyfrowania adresów algorytmami symetrycznymi sprowadza się do dodatkowego poddania adresu URL działaniu funkcji szyfrującej. Poniżej znajduje się przykład tego typu procedury. Zastosowano w niej algorytm szyfrowania AES w trybie *cbc* oraz klucz *my_passwd*. Warto zaznaczyć, że darmowa biblioteka *libGCrypt* udostępnia zdecydowaną większość popularnych algorytmów szyfrujących (np. 3DES) działających w różnych trybach.

```
sub encrypt {
    # pobranie tekstu do zaszyfrowania
    my ($text) = @_;

    # powolanie szyfratora
    use Crypt::GCrypt;
    my $cipher = Crypt::GCrypt->new(
                        type => 'cipher',
                        # wybor algorytmu
                        algorithm => 'aes',
                        # wybor trybu dzialania
                        mode => 'cbc');
```

```
        $cipher->start('encrypting');
        # ustawienie hasla
        $cipher->setkey('my_password');

        # zaszyfrowanie textu
        my $ciphertext = $cipher->encrypt($text);
        $ciphertext .= $cipher->finish;

        # zwrocenie zaszyfrowanego tekstu
        return $ciphertext;
    }
```

Deszyfrowanie tak przygotowanego adresu URL również nie jest kłopotliwe. Do tego celu można wykorzystać poniższą procedurę.

```
    sub decrypt {
        # pobranie tekstu do zaszyfrowania
        my ($ciphertext) = @_;

        # powolanie szyfratora
        use Crypt::GCrypt;
        my $cipher = Crypt::GCrypt->new(
                            type => 'cipher',
                            algorithm => 'aes',
                            mode => 'cbc');
        $cipher->start('decrypting');

        # ustawienie hasla
        $cipher->setkey('my_password');

        # odszyfrowanie textu i zwrocenie wyniku
        return $cipher->decrypt($ciphertext);   }
```

## 2.3. Kompresja

Hiperłącza przygotowane w opisany powyżej sposób stanowią poważną przeszkodę przy wszelkiego rodzaju próbach ataku poprzez modyfikację adresu URL witryny. Szyfrowanie adresu URL w hiperłączach ma dwie poważne wady. Pierwszą z nich jest zdecydowanie większe zapotrzebowanie na zasoby serwera, niż w przypadku wyświetlania adresów w postaci jawnej. Drugą jest długość wynikowego ciągu URL. Ze względu na szyfrowanie, jego długość znacznie się zwiększa i w przypadku starszych przeglądarek mających nałożone ograniczenie na łańcuch *QUERY_STRING* do 255 znaków przeglądanie strony staje się niemożliwe. Programista przygotowujący aplikację internetową nie ma

znaczącego wpływu na zmniejszenie zapotrzebowania na zasoby serwera[3], natomiast może zmniejszyć długość wynikowego ciągu URL przez zastosowanie kompresji. Ciąg *QUERY_STRING* poddany szyfrowaniu składa się prawie wyłącznie ze znaków drukowalnych, zatem doskonałe efekty daje wykorzystanie algorytmów kompresji tekstu. Przykładem jest algorytm *PPMd*, który umożliwia niemal trzykrotne zmniejszenie długości łańcucha poddanego jego działaniu. Poniżej są pokazane procedury realizujące operacje kompresji i dekompresji przy użyciu wymienionego algorytmu.

```
# kompresja
sub compress {
    my ($str) = @_;

    # kompresowanie wejsciowego lancucha
    use Compress::PPMd;
    my $encoder = Compress::PPMd::Encoder->new();
    my $cmps = $encoder->encode($str);

    # zwrocenie skompresowanego lancucha
    return $cmps;
}
# dekompresja
sub decompress {
    my ($cmps) = @_;

    # dekompresowanie wejsciowego lancucha
    use Compress::PPMd;
    my $decoder = Compress::PPMd::Decoder->new();
    my $str = $decoder->decode($cmps);

    # zwrocenie lancucha
    return $str;   }
```

## 3. Modyfikacja treści formularzy

Do przesyłania danych pomiędzy klientem, a serwerem usługi WWW wykorzystywany jest protokół HTTP. Do wysyłania danych od klienta do serwera (np. danych wypełnionego formularza) mogą być wykorzystane metody *GET* lub *POST*. W metodzie *GET* dane są przesyłane w pierwszej linii komunikatu HTTP w postaci *URN*, który jest częścią *URI* (rysunek 6). Natomiast w metodzie *POST* dane są przesyłane w zasadniczej części

[3] Jedyną możliwością zmniejszenia wykorzystania zasobów serwera w przypadku szyfrowania jest odpowiednie dobranie algorytmu kryptograficznego.

komunikatu HTTP (rysunek 7).

```
GET /index.sdy?par1=aaa&par2=bbb HTTP/1.0
User-Agent: ....
Accept: ....
Content-type: ....
Content-lenght: ....
```

**Rys. 6. Sposób przekazywania danych metodą *GET***

```
GET /index.sdy HTTP/1.0
POST /index.sdy HTTP/1.0
User-Agent: ....
Accept: ....
Host: ....
Content-type: ....
Content-lenght: ....

login=zbigniew&passwd=qaz123&zatwierdz=Zatwierd\u017a
```

**Rys. 7. Sposób przekazywania danych metodą *POST***

Serwery WWW interpretują odebrane komunikaty HTTP w taki sposób, że na początku, niezależnie od wykorzystanej metody *GET*, czy *POST*, interpretowana jest pierwsza linia komunikatu. Jeżeli w tej linii występuje niepusty ciąg *URN* to jego zawartość jest wpisywana do zmiennej środowiskowej *QUERY-STRING*. Następnie, jeżeli dane są przekazywane metodą *POST*, to interpretowana jest zasadnicza część komunikatu. Bardzo często twórcy interaktywnych stron budują formularze tak, że część danych jest przesyłana w pierwszej linii komunikatu, a pozostała część w jego zasadniczej części. Wycinek takiego dokumentu HTML jest pokazany na rysunku 8, a odebrany komunikat może mieć postać podaną na rysunku 9.

Jednym z ataków na serwery WWW może być modyfikacja formularzy witryny WWW. Atak ten może polegać na tym, że w danych formularza wprowadza się dane zmieniające działanie formularza (tzw. ciągi formatujące) albo modyfikuje się dane zawarte w samym formularzu. Problemy wykorzystania przez agresora ciągów formatujących zostały poruszone w książce [1].

```
<form method="post"
action="https://www.ita.wat.edu.pl/index.sdy?sid=
                                  40608c809843d9cd23">
    <input type="text" name="login" />
    <input type="password" name="passwd" />
    <input type="submit" name="zatwierdz"
                                value="Zatwierd\u017a" />
</form>
```

**Rys. 8. Wycinek dokumentu HTML**

```
POST /index.sdy?sid=40608c809843d9cd23 HTTP/1.0
User-Agent: ....
Accept: ....
Host: ....
Content-type: ....
Content-lenght: ....

login=zbigniew&passwd=qaz123&zatwierdz=Zatwierd\u017a
```

**Rys. 9. Sposób przekazywania danych metodą *POST***

W pierwszym przypadku możliwym jest, że użytkownik podał niewłaściwe informacje, natomiast w drugim istnieje pewność, że nastąpiła próba ataku na aplikację internetową.

## 3.1. Ciągi formatujące

Atak przeprowadzony za pomocą ciągów formatujących występuje w przypadku wprowadzenia przez użytkownika do pól formularza danych niepoprawnych mogących zmienić standardowe zachowanie aplikacji internetowej przetwarzającej te dane. Jeśli do przetwarzania danych wykorzystywana jest zewnętrzna aplikacja lub skrypt CGI uruchamiany z poziomu powłoki, za atak rozumiane będzie np. wymuszenie uruchomienia dodatkowych programów podczas wywołania systemowego w portalu. Dopuszczenie do wystąpienia opisywanej sytuacji może być tragiczne w skutkach. Dla przykładu rozważmy poniższy kod:

```
# pobranie z formularza zmiennej "text"
my $text = form_get('text');
```

```
# wykonanie w powloce polecenia i
# umieszczenie wyniku w zmiennej $fmt_text
my $fmt_text = `echo $text | fmt`;
```

W przypadku podania przez użytkownika poprawnego łańcucha znaków, zostanie on sformatowany. Jednak, gdy wprowadzony przez użytkownika tekst będzie miał postać:

```
to jest poprawny tekst; rm -Rf /*
```

to do zmiennej **`$text`** będzie przypisany tekst "**`to jest poprawny tekst; rm -Rf /*`**", a następnie podczas próby wykonania polecenia **`echo $text | fmt`** na pobranych danych, zostanie wykonana najpierw komenda **`echo`**, a potem nastąpi usunięcie z systemu plików wszelkich danych (**`rm -Rf`**), do których proces serwera WWW posiada uprawnienia. Na końcu wynik wykonanej operacji zostanie przekazany do polecenia **`fmt`**. Dzieje się tak z powodu wykorzystania w ciągu znaków znaku '**`;`**', który w większości powłok systemowych rozpoznawany jest jako separator komend.

Podstawowym sposobem uniknięcia tego typu sytuacji jest monitorowanie danych wprowadzanych przez użytkownika. Dodatkowo do tzw. *najlepszych praktyk* należy tworzenie filtrów bazujących nie na liście operacji niedozwolonych, lecz na zbiorze operacji dozwolonych. W tym przypadku wszystkie dane pochodzące od użytkownika, które nie są zgodne z założonymi wartościami powinny być odrzucane. Ciekawym rozwiązaniem jest tzw. *system kontroli skażeń*, dostępny między innymi w języku Perl, wymuszający na programiście kontrolę wszelkich danych zewnętrznych. Idea jego działania polega na założeniu, iż wszystkie dane pochodzące ze środowiska zewnętrznego względem aplikacji, a także dane, które miały z nimi kontakt (np. są wynikiem działania, w którym te dane występowały) są *skażone* i nie mogą być wykorzystywane do potencjalnie niebezpiecznych operacji takich, jak usuwanie. Wykorzystanie danych skażonych możliwe jest dopiero po przeprowadzeniu tzw. *odkażania* - czyli sprawdzenia poprawności. Dzięki zastosowaniu tego mechanizmu nawet przypadkowe wystąpienie operacji na niepewnych danych zostanie wykryte.

## 3.2. Modyfikacja danych w formularzu

W przypadku witryn umożliwiających wykonywanie jakichkolwiek operacji finansowych modyfikowanie danych pobieranych od użytkownika za pomocą formularza jest szczególnie niebezpieczne. Mowa tu nie tylko o parametrach przekazywanych za pomocą metody *GET*, czy też pól pozostawionych do tzw. otwartej edycji (pola tekstowe), lecz także o elementach, które teoretycznie nie powinny przyjmować wartości

niepoprawnych. Do takich elementów należą zmienne przekazywane jako *hidden* lub listy wyboru. Jakakolwiek gwarancja, że dane znajdujące się we wspomnianych polach będą poprawne opiera się na przekonaniu, że przeglądarka internetowa użytkownika nie pozwoli na ich zmianę. Bazowanie na tym założeniu jest błędne, gdyż istnieją programy przeznaczone do wysyłania danych z formularzy umożliwiające dowolne modyfikowanie wszystkich parametrów. Oznacza to, że np. w przypadku sklepu internetowego jawne podanie parametrów określających cenę i produkt, bez późniejszej kontroli poprawności może zakończyć się manipulacjami ze strony użytkowników o niepożądanych skutkach.

Podstawowym rozwiązaniem stosowanym w większości serwisów internetowych jest wykorzystanie kondensatów[4] po stronie serwera. Technika ta polega na wykorzystaniu algorytmów mieszających do wytworzenia i zachowania w zasobach serwera skrótów informacji zamieszczonych w formularzu WWW (np. cena i rodzaj produktu) oraz sprawdzeniu na ich podstawie wprowadzonych przez użytkownika danych. Dla przykładu załóżmy, że skrypt powinien wysyłać do użytkownika formularz z polami zawierającymi następujące dane:

- nazwa produktu,
- cena produktu.

Pole *Nazwa Produktu* jest implementowane jako lista wyboru, a pole *Cena Produktu* jest statycznie z nim połączone i zmienia się ilekroć użytkownik zaznaczy inny towar. Projektowany program przed wysłaniem takiego formularza do przeglądarki klienta powinien obliczyć i zapisać kondensat każdej pary produkt - cena, na przykład za pomocą algorytmu MD5. Przykład procedury realizującej takie zadanie został pokazany poniżej:

```
sub save_it {
    # pobranie parametrow
    my ($cena,$nazwa) = @_;

    # obliczenie kondensatu
    use Digest::MD5 qw(md5_hex);
    my $cond = md5_hex($cena,$nazwa,'tajna fraza');

    # zapisanie kondensatu
    save($cond, TRUE);
}
```

[4] Kondensat (skrót, sygnatura) - wynik działania funkcji mieszającej

Po odebraniu danych od użytkownika należy sprawdzić, czy nie nastąpiła modyfikacja wysłanych parametrów formularza. Dokonać tego można przy użyciu następującej procedury:

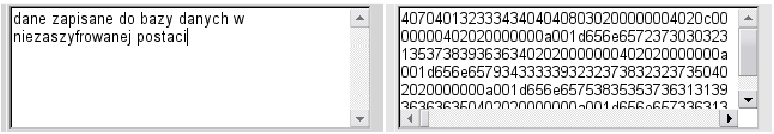
```
sub check_it {
    # pobranie parametrow
    my ($cena_wybrana, $nazwa_wybrana) = @_;

    # obliczenie nowego kondensatu
    # jesli uzytkownik nic nie zmienil, to bedzie taki
    # sam jak $cond obliczony przed wyslaniem danych
    use Digest::MD5 qw(md5_hex);
    my $cond_new =
  md5_hex($cena_wybrana,$nazwa_wybrana,'tajna fraza');

    # sprawdzenie istnienia obliczonego kondensatu
    my $val = restore($cond_new);

    # sprawdzenie poprawnosci danych
    # zmienna $val moze byc niezdefiniowana jedynie w
    # przypadku, gdy $cond_new nie jest rowny $cond
    # czyli, gdy uzytkownik wprowadzil w formularzu
    # zmiany
    if(!$val){
        return FALSE;
    }
    return TRUE;
}
```

## 4. Modyfikacja bazy danych

W przypadku, gdy aplikacja internetowa nie posiada błędów umożliwiających jej niepoprawne wykorzystanie (np. poprzez zastosowanie ciągów formatujących), celem ataku może stać się baza danych, której wspomniane oprogramowanie używa. Sytuacja jest niebezpieczna, gdyż możliwa jest kompromitacja instytucji korzystającej z określonego oprogramowania, a także pociągnięcie do odpowiedzialności autora aplikacji, jeśli jego projekt umożliwił wystąpienie takiego zdarzenia. W celu uniknięcia tego typu problemów można zastosować szyfrowanie i serializację danych w bazie danych.

## 4.1. Szyfrowanie danych

Procedurę szyfrowania należy przeprowadzić przed zapisem do bazy danych (rysunek 10). Należy jednak pamiętać, że jest to operacja kosztowna w sensie wykorzystywanych zasobów serwera i w niektórych wypadkach może znacząco wpłynąć na szybkość działania witryny WWW.




**Rys. 10. Porównanie danych w postaci jawnej i zaszyfrowanej w bazie danych**

Podobnie jak w przypadku adresów URL, do szyfrowania danych może zostać wykorzystana biblioteka *libGCrypt*. Dostępna liczba obsługiwanych algorytmów szyfrowania jest wystarczająca, aby znaleźć kompromis pomiędzy wydajnością i bezpieczeństwem przetwarzanych danych. Kod, który umożliwia wykonanie szyfrowania i deszyfrowania danych przy wykorzystaniu wymienionej biblioteki został już zaprezentowany podczas omawiania problematyki szyfrowania adresów URL.

## 4.2. Serializacja danych

Po dokonaniu analizy sposobu działania aplikacji (głównie internetowych) korzystających z bazy danych można określić następujący schemat postępowania występujący w większości programów:

a) utworzenie struktury danych,
b) pobranie z bazy danych odpowiednich wartości i umieszczenie ich we wcześniej utworzonej strukturze danych,
c) wykonanie operacji wykorzystujących utworzoną strukturę,
d) oddzielne zapisanie do bazy danych poszczególnych zmiennych ze struktury, jeśli podczas przetwarzania uległy modyfikacji.

Standardowe szyfrowanie danych zapisywanych i odczytywanych z bazy danych w aplikacji działającej zgodnie z powyższym schematem byłoby wysoce nieefektywne. Operacje szyfrowania i deszyfrowania dla każdej zmiennej znacznie spowalniałyby działanie, a w wielu przypadkach wręcz uniemożliwiały zapewnienie wydajności serwisu internetowego na akceptowalnym poziomie. Jednak dzięki zastosowaniu zaawansowanych technik, charakterystycznych dla

nowoczesnych systemów CMS, możliwe jest uzyskanie dużej wydajności aplikacji, przy jednoczesnym zachowaniu wysokiego poziomu bezpieczeństwa danych. Podstawą wspomnianych specyficznych metod zarządzania danymi jest spostrzeżenie, że większość spośród danych wczytywanych do struktury występującej we wcześniej wspomnianym schemacie jest wykorzystywana jednocześnie. Zatem logicznym posunięciem byłoby zapisywanie ich w formie łącznej w bazie danych i wczytywanie w ten sam sposób. Osiągnięcie omawianego efektu umożliwia serializacja danych, czyli zapisywanie całych struktur danych w postaci pojedynczego łańcucha znaków, co prezentuje poniższy przykład.

```
# serializacja danych
use Storable;

# deklaracja tablicy asocjacyjnej
my %hash = ( name    => 'Zbigniew',
             surname => 'Kowalski',
             age     => 21,
);

# konwersja struktury danych do
# postaci skalara za pomocą funkcji "freeze"
my $string = freeze \%hash;

#######################################

# deserializacja danych
use Storable;

# przywrocenie do pamieci struktury
# danych uprzednio zmienionej w skalar
# i pobranie na nia referencji przy
# wykorzystaniu funkcji "thaw"
my $hash_ref = thaw $string;
```

Zysk na wydajności osiągnięty w ten sposób całkowicie niweluje obciążenie związane z szyfrowaniem i deszyfrowaniem danych.

## 4.3. Kompresja danych

Rozważając poszczególne aspekty działania aplikacji i związane z nimi fakty należy zwrócić uwagę na poniższe zależności:

1. Odczytanie danych poddanych serializacji wymaga znajomości nie tylko wykorzystywanego języka programowania i postaci zapisanej

struktury danych, ale także biblioteki użytej podczas serializacji. Oznacza to, że zserializowane dane poddane są zaawansowanemu kodowaniu.

2. Pobieranie danych z bazy danych jest operacją stosunkowo powolną. Zatem, aby zwiększyć szybkość trzeba dążyć do ograniczenia połączeń z bazą danych oraz zmniejszenia wielkości danych pobieranych z bazy danych. Oznacza to, iż w wielu przypadkach dane przechowywane w skompresowanej formie w bazie danych będą pobierane przez aplikację szybciej.

W związku z powyższym, logicznym rozwiązaniem jest połączenie serializacji danych z ich kompresją. Biorąc pod uwagę fakt, iż algorytmy kompresujące również zapewniają dodatkową metodę kodowania danych, efektem takiego posunięcia będzie zapewnienie wysokiego bezpieczeństwa zapisanych danych, przy jednoczesnym zachowaniu dużej skuteczności. W takim przypadku dodatkowe szyfrowanie nie wydaje się być konieczne. Należy także zwrócić uwagę na fakt, iż osiągnięte w ten sposób zwiększenie przepustowości serwera stanowi zabezpieczenie dostępności, czyli jednego z podstawowych założeń bezpieczeństwa.

Procedura, która umożliwia wykonanie opisywanych operacji, tzn. serializacji, kompresji i kodowania, została przedstawiona poniżej.

```
# przygotowanie danych do zapisu
sub save_data {
    # pobranie parametru,czyli referencji na strukture
    my ($struct_ref) = @_;

    # serializacja struktury
    use Storable;
    my $string = freeze $struct_ref;

    # kompresowanie lancucha znakow
    use Compress::PPMd;
    my $encoder = Compress::PPMd::Encoder->new();
    my $cmps = $encoder->encode($string);

    # zakodowanie w do postaci heksadecymalnej
    my $hex = unpack "h*", $cmps;

    # zwrocenie wartosci wynikowej
    return $hex;
}

#######################################
# odczytanie danych
```

```
sub retrieve_data {
    # pobranie parametru tzn lancucha haksadecymalnego
    my ($hex) = @_;

    # odkodowanie z postaci
    # heksadecymalnej
    my $cmps = pack "h*", $hex;

    # dekompresowanie lancucha znakow
    use Compress::PPMd;
    my $decoder = Compress::PPMd::Decoder->new();
    my $string = $decoder->decode($cmps);

    # deserializacja struktury
    use Storable;
    my $struct_ref = thaw $string;

    # zwrocenie wartosci wynikowej,
    # czyli referencji na strukture
    return $struct_ref;     }
```

## 5. Podsumowanie

Znaczenie portali i witryn internetowych dla firm (w wielu wypadkach strategiczne) czyni z nich dogodny cel ataków zwłaszcza, że są ogólnie dostępne. Często jednak podatności na poszczególne rodzaje ataków mogą zostać w stosunkowo prosty sposób zminimalizowane. W artykule ukazane zostały podstawowe rodzaje ataków na zasoby instytucji zamieszczone w Internecie wraz z propozycjami obrony przed tymi atakami. Przedstawione propozycje zwykle stanowią przeszkodę nie do przebycia dla większości agresorów - nawet tych zaawansowanych.

## Literatura

[1] Guelich S., Gundavaram S., Birznieks G., *Programowanie CGI w Perlu*, Wydawnictwo RM, Warszawa 2000.

[2] Wall L., Christiansen T., Orwant J., *Perl - Programowanie*, Wydanie drugie, Wydawnictwo RM, Warszawa 2001.

[3] Siegel D., *Tworzenie stron WWW*, Wydawnictwo Optimus Pascal Multimedia, Bielsko-Biała 1998.

[4] http://pl.wikipedia.org, dostępne hasła na dzień 19.12.2006.

# Discarding methods of URL vulnerability on Internet applications attacks

ABSTRACT: In article the Internet application selected security problems related to preserving data and receiving data from user was presented. Several sample solutions decreasing possibility to carrying efficacious portal system attack was described.

KEYWORDS: URL vulnerability, URL encryption, Internet applications security

*Recenzent: dr hab. inż. Antoni Donigiewicz*

*Praca wpłynęła do redakcji 29.12.2006*