JOURNAL of COMPUTER SCIENCES INSTITUTE

JCSI 6 (2018) 12-19
Wysłane: 2017-09-14
Przyjęte: 2017-09-18

Analiza wpływu technik bezpiecznego programowania na wydajność i bezpieczeństwo aplikacji

Tomasz Kobiałka
Politechnika Lubelska, Instytut Informatyki, Nadbystrzycka 36B, 20-618 Lublin, Polska

Streszczenie. Tematyką podejmowaną w niniejszym artykule są zagrożenia, które należy wziąć pod uwagę podczas tworzenia oprogramowania. W ramach artykułu przybliżono wybrane rodzaje zabezpieczeń przed często wykorzystywanymi lukami bezpieczeństwa. W oparciu o napisane programy, przeanalizowano wpływ poszczególnych technik bezpiecznego programowania na wydajność i bezpieczeństwo aplikacji. Niniejszy artykuł prezentuje zarówno teoretyczny opis zabezpieczeń, jak również przykłady ich technicznej implementacji.

Słowa kluczowe: programowanie; bezpieczeństwo; wydajność

Adres e-mail: tskobialka@gmail.com

An analysis of influence of safe programming techniques on applications efficiency and security

Tomasz Kobiałka
Institute of Computer Science, Lublin University of Technology, Nadbystrzycka 36B, 20-618 Lublin, Poland

Abstract. The topics covered in this article are the risks that must be taken into account when developing the software. This article gives you an overview of safeguards against some of the anticipated common security vulnerabilities. Based on the written programs, the impact of the various techniques of safe programming on the performance and security of the application has been analyzed. This article presents both a theoretical description of the protections as well as examples of their technical implementation.

Keywords: programming; efficiency; security

E-mail address: tskobialka@gmail.com

1. Wstęp

Tematem artykułu są techniki bezpiecznego programowania oraz analiza ich wpływu na wydajność i bezpieczeństwo aplikacji.

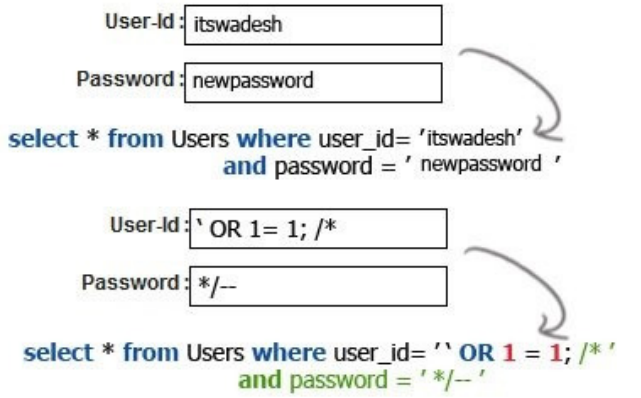
W dzisiejszych czasach osiągnięcie sukcesu w branży programistycznej wiąże się z wytężoną pracą wielu ludzi. Zazwyczaj jest to równoznaczne z poniesieniem dużych nakładów finansowych. Zarówno pieniądze, jak i praca mogą pójść na marne, jeżeli tworzone oprogramowanie nie zostanie dostatecznie zabezpieczone – chociażby przed osobami, które będą umożliwiały korzystanie z produktu w sposób nieautoryzowany. Pod uwagę należy wziąć także możliwość osłabienia wizerunku producenta, jeżeli osoba z zewnątrz zdecyduje się rozpowszechniać niebezpiecznie zmienione programy, nie wspominając o tym, że oprogramowanie – często służące do realizacji celów strategicznych, bądź celów użyteczności publicznej – może zostać zaatakowane przez osoby o złych zamiarach. Jako przykład można podać oprogramowanie medyczne, albo obsługujące bankowość lub lotnictwo. Atak osoby trzeciej może narazić użytkowników na straty nie tylko finansowe, ale także zdrowotne, a w skrajnych przypadkach nawet utratę życia. Każdy z tych przypadków nasuwa myśl o tym, że warto należycie zabezpieczyć stworzony program.

Głównym celem bezpiecznego programowania jest wyeliminowanie możliwości użycia programu niezgodnie z jego przeznaczeniem. Jednakże, oprogramowanie o zbyt restrykcyjnych zabezpieczeniach znacząco zmniejsza swoją użyteczność dla zwykłych użytkowników. Dlatego należy tak dopasować używane techniki, aby znaleźć kompromis pomiędzy zapewnieniem bezpieczeństwa, a pozostawieniem swobody końcowemu odbiorcy.

2. Zagrożenia

2.1. Wstrzykiwanie kodu SQL

Wstrzykiwanie kodu SQL (ang. SQL injection) jest techniką używaną do atakowania aplikacji, których kluczowym komponentem jest baza danych i dane na niej zgromadzone. Technika polega na wprowadzeniu do pola wejściowego aplikacji takiego zapytania SQL, którego wynikiem będzie działanie pożądane przez osobę atakującą, a potencjalnie niebezpieczne dla aplikacji i danych, z których korzysta. SQL injection wykorzystuje luki w bezpieczeństwie aplikacji, np. brak filtrowania danych wejściowych, tudzież brak ustalania konkretnych typów dla każdego wyrażenia w kodzie aplikacji (typowanie słabe), a w konsekwencji powstanie możliwości wykonania wstrzykiwanego zapytania. Atakujący może sfałszować tożsamość, usunąć lub zmienić dane w bazie oraz naruszyć ich integralność, a nawet przyznać sobie uprawnienia administratora bazy. Badania wykazały, że przeciętna aplikacja webowa jest narażona na średnio 4 ataki tego typu w miesiącu, a prób wstrzyknięcia SQL aplikacjom sklepów internetowych jest dwa razy więcej niż innym aplikacjom [1]. Na rysunku 1. zaprezentowano przykładową próbę wstrzyknięcia kodu SQL.

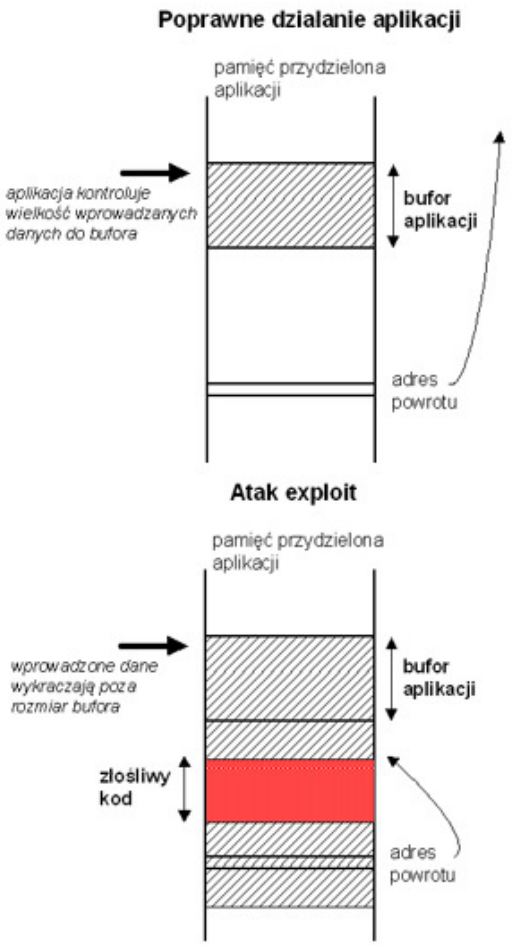


Rys 1: Przykładowa próba wstrzyknięcia kodu SQL

2.2. Przepełnienie bufora

Przepełnienie bufora (ang. buffer overflow) jest anomalią, w której program podczas zapisu danych do bufora przekracza granice pamięci bufora i nadpisuje sąsiadujące lokacje pamięci. Błąd tego typu najczęściej jest wywoływany przez zniekształcone wejścia; jeżeli podczas powstawania programu założono, że wszystkie wejścia będą mniejsze niż pewna wartość i pojemność bufora jest ustawiona dokładnie na tę wartość, nieprawidłowa transakcja, która produkuje więcej danych niż przewidziano powoduje zapis danych poza zakresem pamięci bufora. W momencie, w którym sąsiadujące w pamięci z buforem dane lub kod wykonywalny są nadpisywane, program najczęściej zaczyna zachowywać się niestabilnie, następują błędy w dostępie do pamięci, dane wynikowe są błędne, a często program przedwcześnie kończy swoje działanie [2]. Schemat poglądowy takiej sytuacji został zaprezentowany na rysunku 2. Atakujący powoduje przepełnienie bufora i zmienia adres powrotu. Zamiast powrotu do właściwego wywołania procedury, zmodyfikowany adres oddaje sterowanie złośliwemu kodowi, mieszczącemu się w innym miejscu pamięci procesu.

Wykorzystywanie przepełnienia buforu do przeprowadzania ataków jest powszechnie stosowaną przez hakerów praktyką. W wielu systemach układ pamięci programu, albo całego systemu, jest poprawnie zdefiniowany. Poprzez przesłanie danych mających na celu wywołanie przepełnienia, zapis w miejsca pamięci, mający na celu zastąpienie wykonywanego kodu złośliwym kodem staje się możliwy. Nowoczesne systemy operacyjne używają różnych technik by zwalczać złośliwe przepełnienia bufora, szczególnie poprzez randomizację układu pamięci, a także celowe zachowywanie przerw pomiędzy buforami i analizę zachowań, które powodują zapis danych do tych części pamięci, w których zaalokowany jest bufor. [3]

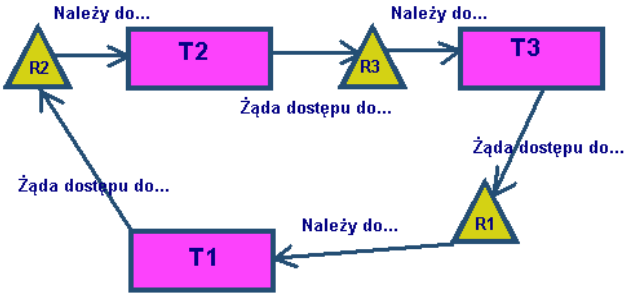


Rys 2: Bufor podczas normalnej pracy oraz w momencie wystąpienia ataku

2.3. Zakleszczenie

Zakleszczenie, inaczej blokada wzajemna (ang. deadlock) jest stanem, w którym każdy uczestnik grupy akcji czeka na zwolnienie blokady przez innego uczestnika. Zakleszczenie jest częstym problemem w systemach wieloprocesowych, programowaniu równoległym oraz obliczeniach rozproszonych, gdzie programowe i sprzętowe blokady są używane do zarządzania zasobami współdzielonymi i w implementacjach synchronizacji procesów. W systemach operacyjnych blokada wzajemna występuje kiedy proces lub wątek przechodzi w stan czekania, ponieważ żądane przez niego zasoby systemowe są wstrzymywane przez inny proces w stanie czekania, który z kolei czeka na dostęp do innych zasobów wstrzymywanych przez kolejny czekający proces. Zakleszczenie następuje wtedy, gdy wątek lub proces nie jest w stanie zmienić swojego stanu z powodu używania żądanych przez jego zasobów przez inny proces w stanie

oczekiwania [4][5].



Rys 3: Schemat poglądowy sytuacji wystąpienia zakleszczenia

2.4. Inżynieria wsteczna

Inżynieria wsteczna (ang. reverse engineering) to czynność polegająca na analizie oprogramowania, mająca na celu pozyskanie wiedzy o jego działaniu i budowie. Informacje te mogą zostać wykorzystane do pozyskania nieautoryzowanej wiedzy o oprogramowaniu i jego funkcjonowaniu, a także do jego modyfikacji. Wiedza ta może być wykorzystana do uzyskania z pozoru niedostępnych informacji o oprogramowaniu czy też jego modyfikacji.

Najczęstsze cele inżynierii wstecznej:
- odzyskiwanie i modyfikacja kodów źródłowych,
- odtwarzanie algorytmów szyfrowania i kompresji,
- analizowanie danych i algorytmów aplikacji po skompilowaniu,
- łamanie szyfrów zabezpieczonych plików (bazy danych, pliki konfiguracyjne),
- analizowanie protokołów sieciowych.

Inżynieria wsteczna to proces analizy aplikacji, której kod nie jest powszechnie dostępny. Dzięki niej, cracker jest w stanie odtworzyć i zrozumieć logikę programu, a także uzyskać wiedzę na temat jego kompletnego działania, która może być wykorzystana do znajdowania i korzystania z luk w programie. Patrząc pod kątem bezpieczeństwa, utrudnianie tego procesu potencjalnym osobom atakującym oprogramowanie jest bardzo ważne [6].

3. Techniki zabezpieczeń

3.1. SQL injection

Przyczyną ataków polegających na wstrzyknięciu kodu SQL jest zazwyczaj wykorzystanie parametru, który może zawierać dane wysłane przez osobę atakującą, bez odpowiedniego przefiltrowania tych danych. W ten sposób osoba trzecia jest w stanie wstrzyknąć fragment kodu, który zmieni wynik całego zapytania [7].

Przykład 1. Przykład skryptu przechwytującego dane z formularza

```
<?
include "db_connect.php";
$login = $_POST['login'];
$pass = $_POST['pass'];
$wynik = mysql_query("SELECT login FROM db_users WHERE
login='$login' AND
password=PASSWORD('$pass')");
if(mysql_num;rows(Swynik) > 0)  {
$rekord = mysql_fetch_array($wynik);
echo "LOGIN OK: ".$rekord['login'];
}
else {
echo "LOGIN FAILED";
include "form.html";
}
?>
```

Przykładowy skrypt PHP zaprezentowany w przykładzie 1 przechwytuje dane z formularza wysłanego przez użytkownika. Kod ten jest potencjalnie niebezpieczny – używa w zapytaniu danych uzyskanych od użytkownika z pominięciem jakiejkolwiek filtracji. Daje to atakującemu możliwość zalogowania się na konto dowolnego użytkownika – wystarczy znajomość loginu. Wystarczy, że w formularzu jako login zostanie podany ciąg znaków: „ user' –- ”, a jako hasło dowolne wyrażenie. Tym sposobem, po pobraniu z formularza danych skrypt PHP wykona zapytanie przedstawione w przykładzie 2.

Przykład 2. Zapytanie SQL z wstrzykniętym kodem rozpoczynającym komentarz

```
SELECT * FROM db_users WHERE login='nazwa_uz' -- ' AND
password='dowolne'
```

Część zapytania, następująca po podwójnym myślniku zostanie zinterpretowana jako komentarz, wykonane jest więc zapytanie tożsame z zaprezentowanym w przykładzie 3.

Przykład 3. Zapytanie SQL, które faktycznie zostanie wykonane

```
SELECT * FROM db_users WHERE login='nazwa_uz'
```

W ten sposób osoba trzecia jest w stanie zalogować się na konto dowolnego użytkownika. W wielu przypadkach konto posiadające uprawnienia administratora ma login „admin” bądź „administrator”. W związku z tym możliwe dla atakującego staje się przejęcie częściowej lub niemal całkowitej kontroli nad systemem.

W przypadku nieznajomości jakiegokolwiek loginu, atakujący również jest w stanie zalogować się do systemu. Wystarczy, że w formularzu logowania poda dowolny login i hasło będące następującym wyrażeniem: „ ') OR 1=1-- ”. Zapytanie wykonane przez skrypt widoczny w przykładzie 2 będzie miało następującą postać zaprezentowaną w przykładzie 4.

Przykład 4. Zapytanie SQL z wstrzykniętym warunkiem

```
SELECT * FROM db_users WHERE login='' AND
password=PASSWORD(' ') OR 1=1 -- ')
```

Sytuację, w której również wykona się tylko część, która nie została zakomentowana przedstawia przykład 5.

Przykład 5. Zapytanie SQL, które faktycznie zostanie wykonane

```
SELECT * FROM db_users WHERE login='' AND
password=PASSWORD(' ') OR 1=1
```

Przykład pokazuje wstrzyknięcie do wyrażenia warunku, który zawsze jest prawdziwy (1=1). Zapytanie zwróci jako wynik wszystkie konta utworzone w systemie, a osoba atakująca zostanie zalogowana na konto utworzone jako pierwsze. Często pierwszym kontem w bazach danych systemów jest konto administratora.

Powyższe sposoby ataków wymagają użycia specjalnych znaków, które są częścią składni języka SQL. W celu zabezpieczania systemu przed wstrzykiwaniem kodu, należy zaimplementować filtrację danych pobranych od użytkownika przed ich przekazaniem do zapytania. W niektórych przypadkach wystarczy usunięcie możliwości korzystania ze znaku „'", co uniemożliwi osobie nieuprawnionej ominięcie znaku otaczającego wartość któregoś z parametru. W przypadku, gdy nie jest to możliwe, należy skorzystać z funkcji addslashes() - funkcja ta poprzedza każdy potencjalnie niebezpieczny znak znakiem „\". Mniej uniwersalnym rozwiązaniem – ponieważ działającym tylko z niektórymi rodzajami baz danych (np. MySQL) jest użycie funkcji (lub jej odpowiednika) mysql_real_escape_string(), która po zidentyfikowaniu zestawu znaków bazy danych dodaje „\" do znaków, które są częścią składni SQL i są potencjalnie niebezpieczne [8]. Istotnym sposobem zabezpieczenia jest również używanie symbolu „'" przy przekazywaniu każdego parametru nawet w przypadkach, w których składnia SQL tego nie wymaga. Uniemożliwia to atakującemu wstrzyknięcie kodu w pasku adresu przeglądarki przy wywoływaniu zapytania. [9]

3.2. Przepełnienie bufora

Zapobieganie błędom przepełnienia bufora wymaga zwracania szczególnej uwagi na używane przy tworzeniu oprogramowania funkcje. Powinny one brać pod uwagę rozmiar danych, które są ich parametrami. Bardzo często za problemy tego typu odpowiadają funkcje obsługujące łańcuchy tekstowe. W przykładzie 6 przedstawiono wywołanie funkcji strcpy.

Przykład 6. Wywołanie strcpy

```
char *strcpy( char *strDestination, const char *strSource )
```

Na listingu widać, że funkcja strcpy w żaden sposób nie sprawdza rozmiaru danych, które przyjmuje. W wielu sytuacjach jej działanie zostanie zakończone wystąpieniem błędu, np. gdy bufor, z którego kopiujemy oraz docelowy nie zawierają żadnych danych, jeżeli strSource nie kończy się jako null, a najczęściej gdy łańcuch źródłowy jest większy od bufora.

Bezpieczniejszym rozwiązaniem jest użycie funkcji strncpy, która sprawdza rozmiar danych wejściowych i bufora docelowego. Listing pokazany jako przykład 7 przedstawia wywołanie funkcji.

Przykład 7. Wywołanie strncpy

```
char *strncpy( char *strDest, const char *strSource, size_t
count )
```

3.3. Zakleszczenie

Przeważająca część reakcji systemów operacyjnych na zakleszczenie opiera się na zapobieganiu wystąpienia jednego z czterech warunków Coffmana (wzajemne wykluczanie, przetrzymywanie i oczekiwanie, brak wywłaszczeń i czekanie cykliczne). W systemach, w których występuje rozpoznawanie zakleszczeń, w momencie wystąpienia zakleszczenia, następuje detekcja stanu systemu w celu wykrycia, czy blokada nastąpiła, a następnie czy została skorygowana. Uruchomiony zostaje algorytm, który śledzi alokację zasobów oraz stan procesów, w wyniku którego następuje cofnięcie systemu do stanu sprzed wystąpienia błędu. Proces, w którym nastąpiło zakleszczenie zostaje uruchomiony ponownie.

Istnieją algorytmy zapobiegania zakleszczeniom. Ich działanie polega na organizacji użycia zasobów przez każdy proces tak, aby zapewnić warunki, w których przynajmniej jeden proces zawsze jest w stanie uzyskać dostęp do zasobów, których potrzebuje. Często używanym rozwiązaniem jest algorytm bankiera, autorstwa Edsgera Dijkstry. Przeciwdziała on nastąpieniu blokady „deadlock" poprzez odmowę lub zawieszenie procesom możliwości dostępu do danych w sytuacji, w której uzyskanie dostępu do danego zasobu mogłoby skutkować stanem niebezpiecznym systemu.

3.4. Inżynieria wsteczna

Jedną z praktyk utrdudniających inżynierię wsteczną jest zaciemnianie kodu. W przypadku języków interpretowanych (np. PHP), należy zmodyfikować kod w taki sposób, aby był rozumiany przez interpreter, ale dla człowieka, który uzyskał do niego dostęp w sposób nieautoryzowany był możliwie mało czytelny. Sposoby zaciemniania kodu:

- szyfrowania i deszyfrowanie kodu podczas jego działania,
- zmiana nazw zmiennych i funkcji na losowe ciągi znaków,
- wyeliminowanie znaków spacji oraz nowego wiersza – kod w jednej linii,
- zapis stałych tekstowych za pomocą kodów ASCII [10].

Korzystając z powyższych metod można znacznie utrudnić lub wręcz uniemożliwić osobie niepowołanej uzyskanie w jakikolwiek użytecznego kodu źródłowego.

W przypadku programów napisanych w językach kompilowalnych, oprócz wymienionych wyżej metod podstawową zasadą bezpieczeństwa jest usunięcie informacji, z których korzysta program debugujący. Niemal wszystkie kompilatory każdego języka mają taką możliwość, ponieważ kwestia ta jest kluczowa dla ochrony programu. W przypadku pominięcia tej zasady, osoba nieuprawniona jest w stanie dokonać skutecznej dekompilacji kodu i w efekcie uzyskać kod aplikacji zbliżony do oryginalnego i umożliwiający zrozumienie szczegółów działania programu [11]. W przykładzie 8 zaprezentowano kod programu wypisującego liczby pierwsze.

Przykład 8. Kod programu przez zaciemnieniem

```
void primes(int cap) {
    int i, j, composite;
    for(i = 2; i < cap; i++) {
     composite = 0;
     for(j = 2; j < i; j++)
          composite += !(i % j);
          if(!composite)
               printf("%d\t", i);
    }
}
int main() {
```

```
    primes(100);
}
```

Przykład 9 przedstawia kod z przykładu 8 po przeprowadzeniu zaciemniania.

Przykład 9. Kod programu po zaciemnieniu

```
_(__,___,____){___/__<=1?_(__,___+1,____):!(___%__)?
_(__,___+1,0):___%__==___/
__&&!____?(printf("%d\t",___/__),_(__,___+1,0)):___%__
>1&&___%__<___/__?_(__,1+
___,____+!(___/__%(___%__))):___<__*__?_(__,___+1,_
___):0;}main(){_(100,0,0);}
```

Innym rozwiązaniem jest kompresja plików wykonywalnych. Pliki te zostają skompresowane przez zewnętrzne oprogramowanie, a następnie są dekompresowane podczas wykonywania programu właściwego. Zaletą tego rozwiązania jest również zmniejszenie rozmiaru plików; spowalnia to jednak pracę programu. Sposób ten zapewnia jednak stosunkowo mały poziom zabezpieczenia. [12]

4. Analiza wpływu zabezpieczeń na działanie

4.1. Wstrzykiwanie kodu SQL

Badanie wpływu technik bezpiecznego programowania na aplikacje webowe wykonano z wykorzystaniem kodu PHP. Posłużono się autorską, niezabezpieczoną aplikacją webową. Badanie objęło obronę przed atakami typu cross-site scripting, SQL injection oraz shell injection. W przykładzie 11 zaprezentowano fragment skryptu przechwytującego dane z formularza wypełnianego przez użytkownika. Dla celów badania, zmierzono czas wykonania skryptu – wynosi on 0,091 s (jest to zaokrąglony średni czas wykonania z dziesięciu prób).

Przykład 10. Niezabezpieczone przechwytywanie danych z formularza

```
if (isset($_POST['zglos'])) {
    $temat = $_POST['temat'];
    $opis = $_POST['opis'];
    $data_rozp = $_POST['data_rozp'];
    $data_zak = $_POST['data_zak'];
    $godz_rozp = $_POST['godz_rozp'];
    $godz_zak = $_POST['godz_zak'];
    $liczba_u = $_POST['liczba_u'];
    $user=$_POST['user'];

    mysql_query("INSERT INTO `wydarzenie` (`temat`,
    `data_rozp`,
    `data_zak`, `godz_zak`, `id_user`, `opis`)
    VALUES ( ' " . Stemat . " ' ,
    STR_TO_DATE ( ' " . date ( ' Y-m-d' ) . " ' , ' %Y-%m-%d'
    $data_zak."', '".$godz_zak."', '".$user."', '".$opis."');");

    header('Location: zaw.php?id='.$id_w);

    ob_flush();
}
```

Kod zaprezentowany w przykładzie 10 przejmuje dane z formularza POST. Instrukcja warunkowa sprawdza, czy nastąpiło ustawienie pola odpowiedzialnego za zatwierdzenie (przesłanie) formularza, zdefiniowanego wcześniej jako „zglos”. Jeżeli warunek jest spełniony, do kolejnych zmiennych przechwytywane są odpowiednio nazwane pola formularza. Następnie z poziomu kodu PHP następuje wykonanie zapytania SQL, wstawiającego dane przesłane w formularzu do nowego rekordu w bazie danych. Po tej operacji użytkownik jest przekierowywany do kolejnej strony i następuje przesłanie danych z bufora wyjściowego.

Dane pobrane od użytkownika nie są w żaden sposób filtrowane. Oznacza to, że użytkownik może przeprowadzić dowolny atak możliwy z poziomu przeglądarki. Podstawową funkcją zabezpieczającą jest htmlentities(), która konwertuje wszystkie zawarte w polu formularza znaczniki HTML do postaci encji. Wszystkie przykładowe znaki specjalne oraz znaczniki html zostały skonwertowane na encje.

W celu zbadania wydajności, zaimplementowano rozwiązanie w skrypcie przechwytującym dane z formularza. Kod źródłowy przedstawiono na listingu w przykładzie 11.

Przykład 11. Przechwytywanie danych po użyciu funkcji htmlentities()

```
if (isset($_POST['zglos'])) {
    Stemat = htmlentities($_POST['temat']);
    Sopis = htmlentities($_POST['opis']);
    $data_rozp = htmlentities($_POST['data_rozp']);
    $data_zak = htmlentities($_POST['data_zak']);
    $godz_rozp = htmlentities($_POST['godz_rozp']);
    $godz_zak = htmlentities($_POST['godz_zak']);
    $liczba_u = htmlentities($_POST['liczba_u']);
    $user=$_POST['user'];
    mysql_query("INSERT INTO `wydarzenie` (`temat`,
    `data_rozp`,
    `data_zak`, `godz_zak`, `id_user`, `opis`)
    VALUES (' " . Stemat . " ' ,
     STR_TO_DATE ( ' " . date ( ' Y-    m-d') . " ' , ' %Y-%m-
    %d'
    $data_zak."', '".$godz_zak."', '".$user."', '".$opis."');");
    header('Location: zaw.php?id='.$id_w);
    ob_flush();
}
```

Zmierzony czas wykonania skryptu wyniósł 0,126 s (jest to zaokrąglony średni czas wykonania z dziesięciu prób). Zastosowanie „htmlentities” w zauważalny sposób wpłynęło na wydajność. Czas wykonania skryptu zwiększył się o około 37 ms, co oznacza wzrost o niemal 41%.

Kolejną funkcją, którą można zastosować jest mysql_real_escape_string. Jest to implementacja funkcji dodającej znaki uniksowe – lewe ukośniki – do znaków mogących być częścią kodu SQL. Zastosowana implementacja to wersja tej funkcji dla bazy MySQL. Ważne jest, żeby w momencie użycia funkcji połączenie z bazą było otwarte, inaczej funkcja nie zadziała.

W celu zbadania wydajności, zaimplementowano rozwiązanie w skrypcie przechwytującym dane z formularza. Czas wykonania skryptu wyniósł 0,153 s (jest to zaokrąglony średni czas wykonania z dziesięciu prób). Zastosowanie „mysql_real_escape_string” istotnie wpłynęło na wydajność. Czas wykonania skryptu zwiększył się o około 61 ms, co oznacza wzrost o o 67% w porównaniu z czasem wykonania niezabezpieczonego skryptu.

W celu zabezpieczenia aplikacji za pomocą obydwu rozwiązań, utworzono funkcję filtruj(), która zwraca łańcuch „oczyszczony” z potencjalnych niebezpieczeństw. Dla dziesięciu powtórzeń, średni czas jednego wykonania skryptu wynosi 0,168 s. Zastosowanie filtracji danych w dość znaczący sposób wpływa więc na szybkość działania skryptu. Czas wykonania zwiększa się niemal dwukrotnie. Ciągle nie jest to jednak wartość zauważalna dla użytkownika. W przypadku formularzy zbliżonych do badanego, czyli posiadających około siedmiu pól, metoda filtracji danych może być z powodzeniem stosowana – wpływ na wydajność będzie znikomy.

Wpływ na bezpieczeństwo aplikacji polega na udaremnieniu prób ataków na aplikację przy pomocy przeglądarki bądź narzędzi służących do złośliwego wykorzystywania luk w formularzach. Filtracja danych zapobiega zmianie warunków wykonania się zapytania SQL, a zamiana znaczników HTML na encje ustrzeże aplikację przed wstrzyknięciem potencjalnie złośliwego skryptu, mogącego zaatakować system lub któregoś z użytkowników.

4.2. Przepełnienie bufora

Występowanie błędów bufora zależy w dużej mierze od ostrożności programisty, zarówno w programowaniu, jak i w wybieraniu technologii tworzenia oprogramowania. Istnieją jednak funkcje narażone na wygenerowanie tego błędu, które posiadają zabezpieczone odpowiedniki – w ramach badania zmierzono wpływ ich użycia na wydajność aplikacji.

Na listingu w przykładzie 12 zaprezentowano propozycję zabezpieczenia przed przepełnieniem bufora funkcji kopiującej łańcuch.

Przykład 12. Bezpieczne użycie strcpy

```
bool obslugaStGCy(const char* input) {
    char buf[80];
    if(input == NULL) {
     assert(false);
     return false;
    }
    if(strlen(input) < sizeof(buf))
          strcpy(buf, input);
    else
     return false;
    return true;
}
```

Funkcja „strcpy” nie przyjmuje jako argument rozmiaru bufora, co czyni ją niebezpieczną w przypadku próby kopiowania łańcucha dłuższego niż pojemność bufora docelowego. Za pomocą instrukcji warunkowej zabezpieczono ją więc przed taką sytuacją; w przypadku zbyt dużego rozmiaru łańcucha wejściowego funkcja zwraca wartość „false”, co oznacza, że operacja kopiowania łańcucha nie została wykonana. To samo dzieje się w przypadku, gdy bufor, z którego kopiujemy nie zawiera żadnych danych. Pomiar czasu wykonania funkcji dla argumentu o długości dwudziestu znaków przeprowadzono w pętli – czas pojedynczego wykonania był trudny do wychwycenia. Pętla wykonała 10000 iteracji, z których wyciągnięto średni czas wykonania funkcji.

Tabela 1. Czas wykonania implementacji obsługi strcpy

Liczba wykonań	Średni czas wykonania [ms]
10000	0.00091188196

Czas wykonania funkcji jest bardzo krótki – wynosi niespełna 1ms. Zabezpieczenie ma znikomy wpływ na wydajność aplikacji – wyniki pomiaru czasu wykonania funkcji bez zabezpieczenia różnią się o pomijalne wartości. Przedstawia je tabela 4.25.

Tabela 2. Czas wykonania implementacji obsługi strcpy – bez zabezpieczenia

Liczba wykonań	Średni czas wykonania [ms]
10000	0.000787545617

Kolejną badaną funkcją jest strncpy. Listing w przykładzie 13 przedstawia implementację obsługi tej funkcji dla potrzeb badawczych.

Przykład 13. Bezpieczne użycie strncpy

```
bool obslugaStrncpy(const char* input) {
    char buf[80];
    if(input == NULL) {
     assert(false);
     return false;
    }
buf[sizeof(buf) - 1] = '\0';
strncpy(buf, input, sizeof(buf));
if(buf[sizeof(buf) - 1] != '\0')
return false;
return true;
}
```

Funkcja „strncpy” jest bezpieczniejsza od „strcpy”, ponieważ przyjmuje jako parametr rozmiar kopiowanego łańcucha wejściowego. Ciągle jednak program należy zabezpieczyć przed pustym parametrem wejściowym oraz brakiem znaku null na końcu łańcucha. W badanej implementacji zabezpieczanie zostało zrealizowane za pomocą instrukcji warunkowych – w przypadku wystąpienia wyżej wymienionych zagrożeń funkcja zwróci wartość „false”, a funkcja „strncpy” nie wykona się. W tabeli 3 przedstawiono wyniki pomiaru czasu wykonania funkcji.

Tabela 3. Czas wykonania implementacji obsługi strncpy

Liczba wykonań	Średni czas wykonania [ms]
10000	0.00182376393

W porównaniu do użycia w implementacji mniej bezpiecznej funkcji „strcpy”, czas wykonania programu nieznacznie wzrósł. Spowodował to nieco dłuższy czas wykonania funkcji kopiującej, która przyjmuje dodatkowy parametr.

Kolejną badaną funkcją jest „snprintf”. Funkcja również realizuje funkcjonalność zapisywania sformatowanego łańcucha źródłowego w zadanym buforze docelowym. Funkcja jest niemal w pełni bezpieczna, ponieważ została zabezpieczona przed wszystkimi argumentami mogącymi wywołać przepełnienie bufora. Dodatkowym zabezpieczeniem jakie należy zastosować jest sprawdzenie

zakończenia łańcucha znakiem „\0”, czyli null. Kod implementacji przedstawiono w przykładzie 14.

Przykład 14. Obsługa snprintf

```
bool obsluga_snprint(int line, unsigned long err, char* msg) {
    char buf[132];
    if(msg == NULL) {
     assert(false);
     return false;
    }
    if(_snprintf(buf, sizeof(buf)-1,
     Błąd w linii %d = %d - %s\n", line, err, msg) < 0)
          return false;
     else
          buf[sizeof(buf)-l] = '\0';
return true;
}
```

Oprócz samego wywołania funkcji, następuje sprawdzenie, czy wiadomość kopiowana nie jest pusta. W przypadku możliwości wystąpienia przepełnienia, funkcja zwróci „false”, a „_snprintf” nie wykona się. Jeżeli dane są poprawne, funkcja wykonuje się, a bufor docelowy zostaje zakończony znakiem null. Wyniki pomiaru czasu wykonania implementacji zaprezentowano w tabeli 4.

Tabela 4. Czas wykonania implementacji obsługi _snprintf

Liczba wykonań	Średni czas wykonania [ms]
10000	0.001826854346

Czas wykonania implementacji funkcji dla tego samego łańcucha jest niemalże taki sam jak

w przypadku implementacji korzystającej z funkcji „strncpy”. Oznacza to podobny, mały negatywny wpływ na wydajność aplikacji – wyniki pomiaru prezentuje tabela 5.

Tabela 5. Czas wykonania implementacji obsługi _snprintf

Implementacja	strcpy	strncpy	snprintf
Zajęta pamięć	17 KB	21 KB	22 KB

Z pomiaru pamięci zajmowanej przez poszczególne implementacje wynika, że implementacja używająca funkcji „strcpy” potrzebuje nieco mniej zasobów. Jest to jednak funkcja najmniej bezpieczna, a różnica w wydajności pomiędzy nią a bardziej bezpiecznymi funkcjami jest bardzo niewielka.

4.3. Zakleszczenie

Zabezpieczanie programu przed błędami związanymi z dostępem procesów do zasobów jest w dużej mierze kwestią podejścia programisty do strony teoretycznej tworzonego oprogramowania. Techniką, która pozwala zapobiec tego typu błędom jest używanie algorytmu bankiera. W ramach badania zabezpieczenia przed błędami związanymi z alokacją procesów, przygotowano implementację algorytmu bankiera w języku Java. Działanie algorytmu polega na stworzeniu bezpiecznego ciągu. Algorytm stara się zaalokować wszystkie procesy.

W ramach badania zbadano szybkość działania algorytmu dla konkretnych danych wejściowych.

Dane wejściowe użyte w algorytmie zaprezentowano w tabeli 6.

Tabela 6. Badanie algorytmu bankiera – dane wejściowe

Rodzaj	Dane
10000	3
Liczba zasobów	4
Tablica alokacji	1 2 2 1 1 0 3 3 1 2 1 0
Tablica maks. liczby egz. zasobu	3 3 2 2 1 1 3 4 1 3 5 0
Tablica przydziału	3 1 1 2

Algorytm poprawnie bezkonfliktowo alokuje wszystkie procesy. Zmierzono czas wykonania operacji alokacji. Wyniki zaprezentowano w tabeli 7.

Tabela 7. Badanie algorytmu bankiera – czas wykonania

Dane wejściowe	Średni czas wykonania [ms]
Tabela 6	0.1054663

Średni czas wykonania operacji alokacji procesów dla podanych danych wejściowych (3 procesy, 4 zasoby) wynosi około 1 ms. Oznacza to, że fakt skorzystania z algorytmu praktycznie nie wpływa na wydajność, za to kompleksowo eliminuje błędy związane z wątkami i zasobami, w przypadku znajomości informacji o żądanych, używanych i aktualnie dostępnych zasobach.

4.4. Inżynieria wsteczna

Badanie zaciemniania kodu przeprowadzono na przykładzie działania programu Dotfuscator. Obiektem badania była przykładowa aplikacja, działająca w środowisku .NET Frawework 4. W celu próby przeprowadzenia inżynierii wstecznej, użyto programu .NET Reflector 9.0.

Kod aplikacji został łatwo zdekompilowany. Widoczna jest kompletna struktura, sygnatury oraz ciała metod. Poniżej przedstawiono pomiar czasu inicjalizacji niezabezpieczonego przed inżynierią wsteczną programu.

Tabela 8. Czasy inicjalizacji programu przed obfuskacją

Przypadek	Średni czas wykonania [s]
Projekt 1, przed użyciem .NET Reflectora	2.821105

Średni czas inicjalizacji programu wyniósł w przybliżeniu 2,8 s.

Następnie przeprowadzono zaciemnianie kodu programem Dotfuscator. Operacja obfuskacji została przeprowadzona bez zmiany standardowych parametrów programu. Dotfuscator umożliwia ustanawianie własnych reguł zaciemniania kodu podczas każdego etapu tego procesu: zmiany nazw, zmiany przebiegu wykonania, szyfrowania łańcuchów, usuwania niepotrzebnych fragmentów kodu i łączenia bibliotek. Po uzyciu programu Dotfuscator, przeprowadzono ponowną próbę inżynierii wstecznej na badanym programie przy pomocy .NET Reflectora. Kod po

zaciemnieniu jest trudny do rozczytania. Przeprowadzono zmianę przebiegu wykonania oraz zmianę nazw (klas, zmiennych obiektów). Dodatkowo, kod programu został zoptymalizowany, a nieużywane fragmenty kodu zostały usunięte. Poniżej przedstawiono wyniki pomiaru czasu inicjalizacji programu po zaciemnieniu i optymalizacji kodu.

Tabela 9. Czasy inicjalizacji programu po obfuskacji

Przypadek	Średni czas wykonania [s]
Projekt 1, po użyciu .NET Reflector	2.391155

Z wyników badania wynika, że czas średni inicjalizacji programu po obfuskacji jego kodu wyniósł około 2,4 s. Oznacza to, że program Dotfuscator oprócz dużego wpływu na bezpieczeństwo aplikacji, wpływa również korzystnie na jej wydajność. W przypadku badanego programu, przyspieszenie inicjalizacji jest minimalne; można się spodziewać, że korzystny wpływ na szybkość wykonania programu teoretycznie powinien rosnąć wraz z rozmiarem projektu – więcej linii kodu zostanie zoptymalizowane. W celu sprawdzenia prawdziwości tej tezy, użyto Dotfuscatora na większym projekcie o niezoptymalizowanym kodzie (zbadano proces generowania GUI). Wyniki zaprezentowano w tabeli 10.

Tabela 10. Czasy inicjalizacji programu po obfuskacji

Przypadek	Średni czas wykonania [s]
Projekt 2, przed użyciem Dotfuscatora	17.25434
Projekt 2, po użyciu Dotfuscatora	16.85349

Z pomiaru wynika, że skuteczność obfuskacji badanym narzędziem nie rośnie wraz z objętością kodu poddawanego obróbce, a zależy od większej liczby czynników (zasoby, z jakich program korzysta, złożoność operacji, które wykonuje itd.). Objętość kodu projektu badanego w przypadku drugim jest większa około ośmiokrotnie od projektu badanego w poprzednim przypadku, a mimo to przyspieszenie jego wykonania przez Dotfuscator również jest minimalne, a nawet mniejsze.

Zmierzono również ilość pamięci zajmowanej przez badane programy po wykonaniu badanych procesów przed i po procesie zaciemniania.

Tabela 11. Porównanie pomiarów zajmowanej pamięci

Projekt	Przed obfuskacją	Po obfuskacji
Projekt 1	6720 KB	6440 KB
Projekt 2	68 543 KB	68 512 KB

Z pomiaru wynika, że użycie programu Dotfuscator spowodowało spadek ilości pamięci operacyjnej zajmowanej przez program pierwszy o 280 KB. Jest to spadek o około 4%. W przypadku drugiego badanego projektu, ilość zajmowanej pamięci zmieniła się o pomijalną wartość. Wynika z tego, że działanie Dotfuscatora ma również korzystny wpływ na oszczędność pamięci potrzebnej do działania badanego programu. Należy jednak pamiętać, że wielkość tego wpływu mocno zależy od indywidualnych właściwości danego kodu [5].

5. Wnioski

W ramach artykułu przeprowadzono analizę wpływu wybranych technik bezpiecznego programowania na wydajność i bezpieczeństwo aplikacji, na podstawie własnych implementacji zabezpieczeń oraz implementacji autorstwa innych autorów. Przybliżono tematykę zabezpieczania programu przed błędami oraz atakami niepowołanych osób.

Wnioski, jakie nasuwają się po przeprowadzeniu analizy, to przede wszystkim fakt, że nie istnieje żadne w stu procentach skuteczne zabezpieczenie programu i danych, którymi operuje. Niezależnie od zastosowanej techniki, zawsze jest ryzyko, że osoba atakująca prędzej czy później będzie w stanie ją obejść. Zabezpieczenia to ciągła walka autorów zabezpieczeń z osobami, które pracują nad sposobem ich obejścia. Co nie oznacza, żeby zaprzestawać prób zabezpieczania oprogramowania – utrudnianie hakerom ich zamiarów jest rzeczą bardzo ważną i istotną dla możliwości normalnego korzystania z wszelakich programów.

W ramach artykułu zawarto analizę wybranych rozwiązań, według autora skutecznych i praktycznych. Mnogość technik bezpiecznego programowania nie pozwala na zbadanie wszystkich w ramach krótkiego artykułu; analiza innych bądź nowszych rozwiązań może być przedmiotem kolejnych eksperymentów i dyskusji. Kwestie bezpieczeństwa oprogramowania zmieniają się na tyle szybko, że tematyka przedstawiona ta zawsze może być przedmiotem badań.

Literatura

[1] K. Sacha, Inżynieria oprogramowania, Helion, Gliwice, 2010.

[2] Sahel Alouneha, Mazen Kharbutlib, Rana AlQuremb, Procedia Computer Science, Volume 21, 2013, 250-256.

[3] Herbert Schildt, Java. Kompendium programisty. Wydanie IX, Helion, Gliwice, 2015

[4] Marcin Lis, Tworzenie bezpiecznych aplikacji internetowych, Helion, Gliwice, 2014.

[5] http://searchsecurity.techtarget.com/ [02-03-2017]

[6] J. C. Foster, Vitaly Osipov, Nish Bhalla, Niels Heinen, Dave Aitel, Buffer Overflow Attacks, 2005, 161-228.

[7] J. Viega, M. Messier, C i C++. Bezpieczne programowanie. Receptury, Helion, Gliwice, 2011.

[8] N. Kalicharan, Java Zaawansowane zastosowania, Helion, Gliwice, 2014.

[9] Grande, J., Boudol, G., Serrano, M., Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming, PPDP 2015, 149-160.

[10] V. Alessandrini, S. Memory, Application Programming, 2016, 83-99.

[11] Welsch, Y., Schäfer, J., Poetzsch-Heffter, A. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 471-500.

[12] Franz, M., Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 12-22.