



Defensive Programming

Kosma Moczek <kosma@kosma.pl>

Geek's Night 2.0 – 20.03.2010

Agenda

- Co kryje się pod słowami “defensive programming”?
- Typowe problemy
- Techniki
- Wsparcie ze strony języków

Czym jest defensive programming?

- To świadomość istnienia prawa Murphy'ego:

„If something can go wrong, it will.”

- Defensive programming to zbiór technik programistycznych minimalizujących negatywne skutki wystąpienia przypadków niestandardowych, błędnych lub wrogich.

Przykład pierwszy

```
$name = $_POST['name'];  
$phone = $_POST['phone'];  
  
$query = "INSERT INTO GEEK(name, phone, ...)  
        VALUES ('$name', '$phone', ...)";  
  
mysql_query($query);
```

- Większość „programistów” PHP nigdy nie przeczytało rozdziału manuala p.t. „Security”.
- Taki kod pozwala na SQL injection.
- Rozwiązanie systemowe ze strony PHP:
magic_quotes_gpc

Przykład pierwszy

- Problemem jest zły design – należało od samego początku wymuszać na programistach stosowanie tzw. *bind values*, co mogłoby wyglądać np. tak:

```
$name = $_POST['name'];  
$phone = $_POST['phone'];  
  
$query = "INSERT INTO GEEK(name, phone, ...)  
VALUES (?, ?, ...)";  
  
mysql_query($query, array($name, $phone));
```

Przykład drugi

```
void have_fun() {  
    char buf[100];  
    puts(„Wpisz coś, ale błagam, nie za długie”);  
    gets(buf);  
}
```

- Funkcja *gets()* nie powinna istnieć.
- Nadpisanie bufora to w praktyce możliwość wykonania dowolnego kodu (zwłaszcza na x86).
- Programiści C dostają dziś poprawione, bezpieczne wersje funkcji do dyspozycji, ale czy naprawdę sądzicie, że z nich korzystają?

Przykład drugi

- Oto przykłady par funkcji – niebezpiecznych i bezpiecznych. Czy typowy leniwy programista będzie pamiętał o tym, żeby podać jeszcze jeden parametr, jeśli bez niego „*też działa*”?

insecure:

```
sprintf(buf, fmt...)  
gets(buf)  
printf(str)  
system(„prog arg...”)
```

secure:

```
snprintf(buf, len, fmt...)  
fgets(buf, len, filehandle)  
printf(„%s”, str)  
fork+exec(„prog”, „arg”...)
```

Przykład trzeci

```
void syscall_fun() {  
    char buf[16];  
    read(fd, buf, 16);  
}
```

...

warning: ignoring return value of 'read', declared
with attribute warn_unused_result

- Czy czytasz komunikaty kompilatora?
- Czy używasz *-Wall -Werror*?
- Czy pamiętasz, jakie wartości może zwrócić `read()`? Czy na pewno reagujesz poprawnie na wszystkie możliwości?

Przykład trzeci

- Możliwe wyniki wywołania `ret = read(..., len, ...)`:
 - `ret == len` – pełen sukces
 - `0 < ret < len` – częściowy sukces (EOF lub odebrany sygnał)
 - `ret == 0` – EOF (lub brak danych dla `O_NONBLOCK`)
 - `ret == -1, errno == EINTR` – odebrany sygnał
 - `ret == -1, errno != EINTR` – błąd
- Uwzględnienie wszystkich możliwości jest nietrywialne; przetestowanie takiego kodu – bardzo trudne.

Fakt:

- Debugowanie kodu jest dwukrotnie trudniejsze, niż jego pisanie.

Wniosek:

- Jeżeli pisząc kod użyjesz całego swojego sprytu, to nie będziesz w stanie takiego kodu w pełni zrozumieć i zdebugować.

Jak się bronić?

1. Nie broń się – zapobiegaj.
2. Stosuj mechanizmy zmuszające do pisania dobrego kodu.
3. Czytaj manuale. Programuj świadomie!

Jakiego języka używasz?

- C?
 - Napisz wrappery wokół syscalli i funkcji bibliotecznych – niech sprawdzają błędy za Ciebie.
 - Używaj gotowych bibliotek, np. GLib, ich wrapperów i obsługi błędów (np. GError).
- Perl?
 - Nieśmiertelna konstrukcja:
`open my $fh, „plik.txt” or die „$!”;`

`No such file or directory at test.pl line 6.`

Jakiego języka używasz?

- Python

- Wyjątki to dobry sposób na zmuszanie programisty do obsługi błędów – jeśli nie będzie ich obsługiwał, jego program bardzo szybko się wysypie, np. na EOFError, oznaczającym koniec pliku przy read().
- Niestety, możliwe jest wycięcie wyjątków poprzez konstrukcję:

try: kod; except: pass

Wielu początkujących programistów tak robi. Moim zdaniem język nie powinien na to pozwalać tak łatwo.

Jakiego języka używasz?

- bash (shell)
 - W bashu istnieje coś na kształt wyjątków. Po wykonaniu „**set -e**” skrypt zakończy działanie, kiedy dowolna z komend zakończy się niepowodzeniem:

```
#!/bin/bash -e
complain() { echo „Bućked” 1>&2; exit 1 }
trap complain ERR
cd /tmp
rm -r .X11-unix/
rm -r .Xvnc/ || true
```

- Zawsze stosuj „**\$zmienna**”, nigdy **\$zmienna**, chyba że naprawdę wiesz, co robisz. Podobnie, stosuj „**\$@**” zamiast **\$@**, **\$*** lub „**\$***”.

Jak się bronić?

1. Nie broń się – zapobiegaj.
2. Stosuj mechanizmy zmuszające do pisania dobrego kodu.
3. Czytaj manuale. Programuj świadomie!

Dobre nawyki

- Upychaj `assert()` i `DEBUG()`, gdzie tylko się da. W wersji finalnej ich nie będzie, więc nie pogorszą wydajności.

- Twórz krótkie opisy funkcji i ich argumentów:

```
/*  
 * frobnicate_args  
 * removes crud from arguments  
 * expects unmodified argc/argv from main  
 * returns amount of crud removed  
 */  
int frobnicate_args(int *argc, char **argv)  
{  
    ...  
}
```

Dobre nawyki

- Nie ufaj funkcji, którą wołasz (zwracana wartość), ani funkcji, która Cię wywołała (argumenty).
- Nie wymyślaj koła na nowo – stosuj funkcje typu `escape_cośtam()` napisane i przetestowane przez innych.
- Pisz unit testy sprawdzające zachowanie modułów/funkcji dla nietypowych argumentów.
- Pod żadnym pozorem nie próbuj wymyślać własnych metod kryptograficznych – chyba, że nazywasz się Bruce Schneier. :D

Co dalej?

<kryptoreklama>

Zapraszam na mój wykład p.t. „Perl dla ludzi”,
we wtorek o godz. 17:00. Więcej informacji na
stronie ASI.

</kryptoreklama>

Kosma Moczek <kosma@kosma.pl>

<http://www.kosma.pl/>