Programowanie Defensywne

Programowanie defensywne i ofensywne
część pierwsza

Arkadiusz Chrobot

Katedra Systemów Informatycznych

6 czerwca 2024

Plan

Wprowadzenie

Pojęcia *defensywnego* i *ofensywnego* programowania nie mają ogólnie przyjętych definicji. W ramach tego wykładu zostaną użyte następujące:

Programowanie defensywne

Programowanie defensywne (ang. *defensive programming*) jest techniką programowania polegającą na tworzeniu kodu programu w taki sposób, aby zawierał on jak najmniej defektów oraz w czasie wykonania jak najwcześniej wracał do poprawnego działania po wystąpieniu wyjątku, spowodowanego niepoprawnymi danymi wejściowymi, niedostępnością zasobu lub innymi nieprzewidzianymi i niekorzystnymi sytuacjami. Ta technika obejmuje swoim zakresem takie zagadnienia jak tolerowanie błędów (ang. *fault-tolerant*), programowanie odporne (ang. *robust programming*), programowanie bezpieczne (ang. *secure programming*) i inne [1].

Wprowadzenie

Programowanie ofensywne

Programowanie ofensywne (ang. *offensive programming*) jest rodzajem programowania defensywnego, polegającym na takiej konstrukcji programu, aby jak najszybciej przerywał on działanie w momencie napotkania błędu, celem uniknięcia propagacji awarii nim spowodowanej. Zazwyczaj komponent, który ulega awarii jest w takim wypadku monitorowany przez inny element oprogramowania, który wznawia jego prawidłową pracę. Ta technika stosowana jest celem ujawnienia *błędów logicznych* w programie, a nie obsługi wyjątków.

Zasady

Choć istnieją pewne ogólne zasady programowania defensywnego [2], to większość z nich jest zależna *od kontekstu*, czyli używanego *języka programowania* oraz *dziedziny zastosowania* tworzonego programu. Wiele firm i instytucji, takich jak Red Hat [3], Microsoft, JPL [4, 5] wypracowało własne zbiory takich zasad. Powstały standardy dla konkretnej gałęzi przemysłu (np. MISRA — ang. *Motor Industry Software Reliability Association* — opracowała standardy dla przemysłu samochodowego, które zostały zaadaptowane również w innych dziedzinach) oraz takie, które od początku były tworzone jako ogólne (np. SEI CERT Coding Standards).

Zasady ogólne

Ogólne zasady programowania defensywnego można zawrzeć w czterech punkach [2]:

Ostrożność Należy założyć, że wystąpią problemy z działaniem programu, które powinny być wykrywane najszybciej jak to możliwe. Program powinien sprawdzać wszystkie dane, których nie generuje oraz wartości zwracane przez podprogramy, czy są one poprawne i nieznieksztalcone.

Błędy użytkowania Programiści muszą zakładać, że osoba, która będzie użytkowała ich kod (w szczególności, jeśli jest on biblioteką) może nie mieć dostępu dokumentacji i w związku z tym podać błędne argumenty wywołania lub dane wejściowe. W takim przypadku zaleca się zakończenie wykonania kodu, aby nie propagować błędu oraz przekazanie klarownego komunikatu opisującego przyczynę awarii. **Komunikat musi być dostarczony programistom użytkującym kod, a nie użytkownikom.**

Zasady ogólne

Ochrona stanu wewnętrznego Zasoby prywatne, od których wartości zależy poprawność działania kodu powinny być niedostępne dla kodu zewnętrznego. Konsekwencją zachowania tej zasady jest wprowadzenie do kodu podziału na moduły (ang. *modularization*).

Obsługa mało prawdopodobnych warunków Każdy warunek, choćby najmniej prawdopodobny powinien być sprawdzany. Jeśli jego wystąpienie bieżąco jest mało prawdopodobne, to na skutek późniejszych zmian w kodzie lub zmianie kontekstu jego użytkowania, może stać się bardzo możliwe.

Zasady ogólne

Te zasady sprowadzają się do unikania niezdefiniowanego (ang. *undefined*) lub nieokreślonego (ang. *unspecified*) zachowania programu. Oba rodzaje zachowań są powiązane z językiem, jaki został użyty do opracowania programu. *Zachowanie niezdefiniowane* powodują konstrukcje językowe, które nie zostały ujęte w standardzie języka, ale np. są stosowane przez twórców translatorów (kompilatorów i interpreterów). Z kolei *zachowanie nieokreślone* wynika z tego, że standard języka przedstawia kilka alternatywnych realizacji danej konstrukcji językowej i pozostawia twórcom translatorów decyzję, który sposób wybiorą.

Standard JPL

Standard JPL dotyczy języka C, który jest używany do tworzenia oprogramowania dla sond, łazików i innego sprzętu kosmicznego [4, 5]. Część tego standardu, która nie jest związana ze standardem MIRSA C (ten ostatni jest płatny), jest dostępna publicznie. Standard ten podzielono na kilka sekcji. W zakresie programowania defensywnego, wyszczególniono następujące reguły:

Ograniczony zasięg Deklaracje zmiennych, stałych itd. **muszą** mieć najmniejszy możliwy zasięg. Niedopuszczalne jest przykrywanie nazw z zewnętrznego zasięgu w zasięgu wewnętrznym.

Zwracane wartości Wartość zwracana przez funkcję **musi** zostać sprawdzona lub rzutowana na typ `void`, jeśli nie jest istotna.

Standard JPL

Weryfikacja argumentów Jeśli funkcja jest funkcją biblioteczną, to **musi** sprawdzać poprawność argumentów z jakimi została wywołana **przed wykonaniem innych zadań**[1]. W przypadku innych funkcji argumenty **muszą** być sprawdzane albo przez funkcję wywołującą, albo przez wywoływaną.

Użycie asercji Asercje **muszą** być używane, aby zapewnić podstawową weryfikację poprawności działania kodu. Funkcje, których kod liczy więcej niż 10 wierszy **powinny** zawierać co najmniej jedną asercję.

Listing 1: Użycie asercji według standardu JPL

```
if (!c_assert(p >= 0) == true) {
        return ERROR;
}
```

[1] Jeśli z tej funkcji korzysta wiele wątków, to standard zakłada, że najczęściej jest ona *wielobieżna* (ang. *re-entrant*).

Standard JPL

Listing 2: Definicja asercji według standardu JPL

```
#define c_assert(e) ((e) ? (true) : \
        tst_debugging("%s,%d: assertion '%s' failed\n", \
                __FILE__, __LINE__, #e), false)
```

Typy Standardowe, podstawowe typy danych **powinny** być zastąpione definicjami typów, określających rozmiar danych i ich znak. *Mój komentarz: Jeśli używany dialekt języka C jest zgodny co najmniej ze standardem ISO C99, to ta reguła może być automatycznie spełniona poprzez użycie* ***standardowych*** *typów, takich jak np.* `uint32_t` *lub* `int64_t`*.*

Standard JPL

Wyrażenia W wyrażeniach złożonych składających się z innych wyrażeń zamierzona kolejność wykonywania działań **musi** być wymuszona poprzez zastosowanie nawiasów okrągłych.

Wyrażenia boolowskie Wyznaczenie wartości wyrażenia boolowskiego **nie może** mieć efektów ubocznych.

Standard firmy Red Hat

Firma Red Hat opracowała standardy defensywnego programowania [3] dla kilku języków programowania (C/C++, Java, Python, powłoki systemowe, Go, Vala), określonych zadań (tworzenie biblioteki, używanie plików tymczasowych, itd., itp.), oraz dla tworzenia mechanizmów zabezpieczających (uwierzytelnianie i autoryzacja, użycie TLS, oprogramowanie dla HSM i inteligentnych kart (ang. *smart cards*)).

Standard firmy Red Hat

Język C — przepełnienie całkowitoliczbowe

Przepełnienie całkowitoliczbowe (ang. *integer overflow*) prowadzi do niezdefiniowanego zachowania. Można mu przeciwdziałać następująco:

1. Użyć większego typu danych by wykonać obliczenia, sprawdzić, czy wynik mieści się w granicach i przeprowadzić jego konwersje do oryginalnego typu. Wszystkie wyniki pośrednie muszą być sprawdzone w ten sposób.
2. Przeprowadzić obliczenia z użyciem równoważnego typu dla liczb naturalnych i zweryfikować wynik przy pomocy operacji bitowych. W przypadku konieczności dodania kilku wyrażeń, każda operacja dodawania musi tak być sprawdzona. Listing 3 zawiera przykład.
3. W przypadku mnożenia należy wyznaczyć wartości graniczne dla danych wejściowych i odrzucić wszystkie, które się w nich nie mieszczą. Listing 4 zawiera przykład.

Standard firmy Red Hat

Język C — przepełnienie całkowitoliczbowe

Listing 3: Obsługa przepełnienia w dodawaniu

```
void report_overflow(void);

unsigned
add_unsigned(unsigned a, unsigned b)
{
  unsigned sum = a + b;
  if (sum < a) { // or sum < b
    report_overflow();
  }
  return sum;
}
```

Standard firmy Red Hat

Język C — przepełnienie całkowitoliczbowe

Listing 4: Obsługa przepełnienia w mnożeniu

```
unsigned
mul(unsigned a, unsigned b)
{
  if (b && a > ((unsigned)-1) / b) {
    report_overflow();
  }
  return a * b;
}
```

Standard firmy Red Hat

Język C — przepełnienie całkowitoliczbowe

Alternatywne rozwiązania w stosunku do tych proponowanych przez standard firmy Red Hat podał Lef Ioannidis w wykładzie wygłoszonym na MIT [6] (konieczne jest włączenie pliku nagłówkowego `limits.h`):

Listing 5: Obsługa przepełnienia całkowitoliczbowego — wersja alternatywna

```
unsigned int ui1, ui2, usum;
/* Initialize ui1 and ui2. */
if(UINT_MAX - ui1 < ui2) {
        /* handle error condition */
} else {
        usum = ui1 + ui2;
}
```

Standard firmy Red Hat

Język C — przepełnienie całkowitoliczbowe

Lef Ioannidis podał również przykład dla obsługi przepełnienia w przypadku odejmowania liczb (konieczne jest włączenie pliku nagłówkowego `limits.h`):

Listing 6: Obsługa przepełnienia całkowitoliczbowego — odejmowanie liczb

```
signed int si1, si2, result;
/* Initialize si1 and si2 */
if((si2>0 && si1<INT_MIN + si2) || (si2<0 && si1 >
↪  INT_MAX + si2)) {
        /* handle error condition*/
} else {
        result = si1 - si2;
}
```

Standard firmy Red Hat

Język C — zmienne globalne

Standard firmy Red Hat zaleca unikanie zmiennych globalnych, aby nie powodowały one problemów w programach wielowątkowych. Jeśli ich użycie jest konieczne, to warto rozważyć zadeklarowanie ich ze słowem kluczowym `static`, aby ograniczyć ich zasię do jednostki translacji (pliku z kodem źródłowym), w których zostały zadeklarowane. W przypadku definicji stałych tablic stałych ciągów znaków należy stosować poniższe rozwiązanie:

Listing 7: Definicja stałej tablicy stałych łańcuchów znaków

```
static const char *const string_list[] = {
  "first",
  "second",
  "third",
  NULL
};
```

Standard firmy Red Hat

Język C — standardowa biblioteka języka C

Użycie funkcji ze standardowej biblioteki języka C nie jest łatwe. Niektóre z nich używają funkcji `malloc()`, aby przydzielić pamięć, np. na zwracane dane. Zatem należy sprawdzić w dokumentacji, czy nie wymagają one zwolnienia pamięci za pomocą `free()`. W przypadku funkcji operujących na łańcuchach znaków należy użyć ich bezpieczniejszych zamienników. Przykładowo zamiast `strcpy()` można użyć `strncpy()`, w następujący sposób:

Listing 8: Przykład użycia `strncpy()`

```
char buf[10];
strncpy(buf, data, sizeof(buf));
buf[sizeof(buf) - 1] = '\0';
```

Proszę zauważyć, że ten kod rozwiązuje tylko problem przepełnienia bufora, a nie utraty części oryginalnego łańcucha.

Standard firmy Red Hat

Język C — zarządzanie pamięcią

Język C nie zarządza pamięcią w sposób bezpieczny (ang. *memory unsafe*). Dlatego warto pamiętać o kilku zasadach używając dynamicznego przydziału pamięci:

- Funkcja `realloc()` nie zwalnia pamięci wskazywanej przez wskaźnik przekazany jej jako argument, jeśli zmiana rozmiaru przydzielonego obszaru się nie powiedzie, ale zwraca wtedy NULL, zatem idiom: `ptr = realloc(ptr, size);` jest niepoprawny, bo może prowadzić do wycieków pamięci.
- Według standardu języka C, po zwolnieniu przez funkcję `free()` pamięci, wskaźnik do niej nie może być porównywany z innym wskaźnikiem, ani nawet wartością `NULL`, bo wynik tego porównania będzie niezdefiniowany.

Standard firmy Red Hat

Język Java

Obiekty zasobów (strumienie) muszą być zawsze zwalniane (zamykane). Zatem zaleca się stosowanie następującej konstrukcji:

Listing 9: Konstrukcja „try-with resources”

```
try (InputStream in = new BufferedInputStream(new
 ↪  FileInputStream(path))) {
    readFile(in);
}
```

Należy unikać wyrzucania błędów (wyjątków dziedziczących po klasie `java.lang.Error` lub `java.lang.Throwable`), ponieważ ich przechwytywanie i obsługa są problematyczne.

Standard firmy Red Hat

Pliki tymczasowe

Pliki tymczasowe mogą być obiektem ataków wykorzystujących sytuacje hazardowe. Aby ich uniknąć należy:

1. Uzyskać lokalizację katalogu na tymczasowe pliki w bezpieczny sposób — należy zignorować niebezpieczne zmienne środowiskowe i np. użyć funkcji `secure_getenv()`, a potem `realpath()` w języku C lub metody `java.lang.System.getenv()` w języku Java.
2. Należy **stworzyć nowy plik**, nie zaś użyć już istniejącego.
3. Plik musi być utworzony tak, aby inne procesy lub wątki nie mogły go otworzyć.
4. Deskryptor pliku tymczasowego nie może wyciec do procesów spokrewnionych i wątków.

Standard firmy Red Hat

Pliki tymczasowe

W języku C dla aplikacji wielowątkowych lub w kodzie należącym do biblioteki należy użyć funkcji `mkostemp()` do tworzenia *nazwanego* pliku tymczasowego. Należy również ustawić flagę `O_CLOEXEC`, aby deskryptor tego pliku nie wyciekł do procesów potomnych i wątków. Jeśli program jest jednowątkowy, to można w nim do tego zadania użyć funkcji `mkstemp()`. W języku Java można użyć metody `java.io.File.createTempFile()`.

Plik tymczasowy *bez nazwy* w języku C może być utworzony przy pomocy funkcji `tmpfile()` lub `fmemopen()`, jeśli jest znany maksymalny rozmiar tego pliku.

Do tworzenia tymczasowych katalogów w języku C służy metoda `mkdtemp()`, ale wynikowy katalog nie jest automatycznie usuwany. W języku Java (począwszy od wersji 7) do tego celu służy metoda `java.nio.file.Files.createTempDirectory()`.

Literatura I

- Spyros Argalias. *Defensive & offensive programming*. 2022. URL: https://programmingduck.com/articles/defensive-programming.
- Red Hat Inc. *Defensive Coding Guide*. 2023. URL: https://developers.redhat.com/articles/defensive-coding-guide.
- Jet Propulsion Laboratory. *JPL Institutional Coding Standard for the C Programming Language*. 2009. URL: https://andrewbanks.com/wp-content/uploads/2019/07/JPL_Coding_Standard_C.pdf.
- Adrian Bledea Georgescu. *NASA coding standards, defensive programming and reliability*. 2017. URL: https://coder.today/tech/2017-11-09_nasa-coding-standards-defensive-programming-and-reliability-a-postmortem-static-analysis./.

Literatura II

Lef Ioannidis. *Secure Programming in C*. 2014. URL: https://ocw.mit.edu/courses/6-s096-effective-programming-in-c-and-c-january-iap-2014/df281b9bb8aa5c4377567454bb839676_MIT6_S096IAP14_Lecture3S.pdf.

Matt Bishop i Chip Elliott. "Robust Programming by Example". W: *Information Assurance and Security Education and Training*. Red. Ronald C. Dodge i Lynn Futcher. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, s. 140–147.

Pytania

?

Koniec

Dziękuję Państwu za uwagę!