Gynvael Coldwind

Diabeł tkwi w szczegółach: C/C++ (część 1)

Programując w językach C lub C++, bardzo łatwo jest popełnić błąd. Powodów można wymienić kilka. Po pierwsze, oba języki są stosunkowo niskopoziomowe, a więc wymagają ręcznego zarządzania pamięcią dynamiczną, dbania o poprawność wskaźników, jak i dbania o wiele innych aspektów. Po drugie, nie wszystkie zachowania obu języków są w pełni zdefiniowane - standardy C i C++ wskazują zachowania nie zdefiniowane w ogóle (ang. *Undefined Behavior*, dalej UB), nie zdefiniowane w standardzie, ale definiowane przez konkretne implementacje (ang. *Implementation-Defined Behavior*), oraz takie, które mogą zostać zaimplementowane na więcej niż jeden sposób, ale niekoniecznie muszą zostać jawnie zdefiniowane w dokumentacji (ang. *Unspecified Behavior*). Niniejszy artykuł ma na celu przedstawienie czytelnikowi kilku przykładów, w których pomimo iluzorycznej trywialności kodu, znajdują się subtelne błędy lub niejasności.

INTEGER OVERFLOW

Arytmetyka w C/C++ jest świetnym przykładem pozornej prostoty. Niestety, w operacjach takich jak dodawanie, odejmowanie, mnożenie, dzielenie czy nawet zmiana znaku kryje się dość spora liczba kruczków, na które trzeba uważać. Większość z nich można zrzucić na karb zjawiska zwanego *integer overflow* (tłumaczone czasem na polski jako „przekroczenie zakresu liczb całkowitych" [1], choć termin angielski jest bardziej popularny).

Integer overflow to zjawisko występujące, gdy wynik danej operacji arytmetycznej wykracza poza zakres danego typu, czyli np. w przypadku typu `int` jest większy niż `INT_MAX` lub mniejszy niż `INT_MIN`. Czasami rozróżnia się *integer overflow* jako przekroczenie zakresu „od góry", oraz *integer underflow* jako przekroczenie zakresu „od dołu". Listing 1a ilustruje kilka przykładowych sytuacji, w których następuje przekroczenie zakresu.

Listing 1a. Integer overflow

```
/* Platforma: x86, 32-bit, GCC */

int a = 2147483647; /* INT_MAX */
a++;
/* wynik 2147483648 wykracza poza zakres */

unsigned char b = 123;
b *= b;
/* wynik 15129 wykracza poza zakres */

short c = -32767;
c -= 123;
/* wynik -32890 wykracza poza zakres */
```

Problem integer overflow w C/C++ rozwiązany jest w następujący sposób:

Standardy i dokumentacje

Kompilatory oraz interpretery języków C oraz C++ są implementowane wg tzw. standardów lub szkiców standardów (ang. *draft*). Obowiązującymi, a zarazem najnowszymi standardami są:

ISO/IEC 9899:2011 (zwany potocznie **C11**; odpowiadający mu *draft* to **N1570** [S1]),
ISO/IEC 14882:2011 (zwany potocznie **C++11**; odpowiadający mu *draft* to **N3337** [S2]).

[W niektórych miejscach w niniejszym artykule powołuję się na konkretne sekcje z najnowszych szkiców standardów - takie miejsca zostaną oznaczone numerem szkicu oraz numerem konkretnej sekcji (np. [N1570 1.2.3] oznacza najnowszy draft C11, sekcja 1.2.3)]

Oprócz powyższych dokumentów warto również wskazać dokumentacje samych kompilatorów oraz interpreterów, m.in. z uwagi na zawarte w nich informacje o przyjętych rozwiązaniach dla niezdefiniowanych zachowań oraz zachowań definiowanych przez implementacje. Dokumentacje kompilatorów zawierają również informacje o rozszerzeniach języka oferowanych przez dany kompilator (rozszerzenia mogą ułatwić pracę programiście, ale niestety sprawiają, że kod przestaje być przenośny między kompilatorami).

Przykładowo kompilatory z rodziny GCC oraz firmy Microsoft posiadają bardzo dobrą dokumentację (kolejno [D1] oraz [D2]).

- W przypadku zmiennych typu *unsigned* (np. `unsigned int`) wynik jest zawsze obliczany modulo górna granica + 1 (chociaż zazwyczaj wygodnie jest myśleć o tym jako o obcinaniu wyniku do dolnych N bitów, gdzie

Nie tylko C/C++...

Problem *integer overflow* oczywiście nie dotyczy jedynie języków C/C++ czy architektury x86 - w zasadzie każdy język czy architektura musi w jakiś sposób poradzić sobie z tym zjawiskiem. Do popularnych rozwiązań należą:

- **nasycenie** (ang. *saturation*) - zmienna nasyca się w wartościach granicznych, czyli np. wynikiem działania `INT_MAX+1` jest nadal `INT_MAX`; przykładowo architektura x86 (rozszerzenie MMX) oraz ARMv6 udostępniają arytmetykę nasyconą
- **"zawinięcie"** (ang. *wrap*) **/ modulo** - po największej możliwej wartości zawsze następuje najmniejsza możliwa wartość, a więc wynikiem działania `INT_MAX+1` jest `INT_MIN`; jest to popularne zachowanie w przypadku platformy x86 oraz np. języka Java
- **wyjątek / więzy integralności** - w przypadku przekroczenia zakresu zmiennej w wyniku działania wygenerowany zostaje wyjątek (lub naruszenie więzów integralności jest sygnalizowane w inny sposób); przykładem może być język Ada
- **zmiana typu zmiennej na typ zmiennoprzecinkowy** - a więc wynik działania `INT_MAX+1` będzie faktycznie poprawny (w matematycznym rozumieniu dodawania), jednak zmienna zmieni swój typ na float; przykładem może być język PHP
- **zwiększenie rozmiaru zmiennej** - w takim wypadku trudno mówić o zjawisku integer overflow - gdy wynik działania wykracza poza zakres zmiennej, jest ona po prostu powiększana; przykładem może być język Python

N to wielkość zmiennej w bitach), a więc przykładowo `UINT_MAX + 1` da w wyniku `0` dla zmiennej typu `unsigned int` (ponieważ wynik obcięty do dolnych 32-bitów wyniesie w tym wypadku zero).

- W przypadku zmiennych typu *signed* (np. `int`) zachowanie przy przekroczeniu zakresu jest niezdefiniowane (patrz ramka: **UB a kod wynikowy**). Warto nadmienić, że niektóre implementacje traktują *signed integer overflow* analogicznie do *unsigned integer overflow*, czyli traktują dolne N bitów wyniku jako ostateczny wynik.

Niestety, „przycinanie" wyniku ma w niektórych przypadkach poważne konsekwencje dla bezpieczeństwa aplikacji. Dzieje się tak, ponieważ wynik danego obliczenia jest niepoprawny z punktu widzenia tej samej operacji wykonanej, korzystając z „normalnych" matematycznych zasad, a więc może być niezgodny z przewidywaniami programisty, a zatem z tym, co programista faktycznie chciał osiągnąć.

UB a kod wynikowy

Natrafiając na kod powodujący pojawienie się niezdefiniowanego zachowania, kompilator nie jest zobowiązany przez standard do (prawie) żadnego konkretnego zachowania (niektórzy żartują, że nawet wygenerowanie kodu formatującego dysk jest w takim wypadku poprawne względem standardu [6]).

Zarówno standard C, jak i C++ wskazują następujący zakres sugerowanych zachowań [N1570 3.4.3] [N3337 1.3.24]:

- całkowite zignorowanie sytuacji - czyli wygenerowanie kodu tak jakby danego dane UB nie było w tym miejscu możliwe (jest to tożsame ze zdaniem się w tym konkretnym wypadku na zachowanie specyficzne dla danej platformy)
- przyjęcie udokumentowanego rozwiązania sytuacji
- przerwanie kompilacji z błędem
- przerwanie wykonania programu z błędem

Dodatkowo niektóre popularne kompilatory w ramach optymalizacji decydują się nie generować kodu wynikowego na podstawie kodu opartego o UB.

Listing 1b. Niepoprawny odbiór testu ze strumienia

```
/* platforma: x86, 32-bit, Windows */

/* odbiór długości tekstu */
/* max długość tekstu: 255 */
unsigned char text_size = read_byte(input);

/* wyliczenie wielkości bufora */
unsigned char buffer_size = text_size + 1;

/* alokacja */
char *text = (char*)malloc(buffer_size);
if(!text) abort();

/* odbiór tekstu */
read_data(text, text_size);
text[text_size] = '\0';
...
```

Aby zilustrować zagrożenie, posłużę się fragmentem kodu widocznym na Listingu 1b. Jest to dość typowy (i niestety niepoprawny) kod odbierający krótki tekst ze strumienia. Działa on w następujący sposób:

1. Odbiera długość tekstu.
2. Wylicza wielkość bufora (czyli do wielkości tekstu dodawany jest jeden bajt potrzebny, aby zakończyć odbierany string terminatorem '\0').
3. Alokuje bufor na ostateczny tekst wg wyliczonej wielkości.
4. Wczytuje faktyczny tekst ze strumienia do bufora.

Integer overflow w tym wypadku następuje w sytuacji, gdy przesłana wielkość tekstu wynosi 255, czyli `UCHAR_MAX` dla rozważanej platformy. Stosując opisaną wyżej zasadę, ostateczna wielkość `buffer_size` będzie wynosić:

```
(255 + 1) % 256 → 0
```

W tym momencie powstaje pytanie: jak funkcja `malloc` zareaguje na 0 jako ilość bajtów do zaalokowania? Standard C wskazuje, że zachowanie w tym wypadku jest zdefiniowane przez konkretną implementację [N1570 7.22.3], natomiast standard C++ wymaga, aby zwrócony wskaźnik był różny od nullptr [N3337 3.7.4.1]. Dodatkowo, dla rozważanej platformy przypadek ten jest opisany w MSDN [2] – na stercie zaalokowany zostanie bufor o wielkości zerowej. A więc `malloc` się powiedzie i zwróci poprawny wskaźnik do bufora różny od `nullptr/NULL`.

Ostatecznie następuje odbiór tekstu i zapisanie go do wcześniej zaalokowanego bufora. Niestety, wywołanie `read_data` korzysta ze zmiennej `text_size` do określenia wielkości odczytywanych danych, a więc w tym wypadku do bufora o wielkości 0 zapisane zostanie 255 bajtów – tak więc konsekwencją *integer overflow* jest przepełnienie bufora na stercie (ang. *heap-based buffer overflow*), co w skrajnym przypadku może doprowadzić do wykonania kodu wstrzykniętego przez atakującego (tematyka ta jednak mocno wykracza poza zakres niniejszego artykułu; zainteresowanym polecam [3] oraz [4]).

Jak więc zabezpieczyć kod w takim przypadku? Oczywiście należy sprawdzić, czy w wyniku danej operacji może nastąpić przekroczenie zakresu (test „przed") lub wykonać operacje i sprawdzić, czy przekroczenie zakresu faktycznie nastąpiło (test „po"). Przykładowa funkcja realizująca inkrementację zmiennej typu `unsigned char` ze zwróceniem uwagi na możliwe przekroczenie zakresu znajduje się na Listingu 1c. Funkcja działa w następujący sposób: jeśli przepełnienie nastąpiło, to wartość po inkrementacji będzie niższa (a konkretniej, będzie równa 0), niż wartość wartość przed. Konkretniej w tym wypadku byłoby to:

```
if(UCHAR_MAX > 0) ...
```

co oczywiście jest warunkiem prawdziwym, a jednocześnie jedyną sytuacją, w której a faktycznie jest większe od a+1.

Listing 1c. Przykładowa funkcja inkrementująca zmienną typu unsigned char

```
bool uc_inc(unsigned char &a) {

 /* czy nastąpi przepełnienie */
 if(a > a + 1) {
   return false;
 }

 a++;
 return true;
}
```

Oczywiście nie jest to jedyna poprawna implementacja funkcji inkrementującej dla zmiennych typu unsigned. Co ważne, implementacja ta jest niepoprawna dla zmiennych signed (Listing 1d).

Listing 1d. Przykładowa niepoprawna funkcja inkrementująca zmienną typu int

```
bool i_inc(int &a) {

 /* czy nastąpi przepełnienie */
 if(a > a + 1) {
   return false;
 }

 a++;
 return true;
}
```

Jak wspomniałem wcześniej, *integer oveflow* dla typów *signed* jest zachowaniem niezdefiniowanym. Co więcej, niektóre popularne kompilatory podczas optymalizacji zakładają, że overflow nigdy miejsca miał nie będzie (ponieważ ufają, że programista nigdy nie dopuści do UB). A więc w tym konkretnym wypadku kompilator może założyć, że warunek `a > a + 1` nigdy nie będzie prawdziwy, a następnie, korzystając z tego założenia, całkowicie usunąć ten blok kodu. Dla przykładu, na Listingu 1e oraz 1f znajduje się funkcja `i_inc` kolejno przed, oraz po optymalizacji (test został wykonany, używając GCC 4.6.2 z pakietu MinGW [5]) – jak widać, sprawdzenie warunku oraz blok `if` zostały faktycznie usunięte.

Listing 1e. Pośrednia forma funkcji i_inc przed optymalizacją

```
// g++ (GCC) 4.6.2 (MinGW)
// -fdump-tree-all -O3 -c test.cpp
// plik: test.cpp.013t.cfg

;; Function bool i_inc(int&) (_Z5i_incRi)

bool i_inc(int&) (int & a)
{
 bool D.1703;
 int D.1700;
 int D.1699;

<bb 2>:
 D.1699 = *a;
 D.1699 = *a;
 D.1700 = D.1699 + 1;
 if (D.1699 > D.1700)
   goto <bb 3>;
 else
   goto <bb 4>;

<bb 3>:
 D.1703 = 0;
 goto <bb 5>;

<bb 4>:
 D.1699 = *a;
 D.1700 = D.1699 + 1;
 *a = D.1700;
 D.1703 = 1;

<bb 5>:
 return D.1703;

}
```

Listing 1f. Pośrednia forma funkcji i_inc po optymalizacji

```
// g++ (GCC) 4.6.2 (MinGW)
// -fdump-tree-all -O3 -c test.cpp
// plik: test.cpp.143t.optimized

bool i_inc(int&) (int & a)
{
 int D.1700;
 int D.1699;

<bb 2>:
 D.1699_7 = *a_2(D);
 D.1700_8 = D.1699_7 + 1;
 *a_2(D) = D.1700_8;
 return 1;

}
```

W związku z powyższym, w przypadku zmiennych typu *signed* nie możemy użyć testu „po". Należy więc sprawdzić, czy parametr jest równy granicznej wartości, ponieważ jest to jedyna wartość, która przy próbie inkrementacji spowoduje przepełnienie. Listing 1g prezentuje poprawny test dla tego przypadku.

Listing 1g. Poprawny test dla inkrementacji zmiennej typu int

```
/* czy nastąpi przepełnienie? */
/* wymaga limits.h */
if(a == INT_MAX) {
 return false;
}
```

NIEJASNA KOLEJNOŚĆ WYKONANIA

Listing 3a. Co zostanie wypisane na *stdout*?

```
int p(char c) { putchar(c); return 1; }
void func(int a, int b, int c) {}

...

/* Przypadek 1 */
int d = p('A') + p('B') * p('C') *
        p('D') + p('E') * (p('F') *
        p('G') + p('H')) - p('I');
putchar('\n');

/* Przypadek 2 */
func(p('X'), p('Y'), p('Z'));
```

Rysunek 1. Wynik wykonania kodu z Listingu 3a

```
# gcc (GCC) 4.6.2 (MinGW)
AEFGHBCDI ZYX

# Microsoft C/C++ 15.0.21022.8
ABCDEFGHI XYZ

# Microsoft C/C++ 15.0.21022.8 (/Ox /Ot)
ABCDEFGHI ZYX

# Microsoft C/C++ 16.00.40219.01
ABCDEFGHI ZYX

# cint C/C++ interpreter 5.16.19
ABCDEFGHI XYZ

# Pelles ISO C 7.00.18
ABCDEFGHI XYZ

# Borland C++ 5.5.1
ABCDEFGHI ZYX
```

Na Listingu 2a znajdują się dwa proste kawałki kodu – w obu z nich wywoływana jest funkcja p() wypisująca podany znak oraz zwracająca wartość, która jest używana podczas dalszego działania programu.

Postawmy więc pozornie proste pytanie: w jakiej kolejności zostaną wypisane znaki na strumień wyjścia? Zgadywać można, że wywołania będą następować od lewej do prawej. Lub być może kolejność wywołań będzie zależna od matematycznych priorytetów operatorów. Można również po prostu sprawdzić – Rysunek 1 pokazuje wynik wykonania przykładowego kodu na kilku różnych kompilatorach – jak widać, testowane implementacje C/C++ wywołują funkcje w różnej kolejności [7].

Listing 2b. Przykład niepoprawnego kodu tworzącego wątek

```
int SpawnThread(func_t *callback, void* userdata);
ThreadStruct *GetNewThreadStruct(void);

int main() {
...
  /* stwórz nowy wątek */
  ThreadStruct *th;

  int tid = SpawnThread(
    th->ThreadFuncPtr,
    th = GetNewThreadStruct()
  );
...
```

Najważniejsze punkty sekwencyjne w C i C++

Standardy C i C++ definiują m.in. następujące punkty sekwencyjne [8] [N1570 Annex C] [N3337 1.9]:

- na końcu pełnego wyrażenia; do pełnych wyrażeń zaliczane są:
  - normalne wyrażenia zakończone średnikiem,
  - warunek w `if`, `switch`, `while` oraz `do ... while`,
  - każde z (opcjonalnych) wyrażeń w `for(;;)`,
  - opcjonalne wyrażenie przy `return`,
- bezpośrednio przed wywołaniem funkcji, ale po ewaluacji argumentów (oraz samego wskaźnika na funkcje),
- między operatorami &&, || oraz , (comma); należy pamiętać, że separator argumentów funkcji nie jest operatorem comma),
- w wyrażeniu `a ? b : c` między operandem a, kolejnym ewaluowanym operandem,
- między kolejnymi inicjalizacjami zmiennych: `int a = 1, b = 2`,
- po pełnej deklaracji, np. `int a`.

Oprócz powyższych istnieją również inne punkty sekwencyjne.

Oczywiście, w przypadku przykładowego kodu kolejność wywoływania funkcji nie ma żadnego znaczenia. Można jednak wyobrazić sobie przypadek, w którym poprawność działania polega na konkretnej kolejności ewaluacji wyrażeń – taki kod przedstawiony jest na Listingu 2b. Podczas jego tworzenia, programista błędnie założył, że ewaluacja przy wywołaniu funkcji następuje zawsze od końca. Błąd jest o tyle złośliwy, że prawdopodobnie program będzie działał dobrze w środowisku danego programisty. Niestety, zmiana flag kompilacji lub choćby zmiana wersji kompilatora może zmienić kolejność ewaluacji, co będzie skutkować użyciem niezainicjalizowanej zmiennej `th`, oraz (w optymistycznym przypadku) natychmiastowym „*crashem*" programu w momencie odczytu zmiennej. W przypadku pesymistycznym w zmiennej `th` znajdzie się wartość wskazująca na istniejący obszar pamięci, a nowy wątek zacznie wykonywać losowy kawałek kodu, który ostatecznie i tak skończy się wyjątkiem i zakończeniem programu, ale bardzo utrudni znalezienie przyczyny.

Można więc wyciągnąć (poprawny) wniosek, że języki C oraz C++ nie definiują dokładnej kolejności ewaluacji wyrażeń w niektórych przypadkach (warto zaznaczyć, że większość współczesnych języków dokładnie definiuje kolejność wykonania ewaluacji). Zamiast tego wprowadzone zostało pojęcie **punktów sekwencyjnych** (ang. *sequence points*) – są to jasno określone miejsca, po których programista może założyć, że wszystkie operacje oraz ich efekty uboczne zostały już wykonane (patrz ramka: **Najważniejsze punkty sekwencyjne w C/C++**). Takie podejście pozwala kompilatorowi m.in. na optymalizację poprzez dobranie najkorzystniejszej kolejności wykonywania działań i ewaluacji (jest to istotne w przypadku nowoczesnych architektur, na których możliwe jest wykonanie kilku dobrze dobranych działań równocześnie).

Wracając do kodu tworzącego wątek – aby uzyskać poprawny program, wystarczy przenieść wyrażenie `th = GetNewThreadStruct()` przed punkt sekwencyjny poprzedzający ewaluacje argumentów funkcji `SpawnThread`, czyli np. do deklaracji zmiennej `th` (patrz Listing 2c).

Listing 2c. Poprawiony kod tworzący wątek

```
/* stwórz nowy wątek */
ThreadStruct *th = GetNewThreadStruct();
int tid = SpawnThread(
  th->ThreadFuncPtr, th
);
```

Innym bardzo dobrym przykładem obrazującym niejednoznaczności wynikające ze sformułowania kodu bez brania pod uwagę punktów sekwencyjnych jest następująca zagadka [B1]:

```
int a = 5; a = a++ + ++a;
/* Ile wynosi a? */
```

Powyższy kod zawiera dwa punkty sekwencyjne: po inicjalizacji zmiennej a, oraz po drugim wyrażeniu. Dodatkowo, standardy mówią, że przypisanie będzie wykonane po obliczeniu wartości obu stron (ale niekoniecznie po zaaplikowaniu efektów ubocznych, takich jak dodatkowe zapisy) [N1570 6.5.16] [N3337 5.17]. Drugie wyrażenie zawiera natomiast stosunkowo dużo operacji zapisu oraz odczytu wykonywanych na zmiennej a, a konkretniej:

- a++ jest operacją odczytu ze zmiennej a
- a++ jest również operacją zapisu do zmiennej a
- analogicznie, ++a jest operacją odczytu ze zmiennej a
- ++a jest również operacją zapisu do zmiennej a
- a = ... jest operacją zapisu do zmiennej a

Ponieważ w międzyczasie nie ma żadnego punktu sekwencyjnego, powyższe zapisy oraz odczyty mogą wystąpić w kilku różnych kolejnościach, co w ostatecznym rozrachunku sprowadza się do kilku różnych możliwych wyników. Jaki jest więc poprawny wynik? Poprawnego wyniku niestety nie ma, ponieważ w omawianych językach wyrażenie, w którym występuje więcej niż jedna modyfikacja danej zmiennej (lub zapis i odczyt danej zmiennej bez ustalonej konkretnej kolejności, w jakiej zapis i odczyt się odbędą), jest niezdefiniowanym zachowaniem (UB).

Listing 3. W jakiej kolejności wykonają się konstruktory?

```
/* plik klasa.h */
class MojaKlasa {
  public:
    MojaKlasa();
};

/* plik a.cpp */
MojaKlasa A;
MojaKlasa B;

/* plik b.cpp */
MojaKlasa C;
MojaKlasa D;
```

Ostatnim w tej części przykładem niejasnej kolejności wykonania jest problem dynamicznej inicjalizacji zmiennych globalnych. Na Listingu 3 przedstawiony jest fragment przykładowego projektu, na który składają się trzy pliki: *klasa.h* zawierająca deklarację klasy `MojaKlasa`, oraz pliki *a.cpp* oraz *b.cpp* deklarujące obiekty klasy `MojaKlasa` o nazwach `A` oraz `B` (a.cpp) i `C` oraz `D` (b.cpp). Powstaje pytanie: w jakiej kolejności zostaną wywołane konstruktory obiektów `A`, `B`, `C` oraz `D`?

Standard C++ wskazuje, że dynamiczna inicjalizacja następuje w kolejności deklaracji w danym pliku źródłowym [N3337 3.6.2]. A więc w tym wypadku mamy pewność, że konstruktor obiektu `A` będzie wywołany przed konstruktorem obiektu `B`, a konstruktor obiektu `C` będzie wywołany przed konstruktorem obiektu `D`. Niestety, nadal daje to dwie możliwości: `A B C D` oraz `C D A B`, tak więc mamy do czynienia z kolejnym zachowaniem typu *unspecified behavior*.

W przypadkach, w których zależy nam na konkretnej kolejności wywoływania konstruktorów, należy na przykład przepisać kod na dynamiczną alokację obiektów w określonej przez nas kolejności. Dodatkowo niektóre kompilatory, jak np. g++ (GCC), udostępniają rozszerzenie języka, które pozwala określić dokładną kolejność, w jakiej ma nastąpić inicjalizacja obiektów globalnych względem całego projektu – służy do tego konstrukt `__attribute__ ((init_priority(N))`, gdzie `N` to priorytet inicjalizacji [9].

PODSUMOWANIE

I tak oto kończymy część pierwszą artykułu o drobnych szczegółach, których pominięcie może skutkować kilkugodzinną sesją z debuggerem, nieprzewidzianymi trudnościami przy przenoszeniu kodu, a czasem nawet skutecznym przełamaniem zabezpieczeń aplikacji przez osoby trzecie. Jak wykazał powyższy tekst, tworząc kod w C czy C++, należy pamiętać o zakresie zmiennych (który czasem jest traktowany po macoszemu lub zupełnie ignorowany), nie zawsze zdefiniowanej kolejności wykonywania kodu, a także o tym, że kod oparty o zachowanie niezdefiniowane nie zawsze musi pojawić się w ostatecznym kodzie wynikowym.

W tym miejscu autor artykułu chciałby raz jeszcze zachęcić czytelników do przejrzenia standardów języków C oraz C++, a także do zapoznania się z zaleceniami na temat tworzenia bezpiecznego i stabilnego kodu, np. autorstwa CERT-u [7].

Wszystkie opinie wyrażone w artykule są prywatnymi opiniami autora.

W sieci

Najnowsze szkice standardów C i C++:

- [S1] http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf
- [S2] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3337.pdf

Dokumentacja popularnych kompilatorów:

- [D1] Free Software Fundation, Inc. „GCC online documentation”: http://gcc.gnu.org/onlinedocs/
- [D2] Microsoft. MSDN, „Visual C++”: http://msdn.microsoft.com/en-us/library/60k1461a

Pozostała bibliografia:

- [1] Wikipedia. Przekroczenie zakresu liczb całkowitych: http://pl.wikipedia.org/wiki/Przekroczenie_zakresu_liczb_ca%C5%82kowitych
- [2] Microsoft: http://msdn.microsoft.com/en-us/library/6ewkz86d(v=vs.110).aspx
- [3] Matt Conover, woowoo Security Team: http://www.cgsecurity.org/exploit/heaptut.txt
- [4] Ben Hawkes, „*Attacking the Vista Heap*”: https://www.lateralsecurity.com/downloads/hawkes_ruxcon-nov-2008.pdf
- [5] MinGW GCC: http://www.mingw.org/
- [6] Chris Lattner, „*What Every C Programmer Should Know About Undefined Behavior* #1/3”: http://blog.llvm.org/2011/05/what-every-c-programmer-should-know.html
- [7] CERT. „CERT *Secure Coding Standards*”: https://www.securecoding.cert.org/confluence/display/seccode/EXP10-C.+Do+not+depend+on+the+order+of+evaluation+of+subexpressions+or+the+order+in+which+side+effects+take+place
- [8] Wikipedia. „*Sequence point*”: http://en.wikipedia.org/wiki/Sequence_point
- [9] Free Software Fundation, Inc. „*C++-Specific Variable, Function, and Type Attributes*”: http://gcc.gnu.org/onlinedocs/gcc/C_002b_002b-Attributes.html#C_002b_002b-Attributes

Materiały własne:

- [B1] http://gynvael.coldwind.pl/?id=369

Gynvael Coldwind gynvael@coldwind.pl

Na co dzień autor pracuje w firmie Google na stanowisku Information Security Engineer. Po godzinach prowadzi bloga oraz nagrywa podcasty o programowaniu (http://gynvael.coldwind.pl/). Hobbystycznie programuje od ponad 20 lat (w tym ponad 10 lat w C i C++).