Gynvael Coldwind

Diabeł tkwi w szczegółach: C/C++ (część 2)

Ciąg dalszy rozważań na temat niezdefiniowanych zachowań (ang. Undefined Behavior, dalej UB) i ich potencjalnych skutków, na przykładzie niezamierzonych wycieków danych, oraz kolejnych problemów związanych z operacjami na liczbach całkowitych.

POPULARNE PLATFORMY

Programista tworzący kod nie powinien polegać na danym, często wywnioskowanym na podstawie testów, niezdefiniowanym zachowaniu (UB). Nawet zmiana sposobu kompilacji (np. użytych flag optymalizacyjnych) czy wersji kompilatora może spowodować zmianę danego zachowania, a więc kod przestanie robić to, czego programista od niego oczekuje - świadczą o tym choćby przykłady z poprzedniej części artykułu (Programista 3/2012).

Niemniej jednak w kontekście danej platformy i kompilatora możemy wskazać najbardziej prawdopodobne, rzeczywiste skutki wykonania kodu zawierającego UB - często są one dobrze znane i rozumiane. W niniejszym artykule zostaną wskazane niektóre prawdopodobne skutki UB dla popularnych platform x86-32 oraz x86-64 (AMD64) przy założeniu, że kod został skompilowany kompilatorem z rodziny GCC dla systemów GNU/Linux (Ubuntu) oraz Windows.

WYCIEKI DANYCH

Niejako oczywistym jest, że w pamięci procesu znajdują się wszystkie dane, na których dany proces operuje. Trochę mniej oczywisty jest fakt, że często znajdują się tam również fragmenty niektórych danych, na których proces wcześniej operował. Co więcej, nowo utworzone zmienne mogą zostać umieszczone w tym samym obszarze pamięci, w którym leżą stare dane. To wszystko powoduje, że możliwym jest nieintencjonalne spowodowanie ujawnienia (wycieku) istotnych danych, np. podczas wysyłania pakietów sieciowych lub zapisu danych do pliku.

Dlaczego w ogóle przejmować się wyciekiem jakichś tam „losowych" bajtów z pamięci? W niektórych przypadkach faktycznie dane, które zostaną ujawnione, nie zawierają żadnych wartościowych informacji. Rozważmy jednak, co może znajdować się w pamięci np. przeglądarki internetowej:

- Fragmenty przeglądanych stron internetowych
- Historia przeglądania
- Wartości ciasteczek
- Ostatnio użyte hasła do logowania na poszczególnych stronach
- Ostatnio wpisywane w formularzach dane
- Różne wewnętrzne struktury danych

Z punktu widzenia prywatności i bezpieczeństwa użytkownika nie było by dobrze, aby któraś z powyższych informacji została ujawniona. Dotyczy to również wewnętrznych struktur danych - zawarte w nich informacje (np. wartości wskaźników) mogą stanowić bardzo ważną podpowiedź dla atakującego, który próbuje obejść współczesne mechanizmy zabezpieczeń utrudniające wykorzystanie podatności (w szczególności ASLR [1]).

Czytelnikowi zainteresowanemu konkretnymi przykładami chciałbym wskazać lukę znalezioną przeze mnie kilka lat temu w popularnych przeglądarkach internetowych [V1], podobną lukę znalezioną przez Mateusza Jurczyka [3], oraz nominowaną do Pwnie Awards[1] lukę w Adobe Flash znalezioną przez Fermina Serna [4].

Wracając do tematu artykułu, czyli języków C oraz C++, można wskazać trzy główne powody niezamierzonego wycieku danych:

- Niezainicjowane zmienne (w tym pola w strukturze itp.)
- *Padding* (pl. dopełnienie, bajty wyrównujące; dalej będę korzystał ze spolszczenia „padding")
- Nieprawidłowy dostęp do pamięci
- Rozważmy szczegółowo poszczególne przypadki.

NIEZAINICJOWANE ZMIENNE

Jedną z pierwszych rzeczy, których programista uczy się o C/C++, jest fakt, że nie wszystkie deklarowane zmienne są zainicjowane (tj. mają od początku istnienia ustaloną wartość) [N1570 6.7.9] [N3337 8.5]. Do grupy zainicjowanych można włączyć:

- Zmienne o tzw. statycznym czasie przechowywania (ang. *static storage duration*), a konkretniej zmienne globalne oraz statyczne są inicjowane zerem (lub **`NULL`**, bądź **`nullptr`** w przypadku wskaźników). W przypadku struktur, itp., każdy element inicjowany jest osobno.
- Zmienne lokalne wątku (ang. *thread-local*) są inicjowane zerem (jw.).
- W pełni jawnie zainicjowane zmienne (np. **`int a = 5;`**).
- Częściowo jawnie zainicjowane instancje struktur oraz tablice (np. **`int a[5] = {1,2};`** - pozostałe elementy tablicy **`a`** zostaną wyzerowane).
- Obiekty z konstruktorami inicjującymi wszystkie pola.

Natomiast pozostałe zmienne przy tworzeniu mają nieokreśloną zawartość, tj. nie można czynić żadnych założeń co do ich wartości (w szczególności oznacza to, że kompilatory nie mają obowiązku czyścić pamięci zajmowanej przez takie zmienne). Do tej grupy zaliczamy:

- Zmienne lokalne (np. **`int a;`**).
- Zmienne alokowane na stercie (np. za pomocą **`malloc`** czy **`new`**, oczywiście z wyłączeniem przypadku wywołania konstruktora inicjującego wszystkie pola obiektu).

1 Pwnie Awards są prestiżowymi nagrodami przyznawanymi za wybitne osiągnięcia w dziedzinie bezpieczeństwa IT, rozdawanymi corocznie na konferencji Black Hat w Las Vegas.

O ile teoretycznie wartość niezainicjowanej zmiennej jest nieokreślona, o tyle w praktyce na przykładowych platformach można wymienić kilka potencjalnych możliwych zawartości takiej zmiennej. Przede wszystkim, należy zaznaczyć, że po użyciu ani stos, ani sterta nie są w żaden sposób czyszczone. Dodatkowo, co zostało wspomniane wcześniej, zarezerwowana pamięć dla nowej zmiennej mogła być już wcześniej używana. Tak więc:

- Zarezerwowana pamięć na stosie może zawierać:
  - "Ostatnie" (najnowsze) wartości zmiennych istniejących w tym samym miejscu wcześniej.
  - Argumenty wywoływanych wcześniej funkcji.
  - Adresy powrotu.
  - Zachowane w pewnym momencie wartości rejestrów.
  - Tzw. ciasteczka bezpieczeństwa (ang. *security cookies*, ew. *stack canaries* [5]).

- Zarezerwowana pamięć na stercie może zawierać:
  - "Ostatnie" wartości wcześniej istniejących w tym miejscu zmiennych.
  - Fragmenty wcześniej używanych wewnętrznych struktur sterty.

Jako przykład niezamierzonego wycieku danych ze stosu niech posłuży kod tworzący i wysyłający prosty pakiet sieciowy (patrz Listing 1). Pakiet składa się z pola typu (**int type**) oraz danych - w tym wypadku jest to tablica 256 znaków przeznaczona na imię. W funkcji **SendNamePacket** najpierw tworzona jest lokalna instancja struktury **pn**, a następnie uzupełniany jest typ oraz za pomocą bezpiecznej funkcji **strcpy_s** [6] kopiowane jest podane w parametrze imię do **pn.name**. Następnie pakiet jest wysyłany.

Niestety, jeśli podane imię nie składa się z dokładnie 255 znaków (w przykładzie składa się jedynie z 4 + terminator), to część tablicy **pn.name** jest niezainicjowana, a więc potencjalnie zawiera sporo "starych" informacji ze stosu.

Listing 1. Przykładowy kod ujawniający dane ze stosu

```
...
const int PACKET_TYPE_NAME = 5;
...
struct PacketName {
   int type;
   char name[256];
};
...
bool SendNamePacket(Socket *s, const char *name) {
   PacketName pn;
   pn.type = PACKET_TYPE_NAME;
   if(strcpy_s(pn.name, sizeof(pn.name), name) != 0)
      return false;
   return s->Send(&pn, sizeof(pn));
}
...
SendNamePacket(s, "John");
```

W tym konkretnym przypadku potencjalnym rozwiązaniem byłoby wyzerowanie całej pamięci zajmowanej przez tablicę **pn.name** (np. za pomocą funkcji **memset**, lub np. alokując strukturę za pomocą funkcji **calloc**, która zeruje zaalokowany fragment pamięci). Niestety, w przypadkach bardziej skomplikowanych struktur ustawienie wartości wszystkich pól może nie wystarczyć.

PADDING

Języki C oraz C++ przewidują możliwość wyrównania zmiennych w pamięci, w tym w tablicach oraz strukturach, do konkretnych adresów (zazwyczaj podzielnych przez 2, 4 lub 8) [N1570 6.2.8] [N3337 3.11]. Ma to związek z optymalizacją - dostęp do zmiennych znajdujących się na wyrównanych adresach jest szybszy na niektórych architekturach procesorów (temat ten wykracza poza tematykę artykułu; patrz [7]), oraz z wymaganiami niektórych architektur (np. częstym wymaganiem jest, aby czterobajtowa zmienna znajdowała się na adresie podzielnym przez 4).

Oczywiście, w przypadku konieczności wyrównania tworzy się "wolna przestrzeń" pomiędzy kolejnymi zmiennymi - ta przestrzeń nazywana jest paddingiem.

W C/C++ padding występuje przede wszystkim w następujących miejscach (zaczynając od najbardziej oczywistych):

- Między polami struktury (zazwyczaj w przypadku gdy kolejne pola mają różną wielkość).
- W uniach (nadmiarowe bajty względem używanego pola unii).
- W polach bitowych (nieużywane bity).
- W zmiennych (np. **long double**).

Ani standard C, ani C++ nie gwarantują konkretnych wartości jeśli chodzi o padding, a w szczególności nie gwarantują, że bajty stanowiące padding będą wyzerowane [N1570 6.2.6.1] [N3337 8.5]. Również, standardy nie stawiają żadnych wymagań względem kopiowania paddingu podczas przypisania (tj. kompilator w konkretnej sytuacji decyduje, czy padding zostanie skopiowany).

Listing 2A. Przykładowy niepoprawny kod ujawniający dane (struct, union)

```
// x86-32, gcc, GNU/Linux
struct my_st {
   char ch; // sizeof(my_st.ch) == 1
            // padding: 3 bajty
   int i;   // sizeof(my_st.i)  == 4
};          // sizeof(my_st)    == 8

void func1(Socket *s) {
   my_st *x = new my_st;
   x->ch = 'A';
   x->i  = 0x12345678;
   Send(s, x, sizeof(*x));
   // Zostały wysłane 3 niewyzerowane
   // bajty paddingu.
   delete x;
}

union my_u {
   int i;
   char ch;
}; // sizeof(my_u) == 4

void func2(Socket *s) {
   my_u u;
   u.i = 0x12345678;
   ...
   u.ch = 'A';
   Send(s, &u, sizeof(u));
   // Zostały wysłane 3 bajty
   // zawierające fragment wartości
   // wcześniej użytego pola u.i.
}
```

Ponieważ przypadki paddingu w strukturach oraz uniach są dość oczywiste (przykład umieszczony został na Listingu 2A), rozważmy implementację typu long double [N1570 6.2.5] [N3337 3.9.1] dla popularnej platformy x86-64/GCC.

Na omawianej platformie wielkość typu long double zwracana przez operator **sizeof** wynosi 16 bajtów. W praktyce typ **long double** implementowany jest przez *x86 Extended Precision Format* [8], którego wielkość to 80 bitów, czyli 10 bajtów. Tak więc pozostałe 6 bajtów stanowi padding.

Przykładowy kod ujawniający dane zawarte w paddingu znajduje się na Listingu 2B.

Listing 2B. Przykładowy niepoprawny kod ujawniający dane (long double)

```
struct V3D {
   long double x, y, z;
};

bool CacheHighPrecision(FILE *cache, V3D *v) {
   return fwrite(v, sizeof(*v), 1, cache);
}
```

W tym miejscu warto przeprowadzić eksperyment, polegający na:

1. Wypełnieniu stosu znaną wartością (np. poprzez stworzenie i wypełnienie lokalnej tablicy, a następnie jej uwolnienie).
2. Stworzeniu i zainicjowanie zmiennych typu long double.
3. Wypisaniu ich w formie zakodowanej razem z paddingiem.

Przykładowy kod takiego eksperymentu znajduje się na Listingu 3, natomiast na Rysunku 1 znajduje się przykładowy wynik wykonania takiego kodu (kod został wykonany na systemie Ubuntu 11.10).

Listing 3. Eksperyment z `long double`

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void HexDump(void *p) {
   unsigned char *uc = p;
   size_t i;

   printf("v: [");
   for(i = 0; i < 10; i++)
      printf("%.2x", uc[i]);
   printf("], padding: [");

   for(i = 10; i < sizeof(long double); i++)
      printf("%.2x", uc[i]);
   printf("]\n");
}

void FillTheStack() {
   char buf[129] = {0};
   memset(buf, 'A', 128);
   fputs(buf, stderr);
}

void Test()
{
   long double a = 5.0;
   long double b = a + 1.0;

   HexDump(&a);
   HexDump(&b);
}

int main(void) {
   FillTheStack();
   Test();
   return 0;
}
```

W tym konkretnym wypadku wypisanie paddingu obu zmiennych spowodowało ujawnienie pewnych danych. W przypadku zmiennej **a** są to dane, którymi wcześniej był wypełniany stos (0x41 to kod znaku 'A'). Natomiast w paddingu zmiennej **b** pojawiła się inna wartość, która okazała się być fragmentem wspomnianego wcześniej ciasteczka bezpieczeństwa.

ODCZYT ZA BUFOREM

Kolejnym częstym powodem wycieków danych jest próba odczytania z bufora (tablicy, struktury) większej ilości danych, niż się tam znajduje - czyli następują odwołania do bajtów znajdujących się w pamięci bezpośrednio za danym buforem. Oczywiście z punktu widzenia standardu jest to UB, natomiast w praktyce spowoduje ujawnienie danych lub "crash" aplikacji (w przypadku gdy dalsza pamięć po buforze nie jest zamapowana).

```
> gcc -Wall -Wextra -std=c99 sz_ld.c -O3
> ./a.out 2>/dev/null
v: [00000000000000a00140], padding: [414141414141]
v: [00000000000000c00140], padding: [dd73f06900b7]
>
```

Rysunek 1. Wynik eksperymentu z long double

Sytuacja jest analogiczna do przepełnieniem bufora (o czym była już mowa w pierwszej części tego artykułu, a także jest mowa pod koniec niniejszej części), z tą różnicą, że dochodzi do próby odczytu, a nie zapisu, danego fragmentu pamięci.

USE-AFTER-FREE

Innym problemem jest tzw. *use-after-free* (dosłownie: użycie po zwolnieniu), czyli odwołanie się do zmiennej (często obiektu lub instancji struktury), która już nie istnieje - tj. została zwolniona, lub skończył się czas jej życia (np. nastąpiło wyjście z bloku kodu, w którym była zadeklarowana).

Przykładem niech będzie Listing 4 (inspirowany pewnym kodem, który ostatnio dostałem do debuggowania), na którym znajduje się niepoprawnie skonstruowana funkcja **`GetCStr`**. W momencie wywołania funkcji z argumentem **`my_string`** wykonywana jest lokalna kopia tego obiektu. Następnie pobierany jest adres wewnętrznego bufora obiektu zawierającego C-string odpowiadający temu stringowi, po czym następuje wyjście z funkcji, wraz ze zniszczeniem obiektu **`s`** (a więc i bufora, którego adres został pobrany). Niestety, adres tego nieistniejącego już bufora został zwrócony i zapamiętany w zmiennej **`n`**, która jakiś czas później jest użyta jako argument funkcji **`puts`**. Wywołanie **`puts`** w tym wypadku spowoduje próbę odczytu z pamięci z nieistniejacego już bufora, co wg. standardu jest UB [N1570 6.2.4] [N3337 3.7], a w praktyce może doprowadzić albo do ujawnienia danych znajdujących się akurat w miejscu wskazywanym przez **`n`** w pamięci (jeśli w międzyczasie zostały tam zapisane inne informacje, co jest możliwe), albo do „crashu" aplikacji, jeśli dany fragment pamięci został odmapowany.

Potencjalnych poprawnych rozwiązań w przypadku tej funkcji jest kilka. Listing 5 prezentuje jedno z nich, polegający na stworzeniu kopii bufora przed jego zniszczeniem (niestety, jednocześnie wprowadza to wymaganie sprawdzenia, czy alokacja wykonywana przez **`strdup`** się powiodła, oraz zwolnienia zaalokowanej pamięci później; w tym konkretnym wypadku najlepsze byłoby całkowite przeprojektowanie problematycznego fragmentu kodu).

Listing 4. Błąd typu `use-after-free`

```
const char* GetCStr(std::string s) {
   return s.c_str();
}
...
const char *n = GetCStr(my_string);
...
puts(n);
```

Listing 5. Potencjalne rozwiązanie problemu

```
const char* GetCStr(std::string s) {
  return strdup(s.c_str());
}
```

Warto w tym miejscu wspomnieć o terminie *dangling pointer* (dosłownie: „zwisający wskaźnik") [9], który określa wskaźnik wskazujący na już nieistniejący obiekt. Dereferencja dangling pointera prowadzi oczywiście do sytuacji opisanej powyżej, stąd

częstym zaleceniem jest umieszczenie wartości NULL/nullptr we wszystkich wskaźnikach wskazujących na dany obiekt po jego zwolnieniu.

PRZECIWDZIAŁANIE WYCIEKOM

Oprócz tworzenia dobrej jakości kodu (inicjowania zmiennych, pamiętania o paddingu, zerowania wskaźników itp.) warto wskazać również kilka narzędzi, które ułatwiają znalezienie tego typu błędów:

- Valgrind - popularny w środowiskach *nixowych program wykrywający niektóre błędy w zarządzaniu pamięcią.
- AddressSanitizer (ASan) [10] - moduł do LLVM pozwalający wykryć wiele różnych nieprawidłowości w trakcie operowania na pamięci.
- PageHeap lub GFlags - narzędzia umożliwiające włączenie dodatkowych opcji w menadżerze sterty pod Windowsem, ułatwiających detekcję nieprawidłowości związanych z operowaniem na stercie.

Dodatkowo, przy tworzeniu formatów plików i protokołów sieciowych warto zainteresować się istniejącymi bibliotekami do serializacji danych, takimi jak np. Protocol Buffers [11], a także pomyśleć o tekstowych formatach przechowywania danych jak np. JSON czy XML (choć mogą wprowadzić dodatkowe problemy innej natury).

DZIELENIE NA INTACH

Na koniec wróćmy na chwilę do tematu zmiennych typu **int** opisywanych w poprzedniej części artykułu. Warto wskazać jeszcze dwa przypadki, w których wynik, będący konsekwencją zakresu zmiennych oraz ich wewnętrznego kodowania, może być zaskoczeniem dla programisty.

Zacznijmy od dzielenia dwóch liczb całkowitych. Oczywiście, każdy praktykujący programista pamięta o specjalnym przypadku, jakim jest dzielenie przez zero - według standardów jest to UB [N1570 6.5.5, N3337 5.6], natomiast na popularnej platformie x86 skutkuje rzuceniem wyjątku #DE (*Division Error*, pl. błąd [podczas] dzielenia) w terminologii Intela [7], EXCEPTION_INT_DIVIDE_BY_ZERO w terminologii Microsoftu lub SIGFPE z parametrem FPE_INTDIV w przypadku systemów zgodnych z POSIX.

Na tej architekturze, oraz innych korzystających z kodowania U2, jest jeszcze jeden specjalny i niestety mniej znany przypadek:

```
int a = INT_MIN;
a /= -1; // *crash*
```

Przeanalizujmy powyższy kod: zmiennej **a** przypisywana jest wartość **INT_MIN**, czyli, zakładając 32-bity typ **int**, -2147483648 (kodowane jako 0x80000000), następnie wartość ta zostaje podzielona przez -1, o czym można myśleć jako o zmianie znaku - a więc wynikiem jest 2147483648. Niestety, ta liczba wykracza poza zakres zmiennej typu **int** (dla przypomnienia, zakres 32-bitowego typu **int** to -2147483648 do 21474836487), a więc nie da się jej zapisać w zmiennej typu **int** - mamy więc do czynienia z UB [N1570 6.5] [N3337 5]. Na rozważanej platformie zostanie wygenerowany wyjątek, podobnie jak w przypadku dzielenia przez zero (#DE). Listing 6 prezentuje poprawną funkcje dzielącą dla rozważanej platformy.

Listing 6. Poprawna funkcja dzieląca (dla kodowania U2)

```
bool Dzielenie(int *wynik, int dzielna, int dzielnik) {
  if(dzielnik == 0)
    return false;

  // wymaga limits.h
  if(dzielna == INT_MIN &&
     dzielnik == -1)
    return false;

  if(!wynik)
    return false;

  *wynik = dzielna/dzielnik;
  return true;
}
```

ZMIANA ZNAKU

Zmianę znaku można oczywiście zapisać również w sposób bezpośredni, wstawiając znak minus przed zmienną. Rozważmy następujący przykład (dla tych samych założeń co poprzednio):

```
int a = INT_MIN;
a = -a;
```

Analogicznie jak w poprzednim przypadku, wynikiem jest 2147483648, którego nie można zapisać w rozważanej zmiennej. W przeciwieństwie jednak do dzielenia, w tym wypadku nie zostanie wygenerowany wyjątek. Spotykanym zachowaniem będzie natomiast przypisanie zmiennej **a** ponownie wartości **INT_MIN**, co wynika ze sposobu zmiany znaku w kodowaniu U2 (patrz ramka **Zmiana znaku liczby w kodowaniu U2**). Ważną obserwacją jest więc fakt, że dla tej konkretnej wartości nie jest możliwe wyliczenie wartości bezwzględnej, ani "ręcznie", ani za pomocą funkcji **abs()** - w obu przypadkach otrzymany wynik będzie w praktyce nadal ujemny.

Jakie są konsekwencje zaniedbania obsłużenia opisanej sytuacji? Poza oczywistym błędem w wynikach lub "crashem" programu, można również w niektórych sytuacjach doprowadzić do możliwości wykonania kodu przez atakującego. Niech przykładem będzie obsługa (niesławnego) formatu zapisu bitmapowych obrazów - BMP.

W nagłówku BMP znajdują się między innymi dwa 32-bitowe pola ze znakiem (o których możemy myśleć jak o intach), opisujące szerokość oraz wysokość obrazu [2]. Format BMP zakłada, że bitmapa w pliku zapisana jest od dołu do góry, chyba że wysokość jest wartością ujemną - w takim wypadku w pliku bitmapa jest zapisana od góry do dołu, a faktyczna wysokość to wartość bezwzględna z wartości zapisanej w nagłówku.

Listing 7. Niepoprawne wyliczenie ilości potrzebnej pamięci w loaderze BMP

```
struct BMP {
   ...
   int w, h;
   int bpp;
   bool topdown;
   ...
};

void* AllocBitmap(BMP *b) {
  if(b->bpp != 8) {
     // Unsupported.
     return NULL;
  }

  // Top-Down?
  if(b->h < 0) {
     b->topdown = true;
     b->h = abs(b->h);
  }
```

Zmiana znaku liczby w kodowaniu U2

Obliczanie zmiany znaku w kodowaniu U2:
Krok 1: negacja wszystkich bitów
Krok 2: dodanie 1 do wyniku

Przykład dla 32-bitowej liczby 1 (zapis hexadecymalny):
00000001 ← początkowa wartość
FFFFFFFE ← po negacji
FFFFFFFF ← po dodaniu 1
Otrzymana wartość: -1

Specjalny przypadek dla 32-bitowej liczby -2147483648 (INT_MIN):
80000000 ← początkowa wartość
7FFFFFFF ← po negacji
80000000 ← po dodaniu 1
Otrzymana wartość: -2147483648 (INT_MIN)

```
    // Limit.
    if(b->w > 4096 || b->h > 4096) {
       return NULL;
    }

    // Alloc.
    size_t sz = b->w * b->h;
    return malloc(sz);
}
```

Listing 7 zawiera przykładowy, niepoprawny kod wyliczający ilość potrzebnej pamięci na podstawie danych z nagłówka. Programista tworzący ten kod założył, że po wywołaniu **abs()** wysokość będzie zawsze dodatnia, co, jak już wiemy, nie jest prawidłowym założeniem w C i C++. Dla wysokości równej **INT_MIN**, po **abs()** na rozważanej platformie wysokość będzie nadal wynosić **INT_MIN**, a więc będzie nadal bardzo dużą ujemną liczbą. Oczywiście, w tym wypadku test **b->h > 4096** zwróci **false**, przez co skrajnie duża liczba nie zostanie wykryta. Idąc dalej, w mnożeniu **b->w * b->h** nastąpi *integer overflow*, a dla parzystych szerokości dodatkowo wynikiem będzie 0 (np. 4 * 0x80000000 → 0x00000000 dla 32-bitowych zmiennych), co z kolei spowoduje alokację zdecydowanie za małej ilości bajtów na stercie (patrz część pierwsza artykułu).

Prawidłowym rozwiązaniem w tym przypadku byłoby sprawdzenie, czy **b->h** jest równe **INT_MIN** przed wywołaniem **abs()**, oraz wyjście z funkcji zwracając **NULL** w takim przypadku. Istotne jest, aby sprawdzenie nastąpiło faktycznie przed próbą zmiany znaku, ponieważ jak pamiętamy z poprzedniej części artykułu, kompilator może zdecydować usunąć kod opierający się na UB w fazie optymalizacji (a z matematycznego punktu widzenia, sprawdzenie, czy wynik jest ujemny po obliczeniu wartości bezwzględnej jest zupełnie zbędne).

PODSUMOWANIE

Kończąc niniejszą krótką serię artykułów, chciałbym raz jeszcze zachęcić czytelnika do przejrzenia standardów języków C oraz C++, zwracania uwagi na niezdefiniowane zachowania podczas tworzenia kodu, a także do zapoznania się z zaleceniami na temat tworzenia bezpiecznego i stabilnego kodu [12].

W sieci

Najnowsze szkice standardów C i C++:

- [N1570] http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf
- [N3337] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3337.pdf

Pozostała bibliografia:

- [1] Wikipedia. http://en.wikipedia.org/wiki/Address_space_layout_randomization
- [2] Microsoft. MSDN, „BITMAPINFOHEADER structure". http://msdn.microsoft.com/en-us/library/windows/desktop/dd183376(v=vs.85).aspx
- [3] Mateusz Jurczyk, Hispasec Labs. „Firefox, Opera, Safari for Windows BMP file handling information leak". http://goo.gl/5dDVW
- [4] Fermin Serna. „CVE-2012-0769, the case of the perfect info leak". http://zhodiac.hispahack.com/my-stuff/security/Flash_ASLR_bypass.pdf
- [5] Wikipedia. http://en.wikipedia.org/wiki/Stack_buffer_overflow#Stack_canaries
- [6] Microsoft. MSDN, „strcpy_s". http://msdn.microsoft.com/en-us/library/td1esda9.aspx
- [7] Intel. „Intel® 64 and IA-32 Architectures Software Developer Manuals". http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html
- [8] Wikipedia. http://en.wikipedia.org/wiki/Extended_precision#x86_Extended_Precision_Format
- [9] Wikipedia. http://en.wikipedia.org/wiki/Dangling_pointer
- [10] Address Sanitizer. http://code.google.com/p/address-sanitizer/
- [11] Protocol Buffers. http://code.google.com/p/protobuf/
- [12] CERT. „Secure Coding Standards". https://www.securecoding.cert.org/confluence/display/seccode/CERT+Secure+Coding+Standards
- [13] Microsoft. MSDN, „GFlags and PageHeap ". http://msdn.microsoft.com/en-us/library/windows/hardware/ff549561.aspx

Materiały własne:

- [V1] „Firefox 2.0.0.11 and Opera 9.50 beta Remote Memory Information Leak" http://vexillium.org/?sec-ff

Wszelkie opinie wyrażone w artykule są prywatnymi opiniami autora.

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++).