

# Wycieki pamięci - analiza popularnych języków programowania

Mariusz Ziętek, Szymon Pitula

11 maja 2019

## Spis treści

|   |   |   |
|---|---|---|
| 1 | Wprowadzenie . . . . .                              | 3 |
| 2 | Metody i narzędzia do wykrywania wycieków . . . . . | 3 |
| 3 | Wycieki pamięci w języku C/C++ . . . . .            | 5 |
| 4 | Wycieki pamięci w języku Rust . . . . .             | 9 |

# 1 Wprowadzenie

Wycieki pamięci to problem związany z alokacją i nie zwalnianiem pamięci programu, która nie jest już potrzebna. Zjawisko to powoduje, że obszar pamięci który nie jest już używany przez program pozostaje oznaczony jako zajęty i nie może zostać ponownie wykorzystany. Problem ten jest najbardziej widoczny w aplikacjach, które działają ciągle lub przez długi okres czasu. Program taki będzie rezerwował coraz więcej pamięci, powodując znaczny spadek wydajności i w końcu zakończenie procesu. Wyciek pamięci jest stosunkowo trudny w wykryciu, ponieważ jego objawy nie są widoczne od razu po uruchomieniu programu. Problem ten jest spotykany w różnych językach programowania, w szeroko rozumianym programowaniu aplikacji. W swojej pracy chcielibyśmy przybliżyć ten problem czytelnikom, pokazać przyczyny i skutki wycieków pamięci oraz metody diagnozowania i unikania tego problemu w najpopularniejszych językach programowania: C/C++, Rust, Java, Python. Każdy z tych języków posiada inny system zarządzania pamięcią oraz wymaga innego udziału użytkownika w tym procesie.

## 2 Metody i narzędzia do wykrywania wycieków

Podstawowym narzędziem używanym przez nas do detekcji wycieków jest **Valgrind**. Jest to program dostarczający narzędzia do debugowania i profilowania działania programów. Najpopularniejszym narzędziem jest **Memcheck**, który będzie używany w tym projekcie najczęściej. Pozwala on na wykrycie błędów związanych z pamięcią (w tym wycieków), które mogą prowadzić do nieprzewidywalnego zachowania programu lub jego zakończenia. Istotną wadą programu jest jego wpływ na wydajność monitorowanej aplikacji. Valgrind potrafi spowolnić działanie programu nawet trzydziestokrotnie. W przypadku tego projektu nie jest to jednak duży problem, ponieważ przytoczone programy będą stosunkowo proste, aby przekaz był jak najbardziej przejrzysty.

Aby uruchomić aplikację z kontrolą Valgrind wystarczy dopisać odpowiednią flagę na przykład:

```
valgrind --leak-check=yes myprog arg1 arg2
```

Powyższe polecenie pozwoli na uruchomienie narzędzia *Memcheck* (domyślnie), wraz ze szczegółową kontrolą wycieków. Jeżeli Valgrind wykryje problem z pamięcią lub wyciek, zwróci odpowiednią wiadomość. Przykładowy raport wygląda następująco:

```
==19182== Invalid write of size 4
==19182==      at 0x804838F: f (example.c:6)
==19182==      by 0x80483AB: main (example.c:11)
==19182== Address 0x1BA45050 is 0 bytes after a block of size
==19182==      40 alloc'd
==19182==      at 0x1B8FF5CD: malloc (vg_replace_malloc.c:130)
==19182==      by 0x8048385: f (example.c:5)
==19182==      by 0x80483AB: main (example.c:11)
```

Informacje które tu widzimy to:

- 19182 to ID procesu, zwykle nie jest ważne.
- Pierwsza linia ("Invalid write ...") wskazuje na rodzaj problemu. W tym przypadku nieudana próba zapisu do pamięci.
- Poniżej znajduje się ślad stosu i nazwy funkcji mówiące o tym, gdzie wystąpił problem. Czytanie tej sekcji może być trudne w przypadku dużych programów, ułatwieniem może być czytanie od dołu.

### 3 Wycieki pamięci w języku C/C++

Konstrukcja tego języka pozwala na wykonanie wycieków pamięci na wiele sposobów. Wybraliśmy te z nich, które w możliwie czytelny sposób pokażą istotę problemu i takie, które są często popełniane. Poniżej przedstawione zostały wycieki związane z alokacją pamięci i zwalnianiem nieużywanego obszaru w przypadku korzystania ze wskaźników. Do alokacji i zwalniania pamięci w języku C++ służą operatory *new* oraz *delete* (można także korzystać z operatorów języka C, np. - *alloc*, *malloc*, *free*. W języku tym odpowiedzialność za zarządzanie pamięcią leży po stronie programisty. Za pomocą *new* rezerwuje się pamięć potrzebną do przechowania obiektu, a gdy ta nie jest już potrzebna należy ją zwolnić. Jeżeli programista nie użyje operatora *delete* do zwolnienia pamięci, dojdzie do wycieku. Można to zaobserwować już w bardzo prostych programach. Poniższy przykład pokazuje różnicę w wykorzystaniu przez program pamięci w przypadku, gdy zostaje ona poprawnie zwolniona i w przypadku, gdy do zwolnienia nie dojdzie. Pierwszy fragment kodu obrazuje poprawne zarządzanie pamięcią:

```
void problematic_function();
int main() {
    problematic_function();
    return 0;
}
void problematic_function(){
    int *data = new int;
    *data = 15;
    delete data;
}
```

Pamięć została zwolniona poprawnie, zatem nie powinien wystąpić wyciek. Potwierdzeniem tej tezy są wyniki programu Valgrind:

```
==6002== Memcheck, a memory error detector
==6002== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==6002== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==6002== Command: ./main
==6002==
==6002==
==6002== HEAP SUMMARY:
==6002==    in use at exit: 0 bytes in 0 blocks
==6002==   total heap usage: 2 allocs, 2 frees, 18,948 bytes allocated
==6002==
==6002== All heap blocks were freed -- no leaks are possible
==6002==
==6002== For counts of detected and suppressed errors, rerun with: -v
==6002== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Raport Valgrind mówi o tym, że zaalokowano i zwolniono dwie struktury, zatem wyciek nie jest możliwy. Całkowita użyta przez program pamięć to 18,948 bajtów.

W kolejnym kroku z kodu programu zostanie usunięta linia odpowiedzialna za zwolnienie pamięci:

```
void problematic_function();
int main() {
    problematic_function();
    return 0;
}
void problematic_function(){
    int *data = new int;
    *data = 15;
}
```

Zmieniony program zostaje przetestowany przez Valgrind i wyniki są następujące:

```
==6055== Memcheck, a memory error detector
==6055== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==6055== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==6055== Command: ./main
==6055==
==6055==
==6055== HEAP SUMMARY:
==6055==   in use at exit: 4 bytes in 1 blocks
==6055==   total heap usage: 2 allocs, 1 frees, 18,948 bytes allocated
==6055==
==6055== 4 bytes in 1 blocks are definitely lost in loss record 1 of 1
==6055==    at 0x483087B: operator new(unsigned int) (in /usr/lib/valgrind/vgpreload_memcheck-x86-linux.so)
==6055==    by 0x108657: problematic_function() (in /home/dsk/Desktop/OiAK-Proj/CppTablica/main)
==6055==    by 0x10862C: main (in /home/dsk/Desktop/OiAK-Proj/CppTablica/main)
==6055==
==6055== LEAK SUMMARY:
==6055==   definitely lost: 4 bytes in 1 blocks
==6055==   indirectly lost: 0 bytes in 0 blocks
==6055==   possibly lost: 0 bytes in 0 blocks
==6055==   still reachable: 0 bytes in 0 blocks
==6055==   suppressed: 0 bytes in 0 blocks
==6055==
==6055== For counts of detected and suppressed errors, rerun with: -v
==6055== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Tym razem Valgrind informuje o wycieku. Według raportu po zakończeniu programu w użyciu zostały 4 bajty, co zgadza się z rozmiarem niezwolnionej liczby całkowitej. Dodatkowo mamy też informację, że wyciek jest definitywny, czyli bezwzględnie wystąpił. Valgrind informuje także o miejscu wystąpienia błędu. Korzystając z tej informacji można szybko zlokalizować błąd. Z informacji wynika, że przyczyną jest operator *new* w funkcji *problematicfunction()* wywołanej w *main()*. Informacja ta jest oczywiście prawdziwa, co łatwo zauważyć analizując kod programu.

Kolejny przykład prezentuje często popełniany błąd, który łatwo popełnić.

```
struct problematicStructure{
    int indexNumber;
    float *grades = new float[1000];
};

int main() {

    problematicStructure *s1 = new problematicStructure();
    delete s1;
    return 0;
}
```

Mogłoby się wydawać, że program jest wykonany poprawnie. Struktura została zaalokowana i zwolniona. Valgrind jednak pokaże, że istnieje wyciek pamięci.

```
==6518== Memcheck, a memory error detector
==6518== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==6518== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==6518== Command: ./main
==6518==
==6518==
==6518== HEAP SUMMARY:
==6518==   in use at exit: 4,000 bytes in 1 blocks
==6518==   total heap usage: 3 allocs, 2 frees, 22,952 bytes allocated
==6518==
==6518== 4,000 bytes in 1 blocks are definitely lost in loss record 1 of 1
==6518==    at 0x4830F6B: operator new[](unsigned int) (in /usr/lib/valgrind/vgpreload_memcheck-x86-linux.so)
==6518==    by 0x10887B: problematicStructure::problematicStructure() (in /home/dsk/Desktop/OiAK-Proj/CppTablica/main)
==6518==    by 0x108792: main (in /home/dsk/Desktop/OiAK-Proj/CppTablica/main)
==6518==
==6518== LEAK SUMMARY:
==6518==   definitely lost: 4,000 bytes in 1 blocks
==6518==   indirectly lost: 0 bytes in 0 blocks
==6518==   possibly lost: 0 bytes in 0 blocks
==6518==   still reachable: 0 bytes in 0 blocks
==6518==   suppressed: 0 bytes in 0 blocks
==6518==
==6518== For counts of detected and suppressed errors, rerun with: -v
==6518== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

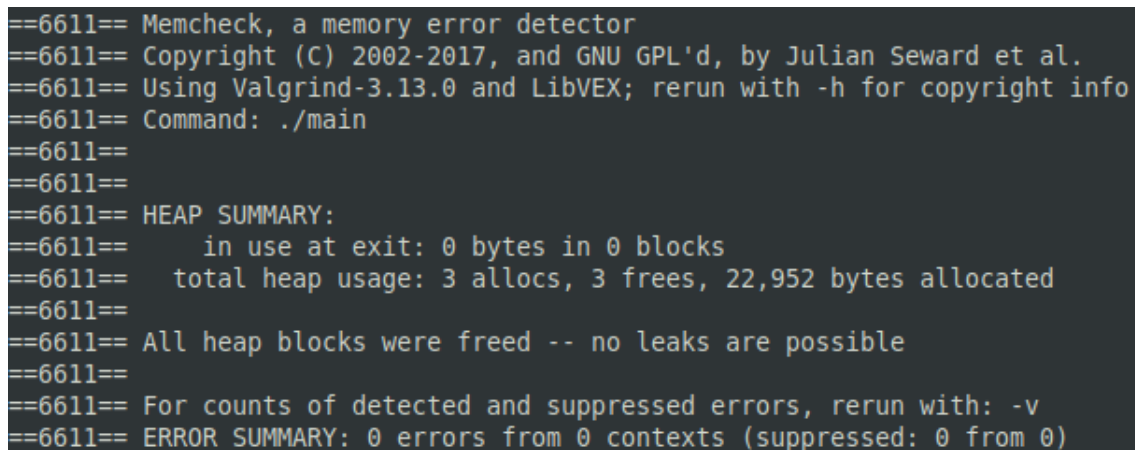
Istotą problemu jest to, że należy usunąć **całą** zarezerwowaną pamięć. Zdarza się, że programista usunie strukturę bądź klasę której już nie potrzebuje, ale zapomni o jej wewnętrznych zasobach. Valgrind informuje o definitywnym wycieku 4000 bajtów, co odpowiada 1000 liczb typu float. W przypadku usunięcia tablicy, ale nie zwolnienia pamięci zajmowanej przez strukturę, Valgrind Również znajdzie wyciek. W tym przypadku będzie to jednak tylko 8 bajtów, odpowiadających rozmiarowi struktury.

Aby całkowicie wyeliminować wyciek pamięci w programie należy zwolnić zarówno strukturę jak i tablicę w niej zawartą, pamiętając o odpowiedniej kolejności:

```
struct problematicStructure{
    int indexNumber;
    float *grades = new float[1000];
};

int main() {

    problematicStructure *s1 = new problematicStructure();
    delete [] s1->grades;
    delete s1;
    return 0;
}
```



```
==6611== Memcheck, a memory error detector
==6611== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==6611== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==6611== Command: ./main
==6611==
==6611==
==6611== HEAP SUMMARY:
==6611==   in use at exit: 0 bytes in 0 blocks
==6611==   total heap usage: 3 allocs, 3 frees, 22,952 bytes allocated
==6611==
==6611== All heap blocks were freed -- no leaks are possible
==6611==
==6611== For counts of detected and suppressed errors, rerun with: -v
==6611== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Powyższe przykłady obrazują, że w języku C oraz C++ do wycieku pamięci można doprowadzić w bardzo prosty sposób. Odpowiedzialność za zarządzanie pamięcią spoczywa na programiście, zatem unikanie wycieków może być trudne. Dodatkowo pokazane przykłady obejmują wąski zakres problemu, który występuje w tym języku. Szczególnie więc warto korzystać z narzędzi takich jak Valgrind, Address Sanitizer, czy menadżer zasobów Visual Studio.



## 4 Wycieki pamięci w języku Rust

Rust różni się od innych języków programowania sposobem zarządzania pamięcią. Sprawdzanie poprawności kodu pod kątem błędnego użycia pamięci odbywa się już na etapie kompilacji, a nie po uruchomieniu aplikacji. Rust nie posiada Garbage Collectora, ale wprowadza mechanizm zwany **Ownership**. Własność ta tak jak w przypadku C++ zmusza programistę do ręcznego zarządzania pamięcią, jednak istotną różnicą jest to, że program napisany w Rust nie skompiluje się, jeżeli wykryty zostanie błąd. Dodatkowo Ownership ułatwia zarządzanie pamięcią, pozwalając na tylko jednego właściciela obiektu, a także poprzez automatyczne zwalnianie zasobów. Oznacza to, że deklarując zmienną, przypisywany jest do niej Ownership, czyli zmienna staje się właścicielem pewnego fragmentu pamięci. Pamięć ta zostanie automatycznie zwolniona w momencie wyjścia poza zakres użycia zmiennej (np. po wyjściu z funkcji). Rust wydaje się zatem być językiem znacznie bardziej bezpiecznym niż C++. Czy to oznacza, że wyciek pamięci jest niemożliwy? Tylko w teorii. Dokumentacja Rust mówi o tym, że język stara się kontrolować wycieki, ale nie jest to całkowicie możliwe, poprzez nieprzewidywalną naturę tego zjawiska.

W języku Rust najłatwiej zobrazować wyciek pamięci wywołując go specjalną funkcją *mem::forget*. Powoduje ona, że Rust usunie zmienną, ale nie uruchomi jej destruktora.

```
use std::mem;
fn main() {
    let xs = vec![0, 1, 2, 3];
    mem::forget(xs);
}
```

Raport Valgrind w przypadku Rust jest znacznie trudniejszy do odczytania niż raport z języka C++. Zawiera dużo więcej informacji, ale są one mniej czytelne dla użytkownika.

```
==6051== HEAP SUMMARY:
==6051==      in use at exit: 16 bytes in 1 blocks
==6051==    total heap usage: 11 allocs, 10 frees, 1,677 bytes allocated
==6051== 16 bytes in 1 blocks are definitely lost in loss record 1 of 1
==6051==    at 0x483021B: malloc (in /usr/lib/valgrind/vgpreload_memcheck
        -x86-linux.so)
==6051==    by 0x1118D9: alloc (alloc.rs:11)
==6051==    by 0x1118D9: __rdl_alloc (alloc.rs:233)
==6051==    by 0x10A0A1: alloc::alloc::alloc (in /home/dsk/Desktop/Rust/
        memleak3/memleak3)
==6051==    by 0x109FED: alloc::alloc::exchange_malloc (in /home/dsk/
        Desktop/Rust/memleak3/memleak3)
==6051==    by 0x10AA49: memleak3::main (in /home/dsk/Desktop/Rust/
        memleak3/memleak3)
==6051==    by 0x10A80B: std::rt::lang_start::{closure} (in /home/dsk/
        Desktop/Rust/memleak3/memleak3)
==6051==    by 0x110445: {{closure}} (rt.rs:49)
==6051==    by 0x110445: std::sys_common::backtrace::__rust_begin_short_
        backtrace (backtrace.rs:135)
==6051==    by 0x11200E: {{closure}} (rt.rs:49)
==6051==    by 0x11200E: std::panicking::try::do_call (panicking.rs:297)
==6051==    by 0x113F67: __rust_maybe_catch_panic (lib.rs:87)
==6051==    by 0x112A95: try<i32, closure> (panicking.rs:276)
==6051==    by 0x112A95: catch_unwind<closure, i32> (panic.rs:388)
==6051==    by 0x112A95: std::rt::lang_start_internal (rt.rs:48)
==6051==    by 0x10A7D1: std::rt::lang_start (in /home/dsk/Desktop/Rust/
        memleak3/memleak3)
==6051==    by 0x10AAF0: main (in /home/dsk/Desktop/Rust/memleak3/memleak3)
==6051== LEAK SUMMARY:
==6051==    definitely lost: 16 bytes in 1 blocks
==6051==    indirectly lost: 0 bytes in 0 blocks
==6051==    possibly lost: 0 bytes in 0 blocks
==6051==    still reachable: 0 bytes in 0 blocks
==6051==    suppressed: 0 bytes in 0 blocks
```

Analizując podsumowanie można zauważyć wyciek 16 bajtów pamięci (odpowiadające 4 liczbom całkowitym z wektora). Odczytanie przyczyny wycieku nie jest jednak proste. Analizując wynik można wywnioskować, że wyciek związany jest z alokacją pamięci w funkcji `main()` programu. Gdyby jednak pominąć funkcję `mem::forget` wyciek nie miałby miejsca.

Ze względu na ilość podawanych informacji, w dalszej części dokumentu będziemy używać skróconych wersji raportów.

Kolejny przykład wycieku również bazuje na użyciu funkcji *mem::forget*, ale tym razem w innym kontekście. W poniższym programie wykorzystana została funkcja *drain()*, która usuwa elementy z wektora i przesuwą pozostałe, tak aby wektor był ciągły. Po usunięciu dwóch pierwszych elementów z wektora, dwa pozostałe powinny trafić na ich miejsce. Używając *mem::forget*, można jednak zapobiec przesuwaniu. Próba odczytania pamięci z początku wektora zakończy się niepowodzeniem i nastąpi wyciek.

```
use std::io::stdin;

fn main() {
    let mut vec = vec![Box::new(0); 4];

    {
        // start draining, vec can no longer be accessed
        let mut drainer = vec.drain(..);

        // pull out two elements and immediately drop them
        drainer.next();
        drainer.next();

        // get rid of drainer, but don't call its destructor
        std::mem::forget(drainer);
    }

    // Oops, vec[0] was dropped, we're reading a pointer into free'd
    // memory!
    println!("{}", vec[0]);
}
```

```
==4133== HEAP SUMMARY:
==4133==      in use at exit: 8 bytes in 2 blocks
==4133==    total heap usage: 22 allocs, 20 frees, 1,886 bytes
                        allocated
==4133==
==4133== LEAK SUMMARY:
==4133==    definitely lost: 8 bytes in 2 blocks
==4133==    indirectly lost: 0 bytes in 0 blocks
==4133==    possibly lost: 0 bytes in 0 blocks
==4133==    still reachable: 0 bytes in 0 blocks
==4133==    suppressed: 0 bytes in 0 blocks
```

Na raporcie Valgrind widać wyciek 8 bajtów pamięci. Powyższe wycieki są jednak stworzone w kontrolowanych warunkach - są wymuszone przez programistę. W języku Rust można jednak wykonać wyciek bez informowania o tym kompilatora. Wyciek związany z cyklami referencji nie zostanie poprawnie wykryty, a program się skompiluje. Kod poniżej tworzy dwie listy, a następnie modyfikuje je w ten sposób, aby ostatni element w każdej liście posiadał referencję to innej listy. W ten sposób powstaje pętla.

```
use std::rc::Rc;
use std::cell::RefCell;
use crate::List::{Cons, Nil};

#[derive(Debug)]
enum List {
    Cons(i32, RefCell<Rc<List>>),
    Nil,
}

impl List {
    fn tail(&self) -> Option<&RefCell<Rc<List>>> {
        match self {
            Cons(_, item) => Some(item),
            Nil => None,
        }
    }
}

fn main() {
    let a = Rc::new(Cons(5, RefCell::new(Rc::new(Nil))));
    let b = Rc::new(Cons(10, RefCell::new(Rc::clone(&a))));

    if let Some(link) = a.tail() {
        *link.borrow_mut() = Rc::clone(&b);
    }
}
```

Tym razem Valgrind informuje użytkownika o problemie z referencjami. Nie udaje się poprawnie usunąć wszystkich powiązań, gdyż w nieskończoność się one zapętłają. Skutkiem jest wyciek pamięci.

```
a initial rc count = 1
a next item = Some(RefCell { value: Nil })
a rc count after b creation = 2
b initial rc count = 1
b next item = Some(RefCell { value: Cons(5, RefCell { value: Nil }) })
b rc count after changing a = 2
a rc count after changing a = 2
==5706==
==5706== HEAP SUMMARY:
==5706==      in use at exit: 40 bytes in 2 blocks
==5706==    total heap usage: 21 allocs , 19 frees , 2,853 bytes allocated
==5706==
==5706== LEAK SUMMARY:
==5706==    definitely lost: 20 bytes in 1 blocks
==5706==    indirectly lost: 20 bytes in 1 blocks
==5706==    possibly lost: 0 bytes in 0 blocks
==5706==    still reachable: 0 bytes in 0 blocks
==5706==    suppressed: 0 bytes in 0 blocks
```

Podobnym przykładem jest program zawierający strukturę, która posiada referencje do samej siebie.

```
use std::cell::RefCell;
use std::rc::Rc;

struct Cycle {
    cell: RefCell<Option<Rc<Cycle>>>,
}

impl Drop for Cycle {
    fn drop(&mut self) {
        println!("freed");
    }
}

fn main() {
    let cycle = Rc::new(Cycle { cell: RefCell::new(None) });
    *cycle.cell.borrow_mut() = Some(cycle.clone());
}
```

```
==7086== HEAP SUMMARY:
==7086==      in use at exit: 16 bytes in 1 blocks
==7086==    total heap usage: 11 allocs, 10 frees, 1,677 bytes allocated
==7086==
==7086== LEAK SUMMARY:
==7086==      definitely lost: 16 bytes in 1 blocks
==7086==      indirectly lost: 0 bytes in 0 blocks
==7086==      possibly lost: 0 bytes in 0 blocks
==7086==      still reachable: 0 bytes in 0 blocks
==7086==      suppressed: 0 bytes in 0 blocks
```

W tym przypadku jednak Valgrind nie wskazuje problemu w tak wyraźny sposób. Wyciek jest jednak definitywny.