ΕΡΓΑΣΤΗΡΙΟ 9

Δυναμική ανάλυση κώδικα

Η **δυναμική ανάλυση του κώδικα ενός προγράμματος** είναι ανάλυση λογισμικού που γίνεται με την εκτέλεση των προγραμμάτων που προκύπτουν από αυτό το σύστημα λογισμικού σε έναν πραγματικό ή εικονικό επεξεργαστή. Για να είναι αποδοτική, πρέπει το πρόγραμμα προς ανάλυση να εκτελεστεί με αρκετές εισόδους ελέγχου για να εμφανιστεί ενδιαφέρουσα συμπεριφορά.

Υπάρχουν πολλές εφαρμογές που μας επιτρέπουν να κάνουμε δυναμική ανάλυση του κώδικα. Μια από αυτές είναι το Valgrind, μια συλλογή εργαλείων που κάνουν πολλά πράγματα, αλλά σε αυτό το εργαστήριο θα επικεντρωθούμε στα εργαλεία που κάνουν:

- **Profiling**: Η διαδικασία του profiling είναι μια σημαντική πτυχή κατά την υλοποίηση ενός προγράμματος και έγκειται στον προσδιορισμό των μερών του κώδικα που είναι χρονοβόρα (θέλουν πολύ χρόνο για να τρέξουν) και θα πρέπει να προσδιοριστούν και να γραφούν ξανά αφού η ταχύτερη εκτέλεση ενός προγράμματος είναι πάντοτε επιθυμητή. Σε πολύ μεγάλα projects, το profiling μπορεί να δώσει σημαντικές πληροφορίες όχι μόνο καθορίζοντας τα τμήματα του προγράμματος που είναι πιο αργά σε εκτέλεση, αλλά μπορεί επίσης να σας βοηθήσει τον προγραμματιστή να βρει πολλά άλλα στατιστικά στοιχεία μέσω των οποίων μπορούν να εντοπιστούν και να διαλυθούν πολλά πιθανά σφάλματα.
- **Memory checking**: Η διαδικασία της ανίχνευσης κακοδιαχείρισης μνήμης είναι σημαντική για να αποκαλύψει τις διαρροές μνήμης, τα σφάλματα αποδέσμευσης μνήμης, κλπ. και να κάνει ένα πρόγραμμα πιο αποδοτικό στη χρήση της μνήμης.

Στο εργαστήριο αυτό θα χρησιμοποιήσουμε το εργαλείο gprof που επιτρέπει το profiling και το εργαλείο Valgrind που μας επιτρέπει να κάνουμε και profiling (μέσω της παραμέτρους **cachegrind)** αλλά και memory checking (μέσω της παραμέτρου **memcheck**). Από το εργαλείο Valgrind θα χρησιμοποιήσουμε το memcheck.

Άσκηση 1 - Χρησιμοποίηση memcheck.

Μεταγλωττίστε το πιο κάτω πρόγραμμα test.c με την πιο κάτω εντολή.

```
gcc test.c -o test
```

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

int main()
{
  char *p;

  // Allocation #1 of 19 bytes
  p = (char *) malloc(19);

  // Allocation #2 of 12 bytes
```

```
  p = (char *) malloc(12);
  free(p);

  // Allocation #3 of 16 bytes
  p = (char *) malloc(16);

  return 0;
}
```

Για να ελέγξετε για memory leaks κατά την διάρκεια της εκτέλεσης του προγράμματος, δώστε την εντολή:

```
valgrind --tool=memcheck --leak-check=full --show-reachable=yes
--num-callers=20 --track-fds=yes --track-origins=yes ./test
```

Αυτή η εντολή θα εμφανίσει στην οθόνη:

```
==8567== Memcheck, a memory error detector
==8567== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==8567== Using Valgrind-3.12.0 and LibVEX; rerun with -h for copyright info
==8567== Command: ./test
==8567==
==8567==
==8567== FILE DESCRIPTORS: 3 open at exit.
==8567== Open file descriptor 2: /dev/pts/0
==8567==    <inherited from parent>
==8567==
==8567== Open file descriptor 1: /dev/pts/0
==8567==    <inherited from parent>
==8567==
==8567== Open file descriptor 0: /dev/pts/0
==8567==    <inherited from parent>
==8567==
==8567==
==8567== HEAP SUMMARY:
==8567==     in use at exit: 35 bytes in 2 blocks
==8567==   total heap usage: 3 allocs, 1 frees, 47 bytes allocated
==8567==
==8567== 16 bytes in 1 blocks are definitely lost in loss record 1 of 2
==8567==    at 0x4C29BE3: malloc (vg_replace_malloc.c:299)
==8567==    by 0x4005B6: main (test.c:16)
==8567==
==8567== 19 bytes in 1 blocks are definitely lost in loss record 2 of 2
==8567==    at 0x4C29BE3: malloc (vg_replace_malloc.c:299)
==8567==    by 0x40058E: main (test.c:9)
==8567==
==8567== LEAK SUMMARY:
==8567==    definitely lost: 35 bytes in 2 blocks
==8567==    indirectly lost: 0 bytes in 0 blocks
==8567==      possibly lost: 0 bytes in 0 blocks
==8567==    still reachable: 0 bytes in 0 blocks
==8567==         suppressed: 0 bytes in 0 blocks
==8567==
==8567== For counts of detected and suppressed errors, rerun with: -v
==8567== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
```

Βλέποντας τον κώδικα μπορούμε να συμπεράνουμε ότι: Η αναφορά στα 19 byte που δεσμεύτηκαν αρχικά (με το πρώτο malloc) χάνεται όταν ο δείκτης p γίνεται malloc για δεύτερη φορά και έτσι αυτές οι θέσεις μνήμης (19 bytes) μένουν απροσπέλαστες (19 bytes in 1 block definitely lost). Έτσι εδώ έχουμε διαρροή μνήμης (memory leak). Το Valgrind μας δείχνει που δεσμεύτηκε αυτός ο χώρος, είναι στο test.c, στην γραμμή 9. Η δεύτερη δέσμευση (12 byte) δεν φαίνεται στα αποτελέσματα επειδή αποδεσμεύεται (free) κανονικά (δεν υπάρχει διαρροή μνήμης). Η τρίτη δέσμευση φαίνεται στα αποτελέσματα για διαρροή μνήμης (memory leak) αν και υπάρχει ο δείκτης p που συνεχίζει να δείχνει στο χώρο δέσμευσης πριν τον τερματισμό του προγράμματος. Εδώ το πρόβλημα είναι ότι δεν αποδεσμεύτηκαν αυτές οι θέσεις πριν τελειώσει το πρόγραμμα. Οπότε και αυτό είναι ένα memory leak και το Valgrind δείχνει την γραμμή στο πρόγραμμα (test.c γραμμή 16).

Μεταγλωττίστε το πιο κάτω πρόγραμμα test2.c με την πιο κάτω εντολή.

```
gcc test2.c -o test2
```

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

#define SIZE 10

int main()
{
  // free in not called
  char *waste = (char *)malloc(SIZE *sizeof(char));

  // uninitialized pointer
  int *a;
  printf("%d\n", *a);

  // write past end of array
  waste[SIZE] = 0;

  return 0;
}
```

Αν εκτελέσετε την εντολή

```
valgrind --tool=memcheck --leak-check=full --show-reachable=yes --num-callers=20 --track-fds=yes --track-origins=yes ./test2
```

θα δείτε ότι υπάρχει ένα memory leak (10 bytes definitely lost). Αν θέλετε να δείτε κι άλλες πληροφορίες σχετικά με το πρόβλημα που προκύπτει από την εκτύπωση μη-αρχικοποιημένου δείκτη ή την προσπέλαση θέσης που βρίσκεται εκτός της δεσμευμένης περιοχής του πίνακα waste εκτελέστε την πιο πάνω εντολή valgrid προσθέτοντας ακόμα ένα όρισμα -v το οποίο παρουσιάσει πιο λεπτομερή ανάλυση των προβλημάτων (verbose).

Άσκηση 2 - Χρησιμοποίηση memcheck.

1. Από την ιστοσελίδα του μαθήματος κατεβάστε τα αρχεία list.h και list.c. Το list.c περιέχει ένα driver αντικείμενου, άρα για την μεταγλώττιση του χρησιμοποιήστε στην εντολή, -DDEBUG.
2. Ελέγξτε με το Valgrind για διαρροές μνήμης.
3. Διορθώστε τα λάθη που υπάρχουν.

Άσκηση 3 - Χρησιμοποίηση gprof.

Η χρήση του εργαλείου gprof δεν είναι καθόλου περίπλοκη. Απλά πρέπει να εκτελέσετε τα εξής βήματα:

- Ενεργοποιήστε το profiling κατά τη μεταγλώττιση του κώδικα. Αυτό γίνεται με την παράμετρο -gp κατά την κλήση του gcc
- Εκτελέστε τον κώδικα προγράμματος για να δημιουργήσετε τα profiling data (αρχείο με όνομα gmon.out)
- Τρέξτε το εργαλείο gprof πάνω στα profiling data (που δημιουργήθηκαν στο προηγούμενο βήμα).

Το τελευταίο βήμα παράγει ένα αρχείο ανάλυσης που είναι σε ανθρώπινη αναγνώσιμη μορφή. Αυτό το αρχείο περιέχει μερικούς πίνακες (flat profile και call graph) εκτός από κάποιες άλλες πληροφορίες. Το flat profile δίνει μια επισκόπηση των πληροφοριών χρονισμού των λειτουργιών όπως η κατανάλωση χρόνου για την εκτέλεση μιας συγκεκριμένης λειτουργίας, πόσες φορές κλήθηκε κλπ. Από την άλλη πλευρά, το call graph παρέχει το δέντρο των κλήσεων των συναρτήσεων που εμπλέκονται στο πρόγραμμα. Έτσι, έτσι μπορεί κανείς να πάρει μια ιδέα του χρόνου εκτέλεσης που δαπανάται και στις συναρτήσεις και στις συναρτήσεις που καλούνται μέσα σε συναρτήσεις κτλ.

Μεταγλωττίστε τα πιο κάτω προγράμματα test_gprof.c και test_gprof_new.c με την πιο κάτω εντολή.

```
//test_gprof.c
#include<stdio.h>

void new_func1(void);

void func1(void)
{
    printf("\n Inside func1 \n");
    int i = 0;

    for(;i<0xffffffff;i++);
    new_func1();

    return;
}

static void func2(void)
{
```

```
    printf("\n Inside func2 \n");
    int i = 0;

    for(;i<0xfffffffaa;i++);
    return;
}

int main(void)
{
    printf("\n Inside main()\n");
    int i = 0;

    for(;i<0xffffff;i++);
    func1();
    func2();

    return 0;
}
```

```
//test_gprof_new.c
#include<stdio.h>

void new_func1(void)
{
    printf("\n Inside new_func1()\n");
    int i = 0;

    for(;i<0xffffffee;i++);

    return;
}
```

Βήμα 1: Μεταγλώττιση

```
gcc -pg test_gprof.c test_gprof_new.c -o test_gprof
```

Από το man page του gcc:
-pg : Generate extra code to write profile information suitable for the analysis program gprof. You must use this option when compiling the source files you want data about, and you must also use it when linking.

Βήμα 2: Εκτέλεση προγράμματος

```
./test_gprof
```

Μετά την ολοκλήρωση της εκτέλεσης του προγράμματος παράγεται το αρχείο gmon.out

Βήμα 3: Εκτέλεση gprof tool

```
gprof test_gprof gmon.out > analysis.txt
```

Δείτε τα περιεχόμενα του αρχείου analysis.txt

```
Flat profile:

Each sample counts as 0.01 seconds.
%    cumulative self          self   total
time seconds    seconds calls s/call s/call name
33.86 15.52     15.52    1    15.52  15.52  func2
33.82 31.02     15.50    1    15.50  15.50  new_func1
33.29 46.27     15.26    1    15.26  30.75  func1
0.07  46.30     0.03                       main

% the percentage of the total running time of the
time program used by this function.

cumulative a running sum of the number of seconds accounted
seconds for by this function and those listed above it.

self the number of seconds accounted for by this
seconds function alone. This is the major sort for this
listing.

calls the number of times this function was invoked, if
this function is profiled, else blank.

self the average number of milliseconds spent in this
ms/call function per call, if this function is profiled,
else blank.

total the average number of milliseconds spent in this
ms/call function and its descendents per call, if this
function is profiled, else blank.

name the name of the function. This is the minor sort
for this listing. The index shows the location of
the function in the gprof listing. If the index is
in parenthesis it shows where it would appear in
the gprof listing if it were to be printed.

Call graph (explanation follows)

granularity: each sample hit covers 2 byte(s) for 0.02% of 46.30 seconds

index % time self children called name

[1]   100.0  0.03  46.27          main [1]
             15.26 15.50    1/1       func1 [2]
             15.52 0.00     1/1       func2 [3]
-----------------------------------------------
             15.26 15.50    1/1       main [1]
[2]   66.4   15.26 15.50    1     func1 [2]
             15.50 0.00     1/1       new_func1 [4]
-----------------------------------------------
             15.52 0.00     1/1       main [1]
[3]   33.5   15.52 0.00     1     func2 [3]
-----------------------------------------------
             15.50 0.00     1/1       func1 [2]
```

```
[4] 33.5     15.50 0.00     1     new_func1 [4]
-----------------------------------------------
```

This table describes the call tree of the program, and was sorted by
the total amount of time spent in each function and its children.

Each entry in this table consists of several lines. The line with the
index number at the left hand margin lists the current function.
The lines above it list the functions that called this function,
and the lines below it list the functions this one called.
This line lists:
index A unique number given to each element of the table.
Index numbers are sorted numerically.
The index number is printed next to every function name so
it is easier to look up where the function in the table.

% time This is the percentage of the `total' time that was spent
in this function and its children. Note that due to
different viewpoints, functions excluded by options, etc,
these numbers will NOT add up to 100%.

self This is the total amount of time spent in this function.

children This is the total amount of time propagated into this
function by its children.

called This is the number of times the function was called.
If the function called itself recursively, the number
only includes non-recursive calls, and is followed by
a `+' and the number of recursive calls.

name The name of the current function. The index number is
printed after it. If the function is a member of a
cycle, the cycle number is printed between the
function's name and the index number.

For the function's parents, the fields have the following meanings:

self This is the amount of time that was propagated directly
from the function into this parent.

children This is the amount of time that was propagated from
the function's children into this parent.

called This is the number of times this parent called the
function `/' the total number of times the function
was called. Recursive calls to the function are not
included in the number after the `/'.

name This is the name of the parent. The parent's index
number is printed after it. If the parent is a
member of a cycle, the cycle number is printed between
the name and the index number.

If the parents of the function cannot be determined, the word
`' is printed in the `name' field, and all the other

```
fields are blank.

For the function's children, the fields have the following meanings:

self This is the amount of time that was propagated directly
from the child into the function.

children This is the amount of time that was propagated from the
child's children to the function.

called This is the number of times the function called
this child `/' the total number of times the child
was called. Recursive calls by the child are not
listed in the number after the `/'.

name This is the name of the child. The child's index
number is printed after it. If the child is a
member of a cycle, the cycle number is printed
between the name and the index number.

If there are any cycles (circles) in the call graph, there is an
entry for the cycle-as-a-whole. This entry shows who called the
cycle (as parents) and the members of the cycle (as children.)
The `+' recursive calls entry shows the number of function calls that
were internal to the cycle, and the calls entry for each member shows,
for that member, how many times it was called from other members of
the cycle.

Index by function name

[2] func1 [1] main
[3] func2 [4] new_func1
```

Άσκηση 4.

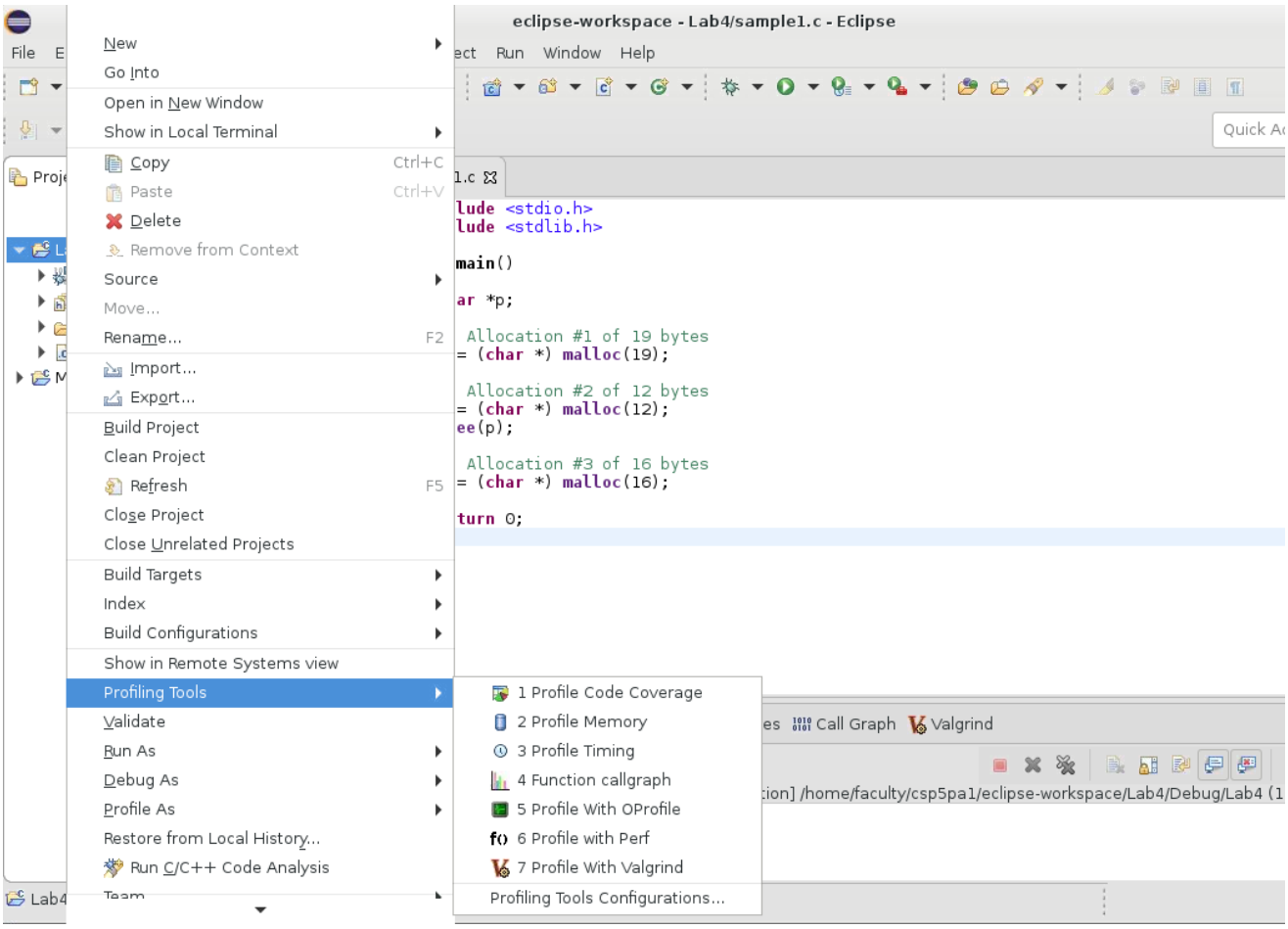
Χρησιμοποιήστε το Valgrind για να ελέγξετε για διαρροές μνήμης για την τρίτη σας εργασία AS3.

Για profiling χρησιμοποιήστε το gprof.

Valgrind with eClipse

Το εργαλείο Valgrind μπορεί να το χρησιμοποιήσετε και από το eClipse. Πρώτα πρέπει να εγκαταστήσετε το plug-in Linux Tools.

Επιλέξτε το project που θέλετε να κάνετε δυναμική ανάλυση διαρροών μνήμης και μετά δεξί κλικ, Profiling Tools → Profile with Valgrind:



Και θα δείτε τα πιο κάτω αποτελέσματα

sample1.c

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

int main()
{
  char *p;

  // Allocation #1 of 19 bytes
  p = (char *) malloc(19);

  // Allocation #2 of 12 bytes
  p = (char *) malloc(12);
  free(p);

  // Allocation #3 of 16 bytes
  p = (char *) malloc(16);

  return 0;
}
```

Problems Tasks Console Properties Call Graph Valgrind

Lab4 (1) [memcheck] valgrind (10/31/17, 1:06 PM)

- 16 bytes in 1 blocks are definitely lost in loss record 1 of 2 [PID: 25061]
- 19 bytes in 1 blocks are definitely lost in loss record 2 of 2 [PID: 25061]

Παράρτημα

From Valgrind website (http://valgrind.org/docs/manual/mc-manual.html)

```
      Pointer chain            AAA Leak Case   BBB Leak Case
      -------------            -------------   -------------
(1)   RRR ------------> BBB                    DR
(2)   RRR ---> AAA ---> BBB    DR              IR
(3)   RRR               BBB                    DL
(4)   RRR      AAA ---> BBB    DL              IL
(5)   RRR ------?-----> BBB                    (y)DR, (n)DL
(6)   RRR ---> AAA -?-> BBB    DR              (y)IR, (n)DL
(7)   RRR -?-> AAA ---> BBB    (y)DR, (n)DL    (y)IR, (n)IL
(8)   RRR -?-> AAA -?-> BBB    (y)DR, (n)DL    (y,y)IR, (n,y)IL, (_,n)DL
(9)   RRR      AAA -?-> BBB    DL              (y)IL, (n)DL


Pointer chain legend:
- RRR: a root set node or DR block
- AAA, BBB: heap blocks
- --->: a start-pointer
- -?->: an interior-pointer


Leak Case legend:
- DR: Directly reachable
- IR: Indirectly reachable
- DL: Directly lost
- IL: Indirectly lost
- (y)XY: it's XY if the interior-pointer is a real pointer
- (n)XY: it's XY if the interior-pointer is not a real pointer
- (_)XY: it's XY in either case
```

Every possible case can be reduced to one of the above nine. Memcheck merges some of these cases in its output, resulting in the following four leak kinds.

- "Still reachable". This covers cases 1 and 2 (for the BBB blocks) above. A start-pointer or chain of start-pointers to the block is found. Since the block is still pointed at, the programmer could, at least in principle, have freed it before program exit. "Still reachable" blocks are very common and arguably not a problem. So, by default, Memcheck won't report such blocks individually.
- "Definitely lost". This covers case 3 (for the BBB blocks) above. **This means that no pointer to the block can be found**. The block is classified as "lost", because the programmer could not possibly have freed it at program exit, since no pointer to it exists. This is likely a symptom of having lost the pointer at some earlier point in the program. Such cases should be fixed by the programmer.
- "Indirectly lost". This covers cases 4 and 9 (for the BBB blocks) above. This means that the block is lost, not because there are no pointers to it, but rather because all the blocks that point to it are themselves lost. For example, if you have a binary tree and the root node is lost, all its children nodes will be indirectly lost. Because the problem will disappear if the definitely lost block that caused the indirect leak is fixed, Memcheck won't report such blocks individually by default.
- "Possibly lost". This covers cases 5--8 (for the BBB blocks) above. This means that a chain of one or more pointers to the block has been found, but at least one of the pointers is an interior-pointer. This could just be a random value in memory that happens to point into a block, and so you shouldn't consider this ok unless you know you have interior-pointers.

(Note: This mapping of the nine possible cases onto four leak kinds is not necessarily the best way that leaks could be reported; in particular, interior-pointers are treated inconsistently. It is possible the categorisation may be improved in the future.)

Furthermore, if suppressions exists for a block, it will be reported as "suppressed" no matter what which of the above four kinds it belongs to.

From Valgrind FAQ:

With Memcheck's memory leak detector, what's the difference between "definitely lost", "indirectly lost", "possibly lost", "still reachable", and "suppressed"?

The details are in the Memcheck section of the user manual.

In short:

- **definitely lost** means your program is leaking memory -- fix those leaks!
- **indirectly lost** means your program is leaking memory in a pointer-based structure. (E.g. if the root node of a binary tree is "definitely lost", all the children will be "indirectly lost".) If you fix the definitely lost leaks, the indirectly lost leaks should go away.
- **possibly lost** means your program is leaking memory, unless you're doing funny things with pointers. This is sometimes reasonable.
  Use --show-possibly-lost=no if you don't want to see these reports.

- **still reachable** means your program is probably ok -- it didn't free some memory it could have. This is quite common and often reasonable.
  Don't use --show-reachable=yes if you don't want to see these reports.
- **suppressed** means that a leak error has been suppressed. There are some suppressions in the default suppression files. You can ignore suppressed errors.