Εντοπισμός σφαλμάτων στη διαχείριση μνήμης

Στο παρόν φυλλάδιο περιγράφουμε τα πιο συχνά και σοβαρά λάθη που σχετίζονται με δυναμική δέσμευση μνήμης, και κάποια εργαλεία που βοηθούν στον εντοπισμό αυτών των σφαλμάτων.

Είδος σφάλματος: Memory leak

Προκύπτει όταν η δυναμικά δεσμευμένη μνήμη δεν αποδεσμεύεται όταν πάψει να χρειάζεται. Σε μεγάλα ή μακρόβια προγράμματα αυτό μπορεί να εξαντλήσει τη διαθέσιμη μνήμη. Γι' αυτό είναι σημαντικό να αποκτήσετε τη συνήθεια να αποδεσμεύετε τη δυναμικά δεσμευμένη μνήμη όταν δε χρειάζεται πια.

Παράδειγμα:

```
1  int *p, x;
2
3  p = (int*) malloc(sizeof(int));
4  p = &x;
```

Μετά τη γραμμή 4 ο δείκτης p δείχνει στη διεύθυνση του x και δεν υπάρχει άλλος δείκτης προς τη μνήμη που δεσμεύτηκε από τη malloc, επομένως δεν είναι πια δυνατό να αποδεσμευθεί αυτή.

Παράδειγμα:

```
1   void addObj(int *objects[], int pos, int val)
2   {
3       int *newObj;
4
5       newObj = (int*) malloc(sizeof(int));
6       *newObj = val;
7
8       if (objects[pos] == NULL) {
9           objects[pos] = newObj;
10      }
11  }
```

Αν η συνθήκη της γραμμής 8 είναι ψευδής, τότε η μνήμη που δεσμεύτηκε στη γραμμή 5 δε μπορεί να αποδεσμευθεί ποτέ γιατί μετά την επιστροφή από τη συνάρτηση δεν υπάρχει πια πρόσβαση σε αυτή (η μεταβλητή newObj είναι τοπική και η διεύθυνση δεν επιστρέφεται).

Είδος σφάλματος: Heap buffer overflow

Προκύπτει όταν γίνεται προσπέλαση μνήμης πέρα του χώρου που έχει δεσμευτεί. Ένα τέτοιο σφάλμα εισάγει τρωτά σημεία στην ασφάλεια του συστήματος όπου εκτελείται το πρόγραμμα.

Παράδειγμα:

```
1  char *name;
2  name = (char *) malloc(5);
3  scanf("%s", name);
```

Το %s δεν περιορίζει το πλήθος χαρακτήρων που θα διαβάσει η scanf, επομένως αν δοθεί ως είσοδος ένα όνομα με περισσότερους από 4 χαρακτήρες, θα προκύψει heap overflow.

Είδος σφάλματος: Use after free

Προκύπτει όταν γίνεται προσπέλαση μνήμης αφότου αυτή έχει αποδεσμευθεί. Αν μετά την αποδέσμευση η μνήμη επαναχρησιμοποιηθεί από άλλη malloc μπορεί να αποθηκευτούν διαφορετικά δεδομένα σε αυτή. Το σφάλμα μπορεί να οδηγήσει σε αλλοίωση των δεδομένων του προγράμματος ή και να δώσει τη δυνατότητα σε κακόβουλο χρήστη να εισάγει δικό του κώδικα προς εκτέλεση.

Το σφάλμα αυτό είναι ύπουλο γιατί ενδέχεται τα περιεχόμενα της αποδεσμευμένης μνήμης να μην αλλάξουν άμεσα, δίνοντας στον προγραμματιστή την ψευδαίσθηση ότι το πρόγραμμα λειτουργεί σωστά.

Παράδειγμα:

```
1  char *msg;
2
3  msg = (char *) malloc(20);
4  strcpy(msg, "Some message...");
5  free(msg);
6  printf("%s\n", msg);
```

Η printf καλεί malloc στα πλαίσια της λειτουργίας της. Εάν η μνήμη που δεσμεύεται από την printf τύχει να είναι η ίδια που είχε δεσμευθεί για το msg, τότε θα αποθηκευτεί κάτι διαφορετικό σε αυτή και το μήνυμα που θα εκτυπωθεί δε θα είναι το "Some message..."

Είναι καλή τακτική μετά από free να τίθεται ο δείκτης σε NULL (π.χ. msg = NULL;). Έτσι, αν γίνει προσπάθεια προσπέλασης αυτού, θα προκληθεί segmentation fault και θα αντιληφθεί άμεσα ο προγραμματιστής ότι έχει γίνει λάθος.

Χρειάζεται ιδιαίτερη προσοχή όταν πολλοί δείκτες δείχνουν στην ίδια μνήμη, γιατί μια free μέσω οποιουδήποτε από αυτούς καθιστά όλους τους υπόλοιπους μη έγκυρους.

Παράδειγμα:

```
1  int *pnum, *copy;
2
3  pnum = (int *) malloc(1*sizeof(int));
4  *pnum = 15;
5  copy = pnum;
6  free(copy);
7  copy = NULL;
8  printf("%d", *pnum);
```

Στη γραμμή 4 ο δείκτης copy περιέχει την ίδια διεύθυνση με τον pnum, δηλαδή δείχνει στη μνήμη που δεσμεύθηκε στη γραμμή 2 και περιέχει το 15. Μετά τη free, ο copy γίνεται NULL, αλλά ο pnum συνεχίζει να δείχνει στην αποδεσμευμένη μνήμη με αποτέλεσμα στην printf να γίνεται παράνομη προσπέλαση μνήμης.

Εργαλείο: AddressSanitizer (linux, macOS) και LeakSanitizer (linux)

Πρόκειται για εργαλεία που παρέχονται μέσω συγκεκριμένων μεταγλωττιστών όπως gcc, clang/llvm και ανιχνεύουν σφάλματα μνήμης (buffer overflows, use-after-free κ.α.). Συχνά ο LeakSanitizer συμπεριλαμβάνεται στον AddressSanitizer.

Μεταγλώττιση κώδικα και χρήση

Μεταγλωττίστε το πρόγραμμά σας με την επιλογή `-fsanitize=address` για να ανιχνευθεί τυχόν παράνομη προσπέλαση μνήμης κατά την εκτέλεση (και memory leaks αν συμπεριλαμβάνεται ο LeakSanitizer).

Μεταγλωττίστε το πρόγραμμά σας με την επιλογή `-fsanitize=leak` για να πάρετε πληροφορίες για memory leaks από τον LeakSanitizer.

Δυστυχώς για μοντέρνο macOS δεν υπάρχει εύκολος τρόπος να γίνει έλεγχος για leaks μέσω τερματικού.

Εργαλείο Valgrind (linux) (προφέρεται "βάλγκριντ")

Πρόκειται για εργαλείο ανάλυσης μνήμης (memory profiler) που εκτελεί το πρόγραμμα και ανιχνεύει παράνομες προσπελάσεις μνήμης, memory leaks κτλ. Είναι διαθέσιμο για linux αλλά όχι για πρόσφατες εκδόσεις του macOS. Είναι πιο αργό από τον address sanitizer αλλά μπορεί να εντοπίσει κάποια παραπάνω λάθη.

Μεταγλώττιση κώδικα και χρήση

Μεταγλωττίστε το πρόγραμμα σας με την επιλογή -g. Μη συμπεριλάβετε το -fsanitize=address αν θέλετε να χρησιμοποιήσετε valgrind, γιατί δε συνεργάζονται και δε θα λειτουργήσει σωστά.

Εκτελέστε το πρόγραμμα γράφοντας valgrind και ακολούθως την εντολή εκτέλεσης ανακατευθύνοντας τη συμβατική έξοδο στο /dev/null, για παράδειγμα:

```
valgrind ./hw1 arg1 arg2 arg3 < input > /dev/null
```

Το /dev/null είναι ένα ειδικό αρχείο στο οποίο ότι γράφεται απορρίπτεται. Η ανακατεύθυνση σε αυτό γίνεται να ελαχιστοποιηθεί ο "θόρυβος" στην οθόνη.

Παράδειγμα 1 (heap buffer overflow)

```
1     int main (int argc, char *argv[])     {
2
3         int **p, i;
4
5         p = (int**) malloc(sizeof(int)*10);
6         for (i = 0; i < 10; i++) {
7             p[i] = (int*) malloc(sizeof(int));
8         }
9         for (i = 0; i < 10; i++) {
10            printf("%p\n", p[i]);
11        }
12        free(p);
13        return 0;
14    }
```

Έξοδος AddressSanitizer για το παράδειγμα 1

```
==23743==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x604000000038 at pc 0x55af0e9182b6
bp 0x7ffec95ea500 sp 0x7ffec95ea4f0
WRITE of size 8 at 0x604000000038 thread T0
   #0 0x55af0e9182b5 in main /srv/homes/vdoufexi/test.c:7
   #1 0x7f73b8d57082 in __libc_start_main ../csu/libc-start.c:308
   #2 0x55af0e91818d in _start (/srv/homes/vdoufexi/test+0x118d)

0x604000000038 is located 0 bytes to the right of 40-byte region [0x604000000010,0x604000000038)
allocated by thread T0 here:
   #0 0x7f73b9032808 in __interceptor_malloc ../../../../src/libsanitizer/asan/asan_malloc_linux.cc:144
   #1 0x55af0e918266 in main /srv/homes/vdoufexi/test.c:5
   #2 0x7f73b8d57082 in __libc_start_main ../csu/libc-start.c:308
```

Ερμηνεία

Στη γραμμή 7 του προγράμματος προσπαθούμε να γράψουμε 8 bytes (WRITE of size 8) σε σημείο που βρίσκεται πέρα (bytes to the right) από τη μνήμη που δεσμεύτηκε στη γραμμή 5 του προγράμματος (allocated by thread T0 here) και η οποία έχει μέγεθος 40 bytes (40-byte region).

Τρόπος σκέψης για τον εντοπισμό του προβλήματος

Ελέγχουμε πρώτα τα όρια της επανάληψης για να βεβαιωθούμε ότι δε βγαίνουμε κατά λάθος εκτός ορίων του πίνακα σε αυτό το σημείο.

Εφόσον αυτά είναι σωστά, η μόνη άλλη εξήγηση είναι ότι η περιοχή δεσμευμένης μνήμης στην οποία δείχνει ο δείκτης p είναι μικρότερη από όσο θα έπρεπε. Παρατηρούμε πως ο πίνακας p έχει μέγεθος 10 και αποτελείται από δείκτες (8 bytes), επομένως θα έπρεπε να καταλαμβάνει 80 bytes και όχι 40.

Πράγματι, στη γραμμή 5 έχουμε κατά λάθος sizeof(int) αντί για το σωστό sizeof(int*).

Αφού διορθωθεί το παραπάνω λάθος, ένας επανέλεγχος παράγει μήνυμα για memory leak:

```
==23768==ERROR: LeakSanitizer: detected memory leaks

Direct leak of 40 byte(s) in 10 object(s) allocated from:
   #0 0x7f9507c97808 in __interceptor_malloc ../../../../src/libsanitizer/asan/asan_malloc_linux.cc:144
   #1 0x5618abbc0292 in main /srv/homes/vdoufexi/test.c:7
   #2 0x7f95079bc082 in __libc_start_main ../csu/libc-start.c:308

SUMMARY: AddressSanitizer: 40 byte(s) leaked in 10 allocation(s).
```

Ερμηνεία

Στη γραμμή 7 δεσμεύτηκαν 40 bytes για 10 συνολικά αντικείμενα, για τα οποία προέκυψε "direct leak", δηλαδή δεν αποδεσμεύθηκε η μνήμη που καταλαμβάνουν. Πράγματι, δεν αποδεσμεύεται πουθενά η μνήμη που δεσμεύτηκε για κάθε ένα p[i].

Έξοδος valgrind για το παράδειγμα 1

Το valgrind έχει το πλεονέκτημα ότι με μία εκτέλεση αναφέρει τόσο τις παράνομες προσπελάσεις μνήμης όσο και τα memory leaks.

```
==23893== Invalid write of size 8
==23893==    at 0x1091D3: main (test.c:7)
==23893==  Address 0x4a67068 is 0 bytes after a block of size 40 alloc'd
==23893==    at 0x483B7F3: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==23893==    by 0x1091A6: main (test.c:5)
==23893==
==23893== Invalid read of size 8
==23893==    at 0x1091FD: main (test.c:10)
==23893==  Address 0x4a67068 is 0 bytes after a block of size 40 alloc'd
==23893==    at 0x483B7F3: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==23893==    by 0x1091A6: main (test.c:5)
==23893==
==23893==
==23893== HEAP SUMMARY:
==23893==     in use at exit: 40 bytes in 10 blocks
==23893==   total heap usage: 12 allocs, 2 frees, 1,104 bytes allocated
==23893==
==23893== LEAK SUMMARY:
==23893==    definitely lost: 40 bytes in 10 blocks
==23893==    indirectly lost: 0 bytes in 0 blocks
==23893==      possibly lost: 0 bytes in 0 blocks
==23893==    still reachable: 0 bytes in 0 blocks
==23893==         suppressed: 0 bytes in 0 blocks
==23893== Rerun with --leak-check=full to see details of leaked memory
==23893==
==23893== For lists of detected and suppressed errors, rerun with: -s
==23893== ERROR SUMMARY: 10 errors from 2 contexts (suppressed: 0 from 0)
```

Ερμηνεία

Το μήνυμα "Invalid write of size 8" σημαίνει το ίδιο με το αντίστοιχο μήνυμα του AddressSanitizer.

Το μήνυμα "Invalid read of size 8" σημαίνει ότι προσπαθούμε να διαβάσουμε δεδομένα σε μνήμη που δεν έχουμε δεσμεύσει και οφείλεται στο ίδιο λάθος με παραπάνω.

Ακολουθούν πληροφορίες για το κατά πόσο αποδεσμεύτηκε η δυναμικά δεσμευμένη μνήμη του προγράμματος (HEAP SUMMARY). Το μήνυμα "in use at exit: 40 bytes in 10 blocks" σημαίνει ότι στο τέλος του προγράμματος υπήρχαν 40 bytes που δεν είχαν γίνει free.

Αν ξανατρέξουμε valgrind με επιλογή --leak-check=full θα πάρουμε πληροφορίες για το πού δεσμεύθηκε η "χαμένη" μνήμη:

```
==23898== 40 bytes in 10 blocks are definitely lost in loss record 1 of 1
==23898==  at 0x483B7F3: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==23898==    by 0x1091D2: main (test.c:7)
```

Παρατηρήστε πως το valgrind αναφέρει "12 allocs, 2 frees, 1,104 bytes allocated " ενώ ο κώδικας δεσμεύει 11 αντικείμενα (1 στη γραμμή 5 και 10 στη γραμμή 7) και σίγουρα όχι 1104 bytes. Αυτό οφείλεται στο ότι το valgrind καταγράφει και ότι δεσμεύσεις/αποδεσμεύσεις γίνονται κατά την εκτέλεση συναρτήσεων βιβλιοθήκης όπως η printf.

Παράδειγμα 2 (Παράνομη προσπέλαση)

```
1    int main (int argc, char *argv[])
2    {
3        int ** p, i;
4        p = (int**) malloc(sizeof(int*)*10);
5        for (i = 0; i < 10; i++) {
6            p[i] = (int*) malloc(sizeof(int));
7        }
8        p = (int**) realloc(p, sizeof(int*)*15);
9        for (i = 0; i < 15; i++) {
10           if (p[i]) {
11               printf("%d\n", *p[i]);
12           }
13       }
14       return 0;
15   }
```

Έξοδος valgrind για το παράδειγμα 2

```
==28025== Conditional jump or move depends on uninitialised value(s)
==28025==    at 0x109218: main (test2.c:10)
==28025==
==28025== Use of uninitialised value of size 8
==28025==    at 0x48CE69B: _itoa_word (_itoa.c:179)
==28025==    by 0x48EA574: __vfprintf_internal (vfprintf-internal.c:1687)
==28025==    by 0x48D4D3E: printf (printf.c:33)
==28025==    by 0x109245: main (test2.c:11)
```

<η αναφορά στο memory leak παραλείπεται γιατί καλύπτεται από το παράδειγμα 1>

Ερμηνεία

Αναφέρει δύο σφάλματα: (α) στη γραμμή 10 (test2.c:10) ο υπολογισμός της τιμής της συνθήκης εξαρτάται από μια ποσότητα που δεν έχει αρχικοποιηθεί ("Conditional jump or move depends on uninitialised value(s)") και (β) στη γραμμή 11 γίνεται χρήση τιμής που δεν έχει αρχικοποιηθεί.

Τρόπος σκέψης για τον εντοπισμό του προβλήματος

Στη γραμμή 10 γίνεται αναφορά στο p[i]. Παρατηρούμε ότι ενώ οι πρώτες 10 θέσεις του πίνακα έχουν αρχικοποιηθεί, δηλαδή δείχνουν σε έγκυρη μνήμη (γρ. 5-7), οι επιπλέον 5 που δεσμεύονται από τη realloc δεν έχουν. Το δεύτερο σφάλμα οφείλεται στο ίδιο πρόβλημα.

Έξοδος AddressSanitizer για το παράδειγμα 2

Ο AddressSanitizer δεν ανιχνεύει το (α), αλλά αναφέρει segmentation fault που προκύπτει λόγω του(β):

```
==24007==ERROR: AddressSanitizer: SEGV on unknown address 0x000000000000 (pc 0x5555d9aee328 bp
0x7ffc84ea8120 sp 0x7ffc84ea80f0 T0)
==24007==The signal is caused by a READ memory access.
==24007==Hint: address points to the zero page.
   #0 0x5555d9aee327 in main /srv/homes/vdoufexi/test2.c:11
   #1 0x7fb134abe082 in __libc_start_main ../csu/libc-start.c:308
   #2 0x5555d9aee14d in _start (/srv/homes/vdoufexi/test2+0x114d)

AddressSanitizer can not provide additional info.
SUMMARY: AddressSanitizer: SEGV /srv/homes/vdoufexi/test2.c:11 in main
```

Το ενδιαφέρον εδώ είναι πως το μήνυμα λάθους δεν είναι ακριβές. Η διεύθυνση p[i] είναι μεν άγνωστη (unknown address) με την έννοια ότι δεν είναι έγκυρη, αλλά δεν είναι NULL (0x000000000000, zero page). Αν ήταν NULL θα ήταν ψευδής η συνθήκη της if και δε θα είχε εκτελεστεί η γραμμή 11.

Παράδειγμα 3 (heap-use-after-free)

```
1   int main (int argc, char *argv[])
2   {
3       int *pnum, *copy;
4       pnum  = (int *)malloc(1*sizeof(int));
5       *pnum = 15;
6       copy = pnum;
7       free(copy);
8       copy = NULL;
9       printf("%d", *pnum);
10      return 0;
11  }
```

Έξοδος valgrind για το παράδειγμα 3

```
==24044== Invalid read of size 4
==24044==    at 0x1091D4: main (test3.c:9)
==24044==  Address 0x4a67040 is 0 bytes inside a block of size 4 free'd
==24044==    at 0x483CA3F: free (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==24044==    by 0x1091C7: main (test3.c:7)
==24044==  Block was alloc'd at
==24044==    at 0x483B7F3: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==24044==    by 0x1091A5: main (test3.c:4)
```

Έξοδος AddressSanitizer για το παράδειγμα 3

```
==24031==ERROR: AddressSanitizer: heap-use-after-free on address 0x602000000010 at pc 0x557e59b532fe bp
0x7ffc6136e320 sp 0x7ffc6136e310
READ of size 4 at 0x602000000010 thread T0
   #0 0x557e59b532fd in main /srv/homes/vdoufexi/test3.c:9
   #1 0x7f6cf0588082 in __libc_start_main ../csu/libc-start.c:308
   #2 0x557e59b5318d in _start (/srv/homes/vdoufexi/test3+0x118d)

0x602000000010 is located 0 bytes inside of 4-byte region [0x602000000010,0x602000000014)
freed by thread T0 here:
   #0 0x7f6cf086340f in __interceptor_free ../../../../src/libsanitizer/asan/asan_malloc_linux.cc:122
   #1 0x557e59b532be in main /srv/homes/vdoufexi/test3.c:7
   #2 0x7f6cf0588082 in __libc_start_main ../csu/libc-start.c:308

previously allocated by thread T0 here:
   #0 0x7f6cf0863808 in __interceptor_malloc ../../../../src/libsanitizer/asan/asan_malloc_linux.cc:144
   #1 0x557e59b53265 in main /srv/homes/vdoufexi/test3.c:4
   #2 0x7f6cf0588082 in __libc_start_main ../csu/libc-start.c:308
```

Ερμηνεία

Στη γραμμή 9 γίνεται προσπέλαση μνήμης που ελευθερώθηκε στη γραμμή 7.

Παράδειγμα 4 (strcpy overlap)

```
1  int main (int argc, char
2  *argv[])
3  {
4      char str[6] = {"Hello"};
5      strcpy(str, &str[1]);
6      printf("%s\n", str);
7      return 0;
8  }
```

Η strcpy (και άλλες συναρτήσεις) έχει απροσδιόριστη συμπεριφορά όταν αντιγράφει μέρος ενός string πάνω στον εαυτό του. Αυτό σημαίνει πως μπορεί να λειτουργήσει σωστά, αλλά μπορεί και όχι.
Να διαβάζετε πάντα τα man pages των συναρτήσεων ώστε να βλέπετε τι περιορισμοί υπάρχουν στη χρήση τους και να **μην** εξαρτάστε από εμπειρικές παρατηρήσεις ("δούλεψε σωστά όταν το δοκίμασα στον υπολογιστή μου").

Έξοδος AddressSanitizer για το παράδειγμα 4

```
==24070==ERROR: AddressSanitizer: strcpy-param-overlap: memory ranges [0x7ffe9407a480,0x7ffe9407a485) and
[0x7ffe9407a481, 0x7ffe9407a486) overlap
   #0 0x7f2102863f01 in __interceptor_strcpy ../../../../src/libsanitizer/asan/asan_interceptors.cc:429
   #1 0x55d13299e329 in main /srv/homes/vdoufexi/test4.c:4
   #2 0x7f21025fa082 in __libc_start_main ../csu/libc-start.c:308
   #3 0x55d13299e14d in _start (/srv/homes/vdoufexi/test4+0x114d)
```

Η έξοδος του valgrind είναι αντίστοιχη, αν και πιο συνοπτική.