Compte rendu Sécurité Logicielle

DYNAMIC/STATIC ANALYSIS

Fait par :
BENARAB Hanane

1- Dangerous functions :

Le programme **getpw.c,** utilise la fonction **getpw()** qui permet de reconstruire l'enregistrement de la ligne de mot de passe de l'utilisateur uid et le place dans le tampon buf. Ce tampon contient en retour une ligne au format suivant : **name:passwd:uid:gid:gecos:dir:shell**

L'utilisation de cette fonction est dangereuse, d'où le warning du linker juste après la compilation.
Le problème vient du fait, qu'on ne prend pas en compte la taille du tampon, ce qui peut nous exposé a des buffer overflow.

En réduisant la taille du tampon, la provocation d'un **seg fault** est possible. En effet, la fonction **getpw()** peut écraser l'adresse de retour car getpw () peut écrire même après la taille du buffer.

En utilisant la commande **chfn,** l'utilisateur peut changer son nom complet ainsi que toutes les informations relatives a son compte qui se trouve dans le champ **gecos.**

Un attaquant peut exploiter cela, il suffit de connaître la taille du buffer, ainsi il peut écrire plus loin afin d'écraser l'adresse de retour et peut mettre ce qu'il veut.

L'utilisation de **mktemp** est dangereuse, parce qu'elle permet de créer un fichier dont le nom est unique. ce nom est composé dont les 6 derniers caractères sont le numéro du processus (**PID)** et une lettre. Ce qui laisse que 26 possibilités.

Un attaquant, peut facilement deviner le nom de ce fichier, il y'a également une condition de concurrence entre le test d'existence du nom et l'ouverture du fichier. Ainsi, il suffit de créer un lien symbolique vers le nom du fichier qu'il a deviné facilement au bon moment.

L'utilisateur va croire qu'il a un nom de fichier unique, mais entre-temps l'attaquant aurait déjà créé un fichier avec le même nom du fichier.

Pour éviter ce problème, il est recommandé d'utiliser **mkstemp,** qui crée et ouvre le fichier, et renvoie un descripteur de fichier ouvert pour ce fichier.

Les 6 derniers caractères de template doivent être XXXXXX, et ils seront remplacés par une chaîne rendant le nom de fichier unique avec certitude contrairement a **mktemp.**

2 – Memory leak reporting :

2-1 Definite loss :

On remarque dans le programme, **test-leak.c,** L'utilisation de deux **Malloc ()** qui permet d'allouer de la mémoire dynamiquemet à deux variables d et c ,qui n'ont pas été libéré par la suite avec **free(),** de plus la variable d n'est même pas utilisée.

En utilisant **cppcheck,** il détecte juste que la variable **'d'** n'a pas été utilisé ni libéré. Mais ne détecte pas la variable **'c'** qui n'a pas été libéré.

```
hbenarab@cardhu:~/secu/TD7/TD7$ cppcheck test-leak.c
Checking test-leak.c ...
[test-leak.c:9]: (error) Memory leak: d
```

En utilisant cette fois-ci, **valgrind** ce dernier permet de detecter les deux memory leak relatif au deux variables c et d. mais ne donne pas de précisions sur la location de ces dernieres.

```
hbenarab@cardhu:~/secu/TD7/TD7$ valgrind ./test-leak
==1077461== Memcheck, a memory error detector
==1077461== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==1077461== Using Valgrind-3.14.0 and LibVEX; rerun with -h for copyright i
==1077461== Command: ./test-leak
==1077461==
0x4aa7040
==1077461==
==1077461== HEAP SUMMARY:
==1077461==     in use at exit: 30 bytes in 2 blocks
==1077461==   total heap usage: 3 allocs, 1 frees, 1,054 bytes allocated
==1077461==
==1077461== LEAK SUMMARY:
==1077461==    definitely lost: 30 bytes in 2 blocks
==1077461==    indirectly lost: 0 bytes in 0 blocks
==1077461==      possibly lost: 0 bytes in 0 blocks
==1077461==    still reachable: 0 bytes in 0 blocks
==1077461==         suppressed: 0 bytes in 0 blocks
==1077461== Rerun with --leak-check=full to see details of leaked memory
==1077461==
==1077461== For counts of detected and suppressed errors, rerun with: -v
==1077461== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Pour plus de précisions, on utilise **« valgrind --leak-check=full ./test-leak ».**

en utilisant ceci, on a plus de détails sur les allocations de mémoire qui n'ont pas été libéré.
Il y'a également une **Backtrace**, qui permet d'afficher les fonctions qui n'ont pas libéré la mémoire après l'utilisation du Malloc (). Le coupable n'est pas forcément la fonction qui a fait le Malloc().
Dans notre cas, la fonction g() aurait dû faire un free() après le printf() et ce n'est donc pas la fonction f qui a effectué le malloc qui est la coupable.

En ce qui concerne la variable d, le coupable est bien f() car elle l'alloue localement sans donner accès au pointeur avant de retourner. En effet, a la fin de l'exécution de f(), cet espace mémoire n'est plus accessible dans le programme alors qu'il est toujours alloué dans le tas.

```
==1238331== HEAP SUMMARY:
==1238331==     in use at exit: 30 bytes in 2 blocks
==1238331==   total heap usage: 3 allocs, 1 frees, 1,054 bytes allocated
==1238331==
==1238331== 10 bytes in 1 blocks are definitely lost in loss record 1 of 2
==1238331==    at 0x483577F: malloc (vg_replace_malloc.c:299)
==1238331==    by 0x109156: f (test-leak.c:5)
==1238331==    by 0x109189: g (test-leak.c:14)
==1238331==    by 0x1091B1: h (test-leak.c:19)
==1238331==    by 0x1091BD: main (test-leak.c:23)
==1238331==
==1238331== 20 bytes in 1 blocks are definitely lost in loss record 2 of 2
==1238331==    at 0x483577F: malloc (vg_replace_malloc.c:299)
==1238331==    by 0x109164: f (test-leak.c:6)
==1238331==    by 0x109189: g (test-leak.c:14)
==1238331==    by 0x1091B1: h (test-leak.c:19)
==1238331==    by 0x1091BD: main (test-leak.c:23)
==1238331==
```

En utilisant, **ASAN,** les fuites mémoires sont également détectées et donne également une backtrace.

```
hbenarab@cardhu:~/secu/TD7/TD7$ ./test-leak.asan
0x602000000010

=================================================================
==1785832==ERROR: LeakSanitizer: detected memory leaks

Direct leak of 20 byte(s) in 1 object(s) allocated from:
    #0 0x7f18c5a1b330 in __interceptor_malloc (/lib/x86_64-linux-gnu/libasan.so.5+
30)
    #1 0x5643108501b4 in f /net/cremi/hbenarab/secu/TD7/TD7/test-leak.c:6
    #2 0x564310850241 in g /net/cremi/hbenarab/secu/TD7/TD7/test-leak.c:14
    #3 0x564310850269 in h /net/cremi/hbenarab/secu/TD7/TD7/test-leak.c:19
    #4 0x564310850275 in main /net/cremi/hbenarab/secu/TD7/TD7/test-leak.c:23
    #5 0x7f18c577509a in __libc_start_main ../csu/libc-start.c:308

Direct leak of 10 byte(s) in 1 object(s) allocated from:
    #0 0x7f18c5a1b330 in __interceptor_malloc (/lib/x86_64-linux-gnu/libasan.so.5+
30)
    #1 0x5643108501a6 in f /net/cremi/hbenarab/secu/TD7/TD7/test-leak.c:5
    #2 0x564310850241 in g /net/cremi/hbenarab/secu/TD7/TD7/test-leak.c:14
    #3 0x564310850269 in h /net/cremi/hbenarab/secu/TD7/TD7/test-leak.c:19
    #4 0x564310850275 in main /net/cremi/hbenarab/secu/TD7/TD7/test-leak.c:23
    #5 0x7f18c577509a in __libc_start_main ../csu/libc-start.c:308

SUMMARY: AddressSanitizer: 30 byte(s) leaked in 2 allocation(s).
hbenarab@cardhu:~/secu/TD7/TD7$
```

Les fuites de mémoire pose des problèmes de sécurité, car elles peuvent causer des dysfonctionnements dans le programme, ce qui peut amener a des attaques de type **DÉNI DE SERVICE.** En effet, un attaquant peut submerger un serveur de requête par exemple et si cette requête ne libère jamais l'espace qu'elle a alloué, le programme va dépasser l'espace mémoire disponible et donc terminer brutalement.

2-2 Still reachable loss :

Le programme **test-reachable-leak.c,** alloue de la mémoire dynamiquement pour le pointeur c qui est défini en dehors de toute fonction (Variable globale). Ce dernier peut étre utilisé par toutes les fonctions.

ASAN et **CPPCHECK,** ne détecte pas cela, car le pointeur reste accessible. De plus, aprés le dernier accés le programme EXIT donc on peut considérer que le free n'est pas nécessaire. (FAUX POSITIF)

Contrairement a **Valgrind,** qui detecte la fuite. En effet, valgrind pour chaque Malloc(), il cherche s'il y'a un Free () associé.

```
hbenarab@cardhu:~/secu/TD7/TD7$ valgrind --leak-check=full --show-leak-kinds=a
-reachable-leak
==2430909== Memcheck, a memory error detector
==2430909== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==2430909== Using Valgrind-3.14.0 and LibVEX; rerun with -h for copyright info
==2430909== Command: ./test-reachable-leak
==2430909==
==2430909==
==2430909== HEAP SUMMARY:
==2430909==     in use at exit: 10 bytes in 1 blocks
==2430909==   total heap usage: 1 allocs, 0 frees, 10 bytes allocated
==2430909==
==2430909== 10 bytes in 1 blocks are still reachable in loss record 1 of 1
==2430909==    at 0x483577F: malloc (vg_replace_malloc.c:299)
==2430909==    by 0x109142: main (test-reachable-leak.c:6)
==2430909==
```

3 – Unintialized Values :

Dans le programme **test-unintialized.c,** on voit que c[0] l'argument de f(), n'a pas été initialisé avant,
donc on ne peut pas garantir sa valeur.

En effet, lors de la libération de la mémoire, on nettoie pas ce qu'il y'avait précédemment dans le tas ainsi le malloc pourrait contenir des anciennes valeurs dans le tas. Par chance, sa valeur correspond a 0 mais ce n'est pas toujours le cas.

On peut utiliser **malloc _perturb** qui va initialiser toutes les valeurs allouées (Autre que par Calloc()), au complément de la valeur de l'octet le moins significatif, du paramètre passé.

```
hbenarab@cardhu:~/secu/TD7/TD7$ ./test-uninitialized
foo is 0
hbenarab@cardhu:~/secu/TD7/TD7$ export MALLOC_PERTURB_=123
hbenarab@cardhu:~/secu/TD7/TD7$ ./test-uninitialized
foo is -124
```

On voit que après l'utilisation de malloc perturb, foo n'est plus du tout à 0.

En effectuant l'analyse avec ASAN et CPPCHECK, ces derniers ne rapporte aucun bugs. Quant à **Valgrind** il reporte le problème en spécifiant où la valeur qui n'est pas initialisé, a été utilisé. Mais ne donne pas ou elle a été alloué.

« JUMP OR MOVE DEPENDS ON UNINTIALIASED VALUE », signifie que valgrind detecte bien que notre programme effectue un if en testant la valeur d'une variable non initialisée, ce qui ne fait aucun sens.

```
^C
hbenarab@hackett:~/secu/TD7/TD7$ valgrind ./test-uninitialized
==3843961== Memcheck, a memory error detector
==3843961== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==3843961== Using Valgrind-3.14.0 and LibVEX; rerun with -h for copyright info
==3843961== Command: ./test-uninitialized
==3843961==
==3843961== Conditional jump or move depends on uninitialised value(s)
==3843961==    at 0x109176: f (test-uninitialized.c:5)
==3843961==    by 0x1091C6: main (test-uninitialized.c:13)
==3843961==
foo is 0
==3843961==
==3843961== HEAP SUMMARY:
==3843961==     in use at exit: 0 bytes in 0 blocks
==3843961==   total heap usage: 2 allocs, 2 frees, 1,034 bytes allocated
==3843961==
==3843961== All heap blocks were freed -- no leaks are possible
==3843961==
==3843961== For counts of detected and suppressed errors, rerun with: -v
==3843961== Use --track-origins=yes to see where uninitialised values come from
==3843961== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

En testant maintenant avec « **--track-origins=yes** », on voit qu'il arrive à détecter l'origine du problème et où cela a été alloué.

```
==4068021== Memcheck, a memory error detector
==4068021== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==4068021== Using Valgrind-3.14.0 and LibVEX; rerun with -h for copyright info
==4068021== Command: ./test-uninitialized
==4068021==
==4068021== Conditional jump or move depends on uninitialised value(s)
==4068021==    at 0x109176: f (test-uninitialized.c:5)
==4068021==    by 0x1091C6: main (test-uninitialized.c:13)
==4068021==  Uninitialised value was created by a heap allocation
==4068021==    at 0x483577F: malloc (vg_replace_malloc.c:299)
==4068021==    by 0x1091B1: main (test-uninitialized.c:12)
==4068021==
```

"Unintialised value was created by a heap allocation." , donc il trouve bien que cette Valeur non initialisé est dû a un malloc.

Concernant le programme **test-unintialized-printf .c,** il s'agit du même programme précédent, juste sans la condition qui vérifie si foo est vide ou pas.

En l'analysant avec valgrind, ce dernier émet beaucoup de warnings sur les printf() alors que le problème vient de notre programme.

Il faut noter que derrière les comportement incertain des programmes, ça peut potentiellement être la raison d'une fuite d'informations qui est déjà présente dans le tas. Il se peut que ses informations soient sensibles, mot de passe par exemple.

4 – Undefined behavior :

Le programme **test-undefined.c,** essaie d'incrémenter un **INT_MAX.** Il s'agit d'un **INTEGER OVERFLOW** qui conduit à un comportement indéfini.
En exécutant le programme, ce dernier nous donne une valeur négative.

```
hbenarab@hackett:~/secu/TD7/TD7$ ./test-undefined
-2147483648
```

Valgrind ne détecte aucun problème, parce que il n'a pas accès au code source, donc pour lui il s'agit d'une instruction d'incrémentation, il ignore si la valeur est un integer ou unsigned.

Quant a **USAN,** il arrive bien a détecté l'integer overflow. En effet, ce dernier étant injecté par le compilateur, il a bien accès au code source, donc il connait le type de la variable d'où sa capacité à effectuer les vérifications approprié. Il nous indique que un integer ne pas contenir la valeur INT_MAX+1.

```
hbenarab@hackett:~/secu/TD7/TD7$ ./test-undefined.usan
test-undefined.c:7:3: runtime error: signed integer overflow: 2147483647 + 1 cannot be represented in
type 'int'
-2147483648
```

5 – Buffer overflow :

5-1 Heap buffer overflow :

Le programme **test-overrun,** alloue un tableau de caractéres d'une taille de 10 éléments, essaie par la suite d'écrire au 11 éme élément, ce dernier fonctionne bien par chance, parce que Malloc arrondi la taille de l'allocation ce qui ne provoque pas de seg fault.

CPPCHECK voit le problème en effet ce dernier a accès au code source donc il rencontre le problème. Valgrind le detecte aussi car il voit un accès dans le tas, juste après un espace mémoire alloué, Asan trouve aussi l'erreur.
Tant dis que USAN non.

```
hbenarab@hackett:~/secu/TD7/TD7$ ./test-overrun.usan
hbenarab@hackett:~/secu/TD7/TD7$ cppcheck test-overrun.c
Checking test-overrun.c ...
[test-overrun.c:6]: (error) Array 'c[10]' accessed at index 10, which is out of bounds.
```

```
0x60200000001a is located 0 bytes to the right of 10-byte region [0x602000000010,0x60200000001a)
allocated by thread T0 here:
    #0 0x7f0fa5fd8330 in __interceptor_malloc (/lib/x86_64-linux-gnu/libasan.so.5+0xe9330)
    #1 0x561e35bac186 in main /net/cremi/hbenarab/secu/TD7/TD7/test-overrun.c:5
    #2 0x7f0fa5d3209a in __libc_start_main ../csu/libc-start.c:308

SUMMARY: AddressSanitizer: heap-buffer-overflow /net/cremi/hbenarab/secu/TD7/TD7/test-overrun.c:6 in m
ain
```

```
hbenarab@hackett:~/secu/TD7/TD7$ valgrind ./test-overrun
==469603== Memcheck, a memory error detector
==469603== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==469603== Using Valgrind-3.14.0 and LibVEX; rerun with -h for copyright info
==469603== Command: ./test-overrun
==469603==
==469603== Invalid write of size 1
==469603==    at 0x109163: main (test-overrun.c:6)
==469603==  Address 0x4aa704a is 0 bytes after a block of size 10 alloc'd
==469603==    at 0x483577F: malloc (vg_replace_malloc.c:299)
==469603==    by 0x109156: main (test-overrun.c:5)
==469603==
==469603==
==469603== HEAP SUMMARY:
==469603==     in use at exit: 0 bytes in 0 blocks
==469603==   total heap usage: 1 allocs, 1 frees, 10 bytes allocated
==469603==
```

("0 bytes after a block of size 10 alloc'd") signifie que l'adresse à laquelle on essaie d'accéder se trouve juste après un espace alloué de taille 10.

Les 3 outils, remontent la ligne 6 comme étant un problème de HEAP Overflow.
valgrind et asan, arrive à remonter également, le fait que le malloc a une taille de 10 éléments.

5-2 Static buffer overflow :

Dans le programme **test-overrun-data.c,** on essaie d'accéder à l'élément 11 et à l'élément -1, or le tableau de caractères est défini sur 10 éléments dans le segment data.
Comme le tableau est défini dans le segment data, or on sait que ce segment est accessible en écriture et en lecture d'où l'exécution normal du programme.

Comme CPPCHECK et USAN ont accès au code source, seuls ces derniers le détecte contrairement a valgrind. ASAN est conçu pour détecter les erreurs d'accès à la mémoire liées aux allocations de mémoire heap et stack, mais il peut ne pas fonctionner comme prévu pour les allocations de mémoire statique.

```
hbenarab@hackett:~/secu/TD7/TD7$ ./test-overrun-data.usan
test-overrun-data.c:7:3: runtime error: index 10 out of bounds for type 'char [10]'
test-overrun-data.c:7:8: runtime error: store to address 0x55866baf0122 with insufficient space for an
 object of type 'char'
0x55866baf0122: note: pointer points here
 00 00  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  00 00 00 00 00 00
            ^
test-overrun-data.c:8:3: runtime error: index -1 out of bounds for type 'char [10]'
```

```
hbenarab@hackett:~/secu/TD7/TD7$ cppcheck test-overrun-data.c
Checking test-overrun-data.c ...
[test-overrun-data.c:7]: (error) Array 'c[10]' accessed at index 10, which is out of bounds.
[test-overrun-data.c:8]: (error) Array index -1 is out of bounds.
```

5-3 Stack buffer overflow :

Le code **test-overrun-stack.c** présente un dépassement de tampon, car il tente d'écrire en dehors des limites du tableau **c**, sachant que le tableau **c** est une variable locale et est donc alloué sur la pile
La ligne **c[10] = 1;** tente d'écrire dans la 11 ème position du tableau, qui n'existe pas car le tableau est déclaré avec une taille de 10.

USAN et CPPCHECK détectent le dépassement de tampon pour les mêmes raisons que mentionnées précédemment, à savoir qu'ils peuvent examiner le code source et détecter les erreurs de programmation.

ASAN, quant à lui, est spécialement conçu pour vérifier l'accessibilité des variables locales et des allocations dynamiques de mémoire, ce qui lui permet de détecter l'accès incorrect en dehors du tableau dans cet exemple.

En revanche, Valgrind ne peut pas détecter l'erreur car il n'a pas accès au code source et ne peut pas déterminer la taille des tableaux et des variables locales. Il ne dispose pas non plus de mécanismes pour suivre les limites des tampons alloués, comme le fait ASAN.

Bien qu'une marge de sécurité soit généralement utilisée pour allouer de la place sur la pile pour les variables locales, cela ne garantit pas que tous les accès mémoire seront corrects.

Dans ce cas particulier, les accès étant juste un octet avant ou après le début du tableau, ils tombent dans la marge de la pile. En effet quand on fait de la place sur la pile pour les variables locales, on le fait avec une certaine marge. Ici l’accès un octet avant ou après ne pose pas de problème du point de vue de Valgrind, il ne peut pas savoir si c’est un accès illégal ou non.