TD 7 : Partie 2

8 – ODD expressions :

Le programme **test-expr.c,** a pour but d'afficher la parité du nombre d'arguments passé au programme.
En exécutant ce dernier, nous donne toujours le même résultat, peu importe le nombre d'arguments passé en entrée, qu'il soit pair ou impair.

```
hbenarab@aragorn:~/secu/TD7/TD7$ ./test-expr 1 1 1
odd number of arguments
hbenarab@aragorn:~/secu/TD7/TD7$ ./test-expr 1 1
odd number of arguments
```

Le problème dans ce programme, vient du test **if** qui vérifie la parité de nombre d'arguments passé en entrée. En effet, l'instruction `if (argc & 1 == 0)` effectue d'abord une comparaison en testant si 1 == 0 parce que «== » est plus prioritaire que le &, ce qui nous donne False. Ensuite, il effectue une comparaison bit a bit (&) entre argc et false (interprété comme un 0), d'où le résultat qui est toujours à 0 (faux) ce qui affiche toujours "odd number of arguments " .

Pour remédier à ça, il suffit juste rajouter des parenthèses comme ci-dessous.
Ainsi on testera vraiment la parité, en comparant le dernier bit de **argc** à 1.

```c
#include <stdio.h>
int main(int argc, char *argv[]) {
    argc--; // program name is not really an argu

    // Extract bit 0 to check for parity of the r
    if ((argc & 1) == 0) {
        printf("even number of arguments\n");
    } else {
        printf("odd number of arguments\n");
    }
}
```

```
hbenarab@aragorn:~/secu/TD7/TD7$ ./test-expr 1 1
even number of arguments
hbenarab@aragorn:~/secu/TD7/TD7$ ./test-expr 1 1 1
odd number of arguments
```

8-2 –

```
<<< CID 1297531: Control flow issues DEADCODE
<<< Execution cannot reach this statement: "fsrprintf(dcgettext(NULL, "...".
```

L'erreur reportée dans ce cas, est un segment de code jamais exécuté, qui est censé effectuer une vérification sur la variable ret si elle est négative. Or, la variable ret est assigné à un booléen qui prend 1 ou 0 comme valeur et donc n'est jamais une valeur négative.

```
while ((ret = xfs_bulkstat(fsfd,
                           &lastino, GRABSZ, &buf[0], &buflenout) == 0)) {
```

Le but du développeur était juste de récupérer la valeur de retour de xfs_bulkstat. Due a une erreur de parenthésage, la valeur de ret est en effet, la comparaison du retour avec 0 parce que la comparaison est plus prioritaire que l'affectation.

L'avantage d'assigner une valeur tout en testant sa valeur de vérité, est l'optimisation du code et le rendre plus concis.

En effet, si on avait voulu récupérer la valeur retour dans ret sans tester en même temps le if, il aurait fallu une ligne de code ret =xfs_bulkstat(…) avant le while, et la même chose à la fin du while.
Mais cela, peut engendrer des erreurs de parenthésage ou de confusion, et ainsi amener un comportement non voulu au programme. Cela peut aussi ne pas être clair immédiatement pour des développeurs relisant votre code.

8-3

```
<<< CID 108554: Incorrect expression ASSERT_SIDE_EFFECT
<<< Assignment "n = (int)p[1]" has a side effect.  This code will work differentl
```

```
    assert(n = int(*(p+1)), "just checkin'..."); // send_code_addr()+1 must
```

Le bug reporté dans ce cas, vient d'un **ASSERT,** sensé effectuer un test de comparaison mais comme le montre le code ci-dessus, au lieu de mettre == (Comparaison), le développeur a mis = ce qui correspond a une assignation.

Dans ce cas-là, le ASSERT peut être dangereux parce qu'il modifie le comportement souhaité du programme. En mode **DEBUG,** Ceci correspondra a une assignation donc la

variable n prendra une nouvelle valeur alors que de base, on voulait juste effectuer une comparaison.

En mode **NON-DEBUG,** cette affectation n'aura pas lieu et donc le programme pourrait avoir un comportement différent qu'en mode **DEBUG.**
Ici, le programme fonctionne bien par chance car la valeur assignée dans le assert était la valeur que l'on souhaitait avoir pour n.

Cela aurait pu se passer autrement en effet, on aurait pu avoir le bon comportement en mode **DEBUG,** grâce à l'assignation du assert et en mode **NON-DEBUG** notre programme aurait pu avoir un autre comportement car n n'aurait pas la bonne valeur.

8-4

```
<<< CID 1308097: Incorrect expression USELESS_CALL
<<< Calling "screen->empty()" is only useful for its return value, which is ignor
```

```
69          screen.empty();
70          screen.setPalette(palette);
```

La fonction **empty(),** sert a effectué une vérification sur la taille de screen si elle est nulle. Elle retourne une valeur booléenne (True si nulle false sinon).Cette fonction ne modifie en aucun cas la valeur de la surface.

L'analyseur statique, renvoie un warning en spécifiant que le retour de la fonction empty() n'a pas été utilisé ou récupéré donc son utilisation est inutile.
Le développeur se rend donc ainsi compte qu'il voulait utiliser la fonction **clear()** et non pas la fonction **empty()**.

8-5

```
<< 18. Checking "i < 21" implies that "i" may be up to 20 on the true branch.

<< 20. Incrementing "i". The value of "i" may now be up to 21.

<<< CID 1003944: Memory - illegal accesses OVERRUN
<<< 21. Overrunning array "Tinsel::g_objArray" of 21 8-byte elements at element index 21 (byte offse
```

```
                    for (i = 0; g_objArray[i] && i < MAX_WCOMP; i++) {
                            MultiMoveRelXY(g_objArray[i], x - center, deltay);
```

Dans ce dernier cas, le problème est un accès en mémoire illégale. En effet, dans le corps de la boucle la variable i va bien de 0 à 20, mais lors de la dernière comparaison pour savoir si on continue de boucler, on effectue un accès en mémoire illégale parce qu'on essaie d'accéder à **g_objArray[i]** avec i qui vaut 21. Comme ce dernier, ne contient que 21 éléments, avec i qui vaut 21 on essaie d'accéder au 22 éme élément. Cela, peut causer un **SEGFAULT** et ainsi le crash du programme.

Ceci, se produit rarement parce qu'il faut effectuer un accès mémoire dans une zone illégale dans la pile ou dans une zone allouée par un malloc() mais cela reste pénible car peut causer des crash occasionnel et donc difficilement corrigeable.

9 – Control flow :

9-1

```
167         if (iface)
168         {
169             rawProcess   = iface->createRawProcessor();
170             bool decoded = rawProcess->decodeRawImage(inUrl, imageData, widt
171         }
172     }
```

```
<< The condition "decoded" cannot be true.
```

```
174     if (decoded)
175     {
176         uchar* sptr  = (uchar*)imageData.data();
177         float factor = 65535.0 / rgbmax;
```

Le bug reporté, est un DEADCODE parce qu'on initialise une variable **decoded** à false, ensuite on la redéfinie localement en lui assignant une nouvelle valeur, donc sa portée sera juste à l'intérieur du scope (if) ou elle a été définie.

Comme elle garde toujours sa valeur à false, lors du test (if decoded) , on exécutera donc jamais le code à l'intérieur du if. Et donc l'analyseur considère cela comme du DEADCODE.

9-2

```
            skip = val - 0x60;
            continue;
        }
```

Au début, la valeur de **SKIP** était à 1, on modifie sa valeur et on tombe juste après sur une instruction continue qui fait réitérer la boucle for, or la première instruction dans le corps de la boucle est d'affecter la valeur 1 à SKIP. Donc on écrase la valeur précédente. La ligne est donc inutile, c'est aussi du DEADCODE

9-3

```
<<< CID 1354182: Control flow issues MISSING_BREAK
<<< The case for value "1" is not terminated by a 'break' statement.
```

```
case SSL_ERROR_SSL:
        {
                unsigned long ssl_err_tmp = ERR_peek_error();
                FT_ERROR_OPENSSL_P("Unexpected error %d  during SSL shutdown, clo
                assert(errno_ssl_cmd == 0);
                _ft_stream_error(this, 0, ssl_err_tmp, "SSL shutdown (unknown)");
        }
        FT_TRACE(FT_TRACE_ID_STREAM, "END " TRACE_FMT " unknown", TRACE_ARGS);
```

Le bug dans ce programme-là, est l'oublie d'un return ou d'un break dans le cas d'un case. En effet, lors d'un test si on rentre dans ce cas d'erreur, ce dernier va exécuter ses instructions et en plus de cela, il exécutera également le cas d'erreur par **defaut** vu que ce dernier ne break pas et ne retourne rien.

Ici cela ne cause pas « trop » de dégats puisque la gestion de l'erreur default et de SSL_ERROR_SSL, semble similaire. Mais on peut imaginer des cas ou la gestion d'un default et d'une autre erreur soit complètement différente, et l'oublie d'un break ou d'un return causerait de gros problèmes.

9-4

```
        struct lov_oinfo      **loi;
        unsigned int i;
```

```
        while (--i >= 0)
                kmem_cache_free(lov_oinfo_slab, lsm->lsm_oinfo[i]);
        kvfree(lsm);
```

On essaie d'effectuer une comparaison pour savoir si i est plus grand ou égal à 0 or i est un unsigned int, donc sa valeur est toujours supérieure ou égale à 0 donc la condition dans le while est toujours true. (Le compilateur nous le dit d'ailleurs lors de la compilation)

Lorsqu'on effectue la décrémentation d'un unsigned int qui a pour valeur 0, celui-ci vaudra UINT_MAX, en effet il ne peut pas être négatif on raisonne donc modulo UINT_MAX+1.
Le problème ici est donc que le while va boucler et en plus on va faire des accès mémoire illégaux sur lsm→lsm_oinfo[i], car on sera en array overflow. De plus si le kernel ne se coupe pas on va passer une mauvaise adresse à kmem_cache_free.
Le programme aura vraiment un comportement chaotique.

9-5

```
                /* Detect PCI clock by looking at cmd_high_time. */
                switch((itr1 >> 8) & 0x07) {
```

```
ID 741147: Control flow issues DEADCODE
xecution cannot reach this statement: "case 9U:".
```

```
                        case 0x09:
                                pci_clk = 40;
                                break;
                        case 0x05:
                                pci_clk = 25;
                                break;
                        case 0x07:
                        default:
                                pci_clk = 33;
                                break;
                }
        }
```

L'expression « (itr1 >> 8) » décale les bits de itr1 de 8 positions vers la droite, cela revient à éliminer les 8 bits les moins significatifs, on compare après bit à bit avec 0x07. Donc toutes les valeurs possibles sont comprises entre 0x00 et 0x07. Ainsi le cas 0x09 ne peut jamais se produire et la section de code n'est pas atteignable c'est du DEADCODE. L'impact est que le CPI clock speed n'atteindra jamais sa valeur maximale.

10– Damn Pointers :

10-1

```
<< 4. Storage is returned from allocation function "operator new[]".

<< 5. Assigning: "buffer" = storage returned from "new char[mn1]".

111             char* buffer = new char[mn1];

<< 6. Resource "buffer" is not freed or pointed-to in "memcpy".

112             memcpy(buffer, p1Int8->get(), mn1 * sizeof(char));
113             p1Copy = new int[mn1];

<< 7. Resource "buffer" is not freed or pointed-to in "memcpy".

114             memcpy(p1Copy, buffer, mn1 * sizeof(int));

< 8. Breaking from switch

<<< CID 1321065: Resource leaks RESOURCE_LEAK
<<< 9. Variable "buffer" going out of scope leaks the storage it points to.
```

Buffer est alloué dynamiquement grâce à l'opérateur new. On l'utilise localement mais on n'utilise jamais delete[] buffer. Et on a aucune variable globale contenant son pointeur, on ne pourra donc jamais le libérer, car son pointeur restera local et disparaîtra. Ainsi l'analyseur statique est sûr que c'est une fuite mémoire

10-2

```
< 1. Condition "this->texture == NULL", taking false branch.

85              if (texture == nullptr) {
86                      return;
87              }

< 2. Condition "pos == NULL", taking true branch.

89              if (pos == nullptr) {
90                      SDL_Rect box = { x, y, w, h };

<< 3. Assigning: "pos" = "&box" (address of local variable "box").

91                      pos = &box;

<< 4. Variable "box" goes out of scope.

92              }

<<< CID 1375004: Memory - illegal accesses RETURN_LOCAL
<<< 5. Using "pos", which points to an out-of-scope variable "box".

94              SDL_RenderCopy(renderer, texture, clip, pos);
95      }
```

Dans ce programme, si pos est nullptr, on définit localement box, et on passe son adresse comme valeur de pos.

Mais on passe son adresse qui est LOCALE. Ainsi, ligne 94 on utilise pos qui pointe vers la variable box mais qui n'est pas accessible dans notre portée actuelle, c'est donc un accès mémoire illégale.

10-3

```
148         this.parent = parent;
< 1. Condition "parent != null", taking false branch
<< 2. Comparing "parent" to null implies that "parent" might be null.
149         if (parent != null) {
150             parent.addChild(this);
151         }
<<< CID 33615: Null pointer dereferences FORWARD_NULL
<<< 3. Calling a method on null object "parent".
152         getContext().setParent(parent.getContext());
153     }
154
```

Ligne 149, on teste si la variable parent est null ou non, mais ligne 152 qui est hors du if, on appelle la méthode getContext() sur parent, mais ici on peut se retrouver avec parent qui est null et donc appeler une méthode sur un objet null ce qui est contradictoire.
La ligne 152 aurait dû être dans le if testant si parent est null ou non.

10-4

```
1711        while ((*lastpos != 0) && ((pos = strchr(lastpos, '&')) != NULL)) {
1712                /* Store current string */
1713                strncat(tmp, lastpos, pos - lastpos);
1714                lastpos = pos;
1715                /* Skip ampersand */
1716                pos++;
1717                /* Detect end of string */
<< At condition "pos == NULL", the value of "pos" cannot be "NULL".
1718                if (pos == 0) break;
1719                /* Find entity length */
1720                pos_end = strchr(pos, ';');
1721                if (pos_end - pos > 6 || pos_end == NULL) {
```

A la ligne 1711, on affecte avec strchr(lastpos, '&'), la fonction strchr renvoie un pointeur vers la première occurrence du caractère & dans lastpos ou null si il n'apparaît pas, or on teste dans le while pour que pos soit différent de NULL.

On est donc sûr que pos contient une adresse, ainsi ligne 1718 c'est impossible que pos soit NULL. On voulait en fait regarder si la valeur pointée par pos est nulle, c'est pourquoi le fix de ce bug est *pos ==0 à la place de pos==0, on comparera donc bien la valeur pointée par pos à 0 et non plus pos directement.

10-5

```
251
252  static inline void
253  _drive_parse(Eldbus_Message_Iter *iter, const char *opath)
254  {

<<< CID 121492: Incorrect expression SIZEOF_MISMATCH
<<< Passing argument "8UL /* sizeof (entry) */" to function "calloc" and then casting the return value to "De

< Did you intend to use "sizeof (*entry)" instead of "sizeof (entry)"?

255      Device_Entry *entry = calloc(1, sizeof(entry));
256
257      Eina_Bool removable = EINA_TRUE;
258
259      if (!_dbus_helper_search_field(iter, "Removable", NULL, NULL, NULL, &removable))
260        ERR("Failed to fetch Removable");
261
```

Le calloc ligne 255, alloue de la mémoire dynamiquement et la taille demandé est entry . Or entry est un pointeur vers un object de type Device_Entry, sa taille est donc de 4 ou 8 octets selon l'architecture, mais ce n'est sûrement pas la taille de la structure.

Device_Entry. Le fix proposé est de remplacer sizeof(entry) par sizeof(*entry), en effet en faisant cela on va alors déréférencer le pointeur entry et obtenir la bonne taille d'une structure Device_Entry. Cependant ici, entry n'a pas été initialisé auparavant on pourrait donc vouloir déréférencer un pointeur non initialisé ce qui pourrait amener a un comportement non voulu. Je pense que la meilleure chose à faire est de remplacer par sizeof(Device_Entry).

10-6

```
55      wmodule *next_module;
< 1. Condition "cur_module", taking true branch
< 4. Condition "cur_module", taking true branch
57      for (cur_module = wmodules; cur_module; wmodules = next_module) {
<<< CID 117766: Memory - illegal accesses USE_AFTER_FREE
<<< 5. Dereferencing freed pointer "cur_module".
58          next_module = cur_module->next;
59          cur_module->context->destroy(cur_module->data);
<< 2. "free" frees "cur_module".
60          free(cur_module);
< 3. Jumping back to the beginning of the loop
61      }
62  }
63
```

Dans ce code on souhaite libérer une sorte de liste chaînée. Or ligne 60, on free cur_module, et lors du prochain tour de boucle ligne 58 on réutilise cur_module qu'on vient de free, une fois qu'un espace mémoire est libéré rien ne garantit que son contenu va rester intact. On déférence donc un pointeur qui a été libéré.

Le fix proposé n'est pas bon en effet, on free cur module, et après dans la condition de boucle on fait cur_module=cur_module→next donc on déférence encore un pointeur qui à été free, et en plus on n'a aucune garanti du comportement.
La bonne façon de faire est de passer par une variable intermédiaire nextmodule comme ceci :

```
for (cur_module = wmodules; cur_module; cur_module = next_module)
        next_module = cur_module->next;
    cur_module->context->destroy(cur_module->data);
        free(cur_module);
        }
```

11 – ERROR Checking

11-1

```
< 1. Condition "!fromDBbackend.open(this->m_parameters)", taking false branch
54      if (!fromDBbackend.open(m_parameters))
55      {
56          return false;
57      }
< 1. Example 1: "this->checkPriv(fromDBbackend, QString const(QLatin1String("CheckPriv_CREATE_TABLE")))" has
< 2. Condition "!this->checkPriv(fromDBbackend, QString const(QLatin1String("CheckPriv_CREATE_TABLE")))", ta
59      if (!checkPriv(fromDBbackend, QLatin1String("CheckPriv_CREATE_TABLE")))
60      {
61          insufficientRights.append(QLatin1String("CREATE TABLE"));
62          result = false;
```

```
< 3. Falling through to end of if statement
63      }
< 1. Example 2: "this->checkPriv(fromDBbackend, QString const(QLatin1String("CheckPriv_ALTER_TABLE")))" has
64      else if (!checkPriv(fromDBbackend, QLatin1String("CheckPriv_ALTER_TABLE")))
65      {
66          insufficientRights.append(QLatin1String("ALTER TABLE"));
67          result = false;
68      }
< 1. Example 3: "this->checkPriv(fromDBbackend, QString const(QLatin1String("CheckPriv_CREATE_TRIGGER")))" h
69      else if (!checkPriv(fromDBbackend, QLatin1String("CheckPriv_CREATE_TRIGGER")))
70      {
71          insufficientRights.append(QLatin1String("CREATE TRIGGER"));
72          result = false;
```

L'analyseur a remarqué que le développeur a utilisé 5 fois la méthode checkPriv en utilisant la valeur retourner par cette dernière. Il indique donc qu'ici on ne l'a prend pas en compte et que cela n'est sûrement pas normal et donc que la valeur de result n'est peut-être pas celle qu'on voudrait qu'elle soit.

11-2

optBuf est une chaîne de caractère remplit par l'utilisateur grâce à un **fgets**(), c'est ce qu'on appelle un tained string, car il provient de l'extérieur et peut être « contaminé » ce qui est très dangereux par ce que la fonction **dbfcmd**, exécute cette requête dans la base de données et nous n'avons même pas vérifier son contenu avant alors qu'elle provient de l'utilisateur.
Ici il faut donc d'abord vérifier l'entrée avant de la fournir a **dbfcmd**.