Grenoble INP- Ensimag
Olivier Benjamin
Étudiant en deuxième année

2010-2011
filière MMIS MCS

Travaux d'Études et de Recherches
Test en frelatage (fuzzing) et code binaire

Laboratoire d'Informatique de Grenoble - Équipe VASCO
Encadrant : Roland Groz

Cadre du TER - découverte du laboratoire

Les Travaux d'Études et de Recherches sont un module optionnel de la deuxième année d'études à l'Ensimag. Ils consistent à réaliser un travail de recherche dans un laboratoire. J'ai effectué mon travail d'études et de recherches au sein de l'équipe VASCO (Validation de systèmes, composants et objets logiciels) du Laboratoire d'informatique de Grenoble de février à mai 2011. J'ai été encadré par un chercheur expérimenté, Roland Groz, et deux doctorants en première année de thèse : Fabien Duchène, et Karim Hossen. J'ai également eu l'occasion d'interagir avec une doctorante en deuxième année de l'équipe : Sofia Bekrar. Le thème de mon travail est inspiré d'une partie de son sujet de thèse, et permet de préparer d'éventuels approfondissements dans ce cadre. Au cours des semaines passées dans cette équipe, j'ai ainsi pu découvrir différents profils de chercheurs, et différents parcours.

Termes spécifiques

Test logiciel

Le test logiciel est la discipline qui s'attache à vérifier la conformité d'un logiciel vis-à-vis d'un cahier des charges.

Vulnérabilité

Une vulnérabilité est un défaut, une faiblesse du système considéré, qui pourrait résulter en une violation des propriétés du système.

Buffer Overflow

Le buffer overflow est une des vulnérabilités les plus connues [15] en sécurité logicielle. Il s'agit d'un comportement anormal qui permet d'écrire dans une partie de la mémoire autre que celle prévue, en débordant de cette dernière.

Métrique (de couverture)

Dans ce sujet, on s'attache à mesurer l'efficacité de tests. Une métrique est une propriété ou une méthode permettant de mesurer de façon pertinente cette efficacité selon un critère particulier.

Couverture d'instructions

La couverture d'instruction est une métrique qui utilise comme critère la proportion d'instructions exécutées. On calcule le rapport du nombre d'instructions exécutées sur le nombre d'instructions totales du programme.

Analyse de teinte

L'analyse de teinte consiste à marquer les données reçues de l'utilisateur, et de propager ensuite cette teinte aux objets qui dépendent ensuite de ces données. Cela permet notamment de garder une trace de la confiance que l'on peut accorder aux données.

I - Introduction

I-1 Analyse biographique

J'ai commencé par me documenter sur le domaine, et lire des études et publications. Le rapport [4] est une bonne introduction aux problématiques de la détection et du test de vulnérabilités logicielles. L'auteur y offre une bonne approche progressive de ce qu'est une vulnérabilité, puis de leur prévention, et enfin un bon panorama des méthodes de détection. Il m'a été utile pour me familiariser avec l'environnement du test logiciel et ses termes spécifiques. La lecture de [11] est une bonne introduction au test en frelatage. L'intérêt de [16], de [10], et de [17] tient au fait qu'ils ont tous un objectif commun, qui est similaire en partie à celui de ce TER. En effet, ils traitent tous trois de la problématique suivante : réduire le coût des tests en frelatage en analysant mieux les programmes testés. Ici, le but est d'augmenter l'efficacité des tests plutôt que d'évaluer, mais beaucoup de considérations restent pertinentes. Dans le cas de [16], la première étape consiste d'ailleurs à évaluer la pertinence d'une donnée d'entrée, en observant la façon dont elle est utilisée, ce qui est donc très proche du sujet de ce travail. Les rapports [10] et [17] introduisent de plus l'analyse de teinte, et son utilisation dans le cadre qui nous intéresse.

I-2 Cadre du sujet

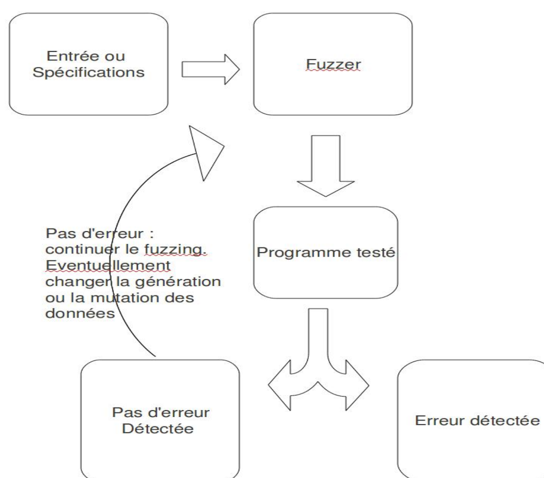
Le test logiciel

L'équipe VASCO s'intéresse à la validation de systèmes et de logiciels. L'informatique prenant toujours plus d'importance dans notre monde, les logiciels sont amenés à manipuler des données de plus en plus sensibles, et ce de plus en plus fréquemment. Le rapport du NIST (National Institute of Standards and Technology) de Mai 2002 [7] a estimé l'impact des défauts des logiciels sur l'économie des États-Unis à 59.5 milliards de dollars par an. Un des exemples les plus connus et les plus coûteux est l'erreur de programmation qui provoqua l'explosion d'Ariane 5 en 1997. Le même rapport du NIST mentionne que l'amélioration des infrastructures de test logiciel pourrait permettre de réduire le coût mentionné précédemment de 22 milliards de dollars par an. Les dommages envisagés ici ne sont que d'ordre matériel, mais il est possible que des vies dépendent du bon fonctionnement de programmes. Ceci souligne encore l'importance du test logiciel, et de son amélioration. Pour cela, une étape est donc de s'assurer qu'un logiciel se comporte conformément à ses spécifications avant sa mise en service, afin de détecter les vulnérabilités présentes, qui pourraient ensuite être exploitées pour compromettre son exécution. Ces enjeux sont connus depuis longtemps, mais le nombre toujours important de vulnérabilités découvertes dans des logiciels grand public démontre l'importance de moderniser le processus de test. Explorer toutes les possibilités demanderait trop de ressources, car elles sont en général nombreuses, et cela ne constitue donc pas une méthode de test viable en pratique. C'est la raison pour laquelle il est nécessaire de tester les programmes de manière plus intelligente et efficace.

Le test en frelatage

Depuis quelques années –le premier fuzzer date de 1988–, une nouvelle forme de test a commencé à être employée : le test en frelatage, ou 'fuzzing'. Le principe de cette méthode de test est de soumettre le programme à des données aléatoires, mal formées –volontairement ou pas–, ou proche de certaines bornes, afin de prévoir son comportement quand confronté à un attaquant. Le fuzzer le plus simple que l'on puisse imaginer, pour un programme qui prendrait en entrée une chaîne de caractères, serait un programme qui produirait la chaîne 'A', puis la chaîne 'AA', et ainsi de suite jusqu'à une taille donnée. Cependant, le frelatage peut être plus évolué, et concerner un type de protocole, ou de format de fichier. Par exemple, un test en frelatage sur la valeur de certains champs dans le format JPEG aurait pu détecter une vulnérabilité dans le traitement de ces images au sein de Windows [9]. Cette méthode de test a pour avantage d'être automatique et rapide, ce qui permet de tester un grand nombre de valeurs.

Cependant, elle présente également des inconvénients. En effet, les contraintes professionnelles en termes de temps et de coût sont importantes. Ainsi, il est attendu de toute méthode de test qu'elle soit la plus efficace possible, notamment en temps. De plus, être certain du fait que le produit remplit les critères exigés nécessite d'être capable d'évaluer ce qui a pu être testé. En effet, un échec à un test signifie toujours que le produit est invalide, mais réussir tous les tests n'est pas toujours suffisant. Cela peut indiquer que tous les critères intéressants n'ont pas été testés. On peut à ce sujet citer Edsger w. Dijkstra, qui écrivait en 1970 : “le test ne peut montrer que la présence d'erreurs, jamais l'absence”. [6]



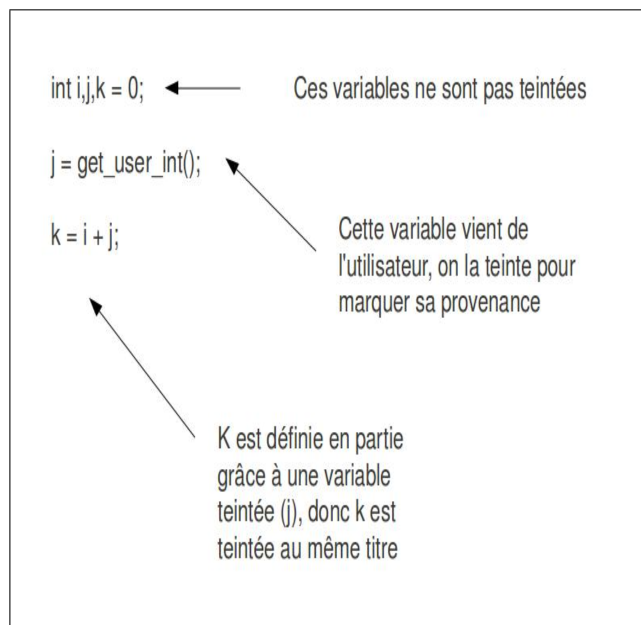
Il est donc nécessaire de trouver un moyen d'évaluer la qualité des tests, par exemple pour modifier la façon dont on génère les données de test. La métrique généralement adoptée pour ce faire est la couverture d'instructions : on exécute le programme, et on examine ensuite la quantité d'instructions qui a été exécutée par rapport à la quantité d'instructions totale du programme. Le sujet sur lequel j'ai travaillé s'inscrit dans le cadre du test de programmes binaires compatibles x86. On se restreint donc au cas où le code source n'est pas disponible, et on ne dispose que de l'exécutable. Ce cas correspond au cas le plus fréquent, puisque les logiciels propriétaires ne dévoilent pas leur code source. Des outils tels que gcov [14] permettent de le faire à condition d'avoir utilisé une option spéciale à la compilation. Dans notre cas, le fait que l'on travaille sur du code binaire ne nous permet pas d'utiliser cet outil. D'autres permettent d'utiliser ce genre de techniques sur des programmes binaires, sans avoir à les recompiler. On peut par exemple penser à Dytan, qui faisait partie des solutions auxquelles Sofia Bekrar avait réfléchi. Cependant, divers programmes tels que IDA pro couplé à des scripts, par exemple, permettent aussi d'obtenir des traces plus précises lors de l'exécution, notamment des traces des instructions exécutées.

I-3 Pistes abordées, réflexion

Le but de ce travail est d'aboutir à une façon de qualifier la réussite d'un test de manière automatique d'une façon plus intéressante et précise que la couverture d'instructions. Le défaut de cette méthode est qu'elle ne différencie pas les instructions du programme alors que certaines instructions méritent plus d'attention que d'autres dans un audit de sécurité logicielle. En effet, pour prendre un exemple, une instruction de manipulation de la mémoire, telle que la fonction strcpy de la libc, qui est connue pour avoir entraîné beaucoup de vulnérabilités de type buffer overflow, est plus importante à tester qu'une instruction affichant une chaîne de caractères. Comme le but est de qualifier la réussite du test, évaluer à quel point on a testé et stressé les parties importantes du code est la façon la plus intéressante d'aborder le problème. Plusieurs critères interviennent dans ce qui rend une portion du programme plus intéressante qu'une autre.

Données utilisateur - analyse de teinte

On s'est déjà assuré au cours de tests précédents que le fonctionnement du programme prévu par le développeur était opérationnel, et on veut maintenant vérifier qu'il n'est pas possible de tirer partie d'un fonctionnement imprévu. Ainsi, on s'intéresse à une réponse du programme à des entrées, et les parties du logiciel qui traitent des données utilisateurs méritent donc une attention particulière, puisqu'elles manipulent des éléments sur lesquels un attaquant a de l'influence. Cet aspect est étudié depuis quelques années par la communauté du test logiciel, et certains outils de frelatage commencent à incorporer ce type de détections. Une solution communément adoptée est une technique d'analyse de teinte, sur laquelle travaillent notamment des chercheurs du laboratoire Verimag, et qui consiste à marquer les données manipulées par l'utilisateur, et de suivre la façon dont elles sont utilisées lors de l'exécution du programmme. Cela permet de ne s'intéresser qu'aux données qui sont suceptibles d'être malfaisantes, car provenant de l'utilisateur. En pratique, l'analyse de teinte peut être réalisée par exemple en ajoutant un bit d'intégrité à chaque mot de 32 bits en mémoire. [5] Ce bit d'intégrité peut être positionné soit à "haute", soit à "basse", et les interactions entre objets sont modifiées par ces bits. Ainsi, un objet ne pourra être modifié que par des ceux ayant une intégrité au moins égale à la sienne, et en lire un fait baisser l'intégrité du lecteur à la sienne. On garde ainsi une trace de la confiance qui lui est accordée.



Cette approche avait été envisagée par Sofia Bekrar au cours de sa thèse, et est prometteuse, car elle se rapproche beaucoup de l'origine des vulnérabilités. Des vunérabilités détectées par cette méthode sont importantes, car comme elles sont dépendantes de données fournies par l'utilisateur, elles sont effectivement exploitable par un attaquant.

L'inconvénient dans ce cas est qu'elle était trop longue à mettre en place pour convenir au cadre du TER.

Modifications importantes de la mémoire

On s'intéresse ensuite à la taille et au type de mémoire manipulée par le programme. En effet, il semble à première vue intuitif qu'une modification majeure du fonctionnement du programme nécessite de toucher à une importante plage mémoire.

Cette métrique a l'avantage d'être relativement simple à mettre en oeuvre, puisqu'il suffit d'examiner les différences entre l'état de la mémoire avant l'exécution de l'instruction, et l'état après son exécution. Cependant, il apparaît rapidement que dans beaucoup d'exploitations de vulnérabilités, notamment du type buffer overflow, il est possible de n'avoir besoin de modifier qu'une taille mémoire correspondant à un pointeur. Cette approche ne pouvait donc pas mener à une bonne mesure de l'efficacité des tests.

Nombre moyen d'exécutions

L'idée suivante part d'un constat simple : un programme informatique ne se comporte pas toujours de la même façon à chaque exécution, mais des parties de ce programme sont toujours exécutées, et certaines sont exécutées plus souvent que d'autres. Les parties qui sont systématiquement exécutées sont à première vue plus exposées, et il semble donc plus important de les tester.

Il s'est cependant avéré que déterminer finement les chemins empruntés inconditionnellement est encore une fois trop long pour être abordé dans le cadre du TER.

Définition de patrons de vulnérabilités

La découverte de vulnérabilités peut également passer par la recherche dans le code assembleur, que des outils permettent d'obtenir à partir du binaire, d'instructions caractéristiques. Cette approche conduit à définir un certain nombre de patrons associant à des vulnérabilités connues, des types de séquences d'instructions. L'étape suivante consiste à repérer dans le code du programme des séquences correspondant à ces patrons, et à vérifier pendant les tests que l'on a particulièrement stressé ces endroits précis, qui représentent les éléments les plus suceptibles d'être vulnérables. C'est cette approche qui a suscité le plus d'intérêt, car elle était la plus proche du besoin réel en pratique.

Cependant, la complexité des programmes et de leurs erreurs ou défauts rend difficile la définition de patrons détaillés et précis. Cet exercice demande de plus beaucoup d'expérience et d'expertise pour aboutir à des patrons proches de la réalité, et une manière de simplifier cette idée a donc été envisagée. En effet, un certain nombre de vulnérabilités consistent en un appel de fonction employé à mauvais escient, ou avec de mauvais arguments. Un patron extrêmement simplifié de ces vulnérabilités est donc simplement l'instruction d'appel de ces fonctions. Ce dernier est à l'inverse trop simple, car il ne différence pas les utilisations légitimes et frauduleuses de ces fonctions, et n'apporte rien à l'état des connaissances dans le domaine. Suite à une discussion d'équipe, il fut donc suggéré qu'une approche intermédiaire serait probablement intéressante.

Il s'agit donc de tracer non plus un appel de fonctions appartenant à un ensemble de fonctions données susceptibles d'être utilisées dangereusement, mais plutôt de repérer dans le programme des enchaînements d'instructions qui pourraient conduire à une vulnérabilité. Le principe de l'analyse est donc dans un premier temps de définir une base de chaînes d'instructions susceptibles d'être utilisées par un attaquant pour détourner le programme

de son utilisation. Ceci doit être fait une seule fois, et, dans un cadre plus général, cette base devrait être maintenue et mise à jour au fur et à mesure des progrès faits dans la découverte de vulnérabilités. Ensuite, il convient d'examiner le programme afin de détecter les instructions présentes qui pourraient être utilisées pour aboutir à une séquence présente dans la base. Ceci doit être fait pour chaque programme. On doit ensuite répertorier les chaînes d'instructions qu'il est à priori possible de construire pour ce programme, et leur attribuer une valeur qui permettra à la fin des tests de calculer leur efficacité. Viennent ensuite les tests, pendant lesquels il faut détecter quelles chaînes ont été exécutées, et dans quelle proportion des tests. La dernière étape consiste à évaluer la qualité des tests en fonction des enchaînements qui ont été atteints, et donc testés, ainsi que des proportions des tests qui ont déclenché au moins un de ces enchaînements.

II - Expérimentations

Une partie importante du travail de TER a consisté à mettre en pratique les solutions théoriques discutées. Pour cela, une première étape était de trouver les outils adéquats permettant de telles expérimentations. Dans un travail tel que celui-ci, cette étape est particulièrement importante, car le temps restreint dont on dispose nécessite de pouvoir travailler rapidement, et cela signifie souvent de travailler avec les outils qui simplifient cette tâche. Dans le cas présent, il était nécessaire d'utiliser un désassembleur, avec des capacités de débuggeur.

Outils utilisés

PaiMei

PaiMei[1][2] est un environnement de rétro-ingénierie pour Windows 32 bits écrit en Python par Pedram Amini. Il a été présenté à la conférence sur la sécurité informatique REcon en 2006. Le but de cet outil est de centraliser des tâches courantes en rétro-ingénierie au sein d'un unique outil. Il est composé de plusieurs sous programmes qui remplissent des tâches spécifiques. Cet outil a l'avantage de présenter une interface graphique simple, et permet de représenter des binaires sous la forme de graphes d'appels très clairs. Il premet de faire de même avec les fonctions, en graphes de contrôle de flux. PaiMei présente cependant de nombreux problèmes. En effet, cet outil a été écrit par un seul développeur, avec des contributions extérieures, mais dans le cadre d'un projet open source. Il ne bénéficie pas d'une infrastructure ou d'une maintenance comparable à des outils propriétaires. Il est en conséquence assez difficile à installer, et à faire fonctionner. Il demande également beaucoup d'installations préalables. En outre, assez peu de documentation est facilement disponible. Je l'ai personnellement trouvé difficile à prendre en main.

PyEmu

PyEmu[12][13] est un émulateur pour l'architecture Intel 32 bits écrit en Python lui aussi, par Cody Pierce. Il a été présenté à la conférence de sécurité Black Hat à Vegas en 2007. Un important avantage de PyEmu, est qu'il permet à l'utilisateur de choisir à quels moments de son éxecution le programme émulé va appelé le script de l'utilisateur. Il permet également de manipuler et modifier la mémoire utilisée par le processus, et de suivre l'évolution de la mémoire, les accès, et les écritures. Cependant, PyEmu étant lui aussi un projet open source, il souffre des mêmes désavantages que PaiMei, bien que moins difficile à installer. De plus, son but et ses objectifs sont différents de ceux de ce TER, et plus éloignés que ceux de PaiMei de ce qui nous intéresse.

IDA pro

IDA pro[8] est un tel outil très utilisé dans le domaine du test logiciel, et l'outil avec lequel j'ai travaillé. Il est développé par la société Hex-Rays, basée en Belgique. Il présente lui aussi une interface graphique, il est plus stable que les deux outils mentionnés précédemment. Étant très utilisé, IDA pro est bien documenté, et il est facile à prendre en main. Il bénéficie également d'une maintenance assurée par la société qui le développe. Il a aussi l'avantage de posséder des plugins, dont l'un en particulier nous a intéressé au cours de ce travail. Le plugin en question s'appelle IDAPython [3]. Il s'agit d'un plugin open

source pour IDA Pro développé par Gergely Erdelyi, Elias Bachaalany, et Ero Carrera. Son but est d'intégrer le langage python dans IDA Pro afin de permettre l'écriture de scripts dans ce langage, et leur éxécution. Il est bien documenté, et son utilisation avec IDA Pro fait que plusieurs exemples de scripts sont disponibles sur internet. Il fournit des interfaces et des encapsulations des fonctions fournies par IDA. A l'aide des références disponibles, ces deux outils sont nettement plus rapides à utiliser que les deux susmentionnés. Ces deux programmes permettent tous les deux d'utiliser IDA et IDAPython, mais sont plus difficiles à utiliser, et moins adaptés. C'est cela qui a motivé mon choix de travailler avec IDA et son plugin.

Expériences

On attend du programme qu'il soit capable d'évaluer la pertinence des tests effectués vis à vis des vulnérabilités potentielles du programme. Plus précisément, dans ce cas, on veut dans un premier temps qu'il puisse déterminer si une fonction répertoriée aurait pu être appelée et ne l'a pas été, puis qu'il puisse déterminer si un tel appel est précédé ou non d'une fonction recherchée.

J'ai dans un premier temps mis en place la version naïve de l'approche des patrons de vulnérabilité discutée précédemment, qui consistait à surveiller les appels à des fonctions jugées dangereuses. Pour tester l'efficacité de cette approche, je l'ai testée sur un exemple simple. Le but de cette expérience, est d'arriver à détecter que la dernière branche conditionnelle est la plus intéressante en termes de vulnérabilités.

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

int main(int argc, char* argv[]){

        if(!strcmp(argv[1], "1")){
                printf("le premier argument vaut 1\n");
        }else if (!strcmp(argv[1],"2")){
                printf("le premier argument vaut 2\n");
        }else if (!strcmp(argv[1],"3")){
                printf("le premier argument vaut 3\n");
        }else if (!strcmp(argv[1],"4")){
                char *p = readline(">>>");
                char buffer[20];
                strcpy(buffer, p);
        }
}
```

Programme d'exemple pour tester l'approche naïve

Dans ce programme, on constate rapidement que seule la dernière séquence d'instructions, déclenchée par un premier argument valant 4, est suceptible d'entraîner une vulnérabilité, ici de type buffer overflow. C'est donc cette séquence d'instructions qui est la plus intéressante à tester. Supposons qu'une plateforme de frelatage fournisse les entrées

1, 2, puis 3 à ce programme. On n'a alors pas déclenché, ni testé la séquence vulnérable, mais une couverture d'instructions indiquerait que le programme a été testé à plus de 75%.

Dans le cas où cette fois 4 fait partie des entrées de test, l'approche mise en place permet de mettre en lumière qu'un dangereux appel à strcpy(), qui est une fonction répertoriée comme dangereuse, n'a pas été testé. Il serait possible d'essayer de donner une évaluation de la couverture des tests, qui serait cette fois très mauvaise.

```
idc.LoadDebugger("win32", 0)
idc.StartDebugger("C:\Documents and Settings\\bom\My
Documents\\testinter4.exe", "4", "C:")
idc.GetDebuggerEvent(WFNE_SUSP, -1)

code = idc.GetEventId()
while (code==16):


  print idc.GetFunctionName(idc.GetEventEa())

  for func in l:
   print("on fait le tour des fonctions")
   if(func['addr']== idc.GetEventEa()):
     func['appel']=func['appel']+1
for antec in func['antecedents']:
  if(antec['appel']==1):
     func['antecappel'].append(antec['ant'])

  for li in liste:
   print("on regarde les antecedents")
   li['appel'] = 0
   if(idc.GetEventEa()in li['addr']):
print("c'est un antecedent")
li['appel'] = 1

  Message("stopped at event "+str(idc.GetEventEa())+", event code is
"+str(idc.GetEventId())+", function is
"+str(idc.GetFunctionName(idc.GetEventEa()))+"\n")
  continue_process()
  idc.GetDebuggerEvent(WFNE_SUSP, -1)
  code = idc.GetEventId()

continue_process()
```

Partie du script pour IDA en python utilisé

Dans un deuxième temps, on veut cette fois ci tracer les instructions exécutées avant l'appel aux fonctions dangereuses. Pour tester, je me suis concentré sur un exemple simple, où un cherche à detecter les appels éventuels à la fonction strlen de la libc avant les appels à strcpy. Dans le cas du programme suivant, il faut être capable de détecter que la quatrième branche est intéressante, mais moins que la cinquième, puisque toutes deux font appel à strcpy, mais la quatrième peut présenter une tentative de contrôle de cet appel.

On a exécuté le programme avec pour entrée "4", ce qui provoque un appel à strlen, puis un à strcpy. On pourrait voir sur la capture d'écran que le script affiche dans IDA les résultats suivants :

```
strlen est present
['strlen']
la fonction strlen a bien ete appelee avant strcpy

On a manque un appel dangereux a strcpy a l adresse 4199677
strlen est present
[]
la fonction strlen n'a pas ete appelee avant strcpy
```

On peut donc constater que le programme a détecté l'appel à strlen suivi de strcpy, mais il a aussi détecté l'appel potentiel à strcpy (avec une entrée "5") qui n'a pas été appelé. Dans ce cas précis, l'appel à strlen montre une volonté de tester pour éviter la vulnérabilité de type buffer overflow. Ainsi, il faudrait donner une importance plus grande à l'appel non testé, car on ne sait pas encore s'il est ou non précédé d'une instruction recherchée, et une moins grande à celui déclenché, car il présente moins de chances d'être exploité, puisque le développeur a essayé de se prémunir contre cette possibilité. On pourrait également faire le test avec une fonction qui prendrait des données de l'utilisteur, telle que par exemple la fonction readline.

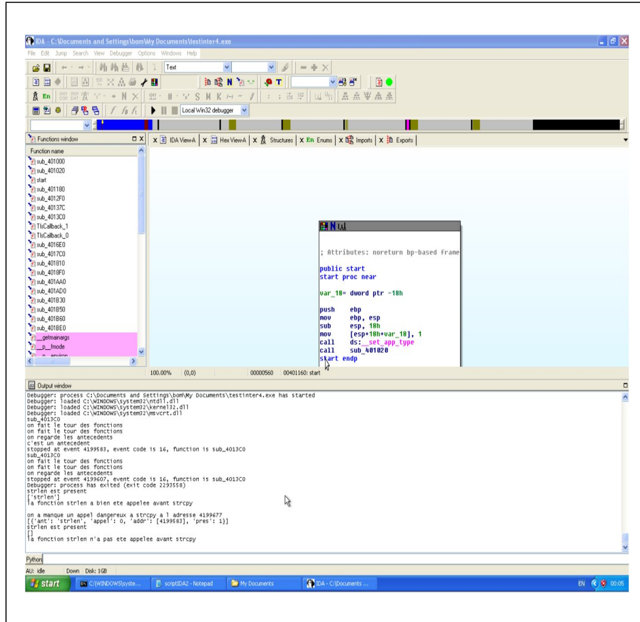
```
#include <stdio.h>
#include <stdlib.h>
#include <readline/readline.h>

int main(int argc, char* argv[]){

        if(!strcmp(argv[1], "1")){
                printf("le premier argument vaut 1\n");
        }else if (!strcmp(argv[1],"2")){
                printf("le premier argument vaut 2\n");
        }else if (!strcmp(argv[1],"3")){
                printf("le premier argument vaut 3\n");
        }else if (!strcmp(argv[1],"4")){
                char *p = readline(">>>");
                char buffer[20];
                strcpy(buffer, p);
        }else if (!strcmp(argv[1],"4")){
                char *p = readline(">>>");
                char buffer[20];
                if(strlen(p)<19){
                        strcpy(buffer, p);
                }
        }else if (!strcmp(argv[1],"5")){
                char *p = readline(">>>");
                char buffer[20];
                strcpy(buffer, p);
        }
}
```

On teste maintenant les chaines d'instructions

Dans le cas d'un appel à readline avant strcpy, il faudrait au contraire augmenter l'importance de tester cette éxécution, car l'utilisateur serait alors peut être à même de manipuler les données fournies, et donc d'exploiter la vulnérabilité. Le programme remplit bien ses objectifs, et on constate que les résultats obtenus auraient de la valeur pour déterminer la pertinence de tests, puisqu'ils sont cohérents avec une appréciation humaine des vulnérabilités.



Affichage du résultat dans IDA Pro

III Conclusion

Réponse au problème

J'ai montré dans ce rapport, qu'il est possible d'envisager différentes façon de mesurer l'efficacité du test en frelatage que la méthode de couverture d'instructions qui est en général utilisée. J'ai également pu obtenir des résultats concrets sur des programmes simples destinés et écrits dans ce but. Dans chaque cas, on a pu constater que les résultats obtenus étaient cohérents avec une analyse de vulnérabilité menée par un humain. On peut cependant regretter de n'avoir pas pu tester ces méthodes sur des programmes plus importants.

Perspectives

Un grand avantage du test en frelatage est son caractère automatique, et la génération de cas de tests selon des règles définies. Cependant, dans le sujet présenté, il reste nécessaire qu'un humain modifie les tests de façon à ce qu'ils prennent en compte l'évaluation qui est calculée. Cette étape est fastidieuse, et ne présente à priori aucune difficulté pouvant empêcher son automatisation. Il serait donc intéressant de mettre en place une rétroaction de cette évaluation sur l'environnement de frelatage afin qu'il modifie lui-même automatiquement la méthode de génération des cas de tests afin d'arriver au-dessus d'une exigence donnée.

Il serait également intéressant d'approfondir la définition des patrons de vulnérabilités qui avaient été envisagés, et dont la conception a dûe être simplifiée dans le cadre de ce travail.

Bilan personnel

J'ai choisi de participer à un TER dans le but de me familiariser avec le monde de la recherche, et de découvrir le domaine de la sécurité logicielle auquel je m'intéresse. Cette expérience m'a permis d'associer une réalité aux concepts qui m'étaient connus, et de découvrir des aspects parfois inattendus de la vie de chercheur. Cela a également été l'occasion pour moi d'apprendre beaucoup et rapidement, tant sur la partie technique que sur la simple connaissance du fonctionnement d'un laboratoire. J'ai maintenant une meilleure connaissance des enjeux quotidiens d'un travail en laboratoire. J'ai également pu, au cours de ces semaines, assister à plusieurs réunions d'équipe portant sur différents aspects des travaux en laboratoire, dont l'écriture et la correction d'un article scientifique destiné à la publication. Une partie du travail à également consisté en la lecture de différents articles scientifiques et techniques en général courts et denses, qui est une expérience différente de la lecture de livres ou de manuels de plusieurs centaines de pages qui peuvent être utiles dans mes études. Ce travail m'a ainsi donné l'occasion de m'intéresser et de m'initier à un domaine technique que je souhaitais découvrir : la sécurité et la sûreté des systèmes à travers la lecture d'articles, et de discussions avec des professionnels du domaine. J'ai été surpris par le quotidien de la vie de chercheur, qui consiste en un apprentissage différent de l'apprentissage scolaire. En effet, une grande partie du travail de chercheur consiste à lire des articles scientifiques et techniques, et assister à des séminaires sur leur sujet de recherche. Mon sentiment sur le monde de la recherche est qu'il s'agit d'un environnement très stimulant intellectuellement, de façon plus pointue que l'environnement universitaire, et probablement plus que le monde de l'entreprise en général. Il est facile de confronter

ses idées, et d'apprendre directement de ses collègues, dans mon cas beaucoup des deux doctorants qui m'on encadré. A ce titre, cet environnement m'a eu l'air très porté sur le partage des connaissances. Une partie importante de ce que m'a apporté le TER, est que j'ai été confronté à un encadrement et à une façon de travailler différents de ceux auxquels on peut être habitué en tant qu'étudiant. Il est difficile de passer de l'encadrement très guidé auquel on est confronté en tant qu'étudiant à celui plus flottant de la recherche. Bien que mes encadrants m'aient accompagnés et se soient rendus très disponibles tout au long de ce travail, l'autonomie demandée est beaucoup plus importante que celle qui est attendue d'un étudiant au quotidien. Je connaissais mal ce domaine, et il m'a été difficile de trouver facilement les informations pertinentes, et de bien visualiser dans quelle direction je devais m'orienter. En conclusion, ce TER a été l'occasion pour moi de toucher à la recherche, et mon regret principal est de n'avoir pas eu les connaissances et le temps pour que cette expérience soit encore plus profitable. Cela m'a donné l'envie de me confronter à la recherche de manière plus sérieuse, dans le cadre d'un master en troisième année, ou d'un stage ou projet de fin d'études.

Remerciements

Je tiens à remercier mon encadrant, Roland Groz, d'avoir bien voulu me proposer un sujet dans un domaine qui m'intriguait et m'intéressait. Je veux aussi le remercier au même titre que ses deux doctorants Karim Hossen et Fabien Duchène de s'être toujours mis à ma disposition, ainsi que pour leur patience et leurs corrections. Merci également à Jean-Luc Richier pour ses contributions, lors des réunions d'équipes et lors de l'étape de correction. Ma gratitue va également à Sofia Bekrar, d'avoir pris du temps pour me conseiller et m'aiguiller. Enfin, j'aimerais remercier Florence Maraninchi et la direction des études de l'Ensimag de proposer en deuxième année une option permettant de découvrir la recherche, avant de se diriger ou non vers un master recherche.

Bibliographie

[1] Pedram Amini. Paimei - reverse engineering framework. http://code.google.com/p/paimei/.

[2] Pedram Amini. Paimei - reverse engineering framework. 2006.

[3] G. Erdelyi ; E. Bachaalany ; E. Carrera. idapython - python plugin for interactive disassembler pro. http ://code.google.com/p/idapython/.

[4] W. Jimenez ; A. Mammar ; A. Cavalli. Software vulnerabilities, prevention and detection methods : A review. Technical report, Telecom SudParis, 2009.

[5] Crandall ; Chong. Minos : Control data attack prevention orthogonal to memory model. Technical report, University of California at Davis, 2004.

[6] Edsger W. Dijkstra. *Notes on structured programming*, chapter On the reliability of mechanisms. Technological University Eindhoven, 1970.

[7] National Institute for Standards and Technology. The economic impacts of inadequate infrastructure for software testing. Technical report, NIST, 2002.

[8] Hex-Rays. The ida pro disassembler and debugger. http ://www.hex-rays.com/idapro/.

[9] Microsoft. Microsoft Security Bulletin MS04-028 : Buffer Overrun in JPEG Processing (GDI+) Could Allow Code Execution (833987). http://www.microsoft.com/technet/security/bulletin/ms04-028.mspx.

[10] A. Lanzi ; L. Martignoni ; M. Monga ; R. Paelari. A smart fuzzer for x86 executables. Technical report, Milan University, 2007.

[11] C. Miller ; Z.N.L. Peterson. Analysis of mutation and generation-based fuzzing. Technical report, Independent Security Evaluators, 2007.

[12] Cody Pierce. Pyemu : A multi-purpose scriptable ia-32 emulator. http ://-code.google.com/p/pyemu/.

[13] Cody Pierce. Pyemu : A multi-purpose scriptable ia-32 emulator. 2007.

[14] The GNU project. Gcov - a test coverage program. http://gcc.gnu.org/onlinedocs/gcc/Gcov.html#Gcov.

[15] The Open Web Application Security Project. Buffer overflow. https://www.owasp.org/index.php/Buffer_Overflow.

[16] Martin Vuagnoux. Autodafe : an act of software torture. Technical report, Swiss Federal Institute of Technology, Cryptography and Security Laboratory, 2006.

[17] Wang ; Wei ; Gu ; Zou. Taintscope : A checksum-aware directed fuzzing tool for automatic software vulnerability detection. Technical report, Chinese ministry of Education, Peking University, Texas A and M University, 2010.