ENSEEIHT **3e année**
Département SN **TP**

Vulnérabilités logicielles - TP1

20 janvier 2023

Ce TP à pour objectif d'expérimenter les mécanismes liés aux attaques par dépassement de tampon qui ont été exposés en cours. Pour ce faire, de premiers exemples simples sont étudiés. Par la suite, il sera exposé comment utiliser les dépassements de tapons pour contrôler le flux d'exécution d'un programme. Enfin, il est exposé quels sont les problèmes et concepts auxquels l'attaquant est confronté afin de développer une charge utile malveillante.

Ce TP apporte aussi quelques notions autour des interfaces binaire applicatives et noyau, de rétro ingénierie et d'utilisation d'outils de débogage de logiciels.

---

1 Préparations

Si vous travaillez en salle réseau / sécurité ou en salle classique

```
$ mkdir /work/fw
$ cd /work/fw
$ wget -c 'http://flash.enseeiht.fr/bemorgan/lubuntu-18.04.15-tp-secu-logiciel-64.tgz'
$ tar xvf lubuntu-18.04.15-tp-secu-logiciel-64.tgz
$ cd lubuntu-18.04.15-tp-secu-logiciel-64
# sudo si vous n'avez pas le droit de créer des interfaces et de changer kvm
$ ./start.sh
```

Une fois la VM lancée, rendez-vous dans le dossier `~n7/tp1`

Si vous travaillez depuis chez vous sur une machine GNU / linux

— Installez `qemu`
— Connectez-vous au VPN de l'n7
— Suivez les mêmes étapes que la section suivante

Accès au fichiers de la VM

Qemu relaye en DNAT le couple `ip:port` hôte `localhost:2222` vers le couple de le couple `ip-vm:22` de la VM. Il est donc possible de se connecter sur le serveur OpenSSH de la VM pour télécharger les fichiers. Depuis l'hôte :

```
$ scp -P 2222 -r n7@localhost:/home/n7/tp1 .
$ sftp -P 2222 n7@localhost
```

2 Dépassements de tampon

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.

En ce qui concerne la gestion des structures de données de taille dynamique, comme les tableaux, le problème majeur de ce type de langages est que la plupart d'entre eux n'associent pas de contrôle de taille lors de leur manipulation en lecture où écriture.

Le but de cette partie est de comprendre : comment sont allouées en mémoire les variables de type primitifs ou structures de données ; comment elles sont manipulés par le langage C ; où sont situées les faiblesses de leur mise en œuvre ; et enfin comment en tirer partie intelligemment en exécutant des dépassements de tampons.

2.1 Allocation mémoire des variables de types primitifs

Soit le programme `tp1-1.c` :

```
#include <stdio.h>

int main(int argc, char *argv[]) {
  int a = 1;
  int b = -1;
  long int c = -4;
  printf("Vars : a(%d), b(%d), c(%ld)\n", a, b, c);
  return 0;
}
```

Compilez le programme avec le compilateurs et la commande suivante :

```
gcc -o tp1 tp1-1.c -mpreferred-stack-boundary=3
```

Cette ligne de commande invoque le compilateur sur le fichier source `tp1.c` en lui demandant de tenter d'allouer les variables locales alignées sur 8 octets. Ce type de directive permet de gagner de l'espace mémoire, mais rentre parfois en conflit avec des besoins d'optimisation. Les environnements embarqués, disposant d'un espace mémoire restreint, peuvent être des utilisateurs de ce type de configuration.

Le programme déclare et initialise trois variables de types primitifs et écrit leur valeur sur la sortie standard.

1. Compilez et exécutez le programme pour constater son fonctionnement.
2. Modifiez le programme de manière à afficher les adresses des trois variables que vous avez déclaré, que constatez-vous ?
3. Exécutez le programme plusieurs fois, que constatez vous ?

2.2 Allocation mémoire des structures de données

Soit le programme `tp1-2.c`

```
#include <stdio.h>

struct enregistrement {
  int a;
  int b;
  int c;
  int d;
};

int main(int argc, char *argv[]) {
  struct enregistrement a = {.a = 1, .b = 2, .c = 3, .d = 4};
  long int b [4] = {0, 1, 2, 3};
  char c [] = "chaine1";
  char d [] = "chaine2";
  printf("Vars : a(%d), b(%ld), c(\"%s\"), d(\"%s\")\n",
      a.a, b[0], c, d);
  return 0;
}
```

Ce programme déclare une variable de type primitif ainsi que deux structures de données de type enregistrement et tableau.

1. Compilez et exécutez le programme pour constater son fonctionnement.
2. Modifiez le programme de manière à afficher les adresses des trois structures de données que vous avez déclaré, que constatez-vous ?

2.3 Copie de chaîne de caractères

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

La plupart des fonctions de gestion de chaînes de caractères s'appuyent directement sur cette propriété lors de copies ou d'affiche age. 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.

1. Consultez la page `string` du manuel, section bibliothèque standard GNU, pour trouver la fonction de copie de chaîne de caractères.
2. Modifiez le programme `tp1-2.c` afin de copier dans la chaîne `d` le premier argument du programme.
3. Compilez le programme avec la commande suivante :

   ```
   gcc -o -O0 tp1-2 tp1-2.c -g -mpreferred-stack-boundary=3 -Wall  -Werror -fno-
       stack-protector
   ```

4. Exécutez le programme pour constater son fonctionnement.

5. Entrez entre 8 et 15 caractères en argument du programme, que ce passe-t-il ?
6. Entrez plus de 15 caractères, que ce passe-t-il ?

2.4 Premières attaques : confidentialité et intégrité des données

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

Soit le programme `tp1-3.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, 16);
  printf("Écris 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("Échec de l'authentfication,"
        " le mot de passe %s est faux\n", b);
  } else {
    printf("Succès de l'authentification\n");
  }
  return 0;
}
```

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.

Vous noterez 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.

1. Compilez le programme avec la commande suivante :

   ```
   gcc -o -O0 tp1-3 tp1-3.c -g -mpreferred-stack-boundary=3 -Wall  -Werror -fno-
       stack-protector
   ```

2. Constatez son fonctionnement.
3. Analysez le programme et identifiez où se situe le dépassement de tampon.
4. Comment est-ce que le développeur s'assure d'obtenir une chaine de caractères correcte après le traitement des entrées utilisateur ?
5. Proposez au programme une entrée bien choisie afin de réussir à réveler le secret placé en mémoire.
   — Souvenez-vous qu'il est possible de rediriger l'entrée standard du programme depuis un fichier.
   — Ci-besoin, pensez à utiliser le débogueur de programmes `gdb` pour vous simplifier la phase de compréension et d'exploitation de la vulnérabilité.

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.

Soit le programme `tp1-4.c`

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

int main(int argc, char *argv[]) {
  int session = 0;
  int uid = getuid();
  char secret[] = "secretFOU";
  char buf[32];
  printf("Écris 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; // Tout va bien !
  } else {
    printf("Le mot de passe n'est pas correct\n");
    return 1; // Tout va MAL !
  }
  return 0;
}
```

Ce programme demande à l'utilisateur de taper un mot de passe si celui-ci n'est pas administrateur du système. Comme nous l'avons vu dans les exercices précédents, l'identité de l'utilisateur est stockée en mémoire proche du tampon dans lequel l'entrée utilisateur est recueillie.

1. Compilez le programme et constatez son fonctionnement.
2. Analysez le programme et identifiez où se situe le dépassement de tampon.

3. Quelle valeur de la variable `uid` serait adéquate pour obtenir une session sans connaître le mot de passe ?
4. Proposez au programme une entrée bien choisie afin de place cette valeur dans `uid`
   — Utilisez un débogeur !
   — La commande `perl` suivante permet d'obtenir une chaine de caractères commençant par `"aaa"` et finissant par le nombre `0xaaaa`

   ```
   $ perl -e 'print "a"x4 . "\xaa\xaa"'
   ```

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

Soit le programme `tp1-5.c`

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

void a() {
  printf("Voici le secret : secret\n");
}

void b() {
  printf("Le mot de passe n'est pas bon :)\n");
}

int check() {
  char buf[32];
  printf("Écris moi le mot de passe sur fd 0\n");
  scanf("%s", &buf[0]);
  return !strcmp(buf, "secretFOU");
}

int main(int argc, char *argv[]) {
  if (check()) {
    a();
  } else {
    b();
  }
}
```

Ce programme demande à l'utilisateur de rentrer un mot de passe sur l'entrée standard et si et seulement si celui-ci est bon, affiche un secret à l'aide de la fonction `b()`.

1. Où se situe la vulnérabilité ?
2. Quelle différence fondamentale trouvez-vous entre le programme précédent et celui-ci en ce qui concerne la vérification du mot de passe ?
3. Compilez le programme avec la commande suivante :

   ```
   $ gcc -o tp1 tp1-5.c -g -mpreferred-stack-boundary=3 -Wall        -Werror -
       fno-stack-protector -no-pie
   ```

4. Constatez son fonctionnement.
5. Est-il possible d'utiliser le même type d'attaque que précédemment ? Pourquoi ?

Il est nécessaire d'adopter une autre stratégie pour pouvoir afficher le secret de la fonction `a()`.

Comme nous l'avons vu en cours, la pile est une zone mémoire utilisée pour la déclaration des variables locales, mais aussi pour sauvegarder le contexte des fonctions. En effet, l'adresse de retour de la fonction appelante est empilée au moment d'un appel de fonction et est dépilée lors de son retour.

Cette zone mémoire est donc essentielle pour le bon fonctionnement des programmes exécutés sur un processeur.

1. Déboguez le programme `tp1-5.c` avec `gdb` afin de reconstituer précisément son contexte de pile (papier crayons ou fichier texte !).
2. Constituez l'entrée adéquate afin de réécrire l'adresse de retour de la fonction `check` avec une adresse bien choisie afin d'afficher le secret de la fonction `a()`. Cette addresse est absolue. Le système se protège contre l'exploiration de ce type de vulnérabilité en rendant aléatoire l'adresse de chargement des section du programme (ASLR). Cette mesure de protection rend la recherche de l'adresse de la fonction `a()` difficile. Si vous travaillez sur machine virtuelle nous allons désactiver la `randomisation` de l'espace d'adressage :

   ```
   $ sudo sysctl kernel.randomize_va_space=0
   ```

   Si vous travaillez sur machine fixe, la suite de ce TP doit s'effectuer sous `gdb`, avec la `randomisation` de l'espace d'adressage désactivée.
3. Que se passe-t-il une fois que le secret est affiché ? Pourquoi ?

ATTENTION dans ce TP, les machines utilisées possèdent un processeur supportant un mode d'exécution 64-bits. Ceci signifie que les registres généraux (eax, ebx, etc.) ont une taille de 64-bits et que l'unité arithmétique et logique du processeur peut les traiter. Ceci à aussi pour effet de bord d'augmenter l'espace d'adressage virtuel disponible pour les applications (48-bits). Les sauvegardes de contexte et adresses de retour ont par conséquent une taille de 64-bits une fois empilées (si on aligne sur une puissance de 2). Enfin, les arguments de fonctions sont de préférence passés par registre (utilisation du paradigme *fast call*). L'ABI utilisée par GCC est la System v ABI.

4 Exécution de code arbitraire : *shellcode*

Dans l'exercice précédent nous avons vu qu'il était possible de contrôler le flux 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 est possible de réutiliser du code déjà présent en mémoire comme suite d'exécution du programme.

Dans cet exercice, nous allons utiliser les dépassement de tampon pour exécuter du code arbitrairement placé en mémoire lors de l'exécution dudit dépassement. Enfin, nous allons nous placer dans la peau d'un attaquant dont l'objectif est d'obtenir l'exécution d'un *shell* à distance.

4.1 Conception de l'attaque

Il existe plusieurs solutions pour concevoir un *shellcode*. Certaines étapes de génération de l'entrée malveillante sont indispensable à un fonctionnement correct. Il est donc important de bien comprendre ses différentes étapes de génération.

1. En utilisant les notions vues en cours, donnez la conception ainsi qu'un algorithme informel de génération de *shellcode*.
   — Comment obtenir en mémoire la localisation de l'adresse de retour et du tableau écrit par la copie vulnérable ?
   — Comment calculer la taille que doit avoir l'entrée malveillante afin de réécrire l'adresse de retour ?
   — Par quelle adresse celle-ci doit être remplacée ?

4.2 Conception de la charge utile : exécution d'un *shell*

L'appel système permettant de charger et d'exécuter un nouveau programme est `execve`. Il est mise en œuvre dans la fonction portant le même nom de la bibliothèque standard C.

1. L'interface de l'appel système `execve()` est document dans la page de manuel associée. Cela vous permettra de connaître les arguments attendus par l'appel système. Attention au format attendu pour le tableau `argv`.
2. Développez un programme C simple `tp1-6.c` utilisant cet appel système.
3. Analysez en détail les types des paramètres de la fonction `execve()`. Vous pouvez utilisez le débogueur `gdb` pour confirmer vos intuitions.

La difficulté de l'injection de code dans l'espace mémoire d'un programme distant est liée à la méconnaissance de son environnement d'exécution. Par exemple, il est difficile de déterminer où sont chargées les bibliothèques partagées en mémoire, et quelles sont-elles. On parle dans ce cas de programmes *bare-metal* qui peut être traduit dans ce cadre là comme auto suffisant. Nous avons vu que l'appel système dont nous avons besoin pour exécuter un programme est mis en œuvre dans la bibliothèque C. Or, notre code malveillant exécuté depuis la pile ne saura pas où cette bibliothèque est chargée. Enfin, adapter notre *shellcode* à une version de bibliothèque limiterait fortement sa portabilité entre versions de systèmes.

Il est donc nécessaire de trouver un autre moyen d'interaction avec le noyau qui soit fonctionnel et utilisable sur différentes machines vulnérables.

Le noyau linux a standardisé l'interface de ses appels systèmes en suivant la norme `POSIX`. Aussi, dans les version modernes de `linux` l'instruction utilisée pour effectuer

un appel système est l'instruction `syscall`. Cette fonction invoque explicitement le noyau pour exécuter un appel système. Interagir directement avec le noyau limite la connaissance à priori à avoir sur le système, mais aussi maximise sa portabilité : tous les systèmes `POSIX`.

1. L'instruction `syscall` se contente de donner la main au noyau. Elle ne spécifie par d'interface bas niveau, ou *Application Binary Interface* (ABI), spécifiant comment renseigner concrètement les arguments [1]. Consultez le fichier du noyau linux `arch/x86/entry/entry_64.S` et plus particulièrement l'étiquette `entry_SYSCALL_64` afin de connaître l'ABI du noyau pour l'instruction `syscall`.
2. Le type d'appel système à exécuter est donné par son numéro. Consultez le fichier du noyau linux `entry/syscalls/syscall_64.tbl` pour déterminer le numéro de l'appel système `execve()`. Confirmez aussi la version de l'ABI.

Nous avons maintenant toutes les informations nécessaires afin pas passer à l'étape du développement. Afin de simplifier la mise en œuvre, les attaquants séparent en général la phase de développement de la "charge utile" et la phase d'intégration dans l'entrée malveillante. La charge utile est développée dans un environnement favorable puis ensuite est extraite pour être intégrée dans l'entrée malveillante.

Soit le programme `tp1-7.c` :

```
int main(int argc, char *argv[]) {
  __asm__ __volatile__("jmp shellcode;");
  return 0;
}
```

Et le programme `tp1-7.s` :

```
.global shellcode
shellcode:
  nop
  nop
  nop
//
// Rappel assembleur:
//  mov $0x1, %rax // Met la valeur 0x1 dans le registre rax
//  syscall // execute l'appel système
//  mov 0x8, %rax // copie depuis la mémoire à l'adresse 0x8 dans rax
//  mov %rax, 0x8 // Copie le registre raxdans la mémoire à l'addresse 0x8
//  push %rax // empile rax
//  pop %rax // depile rax
```

Le fichier `tp1-7.c` met en œuvre l'interface classique des programmes C GNU. Puis, il saute directement vers l'étiquette assembleur *shellcode* où vous devrez placer votre charge utile pour l'attaque.

1. Compilez le programme avec la commande suivante :

1. Pour rappel sur linux 32 bits les arguments sont empilés, alors que sous linux 64 bits, les arguments sont renseignés à l'aide des registres généraux. On parle alors de *fast call*.

```
$ gcc -o tp1 -Wall -Werror tp1-7.c tp1-7.s
```

2. Utilisez le programme assembleur `tp1-7.s` pour développer votre *shellcode*. Demandez de l'aide à l'enseignant pour le language d'assemblage.
   (a) Placez la chaine de caractère `"/bin/sh"` sur la pile. Attention, comment se termine une chaîne correcte ?
   (b) Renseignez son pointeur dans le premier argument de l'appel système. **NOTA BENE** : les étapes suivantes seraient normalement nécessaires pour plus de furtivité. Dans ce TP nous allons renseigner uniquement l'argument 1, c'est à dire le programme à exécuter. Vous pouvez à votre guise développer la suite, mais ne passez pas trop de temps dessus.
   (c) Placez le tableau argv sur la pile. Attention à sa terminaison.
   (d) Renseignez son pointeur dans le deuxième argument de l'appel système.
   (e) Renseignez le troisième argument de l'appel système.

Nota bene Nous avons vu dans le cours qu'il est préférable de ne pas utiliser la pile le plus possible. Ceci est réalisable en plaçant une section de données en fin de *shellcode*. Pour simplifier ce travail pratique, il est demandé d'empiler directement la chaîne `"/bin/sh"` pour simplifier la conception de l'attaque. Nous considérons que la fonction vulnérable a alloué assez de mémoire dans le pile pour fonctionner de la sorte. Aussi, nous ne chiffrerons pas le binaire.

Une fois que votre charge utile est développée, il faut s'assurer qu'elle pourra être copiée avec l'entrée malveillante.

1. Quels sont les points indispensables pour une copie de chaîne de caractère réussie ?
2. Observez le contenue binaire de votre étiquette *shellcode* pour vérifier sa compatibilité avec les exigences précédentes :

   ```
   $ objdump -D tp1 | less
   /<shellcode>:
   ```

3. Apportez les modification nécessaires afin de rendre compatible votre charge utile avec les exigences précédentes. Demandez de l'aide à l'enseignant !

Le *shellcode* est maintenant développé de manière intégrée à un programme GNU/linux. Il ne reste plus qu'à le distribuer sous la forme d'une suite brute d'instructions.

1. Compilez de manière séparée le *shellcode* `tp1-7.s` de manière à ne traduire que la charge utile :

   ```
   $ gcc -c -Wall -Werror <programme>.s
   ```

2. Extrayez la section de code `.text` du fichier objet compilé :

   ```
   $ objcopy -O binary --only-section=.text <programme>.o <shellcode>
   ```

3. Vérifiez les instructions de votre `shellcode` généré

   ```
   $ objdump -b binary -D -mi386 -Mx86_64 <shellcode>
   ```

Une fois que le `shellcode` est prêt nous pouvons l'intégrer dans une attaque finale.

4.3 Intégration et exécution de l'attaque

Soit le programme tp1-8.c :

```
#include <stdio.h>
#include <string.h>
#include <bsd/string.h>
#include <stdint.h>
#include <stdlib.h>

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

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

  a[0x3f] = '\0';
  b[0x3f] = '\0';

  printf("Before copy %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;
}

void usage(void) {
  fprintf(stderr, "Missing Parameter\n");
}

int main(int argc, char *argv[]){
  char buf[256];
  if (argc < 2) {
    usage();
    exit(1);
  }
  strlcpy(&buf[0], argv[1], 256);
```

```
  f(&buf[0]);
}
```

À l'aide des différents points abordés lors de cet exercice, exécutez un dépassement de tampon sur le programme précédent afin d'exécuter un *shellcode*.

1. Compilez le programme avec la commande suivante :

   ```
   $ gcc -o tp1 tp1-8.c -g -mpreferred-stack-boundary=3 -Wall -Werror -fno-stack
       -protector -z execstack -no-pie -lbsd
   ```

2. Identifiez la vulnérabilité. Commentez.
3. Vous devez maintenant identifier clairement une adresse de retour vers laquelle sauter proche tu tableau `b`.
4. Concevez l'entrée malveillante incluant le *shellcode*. Vous pouvez générer un argument de la manière suivante :

   ```
   $ echo -n `perl -e 'print "l"x20'``cat shellcode``perl -e 'print "m"x40 . "\
       xca\xfe\xba\xbe"'`
   ```

5. Exécutez l'attaque.
6. Reposez-vous.