TP - Exploitation binaire - Partie 2

Objectifs

- Se familiariser avec la pratique des attaques par buffer overflow
- Comprendre en profondeur le fonctionnement de la stack et la manipulation de données
- Mettre en place des attaques liées au buffer overflow
- Détourner certaines protections

Prérequis

- Une machine Linux sous architecture x86_64
- Compilateur (GCC) et débugger (GDB) installés avec certains plugins (notamment `gef`).
- ghidra et/ou IDA pour désassembler/décompiler les binaires si besoin
- Librairie python `pwntools` pour faciliter l'exploitation

Pour l'installation de `gcc` si vous ne l'avez pas encore sur votre machine :

```
sudo apt update
sudo apt install build-essential

# Vérifiez la version de gcc
gcc --version
```

`gdb` est par défaut installé sur la plupart des distributions mais si jamais vous avez besoin de le faire :

```
sudo apt update
sudo apt install gdb
```

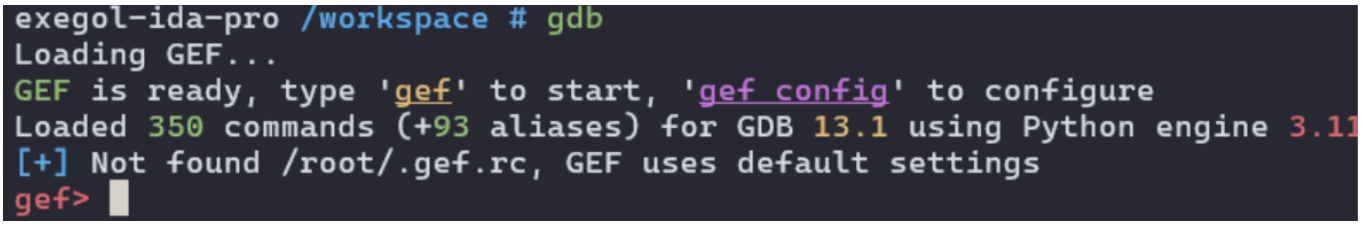
Installer le plugin `gef` pour `gdb` qui vous aidera grandement lors de vos phases de debugging :

```
wget -q https://raw.githubusercontent.com/bata24/gef/dev/install.sh -O- | sed -e 's/pip3 install/pip3 install --break-system-packages/g' | sh
```

Github associé : https://github.com/bata24/gef

Si `gef` est bien installé, vous devriez obtenir quelque chose qui ressemble à ça lors que vous ouvrirez `gdb` :

```
gdb
```



Installer la version d'essai d'IDA Pro : https://hex-rays.com/ida-pro

Pour ghidra : https://github.com/NationalSecurityAgency/ghidra/releases

Si jamais vous voulez jouer avec des versions 32 bits, vous aurez besoin du paquet `gcc-multilib` :

```
sudo apt update
sudo apt install gcc-multilib
```

Installation de `pwntools` : https://docs.pwntools.com/en/dev/install.html

---

Rappels

Certains binaires sont fournis tels quels, d'autres possèdent également un code source. Dans certains cas vous devrez donc comprendre son fonctionnement à l'aide de débuggeurs et de désassembleurs (voir décompilateurs).

Avant toute chose, n'oubliez pas de désactiver la protection ASLR sur votre machine afin de ne pas avoir de mauvaise surprise lors de la réalisation de ce TP :

```
sudo su
echo 0 > /proc/sys/kernel/randomize_va_space
```

Attention, si vous êtes sur un container Docker par exemple, vous devez le faire sur votre hôte et pas à l'intérieur du container.

Pour réactiver l'ASLR il faudra juste reboot votre machine ou remodifier la valeur du fichier :

```
echo 2 > /proc/sys/kernel/randomize_va_space
```

Rendu

Le TP devra être rendu en format `PDF` de préférence (Word ou Markdown également possible) à l'adresse mail : quentin.simier@synoslabs.com

Le rapport doit contenir l'ensemble des exercices avec les explications associées et les réponses demandées.
Vous devez également fournir le code concernant l'exploit utilisé (fichier python avec le code `pwntools` utilisé). Vous pouvez le faire via un fichier ZIP ou directement sur un Github public avec partage du lien pour que je puisse y accéder.

Vous pouvez faire le TP par groupe de 4 max sachant qu'il y a pas mal d'exercices, cela vous permettra de mieux répartir les tâches. N'hésitez pas cependant à vous renseigner sur ce qu'à fait votre camarade pour avoir une vue d'ensemble des différentes techniques utilisées.

N'hésitez pas à intégrer des captures d'écran dans votre rapport lorsque vous effectuez vos tests sur `gdb` par exemple pour montrer les valeurs des registres, de la stack, etc...

Dans les cas où les codes sources sont fournis et que vous exploitez la vulnérabilité sur un binaire que vous avez compilé vous même, mettez-le également dans les fichiers rendus pour que je puisse vérifier votre solution sur le même binaire que vous avez utilisé.

Énoncé

Voici la liste des "challenges" qu'il faut réaliser. A chaque fois il vous sera indiqué ce qu'il faut faire et si besoin les commandes de compilation si le fichier C vous est fourni pour que vous puissiez obtenir le même résultat.

Comme chaque machine est différente, il se peut que certaines valeurs d'adresses ne soient pas les mêmes entre les différentes machines. N'oubliez pas ça lors de vos tests si jamais vous échangez des fichiers.

Exercice 1 : Buffer overflow basique

Lien vers le binaire :
https://drive.google.com/file/d/1FpSY0xAghWr7Vq_Xkb2H0THct5ilbtQ4/view?usp=sharing

Le but de cet exercice est de comprendre ce que fait un programme donné, comment il fonctionne et comment retrouver les instructions utiles du binaire en utilisant des outils comme `ghidra`, `ida` et `gdb`, puis de l'exploiter avec un buffer overflow pour afficher le message attendu.
Si jamais vous souhaitez utiliser d'autres outils vous avez le droit mais n'oubliez pas d'indiquer les différentes étapes effectuées ainsi qu'une explication si possible détaillée.

Attention, vous ne devez en aucun cas modifier le binaire. Vous n'avez que le droit de l'exécuter et d'y passer les données que vous souhaitez.

Vous pouvez le faire manuellement (dans ce cas, attention au little-endian) ou avec un script python qui utilise `pwntools`.

Code source pour vous aider :

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

void main() {
    int64_t is_admin = 5;
    char buffer[32];

    printf("Entrez votre nom : ");
    gets(buffer);

    if (is_admin) {
        printf("Accès autorisé\n");
    } else {
        printf("Accès refusé ...\n");
    }
}
```

Exercice 2 : Canary

Lien vers le binaire : https://drive.google.com/file/d/14Sep6buqjN-9GYrOherE4x4KVZHdrkD7/view?usp=sharing

Ce binaire contient une protection de type canary qui a été mis en place par un développeur pas très doué.
En effet, il l'a implémenté manuellement et en dur dans le code sans utiliser l'options fournie par le compilateur lors de la compilation du programme.
De ce fait, il est possible de retrouver cette valeur et de la réécrire pour finalement réussir le buffer overflow.

Rappel : le principe du canary est d'écrire une valeur intermédiaire dans le code de façon à ce que dès qu'il y a une tentative de buffer overflow, cette valeur est modifiée et est donc différente entre le début du programme et la fin. Si une modification est détectée alors le programme s'arrête.

Votre but est d'analyser ce binaire, comprendre comment il fonctionne pour réussir à bypass le canary et finalement afficher le message de victoire !

Exercice 3 : ret2win

Lien vers le binaire :
https://drive.google.com/file/d/1fXOkvN6xmXc1rZa0fnVg0Rq48N0pr9fG/view?usp=sharing

Comme vu en cours, ce binaire a pour but de vous entraîner à effectuer un buffer overflow sur celui-ci afin de réécrire la valeur de retour et d'exécuter la fonction cachée qui vous mènera au shell sur la machine !

Lors qu'on commence à jouer avec des adresses mémoire il est préférable d'automatiser certaines actions (notamment avec `pwntools`) et de visualiser l'état des différents éléments : stack, registres, etc.. avec `gdb`.

N'oubliez pas qu'avec GDB vous pouvez vous attacher à un processus en cours afin de visualiser ces informations (et notamment la stack) : `attach <PID>`

Exercice bonus

Reprenez le même binaire que pour l'exercice sur le bypass du canary.
Avez-vous remarquer quelque chose d'intéressant dedans ? Est-il possible de faire en sorte d'obtenir quelque chose de beaucoup plus sensible qu'un simple affichage d'un message de victoire ?

Indice : il s'agit d'un mix entre l'exercice 2 et 3.

Exploitez cette vulnérabilité et prenez le contrôle de la machine.

Aide

Voici quelques rappels de commandes qui pourraient vous être utiles lors du TP.

`gdb` cheatsheet : https://darkdust.net/files/GDB%20Cheat%20Sheet.pdf

```
gdb ./binary
```

En fonction de si vous avez du code C ou votre propre assembleur avec le label `_start` par exemple :

```
disass main
# ou
disass _start
```

Mettre un breakpoint sur main pour arrêter l'exécution à ce point :

```
b* main
```

Afficher le contenu de la stack :

```
tele
```

Aller à la prochaine instruction (next instruction). Attention ne rentre pas en détail dans les fonctions. Préférez utiliser `si` dans ce cas :

```
ni
```

Aller à la prochaine instruction (step into). Peut rentrer en détail dans les fonctions

```
si
```

Lancer un programme :

```
run
```

Afficher les infos sur les fonctions disponibles :

```
info func
```

Afficher les infos sur les registres actuels (préférer utiliser `tele` avec `gef`) :

```
info register
```

S'attacher avec GDB au process lancé avec `pwntools` afin de visualiser l'état de la mémoire au milieu de l'exécution du programme :

- Mettez un `pause()` dans votre script python avant d'envoyer les infos au binaire.
- Lancez GDB sur le binaire.
- Positionnez un breakpoint juste après un `gets`, `read` ou toute autre lecture utilisateur.
- Lancez votre script python, il devrait vous donner le PID du process en cours.
- Dans GDB attachez vous au numéro de PID avec `at <PID>`
- Toujours dans GDB, faites `continue` ou `c`
- Normalement GDB doit être en attente d'une entrée utilisateur. Sur votre script python, faites "Entrée" afin de sortir de la pause.
- Bien joué ! Vous êtes actuellement sur le breakpoint que vous avez positionné juste avant et vous pouvez visualiser l'état de la stack et des variables dans GDB par rapport aux données qui ont été envoyées par le script python.