Chapitre 6 : Sécurité de la Mémoire
Buffer Overflow sur Systèmes 32-bits

Cours de Sécurité Informatique - Niveau Universitaire
Partie 4 : Software Security

12 janvier 2026

Table des matières

1 Introduction

1.1 Problématique

Les langages comme C/C++ offrent un accès direct à la mémoire mais sans vérification automatique des bornes. Cela permet des erreurs de programmation dangereuses : **buffer overflows**.

Conséquence : Exploitation pour exécution de code arbitraire, escalade de privilèges, DoS.

1.2 Statistiques

— 70% des vulnérabilités Microsoft (historiquement) : memory safety

— CVE les plus critiques : souvent buffer overflows

— Impact : Wannacry, Heartbleed, Stuxnet, etc.

2 Organisation de la mémoire (32-bits)

2.1 Layout mémoire d'un processus

```
Adresses hautes
+------------------+
|   Kernel space   |  (réservé OS)
+------------------+
|      Stack       |  (croît vers le bas)
|        |         |
|        v         |
+------------------+
|       Heap       |  (croît vers le haut)
|        ^         |
|        |         |
+------------------+
|   .bss (données  |  (variables non initialisées)
|    non init.)    |
+------------------+
|   .data (données |  (variables globales initialisées)
|    initialisées) |
+------------------+
|      .text       |  (code exécutable)
+------------------+
Adresses basses
```

2.2 Organisation de la pile (Stack)

Chaque fonction appel crée un stack frame :

```
Stack frame de f():
+-------------------+ <- ESP (Stack Pointer)
| Variables locales |
+-------------------+
| Saved EBP         | <- EBP (Base Pointer)
+-------------------+
| Return Address    | <- Adresse de retour
+-------------------+
```

```
| Arguments          |
+--------------------+
```

Instructions :

- `CALL func` : Empile return address, saute à func
- `RET` : Dépile return address, saute à cette adresse
- `PUSH / POP` : Manipule la pile

3 Buffer Overflow sur la pile

3.1 Principe

Code vulnérable :

```
void vulnerable_function(char *input) {
    char buffer[64];
    strcpy(buffer, input);  // Pas de v rification de taille !
}

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

Problème : Si `input` $>$ 64 bytes, débordement du buffer.

3.2 Exploitation : Écraser le return address

Objectif : Contrôler le return address pour rediriger l'exécution.
Stratégie :

1. Écrire du shellcode dans le buffer
2. Écraser return address avec l'adresse du shellcode
3. Quand la fonction retourne, exécution du shellcode

Payload structure :

```
[Shellcode (ex: spawn shell)] + [Padding] + [Return Address]
```

3.3 Shellcode

Définition : Code machine pour exécuter une action (ex : lancer `/bin/sh`).
Exemple x86 (Linux) : Syscall `execve("/bin/sh")`

```
Shellcode (hex):
\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e...
```

Contraintes :

- Pas de bytes NULL (\x00) si exploit passe par `strcpy()`
- Taille réduite

3.4 Déterminer l'adresse de retour

Méthode 1 : Debugger (gdb)

```
gdb ./vulnerable
(gdb) break vulnerable_function
(gdb) run $(python -c 'print "A"*100')
(gdb) x/100x $esp  # Examiner la pile
```

Méthode 2 : NOP sled (glissière de NOPs)

- Préfixer shellcode par beaucoup de `NOP` (\x90)
- Viser approximativement au milieu
- Si on atterrit sur un NOP, glisse jusqu'au shellcode

```
[NOPs x 100] + [Shellcode] + [Padding] + [Adresse dans NOPs]
```

4 Contre-mesures

4.1 Programmation sécurisée

- **Éviter fonctions dangereuses** : `strcpy`, `gets`, `sprintf`
- **Utiliser versions sûres** : `strncpy`, `fgets`, `snprintf`
- **Vérifier bornes** : Toujours valider taille inputs
- **Static analysis** : Outils comme Coverity, Clang Static Analyzer

4.2 Stack Canaries

Principe : Placer une valeur aléatoire (canary) entre buffer et return address.

```
Stack frame avec canary:
+-------------------+
| Variables locales |
+-------------------+
| Stack Canary      | <- Valeur aléatoire secrète
+-------------------+
| Saved EBP         |
+-------------------+
| Return Address    |
+-------------------+
```

Mécanisme : Avant de retourner, vérifier que le canary n'a pas changé. Si modifié, crasher le programme.
Activation : GCC avec `-fstack-protector` ou `-fstack-protector-all`
Contournement : Si attaquant peut lire le canary (info leak), il peut le préserver.

4.3 NX bit (Data Execution Prevention)

Principe : Marquer la pile comme non-exécutable.
Effet : Même si attaquant injecte shellcode, ne pourra pas l'exécuter.
Hardware support : CPU moderne (NX bit x86, XD bit Intel, XN bit ARM)
OS : Windows DEP, Linux NX
Contournement : Return-Oriented Programming (ROP) - réutiliser code existant.

4.4 ASLR (Address Space Layout Randomization)

Principe : Randomiser les adresses mémoire à chaque exécution.
Zones randomisées : Stack, heap, bibliothèques, exécutable (PIE)
Effet : Attaquant ne peut pas prédire où placer son shellcode ou quelle adresse cibler.
Activation :

```
Linux: echo 2 > /proc/sys/kernel/randomize_va_space
Windows: ASLR activé par défaut depuis Vista
```

Contournement : Info leaks (révéler adresses), brute force (32-bits), ROP.

4.5 Compilation avec protections

```
gcc -fstack-protector-all -D_FORTIFY_SOURCE=2 -fPIE -pie \
    -Wl,-z,relro,-z,now -o program program.c
```

- `-fstack-protector-all` : Stack canaries
- `-fPIE -pie` : Position Independent Executable (ASLR)
- `-Wl,-z,relro,-z,now` : Read-only GOT/PLT
- `-D`$FORTIFY_SOURCE = 2$: *Runtimechecks*

5 Techniques avancées d'exploitation

5.1 Return-Oriented Programming (ROP)

Contexte : NX activé, impossible d'exécuter shellcode injecté.
Idée : Chaîner des "gadgets" (petits bouts de code se terminant par `RET`).
Gadget :

```
pop eax ; ret   # Gadget 1
pop ebx ; ret   # Gadget 2
int 0x80 ; ret  # Gadget 3 (syscall)
```

Payload ROP :

```
[Padding] + [Addr Gadget1] + [Data] + [Addr Gadget2] + ...
```

Outils : ROPgadget, pwntools

5.2 Heap overflow

Principe : Débordement sur le tas (heap) au lieu de la pile.
Exploitation : Plus complexe, cible métadonnées malloc/free.
Techniques : Use-After-Free, Double-Free, Heap spraying.

6 Langages sûrs et alternatives

Avertissement

La vraie solution : Utiliser des langages memory-safe.
Langages sans buffer overflows :

- **Rust** : Ownership, borrow checker (sécurité à la compilation)
- **Go** : Garbage collection, vérification bornes

— **Python, Java, C#** : Managed memory

Tendance industrie : Migration progressive vers Rust (ex : Microsoft, Google, Linux kernel).

7 Travaux pratiques

7.1 Exercices théoriques

1. Dessiner le stack frame d'une fonction et identifier où se trouve le return address.
2. Calculer l'offset nécessaire pour écraser le return address dans un buffer de 64 bytes.
3. Expliquer pourquoi NX seul n'est pas suffisant (ROP).
4. Décrire comment ASLR + stack canaries combinés renforcent la sécurité.

7.2 Exercices pratiques

— `06_demo_buffer_overflow.c` : Code vulnérable + exploitation

— `06_demo_rop.py` : Génération de chaîne ROP

— `06_demo_protections.sh` : Tester stack canaries, NX, ASLR

Environnement : VM Linux 32-bits avec protections désactivées (pour apprentissage).

Avertissement

Éthique : Ces techniques sont enseignées à des fins défensives (comprendre pour mieux protéger). Toute exploitation malveillante est illégale.

8 Conclusion

Points clés :

— Buffer overflows : exploitation classique mais toujours dangereuse

— Défenses : Stack canaries, NX, ASLR (défense en profondeur)

— ROP : Technique avancée pour contourner NX

— Solution long terme : Langages memory-safe (Rust, Go)

Évolution : Exploitation de plus en plus difficile grâce aux protections modernes, mais toujours possible (complexité accrue).

"The best way to learn security is to break things (legally)."
— Principe du hacking éthique