Exploitation des chaînes de format

CSC_4CS03_TP – Cyberattaques : menaces et mises en œuvre

Thomas Cadegros Yahya Moustahsane

Février 2026

Sommaire

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

1. Analyse préliminaire du programme

Détection de la vulnérabilité

Payload injecté : `AAAA %p %p %p %p %p`

```
$ ./vuln
Please Insert an IP address to ping:
AAAA %p %p %p %p %p
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

Session de débogage

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

```
(gdb) run
Please Insert an IP address to ping:
AAAAAAAA %7$p %27$p
AAAAAAAA 0x7fffffffdc38 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)

Analyse des données exfiltrées

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 : $\texttt{0x54f3} - 5 = \texttt{0x54ee}$ (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 :

```
0x555555555562 <+98>: pop %r15 ; opcode: 41 5f
0x555555555564 <+100>: ret ; opcode: c3
```

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

$$\text{Adresse gadget} = \texttt{0x555555555562} + 1 = \texttt{0x555555555563}$$

Calcul des offsets

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

Distances relatives :

- $\Delta_{\text{Gadget}} = \texttt{0x5563} - \texttt{0x54f3} = \mathbf{+0x70}$ (112 octets)
  - $\rightarrow$ < 256 : atteignable en modifiant 1 seul octet (LSB)
- $\Delta_{\text{System}} = \texttt{0x5100} - \texttt{0x54f3} = \mathbf{-0x3F3}$ (–1011 octets)
  - $\rightarrow$ 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`

```
0x5555555554ee <+24>: call makePing ; <-- cible (LSB =0xee)
0x5555555554f3 <+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)

Algorithme de la boucle d'attaque

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**