Exploitation des chaînes de format (Format String)

CSC_4CS03_TP - Cyberattaques : menaces et mises en œuvre

Thomas Cadegros, Yahya Moustahsane

Février 2026

Pour identifier la faille, nous avons exécuté le programme en fournissant une chaîne de caractères contenant des spécificateurs de format. Si le programme est vulnérable, il interprétera ces spécificateurs au lieu de les afficher littéralement.
Nous avons utilisé le payload suivant : `AAAA %p %p %p %p %p` .

- `AAAA` : Sert de motif reconnaissable (0x41414141) pour repérer notre buffer sur la pile.
- `%p` : Spécificateur de format demandant l'affichage d'un pointeur (adresse mémoire) situé sur la pile.

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

Listing 1 – Test d'injection de chaînes de format

```
thomas@zephyrus:~/work/devoir2026$ ./vuln
Checking ping is on the system...
/usr/bin/ping
Please Insert an IP address to ping:
AAAA %p %p %p %p %p
AAAA 0x5acad81676b1 0xfbad2288 0x7e170491ba91 0
    x5acad81676c4 0x410
ping: %p: Name or service not known
```

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. Comme expliqué dans le cours, la fonction `printf` (ou famille) dépile les arguments correspondants aux spécificateurs `%p` même s'ils n'ont pas été fournis par le programmeur.

La vulnérabilité détectée est une **Format String Vulnerability** (CWE-134).
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);
```

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.
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.
Nous avons exécuté le programme vulnérable sous GDB en injectant une chaîne de format spécifique pour inspecter le $7^{\text{ème}}$ et le $27^{\text{ème}}$ mot de la pile.
Commande GDB utilisée : `run` puis injection de `AAAAAAAA %7$p %27$p`.

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

```
(gdb) run
Starting program: /home/thomas/work/devoir2026/vuln
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu
    /libthread_db.so.1".
Checking ping is on the system...
```

```
/usr/bin/ping
Please Insert an IP address to ping:
AAAAAAAA %7$p %27$p
AAAAAAAA 0x7fffffffdc38 0x5555555554f3
[Detaching after vfork from child process 745608]
ping: %27: Name or service not known
```

L'architecture x86_64 utilise les registres `RDI`, `RSI`, `RDX`, `RCX`, `R8`, `R9` pour les 6 premiers arguments. Les arguments suivants sont placés sur la pile. Le spécificateur `%7$p` pointe donc vers la première valeur disponible au sommet de la pile (`RSP`) au moment de l'appel à `printf`.
La valeur obtenue est `0x7fffffffdc38`.

- **Observation :** Il s'agit d'une adresse de la pile (commençant par `0x7fff...`).
- **Interprétation :** Contrairement à une exécution où l'on verrait `0x41414141` (la valeur hexadécimale de "AAAA"), nous voyons ici un pointeur. Cela indique que le sommet de la pile à cet instant contient une adresse pointant vers une zone de la stack (probablement une variable locale ou le buffer lui-même).

- **Rôle dans l'attaque :** C'est un point d'ancrage essentiel. Si nous utilisons le spécificateur `%7$n` (écriture), nous écrirons à l'adresse `0x7fffffffdc38`.

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** de la fonction vulnérable vers la fonction `main`.
- **Preuve par le calcul :** Le script d'exploitation indique que l'adresse de retour cible (le début de l'instruction après l'appel) se termine par l'octet `0xee`.

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

L'adresse obtenue dans GDB (...`54f3`) est située exactement 5 octets après l'adresse cible mentionnée dans l'énoncé. Cela confirme que `0x5555555554f3` est bien l'adresse de

l'instruction suivant immédiatement l'appel à `makePing` dans le `main`.

Nous pouvons reconstituer l'état de la mémoire lors de la vulnérabilité :

Offset printf	Adresse (Exemple)	Contenu	Description
%7$p	`0x7fffffffdc38`	Pointeur Stack	Sommet de la p
...	...	...	Variables locales
...	...	Saved RBP	Sauvegarde du
%27$p	`0x5555555554f3`	**Saved RIP**	**Adresse de ret**
%28$p	...	...	Zone stable pou

Table – Organisation de la pile lors de l'appel à printf

Conclusion de l'analyse dynamique : Nous avons identifié l'emplacement de l'adresse de retour (Offset 27). La distance entre le sommet de la pile (Offset 7) et l'adresse de retour est de 20 "mots" machine (soit $20 \times 8 = 160$ octets). Ces informations sont suffisantes pour calculer les distances nécessaires à la construction

de 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.
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 :

```
0x000055555555554ee <+24>: call 0x5555555553b3 <makePing>
0x000055555555554f3 <+29>: mov $0x0,%eax <-- ADRESSE REF
```

- **Adresse de System (*System*)** : L'entrée de la fonction dans la PLT.
- **Adresse du Gadget (*Gadget*)** : Nous avons identifié un "Universal Gadget" dans la fonction `__libc_csu_init`.

L'instruction explicite `pop rdi` n'est pas présente. Cependant,

l'instruction `pop r15` située à la fin de la fonction d'initialisation permet de la simuler.

Listing 3 – Désassemblage partiel de `__libc_csu_init`

```
1 0x000055555555562 <+98>: pop %r15
2 0x000055555555564 <+100>: ret
```

En code machine, `pop r15` est encodé par `41 5f`. L'instruction `pop rdi` correspond à l'opcode `5f`. En sautant le premier octet (offset +1), le processeur exécute `5f` (pop rdi) suivi de `c3` (ret).

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

Élément	**Description**	**Adresse (Hex)**
Référence (*Ref*)	Adresse de retour (dans main)	`0x5555555554f3`
Cible 1 (*Gadget*)	`pop rdi ; ret` (csu_init)	`0x555555555563`
Cible 2 (*System*)	`system@plt`	`0x555555555100`

Table – Adresses relevées sous GDB

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

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

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

Interprétation : Le gadget se trouve exactement 112 octets (0x70) après l'adresse de retour. Cette distance est très faible (< 256). Cela confirme qu'il est possible d'atteindre ce gadget en modifiant uniquement l'**octet de poids faible (LSB)** de l'adresse de retour (remplacement de `0xf3` par `0x63` et ajustement éventuel du nibble précédent), validant la stratégie de l'attaque sans connaître l'adresse de base complète (ASLR).

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

$$\Delta_{System} = 0x555555555100 - 0x5555555554f3 = \mathbf{-0x3F3}$$

Interprétation : La fonction `system` se situe 1011 octets avant l'adresse de retour. Ce décalage constant sera utilisé dans le script

d'exploitation pour calculer l'adresse réelle de `system` à partir de la fuite d'adresse (leak).
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 devons utiliser la boucle d'exploitation (créée en modifiant le LSB de l'adresse de retour) pour écrire la chaîne ROP élément par élément, itération après itération.
Pour obtenir un shell, nous devons appeler la fonction `system("/bin/sh")`. Selon la convention d'appel x86_64, le premier argument d'une fonction doit être placé dans le registre `RDI`. Le registre `RDI` devra donc contenir **l'adresse mémoire** où est stockée la chaîne `"/bin/sh"`.

Choix de l'emplacement : Le Mot n°28

- L'analyse GDB a montré que le **Mot n°27** correspond à l'adresse de retour (Saved RIP) que nous utilisons pour boucler.

- Le **Mot n°28** se situe juste au-dessus (adresse mémoire supérieure de 8 octets).
- **Justification :** Cet emplacement se trouve dans la stack frame de la fonction appelante (`main`). Contrairement aux variables locales de `makePing` qui sont réinitialisées à chaque appel, cette zone reste stable au fil des itérations de la boucle. C'est une "Safe Zone" idéale pour stocker nos données.

Encodage Little Endian : La mémoire étant adressée en Little Endian, nous devons inverser la chaîne hexadécimale pour qu'elle soit lisible correctement une fois en mémoire.

- Chaîne : `/bin/sh\0`
- Hexadécimal brut : `2f 62 69 6e 2f 73 68 00`
- Entier 64 bits (Little Endian) : `0x0068732f6e69622f`

La chaîne ROP finale, pour être exécutée séquentiellement par le processeur lors des instructions `RET`, doit être organisée ainsi sur la pile (du bas vers le haut) :

Ordre d'exécution	Offset Pile	Contenu	Descrip
1	Saved RIP	`Adresse Gadget`	`POP RDI`
2	RIP + 8	`Adresse de "/bin/sh"`	Argumer
3	RIP + 16	`Adresse de System`	Appel de

Table – Organisation de la ROP Chain en mémoire

Justification de l'écriture en ordre inverse : Nous devons écrire ces éléments en mémoire dans l'ordre inverse de leur position sur la pile (en commençant par RIP+16, puis RIP+8, et enfin RIP).

1. **Maintien de la boucle d'exploitation :** L'adresse de retour actuelle (Saved RIP) contient l'adresse qui permet de relancer la fonction `makePing`. Si nous écrasons cette adresse dès le début par l'adresse du Gadget, la boucle se brise immédiatement.
2. **Prévention des crashs :** Si la boucle se brise et que le programme saute sur le Gadget alors que les arguments (situés plus haut, en RIP+8 et RIP+16) n'ont pas encore été écrits, le processeur dépilera des données invalides (des zéros ou des résidus de mémoire), provoquant un *Segmentation Fault* avant

l'ouverture du shell.

3. **Méthodologie :**
   - **Étape 1 :** Écrire l'adresse de `system` en **RIP+16** (Mot 29). La boucle continue.
   - **Étape 2 :** Écrire l'adresse de la chaîne `"/bin/sh"` en **RIP+8** (Mot 28). La boucle continue.
   - **Étape 3 (Finale) :** Écraser le **Saved RIP** (Mot 27) avec l'adresse du Gadget. La boucle s'arrête, la chaîne ROP s'exécute.

L'exploitation complète nécessite l'écriture de trois éléments distincts sur la pile (Gadget, Argument, Fonction System). Cependant, le buffer d'entrée est trop petit pour injecter la chaîne de format nécessaire à ces trois écritures simultanées.
Pour contourner cette limitation, nous mettons en place 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.
Normalement, à la fin de la fonction `makePing`, l'instruction `ret` récupère l'adresse de retour (Saved RIP) sur la pile pour revenir à la fonction `main` et continuer l'exécution (instruction `mov $0x0,`

%eax).
Pour relancer makePing, nous devons modifier cette adresse de retour pour qu'elle pointe non pas vers l'instruction suivante, mais vers l'instruction d'appel (call) précédente.
D'après notre analyse GDB précédente (Partie 3), voici le désassemblage autour de l'appel :

Listing 4 – Adresses autour de l'appel makePing

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

- **Adresse actuelle sur la pile :** ...54f3 (Retour normal).
- **Adresse cible pour boucler :** ...54ee (Instruction call makePing).

Nous constatons que seule la fin de l'adresse change. La distance est de 5 octets ($0xF3 - 0xEE = 5$).
Pourquoi ne modifier que le dernier octet (LSB) ?

1. **Contournement de PIE/ASLR :** Comme le programme est compilé en mode PIE (*Position Independent Executable*), les adresses complètes (ex : 0x5555...) changent à chaque

exécution. Cependant, les pages mémoire étant alignées sur 4 Ko (0x1000), les 12 bits de poids faible (les 3 derniers chiffres hexadécimaux) restent constants relativement au début de la page.

2. **Stabilité :** L'octet de poids faible de l'instruction `call` sera toujours `0xee`, quelle que soit l'adresse de base choisie par le système.
3. **Mise en œuvre :** En utilisant le spécificateur de format `%hhn`, nous pouvons écraser uniquement le dernier octet de l'adresse de retour sans toucher aux octets supérieurs randomisés qui restent valides.

La stratégie consiste à envoyer plusieurs payloads successifs. À chaque itération (sauf la dernière), nous restaurons la boucle.

1. **Itération 1 (Préparation System) :**
   - Écrire l'adresse de `system` plus haut sur la pile (Offset RIP+16).
   - Écraser le LSB du Saved RIP (actuellement `0xf3`) avec `0xee`.
   - **Résultat :** Au `ret`, le programme saute sur `call makePing`. La fonction redémarre.
2. **Itération 2 (Préparation "/bin/sh") :**

- Écrire l'adresse de la chaîne `"/bin/sh"` sur la pile (Offset RIP+8).
- Écraser à nouveau le LSB du Saved RIP avec `0xee`.
- **Résultat :** La fonction redémarre encore une fois.

3. **Itération 3 (Déclenchement ROP) :**
   - Cette fois, nous n'écrivons pas `0xee`.
   - Nous écrasons le Saved RIP entier (ou son LSB et le suivant) par l'adresse du gadget `pop rdi ; ret` (dont le LSB est `0x63`).
   - **Résultat :** Au `ret`, le programme saute sur le gadget. RDI est chargé avec l'argument, puis `system` est appelé.