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

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

```
thomas@zephyrus:~/work/devoir2026$ ./vuln
Checking ping is on the system...
/usr/bin/ping
Please Insert an IP address to ping:
AAAA %16$p
AAAA 0x41414141
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 :

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

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

```
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

```
(gdb) run
Starting program: /home/thomas/work/devoir2026/vuln
...
Checking ping is on the system...
```

```
/usr/bin/ping
Please Insert an IP address to ping:
AAAA | Off7=%7$p | Off16=%16$p | Off26=%26$p | Off27=%27$p
AAAA | Off7=0x7fffffffdcc8 | Off16=0x4f207c2041414141 | Off26=0x7fffffffdce0 |
    Off27=0x5555555554f3
[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 devrons 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	Adresse (Exemple)	Contenu	Description
%7$p	`0x7fffffffdcc8`	Stack Pointer	Sommet de la pile (RSP)
...	...	...	Padding et variables locales
%16$p	`0x41414141`	**Buffer Start**	**Début de notre payload**
...	...	...	Données utilisateur
%26$p	`0x7fffffffdce0`	**Saved RBP**	Pointeur de base sauvegardé
%27$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 :

```
0x00005555555554ee <+24>: call 0x5555555553b3 <makePing>
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" : L'instruction explicite `pop rdi` n'est pas présente. Cependant, l'instruction `pop r15` (`41 5f`) située à la fin de la fonction d'initialisation permet de la simuler en sautant le premier octet.

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

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

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

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

3.2 Tableau récapitulatif des adresses

Élément	Description	Adresse (Hex)
Référence (Ref)	Adresse de retour (dans main)	`0x5555555554f3`
Cible 1 ($Gadget_{POP}$)	`pop rdi ; ret` (csu_init)	`0x555555555563`
Cible 2 ($System$)	`system@plt`	`0x555555555100`
Cible 3 ($Gadget_{RET}$)	`ret` (fin de main)	`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

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

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

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`	`addr_rip`	`...EE`	Maintien de la boucle
`RBP + 16`	`addr_rbp + 16`	`Gadget POP RDI`	Charge RDI avec la valeur suivante
`RBP + 24`	`addr_rbp + 24`	`Adresse Chaîne`	Argument (Pointeur vers RBP+40)
`RBP + 32`	`addr_rbp + 32`	`Adresse System`	Appel de `system()`
`RBP + 40`	`addr_rbp + 40`	`/bin/sh...`	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 réinvocation (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

```
0x00005555555554ee <+24>: call 0x5555555553b3 <makePing>
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.