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*) :** Nous avons identifié un "Universal Gadget" dans la fonction `__libc_csu_init`.

3.1.1 Analyse du Gadget "POP RDI" dans `__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

```
0x0000555555555562 <+98>: pop %r15
0x0000555555555564 <+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).

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

3.2 Tableau récapitulatif des adresses

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 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)

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

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

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).

3.3.2 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 sera utilisé dans le script d'exploitation pour calculer l'adresse réelle de `system` à partir de la fuite d'adresse (leak).

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 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.

4.1 Placement de la chaîne "/bin/sh"

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`

4.2 Stratégie d'insertion : L'Ordre Inverse

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	Description
1	Saved RIP	`Adresse Gadget`	`POP RDI ; RET`
2	RIP + 8	`Adresse de "/bin/sh"`	Argument dépilé dans RDI
3	RIP + 16	`Adresse de System`	Appel de la fonction

TABLE 3 – 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.

5 Construction de la boucle d'exploitation

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.

5.1 Mécanisme de réinvocation (LSB Overwrite)

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.

5.1.1 Analyse des adresses dans le `main`

D'après notre analyse GDB précédente (Partie 3), voici le désassemblage autour de l'appel :

Listing 4 – Adresses autour de l'appel makePing

```
0x00005555555554ee <+24>: call 0x5555555553b3 <makePing>
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 ($0xF3 - 0xEE = 5$).

5.2 Justification de la technique "Partial Overwrite"

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.

5.3 Algorithme de la boucle d'attaque

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.