

# Exploitation binaire moderne

INF600C

Sécurité des logiciels et exploitation de vulnérabilités

---

Hiver 2026

Université du Québec à Montréal

## Contre-mesures modernes

- Protection de l'espace exécutable (NX, *no-execute*)
- ASLR (*address space layout randomization*)
- & PIE (*position independent executable*)
- Canari de pile (*stack canary*)
- Renforcement de source (FORTIFY\_SOURCE)
- RELRO (*relocation read-only*)

## Contre-mesures modernes

- Protection de l'espace exécutable (NX, *no-execute*)
- ASLR (*address space layout randomization*)
- & PIE (*position independent executable*)
- Canari de pile (*stack canary*)
- Renforcement de source (FORTIFY\_SOURCE)
- RELRO (*relocation read-only*)

```
$ gdb ./pass
```

```
> checksec
```

```
CANARY      : disabled
```

```
FORTIFY     : disabled
```

```
NX          : disabled
```

```
PIE         : disabled
```

```
RELRO      : disabled
```

NX

ret2libc

ROP

ASLR

PIE

**NX**

---

## Bit NX (*no-execute*)

- Fonctionnalité matérielle
- Permet de marquer des pages comme non exécutables
- Existe dans x86\_64 (même en mode 80386)

## Intégré au système d'exploitation

- Depuis Linux 2.6.8 (2004)
- DEP (*data execution prevention*): chez Windows XP SP2 (2004)
- W<sup>X</sup> (*write xor execute*): chez OpenBSD 3.3 (2003)

## Éditeur de liens

- Option `-z execstack` de GNU `ld` (et `gcc`)
- Indique que l'objet requiert quand-même une pile exécutable

```
$ gcc -z execstack $opts pass.c -o pass
$ gcc -z noexecstack $opts pass.c -o pass_nx
```

```
$ ls -l pass pass_nx
-rwxr-xr-x 5888 pass
-rwxr-xr-x 5888 pass_nx
```

```
$ diffoscope pass pass_nx
- GNU_STACK 0x00000000 0x000000 0x000000 RWE 0x10
+ GNU_STACK 0x00000000 0x000000 0x000000 RW 0x10
```

Un processus a

- Différentes zones mémoire avec
- Une origine différente
- Des droits différents: lire, écrire, exécuter ( $rwx$ )  
exécuter = avoir compteur ordinal dessus

Segments simplistes

- Code machine ( $r-x$ )
- Données globales en lecture seule ( $r--$ )
- Données globales en lecture écriture ( $r-w$ )
- Tas ( $rw-$ ) qui croit vers le bas
- Espace libre
- Pile ( $rw-$ ) qui croit vers le haut

# Organisation de la mémoire — pass

```
$ cat /proc/`pgrep pass`/maps
```

address	perms	offset	dev	inode	pathname
08048000-08049000	r-xp	00000000	08:07	1984644	pass
08049000-0804a000	rxp	00000000	08:07	1984644	pass
0804a000-0806b000	rxp	00000000	00:00	0	[heap]
f7dde000-f7f8f000	r-xp	00000000	08:07	8519781	libc-2.24.so
f7f8f000-f7f91000	r-xp	001b0000	08:07	8519781	libc-2.24.so
f7f91000-f7f92000	rxp	001b2000	08:07	8519781	libc-2.24.so
f7f92000-f7f95000	rxp	00000000	00:00	0	
f7fd1000-f7fd4000	rxp	00000000	00:00	0	
f7fd4000-f7fd7000	r--p	00000000	00:00	0	[vvar]
f7fd7000-f7fd9000	r-xp	00000000	00:00	0	[vdso]
f7fd9000-f7ffc000	r-xp	00000000	08:07	8519721	ld-2.24.so
f7ffc000-f7ffd000	r-xp	00022000	08:07	8519721	ld-2.24.so
f7ffd000-f7ffe000	rxp	00023000	08:07	8519721	ld-2.24.so
fffdd000-ffffe000	rxp	00000000	00:00	0	[stack]

## Organisation de la mémoire — pass\_nx

```
$ cat /proc/`pgrep pass_nx`/maps
```

address	perms	offset	dev	inode	pathname
08048000-08049000	r-xp	00000000	08:07	1984676	pass_nx
08049000-0804a000	rw-p	00000000	08:07	1984676	pass_nx
0804a000-0806b000	rw-p	00000000	00:00	0	[heap]
f7dde000-f7f8f000	r-xp	00000000	08:07	8519781	libc-2.24.so
f7f8f000-f7f91000	r--p	001b0000	08:07	8519781	libc-2.24.so
f7f91000-f7f92000	rw-p	001b2000	08:07	8519781	libc-2.24.so
f7f92000-f7f95000	rw-p	00000000	00:00	0	
f7fd1000-f7fd4000	rw-p	00000000	00:00	0	
f7fd4000-f7fd7000	r--p	00000000	00:00	0	[vvar]
f7fd7000-f7fd9000	r-xp	00000000	00:00	0	[vdso]
f7fd9000-f7ffc000	r-xp	00000000	08:07	8519721	ld-2.24.so
f7ffc000-f7ffd000	r--p	00022000	08:07	8519721	ld-2.24.so
f7ffd000-f7ffe000	rw-p	00023000	08:07	8519721	ld-2.24.so
fffdd000-ffffe000	rw-p	00000000	00:00	0	[stack]

**ret2libc**

---

Objectifs: flags 1, 2, 3 et 4

Objectifs: flags 1, 2, 3 et 4

- flags 1, 2, 3: rien ne change. NX ne protège rien
  - flag 4: erreur de segmentation
- Dénis de service :) mais pas de flag :(

Objectifs: flags 1, 2, 3 et 4

- flags 1, 2, 3: rien ne change. NX ne protège rien
- flag 4: erreur de segmentation

→ Dénis de service :) mais pas de flag :(

Objectif: avoir le flag 4 quand même

## L'attaquant

- Ne peut injecter du nouveau code machine
- Contrôle quand même l'adresse de retour du `ret`
- Peut brancher sur code existant (qui a les droits x)

## De flag3 à flag4

- Plutôt que `ret` sur `getflag3`
- Pourquoi ne pas `ret` sur `system` dans la `libc`?

- Identifier l'adresse de `system`  
Merci gdb: `p system`
- Identifier une chaîne `"/bin/sh"`  
Merci pwndbg: `search -t string "/bin/sh\0"`
- Bricoler la pile  
Pour que `system` s'exécute comme il faut...

- Identifier l'adresse de `system`  
Merci gdb: `p system`
- Identifier une chaîne `"/bin/sh"`  
Merci pwndbg: `search -t string "/bin/sh\0"`
- Bricoler la pile  
Pour que `system` s'exécute comme il faut...  
Mais comment?

## *Application Binary Interface (ABI)*

Description bas niveau pour interconnecter

- Un programme et le système d'exploitation
- Un programme et des bibliothèques
- Les composantes d'un programme

Quoi spécifier?

- Représentation des données: taille, agencements, alignements
- Convention d'appel
- Nommage des entités (C++)
- Exceptions (C++)
- Format des fichiers binaires (exe, elf, etc.)

Qui s'en soucie?

- Les compilateurs (et cie) principalement
  - Les noyaux des système d'exploitation pour les appels système
  - Les distributions
- pour assurer la cohérence et la compatibilité des paquets

## Bibliothèques vs programmes clients

- Au niveau machine: `call` et `ret` gèrent le flot de contrôle
- Le reste se fait par convention

## Quoi conventionner?

- Comment passer les arguments?
- Dans quelle ordre?
- Sous quelle forme?
- Quel alignement?
- Comment récupérer les résultats?
- Qui restaure la pile?
- Quel est le rôle des registres?
- Quels registres doivent être sauvegardés?

ABI System V, supplement IA32

- 1996, Version officielle
- 2005, Version avec mises à jour

En bref pour les cas simples:

- On empile les arguments en ordre inverse  
Le premier argument est empilé en dernier
- Le retour est dans %eax
- Mais il y a des subtilités dans bien des cas  
flottants, structures, truc plus grands que 32 bits, varargs, etc.

`system` s'attend à

- `%esp` pointe sur l'adresse de retour  
On s'en fiche!
- `%esp+4` pointe sur l'argument = l'adresse de `"/bin/sh"`

Payload

- "A" \* décalage
- Adresse de `system`: écrase le *retaddr* original
- "BBBB": retour de `system`, on s'en fiche
- Adresse de `"/bin/sh"`: l'argument de `system`

API System V, supplément AMD64

- 2018, Version 1.0

En bref pour les cas simples

- Les 6 premiers arguments vont dans:  
%rdi, %rsi, %rdx, %rcx, %r8, %r9
- Les autres arguments sont empilés
- Le retour revient dans %rax
- Mais il y a des subtilités dans bien des cas

**ROP**

---

Objectif: le flag 4

Objectif: le flag 4

Problème

- *shellcode* dans la pile?

Difficile, car on est NX

- *ret2libc* vers `system`?

Difficile, car l'argument doit être dans `%rdi`

# Return Oriented Programming (ROP)

- L'argument de `system` va dans `%rdi`
- Comment mettre une valeur dans `%rdi`?
- Y a-t-il un morceau de code qui le fait pour nous?

## Idée du ROP

- Trouver des morceaux de code utile qui terminent par `ret`  
On appelle ça des **gadgets**
- Mettre dans la pile les adresses des gadgets dans l'ordre  
On appelle ça une **ROP chain**

## À la main dans le binaire et les bibliothèques

- `objdump -d | grep`
- `ghex`

À la main dans le binaire et les bibliothèques

- `objdump -d | grep`
- `ghex`

Avec `pwndbg`

```
$ gdb ./pass_64_nx
> start
> rop --grep 'pop rdi ; ret'
0x0040129b : pop rdi ; ret
```

Avec d'autres outils:

- `ropgadget`
- `ropper`
- <http://ropshell.com/>

# ROP: mise en œuvre

- Identifier l'adresse de `system`  
Merci gdb: `p system`
- Identifier une chaîne `"/bin/sh"`  
Merci peda: `searchmem "/bin/sh\0"`
- Identifier un gadget `{pop rdi; ret}`  
Merci peda: `ropsearch "pop rdi; ret"`

## Payload

- "A" \* décalage
- Adresse de `{pop rdi; ret}`
- Adresse de `"/bin/sh"`
- Adresse de `system`

**ASLR**

---

Contre-mesure du **système d'exploitation** 

- Pas besoin de recompiler (modulo PIE, c.f. plus loin)
- Linux: depuis 2.6.12 (2005)
- Windows: depuis Vista (2007)

Réduit l'exploitabilité des bugs

- Le bug de programmation est toujours là
- Toujours un comportement indéterminé
- Mais **seulement** une erreur de segmentation

→ Mieux vaut *segv* que *pwn*

Le noyau positionne **aléatoirement** les zones mémoires

- La pile
- Les projections mémoire (`mmap`)
- Les bibliothèques dynamiques (en fait c'est `mmap` aussi)
- Le tas
- etc.

À **chaque exécution** du programme

Prévient les attaques qui nécessitent une adresse précise

- shellcode: l'adresse de la pile
- ret2libc: les adresses des fonctions de bibliothèques
- ROP: les adresses des gadgets

→ Même si l'attaquant contrôle `%ip`, il ne sait pas où pointer

On active ASLR (ou plutôt on le réactive)

```
$ sudo sysctl kernel/randomize_va_space=2 # Voir proc(5)
```

```
$ cat /proc/`pgrep pass`/maps
```

```
08048000-08049000 r-xp 00000000 08:07 1984785 pass_nx
08049000-0804a000 rw-p 00000000 08:07 1984785 pass_nx
09bea000-09c0c000 rw-p 00000000 00:00 0 [heap]
f7d34000-f7f06000 r-xp 00000000 08:07 8520035 libc-2.27.so
f7f06000-f7f07000 ---p 001d2000 08:07 8520035 libc-2.27.so
f7f07000-f7f09000 r--p 001d2000 08:07 8520035 libc-2.27.so
f7f09000-f7f0a000 rw-p 001d4000 08:07 8520035 libc-2.27.so
f7f0a000-f7f0d000 rw-p 00000000 00:00 0
f7f49000-f7f4b000 rw-p 00000000 00:00 0
f7f4b000-f7f4e000 r--p 00000000 00:00 0 [vvar]
f7f4e000-f7f50000 r-xp 00000000 00:00 0 [vdso]
f7f50000-f7f76000 r-xp 00000000 08:07 8519959 ld-2.27.so
f7f76000-f7f77000 r--p 00025000 08:07 8519959 ld-2.27.so
f7f77000-f7f78000 rw-p 00026000 08:07 8519959 ld-2.27.so
ff92d000-ff94e000 rw-p 00000000 00:00 0 [stack]
```

Objectifs: avoir les 4 flags de `pass_nx` (32 bits)

Objectifs: avoir les 4 flags de `pass_nx` (32 bits)

- Flag 1 & 2: ASLR change rien

Objectifs: avoir les 4 flags de `pass_nx` (32 bits)

- Flag 1 & 2: ASLR change rien
- Flag 3: ça passe aussi !

La zone du programme n'a pas bougé en RAM

Toujours 0x08048000–0x0804a000

Objectifs: avoir les 4 flags de `pass_nx` (32 bits)

- Flag 1 & 2: ASLR change rien
- Flag 3: ça passe aussi !

La zone du programme n'a pas bougé en RAM

Toujours 0x08048000–0x0804a000

- Flag 4: `ret2libc segfault...` logique, on sait pas où est la `libc`

3 flags sur 4, c'est pas si mal!

Objectifs: avoir les 4 flags de `pass_nx` (32 bits)

- Flag 1 & 2: ASLR change rien
- Flag 3: ça passe aussi !

La zone du programme n'a pas bougé en RAM

Toujours 0x08048000–0x0804a000

- Flag 4: `ret2libc segfault...` logique, on sait pas où est la `libc`

3 flags sur 4, c'est pas si mal!

Nouvel objectif: avoir le 4ème flag quand même

## Jeu de hasard: vaincre par la force brute

ASLR ne casualise que certains octets

- Le décalage dans la page (0x1000) est constant:  
octets de poids faible
- Les octets de poids forts sont invariables

→ seuls les octets du milieu sont aléatoires

En 32 bits

- Début de la libc entre 0xf7c00000 à 0xf7dff000
- Il faut juste **deviner** les bits du milieu
- Entropie: 9 bits, soit un chance sur  $2^9$ , soit 0,2%
- Un essai par 0,01s

50% de succès après 4s

85% de succès après 10s

→ C'est très efficace !!!

## pass\_nx: mise en œuvre pour contourner ASLR

- gdb désactive par défaut ASLR
- Mais les adresses données sont une **possibilité** valide avec ASLR activé
- On détermine les adresse de `system` de `"/bin/sh"` et du gadget

### Payload

- Le même que pour ROP
- Mais on teste jusqu'à ce que ça marche  
Car on a calculé les probabilités avant!
- Attention aux tampons: on ajoute un `sleep(1)` pour éviter que le `read(2)` de `fgets(3)` mange aussi la commande shell

## Jeu de hasard: vaincre 64 bits?

En 64 bits

- Début de la libc entre 0x7f0000000000 et 0x7fffffff0000
  - Entropie: 32 bits, soit un chance sur  $2^{32}$ , soit 0,000 000 02%
  - Un essai par 0,01s  
0,02% de succès après 1 jour  
50% de succès après 1 an
- Ce n'est pas très efficace.

L'attaquant n'écrase que les octets de poids faible

- Permet de viser une information dans la même zone mémoire
- Sans connaître l'adresse exacte
- Quitte à attaquer quelques bits par force brute

ASLR ne change que l'adresse de départ des zones mémoires

- Les fichiers sont chargés à cette adresse de départ par le chargeur et éditeur de liens dynamique  $\tau_d(8)$
- Les adresses relatives sont déjà prédéterminées par l'éditeur de liens statique  $\tau_d(1)$

L'attaquant qui obtient un pointeur lors de l'exécution

- Connaît l'adresse précise de l'information pointée
- Peut déterminer l'adresse des autres informations dans la zone

Mais ça nécessite une **attaque dynamique**

Car les adresses changent à chaque exécution

**PIE**

---

Le programme principal n'est pas casualisé!

`ret2text` fonctionne toujours

- L'attaquant a eu `flag3`!
- L'attaquant peut ROPer avec des gadgets de l'exécutable  
Il y en a des biens dans `__libc_csu_init`
- L'attaquant a accès à des trucs cool avancés:  
GOT, PLT, `_dl_resolve`, etc.

Pourquoi pas ASLR aussi l'exécutable lui-même?

## Espace d'adressage et exécutables

- Dans un ELF **exécutable** standard (programme principal)
- Les adresses locales sont **déterminées** (par l'éditeur de liens)
- Avec des valeurs **fixes** dans le code machine

```
$ file pass_64
```

```
pass: ELF 64-bit LSB executable, dynamically linked,  
       interpreter /lib/ld-linux.so.2
```

```
$ objdump -d -Mintel pass_64
```

```
00000000004006b2 <getflag2>: ...  
    4006b6: bf 67 08 40 00    mov     edi,0x400867  
    4006bb: e8 20 fe ff ff   call   4004e0 <puts@plt>  
    ...  
00000000004006d4 <main>: ...  
    4006f5: e8 b8 ff ff ff   call   4006b2 <getflag2>  
    ...
```

## Position-independent code (PIC)

- Dans une bibliothèque partagée dynamique
- On ne peut pas utiliser des adresses fixes dans le code machine
- On doit générer du **code** machine **indépendant** de la **position**

### Technique PIC simple

- Même si l'adresse absolue est **inconnue**,
- Les adresses relatives sont **prédéterminables**
- On utilise `%ip` pour localiser le reste des choses

```
$ gcc -fpic pass.c pass_64_pic $opts
```

```
$ objdump -d -Intel pass_64_pic
```


```
00000000004006f1 <getflag2>: ...
```

```
4006f5: 488d3dbb010000 lea    rdi,[rip+0x1bb]
```


```
4006fc: e80ffeffff      call   400510 <puts@plt>
```

```
...
```

## *Position-independent executable (PIE)*

Le compilateur (`gcc`) 

- Doit générer du code machine indépendant de la position
- Pour exécutable (un peu différent du PIC de bibliothèque)
- `gcc -fpie`

L'éditeur de lien (`ld`) 

- Doit fabriquer un exécutable chargeable n'importe où
- Sans adresse de départ fixe
- `ld -pie` (et `gcc -pie`)

Le chargeur (noyau) et chargeur dynamique (`ld.so`) 

- Doivent adapter le chargement pour profiter d'ASLR

## Options de gcc(1)

- `-fpic` Génère de l'assembleur pic pour bibliothèques
- `-fPIC` Sur certaines architectures `-fpic` ne fonctionne pas toujours. `-fPIC` fonctionne toujours mais peut générer du code plus gros.
- `-fpie` Génère de l'assembleur pic pour exécutable
- `-fPIE` Même remarque que pour `-fPIC`

La différence `-fpie` / `-fpic` concerne les appels internes de symboles exposés: fonctions et variables globales

- `-fpie` ces références sont directes
- `-fpic` ces références sont résolues dynamiquement
- Voir [`ld.so\(8\)`](http://ld.so(8)) pour les détails

## Options de `ld(1)` (et `gcc(1)`)

- `-no-pie`: Fabrique un exécutable normal
- `-pie`: Fabrique un exécutable PIE (*shared object*)
- `-shared`: Fabrique une bibliothèque dynamique (*shared object*)

## Attention

Il faut combiner les options pour éviter les mauvaises surprises

- Exécutable PIE

```
gcc -pie -fpie prog.c -o prog
```

C'est le comportement par défaut dans les `gcc` modernes

- Bibliothèque

```
gcc -shared -fpic prog.c -o prog.so
```

- Exécutable non-PIE

```
gcc -no-pie -fno-pie prog.c -o prog
```

## Contre-mesures modernes

- Attaque: shellcode

→ défense NX

- Attaque: ret2libc et ROP

→ défense ASLR (et PIE)

## Les contourner c'est

- souvent de la bidouille
- souvent technique
- souvent spécifique

Il y a encore encore plein de choses à voir

- des classes de fragilités
- des types de vulnérabilités
- des styles d'attaques
- et des contre-mesures

```
$ gdb pass_pie
```

```
> checksec
```

```
CANARY      : disabled
```

```
FORTIFY     : disabled
```

```
NX          : ENABLED
```

```
PIE         : ENABLED
```

```
RELRO      : disabled
```