

Debugging Système et Noyau

Aurélien Cedeyn

École Nationale Supérieure d'Informatique pour l'Industrie et l'Entreprise

2017-2018

Sommaire

1 Debugger

- Définitions
 - Les différents type d'analyse
 - Les différents univers
 - Ce que couvre ce cours

Définitions

Définition

To search for and eliminate malfunctioning elements or errors in

- Quand doit-on debugger ?
 - Une application ne se comporte pas comme elle le devrait
 - Les performances du système ne correspondent pas aux attentes
 - Problèmes de sécurité
 - Plantage du système
- Quels outils ?
 - Pour chaque élément du système à observer, il faut choisir le ou les outils les mieux adaptés à la situation.
 - Certains outils sont orientés utilisateur, d'autre système, certains autres font la jonction entre les deux univers.
- Que chercher ?
- Se poser des questions et faire preuve d'imagination

Définitions

Définition

To search for and eliminate malfunctioning elements or errors in

- Quand doit-on debugger ?
 - Une application ne se comporte pas comme elle le devrait
 - Les performances du système ne correspondent pas aux attentes
 - Problèmes de sécurité
 - Plantage du système
 - Quels outils ?
 - Pour chaque élément du système à observer, il faut choisir le ou les outils les mieux adaptés à la situation.
 - Certains outils sont orientés utilisateur, d'autre système, certains autres font la jonction entre les deux univers.
 - Que chercher ?
 - **Se poser des questions et faire preuve d'imagination**

Les différents type d'analyse

- Activer, lire et comprendre les messages de log
 - Tracer un processus
 - Récupérer la pile d'exécution d'un processus
 - Analyse la mémoire d'un processus

Les différents univers

Définition

Au sein du système d'exploitation, on peut distinguer trois univers différents :

- L'espace utilisateur : tout ce qui est lancé par un utilisateur
 - L'espace noyau : l'ensemble des composants internes du système d'exploitation (les drivers, le noyau, les modules...)
 - Le matériel : les périphériques, la mémoire, le CPU...

Les différents univers

UserSpace et KernelSpace

Les différents univers

UserSpace

Terminal / Window Manager

Systemd / Init

Code utilisateur

Librairie GNU C (libc)

Les différents univers

UserSpace

Terminal / Window Manager
Systemd / Init
Code utilisateur

Librairie GNU C (libc)

KernelSpace

Appels systèmes (syscall)

Modules et drivers noyaux

L'espace utilisateur accÈde aux fonctionnalitÈs du noyau exclusivement via les appels systÈmes (syscall).

Les différents univers

UserSpace

Terminal / Window Manager

Systemd / Init

Code utilisateur

Librairie GNU C (libc)

KernelSpace

Appels systèmes (syscall)

Services noyaux

Modules et drivers noyaux

Hardware

CPU

Mémoire

Périphériques

Via ses drivers, le noyau accède au matériel et les présente à l'espace utilisateur.

Ce que couvre ce cours

Ce que couvre ce cours

- Appréhender la construction d'un fichier binaire et comprendre les mécanismes en jeu lors de son exécution.

Ce que couvre ce cours

Ce que couvre ce cours

- Appréhender la construction d'un fichier binaire et comprendre les mécanismes en jeux lors de son exécution.
 - Explorer les principaux outils disponibles permettant d'approcher le debugging des processus en espace utilisateur.

Ce que couvre ce cours

Ce que couvre ce cours

- Appréhender la construction d'un fichier binaire et comprendre les mécanismes en jeu lors de son exécution.
 - Explorer les principaux outils disponibles permettant d'approcher le debugging des processus en espace utilisateur.
 - S'initier à la mise en place de système de debugging en espace noyau.

Sommaire

1 Debugger

2 Les instructions

- Introduction
- La compilation
- L'exécution

3 Processus utilisateur

4 Noyau

5 Question

Introduction

Et si on commençait doucement ?

Avant tout, nous avons besoin de comprendre
un minimum d'instructions Assembleur x86 !

Introduction

Et si on commençait doucement ?



Introduction

Et si on commençait doucement ?

Les registres

%rax, %rbx, %rcx, %rdx, %rdi, %rsi, %r[8-15]

- **%rax** : Accumulateur, sert à effectuer des calculs arithmétiques ou à envoyer un paramètre à une interruption.
 - **%rbx** : Registre auxiliaire de base, sert à effectuer des calculs arithmétiques ou bien des calculs sur les adresses.
 - **%rcx** Registre auxiliaire (compteur), sert généralement comme compteur dans des boucles.
 - **%rdx** Registre auxiliaire de données, sert à stocker des données destinées à des fonctions.
 - **%rdi** Registre contenant un index de destination : utilisée comme adresse source pour les copies de données
 - **%rsi** Registre contenant un index de source : utilisé comme adresse source pour les copies de données
 - **%r[8-15]** Registres complémentaires

rax (64bits)		eax (32bits)		ax (16 bits)	
				ah (8bits)	al (8bits)

Introduction

Et si on commençait doucement ?

Les instructions

mov

mov src, dest

- copie la source (*src*) vers la destination (*dest*)
- l'instruction peut-être suffixée par (*q,l,w,s,b*) et correspond au nombre de bits à copier (ex : *l - long (4 octets)*)

syscall

syscall

- Instruction d'exécution d'une interruption logicielle.

Les instructions

Des sources au fichier binaire

Use the source Luke.

```
$ vim add.c
```

Le code source - langage C

```
void main(void){  
    exit(2);  
}
```

```
$ vim add.s
```

Le code source - Assembleur x86_64

```
.text  
.globl _start  
  
start:  
    mov $60, %rax  
    mov $2, %rdi  
    syscall
```

Plus d'informations sur l'interface avec les appels systèmes et Linux en assembleur :

https://en.wikibooks.org/wiki/X86_Assembly/Interfacing_with_Linux

Les instructions

Des sources au fichier binaire

Use the source Luke.

```
$ vim add.c
```

Le code source - langage C

```
void main(void){  
    exit(2);  
}
```

```
$ vim add.s
```

Le code source - Assembleur x86_64

```
.text  
.globl _start  
  
.start:  
    mov $60, %rax  
    mov $2, %rdi  
    syscall
```

- main : fonction principale du programme
- exit : appel système

Plus d'informations sur l'interface avec les appels systèmes et Linux en assembleur :

https://en.wikibooks.org/wiki/X86_Assembly/Interfacing_with_Linux

Les instructions

Des sources au fichier binaire

Use the source Luke.

```
$ vim add.c
```

Le code source - langage C

```
void main(void){  
    exit(2);  
}
```

```
$ vim add.s
```

Le code source - Assembleur x86_64

```
.text  
.globl _start  
  
_start:  
    mov $60, %rax  
    mov $2, %rdi  
    syscall
```

- mov \$60, rax : numéro de l'appel système
- mov \$2, rdi : paramètre de l'appel système
- syscall : appel système

Plus d'informations sur l'interface avec les appels systèmes et Linux en assembleur :

https://en.wikibooks.org/wiki/X86_Assembly/Interfacing_with_Linux

Les instructions

Des sources au fichier binaire

Use the source Luke.

```
$ vim add.c
```

Le code source - langage C

```
void main(void){  
    exit(2);  
}
```

```
$ vim add.s
```

Le code source - Assembleur x86_32

```
.text  
.globl _start  
  
_start:  
    movl $1, %eax  
    movl $2, %ebx  
    int $0x80
```

- mov \$1, eax : numéro de l'appel système
- mov \$2, ebx : paramètre de l'appel système
- int \$0x80 : appel système

Plus d'informations sur l'interface avec les appels systèmes et Linux en assembleur :

https://en.wikibooks.org/wiki/X86_Assembly/Interfacing_with_Linux

Les instructions

Le compilateur se charge de transformer le code source en code objet binaire.

```
$ as add.s -o add.o
```

Le code machine binaire - Compilateur (gcc, as)

```
add.o: file format elf64-x86-64

Sections:
Idx Name      Size    VMA          LMA          File off  Algn
 0 .text      00000010  0000000000000000  0000000000000000  00000040  2**0
               CONTENTS, ALLOC, LOAD, READONLY, CODE
 1 .data      00000000  0000000000000000  0000000000000000  00000050  2**0
               CONTENTS, ALLOC, LOAD, DATA
 2 .bss      00000000  0000000000000000  0000000000000000  00000050  2**0
               ALLOC

Contents of section .text:
0000 48c7c03c 00000048 c7c70200 00000f05 H...<...H.......
```

Disassembly of section .text:

```
0000000000000000 <.text>:
 0: 48 c7 c0 3c 00 00 00          mov    $0x3c,%rax
 7: 48 c7 c7 02 00 00 00          mov    $0x2,%rdi
 e: 0f 05                          syscall
```

Les instructions

Le linker se charge de transformer l'objet binaire obtenu en code exécutable.

```
$ ld add.o -o add
```

L'exécutable - Linker (ld)

```

add:      file format elf64-x86-64

Sections:
Idx Name      Size    VMA          LMA          File off  Alg
0 .text      00000010  0000000000400078  0000000000400078  00000078  2**0
          CONTENTS, ALLOC, LOAD, READONLY, CODE

Contents of section .text:
400078 48c7c03c 00000048 c7c70200 00000f05 H..<...H.....
Disassembly of section .text:
0000000000400078 <.text>:
400078: 48 c7 c0 3c 00 00 00      mov    $0x3c,%rax
40007f: 48 c7 c7 02 00 00 00      mov    $0x2,%rdi
400086: 0f 05                  syscall

```

Les instructions

Des sources au fichier binaire

Définition

Une instruction est simplement un ensemble d'octets transmis au processeur.

Les instructions

Des sources au fichier binaire

Définition

Une instruction est simplement un ensemble d'octets transmis au processeur.

Instruction x86_64 (big indian)

```
add:    file format elf64-x86-64

Sections:
Idx Name      Size    VMA          LMA          File off  Align
 0 .text      00000010  0000000000400078  0000000000400078  00000078  2**0
          CONTENTS, ALLOC, LOAD, READONLY, CODE
Contents of section .text:
400078 48c7c03c 00000048 c7c70200 00000f05 H...H.....
Disassembly of section .text:
0000000000400078 :.text:
400078: 48 c7 c0 3c 00 00 00      mov    $0x3c,%rax
40007f: 48 c7 c7 02 00 00 00      mov    $0x2,%rdi
400086: 0f 05                   syscall
```

Les instructions

Des sources au fichier binaire

Définition

Une instruction est simplement un ensemble d'octets transmis au processeur.

Instructions ARM (little indian)

```
add.arm:      file format elf32-littlearm

Sections:
Idx Name      Size    VMA      LMA      File off  Align
 0 .text      00000010 00008054 00008054 00000054 2**2
              CONTENTS, ALLOC, LOAD, READONLY, CODE
 1 .ARM.attributes 00000014 00000000 00000000 00000064 2**0
              CONTENTS, READONLY
```

Contents of section .text:
8054 0200a0e3 0170a0e3 04002de5 000000efp....=....

Contents of section .ARM.attributes:
0000 41130000 00616561 62690001 09000000 A....aeabi.....
0010 06010801
.....

Disassembly of section .text:

```
00008054 <_start>:
8054:   e3a00002      mov      r0, #2
8058:   e3a07001      mov      r7, #1
805c:   e52d0004      push     [r0]          ; (str r0, [sp, #-4]!)
8060:   ef000000      svc      0x00000000
```

Les instructions

Des sources au fichier binaire

Définition

Une instruction est simplement un ensemble d'octets transmis au processeur.

Lecture du fichier binaire

- `$ objdump -dsh add.o`
- `$ xxd add.o`

Les instructions

ELF¹⁰¹ a Linux executable walkthrough

DISSECTED FILE

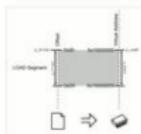


LOADING PROCESS

1 HEADER
THE ELF HEADER IS PARSED
THE PROGRAM HEADER IS PARSED
[SECTIONS ARE NOT USED]

2 MAPPING

3 EXECUTION
ENTRY IS CALLED
SYSCALLS ARE ACCESSED VIA:
- SYSCALL NUMBER IN THE RAX REGISTER
- CALLING INSTRUCTION SYSCALL



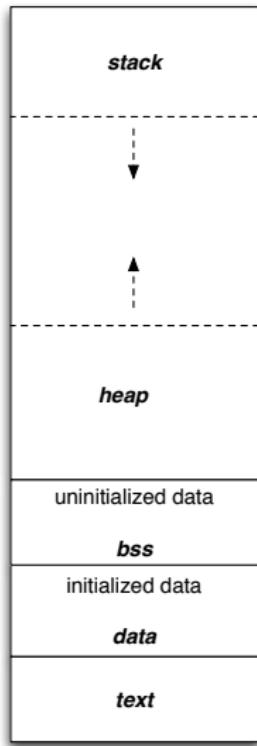
TRIVIA

THE ELF WAS FIRST SPECIFIED BY US, L AND UI FOR UNIX SYSTEM V IN 1989.

- THE ELF IS USED, AMONG OTHERS, IN:
 - LINUX, ANDROID, *BSD, SOLARIS, BEOS
 - PSP, PLAYSTATION 2-4, DREAMCAST, GAMECUBE, WII
 - VARIOUS OSES MADE BY SAMSUNG, ERICSSON, NOKIA,
 - MICROCONTROLLERS FROM ATMEL, TEXAS INSTRUMENTS

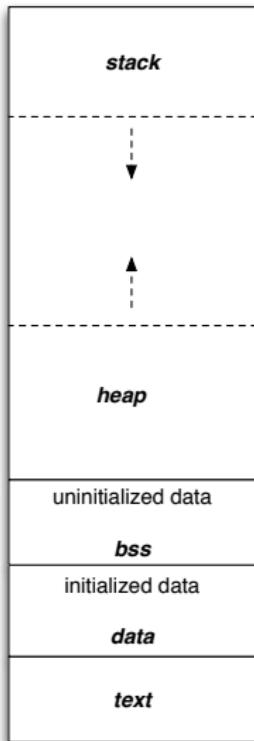
L'exécution

Mise en place en mémoire



L'exécution

Mise en place en mémoire

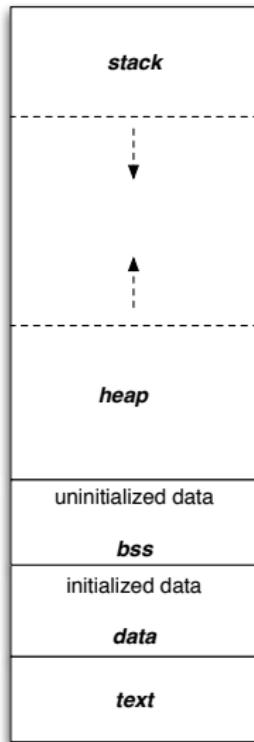


text (fixé à la compilation)

Appelée aussi *code section* : ensemble du code exécutable

L'exécution

Mise en place en mémoire



data (fixé à la compilation)

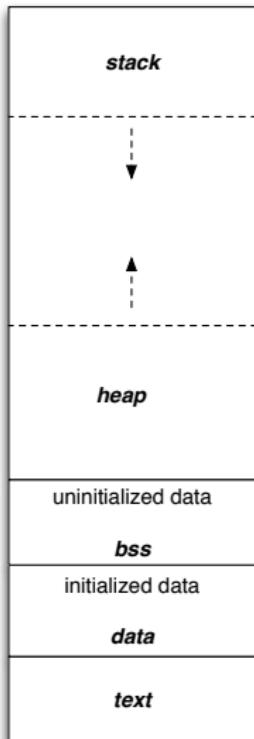
Variables globales initialisées du code.

text (fixé à la compilation)

Appelée aussi *code section* : ensemble du code exécutable.

L'exécution

Mise en place en mémoire



bss (fixé à la compilation)

Block Started by Symbol : Variables globales non initialisées du code.

data (fixé à la compilation)

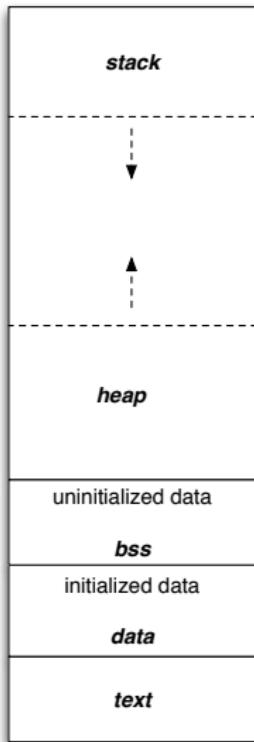
Variables globales initialisées du code

text (fixé à la compilation)

Appelée aussi *code section* : ensemble du code exécutable

L'exécution

Mise en place en mémoire



heap (allouée à l'exécution)

Le **tas** : section mémoire utilisée pour allouer les variables dynamiques.

bss (fixé à la compilation)

Block Started by Symbol : Variables globales non initialisées du code.

data (fixé à la compilation)

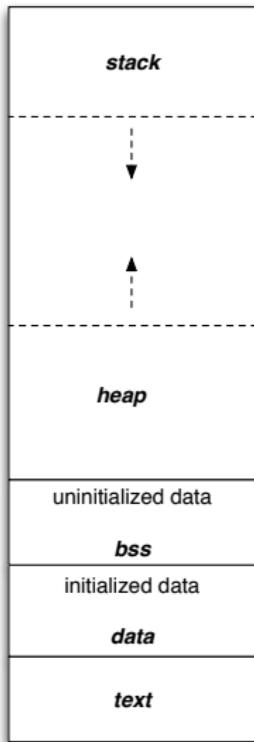
Variables globales initialisées du code.

text (fixé à la compilation)

Appelée aussi *code section* : ensemble du code exécutable.

L'exécution

Mise en place en mémoire



stack (allouée à l'exécution)

La *pile* contient la pile d'appel des fonctions, leur variables locales et les adresses de fonctions externes.

heap (allouée à l'exécution)

Le *tas* : section mémoire utilisée pour allouer les variables dynamiques.

bss (fixé à la compilation)

Block Started by Symbol : Variables globales non initialisées du code.

data (fixé à la compilation)

Variables globales initialisées du code.

text (fixé à la compilation)

Appelée aussi *code section* : ensemble du code exécutable.

L'exécution

Le déplacement dans la pile

À partir de ce moment nous avons un exécutable avec son code qui est monté en mémoire à l'exécution.

C'est sur la pile que va se jouer l'exécution des différentes fonctions ainsi que le passage de leurs paramètres et de leurs valeurs de retour.

Les registres

`%rbp`

Base Pointer : Adresse de base d'appel de la fonction

`%rsp`

Stack Pointer : Adresse courante de la pile (*stack*)

`%rip`

Instruction Pointer : Adresse de la prochaine instruction

L'exécution

Le déplacement dans la pile

À partir de ce moment nous avons un exécutable avec son code qui est monté en mémoire à l'exécution.

C'est sur la pile que va se jouer l'exécution des différentes fonctions ainsi que le passage de leurs paramètres et de leurs valeurs de retour.

Les instructions

push

Pousse sur la *stack* le contenu du registre en paramètre.

pop

Récupère la dernière valeur de la *stack* et place son contenu dans le registre passé en paramètre.

L'exécution

Le déplacement dans la pile

À partir de ce moment nous avons un exécutable avec son code qui est monté en mémoire à l'exécution.

C'est sur la pile que va se jouer l'exécution des différentes fonctions ainsi que le passage de leurs paramètres et de leurs valeurs de retour.

Les instructions

call

Instruction d'appel de fonction. Elle procède en 2 étapes :

- Sauvegarde de %rip sur la stack
- Déplacement de %rip à l'adresse demandée

ret

Instruction de retour de fonction. Restauration de %rip sur la stack

Debugger
ooooo
L'exécution

Les instructions

Processus utilisateur

Noyau
oooooooooooooooooooo

Question 00

Attention, on s'accroche !



L'exécution

Exemple d'exécution dans la stack (pile)

On entre dans la fonction main, %rbp est sauvegardé

%rsp	Stack	%rip	Assembleur - section text
→0xf0	0xff (%rbp)	→ 0x01	push %rbp
0xe0		0x02	mov %rsp,%rbp
0xd0		0x05	callq 0x12 <test>
0xc0		0x09	mov \$0x0,%eax
0xb0		0x0d	retq
0xa0		0x12	push %rbp
		0x13	mov %rsp,%rbp
		0x1f	callq 0x26 <in_test>
		0x24	pop %rbp
		0x25	retq
		0x26	push %rbp
		0x27	mov %rsp,%rbp
		0x2a	nop
		0x2b	pop %rbp
		0x2c	retq

Code C

```
→ void main(void){  
    test();  
}  
void test(void){  
    return in_test();  
}  
void in_test(void){  
    return;  
}
```

L'exécution

Exemple d'exécution dans la stack (pile)

On place l'adresse de base de la stack à sa nouvelle valeur

%rsp	Stack	%rip	Assembleur - section text
0xf0	0xff (%rbp)	0x01	push %rbp
0xe0		→ 0x02	mov %rsp,%rbp
0xd0		0x05	callq 0x12 <test>
0xc0		0x09	mov \$0x0,%eax
0xb0		0x0d	retq
0xa0		0x12	push %rbp
		0x13	mov %rsp,%rbp
		0x1f	callq 0x26 <in_test>
		0x24	pop %rbp
		0x25	retq
		0x26	push %rbp
		0x27	mov %rsp,%rbp
		0x2a	nop
		0x2b	pop %rbp
		0x2c	retq

Code C

```
→ void main(void){  
    test();  
}  
void test(void){  
    return in_test();  
}  
void in_test(void){  
    return ;  
}
```

L'exécution

Exemple d'exécution dans la stack (pile)

Appel de la fonction test, %rip est poussé sur la stack et %rip prend la nouvelle valeur de la fonction test

%rsp	Stack	%rip	Assembleur - section text
0xf0	0xff (%rbp)	0x01	push %rbp
→0xe0	0x09 (main)	0x02	mov %rsp,%rbp
0xd0		→ 0x05	callq 0x12 <test>
0xc0		0x09	mov \$0x0,%eax
0xb0		0x0d	retq
0xa0		0x12	push %rbp
		0x13	mov %rsp,%rbp
		0x1f	callq 0x26 <in_test>
		0x24	pop %rbp
		0x25	retq
		0x26	push %rbp
		0x27	mov %rsp,%rbp
		0x2a	nop
		0x2b	pop %rbp
		0x2c	retq

```
Code C
void main(void){
    → test();
}
void test(void){
    return in_test();
}
void in_test(void){
    return ;
}
```

L'exécution

Exemple d'exécution dans la stack (pile)

Entrée dans la fonction test, et même principe que pour la fonction main

%rsp	Stack	%rip	Assembleur - section text
0xf0	0xff (%rbp)	0x01	push %rbp
0xe0	0x09 (main)	0x02	mov %rsp,%rbp
→0xd0	0xe0 (%rbp)	0x05	callq 0x12 <test>
0xc0		0x09	mov \$0x0,%eax
0xb0		0x0d	retq
0xa0		→ 0x12	push %rbp
		0x13	mov %rsp,%rbp
		0x1f	callq 0x26 <in_test>
		0x24	pop %rbp
		0x25	retq
		0x26	push %rbp
		0x27	mov %rsp,%rbp
		0x2a	nop
		0x2b	pop %rbp
		0x2c	retq

Code C

```
void main(void){  
    test();  
}
```

```
→ void test(void){  
    return in_test();  
}
```

```
void in_test(void){  
    return;  
}
```

L'exécution

Exemple d'exécution dans la stack (pile)

Entrée dans la fonction test, et même principe que pour la fonction main

%rsp	Stack	%rip	Assembleur - section text
0xf0	0xff (%rbp)	0x01	push %rbp
0xe0	0x09 (main)	0x02	mov %rsp,%rbp
0xd0	0xe0 (%rbp)	0x05	callq 0x12 <test>
0xc0		0x09	mov \$0x0,%eax
0xb0		0x0d	retq
0xa0		0x12	push %rbp
		→ 0x13	mov %rsp,%rbp
		0x1f	callq 0x26 <in_test>
		0x24	pop %rbp
		0x25	retq
		0x26	push %rbp
		0x27	mov %rsp,%rbp
		0x2a	nop
		0x2b	pop %rbp
		0x2c	retq

```
Code C
void main(void){
    → test();
}
→ void test(void){
    return in_test();
}
void in_test(void){
    return ;
}
```

L'exécution

Exemple d'exécution dans la stack (pile)

Appel de la fonction `in_test`, même manipulation sur `%rip`

<code>%rsp</code>	Stack	<code>%rip</code>	Assembleur - section text
0xf0	0xff (%rbp)	0x01	push %rbp
0xe0	0x09 (main)	0x02	mov %rsp,%rbp
0xd0	0xe0 (%rbp)	0x05	callq 0x12 <test>
→0xc0	0x24 (test)	0x09	mov \$0x0,%eax
0xb0		0x0d	retq
0xa0		0x12	push %rbp
		0x13	mov %rsp,%rbp
		→ 0x1f	callq 0x26 <in_test>
		0x24	pop %rbp
		0x25	retq
		0x26	push %rbp
		0x27	mov %rsp,%rbp
		0x2a	nop
		0x2b	pop %rbp
		0x2c	retq

```
Code C
void main(void){
    → test();
}
void test(void){
    → return in_test();
}
void in_test(void){
    return ;
}
```

L'exécution

Exemple d'exécution dans la stack (pile)

Entrée dans la fonction in_test

%rsp	Stack
0xf0	0xff (%rbp)
0xe0	0x09 (main)
0xd0	0xe0 (%rbp)
0xc0	0x24 (test)
→0xb0	0xc0 (%rbp)
0xa0	

%rip	Assembleur - section text
0x01	push %rbp
0x02	mov %rsp,%rbp
0x05	callq 0x12 <test>
0x09	mov \$0x0,%eax
0x0d	retq
0x12	push %rbp
0x13	mov %rsp,%rbp
0x1f	callq 0x26 <in_test>
0x24	pop %rbp
0x25	retq
→ 0x26	push %rbp
0x27	mov %rsp,%rbp
0x2a	nop
0x2b	pop %rbp
0x2c	retq

```
Code C
void main(void){
    →     test();
    }
void test(void){
    →     return in_test();
}
→     void in_test(void){
    →         return ;
    }
```

L'exécution

Exemple d'exécution dans la stack (pile)

Entrée dans la fonction in_test

%rsp	Stack
0xf0	0xff (%rbp)
0xe0	0x09 (main)
0xd0	0xe0 (%rbp)
0xc0	0x24 (test)
0xb0	0xc0 (%rbp)
0xa0	

%rip	Assembleur - section text
0x01	push %rbp
0x02	mov %rsp,%rbp
0x05	callq 0x12 <test>
0x09	mov \$0x0,%eax
0x0d	retq
0x12	push %rbp
0x13	mov %rsp,%rbp
0x1f	callq 0x26 <in_test>
0x24	pop %rbp
0x25	retq
0x26	push %rbp
→ 0x27	mov %rsp,%rbp
0x2a	nop
0x2b	pop %rbp
0x2c	retq

Code C

```
void main(void){  
    test();  
}
```

```
→ void test(void){  
    return in_test();  
}
```

```
→ void in_test(void){  
    return;  
}
```

L'exécution

Exemple d'exécution dans la stack (pile)

Entrée dans la fonction in_test

%rsp	Stack
0xf0	0xff (%rbp)
0xe0	0x09 (main)
0xd0	0xe0 (%rbp)
0xc0	0x24 (test)
0xb0	0xc0 (%rbp)
0xa0	

%rip	Assembleur - section text
0x01	push %rbp
0x02	mov %rsp,%rbp
0x05	callq 0x12 <test>
0x09	mov \$0x0,%eax
0x0d	retq
0x12	push %rbp
0x13	mov %rsp,%rbp
0x1f	callq 0x26 <in_test>
0x24	pop %rbp
0x25	retq
0x26	push %rbp
0x27	mov %rsp,%rbp
→ 0x2a	nop
0x2b	pop %rbp
0x2c	retq

```
Code C
void main(void){
    → test();
}
void test(void){
    → return in_test();
}
void in_test(void){
    → return ;
}
```

L'exécution

Exemple d'exécution dans la stack (pile)

Retour de la fonction, %rbp est restauré et retq restaure %rip

%rsp	Stack	%rip	Assembleur - section text
0xf0	0xff (%rbp)	0x01	push %rbp
0xe0	0x09 (main)	0x02	mov %rsp,%rbp
0xd0	0xe0 (%rbp)	0x05	callq 0x12 <test>
0xc0	0x24 (test)	0x09	mov \$0x0,%eax
0xb0		0x0d	retq
0xa0		0x12	push %rbp
		0x13	mov %rsp,%rbp
		0x1f	callq 0x26 <in_test>
		0x24	pop %rbp
		0x25	retq
		0x26	push %rbp
		0x27	mov %rsp,%rbp
		0x2a	nop
		→0x2b	pop %rbp
		0x2c	retq

```
Code C
void main(void){
    → test();
}
void test(void){
    → return in_test();
}
void in_test(void){
    → return ;
}
```

L'exécution

Exemple d'exécution dans la stack (pile)

Retour de la fonction, %rbp est restauré et retq restaure %rip

%rsp	Stack	%rip	Assembleur - section text
0xf0	0xff (%rbp)	0x01	push %rbp
0xe0	0x09 (main)	0x02	mov %rsp,%rbp
0xd0	0xe0 (%rbp)	0x05	callq 0x12 <test>
0xc0		0x09	mov \$0x0,%eax
0xb0		0x0d	retq
0xa0		0x12	push %rbp
		0x13	mov %rsp,%rbp
		0x1f	callq 0x26 <in_test>
		0x24	pop %rbp
		0x25	retq
		0x26	push %rbp
		0x27	mov %rsp,%rbp
		0x2a	nop
		0x2b	pop %rbp
		→0x2c	retq

Code C

```
void main(void){
    →    test();
}
void test(void){
    →    return in_test();
}
void in_test(void){
    →    return ;
}
```

L'exécution

Exemple d'exécution dans la stack (pile)

Retour de la fonction, %rbp est restauré et retq restaure %rip

%rsp	Stack	%rip	Assembleur - section text
0xf0	0xff (%rbp)	0x01	push %rbp
0xe0	0x09 (main)	0x02	mov %rsp,%rbp
0xd0		0x05	callq 0x12 <test>
0xc0		0x09	mov \$0x0,%eax
0xb0		0x0d	retq
0xa0		0x12	push %rbp
		0x13	mov %rsp,%rbp
		0x1f	callq 0x26 <in_test>
		→0x24	pop %rbp
		0x25	retq
		0x26	push %rbp
		0x27	mov %rsp,%rbp
		0x2a	nop
		0x2b	pop %rbp
		0x2c	retq

```
Code C
void main(void){
    →    test();
}
void test(void){
    →    return in_test();
}
void in_test(void){
    →    return ;
}
```

L'exécution

Exemple d'exécution dans la stack (pile)

Retour de la fonction, %rbp est restauré et retq restaure %rip

%rsp	Stack	%rip	Assembleur - section text
0xf0	0xff (%rbp)	0x01	push %rbp
0xe0		0x02	mov %rsp,%rbp
0xd0		0x05	callq 0x12 <test>
0xc0		0x09	mov \$0x0,%eax
0xb0		0x0d	retq
0xa0		0x12	push %rbp
		0x13	mov %rsp,%rbp
		0x1f	callq 0x26 <in_test>
		0x24	pop %rbp
		→0x25	retq
		0x26	push %rbp
		0x27	mov %rsp,%rbp
		0x2a	nop
		0x2b	pop %rbp
		0x2c	retq

Code C

```
void main(void){  
    test();  
}
```

```
void test(void){  
    return in_test();  
}
```

```
void in_test(void){  
    return;  
}
```

L'exécution

Exemple d'exécution dans la stack (pile)

Retour de la fonction, %rbp est restauré et retq restaure %rip

%rsp	Stack	%rip	Assembleur - section text
0xf0	0xff (%rbp)	0x01	push %rbp
0xe0		0x02	mov %rsp,%rbp
0xd0		0x05	callq 0x12 <test>
0xc0		→0x09	mov \$0x0,%eax
0xb0		0x0d	retq
0xa0		0x12	push %rbp
		0x13	mov %rsp,%rbp
		0x1f	callq 0x26 <in_test>
		0x24	pop %rbp
		0x25	retq
		0x26	push %rbp
		0x27	mov %rsp,%rbp
		0x2a	nop
		0x2b	pop %rbp
		0x2c	retq

Code C

```
void main(void){  
    test();  
}
```

```
void test(void){  
    return in_test();  
}
```

```
void in_test(void){  
    return;  
}
```

L'exécution

Exemple d'exécution dans la stack (pile)

Retour de la fonction, %rbp est restauré et retq restaure %rip

%rsp	Stack	%rip	Assembleur - section text
0xff	0xff (%rbp)	0x01	push %rbp
0xe0		0x02	mov %rsp,%rbp
0xd0		0x05	callq 0x12 <test>
0xc0		0x09	mov \$0x0,%eax
0xb0		→0x0d	retq
0xa0		0x12	push %rbp
		0x13	mov %rsp,%rbp
		0x1f	callq 0x26 <in_test>
		0x24	pop %rbp
		0x25	retq
		0x26	push %rbp
		0x27	mov %rsp,%rbp
		0x2a	nop
		0x2b	pop %rbp
		0x2c	retq

Code C

```
void main(void){  
    test();  
}
```

```
void test(void){  
    return in_test();  
}
```

```
void in_test(void){  
    return;  
}
```

Sommaire

1 Debugger

2 Les instructions

3 Processus utilisateur

- ps, top...
- strace, ltrace
- Les sources
- gdb
- gdb
- gdb
- gdb

4 Noyau

5 Question

Debugger
oooooo
ps, top...

Les instructions
oooooooo

Processus utilisateur
o●oooooooooooo

Noyau
oooooooooooooooooooo

Question
oo

ps, top...

Les deux outils ps et top permettent de consulter la structure mémoire d'un processus (*task_struct*).

ps, top...

On consulte la consommation CPU et mémoire de chaque processus, ainsi que l'état dans lequel il se trouve (en cours d'exécution, stoppé, bloqué, zombie...).

- ps
 - permet d'observer un ou plusieurs processus à un instant T
 - système de filtrage par utilisateur, PID ...
- top
 - Visualisation en continue des processus les plus actifs du système
 - Consultation unifiée de l'uptime de la machine et de l'utilisation de processeurs

Ces deux outils permettent de consulter la santé du système d'exploitation.

Exemple de sortie avec la commande ps

USER	PID	%CPU	%MEM	VSZ	RSS	TTY	ps aux f			
							STAT	START	TIME	COMMAND
root	2	0.0	0.0	0	0	?	S	Oct19	0:00	[kthreadd]
root	3	0.0	0.0	0	0	?	S	Oct19	0:05	_ [ksoftirqd/0]
root	5	0.0	0.0	0	0	?	S<	Oct19	0:00	_ [kworker/0:0H]
root	7	0.0	0.0	0	0	?	S	Oct19	4:58	_ [rcu_sched]
root	8	0.0	0.0	0	0	?	S	Oct19	0:00	_ [rcu_bh]
root	9	0.0	0.0	0	0	?	S	Oct19	0:01	_ [migration/0]
root	10	0.0	0.0	0	0	?	S	Oct19	0:00	_ [watchdog/0]
root	11	0.0	0.0	0	0	?	S	Oct19	0:00	_ [watchdog/1]
root	12	0.0	0.0	0	0	?	S	Oct19	0:01	_ [migration/1]
root	13	0.0	0.0	0	0	?	S	Oct19	0:02	_ [ksoftirqd/1]
root	15	0.0	0.0	0	0	?	S<	Oct19	0:00	_ [kworker/1:0H]
root	16	0.0	0.0	0	0	?	S	Oct19	0:00	_ [watchdog/2]
root	17	0.0	0.0	0	0	?	S	Oct19	0:01	_ [migration/2]
root	18	0.0	0.0	0	0	?	S	Oct19	0:02	_ [ksoftirqd/2]

Les différents champs (*ps aux f*)

- PID : Id du processus
 - %CPU : Pourcentage d'occupation CPU
 - %MEM : Pourcentage d'occupation Mémoire
 - VSZ : Mémoire virtuellement utilisable
 - RSS : Mémoire réellement utilisée
 - TTY : Terminal attaché au processus
 - STAT : Status du processus
 - START : Date de début du processus
 - TIME : Temps système consommé
 - COMMAND : Ligne de commande du processus

Exemple de sortie avec la commande ps

USER	PID	%CPU	%MEM	VSZ	RSS	TTY	ps aux f		
							STAT	START	TIME COMMAND
root	2	0.0	0.0	0	0	?	S	Oct19	0:00 [kthreadd]
root	3	0.0	0.0	0	0	?	S	Oct19	0:05 _ [ksoftirqd/0]
root	5	0.0	0.0	0	0	?	S<	Oct19	0:00 _ [kworker/0:0H]
root	7	0.0	0.0	0	0	?	S	Oct19	4:58 _ [rcu_sched]
root	8	0.0	0.0	0	0	?	S	Oct19	0:00 _ [rcu_bh]
root	9	0.0	0.0	0	0	?	S	Oct19	0:01 _ [migration/0]
root	10	0.0	0.0	0	0	?	S	Oct19	0:00 _ [watchdog/0]
root	11	0.0	0.0	0	0	?	S	Oct19	0:00 _ [watchdog/1]
root	12	0.0	0.0	0	0	?	S	Oct19	0:01 _ [migration/1]
root	13	0.0	0.0	0	0	?	S	Oct19	0:02 _ [ksoftirqd/1]
root	15	0.0	0.0	0	0	?	S<	Oct19	0:00 _ [kworker/1:0H]
root	16	0.0	0.0	0	0	?	S	Oct19	0:00 _ [watchdog/2]
root	17	0.0	0.0	0	0	?	S	Oct19	0:01 _ [migration/2]
root	18	0.0	0.0	0	0	?	S	Oct19	0:02 _ [ksoftirqd/2]

Ensemble des processus noyaux

Exemple de sortie avec la commande ps

USER	PID	%CPU	%MEM	VSZ	RSS	TTY	ps aux f			
							STAT	START	TIME	COMMAND
root	2	0.0	0.0	0	0	?	S	Oct19	0:00	[kthreadd]
root	3	0.0	0.0	0	0	?	S	Oct19	0:05	_ [ksoftirqd/0]
root	5	0.0	0.0	0	0	?	S<	Oct19	0:00	_ [kworker/0:0H]
root	7	0.0	0.0	0	0	?	S	Oct19	4:58	_ [rcu_sched]
root	8	0.0	0.0	0	0	?	S	Oct19	0:00	_ [rcu_bh]
root	9	0.0	0.0	0	0	?	S	Oct19	0:01	_ [migration/0]
root	10	0.0	0.0	0	0	?	S	Oct19	0:00	_ [watchdog/0]
root	11	0.0	0.0	0	0	?	S	Oct19	0:00	_ [watchdog/1]
root	12	0.0	0.0	0	0	?	S	Oct19	0:01	_ [migration/1]
root	13	0.0	0.0	0	0	?	S	Oct19	0:02	_ [ksoftirqd/1]
root	15	0.0	0.0	0	0	?	S<	Oct19	0:00	_ [kworker/1:0H]
root	16	0.0	0.0	0	0	?	S	Oct19	0:00	_ [watchdog/2]
root	17	0.0	0.0	0	0	?	S	Oct19	0:01	_ [migration/2]
root	18	0.0	0.0	0	0	?	S	Oct19	0:02	_ [ksoftirqd/2]
root	1032	0.0	0.0	304632	2956	?	Ssl	Oct19	0:06	/usr/lib/snapd/snapd
root	1036	0.0	0.0	274964	1332	?	Ssl	Oct19	0:00	/usr/sbin/cups-browsed
root	1046	0.0	0.0	29028	532	?	Ss	Oct19	0:00	/usr/sbin/cron -f
root	1048	0.0	0.0	276204	368	?	Ssl	Oct19	0:03	/usr/lib/accountsservice/accounts-daemon
root	1050	0.0	0.0	166456	240	?	Ssl	Oct19	0:11	/usr/sbin/thermald --no-daemon --dbus-enable
root	1054	0.0	0.0	337360	432	?	Ssl	Oct19	0:00	/usr/sbin/ModemManager
root	1056	0.0	0.0	449528	2804	?	Ssl	Oct19	0:00	/usr/sbin/NetworkManager --no-daemon
nobody	1316	0.0	0.0	52948	0	?	S	Oct19	0:00	_ /usr/sbin/dnsmasq --no-resolv --keep-in-foreground --no-hup
root	21637	0.0	0.0	16128	92	?	S	Oct22	0:00	_ /sbin/dhcpcd -d -q -sf /usr/lib/NetworkManager/nm-dhcp-client
root	1176	0.0	0.0	276816	372	?	SLsl	Oct19	0:00	/usr/sbin/lightdm

Suivis des processus utilisateurs

Exemple de sortie avec la commande top

top

```
top - 23:37:03 up 6 days, 1:54, 9 users, load average: 0,29, 0,35, 0,26
Tâches: 260 total, 1 en cours, 258 en veille, 0 arrêté, 1 zombie
%Cpu0 : 1,7 ut, 0,7 sy, 0,0 ni, 97,7 id, 0,0 wa, 0,0 hi, 0,0 si, 0,0 st
top - 23:37:29 up 6 days, 1:54, 9 users, load average: 0,27, 0,33, 0,26
Tasks: 260 total, 1 running, 258 sleeping, 0 stopped, 1 zombie
%Cpu0 : 3,4 us, 1,1 sy, 0,0 ni, 95,5 id, 0,0 wa, 0,0 hi, 0,0 si, 0,0 st
%Cpu1 : 0,0 us, 0,0 sy, 0,0 ni, 100,0 id, 0,0 wa, 0,0 hi, 0,0 si, 0,0 st
%Cpu2 : 1,1 us, 0,0 sy, 0,0 ni, 98,9 id, 0,0 wa, 0,0 hi, 0,0 si, 0,0 st
%Cpu3 : 0,0 us, 1,1 sy, 0,0 ni, 98,9 id, 0,0 wa, 0,0 hi, 0,0 si, 0,0 st
KiB Mem : 5831144 total, 410448 free, 3567680 used, 1853016 buff/cache
KiB Swap: 6273020 total, 6043320 free, 229700 used. 1592284 avail Mem
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
1231	root	20	0	375884	86940	35752	S	4.5	1.5	56:16.89	Xorg
22000	rachel	20	0	2905836	538740	154532	S	2.2	9.2	4:02.72	firefox
2345	mat	20	0	299152	42664	7084	S	1.1	0.7	82:23.29	awesome
5597	mat	20	0	39028	3432	2796	R	1.1	0.1	0:00.01	top
22103	rachel	20	0	2593304	761204	138880	S	1.1	13.1	6:30.29	Web Content
1	root	20	0	119904	4060	2388	S	0.0	0.1	0:03.34	systemd
2	root	20	0	0	0	0	S	0.0	0.0	0:00.04	kthreadd
3	root	20	0	0	0	0	S	0.0	0.0	0:05.30	ksoftirqd/0
5	root	0	-20	0	0	0	S	0.0	0.0	0:00.00	kworker/0:0H
7	root	20	0	0	0	0	S	0.0	0.0	5:00.94	rcu_sched

Les premières lignes donnent une vue synthétique de l'utilisation de la machine.

Sous le capot

Chacune de ces commandes vont parcourir `/proc`, un pseudo filesysteme dans lequel on peut trouver les informations de l'ensemble des processus de la machine.

procfs

```
$ cat /proc/self/stat
15933 (cat) R 10118 15933 10118 34820 15933 4194304 88 0 0 0 0 0 0 0 20 0 1 0 30568230 7647232 193 3121741824
→ 4194304 4240236 140732349587456 140732349586808 140640986640944 0 0 128 0 0 0 17 0 0 0 0 0 0 6340112 6341364
→ 36347904 140732349592700 140732349592720 140732349592720 140732349595631 0
```

D'autres informations sont également disponible dans `procfs` et sont lus par d'autre outils.

procfs

```
$ ls /proc/self/
attr      clear_refs      cpuset      fd      limits      mem      net      oom_score      projid_map
→ sessionid  stat      task
autogroup cmdline      cwd      fdinfo      loginuid      mountinfo      ns      oom_score_adj      root
→ setgroups statm      timers
auxv      comm      environ      gid_map      map_files      mounts      numa_maps      pagemap      sched      smaps
→ status      uid_map
cgroup      coredump_filter      exe      io      maps      mountstats      oom_adj      personality      schedstat      stack
→ syscall      wchan
```

strace

L'outil *strace* permet de voir l'ensemble des appels systèmes effectués par un processus. Les appels systèmes sont également appelés *syscall* et correspondent aux fonctions exposées par le noyau vers l'espace utilisateur.

Exemple d'utilisation de strace

```
# strace -e open, getdents ls
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
open("/lib/x86_64-linux-gnu/libselinux.so.1", O_RDONLY|O_CLOEXEC) = 3
open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
open("/lib/x86_64-linux-gnu/libpcre.so.3", O_RDONLY|O_CLOEXEC) = 3
open("/lib/x86_64-linux-gnu/libdl.so.2", O_RDONLY|O_CLOEXEC) = 3
open("/lib/x86_64-linux-gnu/libpthread.so.0", O_RDONLY|O_CLOEXEC) = 3
open("/proc/filesystems", O_RDONLY)      = 3
open("/usr/lib/locale-archive", O_RDONLY|O_CLOEXEC) = 3
open(".", O_RDONLY|O_NONBLOCK|O_DIRECTORY|O_CLOEXEC) = 3
getdents(3, /* 2 entries */, 32768)    = 48
getdents(3, /* 0 entries */, 32768)    = 0
+++ exited with 0 +++
```

strace est utilisé pour comprendre comment un processus en espace utilisateur inter-agit avec le noyau via les appels systèmes.

strace

L'outil *strace* permet de voir l'ensemble des appels systèmes effectués par un processus. Les appels systèmes sont également appelés *syscall* et correspondent aux fonctions exposées par le noyau vers l'espace utilisateur.

Exemple d'utilisation de strace

```
# strace -e open, getdents ls
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
open("/lib/x86_64-linux-gnu/libselinux.so.1", O_RDONLY|O_CLOEXEC) = 3
open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
open("/lib/x86_64-linux-gnu/libpcre.so.3", O_RDONLY|O_CLOEXEC) = 3
open("/lib/x86_64-linux-gnu/libdl.so.2", O_RDONLY|O_CLOEXEC) = 3
open("/lib/x86_64-linux-gnu/libpthread.so.0", O_RDONLY|O_CLOEXEC) = 3
open("/proc/filesystems", O_RDONLY)      = 3
open("/usr/lib/locale-archive", O_RDONLY|O_CLOEXEC) = 3
open(".", O_RDONLY|O_NONBLOCK|O_DIRECTORY|O_CLOEXEC) = 3
getdents(3, /* 2 entries */, 32768)    = 48
getdents(3, /* 0 entries */, 32768)    = 0
+++ exited with 0 +++
```

strace est utilisé pour comprendre comment un processus en espace utilisateur inter-agit avec le noyau via les appels systèmes.

syscall - int 0x80

strace les options intéressantes

Ne sélectionner que certains appels : -e

```
# strace -e open,gettents ls
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
open("/lib/x86_64-linux-gnu/libselinux.so.1", O_RDONLY|O_CLOEXEC) = 3
open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
open("/lib/x86_64-linux-gnu/libpcre.so.3", O_RDONLY|O_CLOEXEC) = 3
open("/lib/x86_64-linux-gnu/libdl.so.2", O_RDONLY|O_CLOEXEC) = 3
open("/lib/x86_64-linux-gnu/libpthread.so.0", O_RDONLY|O_CLOEXEC) = 3
open("/proc/filesystems", O_RDONLY) = 3
open("/usr/lib/locale/locale-archive", O_RDONLY|O_CLOEXEC) = 3
open(".", O_RDONLY|O_NONBLOCK|O_DIRECTORY|O_CLOEXEC) = 3
gettents(3, /* 2 entries */, 32768) = 48
gettents(3, /* 0 entries */, 32768) = 0
+++ exited with 0 +++
```

strace les options intéressantes

Voir le temps passé dans chaque appel : -tt

strace les options intéressantes

Voir le nombre d'appel utilisés : -c

% time	seconds	usecs/call	calls	errors	syscall
0.00	0.000000	0	7		read
0.00	0.000000	0	1		write
0.00	0.000000	0	8		open
0.00	0.000000	0	10		close
0.00	0.000000	0	9		fstat
0.00	0.000000	0	18		mmap
0.00	0.000000	0	12		mprotect
0.00	0.000000	0	1		munmap
0.00	0.000000	0	3		brk
0.00	0.000000	0	2		rt_sigaction
0.00	0.000000	0	1		rt_sigprocmask
0.00	0.000000	0	2		ioctl
0.00	0.000000	0	7	7	access
0.00	0.000000	0	1		execve
0.00	0.000000	0	2		getdents
0.00	0.000000	0	1		getrlimit
0.00	0.000000	0	2	2	statfs
0.00	0.000000	0	1		arch_prctl
0.00	0.000000	0	1		set_tid_address
0.00	0.000000	0	1		set_robust_list
<hr/>					
100.00	0.000000		90	9	total

strace les options intéressantes

S'attacher à un processus existant : -p

ltrace

L'outil *ltrace* permet de voir l'ensemble des appels à la libc. Les informations fournies ici correspondent aux différentes fonctions appelées dans les librairies du système. Les appels systèmes ne sont pas visibles.

```
ltrace -e opendir -e readdir ls
# ltrace -e opendir -e readdir ls
ls->opendir(".")

ls->readdir(0xf6bca0)
ls->readdir(0xf6bca0)
ls->readdir(0xf6bca0)
ls->readdir(0xf6bca0)
*** exited (status 0) ***
```

- 0xf6bca0
- 0xf6bcd0
- 0xf6bce8
- 0

ltrace est utilisé pour comprendre le comportement d'un processus utilisateur en dehors des appels système.

Les options

ltrace possède les mêmes options que *strace* :

- -e : sélection des fonctions à observer
- -tt : temps passé dans chaque fonction
- -c : nombre d'appel utilisé pour chaque fonction
- -p : PID sur lequel s'attacher

ltrace

L'outil *ltrace* permet de voir l'ensemble des appels à la libc. Les informations fournies ici correspondent aux différentes fonctions appelées dans les librairies du système. ~~Les appels systèmes ne sont pas visibles.~~

```
ltrace -e opendir -e readdir ls
# ltrace -e opendir -e readdir ls
ls->opendir("."          □ 0xf6bca0
ls->readdir(0xf6bca0)   □ 0xf6bcd0
ls->readdir(0xf6bca0)   □ 0xf6bce8
ls->readdir(0xf6bca0)   □ 0
*** exited (status 0) ***
```

ltrace est utilisé pour comprendre le comportement d'un processus utilisateur en dehors des appels système.

Les options

ltrace possède les mêmes options que *strace* :

- -e : sélection des fonctions à observer
- -tt : temps passé dans chaque fonction
- -c : nombre d'appel utilisé pour chaque fonction
- -p : PID sur lequel s'attacher
- -S : affiche également les appels systèmes

Les sources



Les sources

Le point fondamental permettant de debugger efficacement est d'avoir accès aux sources. Qu'il s'agisse du noyau ou d'un autre programme, les sources facilitent grandement la compréhension d'un problème.

Pour avoir un environnement de debugging complet, il faut :

- La version du logiciel à debugger
- Les sources correspondantes
- Les symboles de debug : fournis par la distribution ou à compiler soi-même

Les sources

Les sources du noyau sont très importantes et volumineuses, rechercher dans le code source une définition de fonction ou un symbole particulier peut s'avérer très compliqué.

Pour nous faciliter la tâche, l'outil *cscope* permet de naviguer facilement dans un gros projet écrit en C.

- Le noyau supporte directement l'indexation de *cscope* : *make cscope*
- *cscope -d -R*

Les sources

Global definition: task_struct

File	Line
0 profile.h	66 struct task_struct;
1 regset.h	20 struct task_struct;
2 regset.h	39 typedef int user_regset_active_fn(struct task_struct *target,
3 regset.h	58 typedef int user_regset_get_fn(struct task_struct *target,
4 regset.h	79 typedef int user_regset_set_fn(struct task_struct *target,
5 regset.h	105 typedef int user_regset_writeback_fn(struct task_struct *target,
6 resource.h	7 struct task_struct;
7 sched.h	483 struct task_struct {
8 autogroup.h	5 struct task_struct;
9 debug.h	8 struct task_struct;
a jobctl.h	6 struct task_struct;

...

* Press the space bar to display the first lines again *

Find this C symbol:

Find this global definition: task_struct

Find functions called by this function:

Find functions calling this function:

Find this text string:

Change this text string:

Find this egrep pattern:

Find this file:

Find files *#including this file*:

Find assignments to this symbol:

gdb

gdb est l'outil par excellence de debug. Il permet d'inspecter le comportement et le contenu d'un exécutable.

- Permet de lancer l'exécution d'un programme pas à pas (breakpoints)
- Visualise les différentes variables des fonctions en cours
- Affiche la pile d'exécution
- Affiche les registres
- Peut aussi modifier le comportement du programme (en modifiant des variables par exemple)

Il existe 2 modes de lancement pour *gdb*

Mode classique au lancement de l'exécutable

```
gdb <exécutable> [paramètres de l'exécutable]
```

Mode attachement à un processus en cours

```
gdb -p <PID>
```

Les commandes principales

- `break <ligne>` : place un point d'arrêt dans le code

Exemple d'utilisation de gdb

```
(gdb) break opendir
Breakpoint 1 at 0x7ffff78b3140: file ../sysdeps posix/opendir.c, line 181.
(gdb) run /
Starting program: /bin/ls /
[Thread debugging using libthread_db enabled]
Using host libthread_db library `/lib/x86_64-linux-gnu/libthread_db.so.1'.

Breakpoint 1, _{opendir (name=0x625cc0 `') at ../sysdeps posix/opendir.c:182
182 ../sysdeps posix/opendir.c: Aucun fichier ou dossier de ce type.
(gdb) where
#0  _{opendir (name=0x625cc0 `') at ../sysdeps posix/opendir.c:182
→
#1  0x00000000000403849 in ?? ()
#2  0x00007ffff780b830 in _{libc_start_main (main=0x402a00, argc=2,
    argv=0x7fffffe118, init=<optimized out>, fini=<optimized out>,
    rtld_fini=<optimized out>, stack_end=0x7fffffe108)
    at ../csu/libc-start.c:291
#3  0x000000000004049c9 in ?? ()})}
```

gdb

Les commandes principales

- `break <ligne>` : place un point d'arrêt dans le code
- `run <args>` : lance l'exécutable avec les arguments spécifiés

Exemple d'utilisation de gdb

```
(gdb) break opendir
Breakpoint 1 at 0x7ffff78b3140: file ../sysdeps/posix/opendir.c, line 181.
(gdb) run /
Starting program: /bin/ls /
[Thread debugging using libthread_db enabled]
Using host libthread_db library `/lib/x86_64-linux-gnu/libthread_db.so.1'.

Breakpoint 1, _{opendir (name=0x625cc0 `') at ../sysdeps/posix/opendir.c:182
182 ../sysdeps/posix/opendir.c: Aucun fichier ou dossier de ce type.
(gdb) where
#0  _{opendir (name=0x625cc0 `') at ../sysdeps/posix/opendir.c:182
→
#1  0x00000000000403849 in ?? (
#2  0x00007ffff780b830 in _{libc_start_main (main=0x402a00, argc=2,
    argv=0x7fffffe118, init=<optimized out>, fini=<optimized out>,
    rtld_fini=<optimized out>, stack_end=0x7fffffe108)
    at ../csu/libc-start.c:291
#3  0x000000000004049c9 in ?? ()})}
```

gdb

Les commandes principales

- `break <ligne>` : place un point d'arrêt dans le code
- `run <args>` : lance l'exécutable avec les arguments spécifiés
- `where` : affiche la pile d'appel d'exécution

Exemple d'utilisation de gdb

```
(gdb) break opendir
Breakpoint 1 at 0x7ffff78b3140: file ../sysdeps posix/opendir.c, line 181.
(gdb) run /
Starting program: /bin/ls /
[Thread debugging using libthread_db enabled]
Using host libthread_db library `/lib/x86_64-linux-gnu/libthread_db.so.1'.

Breakpoint 1, _{opendir (name=0x625cc0 `') at ../sysdeps posix/opendir.c:182
182 ../sysdeps posix/opendir.c: Aucun fichier ou dossier de ce type.
(gdb) where
#0  _{opendir (name=0x625cc0 `') at ../sysdeps posix/opendir.c:182
→
#1  0x00000000000403849 in ?? (
#2  0x00007ffff780b830 in _{libc_start_main (main=0x402a00, argc=2,
    argv=0x7fffffe118, init=<optimized out>, fini=<optimized out>,
    rtd_fini=<optimized out>, stack_end=0x7fffffe108)
    at ../csu/libc-start.c:291
#3  0x000000000004049c9 in ?? ()})}
```

gdb

Les commandes principales

- `break <ligne>` : place un point d'arrêt dans le code
- `run <args>` : lance l'exécutable avec les arguments spécifiés
- `where` : affiche la pile d'appel d'exécution

Exemple de backtrace de tous les threads d'un processus utilisateur

```
(gdb) thread apply all where
```

```
Thread 1 (LWP 20141):  
#0 0x000007fd16877807a in __GI___waitpid (pid=-1, stat_loc=0x7fffbcce8c220, options=10) at  
  ./sysdeps/unix/sysv/linux/waitpid.c:29  
#1 0x0000000000044706d in ?? ()  
#2 0x0000000000044854b in wait_for ()  
#3 0x000000000004384bf in execute_command_internal ()  
#4 0x0000000000043851e in execute_command ()  
#5 0x0000000000042139e in reader_loop ()  
#6 0x0000000000041fdb1 in main ()
```

gdb

Autres commandes

- up/down : déplacement dans la stack d'exécution

Registres gdb

```
(gdb) info registers
rax          0xfffffffffffffdfc      -516
rbx          0x0          0
rcx          0x7f0d6900a1b1      139695572951473
rdx          0x0          0
rsi          0x0          0
rdi          0x7fff561480b0      140734637572272
rbp          0x7fff561480f0      0x7fff561480f0
rsp          0x7fff561480e0      0x7fff561480e0
r8           0x557f6f251770      94005813909360
r9           0x7f0d6930d9d0      139695576111568
r10          0x62f          1583
r11          0x246          582
r12          0x557f6f251590      94005813908880
r13          0x7fff561481e0      140734637572576
r14          0x0          0
r15          0x0          0
rip          0x557f6f2516ca      0x557f6f2516ca <loop+27>
...
```

gdb

Autres commandes

- up/down : déplacement dans la stack d'exécution
- info registers : affiche les registres

Registres gdb

```
(gdb) info registers
rax          0xfffffffffffffdfc      -516
rbx          0x0          0
rcx          0x7f0d6900a1b1      139695572951473
rdx          0x0          0
rsi          0x0          0
rdi          0x7fff561480b0      140734637572272
rbp          0x7fff561480f0      0x7fff561480f0
rsp          0x7fff561480e0      0x7fff561480e0
r8           0x557f6f251770      94005813909360
r9           0x7f0d6930d9d0      139695576111568
r10          0x62f          1583
r11          0x246          582
r12          0x557f6f251590      94005813908880
r13          0x7fff561481e0      140734637572576
r14          0x0          0
r15          0x0          0
rip          0x557f6f2516ca      0x557f6f2516ca <loop+27>
...
```

gdb

Autres commandes

- up/down : déplacement dans la stack d'exécution
- info registers : affiche les registres
- info source : Information sur les sources du binaire actuellement analysé.

Information sur les sources

```
(gdb) info source
Current source file is src.c
Compilation directory is /some/where
Located in /some/where/src.c
Contains 16 lines.
Source language is c.
Producer is GNU C11 7.2.0 -mtune=generic -march=x86-64 -g.
Compiled with DWARF 2 debugging format.
Does not include preprocessor macro info.
```

Autres commandes

- up/down : déplacement dans la stack d'exécution
- info registers : affiche les registres
- info source : Information sur les sources du binaire actuellement analysé.
- disassemble : affiche le code assembleur d'une fonction

Code assembleur d'une fonction

```
(gdb) disasemble func
Dump of assembler code for function func:
0x0000557f6f2516af <+0>:    push    %rbp
0x0000557f6f2516b0 <+1>:    mov     %rsp,%rbp
0x0000557f6f2516b3 <+4>:    sub    $0x10,%rsp
0x0000557f6f2516b7 <+8>:    movl   $0x0,-0x4(%rbp)
0x0000557f6f2516be <+15>:   jmp    0x557f6f2516ca <func+27>
0x0000557f6f2516c0 <+17>:   mov    $0x3e8,%edi
0x0000557f6f2516c5 <+22>:   callq  0x557f6f251580 <test+5>
-> 0x0000557f6f2516ca <+27>:  cmpl   $0x0,-0x4(%rbp)
0x0000557f6f2516ce <+31>:  sete   %al
0x0000557f6f2516d1 <+34>:  movzbl %al,%eax
0x0000557f6f2516d4 <+37>:  mov    %eax,%edi
0x0000557f6f2516d6 <+39>:  callq  0x557f6f25169a <print>
0x0000557f6f2516db <+44>:  test   %eax,%eax
0x0000557f6f2516dd <+46>:  jne    0x557f6f2516c0 <func+17>
0x0000557f6f2516df <+48>:  nop
0x0000557f6f2516e0 <+49>:  leaveq 
0x0000557f6f2516e1 <+50>:  retq
```

End of assembler dump.

gdb

Un des gros intérêt de gdb est qu'il peut analyser ce qu'il s'est passé sur un processus de façon post-mortem.

Les corefile

Les fichiers core ou corefiles contiennent l'ensemble de la mémoire d'un processus. Ces fichiers sont générés de différentes façons :

- lorsque le processus a effectué une erreur de segmentation.
- lorsque le processus a effectué une erreur système.
- à la demande : il est possible de demander au système de créer un corefile à tout moment

Exemple de génération d'un corefile

```
$ cat /proc/sys/kernel/core_pattern
|/usr/share/apport/apport %p %s %c %P
$ ulimit -c unlimited
$ gcore 21768
```

Sommaire

1 Debugger

2 Les instructions

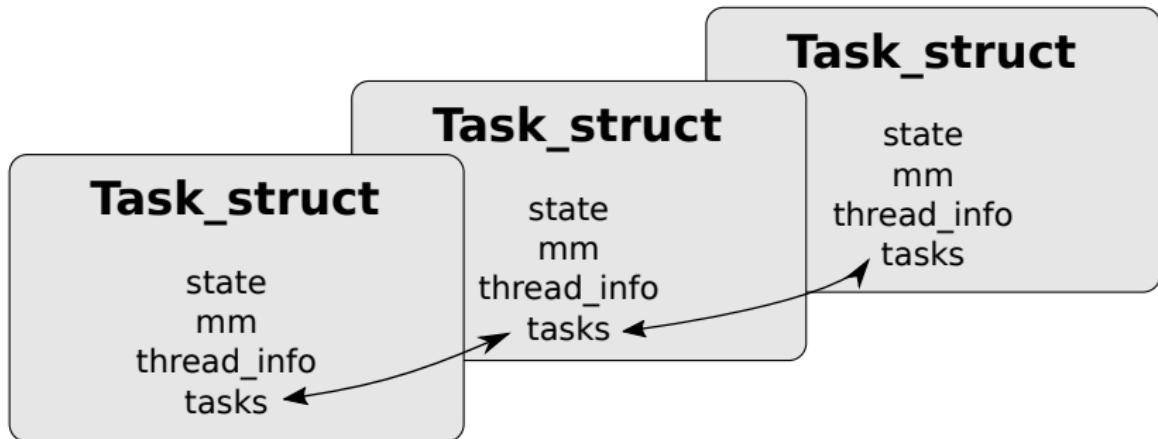
3 Processus utilisateur

4 Noyau

- Au delà de l'espace utilisateur
- Logs kernel
- debugfs
- Crash
- Autres outils

5 Question

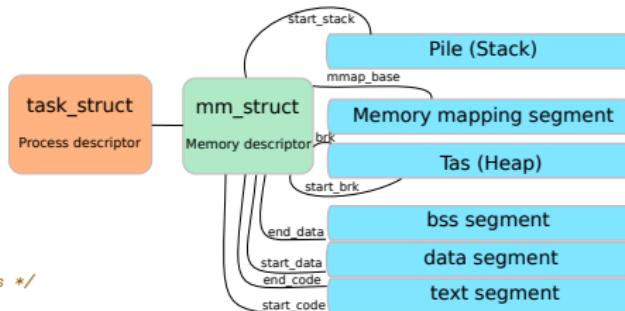
Mais qu'est-ce donc que cette `task_struct` ? ?



Au delà de l'espace utilisateur

Mais qu'est-ce donc que cette `task_struct` ??

```
struct task_struct {  
    ...  
    /* -1 unrunnable, 0 runnable, >0 stopped: */  
    volatile long state;  
    void *stack;  
    atomic_t usage;  
    ...  
    unsigned int cpu;  
    ...  
    struct mm_struct *mm;  
    struct mm_struct *active_mm;  
    /* Per-thread vma caching: */  
    struct vmacache vmacache;  
    struct mm_struct {  
        struct vm_area_struct *mmap; /* list of VMAs */  
    ...  
        unsigned long mmap_base; /* base of mmap area */  
    ...  
        unsigned long start_code, end_code, start_data, end_data;  
        unsigned long start_brk, brk, start_stack;  
    };  
};
```



- La `task_struct` est une représentation d'un processus au sein du noyau
- Elle contient une structure mémoire, la `mm_struct` dans laquelle le noyau chargera les différents segments du binaire
- C'est ensuite le scheduler qui se chargera d'organiser le lancement des différents processus représentés par ces `task_struct`

Debugger
ooooo

Les instructions
oooooooo

Processus utilisateur
oooooooooooo

Noyau
oo●oooooooooooooooooooo

Question
oo

Au delà de l'espace utilisateur

Attention, on s'accroche (bis)



Et les *syscalls* dans le noyau ?

Reprendons notre exemple d'appel à un *syscall* :

Le code source - Assembleur x86_64

```
.text
.globl _start

_start:
    mov $60, %rax
    mov $2, %rdi
    syscall
```

- Les *syscalls* sont définis par architecture
 - *arch/x86/entry/syscalls/syscall_32.tbl* : processeur x86
 - *arch/x86/entry/syscalls/syscall_64.tbl* : processeur x86_64
 - *arch/arm/tools/syscall.tbl* : processeur arm

Au delà de l'espace utilisateur

Et les *syscalls* dans le noyau ?

Reprendons notre exemple d'appel à un syscall :

Le code source - Assembleur x86 64

```
.text
.globl _start

_start:
        mov $60, %rax
        mov $2, %rdi
        syscall
```

```
syscall_64.tb

#
# 64-bit system call numbers and entry vectors
#
# The format is:
# <number> <abi> <name> <entry point>
#
# The abi is "common", "64" or "x32" for this file.
#
0      common      read          sys_read
1      common      write         sys_write
2      common      open          sys_open
3      common      close         sys_close
...
60     common      exit          sys_exit
...
```

L'implémentation des *syscalls*

- Le code déclenché par l'appel à un *syscall* est déclaré via les macros *DEFINE_SYSCALLn*.
- n* correspond au nombre de paramètres de l'appel système.
- Dans notre exemple *sys_exit* n'a qu'un paramètre.
- Il est donc défini avec la macro *DEFINE_SYSCALL1*.

```
SYSCALL_DEFINE1(exit, int, error_code)
{
    do_exit((error_code&0xff) << 8);
}
```

exit.c

Au delà de l'espace utilisateur

L'implémentation des *syscalls*

```
void __noreturn do_exit(long code)                                exit.c
{
    struct task_struct *tsk = current;
    int group_dead;
    TASKS_RCU(int tasks_rcu_i);

    profile_task_exit(tsk);
    kcov_task_exit(tsk);

    WARN_ON(blk_needs_flush_plug(tsk));

    if (unlikely(in_interrupt()))
        panic("Aiee, killing interrupt handler!");
    if (unlikely(!tsk->pid))
        panic("Attempted to kill the idle task!");

    /*
     * If do_exit is called because this processes oopsed, it's possible
     * that get_fs() was left as KERNEL_DS, so reset it to USER_DS before
     * continuing. Amongst other possible reasons, this is to prevent
     * mm_release()->clear_child_tid() from writing to a user-controlled
     * kernel address.
     */
    set_fs(USER_DS);

    ptrace_event(PTRACE_EVENT_EXIT, code);

    validate_creds_for_do_exit(tsk);

    /*
     * We're taking recursive faults here in do_exit. Safest is to just
     * leave this task alone and wait for reboot.
     */
    if (unlikely(tsk->flags & PF_EXITING)) {
```

Debugger
ooooo

Les instructions
ooooooo

Processus utilisateur
oooooooooooo

Noyau
ooooo●oooooooooooooooooooo

Question
oo

Au delà de l'espace utilisateur

Les outils...

dmesg

dmesg permet d'accéder aux messages du noyau.

- Donne les premières informations permettant de comprendre un problème provenant du noyau.
- Accès à tous les messages depuis le dernier démarrage de la machine.
- En interne, c'est le fichier */proc/kmsg* qui est lu dans un ring buffer.

dmesg

dmesg permet d'accéder aux messages du noyau.

- Donne les premières informations permettant de comprendre un problème provenant du noyau.
- Accès à tous les messages depuis le dernier démarrage de la machine.
- En interne, c'est le fichier `/proc/kmsg` qui est lu dans un ring buffer.

```
dmesg
[ 0.072938] PM: Registering ACPI NVS region [mem 0xbff641000-0xbff683fff] (274432 bytes)
[ 0.073006] clocksource: jiffies: mask: 0xffffffff max_cycles: 0xffffffff, max_idle_ns: 7645041785100000 ns
[ 0.073019] futex hash table entries: 1024 (order: 4, 65536 bytes)
[ 0.073091] pinctrl core: initialized pinctrl subsystem
[ 0.073189] RTC time: 7:49:09, date: 11/27/17
[ 0.073278] NET: Registered protocol family 16
[ 0.082796] cpuidle: using governor ladder
[ 0.087453] cpuidle: using governor menu
[ 0.087462] PCCT header not found.
[ 0.087548] ACPI: bus type PCI registered
[ 0.087552] acpiphp: ACPI Hot Plug PCI Controller Driver version: 0.5
[ 0.087617] PCI: MMCONFIG for domain 0000 [bus 00-ff] at [mem 0xe0000000-0xefffffff] (base 0xe0000000)
[ 0.087623] PCI: not using MMCONFIG
[ 0.087626] PCI: Using configuration type 1 for base access
[ 0.087744] NMI watchdog: enabled on all CPUs, permanently consumes one hw-PMU counter.
```

- `-T` : Affiche les timestamps dans un format lisible
- `-w` : Affiche les nouveaux messages dès leur réception

Debugfs

Debugfs est ce qu'on appelle un pseudo système de fichiers, il permet d'accéder aux fonctions dites de debug du noyau.

Accéder à debugfs

```
# mkdir /mnt/debug
# mount -t debugfs none /mnt/debug
```

Contenu de debugfs

```
# ls /sys/kernel/debug/
acpi           intel_powerclamp  regulator
bdi            iosf_sb          sched_features
btrfs           kprobes          sleep_time
cleancache      kvm              sunrpc
clk              mce              suspend_stats
dma_buf          mei0              tracing
dri              pinctrl          usb
dynamic_debug    pkg_temp_thermal virtio-ports
extfrag          pm_qos            wakeup_sources
fault_around_bytes pstate_snb    x86
frontswap        pwm              zswap
gpio             ras
hid              regmap
```

De nombreuses possibilités y sont offertes, nous allons uniquement nous concentrer sur deux d'entre-elles :

- `dynamic_debug`
- `tracing`

dynamic_debug

Le `dynamic_debug` est une fonctionnalité du noyau permettant d'activer les messages de debugging du noyau.

Le fichier `dynamic_debug/control` permet de lister et de contrôler ces activations.

Messages de debug activés

```
# awk '$$3 != "="' /sys/kernel/debug/dynamic_debug/control
# filename:lineno [module]function flags format
init/main.c:741 [main]initcall_blacklisted =p "initcall %s blacklisted\012"
init/main.c:717 [main]initcall_blacklist =p "blacklisting initcall %s\012"
arch/x86/kernel/cpu/mtrr/main.c:491 [main]mtrr_del_page =p "mtrr: no MTRR for %lx000,%lx000 found\012"
arch/x86/kernel/cpu/mtrr/main.c:399 [main]mtrr_check =p "mtrr: size: 0x%lx base: 0x%lx\012"
arch/x86/kernel/cpu/mtrr/generic.c:444 [generic]print_mtrr_state =p "TOM2: %016llx aka %lldM\012"
arch/x86/kernel/cpu/mtrr/generic.c:441 [generic]print_mtrr_state =p " %u disabled\012"
arch/x86/kernel/cpu/mtrr/generic.c:439 [generic]print_mtrr_state =p " %u base %0*X%05X000 mask %0*X%05X000 %s\012"
arch/x86/kernel/cpu/mtrr/generic.c:426 [generic]print_mtrr_state =p "MTRR variable ranges %sabled:\012"
```

Activation d'un message de debug

```
# echo "func SYSC_init_module +p" >/sys/kernel/debug/dynamic_debug/control
awk '/SYSC_init_module/' /sys/kernel/debug/dynamic_debug/control
kernel/module.c:3604 [module]SYSC_init_module =p "init_module: umod=%p, len=%lu, uargs=%p\012"
```

Une fois activés, ces messages sont consultables via `dmesg`.

tracing

Le tracing dans le noyau permet de suivre les différentes fonctions activées pendant une certaine période pour un ou plusieurs processus.

Le principe consiste à définir un ensemble de filtre, de choisir une fonction de suivi (tracing) et de l'activer.

Le résultat est stocké dans un ringbuffer dans le fichier :

- `/sys/kernel/debug/tracing/trace`

perf

La mise en place du tracing étant relativement complexe, l'outil *perf* permet de faciliter son utilisation.

- `perf list` : liste l'ensemble des trace disponibles
- `perf record` : enregistre une trace
- `perf report` : affiche le résultat d'une trace précédemment effectuée

Utilisation de `perf`

```
# cd /tmp
# perf record -e ext4:ext4_free_inode -a
^C[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.823 MB perf.data (2 samples) ]

# perf report
```

- Crash est une version améliorée de gdb destinée à faciliter le debugging du noyau.
- Il se base sur le fichier `/proc/kcore` ou une copie qui contient la mémoire du noyau à un instant T
- Permet une analyse post-mortem ou en live d'un problème ou d'un mauvais comportement

Pour pouvoir utiliser *crash*, il est impératif d'obtenir les symboles de debug du noyau que l'on souhaite debugger

- Soit le noyau est compilé par la distribution et il n'y a qu'à installer les packages :
 - `linux-image-XXX.YYY-ZZZ-generic-dbgsym` (Ubuntu/Debian)
 - `kernel-debuginfo` (RedHat/Fedora/CentOS)
 - ...
- Soit le noyau a été compilé depuis les sources et il faut recompiler le noyau avec les symboles de debug :
 - `apt source linux` (Ubuntu/Debian)
 - `kernel-XXX.YYY.src.rpm` (RedHat/Fedora/CentOS)
 - <https://www.kernel.org/>

Crash

Kexec/Kdump

Le challenge est de pouvoir récupérer la mémoire du noyau qui vient de planter.

- Kdump est un service qui pré-chARGE via PARTIR de kexec un noyau et d'un initrd minimal
- La mémoire du noyau de capture doit être réservée au boot du noyau
 - *crashkernel=auto*
- Une fois le service activé, le noyau de capture est prêt à être déclenché et à procéder à une capture.

Crash

Kexec/Kdump

Comment déclencher cette récupération ?

- Via les fichiers contenus dans `/proc/sys/kernel/panic*`, on définit le comportement du noyau qui va déclencher un *kernel panic*
- Pour effectivement déclencher le *panic* soit :
 - Le kernel panic de façon autonome
 - Via les magic sysrq
 - Via un nmi externe
- Lorsque le noyau se trouve dans cet état, le noyau kdump prend le relais et déclanche la sauvegarde de la mémoire contenu dans `/proc/kcore`.

```
# ls /proc/sys/kernel/panic*
```

<code>/proc/sys/kernel/panic</code>	<code>/proc/sys/kernel/panic_on_oops</code>	<code>/proc/sys/kernel/panic_on_warn</code>
<code>/proc/sys/kernel/panic_on_io_nmi</code>	<code>/proc/sys/kernel/panic_on_unrecoverable_nmi</code>	

Lancement

`crash [noyau corefile]`

- Sans argument, `crash` analyse le système en live
- En lui spécifiant le noyau avec ses symboles de debug et un corefile précédemment généré, on lance une analyse post-mortem

Lancement

crash [noyau corefile]

```
crash
KERNEL: /usr/lib/debug/boot/vmlinux-4.4.0-97-generic
DUMPFILE: /proc/kcore
CPUS: 4
DATE: Sun Nov 26 21:28:48 2017
UPTIME: 12 days, 16:41:39
LOAD AVERAGE: 1.01, 1.19, 1.14
TASKS: 713
NODENAME: dakoro
RELEASE: 4.4.0-97-generic
VERSION: #120-Ubuntu SMP Tue Sep 19 17:28:18 UTC 2017
MACHINE: x86_64 (3093 Mhz)
MEMORY: 6 GB
PID: 28479
COMMAND: "crash"
TASK: fffff8800ad5faa00 [THREAD_INFO: fffff8800799ec000]
CPU: 0
STATE: TASK_RUNNING (ACTIVE)
```

Crash
Utilisation

Lancement

crash [noyau corefile]

```
crash
KERNEL: /usr/lib/debug/boot/vmlinux-4.4.0-97-generic
DUMPFILE: /proc/kcore
CPUS: 4
DATE: Sun Nov 26 21:28:48 2017
UPTIME: 12 days, 16:41:39
LOAD AVERAGE: 1.01, 1.19, 1.14
TASKS: 713
NODENAME: dakoro
RELEASE: 4.4.0-97-generic
VERSION: #120-Ubuntu SMP Tue Sep 19 17:28:18 UTC 2017
MACHINE: x86_64 (3093 Mhz)
MEMORY: 6 GB
PID: 28479
COMMAND: "crash"
TASK: ffff8800ad5faa00 [THREAD_INFO: ffff8800799ec000]
CPU: 0
STATE: TASK_RUNNING (ACTIVE)
```

- Version du noyau
- Date de prise du crash
- Nombre de tâches
- Architecture du processeur
- Informations sur la tâche actuelle

Lister l'ensemble des tâches du système

`ps [PID | task | commande]`

Lister l'ensemble des tâches du système

ps [PID | task | commande]

PID	PPID	CPU	TASK	ST	%MEM	VSZ	ps	RSS	COMM
0	0	0	ffffffffff81e11500	RU	0.0	0	0	0	[swapper/0]
0	0	1	ffff8801b7a38e00	RU	0.0	0	0	0	[swapper/1]
0	0	2	ffff8801b7a39c00	RU	0.0	0	0	0	[swapper/2]
0	0	3	ffff8801b7a3aa00	RU	0.0	0	0	0	[swapper/3]
1	0	1	ffff8801b79e8000	IN	0.1	185492	4296	systemd	
...									
964	1	0	ffff8801b2d79c00	IN	0.0	44920	1556	avahi-daemon	
970	1	1	ffff8801b34e8e00	IN	0.0	26044	184	atd	
972	1	0	ffff8801b34e9c00	IN	0.1	138412	3676	freshclam	
974	1	2	ffff8801b34e8000	IN	0.0	166456	2012	thermald	
990	1	3	ffff8801b53a8e00	IN	0.0	276204	2268	accounts-daemon	
997	1	3	ffff8801b53ab800	IN	0.0	256396	1220	rsyslogd	
1009	1	2	ffff8800b5f30000	IN	0.0	44332	3204	dbus-daemon	
1052	1	1	ffff8800b5f34600	IN	0.0	495104	1020	osspd	
1053	1	1	ffff8800b5f35400	IN	0.0	495104	1020	osspd	
1054	1	3	ffff8800b5f32a00	IN	0.0	495104	1020	osspd	
1055	1	3	ffff8800b5f36200	IN	0.0	495104	1020	osspd	

Lister l'ensemble des tâches du système

`ps [PID | task | commande]`

PID	PPID	CPU	TASK	ST	%MEM	VSZ	ps	RSS	COMM
0	0	0	ffffffffff81e11500	RU	0.0	0		0	[swapper/0]
0	0	1	ffff8801b7a38e00	RU	0.0	0		0	[swapper/1]
0	0	2	ffff8801b7a39c00	RU	0.0	0		0	[swapper/2]
0	0	3	ffff8801b7a3aa00	RU	0.0	0		0	[swapper/3]
1	0	1	ffff8801b79e8000	IN	0.1	185492	4296	systemd	
...									
964	1	0	ffff8801b2d79c00	IN	0.0	44920	1556	avahi-daemon	
970	1	1	ffff8801b34e8e00	IN	0.0	26044	184	atd	
972	1	0	ffff8801b34e9c00	IN	0.1	138412	3676	freshclam	
974	1	2	ffff8801b34e8000	IN	0.0	166456	2012	thermald	
990	1	3	ffff8801b53a8e00	IN	0.0	276204	2268	accounts-daemon	
997	1	3	ffff8801b53ab800	IN	0.0	256396	1220	rsyslogd	
1009	1	2	ffff8800b5f30000	IN	0.0	44332	3204	dbus-daemon	
1052	1	1	ffff8800b5f34600	IN	0.0	495104	1020	osspd	
1053	1	1	ffff8800b5f35400	IN	0.0	495104	1020	osspd	
1054	1	3	ffff8800b5f32a00	IN	0.0	495104	1020	osspd	
1055	1	3	ffff8800b5f36200	IN	0.0	495104	1020	osspd	

- Affichage de *PID PPID*
- Adresse de la *task_struct*
- Status de la tâche
- Mémoire virtuellement disponible
- Mémoire effectivement utilisée
- Ligne de commande

Choisir une autre tâche

`set [PID | task]`

- Par défaut l'ensemble des commandes prennent la tâche courante comme référence
- Pour changer de tâche courante, on utilise la commande `set`
- La sélection de la tâche se fait par *PID* ou par adresse de *task_struct*

Choisir une autre tâche

set [PID | task]

- Par défaut l'ensemble des commandes prennent la tâche courante comme référence
- Pour changer de tâche courante, on utilise la commande *set*
- La sélection de la tâche se fait par *PID* ou par adresse de *task_struct*

```
crash> set 1
      PID: 1
COMMAND: "systemd"
  TASK: ffff8801b79e8000  [THREAD_INFO: ffff8801b79f0000]
    CPU: 1
  STATE: TASK_INTERRUPTIBLE
```

Afficher la pile d'exécution d'une tâche

bt [PID]

- Affiche la pile d'exécution d'une tâche.
- Avec l'option *-f* affiche le contenu complet de la pile.
- C'est cette fonction qui rend crash incontournable pour comprendre ce qu'il se passe sur le système.

Afficher la pile d'exécution d'une tâche

bt [PID]

- Affiche la pile d'exécution d'une tâche.
- Avec l'option *-f* affiche le contenu complet de la pile.
- C'est cette fonction qui rend crash incontournable pour comprendre ce qu'il se passe sur le système.

```
crash> bt
PID: 1      TASK: ffff8801b79e8000  CPU: 0      COMMAND: "systemd"
#0 [ffff8801b79f3d40] __schedule at ffffffff8183efce
#1 [ffff8801b79f3d90] schedule at ffffffff8183f6b5
#2 [ffff8801b79f3da8] schedule_hrtimeout_range_clock at ffffffff81842ca3
#3 [ffff8801b79f3e50] schedule_hrtimeout_range at ffffffff81842cd3
#4 [ffff8801b79f3e60] ep_poll at ffffffff81259c40
#5 [ffff8801b79f3f10] sys_epoll_wait at ffffffff8125af28
#6 [ffff8801b79f3f50] entry_SYSCALL_64_fastpath at ffffffff818437f2
RIP: 00007f5e7aa689d3  RSP: 00007fff1c6f6470  RFLAGS: 00000293
RAX: ffffffffffffd8  RBX: 00005610aabade00  RCX: 00007f5e7aa689d3
RDX: 0000000000000000  RSI: 00007fff1c6f6480  RDI: 0000000000000004
RBP: 0000000000000000  R8: 00007fff1c6f6480  R9: 225870c9894f4527
R10: 00000000fffffff8  R11: 0000000000000293  R12: 0000000000000000
R13: 00007fff1c6f4810  R14: 00007fff1c6f4820  R15: 00005610a8ad28e3
ORIG_RAX: 00000000000000e8  CS: 0033  SS: 002b
```

Afficher le code assembleur d'une fonction

dis fonction

- Permet de comprendre l'enchaînement de la pile d'appel

Afficher le code assembleur d'une fonction

dis fonction

- Permet de comprendre l'enchaînement de la pile d'appel

```
dis
crash> dis schedule
0xffffffff8183f680 <schedule>: callq 0xffffffff818460c0 <ftrace_graph_caller>
0xffffffff8183f685 <schedule+5>:    push  %rbp
0xffffffff8183f686 <schedule+6>:    mov   %gs:0xd400,%rax
0xffffffff8183f68f <schedule+15>:   mov   %rsp,%rbp
0xffffffff8183f692 <schedule+18>:   push  %rbx
0xffffffff8183f693 <schedule+19>:   mov   (%rax),%rdx
0xffffffff8183f696 <schedule+22>:   test  %rdx,%rdx
0xffffffff8183f699 <schedule+25>:   je    0xffffffff8183f6a5 <schedule+37>
0xffffffff8183f69b <schedule+27>:   cmpq $0x0,0x6f0(%rax)
0xffffffff8183f6a3 <schedule+35>:   je    0xffffffff8183f6c3 <schedule+67>
0xffffffff8183f6a5 <schedule+37>:   mov   %gs:0x14304,%rbx
0xffffffff8183f6aa <schedule+46>:   xor   %edi,%edi
0xffffffff8183f6b0 <schedule+48>:   callq 0xffffffff8183ec50 <__schedule>
0xffffffff8183f6b5 <schedule+53>:   mov   -0x3ff8(%rbx),%rax
0xffffffff8183f6bc <schedule+60>:   test  $0x8,%al
0xffffffff8183f6be <schedule+62>:   jne   0xffffffff8183f6ae <schedule+46>
0xffffffff8183f6c0 <schedule+64>:   pop   %rbx
0xffffffff8183f6c1 <schedule+65>:   pop   %rbp
0xffffffff8183f6c2 <schedule+66>:   retq
```

Afficher le code assembleur d'une fonction

dis fonction

- Permet de comprendre l'enchaînement de la pile d'appel

```
dis
crash> dis schedule
0xffffffff8183f680 <schedule>: callq 0xffffffff818460c0 <ftrace_graph_caller>
0xffffffff8183f685 <schedule+5>:    push  %rbp
0xffffffff8183f686 <schedule+6>:    mov   %gs:0xd400,%rax
0xffffffff8183f68f <schedule+15>:   mov   %rsp,%rbp
0xffffffff8183f692 <schedule+18>:   push  %rbx
0xffffffff8183f693 <schedule+19>:   mov   (%rax),%rdx
0xffffffff8183f696 <schedule+22>:   test  %rdx,%rdx
0xffffffff8183f699 <schedule+25>:   je    0xffffffff8183f6a5 <schedule+37>
0xffffffff8183f69b <schedule+27>:   cmpq $0x0,0x6f0(%rax)
0xffffffff8183f6a3 <schedule+35>:   je    0xffffffff8183f6c3 <schedule+67>
0xffffffff8183f6a5 <schedule+37>:   mov   %gs:0x14304,%rbx
0xffffffff8183f6aa <schedule+46>:   xor   %edi,%edi
0xffffffff8183f6b0 <schedule+48>:   callq 0xffffffff8183ec50 <__schedule>
0xffffffff8183f6b5 <schedule+53>:   mov   -0x3ff8(%rbx),%rax
0xffffffff8183f6bc <schedule+60>:   test  $0x8,%al
0xffffffff8183f6be <schedule+62>:   jne   0xffffffff8183f6ae <schedule+46>
0xffffffff8183f6c0 <schedule+64>:   pop   %rbx
0xffffffff8183f6c1 <schedule+65>:   pop   %rbp
0xffffffff8183f6c2 <schedule+66>:   retq
```

- Adresse de l'instruction
- Offset de l'instruction
- Code assembleur

Afficher la définition d'un symbole

what is symbol

- Donne la définition d'une structure
- Donne le type d'un symbole
- Donne le prototype d'une fonction

Afficher la définition d'un symbole

whatis symbole

- Donne la définition d'une structure
- Donne le type d'un symbole
- Donne le prototype d'une fonction

```
crash> whatis struct mm_struct
struct mm_struct {
    struct vm_area_struct *mmap;
...
    unsigned long mmap_base;
    unsigned long mmap_legacy_base;
...
    unsigned long start_code;
    unsigned long end_code;
    unsigned long start_data;
    unsigned long end_data;
    unsigned long start_brk;
    unsigned long brk;
    unsigned long start_stack;
    unsigned long arg_start;
    unsigned long arg_end;
...
    struct uprobes_state uprobes_state;
    void *bd_addr;
    atomic_long_t hugetlb_usage;
}
SIZE: 968
```

Afficher la définition d'un symbole

`print[/format] symbole`

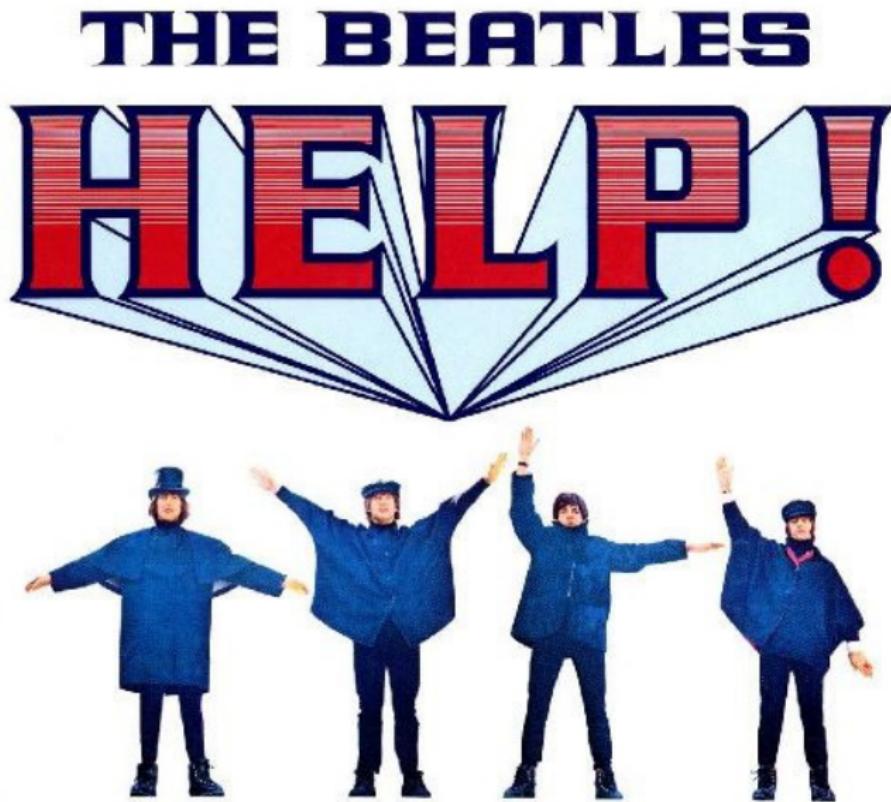
- Affiche la valeur d'un symbole
- Si `/format` est spécifié, affiche toutes les valeurs dans ce format (x : hexadécimal, d : décimal).
- Pour les structures, la commande `struct` est plus rapide à utiliser

Afficher la définition d'un symbole

`print[/format] symbole`

- Affiche la valeur d'un symbole
- Si `/format` est spécifié, affiche toutes les valeurs dans ce format (x : hexadecimal, d :décimal).
- Pour les structures, la commande `struct` est plus rapide à utiliser

```
crash> print (struct list_head) modules
$11 = {
  next = 0xfffffffffc09f2508,
  prev = 0xfffffffffc00052c8
}
crash> print/x modules
$12 = {
  next = 0xfffffffffc09f2508,
  prev = 0xfffffffffc00052c8
}
crash> print/d modules
$13 = {
  next = -1063312120,
  prev = -1073720632
}
crash> print (struct list_head *) modules
$14 = (struct list_head *) 0xfffffffffc09f2508
```



Afficher l'aide en ligne d'une commande

help commande

- Affiche l'aide et les options de la commande

Afficher l'aide en ligne d'une commande

help commande

- Affiche l'aide et les options de la commande

NAME
`help` - get help

SYNOPSIS
`help` [`command` | `all`] [`-<option>`]

DESCRIPTION
When entered with no argument, a list of all currently available crash commands is listed. If a name of a crash `command` is entered, a man-like page for the `command` is displayed. If "all" is entered, `help` pages for all commands will be displayed. If neither of the above is entered, the argument string will be passed on to the `gdb help` command.

A number of internal debug, statistical, and other dumpfile related data is available with the following options:

`-a` - `alias` data
`-b` - shared buffer data
`-B` - build data

...

Et il y en a encore!!

- **foreach**

Applique la commande sur l'ensemble des processus passés en argument.

- **mod**

Manipule les modules externes du noyau.

- **kmem**

Permet de récupérer les informations sur les structures mémoire du noyau.

- **rd**

Lit brutalement la mémoire.

- **files**

Liste les fichiers du processus courant

- **net**

Manipule les interfaces réseaux.

- **gdb**

Crash permet d'appeler des fonctions gdb.

- **gdb list**

Affiche le source d'une fonction.

- **gdb set**

Manipule la mémoire du noyau (DANGEUREUX !!).

Berkley Packet Filter

Les outils BCC basés sur BPF sont les derniers outils de tracing des fonctions kernel. Ils permettent de façon très efficace de visualiser les fonctions kernel sollicitées à un instant t .

SystemTap

SystèmeTap permet de modifier le comportement du noyau en insérant du code à n'importe quelle adresse.

- Modification de modules
- Visualisation de variables et de structures
- ...

Debugger
oooooo

Les instructions
oooooooo

Processus utilisateur
oooooooooooo

Noyau
oooooooooooooooooooooooooooo

Question
●○

Sommaire

1 Debugger

2 Les instructions

3 Processus utilisateur

4 Noyau

5 Question

Debugger
ooooo

Les instructions
ooooooo

Processus utilisateur
oooooooooooo

Noyau
oooooooooooooooooooo

Question
o●

Questions

