DANS LES ENTRAILLES DU LANGAGE C

Notions de bases pour comprendre les technique d’exploit en C

ATTENTION !

Cette présentation s'appuie en grande partie sur l'excellent livre de Jon Erickson "Hacking, the art of exploitation".

Les programmes en C qui seront écrits puis étudiés ne sont que le prétexte à un décorticage un peu poussé du code.

Cette présentation est un cours introductif à une seconde session de pratique d'exploit en C sur un vieux noyau Linux .

SOMMAIRE

- **Back to the basics**
- **Concepts de C à travers le scope du deboggueur**
- **Memory segmentation**

Back to the basics

- ✧ Programmes simples en C
- ✧ Le GNU Compiler Collection (GCC)
- ✧ Le processeur x86 (Intel)
- ✧ Instructions assembleur pour x86
- ✧ Quelques commandes utiles pour GDB

➢ Concepts de C à travers le scope du deboggueur

➢ Memory segmentation

Back to the basics

➢ Concepts de C à travers le scope du deboggueur

- ✧ Chaînes de caractères (*strings*)
- ✧ *Signedness*
- ✧ Pointeurs
- ✧ *Typecasting*
- ✧ Arguments en ligne de commande
- ✧ Portée des variables

➢ Memory segmentation

Back to the basics

➢ Concepts de C à travers le scope du deboggueur

➢ Memory segmentation

- ✧ Segments de la mémoire
- ✧ Notion de *stack* (« pile » mémoire)
- ✧ Notion de *heap* (« tas »)
- ✧ Principes des techniques d'overflow

La prochaine fois !

BACK TO THE BASICS

CODING TIME !

Ecrire un programme en C pour

- ➢ Afficher 10 fois la chaine de caractères "Hello Word !"

- ➢ A la compilation: ne pas oublier l'option –g
(utile ultérieurement pour le debug)

Un programme simple en C

```
#include <stdio.h>

int main(){
        int i;
        for (i = 0; i < 10; i++){
                printf("Hello world !\n");
        }
        return 0;
}
```

Compiler le programme pour l'éxecuter

Avec le GNU Compiler Collection (GCC):

gcc -g prog.c -> a.out*

ou

gcc –g prog.c –o prog -> prog*

Le rôle du compilateur

C'est un traducteur de code d'un langage (langage source) vers un autre (langage cible).

Exemples:
- GCC: GNU Compiler Collection pour le C, C++ essentiellement
- Javac, compilateur Java
- Clang pour langages de la famille C
- GHC pour le Haskell

En ce qui nous concerne, le langage cible est le langage compréhensible par l'ordinateur pour qu'il exécute nos commandes.
- ➢ Langage assembleur

Optimisation du code par le compilateur

Le compilateur va généralement chercher à optimiser le code cible:

- Amélioration de la vitesse d'exécution (ré-ordonnancement des instructions)
- Réduction de l'occupation mémoire du programme

❖ Que dit man gcc ?

Without any optimization option, the compiler's goal is to reduce the cost of compilation and to make debugging produce the expected results.

Turning on optimization flags makes the compiler attempt to improve the performance and/or code size at the expense of compilation time and possibly the ability to debug the program.

Options: –O0 (default) –O1 –O2 –O3...

Compilateur vs interpréteur

Certains langages ne sont pas compilés. L'interpréteur lit et analyse un "script" pour en exécuter les instructions à la volée.

Exemples:

- bash
- SQL

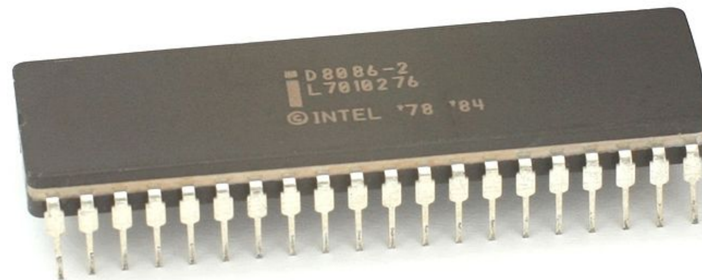
En Unix, le shell est un interpréteur de ligne de commande.

Le langage assembleur pour x86

Sur la plupart des ordinateurs, le processeur qui exécute les instructions appartient à la famille x86.

Il s'agit de la famille de processeurs Intel IA 32 bits, basée sur le premier microprocesseur de ce type d'architecture: le Intel 8086.

Le langage assembleur x86 est un jeu d'instructions compatibles avec les processeurs de la famille x86.



DISASSEMBLING TIME !

Sur le programme précédent:

- Visualiser le fichier exécutable simple_prog

Comment afficher le programme en langage assembleur ?

- Avec GDB:

  # gdb simple_prog
  (gdb) list
  (gdb) disassemble main

- Avec objdump

  #objdump –M intel –D simple_prog

Notre programme en langage assembleur (Linux 2.6.40.3-0.fc15.i686)

Syntaxe AT & T:

```
0x080483b4 <+0>:     push   %ebp
0x080483b5 <+1>:     mov    %esp,%ebp
0x080483b7 <+3>:     and    $0xfffffff0,%esp
0x080483ba <+6>:     sub    $0x20,%esp
0x080483bd <+9>:     movl   $0x0,0x1c(%esp)
0x080483c5 <+17>:    jmp    0x80483d8 <main+36>
0x080483c7 <+19>:    movl   $0x80484d0,(%esp)
0x080483ce <+26>:    call   0x80482f0 <puts@plt>
0x080483d3 <+31>:    addl   $0x1,0x1c(%esp)
0x080483d8 <+36>:    cmpl   $0x9,0x1c(%esp)
0x080483dd <+41>:    jle    0x80483c7 <main+19>
...
```

Notre programme en langage assembleur (Linux 2.6.40.3-0.fc15.i686)

Syntaxe AT & T:

```
0x080483b4 <+0>:     push   %ebp
0x080483b5 <+1>:     mov    %esp,%ebp
0x080483b7 <+3>:     and    $0xfffffff0,%esp
0x080483ba <+6>:     sub    $0x20,%esp
0x080483bd <+9>:     movl   $0x0,0x1c(%esp)
0x080483c5 <+17>:    jmp    0x80483d8 <main+36>
0x080483c7 <+19>:    movl   $0x80484d0,(%esp)
0x080483ce <+26>:    call   0x80482f0 <puts@plt>
0x080483d3 <+31>:    addl   $0x1,0x1c(%esp)
0x080483d8 <+36>:    cmpl   $0x9,0x1c(%esp)
0x080483dd <+41>:    jle    0x80483c7 <main+19>
...
```

Les registres de l'architecture x86

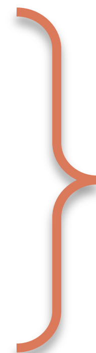
- (gdb) info registers

EAX – Accumulator register

ECX – Counter register

EDX – Data register

EBX – Base register



En général, ils stockent temporairement des variables pour le CPU.

ESP – Stack Pointer

EBP – Base Pointer

Enregistre des adresses 32-bits

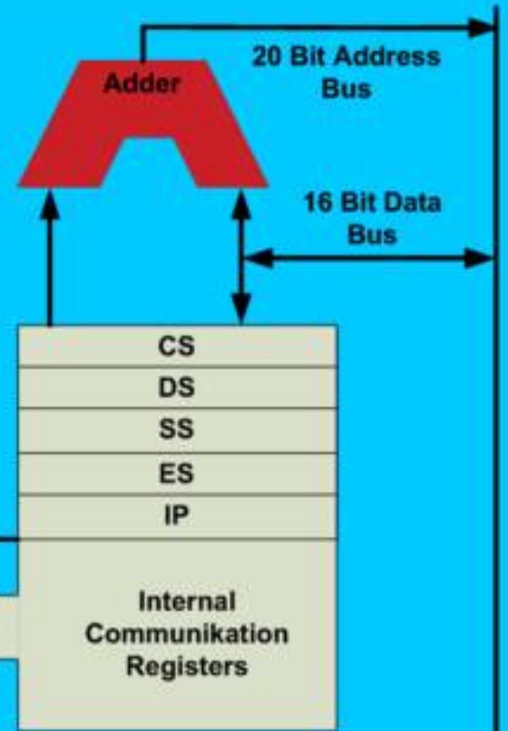
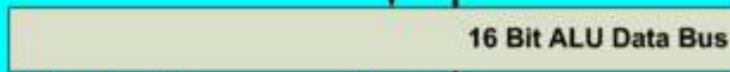
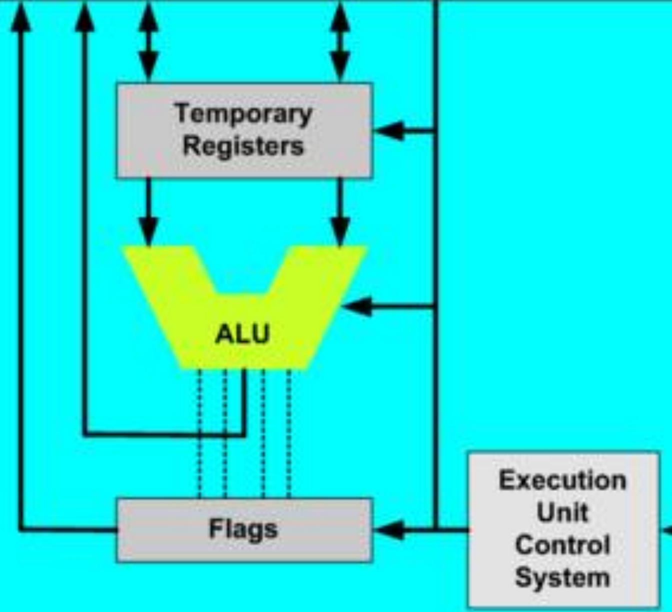
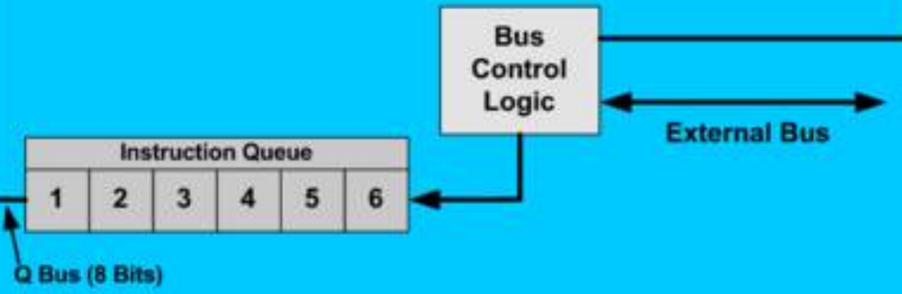
ESI – Source Index

EDI – Destination Index

Pointeurs utilisés quand des données doivent être lues ou écrites à un emplacement donné.

EIP – Instruction Pointer

Pointe à l'instruction courante (que le processeur est en train de lire)

General Registers
AH
AL
BH
BL
CH
CL
DH
DL
SP
BP
SI
DI

Adder
20 Bit Address Bus
16 Bit Data Bus
CS
DS
SS
ES
IP
Internal Communikation Registers

16 Bit ALU Data Bus

Temporary Registers
ALU
Flags

Execution Unit Control System
Instruction Queue
1
2
3
4
5
6
Q Bus (8 Bits)
Bus Control Logic
External Bus
Execution Unit
Bus Interface Unit

Notre programme en langage assembleur (Linux 2.6.40.3-0.fc15.i686)

Syntaxe AT & T:

```
0x080483b4 <+0>:     push   %ebp
0x080483b5 <+1>:     mov    %esp,%ebp
0x080483b7 <+3>:     and    $0xfffffff0,%esp
0x080483ba <+6>:     sub    $0x20,%esp
0x080483bd <+9>:     movl   $0x0,0x1c(%esp)
0x080483c5 <+17>:    jmp    0x80483d8 <main+36>
0x080483c7 <+19>:    movl   $0x80484d0,(%esp)
0x080483ce <+26>:    call   0x80482f0 <puts@plt>
```

- **Syntaxe: operation *<source>*, *<destination>***

  mov %esp, %ebp

  copie le contenu de esp (source) dans ebp (destination)

Notre programme en langage assembleur (Linux 2.6.40.3-0.fc15.i686)

Syntaxe Intel: plus claire à déchiffrer dans un premier temps

```
0x080483b4 <+0>:     push   ebp
0x080483b5 <+1>:     mov    ebp,esp
0x080483b7 <+3>:     and    esp,0xfffffff0
0x080483ba <+6>:     sub    esp,0x20
0x080483bd <+9>:     mov    DWORD PTR [esp+0x1c],0x0
0x080483c5 <+17>:    jmp    0x80483d8 <main+36>
0x080483c7 <+19>:    mov    DWORD PTR [esp],0x80484d0
0x080483ce <+26>:    call   0x80482f0 <puts@plt>
```

- Syntaxe: **operation *<destination>, <source>***

  mov ebp, esp

  copie le contenu de esp (source) dans ebp (destination)

Configurer GDB pour utiliser la syntaxe Intel

- A chaque lancement:

```
$ gdb
(gdb) set disassembly-flavor intel
```

- Dans .gdbinit:

```
$ echo "set disassembly-flavor intel" > ~/.gdbinit
```

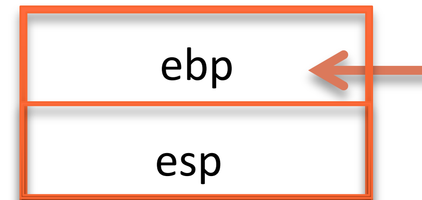
Décomposons un petit peu tout ça... (code Linux)

```
(gdb) disass main
Dump of assembler code for function main:
 0x080483b4 <+0>:    push   ebp
 0x080483b5 <+1>:    mov    ebp,esp
 0x080483b7 <+3>:    and    esp,0xfffffff0
 0x080483ba <+6>:    sub    esp,0x20
 0x080483bd <+9>:    mov    DWORD PTR [esp+0x1c],0x0
 0x080483c5 <+17>:   jmp    0x80483d8 <main+36>
```

On sauve les infos !

- On sauve l'adresse pointée esp dans ebp.
- ✓ Il faut savoir où revenir une fois fini le programme !

En image...

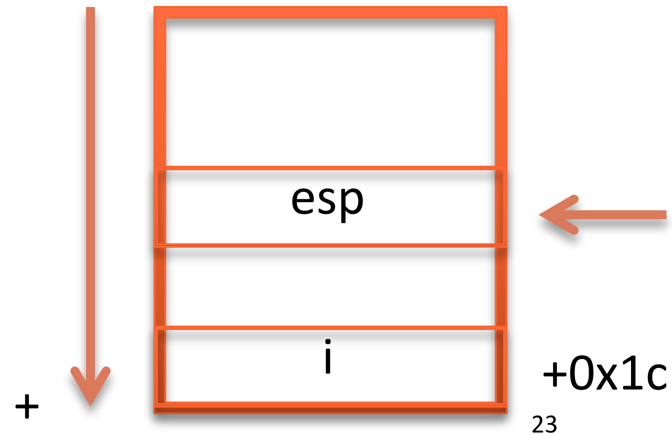


Décomposons un petit peu tout ça...

```
(gdb) disass main
Dump of assembler code for function main:
 0x080483b4 <+0>:    push   ebp
 0x080483b5 <+1>:    mov    ebp,esp
 0x080483b7 <+3>:    and    esp,0xfffffff0
 0x080483ba <+6>:    sub    esp,0x20
 0x080483bd <+9>:    mov    DWORD PTR [esp+0x1c],0x0
 0x080483c5 <+17>:   jmp    0x80483d8 <main+36>
```

Allocation de mémoire

- On soustraire 0x20 à esp, et on enregistre la valeur dans esp
- On met la valeur esp+0x1c à 0: déclaration de la variable i
- On saute à l'instruction <main+36>



Décomposons un petit peu tout ça...

```
0x080483bd <+9>:  mov    DWORD PTR [esp+0x1c],0x0
0x080483c5 <+17>:        jmp    0x80483d8 <main+36>
0x080483c7 <+19>:        mov    DWORD PTR [esp],0x80484d0
0x080483ce <+26>:        call   0x80482f0 <puts@plt>
0x080483d3 <+31>:        add    DWORD PTR [esp+0x1c],0x1
0x080483d8 <+36>:        cmp    DWORD PTR [esp+0x1c],0x9
0x080483dd <+41>:        jle    0x80483c7 <main+19>
0x080483df <+43>:        mov    eax,0x0
0x080483e4 <+48>:        leave
```

La boucle for

- Comparaison dans la boucle for

Décomposons un petit peu tout ça...

```
0x080483bd <+9>:  mov     DWORD PTR [esp+0x1c],0x0
0x080483c5 <+17>:         jmp     0x80483d8 <main+36>
0x080483c7 <+19>:         mov     DWORD PTR [esp],0x80484d0
0x080483ce <+26>:         call    0x80482f0 <puts@plt>
0x080483d3 <+31>:         add     DWORD PTR [esp+0x1c],0x1
0x080483d8 <+36>:         cmp     DWORD PTR [esp+0x1c],0x9
0x080483dd <+41>:         jle     0x80483c7 <main+19>
0x080483df <+43>:         mov     eax,0x0
0x080483e4 <+48>:         leave
```

Exécution de la boucle for

- Appel de la fonction printf()
- On incrémente i pour le prochain tour.

Décomposons un petit peu tout ça...

```
0x080483c7 <+19>:    mov    DWORD PTR [esp],0x80484d0
0x080483ce <+26>:    call   0x80482f0 <puts@plt>
0x080483d3 <+31>:    add    DWORD PTR [esp+0x1c],0x1
0x080483d8 <+36>:    cmp    DWORD PTR [esp+0x1c],0x9
0x080483dd <+41>:    jle    0x80483c7 <main+19>
0x080483df <+43>:    mov    eax,0x0
0x080483e4 <+48>:    leave
0x080483e5 <+49>:    ret
```

Sortie du programme

- leave: mov esp, ebp
- ret: "return from procedure", à une adresse indiquée en haut de la pile

GDB TIME !

On va suivre le déroulement pas à pas de simple_prog

- But: comprendre le rôle de certains registres

GDB TIME !

- **List *fonction*** : affiche les lignes de code du programme

- **break *n*** : pose un point d'arrêt à la ligne n ou à la fonction n.
- **run** : exécute le programme en entier ou jusqu'au prochain break

- **step** : exécute l'instruction suivante (mode pas-à-pas)
  nexti : exécute l'instruction courante

- **info r** ou **info register**: affiche l'état des registres d'instructions
- **print *exp***: affiche la valeur de exp
- **info addr *exp***: affiche l'addresse de exp

La commande *examine*

- Intérêt: examiner l'état de la mémoire

- Syntaxe de la commande: x/*format addr*
  - o : en octal
  - x : en hexadécimal
  - u : unsigned (base décimal standard)
  - t : en binaire

Suivre l'état de la mémoire de EIP avec x

- Le registre EIP (Instruction Pointer) contient les adresses mémoire qui pointent vers les instructions du programme en cours d'exécution.

```
(gdb) i r eip          # raccourci pour info register eip
(gdb) x/x $eip         # $eip désigne l'adresse contenue dans eip
```

- On peut afficher plusieurs "mots" (un mot = 4 octets):

```
(gdb) x/2x $eip
```

- L'option i permet d'afficher la mémoire comme une instruction de langage assembleur:

```
(gdb) x/i $eip
```

Suivre l'état de la mémoire de EIP avec x

- On peut moduler la taille des unités d'adresses affichées (par défaut, un mot) :
  - b : 1 octet
  - h : 2 octets (un demi-mot ou halfword ou short)
  - w : 1 mot (word)
  - g : 8 octets (giant word)

Attention ! Un mot peut aussi désigner 2 octets.

Un double word ou DWORD désigne alors 4 octets.

```
(gdb) x/8xb $eip
(gdb) x/8xw $eip
```

Petit Indien ou Grand Indien

(gdb) x/8xb $eip

0x1fc3 <main+13>: 0x**c7** 0x45 0xf4 0x00 0x00 0x00 0x00 0xeb

(gdb) x/xw $eip

0x1fc3 <main+13>: 0x00f445**c7**

- Sur les processeurs x86, les valeurs sont stockées en **little-endian**: les octets de poids faible sont stockés en premier.

  (poids faible == de puissances les plus petites dans la base de notation)

/!\ Sur le réseau, c'est le contraire: valeurs stockées **en big-endian** !

Déclaration de la variable i

Chaîne ASCII

```
0x080483c7 <+19>: mov    DWORD PTR [esp],0x80484d0
```

(gdb) x/8xb 0x80484d0

(gdb) x/6ub 0x80484d0

- Table ASCII

Dec	Hx	Oct	Char	
0	0	000	NUL	(null)
1	1	001	SOH	(start of heading)
2	2	002	STX	(start of text)
3	3	003	ETX	(end of text)
4	4	004	EOT	(end of transmission)
5	5	005	ENQ	(enquiry)
6	6	006	ACK	(acknowledge)
7	7	007	BEL	(bell)
8	8	010	BS	(backspace)
9	9	011	TAB	(horizontal tab)
10	A	012	LF	(NL line feed, new line)
11	B	013	VT	(vertical tab)
12	C	014	FF	(NP form feed, new page)
13	D	015	CR	(carriage return)
14	E	016	SO	(shift out)
15	F	017	SI	(shift in)
16	10	020	DLE	(data link escape)
17	11	021	DC1	(device control 1)
18	12	022	DC2	(device control 2)
19	13	023	DC3	(device control 3)
20	14	024	DC4	(device control 4)
21	15	025	NAK	(negative acknowledge)
22	16	026	SYN	(synchronous idle)
23	17	027	ETB	(end of trans. block)
24	18	030	CAN	(cancel)
25	19	031	EM	(end of medium)
26	1A	032	SUB	(substitute)
27	1B	033	ESC	(escape)
28	1C	034	FS	(file separator)
29	1D	035	GS	(group separator)
30	1E	036	RS	(record separator)
31	1F	037	US	(unit separator)

Dec	Hx	Oct	Html	Chr
32	20	040		Space
33	21	041	!	!
34	22	042	"	"
35	23	043	#	#
36	24	044	$	$
37	25	045	%	%
38	26	046	&	&
39	27	047	'	'
40	28	050	(	(
41	29	051	)	)
42	2A	052	*	*
43	2B	053	+	+
44	2C	054	,	,
45	2D	055	-	-
46	2E	056	.	.
47	2F	057	/	/
48	30	060	0	0
49	31	061	1	1
50	32	062	2	2
51	33	063	3	3
52	34	064	4	4
53	35	065	5	5
54	36	066	6	6
55	37	067	7	7
56	38	070	8	8
57	39	071	9	9
58	3A	072	:	:
59	3B	073	;	;
60	3C	074	<	<
61	3D	075	=	=
62	3E	076	>	>
63	3F	077	?	?

Dec	Hx	Oct	Html	Chr
64	40	100	@	@
65	41	101	A	A
66	42	102	B	B
67	43	103	C	C
68	44	104	D	D
69	45	105	E	E
70	46	106	F	F
71	47	107	G	G
72	48	110	H	H
73	49	111	I	I
74	4A	112	J	J
75	4B	113	K	K
76	4C	114	L	L
77	4D	115	M	M
78	4E	116	N	N
79	4F	117	O	O
80	50	120	P	P
81	51	121	Q	Q
82	52	122	R	R
83	53	123	S	S
84	54	124	T	T
85	55	125	U	U
86	56	126	V	V
87	57	127	W	W
88	58	130	X	X
89	59	131	Y	Y
90	5A	132	Z	Z
91	5B	133	[	[
92	5C	134	\	\
93	5D	135	]	]
94	5E	136	^	^
95	5F	137	_	_

Dec	Hx	Oct	Html	Chr
96	60	140	`	`
97	61	141	a	a
98	62	142	b	b
99	63	143	c	c
100	64	144	d	d
101	65	145	e	e
102	66	146	f	f
103	67	147	g	g
104	68	150	h	h
105	69	151	i	i
106	6A	152	j	j
107	6B	153	k	k
108	6C	154	l	l
109	6D	155	m	m
110	6E	156	n	n
111	6F	157	o	o
112	70	160	p	p
113	71	161	q	q
114	72	162	r	r
115	73	163	s	s
116	74	164	t	t
117	75	165	u	u
118	76	166	v	v
119	77	167	w	w
120	78	170	x	x
121	79	171	y	y
122	7A	172	z	z
123	7B	173	{	{
124	7C	174			|
125	7D	175	}	}
126	7E	176	~	~
127	7F	177		DEL

www.LookupTables.com

Source: www.LookupTables.com

Chaîne ASCII


```
0x080483c7 <+19>: mov    DWORD PTR [esp],0x80484d0
```

(gdb) x/8xb 0x80484d0

(gdb) x/6ub 0x80484d0

- Table ASCII

- Directement avec GDB:

(gdb) x/6cb 0x80484d0 # regarde la correspondance octet par octet

(gdb) x/s 0x80484d0 # affiche directement la chaîne de caractères

Un programme simple en C

```
#include <stdio.h>

int main(){
	int i;
	for (i = 0; i < 10; i++){
		printf("Hello world !\n");
	}
	return 0;
}
```

La librairie stdio.h

Standart Input/Output library

- /usr/include/stdio.h

Contient le prototype de la fonction printf()

Accéder au code de printf() ?

- Où est le code de printf() ?
- Comment y accéder ?

Trouver le code source de printf()

Le code de printf() se trouve dans des paquets précompilés de la libc.

- Il faut aller regarder par exemple les sources de la **glibc**.
  - Télécharger les sources: http://ftp.gnu.org/gnu/glibc/
  - $ apt-get source glibc-devel

- Le code devient rapidement complexe et long.

Trouver le code source de printf()

Le code de printf() se trouve dans des paquets précompilés de la libc.

- Il faut aller regarder par exemple les sources de la **glibc**.
  - Télécharger les sources: http://ftp.gnu.org/gnu/glibc/
  - $ apt-get source glibc-devel

Le code devient rapidement complexe et long.

- Une alternative est de regarder dans le code de la **dietlibc**.
  - http://dietlibc.sourcearchive.com/

Retour sur quelques commandes assembleur

Le jeu d'instruction assembleur pour x86 a évolué avec le temps et les architectures.

Instructions initiales pour les 8086/8088: une centaine d'instruction.

not	and	or	xor	neg
push	pop	mov		
jmp	cmp	jle		
add	sub	mul		
inc	dec			
lea				

CONCEPTS DE C A TRAVERS LE SCOPE DU DEBOGGUEUR

CODING TIME – Chaînes de caractères

Ecrire un programme qui créé une chaîne de caractère et qui l' imprime à l'écran.

- Cette chaîne devra être construite caractère par caractère.

Puis avec GDB, désassemblez le code.

CODING TIME – Chaînes de caractères

Ecrire un programme qui créé une chaîne de caractère et qui l' imprime à l'écran.

- Cette chaîne devra être construite caractère par caractère.

Puis avec GDB, désassemblez le code.

- Faire de même en utilisant **strcpy()**.

Tailles de certains types de variables

- **Signed int** : entiers positifs et négatifs
- **Unsigned int** : seulement des entiers positifs

Sur un processeur x86_32bits, il y a 2^32 combinaisons possibles pour un unsigned int (4 294 967 295 possibilités).

Pour un signed int : entre -2 147 483 648 à 2 147 483 647.

- La taille d'un type dépend du type d'architecture.

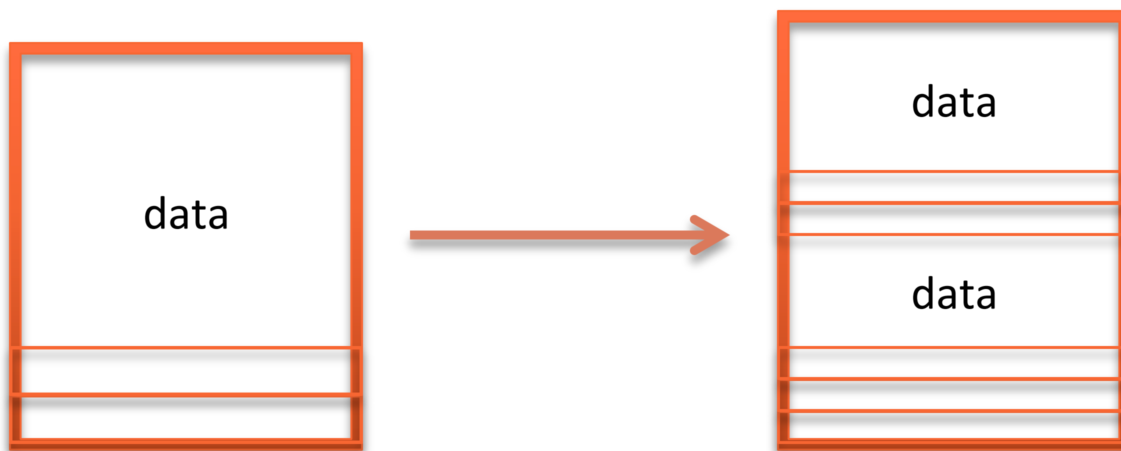
Les pointeurs

- EIP, l'Instruction Pointer, est un pointeur qui pointe vers l'instruction en cours car il contient son adresse mémoire.

- On ne peut pas bouger "réellement" des adresses de mémoires physiques.

  On doit donc copier l'information stockée à cet emplacement mémoire.

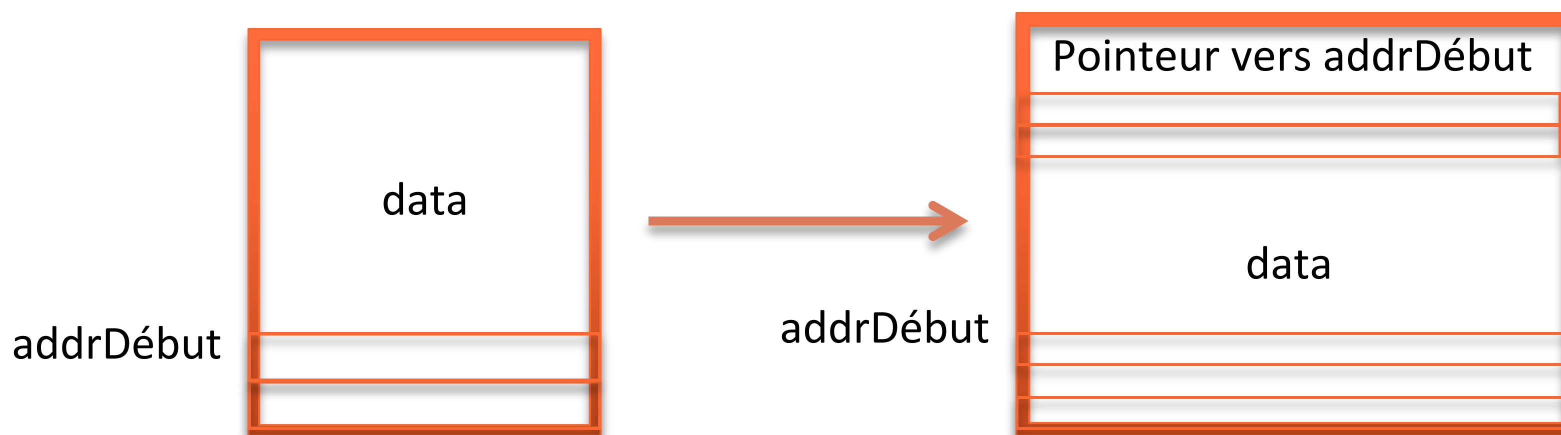
  C'est coûteux : gros chunks de mémoire à allouer/copier.



Les pointeurs

La solution : les pointeurs.

- Au lieu de copier des gros morceaux de mémoire, on utilise l'adresse du début du bloc de mémoire en question.



Les pointeurs

Un pointeur est défini comme quelque chose qui pointe vers une donnée d'un certain type:

```
int *ptr;                # ptr pointe vers une donnée de type int.
```

Les pointeurs

Un pointeur est défini comme quelque chose qui pointe vers une donnée d'un certain type:

int *ptr; # ptr pointe vers une donnée de type int.

Comment voir les données qui sont enregistrées dans le pointeur lui-même ?

- opérateur &

(gdb) print pointer #print la valeur du pointeur (une adresse)

(gdb) print &pointer #print l'adresse du pointeur

- déréférencement *

(gdb) print *pointer #print la valeur des données présentes à l'adresse contenue dans le poin

Typecasting

Typecasting : changement temporaire du type d'une variable, quelque soit le type avec lequel a été défini cette variable.

Syntaxe : (typecast_data_type) variable

CODING TIME !

Ecrire en C un programme qui:

- Définit 2 tableaux de 5 chiffres et 5 lettres;
- Affiche le contenu des deux tableaux (on définira un pointeur pour afficher le contenu dans une boucle for).

GDB TIME !

Afficher l'adresse du premier élément de chacun des tableaux.

Deviner les adresses des éléments suivants ?

Vérifier le résultat avec GDB.

Les types avec les pointeurs

- Le format %d est équivalent à 0x%08x pour afficher des adresses mémoire. (donc pratique pour afficher les valeurs de pointeurs)

On va jouer un peu avec les types de pointeurs: exécuter le programme pointer_types2.c

Corriger le programme !

Les types avec les pointeurs

- Le format %d est équivalent à 0x%08x pour afficher des adresses mémoire. (donc pratique pour afficher les valeurs de pointeurs)

On va jouer un peu avec les types de pointeurs: exécuter le programme pointer_types2.c

Corriger le programme !

2 autres exmples avec pointer_types4.c et pointer_types5.c

Passez des arguments en ligne de commande

- Le format %d est équivalent à 0x%08x pour afficher des adresses mémoire. (donc pratique pour afficher les valeurs de pointeurs)

On va jouer un peu avec les types de pointeurs: exécuter le programme pointer_types2.c

Corriger le programme !

2 autres exmples avec pointer_types4.c et pointer_types5.c

GDB TIME !

On a créé un programme buggué: debug.c

- But du jeu: trouver où est le bug avec GDB et uniquement avec GDB.

NE PAS UTILISER LIST ! On ne doit pas voir le code du programme !

GDB TIME !

- **display *exp*** : équivaut à un **print *exp*** effectué chaque fois que le programme stoppe.
- **delete display *dnums*** : annule l'affichage des variables de numéro *dnums*.

- **show values *n*** : afficher les n dernières valeurs affichées

- **whatis *exp*** : donne le type d'une variable
- **info address *exp***: donne l'adresse de exp

MEMORY SEGMENTATION

Notre programme en langage assembleur (Darwin Kernel Version 9.8.0)

Syntaxe AT & T:

```
0x00001fb6 <main+0>:    push   %ebp
0x00001fb7 <main+1>:    mov    %esp,%ebp
0x00001fb9 <main+3>:    push   %ebx
0x00001fba <main+4>:    sub    $0x24,%esp
0x00001fbd <main+7>:    call   0x1fc2 <main+12>
0x00001fc2 <main+12>:   pop    %ebx
0x00001fc3 <main+13>:   movl   $0x0,-0xc(%ebp)
0x00001fca <main+20>:   jmp    0x1fdf <main+41>
0x00001fcc <main+22>:   lea    0x2e(%ebx),%eax
0x00001fd2 <main+28>:   mov    %eax,(%esp)
0x00001fd5 <main+31>:   call   0x3005 <dyld_stub_puts>
...
```

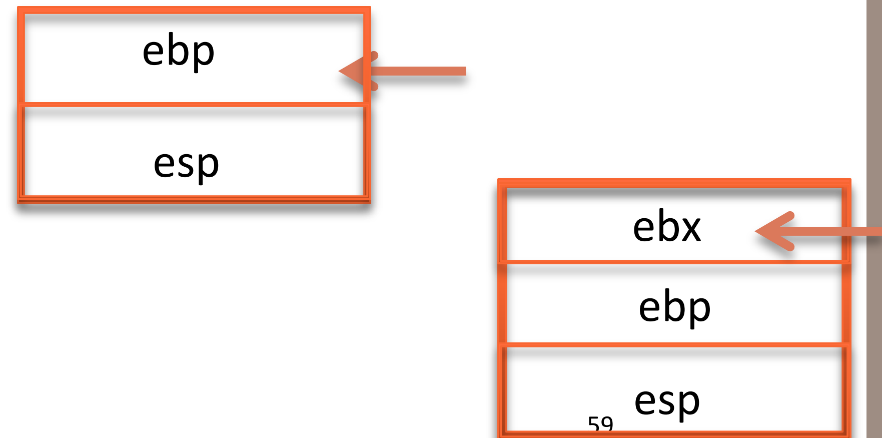
Avec un compilateur/processeur différent:

```
(gdb) disass main
Dump of assembler code for function main:
0x00001fb6 <main+0>:        push   %ebp
0x00001fb7 <main+1>:        mov    %esp,%ebp
0x00001fb9 <main+3>:        push   %ebx
0x00001fba <main+4>:        sub    $0x24,%esp
0x00001fbd <main+7>:        call   0x1fc2 <main+12>
0x00001fc2 <main+12>:       pop    %ebx
0x00001fc3 <main+13>:       movl   $0x0,-0xc(%ebp)
```

On sauve les infos !

- ➢ On sauve l'adresse pointée par ebp dans la stack.
- ✓ Il faut savoir où revenir une fois fini le programme !

En image...



Décomposons un petit peu tout ça...

```
(gdb) disass main
Dump of assembler code for function main:
0x00001fb6 <main+0>:        push   %ebp
0x00001fb7 <main+1>:        mov    %esp,%ebp
0x00001fb9 <main+3>:        push   %ebx
0x00001fba <main+4>:        sub    $0x24,%esp
0x00001fbd <main+7>:        call   0x1fc2 <main+12>
0x00001fc2 <main+12>:       pop    %ebx
0x00001fc3 <main+13>:       movl   $0x0,-0xc(%ebp)
```

Allocation de mémoire

- On alloue de la mémoire
- On appelle l'instruction <main+12>
- On pop ebx.

Remarques

- Ici, on fait intervenir un registre ebx. A priori, difficile de voir à quoi cela sert.
- $ *exp* : valeur *exp*
- % *reg* : adresse du registre *reg*

Décomposons un petit peu tout ça...

```
0x00001fc3 <main+13>:     movl   $0x0,-0xc(%ebp)
0x00001fca <main+20>:     jmp    0x1fdf <main+41>
0x00001fcc <main+22>:     lea    0x2e(%ebx),%eax
0x00001fd2 <main+28>:     mov    %eax,(%esp)
0x00001fd5 <main+31>:     call   0x3005 <dyld_stub_puts>
0x00001fda <main+36>:     lea    -0xc(%ebp),%eax
0x00001fdd <main+39>:     incl   (%eax)
0x00001fdf <main+41>:     cmpl   $0x9,-0xc(%ebp)
0x00001fe3 <main+45>:     jle    0x1fcc <main+22>
```

Question !

Qu'est-ce que -0xc(%ebp) ?

Déclaration de la variable i

- On met 4 octets à 0, 12 cases mémoire au dessus de ebp

Décomposons un petit peu tout ça...

```
0x00001fc3 <main+13>:     movl    $0x0,-0xc(%ebp)
0x00001fca <main+20>:     jmp     0x1fdf <main+41>
0x00001fcc <main+22>:     lea     0x2e(%ebx),%eax
0x00001fd2 <main+28>:     mov     %eax,(%esp)
0x00001fd5 <main+31>:     call    0x3005 <dyld_stub_puts>
0x00001fda <main+36>:     lea     -0xc(%ebp),%eax
0x00001fdd <main+39>:     incl    (%eax)
0x00001fdf <main+41>:     cmpl    $0x9,-0xc(%ebp)
0x00001fe3 <main+45>:     jle     0x1fcc <main+22>
```

La boucle for

- Comparaison dans la boucle for

Remarque

On utilise eax ou ebp au lieu du stack pointer.

Décomposons un petit peu tout ça...

```
0x00001fc3 <main+13>:    movl   $0x0,-0xc(%ebp)
0x00001fca <main+20>:    jmp    0x1fdf <main+41>
0x00001fcc <main+22>:    lea    0x2e(%ebx),%eax
0x00001fd2 <main+28>:    mov    %eax,(%esp)
0x00001fd5 <main+31>:    call   0x3005 <dyld_stub_puts>
0x00001fda <main+36>:    lea    -0xc(%ebp),%eax
0x00001fdd <main+39>:    incl   (%eax)
0x00001fdf <main+41>:    cmpl   $0x9,-0xc(%ebp)
0x00001fe3 <main+45>:    jle    0x1fcc <main+22>
```

Exécution de la boucle for

- String "hello world ! \n" contenue à l'adresse 0x2e(%ebx)
- Appel de la fonction printf()
- On incrémente i pour le prochain tour.

Remarque

Décomposons un petit peu tout ça...

```
0x00001fe5 <main+47>:      mov    $0x0,%eax
0x00001fea <main+52>:      add    $0x24,%esp
0x00001fed <main+55>:      pop    %ebx
0x00001fee <main+56>:      leave
0x00001fef <main+57>:      ret
```

Fin et sorite du programme

On replace le stack pointer à sa position initiale.

On réinitialise eax à 0.

Remarque