



IDA



Reverse Engineering

ΜΙΑ ΠΡΑΚΤΙΚΗ ΕΙΣΑΓΩΓΗ

By:

Μιχάλης Ευαγγέλου

Έρικα Σταμπούρλου

Και όλα τα παιδιά που βοήθησαν με τις ιδέες και το beta testing <3

Reverse Engineering 101

Συνοπτικά:

Πάρε ένα κομμάτι λογισμικού και πες μου τι κάνει. Συνήθως χρειάζεται να αντιστρέψεις τη λογική του, πραγματοποιώντας την αντίστροφη λειτουργία.

Το reverse engineering είναι απαραίτητο skill ενός **malware analyst**.

Τι περιλαμβάνει:

Στατική ανάλυση → Κοίτα τον κώδικα του προγράμματος και αντίστρεψε το χωρίς να το τρέξεις

Δυναμική ανάλυση → Τρέξε το πρόγραμμα για να καταλάβεις τι κάνει

Υβριδική ανάλυση → Συνδυασμός των δύο παραπάνω μεθόδων

Τι είναι ένα εκτελέσιμο αρχείο;

Πρόκειται συνήθως για ένα compiled αρχείο κώδικα, το οποίο προκαλεί την εκτέλεση εντολών της εκάστοτε αρχιτεκτονικής.

Δηλαδή ένα πρόγραμμα ρε αδερφέ!!

Ανάλογα με το λειτουργικό έχουμε και διαφορετικό format στο εκτελέσιμο.

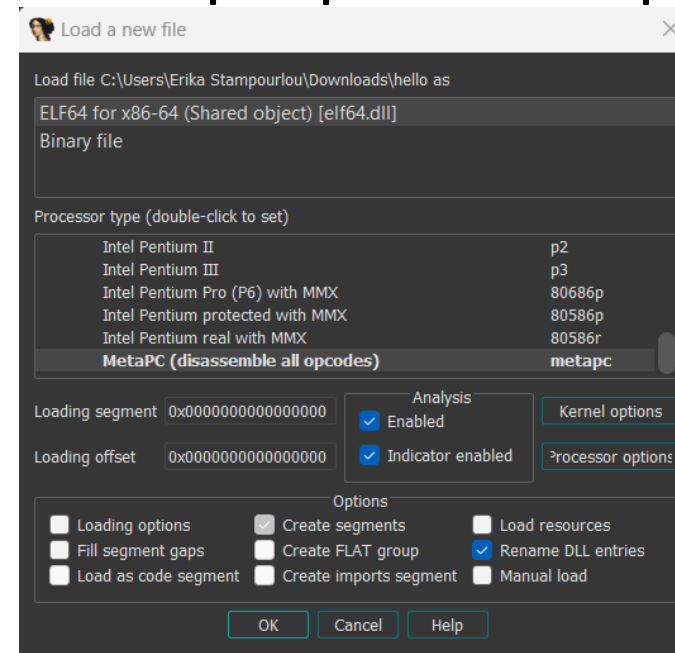
Για παράδειγμα, στα Windows έχουμε τα **exe** files, στα linux τα executables ονομάζονται **elfs** (λόγω του ELF header) κτλ.

Εκτελέσιμο θεωρείται επίσης ένα απλό script (σε **bash** ή **powershell**), μια δυναμική βιβλιοθήκη (**dll** ή **shared library**) και γενικά ό,τι έχει execution permissions.

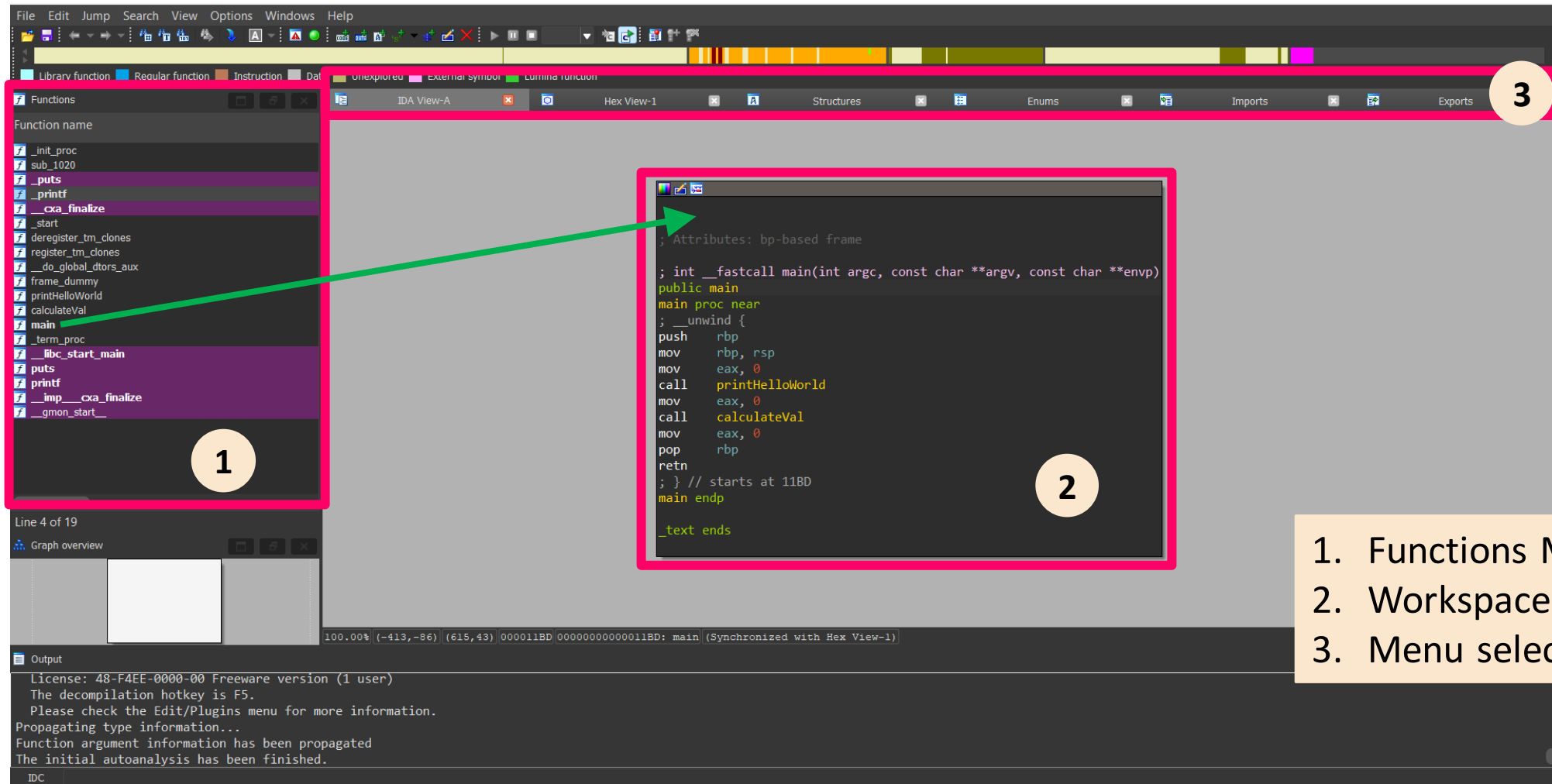
IDA – Ένα πανίσχυρο εργαλείο για Reversing

Η IDA είναι ένα πρόγραμμα που πραγματοποιεί στατικό και δυναμικό reversing.

Η φόρτωση ενός εκτελέσιμου γίνεται από το μενού File→Open και επιλογή του αρχείου. Η IDA μπορεί να αναγνωρίσει αυτόματα την αρχιτεκτονική του προγράμματος:



Λίγα λόγια για το UI



IDA Menus

1. IDA View: Το disassembly του προγράμματος
2. Pseudocode: Το decompilation του προγράμματος
3. Strings: Οι συμβολοσειρές που ανιχνεύονται από την IDA
4. Hex View: Το πρόγραμμα σε raw μορφή (bytes)
5. Structures: Δομές δεδομένων που φορτώνονται στην IDA
6. Λοιπά Menus

Η πλήρης λίστα των Menus (Views) της IDA βρίσκεται στην καρτέλα View
→ Open Subviews

```

1 int __fastcall main(int argc, const
2 {
3     printHelloWorld(argc, argv, envp);
4     calculateVal();
5     return 0;
6 }

```

2

```

; Attributes: bp-based frame

; int __fastcall main(int argc, const char **argv, const char **envp)
public main
main proc near
; __unwind {
push    rbp
mov     rbp, rsp
mov     eax, 0
call    printHelloWorld
mov     eax, 0
call    calculateVal
mov     eax, 0
pop     rbp
retn
; } // starts at 11BD
main_endp

```

1

```

0000000001170 F8 00 00 00 00 EB 29 8B 45 FC C1 E0 0F 89 C2 8B .....E.....
0000000001180 45 FC C1 F8 03 89 C1 8B 45 F8 01 C8 09 C2 8B 45 E.....E.....
0000000001190 F8 83 C0 09 0F AF C2 01 C0 89 45 FC 83 .....E..E...
00000000011A0 83 7D F8 63 7E D1 48 8D 05 6B 0E 00 00 .....}.c~....k...H
00000000011B0 B8 00 00 00 00 E8 86 FE FF FF 90 C9 C3 .....UH.....
00000000011C0 E5 B8 00 00 00 00 E8 7E FF FF FF B8 00 .....].....
00000000011D0 E8 8A FF FF FF B8 00 00 00 00 5D C3 48 83 EC 08 .....].....
00000000011E0 48 83 C4 08 C3 .....H.....
0000000001200 01 00 02 00 00 00 00 00 48 65 6C 6C 6F 20 57 6F .....Hello-W
0000000001210 72 6C 64 21 21 00 00 00 4F 6F 70 73 20 49 20 66 rld!!...Oops-I
0000000001220 6F 72 67 6F 74 20 74 6F 20 70 72 69 6E 74 20 74 orgot-to-print-
0000000001230 68 65 20 76 61 6C 75 65 20 6F 66 20 74 61 72 67 he-value-of-tar
0000000001240 65 74 2C 20 73 6F 72 72 79 2E 2E 2E 00 00 00 00 et,.sorry.....
0000000001250 01 1B 03 3B 3C 00 00 00 06 00 00 00 D0 EF FF FF ...;<.....
0000000001260 88 00 00 00 00 F0 FF FF B0 00 00 00 10 F0 FF FF .....X.....
0000000001270 58 00 00 00 F9 F0 FF FF C8 00 00 00 0F F1 FF FF .....m.....
0000000001280 E8 00 00 00 6D F1 FF FF 08 01 00 00 00 00 00 00 .....zR..x..
0000000001290 14 00 00 00 00 00 00 00 01 7A 52 00 01 78 10 01 .....zR..x..
00000000012A0 1B 0C 07 08 90 01 07 10 14 00 00 00 1C 00 00 00 .....".
00000000012B0 B0 EF FF FF 22 00 00 00 00 00 00 00 00 00 00 00 .....$.....
00000000012C0 14 00 00 00 00 00 00 00 01 7A 52 00 01 78 10 01 .....zR..x..
00000000012D0 1B 0C 07 08 90 01 00 00 24 00 00 00 1C 00 00 00 .....$.
00000000012E0 40 EF FF FF 30 00 00 00 00 0E 10 46 0E 18 4A 0F @...0...F...
00000000012F0 0B 77 08 80 00 3F 1A 3B 2A 33 24 22 00 00 00 00 .w...?.;*3$"...
0000000001300 14 00 00 00 44 00 00 00 48 EF FF FF 08 00 00 00 ...D...H.....
0000000001310 00 00 00 00 00 00 00 00 1C 00 00 00 5C 00 00 00 .....\.
0000000001320 29 F0 FF FF 16 00 00 00 00 41 0E 10 86 02 43 0D .....A....

```

4

```

00000000 ; Ins/Del : create/delete structure
00000000 ; D/A/* : create structure member
00000000 ; N : rename structure or str
00000000 ; U : delete structure member
00000000 ; -----
00000000 Elf64_Sym      struc ; (sizeof=0x1
00000000
00000000 st_name          dd ?
00000004 st_info          db ?
00000005 st_other          db ?
00000006 st_shndx          dw ?
00000008 st_value          dq ?
00000010 st_size          dq ?
00000018 Elf64_Sym      ends
00000018

```

5

Address	Length	Type	String
00000000...	0000001C	C	/lib64/ld-linux-x86-64.so.2
00000000...	00000012	C	__libc_start_main
00000000...	0000000F	C	__cxa_finalize
00000000...	00000007	C	printf
00000000...	0000000A	C	libc.so.6
00000000...	0000000C	C	GLIBC_2.2.5
00000000...	0000000B	C	GLIBC_2.34
00000000...	0000001C	C	_ITM_deregisterTMCloneTable
00000000...	0000000F	C	__gmon_start__
00000000...	0000001A	C	_ITM_registerTMCloneTable
00000000...	0000000E	C	Hello World!!
00000000...	00000035	C	Oops I forgot to print the value of targ
00000000...	00000006	C	;*3\$\"

3

Πώς ξεκινάω;

Εμπειρικός κανόνας: Ο προγραμματιστής γράφει μια κύρια συνάρτηση (`main`). Οπότε ξεκίνα από αυτήν (Για decompilation επιλέγουμε View → Open Subviews → Generate pseudocode).

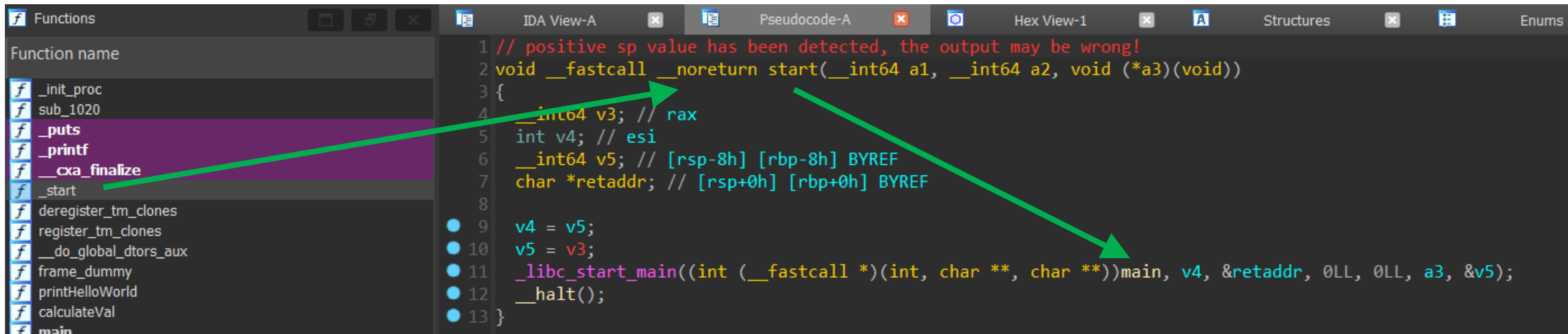
Αλλά: Η `main` δεν εκτελείται αμέσως από το πρόγραμμα.

Αντιθέτως φορτώνεται ο κώδικας του entry point (**start**), ο οποίος πραγματοποιεί τις κατάλληλες λειτουργίες ανάλογα με την πλατφόρμα (Windows, Linux) για την οποία προορίζεται το εκτελέσιμο (όπως για παράδειγμα το setup των arguments `argc`, `argv`, `envp`). Στη συνέχεια, καλείται κανονικά η `main`.

Οι κλήσεις συναρτήσεων πριν τη `main`, είναι κοινός τρόπος σε `malware developers`, ώστε να κάνουν δυσκολότερο το έργο των αναλυτών (βλέπε `TLS_Callbacks` στα Windows)

Υπάρχουν επίσης γλώσσες προγραμματισμού που δεν απαιτούν την ύπαρξη `main function`!

Παράδειγμα κώδικα start



```
1 // positive sp value has been detected, the output may be wrong!
2 void __fastcall __noreturn start(__int64 a1, __int64 a2, void (*a3)(void))
3 {
4     __int64 v3; // rax
5     int v4; // esi
6     __int64 v5; // [rsp-8h] [rbp-8h] BYREF
7     char *retaddr; // [rsp+0h] [rbp+0h] BYREF
8
9     v4 = v5;
10    v5 = v3;
11    _libc_start_main((int (__fastcall *)(int, char **, char **))main, v4, &retaddr, 0LL, 0LL, a3, &v5);
12    __halt();
13 }
```

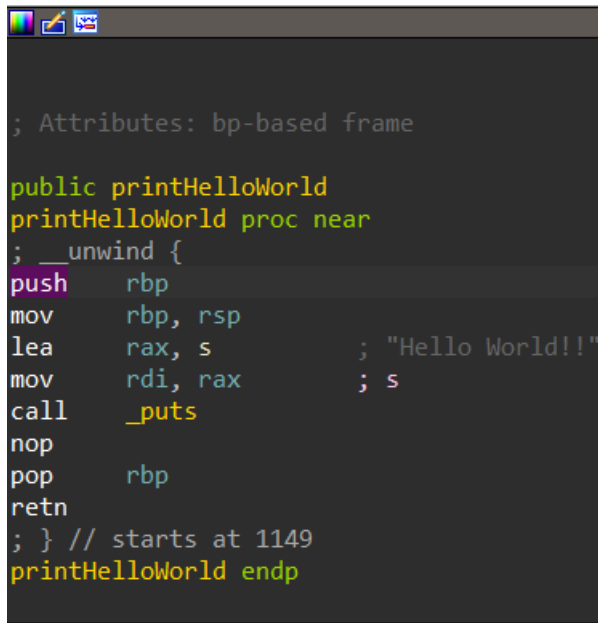
Η start καλεί τη main

Γενικότερα αν υπάρχει main, μπορούμε να ξεκινήσουμε από αυτήν την ανάλυση. Αλλά καλό είναι να εξετάζουμε και τη start, ειδικά σε περίπτωση που η IDA δεν καταφέρει να βρει τη main, ή/και το εκτελέσιμο είναι stripped (του έχουν αφαιρεθεί τα debugging symbols και άρα και τα ονόματα συναρτήσεων)

Static reversing (Disassembly / Decompilation)

- Μπορώ να ανακτήσω τον κώδικα από ένα εκτελέσιμο πρόγραμμα?
- **ΟΧΙ** (με κάποιες εξαιρέσεις)

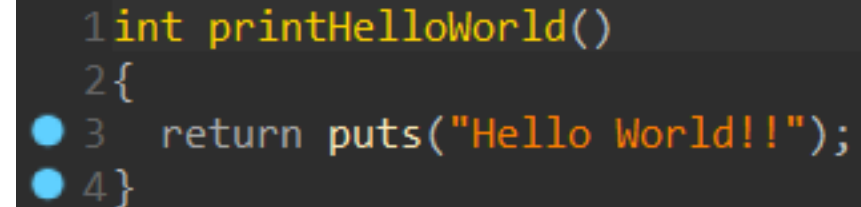
Αυτό που μπορείς να κάνεις όμως είναι να πάρεις την assembly του προγράμματος (disassembler) ή κάποιας μορφής ψευδοκώδικα τύπου C (decompilation)



```
; Attributes: bp-based frame

public printHelloWorld
printHelloWorld proc near
; __unwind {
push    rbp
mov     rbp, rsp
lea     rax, s          ; "Hello World!!"
mov     rdi, rax         ; s
call    _puts
nop
pop     rbp
retn
; } // starts at 1149
printHelloWorld endp
```

Disassembly



```
1 int printHelloWorld()
2 {
3     return puts("Hello World!!");
4 }
```

Decompilation

Και για τους άπιστους...

```
25
26 v2 = a2;
27 v4 = *(__int64 *)((char *)a1 + *(int *)(*a1 + 4) + 72);
28 if ( v4 )
29     (*(void (__fastcall *))(__int64))(*(_QWORD *)v4 + 8i64))(v4);
30 v5 = *a1;
31 v6 = *(int *)(*a1 + 4);
32 if ( *(_DWORD *)((char *)a1 + v6 + 16) )
33 {
34     v7 = 0;
35 }
36 else
37 {
38     v8 = *(__int64 **)((char *)a1 + v6 + 80);
39     if ( !v8 || v8 == a1 )
40     {
41         v7 = 1;
42     }
43     else
44     {
45         sub_140003AD0(v8);
46         v5 = *a1;
47         v7 = *(_DWORD *)((char *)a1 + *(int *)(*a1 + 4) + 16) == 0;
48     }
49 }
50 if ( v7 )
51 {
52     v11 = *(_QWORD **)((char *)a1 + *(int *)(v5 + 4) + 72);
53     if ( *(_QWORD *)v11[8] && (v12 = (int *)v11[11], *v12 > 0) )
54     {
```

Ο ψευδοκώδικας που παίρνουμε δεν είναι ίδιος με τον κώδικα που έγραψε ο προγραμματιστής

Dynamic reversing (debugging)

Καμία φορά, η στατική ανάλυση δεν επαρκεί για να κατανοήσουμε τη λειτουργία ενός προγράμματος. Σε αυτήν την περίπτωση χρησιμοποιούμε **debugger**.

Όταν χρησιμοποιούμε debugger, 'τρέχουμε' το πρόγραμμα => Για την ανάλυση **ιών**, η εκτέλεση πρέπει να γίνεται **σε κατάλληλα διαμορφωμένο isolated περιβάλλον!**

Ο debugger επιτρέπει το 'πάγωμα' της εκτέλεσης του προγράμματος στην εντολή που επιθυμούμε, ώστε να μπορούμε να εξετάσουμε την εικόνα της στοίβας, των καταχωρητών, της μνήμης κτλ (High level: μπορούμε να δούμε τιμές μεταβλητών).

Dynamic reversing (debugging)

The screenshot displays the IDA Pro interface during a dynamic debugging session. The main window shows the C pseudocode for a function named `calculateVal`. The function initializes `v2` to 15, enters a `for` loop from `i = 0` to `99`, and calculates a value based on `v2` and `i`. A `printf` statement is highlighted, indicating a breakpoint. The function returns `0x9F1D53E8`.

The right-hand side of the interface shows several panels:

- General registers:** Lists registers RAX, RBX, RCX, RDX, and RSI with their current values and pointers to memory locations.
- Modules:** Shows the loaded module `/home/kali/hello`.
- Threads:** Displays a single thread with ID 7250, Hex 1C52, State Ready, and Name `hello`.
- Stack view:** Shows the stack frame for `calculateVal` at address `00007FFFE28173A0`, including local variables and return values.

The bottom-left panel shows the Hex View-1, displaying the raw bytes of the function's code.

Breakpoint έπειτα από το for loop που αλλάζει την τιμή της v2. Η τελική τιμή είναι 0x9F1D53E8 (2669499368)

Σχηματικά

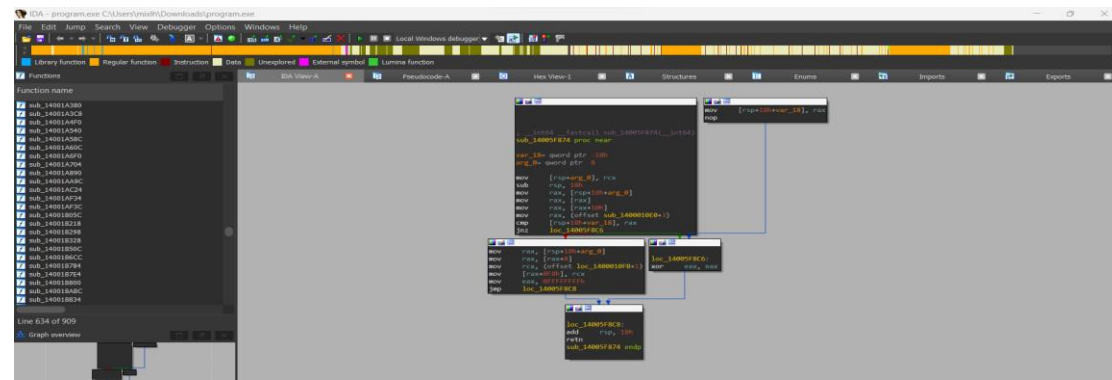
Debugger (dynamic analysis)



program.exe
1/12/2023 8:29 μμ
409 KB

Understand the
functionality of
the program

Disassembler/Decompiler (static analysis)



Come closer, I don't byte

Ξέρετε ότι 1 byte = 8 bits = 2 hexadecimal characters. Παίρνει τιμές 0-255.
e.g. $134_{10} = 0b10001000_2 = 0x86_{16}$ (τα 0b, 0x μπορούν να παραληφθούν)

Όσον αφορά το reversing, είναι **βασικό** να έχετε υπόψη σας τον παρακάτω πίνακα:

Data Type	Unsigned Version	Length in Bytes	Length in Bits
long int	uint64_t	8*	64*
int	uint32_t	4	32
short	uint16_t	2	16
char	uint8_t	1	8

Σημείωση: Οι προσημασμένοι αριθμοί δε διαφέρουν από τους μη προσημασμένους στη χωρητικότητα. Η διαφορά τους βρίσκεται στην ερμηνεία του προσήμου κατά τις αριθμητικές πράξεις.

Βασικά, τα πάντα είναι bytes

Παράδειγμα: Ο χρήστης δίνει input 'ntua' μέσω του πληκτρολογίου. Το input αυτό μεταφράζεται σε bytes, μέσω ενός ascii table:

Hex	Value	Hex	Value	Hex	Value	Hex	Value	Hex	Value	Hex	Value	Hex	Value	Hex	Value
00	NUL	10	DLE	20	SP	30	0	40	@	50	P	60	`	70	p
01	SOH	11	DC1	21	!	31	1	41	A	51	Q	61	a	71	q
02	STX	12	DC2	22	"	32	2	42	B	52	R	62	b	72	r
03	ETX	13	DC3	23	#	33	3	43	C	53	S	63	c	73	s
04	EOT	14	DC4	24	\$	34	4	44	D	54	T	64	d	74	t
05	ENQ	15	NAK	25	%	35	5	45	E	55	U	65	e	75	u
06	ACK	16	SYN	26	&	36	6	46	F	56	V	66	f	76	v
07	BEL	17	ETB	27	'	37	7	47	G	57	W	67	g	77	w
08	BS	18	CAN	28	(38	8	48	H	58	X	68	h	78	x
09	HT	19	EM	29)	39	9	49	I	59	Y	69	i	79	y
0A	LF	1A	SUB	2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
0B	VT	1B	ESC	2B	+	3B	;	4B	K	5B	[6B	k	7B	{
0C	FF	1C	FS	2C	,	3C	<	4C	L	5C	\	6C	l	7C	
0D	CR	1D	GS	2D	-	3D	=	4D	M	5D]	6D	m	7D	}
0E	SO	1E	RS	2E	.	3E	>	4E	N	5E	^	6E	n	7E	~
0F	SI	1F	US	2F	/	3F	?	4F	O	5F	_	6F	o	7F	DEL

Σύμφωνα με τον παραπάνω πίνακα, το input του χρήστη αντιστοιχεί στα bytes:

n	t	u	a
0x6e	0x74	0x75	0x61


```

int __fastcall main(int argc, const char **argv, const char **envp)
{
    int c[25]; // [rsp+0h] [rbp-70h]
    int k; // [rsp+64h] [rbp-Ch]
    int j; // [rsp+68h] [rbp-8h]
    int i; // [rsp+6Ch] [rbp-4h]

    c[0] = 65;
    c[1] = 109;
    c[2] = 32;
    c[3] = 73;
    c[4] = 32;
    c[5] = 97;
    c[6] = 32;
    c[7] = 110;
    c[8] = 117;
    c[9] = 109;
    c[10] = 98;
    c[11] = 101;
    c[12] = 114;
    c[13] = 32;
    c[14] = 111;
    c[15] = 114;
    c[16] = 32;
    c[17] = 97;
    c[18] = 32;
    c[19] = 99;
    c[20] = 104;
    c[21] = 97;
    c[22] = 114;
    c[23] = 63;

    puts("Values in hex:");
    for ( i = 0; i <= 23; ++i )
        printf("%x ", (unsigned int)c[i]);
    putchar(10);
    puts("Values in decimal:");
    for ( j = 0; j <= 23; ++j )
        printf("%d ", (unsigned int)c[j]);
    putchar(10);
    puts("Values in chars:");
    for ( k = 0; k <= 23; ++k )
        putchar(c[k]);
}

```

Στο πρόγραμμα αριστερά, παρατηρούμε ότι ο πίνακας `c` αρχικοποιείται με 24 ακεραίους.

Οι πράξεις εσωτερικά του προγράμματος γίνονται με την ακέραια αναπαράσταση των αριθμών (bytes).

Αυτό, όμως που έχει σημασία κατά την εκτύπωση, είναι η μορφή με την οποία θα επιλέξουμε να τους εμφανίσουμε στην οθόνη. Έτσι:

%x	%d	%c
hexadecimal	Decimal	character
Values in hex:		
41 6d 20 49 20 61 20 6e 75 6d 62 65 72 20 6f 72 20 61 20 63 68 61 72 3f		
Values in decimal:		
65 109 32 73 32 97 32 110 117 109 98 101 114 32 111 114 32 97 32 99 104 97 114 63		
Values in chars:		
Am I a number or a char?		

```

unsigned __int8 v4; // [rsp+Fh] [rbp-71h] BYREF
puts("Now tell me, what is your favourite character?");
__isoc99_scanf("%c", &v4);
printf(
    "Wow, you chose %c, which is %d in decimal and %x in hex\n",
    (unsigned int)v4,
    (unsigned int)v4,
    (unsigned int)v4);
v4 += 30;
if ( v4 <= 32 || v4 == 127 )
    printf(
        "Hmm I added 30 to it and now it is unprintable... Well here is its value in decimal: %x\n",
        (unsigned int)v4);
else
    printf("After adding 30, the character I got is %c, or %d in decimal\n", (unsigned int)v4, (unsigned int)v4);
return 0;
}

```

Οι printable χαρακτήρες, βρίσκονται στο εύρος [32,127] ή αλλιώς [0x20,0x7e] και περιλαμβάνουν τους αριθμούς 0-9, τα γράμματα a-z/A-Z και ειδικούς χαρακτήρες όπως !{}_#@ κτλ.

Έτσι, ανάλογα την είσοδο μπορούμε να πάρουμε:

```

Now tell me, what is your favourite character?
C
Wow, you chose C, which is 67 in decimal and 43 in hex
After adding 30, the character I got is a, or 97 in decimal

```

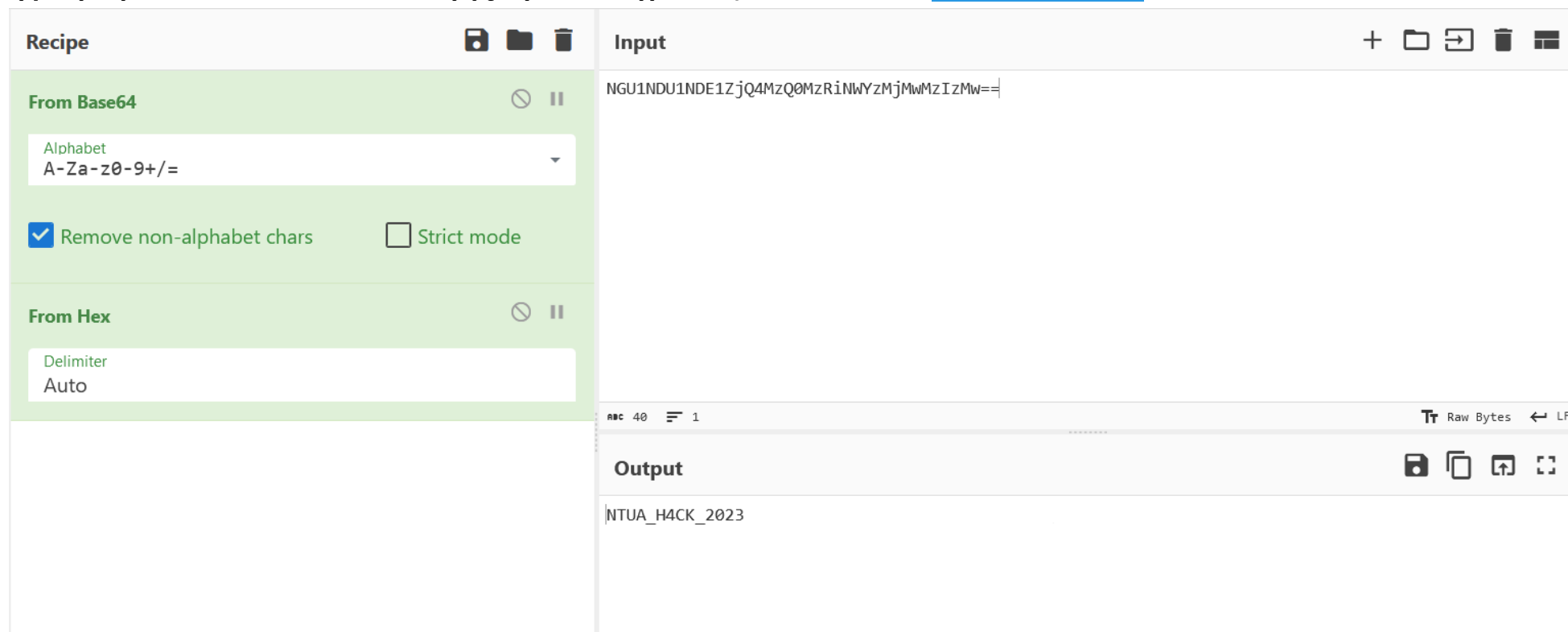
```

Now tell me, what is your favourite character?
x
Wow, you chose x, which is 120 in decimal and 78 in hex
Hmm I added 30 to it and now it is unprintable... Well here is its value in decimal: 150

```

Κυβερνομάγειρας to the rescue

Ένα απλό και εύχρηστο εργαλείο για πραγματοποίηση encoding/decoding (hex/ascii/decimal/octal/base64), encryption/decryption αλλά και άλλων λειτουργιών (πχ γρήγορη αντικατάσταση χαρακτήρων) είναι το [CyberChef](#).



Με τη συνταγή (αριστερά) αποκωδικοποιούμε το input (πάνω δεξιά) σε output (κάτω δεξιά) σε NTUA_HACK 2023



Ας μπορούμε σε
λίγο μεγαλύτερο
βάθος...

Δήλωση και ανάθεση στην IDA

```
1 int __fastcall main(int argc, const char **argv, const char **envp)
2 {
3     __int64 v4; // [rsp+0h] [rbp-20h] BYREF → 8 bytes
4     unsigned int *v5; // [rsp+8h] [rbp-18h]
5     _DWORD *v6; // [rsp+10h] [rbp-10h]
6     unsigned __int8 v7; // [rsp+19h] [rbp-7h] → 1 byte
7     unsigned __int16 v8; // [rsp+1Ah] [rbp-6h] → 2 bytes
8     unsigned int v9; // [rsp+1Ch] [rbp-4h] → 4 bytes
```

Variable Declaration

```
10 __isoc99_scanf(&unk_2004, &v4, envp); → Read 8 bytes into v4
11 v9 = v4;
12 v8 = v4;
13 v7 = v4;
14 printf("All 8 bytes %lx\n", v4);
15 printf("First 4 bytes %x\n", v9);
16 printf("First 2 bytes %x\n", v8);
17 printf("First byte %x\n", v7);
```

Cast v4 to 4 bytes (v9)
Then to 2 bytes (v8)
Then to 1 byte (v7)

Variable Initialization

Για παράδειγμα, με είσοδο 1384798606446149359
(0x1337cafedeadeadbeef) το πρόγραμμά μας βγάζει ως έξοδο:

```
All 8 bytes 1337cafedeadeadbeef
First 4 bytes deadbeef
First 2 bytes beef
First byte ef
```

Now some pointer stuff

Δείκτης (Pointer): Μεταβλητή που δείχνει σε τόσες θέσεις μνήμης όσες και ο τύπος των δεδομένων που προσπελάζει. **Κάθε θέση μνήμης έχει μέγεθος 1 byte.**

Π.χ.

(char *) p →

0x1001	0x45
--------	------

(int *) q →

0x1001	0x45
0x1002	0x57
0x1003	0x70
0x1004	0x6d

Ένας δείκτης χαρακτηρίζεται από την πρώτη διεύθυνση μνήμης στην οποία δείχνει. Για να πάρουμε το περιεχόμενο αυτής της διεύθυνσης, πρέπει να αποδεικτοδοτήσουμε (dereference) τον δείκτη (με ένα '*'). Έτσι, για παράδειγμα:

p = 0x1001

*p = 0x45

q = 0x1001

*q = 0x6d705745 (???)

Pointer Arithmetic

Η πρόσθεση του x σε έναν pointer, έχει ως αποτέλεσμα την μετακίνησή του κατά τόσες θέσεις όσες και ο τύπος δεδομένων που προσπελάζει.

Π.χ.

`(char *) p` →

0x1001	0x45
--------	------

`(int *) q` →

0x1001	0x45
0x1002	0x57
0x1003	0x70
0x1004	0x6d

`(char *) (p + 1)` →

0x1002	0x57
--------	------

`(int *) (q + 1)` →

0x1005	0x99
0x1006	0xa2
0x1007	0x21
0x1008	0x48

Pointer Arithmetic

Μπορούμε επίσης (συνήθως) να αλλάξουμε την τιμή των δεδομένων στα οποία δείχνει ένας pointer:

Π.χ.

`(char *) p` →

0x1001	0x45
--------	------

`*p = 0x39` →

0x1001	0x39
--------	------

`(int *) q` →

0x1001	0x45
0x1002	0x57
0x1003	0x70
0x1004	0x6d

`*q = 0x39` →

0x1001	0x39
0x1002	0x00
0x1003	0x00
0x1004	0x00

Προσοχή: Εδώ αλλάζουμε 4 τιμές κάθε φορά (*int pointer*)

Pointers στην IDA

Στην IDA πολλές φορές μπορεί να δούμε τις εξής λέξεις μπροστά από έναν pointer:

Keyword	Size in bytes
WORD	2
DWORD (double word)	4
QWORD (quad word)	8

Π.χ

`(_DWORD *) q` →

0x1001	0x45
0x1002	0x57
0x1003	0x70
0x1004	0x6d

`(_WORD *) k` →

0x1001	0x45
0x1002	0x57

Επιστρέφουμε στο πρόγραμμά μας

```
5  _DWORD *v6; // [rsp+10h] [rbp-10h]
```

DWORD: 2 λέξεις (2 Words). Κάθε WORD στη μνήμη ισοδυναμεί με 2 bytes

DWORD *: Δείκτης σε 4 συνεχόμενες θέσεις (bytes) μνήμης

```
18  v6 = malloc(8uLL);  
19  *v6 = 0x13371337;  
20  v6[1] = 0xDEADBEEF;
```

- **malloc**: Συνάρτηση της C που δεσμεύει δυναμικά χώρο στη μνήμη (σε bytes)

- **8uLL**: 8 bytes (uLL → unsigned long long)

Η εντολή **malloc(8uLL)** δεσμεύει 8 bytes στη μνήμη

```
[heap]:000055FEE1411AC0 db 37h ; 7  
[heap]:000055FEE1411AC1 db 13h  
[heap]:000055FEE1411AC2 db 37h ; 7  
[heap]:000055FEE1411AC3 db 13h  
[heap]:000055FEE1411AC4 db 0EFh  
[heap]:000055FEE1411AC5 db 0BEh  
[heap]:000055FEE1411AC6 db 0ADh  
[heap]:000055FEE1411AC7 db 0DEh
```

$*v6 \equiv v6[0]$

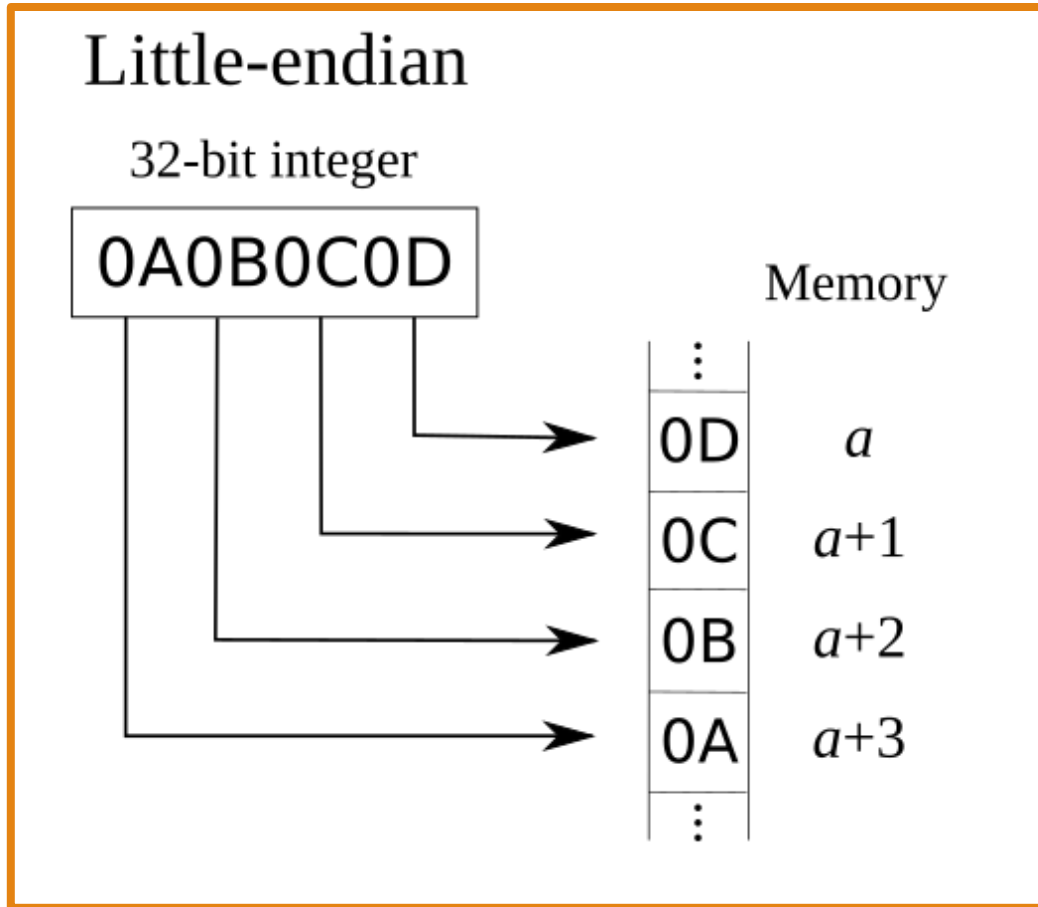


$*(v6+1) \equiv v6[1]$



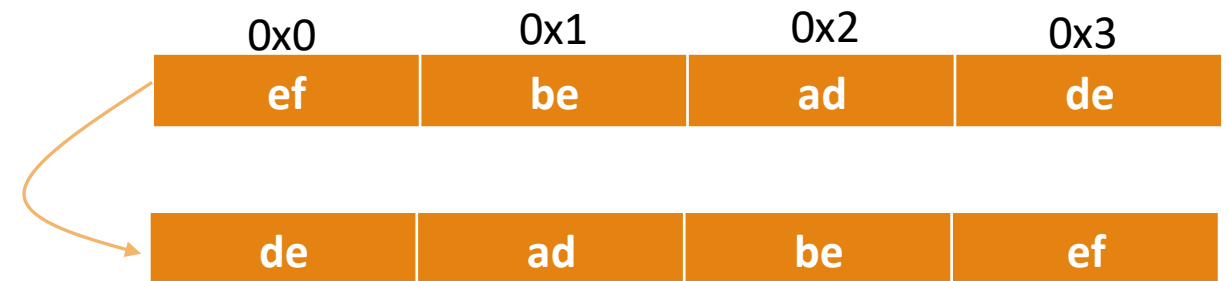
Wtf, έγγραψα deadbeef, γιατί βλέπω efbeadde?

Είναι θέμα endianness...



Σε μια little-endian αρχιτεκτονική, τα bytes στη μνήμη αποθηκεύονται σε αντίστροφη σειρά (lsb \rightarrow msb) από αυτήν που διαβάζονται (msb \rightarrow lsb).

Δηλαδή πρώτα το Least Significant Byte,..., και τέλος το Most Significant Byte



```
(kali@kali)-[~]  
$ file vartypes  
vartypes: ELF 64-bit LSB
```

Ας συνεχίσουμε...

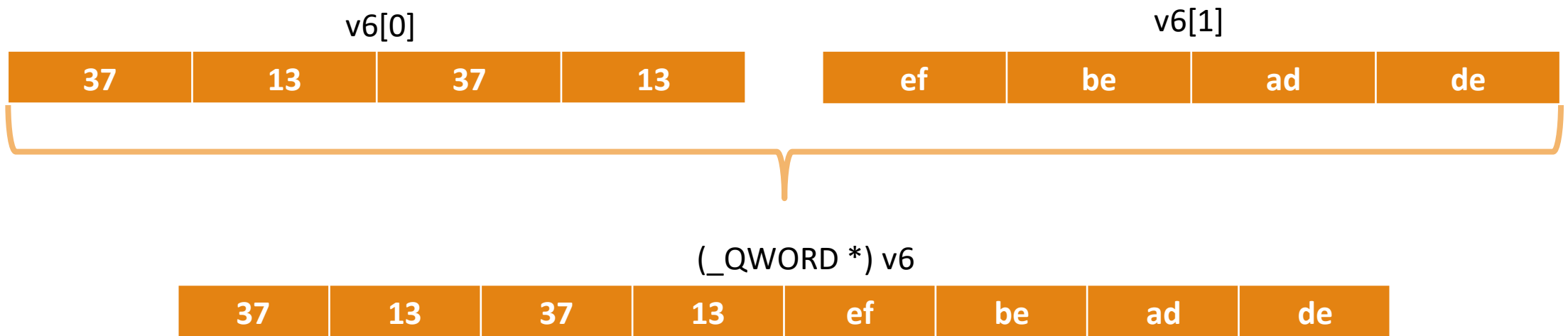
```
5  _DWORD *v6; // [rsp+10h] [rbp-10h]
3  __int64 v4; // [rsp+0h] [rbp-20h] BYREF
```

```
● 18  v6 = malloc(8uLL);
● 19  *v6 = 0x13371337;
● 20  v6[1] = 0xDEADBEEF;
● 21  v4 = *(_QWORD *)v6;
● 22  printf("All 8 bytes of a %lx\n", v4);
```

Ο v6 είναι δείκτης σε 4 θέσεις μνήμης και εμείς έχουμε δεσμεύσει 8 θέσεις στη μνήμη (κάθε θέση αντιστοιχεί σε 1 byte). Αρχικά αποθηκεύουμε στις πρώτες 4 θέσεις (*v6) την τιμή 0x13371337. Στη συνέχεια προσπελάζουμε τις επόμενες 4 θέσεις μνήμης (v6[1]) και αποθηκεύουμε την τιμή 0xdeadbeef. Συνολικά έχουμε χρησιμοποιήσει και τις 8 θέσεις!

```
5  _DWORD *v6; // [rsp+10h] [rbp-10h]
3  __int64 v4; // [rsp+0h] [rbp-20h] BYREF

18 v6 = malloc(8uLL);
19 *v6 = 0x13371337;
20 v6[1] = 0xDEADBEEF;
21 v4 = *(_QWORD *)v6;
22 printf("All 8 bytes of a %lx\n", v4);
```



Ο τύπος `(*_QWORD *)` κάνει cast τον pointer `v6`, ώστε από 4 θέσεις μνήμης να δείχνει σε 8 θέσεις
(`QWORD` = `QUADWORD` = 4 Words = 8 bytes)

```
5  _DWORD *v6; // [rsp+10h] [rbp-10h]
3  __int64 v4; // [rsp+0h] [rbp-20h] BYREF

18  v6 = malloc(8uLL);
19  *v6 = 0x13371337;
20  v6[1] = 0xDEADBEEF;
21  v4 = *(_QWORD *)v6;
22  printf("All 8 bytes of a %lx\n", v4);
```

→ v4 = 0xdeadbeef13371337

Στη συνέχεια η τιμή του `(_QWORD *) v6` ανατίθεται στον `v4`. Προσοχή, το `(_QWORD *) v6` είναι pointer, άρα πρέπει να τον αποδεικτοδοτήσουμε, οπότε θέλουμε την τιμή `*(_QWORD *) v6`.

Φανταστείτε έναν ενδιάμεσο pointer **intermediate**.

Ουσιαστικά συμβαίνουν τα εξής:

1. Cast `v6` from `DWORD` to `QWORD` `(_QWORD *)` (**temporarily**)
2. **Intermediate pointer** points to `(_QWORD *)v6`
3. Assign to `v4` the value `*intermediate` (\equiv `intermediate[0]`)

Question time!

```
4 unsigned int *v5; // [rsp+8h] [rbp-18h]
```

```
● 23 v5 = (unsigned int *)malloc(4uLL);  
● 24 *(_WORD *)v5 = 0x1337;  
● 25 *((_WORD *)v5 + 1) = 0xBABE;  
● 26 printf("All 4 bytes of b %x\n", *v5);  
● 27 return 0;
```

Μπορεί κάποιος να μου πει πώς δουλεύει ο παραπάνω κώδικας;

Χρήσιμο Υλικό

Disassemblers/Decompilers:

- [IDA – HexRays](#) – Η official ιστοσελίδα της HexRays για την IDA
- [Ghidra](#) – Open source Disassembler/Decompiler της HexRays
- [DnSpy](#) – Disassembler/Decompiler για .NET executables (C#)
- [Pycdc](#) – Disassembler/Decompiler για .pyc αρχεία

Debuggers:

- [DnSpy](#) , [IDA](#)
- [x64dbg](#), (για windows executables)
- [GEF](#) (για linux elfs)

Course:

- [Reverse Engineering 101,102](#) – by Malware Unicorn

Χρήσιμο Υλικό

Επίσης ρίξτε μια ματιά στο github μας με χρήσιμο υλικό και περιγραφές για όλες τις κατηγορίες:

<https://github.com/CS-IEEE-SB-NTUA/Ethical-Hacking/wiki/Hacking-Resources>

Ευχαριστώ πολύ για τον χρόνο σας!!!

