

!error	Display error	====	xax (64 bits)
!address	Display information about memory	=====	eax (32 bits)
-	List threads	====	ax (16 bits)
bl	List breakpoints	====	ah (8 bits)
bc	Cancel breakpoints	====	al (8 bits)
be	Enable breakpoints	{ IDA Pro shortcuts }	
bd	Disable breakpoints	Navigation:	
bp [Addr]	Set breakpoint at the address	Enter	: Jump to operand
bm SymPattern	Set breakpoint at the symbol	G	: Go to address
ba [r w el] Addr	Set breakpoint on Access	CTRL+P	: Jump to function
k	Display call stack	CTRL+E	: Jump to entry point
r	Dump all registers	Search:	
u	Disassemble	Alt+C	: Next code
dN	Display where N:	Alt+I	: Immediate value
a: ascii chars u: Unicode char		Alt+T	: Text
b: byte + ascii w: word		Alt+S	: Sequence of bytes
M: word + ascii d: dword		Graphing:	
c: dword + ascii q: qword		F12	: Flow chart
b: bin + byte l: bin + dword		Subviews:	
eN Addr Value	Edit memory	Shift+F4	: Name
.writemem f A S	Dump memory	Shift+F12	: Strings
f: file name		Debugger:	
A: Address		F9	: Start
S: Size (lx)		F7	: Step into
Ctrl+Shift+F7		Ctrl+Shift+F7	: Run until return
dec hex char dec hex char dec hex char dec hex char			
0 0x00 NUL	32 0x20 SPACE	64 0x40 8	96 0x60 -
1 0x01 SOH	33 0x21 !	65 0x41 A	97 0x61 a
2 0x02 STX	34 0x22 *	66 0x42 B	98 0x62 b
3 0x03 ETX	35 0x23 #	67 0x43 C	99 0x63 c
4 0x04 EOT	36 0x24 \$	68 0x44 D	100 0x64 d
5 0x05 ENQ	37 0x25 %	69 0x45 E	101 0x65 e
6 0x06 ACK	38 0x26 &	70 0x46 F	102 0x66 f
7 0x07 BEL	39 0x27 ^	71 0x47 G	103 0x67 g
8 0x08 BS	40 0x28 _	72 0x48 H	104 0x68 h
9 0x09 TAB	41 0x29 ~	73 0x49 I	105 0x69 i
10 0x0A LF	42 0x2A *	74 0x4A J	106 0x6A j
11 0x0B VT	43 0x2B +	75 0x5B K	107 0x70 k
12 0x0C FF	44 0x2C ,	76 0x5C L	108 0x7C l
13 0x0D CR	45 0x2D -	77 0x5D M	109 0x7D m
14 0x0E SO	46 0x2E ,	78 0x5E N	110 0x7E n
15 0x0F SI	47 0x2F /	79 0x5F O	111 0x7F o
16 0x10 DLE	48 0x30 0	80 0x50 P	112 0x70 p
17 0x11 DC1	49 0x31 1	81 0x51 Q	113 0x71 q
18 0x12 DC2	50 0x32 2	82 0x52 R	114 0x72 r
19 0x13 DC3	51 0x33 3	83 0x53 S	115 0x73 s
20 0x14 DC4	52 0x34 4	84 0x54 T	116 0x74 t
21 0x15 NAK	53 0x35 5	85 0x55 U	117 0x75 u
22 0x16 SYN	54 0x36 6	86 0x56 V	118 0x76 v
23 0x17 EOB	55 0x37 7	87 0x57 W	119 0x77 w
24 0x18 CAN	56 0x38 8	88 0x58 X	120 0x78 x
25 0x19 EM	57 0x39 9	89 0x59 Y	121 0x79 y
26 0x1A SUB	58 0x3A :	90 0x5A Z	122 0x7A z
27 0x1B ESC	59 0x3B :	91 0x5B \	123 0x7B \
28 0x1C FS	60 0x3C <	92 0x5C \	124 0x7C \
29 0x1D GS	61 0x3D =	93 0x5D]	125 0x7D]
30 0x1E RS	62 0x3E >	94 0x5E ^	126 0x7E ^
31 0x1F US	63 0x3F ?	95 0x5F DEL	127 0x7F DEL

Máster en Análisis de Malware, Reversing y Bug Hunting

ENIIT
INNOVAT BUSINESS SCHOOL



Campus Internacional
CIBERSEGURIDAD



UCAM
UNIVERSIDAD
CATÓLICA DE MURCIA

MÓDULO 3 – Análisis de Código Fuente

<https://campusciberseguridad.com>

INDICE DE CONTENIDOS

1.	Introducción.....	7
1.1.	Lenguajes de programación	8
1.2.	Evolución de los lenguajes de programación	8
1.2.1.	Lenguaje máquina (primera generación)	9
1.2.2.	Lenguaje ensamblador (segunda generación)	10
1.2.3.	Lenguajes de alto nivel (tercera generación)	12
1.2.4.	Lenguajes de cuarta y quinta generación.....	13
1.3.	Clasificación de los lenguajes de programación.....	13
1.3.1.	Lenguajes compilados	14
1.3.2.	Lenguajes interpretados.....	16
1.3.3.	Compilados vs interpretados.....	17
1.3.4.	Máquinas virtuales o el modelo “híbrido” de ejecución.....	18
1.3.5.	Lenguajes intermedios	20
1.4.	Sistemas de tipos.....	20
1.4.1.	Tipos estáticos	21
1.4.2.	Tipos dinámicos	21
1.4.3.	Diferencias entre tipos estáticos y dinámicos	22
1.5.	Paradigmas de programación.....	23
1.5.1.	¿Qué es un paradigma, respecto a los lenguajes de programación?	23
1.6.	Identificar un lenguaje de programación a partir de su código fuente	24
1.6.1.	Identificando el código fuente de un shellcode	25
1.6.2.	Identificando un lenguaje de programación	26
1.7.	¿Por qué es útil esta información?.....	31
1.8.	¿Qué tengo que haber aprendido?	31
2.	Elementos básicos del análisis de código (I)	32
2.1.	De la estructura monolítica al diseño estructurado.....	32
2.2.	Funciones y procedimientos.....	33
2.2.1.	Signatura de una función.....	33
2.2.2.	Tipos de funciones.....	34
2.2.3.	Parámetros de las funciones	38
2.2.4.	Paso por valor, paso por referencia (punteros)	39

2.2.5.	Parámetros constantes.....	43
2.2.6.	Parámetros por defecto y número variable de parámetros	44
2.3.	Modularidad y ámbito (visibilidad)	46
2.3.1.	Importaciones y exportaciones	47
2.3.2.	Grafo relacional de dependencias.....	50
2.3.3.	Ámbito y visibilidad	51
2.4.	¿Por qué es útil esta información?.....	57
2.5.	¿Qué tengo que haber aprendido?	57
3.1.	3. Elementos básicos del análisis de código (II)	58
	Secuencia.....	58
3.2.	Selección.....	61
3.3.	Iteración y recursión.....	68
3.4.	¿Por qué es útil esta información?	70
3.5.	¿Qué tengo que haber aprendido?	71
4.1.	4. La memoria	72
	Los espacios del kernel y el usuario	72
4.2.	Segmento de texto o el programa en si mismo	73
4.3.	Segmento de memoria mapeada	74
4.4.	Segmento de datos y segmentos de datos no inicializados.....	75
4.5.	Segmento de montículo o heap	76
4.6.	Segmento de entorno o environment.....	78
4.7.	Segmento de la pila o stack EN LA ARQUITECTURA x86 (32 bits)	79
4.7.1.	PUSH y POP, operando con la pila.....	79
4.7.2.	¿Pero para qué sirve la pila?	80
4.7.3.	Desensamblado del programa de ejemplo	81
4.7.4.	Convenciones de llamada.....	82
4.7.5.	Marco de pila.....	83
4.7.6.	Introduciendo parámetros para la función a llamar	85
4.7.7.	Llamando a la función	85
4.7.8.	Stack Base Pointer, recordando el marco de pila.....	86
4.7.9.	Creando espacio para las variables locales	88
4.7.10.	El retorno	89
4.7.11.	La pila en la arquitectura AMD64 (64 bits).....	90

MODULO 3 – ANÁLISIS DE CÓDIGO FUENTE

4.7.12.	¿Es posible evitar el registro EBP para crear marcos de pila?.....	92
4.8.	La memoria, foto final	94
4.9.	Código independiente de la posición y medidas antiexplotación.....	94
4.10.	¿Por qué es útil esta información?.....	95
4.11.	¿Qué tengo que haber aprendido?	95
		96
5.1.	5. Punteros	96
	¿Qué son?	96
5.2.	¿Para qué sirven?	96
5.3.	¿Cómo funcionan?.....	98
5.3.1.	Punteros a estructuras	101
5.3.2.	Punteros a punteros	102
5.3.3.	Punteros a funciones	103
5.3.4.	Punteros a arrays.....	105
5.4.	Aritmética de punteros	108
5.5.	Complejos, potentes pero peligrosos.....	110
5.6.	¿Por qué es útil esta información?.....	110
5.7.	¿Qué tengo que haber aprendido?	110
		111
6.1.	6. Herramientas y entorno de trabajo	111
	Editores de texto	111
6.1.1.	Editor de texto basado en interfaz de usuario	111
6.1.2.	Editor de texto basado en terminal de comandos	112
6.1.3.	Editor de texto basado en navegador web	113
6.1.4.	Características deseables en un editor de texto	113
6.2.	Indexadores simbólicos	117
6.2.1.	Ctags	121
6.2.2.	GNU Global	121
6.2.3.	cscope	123
6.3.	Navegadores de código	126
6.3.1.	CodeQuery.....	126
6.3.2.	OpenGrok	127
6.3.3.	Sourcetrail	128
6.4.	Herramientas de búsqueda	129
6.5.	Control de versiones de código	131

6.6.	¿Por qué es útil esta información?	133
6.7.	¿Qué tengo que haber aprendido?	133
7.1.	7. Desensamblado, decompilación y código ofuscado	134
	Desensamblado	135
7.1.1.	Herramientas visuales de desensamblado.....	136
7.1.2.	Herramientas de consola para desensamblar.....	139
7.1.3.	Comparativa sintaxis AT&T vs Intel	140
7.2.	Decompilado.....	141
7.2.1.	Un ejemplo práctico de decompilación.....	142
7.2.2.	Decompilado de un ejecutable binario en Ghidra	146
7.3.	Código ofuscado	146
7.3.1.	Un ejemplo práctico de desofuscación	148
7.4.	¿Por qué es útil esta información?	155
7.5.	¿Qué tengo que haber aprendido?	156
8.1.	8. Análisis de código fuente en lenguajes de bajo nivel	156
	Ensamblador y código objeto desensamblado	156
8.2.	Detección de funciones	157
8.2.1.	El prólogo de una función.....	157
8.2.2.	El epílogo de una función	158
8.3.	Detección de estructuras de control de flujo.....	158
8.3.1.	Estructuras iterativas.....	158
8.3.2.	Estructuras de decisión	162
8.4.	Detección de estructuras de datos.....	163
8.4.1.	Estructuras.....	163
8.4.2.	Arrays.....	164
8.5.	¿Por qué es útil esta información?	166
8.6.	¿Qué tengo que haber aprendido?	166
9.1.	9. Análisis de código fuente en lenguajes de alto nivel	167
	Descripción del proyecto Mirai	167
9.2.	Objetivo	168
9.3.	Herramientas empleadas	168
9.4.	Módulos empleados y referencias	168
9.5.	Analizando el contexto de las funciones implicadas.....	171

9.6.	Analizando las funciones implicadas	175
9.7.	Conclusión	182
9.8.	¿Por qué es útil esta información?	183
9.9.	¿Qué tengo que haber aprendido?	183
10.1.	10. Apéndice A: Guía básica de ensamblador	184
	Registros	184
10.1.1.	Registros de la CPU.....	184
10.1.2.	Registros de propósito general	185
10.1.3.	Registros de segmento	187
10.1.4.	EFLAGS o RFLAGS.....	187
10.1.5.	Puntero de instrucción	188
10.2.	Direccionamiento	188
10.2.1.	Direccionamiento inmediato.....	188
10.2.2.	Direccionamiento a registro	189
10.2.3.	Direccionamiento directo (o, Absoluto)	189
10.2.4.	Direccionamiento INDIRECTO	189
10.3.	OPCODES	189
10.4.	Instrucciones comunes.....	190
10.4.1.	Operadores aritméticos.....	190
10.4.2.	Operadores de comparación, lógicos y de desplazamiento	191
10.4.3.	Operadores de salto	192
10.4.4.	Operadores de pila y funciones.....	193
10.5.	Leer el desensamblado.....	194

1. INTRODUCCIÓN

Una de las principales tareas que nos encontramos en la ingeniería inversa, el análisis de malware o la búsqueda de vulnerabilidades es **entender que hace un programa**. Es imposible obtener un resultado en cualquiera de estas áreas sin leer código, analizarlo, comprenderlo y extraer el sentido original que su creador o creadores quisieron otorgarle. Por utilizar una analogía, no poseer estas habilidades sería el equivalente a intentar hacer un resumen de una novela escrita en un idioma que desconoces completamente.

El objetivo de esta asignatura es ofrecerte las herramientas, técnicas y el conocimiento básico para facilitar la tarea de entender el código con el cual se está trabajando. Sin ellas, aunque finalmente obtengamos una representación en código del programa u objeto de estudio, no sabremos dilucidar su finalidad y simplemente estaríamos divagando por las líneas de código de arriba abajo sin saber como descifrar qué tenemos delante de la pantalla.

Un ejemplo, bastante popular en lenguaje C:

```
void
send(short *to, short *from, int count)
{
    int n=(count+7)/8;
    switch(count%8){
        case 0: do{ *to = *from++;
        case 7:   *to = *from++;
        case 6:   *to = *from++;
        case 5:   *to = *from++;
        case 4:   *to = *from++;
        case 3:   *to = *from++;
        case 2:   *to = *from++;
        case 1:   *to = *from++;
                    }while(--n>0);
    }
}
```

Si tuvieses que enfrentarte a esta pieza de código ¿Por dónde empezarías?

Comprender el código, tanto en un **análisis dinámico** como **estático** de un programa, nos aporta una visión única e imprescindible. En la ingeniería inversa es un pilar fundamental e ineludible que nos permitirá llevar a buen puerto la tarea de deducir cual es el objetivo de un malware, como funciona un exploit o descubrir una vulnerabilidad con solo leer el código fuente.

IMPORTANTE: El curso no pretende que aprendas a programar en un lenguaje determinado. Se da por hecho que se poseen conocimientos básicos de programación y se es capaz de leer código ensamblador y lenguaje C a nivel básico.

Aunque se discuten y muestran las estructuras básicas de programación, se hace con la intención de que se familiarice con ellas y se sea capaz de detectar su presencia y entender su funcionamiento en cualquier lenguaje de programación.

1.1. LENGUAJES DE PROGRAMACIÓN

El objeto central del curso es el **código fuente** el cual estará escrito en un **lenguaje de programación**. Estos son el medio para que un humano pueda comunicarse con la máquina. De no ser así, por cada necesidad que surgiese deberíamos diseñar una máquina expresamente para realizar la tarea que nos propongamos.

De hecho, los primeros computadores poseían una limitada capacidad para ser programados, como, por ejemplo, el Colossus de Alan Turing¹; computadora que se uso durante la segunda guerra mundial para descifrar los códigos del ejercito alemán. Se podría decir que la “programación” en aquella época consistía en *recablear* el computador para conseguir un propósito distinto.

Esto, fue solucionado progresivamente por la programación. Es decir, la habilidad de usar una computadora para múltiples tareas, definidas por un algoritmo codificado en un lenguaje procesable por aquella. Poco a poco, se fue desplazando el engorroso trabajo de reconnectar cables a **codificar** las instrucciones en un medio que después era interpretado y ejecutado por el procesador.

Un procesador, por lo tanto, no hace nada extraordinario sin un programa que ejecutar. La única forma que tenemos para comunicarle nuestras intenciones es mediante un lenguaje de programación. En cierto modo, es como hablar un idioma particular. Cuanto más conoczamos el lenguaje de programación, mejor podremos expresarnos y seremos más concisos y específicos en aquellas tareas que necesitemos codificar.

1.2. EVOLUCIÓN DE LOS LENGUAJES DE PROGRAMACIÓN

A pesar de los avances en la tecnología, por norma general, un procesador no va a entender el código de un lenguaje de programación de **alto nivel** de forma directa. Para ello, necesita de un programa que traduzca ese lenguaje de alto nivel en otro más adecuado para la máquina; cercano a esta o de más **bajo nivel**. Ahí, entra el rol del compilador.

Utilizamos esta definición inicial para categorizar a los lenguajes de programación respecto a la cercanía o no de la máquina. Así, si hablamos de lenguajes de alto nivel o muy alto nivel, querremos decir que son lenguajes que se alejan de la máquina y se acercan de un modo u otro al lenguaje humano. Este tipo de lenguajes están pensados para que las personas podamos expresar con mayor facilidad las tareas o diseños a programar que solucionen problemas.

Por el contrario, un lenguaje de bajo nivel está pensado para la ejecución e incluso, como es el caso de los **lenguajes ensambladores**, para una arquitectura en particular. Estos lenguajes son los que se utilizaban en los primeros computadores programables y se consideran, aun así, de **segunda generación**, pues la primera generación consistía en la programación directa en **lenguaje máquina**. Como, por ejemplo, las primeras tarjetas o cintas perforadas.

Por lo tanto, podemos obtener nuestra primera clasificación de lenguajes de programación atendiendo a la distancia entre persona y computador:

¹ <https://es.wikipedia.org/wiki/Colossus>

1.2.1. LENGUAJE MÁQUINA (PRIMERA GENERACIÓN)

Es lo que popularmente habremos escuchado alguna vez: “los ceros y unos”. Una secuencia de dígitos binarios que trasladarán su significado final a si habrá o no señal eléctrica en un punto determinado de uno de los millones de puertas lógicas que componen un procesador.

Observemos que apariencia posee un programa realmente:

```
00003fae: 00000000 00000000 00000011 00000000 00000000 00000000 .....  
00003fb4: 00000000 00000000 00000000 00000000 00110000 01000000 .....@  
00003fba: 00000000 00000000 00000000 00000000 00000000 00000000 .....  
00003fc0: 00110000 00110000 00000000 00000000 00000000 00000000 00....  
00003fc6: 00000000 00000000 00001000 00000000 00000000 00000000 .....  
00003fcc: 00000000 00000000 00000000 00000000 00000000 00000000 .....  
00003fd2: 00000000 00000000 00000000 00000000 00000000 00000000 .....  
00003fd8: 00000001 00000000 00000000 00000000 00000000 00000000 .....  
00003fde: 00000000 00000000 00000000 00000000 00000000 00000000 .....  
00003fe4: 00000000 00000000 00000000 00000000 11111110 00000000 .....  
00003fea: 00000000 00000000 00000001 00000000 00000000 00000000 .....  
00003ff0: 00110000 00000000 00000000 00000000 00000000 00000000 0....  
00003ff6: 00000000 00000000 00000000 00000000 00000000 00000000 .....  
00003ffc: 00000000 00000000 00000000 00000000 00110000 00110000 ....00  
00004002: 00000000 00000000 00000000 00000000 00000000 00000000 .....  
00004008: 00100110 00000000 00000000 00000000 00000000 00000000 5....  
0000400e: 00000000 00000000 00000000 00000000 00000000 00000000 .....  
00004014: 00000000 00000000 00000000 00000000 00000001 00000000 .....
```

Si quieres experimentar, puedes obtener la salida de un ejecutable cualquiera en binario con la utilidad “`xxd -b <ejecutable>`” en cualquier sistema Linux

Probablemente, ya hayas leído en lenguaje máquina sin saberlo. Si has estudiado un volcado de memoria o desensamblado en **hexadecimal** ya lo has hecho. En general, como representación alternativa del formato binario propio del lenguaje máquina, se usa el formato hexadecimal, mucho más conveniente y extendido.

Por cierto, esta representación del código se denomina **código objeto o lenguaje máquina** (incluso, código máquina). Más adelante la definiremos mejor, aunque para ser precisos, el código objeto es aquel que siendo código máquina o lenguaje máquina es el producto o salida de un compilador.

Debes saber, no obstante, que el formato hexadecimal no es más que una ayuda para los ojos de un analista. Al final, la máquina leerá los datos e instrucciones en una larga línea de ceros y unos. Eso sí, debes acostumbrarte, sin bien no a leer en hexadecimal, al menos familiarizarte con su interpretación puesto que junto con el ensamblador, (y hay cientos de clases de ensamblador...), es la lengua franca de la **ingeniería inversa**.

En este formato, no hay distinción entre datos e instrucciones. Todo dependerá de como interpretaremos la secuencia de bytes. De hecho, en ingeniería inversa es común que cuando se esté analizando un programa o archivo, se interprete porciones de datos como instrucciones y viceversa

hasta que se da con el formato adecuado.

Una vista bien entrenada es capaz de detectar prólogos de funciones y determinadas estructuras del código ensamblador con solo ver la secuencia de valores hexadecimales; sin llegar incluso a desensamblar el código.

Naturalmente, por cada tipo de procesador el **lenguaje máquina** cambiará, por lo que un analista versado en la arquitectura Intel, podría no ver con la misma facilidad esas mismas estructuras en el código de una arquitectura del tipo ARM.

Por último, este tipo de lenguajes se conocen como de **primera generación**. Como es de figurar, antes del lenguaje máquina no existía algo que pudiera encajar con la definición, puesto que el recableado de un ordenador de la época, aunque se considera programación, no utilizaba lenguaje alguno para trasladar órdenes a la máquina.

1.2.2. LENGUAJE ENSAMBLADOR (SEGUNDA GENERACIÓN)

Aunque el progreso desde el recableado hasta el lenguaje máquina podría considerarse un salto tecnológico considerable, era evidente que, aun así, la programación era compleja, difícil y sobre todo con una tasa de errores altísima. Una sola variación de un uno o un cero en su lugar daba al traste con horas enteras de programación. Por no mencionar la casi nula capacidad de diagnóstico de errores y herramientas de depuración. Frente a esta necesidad, se desarrollan los lenguajes denominados ensambladores.

Si observamos un programa en lenguaje ensamblador, ya no veremos una cadena de ceros y unos, sin embargo, hay una alta proliferación de valores en hexadecimal junto con las instrucciones (de nuevo, por ser un formato y base numérica de conveniencia) que sigue recordándonos lo cerca que se está de la máquina. Observemos la misma salida de antes, pero vista en ensamblador:

```
root@kali:~# objdump a.out -d

a.out:      file format elf64-x86-64

Disassembly of section .init:
0000000000001000 <_init>:
1000:   48 83 ec 08      sub    $0x8,%rsp
1004:   48 8b 05 dd 2f 00 00  mov    %rax,%rax      # 3fe8 <__gmon_start__>
100b:   48 85 c0          test   %rax,%rax
100e:   74 02             je    1012 <_init+0x12>
1010:   ff d0             callq  *%rax
1012:   48 83 c4 08      add    $0x8,%rsp
1016:   c3                retq

Disassembly of section .plt:
0000000000001020 <.plt>:
1020:   ff 35 e2 2f 00 00  pushq  %rip      # 4008 <_GLOBAL_OFFSET_TABLE_+0x8>
1026:   ff 25 e4 2f 00 00  jmpq   *0x2fe4(%rip)      # 4010 <_GLOBAL_OFFSET_TABLE_+0x10>
102c:   0f 1f 40 00         nopl   %rax

0000000000001030 <printf@plt>:
1030:   ff 25 e2 2f 00 00  jmpq   *0x2fe2(%rip)      # 4018 <printf@GLIBC_2.2.5>
1036:   68 00 00 00 00      pushq  $0x0
103b:   e9 e0 ff ff ff    jmpq   1020 <.plt>

Disassembly of section .plt.got:
0000000000001040 <_cxa_finalize@plt>:
1040:   ff 25 b2 2f 00 00  jmpq   *0x2fb2(%rip)      # 3ff8 <_cxa_finalize@GLIBC_2.2.5>
1046:   66 90             xchg   %ax,%ax

Disassembly of section .text:
0000000000001050 <_start>:
1050:   31 ed             xor    %ebp,%ebp
1052:   49 89 d1          mov    %rdx,%r9
1055:   5e                pop    %rsi
1056:   48 89 e2          mov    %rsp,%rdx
1059:   48 83 e4 f0      and    $0xfffffffffffffff0,%rsp
105d:   50                push   %rax
105e:   54                push   %rsp
105f:   4c 8d 05 7a 01 00 00  lea    %r17(%rip),%r8      # 11e0 <_libc_csu_fini>
1066:   48 8d 0d 13 01 00 00  lea    %r11(%rip),%rcx      # 1180 <_libc_csu_init>
106d:   48 8d 3d c1 00 00 00  lea    %r1(%rip),%rdi      # 1135 <main>
1074:   ff 15 66 2f 00 00      callq  *0x2f66(%rip)      # 3fe0 <_libc_start_main@GLIBC_2.2.5>
107a:   f4                hit
107b:   0f 1f 44 00 00      nopl   %r9(%rax,%rax,1)
```

Si quieres experimentar, puedes obtener la salida de un ejecutable cualquiera en ensamblador con la utilidad “**objdump -d <ejecutable>**” en cualquier distribución Linux

Si observamos las instrucciones en ensamblador (columna más a la derecha), podremos ver que al menos poseen cierto sentido. Aunque lo veremos más adelante, en el capítulo dedicado al análisis de código ensamblador, este lenguaje posee una característica común en casi todas sus variantes: el empleo de **mnemotécnicos**².

El lenguaje es sencillo, sin complicaciones, básico. Si observamos, se trata de instrucciones muy concretas: “pon este valor aquí”, “mueve este valor allí”, “suma esto a ese valor”. No posee una sintaxis complicada; e incluso tratándose de un conjunto de instrucciones **CISC**³, poseen una clara estructura: mnemotécnico y uno o dos valores (ni siquiera pueden ser llamados parámetros)⁴ en la mayoría de los ensambladores.

² Cada instrucción poseía un nombre simbólico abreviado para no ocupar espacio pero que ayudase a ser recordado.

³ *Complex Instruction Set Computer*.

⁴ Siempre nos referiremos al ensamblador de Intel x86 o x86-64.

No obstante, aun estamos cerca del procesador. Se trata de instrucciones con un mapeado directo a la arquitectura empleada por el fabricante. Si intentáramos programar en el ensamblador de un x86 y tratáramos de ensamblar ese programa para un ARM sin conversión alguna no encontraríamos la forma de ejecutarlo.

Gracias al avance del ensamblador se consigue empezar a vislumbrar los esquemas básicos de la programación estructurada: **repetición** (iteración y recursividad), **selección** y **secuencia**. Se puede hablar ya de subrutinas y bloques e incluso de uno de los pilares fundamentales del avance en programación: la **reutilización de código**.

El siguiente reto por superar era la **portabilidad**: escribir código que pudiese ser ejecutado en las distintas arquitecturas y sistemas operativos que comenzaban a proliferar. Dado que el ensamblador está atado a la arquitectura, los desarrollos para las distintas plataformas requerían una inmensa inversión en la conversión de estos programas.

1.2.3. LENGUAJES DE ALTO NIVEL (TERCERA GENERACIÓN)

Son los lenguajes de programación tal y como los conocemos actualmente. Se encuentran en un punto intermedio entre el humano y la máquina. Estos lenguajes poseen una estructura similar al lenguaje humano, aunque, como en el caso muchos lenguajes, aun debemos tener conciencia del tamaño de las variables, gestión de memoria (sobre todo si el lenguaje no posee recolección de memoria) y otros recursos, etc.

Existe una gran riqueza de **paradigmas de programación**, **sistemas de tipos**, implementación y técnicas usadas. Veremos una completa clasificación de los lenguajes de programación de tercera generación, que nos ayudará a enfocar el estudio de su código y a elegir las herramientas adecuadas para su análisis.

La aparición de la tercera generación era una conclusión obvia por un motivo importante que ya hemos comentado: portabilidad. Era complejo trasladar el código ensamblador de una arquitectura a otra, además, la programación seguía siendo tediosa y con mucha tendencia a generar errores que eran difíciles de detectar y más aun de depurar y corregir.

Los lenguajes de programación de alto nivel dispararon el crecimiento del software y permitieron construir toda una ingeniería alrededor del desarrollo de software. Hasta entonces, la programación ligada a la máquina restringía el uso de computadoras al ámbito científico y militar, pero el hecho de poder generar la solución de un problema con un lenguaje que se abstraía de la máquina en la que se ejecutaba, ensanchó los límites de su uso hasta el estado en el que actualmente se encuentra.

```

Music Terminal - 
File Edit View Terminal Tabs Help
#include <stdio.h>
int main() {
    const unsigned int cantidad = 100;
    for (int i = 0; i <= cantidad ; ++i) {
        printf("%i\n", i);
    }
    return 0;
}

```

En la imagen de arriba podemos ver el programa utilizado en los apartados anteriores. Observando su concisión y expresividad, nos podemos hacer una idea del esfuerzo titánico que se hacía hace décadas para poder obtener un programa correcto.

Desde la imagen de los ceros y unos, pasando por el ensamblador y finalizando por el código fuente en C, donde se oculta gracias al uso de librerías la complejidad tras “printf”, existen grandes avances en la ingeniería de computadoras y lenguajes de programación.

1.2.4. LENGUAJES DE CUARTA Y QUINTA GENERACIÓN

Brevemente, por no pertenecer al ámbito de esta asignatura: los lenguajes de cuarta generación son aquellos que permiten describir la solución a un problema sin indicar, mediante la programación, como solucionar dicho problema. Por ejemplo, las herramientas CASE⁵.

Los lenguajes de quinta generación se asocian a procesos de desarrollo en inteligencia artificial; aunque aun no se ha demostrado un efecto práctico.

1.3. CLASIFICACIÓN DE LOS LENGUAJES DE PROGRAMACIÓN

Ahora nos centraremos en los lenguajes de programación de tercera generación o de alto nivel. Debido a la amplia gama existente, se introducirán varios conceptos claves compartidos por grupos de ellos que nos permitirá clasificarlos según estas características.

Aunque comparten estructuras e incluso sintaxis, existen diferencias esenciales que los hacen más apropiados para algunas tareas que otros. La clasificación no es ni pretende ser exhaustiva. De hecho, existen numerosos autores que difieren entre ellos respecto a qué elementos han de ser tenidos en cuenta para abordar la cuestión de agrupar unos lenguajes con otros.

⁵ Computed Aided Software Engineering

Como elementos prácticos, nos enfocaremos en si el lenguaje es **interpretado o compilado**. Si se ejecuta de forma **nativa** o en una **máquina virtual**. Si su sistema de tipos es **fuerte o débil** o si las variables son asignadas a un tipo concreto durante la ejecución del programa (**enlace dinámico**) o en la fase de compilación (**enlace estático**).

1.3.1. LENGUAJES COMPILADOS

Si escribimos código en un lenguaje de alto nivel ¿Cómo llega el procesador a entenderlo? Evidentemente, mediante un proceso de “traducción” que sustituya el código de alto nivel a un nivel entendible por la máquina **sin alterar la intención de su creador**. Es decir, cuando el compilador efectúa su trabajo, el código fuente original del programa en alto nivel dejará de ser usado en la ejecución y el **código objeto** resultante será el que finalmente procese la máquina.

El compilador es un programa que obtiene como entrada el código fuente creado por un programador y produce como salida un código, denominado **código objeto**, que puede ser ejecutado por una arquitectura en particular. Dicho proceso tiene por nombre: **compilación**.

Veamos un ejemplo (tomado del apartado anterior), supongamos el siguiente código fuente en el lenguaje de programación C:

```
#include <stdio.h>

int main() {
    const unsigned int cantidad = 100;
    for (int i = 0; i <= cantidad; ++i) {
        printf("%i\n", i);
    }
    return 0;
}
```

Utilizando un compilador, vamos a traducir este fragmento de código fuente en lenguaje ensamblador:

```
clang -S -mllvm -x86-asm-syntax=intel codigo_1.c
```

Observemos la salida:

```
.section __TEXT,__text,regular,pure_instructions
.build_version macos, 10, 15      sdk_version 10, 15
.intel_syntax noprefix
.globl _main          ## -- Begin function main
.p2align 4, 0x90
_main:             ## @main
.cfi_startproc
## %bb.0:
    push    rbp
    .cfi_def_cfa_offset 16
    .cfi_offset rbp, -16
    mov     rbp, rsp
    .cfi_def_cfa_register rbp
```

```

push    r14
push    rbx
.cfi_offset rbx, -32
.cfi_offset r14, -24
lea     r14, [rip + L_.str]
xor    ebx, ebx
.p2align 4, 0x90
LBB0_1:      ## =>This Inner Loop Header: Depth=1
    mov    rdi, r14
    mov    esi, ebx
    xor    eax, eax
    call   _printf
    inc    ebx
    cmp    ebx, 101
    jne    LBB0_1
## %bb.2:
    xor    eax, eax
    pop    rbx
    pop    r14
    pop    rbp
    ret
.cfi_endproc
        ## -- End function
.section __TEXT,__cstring,cstring_literals
L_.str:      ## @.str
    .asciz "%i\n"
.subsections_via_symbols

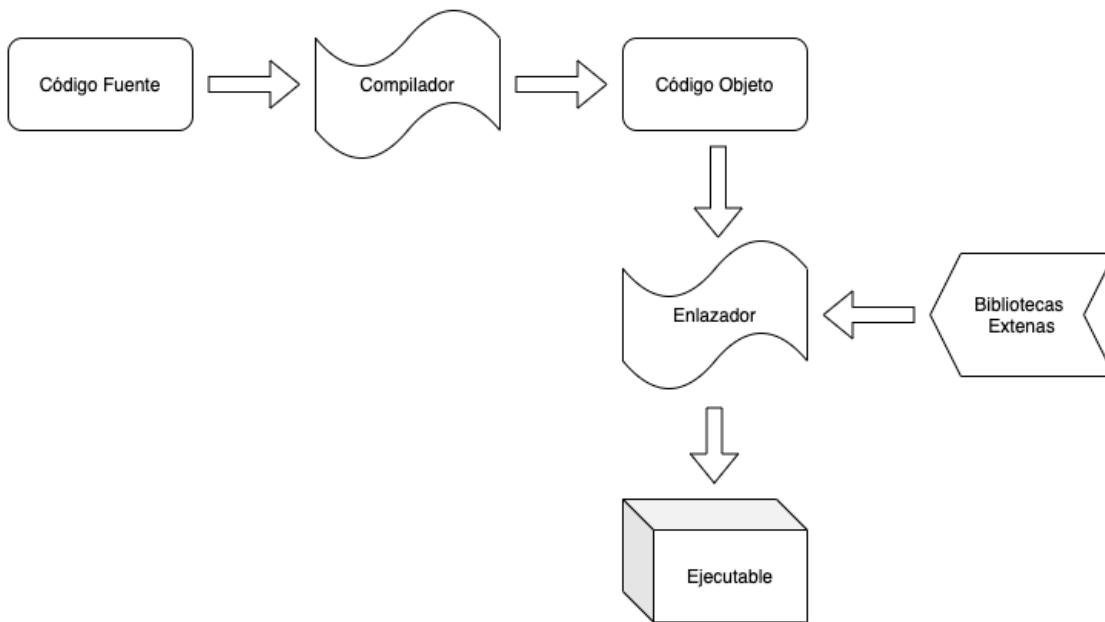
```

Como podemos ver, hemos descendido un nivel y sin experiencia previa en ensamblador, nos resultaría bastante complicado concluir que este programa solo tiene por objeto imprimir la lista de números del 0 al 100, si tan solo dispusiéramos del código fuente en ensamblador.

También debemos confesar que el código es el resultado de un compilador, que tiene muchas más tareas que hacer además de imprimir por pantalla el listado de números; por ejemplo, adecuar el código para su ejecución en el sistema operativo objetivo. Si hubiésemos hecho el mismo programa directamente en lenguaje ensamblador hubiese sido bastante más conciso y probablemente más legible.

Esta distinción es importante: “*una arquitectura en particular*”. Una de las virtudes de los compiladores es que pueden (si han sido diseñados para ello) compilar código para una o varias arquitecturas diferentes. Esta particularidad, es denominada **compilación cruzada**. Un ejemplo sería el uso de un compilador en un sistema bajo arquitectura x86-64 capaz de producir código para un procesador de la familia ARM.

Respecto a la etapa final: traducir el lenguaje ensamblador a lenguaje máquina, es tarea de un programa denominado **ensamblador**. El será, junto con el **enlazador**, el que producirá la imagen final del ejecutable o biblioteca lista para su uso en un sistema operativo o arquitectura en particular.



En la figura anterior, podemos ver como encajan todos los elementos a grandes rasgos. El código fuente es introducido en el compilador y de él saldrá un código objeto (a la vista es binario, es decir, código máquina), opcionalmente, si hacemos uso de librerías externas (a su vez, estas, código máquina), el enlazador resolverá los símbolos externos (por ejemplo, la llamada a una función de una librería) y creará un ejecutable con el formato impuesto por el sistema operativo.

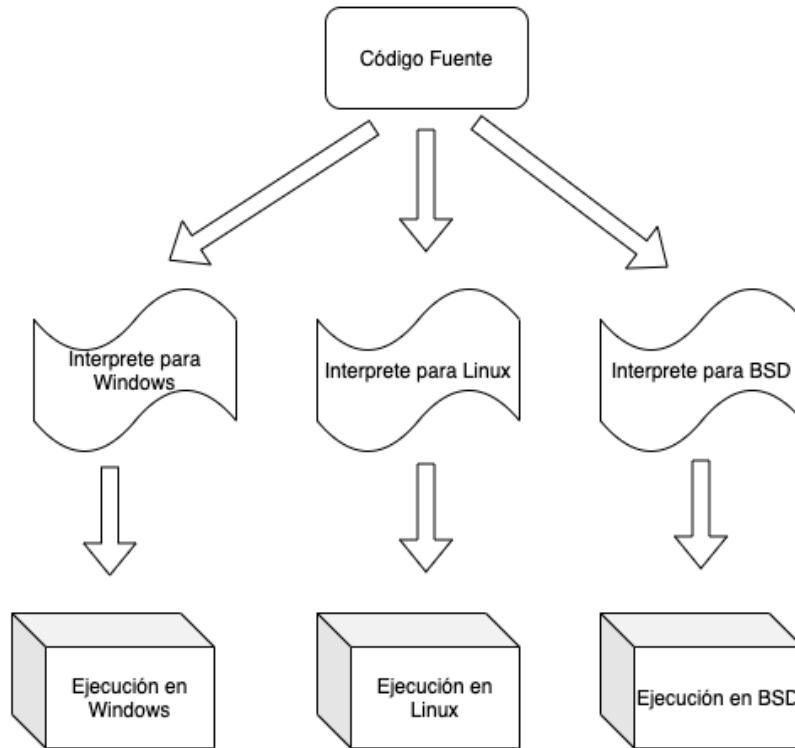
Ahora que ya sabemos como funciona la compilación, pasaremos a ver la interpretación. Antes de nada, debemos comentar que no existe impedimento para que un lenguaje compilado no posea interprete y viceversa. Es decir, el lenguaje existe por encima de la forma en el que el código fuente de este es procesado.

1.3.2. LENGUAJES INTERPRETADOS

En el esquema esencial de un interprete no hay etapas intermedias como las que hemos visto en un compilador (Posteriormente, veremos como con las máquinas virtuales esto se ha difuminado hacia un esquema híbrido). El código fuente es leído línea a línea por el interprete, el cual traduce a código máquina y ejecuta su significado al vuelo.

Por cada ejecución de un interprete, el código fuente es leído, traducido y ejecutado una y otra vez. Esto posee sus ventajas e inconvenientes. Por un lado, existe una gran penalización en la ejecución, puesto que el propio consumo de recursos cuando se produce la lectura e interpretación se añade a la posterior ejecución de esa línea. Por otro lado, esto permite una gran flexibilidad de cara a la portabilidad. Debido a que solo sería necesario portar el interprete a un sistema en particular, mientras que el código ya escrito se podría ejecutar sin cambio alguno.

Como podemos ver, el esquema de trabajo de un interprete es muy simple:



Por supuesto, esto requiere que los diferentes interpretes posean un rendimiento similar en cada plataforma e independientemente de la plataforma, deben ser programas muy optimizados para contrarrestar la penalización del proceso de interpretación.

Aunque los llamados lenguajes interpretados se suponen una traducción “directa”, veremos que esto no siempre es así. Habitualmente, se traducen a un código intermedio altamente optimizado y posteriormente se ejecutan en un interprete especializado en dicho código intermedio.

1.3.3. COMPILADOS VS INTERPRETADOS

Compilar código fuente es similar al proceso de traducción de un manuscrito de un idioma a otro. Imaginad que entregan para su traducción un documento en ruso y se requiere su traducción al español ¿Va a ser leído de forma inmediata el documento en español? No, quien lo traduzca lo hará tranquilamente, entregará el documento en español y ya se leerá cuando se necesite; además, **se podrá leer cuantas veces se requiera porque el documento ya estará traducido** y no tendremos que volver a traducirlo cada vez que necesitemos recurrir a él.

En el caso de los lenguajes interpretados, el proceso se asemeja más a la traducción simultánea. Se reúnen dos personas que desconocen sus respectivas lenguas y recurren a un servicio de traducción. La conversación fluye entre ellas en ese momento. Digamos que el proceso es directo y **cada vez que necesiten comunicar algo mutuamente, deberán recurrir a su traducción inmediata** a través de los

“intérpretes”.

Ahora bien, la pregunta que nos puede surgir es: ¿Qué aproximación es mejor? La respuesta es, por supuesto, que depende de la necesidad y requerimientos. Ningún modelo es mejor que otro. Básicamente, lo que no posee uno lo tiene el otro y viceversa.

En general, los lenguajes compilados ofrecen un gran rendimiento en ejecución y ahorro de recursos, por el contrario, el desarrollo suele ser más tedioso debido a los ciclos de programación-compilación-prueba-depuración. Pensemos que, para un pequeño cambio en el código, este ha de ser compilado de nuevo antes que pueda ser ejecutado.

Por el contrario, el desarrollo en lenguajes interpretados posee un característico ciclo de programación-ejecución-depuración más ágil y sencillo. Eso sí, se ha de tener en cuenta que cuando se ejecuten en producción, los programas ofrecerán un rendimiento algo más bajo que sus contrapartes compilados.

Una aproximación que suele emplearse es hacer un prototipo en un lenguaje interpretado y luego emplear el diseño en un programa compilado. Otra alternativa que ofrecen algunos lenguajes interpretados es derivar las partes más exigentes a un lenguaje compilado. Al final se trata, como vemos, de no cerrarnos a una solución cuando podemos emplear lo mejor de los dos mundos.

1.3.4. MÁQUINAS VIRTUALES O EL MODELO “HÍBRIDO” DE EJECUCIÓN

Cuando hablamos de lenguajes interpretados vimos que algunos de ellos no traducían inmediatamente el código fuente y procedían a su ejecución. Esto es debido a que el lenguaje de programación suele estar en un nivel bastante alto y cercano al lenguaje humano.

Cómo recordamos, cuanto más sencillo sea un lenguaje (por ejemplo, el ensamblador) y más cercano a la máquina esté, mayor será su rendimiento. Pero, por otro lado, esto suponía sacrificar la portabilidad y tener que reescribir el código para otras plataformas.

Para solucionar esto, se pensó en un modelo que tomase las ventajas de unos y otros e intentase paliar las debilidades de ambos. Dado que un lenguaje sencillo permitía una ejecución más rápida y optimizada, se ideó una compilación de un lenguaje fuente a uno denominado **lenguaje intermedio**.

Este lenguaje intermedio se asemeja (solo en apariencia) al lenguaje ensamblador en cuanto a sencillez y adecuación a la máquina. Ahora bien, ¿en qué máquina ejecutamos ese lenguaje intermedio?

Dado que esto no podía emplearse para máquinas reales, se inventó un concepto de **máquina virtual**. Un procesador no real, que interpreta el código intermedio y portada a cada sistema operativo deseado.

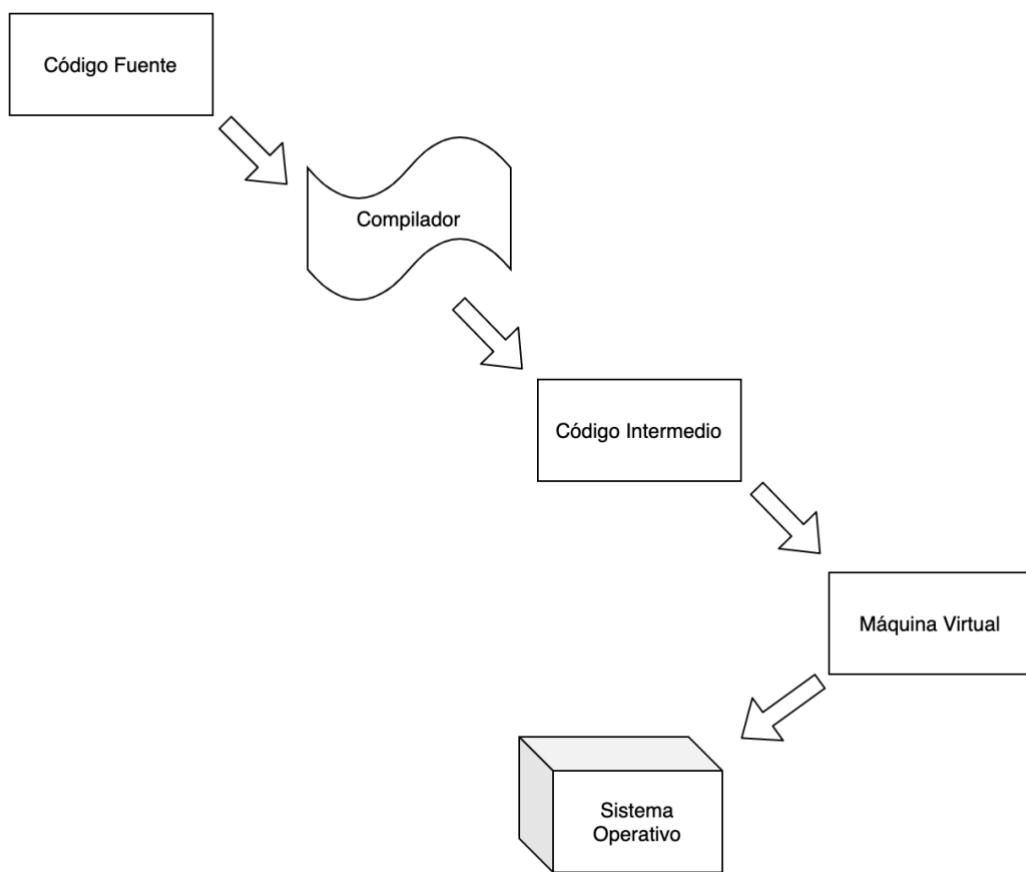
Un ejemplo muy claro de funcionamiento es la máquina virtual Java o JVM⁶. Este sistema posee un lenguaje característico y muy conocido: Java. Java no se ejecuta directamente sobre el sistema, a pesar de que dicen que es un lenguaje “compilado” este, es compilado a una representación

⁶ Java Virtual Machine

intermedia de su código a otro más sencillo denominado: **bytecode**.

Dicho bytecode es en realidad código intermedio que ahora sí, va a ser ejecutado por la máquina virtual JVM. Esto permite, por un lado, que todo el código Java producido pueda ser ejecutado en todos los sistemas operativos (e independientemente del procesador) donde exista una versión de la JVM y, por otro lado, dado que su código es compilado a un código intermedio y no nativo, agilizar los tiempos de compilación de forma que mejoren los tiempos de desarrollo.

El esquema sería el siguiente:



Como se observa, es un modelo intermedio con ambas aproximaciones. Si deseamos ejecutar nuestro código fuente, tan solo deberíamos asegurarnos de que existe una máquina virtual para dicho sistema.

Por supuesto, al ejecutar la máquina virtual código intermedio, para crear un nuevo lenguaje de programación, tan solo necesitamos un compilador que ofrezca una traducción a dicho código intermedio del lenguaje creado.

1.3.5. LENGUAJES INTERMEDIOS

Hemos de remarcar aquí el concepto de **lenguaje intermedio**. Cuando en nuestra tarea de ingeniería inversa tengamos que enfrentarnos a un programa escrito para una máquina virtual, con bastante probabilidad nos toparemos con un conjunto de archivos en este lenguaje; habitualmente dentro de un paquete o *bundle*, como por ejemplo un APK del sistema operativo Android.

La ventaja de los lenguajes intermedios es que es relativamente fácil **decompilar** el código hacia su forma original; el código fuente. Al contrario que sucede con los binarios nativos cuyo código decompilado (tras su desensamblado) no suele reproducir con tanta exactitud el código fuente original.

Otro ejemplo, aunque difiere sustancialmente, es Microsoft .Net. En dicho sistema, el código intermedio es finalmente compilado a un ejecutable nativo; aunque estos necesitan la presencia de las bibliotecas del entorno de ejecución .Net para funcionar.

1.4. SISTEMAS DE TIPOS

¿Qué es un tipo? Supongamos que una dirección de memoria cualquiera (0xFFFF0000) posee el siguiente aspecto.

0x616c6f68

Ese valor no nos dice nada. Pero si dijéramos que esos cuatro bytes son del tipo *entero sin signo*, el número en decimal sería: 1.634.496.360, pero si dijéramos que son los caracteres en hexadecimal de una cadena, esta sería “hola” (en arquitectura Intel, LittleEndian⁷)

Por lo tanto, un valor posee un significado gracias a su **tipo**. Sin él, solo es una cadena de ceros y unos sin sentido alguno. El tipo definirá el valor y dotará de significado su expresión en un programa.

Ahora bien, ¿Cómo declaramos un tipo? ¿Cómo le decimos a un procesador que una variable es un entero con signo o sin signo? Pues bien, esto se hace (a grandes rasgos) cuando se está compilando el programa desde código fuente del programa o cuando este se está ejecutando. Veremos.

⁷ <https://es.wikipedia.org/wiki/Endianness>

1.4.1. TIPOS ESTÁTICOS

Los tipos estáticos son así denominados porque se declaran en tiempo de compilación. Son estáticos porque un tipo asignado a un valor no cambia durante la ejecución del programa; con la salvedad de una operación denominada **casting** que permite cambiar su interpretación en forma de expresión o función)

Por ejemplo, en lenguaje C:

```
int valor = 5;
```

A partir de ahora, esa variable queda declarada e inicializada a un valor escalar, el 5. Su tipo es “int” o Integer (entero). Este tipo vendrá definido por el estándar del lenguaje e implementado por el compilador. Como nota adicional, este **tipo de dato** es también denominado **primitivo**, por el hecho de que ya viene determinado por el propio lenguaje y compilador.

Los tipos no primitivos son denominados **tipos derivados**. Son aquellos que son creados por el desarrollador a partir de los tipos primitivos; de ahí que se llamen derivados. Un ejemplo de tipo derivado sería una estructura, también llamado registro (record) o tipo agregado:

```
typedef struct coche {
    int ruedas;
    int marchas;
    int velocidad;
    int cilindrada;
    ...
} coche;
```

La repetición que vemos de *coche* no es un error. Estamos definiendo un tipo derivado llamado *coche* y a la vez, declarando una estructura de tipo *coche* llamada...*coche*. Como podemos observar, el nuevo tipo *coche* es un derivado que contiene cuatro enteros.

Las cosas se pueden complicar incluso más, debido a que podemos agregar tipos derivados a otros tipos derivado. Es más, el mismo tipo puede poseer un puntero o referencia hacia el mismo. Lo veremos más adelante.

1.4.2. TIPOS DINÁMICOS

Si no asignamos un tipo a un valor concreto en una posición de memoria durante la compilación, podríamos asignarlo durante la ejecución. Es lo que ocurre con muchos lenguajes interpretados.

Según el interprete de un lenguaje va leyendo las instrucciones de un programa, este va asignando un tipo en base al valor real que observa. Tomemos un ejemplo:

```
A = 100
B = "200"
```

$$C = A + B$$

Como podemos observar, la variable ‘A’ toma el valor de entero 100. Suponiendo que el lenguaje de programación posea un tipo “Integer” para los valores enteros, el interprete asignará este último a dicha variable.

Con la variable ‘B’, hará lo mismo. Su tipo será un hipotético “String”, ya que aunque dentro de las comillas veamos un número, prevalece su significado como cadena.

¿Qué ocurre con C? ¿Qué tipo le asignará el interprete? La respuesta es que depende completamente del lenguaje de programación y sus **reglas de conversión** frente al operador suma.

En el caso que observamos podrían suceder varias cosas y todas ellas serían perfectamente justificables:

- El interprete intenta sumar un entero y una cadena (tipos Integer y String respectivamente) y como el lenguaje no establece conversión da un error.
- El interprete observa la suma entre el entero y la cadena y decide que como lo que hay dentro de la cadena es un entero, lo convierte a este y se realiza la suma. C valdrá 300 y será un tipo Integer.
- El interprete convierte el valor de “A” a una cadena y decide que “+” en ese contexto es una concatenación de cadenas. C termina siendo interpretado como tipo “String” con valor: “100200”.

En el primer caso, se diría que el lenguaje posee **tipos estrictos o fuertes**. En los dos casos posteriores, se diría que el lenguaje de programación posee **tipos débiles**. Como vemos, la diferencia estriba en si se efectúa conversión frente a operaciones observables en dos tipos distintos. Esta última distinción también afecta a los tipos estáticos.

1.4.3. DIFERENCIAS ENTRE TIPOS ESTÁTICOS Y DINÁMICOS

Desde el punto de vista del desarrollador o programador, el sistema de tipos es bastante importante y definirá su ciclo de trabajo de una forma u otra. En el caso de los tipos estáticos, pasará más tiempo compilando el programa y observando que las variables y constantes sean del tipo adecuado, mientras que en los lenguajes con tipos dinámicos es durante la ejecución donde se verá si se poseen los tipos adecuados a una operación concreta.

Desde el punto de vista de la ingeniería inversa y más concretamente, desde el punto de vista del análisis de código, es más complejo seguir el hilo de un lenguaje con tipos dinámicos, ya que no se nos ofrece una riqueza en datos al no estar codificada la información que toman las variables.

Esto último es observable en los denominados IDE⁸, donde, el resultado de sintaxis y el autocompletado o herramientas de análisis son mucho más completas en lenguajes de tipos

⁸ Integrated Development Environment

estáticos que en los lenguajes con tipos dinámicos.

1.5. PARADIGMAS DE PROGRAMACIÓN

Existen diferentes aproximaciones a la hora de programar. Los **paradigmas** más empleados por los lenguajes de programación más usados⁹ son, el Orientado a Objetos e Imperativo. Dicho esto, es bastante común que los lenguajes de programación actuales soporten varios tipos de paradigmas, denominándose así **multiparadigma**.

1.5.1. ¿QUÉ ES UN PARADIGMA, RESPECTO A LOS LENGUAJES DE PROGRAMACIÓN?

Podría decirse que es el conjunto de formas y métodos para resolver un problema mediante un lenguaje de programación. Formas porque de un lenguaje a otro que comparten el mismo paradigma son reconocibles las estructuras empleadas en las que ambos resuelven un problema similar. Métodos porque existe un cuerpo de procedimientos comunes entre un mismo paradigma; una aproximación a la resolución de problemas que es compartida por todos los lenguajes dentro del mismo paradigma.

Tomando como ejemplo la **programación imperativa**, es bastante conocido el símil de esta respecto a la elaboración de una receta de cocina. Si observamos, una receta de cocina no nos dice nada acerca del problema que quiere resolver, sino que se limita a describir de forma muy clara las acciones que el usuario debe realizar.

Este es uno de los paradigmas clásicos y que nace directamente de los lenguajes ensambladores. Es el nivel más básico de entendimiento. “Da órdenes” de como se deben hacer las cosas y en vez de describir un problema, se limita a indicar paso a paso su solución.

Otra aproximación, algo diferente (aunque aun es declarativa), puede ser tomada de la **programación funcional**. Este paradigma nace de los inicios de la programación en el campo de la inteligencia artificial y la influencia de los matemáticos de la época en la que surgió.

El paradigma funcional intenta eliminar uno de los mayores problemas de la programación imperativa: el cambio de estado en un programa y sus efectos colaterales. Es decir, que, por ejemplo, cuando se llame a una función o método el resultado de esa llamada no sea siempre el mismo. Estos efectos colaterales, han ocasionado múltiples errores de programación (con grandes costes económicos e incluso, peor, de vidas humanas) y suelen ser la causa de muchas vulnerabilidades de seguridad.

Respecto al paradigma dominante en la actualidad, la **orientación a objetos** nace de la conjunción en una misma unidad, la clase, de los datos y las funciones necesarias para manipular a estos. Además, la orientación a objetos promueve tres formas destacables: **herencia, polimorfismo** y

⁹ <https://www.tiobe.com/tiobe-index/>

encapsulamiento.

La **herencia** permite la reutilización de código común basándose en una jerarquía de clases. Es decir, por ejemplo, la clase “Vehículo” puede contener un campo denominado “velocidad” porque todos los vehículos en algún momento deben desplazarse. Una clase que herede de “Vehículo”, como “Camión” no tendrá que volver a declarar un campo “velocidad” puesto que lo hereda de su clase superior.

El **polimorfismo** es una interesante característica que permite que diversos objetos diferentes puedan ser invocados usando un método común. Supongamos una clase “Garaje” que almacena “vehículos”. Dentro de un “garaje” podemos tener “camiones”, “coches” y “motos”. Ahora bien, dentro del “garaje” todos ellos pueden moverse si llamamos al método común “mover” que procede de su clase superior “Vehículo”. Así, todos se comportan como un “vehículo” pero atendiendo a sus particulares características.

Por último, tenemos el **encapsulamiento**. El encapsulamiento nos indica que el estado de una clase y sus métodos se circunscriben solo a la clase y no al exterior de esta. Es decir, si tenemos una clase “radio”, no tiene sentido que para cambiar de emisora se utilice un método externo a esta o tenga una propiedad exterior que indique a qué emisora está sintonizada. El encapsulamiento no debe ser confundido (aunque estrechamente relacionado) con otro principio de la orientación a objetos: el **ocultamiento de la información** o la **visibilidad**.

En definitiva, debemos estar preparados para leer código que no está basado en una sola escuela. Aunque hoy día el paradigma imperativo y el orientado a objetos sea mayoría, es fácil ver como algunos principios de otros paradigmas (en particular, el paradigma funcional) son adoptados e integrados como una técnica más. No demos por sentado que siempre vamos a encontrarnos estructuras, funciones, punteros y macros, etc.

1.6. IDENTIFICAR UN LENGUAJE DE PROGRAMACIÓN A PARTIR DE SU CÓDIGO FUENTE

El primer paso antes de analizar un programa es identificar el lenguaje o lenguajes de programación que lo componen. A menudo, esta tarea es sencilla; tanto como simplemente mirar la extensión de del archivo de texto que le sirve de soporte.



Sin embargo, a veces solo tendremos un fragmento de código. Imaginad un trozo de exploit rescatado de un pcap de una captura de tráfico de una máquina explotada. Sin conocer los detalles de la máquina ¿sabríamos averiguar el tipo de arquitectura que ha sido objeto de ataque? ¿x86? ¿x64? ¿O era ARM? ¿RISC, quizás?

1.6.1. IDENTIFICANDO EL CÓDIGO FUENTE DE UN SHELLCODE

Observad el código de la siguiente imagen (es un fragmento de un shellcode):

```
xor eax,eax
cdq
mov al,mmap2
xor ebx,ebx      ;addr
xor ecx,ecx
inc ecx          ;size
shl ecx,15        ;0x8000 potentially compute at runtime
;;      xor edx,edx; done with cdq
mov dl, PROT_READ|PROT_WRITE|PROT_EXEC ;prot
push byte MAP_PRIVATE | MAP_ANONYMOUS ;flags
pop esi
xor edi, edi      ;FD
dec edi           ;-1
xor ebp,ebp      ;offset
```

¿Sabrías identificar la arquitectura y plataforma (sistema operativo) con solo mirar el código? Veamos las pistas que nos deja el código.

Cojamos la instrucción XOR. Esta instrucción es muy común en ensamblador (OR exclusivo) pero en arquitecturas ARM, por ejemplo, es EOR¹⁰ por lo que se descarta que sea ARM. Nos decantamos por Intel, ya que el xor de MISP, por ejemplo, requiere de tres parámetros¹¹.

xor ebx,ebx ;addr

¿Pero es x86 o x64? Pues fíjemonos como hace referencia a los registros. EBX existe en las dos plataformas, pero si nos fijamos, el shellcode solo está utilizando los registros de 32 bits. No hay rastro de registros de x64 cuya contraparte, en el caso de EBX, sería RBX¹².

¹⁰ <https://developer.arm.com/documentation/dui0489/h/arm-and-thumb-instructions/and--orr--eor--bic--and--orn>

¹¹ <http://www.mrc.uidaho.edu/mrc/people/jff/digital/MIPSir.html>

¹² <https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/x64-architecture>

Entonces ya sabemos que con mucha probabilidad este shellcode es para una arquitectura Intel de 32 bits.

Elementos para tener en cuenta cuando estamos identificando un shellcode:

- Miramos las instrucciones que utiliza y las comparamos con las que soportan las distintas arquitecturas que conocemos, en cuanto encontramos una instrucción que no está definida para esa arquitectura la descartaremos.
- Nos fijaremos en como maneja los registros. Si hace referencia a registros o palabras que son de 64 bits sabremos que debemos descartar arquitecturas de menor ancho de palabra¹³.
- Si se trata de un shellcode en ensamblador, ver si existen comentarios, instrucciones macro, definiciones que hagan referencia a la plataforma y ensamblador exacto (NASM, gas, HLA, etc).

Observad las diferencias entre un shellcode en ensamblador de Intel x86, que hemos visto, y un ARM:

```

xor ebx,ebx          ;addr
xor ecx,ecx
inc ecx             ;size
shl ecx,15          ;0x8000 potentially compute at runtime
;;      xor edx,edx; done with cdq
mov dl, PROT_READ|PROT_WRITE|PROT_EXEC ;prot
push byte MAP_PRIVATE | MAP_ANONYMOUS ;flags
pop esi
xor edi, edi         ;FD
dec edi              ;-1
xor ebp,ebp          ;offset
                                .code 16
                                adr    r0, SHELL
                                eor    r2, r2, r2
                                strb   r2, [r0, #7]
                                push   {r0, r2}
                                mov    r1, sp
                                mov    r7, #11
                                svc    #1

```

1.6.2. IDENTIFICANDO UN LENGUAJE DE PROGRAMACIÓN

Como vimos, puede ser tan fácil como observar el archivo; es decir, su extensión. Pero esto no siempre va a ser así por dos razones: puede estar cambiada (la extensión) o puede que nos den el código en un simple archivo de texto sin ninguna identificación o directamente, nos dan solo fragmentos.

Observando el código de la imagen. ¿Dirías que es código C o C++? ¿Compatible con ambos?

```

#include <stdio.h>

struct point {
    int x;
    int y;
};

int main()
{
    point mypoint = {50, 50};
    printf("My point is at %u, %u\n", mypoint.x, mypoint.y);
    return 0;
}

```

¹³ [https://es.wikipedia.org/wiki/Palabra_\(inform%C3%A1tica\)](https://es.wikipedia.org/wiki/Palabra_(inform%C3%A1tica))

El código que muestra la imagen solo es compatible con C++. No es un programa correcto si intentamos compilarlo con C. Como vemos:

```
→ master_ACF gcc not_for_c.c
not_for_c.c:9:3: error: must use 'struct' tag to refer to type 'point'
    point mypoint = {50, 50};
    ^
 struct
1 error generated.
```

Hemos de tener un conocimiento más que superficial para atender a detalles tan pequeños. Como no vamos a tener ese conocimiento para todos los lenguajes de programación tendremos que disponer de ciertas técnicas para apoyarnos.

En primer lugar:

¿Es un lenguaje con tipos estáticos o dinámicos?

Esto es fácil de responder dado que la mayoría de los lenguajes con tipos estáticos declaran el tipo junto con la variable. Así, si vemos una variable que es usada sin que anteriormente hayamos encontrado su declaración, lo más probable es que estemos frente a un lenguaje dinámico. Con excepciones...

Observemos a Go o también conocido como Golang, un lenguaje de tipos estáticos:

```
7
8 type coded interface {
9     Code() int
10}
11
12 var checkCoded = func(err error) bool {
13     _, ok := err.(coded)
14     return ok
15}
16
```

En la línea 12 podemos ver que ‘checkCoded’ es una variable con el tipo: “función que toma un parámetro de tipo ‘err’ y devuelve un tipo booleano” ¿Correcto?

Sin embargo, en la línea 13 vemos que está declarando dos variables sin tipo. Eso puede despistarnos. Y es que algunos lenguajes de tipo estático permiten derivar el tipo y que sea el compilador el que lo identifique. ¿Cómo lo hace? Simplemente observando que tipo retorna la expresión que posee a la derecha. En este caso, el “err.(coded)”.

Existen variaciones en otros lenguajes (C#, C++) de este “atajo” para facilitar un poco la vida del programador y no tener que teclear el tipo de variables locales; esto es, dentro del cuerpo de una función.

Vamos a complicar un poco más esto. Veamos:

```
def add_visit(feed: Feed, domains: List[str], visit_result=VisitResult) -> None:
    length: int = len(domains)
    size: int = len("".join(domains))
    list_hash: str = hashlib.sha1("".join(domains).encode("utf-8")).hexdigest()
    visit_at: datetime.datetime = datetime.datetime.utcnow()

    new_domains: int = add_domains(feed, domains, visit_at)
```

¿Qué tenemos aquí? Una función con sus parámetros, tipos, variables locales, tipos... ¿Diríamos que es un lenguaje con tipos estáticos?

Es Python, de sobras conocido por ser un lenguaje con tipos dinámicos. Despista, porque el lenguaje no fuerza la corrección de tipos al ser ejecutado el script, sino que “permite” la escritura de “anotaciones” para que herramientas de terceros permitan sacar partido a dichas anotaciones.

Identificación de palabras clave

Todos los lenguajes de programación utilizan palabras reservadas¹⁴. Estas palabras no podemos redefinirlas en el código (o no deberíamos) y se presentan en abundancia en el código, junto con operadores lógicos, condicionales y matemáticos.

Uno de los casos clave es como define el lenguaje una función. Por la posición del valor de retorno, la palabra (o ausencia de esta) para definirla, etc.

Ejemplos:

Ruby:

```
def calculate_value(x,y)
  x + y
end
```

¹⁴ https://en.wikipedia.org/wiki/Reserved_word

Python:

```
def calculate_value(x, y):
    return x+y
```

C/C++:

```
int calculate_value(int x, int y) {
    return x+y;
}
```

Go:

```
func calculate_value(x int, y int) int {
    return x + y
}
```

Java (que siempre irá dentro de una clase):

```
class Playground {
    public static void main(String[ ] args) {
        System.out.println("Hello Java");
    }

    int calculate_value(int x, int y) {
        return x+y;
    }
}
```

C# (de igual modo, dentro de una clase):

```

1  using System;
2
3  public class Program
4  {
5      public static void Main()
6      {
7          Console.WriteLine("Hello World");
8      }
9
10     int calculate_value(int x, int y) {
11         return x + y;
12     }
13 }
```

Estamos viendo como en algunos casos necesitamos “más” información. Esto es, más código para poder separar un lenguaje de otro. Por ejemplo, se nota la influencia de C en C++, Java y C#. Aunque no está tan clara para Python y Ruby.

Curiosamente, estos dos últimos lenguajes poseen “parecidos razonables”, mientras que Java y C#, que descienden de la familia de lenguajes de C, cuesta distinguirlos y necesitaremos ampliar la lectura del código para entender quien es quien.

Identificación por el token empleado para los comentarios

Por absurdo que parezca, es posible identificar, o al menos obtener pistas fundamentales, observando que carácter o caracteres son empleados para usar comentarios en el código. Naturalmente, si es que este presenta comentarios.

Observemos la tabla (tomada de la Wikipedia inglesa¹⁵):

¹⁵ [https://en.wikipedia.org/wiki/Comparison_of_programming_languages_\(syntax\)#Comments](https://en.wikipedia.org/wiki/Comparison_of_programming_languages_(syntax)#Comments)

Symbol	Languages
C	Fortran I to Fortran 77 (C in column 1)
REM	BASIC, Batch files
::	Batch files, COMMAND.COM, cmd.exe
NB.	J; from the (historically) common abbreviation Nota bene , the Latin for "note well".
Ⓐ	APL; the mnemonic is the glyph (jot overstruck with shoe-down) resembles a desk lamp, and hence "illuminates" the foregoing.
#	Bourne shell and other UNIX shells, Cobra, Perl, Python, Ruby, Seed7, Windows PowerShell, PHP, R, Make, Maple, Elixir, Nim ^[10]
%	TeX, Prolog, MATLAB, ^[11] Erlang, S-Lang, Visual Prolog
//	ActionScript, C (C99), C++, C#, D, F#, Go, Java, JavaScript, Kotlin, Object Pascal (Delphi), Objective-C, PHP, Rust, Scala, SASS, Swift, Xojo
'	Monkey, Visual Basic, VBScript, Small Basic, Gamas, Xojo
!	Fortran, Basic Plus, Inform, Pick Basic
;	Assembly x86, AutoHotkey, AutoIt, Lisp, Common Lisp, Clojure, Rebol, Red, Scheme
--	Euphoria, Haskell, SQL, Ada, AppleScript, Eiffel, Lua, VHDL, SGML, PureScript
*	Assembler S/360 (* in column 1), COBOL I to COBOL 85, PAW, Fortran IV to Fortran 77 (* in column 1), Pick Basic
	Curl
"	Vimscript, ABAP
\	Forth
*>	COBOL 90

1.7. ¿POR QUÉ ES ÚTIL ESTA INFORMACIÓN?

Identificar en que lenguaje de programación está hecho un programa va a definir que tipo de herramientas y técnicas debemos emplear para estudiarlo. Es el primer paso en la tarea de la ingeniería inversa: conocer e informarnos sobre qué tenemos delante. De nada nos servirá poseer una gran destreza en una herramienta determinada si desconocemos como aplicarla a un programa cualquiera porque desconocemos su origen.

Del mismo modo, enfrentarnos a un script ofuscado o a un repositorio con veinte mil líneas de código (o muchísimas más) supone identificar el lenguaje, su implementación concreta y el estilo que define respecto a como trata las variables, sus tipos y los métodos que pueden ser empleados sobre ellas.

Es más, hoy día no es raro ver como una aplicación se divide en distintas partes: servidor, cliente web, aplicación móvil, etc. Cada una desarrollada con un lenguaje y herramientas diferentes. Saber conectar los distintos puntos de estas arquitecturas y no perderse es fundamental en el análisis de sistemas.

1.8. ¿QUÉ TENGO QUE HABER APRENDIDO?

- **Entender** que un analista de malware o investigador de vulnerabilidades debe saber **interpretar el código** de un programa ya sea porque posee o tiene acceso al código fuente o es posible obtener un desensamblado o decompilado de este.
- Al **reconocer un código fuente**, debes saber **identificar** a que lenguaje de programación pertenece.
- Entender los dos tipos principales de sistemas de tipos dominantes, el **dinámico** y el **estático**.
- Determinar si es código **compilado** o **interpretado**, si requiere de una **máquina virtual** para ejecutarse y si ésta posee una representación en **lenguaje intermedio**.

2. ELEMENTOS BÁSICOS DEL ANÁLISIS DE CÓDIGO (I)

La programación estructurada, como ya vimos, es el paradigma más básico de la programación y uno de los más extendidos. Incluso lenguajes orientados a objetos poseen como base este paradigma; por ejemplo, C++, que agrega objetos y programación genérica (vía plantillas) al lenguaje C, el cual es esencialmente un lenguaje estructurado.

En este módulo (dividido en dos partes) veremos como se organiza el código y en el siguiente las estructuras de control clásicas de la programación estructurada.

2.1. DE LA ESTRUCTURA MONOLÍTICA AL DISEÑO ESTRUCTURADO

En los inicios de la programación, todo el programa formaba una única pieza donde se entrelazaban los datos (variables) con los procedimientos (funciones). Las estructuras de control eran simples saltos y a lo sumo, se disponía de una sentencia tipo **goto**¹⁶ para realizar saltos a uno u otro lugar del código.

Tampoco existía la noción de parámetros ni valores de retorno, pues los datos eran objetos variables a la vista de todo el programa (variables globales). Es fácil emular este tipo de programación en cualquier lenguaje que implemente el paradigma de la programación estructurada.

Sea como fuere, este tipo de organización presentó muy pronto muchos inconvenientes¹⁷. Uno de ellos, el más evidente: la no **reutilización del código**. Esto supuso la aparición de funciones y procedimientos. Trozos de código que tomaban parámetros y podían retornar un valor.

Esto supuso una revolución y el siguiente paso fue la agrupación de funciones y estructuras de datos relacionadas en un solo conjunto que podía ser reutilizado cada vez que un programa necesitase su funcionalidad. Este concepto es lo que denominamos **programación modular** que facilita el **diseño estructurado**; una de las metodologías más básicas del desarrollo de software.

¹⁶ <https://en.cppreference.com/w/cpp/keyword/goto>

¹⁷ https://es.wikipedia.org/wiki/C%C3%B3digo_espagueti

2.2. FUNCIONES Y PROCEDIMIENTOS

¿Funciones o procedimientos¹⁸? Algunos autores llaman procedimientos a aquellas funciones que no devuelven ningún valor. Es decir, son llamadas, efectúan cambios en el estado del programa y no retornan ningún resultado que queramos inspeccionar o reutilizar.

Al margen de las denominaciones, nos interesa saber que en ocasiones las librerías y programas poseen funciones que no devuelven nada. Su retorno es meramente semántico (el ejemplo en lenguaje C sería una función que retorna **void**), pero habitualmente son funciones que al no devolver nada mutan el estado global del programa (a través de las variables globales) o bien mutan un parámetro que les es pasado por su **signatura**; por ello, deberemos estar atento a qué exactamente están mutando y como.

2.2.1. SIGNATURA DE UNA FUNCIÓN

Una función se identifica por su nombre, pero dado que en ciertos lenguajes se soporta una característica que permite poseer funciones con idéntico nombre, denominada **sobrecarga**¹⁹, la mejor forma de caracterizar una función es a través de su **signatura**.

La signatura de una función es el conjunto de los tipos de los parámetros²⁰ que recibe y el valor devuelto por esta en un orden determinado.

```
int main (int argc, char **argv) {
    return 0;
}
```

La función *main* típica del lenguaje C tendría la signatura: “recibe un valor de tipo entero sin signo, un puntero a un tipo puntero a *char* y devuelve un entero sin signo”.

Dependiendo del número de parámetros que posee la signatura de una función, podemos denominarlas por su **n-aridad**. Por ejemplo, si recibe un solo parámetro serían unitarias. Dos, binarias. Tres, ternarias...

Lo importante y que debemos tener en cuenta es que cuando hablamos de caracterizar una función comenzamos por su signatura: qué toma y qué devuelve.

¹⁸ También se aceptan más términos: rutina, subrutina, subprograma, método, etc.

¹⁹ No confundir con la sobrecarga de operadores característica de C++.

²⁰ También podemos llamarlos “argumentos”.

2.2.2. TIPOS DE FUNCIONES

Ya sabemos que las funciones que no devuelven nada las podemos denominar **procedimiento**. También sabemos que por su número de parámetros podemos llamarlas por su **n-aridad**: unitarias, binarias, etc.

Por mera combinación de tomas de parámetros y valores de retorno tenemos funciones qué:

- No toman parámetros y no devuelven nada.
- Toman parámetros y devuelve un valor.
- No toman parámetros y devuelve un valor.
- Toman parámetros y no devuelve nada.

Predicado

Cuando una función es usada para devolver un valor booleano (0 o 1, falso o verdadero) se le suele denominar **predicado**. Son funciones que suelen ser usadas para consultar un estado o mutarlo y devolver si ha tenido éxito o no. Suelen poseer como mucho un parámetro o ninguno y como condición sine qua non que solo devuelve un valor booleano.

Leaf Function (Función hoja)

Son funciones que, simplemente, no llaman a otra función dentro de su cuerpo. Es decir, actúan como una función ordinaria solo que el código que la implementa no realiza ninguna llamada. Son funciones cuyos análisis suelen ser sencillos ya que al no interrelacionarse con el resto de funciones (es posible que incluso ni siquiera toque memoria de pila) su análisis se circumscribe alrededor de su código exclusivamente.

El nombre “leaf”²¹ hace alusión a que si dibujamos un grafo de relaciones entre funciones, este tipo de funciones serían hojas de tal grafo.

Función del sistema, system call o syscall

Son aquellas funciones que posee el sistema operativo y expone a través del API para programadores. En principio, todos los programas que se ejecutan en el sistema tienen acceso a ellas. Por ejemplo, dos funciones similares del sistema operativo Windows y el kernel Linux:

<https://docs.microsoft.com/en-us/windows/win32/api/libloaderapi/nf-libloaderapi-loadlibrarya>

<https://linux.die.net/man/3/dlopen>

Con mucha frecuencia escucharéis y leeréis el término **syscall** para referirse a este tipo de funciones en el mundo del reversing

²¹ https://en.wikipedia.org/wiki/Leaf_subroutine

Función importada o externa

Son aquellas funciones que obtiene cualquier programa de una librería de terceros. Por ejemplo, si el programa que estamos analizando hace uso de funciones de compresión tiene dos opciones: o bien las implementa con funciones propias (siguiente definición) o las **importa** de una librería preexistente.

Por ejemplo, la librería zlib²², en caso de querer trabajar con compresión o curl²³ en caso de interactuar con servidores, etc. Ahora bien, existen distintos modos en los que un programa puede importar funciones de una librería. Mediante enlace dinámico, estático o dinámicamente durante la ejecución (carga dinámica).

Enlace dinámico significa que la librería usada por el programa se cargará cuando este se ejecute. Solo son necesarios los archivos de cabecera (típicamente los .h) para que el compilador pueda inspeccionar la firma y nombre de las funciones. Más tarde, cuando el programa se ejecute, el cargador dinámico del sistema operativo resolverá las dependencias. La desventaja es que el programa es sensiblemente más lento al necesitar resolver esas dependencias.

Enlace estático significa que se ha copiado el código binario de la librería en el programa del usuario. De esta forma, no necesita que esta se cargue durante la ejecución ya que el programa dispone de una copia para él. La desventaja es obvia: ejecutables de mayor tamaño.

Carga dinámica significa que el programa, durante su compilación, no hace referencia alguna a la función o funciones, ni de forma estática ni dinámica. Es más, si inspeccionamos el ejecutable producido no veremos referencia alguna. ¿Cómo es esto posible?

Mediante la carga dinámica de una librería. ¿Os acordáis de las funciones de sistema referenciadas arriba? *dllopen* y *LoadLibraryA*? Pues ambas sirven para utilizar el cargador dinámico del sistema operativo en tiempo de ejecución.

Es decir, podemos decirle al cargador dinámico: “busca una librería que se llama ‘x’ y dentro de esta una función llamada ‘y’, con una firma ‘z’”. El valor que devuelve esta función es un puntero a una función lista para ser usada; sin necesidad de haberla importado ni estática ni dinámicamente.

Esta técnica es frecuentemente usada, sobre todo, por los creadores de malware para ocultar el uso de funciones del sistema. Tanto para dificultar el análisis como la detección automatizada. Vamos a verlo claramente con un ejemplo:

²² <http://zlib.net/>

²³ <https://curl.haxx.se/libcurl/>

```
#include <dlfcn.h>
int main()
{
    void (*p_funcion_mrev) (const char *); // Function pointer
    void *handler;

    handler = dlopen("./Libmrev", RTLD_LAZY);
    p_funcion_mrev = dlsym(handler, "mrev");
    p_funcion_mrev("Hola desde main");
    dlclose(handler);

    return 0;
}

#include <stdio.h>
void mrev(const char*cad)
{
    printf("From mrev function: %s\n", cad);
}
```

A la derecha tenemos una simple, muy simple librería de una sola función “mrev” que toma un parámetro (un puntero a un tipo constante *char*). A la izquierda un programa que “usará” dicha función, pero no va a importar ni estática ni dinámicamente cuando lo compilemos:

Compilamos la librería:

```
gcc -shared -fPIC -o libmrev
```

Compilamos el programa (observad que -ldl significa que usamos el cargador dinámico del sistema operativo):

```
gcc main.c -ldl
```

Ejecutamos:

```
./a.out
```

```
root@kali: ~ /mrev
→ mrev ./a.out
from mrev function: Hola desde main
→ mrev [REDACTED]
```

Si inspeccionásemos el ejecutable no veríamos ninguna referencia a la función cargada dinámicamente. Ahora pensad, si además ofuscamos el nombre tanto de la librería como de la función ¿dificultaría eso el análisis?

Funciones de usuario

Son aquellas funciones que son implementadas por el programador. No están implementadas por una librería externa o por el sistema. Estas funciones son las que realmente nos interesa de cara a un análisis de código ¿Por qué?

Las funciones importadas son relativamente conocidas. Suelen estar documentadas, accesibles y su comportamiento suele ser el esperado. En pocas palabras, tenemos poco trabajo ahí.

La diferencia la marcan las funciones de usuario. Desconocidas, no documentadas... Es decir, su comportamiento es oscuro, opaco y el objeto de nuestro estudio. Solo pasándolas por nuestra mirada de analista y mucha paciencia podremos descubrir su finalidad. Este va a ser el verdadero núcleo del análisis de código fuente.

Transparencia referencial y funciones puras

Una función que no cambia el estado del programa posee una cualidad muy buscada por los programadores: la **transparencia referencial**. Es mejor verlo con un ejemplo:

```
int double_it(int value) {  
    return value * value;  
}
```

¿Cambia el estado del programa? ¿Muta el objeto que recibe como parámetro? No y no. De hecho, si pudiésemos sustituir el valor que produce esta función allí donde es invocada no cambiaría nada.

Es una **función pura**, porque cumple con las dos condiciones para serlo. Primero: si la llamamos dos veces con el mismo argumento produce el mismo resultado. Segundo: la llamada a la función no produce **efectos colaterales**.

Analizar una función sin efectos colaterales y **referencialmente transparente** no posee muchas complicaciones. Hace una sola cosa y la hace bien. El problema es el inverso: funciones que cambian el estado del programa, muta el objeto que recibe y además dos invocaciones con los mismos parámetros no producen el mismo valor de retorno o resultado. Una pesadilla para el analista y, además, es el caso que más abunda...

2.2.3. PARÁMETROS DE LAS FUNCIONES

De forma indisociable, salvo en el caso de funciones que no toman parámetros, las funciones toman parámetros, según el tipo de estos podemos definir ciertas categorías que van a ayudarnos a entender el comportamiento de estas.

Signaturas, tipos y lenguajes dinámicos

Vamos a hacer un alto aquí y explicar un concepto que probablemente os habréis preguntado antes. Estamos hablando de **tipos**, pero hay lenguajes para los que la definición de tipo no es estática sino dinámica, por ejemplo, en Python:

```
def double_it(value):
    return value * value
```

¿Cuál sería la signatura de esta función? Solo podemos saber que toma un parámetro y que devuelve un valor. Y esto último lo sabemos porque tenemos un ‘return’ en el código porque no hay otra forma de señalizar, de forma estricta que la función retorna un valor²⁴.

En los lenguajes dinámicos el tipo de una variable se resuelve en tiempo de ejecución, no antes, por lo que el compilador no se quejará si lee en el código que ‘double_it’ recibe una cadena o un entero o una lista de enteros.

Hasta que no se ejecute la llamada a la función y el código de esta, no mostrará fallo alguno si el operador '*' tiene sentido para el tipo que le pasemos como parámetro. Sin embargo, en los lenguajes con tipos estáticos esta información es vinculante y ayuda al compilador a detectar errores antes de que el programa se ejecute. De hecho, el programa proseguirá compilándose para detectar más errores o advertencias, pero no culminará con la elaboración de un ejecutable.

Nos centraremos en el tipado estático (en concreto en los de la familia de lenguajes relacionados con C), para que nos proporcione una base de conocimiento que posteriormente podremos aplicar a los lenguajes dinámicos.

²⁴ Python, a partir de la versión 3.5 soporta anotaciones en las declaraciones. Aunque son ignoradas por el compilador, podemos utilizar herramientas como mypy para detectar errores en el uso de tipos.

2.2.4. PASO POR VALOR, PASO POR REFERENCIA (PUNTEROS)

Respecto a los parámetros, existen dos clases fundamentales que deberemos aprender a distinguir bien porque dependiendo a la clase que pertenezca un parámetro nos va a complicar más o menos el análisis.

Antes, para comprenderlo, vamos a ver un ejemplo:

```

1 #include <stdio.h>
2
3 struct Coordenadas {
4     int x;
5     int y;
6 };
7
8 void mover(struct Coordenadas coors, int dx, int dy) {
9     coors.x += dx;
10    coors.y += dy;
11 }
12
13 int main() {
14     struct Coordenadas c = {.x = 100, .y = 100};
15
16     printf("Objeto en x = %i | y = %i\n", c.x, c.y);
17
18     mover(c, 50, 50);
19
20     printf("Objeto en x = %i | y = %i\n", c.x, c.y);
21
22     return 0;
23 }
```

Ahí vemos una estructura que contiene dos coordenadas, 'x' e 'y', una función 'mover' que toma una estructura del tipo 'Coordenada' y dos enteros que representan los ejes 'x' e 'y'.

A plena vista, diríamos que se entiende la intención del programa ¿verdad? Declaramos una variable y la inicializamos a dos valores (uno por coordenada), inspeccionamos su estado, la pasamos ala función para moverla y volvemos a inspeccionar. ¿Qué ocurrirá?

```

→ master_ACF ./a.out
Objeto en x = 100 | y = 100
Objeto en x = 100 | y = 100
→ master_ACF
```

Vaya. ¡Pues el objeto no se ha movido! ¿Por qué?

Si observamos la firma de la función, vemos lo siguiente:

```
void mover(struct Coordenadas coors, int dx, int dy)
```

Toma una estructura del tipo “Coordenadas” y dos enteros. ¿Pero como los toma? De nuevo, ¿Por qué no ha modificado la estructura que le hemos pasado?

La respuesta es porque no hemos pasado nada. Son copias, no los objetos originales. Cuando invocamos una función y sus respectivos parámetros, sino lo indicamos de alguna forma, el compilador **copiará** los valores sin tocar los originales.

La copia de los valores pasados como parámetros tiene sus ventajas y desventajas. El programador es responsable de averiguar si es óptimo pasarlos por valor (copia) o **referencia**.

Cuando veamos parámetros pasados por copia sabremos que no estarán sujetos a mutación, ya que son copias y serán destruidos cuando la función retorne. De hecho, esos objetos existen en la **pila de llamadas de la función**. Un área de memoria cuyos fragmentos son descartados cuando retornan las funciones que los usaron.

Existen lenguajes que por defecto pasan las clases por referencia, otros (Java, por ejemplo), los pasan siempre por valor (tiene truco, porque lo que se copia es una referencia al objeto).

En definitiva, lo que nos tiene que quedar claro es que hay paso por valor y paso por referencia. Además, en dependencia del lenguaje deberemos aplicar o no una notación para indicarlo.

En C, utilizaríamos la dirección de memoria de la estructura que queremos modificar. Esto se hace a través de punteros. Un puntero no es más que un tipo cuyo valor es siempre una dirección de memoria. Una única dirección. Todas las direcciones son del mismo tamaño, así que siempre sabremos (en dependencia de la plataforma) que un puntero posee un valor de n-bytes (o bits, como prefiramos).

Ahora bien, todas las direcciones son del mismo tamaño, pero seguro que habrás notado que tenemos punteros a *int*, a *char*, a *struct*... Es decir, junto con el puntero, le indicamos al compilador **a qué está apuntando**. ¿Para qué? Para que sepa que tamaño tiene el objeto apuntado y de esta forma poder efectuar algo denominado **aritmética de punteros o reservar memoria**; entre otras muchas cosas.

Volvamos al ejemplo. Ahora, cambiamos la firma de la función ‘mover’ y le decimos al compilador que en vez de una copia acepte una dirección de memoria que esté apuntando a una estructura del tipo ‘Coordenada’.

```
void mover(struct Coordenadas *coors, int dx, int dy) {
    coors->x += dx;
    coors->y += dy;
}
```

Observar, simplemente añadiendo un '*' al nombre de la variable le decimos al compilador que lo que esperamos en una dirección a una estructura del tipo 'Coordenada'. (también veréis un cambio cuando accedemos a los campos de la estructura, esas "flechas" en vez de puntos)

También, al invocar a la función, en vez de pasar el nombre del objeto, lo acompañamos de un carácter '&'. Con esto le decimos al compilador que estamos invocando la función y queremos que deposite la dirección de ese objeto y no el objeto en si; esto último, provocaría un error fatal.

```
mover(&c, 50, 50);
```

Compilamos y observamos el cambio:

```
→ master_ACF ./a.out
Objeto en x = 100 | y = 100
Objeto en x = 150 | y = 150
→ master_ACF
```

Como vemos, ahora si hemos cambiado el valor tal y como queríamos. Ahora se cambia el objeto dentro de la función y no una copia, que a pesar de que sus valores se cambien esta copia es destruida cuando se retorna.

Observemos los cambios (mínimos) en global:

```

1 #include <stdio.h>
2
3 struct Coordenadas {
4     int x;
5     int y;
6 };
7
8 void mover(struct Coordenadas *coors, int dx, int dy) {
9     coors->x += dx;
10    coors->y += dy;
11 }
12
13 int main() {
14     struct Coordenadas c = {.x = 100, .y = 100};
15
16     printf("Objeto en x = %i | y = %i\n", c.x, c.y);
17
18     mover(&c, 50, 50);
19
20     printf("Objeto en x = %i | y = %i\n", c.x, c.y);
21
22     return 0;
23 }

```

Importante retener esta información: unos parámetros son pasados por **valor**, eso son copias que no van a modificar el objeto que les es pasado, el original. La función utilizará esa copia y cuando retorne se descartará de la memoria.

Otros parámetros, son pasados por **referencia**. Esos valores son los originales, cualquier modificación se efectuará sobre ellos y mutarán de estado. Cuando la función retorne los cambios en los objetos pasados por parámetros persistirán.

2.2.5. PARÁMETROS CONSTANTES

En ocasiones, para indicar que no se quiere mutar un objeto que es pasado por referencia, se indica al compilador que “vigile” que durante el código de la función dicho objeto no es mutado.

En la familia de lenguajes C, se añade la palabra reservada “const”. Cuando veamos esta palabra junto al parámetro sabremos que la función solo podrá “leer” ese valor, pero no mutarlo.

Veamos que ocurre en nuestro ejemplo anterior si agregamos esa palabra a la firma de la función “mover” e intentamos compilar:

```
void mover(const struct Coordenadas *coors, int dx, int dy) {
    coors->x += dx;
    coors->y += dy;
}
```

Y ahora compilamos:

```
→ master_ACF gcc ref_o_valor.c
ref_o_valor.c:9:12: error: cannot assign to variable 'coors' with const-qualified type 'const struct Coordenadas *'
  coors->x += dx;
  ^~~~~~
ref_o_valor.c:8:38: note: variable 'coors' declared const here
void mover(const struct Coordenadas *coors, int dx, int dy) {
  ^~~~~~
ref_o_valor.c:10:12: error: cannot assign to variable 'coors' with const-qualified type 'const struct Coordenadas *'
  coors->y += dy;
  ^~~~~~
ref_o_valor.c:8:38: note: variable 'coors' declared const here
void mover(const struct Coordenadas *coors, int dx, int dy) {
  ^~~~~~
2 errors generated.
→ master_ACF
```

Como vemos, el compilador vigila que no se mute el estado de la estructura, ya que esta se ha declarado como **constante**. Cualquier cambio que intentemos hacer, será reprobado por el compilador.

Aunque existen formas retorcidas de forzar cambios, cuando encontraremos esta declaración sabremos que ese objeto pasado como parámetros no debería sufrir cambios. Así que cuando estemos analizando código sabremos que el valor no mutará en el código de la función.



2.2.6. PARÁMETROS POR DEFECTO Y NÚMERO VARIABLE DE PARÁMETROS

Parámetros por defecto

Como sabemos, cuando llamamos a una función, los parámetros se sustituyen por los valores adecuados al tipo. El lenguaje C no soporta valores por defecto en los parámetros, esto es, valores que si se omiten cuando se llama a una función se toman por estos, pero esta característica si existe, y la vais a encontrar, en otros lenguajes, por ejemplo, C++:

```
1 #include <stdio.h>
2
3 void f(int i = 100) { printf("%i\n", i); }
4
5 int main() {
6     f(90);
7     f();
8     return 0;
9 }
```

Si compilamos el programa anterior en C dará error, pero no en C++. La salida una vez ejecutado:

```
→ master_ACF ./a.out
90
200
→ master_ACF
```

Número variable de parámetros

En determinado tipo de funciones nos vamos a encontrar que toman parámetros de la forma habitual (o incluso no toman) y posteriormente, en la signatura, vemos una elipsis característica, por ejemplo:

```

1 #include <stdarg.h>
2 #include <stdio.h>
3
4 Void multiple_print(int count, ...) {
5     va_list args;
6     int current;
7
8     va_start(args, count);
9
10    while (count-- != 0) {
11        current = va_arg(args, int);
12        printf("%i\n", current);
13    }
14
15    va_end(args);
16 }
17
18 int main() {
19     multiple_print(3, 1, 2, 3);
20     return 0;
21 }
```

La sintaxis es compleja, ya que se ayuda de macros definidas en el archivo de cabeceras "stdarg".

Por fortuna, en otros lenguajes la sintaxis es más amigable, un ejemplo en Java:

The screenshot shows a file structure on the left with files: MyClass.java, Var.java (selected), and maven-lib. On the right, the code for Var.java is displayed:

```

1 public class Var{
2     public static void variable_args(int ...args) {
3         for (int i: args) {
4             System.out.println(i);
5         }
6     }
7 }
```

Below the code, there's an "Execute Mode, Version, Inputs & Arguments" section with "JDK 11.0.4" selected, "Interactive" mode checked, and "Stdin Inputs" empty. Under "CommandLine Arguments", there's a text input field containing "1 2 3 4". At the bottom are "Execute" and "..." buttons.

Result
CPU Time: 0.12 sec(s), Memory: 30384 kilobyte(s)

```

1
2
3
4
```

Lo importante: recordad que algunas funciones toman un número variable de parámetros. Es muy común el uso de la elipsis (...) en la firma.

2.3. MODULARIDAD Y ÁMBITO (VISIBILIDAD)

Ahora que los ingenieros de software disponían de una poderosa herramienta: la función. Vieron que era más fácil agrupar un conjunto de funciones de concepto familiar listo para ser reutilizado cuando hiciese falta. De este modo, podrían aislar un grupo de funciones relacionadas del programa principal y aprovechar ese código escrito para otros programas.

Nace el concepto de módulo. Conjunto de funciones y/o clases (dependiendo del soporte de paradigmas del lenguaje) relacionadas entre si y exportables. Con ello, despegó la reutilización del código, lo que supondrá un ahorro impresionante de costes.

En el análisis de código el concepto de módulo es muy importante. Nos ayudará a ver las relaciones entre estos y el programa principal objeto de análisis.

Seguro que has visto la inclusión típica de un archivo de cabecera en C o C++ y de una importación de módulos (también llamado paquete) en Java o Python. Diversos ejemplos:

C:

```
#include <stdio.h>
```

C++:

```
#include <iostream>
```

C++20:

```
import <iostream>;
```

Java:

```
import java.util.Collection;
```

Python:

```
import sys
```

La sintaxis es similar y suele encontrarse justo en la cabecera de los archivos que estemos analizando. Con excepciones, veremos importaciones dentro de otras estructuras, se trata de una mala práctica de programación que suele ser usada.

Con este tipo de declaraciones, el archivo objeto de análisis está demandando funcionalidad que no posee y que necesita. Son pistas para nosotros, puesto que viendo la organización de importaciones podremos incluso medir la complejidad de dicho módulo, su acoplamiento con el resto y, sobre todo, pistas de que está haciendo (por ejemplo, si importa funcionalidad de red o de cifrado)

2.3.1. IMPORTACIONES Y EXPORTACIONES

Rara vez un módulo existe aislado, sin necesidad de importar funcionalidad del exterior. Tanto puede importarla como exportarla. Para ello, como hemos visto, utiliza un mecanismo de importación (veremos también la exportación).

Bien, cuando analizamos un archivo cualquiera comenzamos por ver como se relaciona con su entorno. Vemos sus importaciones y las dotamos de un nivel:

- Sistema
- Biblioteca estándar
- Biblioteca de terceros
- Módulos propios

Recordad cuando hablamos de “Tipos de funciones” anteriormente. Como estas están categorizadas de varias formas y una de ellas es su procedencia. En los módulos, por ser estos un conjunto de funciones, también se da una relación parecida.

Sistema

Son las funciones que expone el sistema operativo en forma de API²⁵ Por lo general, solo en lenguajes C y C++ están expuestas directamente (con excepciones). La otra forma de interactuar con ellas es a través de una librería que incluya llamadas a estas funciones (habitualmente suele ser una librería de funciones que por debajo consume la librería de C)

Este tipo de funciones conectan con los recursos que administra el sistema operativo. Es decir, el archivo que importa este tipo de funciones va a necesitar o hace uso, con mucha probabilidad, de un **recurso del sistema operativo**.

Para conocer las distintas API de sistemas operativos podemos recurrir a recursos de documentación tales como:

Linux: <https://man7.org/tlpi/>

Windows: <https://docs.microsoft.com/en-us/windows/win32/api/>

Afortunadamente, aunque con excepciones, la documentación disponible es abundante, comentada y con numerosos ejemplos de su funcionamiento.

²⁵ Application Programming Interface

Biblioteca estándar

Habitualmente, para la mayoría de los lenguajes de programación no esotéricos²⁶, el lenguaje viene acompañado de una librería completa de módulos disponibles para usar “de serie”. Es una buena ventaja y noticia, ya que, de nuevo, la documentación suele ser de una alta calidad, disponibilidad y fiable; como siempre, con excepciones.

Para C y C++ tenemos recursos tales como:

<https://en.cppreference.com/w/>

Para Java y Python, respectivamente:

<https://docs.oracle.com/en/java/javase/15/index.html>

<https://docs.python.org/3/>

Bibliotecas de terceros

Obviamente, no todas las API de sistema junto con las bibliotecas pueden implementar todos los casos de uso y necesidades, por lo que el ecosistema de librerías de terceros disponibles suele ser abundante.

Casi todos los lenguajes de programación disponen de un repositorio de paquetes (módulos) disponible para descargar. Quizás, las excepciones con C y C++, donde no existe un solo lugar que unifique las librerías disponibles, por lo que las encontraremos dispersas y con sus propias particularidades como, métodos de compilación y enlace, documentación, uso y licencias.

La documentación va a depender del apoyo que posea el proyecto y el tiempo que sus mantenedores le dediquen a esta. Una advertencia al respecto: **la documentación suele ser obsoleta**. Es bastante posible que veáis usos de una biblioteca de terceros que no esté documentada; en tales casos, solo nos queda recurrir directamente al código.

Las ventajas del uso de estas bibliotecas son claras: en la mayoría de los casos están disponibles públicamente o podemos indagar en su código. Además, si podemos asegurarnos de que esta no ha sido manipulada (mediante comparación o cálculo de hashes) podremos descartar comportamiento malicioso a priori; es decir, podemos descartar profundizar en su análisis.

Algunos ejemplos de repositorios centrales de código:

Javascript (Node):

<https://www.npmjs.com/>

Python:

<https://pypi.org/>

²⁶ https://esolangs.org/wiki/Main_Page

Módulos propios

En este caso, nos estamos refiriendo a conjuntos de archivos que proveen de funcionalidad al resto del programa. Son bibliotecas de funciones (o clases) que el propio programa posee, pero que no son la línea central del programa, están ahí para apoyar la función principal del programa.

Cuando el programa principal llame a estas funciones no nos quedará más remedio que estudiarlas y crear anotaciones. No estarán documentadas (salvo que estemos, precisamente, analizando un programa o librería de código abierto) y posiblemente, el código no esté disponible o esté, pero ofuscado, peor aun, será una librería binaria que tendremos que desensamblar o decompilar.

Aquí entramos ya en la verdadera tarea del analista. El resto es “hacerse una idea” e ir despejando el camino formándonos un mapa. El resto, como dirían los exploradores: territorio sin cartografiar.

Un ejemplo, el código fuente de Mirai (una botnet), uno de sus módulos es un conjunto de cabecera y definición (archivos con extensiones .h y .c respectivamente) para simplemente calcular dos tipos de checksum²⁷:

```

1 #define __GNU_SOURCE
2
3 #include <arpa/inet.h>
4 #include <linux/ip.h>
5
6 #include "includes.h"
7 #include "checksum.h"
8
9 uint16_t checksum_generic(uint16_t *addr, uint32_t count)
10 {
11     register unsigned long sum = 0;
12
13     for (sum = 0; count > 1; count -= 2)
14         sum += *addr++;
15     if (count == 1)
16         sum += (char)*addr;
17
18     sum = (sum >> 16) + (sum & 0xFFFF);
19     sum += (sum >> 16);
20
21     return ~sum;
22 }
23
24 uint16_t checksum_tcpudp(struct iphdr *iph, void *buff, uint16_t data_len, int len)
25 {
26     const uint16_t *buf = buff;
27     uint32_t ip_src = iph->saddr;
28     uint32_t ip_dst = iph->daddr;
29     uint32_t sum = 0;
30     int length = len;
31
32     while (len > 1)
33     {
34         sum += *buf;
35         buf++;
36         len -= 2;
37     }
38
39     if (len == 1)
40         sum += *((uint8_t *) buf);
41
42     sum += (ip_src >> 16) & 0xFFFF;
43     sum += ip_src & 0xFFFF;
44     sum += (ip_dst >> 16) & 0xFFFF;
45     sum += ip_dst & 0xFFFF;
46     sum += htons(iph->protocol);
47     sum += data_len;
48
49     while (sum >> 16)
50         sum = (sum & 0xFFFF) + (sum >> 16);
51
52     return ((uint16_t) (~sum));
53 }
```

²⁷ https://es.wikipedia.org/wiki/Suma_de_verificación

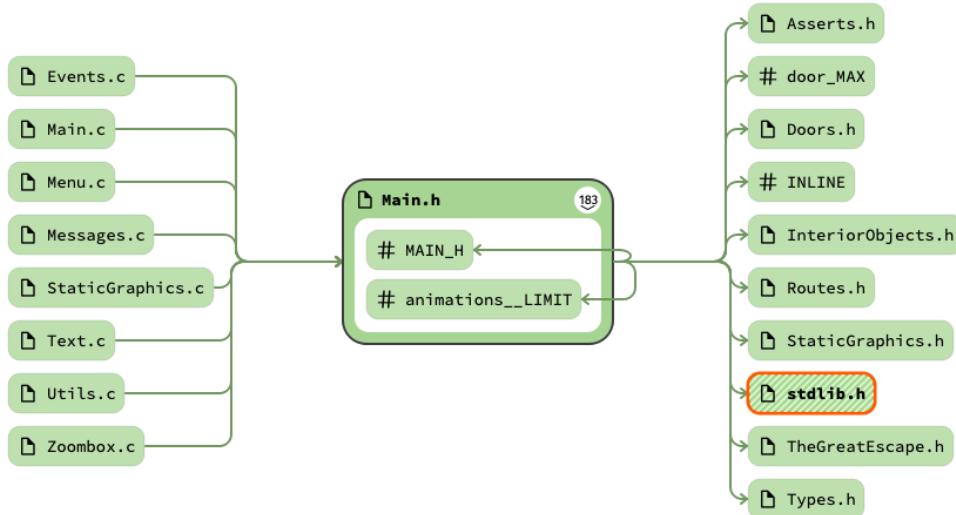
El código en si es inofensivo, fácilmente de entender y cumple una sola función (en el sentido de funcionalidad). Es usado por el programa central para apoyarse, pero no forma parte del núcleo o intención del programa. Está ahí para **realizar una tarea auxiliar**.

Naturalmente, huelga decir que si lo que estamos realizando es una búsqueda de vulnerabilidades si entraría en nuestro ámbito; no nos olvidemos que formará parte del programa y tarde o temprano si encontramos un fallo en alguna función de este tipo de bibliotecas tan solo tenemos que buscar donde son usadas para examinar su viabilidad para explotarlas.

2.3.2. GRAFO RELACIONAL DE DEPENDENCIAS

Como hemos estado viendo, los módulos permiten importar la funcionalidad de otros. Esto forma una relación en forma de grafo, siendo los vértices los módulos y las aristas las importaciones que hacen entre ellos.

En la siguiente imagen podemos ver un ejemplo de un grafo parcial de un archivo que pertenece a un videojuego. En el centro tenemos un archivo de cabecera, "Main.h" que es **dependencia-de** e incluye **dependencias**.



Esto nos permite saber tanto donde son usadas las definiciones que porta un archivo como las que usa. Observad, que es incluido en ciertos archivos (grupo de la izquierda) y, a su vez, el mismo incluye otros (grupo de la derecha)

Importante: El hecho de que un archivo incluya módulos o sea incluido como módulo de otro, no significa necesariamente que use las definiciones que contiene. Aunque se trata de una mala praxis de programación (incluir un módulo para luego no usarlo) es común y hemos de estar atentos a esta particularidad. Podemos con total libertad eliminar dicha importación si hemos comprobado previamente que es neutra.

2.3.3. ÁMBITO Y VISIBILIDAD

Cuando hablamos de funciones y variables (o clases) debemos tener en cuenta algo bastante importante de cara a los análisis de código: que alcance posee una variable. Es decir, desde donde puede ser vista. Existen variables que cualquier módulo puede referenciar: las **variables globales**. Tenemos variables que solo pueden ver las funciones o clases declaradas en el mismo módulo: **variables estáticas, unitarias o de módulo** (su nombre puede variar según el lenguaje de programación). Finalmente, tenemos las variables definidas dentro de las funciones, métodos o clases, denominadas generalmente **variables locales**.

El ámbito, visibilidad o vida de una variable viene definido o limitado por el lugar de declaración. Así, una variable local a una función solo existirá (salvo una excepción que comentaremos) cuando se llame a esa función, mientras que una variable global estará siempre disponible, desde que comienza a ejecutarse el programa hasta el término de su ejecución.

Variables globales

Las variables globales son, a su vez, desaconsejables pero necesarias en algunos entornos. Lo son porque mantener un estado global dificulta la programación concurrente y es un claro antipatrón²⁸ que da bastantes problemas.

Por otro lado, se hacen inevitables cuando, por ejemplo, debemos leer una variable de entorno (que no es más que una variable global en el contexto de ejecución) para obtener cierto valor de configuración o similar.

Cuando analizamos código, el mayor problema de las variables globales es que aumentan el ámbito de influencia (en inglés, *scope*) de la función que estemos analizando en ese momento.

Básicamente, cuando mayor es el ámbito de una variable, mayor probabilidad existe de que su estado sea mutado por una función a un valor que posteriormente usemos en otra y no sea el que se está esperando.

En lenguajes como C o Javascript, donde abundan las variables globales, basta con declararlas en cualquier librería o archivo, fuera del cuerpo de una función, clase o estructura. En cuanto hagamos uso de dicho archivo o librería, la variable pertenecerá al ámbito global.

De hecho, el abuso de variables globales posee una definición en el estándar CWE de vulnerabilidades:

<https://cwe.mitre.org/data/definitions/1108.html>

Veamos un ejemplo, observad la variable “id_tag” de los fragmentos de código que se muestran:

²⁸ <https://wiki.c2.com/?GlobalVariablesAreBad>

loader/src/main.c

```

13  #include "headers/util.h"
14
15  static void *stats_thread(void *);
16
17  static struct server *srv;
18
19  char *id_tag = "telnet";
...
41  if (argc == 2)
42  {
43      id_tag = args[1];
44  }
45
46  if (!binary_init())
47  {
48      printf("Failed to load bins/dlr.* as dropper\n");

```

● C Showing the top two matches Last indexed on 5 Jul 2018

loader/src/headers/includes.h

```

30  #define EXEC_QUERY    "/bin/busybox IHCEE"
31  #define EXEC_RESPONSE "IHCEE: applet not found"
32
33  #define FN_DROPPER   "upnp"
34  #define FN_BINARY    "dvrHelper"
35
36  extern char *id_tag;

```

● C Showing the top match Last indexed on 5 Jul 2018

El primer fragmento muestra una variable con el tipo: “puntero a char”, que básicamente en C es que el puntero está apuntando al primer carácter de una cadena de texto, en este ejemplo concreto, la cadena “telnet”. Como vemos, la variable está declarada e inicializada en el archivo “loader/src/main.c”.

Pero también vemos que está “referenciada” de alguna forma en el archivo “loader/src/headers/includes.h”. Pero no la inicializa, simplemente la declara y además antepone una palabra reservada “extern”.

Dicha palabra, le indica al compilador que el nombre “id_tag” está definido en otro archivo y cuando el programa se compile, entonces y solo entonces, podrá resolverla a su valor.

Ahora, todo archivo que incluya el archivo “loader/src/headers/includes.h” podrá ver la variable “id_tag” y lo peor de todo no es que puedan verla, sino que pueden cambiar su valor sin que el resto lo advierta (por ello, las variables globales, son un patrón a evitar en programación concurrente)

En C, una manera de mantener las variables fuera de ámbito de función o estructura es añadiéndoles la palabra reservada **static**. Esto convertiría a la variable global en estática al módulo y no podría ser alcanzada fuera de este. Lo veremos en el siguiente subapartado.

Variables estáticas, de módulo o unidad

Estas variables existen dentro de un archivo de código fuente. No son globales y por lo tanto otros archivos no pueden referenciar a estas. Con la llegada de la orientación a objetos, este tipo de variables suelen ser encapsuladas en una clase, mientras que en lenguajes como C se siguen manteniendo y son tremadamente comunes.

En C, es común, para separarla del caso de las variables globales, añadir el cualificador “static” cuando se está declarando la variable o función (en realidad, a lo que afecta es al “símbolo” y no al tipo)

Vamos a observar un caso práctico, declaramos una variable global y otra “static”:

```

1 #include <stdio.h>
2
3 #include "constatic.h"
4
5 static const char *si_static = "variable solo disponible en este modulo";
6 const char *no_static = "variable disponible globalmente";
7
8 void funcion_de_static(const char *cad) { printf("%s\n", cad); }

```

Simple ¿verdad? Llamaremos a este archivo “constatic.c” y creamos otro que haga uso de la función “función_de_static” con los dos casos:

Primer caso: desde nuestro programa la indicamos al compilador que haremos uso de la variable global “no_static” y la usamos en la función también importada. Intentamos compilar y ejecutamos.

```

1 #include <stdio.h>
2
3 #include "constatic.h"
4
5 extern const char *no_static;
6
7 int main() {
8     funcion_de_static(no_static);
9     return 0;
10 }

```

Como vemos, podemos compilar y ejecutar de forma normal.

```

→ master_ACF gcc static1.c constatic.c
→ master_ACF ./a.out
variable disponible globalmente
→ master_ACF

```

Ahora intentamos hacer uso de la variable cualificada con “static” denominada “si_static”:

```

1 #include <stdio.h>
2
3 #include "constatic.h"
4
5 extern const char *si_static;
6
7 int main() {
8     funcion_de_static(si_static);
9     return 0;
10 }

```

Compilamos y observamos que el enlazador (el programa que debe asegurarse que un ejecutable puede hacer uso de todos los símbolos -nombres- que posee el código) muestra un error advirtiendo que no puede hallar el símbolo pedido; esto es, “si_estatic”:

```

→ master_ACF gcc static1.c constatic.c
Undefined symbols for architecture x86_64:
  "_si_static", referenced from:
    _main in static1-01dc1d.o
ld: symbol(s) not found for architecture x86_64
clang: error: linker command failed with exit code 1 (use -v to see invocation)
→ master_ACF ┌

```

Cuando veáis una variable en el código con el cualificador “**static**” pensad que solo puede ser usada allí donde es declarada. Es una forma de limitar la influencia de un símbolo a su unidad de compilación y una práctica absolutamente recomendada en la programación en C.

Variables locales

Aquí, sin ninguna duda, nos estamos refiriendo a aquellas variables que están asociadas a un **bloque de código**. Salvo en lenguajes de programación como Python o Ruby, donde se usa el espacio horizontal para definir bloques, es muy común el uso de paréntesis “{}”.

Los paréntesis (por generalizar) definen, como hemos dicho, un bloque de código. Este concepto es importante, dado que como veremos posteriormente, los análisis pasan por fragmentar el código en piezas cada vez más pequeñas hasta llegar a la unidad de ejecución más pequeña.

Habitualmente, escucharéis el concepto de variable local muy asociado a funciones. Esto es cierto, pero también se usa en variables que pertenecen a bloques de código de bucles o de elementos de bifurcación (más adelante los veremos).

Un ejemplo de localidad, bastante esclarecedor que prácticamente explica de forma perfecta lo que queremos definir:

```

1 #include <stdio.h>
2
3 struct point {
4     int x;
5     int y;
6 };
7
8
9 int main()
10 {
11     const char *cadena = "fuera del bloque";
12     {
13         const char *cadena = "dentro del bloque";
14         printf("%s\n", cadena);
15     }
16
17     printf("%s\n", cadena);
18     return 0;
19 }
20

```

dentro del bloque
 fuera del bloque

Si observáis con atención, dentro de la misma función “main” existen dos variables declaradas con idéntico símbolo. Hagamonos las siguientes preguntas:

- 1.- ¿Por qué nos ha permitido el compilador declarar el mismo símbolo?
- 2.- ¿Por qué la segunda llamada a “printf” no ha imprimido “dentro del bloque”?
- 3.- Si elimininássemos la segunda declaración de cadena ¿qué habría imprimido el primer “printf”?

Respuestas

1.- Porque a pesar de coincidir en el nombre, no son el mismo símbolo. Existe una variable “cadena” en main y existe otra variable “cadena” en un bloque de código anónimo dentro de “main”. A pesar del nombre, los ámbitos son diferente y eso se tiene en cuenta cuando se está compilando el programa.

2.- Porque cuando se busca la definición de un símbolo nunca se busca en bloques de código internos, siempre se busca el símbolo “desde dentro hacia afuera”.

3.- Consecuencia de la respuesta a la pregunta 2.

Variables locales y estáticas

Para terminar de complicar las definiciones. Como sabemos, las variables se construyen cuando la ejecución pasa dentro de una función y se destruyen cuando se sale de la función. Como consecuencia de esta destrucción de variables se crea, en ocasiones, fugas de memoria al olvidar liberar memoria, recursos, etc. Esto es así por el funcionamiento propio de la zona de memoria de un proceso conocida como pila²⁹.

No obstante, existe una forma curiosa de hacer que una variable declarada dentro de una función

²⁹ https://en.wikipedia.org/wiki/Stack-based_memory_allocation

persiste llamada tras llamada. Es decir, dentro de la función toma un valor y cuando se vuelve a llamar a la función el valor sigue allí.

Es una forma de decirle al compilador: no destruyas el valor de esta variable. Para ello, en lenguaje C, usaremos un cualificador que ya habéis visto antes, nuestro **static**.

Veamos un ejemplo:

```
1 #include <stdio.h>
2
3 int incrementar_x() {
4     static int x = 0;
5     x++;
6     return x;
7 }
8
9
10 int main()
11 {
12     int i = 0;
13     while (i++ < 10) printf("valor de x: %i\n", incrementar_x());
14
15     return 0;
16 }
17
```

```
valor de x: 1
valor de x: 2
valor de x: 3
valor de x: 4
valor de x: 5
valor de x: 6
valor de x: 7
valor de x: 8
valor de x: 9
valor de x: 10
```

Fíjate como “guarda” el valor inicial y lo va actualizando con cada llamada.

Cuando veáis un static acompañando a una variable dentro de un bloque de código, dicho valor no se destruirá y persistirá a lo largo del programa.

¿Por qué?

En vez de crear un espacio en la pila para esta variable, el compilador la sitúa en una zona de la memoria denominada “memoria inicializada”. Como el compilador ya sabe cuál es el valor de estas variables y que estarán presentes a lo largo de todo el programa no tiene porque guardarlas en una pila o en el montículo, crea una zona especial que siempre tiene el mismo tamaño y así se ahorra su gestión o la minimiza.

Por zona de “memoria inicializada” entendemos allí donde se guardan todas las variables globales y estáticas y cadenas que poseen un valor cuando se declaran en el código. Si no las declaramos, es decir, si tenemos variables globales o estáticas sin un valor predeterminado, se crea otra zona aneja

a esta denominada: BSS³⁰ (block starting symbol) o “memoria no inicializada” por oposición a la anterior denominación.

2.4. ¿POR QUÉ ES ÚTIL ESTA INFORMACIÓN?

Análisis, como palabra, significa³¹: distinguir y separar las partes de un todo hasta llegar a conocer sus principios y elementos.

Como analistas de malware, *bughunters* o ingenieros de *reversing*, es precisamente lo que debemos hacer para llegar a comprender un programa o librería; que es precisamente el objeto central de estudio de la profesión.

Ahora bien, para distinguir y separar las partes necesariamente debemos aprender, comprender y asimilar cuales son estas partes. Sino interiorizamos esto, no podremos, de ningún modo, emprender la tarea del análisis.

Esta primera parte de “Elementos Básicos del Análisis de Código” muestra precisamente los principios y elementos.

Como capas de un mismo objeto, un programa es el conjunto de módulos propios, librerías de terceros y del sistema. Dentro de estas tenemos las funciones y objetos, partículas más pequeñas llenas de la funcionalidad real del programa. Son “aquellos que se ejecuta”.

En la segunda parte, veremos el resto de los elementos con los que podremos separar en partículas más pequeñas aún.

2.5. ¿QUÉ TENGO QUE HABER APRENDIDO?

- Entender que un programa no es un todo sino un conjunto de piezas orquestadas por un punto de inicio, entrada o ejecución.
- Poder, a grandes rasgos, entender la estructura de un programa viendo las importaciones que realizan los distintos módulos.
- Identificar y clasificar las librerías que usa un programa, ya sean propias, de terceros, estándar o del sistema.
- Identificar y clasificar las funciones que usa un módulo y clasificarlas según los distintos tipos que existen.
- Observar los parámetros (signatura de una función) y diferenciar entre copia del valor y referencia (especialmente los punteros)

³⁰ <https://en.wikipedia.org/wiki/.bss>

³¹ <https://www.rae.es/drae2001/an%C3%A1lisis>

3. ELEMENTOS BÁSICOS DEL ANÁLISIS DE CÓDIGO (II)

La programación estructurada, como ya vimos, es el paradigma más básico de la programación y uno de los más extendidos. Incluso lenguajes orientados a objetos poseen como base este paradigma; por ejemplo, C++ que agrega objetos y programación genérica (vía plantillas) al lenguaje C, el cual es esencialmente un lenguaje estructurado.

El paradigma estructurado se basa en ciertas estructuras básicas llamadas **estructuras de control** que, combinadas entre sí, permiten controlar el flujo de un programa determinado por los parámetros de entrada.

Las estructuras de control son las siguientes:

- Secuencia
- Selección
- Iteración
- Recursión

Algunos autores indican que iteración y recursión pueden situarse en la misma categoría: “repetición”. Aunque poseen idéntico significado, vamos a verlas por separado.

3.1. SECUENCIA

Una secuencia es un conjunto de sentencias que son ejecutados en orden (generalmente, de arriba abajo). Un programa es en realidad un gran conjunto de sentencias, que, si bien son ejecutadas en orden mediante la selección e iteración (o recursión) es posible modificar la dirección del flujo de ejecución.

Un bloque simple de código ya es una secuencia:

```
{  
    let x = 100;  
    let y = x * 2;  
}
```

Primero tendrá lugar la declaración de la variable ‘x’, después la de la variable ‘y’. Ese orden se mantendrá así por el flujo de ejecución. Demasiado básico ¿verdad? Vamos a complicarlo.

Por el hecho de que su ejecución sea secuencial no significa que deban ejecutarse en el mismo espacio o localización. De hecho, cuando llamamos a una función lo que en realidad estamos haciendo es cambiar el flujo de ejecución a otra **localización**; ya sea dentro de nuestro programa o -y esto es un concepto importante- a código externo a nuestro programa, es decir, a una librería.

Podemos ver la llamada a funciones como un salto que traslada temporalmente el flujo de ejecución a otro lugar.

```
{
    let x = 100;
    print (x);
    let y = x * 2;
}
```

Como vemos en el ejemplo de arriba, el código va a ejecutarse de forma secuencial, pero cuando lleguemos a la llamada a la función “print”, el flujo saltará a dicha localización y la ejecución proseguirá hasta que retorne desde el conjunto de sentencias que llamamos “print”.

Otra forma de ver este salto es como **salto no condicional o salto incondicional**. Esto de debido a que el salto de localización se efectuará sin que ninguna condición intervenga en dicha decisión.

Probablemente, habrás leído acerca de la sentencia “GOTO” o “goto” de algunos lenguajes de programación. Si bien, se desaconseja su uso³², en determinados patrones su uso puede dar lugar a un código bien estructurado. Por ejemplo, en C se usa dentro de una misma función para tareas de liberación de recursos, para tratamiento de errores, etc.

Un ejemplo muy simple tomado del código fuente del kernel Linux³³:

```
{
    int result = 0;
    char *buffer;

    buffer = kmalloc(SIZE, GFP_KERNEL);
    if (!buffer)
        return -ENOMEM;

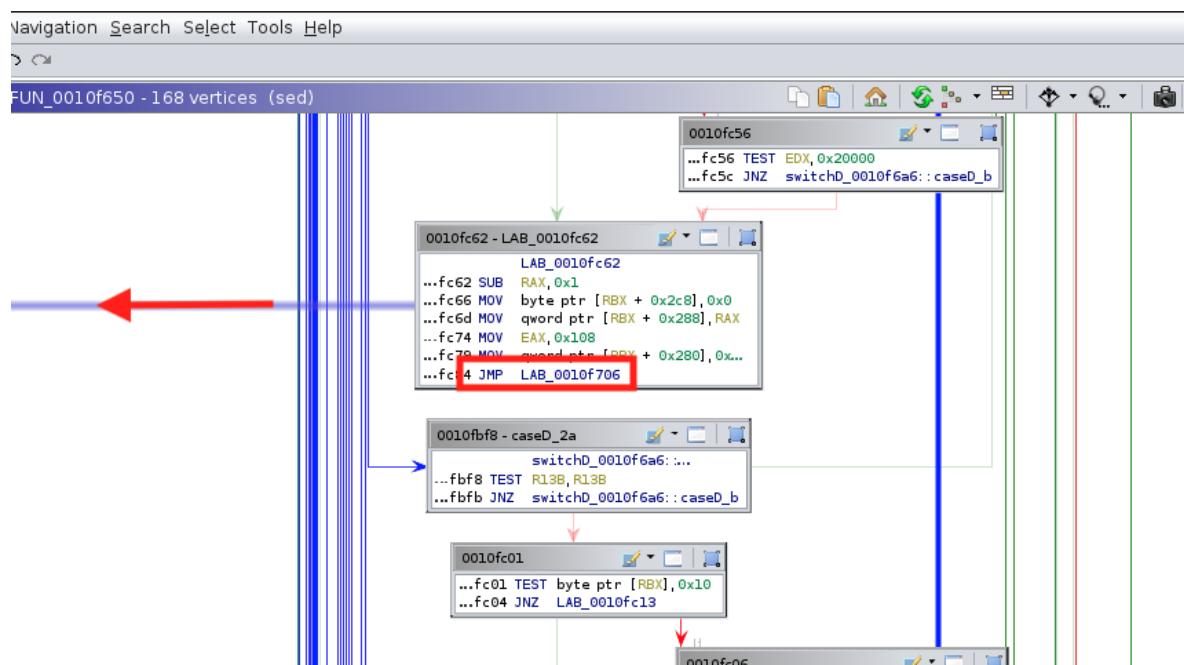
    if (condition1) {
        while (loop1) {
            ...
        }
        result = 1;
        goto out_free_buffer;
    }
    ...
out_free_buffer:
    kfree(buffer);
    return result;
}
```

En el lenguaje ensamblador el uso de saltos incondicionales es bastante habitual. En concreto, en el ensamblador x86 o x64, en la forma de instrucción mnemotécnica “JMP” o “JUMP”³⁴.

³² <https://en.wikipedia.org/wiki/Goto#Criticism>

³³ <https://www.kernel.org/doc/html/v4.18/process/coding-style.html#centralized-exiting-of-functions>

³⁴ https://en.wikibooks.org/wiki/X86_Assembly/Control_Flow#Unconditional_Jumps



Detalle del desensamblado de una función en un ejecutable. Obsérvese el salto incondicional al finalizar el segmento de código. La única ruta posible tras dicha secuencia de código es la indicada por la flecha roja a la izquierda.

Nos ha de quedar claro que una secuencia es un conjunto de instrucciones, expresiones y sentencias que van a ser ejecutadas en orden (habitualmente, representado de arriba abajo). Sin embargo, “en orden” no significa que este no pueda desplazarse de localización, algo que complica los esfuerzos de los analistas dado que los saltos complican el seguimiento del flujo de ejecución.

Construcciones típicas de salto incondicionales en la secuencia son las llamadas a función “call” o el propio retorno de una función “return”, además de los ya vistos “jmp” o “jump” y “goto”.

La representación gráfica de una secuencia es bastante simple y autoexplicativa:



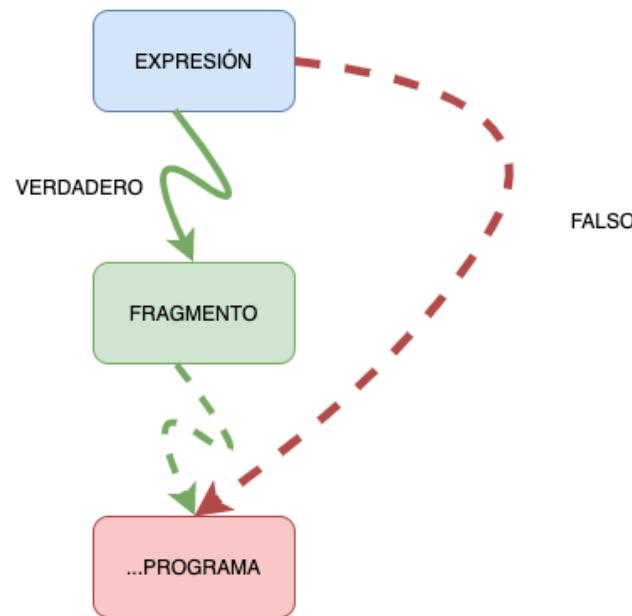
3.2. SELECCIÓN

La **selección** permite realizar saltos en el código, pero de manera **condicional**. Es decir, si en la secuencia los saltos que vimos en la **secuencia** eran **incondicionales** (se salta si o si) ahora veremos que es posible efectuar un salto en el código en dependencia del resultado de una expresión.

En primer lugar, debemos de saber que toda instrucción de selección implica que habrá una parte que no se ejecute. Eso complica el análisis puesto que en una lectura del código no podremos, en la mayoría de los casos, determinar qué ocurrirá; puesto que dependerá de los valores que tome la expresión lógica o relacional que, habitualmente, contiene toda selección.

Algunos autores nombran a la **selección** como **bifurcación**.

Lo que debemos tener muy claro es que una selección consta siempre de una expresión a evaluar y un salto o bien a una zona de código o precisamente para que no se ejecute ese fragmento. Vamos a verlo de forma visual:



Es el esquema más simple de la selección, de hecho, se le llama **selección o bifurcación simples**. Esto es debido a que la condición se evalúa para disponer si se va a ejecutar o no cierto fragmento de código. Es decir, dependiendo del valor de la expresión el programa ejecutará las siguientes líneas o realizará un salto que las omitirá. Un ejemplo en código (procede del kernel Linux):

```

link = readl(&regs->GigLnkState);
if (link & LNK_1000MB)
    speed = SPEED_1000;
else {
    link = readl(&regs->FastLnkState);
}
  
```

Como podemos apreciar, si el valor que toma la variable “link” de la salida o retorno de la función “readl” es igual a la constante “LNK_1000MB”, la variable “speed” tomará el valor de la constante “SPEED_1000”. De lo contrario, “link” volverá a tomar un nuevo valor de la función “readl”.

¿Qué es importante en un análisis de código viendo esta estructura?

Si alguna vez has jugado al ajedrez, habrás observado que mucho del análisis de éste se efectúa intentando adivinar qué ocurrirá si movemos cierta pieza a cierta casilla. ¿Qué hará el contrario? Y así, sucesivamente.

En el análisis de código, y en particular en este tipo de estructura, ocurre exactamente lo mismo: ¿Qué valor tomará cierta expresión lógica? ¿Verdadero o falso? Al final, se forma un árbol de búsqueda³⁵ o decisiones que va a definir la complejidad del código que estamos tratando.

Nuestro análisis respecto al código teniendo en cuenta estas estructuras debe recorrer todas las opciones. No obstante, existen algunas estrategias para “podar” dicho árbol y no terminar abrumados por la dimensión del problema³⁶.

La principal de ellas es tomar siempre la línea de decisión más probable. ¿Cómo sabemos cual es? Por ejemplo, en determinadas ocasiones nos veremos con bifurcaciones que son simplemente estructuras para controlar errores y finalizar el programa. Eliminaremos dichas opciones fluyendo siempre por el trazado que sí se va a ejecutar; en otras palabras, seremos optimistas respecto a la ejecución.

Observemos el siguiente fragmento donde se analiza un malware que realiza detección de máquinas virtuales (tomada de ³⁷):

```

0001F48C E9 70 01+call  FormatString
0001F491 8B 55 C0 mov    edx, [ebp+var_48]
0001F494 8B 6C FC+mov    eax, offset aVMware ; "VMWARE"
0001F499 E8 E6 54+call  cmpString
0001F49E 85 C0 test   eax, eax
0001F4A0 0F 85 91+jnz loc_B01FC37

0001F4A6 8D 45 B4 lea    eax, [ebp+var_4C]
0001F4A9 E8 EE D3+call  enumDisplays
0001F4AE 8B 45 B4 mov    eax, [ebp+var_4C]
0001F4B1 8D 55 B8 lea    edx, [ebp+var_48]
0001F4B4 E8 53 81+call  FormatString
0001F4B9 8B 55 B8 mov    edx, [ebp+var_48]
0001F4BC 8B 7C FC+mov    eax, offset aVirtual ; "VIRTUAL"
0001F4C1 E8 BE 54+call  cmpString
0001F4C6 85 C0 test   eax, eax
0001F4C8 0F 85 69+jnz loc_B01FC37

0001F4CE 8D 45 AC lea    eax, [ebp+var_54]
0001F4D1 E8 C6 D3+call  enumDisplays
0001F4D6 8B 45 AC mov    eax, [ebp+var_54]
0001F4D9 8D 55 B0 lea    edx, [ebp+var_50]
0001F4DC E8 2B 81+call  FormatString
0001F4E1 8B 55 B0 mov    edx, [ebp+var_50]
0001F4E4 8B 8C FC+mov    eax, offset aUnbox ; "UNBOX"
0001F4E9 E8 96 54+call  cmpString
0001F4EE 85 C0 test   eax, eax

```

262) UNKNOWN 0B01F500: unpackedMain+AC

Si observamos que las rutinas de detección **finalizan en su conjunto en una salida del programa**, daremos por hecho que no detecta la máquina virtual y seguiremos analizando el comportamiento de la muestra para ver lo que hace realmente. Fijaros que es importante observar que la salida neutraliza el programa o nos lleva a áreas de este que son insustanciales para nuestro análisis. En

³⁵ https://www.chessprogramming.org/Search_Tree

³⁶ https://es.wikipedia.org/wiki/Complejidad_en_los_juegos

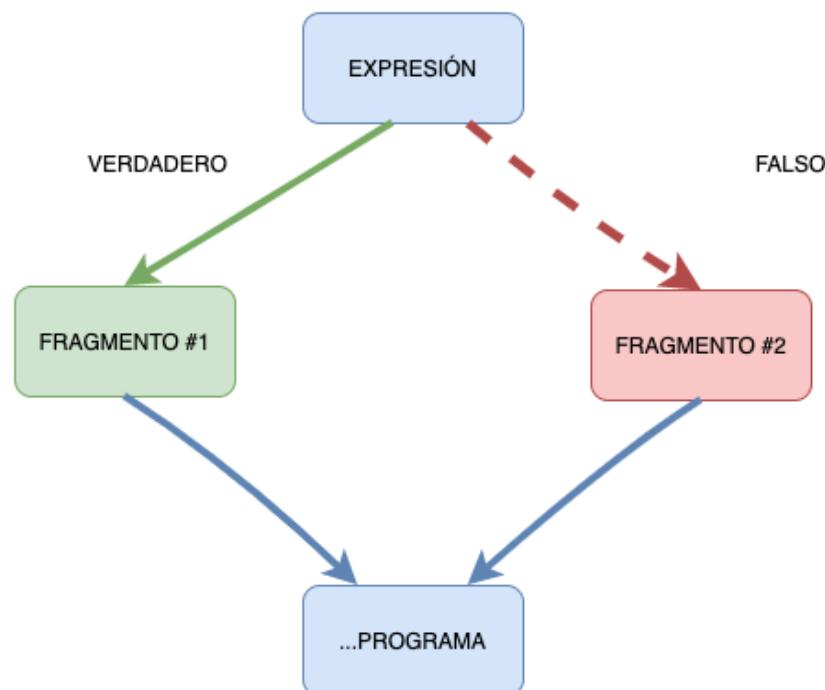
³⁷ <https://blog.malwarebytes.com/threat-analysis/2014/02/a-look-at-malware-with-virtual-machine-detection/>

otras palabras, deberemos centrarnos en las bifurcaciones de interés e ir descartando aquellas que finalizan en vías muertas.

En ocasiones, las estructuras de selección se limitan a comprobar que el valor de retorno de las funciones es el adecuado. De nuevo, seremos optimistas y seguiremos analizando y dando por hecho que no va a darse el error. En definitiva, se trata de tomar aquellas alternativas en las que la ejecución es optimista respecto a las expresiones que dominan la selección.

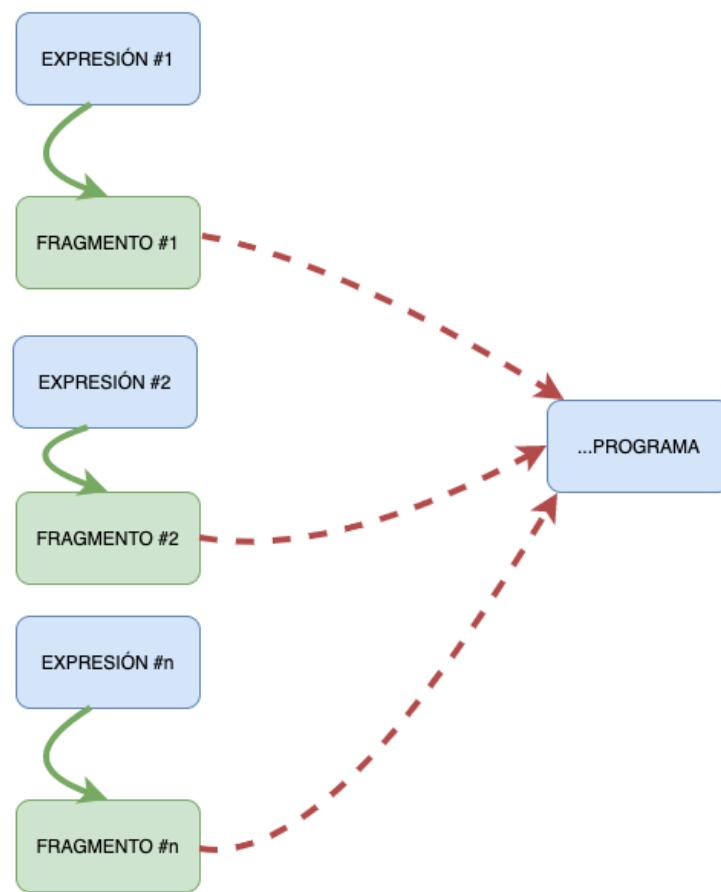
Selección múltiple

Otro tipo de bifurcación o selección es una estructura que decide que fragmento de código va a ser ejecutado de dos o más opciones.



En este caso, la expresión nos lleva a dos posibles bloques de código, pero solo uno de ellos se ejecutará. Esta construcción es muy habitual en segmentos de código del tipo **if-else**. También podemos denominarla **selección múltiple** o **bifurcación compleja**.

Una construcción típica de selección múltiple con más de dos opciones, la encontramos en sentencias del tipo “switch”:



Estos casos complican el análisis ya que si estamos trazando una línea de ejecución nos veremos con tantos subcasos como expresiones “case” tenga el bloque “switch”.

Un ejemplo en código (de nuevo, procedente del código fuente del kernel Linux):

```

switch (iomap->type) {
case IOMAP_HOLE:
/*
 * If the buffer is not up to date or beyond the current EOF,
 * we need to mark it as new to ensure sub-block zeroing is
 * executed if necessary.
 */
if (!buffer_uptodate(bh) ||
(offset >= i_size_read(inode)))
set_buffer_new(bh);
break;
case IOMAP_DELALLOC:
if (!buffer_uptodate(bh) ||
(offset >= i_size_read(inode)))
set_buffer_new(bh);
set_buffer_uptodate(bh);
set_buffer_mapped(bh);
set_buffer_delay(bh);
break;
case IOMAP_UNWRITTEN:
/*
 * For unwritten regions, we always need to ensure that regions
 * in the block we are not writing to are zeroed. Mark the
 * buffer as new to ensure this.
*/
set_buffer_new(bh);
set_buffer_unwritten(bh);
fallthrough;
case IOMAP_MAPPED:
if ((iomap->flags & IOMAP_F_NEW) ||
offset >= i_size_read(inode))
set_buffer_new(bh);
bh->b_blocknr = (iomap->addr + offset - iomap->offset) >>
inode->i_blkbits;
set_buffer_mapped(bh);
break;
}
}

```

Como podemos observar, se obtiene un valor a comparar de forma lógica desde “iomap->type” y se compara sucesivamente en los distintos “case”. Solo se ejecutará aquel fragmento de código cuya comparación coincida con el valor a comparar³⁸...

Siempre que el bloque finalice con una sentencia “break”. En caso contrario, se ejecutarán en secuencia todos los bloques debajo de este. Un efecto a veces deseado, pero en ocasiones fuente de muchos errores de programación.

³⁸ El lector atento habrá visto que un bloque que posee la expresión “fallthrough”. Dicha expresión no es una palabra estándar del lenguaje C sino un atributo no estándar del compilador GCC para que no emita una advertencia de que se está omitiendo una sentencia “break”. Es decir, el programador es consciente de los efectos colaterales de omitir el “break”.

Equivalencia de “switch” e “if-else”

La sentencia “switch” que poseen muchos lenguajes puede ser perfectamente sustituida por combinaciones de “if-else-if” anidadas. Por ejemplo, el lenguaje de programación Python no posee “switch” y la única posibilidad es sustituir dicha estructura por cadenas de “if-elif”.

En el caso anterior, el código vendría a ser algo similar a:

```

1 if (iomap->type == IOMAP_HOLE) {
2 /*
3  * If the buffer is not up to date or beyond the current EOF,
4  * we need to mark it as new to ensure sub-block zeroing is
5  * executed if necessary.
6 */
7 if (!buffer_uptodate(bh) || (offset >= i_size_read(inode)))
8     set_buffer_new(bh);
9
10 } else if (iomap->type == IOMAP_DELALLOC) {
11     if (!buffer_uptodate(bh) || (offset >= i_size_read(inode)))
12         set_buffer_new(bh);
13     set_buffer_uptodate(bh);
14     set_buffer_mapped(bh);
15     set_buffer_delay(bh);
16 } else if (iomap->type == IOMAP_UNWRITTEN) {
17 /*
18  * For unwritten regions, we always need to ensure that regions
19  * in the block we are not writing to are zeroed. Mark the
20  * buffer as new to ensure this.
21 */
22     set_buffer_new(bh);
23     set_buffer_unwritten(bh);
24     fallthrough;
25 } else if (iomap->type == IOMAP_MAPPED) {
26     if ((iomap->flags & IOMAP_F_NEW) || offset >= i_size_read(inode))
27         set_buffer_new(bh);
28     bh->b_blocknr = (iomap->addr + offset - iomap->offset) >> inode->i_blkbits;
29     set_buffer_mapped(bh);
30 }
```

Al final lo que deberemos tener en cuenta es que tenemos una comparación lógica y varias salidas posible o líneas de ejecución. En algunos casos podremos eliminar bifurcaciones y en otros no nos quedará más remedio que analizar las múltiples variantes que se nos presenta.

Afortunadamente también, en muchos casos, las diferentes ejecuciones no representan saltos sino manipulaciones de valores pasados como parámetros u objetos globales con lo que el flujo no variará en la mayoría de los casos.

3.3. ITERACIÓN Y RECURSIÓN

Otra de las estructuras básicas de la programación estructurada es la **repetición**. Esta puede darse de dos formas posibles: **iteración** y **recursión**. La iteración es la repetición de un fragmento de código hasta que se de cierta condición lógica mientras que la recursión, que casi merecería un capítulo aparte, es una función o estructura que puede llamarse o referenciarse a si misma dentro de su propia definición; ocasionando así una estructura repetitiva.

Es fácil detectar la presencia de una función o estructura recursiva. Da igual el lenguaje, la característica común es, como hemos comentado, la presencia de una llamada o referencia a la propia función o estructura. El ejemplo canónico, una función para calcular el factorial de un número:

```
long factorial(long from) {
    return from ? factorial(from - 1) * from : 1;
}
```

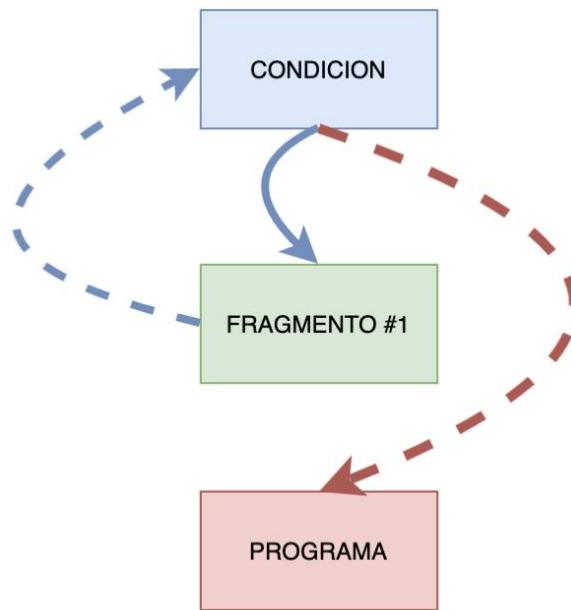
No dedicaremos mucho espacio a las formas recursivas de esta estructura dado que no es común encontrarla en el código. De forma natural, suele emerger en iteraciones cuyo límite no está definido o no es, a priori, determinista (exploración de grafos, etc.) Dado que su codificación es más compleja, la repetición suele presentarse en su forma habitual como iteración.

Iteración: Condiciones de salida y finalización

Salvo en iteraciones infinitas, siempre existe al menos una condición lógica que determina cuando debe la iteración parar o proseguir con el flujo de ejecución. Remarcamos lo de “al menos una” dado que es posible que dentro del mismo bucle de ejecución tengamos condiciones que produzcan la salida de la iteración.

Es muy importante tener en cuenta esta última consecuencia. Cuando analicemos un código debemos interpretar las distintas salidas que posee un fragmento. Dentro de una iteración, salvo las llamadas a funciones con retorno, deberemos estar atentos a las condiciones de ruptura del bucle.

En líneas generales, la estructura básica de la iteración es:



En código, existen varios tipos de implementaciones para expresar iteración, la más característica es el bucle “for”, prácticamente omnipresente en casi todos los lenguajes de programación.

```

for (int i = 1; i <= 10; i++) {
    printf("%i\n", i);
}
  
```

En muchos lenguajes también disponemos de la instrucción “while”:

```

int i = 1;
while (i <= 10) {
    printf("%i\n", i);
    i++;
}
  
```

En particular, en el lenguaje C también disponemos de “do-while” para que al menos se ejecute una vez la iteración por el bucle:

```
int i = 1;
do {
    printf("%i\n", i);
    i++;
} while (i <= 10);
```

Como advertimos al inicio, los bucles pueden poseer puntos de ruptura internos, habitualmente bajo condiciones:

```
int i = 1;
do {
    printf("%i\n", i);
    i++;
    if (i > 10) break;
} while (1);
```

Advirtamos que en la última versión se trata de un bucle infinito, ya que ese “while (1)” mantendrá el bucle iterando de forma continua. No obstante, ya que dentro del mismo poseemos una ruptura gobernada por una expresión lógica, se podrá salir del bucle del mismo modo que si esta hubiese estado fuera.

3.4. ¿POR QUÉ ES ÚTIL ESTA INFORMACIÓN?

Lo dijimos en la primera parte y repetimos aquí:

Análisis, como palabra, significa: distinguir y separar las partes de un todo hasta llegar a conocer sus principios y elementos.

Dentro de los fragmentos de código (bloques o funciones) disponemos de varias estructuras básicas: secuencia, selección y repetición (iteración y recursión) que son estructuras más pequeñas aun de las estudiadas en el capítulo anterior.

El gran problema del análisis es la complejidad que supone que estas estructuras puedan combinarse unas dentro de otras, produciendo distintas líneas de ejecución en dependencia a los valores que reciben de entrada. Equivalente a las llamadas a funciones dentro de funciones...

Por lo tanto, es fundamental que tengamos una idea muy clara de como son estas estructuras para detectarlas, aislarlas y comprender su función y acoplamiento con el resto de la función, módulo o

programa.

En pocas palabras, y como decíamos al principio, el análisis significa identificar y separar las partes para entender como funciona el todo.

3.5. ¿QUÉ TENGO QUE HABER APRENDIDO?

- Detectar la existencia de las diferentes estructuras fundamentales: secuencia, selección y repetición.
- Entender como funcionan estas partículas, los diferentes tipos que existen y los elementos comunes en todas ellas que poseen los lenguajes estructurados y orientados a objetos.

4. LA MEMORIA

Antes de entrar de lleno en el análisis de código (tanto de bajo como de alto nivel) debemos aprender como funciona la memoria cuando un ejecutable se convierte en un proceso. De hecho, un ejecutable no es más que un conjunto de instrucciones con un formato determinado que le indica al sistema operativo como debe formar un proceso y a partir de donde debe comenzar su ejecución.

Para comenzar, debemos recordar que una gran mayoría de las vulnerabilidades atribuidas al software provienen de una mala gestión de la memoria en los programas: desbordamiento de buffer, punteros nulos, punteros colgados, etc.

Conocer como funciona la memoria es fundamental, no solo en el análisis dinámico, sino para comprender simplemente leyendo el código que está ocurriendo. El hecho de que una variable contenga un puntero a un objeto que existe en el montículo no es lo mismo que una dirección de memoria atribuida a un objeto que está ubicado en el segmento de datos.

4.1. LOS ESPACIOS DEL KERNEL Y EL USUARIO

Este es el aspecto de **la memoria tal y como lo ve un proceso** en el sistema operativo Linux. Variará un poco en Windows y por supuesto, el direccionamiento en sistemas de 64 bits, pero básicamente la estructura que veremos será muy similar:



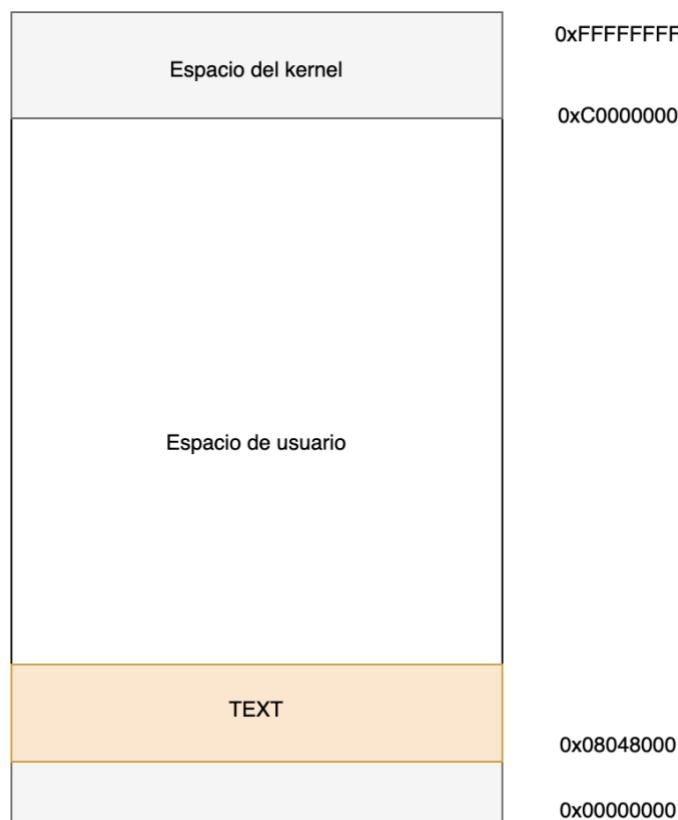
Como vemos, existe una sección desde la dirección más baja, 0x00000000 hasta 0x0804800 a la que el proceso no puede tener acceso, está mapeada por el kernel. Del mismo modo, existe una zona en la parte superior de la memoria, de 0xC0000000 a 0xFFFFFFFF, donde está mapeado el propio kernel (esto no significa que en cada proceso “se copie” el kernel, sino que se mapea “virtualmente” allí).

El resto, es una zona virtual donde residen los distintos segmentos que pertenecen al proceso. Todo ese espacio está “virtualmente” disponible para el proceso. Decimos virtual, porque en realidad la memoria física está limitada, muy limitada, pero el kernel le hace creer al proceso que está completamente disponible para él. Luego, el MMU (memory management unit) hará una traducción de direcciones virtuales a físicas. Lo importante es que desde nuestro punto de vista y del proceso la memoria se ve así, con dicho reparto.

4.2. SEGMENTO DE TEXTO O EL PROGRAMA EN SI MISMO

Si viésemos el código como una especie de partitura que la CPU debe ir interpretando, esta debe localizarse en algún punto ¿no? La CPU posee un registro denominado PC (program counter) que apunta a la instrucción actual que debe procesar y ese PC debe, necesariamente, apuntar a una dirección de memoria que es donde estará nuestra “partitura”.

Ese segmento, es el segmento de texto o código del programa. Es justo el segmento que va primero, a partir del espacio reservado al comienzo por el propio kernel. El segmento no crece, es fijo:



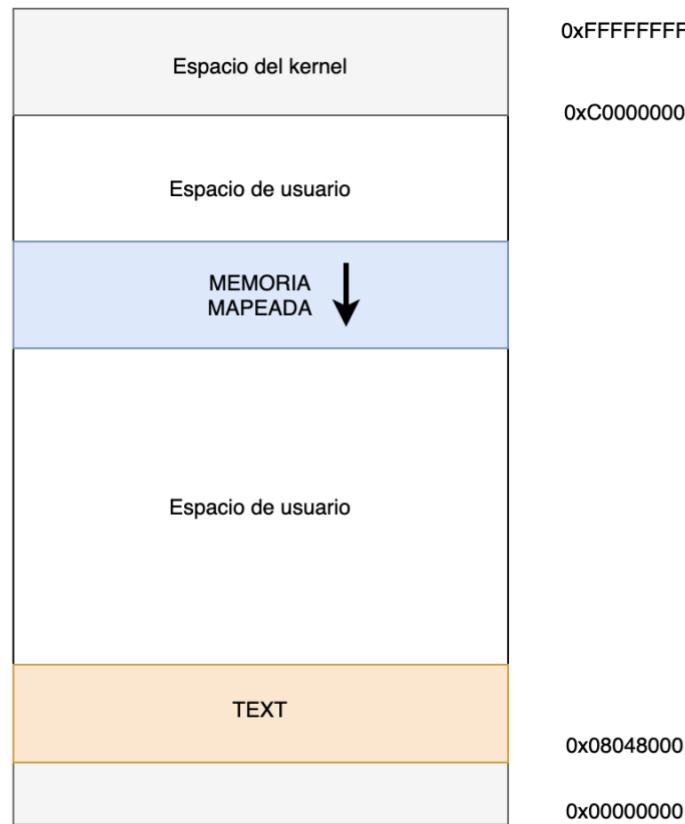
Cuanto estamos desensamblando un programa, ese segmento es el que nos interesa cuando se despliega en forma de proceso. Lo que vemos, el código ensamblador irá a esa zona de la memoria para que la CPU vaya saltando entre el código del programa.

4.3. SEGMENTO DE MEMORIA MAPEADA

Hablando de saltos ¿A dónde va la CPU cuando el propio programa llama a una función externa, que no pertenece al código de éste?

La respuesta es: a una zona de la memoria de usuario que mapea las librerías que el ejecutable enlazó cuando fue compilado. Estas librerías existen en memoria física, pero para que todos los procesos puedan compartirlas solo existe un original y luego se crean copias “virtuales”. Es decir, el proceso cree que las librerías están ahí para él, cuando en realidad están compartidas por todos.

Pero no solo sirve para las librerías. Cualquier archivo que abrimos dentro del proceso o cuando utilizamos la función **mmap**³⁹ directamente, se mapea en esa zona o segmento de memoria:



³⁹ <https://man7.org/linux/man-pages/man2/mmap.2.html>

4.4. SEGMENTO DE DATOS Y SEGMENTOS DE DATOS NO INICIALIZADOS

Un proceso no parte de la nada, posee un grupo de variables al inicio del programa, ya establecidas en el código. Lo vimos anteriormente en el capítulo dedicado a los elementos básicos. Estas son las variables globales y estáticas.

Como sabemos, estas variables pueden estar inicializadas o no a algún valor. Veamos esto con un ejemplo de código que contiene varias de estas variables:

```

1 #include <stdio.h>
2
3 int global_no_inicializada;
4 int global_inicializada = 99;
5
6 void f() {
7     static int estatica_no_inicializada;
8     static int estatica_inicializada = 99;
9
10    printf("Dir: %x Val: %i\n", &estatica_no_inicializada, estatica_no_inicializada);
11    printf("Dir: %x Val: %i\n", &estatica_inicializada, estatica_inicializada);
12 }
13
14 int main()
15 {
16     f();
17     printf("Dir: %x Val: %i\n", &global_no_inicializada, global_no_inicializada);
18     printf("Dir: %x Val: %i\n", &global_inicializada, global_inicializada);
19     return 0;
20 }
21

```

Dir: 600a30 Val: 0
 Dir: 600a20 Val: 99
 Dir: 600a2c Val: 0
 Dir: 600a24 Val: 99

¿Se puede ver como las inicializadas, a pesar de estar declaradas en lugares diferentes, tienen direcciones más cercanas entre ellas?

Es decir, 0x600a30 que es **global** está más cerca de 0x600a2c que es **estática** dentro de la función. Lo mismo podríamos decir con las otras dos.

¿Qué tienen en común?

Lo que las diferencia, realmente, es que unas están inicializadas y otras no, aunque por defecto se les asigna un cero.

Curiosamente, no van asignadas a la misma región. Las variables **inicializadas** van a la región o segmento conocido como “datos” (data segment). Mientras que las variables **no inicializadas** van al segmento denominado BSS (bss segment).

Dado que van por separado, como las no inicializadas no poseen valor, es mejor agruparlas. De este modo, resulta óptimo inicializar a cero el segmento BSS entero.

Por ende, disponemos de dos regiones fijas (no crecen) y próximas que contienen las variables globales y estáticas. Se ven así:



4.5. SEGMENTO DE MONTÍCULO O HEAP

¿Qué ocurre si un programa no puede anticipar en su código cuantas variables y de qué tipo va a necesitar?

Supongamos un problema. Un programa que descarga una lista de palabras de un servidor. No sabemos cuántas líneas posee el archivo, pero debemos almacenarlas en una lista enlazada para posteriormente procesarlas una a una. ¿Cómo hacemos para resolver el problema? ¿Creamos una variable global “grande” para no quedarnos corto? ¿Cuánto es grande? ¿Y si la lista es pequeña, no estaríamos ocupando una zona inmensa de memoria para nada?

Lo ideal sería **reservar memoria de forma dinámica**, según vayamos necesitándola y la liberaremos cuando ya no haga falta. De esa forma optimizaríamos los recursos del sistema, gastando únicamente lo que hiciese falta.

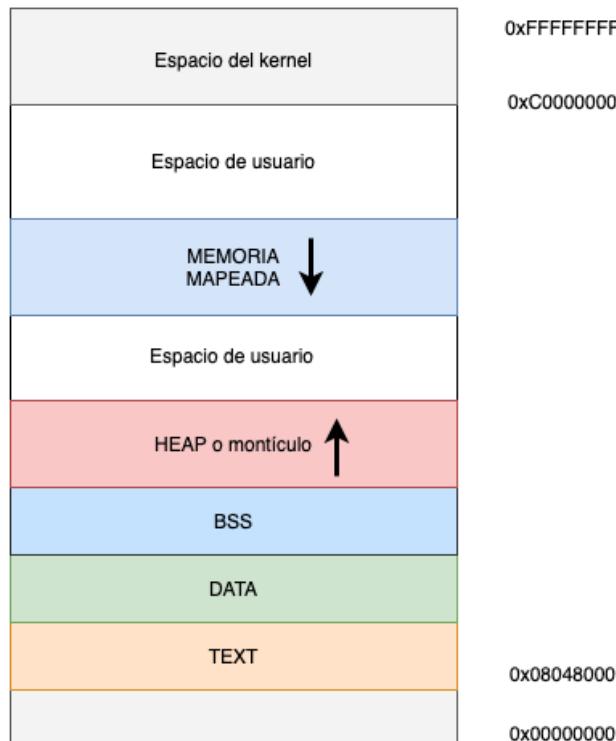
Esto, que denominamos memoria dinámica, nos lo ofrece el montículo o heap, a través de un grupo de funciones que nos permiten reservar y liberar bloques de memoria de ese segmento. Este invento revolucionó la optimización de recursos en la programación y... a la vez creó uno de los mayores quebraderos de cabeza para los programadores y fuente de muchas vulnerabilidades.

En todos los sistemas operativos donde podemos tener la opción de reservar memoria de forma dinámica, el sistema ofrece una interfaz a través de un grupo de funciones y librerías (también es posible tratar el montículo (heap) a través de bibliotecas de terceros).

Tanto en Linux como en Windows está la familia de funciones conocidas como **malloc**⁴⁰ (memory allocation) y, además, en Windows, podemos usar la familiar **HeapAlloc**⁴¹.

El heap o montículo **crece hacia direcciones altas de la memoria**. Pero ojo, no significa que crezca de forma indefinida, no. No crece en función “cada llamada a malloc o reserva que se pide es un crecimiento”. No es así. El montículo es un área que comienza con un tamaño definido y según se va demandando memoria va creciendo, pero, cuando se libera memoria de este segmento vuelve a estar disponible. Se suele decir que el **montículo recicla la memoria**.

Al final, cuando se producen muchas llamadas de reservas y liberaciones de memoria se termina con un efecto denominado **fragmentación del heap** (o montículo). Según sea la estrategia empleada por la librería que administre el montículo, así de buena o mala será la fragmentación. Cuanta más fragmentación y peor gestionada más crecerá el montículo. Imaginad hacer una pared con ladrillos de desigual altura y longitud...



⁴⁰ <https://man7.org/linux/man-pages/man3/malloc.3.html>

⁴¹ <https://docs.microsoft.com/en-us/windows/win32/memory/memory-management-functions#heap-functions>

4.6. SEGMENTO DE ENTORNO O ENVIRONMENT

Un proceso no vive aislado. Debe ejecutarse en un sistema donde convive con cientos e incluso hasta miles de proceso que se ejecutan de forma concurrente e incluso paralela.

Para comunicarnos con un proceso tenemos varios métodos, sockets, pipes, buckets, archivos, etc. Los más primitivos son el paso de parámetros y las variables de entorno.

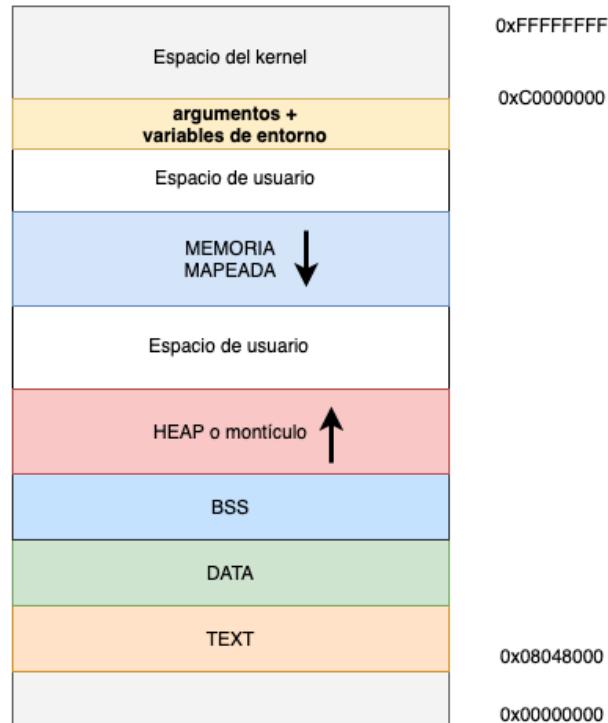
Los parámetros ya los conocemos. A poco que hayamos usado un programa en línea de comandos nos sonará. Las variables de entorno son otra forma de pasar datos a un programa. Se crean en el entorno de ejecución y perviven en él hasta que el proceso muere.

Las variables de entorno son variables clave-valor, como un diccionario. Habitualmente las habréis podido ver en una sesión de terminal. Por ejemplo, en Linux tenemos el comando “env”. Con dicho comando podremos ver qué variables tiene el entorno.

Dado que cada vez que ejecutamos un programa desde la línea de comando se realiza un fork del proceso mismo de la terminal, este hereda todas esas variables de entorno.

¿Dónde van copiadas esas variables de entorno para que el programa pueda usarlas? Por supuesto, en un segmento de memoria reservado para tal uso.

Tanto las variables de entorno como los parámetros que pasamos a cualquier programa poseen una zona en la **memoria fija**, denominada “environ”, allí se encuentra toda esta información:



4.7. SEGMENTO DE LA PILA O STACK EN LA ARQUITECTURA X86 (32 BITS)

La pila es una zona de memoria que se ubica en una zona alta de la memoria, es dinámica y crece hacia direcciones más bajas, en oposición al montículo o heap.

De hecho, cuando alguien se refiere a la “cima de la pila” (en inglés: *top of the stack*) en realidad no se refiere a “la dirección más alta ocupada por la pila” sino todo lo contrario, “la dirección más baja ocupada por la pila”; que además irá cambiando, como veremos, en cada llamada a una función.

Supón que tienes 10 platos y pintas su número en cada uno de ellos. Ahora coges el número uno y lo depositas en la mesa, encima de él depositas el dos, encima del dos el tres, etc.

¿Qué plato queda “en la cima de la pila” de platos? El, 10, claro.

¿Cómo harías si quisieras llegar al número cinco? Pues seguramente quitas el número 10, el número 9, etc, hasta llegar al número 5.

Así funciona la pila. Es una estructura LAST IN, FIRST OUT (LIFO) o último en entrar, primero en salir.

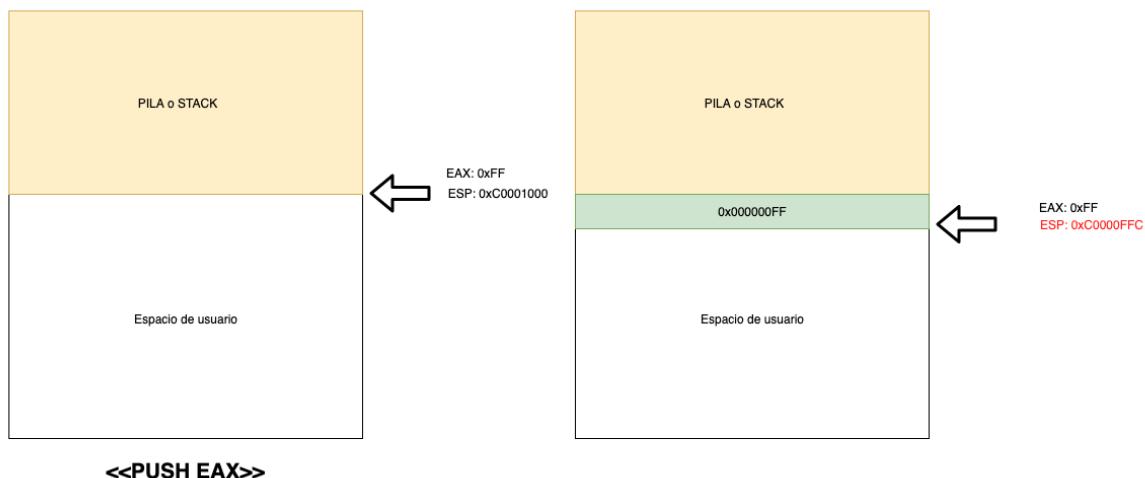
4.7.1. PUSH Y POP, OPERANDO CON LA PILA

En el ensamblador x86 habrás visto muchas veces los mnemotécnicos PUSH-POP. Son los encargados de meter valores en la pila (PUSH) y sacarlos (POP). También tenemos un puntero a la cima de la pila (recordad, la dirección más baja actual de la pila) denominado ESP o RSP en 64 bits.

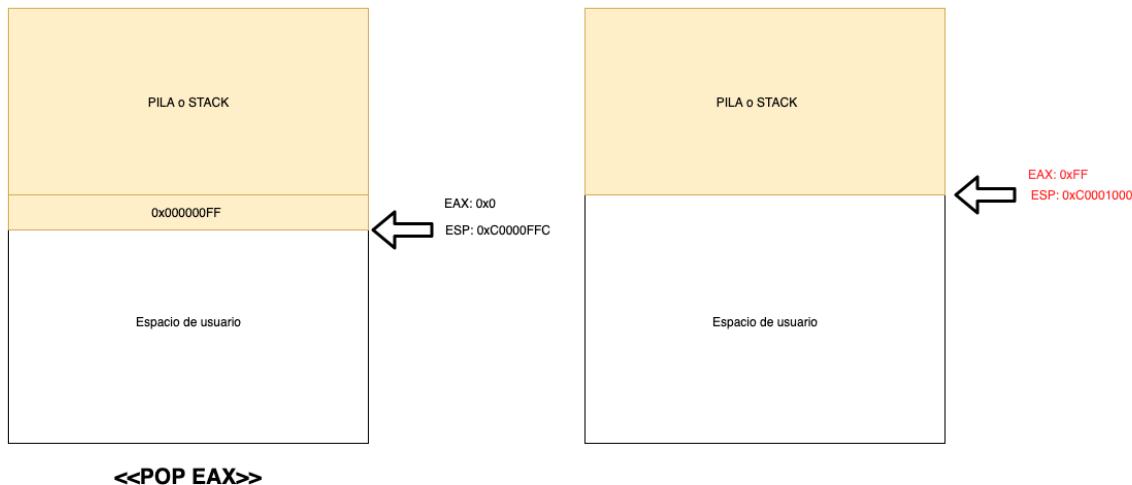
Cada vez que hacemos PUSH, el valor de ESP decremente (haciendo el segmento de pila más grande) hacia una dirección más baja y cuando sacamos valores de la pila con POP, el valor de ESP aumenta (haciendo el segmento de pila más pequeño).

De hecho, como veremos, una de las formas de detectar una función en el código desensamblado es observando donde se producen los PUSHes y POPs.

Observemos como tenemos un valor en EAX y lo “empujamos” a la pila con PUSH:



Ahora supongamos la operación inversa: sacamos un valor de la pila con POP hacia EAX:



Como vemos, el funcionamiento en sí mismo es fácil, tan solo hay que recordar que se “crece hacia abajo” y se “decrece hacia arriba”. Además, que tenemos el registro ESP o RSP apuntando siempre a la cima de la pila y está “cima” es la dirección más baja.

4.7.2. ¿PERO PARA QUÉ SIRVE LA PILA?

Hemos hablado de la pila, donde está, como crece y sus particularidades, pero no hemos hablado para qué exactamente queremos este segmento.

Vamos a descubrir para qué sirve planteando un problema y veremos como llegamos a una conclusión.

Cuando llamamos a una función decíamos que la CPU “salta” hacia donde está esa función, realiza lo que su código indica y vuelve de vuelta con, posiblemente, un valor. ¿No? Veamos un código que ilustre lo que acabamos de decir:

```

1 #include <stdio.h>
2
3 int funcion_llamada(int argumento) {
4     int factor = 5;
5     int y = argumento * factor;
6     return y;
7 }
8
9 int funcion_que_llama_a_funcion(int argumento) {
10    int x = funcion_llamada(argumento);
11    return x;
12 }
13
14 int main()
15 {
16    int z = funcion_que_llama_a_funcion(10);
17    printf("resultado: %i\n", z);
18    return 0;
19 }
20

```

Como vemos es un programa muy simple con tres funciones. “main”, obligatoria por definición y dos más creadas con nombre muy clarificadores: “función_llamada” y “función_que_llama_a_funcion”. Ambas funciones toman un parámetro, contienen variables locales y devuelven un valor.

Vamos a obviar el hecho de que “main” es y una función. Como vemos, dentro de “main” se produce una llamada a “función_que_llama_a_funcion” que a su vez llama, dentro de su cuerpo, a “función_llamada”.

Ahora preguntémonos:

- ¿Dónde almacenamos las variables locales de “función_que_llama_a_funcion” y las de “función_llamada”?
- ¿Cómo sabe la CPU a qué dirección volver cuando ha terminado de ejecutar todo el cuerpo de la función “función_llamada”?
- ¿Cómo le comunica el valor de retorno “función_llamada” a “función_que_llama_a_funcion”?

Pensemos por un momento como solucionaríamos estos tres problemas si solo tuviésemos una estructura de pila y los dos operadores PUSH y POP.

4.7.3. DESENSAMBLADO DEL PROGRAMA DE EJEMPLO

Observemos aquí el desensamblado del código de la imagen de arriba. Nos servirá para ilustrar el funcionamiento de la pila:

```

1  ↵ funcion_llamada(int):
2      push    ebp
3      mov     ebp, esp
4      sub     esp, 16
5      mov     DWORD PTR [ebp-4], 5
6      mov     eax, DWORD PTR [ebp+8]
7      imul   eax, DWORD PTR [ebp-4]
8      mov     DWORD PTR [ebp-8], eax
9      mov     eax, DWORD PTR [ebp-8]
10     leave
11     ret
12 ↵ funcion_que_llama_a_funcion(int):
13     push    ebp
14     mov     ebp, esp
15     sub     esp, 16
16     push    DWORD PTR [ebp+8]
17     call    funcion_llamada(int).
18     add    esp, 4
19     mov     DWORD PTR [ebp-4], eax
20     mov     eax, DWORD PTR [ebp-4]
21     leave
22     ret
23 ↵ .LC0:
24     .string "resultado: %i\n"
25 ↵ main:
26     lea     ecx, [esp+4]
27     and    esp, -16
28     push   DWORD PTR [ecx-4]
29     push   ebp
30     mov    ebp, esp
31     push   ecx
32     sub    esp, 20
33     push   10
34     call   funcion_que_llama_a_funcion(int).
35     add    esp, 4
36     mov    DWORD PTR [ebp-12], eax
37     sub    esp, 8
38     push   DWORD PTR [ebp-12]
39     push   OFFSET FLAT:.LC0
40     call   printf
41     add    esp, 16
42     mov    eax, 0
43     mov    ecx, DWORD PTR [ebp-4]
44     leave
45     lea    esp, [ecx-4]
46     ret

```

4.7.4. CONVENCIONES DE LLAMADA

Aunque no hayamos resuelto el problema planteado en anteriormente, es probable que intuyamos la respuesta o al menos le damos algo de forma. Vamos a ver como las funciones usan la pila para gestionar sus variables, argumentos y valores de retorno.

En primer lugar, entre funciones siempre hay una función que llama y otra que es llamada. Aunque parezca una obviedad, en inglés se usan los términos “caller” y “callee” respectivamente y son ampliamente usados cuando en el mundo del reversing se discute sobre **convenciones de llamada**⁴².

Una convención de llamada es un protocolo de como usar la pila. Básicamente:

- En que orden se introducen los parámetros en la pila.
- Que registros se guardan y quien ha de guardarlos.
- Quién (o caller o callee) se encarga de “limpiar” la pila.

Una vez se acuerdan (hay varios métodos⁴³) estas tres condiciones, se produce lo que denominamos convención de llamada; y es que es la que usará el compilador para generar código acorde a esta.

Nosotros veremos la **cdecl**, por ser la mas usada en el mundo UNIX y recomendada por el estándar del lenguaje C. En realidad, con ver una, tan solo hay que tener en cuenta los tres aspectos antes comentados.

En cdecl, los argumentos de la función se pasan de derecha a izquierda, el caller guarda los registros EAX, ECX y EDX, el resto de los registros de la CPU los guarda el callee. El valor de retorno de la función es puesto por el callee en el registro EAX. Además, es el caller el que se responsabiliza de limpiar la pila una vez se ha retornado de la función llamada.

4.7.5. MARCO DE PILA

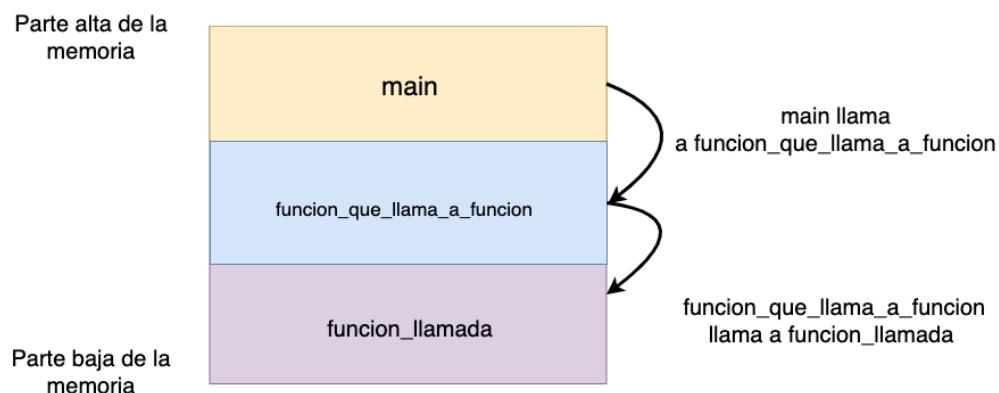
¿Recordáis los platos? Pues cada función tiene una especie de plato donde deja todo lo que le pertenece. Sus argumentos, variables locales, etc. Es decir, toda la información que necesita una función para ejecutarse incluyendo la dirección de vuelta para regresar al lugar desde donde fue llamada.

Así, la pila se compone de muchos fragmentos (platos) y cada fragmento pertenece a una función. Estos fragmentos se denominan “marcos” (frameworks). No hay nada más en la pila que no sea marcos que pertenecen cada uno a una función. Además, la pila funciona como tal (recordad LAST IN, FIRST OUT).

¿Qué forma tendrá esos marcos en el caso anterior? Pues como se amontonan (stack) según se van llamando funciones de forma encadenada, la pila presentaría el siguiente aspecto:

⁴² <https://devblogs.microsoft.com/oldnewthing/20040102-00/?p=41213>

⁴³ https://en.wikibooks.org/wiki/X86_Disassembly/Calling_Conventions

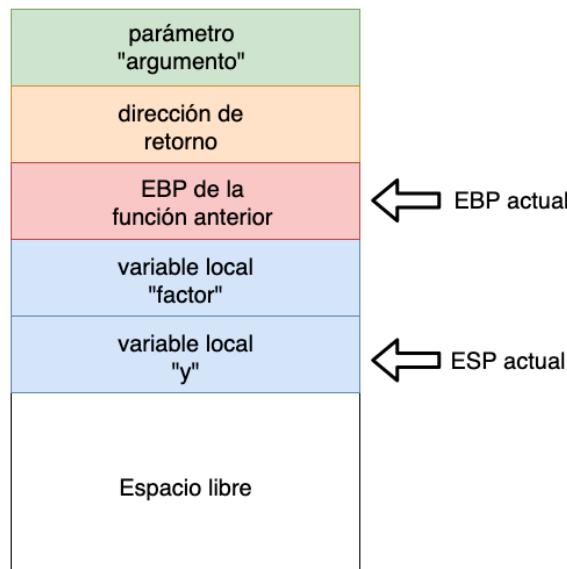


Recordemos: la pila empieza en una dirección de memoria alta y “crece” hacia abajo, como una estalactita; del techo al suelo.

Es decir, estamos amontonando en la pila los distintos marcos que pertenecen cada uno a una función. Según se van encadenando funciones, se van amontonando marcos de pila; es decir, estamos apilando marcos nunca mejor dicho.

Y, a la inversa, cada vez que retornamos de una función se van quitando fragmentos (recordad cuando “quitamos” platos para “volver” al plato número 5)

Ahora, vamos a coger uno de esos fragmentos, en concreto el de la función “función_llamada”, lo abriremos y veremos que contiene:



Son muchas cosas, empezamos por el principio. Ya sabemos que el registro de la CPU denominado Stack Pointer (ESP en 32 bits y RSP en 64 bits) tiene el valor de la dirección de memoria donde está la cima de la pila (recordad, la dirección más baja, puesto que la pila crece “hacia abajo”) ¿Pero como recordamos donde empiezan o acaban todos los marcos de pila que tenemos? Primero veremos algo

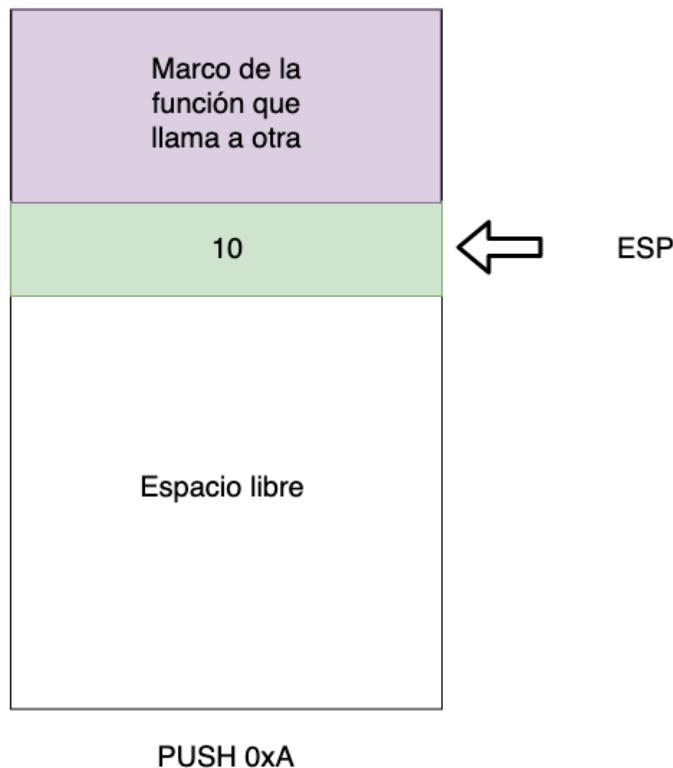
fácil: pasarle parámetros a la función que vamos a llamar desde “main”: “función_que_llama_a_funcion”.

4.7.6. INTRODUCIENDO PARÁMETROS PARA LA FUNCIÓN A LLAMAR

Como vamos a llamar a una función que necesita parámetros (según su signatura) para hacer su trabajo necesitamos una vía para comunicarle a la función donde recogerlos.

La función anterior “función_que_llama_a_funcion” solo necesita de un parámetro, un entero. Así que tan solo tenemos que “comunicarle” un valor.

¿Cómo realizamos esa comunicación? Empleamos la pila. Hacemos **PUSH <valor>** y metemos en la pila el valor que deseamos pasar a la función que estamos a punto de llamar. Si vemos el listado en ensamblador se trata de la **Línea 33**. La pila queda así:



Es simple y sencillo, antes de llamar a la función vamos introduciendo sus parámetros desde el final de la signatura hasta el comienzo.

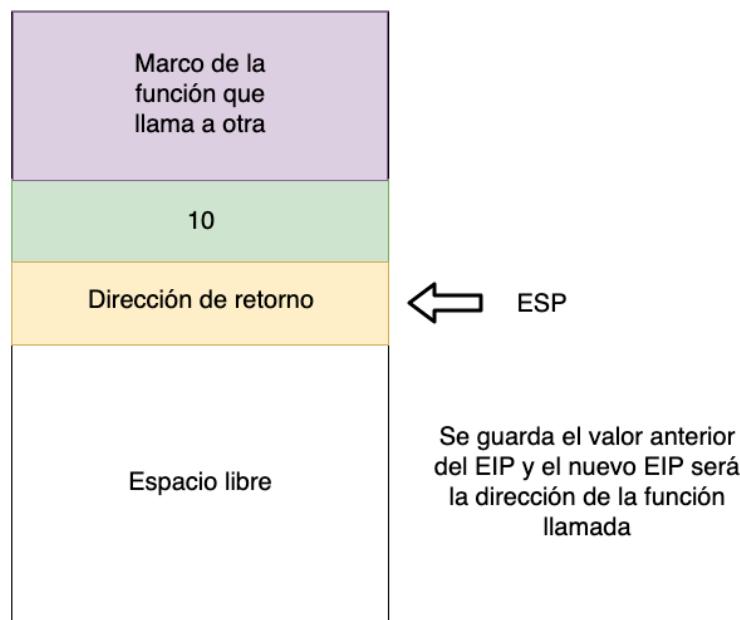
4.7.7. LLAMANDO A LA FUNCIÓN

Lo siguiente también es simple, una llamada a una función se representa por el mnemotécnico **CALL <nombre_funcion>** y podemos verlo en acción en la **Línea 34** del listado.

Aunque la acción sea simple, cuando llamamos a una función la acción de CALL provoca que automáticamente la dirección actual en ejecución, la que está en el registro EIP (Instruction Pointer o RIP en 64bits) se guarde en la pila.

Hacer **CALL función_que_llama_a_funcion** es equivalente a hacer lo siguiente:

PUSH EIP	; Guarda EIP en la pila
MOV EIP, [función_llamada]	; Guarda la dirección de “función_llamada” en el registro EIP



CALL función_que_llama_a_funcion

Esto, representa un salto en el programa. Supongamos que la función “función_que_llama_a_funcion” se encuentra en la dirección (del segmento de texto) 0x08483000 y nuestro EIP, el que acaba de guardarse en la pila, estaba en 0x08482560; que correspondía al código de “main”.

Lo que ocurre ahora es que el EIP toma el valor donde comienza la función llamada, es decir, 0x08483000 y comienza a ejecutar su código. Como vemos, en la pila quedan guardados los parámetros con los que se va a llamar a la función y la dirección antigua, así que cuando se regrese de esa función, sabremos donde continuar ejecutando el código.

Proseguimos, pero ahora estamos en la línea donde comienza “función_que_llama_a_funcion”, es decir, en la 13 del código desensamblado que hay más arriba.

4.7.8. STACK BASE POINTER, RECORDANDO EL MARCO DE PILA

Ahora bien ¿Cómo sabemos donde termina un marco de pila y donde empieza el siguiente? No podemos tener un registro de la CPU por cada marco, eso es evidente. De nuevo: lo guardamos en la

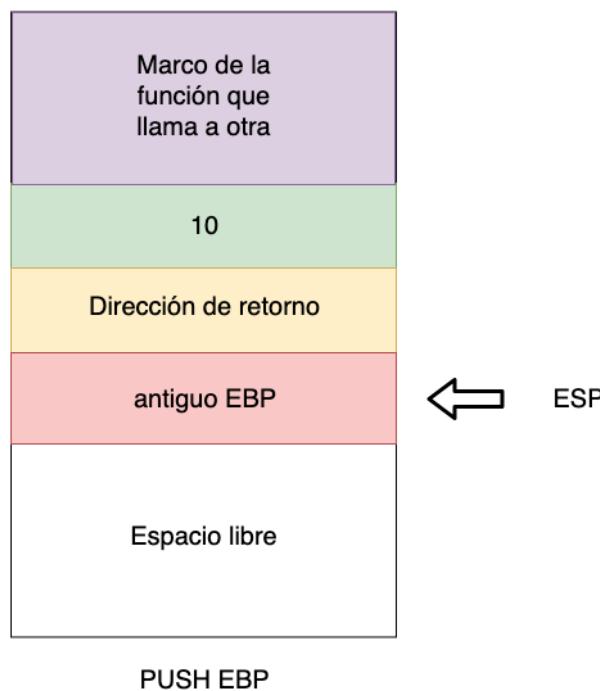
pila, como parte de la propia información del marco y tan solo mantenemos un registro de donde está ubicado el marco “activo”; es decir, el marco de pila de la función que actualmente se está ejecutando.

A ese registro lo llamamos EBP o RBP (32 y 64 bits, respectivamente). Y contendrá la dirección donde se ubica el marco de pila de la función en ejecución. BP significa Base Pointer y es, como su nombre indica “la base del marco actual de la pila”.

A partir del valor de este registro se indexan los argumentos de la función o las variables locales de esta. Es decir, todo lo que vemos como **EBP + <valor>**, es parámetro y todo lo que vemos con **EBP - <valor>** es variable local.

Así, lo primero que hace una función cuando llama a otra es guardar el EBP anterior haciendo “**PUSH EBP**”, para que cuando retorne la función llamadora se recupere su EBP y no perdamos la cadena de marcos entre llamada y llamada.

La pila se vería entonces de este modo cuando se hace **PUSH EBP** para guardarlo (lo vemos en la línea 13):



Ya tenemos guardado el EBP antiguo para recuperarlo más tarde. Es entonces, cuando creamos el marco de pila para la función que vamos a llamar y esto lo hacemos moviendo al EBP, el registro base de la pila, el valor actual del ESP (donde está la cima): **MOV EBP, ESP**

Esto viene a decir: “Como ya he guardado el EBP anterior, ahora uso el registro ESP para crear el marco actual, mi marco”. De este modo, se crea el marco para la función en curso.

Es importante notar como el marco gira entorno al EBP y este se crea a partir del ESP tal y como está cuando se guarda el anterior EBP. En secuencia:

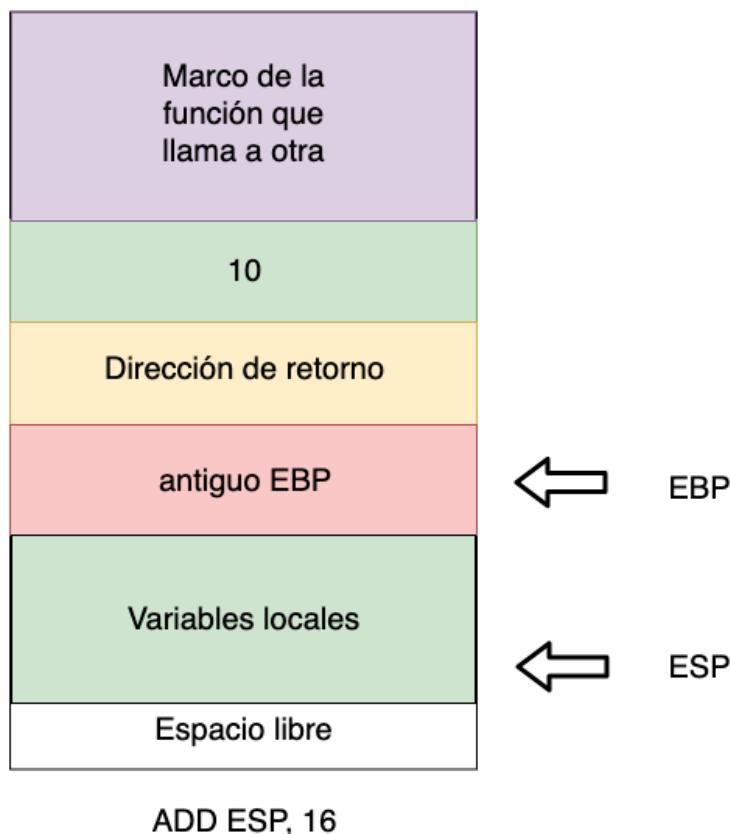
- Guardo el EBP antiguo en la pila (**PUSH EBP; línea 13**)
- El ESP aumenta y me quedo con ese valor para crear mi marco (**MOV EBP, ESP; línea 14**)

4.7.9. CREANDO ESPACIO PARA LAS VARIABLES LOCALES

Ahora, la función en curso necesita espacio para sus variables. Ella es responsable de acomodarlas en la pila. ¿Cómo generamos espacio en la pila? ¿Cómo la hacemos crecer? **Restando la cantidad de espacio necesario al registro ESP** (ver línea 15). Es decir, haciendo crecer la pila hacia abajo (recordad la estalactita)

Atención, porque no siempre equivale el espacio reservado para variables locales con el número y tipo de variables locales que posee la función. Una cosa es el espacio que calculamos y otra la que el compilador, que puede o no optimizar la función, necesita para la función.

En esta ocasión reserva 16 bytes y lo hace sumando a la cima de la pila, el ESP, 16 o 0xF en hexadecimal:



Como vemos en el código fuente original, la función “función_que_llama_a_funcion” usa una sola variable local que contiene el valor retorno de la función “función_llamada”.

Para no repetirnos, observemos en el código como para inicializar esa variable global que estará en

EBP-4, hace PUSH de **[EBP+8]**, que es donde se encuentra...**10**...el valor que precisamente es parámetro de “función_que_llama_a_funcion”.

Llama a “función_llamada” y esta le devuelve en el registro **EAX** el valor de retorno de la función. Observad en el código como “función_llamada” hace referencia a su parámetro **[EBP+8]** y sus variables locales: **[EBP-4] == factor** y **[EBP-8] == y**.

Si leemos el ensamblador de “función_llamada” parece que dice: “Inicio una variable local en [EBP-4] con el valor ‘5’. Muevo a EAX el parámetro que me han pasado en [EBP+8]. Multiplico lo que haya en EAX por mi variable local [EBP-4]. Inicializo mi variable local en [EBP-8] con el valor que halla en EAX y dejo en EAX el resultado de la operación, que precisamente está en mi variable local [EBP-8]**”.

(* Si un alumno o alumna se da cuenta de qué se puede hacer mejor en ese fragmento en ensamblador, se ganará una décima)

4.7.10. EL RETORNO

Cuando se vuelve de “función_llamada”, esta nos deja el valor de retorno en EAX para que “función_que_llama_a_funcion” lo use para inicializar su variable global en [EBP-4], es decir, su variable ‘x’ en el código.

Pero ¿Qué pasa con EBP, no hay una instrucción para restablecerlo? ¿Y con el valor de retorno?

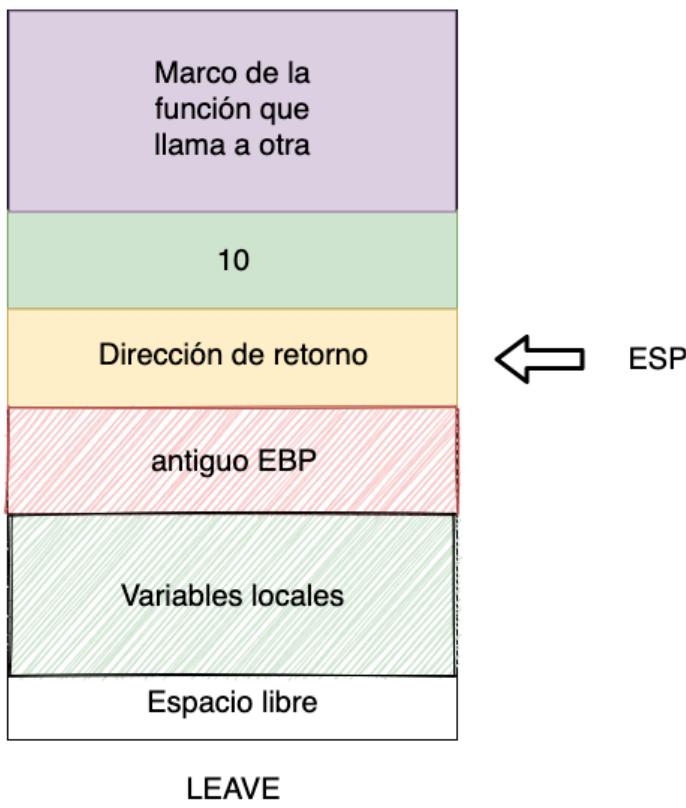
No estamos viendo el resto de las operaciones porque existen dos mnemotécnicos para ello: **LEAVE** y **RET**.

Cuando cualquier de las funciones alcanza un LEAVE la CPU hace exactamente el equivalente a esto:

MOV ESP, EBP
POP EBP

Es decir, “libera” el espacio que antes había reclamado la función para si misma “subiendo” el **ESP** hacia la dirección de la pila donde apunta el **EBP**. Y ¿Qué se guarda allí? El antiguo puntero al EBP de la función anterior.

El POP EBP movería el valor de la pila donde se encuentre el ESP a EBP y añade +4 al ESP, descartando ya todo lo que haya debajo de “Dirección de retorno”. Es decir, el LEAVE nos deja así:



Posteriormente, se ejecuta el **RET**. No es difícil adivinar que se trata de “retornar” (return) y ¿dónde está apuntando el **ESP** ahora mismo? Exacto, a la dirección de la función que llamó a la que estamos a punto de abandonar.

¿Estamos preparados?

- Hemos descartado el espacio de pila usado para acomodar las variables locales.
- Hemos restaurado el antiguo **EBP**, para que cuando volvamos la función que nos llamó tenga localizados sus parámetros y variables locales.
- Hemos avanzado el **ESP** hasta apuntar a la dirección de salto

Pues ya solo queda volver. Hacemos **RET** y el equivalente es un hipotético:

POP EIP

Es decir, sacamos de la pila y metemos en **EIP** el valor que haya guardado en **[ESP]** ¿Dónde decíamos que apuntaba **ESP**? Listo, volvemos al lugar donde justo se estaba ejecutando la función “función_que_llama_a_funcion” y después del **RET** de esta, volveremos al “main”, justo antes de salir del programa.

4.7.11. LA PILA EN LA ARQUITECTURA AMD64 (64 BITS)

Hasta ahora, hemos visto el funcionamiento clásico de la pila en un sistema de 32 bits y arquitectura

Intel o x86. Pero con el avance de la tecnología ha hecho que esta arquitectura esté en prácticamente desuso. Es decir, la gran mayoría (por no decir todos) los procesadores a fecha de la redacción de este documento son de 64 bits.

El diseño de estos trajo también una ampliación de los registros disponibles. Si en 32 bits tenemos los clásicos EAX, EBX, ECX, EDX, etc. La arquitectura 64 bits amplía estos a ocho más y además cambia algunas nomenclaturas (y, por supuesto, son registros de 64 bits/4 bytes en vez de 32 bits/4 bytes):

32 bits	64 bits
EAX	RAX
EBX	RBX
ECX	RCX
EDX	RDX
EBP	RBP
ESP	RSP
EDI	RDI
ESI	ESI
-	R8
-	R9
-	R10
-	R11
-	R12
-	R13
-	R14
-	R15

Con esta ampliación (más bien, duplicación) de registros, se determinó que los parámetros de las funciones podrían utilizar los registros y evitar el uso de la pila. Al fin y al cabo, operar en todos los niveles con un registro es más óptimo que tocar memoria.

Por lo tanto, en la arquitectura de 64 bits, aunque posible, no se utiliza habitualmente la pila para pasar los parámetros de las funciones, sino los registros disponibles.

En concreto, es posible emplear solo registros si una función tiene seis parámetros. Efectivamente, si la función tuviera más de seis parámetros entonces sí se emplearía la pila.

Por orden, de izquierda a derecha si miramos la firma de una función, los registros a emplear son estos:

RDI, RSI, RDX, RCX, R8, R9

Recordemos. Más allá del sexto parámetro no se emplearían más registros, sino la pila; como hacíamos en los sistemas de 32 bits.

Observemos el detalle del ensamblador con un pequeño ejemplo:

```

1  [[gnu::noinline]] int add(int a, int b)
2  {
3      int c, d;
4      c = 11;
5      d = 12;
6      return a + b + c + d;
7  }
8
9  int foo()
10 {
11     return add(11, 42);
12 }
13

```

```

1  add(int, int):
2      push    rbp
3      mov     rbp, rsp
4      mov     DWORD PTR [rbp-20], edi
5      mov     DWORD PTR [rbp-24], esi
6      mov     DWORD PTR [rbp-4], 11
7      mov     DWORD PTR [rbp-8], 12
8      mov     edx, DWORD PTR [rbp-20]
9      mov     eax, DWORD PTR [rbp-24]
10     add    edx, eax
11     mov     eax, DWORD PTR [rbp-4]
12     add    edx, eax
13     mov     eax, DWORD PTR [rbp-8]
14     add    eax, edx
15     pop    rbp
16     ret
17 foo():
18     push    rbp
19     mov     rbp, rsp
20     mov     esi, 42
21     mov     edi, 11
22     call   add(int, _int)
23     pop    rbp
24     ret

```

Si observamos a la derecha:

MOV esi, 42
MOV edi, 11

No se emplea en ningún momento la pila para “empujar” los parámetros necesarios para la función.

Existe una salvedad importante para sistemas operativos **Microsoft Windows**:

Solo es posible utilizar los registros para cuatro parámetros. Son los registros: **RCX, RDX, R8 y R9**. Es decir, no se emplean los registros: RSI, RDI.

4.7.12. ¿ES POSIBLE EVITAR EL REGISTRO EBP PARA CREAR MARCOS DE PILA?

Correcto. En los procesadores modernos, y dependiendo del contexto y compilador, es posible realizar un seguimiento de los marcos de pila sin que sea necesario el empleo del registro RBP y su almacenamiento y recuperación a través de la pila.

Un ejemplo basado en el código del anterior apartado utilizando la opción ‘-fomit-frame-pointer’ del compilador GCC:

```

1  [[gnu::noinline]] int add(int a, int b)
2  {
3      int c, d;
4      c = 11;
5      d = 12;
6      return a + b + c + d;
7  }
8
9  int foo()
10 {
11     return add(11, 42);
12 }
13

```

```

1  add(int, int):
2      sub    esp, 16
3      mov    DWORD PTR [esp+12], 11
4      mov    DWORD PTR [esp+8], 12
5      mov    edx, DWORD PTR [esp+20]
6      mov    eax, DWORD PTR [esp+24]
7      add    edx, eax
8      mov    eax, DWORD PTR [esp+12]
9      add    edx, eax
10     mov   eax, DWORD PTR [esp+8]
11     add   eax, edx
12     add   esp, 16
13     ret
14 foo():
15     push   42
16     push   11
17     call   add(int, int)
18     add    esp, 8
19     ret

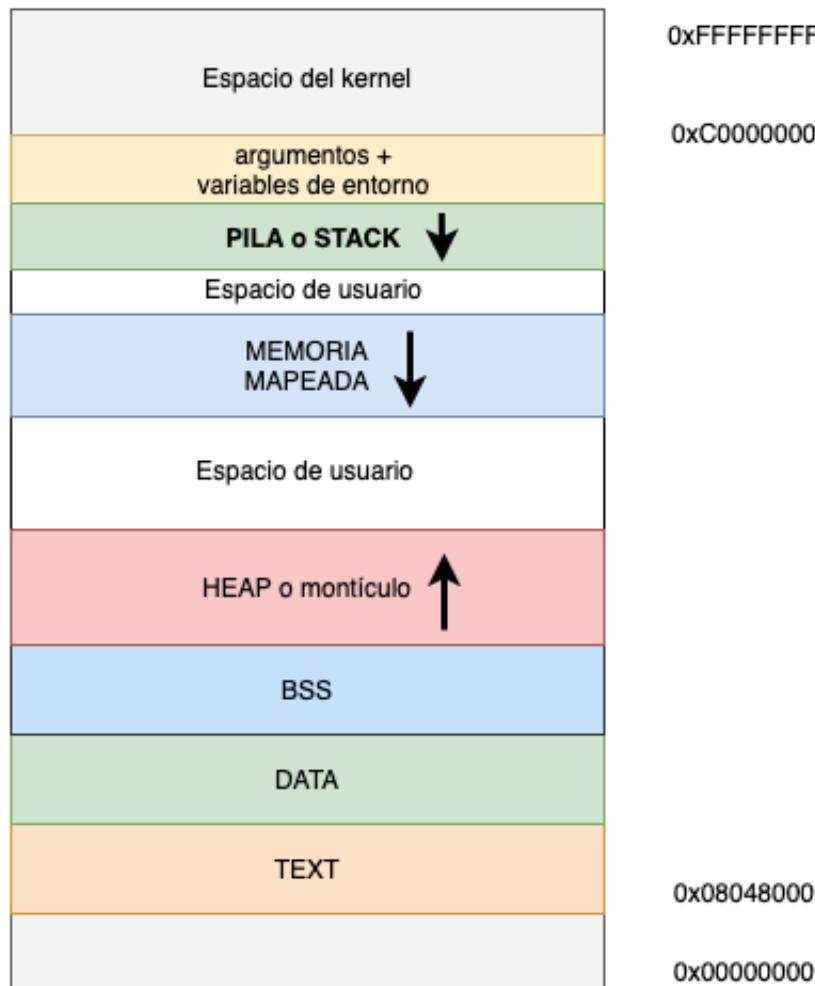
```

Como vemos, en ningún momento se emplea EBP, ni sus operaciones asociadas y sin embargo todo sigue funcionando con normalidad.

Esto, es necesario normalizarlo para no llevarnos sorpresas cuando realicemos el desensamblado de un código y las operaciones de pila contemplen esta posibilidad.

4.8. LA MEMORIA, FOTO FINAL

Finalmente, volvemos a la vista del segmento de pila con todas las piezas del puzzle:



4.9. CÓDIGO INDEPENDIENTE DE LA POSICIÓN Y MEDIDAS ANTIEXPLOTAÇÃO

Tal y como hemos presentado el mosaico de segmentos de la memoria, parece que existe un orden que nunca se altera, pero esto no siempre es así. Como defensa contra los exploits, se ideo un sistema de enlace en la que la posición de los segmentos podía variar de direcciones.

Esto hace que los exploits que necesitan de referencias fijas sobre las que apoyarse, no puedan asegurar la posición al estar determinados segmentos en direcciones distintas.

Dado que los procesos corren en un espacio de memoria virtual, no es necesario que las direcciones de los segmentos sean las mismas siempre, de su traducción a memoria real ya se encarga la MMU. Esto puede aprovecharse para cargar ciertos segmentos en posiciones aproximadamente aleatorias, lo suficiente para hacer la explotación más compleja dado que se necesita información sobre la posición en memoria de los segmentos.

De hecho, la base para evadir la protección que se encarga de esta reposición de los segmentos, ASLR (Address Space Layout Randomization) es la necesidad de una fuga de información de memoria que permita tomar una referencia y a partir de ahí, ir pivotando.

Por ello, aunque veamos una foto fija, debemos pensar que los segmentos, realmente no se encuentran en el orden que observamos. No obstante, no todos los segmentos son reposicionados en direcciones aleatorias. En un inicio, fue la pila, posteriormente se añadió aleatoriedad al propio texto del programa (text), montículo (heap) y las librerías enlazadas.

https://es.wikipedia.org/wiki/Aleatoriedad_en_la_disposición_del_espacio_de_direcciones

4.10. ¿POR QUÉ ES ÚTIL ESTA INFORMACIÓN?

El funcionamiento de la memoria es tan vital y obvio en la ingeniería inversa que probablemente esta sección sobra. Sin embargo, sería lícito preguntar porque en un módulo de análisis de código posee su propio capítulo.

La respuesta es que entendiendo como funciona la memoria nos hace ver con muchísima claridad como funcionan los programas y que impacto tiene, por ejemplo, liberar dos veces memoria del heap o observar una copia de un buffer en la pila que sabemos podemos machacar con un parámetro.

4.11. ¿QUÉ TENGO QUE HABER APRENDIDO?

- Entender como se estructura la memoria y como se organiza. Cuales son las secciones fijas y cuales dinámicas (cambiantes a lo largo del programa).
- Comprender porque existe la sección de datos inicializados, no inicializados y donde se ubica el código (texto) de un programa.
- Determinar cuales son las secciones dinámicas y cual es su funcionamiento y papel. Por qué el montículo o heap se fragmenta, por qué la pila avanza y se contrae y por qué existe una zona de mapeo donde se pueden referenciar las librerías externas.

5. PUNTEROS

Vamos a acercarnos a los punteros. Mecanismo imprescindible en la programación, tanto en su uso director por parte del programador como internamente en numerosos (si no todos) los lenguajes de programación.

No vamos a negarlo. Son difíciles de aprender y complejos de dominar. Tanto, que son la raíz de miles y miles de vulnerabilidades que se derivan de un incorrecto uso por parte del programador.

Por esto último, es necesario entender su funcionamiento para que al menos, podamos leer código y comprender que se está haciendo con ellos y donde puede estar escondiéndose una vulnerabilidad latente.

En el ‘Bug hunting’ en caso de que tengamos oportunidad de leer el código libremente, su entendimiento se hace esencial. También nos ayudará a comprender el manejo de la memoria y los tropiezos derivados de su mal uso.

5.1. ¿QUÉ SON?

Los punteros son un concepto de programación tremadamente útil y potente. Todos los lenguajes de programación hacen uso de ellos, aunque (afortunadamente quizás) muchos de ellos los emplean de forma transparente (el programador se abstrae de su uso) siendo ocultados bajo mecanismos de más alto nivel.

En lenguajes con gestión de memoria automática los punteros son prácticamente invisibles, sin embargo, están ahí. Un ejemplo evidente y claro es Java. Este lenguaje popularizó el concepto de máquina virtual y recolectores de basura (ojo, decimos “popularizó”, porque los conceptos y sus aplicaciones poseen larga data). No vemos los punteros por ninguna parte, sin embargo...

<https://docs.oracle.com/javase/7/docs/api/java/lang/NullPointerException.html>

Existe una excepción que se denomina “NullPointerException”.

Y es que, aunque no tengamos que vernos con los punteros en dicho lenguaje, este, internamente los emplea denominándolos conceptualmente, **referencias**. Si una variable en Java apunta a ‘null’ e intentamos usarla nos las veremos con esta excepción, precisamente porque dicha variable no estará apuntando a nada o apunta a ‘null’ que básicamente es la nada.

Fijemos en nuestra cabeza lo que acabamos de decir con “apunta a”. Esa expresión ya deja claro para que sirve un “puntero”. Precisamente, para eso, para apuntar a.

5.2. ¿PARA QUÉ SIRVEN?

¿Para qué sirve una variable? vamos a poner un ejemplo sencillo, una variable que apunta a un

entero:

```
int valor = 100;
```

¿Qué significa eso? Es fácil.

Cuando compilemos el programa, el compilador creará una etiqueta en la que escribirá “valor”, le asignará una posición de memoria y en esa dirección pondrá un entero que será el número 100.

Cuando el compilador lee esa línea estará diciendo. “Vale, quieres que almacene en la memoria el entero 100, y que cuando digas ‘valor’ te estarás refiriendo a ese lugar.”

Posteriormente, cuando aparece ‘valor’ en el programa lo hará principalmente de dos formas:

```
int otro_valor = valor;
valor = 200;
```

En la primera línea, el programa va a la dirección de ‘valor’, “lee” el entero que hay allí (que será 100) y asigna idéntica cantidad a la variable ‘otro_valor’. El 100 de ‘valor’ es de ‘valor’ y el 100 de ‘otro_valor’ será otro 100, **no lo comparten, se copia**.

En la segunda línea, va a la dirección de ‘valor’ y “escribe” el entero 200 sin preocuparse por lo que había en dicha dirección. Es decir, en la misma dirección pone 200 y sustituye el valor anterior, 100.

Funciones que no pueden cambiar variables

Supongamos una función que toma un entero como parámetro y cambia su valor a 200. Es decir, lo que hemos visto antes, pero aislado en forma de función para que pueda ser reutilizado en cualquier parte del programa.

```
void asigna200 (int valor)
{
    valor = 200;
}
```

La función es clara, ¿verdad?

Lo esperable sería que ocurriese esto:

```
int valor = 100;
puts(valor);           // muestra por consola 100
asigna200(valor);
puts(valor);
```

¿Qué creeis que aparecerá por la consola? ¿100 o 200?

100, por supuesto. Nada ha cambiado porque la función ‘asigna200’ se ha comportado de la siguiente manera:

```
asigna200 ( <dirección del primer parámetro> )
{
    <dirección del primer parámetro> = 200;
}
```

Cuando llamamos a ‘asigna200’ y le pasamos el parámetro 100, lo único que estamos haciendo es inicializar la dirección del primer parámetro con dicho valor. Si posteriormente asignamos a ese mismo nombre, ‘valor’, la cantidad de 200, estamos cambiando el parámetro y no la variable que le hemos pasado.

Introduzcamos los punteros

Entonces ¿Cómo podemos hacer para asignar un nuevo valor a una variable externa? ¿Cómo podemos mutarla?

Claro, una forma de hacerlo sería así:

```
int devuelve_valor()
{
    return 200;
}
valor = 100;
puts(valor);
valor = devuelve_valor();
puts(valor);
```

Ahí tenemos garantizado que estamos cambiando a ‘valor’. Pero no lo estamos haciendo dentro de la función, sino reciclando un valor de retorno. Nada cambia en realidad.

5.3. ¿CÓMO FUNCIONAN?

¿Y si en vez de la cantidad que guarda ‘valor’ le **pasamos su dirección**?

Todas las variables, objetos, estructuras tienen una dirección en memoria, eso lo tenemos claro. Han de residir en algún lugar, de lo contrario ¿Dónde almacenaríamos sus valores?

C, C++ y otros lenguajes poseen operadores para indicar que **no se quiere recuperar el valor asignado sino su dirección en memoria**.

En C, por ejemplo, es el operador *ampersand*:

```

1 #include <stdio.h>
2
3
4 int main()
5 {
6     int valor = 100;
7     printf("%p\n", &valor);
8 }
9
10

```

0x7ffcdd3a01ec

Eso imprimirá la dirección de ‘valor’ en vez del entero que tenga depositado en su dirección de memoria.

Es decir, en C, una variable como ‘valor’ tiene dos formas de poder obtener información sobre ella. El entero que almacena y la dirección de memoria donde se encuentra.

Intentemos guardar la dirección de memoria en otra variable:

```

1 #include <stdio.h>
2
3
4 int main()
5 {
6     int valor = 100;
7     printf("%p\n", &valor);
8
9     int direccion_valor = &valor;
10
11

```

main.cpp: In function 'int main()':
main.cpp:9:27: error: invalid conversion from 'int*' to 'int' [-fpermissive]
9 | int direccion_valor = &valor;
| ^~~~~~
| |
| int*

Error, por supuesto. No podemos almacenar (de forma directa) la dirección de ‘valor’ en otra variable declarada como entero.

La forma correcta de almacenar direcciones es con una sintaxis especial, **utilizando el asterisco después del tipo de variable declarado**, que le indicará al compilador que lo que realmente queremos es almacenar la dirección de una variable:

```
int *dirección_valor = &valor;
```

Vamos a ver que ocurre:

```

1 #include <stdio.h>
2
3
4 int main()
5 {
6     int valor = 100;
7     printf("%p\n", &valor);
8
9     int *direccion_valor = &valor;
10 }
11

```

0x7fff41b0842c

Ahora vamos a rediseñar la función ‘asigna200’ de forma que, en vez de aceptar un entero, **acepte la dirección de un entero**.

```

1 #include <stdio.h>
2
3 void asigna200(int* direccion_valor)
4 {
5     *direccion_valor = 200;
6 }
7
8 int main()
9 {
10     int valor = 100;
11     printf("%i\n", valor);
12
13     asigna200(&valor);
14
15     printf("%i\n", valor);
16 }
17

```

¡Aja!, ya lo tenemos.

Si observamos bien, **hemos utilizado de nuevo el asterisco para indicar que no queremos cambiar donde apunta el puntero sino el valor apuntado**.

Así, que, con los punteros utilizamos el asterisco de tres formas.

- Para declarar un puntero (que alojará la dirección de una variable).
- Para indicar que queremos cambiar el valor apuntado.
- Para indicar que queremos “recuperar” el valor apuntado.

Esto último se denomina “desreferenciación” y funciona tal que así:

```

1 #include <stdio.h>
2
3 void asigna200(int* direccion_valor)
4 {
5     *direccion_valor = 200;
6
7     int valor_interno = *direccion_valor;
8
9     printf("valor_interno es: %i\n", valor_interno);
10 }
11
12 int main()
13 {
14     int valor = 100;
15     printf("%i\n", valor);
16
17     asigna200(&valor);
100
valor_interno es: 200
200

```

Como vemos, si declaramos una variable con el tipo “puntero a <tipo>” emplearemos el asterisco.

Si deseamos cambiar el valor apuntado, emplearemos el asterisco.

De nuevo, si deseamos obtener el valor al que estamos apuntando...emplearemos el asterisco.

```

1 #include <stdio.h>
2
3 void asigna200(int* direccion_valor)          // Declaramos el parametro como "puntero que apunta a tipo int"
4 {
5     *direccion_valor = 200;                      // Cambiando el valor apuntado
6
7     int valor_interno = *direccion_valor;        // Obteniendo el valor apuntado
8
9     printf("valor_interno es: %i\n", valor_interno);
10 }
11
12 int main()
13 {
14     int valor = 100;
15     printf("%i\n", valor);
16
17     asigna200(&valor);                         // Dirección de la variable valor, no el entero que almacena
18
19     printf("%i\n", valor);
20 }
21
100
valor_interno es: 200
200

```

Los punteros no solo pueden apuntar a valores enteros, sino que pueden apuntar a la dirección de funciones, estructuras, instancias de clases e incluso...a otros punteros. Efectivamente, podemos tener punteros que apuntan a punteros de manera repetitiva.

5.3.1. PUNTEROS A ESTRUCTURAS

```

1 #include <stdio.h>
2
3 struct test
4 {
5     int valor;
6 };
7
8 typedef struct test test;
9
10 int main()
11 {
12     test t {.valor = 100};
13
14     test *puntero_a_test = &t;
15
16     printf("valor: %i\n", puntero_a_test->valor);      // usando la notacion flecha '->'
17     printf("valor: %i\n", (*puntero_a_test).valor);    // desreferenciando el puntero y accediendo al miembro 'valor'
18
19 }
20
21
valor: 100
valor: 100

```

Como podemos ver, en el caso particular de las estructuras, podemos usar dos variantes: la de “desreferencia” y usando el operador flecha “->”. En el caso de la primera debemos asegurarnos de interponer paréntesis ya que el operador ‘.’ posee mayor **precedencia**⁴⁴ que el operador ‘->’ y daría error al compilar.

5.3.2. PUNTEROS A PUNTEROS

Como dijimos, el compilador solo necesita que por cada variable le indiquemos el tipo. El tipo es la idea central en todos los lenguajes de programación con tipado estático y cobra una importancia capital. Al contrario que los dinámicos (más centrados en el qué referencia un nombre) debemos declarar el tipo antes de usarlo e inicializarlo.

Como hemos visto declarar un tipo es algo trivial:

```
int valor = 100;
```

El compilador reserva, habitualmente, 4 bytes en una dirección de memoria y pone allí el número 100. De ahora en adelante, allí donde se use el nombre ‘valor’ querrá decir “la dirección xxxxxxxx”.

Como hemos visto, declarar un puntero es fácil. Agregamos el asterisco **antes del nombre** y lo inicializamos con una dirección (a ser posible una dirección válida o a NULL):

```
int *puntero_a_valor = &valor;

o

int *puntero_nulo = NULL;
```

Pues bien, como ser “puntero a entero” es un tipo, nada nos detiene a declarar un puntero al tipo “puntero a entero”. Con lo cual:

```
int **puntero_a_puntero = &puntero_a_valor;
```

⁴⁴ https://en.cppreference.com/w/c/language/operator_precedence

```

1 #include <stdio.h>
2
3 struct test
4 {
5     int valor;
6 };
7
8 typedef struct test test;
9
10 int main()
11 {
12     test t {.valor = 100};
13
14     test *puntero_a_test = &t;
15
16     test **puntero_a_puntero = &puntero_a_test;
17
18     printf("valor: %i\n", (**puntero_a_puntero).valor);
19
20
21

```

valor: 100

Los punteros a punteros son muy empleados en C. De hecho, si has visto muchas funciones ‘main’ (la función que todo programa ejecutable debe poseer definida en C) de este estilo:

```
int main(int argc, char **argv)
```

5.3.3. PUNTEROS A FUNCIONES

La sintaxis de los punteros a funciones es complicada. Tanto como complicada sea la firma de una función.

Existe una llamativa curiosidad en el caso de las funciones y como veremos de los arrays. **El nombre de una función ya representa su dirección**, por lo que **no es necesario el operador &** cuando asignamos esta a un puntero.

Veamos un ejemplo básico:

```

1 #include <stdio.h>
2
3 int suma (int a, int b)
4 {
5     return a+b;
6 }
7
8 int main()
9 {
10    int resultado = suma(10, 20);
11
12    printf("%i\n", resultado);
13
14    int(*puntero_a_funcion)(int, int) = suma;
15
16    int otro_resultado = puntero_a_funcion(40, 50);
17
18    printf("%i\n", otro_resultado);
19
20 }
21
30
90

```

Como vemos, la dificultad aquí es que debemos replicar la firma de la función y además indicar que se trata de un puntero con el asterisco posicionado delante del nombre.

No nos valdría el puntero ‘puntero_a_funcion’ para otra función que no posea idéntica firma. Daría error por no ser del mismo tipo.

Este tipo de asignación no se suele realizar de forma directa. Para mejorar la reutilización y sobre todo, la legibilidad del código, los programadores se apoyan en **tipos definidos por el usuario**.

Básicamente, es un mecanismo para dotar de un alias a un tipo cualquiera. Ejemplo, en el caso de punteros a funciones:

```

1 #include <stdio.h>
2
3 int suma (int a, int b)
4 {
5     return a+b;
6 }
7
8 int main()
9 {
10    int resultado = suma(10, 20);
11
12    printf("%i\n", resultado);
13
14    typedef int(*puntero_a_funcion)(int, int);
15
16    puntero_a_funcion p_a_f = suma;
17
18    int otro_resultado = p_a_f(40, 50);
19
20    printf("%i\n", otro_resultado);
21
30
90

```

Como vemos, aislamos la definición del tipo con su uso determinado. “p_a_f” es ahora del tipo:

puntero a función que devuelve un entero y toma dos enteros como parámetros.

Por último, vemos como en el caso de las funciones no tenemos que efectuar la “desreferenciación” por el mismo motivo que comentábamos antes. Un nombre de función ya representa su dirección. Es lógico ya que cabe comprender que **las funciones no son un valor determinado o conjunto de estos, sino que apuntan a “comienzos de trozos de código”**.

5.3.4. PUNTEROS A ARRAYS

Los punteros a arrays son otro caso particular. En C, un array es un conjunto de valores del mismo tipo, contiguos en memoria y de una longitud definida. Así, declaramos un array como:

```

1 #include <stdio.h>
2
3 int main()
4 {
5     char array[3] = {1,2,3};
6 }
7
8 |

```

Para declarar un puntero a un array, lo hacemos como si estuviéramos declarando un **puntero al primer elemento**.

Es decir, no hacemos:

```

1 #include <stdio.h>
2
3 int main()
4 {
5     char array[3] = {1,2,3};
6
7     char *p[3] = array;
8 }
9
10

main.cpp: In function 'int main()':
main.cpp:7:18: error: array must be initialized with a brace-enclosed initializer
    7 |         char *p[3] = array;
          ^~~~~~

```

Como vemos, da error. Sin embargo, declaramos el puntero como si solo apuntase al primer elemento:

```

1 #include <stdio.h>
2
3 int main()
4 {
5     char array[3] = {1,2,3};
6
7     char *p = &array[0];
8
9     printf("%i\n", *p);
10}
11
12

```

1

Es decir, declaro un puntero al tipo 'char' y lo inicializo con la dirección del primer elemento de un array de 'char'.

Afortunadamente, existe una forma de expresión más sencilla. Asignar el puntero al array por su nombre:

```

1 #include <stdio.h>
2
3 int main()
4 {
5     char array[3] = {1,2,3};
6
7     char *p = array;
8
9     printf("%i\n", *p);
10}
11
12

```

1

Y es que, cuando el compilador de C ve el nombre de un array lo convierte a la dirección en memoria de su primer elemento, lo que equivale a un puntero.

Curiosamente, cuando pasamos un array a una función como parámetro, C convierte dicho array a un puntero a su primer elemento:

```

1 #include <stdio.h>
2
3 void funcion(char arr[])
4 {
5     char *p = arr;
6     printf("desde la funcion: %i\n", *p);
7 }
8
9 int main()
10 {
11     char array[3] = {1,2,3};
12     funcion(array);
13 }
14
15
16

```

desde la funcion: 1

Dicho efecto se denomina “array to pointer decay”.

Observemos el siguiente ejemplo:

```

1 #include <stdio.h>
2
3 void funcion (char* array_pointer)
4 {
5     printf("%i\n", array_pointer[0]);
6 }
7
8 void funcion2 (char* array_pointer)
9 {
10    printf("%i\n", *array_pointer);
11 }
12
13 int main()
14 {
15     char array[] = {1,2,3};
16     char valor = 100;
17
18     funcion(array);
19     funcion2(array);
20
21     funcion(&valor);
22     funcion2(&valor);
23 }
24

```

1
 1
 100
 100

Ninguna de las dos funciones indica que la función ha de ser llamada con un array, sin embargo, admite tanto arrays (que como estamos diciendo, su nombre es ya la dirección de su primer elemento) como la dirección, de forma explícita, de un objeto del tipo ‘char’.

Es más, observemos como dentro de cada una de las funciones, podemos tratar al puntero como tal o ¡como un array!

5.4. ARITMÉTICA DE PUNTEROS

Los punteros son direcciones, siempre. Además, son **direcciones con un tamaño establecido** por el compilador y la plataforma. En 32 bits, un puntero posee 4 bytes. En 64 bits un puntero son 8 bytes.

No existen medias tintas. Un puntero apunta a una dirección. **Una dirección posee un tamaño fijo.**

Lo que si varía es el tamaño del objeto apuntado. Por lo que **un objeto apuntado ocupar varias direcciones de memoria de forma contigua.**

Veamos, un array es el ejemplo más claro. Posee un tamaño definido y su nombre representa la dirección de su primer elemento.

¿Qué ocurre si sumamos 1 a un puntero? Veámoslo:

```
1 #include <stdio.h>
2
3 void funcion (char *puntero)
4 {
5     printf("%i\n", *puntero);
6     printf("%i\n", *(puntero+1));
7     printf("%i\n", *(puntero+2));
8 }
9
10
11 int main()
12 {
13     char array[] = {1,2,3};
14
15     funcion(array);
16 }
```

1
2
3

Es decir, estamos recorriendo el array si vamos sumando '1' al puntero. Dado que el puntero, como valor, es una dirección de memoria (la del objeto apuntado) si le sumamos una unidad, aumentamos 4 u 8 bytes (o el tamaño de dirección fijo que se posea) y pasa a la siguiente "casilla" u objeto del array.

¿Funcionaría hacia atrás, restando? Vamos a verlo, invocando a la función con el último elemento del array en vez del primero:

```

1 #include <stdio.h>
2
3 void funcion (char *puntero)
4 {
5     printf("%i\n", *(puntero));
6     printf("%i\n", *(puntero+1));
7     printf("%i\n", *(puntero+2));
8 }
9
10
11 int main()
12 {
13     char array[] = {1,2,3};
14
15     funcion(&array[2]);
16 }
17

```

```

3
2
1

```

Curioso, ¿verdad?

No obstante, es un ejemplo. La función no sabe cual es el tamaño de un array y no puede controlar el puntero así porque si le pasamos un array más pequeño no podemos asegurar lo que podría pasar. Estaríamos ante una situación fuera de control y con una alta probabilidad de que generemos una vulnerabilidad.

Habitualmente, en C, se pasa el tamaño de un array junto con este en los argumentos de las funciones.

```

1 #include <stdio.h>
2
3 void funcion (char *puntero, size_t tamanyo)
4 {
5     size_t c = 0;
6     while (c < tamanyo)
7     {
8         printf("%i\n", *(puntero+c));
9         c++;
10    };
11 }
12
13 int main()
14 {
15     char array[] = {1,2,3};
16     funcion(array, sizeof array);
17 }

```

```

1
2
3

```

Aun así, ¿Garantizamos que ese es el tamaño? ¿Qué ocurre si estamos equivocados?

Estaríamos, de nuevo, ante una más que posible vulnerabilidad. Sobre todo, si ese parámetro que controla el tamaño puede ser modificado desde un valor controlado por el usuario.

5.5. COMPLEJOS, POTENTES PERO PELIGROSOS

Los punteros, en los lenguajes C y C++, se relacionan con la gestión de memoria. Por ejemplo, si necesitamos un bufé de tamaño variable acudimos a funciones como ‘malloc’. Esta función, reserva memoria en el montículo y devuelve un puntero al comienzo de bloque de memoria reservado.

Podemos hacernos muchas preguntas solamente respecto a esa funcionalidad:

- ¿Qué pasa si no ha sido posible reservar memoria? ¿En qué estado se encuentra el puntero?
- Declaramos un puntero y no lo inicializamos a una dirección. ¿Qué ocurre si usamos ese puntero, si lo desreferenciamos? ¿Qué contenido poseerá? ¿Qué pasará si escribimos usando su dirección apuntada?
- ¿Y si utilizamos aritmética de punteros y escribimos más allá del tamaño de un array o bloque de memoria reservada?
- ¿Qué pasaría si devolvemos ese bloque reservado con ‘free’ y a continuación volvemos a utilizar el puntero?
- ¿Qué ocurre si llamamos dos veces (o más) a ‘free’?

Todas estas preguntas llevan a distintas clases de vulnerabilidades relacionadas con punteros y gestión manual de la memoria. Responsables, como ya hemos comentado, de incontables vulnerabilidades.

5.6. ¿POR QUÉ ES ÚTIL ESTA INFORMACIÓN?

Los punteros son importantes, muy importantes en programación. Su concepto trasciende más allá de su uso en lenguajes como C.

Sin embargo, un manejo directo por parte del programador puede arruinar la seguridad de un programa. Existen múltiples categorías que definen vulnerabilidades creadas por punteros y una gran parte de las vulnerabilidades encontradas son debidas a un manejo defectuoso de estos.

5.7. ¿QUÉ TENGO QUE HABER APRENDIDO?

- A ver un puntero como un objeto que almacena la dirección de otro objeto
- Lo importante en un puntero es a “donde” apunta y a “qué” apunta.
- Los punteros poseen aritmética, pueden sumarse y restarse.
- Es importante retener que, en C, el nombre de una función y el nombre de un array, por sí mismos, representan su dirección y no es necesario anteponer el operador ‘&’ delante del nombre.
- Comprender que el uso manual por parte del programador representa un riesgo altísimo al ser un mecanismo complejo de utilizar y entender.

6. HERRAMIENTAS Y ENTORNO DE TRABAJO

Empezaremos viendo algunas de las herramientas que podemos usar para ayudarnos en la tarea de entender el código fuente. Sin ellas, la tarea sigue siendo factible pero innecesariamente dificultosa. Sin lugar a duda, el código fuente de un programa puede llegar a analizarse con un simple **editor de texto** sin tan siquiera **resaltado de sintaxis**.

De hecho, una vista entrenada es capaz de seguir la pista a las distintas trazas posibles de ejecución de una parte del código tal y como haría un jugador de ajedrez al calcular las distintas ramificaciones de una partida. Un ejercicio muy práctico consiste en saltar a un archivo cualquiera de un repositorio de código y tratar de entender qué hace una función o clase.

No obstante, utilizar herramientas es fundamental si queremos ser productivos, además, aprender su funcionamiento a fondo es primordial para sacarles provecho. Veremos como existen herramientas que funcionarán con todo tipo de lenguajes y algunas especializadas en algunos de ellos o exclusivamente centrado en uno.

Comenzaremos por una ineludible: el editor de texto.

6.1. EDITORES DE TEXTO

Es oportuno señalar aquí, que los **procesadores de texto** no son herramientas adecuadas para visualizar ni navegar por el código fuente. Se ha de evitar a toda costa su uso y no confundirlos con los editores de texto.

Los editores de texto trabajan sobre lo que se conoce como texto plano. Un archivo de texto plano se representa tal cual. No posee interpretación. Por el contrario, y como ejemplo, un archivo de texto enriquecido se representa de una forma y su contenido, si abriésemos el archivo, sería diferente. En concreto, llevaría un formato de instrucciones que indica cómo se debe representar este.

Cuando lidiermos con código, este se nos presentará como un archivo o un conjunto de archivos de texto. Para representar dichos archivos tenemos varias opciones:

6.1.1. EDITOR DE TEXTO BASADO EN INTERFAZ DE USUARIO

Es el clásico editor de texto que podríamos encontrar como, por ejemplo, el omnipresente **notepad** en los sistemas operativos Microsoft Windows. Este tipo de editores se basan en un estilo visual y están orientados a trabajar con ambientes de escritorio. Visualmente son potentes y permiten exponer información de manera muy clara.

```

C:\Program Files\Notepad++\change.log - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
change.log x
1 Notepad++ v7.7 new features and bug-fixes:
2
3 1. Continue Microsoft binary code signing thanks to the offer from DigiCert (again).
4 2. Upgrade Scintilla from v3.56 to v4.14.
5 3. Fix a regression about memory issue while reloading a file.
6 4. Fix cursor flickering problem after double clicking on URL.
7 5. Make Python files default to using spaces instead of tabs.
8 6. Add "Count in selection" option in Find dialog.
9 7. Add Ctrl + R shortcut for "Reload from disk" command.
10 8. Fix '\' display problem in CSS while using themes (Remove Batang font for CSS).
11 9. Fix crash while right clicking on DocSwitcher's column bar.
12 10. Fix all plugins being removed problem while Plugin Admin removes an old plugin.
13
14
15 Included plugins:
16
17 1. NppExport v0.2.8 (32-bit x86 only)
18 2. Converter 4.2.1
19 3. Mime Tool 2.5
20
21 Undated (Installer only):
< >
length : 904  lines : 23      Ln: 1  Col: 1  Sel: 0 | 0      Windows (CR LF)  UTF-8      IN

```

6.1.2. EDITOR DE TEXTO BASADO EN TERMINAL DE COMANDOS

Este tipo de editores son los que prácticamente inventaron el término. Aun hoy en día, son ampliamente usados en la administración de sistemas y programación de aplicaciones. A pesar de su relativa antigüedad, son programas que permiten una gran integración con herramientas de comando (las cuales usaremos de forma amplia) y otorgan un complemento perfecto para usuarios que prefieren trabajar más en un terminal que en el escritorio.

```

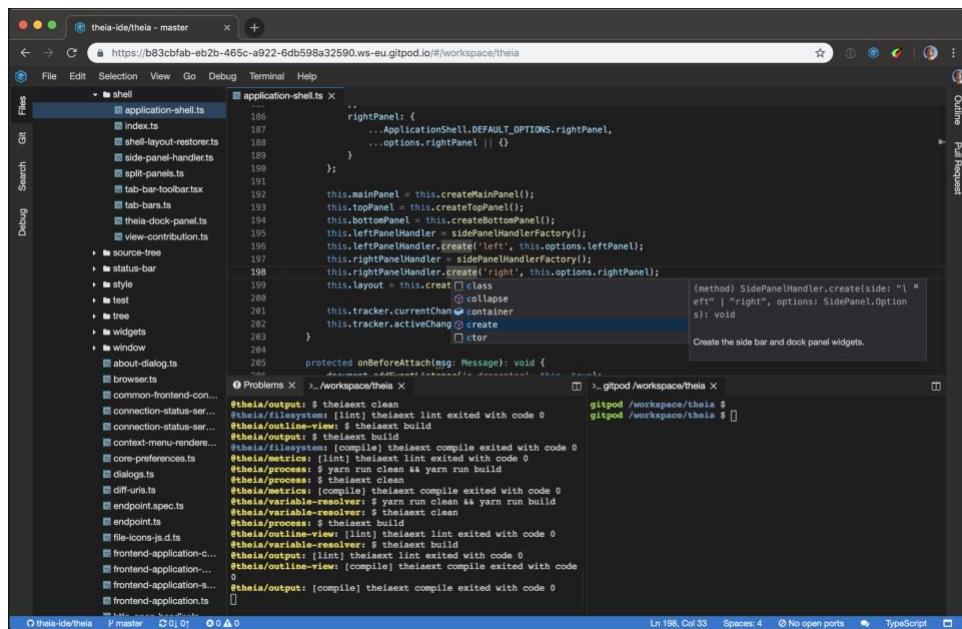
747         lf bufer["%"] != buffer | continue | endif
748         call snewleibang()
749     endifr
750
751     " Because tabbars and other appearing/disappearing windows change
752     " the window number, we have to make sure we're reading the right one
753     let lnum = filter(range(1, winnr("topleft"), winnr("winbelow")))[0]
754     if back | exe back . "wincmd w" | unlet wibye_back | endif
755
756     " If it hasn't been already deleted by buflisted, and its pains now,
757     " unless it previously was an unnamed buffer and :new returned it again.
758     " Using buflisted() over buflisted() because buflisted+=late causes the
759     " buffer to still exist, even though it won't be :bdeletable.
760     if buflisted(buffer) && buffer != bufer["%"]
761         exe 'action'. . ibang. " ". buffer
762     endif
763     endif
764     endifn
765
766     function! slist2bufnr(buffer)
767         if empty(a:buffer)
768             return buffer
769         elseif a:buffer ==# "<d>v"
770             return bufer(str2arr(a:buffer))
771         else
772             return bufer(a:buffer)
773         endif
774     endfunction
775
776     function! snewleibang()
777         exe "new" . ibang
778     endfunction
779
780     set noswapfile
781     " If empty and out of sight, delete it right away:
782     set nobuflisted
783     " Regular buffer warns people if they have unsaved text there. Wouldn't
784     " want to lose someone's data!
785     set nobuflist
786     " Hide the buffer from buffer explorers and tabbars:
787     set nobuflisted
788     endfunction
789
790     " Using the built-in :echomsg prints a stacktrace, which isn't that nice.
791     function! srmfile(arg)
792         silent Erromsg
793         echomsg a:arg
794         echonl NONE
795         let verrormsg = a:arg
796     endfunction
797
798     command! -bang -complete=buffer -nargs=? Bdelete
799         \ call sibdelete("delete", <-q-Bang>, <-q-args>)
800
801     command! -bang -complete=buffer -nargs=? Bipeout
802         \ call sibdelete("bipeout", <-q-Bang>, <-q-args>)
803
NORMAL > -m -2 /Users/davidgarcia/.config/nvim/init.vim

```

6.1.3. EDITOR DE TEXTO BASADO EN NAVEGADOR WEB

Probablemente extrañe la inclusión de un navegador web como editor de textos, pero los avances en aplicaciones en la nube y clientes ricos permiten obtener un producto de gran calidad y presentación que iguala a sus contrapartes nativos.

De hecho, una de las ventajas es precisamente la separación de presentación (cliente) y análisis (servidor). Además, con la tendencia a emplear **servidores de lenguaje⁴⁵** y trabajo en equipo, este tipo de arquitectura se adapta fielmente a dicho esquema.



6.1.4. CARACTERÍSTICAS DESEABLES EN UN EDITOR DE TEXTO

Existe una gran cantidad de editores de texto. Podemos escoger casi cualquiera de ellos, pero daremos una serie de características que debería poseer para facilitar el análisis de código. Nótese que no estamos buscando características para programación, tales como completado de código, etc.

RESALTO DE SINTAXIS

Existen varias opiniones respecto a la necesidad o no del resaltado de sintaxis en el código. Hay

⁴⁵ <https://langserver.org/>

personas que les resulta accesorio e incluso molesto, además, el resaltado de sintaxis ralentiza la carga de los archivos, puesto que deben ser analizados sintácticamente para asignar un determinado color a cada partícula significativa del código.

En el lado opuesto, el resaltado de sintaxis permite identificar rápida y visualmente una palabra clave, variable o separar el código de los comentarios. Además, los últimos avances en servidores de lenguaje hacen que el resaltado de sintaxis se transforme en resultado semántico, ya que añaden propiedades analíticas como, por ejemplo, información contextual.

Véase la diferencia entre resultado vs no resaltado sobre el mismo fragmento de código:

```

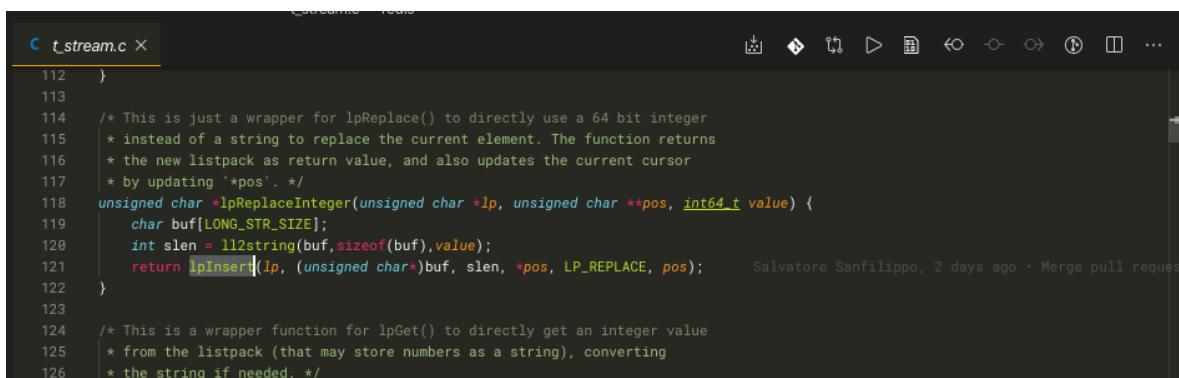
182     |     filename, strerror(errno));
183     |
184     }
185 }
186
187 /* Check if the file is zero-length: if so return C_ERR to signal
188 * we have to write the config. */
189 if (fstat(fileno(fp),&sb) != -1 && sb.st_size == 0) {
190     fclose(fp);
191     return C_ERR;
192 }
193
194
195 /* Parse the file. Note that single lines of the cluster config file can
196 * be really long as they include all the hash slots of the node.
197 * This means in the worst possible case, half of the Redis slots will be
198 * present in a single line, possibly in importing or migrating state, so
199 * together with the node ID of the sender/receiver.
200 *
201 * To simplify we allocate 1024+CLUSTER_SLOTS*128 bytes per line. */
202 maxline = 1024+CLUSTER_SLOTS*128;
203 line = zmalloc(maxline);
204 while(fgets(line,maxline,fp) != NULL) {
205     int argc;
206     sds argv;
207     clusterNode *n, *master;
208     char *p, *s;
209
210     /* Skip blank lines, they can be created either by users manually
211      * editing nodes.conf or by the config writing process if stopped
212      * before the truncate() call. */
213     if (line[0] == '\n' || line[0] == '\0') continue;
214
215     /* Split the line into arguments for processing. */
216     argv = sdssplitargs(line,&argc);
217     if (argc == NULL) goto fmterr;
218
219     /* Handle the special "vars" line. Don't pretend it is the last
220      * line even if it actually is when generated by Redis. */
221     if (strcasecmp(argv[0],"vars") == 0) {
222         if ((argc % 2)) goto fmterr;
223         for (j = 1; j < argc; j += 2) {
224             if (strcasecmp(argv[j],"currentEpoch") == 0) {
225                 server.cluster->currentEpoch =
226                     strtoull(argv[j+1],NULL,10);
227             } else if (strcasecmp(argv[j],"lastVoteEpoch") == 0) {
228                 server.cluster->lastVoteEpoch =
229                     strtoull(argv[j+1],NULL,10);
230             } else {
231                 serverLog(LL_WARNING,
232                         "Skipping unknown cluster config variable '%s'",
233                         argv[j]);
234             }
235         }
236     }
237
238     /* Log the cluster node config from ss: ss */
239     filename, strerror(errno));
240     exit(1);
241 }
242
243 /* Check if the file is zero-length: if so return C_ERR to signal
244 * we have to write the config. */
245 if (fstat(fileno(fp),&sb) != -1 && sb.st_size == 0) {
246     fclose(fp);
247     return C_ERR;
248 }
249
250 /* Parse the file. Note that single lines of the cluster config file can
251 * be really long as they include all the hash slots of the node.
252 * This means in the worst possible case, half of the Redis slots will be
253 * present in a single line, possibly in importing or migrating state, so
254 * together with the node ID of the sender/receiver.
255 *
256 * To simplify we allocate 1024+CLUSTER_SLOTS*128 bytes per line. */
257 maxline = 1024+CLUSTER_SLOTS*128;
258 line = zmalloc(maxline);
259 while(fgets(line,maxline,fp) != NULL) {
260     int argc;
261     sds argv;
262     clusterNode *n, *master;
263     char *p, *s;
264
265     /* Skip blank lines, they can be created either by users manually
266      * editing nodes.conf or by the config writing process if stopped
267      * before the truncate() call. */
268     if (line[0] == '\n' || line[0] == '\0') continue;
269
270     /* Split the line into arguments for processing. */
271     argv = sdssplitargs(line,&argc);
272     if (argc == NULL) goto fmterr;
273
274     /* Handle the special "vars" line. Don't pretend it is the last
275      * line even if it actually is when generated by Redis. */
276     if (strcasecmp(argv[0],"vars") == 0) {
277         if ((argc % 2)) goto fmterr;
278         for (j = 1; j < argc; j += 2) {
279             if (strcasecmp(argv[j],"currentEpoch") == 0) {
280                 server.cluster->currentEpoch =
281                     strtoull(argv[j+1],NULL,10);
282             } else if (strcasecmp(argv[j],"lastVoteEpoch") == 0) {
283                 server.cluster->lastVoteEpoch =
284                     strtoull(argv[j+1],NULL,10);
285             } else {
286                 serverLog(LL_WARNING,
287                         "Skipping unknown cluster config variable '%s'",
288                         argv[j]);
289             }
290         }
291     }
292 }
```

Como ya dijimos, una vista entrenada en un lenguaje de programación verá perfectamente donde están situadas las declaraciones de variables, los tipos, bucles, llamadas de funciones, etc., sin necesidad de activar el resaltado. Sin embargo, es cuestión de gustos, ambas aproximaciones son válidas mientras se adapte a las necesidades de cada cual.

INTEGRACIÓN DE HERRAMIENTAS DE ANÁLISIS DE CÓDIGO

Dado que el código fuente es solo texto, es necesario disponer de herramientas externas que ayuden a constituir una base de conocimiento acerca de la estructura del código. Por ejemplo, señalar una función y poder navegar hacia su definición:

Supongamos que nos encontramos analizando el siguiente fragmento de código:



```

 112 }
113
114 /* This is just a wrapper for lpReplace() to directly use a 64 bit integer
115 * instead of a string to replace the current element. The function returns
116 * the new listpack as return value, and also updates the current cursor
117 * by updating '*pos'. */
118 unsigned char *lpReplaceInteger(unsigned char *lp, unsigned char **pos, int64_t value) {
119     char buf[LONG_STR_SIZE];
120     int slen = ll2string(buf,sizeof(buf),value);
121     return lpInsert(lp, (unsigned char*)buf, slen, *pos, LP_REPLACE, pos);      Salvatore Sanfilippo, 2 days ago · Merge pull request
122 }
123
124 /* This is a wrapper function for lpGet() to directly get an integer value
125 * from the listpack (that may store numbers as a string), converting
126 * the string if needed. */

```

Necesitamos encontrar la definición de la función señalada ‘lpInsert’. Tendríamos varias opciones, una de ellas sería recurrir a herramientas de búsqueda de texto y mediante un escaneo de todo el código fuente acceder a todos los símbolos ‘lpInsert’ hasta dar con la localización de la definición.

Otra opción es apoyarnos en herramientas de análisis para crear un índice donde ya se encuentre la definición y accedamos directamente mediante una opción del propio editor de textos.

Veamos la primera opción, usando una herramienta de búsqueda especializada (ripgrep):

```

(venv) + redls git:(unstable) rg lpInsert
src/listpack.c
595:unsigned char *lpInsert(unsigned char *lp, unsigned char *ele, uint32_t size, unsigned char *p, int where, unsigned char **newp) {
734: * listpack. It is implemented in terms of lpInsert(), so the return value is
735: * the same as lpInsert(). */
739:     return lpInsert(lp,ele,size,eofptr,LB_BEFORE,NULL);
747:     return lpInsert(lp,NULL,0,p,LB_REPLACE,newp);

src/t_stream.c
121:     return lpInsert(lp, (unsigned char*)buf, slen, *pos, LP_REPLACE, pos);

src/listpack.h
42:/* lpInsert() where argument possible values: */
49:unsigned char *lpInsert(unsigned char *lp, unsigned char *ele, uint32_t size, unsigned char *p, int where, unsigned char **newp);
(venv) + redls git:(unstable) 

```

Invocaríamos el comando: “rg lpInsert” y tendríamos que observar los resultados hasta ver donde está la definición de esta función. En este caso en concreto, nos señala el archivo “src/listpack.c” línea 595.

En la segunda opción (usando el editor de código Visual Studio Code):

```

116  * the new listpack as return value, and also updates the current cursor
117  * by updating 'pos'. */
118  unsigned char *lpReplaceInteger(unsigned char *lp, unsigned char **pos, int64_t value) {
119      char buf[LONG_STR_SIZE];
120      int slen = ll2string(buf, sizeof(buf), value);
121      return lpInsert(lp, pos, slen, &buf[0], LP_REPLACE);
122 }
123 /* This is a wrapper
124 * from the list
125 * the string if
126 * int64_t lpGetInt
127     int64_t v;
128     unsigned cha
  
```

The screenshot shows a context menu in Visual Studio Code with the following options:

- Run Code
- Go to Definition** (highlighted)
- Go to Declaration
- Go to References
- Peek
- Search in devDocs.io [⌘K ⌘K]

Y accedemos directamente a su definición:

```

3  * element, on the right of the deleted one, or to NULL if the deleted element
4  * was the last one. */
5  unsigned char *lpInsert(unsigned char *lp, unsigned char *ele, uint32_t size, unsigned char *p, int where, unsigned char **newp) {
6      unsigned char intenc[LP_MAX_INT_ENCODING_LEN];
7      unsigned char backlen[LP_MAX_BACKLEN_SIZE];
8
9      uint64_t enclen; /* The length of the encoded element. */
10
11     /* An element pointer set to NULL means deletion, which is conceptually
12      * replacing the element with a zero-length element. So whatever we
  
```

INTEGRACIÓN DE LA DOCUMENTACIÓN

Otro de los recursos que necesitaremos y al que acudiremos con muy alta frecuencia es la documentación del código. Por supuesto, esta no siempre estará disponible. De hecho, en ingeniería inversa es poco común encontrarnos con una documentación salvo en el caso de API REST documentadas (en el caso de aplicaciones web), llamadas al sistema (por ejemplo, el API de Windows).

No obstante, en el caso de que estemos trabajando sobre una base de código documentada, resulta muy adecuado disponer de un acceso rápido a la documentación del código. Mientras que, en lenguajes con tipos estáticos esta información ya podemos derivarla de los tipos de los parámetros y de los valores de retorno de las funciones (lo veremos más adelante), en lenguajes con tipos dinámicos no está disponible y deberemos de acudir a su documentación o a la definición de dicho objeto en el código fuente.

Por ello, es necesario (aunque no es esencial) que el editor de texto nos permita ver o enlazar al menos la documentación; si es que esta existe o es accesible. En la imagen de abajo podemos ver como mediante una función proveída por un complemento de Vim, es posible abrir la documentación en línea de la función “printf” de la librería estándar de C.

The screenshot shows the std::printf function page on cppreference.com. It includes the function signature, detailed documentation, parameters, and the source code implementation. The source code is written in C and shows how it handles arguments, opens files, and writes to them.

```

4 // Copyright (c) 2018 Salvatore Sanfilippo.
5 // This software is released in the 3-clause BSD license. */
6
7 #include <stdio.h>
8 #include <unistd.h>
9 #include <sys/stat.h>
10 #include <stdlib.h>
11 #include <time.h>
12
13 int main(int argc, char **argv) {
14     struct stat stat;
15     int fd, cycles;
16
17     if (argc != 3) {
18         fprintf(stderr, "Usage: <filename> <cycles>\n");
19         exit(1);
20     }
21
22     srand(time(NULL));
23     char filename[argc-1];
24     strcpy(filename, argv[1]);
25     fd = open(filename, O_RDWR);
26     if (fd == -1) {
27         perror("open");
28         exit(1);
29     }
30     fstat(fd, &stat);
31
32     while(cycles--) {
33         unsigned char buf[32];
34         unsigned long offset = rand()&stat.st_size;
35         int writelen = lrand()&31;
36         int j;
37
38         for (j = 0; j < writelen; j++) buf[j] = (char)rand();
39         lseek(fd, offset, SEEK_SET);
40         write(fd, buf, writelen);
41         printf("writing %d bytes at offset %lu\n", writelen, offset);
42         write(fd, buf, writelen);
43     }
44 }

```

Disponer de la documentación es una gran ventaja, ya que nos permite entender que hace la función sin necesidad de leer el código fuente de su implementación. No obstante, salvo casos concretos, deberemos tomar la documentación como una guía y no como algo escrito en piedra. Las razones son sencillas: la documentación puede estar anticuada o puede estar incompleta o simplemente es errónea.

6.2. INDEXADORES SIMBÓLICOS

También denominados de creación de referencias cruzadas (cross reference), estas herramientas realizan una lectura por todos los archivos fuentes de un programa y sus librerías asociadas, creando una tabla índice donde se describe el símbolo, su tipo, localización y en algunos casos, la asociación con el resto de los símbolos.

Saber trabajar con y disponer de estas herramientas supone una gran ventaja estratégica. Tan solo debemos apuntar a un símbolo y operar de forma adecuada para observar las referencias, tipo, tipo de retorno y argumentos en funciones, etc. En algunos casos, también podremos saber en qué sitios se llama a una función concreta o cuando se está manipulando una variable en concreto.

Observemos un simple uso de “ctags” sobre la base del código fuente de la base de datos “Redis”:

```
+ redis git:(unstable) x l
total 344
drwxr-xr-x 25 davidgarcia staff 800B 4 mar 21:15 .
drwxr-xr-x 38 davidgarcia staff 1,2K 4 mar 09:53 ..
drwxr-x--- 4 davidgarcia staff 128B 4 mar 21:15 .ccls-cache
drwxr-xr-x 14 davidgarcia staff 448B 4 mar 22:20 .git
drwxr-xr-x 3 davidgarcia staff 96B 4 mar 06:23 .github
-rw-r--r-- 1 davidgarcia staff 400B 4 mar 06:23 .gitignore
-rw-r--r-- 1 davidgarcia staff 634B 4 mar 06:23 00-RELEASENOTES
-rw-r--r-- 1 davidgarcia staff 53B 4 mar 06:23 BUGS
-rw-r--r-- 1 davidgarcia staff 2,3K 4 mar 06:23 CONTRIBUTING
-rw-r--r-- 1 davidgarcia staff 1,5K 4 mar 06:23 COPYING
-rw-r--r-- 1 davidgarcia staff 11B 4 mar 06:23 INSTALL
-rw-r--r-- 1 davidgarcia staff 6,7K 4 mar 06:23 MANIFESTO
-rw-r--r-- 1 davidgarcia staff 151B 4 mar 06:23 Makefile
-rw-r--r-- 1 davidgarcia staff 20K 4 mar 06:23 README.md
-rw-r--r-- 1 davidgarcia staff 3,1K 4 mar 06:23 TLS.md
drwxr-xr-x 9 davidgarcia staff 288B 4 mar 06:23 deps
-rw-r--r-- 1 davidgarcia staff 78K 4 mar 06:23 redis.conf
-rw-r--r-- 1 davidgarcia staff 275B 4 mar 06:23 runtest
-rwrxr-xr-x 1 davidgarcia staff 280B 4 mar 06:23 runtest-cluster
-rwrxr-xr-x 1 davidgarcia staff 679B 4 mar 06:23 runtest-moduleapi
-rwrxr-xr-x 1 davidgarcia staff 281B 4 mar 06:23 runtest-sentinel
-rw-r--r-- 1 davidgarcia staff 9,5K 4 mar 06:23 sentinel.conf
drwxr-xr-x 142 davidgarcia staff 4,4K 4 mar 06:23 src
drwxr-xr-x 13 davidgarcia staff 416B 4 mar 06:23 tests
drwxr-xr-x 24 davidgarcia staff 768B 4 mar 06:23 utils
+ redis git:(unstable) x ctags -R
+ redis git:(unstable) x head tags
!.TAG_FILE_FORMAT 2 /extended format; --format=1 will not append ; to lines/
!.TAG_FILE_SORTED 1 /=unsorted, 1=sorted, 2=foldcase/
!.TAG_OUTPUT_MODE u-ctags /u-ctags or e-ctags/
!.TAG_PROGRAM_AUTHOR Universal Ctags Team //
!.TAG_PROGRAM_NAME Universal Ctags /Derived from Exuberant Ctags/
!.TAG_PROGRAM_URL https://ctags.io/ /official site/
!.TAG_PROGRAM_VERSION 0.8.0 /cb84253/
$(CPP_OBJS) deps/jemalloc/Makefile.in /*$(CPP_OBJS) $(CPP_PIC_OBJS) $(TESTS_CPP_OBJS): %.$(O)$:/; t
$(CPP_OBJS) deps/jemalloc/Makefile.in /*$(CPP_OBJS) $(C_SYM_OBJS) $(C_JET_OBJS) $(C_JET_SYM_OBJS): CPPFLAGS += -DDLEXPORT$/; t
$(CPP_OBJS) deps/jemalloc/Makefile.in /*$(CPP_OBJS): $(objroot)src/%.$(O): $(srcroot)src/%.cpp$/; t
+ redis git:(unstable) x
```

Para operar con “ctags” de forma sencilla le pasamos simplemente el parámetro “-R” para que efectúe un análisis recursivo. “ctags” recorrerá el árbol de directorios en busca de archivos de código fuente y formará un archivo (nombrado “tags”) con todo el contenido indexado.

Observemos el contenido de dicho archivo, con el índice ya creado con todos los símbolos del código:

```
4520 T_NUMBERdeps/lua/src/luajson.c "/NUMBER;" e enum:_anon3a73c778103 file:
4521 T_0B3_BEGINdeps/lua/src/luajson.c "/_0B3_BEGIN://" e enum:_anon3a73c778103 file:
4522 T_0B3_ENDdeps/lua/src/luajson.c "/_0B3_END://" e enum:_anon3a73c778103 file:
4523 T_STRINGdeps/lua/src/luajson.c "/_STRING://" e enum:_anon3a73c778103 file:
4524 T_UNKNOWNdeps/lua/src/luajson.c "/_UNKNOWN://" e enum:_anon3a73c778103 file:
4525 T_WHITESPACEdeps/lua/src/luajson.c "/_WHITEPACE://" e enum:_anon3a73c778103 file:
4526 Tabledeps/lua/src/lobject.h "/typedef struct Table {" s
4527 Tabledeps/lua/src/lobject.h "/} Table;" t typepref:struct:Table
4528 Tabledeps/lua/src/tracking_collisions.c "/Int Table[TABLE_SIZE];/" v typepref:typename:int
4529 Terminalsin 2010, deps/linenoise/README.markdown ## Terminals, in 2010.4; " s
4530 TerminateModuleForChildsrc/module.c /"int TerminateModuleForChild(int child_pid, int wait) {"/; f typepref:typename:int
4531 TestAssertIntegerReplysrc/modules/testmodule.c /"int TestAssertIntegerReply(RedisModuleCtx *ctx, RedisModuleCallReply *reply, long long expected);/" f typepref:typename:int
4532 TestAssertStringReplysrc/modules/testmodule.c /"int TestAssertStringReply(RedisModuleCtx *ctx, RedisModuleCallReply *reply, const char *str, size_t len);/" f typepref:typename:int
4533 TestCallsrc/modules/testmodule.c /"int TestCall(RedisModuleCtx *ctx, RedisModuleString **argv, int argc) {"/; f typepref:typename:int
4534 TestTxFlagssrc/modules/testmodule.c /"int TestTxFlags(RedisModuleCtx *ctx, RedisModuleString **argv, int argc) {"/; f typepref:typename:int
4535 TestItsrc/modules/testmodule.c /"int TestIt(RedisModuleCtx *ctx, RedisModuleString **argv, int argc) {"/; f typepref:typename:int
4536 TestMatchReplysrc/modules/testmodule.c /"int TestMatchReply(RedisModuleCallReply *reply, char *str) {"/; f typepref:typename:int
4537 TestNotificationssrc/modules/testmodule.c /"int TestNotifications(RedisModuleCtx *ctx, RedisModuleString **argv, int argc) {"/; f typepref:typename:int
4538 TestStringAppendsrc/modules/testmodule.c /"int TestStringAppend(RedisModuleCtx *ctx, RedisModuleString **argv, int argc) {"/; f typepref:typename:int
4539 TestStringAppendAMsrc/modules/testmodule.c /"int TestStringAppendAM(RedisModuleCtx *ctx, RedisModuleString **argv, int argc) {"/; f typepref:typename:int
4540 TestStringPrintfsrc/modules/testmodule.c /"int TestStringPrintf(RedisModuleCtx *ctx, RedisModuleString **argv, int argc) {"/; f typepref:typename:int
4541 Testunlinksrc/modules/testmodule.c /"int Testunlink(RedisModuleCtx *ctx, RedisModuleString **argv, int argc) {"/; f typepref:typename:int
4542 Tested with...deps/linenoise/README.markdown ## Tested with...$/; s
4543 TestsTLS.md /## Tests$/; s
4544 The APIdeps/lumenuse/README.markdown /* The API$/; c
4545 TimeoutStartSecutils/systemd/multiple_servers.service /*TimeoutStartSec=infinity$/; k section:Service
4546 TimeoutStartSecutils/systemd/redis_server.service /*TimeoutStartSec=infinity$/; k section:Service
4547 TimeoutStopSecutils/systemd/multiple_servers.service /*TimeoutStopSec=infinity$/; k section:Service
4548 TimeoutStopSecutils/systemd/redis_server.service /*TimeoutStopSec=infinity$/; k section:Service
4549 TimerCommandRedisCommandsrc/modules/hellotimer.c /"int TimerCommand.RedisCommand(RedisModuleCtx *ctx, RedisModuleString **argv, int argc) {"/; f typepref:typename:int
4550 Timerssrc/module.c /*static rax *Timers; \V* The radix tree of all the timers sorted by expire. */$/; v typepref:typename:rax * file:
4551 To-Do ListTLS.md /To-Do List$/; s
```

Como podemos observar, cada línea posee información concreta de un objeto determinado. Su nombre, archivo, comando EX⁴⁶ que fue usado para la extracción del símbolo, abreviatura y tipo. Mirar dicho archivo no nos conduce a nada, salvo qué, mediante una búsqueda (por ejemplo, con el comando “grep”), queramos observar las distintas declaraciones de un nombre. El verdadero poder de dicha herramienta se encuentra dentro de un editor, cuando es invocada para navegar por el código.

Supongamos que estamos leyendo el archivo de código fuente en C, “deps/hiredis/async.h” y nos

⁴⁶ Ex es un popular editor de texto basado en línea, cuando se interactuaba con teletipos en vez de pantallas.
[https://en.wikipedia.org/wiki/Ex_\(text_editor\)](https://en.wikipedia.org/wiki/Ex_(text_editor))

topamos con una estructura con un miembro del tipo puntero a “redisContext”, pero no sabemos nada acerca de él.

```

54     redisCallback *head, *tail;
55 } redisCallbackList;
56
57 /* Connection callback prototypes */
58 typedef void (redisDisconnectCallback)(const struct redisAsyncContext*, int status);
59 typedef void (redisConnectCallback)(const struct redisAsyncContext*, int status);
60 typedef void (redisTimerCallback)(void *timer, void *privdata);
61
62 /* Context for an async connection to Redis */
63 typedef struct redisAsyncContext {
64     /* Hold the regular context, so it can be realloc'ed. */
65     redisContext c;
66
67     /* Setup error flags so they can be used directly. */
68     int err;
69     char *errstr;
70
71     /* Not used by hiredis */
72     void *data;
73
74     /* Event library data and hooks */
75     struct {
76         void *data;
77
78         /* Hooks that are called when the library expects to start
79         * reading/writing. These functions should be idempotent. */
80         void (*addRead)(void *privdata);
81         void (*delRead)(void *privdata);
82         void (*addWrite)(void *privdata);
83         void (*delWrite)(void *privdata);
84         void (*cleanup)(void *privdata);
85         void (*scheduleTimer)(void *privdata, struct timeval tv);
86     } ev;
87
88     /* Called when either the connection is terminated due to an error or per

```

El procedimiento sin tener un índice de símbolos sería ver que archivos de inclusión poseen la definición de este símbolo. Eso nos llevaría a abrir todos esos archivos y realizar una búsqueda. Los usuarios más avanzados podrán hacer un “grep” recursivo (una herramienta para realizar búsqueda de texto mediante línea de comandos) o mediante herramientas similares.

Sin embargo, poseer un editor de código con soporte para las “tags” o etiquetas generadas por “ctags” nos permiten ir hacia la definición con tan solo una combinación de teclas:

```

205 /* Context for a connection to Redis */
206 typedef struct redisContext {
207     const redisContextFuncs *funcs; /* Function table */
208
209     int err; /* Error flags, 0 when there is no error */
210     char errstr[128]; /* String representation of error when applicable */
211     redisFD fd;
212     int flags;
213     char *obuf; /* Write buffer */
214     redisReader *reader; /* Protocol reader */
215
216     enum redisConnectionType connection_type;
217     struct timeval *timeout;
218
219     struct {
220         char *host;
221         char *source_addr;
222         int port;
223     } tcp;
224
225     struct {
226         char *path;
227     } unix_sock;
228
229     /* For non-blocking connect */
230     struct sockaddr *saddr;
231     size_t addrlen;
232
233     /* Additional private data for hiredis addons such as SSL */
234     void *privdata;
235 } redisContext;
236

```

Hay tenemos la definición del símbolo, con toda la información que necesitamos para interpretar el uso que se pueda hacer de esa variable dentro de la estructura que vimos anteriormente.

Ahora veremos algunas herramientas que nos servirán de apoyo para la generación de “tags” o índice de símbolos.

6.2.1. CTAGS

Fue de las primeras herramientas de este tipo. Programada por Ken Arnold para el sistema operativo BSD, su popularidad alcanzó finalmente a todos los sistemas de la familia UNIX. Además, casi todos los editores de texto poseen un soporte para ctags, ya sea integrado en el editor o a través de complementos.

Aunque el binario se llame ctags, por ejemplo, en el sistema operativo Linux, el proyecto se refundó en “Exhuberant ctags”. Actualmente es mantenido vía un fork denominado Universal Ctags⁴⁷.

Ctags posee un rico abanico de opciones, no obstante, es posible obtener un resultado aceptable con simplemente indicar que analice los archivos de forma recursiva cuando se invoca dentro de un directorio, por ejemplo, con:

```
$ ctags -R .
```

Ctags es bastante rápido, incluso para proyectos tan grandes como el kernel Linux (y dependiendo del hardware) es posible generar un archivo índice en relativamente poco tiempo.

6.2.2. GNU GLOBAL

Aunque de similar uso, integración y capacidades, GNU Global⁴⁸ (en adelante, “Global”) es independiente del editor y puede ser usado directamente en la línea de comandos. Además, incluye funcionalidades adicionales que no se encuentran en ctags, por ejemplo, poder listar todas las referencias de una función determinada. No obstante, para funcionar depende de ctags para algunos lenguajes; debido a que utiliza dicha herramienta para generar el índice de las definiciones.

Para usar Global, en primer lugar, se debe llamar a ‘gtags’, con la cual se generan tres archivos que son bases de datos con todo el contenido resultante de la indexación. Por un lado, se genera GPATH, que contiene las referencias a los archivos que contienen el código fuente. GRTAGS, que contiene todas las referencias de los símbolos. Esto es, cuando y donde es llamado un símbolo dentro del programa. GTAGS, que contiene la definición de los símbolos tal y como lo encontramos en ctags, pero en formato base de datos.

Una vez se generan estas bases de datos, es posible realizar consultas de cualquier símbolo. Por ejemplo:

⁴⁷ <https://github.com/universal-ctags/ctags>

⁴⁸ <https://www.gnu.org/software/global/global.html>

Búsqueda mostrando los usos de la función ‘raxSeek’ a lo largo del código de Redis:

```
- redis git:(unstable) ✘ global -xr raxSeek
raxSeek      1379 src/acl.c          raxSeek(&rl,"^",NULL,0);
raxSeek      1717 src/acl.c          raxSeek(&rl,"^",NULL,0);
raxSeek      1237 src/aofc.c         raxSeek(&rl,"^",NULL,0);
raxSeek      1253 src/aofc.c         raxSeek(&rl_cons,"^",NULL,0);
raxSeek      1260 src/aofc.c         raxSeek(&rl_pel,"^",NULL,0);
raxSeek      1231 src/config.c       raxSeek(&rl,">",INDEXED,2);
raxSeek      1668 src/db.c          raxSeek(&rlter,>,"INDEXED,2");
raxSeek      1688 src/db.c          raxSeek(&rlter,">","INDEXED,2");
raxSeek      1688 src/db.c          raxSeek(&rlter,">","INDEXED,2");
raxSeek      667 src/defrag.c        raxSeek(&rl,"^",NULL,0);
raxSeek      670 src/defrag.c        if (!raxSeek(&rl,>,"last",sizeof(last))) {
raxSeek      718 src/defrag.c        raxSeek(&rl,"^",NULL,0);
raxSeek      5249 src/module.c       raxSeek(&rl,"^",NULL,0);
raxSeek      5380 src/module.c       raxSeek(&rl,"^",NULL,0);
raxSeek      5678 src/module.c       raxSeek(&rl->rl.op.key,keylen);
raxSeek      5781 src/module.c       return raxSeek(&rl->i.op.key,keylen);
raxSeek      6027 src/module.c       raxSeek(&rl,"^",NULL,0);
raxSeek      869 src/object.c        raxSeek(&rl,"^",NULL,0);
raxSeek      883 src/object.c        raxSeek(&rl,"$",NULL,0);
raxSeek      894 src/object.c        raxSeek(&rl,"^",NULL,0);
raxSeek      908 src/object.c        raxSeek(&rl_i,"^",NULL,0);
raxSeek      1545 src/rax.c          raxSeek(&rl,">","NULL,0");
raxSeek      200 src/rax.h          int raxSeek(raxIterator *it, const char *op, unsigned char *ele, size_t len);
raxSeek      696 src/rdb.c          raxSeek(&rl,"^",NULL,0);
raxSeek      732 src/rdb.c          raxSeek(&rl,"^",NULL,0);
raxSeek      911 src/rdb.c          raxSeek(&rl,"^",NULL,0);
raxSeek      944 src/rdb.c          raxSeek(&rl,"^",NULL,0);
raxSeek      214 src/t_stream.c     raxSeek(&rl,"$",NULL,0);
raxSeek      403 src/t_stream.c     raxSeek(&rl,"^",NULL,0);
raxSeek      415 src/t_stream.c     raxSeek(&rl,">","rl.key,rl.key_len");
raxSeek      533 src/t_stream.c     raxSeek(&rl_i->rl,"<=",(unsigned char*)si->start_key,
raxSeek      535 src/t_stream.c     if ((rxEof(&rl_i->r1)) raxSeek(&rl_i->r1,"^",NULL,0));
raxSeek      537 src/t_stream.c     raxSeek(&rl_i->rl,"^",NULL,0);
raxSeek      541 src/t_stream.c     raxSeek(&rl_i->rl,"<=",(unsigned char*)si->end_key,
raxSeek      543 src/t_stream.c     if ((rxEof(&rl_i->r1)) raxSeek(&rl_i->r1,"$",NULL,0));
raxSeek      545 src/t_stream.c     raxSeek(&rl_i->rl,"^",NULL,0);
raxSeek      1058 src/t_stream.c    raxSeek(&rl,">","startkey",sizeof(startkey));
raxSeek      1717 src/t_stream.c    raxSeek(&rl,"^",NULL,0);
raxSeek      2010 src/t_stream.c    raxSeek(&rl,"^",NULL,0);
raxSeek      2010 src/t_stream.c    raxSeek(&rl,"$",NULL,0);
raxSeek      2024 src/t_stream.c    raxSeek(&rl,"^",NULL,0);
raxSeek      2061 src/t_stream.c    raxSeek(&rl,">","startkey",sizeof(startkey));
raxSeek      2511 src/t_stream.c    raxSeek(&rl,"^",NULL,0);
raxSeek      2527 src/t_stream.c    raxSeek(&rl,"^",NULL,0);
raxSeek      73 src/tracking.c    raxSeek(&rl,"^",NULL,0);
raxSeek      251 src/tracking.c    raxSeek(&rl,"^",NULL,0);
raxSeek      278 src/tracking.c    raxSeek(&rl,"^",NULL,0);
raxSeek      367 src/tracking.c    raxSeek(&rl,"^",NULL,0);
raxSeek      306 src/tracking.c    raxSeek(&rl_i,"^",NULL,0);
raxSeek      410 src/tracking.c    raxSeek(&rl_i,"^",NULL,0);
raxSeek      423 src/tracking.c    raxSeek(&rl_i,"^",NULL,0);
+ redis git:(unstable) ✘
```

Cómo podemos observar, tenemos agrupados los archivos y líneas donde se está haciendo uso de dicha función.

De manera aislada, preguntamos a Global donde se encuentra la definición de este símbolo:

```
- redis git:(unstable) ✘ global -x raxSeek
raxSeek      1508 src/rax.c      int raxSeek(raxIterator *it, const char *op, unsigned char *ele, size_t len) {
```

Global es muy potente y al igual que ctags soporta numerosos lenguajes de programación. Será una herramienta esencial para observar donde se define un símbolo y que usos se hace de aquel a lo largo de un programa.

6.2.3. CSCAPE

cscope⁴⁹ añade una dimensión más a los resultados que ya podíamos obtener con Global. Además de lo que podíamos hacer con las anteriores herramientas, ahora podemos preguntar a cscope, por ejemplo, qué funciones llama cierta función.

Lo primero es construir la base de datos que utilizará cscope, aunque primero se le debe señalar que archivos del código fuente deberán ser agregados al análisis. Por ejemplo, para un proyecto en lenguaje C, usamos la herramienta ‘find’ para que nos liste todos los archivos que pertenecen a este lenguaje y queramos indexar:

```
find . -name '*.[h,c]' > cscope.files
```

Ahora disponemos de un archivo con un listado de rutas para añadir a la base de datos de cscope. Generamos esta con:

```
cscope -b -q
```

Si ahora ejecutásemos cscope en la raíz del proyecto nos aparecería un menú con diversas opciones para realizar búsquedas:

```
Find this C symbol: █
Find this global definition:
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:
```

Como podemos ver, ahora disponemos de más opciones y tipos de búsqueda.

⁴⁹ <http://cscope.sourceforge.net/>

Veamos un ejemplo con el mismo símbolo anterior ‘raxSeek’ del código fuente de Redis, supongamos que queremos saber de un golpe, qué funciones llama la mencionada función:

```
Functions called by this function: raxSeek
  File Function      Line
  0 rax.c raxSeek    1545 return raxSeek(it,>=,NULL,0);
  1 rax.c raxSeekGreatest 1552 if (!raxSeekGreatest(it)) return 0;
  2 rax.c assert     1553 assert(it->node->iskey);
  3 rax.c raxGetData 1554 it->data = raxGetData(it->node);
  4 rax.c raxLowWalk 1562 size_t i = raxLowWalk(it->rt,ele,len,&it->node,NULL,&splitpos,&it->stack);
  5 rax.c raxIteratorAddChars 1572 if (!raxIteratorAddChars(it,ele,len)) return 0;
  6 rax.c raxGetData 1573 it->data = raxGetData(it->node);
  7 rax.c raxStackPush 1580 if (!raxStackPush(&it->stack,it->node)) return 0;
  8 rax.c raxIteratorAddChars 1585 if (!raxIteratorAddChars(it,parent->data,parent->size))
  9 rax.c raxNodeFirstChildPtr 1588 raxNode **cp = raxNodeFirstChildPtr(parent);
 a rax.c memcpy       1592 memcpy(&aux,cp,sizeof(aux));
 b rax.c raxIteratorAddChars 1597 if (!raxIteratorAddChars(it,p,1)) return 0;
 c rax.c raxStackPop 1600 raxStackPop(&it->stack);
 d rax.c debugf     1604 debugf("After initial seek: i=%d len=%d key=%.*s\n",
 e rax.c raxIteratorAddChars 1612 if (!raxIteratorAddChars(it,ele+i,1)) return 0;
 f rax.c debugf     1613 debugf("Seek normal node on mismatch: %.*s\n",
 g rax.c raxIteratorPrevStep 1617 if (lt && !raxIteratorPrevStep(it,1)) return 0;
 h rax.c raxIteratorNextStep 1618 if (gt && !raxIteratorNextStep(it,1)) return 0;
 i rax.c debugf     1621 debugf("Compressed mismatch: %.*s\n",
 j rax.c raxIteratorNextStep 1632 if (!raxIteratorNextStep(it,0)) return 0;
 k rax.c raxIteratorAddChars 1634 if (!raxIteratorAddChars(it,it->node->data,it->node->size))
 l rax.c raxIteratorNextStep 1636 if (!raxIteratorNextStep(it,1)) return 0;
 m rax.c raxSeekGreatest 1645 if (!raxSeekGreatest(it)) return 0;
 n rax.c raxGetData    1646 it->data = raxGetData(it->node);
 o rax.c raxIteratorAddChars 1648 if (!raxIteratorAddChars(it,it->node->data,it->node->size))
 p rax.c raxIteratorPrevStep 1650 if (!raxIteratorPrevStep(it,1)) return 0;
 q rax.c debugf     1655 debugf("No mismatch: %.*s\n",
 r rax.c raxGetData    1676 it->data = raxGetData(it->node);
 s rax.c raxIteratorNextStep 1678 if (gt && !raxIteratorNextStep(it,0)) return 0;
 t rax.c raxIteratorPrevStep 1679 if (lt && !raxIteratorPrevStep(it,0)) return 0;
```

Si queremos ver donde exactamente es llamada una determinada función, pulsamos enter y nos invoca el editor que tengamos definido en la variable de entorno \$EDITOR, abriendo el archivo y desplazando el cursor en la misma línea de la referencia, por ejemplo, veamos donde llama ‘raxSeek’ a la función estándar ‘memcpy’:

```
1574     } else { if (lt || gt) {
1575         /* Exact key not found or eq flag not set. We have to set as current
1576         * key the one represented by the node we stopped at, and perform
1577         * a next/prev operation to seek. To reconstruct the key at this node
1578         * we start from the parent and go to the current node, accumulating
1579         * the characters found along the way. */
1580     if (!raxStackPush(&it->stack,it->node)) return 0;
1581     for (size_t j = 1; j < it->stack.items; j++) {
1582         raxNode *parent = it->stack.stack[j-1];
1583         raxNode *child = it->stack.stack[j];
1584         if (parent->iscompr) {
1585             if (!raxIteratorAddChars(it,parent->data,parent->size))
1586                 return 0;
1587         } else {
1588             raxNode **cp = raxNodeFirstChildPtr(parent);
1589             unsigned char *p = parent->data;
1590             while(1) {
1591                 raxNode *aux;
1592                 memcpy(&aux,cp,sizeof(aux));
1593                 if (aux == child) break;
1594                 cp++;
1595                 p++;
1596             }
1597             if (!raxIteratorAddChars(it,p,1)) return 0;
1598         }
1599     }
1600     raxStackPop(&it->stack);
1601
1602     /* We need to set the iterator in the correct state to call next/prev
1603     * step in order to seek the desired element. */
1604     debugf("After initial seek: i=%d len=%d key=%.*s\n",
1605     (int)i, (int)len, (int)it->key_len, it->key);
1606     /* If the last node is compressed, we must
1607     * free it before we can move to the next node. */
1608 }
```

Naturalmente, también poseemos la funcionalidad ya vista con Global, la referencia cruzada de

todas las localizaciones donde se llama a 'raxSeek':

```
Functions calling this function: raxSeek
  File      Function
  0 src@rax.h unknown
  1 acl.c    ACLSaveToFile
  2 acl.c    aclCommand
  3 aof.c    rewriteStreamObject
  4 aof.c    rewriteStreamObject
  5 aof.c    rewriteStreamObject
  6 config.c configSetCommand
  7 db.c     delKeysInSlot
  8 db.c     delKeysInSlot
  9 defrag.c scanLaterStreamListpacks
 a defrag.c scanLaterStreamListpacks
 b defrag.c defragRadixTree
 c module.c moduleTimeHandler
 d module.c RM_CreateTimer
 e module.c RM_DictIteratorStartC
 f module.c RM_DictIteratorReseekC
 g module.c RM_FreeServerInfo
 h object.c objectComputeSize
 i object.c objectComputeSize
 j object.c objectComputeSize
 k object.c objectComputeSize
 l rax.c    raxSeek
 m rax.h    raxSeek
 n rdb.c    rdbSaveStreamPEL
 o rdb.c    rdbSaveStreamConsumers
 p rdb.c    rdbSaveObject
 q rdb.c    rdbSaveObject
 r t_stream.c streamAppendItem
 s t_stream.c streamTrimByLength
 t t_stream.c streamTrimByLength
 u t_stream.c streamIteratorStart
 v t_stream.c streamIteratorStart
 w t_stream.c streamIteratorStart
 x t_stream.c streamIteratorStart
 y t_stream.c streamIteratorStart
 z t_stream.c streamIteratorStart
 A t_stream.c streamReplyWithRangeFromConsumerPEL
 B t_stream.c streamDelConsumer
 C t_stream.c xpendingCommand
 D t_stream.c xpendingCommand
 E t_stream.c xpendingCommand
 F t_stream.c xpendingCommand
 G t_stream.c xinfoCommand
 H t_stream.c xinfoCommand
 I tracking. disableTracking
 J tracking. trackingRememberKeyToBroadcast
 K tracking. trackingInvalidatedKey
 L tracking. trackingLmtUsedSlots
 M tracking. trackingBroadcastInvalidationMessages
 N tracking.c trackingBroadcastInvalidationMessages
 O tracking.c trackingBroadcastInvalidationMessages

Line
 200 int raxSeek(raxIterator *it, const char *op, unsigned char *ele, size_t len);
1379 raxSeek(&rl, ">", NULL, 0);
1717 raxSeek(&rl, "<", NULL, 0);
1237 raxSeek(&rl, "^^", NULL, 0);
1253 raxSeek(&rl_cons, "^^", NULL, 0);
1260 raxSeek(&rl_pel, "^^", NULL, 0);
1231 raxSeek(&rl, "~~", NULL, 0);
1668 raxSeek(&iter, ">=", Indexed, 2);
1688 raxSeek(&iter, ">=", Indexed, 2);
667 raxSeek(&rl, "~~", NULL, 0);
670 if (!raxSeek(&rl, ">", last, sizeof(last))) {
  718 raxSeek(&rl, "~~", NULL, 0);
5249 raxSeek(&rl, "~~", NULL, 0);
5385 raxSeek(&rl, "~~", NULL, 0);
5676 raxSeek(&dl->r1, op, key, keylen);
5781 return raxSeek(dl->r1, op, key, keylen);
6027 raxSeek(&rl, "~~", NULL, 0);
869 raxSeek(&rl, "~~", NULL, 0);
883 raxSeek(&rl, "$", NULL, 0);
895 raxSeek(&rl, "~~", NULL, 0);
986 raxSeek(&rl, "~~", NULL, 0);
1545 return raxSeek(it, ">", NULL, 0);
200 int raxSeek(raxIterator *it, const char *op, unsigned char *ele, size_t len);
696 raxSeek(&rl, "~~", NULL, 0);
732 raxSeek(&rl, "~~", NULL, 0);
911 raxSeek(&rl, "~~", NULL, 0);
944 raxSeek(&rl, "~~", NULL, 0);
214 raxSeek(&rl, "$", NULL, 0);
403 raxSeek(&rl, "~~", NULL, 0);
415 raxSeek(&rl, ">=", rl.key.rl.key_len);
533 raxSeek(&sl->i, "<", (unsigned char *)si->start_key,
535 if (raxEOF(&sl->r1)) raxSeek(&sl->r1, "~~", NULL, 0);
537 raxSeek(&sl->i, "~~", NULL, 0);
541 raxSeek(&sl->i, "<=", (unsigned char *)si->end_key,
543 if (raxEOF(&sl->r1)) raxSeek(&sl->r1, "k", NULL, 0);
545 raxSeek(&sl->i, "$", NULL, 0);
1058 raxSeek(&rl, ">=", startkey, sizeof(startkey));
1717 raxSeek(&rl, "~~", NULL, 0);
2018 raxSeek(&rl, "~~", NULL, 0);
2016 raxSeek(&rl, "$", NULL, 0);
2024 raxSeek(&rl, "~~", NULL, 0);
2061 raxSeek(&rl, ">=", startkey, sizeof(startkey));
2511 raxSeek(&rl, "~~", NULL, 0);
2537 raxSeek(&rl, "~~", NULL, 0);
73 raxSeek(&rl, "~~", NULL, 0);
251 raxSeek(&rl, "~~", NULL, 0);
278 raxSeek(&rl, "~~", NULL, 0);
367 raxSeek(&rl, "~~", NULL, 0);
396 raxSeek(&rl, "~~", NULL, 0);
410 raxSeek(&r2, "~~", NULL, 0);
423 raxSeek(&r2, "~~", NULL, 0);

Find this C symbol:
Find this global definition:
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:
```

No obstante, cscope es una herramienta muy centrada en el código del lenguaje C. Aunque posee un buen soporte de Java y C++, en determinadas construcciones de estos lenguajes puede que no se obtengan resultados satisfactorios.

6.3. NAVEGADORES DE CÓDIGO

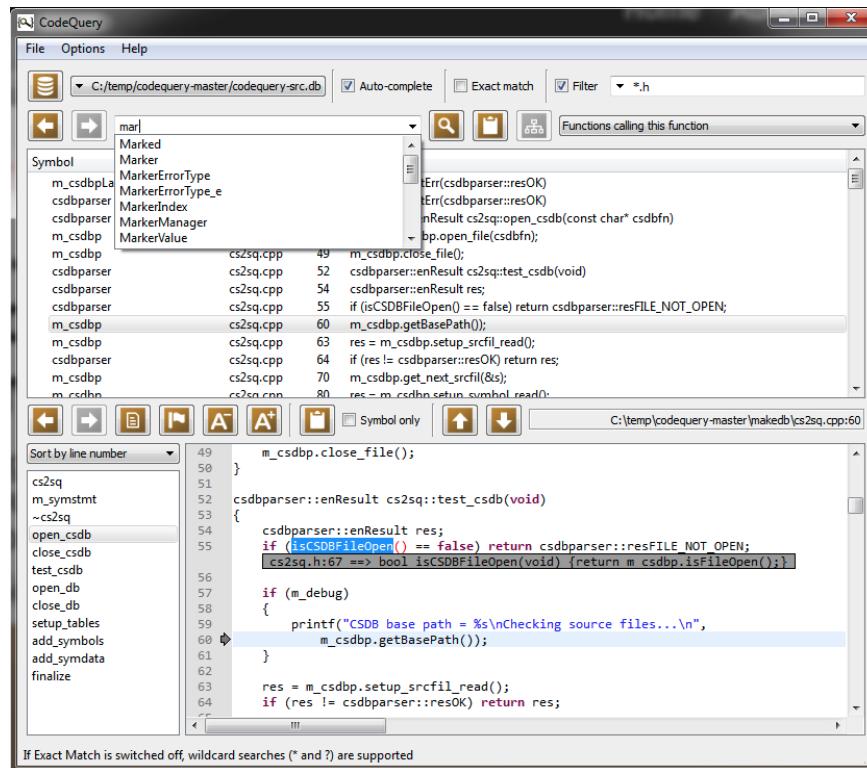
Hemos descrito con cierto nivel de detalle las herramientas anteriores, ya que estas son la base de posteriores desarrollos de utilidades adaptadas a otros lenguajes de programación. Ctags, Global y cscope nacieron cuando el desarrollo en sistemas UNIX era dominado por el lenguaje C. Por ello, están centradas sobre todo en este lenguaje, aunque poseen cierto soporte para otros.

Presentamos ahora herramientas que además del indexado nos permiten muchas más opciones, algunas indispensables como navegación por el código e incluso el dibujo de grafos de relaciones entre módulos y símbolos.

Son autenticos todo en uno que además de una relativa fácil instalación poseen un interfaz interactivo, moderno y amigable.

6.3.1. CODEQUERY

Codequery⁵⁰ es, básicamente, un frontend de ‘cscope’ y ‘exhuberant tags’ (combina las dos), aunque con muchas más opciones que nos permitirá navegar por el código, realizar búsquedas avanzadas, autocompletado y creación de grafos entre otras características.



⁵⁰ <https://github.com/ruben2020/codequery>

6.3.2. OPENGROK

También disponemos de completas suites de navegación de código y referencias con los proyectos ‘Elixir⁵¹’, ‘lxr’⁵² y ‘OpenGrok’⁵³ que incluyen una arquitectura cliente (web) – servidor, con lo que podemos centralizar el indexado y dotar a un equipo de analistas de acceso via navegador web.

En la imagen podemos ver una instalación de OpenGrok buscando referencias del símbolo de ejemplo (la función ‘raxSeek’) del proyecto Redis que hemos estado usando:

The screenshot shows the OpenGrok search interface at localhost:8080/search?project=src&full=raxSeek&defs=&refs=&path=&hist=&type=&si=full. The search term 'full:raxseek' is entered in the search bar. The results list shows 12 matches across various Redis source files, each with line numbers and a snippet of the code containing the 'raxSeek' function.

```

Searched full:raxseek (Results 1 – 12 of 12) sorted by relevance
/src/
  73 raxSeek(zri,"",NULL,0); in disableTracking()
  251 raxSeek(zri,"",NULL,0); in trackingForMemberKeyToBroadcast()
  278 raxSeek(zri,"",NULL,0); in trackingValidateKey()
  367 raxSeek(zri,"",NULL,0); in trackingLimitUsedSlots()
  396 raxSeek(zri,"",NULL,0); in trackingBroadcastInvalidationMessages()
  410 raxSeek(zrl2,"",NULL,0); in trackingBroadcastInvalidationMessages()
  423 raxSeek(zrl2,"",NULL,0); in trackingBroadcastInvalidationMessages()
  214 raxSeek(zri,$,NULL,0); in streamAppendItem()
  403 raxSeek(zri,">",NULL,0); in streamTrimByLength()
  415 raxSeek(zri,>"$zr.key,r1.key_len); in streamTrimByLength()
  537 raxSeek(zri->r1,"",NULL,0); in streamIteratorStart()
  545 raxSeek(zri->r1,$,NULL,0); in streamIteratorStart()
  2511 raxSeek(zrl2,"",NULL,0); in xinfoCommand()
  [all ...]
  869 raxSeek(zri,"",NULL,0); in objectComputeSize()
  883 raxSeek(zri,$,NULL,0); in objectComputeSize()
  895 raxSeek(zri,"",NULL,0); in objectComputeSize()
  906 raxSeek(zrl1,"",NULL,0); in objectComputeSize()
  667 raxSeek(zrl1,"",NULL,0); in scanLaterStreamListpacks()
  678 if (!raxSeek(zrl1,>"",last, sizeof(last))) { in scanLaterStreamListpacks()
  718 raxSeek(zrl1,"",NULL,0); in defragRadixtree()
  200 int raxSeekIterator(it, const char *op, unsigned char *ele, size_t len);
  1254 * to initialize the iterator, and must be followed by a raxSeek() call,
  1305 * the parent will be skipped. This option is used by raxSeeki() when
  1598 return raxSeekIterator(it, const char *op, unsigned char *ele, size_t len) { in raxSeek()
  1545 return raxSeek(it,>"",NULL,0); in raxSeek()
  1810 /* Return if the iterator is in an EOF state. This happens when raxSeek()
  696 raxSeek(zrl1,"",NULL,0); in rdSaveStreamElt()
  732 raxSeek(zrl1,"",NULL,0); in rdSaveStreamConsumers()
  911 raxSeek(zrl1,"",NULL,0); in rdSaveObject()
  944 raxSeek(zrl1,"",NULL,0); in rdSaveObject()
  1227 raxSeek(zrl1,"",NULL,0); in rewriteStreamObject()
  1228 raxSeek(zrl1_cons,"",NULL,0); in rewriteStreamObject()
  1260 raxSeek(zrl1_pel,"",NULL,0); in rewriteStreamObject()
  1668 raxSeek(zrl1,>"",indexEd,2); in gethysinSlot()
  1688 raxSeek(zrl1,>"",indexEd,2); in delhysinSlot()
  1379 raxSeek(zrl1,"",NULL,0); in ACLSaveToFile()
  1717 raxSeek(zrl1,"",NULL,0); in aclCommand()
  5249 raxSeek(zrl1,"",NULL,0); in moduleTimerHandler()
  5305 raxSeek(zrl1,"",NULL,0); in RM_CreateTimer()
  5676 raxSeek(zdi->r1,op,key,keylen); in RM_DictIteratorStartC()
  5701 return raxSeek(zdi->r1,op,key,keylen); in RM_DictIteratorReseekC()
  6027 raxSeek(zrl1,"",NULL,0); in RM_FreeServerInfo()
  HAD config.c 1231 raxSeek(zrl1,"",NULL,0); in configSetCommand()

```

51 <https://github.com/bootlin/elixir>

52 <https://sourceforge.net/projects/lxr/>

53 <https://github.com/oracle/opengrok>

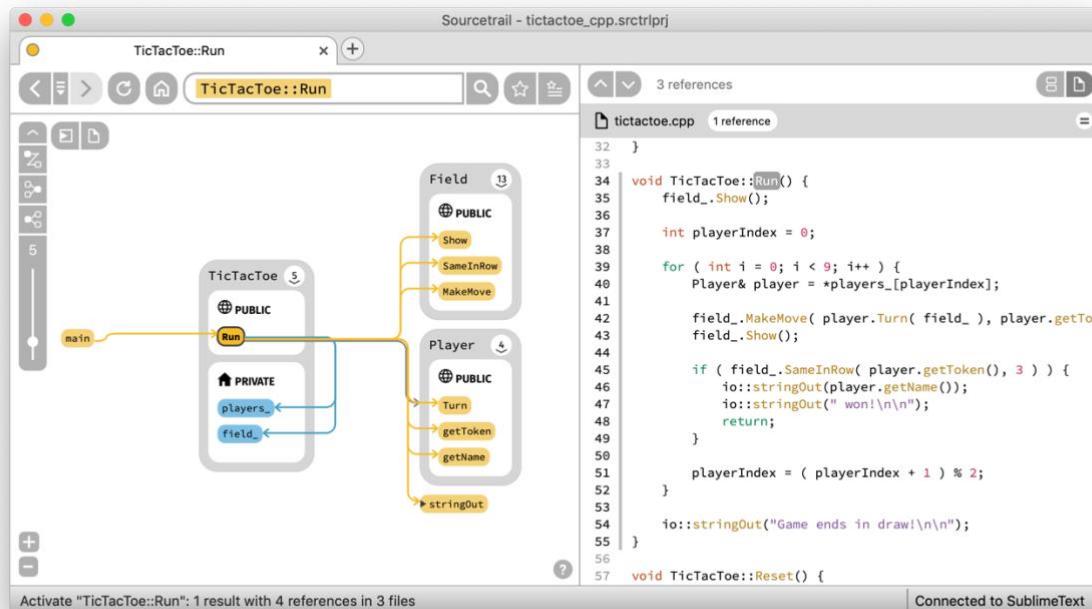
Aunque la instalación de OpenGrok es medianamente compleja, es posible usar un contenedor Docker sobre una imagen ya creada de OpenGrok (disponible en Docker Hub) para proyectos de rango medio.

Este tipo de herramientas son esenciales para analizar código. Durante nuestra tarea de análisis, podemos vernos profundizando en el uso de una función a varios niveles o capas de llamadas a funciones que pueden, a su vez, convertirse en otros tantos niveles de llamadas.

Por lo tanto, es fundamental dominar dicha complejidad a través de estas herramientas. Saber configurarlas y usarlas con soltura (navegar de forma transversal y vertical) nos proporcionará un gran control sobre el código y sobre todo no romper el hilo de pensamiento sobre el que estemos discutiendo; especialmente, si tenemos que emplear recursos de tiempo y concentración en tareas de búsqueda manual.

6.3.3. SOURCETRAIL

Sourcetrail⁵⁴ es una herramienta similar a Codequery, en el sentido de que también es una aplicación stand-alone basada en Qt. Es muy potente visualmente y además permite la interconexión con numerosos IDE y editores de texto como Visual Studio Code o Vim.



Fuente de la imagen: www.sourcetrail.com

Es, además de potente, utiliza un motor propio de indexado que además es extensible a través de

⁵⁴ [https://www.sourcetrail.com/](http://www.sourcetrail.com)



un SDK publicado por la propia organización.

6.4. HERRAMIENTAS DE BÚSQUEDA

En determinados momentos, no solo querremos buscar entre el código de forma simbólica sino buscar patrones de texto. Esto puede surgir por diferentes factores, por ejemplo: buscar en los comentarios, detectar el uso de estructuras de código específicas o simplemente, no disponemos de una herramienta de indexado de símbolos.

Afortunadamente, disponemos de un autentico arsenal de este tipo de herramientas. Comenzando por ‘grep’⁵⁵ la más veterana hasta implementaciones en lenguajes modernos (rust) como ‘ripgrep’⁵⁶ o ‘the silver searcher’⁵⁷.

Si bien difieren en cuanto a opciones particulares, en general, este tipo de herramientas posee un cuerpo común de funcionalidades compartidas entre ellas. A saber: búsqueda recursiva por directorios, empleo de expresiones regulares, ignorar archivos, etc.

Veamos un uso estándar de una de estas herramientas, en concreto de ‘ripgrep’. Por ejemplo, vamos a hacer una búsqueda del mismo símbolo de ejemplo que hemos estado usando anteriormente, la función ‘raxSeek’:

⁵⁵ <http://man7.org/linux/man-pages/man1/grep.1.html>

⁵⁶ <https://github.com/BurntSushi/ripgrep>

⁵⁷ https://github.com/ggreer/the_silver_searcher

```
# Redis git:(unstable) ✘ rg raxSeek *
Binary file GTAGS matches (found "\u{0}" byte around offset 3)
Binary file GRTAGS matches (found "\u{0}" byte around offset 3)

src/object.c
869:     raxSeek(&ri,"^",NULL,0);
883:     raxSeek(&ri,"$",NULL,0);
895:     raxSeek(&ri,"^",NULL,0);
996:     raxSeek(&cri,"^",NULL,0);

src/tracking.c
73:     raxSeek(&ri,"^",NULL,0);
251:     raxSeek(&ri,"^",NULL,0);
278:     raxSeek(&ri,"^",NULL,0);
367:     raxSeek(&ri,"^",NULL,0);
396:     raxSeek(&ri,"^",NULL,0);
410:     raxSeek(&ri2,"^",NULL,0);
423:     raxSeek(&ri2,"^",NULL,0);

src/t_stream.c
214:     raxSeek(&ri,"$",NULL,0);
493:     raxSeek(&ri,"^",NULL,0);
415:     raxSeek(&ri,">",rl.key.rl.key_len);
533:     raxSeek(&si->r1,"cm", (unsigned char*)si->start_key,
535:             if (raxEOF(&si->r1)) raxSeek(&si->r1,"^",NULL,0);
537:     raxSeek(&si->r1,"^",NULL,0);
541:     raxSeek(&si->r1,"cm", (unsigned char*)si->end_key,
543:             if (raxEOF(&si->r1)) raxSeek(&si->r1,"$",NULL,0);
545:     raxSeek(&ri,"^",NULL,0);
1958: raxSeek(&ri,"^",startkey,sizeof(startkey));
1717: raxSeek(&ri,"^",NULL,0);
2910:     raxSeek(&ri,"^",NULL,0);
2916:     raxSeek(&ri,"$",NULL,0);
2924:     raxSeek(&ri,"^",NULL,0);
2961:     raxSeek(&ri,"^",startkey,sizeof(startkey));
2511:     raxSeek(&ri,"^",NULL,0);
2537:     raxSeek(&ri,"^",NULL,0);

src/aof.c
1237:     raxSeek(&ri,"^",NULL,0);
1253:     raxSeek(&ri_cons,"^",NULL,0);
1260:     raxSeek(&ri_pel,"^",NULL,0);

src/rax.h
290:int raxSeek(raxIterator *it, const char *op, unsigned char *ele, size_t len);

src/rdb.c
696:     raxSeek(&rl,"^",NULL,0);
732:     raxSeek(&rl,"^",NULL,0);
911:     raxSeek(&rl,"^",NULL,0);
944:     raxSeek(&rl,"^",NULL,0);

src/module.c
5249:     raxSeek(&ri,"^",NULL,0);
5305:     raxSeek(&ri,"^",NULL,0);
5676: raxSeek(&dl->i.op,key,keylen);
5701: return raxSeek(&dl->r1.op,key,keylen);
6927:     raxSeek(&ri,"^",NULL,0);

src/config.c
1231:     raxSeek(&ri,"^",NULL,0);

src/acl.c
1379:     raxSeek(&ri,"^",NULL,0);
1717:     raxSeek(&ri,"^",NULL,0);

src/defrag.c
667:     raxSeek(&rl,"^",NULL,0);
670:     if (!raxSeek(&rl,>, last, sizeof(last))) {
718:     raxSeek(&rl,"^",NULL,0);

src/db.c
1668: raxSeek(&iter,"^",indexed,2);
1688:     raxSeek(&iter,"^",indexed,2);

src/rax.c
1254: /* to initialize the iterator, and must be followed by a raxSeek() call,
1305: * the parent will be skipped. This option is used by raxSeek() when
1412:int raxSeekGreatest(raxIterator *it) {
1490:     if (!raxSeekGreatest(it)) return 0;
1500:     raxSeek(&riter,*it, const char *op, unsigned char *ele, size_t len) {
1545:         return raxSeek(&riter,>,NULL,0);
1552:         if (!raxSeekGreatest(it)) return 0;
1645:             if (!raxSeekGreatest(it)) return 0;
1810:/* Return if the iterator is in an EOF state. This happens when raxSeek()
```

Como podemos observar, la presentación de resultados es bastante correcta y podemos encontrar rápidamente las referencias deseadas. Ahora bien, si se observa con detenimiento la imagen, podemos observar como también se añaden los resultados de esa cadena, ‘raxSeek’, en los comentarios.

A menos que deseemos precisamente buscar documentación acerca de esta función en concreto, no nos interesarán. Esta es la diferencia fundamental entre un indexador de código y una herramienta de búsqueda: estas últimas no distinguen código de texto.

6.5. CONTROL DE VERSIONES DE CÓDIGO

Si con la navegación de código disponemos de la facilidad de navegar por niveles de forma vertical y horizontal, el versionado de código nos permite agregar una nueva dimensión: el tiempo. En determinados casos no solo dispondremos del código fuente tal cual, sino de toda o parte de su historia.

Por ejemplo, supongamos que estamos analizando el código fuente de OpenSSL⁵⁸ en busca de bugs. Disponemos no solo del código sino de todo el desarrollo que ha ocurrido a lo largo del tiempo desde que se comenzó a introducir código en el sistema de versionado. Gracias a este tipo de software, podremos ver la evolución del código.

Aunque el versionado de código ha estado disponible desde hace décadas, con los primeros sistemas, por ejemplo, el venerable RCS⁵⁹, actualmente el más popular y extendido es Git⁶⁰.

Observemos un uso de esta misma herramienta, Git, para ver la evolución de la función que hemos estado usando:

```

rax.c [1409:54] -> rax.c [39:232] X
1337 // If no key is returned, then 0 is returned.
1338 int raxSeek(raxIterator *it, unsigned char *ele, size_t len, const char *op) {
1339     int eq = 0; /* Just resetting. Initialized by raxStart(), etc */
1340     int gt = 0; /* Just resetting. Initialized by raxStart(), etc */
1341     int lt = 0; /* Just resetting. Initialized by raxStart(), etc */
1342     int flags = RAX_ITER_JUST_SEEKED;
1343     it->flags = RAX_ITER_EOF;
1344     it->key_len = 0;
1345     it->node = NULL;
1346
1347     /* Set flags according to the operator used to perform the seek. */
1348     if (op[0] == '<') {
1349         if (op[1] == '=') eq = 1;
1350         else if (op[1] == '<') lt = 1;
1351         else if (op[1] == '>') gt = 1;
1352         else if (op[1] == '=') eq = 1;
1353         else if (op[1] == '>') gt = 1;
1354         else if (op[1] == '<=') {
1355             eq = 1;
1356         } else if (op[1] == '>=') {
1357             gt = 1;
1358         } else if (op[1] == '<>') {
1359             lt = 1;
1360         }
1361     }
1362     else {
1363         /* If there are no elements, set the EOF condition immediately and
1364         * return. */
1365         if (it->rt->node == 0) {
1366             it->flags |= RAX_ITER_EOF;
1367             return 1;
1368         }
1369     }
1370
1371     /* If (first) {
1372        * Seeking the first key greater or equal to the empty string
1373        * is equivalent to seeking the smaller key available. */
1374     if (first) {
1375         /* Find the greatest key taking always the last child till a
1376         * final node is found. */
1377         it->node = it->rt->head;
1378         if (!raxSeekerFirst(it)) return 0;
1379         assert(it->node->key);
1380         return 1;
1381     }
1382
1383     if (last) {
1384         /* Find the greatest key taking always the last child till a
1385         * final node is found. */
1386         it->node = it->rt->head;
1387         if (!raxSeekerLast(it)) return 0;
1388         assert(it->node->key);
1389         return 1;
1390     }
1391
1392     /* We need to seek the specified key. What we do here is to actually
1393     * perform a lookup, and later invoke the prev/next key code that
1394     * is already use for iteration. */
1395     int f_size = raxNodeSize(it->rt->ele, len, it->node, NULL, &splitpos, it->stack);
1396     size_t f_size = raxNodeSize(it->rt->ele, len, it->node, NULL, &splitpos, it->stack);
1397
1398     /* Return DOM on incomplete stack info. */
1399     if (!it->stack.dom) return 0;
1400
1401     if (eq && lt == len && (!it->node->iscomp || splitpos == 0) &&
1402         !it->node->iskey) {
1403         /* We found our node, since the key matches and we have an
1404         * 'equal' condition */
1405         if (!raxIteratorAddChar(it, ele, len)) return 0; /* DOM. */
1406
1407         /* If we found our node, we have to set as current
1408         * key the one represented by the node we stopped at, and perform
1409         * a next/prev operation to seek. To reconstruct the key at this node
1410         */
1411     }
1412
1413     /* Exact key not found or eq flag not set. We have to set as current
1414     * key the one represented by the node we stopped at, and perform
1415     * a next/prev operation to seek. To reconstruct the key at this node
1416     */
1417 }
```

⁵⁸ <https://github.com/openssl/openssl>

⁵⁹ <https://www.gnu.org/software/rcs/>

⁶⁰ <https://git-scm.com/>

Con detalle, observemos como se han ido agregando cambios, algunos incluso bastante notorios, por ejemplo, el ajuste de la variable global ‘errno’ (usada en lenguaje C para indicar un estado de error determinado):

<pre> } else if (op[0] == '\$') { last = 1; } else { return 0; /* Error. */ } /* If there are no elements, set the EOF condition immediately and */ </pre>	<pre> 1386 } else if (op[0] == '\$') { 1387 last = 1; 1388 } else { 1389 errno = 0; 1390 return 0; /* Error. */ 1391 } 1392 /* If there are no elements, set the EOF condition immediately and */ 1393 </pre>
--	--

Un cambio curioso en el orden de los parámetros de una llamada a si misma (‘raxSeek’ es recursiva):

<pre> 2 /* Seeking the first key greater or equal to the empty string 3 * is equivalent to seeking the smaller key available. 4 return raxSeek(it,NULL,0,>=); 5 } 6 7 if (last) { 8 /* Find the greatest key taking always the last child till a */ </pre>	<pre> 1481 /* Seeking the first key greater or equal to the empty 1482 * is equivalent to seeking the smaller key available 1483 return raxSeek(it,>=,NULL,0); 1484 } 1485 1486 if (last) { 1487 /* Find the greatest key taking always the last child */ 1488 </pre>
--	---

Un cambio condicional que protege las operaciones en dependencia del operador que se le pase a la función:

<pre> + "equal" condition. */ if (!raxIteratorAddChars(it,ele,len)) return 0; /* OOM. */ } else { /* Exact key not found or eq flag not set. We have to set as current * key the one represented by the node we stopped at, and perform * a next/prev operation to seek. To reconstruct the key at this node */ } </pre>	<pre> 1428 * "equal" condition. */ 1429 if (!raxIteratorAddChars(it,ele,len)) return 0; /* OOM. */ 1430 } else if (!lt gt) { 1431 /* Exact key not found or eq flag not set. We have to set as current 1432 * key the one represented by the node we stopped at, and perform 1433 * a next/prev operation to seek. To reconstruct the key at this node 1434 */ </pre>
--	--

Finalmente, un cambio que modifica el iterador proporcionado a la función a través de los parámetros (it-> flags |= RAX_ITER_OF) para indicar que se ha llegado al final de la iteración por la estructura de datos y no se ha encontrado un resultado (return 1)

<pre> if (gt && !raxIteratorNextStep(it,0)) return 0; if (lt && !raxIteratorPrevStep(it,0)) return 0; it->flags = RAX_ITER JUST_SEEKED; /* Ignore next call. */ } } return 1; </pre>	<pre> 1518 if (gt && !raxIteratorNextStep(it,0)) return 0; 1519 if (lt && !raxIteratorPrevStep(it,0)) return 0; 1520 it->flags = RAX_ITER JUST_SEEKED; /* Ignore next call. */ 1521 } 1522 } else { 1523 /* If we are here just eq was set but no match was found. */ 1524 it->flags = RAX_ITER_EOF; 1525 } 1526 1527 return 1; 1528 </pre>
---	---

Como podemos observar, la dimensión aportada por un control de versiones es única y nos proporciona una mirada a través del tiempo de la evolución del código. En este caso, podemos ver como el o los programadores han ido agregando cambios a esta función para mejorar su resiliencia a posibles errores y comunicar mejor su estado en dicho sentido.

El uso que podemos dar de esta nueva perspectiva es fundamental para adquirir conocimientos acerca de como ha ido evolucionando el desarrollo o como se han corregido los errores; algo fundamental en el campo del Bug Hunting.

6.6. ¿POR QUÉ ES ÚTIL ESTÁ INFORMACIÓN?

Si hay algo que tienen todas las especialidades de ciberseguridad son **las herramientas**. Incluso existen distribuciones (y varias) dedicadas a recolectar herramientas de seguridad. Es complicado realizar las distintas y complejas tareas de análisis de malware, reversing y búsqueda de vulnerabilidades prescindiendo de las herramientas.

Este capítulo ha estado dedicado precisamente a eso, a las herramientas que podemos usar en el análisis de código fuente. Dado que es una tarea que no precisa de análisis dinámico (entraríamos ya en el campo del análisis vivo tal, que se mostrará en el módulo del curso dedicado a dicha tarea) las herramientas se centran en búsquedas, indexación de símbolos, navegadores de código, etc.

Es esencial, no usarlas todas, pero si haberlas probado y quedarse con aquellas que mejor casan con nuestro estilo de analista. Aquellas que elijamos como nuestras, deberemos dedicarles tiempo para dominarlas y ser productivos con ellas.

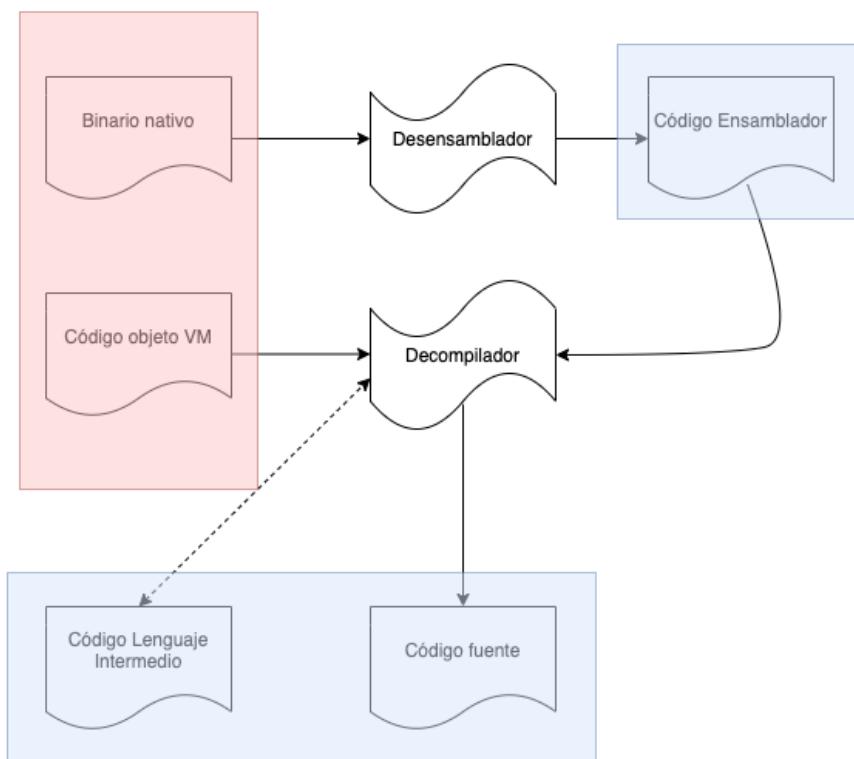
6.7. ¿QUÉ TENGO QUE HABER APRENDIDO?

- Conocer que herramientas tienes al alcance para analizar código.
- Entender que para la tarea de análisis de código va a ser casi imprescindible un editor de código, un indexador y un buscador de cadenas.
- El código fuente puede provenir de un repositorio con control de versiones (Git, SVN, etc) es imprescindible entender que se trata de “máquinas del tiempo”, donde podremos viajar hasta otra versión de un mismo fragmento de código y entender como ha evolucionado con el tiempo.

7. DESENSAMBLADO, DECOMPILEACIÓN Y CÓDIGO OFUSCADO

Si bien la asignatura versa sobre el estudio y análisis del código fuente, este no siempre va a estar disponible. Habrá veces que nos enfrentemos a un binario y poco más. Sin embargo, es posible realizar un proceso para revertir un ejecutable a formatos que nos permitan obtener la lógica del programa en forma de código. Estas herramientas son denominadas **desensambladores** y **decompiladores**.

Aunque anteriormente hemos hablado de este proceso, veremos brevemente en que consiste mediante un esquema:



Como podemos ver, en sombreado rojo, inicialmente partimos de un binario nativo o código objeto para una VM concreta (como .Net o Java). No podríamos efectuar análisis de código puesto que este no existe como tal.

Si disponemos de un binario lo pasariamos por un desensamblador para intentar obtener su código ensamblador. Esto ya lo vimos en el apartado del módulo de introducción. Por otro lado, y como ejemplo, si dispusiésemos de una aplicación Java lo haríamos con un decompilador para esta plataforma.

En ambos casos, ya tendríamos código disponible para su lectura y comprensión (zona sombreada de azul).

No obstante, aun así, para el ensamblador del binario podríamos efectuar una nueva iteración bajo

un descompilador e intentar traducir el código ensamblador a código fuente (normalmente C y C++). Decimos intentar, porque es complicado para un descompilador traducir un ensamblado, máxime si este es complejo o ha sido previamente ofuscado.

Para el caso de un objeto ejecutable para una VM en particular, podríamos quedarnos con el código intermedio si somos capaces de interpretarlo. Este código intermedio suele ser similar al código ensamblador de un procesador real (habitualmente, mucho más simple) solo que adaptado a una máquina virtual determinada.

Afortunadamente, este código intermedio puede ser descompilado con mejores resultados (nuevamente, si este no ha sido ofuscado) que la descompilación de código nativo.

7.1. DESENSAMBLADO

Como ya hemos visto anteriormente, el desensamblado de código binario es relativamente fácil de conseguir. Existen multitud de herramientas disponibles para todo tipo de binarios para casi todas las arquitecturas y sistemas operativos. En especial son abundantes y variadas en Windows y derivados de UNIX.

Una herramienta versátil para desensamblar y prácticamente en toda distribución Linux es **objdump**⁶¹. Ya la vimos en un capítulo anterior. También podemos añadir otras herramientas que nos vendrán bien en la tarea de desensamblado: **readelf**, **nm** o la muy versatil suite **radare2**⁶².

Para sistemas Linux o UNIX son más que suficiente. Como es habitual, en estos sistemas se trabaja con asiduidad desde la consola o terminal virtual. Si preferimos un interfaz gráfico, podemos agregar **Cutter**⁶³ a radare2. Un proyecto de código abierto que ofrece un completo frontend con gran capacidad visual.

En sistemas tanto Microsoft Windows como Linux o OSX tenemos **IDAPro** (que posee una versión gratuita, aunque con funcionalidad limitada) y **Ghidra**⁶⁴ como herramientas multiuso. Es decir, poseen capacidad para procesar diferentes tipos de binarios y librerías, arquitecturas, depuran, desensamblan y también descompilan; es decir, son capaces de transformar el código ensamblador en una aproximación en C para mejorar la lectura de este (aunque hay quienes prefieren la lectura directamente en ensamblador)

A pesar de ser más un depurador, **x64dbg**⁶⁵ puede servirnos en Windows de herramienta para desensamblar, producir gráficos de funciones, anotaciones, etc. Posee un completo ecosistema de plugins con los que extender su funcionalidad e interoperacionalidad con otras herramientas.

⁶¹ <https://sourceware.org/binutils/docs/binutils/objdump.html>

⁶² <https://www.radare.org/n/>

⁶³ <https://cutter.re/>

⁶⁴ <https://ghidra-sre.org/>

⁶⁵ <https://github.com/x64dbg/x64dbg>

7.1.1. HERRAMIENTAS VISUALES DE DESENSAMBLADO

Casi todas las herramientas de este tipo son similares, aunque con pequeñas diferencias que las hacen más útiles según el objetivo de nuestro análisis. Veamos, por ejemplo, una sesión de desensamblado en Ghidra. Las funcionalidades que veremos suelen estar disponibles en casi todos los programas de este tipo:

The screenshot shows the Ghidra interface with the following windows:

- Program Trees**: Shows the file structure of 'a7.out' with sections like .bss, .data, .got, .dynamic, .fini_array, .init_array, .eh.frame, .eh.frame_hdr, .rodata, .fini, .text, .plt.got, .plt, .init, and .rela.plt.
- Listing: a7.out**: Displays assembly code for several functions. One function shown is:

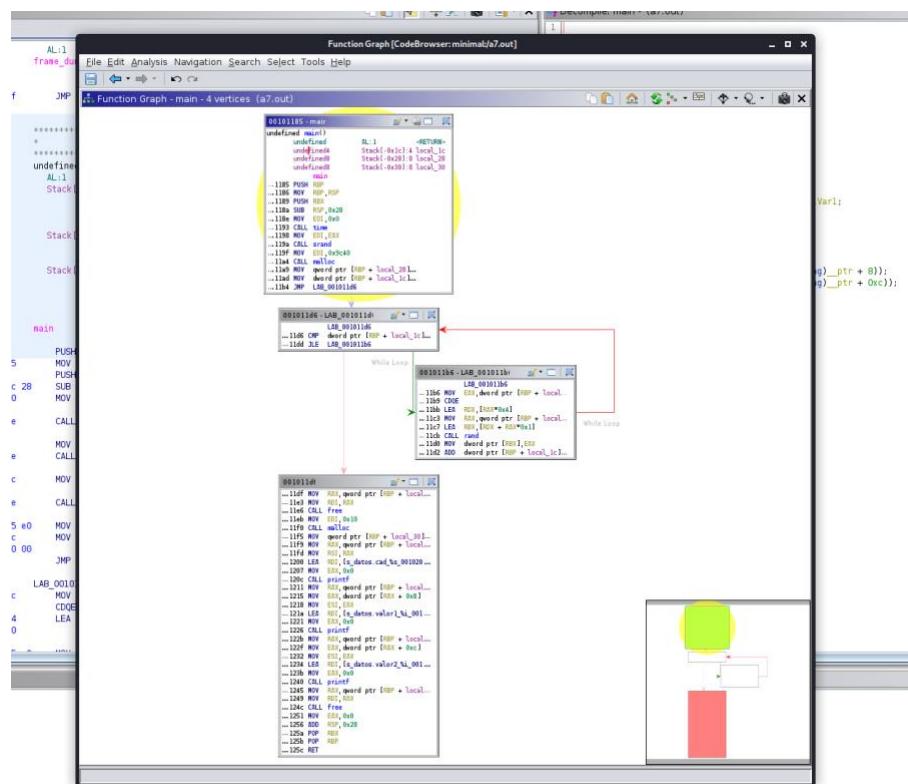

```

      2f 00 00 -- Flow.Override: CALL_RETURN (COMPUTED_CALL_TERMINATOR)
00101046 68 01 00 PUSH 0x1
0010104b 00 00 JMP LAB_00101020

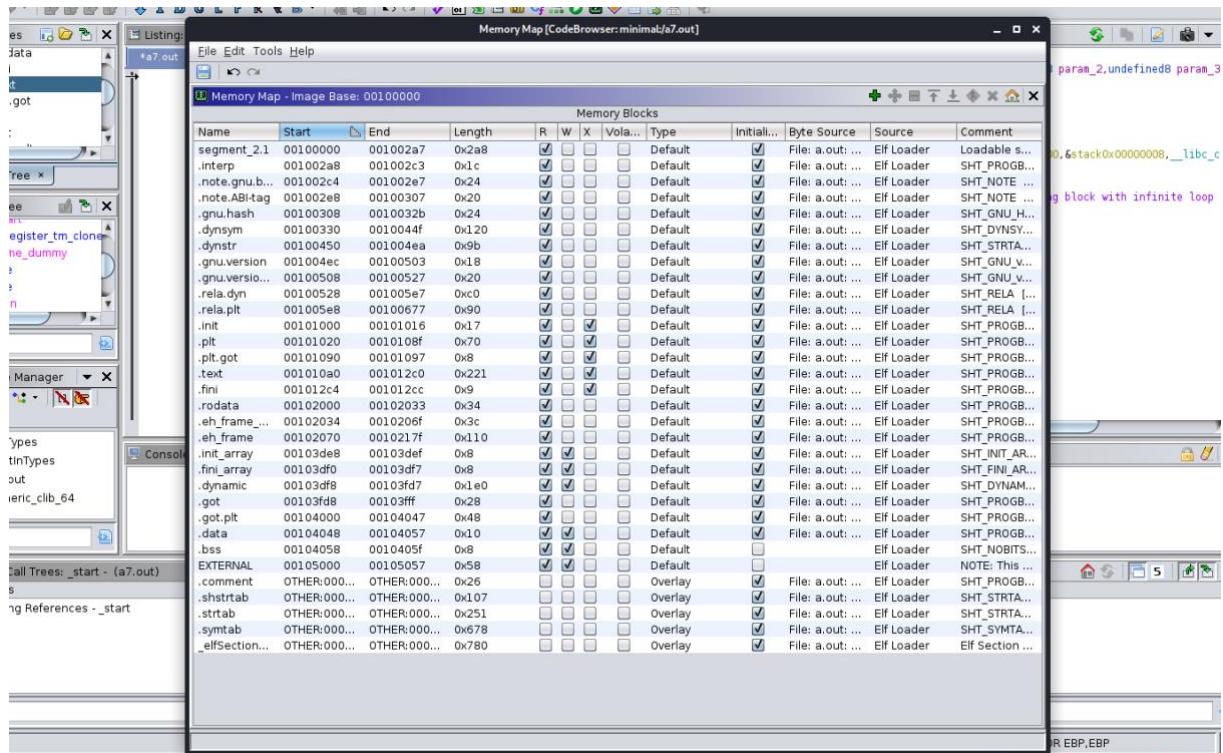
*****
* THUNK FUNCTION
*****
thunk void srand(uint __seed)
    Thunked-Function: srand
    <VOID> <RETURN>
    EDI:4 __seed
    srand
    JNP qword ptr [->srand] XREF[1]: main:0010119a(c) void srand(uint __seed)

00101050 ff 25 d2 2f 00 00 -- Flow.Override: CALL_RETURN (COMPUTED_CALL_TERMINATOR)
00101056 68 02 00 PUSH 0x2
0010105b e9 c0 ff ff ff JMP LAB_00101020
      
```
- Symbol Tree**: Shows symbols such as free, main, malloc, rand, printf, rand, register_tm_clones, srand, and time.
- Data Type Manager**: Shows data types including BuiltInTypes, a7.out, and generic_clib_64.

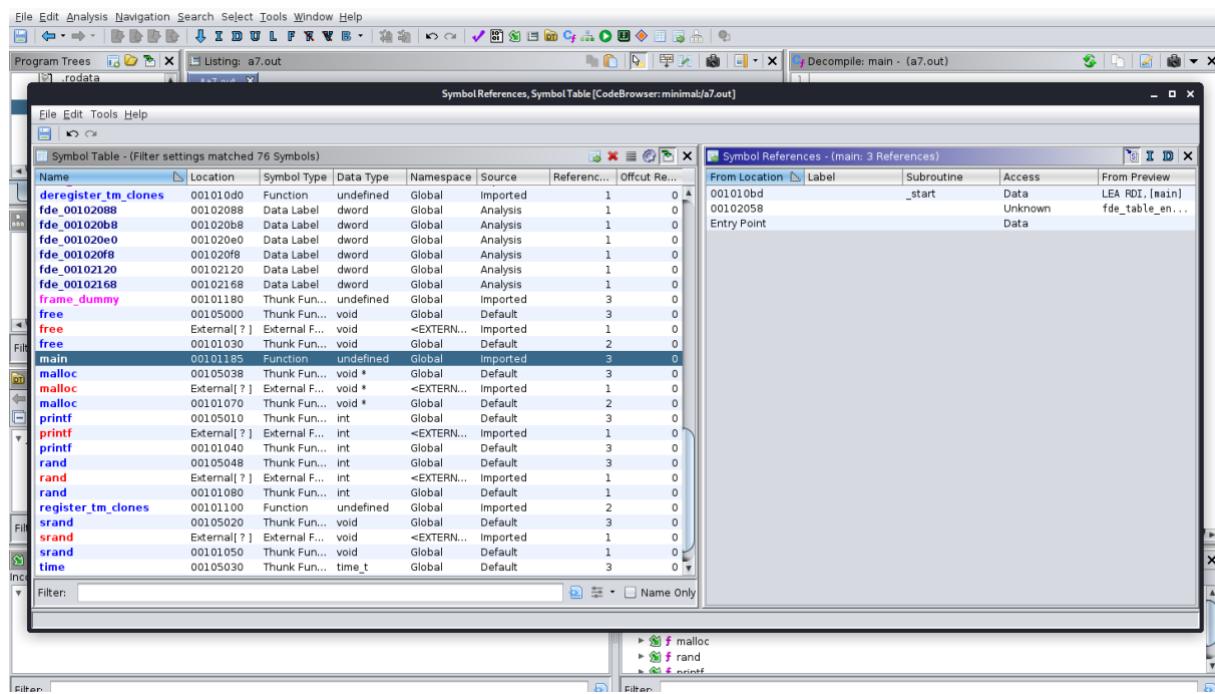
A partir del desensamblado, Ghidra puede construirnos un árbol de la función actual que estemos estudiando (menú Window -> Function Graph):



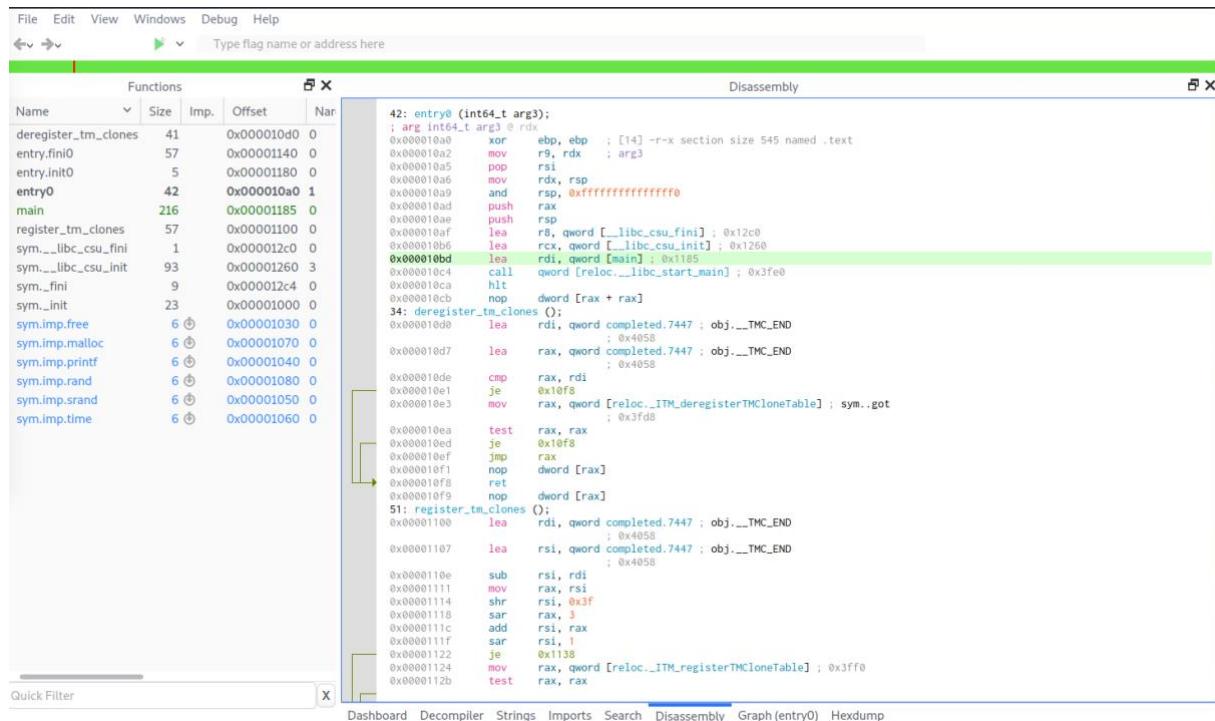
También podremos obtener un mapa de la memoria, donde los distintos segmentos de un ejecutable son ubicados para dar vida al proceso (menú Window -> Memory Map):



Otra opción útil es poder obtener una referencia cruzada de las funciones y variables, es decir, donde se está llamando una función determinada, donde se declara, etc (Menu -> Symbol Reference):



Cutter, el frontend para radare2 no se queda corto en opciones. Como hemos comentado, IDAPro, Ghidra o Cutter (radare2) poseen funcionalidades bastante parecidas. Solo nuestra experiencia usándolos nos puede ayudar a profundizar en uno o algunos de ellos.



Hemos de comentar aquí, que estas herramientas no se limitan, por supuesto, a desensamblar un

ejecutable y navegar por el código. Son también (y casi más importante) autenticos estudios de análisis multipropósito que nos permitirán depurar mientras analizamos e incluso emular la ejecución sin necesidad de crear un proceso real en la máquina.

7.1.2. HERRAMIENTAS DE CONSOLA PARA DESENSAMBLAR

Más modestas y centradas en la tarea de desensamblar, tenemos las herramientas de consola. Algunas ya las hemos comentado: objdump, readelf, nm u obtool, etc.

El funcionamiento básico es similar en casi todas ellas, variando en todo caso las banderas (flags) de opciones.

En la captura vemos el desensamblado parcial de un programa con “objdump”. En concreto, de la función “main”:

```
→ ~ objdump --disassemble=main a.out
a.out:   file format elf64-x86-64

Disassembly of section .init:
Disassembly of section .plt:
Disassembly of section .plt.got:
Disassembly of section .text:
0000000000000185 <main>:
1185: 55          push %rbp
1186: 48 89 e5    mov %rsp,%rbp
1189: 53          push %rbx
118a: 48 83 ec 28 sub $0x28,%rsp
118e: bf 00 00 00  mov $0x0,%edi
1193: e8 c8 fe ff  callq 1060 <time@plt>
1198: 89 c7        mov %eax,%edi
119a: eb b1 fe ff  callq 1059 <>rand@plt>
119f: b8 4c 00 00  mov $0x4c0,%edi
11a4: e8 c7 fe ff  callq 1070 <malloc@plt>
11a9: 48 8d 45 e0  mov %rdi,%rbp
11ad: c7 45 ec 00 00 00  movl $0x0,-0x1c(%rbp)
11b4: eb 20        jmp 11d6 <main+0x15>
11b6: b8 45 ec    mov $0x14(%rbp),%eax
11b9: 48 98        cltq
11bb: 48 8d 14 85 00 00 00  lea 0x0(%rax,4),%rdx
11c2: 00          xorl %eax,%eax
11c3: 48 8b 45 e0  mov -0x20(%rbp),%rax
11c7: 48 8d 1c 02  lea (%rdx,%rax,1),%rbx
11cb: e8 b6 fe ff  callq 1080 <rand@plt>
11d0: 89 03        mov %eax,%rbx
11d2: 83 45 ec 01  addl $0x1,-0x14(%rbp)
11d6: 81 7d ec f7 27 00 00  cmpl $0x270f,-0x14(%rbp)
11dd: 7e d7        jle 11b6 <main+0x31>
11df: 48 8b 45 e0  mov -0x20(%rbp),%rax
11e3: 48 89 c7        mov %rax,%rdi
11e6: e8 45 ec ff ff  callq 1030 <free@plt>
11eb: bf 10 00 00 00  mov $0x10,%edi
11f0: e8 7b fe ff  callq 1070 <malloc@plt>
11f5: 48 89 45 d8  mov %rax,-0x28(%rbp)
11f9: 48 8b 45 d8  mov -0x28(%rbp),%rax
11fd: 48 89 c6        mov %rax,%rsi
1200: 48 8d 3d fd 0d 00 00  lea 0xdffd(%rsi),%rdi      # 2004 <_IO_stdin_used+0x4>
1207: b8 00 00 00 00  mov $0x0,%eax
120c: e8 2f fe ff ff  callq 1040 <printf@plt>
```

La opción dada a objdump es:

```
objdump --disassemble=main a.out
```

Dado que cumple con la filosofía UNIX, objdump no posee cualidades de navegación así que deberemos utilizar una tubería (pipe) para o bien volcar la salida a un archivo o hacia la herramienta “less”:

```
objdump --disassemble=main a.out | less
objdump --disassemble=main a.out > volcado.txt
```

Como casi todas las herramientas de desensamblado en Linux o derivados de UNIX, la sintaxis por defecto es AT&T. Es fácil indicar a objdump que la cambie a Intel si estamos más acostumbrados a

esta:

```
objdump --disassemble=main -M intel a.out
```

```

0000000000001185 <main>:
1185:    55                      push   rbp
1186:    48 89 e5                mov    rbp,rbp
1189:    53                      push   rbx
118a:    48 83 ec 28              sub    rbp,0x28
118e:    bf 00 00 00 00          mov    edi,0x0 or greater
1193:    e8 c8 ff ff ff          call   1060 <timeopl>
1198:    89 c7                  mov    edi,eax
119a:    e8 b1 fe ff ff          call   1050 <r srand@plt>
119f:    bf 40 9c 00 00          mov    edi,0xc40
11a4:    e8 c7 fe ff ff          call   1070 <malloc@plt>
11a9:    48 89 45 e0              mov    QWORD PTR [rbp-0x20],rax
11ad:    c7 45 ec 00 00 00 00      mov    DWORD PTR [rbp-0x14],0x0
11b4:    eb 20                  jmp   11de <main+0x51>
11b6:    8b 45 ec                mov    eax,DWORD PTR [rbp-0x14]
11b9:    48 98                  cdqe
11bc:    48 8d 14 85 00 00 00      lea    rdx,[rax*4+0x0]
11c2:    00
11c3:    48 bb 45 e0              mov    rax,QWORD PTR [rbp-0x20]
11c7:    48 8d 1c 02              lebx  [rdx-rax*1]
11cb:    e8 b0 fe ff ff          call   1080 <r rand@plt>
11d0:    89 03                  mov    DWORD PTR [rbx],eax
11d2:    83 45 ec 01              add    DWORD PTR [rbp-0x14],0x1
11d6:    81 7d 0f 27 00 00      cmp    DWORD PTR [rbp-0x14],0x27f0
11dd:    7e d7                  jle   11be <main+0x31>

```

Respecto a la sintaxis del ensamblador, merece la pena que dediquemos un momento a comentar algunos aspectos:

7.1.3. COMPARATIVA SINTAXIS AT&T VS INTEL

Como hemos visto, el código desensamblado es una traducción directa del código objeto a código ensamblador de esa plataforma y arquitectura concreta. Es decir, el código objeto de una plataforma Intel no puede desensamblarse a una ARM (con confundir con la compilación cruzada).

Además, en algunos casos, el código saldrá con una sintaxis determinada. Por ejemplo, podemos desensamblar un binario para la plataforma x86 de Intel y elegir la sintaxis Intel o AT&T. La primera es más típica o usual en analistas de sistemas Windows, mientras que la sintaxis de AT&T es más común en el mundillo UNIX o Linux.

Observemos un ejemplo. Tenemos un código en ensamblador, lo compilamos y producimos un ejecutable. Vamos a desensamblarlo en las dos sintaxis: AT&T e Intel:

```
→ Desktop objdump -d a.out
a.out:      file format elf64-x86-64

Disassembly of section .text:
0000000000401000 <_start>:
401000: b8 01 00 00 00    mov    $0x1,%eax
401005: bf 01 00 00 00    mov    $0x1,%edi
40100a: 48 be 20 40 00 00  movabs $0x402000,%rsi
401011: 00 00 00          mov    $0x0,%rax
401014: ba 0d 00 00 00    mov    $0xd,%edx
401019: 0f 05             syscall
40101b: b8 3c 00 00 00    mov    $0x3c,%eax
401020: 48 31 ff          xor    %rdi,%rdi
401023: 0f 05             syscall

→ Desktop ┌
→ Desktop objdump -d a.out -M intel-mnemonic
a.out:      file format elf64-x86-64

Disassembly of section .text:
0000000000401000 <_start>:
401000: b8 01 00 00 00    mov    eax,0x1
401005: bf 01 00 00 00    mov    edi,0x1
40100a: 48 be 20 40 00 00  movabs rsi,0x402000
401011: 00 00 00          mov    $0x0,%rax
401014: ba 0d 00 00 00    mov    edx,0xd
401019: 0f 05             syscall
40101b: b8 3c 00 00 00    mov    eax,0x3c
401020: 48 31 ff          xor    rdi,rdi
401023: 0f 05             syscall

→ Desktop ┌
```

El programa es exactamente el mismo, pero la sintaxis varía entre una y otra.

Por ejemplo, las instrucciones en AT&T pueden llevar sufijos que indican el tamaño de la palabra que la instrucción está manejando. También es muy característico de la sintaxis AT&T el uso del carácter \$ cuando se indica una dirección y el carácter % cuando se referencia un registro.

Hay que tener cuidado especial en la posición de los operadores que van tras una instrucción. Por ejemplo, fíjemonos en la instrucción siguiente:

MOV EAX, 0x1	← Intel
mov \$0x1, %eax	← AT&T

Como vemos, el registro destino va el primero en Intel y el segundo (invierte la posición) en AT&T. Es bastante común cuando uno se acostumbra al uso de una sintaxis confundir la posición en la lectura de la otra.

Esto se entiende mejor cuando se dice como:

MOV EAX, 0x1	← Intel	“Mueve a EAX el valor 0x01”
mov \$0x1, %eax	← AT&T	“Mueve el valor 0x01 a %eax”

Fíjate, como habíamos dicho, la diferencia en la mención a los registros (el % en AT&T) y las constantes numéricas (el % en el valor 0x01).

Consejo: Si vais a trabajar más o preferís trabajar sobre sistemas Windows, aprended a manejarlos con soltura en la sintaxis Intel. Elegid la sintaxis AT&T si optáis más por sistemas UNIX y derivados. En cualquier caso, es relativamente fácil convertir una sintaxis en la otra con las herramientas disponibles.

7.2. DECOMPILEADO

El decompilado es una fase más, en la cual se pretende reconstruir el código original mediante una traducción de un lenguaje de bajo nivel (típicamente, ensamblador) hacia un lenguaje de alto nivel.

Nótese que el decompilado no está restringido a solo las plataformas nativas, sino que es posible obtener el código intermedio de una máquina virtual y decompilarlo hacia un lenguaje de alto nivel. El ejemplo más claro es Java.

Los decompiladores suelen ser programas complicados, en el sentido de difíciles de programar debido a la gran cantidad de posibilidades e interpretaciones que se pueden derivar del análisis del código ensamblador.

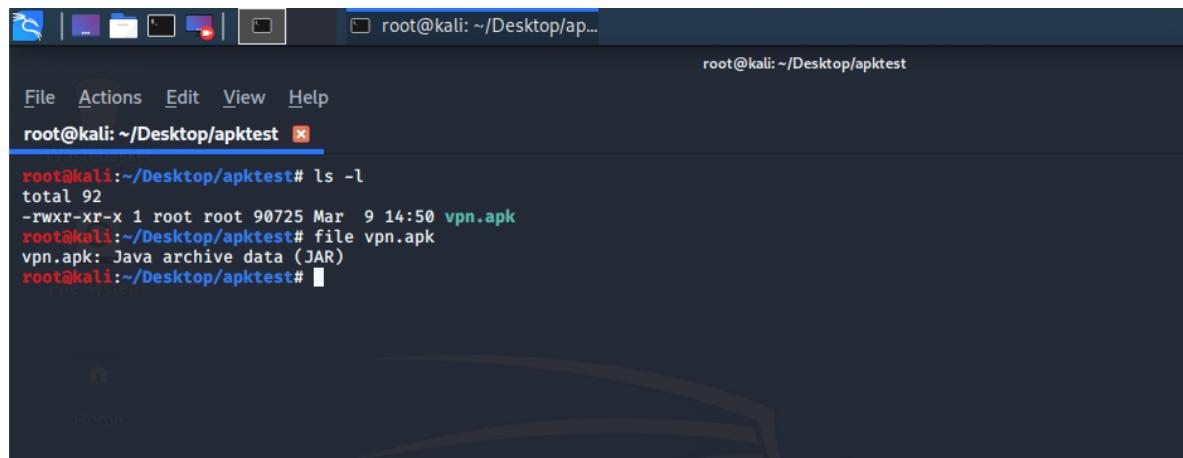
Sin duda, la mayor de las dificultades es la de producir código de decompilado de archivos ejecutables o bibliotecas que no portan información sobre los símbolos (nombres y tipos), que habitualmente sí encontramos en los programas cuando han sido compilados con esta información. En particular, los ejecutables nativos.

En lenguajes de programación que no posee equivalente a un ejecutable nativo la decompilación ofrece mejores resultados. El código intermedio de las máquinas virtuales e interpretadores ya ofrece bastante información como para invertir el proceso de traducción y esto se nota en los resultados.

7.2.1. UN EJEMPLO PRÁCTICO DE DECOMPILACIÓN

Vamos a ver un proceso práctico efectuado sobre un APK (una aplicación para Android). Usaremos la herramienta de desensamblado smali⁶⁶.

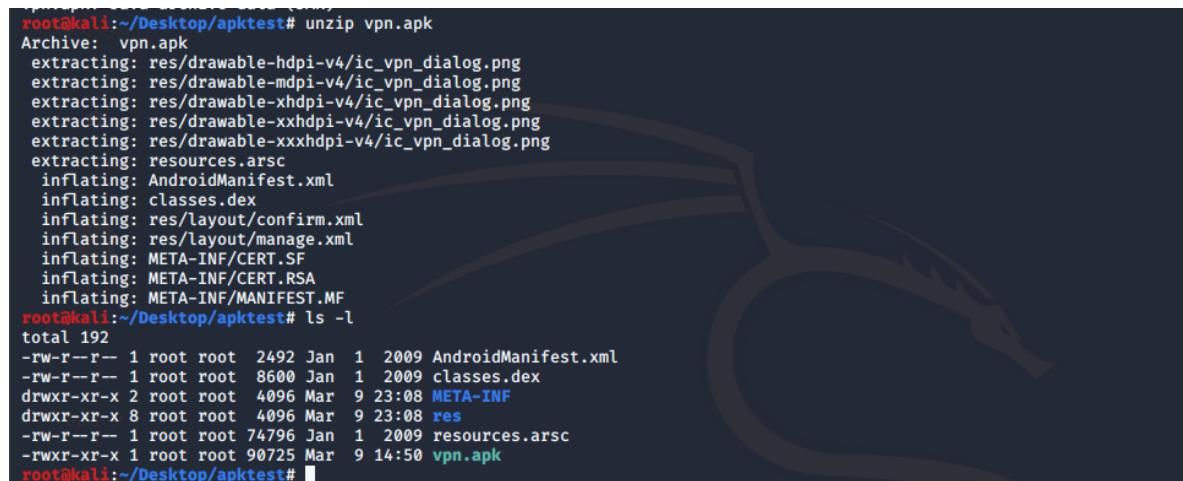
Supongamos que tenemos un APK cualquiera que queremos investigar:



```
root@kali: ~/Desktop/apktest# ls -l
total 92
-rwxr-xr-x 1 root root 90725 Mar  9 14:50 vpn.apk
root@kali:~/Desktop/apktest# file vpn.apk
vpn.apk: Java archive data (JAR)
root@kali:~/Desktop/apktest#
```

El formato APK es simplemente un archivo comprimido, es decir, podríamos procesarlo como este tipo de archivo con un descomprimidor y obtener su contenido, tal que así:

unzip vpn.apk



```
root@kali:~/Desktop/apktest# unzip vpn.apk
Archive: vpn.apk
extracting: res/drawable-hdpi-v4/ic_vpn_dialog.png
extracting: res/drawable-mdpi-v4/ic_vpn_dialog.png
extracting: res/drawable-xhdpi-v4/ic_vpn_dialog.png
extracting: res/drawable-xxhdpi-v4/ic_vpn_dialog.png
extracting: res/drawable-xxxhdpi-v4/ic_vpn_dialog.png
extracting: resources.arsc
  inflating: AndroidManifest.xml
  inflating: classes.dex
  inflating: res/layout/confirm.xml
  inflating: res/layout/manage.xml
  inflating: META-INF/CERT.SF
  inflating: META-INF/CERT.RSA
  inflating: META-INF/MANIFEST.MF
root@kali:~/Desktop/apktest# ls -l
total 192
-rw-r--r-- 1 root root 2492 Jan  1  2009 AndroidManifest.xml
-rw-r--r-- 1 root root 8600 Jan  1  2009 classes.dex
drwxr-xr-x 2 root root 4096 Mar  9 23:08 META-INF
drwxr-xr-x 8 root root 4096 Mar  9 23:08 res
-rw-r--r-- 1 root root 74796 Jan  1  2009 resources.arsc
-rw-r--r-x 1 root root 90725 Mar  9 14:50 vpn.apk
root@kali:~/Desktop/apktest#
```

En concreto, el código para la máquina virtual Dalvik (o Android ART) reside en el archivo 'classes.dex'. No podríamos ver directamente su contenido, ya que se encuentra en un formato binario:

⁶⁶ <https://github.com/JesusFreke/smali>



head classes.dex

Ahora bien, usando smali podremos procesar dicho archivo y desensamblar su contenido. Para ello emplearemos (forma parte de la suite de smali) la herramienta ‘baksmali’:

```
baksmali d /root/Desktop/apktest/vpn.apk
```

La salida de baksmali la encontraremos en la ruta /usr/share/smali/out, en concreto, la misma estructura de directorios, pero ya con las clases desensambladas:

tree /usr/share/smali/out/

```
0 directories, 2 files
root@kali:~/Desktop/apktest# tree /usr/share/smali/out/
/usr/share/smali/out/
└── com
    └── android
        └── vpndialogs
            ├── ConfirmDialog.smali
            └── ManageDialog.smali

3 directories, 2 files
root@kali:~/Desktop/apktest#
```

Si ahora abrimos uno de los archivos con un editor de textos, podremos leer código **smali**, una representación aproximada de lo que ejecutaría la máquina virtual Dalvik o Android ART:

```
root@kali: ~/Desktop/apktest ✘
.class public Lcom/android/vpndialogs/ManageDialog;
.super Lcom/android/internal/app/AlertActivity;
.source "ManageDialog.java"

# interfaces
implements Landroid/content/DialogInterface$OnClickListener;
implements Landroid/os/Handler$Callback;

# instance fields
.field private mConfig:Lcom/android/internal/net/VpnConfig;
.field private mDataReceived:Landroid/widget/TextView;
.field private mDataRowsHidden:Z;
.field private mDataTransmitted:Landroid/widget/TextView;
.field private mDuration:Landroid/widget/TextView;
.field private mHandler:Landroid/os/Handler;
.field private mService:Landroid/net/IConnectivityManager;

# direct methods
.method public constructor <init>()
    .registers 1

    .prologue
    .line 38
    invoke-direct {p0}, Lcom/android/internal/app/AlertActivity;→<init>()

    return-void
.end method

.method private getNumbers()[Ljava/lang/String;
    .registers 10

    .prologue
    .line 172
    const/4 v2, 0x0

    .line 175
    .local v2, "in":Ljava/io/DataInputStream;
    :try_start_1
    new-instance v3, Ljava/io/DataInputStream;

    new-instance v7, Ljava/io/FileInputStream;
    const-string/jumbo v8, "/proc/net/dev"
    invoke-direct {v7, v8}, Ljava/io/FileInputStream;→<init>(&Ljava/lang/String;)
"/usr/share/smali/out/com/android/vpndialogs/ManageDialog.smali" 899L, 24916C
12,0-1      Top
```

Ahora podremos quedarnos con dicho código e intentar analizar la aplicación a dicho nivel o pasar a la siguiente fase y realizar la **decompilación**.

Para obtener el decompilado de un APK lo que hacemos es apoyarnos en herramientas que traduzcan el bytecode para Dalvik en bytecode para la JVM (Java Virtual Machine) y a partir de ahí, producir código fuente en lenguaje Java.

Utilizar este método es sencillo con, por ejemplo, dex2jar. Tan solo tenemos que indicar la localización del archivo ‘classes.dex’ y trasladará dicho bytecode para Android en bytecode para la JVM:

```
d2j-dex2jar classes.dex
```

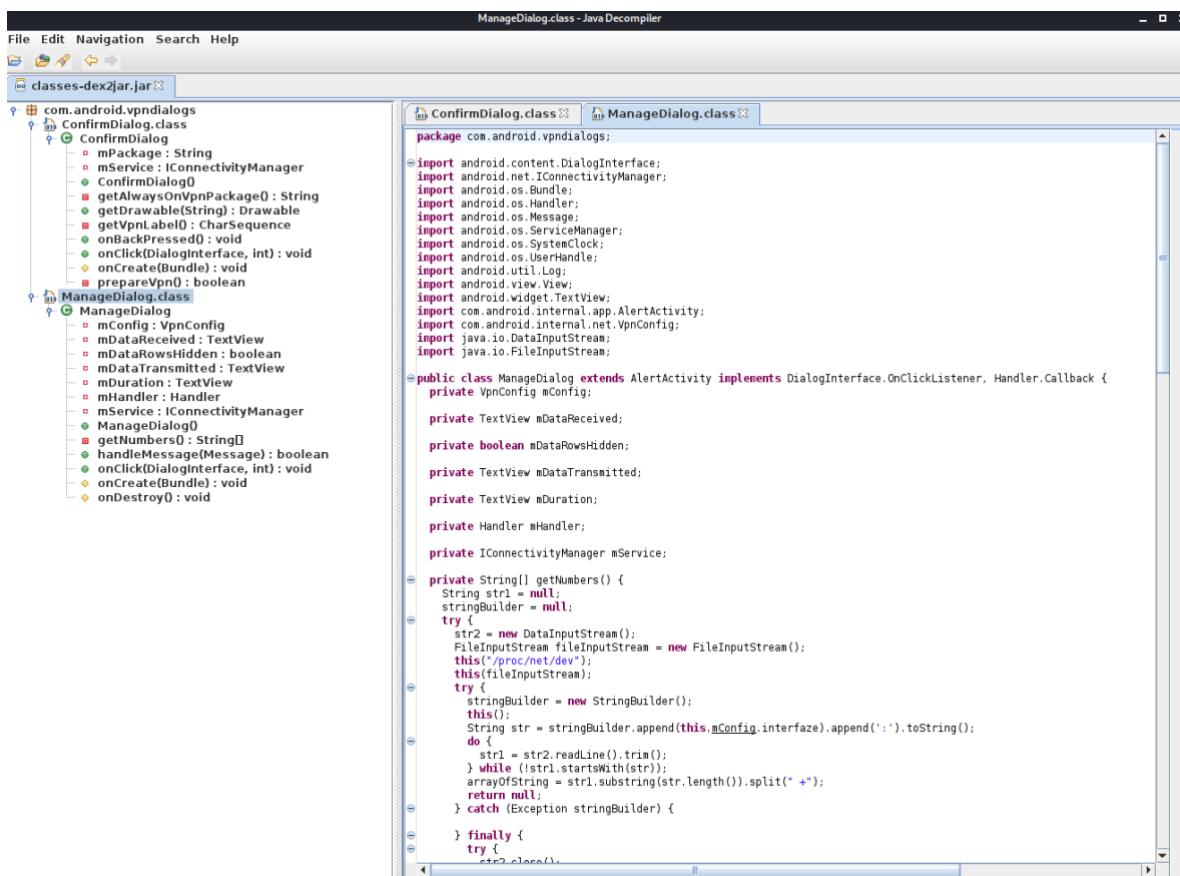
```

File Actions Edit View Help
root@kali:~/Desktop/apktest x
root@kali:~/Desktop/apktest# d2j-dex2jar classes.dex
Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSettings=on -Dswing.aatext=true
dex2jar classes.dex → ./classes-dex2jar.jar
root@kali:~/Desktop/apktest# ls -l
total 200
-rw-r--r-- 1 root root 2492 Jan 1 2009 AndroidManifest.xml
-rw-r--r-- 1 root root 8600 Jan 1 2009 classes.dex
-rw----- 1 root root 5429 Mar 9 23:38 classes-dex2jar.jar
drwxr-xr-x 2 root root 4096 Mar 9 23:08 META-INF
drwxr-xr-x 8 root root 4096 Mar 9 23:08 res
-rw-r--r-- 1 root root 74796 Jan 1 2009 resources.arsc
-rwrxr-xr-x 1 root root 90725 Mar 9 14:50 vpn.apk
root@kali:~/Desktop/apktest#

```

Y ahora podremos descompilar el jar generado por dex2jar y visualizarlo en jd-gui:

jd-gui classes-dex2jar.jar



Como podemos observar, resulta mucho más legible y comprensible obtener el código fuente en un lenguaje de alto nivel que trabajar con un ensamblador.

7.2.2. DECOMPILEDO DE UN EJECUTABLE BINARIO EN GHIDRA

Como vimos anteriormente, Ghidra es una herramienta muy versátil en cuanto a funcionalidad tanto para desensamblar como decompilar. Antes, en la sección anterior lo vimos en acción desensamblando un binario, ahora veremos la captura completa.

Observad, en la zona derecha, como además del código ensamblador de la función “main”, el decompilador de Ghidra es capaz de proporcionarnos un código C que puede leerse e interpretarse con bastante claridad:

```

Decompile: main - (a7.out)
1 undefinedB main(void)
2 {
3     int iVar1;
4     time_t tVar2;
5     void *__ptr;
6     int local_1c;
7
8     tVar1 = time((time_t *)0x0);
9     srand((uint)tVar2);
10    __ptr = malloc(40000);
11    local_1c = 0;
12    while (local_1c < 10000) {
13        iVar1 = rand();
14        *(int)((long)local_1c * 4 + (long)__ptr) = iVar1;
15        local_1c = local_1c + 1;
16
17        free(__ptr);
18        __ptr = malloc(0x10);
19        printf("datos.cad %s\n", __ptr);
20        printf("datos.valor1 %i\n", (ulong)*(uint*)((long)__ptr + 8));
21        printf("datos.valor2 %i\n", (ulong)*(uint*)((long)__ptr + 0xc));
22        free(__ptr);
23        return 0;
24    }
25
26
27
***** FUNCTION *****
undefined undefined4
AL[1] undefined main()
<RETURN> Stack[-0x1c]:4 local_1c
XREF[4]: 0010118d(R),
001011b6(R),
00101142(R),
00101146(R)
undefined8 Stack[-0x28]:8 local_28
XREF[3]: 001011a9(R),
001011c3(R),
001011d4(R),
001011e5(R),
001011f9(R),
00101211(R),
0010122b(R),
00101245(R)
undefined8 Stack[-0x30]:8 local_30
XREF[5]: 001011a5(R),
001011b5(R),
001011c5(R),
001011d5(R),
001011e5(R),
001011f5(R),
00101211(R),
0010122b(R),
00101245(R)
main
XREF[3]: Entry Point(*),
_start:001010bd(*), 00102058
00101185 55 PUSH RBP
00101186 48 89 e5 MOV RBP,RSP

```

Evidentemente, esto no siempre será así. De hecho, existen técnicas para dificultar tanto el desensamblado como el decompilado. Entre ellas, mediante la ofuscación de código que veremos en la siguiente sección.

7.3. CÓDIGO OFUSCADO

Hay muchas razones por las cuales el código fuente puede ofuscarse. La finalidad es siempre la misma: dificultar la legibilidad y compresión del código fuente. La ofuscación es usada para proteger de algún modo la propiedad intelectual inserta en el código.

Un ejemplo claro sería una aplicación para Android en la que se quiere impedir o dificultar el análisis inverso. Como hemos visto, resulta sencillo obtener un desensamblado y decompilado de una aplicación para este sistema. La única arma que posee el creador de estos programas es complicar ese proceso (en realidad, también existen técnicas para evitar el desensamblado, pero caen fuera del ámbito de esta asignatura) mediante la aplicación de ofuscado.



Código ofuscado en lenguaje C para el concurso anual de ofuscación IOCCC⁶⁷:

¿Qué hace la ofuscación?

La ofuscación transforma un objeto de texto de entrada (código fuente) en una salida diferente. En ese proceso, se realiza una serie de transformaciones que tienen por objeto complicar la lectura de dicha entrada.

Algunas transformaciones son: renombrar las variables, funciones, parámetros a otros objetos con nombres muy similares y difíciles de entender o leer, sustitución de cadenas de texto por generadores de texto codificado, reformato del código hacia una forma desestructurada, inserción de llamadas a función adicionales, etc.

¿Cómo combatir la ofuscación?

Es complicado. Va a depender de como se ofuscó y de que lenguaje se trata. A veces, es posible mejorar partes a través del reformato, renombrado de variables, descarte de partes de código muerto, etc.

Existen herramientas que pueden ayudar a combatir el código ofuscado, pero como analistas, no podremos esquivar totalmente el análisis manual del código. Nuestras armas aquí son la detección de formas: funciones, estructuras, condicionales, etc y la paciencia.

Lo recomendable es mantener el código en un sistema de versionado (Git y similares) e ir haciendo

⁶⁷ <https://www.ioccc.org/>

commits de cada modificación (así como ramas en caso necesario) para poder volver atrás en un momento dado.

Una de las partes esenciales del tratamiento de la ofuscación es que no cambiemos el sentido del código. Para ello, deberemos ser extremadamente cuidadosos donde y que tocamos. Un cambio en un condicional podría mutar la acción del código y perder el sentido original; lo cual, daría al traste con todo el análisis.

7.3.1. UN EJEMPLO PRÁCTICO DE DESOFUSCACIÓN

Vamos a tomar un sencillo trozo de código Javascript de ejemplo:

```

1 // Hola mundo
2 function hola() {
3   console.log("Hola Mundo!");
4 }
5 hola();

```

Lo pasamos por un ofuscador y obtenemos el resultado:

```
var _0x290d=['log','Hola\x20Mundo!'];(function(_0x321332,_0x290d63){var _0x192573=function(_0x20fba8){while(--_0x20fba8){_0x321332['push'](_0x321332['shift']());}};_0x192573(++_0x290d63);}(_0x290d,0x136));var _0x1925=function(_0x321332,_0x290d63){_0x321332=_0x321332-0x0;var _0x192573=_0x290d[_0x321332];return _0x192573;};function hola(){console[_0x1925('0x0')](_0x1925('0x1'));}hola();
```

Ahora pasamos un editor de texto para facilitar el ciclo modificación-prueba:

```

+ js cat ofuscate.js
var _0x290d=['log','Hola\x20Mundo!'];(function(_0x321332,_0x290d63){var _0x192573=function(_0x20fba8){while(--_0x20fba8){_0x321332['push'](_0x321332['shift']());}};_0x192573(++_0x290d63);}(_0x290d,0x136));var _0x1925=function(_0x321332,_0x290d63){_0x321332=_0x321332-0x0;var _0x192573=_0x290d[_0x321332];return _0x192573;};function hola(){console[_0x1925('0x0')](_0x1925('0x1'));}hola();
+ js node ofuscate.js
Hola Mundo!
+ js

```

En primer lugar, vamos a separar las definiciones desde el ámbito más global. Es decir, todo aquello que se está definiendo a nivel de unidad de compilación o archivo ¿Cómo definimos o encontramos esto? Vemos las asignaciones y donde se producen estas, además de los niveles de objetos del tipo {}, () y [] y las terminaciones “;”

```

1 var _0x290d=['log','Hola\x20Mundo!'];
2
3 (function(_0x321332,_0x290d63){var _0x192573=function(_0x20fba8){while(--_0x20fba8){_0x321332['push'](_0x321332['shift']());}};
4 _0x192573(++_0x290d63);}(_0x290d,0x136));
5
6 var _0x1925=function(_0x321332,_0x290d63){_0x321332=_0x321332-0x0;var _0x192573=_0x290d[_0x321332];return _0x192573;};
7
8 function hola(){console[_0x1925('0x0')](_0x1925('0x1'));}
9
10
11 hola();

```

Vamos a darle un mejor formato a esos objetos:

```

1 var _0x290d = ["log", "Hola\x20Mundo!"];
2
3 (function(_0x321332, _0x290d63) {
4     var _0x192573 = function(_0x20fba8) {
5         while (--_0x20fba8) {
6             _0x321332["push"](_0x321332["shift"]());
7         }
8     };
9     _0x192573(++_0x290d63);
10 })(_0x290d, 0x136);
11
12 var _0x1925 = function(_0x321332, _0x290d63) {
13     _0x321332 = _0x321332 - 0x0;
14     var _0x192573 = _0x290d[_0x321332];
15     return _0x192573;
16 };
17
18 function hola() {
19     console[_0x1925("0x0")](_0x1925("0x1"));
20 }
21
22 hola();

```

Fijémonos en la variable global declarada con el identificador `_0x290d`, solo tiene tres apariciones (dos si descontamos la declaración) y en ninguna de ellas es modificada, por lo que directamente lo podemos sustituir por su valor.

Ahora que tenemos una visual más clara, vamos a seguirle la pista a las funciones que son llamadas y qué están llamando estas a su vez:

```

1 var _0x290d = ["log", "Hola\x20Mundo!"];
2
3 (function(_0x321332, _0x290d63) {
4     var _0x192573 = function(_0x20fba8) {
5         while (--_0x20fba8) {
6             _0x321332["push"](_0x321332["shift"]());
7         }
8     };
9     _0x192573(++_0x290d63);
10 })(_0x290d, 0x136);
11
12 var _0x1925 = function(_0x321332, _0x290d63) {
13     _0x321332 = _0x321332 - 0x0;
14     var _0x192573 = _0x290d[_0x321332];
15     return _0x192573;
16 };
17
18 function hola() {
19     console[_0x1925("0x0")](_0x1925("0x1"));
20 }
21
22 hola();

```

Bien, esos dos bloques de código son llamados justo cuando se invoca o ejecuta el archivo. Ahora debemos ver que acoplamiento tienen esas funciones con su entorno. Es decir, debemos preguntarnos ¿A qué objeto fuera del ámbito de esas funciones están produciendo una mutación? ¿Modifican algo el entorno las llamadas a esas funciones?

Si nos fijamos en la primera función, no tiene nombre. Es una función anónima que se ejecuta justo después de su declaración con dos parámetros, el contenido de la variable `_0x290d` y el valor hexadecimal `0x136`; por cierto, este ofuscador cambia los nombres de variables y asigna nuevas variables con nombres que pretendidamente son asociados a valores hexadecimales.

Vamos a centrarnos en esta función. En primer lugar, vamos a cambiar los parámetros por su valor correspondiente el primero y el valor en decimal el segundo:

```

3 (function(array_log_holamundo, valor_numerico) {
4     var _0x192573 = function(_0x20fba8) {
5         while (--_0x20fba8) {
6             array_log_holamundo["push"]([array_log_holamundo["shift"]()]);
7         }
8     };
9     _0x192573(++valor_numerico);
0 })(["log", "Hola\x20Mundo!"], 310);

```

Por un lado, tenemos los valores que se están pasando por los parámetros y por otro, hemos cambiado los nombres de las variables en hexadecimal por otros, al menos, más descriptivos. Seguimos empleando el mismo sistema apuntando a la función definida y usada dentro de esta. Si nos fijamos en la definición de la variable interna `_0x192573` se trata de una función que toma un parámetro. De entrada, cambiamos el nombre de la función a otro más descriptivo:

```

2
3 (function(array_log_holamundo, valor_numerico) {
4     var funcion_interna = function(_0x20fba8) {
5         while (--_0x20fba8) {
6             array_log_holamundo["push"]([array_log_holamundo["shift"]()]);
7         }
8     };
9     funcion_interna(++valor_numerico);
0 })(["log", "Hola\x20Mundo!"], 310);
1

```

Vemos, por observación, que dicho parámetro es el valor numérico que ya se pasó por parámetro a su función externa y que se usa como condición en un bucle ‘while’. Le cambiamos el nombre, de nuevo, a algo más descriptivo:

```

2
3 (function(array_log_holamundo, valor_numerico) {
4     var funcion_interna = function(valor_a_interna) {
5         while (--valor_a_interna) {
6             array_log_holamundo["push"]([array_log_holamundo["shift"]()]);
7         }
8     };
9     funcion_interna(++valor_numerico);
10 })(["log", "Hola\x20Mundo!"], 310);
11

```

Por observación directa, vemos la función externa toma un array con dos cadenas y el valor decimal 310. Declara una función interna y le pasa como parámetro el valor decimal pero aumentado a una unidad, es decir, 311. No obstante, ese 311 volverá a ser usado como 310, porque justo en el ‘while’ volverá a disminuir su valor en exactamente una unidad antes de ser ejecutado.

Resumiendo, podemos eliminar la función interna y convertirla en código de la externa:

```

3 (function(array_log_holamundo, valor_numerico) {
4     while (valor_numerico--) {
5         array_log_holamundo["push"]([array_log_holamundo["shift"]()]);
6     }
7 })(["log", "Hola\x20Mundo!"], 310);
8

```

El lector atento habrá observado el cambio en el sentido del ‘—’ en el ‘while’. Esto es así porque antes de inyectar el parámetro en la función interna incrementaba en una unidad: 310 -> 311. Pero antes del test del bucle ‘while’, se decrementaba, luego el bucle se ejecutará 310 veces. Para compensar el ciclo añadido, simplemente hacemos que primero pase por el ‘while’ y posteriormente decremente el valor, de este modo añadirá una ejecución extra al final sin necesidad de incrementar el parámetro.

Como este tipo de trucos es bastante habitual, vamos a exponer este último razonamiento con código ilustrativo:

```

> f1 = function(a){while(--a){console.log(a);}}
[Function: f1]
> f2 = function(a){while(a--){console.log(a);}}
[Function: f2]
> let c1 = 10; let c2 = 10;
undefined
> f1(++c1)
10
9
8
7
6
5
4
3
2
1
undefined
> f2(c2)
9
8
7
6
5
4
3
2
1
0
undefined
> █

```

Esto se podrá hacer mientras que no se esté usando el valor en sí de la variable. En caso contrario si deberemos respetar su valor incrementado y el orden de decremento anterior. Como lo que se busca es ofuscar el número de repeticiones del bucle lo vemos de esta forma.

Ahora nos centramos en el cuerpo del bucle ‘while’ ¿Qué hace 310 veces?

```

4     while (valor_numerico--) {
5         array_log_holamundo["push"]([array_log_holamundo["shift"]()]);
6     }

```

Extraemos ese código, lo simplificamos y vemos que efectos tiene por separado. Para ello, añadimos un log simple para ver que ocurre dentro del bucle y como resulta mutada el array con las dos cadenas pasadas:

```
1 let f = function() {
2     let contador = 310;
3     let cadena = ["log", "Hola\x20Mundo!"];
4
5     while(contador--) {
6         cadena["push"](cadena["shift"]());
7         console.log(cadena);
8     }
9
10    console.log(cadena);
11 }
12
13 f();
```

Ejecutamos y comprobamos que ocurre:

Bien. Ahora sabemos que hace esa función: absolutamente nada que mute o modifique el valor inicial que teníamos en la variable global. Por completitud y curiosidad, el código que ejecuta es una versión retorcida de:

```
1 let f = function() {  
2     let contador = 310;  
3     let cadena = ["log", "Hola\x20Mundo!"];  
4  
5     while(contador--) {  
6         cadena.push(cadena.shift());  
7         console.log(cadena);  
8     }  
9  
10    console.log("FINAL: ");  
11    console.log(cadena);  
12  
13 }  
14  
15 f();
```

Cambia el tratamiento de las llamadas a métodos de un objeto. Algo muy particular del lenguaje Javascript.

Seguimos con nuestro código, borrando directamente el bloque de código de la función que hemos comprobado...no hace nada.

```

1 var _0x290d = ["log", "Hola\x20Mundo!"];
2
3 var _0x1925 = function(_0x321332, _0x290d63) {
4   _0x321332 = _0x321332 - 0x0;
5   var _0x192573 = _0x290d[_0x321332];
6   return _0x192573;
7 };
8
9 function hola() {
10   console[_0x1925("0x0")](_0x1925("0x1"));
11 }
12
13 hola();
```

Esa función que acabamos de eliminar se denomina **código muerto**. No porque no se procese, sino porque su ejecución o no, no interfiere con el objetivo del código. Solo está ahí para estorbar al análisis.

Como vemos, ahora esa variable solo es referenciada una única vez:

```

1 var _0x290d = ["log", "Hola\x20Mundo!"];
2
3 var _0x1925 = function(_0x321332, _0x290d63) {
4   _0x321332 = _0x321332 - 0x0;
5   var _0x192573 = _0x290d[_0x321332];
6   return _0x192573;
7 };
8
9 function hola() {
10   console[_0x1925("0x0")](_0x1925("0x1"));
11 }
12
13 hola();
```

Del mismo modo la función _0x1925, por lo que hacemos cambios para ir eliminando código y aclarando nombres:

```

1 function unica_funcion (_0x321332, _0x290d63) {
2   _0x321332 = _0x321332 - 0x0;
3   var _0x192573 = ["log", "Hola\x20Mundo!"][_0x321332];
4   return _0x192573;
5 };
6
7 function hola() {
8   console[unica_funcion("0x0")](unica_funcion("0x1"));
9 }
10
11 hola();
```

Vamos a ver algo curioso y que también resulta ser propio del lenguaje Javascript. Observemos la línea 2. Hace una sustracción del primer parámetro con 0x0. Es decir, resta literalmente cero al primer parámetro y asigna a el mismo el resultado. ¿Para qué restar cero?

Lo hace por la conversión de cadena a entero. Javascript promociona la cadena a número cuando observa una operación en la que no tiene sentido operar. Vamos a entenderlo con un pequeño ejemplo:

```
> "0x1" - 0x0
1
> "0x1" - "0x0"
1
> "0x1" + 0x0
'0x10'
> "0x1" + "0x0"
'0x10x0'
> █
```

Como vemos, este tipo de conversiones resultan poco intuitivas, sin embargo, poseen su sentido tal y como está definido el lenguaje y son explotadas como trucos para este tipo de ofuscaciones. Así pues, podemos, de nuevo, sustituir más código y eliminar partículas que no se usen.

De nuevo, el lector atento habrá percibido que el segundo parámetro de la función ahora nombrada ‘unica_funcion’ ni tan siquiera se utiliza:

```
function unica_funcion (_0x321332) {
    var _0x192573 = ["log", "Hola\x20Mundo!"][_0x321332];
    return _0x192573;
};

function hola() {
    console[unica_funcion(0x0)](unica_funcion(0x1));
}

hola();
```

Seguimos quitando código, por ejemplo, dado que no tiene más uso que retornar de la función, podemos deshacernos de la variable _0x192573 y ya de paso renombramos el único parámetro a ‘índice’, pues ese parece ser su cometido y convertimos los valores hexadecimales en decimales:

```

1 function unica_funcion (indice) {
2   return ["log", "Hola\x20Mundo!"] [indice];
3 }
4
5 function hola() {
6   console[unica_funcion(0)](unica_funcion(1));
7 }
8
9 hola();
```

Bastante más claro ¿verdad?

Vemos como en la línea 6 se vuelve a usar el truco de llamar a funciones internas por sus nombres de objeto, de decir se está haciendo lo siguiente:

console.log por console["log"]

La primera llamada nos devuelve “console.log” y la segunda “Hola Mundo!”. Sustituimos y aclaramos:

```

1 function hola() {
2   console["log"]("Hola\x20Mundo!");
3 }
4
5 hola();
```

Volvemos a sustituir por la llamada tradicional y además cambiamos el hexadecimal de la cadena por el carácter espacio original:

```

1 function hola() {
2   console.log("Hola Mundo!");
3 }
4
5 hola();
```

Resultado: volvemos al código original. No obstante, debemos recordar que en el día a día posiblemente no nos encontraremos el código como lo vemos ahí arriba sino como lo vemos aquí abajo, es decir, así:

```
var _0x290d=['log','Hola\x20Mundo!'](function(_0x321332,_0x290d63){var _0x192573=function(_0x20fba8){while(--_0x20fba8)_0x321332['push'](_0x321332['shift']());};_0x192573(++_0x290d63);}_0x290d,0x136);var _0x1925=function(_0x321332,_0x290d63){_0x321332=_0x321332-0x0;var _0x192573=_0x290d[_0x321332];return _0x192573;};function hola(){console[_0x1925('0x0')](_0x1925('0x1'));}hola();}
```

7.4. ¿POR QUÉ ES ÚTIL ESTA INFORMACIÓN?

Cuando no disponemos del código fuente, y será algo común cuando no analizamos proyectos de código abierto, debemos de ingenierías para obtener el máximo de claridad posible. Esto vendrá dado por nuestro conocimiento sobre el objeto que tenemos delante y las herramientas y nuestra destreza para manejarlas.

El objetivo es, como hemos dicho, obtener “claridad”. Existen tres técnicas a tener en cuenta: desensamblar, decompilar y desofuscar. Cada una de ellas supone un nivel de dureza a solventar.

Afortunadamente, no todas vienen superpuestas. Es decir, habrá veces que solo tendremos que desofuscar, otras obtener un decompilado de un lenguaje intermedio para una virtual. Todo dependerá de que tenemos delante.

7.5. ¿QUÉ TENGO QUE HABER APRENDIDO?

- Desensamblar un binario empleando alguna de las herramientas vistas.
- Cambiar la sintaxis del código producido para acomodarla a nuestras preferencias.
- Decompilar código ayudándonos de una herramienta para obtener algo cercano al código original.
- Entender que es el código ofuscado y aplicar técnicas para intentar mejorar su legibilidad.

8. ANÁLISIS DE CÓDIGO FUENTE EN LENGUAJES DE BAJO NIVEL

En este módulo nos vamos a dedicar al estudio y análisis del código en ensamblador x86 y x64.

Podría decirse, que el ensamblador es la lengua franca de la ingeniería inversa. Máxime de las mencionadas plataformas x86 y x64; aunque dado el auge de los dispositivos móviles, también lo ha hecho en la arquitectura ARM.

Históricamente, el ensamblador está muy ligado al proceso de ingeniería inversa debido sobre todo a que cuando se obtenía un programa, este, venía en formato binario y acompañado de diversas librerías ya compiladas. Hoy existen multitud de programas que no son binarios y su decompilado es relativamente sencillo si su código no ha sido ofuscado.

8.1. ENSAMBLADOR Y CÓDIGO OBJETO DESENSAMBLADO

En primer lugar, tenemos que hacer una distinción entre el código que produce un desensamblador y el que estamos viendo en un proyecto de código ensamblador. Aunque son lenguajes ensambladores, no son exactamente lo mismo.

Programar en lenguaje ensamblador es una tarea común. Aun más hoy en día en el que el Internet de las cosas y los microcontroladores han creado un autentico mercado y demanda, cada día más, programadores en los distintos ensambladores que existe.

De hecho, ni tan siquiera en algunas plataformas el lenguaje C es una opción. Debido a las restricciones de espacio de memoria y recursos computacionales, se debe programar directamente en ensamblador. Eso si, esto no significa que el ensamblador sea una replica del código que vemos cuando desensamblamos un ejecutable.

A pesar de ser el mismo juego de instrucciones, el ensamblador como lenguaje de programación suele poseer “ayudas” al programador, tales como macros, directivas, expresiones lógicas, pseudo instrucciones, etc. Ayudas que cuando ensamblamos el programa se convierten en algo más parecido

al código que vemos cuando realizamos el desensamblado.

En este capítulo haremos referencia a la sintaxis Intel, más usada en reversing y análisis de malware.

8.2. DETECCIÓN DE FUNCIONES

Uno de los primeros pasos en la lectura del código que obtenemos de un desensamblado es la detección de funciones. Afortunadamente, las herramientas con las que trabajamos ya hacen esa detección por nosotros. No obstante, en primer lugar, debemos reconocerlas por nosotros mismos y sin ayuda de automatismos. En segundo lugar, es posible que en algún momento el desensamblador no obre bien y tengamos que analizar un fragmento de forma manual.

Ya en el capítulo de la memoria vimos de forma extensiva como funcionaba la pila. Eso nos dará muchísimas pistas sobre la presencia de una función. De hecho, con ver PUSH al comienzo podremos empezar a otear el comienzo de una función y solo tenemos que ir buscando un LEAVE y RET o una serie de POP más abajo, en el código, para concluir que es el cuerpo de una función.

8.2.1. EL PRÓLOGO DE UNA FUNCIÓN

En ensamblador, una función necesita crear su marco de pila y, como hemos visto, una de las primeras cosas que hace es guardar en la pila la dirección del EBP anterior y poner en el registro EBP la dirección de la cima de la pila, guardada en el registro ESP.

A continuación, hará que la pila crezca (de nuevo, recordad, crece menguando la dirección) para hacer sitio a las variables locales para hacer su trabajo.

Salvo cambios en la convención de llamadas de un programa, el prólogo de una función tendrá el siguiente aspecto:

```
funcion_que_llama_a_funcion(int):
    push    ebp
    mov     ebp, esp
    sub     esp, 16
```

Como podemos apreciar, los primeros pasos de una función pasan por guardar el EBP y crear uno nuevo a partir de ESP; posteriormente, añadir el espacio necesario para trabajar.

Es posible, también, encontrar prólogos de funciones que guardan el contenido de los registros en la pila. Cuando esto sucede es porque va a operar con los registros en vez de direcciones de la pila (se optimiza la velocidad de ejecución), al final, cuando la función va a retornar, vuelve a reponer el valor en los registros con una serie de POPs.

8.2.2. EL EPÍLOGO DE UNA FUNCIÓN

El epílogo de una función es su preparación para retornar al lugar desde donde fue llamada. Esto lo hace de forma muy típica así:

```
mov    eax, DWORD PTR [ebp-8]
leave
ret
```

El mov hacia EAX es para poner ahí el valor de retorno de la función. LEAVE y RET fueron discutidos ya en la sección de la pila, en el capítulo dedicado a la memoria. En breve, LEAVE restaura el ESP de la pila moviendo el valor de EBP a este registro. De esta forma, se “limpia” el espacio utilizado en la pila. RET, hace una especie de POP de la dirección guardada para el retorno a EIP.

De nuevo lo explicado en el prólogo: podemos ver una consecución de POPs. Esto es debido, como hemos comentado, a la restauración del contexto que existía en los registros antes de que nuestra función fuese llamada.

Esto es especialmente cierto en sistemas de 64 bits (arquitectura Intel), donde se trabaja más con registros ya que esta arquitectura posee más registros que la arquitectura de 32 bits.

8.3. DETECCIÓN DE ESTRUCTURAS DE CONTROL DE FLUJO

Vamos a ver un ejemplo de estructura de iteración y dos de bifurcación para ver como se construyen en ensamblador estos bloques y los caracterizaremos.

8.3.1. ESTRUCTURAS ITERATIVAS

Con la iteración hay que tener muy clara una cosa: hay al menos un valor que va incrementando o decrementando su valor de forma lineal, una instrucción de comparación (CMP) y otra de salto. Ellas juntas parecen decir: “incremento el valor, miro si ese valor es igual al número de iteraciones que me piden, si lo es salto y dejo de iterar”.

Aunque con variaciones, ese es el quid de una iteración. En código de más alto nivel podemos encontrar diferentes estructuras, como hemos visto, pero en ensamblador, por normal, se sigue el patrón descrito; al menos en código desensamblado procedente de un lenguaje compilado.

Veamos un código de ejemplo:

```

1 #include <stdio.h>
2 #define LEN(x) (sizeof((x))/sizeof((x[0])))
3
4 int sum(int *a, int len) {
5     int suma = 0;
6     for (int i = 0; i < len ; ++i) {
7         suma += a[i];
8     }
9
10    return suma;
11}
12
13 int main()
14 {
15     int array[] = {1,2,3,4,5};
16
17     printf("%i\n", sum(array, LEN(array)));
18     return 0;
19 }

```



Como vemos, es muy sencillo. Tenemos una función, “sum”, que toma un puntero a un objeto del tipo “int” y un valor que será el número de iteraciones.

“main” llama a “sum” pasándole un puntero a un array (que es una variable local de “main”) de “int” y la longitud de este array; extraída de una macro que hemos definido.

Ahora veamos el código desensamblado de forma parcial, en concreto la función “sum”:

```

sum:
55
8049172 push    ebp
89 e5
8049173 mov     ebp,esp
83 ec 10
8049175 sub     esp,0x10
c7 45 fc 00 00 00 00
8049178 mov     DWORD PTR [ebp-0x4],0x0
c7 45 f8 00 00 00 00
804917f mov     DWORD PTR [ebp-0x8],0x0
eb 18
8049186 jmp     80491a0 <sum+0x2e>
8b 45 f8
8049188 mov     eax,WORD PTR [ebp-0x8]
8d 14 85 00 00 00 00
804918b lea     edx,[eax*4+0x0]
8b 45 08
8049192 mov     eax,WORD PTR [ebp+0x8]
01 d0
8049195 add     eax,edx
8b 00
8049197 mov     eax,WORD PTR [eax]
01 45 fc
8049199 add     WORD PTR [ebp-0x4],eax
83 45 f8 01
804919c add     WORD PTR [ebp-0x8],0x1
8b 45 f8
80491a0 mov     eax,WORD PTR [ebp-0x8]
3b 45 0c
80491a3 cmp     eax,WORD PTR [ebp+0xc]
7c e0
80491a6 jl     8049188 <sum+0x16>
8b 45 fc
80491a8 mov     eax,WORD PTR [ebp-0x4]
c9
80491ab leave
c3
80491ac ret
-----
```

Localicemos la variable que se incrementa, la comparación y el salto condicional:

```

804919c add     WORD PTR [ebp-0x8],0x1
8b 45 f8
80491a0 mov     eax,WORD PTR [ebp-0x8]
3b 45 0c
80491a3 cmp     eax,WORD PTR [ebp+0xc]
7c e0
80491a6 jl     8049188 <sum+0x16>
-----
```

El “add” suma uno a una variable local, la situada en **[EBP-0x8]** que resulta ser el “int x” del bucle “for”.

El **MOV** que vemos después hacia **EAX** es para preparar la comparación con “5”, que es el tamaño del array y es pasado como argumento en **[EBP+0xc]**.

El **JL** es un mnemotécnico que básicamente está diciendo “JUMP IF IS LESS”, cumpliendo así la condición del “for”: $i < len$, siendo “len” igual a 5.

Cuando se cumpla la condición, el bucle habrá finalizado las iteraciones y proseguirá por debajo de la instrucción JL, es decir:

```
80491a8  mov    eax,DWORD PTR [ebp-0x4]
c9
80491ab  leave
c3
80491ac  ret
main:
```

Esto lo que hace es, básicamente, dejar en **EAX** el resultado de la suma, que la función fue almacenando en su variable local “suma” situada en la pila en **[EBP-0x4]**.

Por curiosidad ¿Cómo actualiza “suma” o lo que es lo mismo **[EBP-0x4]**? Como habíamos visto si se cumplía la instrucción JL de la comparación (menor de 5) entonces saltaba a 0x8049188 que es:

```
8049188  mov    eax,DWORD PTR [ebp-0x8]
8d 14 85 00 00 00 00
804918b  lea    edx,[eax*4+0x0]
8b 45 08
8049192  mov    eax,DWORD PTR [ebp+0x8]
01 d0
8049195  add    eax,edx
8b 00
8049197  mov    eax,DWORD PTR [eax]
01 45 fc
8049199  add    DWORD PTR [ebp-0x4],eax
```

Mueve a **EAX** el valor que está en **[EBP-0x8]** que es la variable local ‘i’ que se declaró dentro del bucle “for”, que sabemos irá desde 0 a 4 (siempre menor que 5).

Ejecuta **LEA**, que significa **LOAD EFFECTIVE ADDRESS**. Lo que hace es calcular la expresión del segundo operador y tratarla como una dirección, entonces carga esa dirección en el registro **EDX**. Aquí se está usando para calcular el desplazamiento del puntero sobre el array. En este momento **EAX** contiene **[EBP-0x8]**, que es la variable local que posee ‘i’.

Es decir, si el primer elemento del array, el $a[0]$, está en la dirección 0x8000 (dirección hipotética) y cada entero son 4 bytes, el siguiente elemento del array, $a[1]$, estará en 0x8004. El otro, $a[2]$, en 0x8008...

Mueve a **EAX** el valor que hay en **[EBP+0x8]**, que es la dirección o puntero que se le pasó a la función “suma”.

Suma a **EAX** el valor del registro **EDX**. Es decir, el elemento ‘i’-esimo del array al que está apuntando **[EBP+0x8]**. Es como hacer el **a[i]** del código. **EAX** contiene la dirección de comienzo del array, mientras que **EDX** contiene el desplazamiento para llegar al siguiente elemento de la iteración.

Mueve a **EAX** el valor que hay en la dirección a la que apunta...precisamente **[EAX]**, que no es otro que el valor que almacena el array en esa dirección.

Finalmente, suma en **[EBP-0x4]**, que es la variable local “suma”, el valor que haya en **EAX**.

Dejamos esta tabla para que se entienda un poco mejor la relación parámetros y variables locales en la pila:

EBP + 0xC	5
EBP + 0x8	*a
EBP - 0x4	suma
EBP - 0x8	i

8.3.2. ESTRUCTURAS DE DECISIÓN

El caso switch es bastante obvio y presenta el siguiente aspecto:

```

3 int decidir(int valor) {
4     switch (valor) {
5         case 0: {
6             return 2;
7         }
8
9         case 1: {
10            return 4;
11        }
12
13         case 2: {
14             return 8;
15         }
16
17         case 3: {
18             return 16;
19         }
20     }
21 }
22
23 int main()
24 {
25     decidir(3);
26     return 0;
27 }
```

decidir:

```

55
8049162 push    ebp
89 e5
8049163 mov     ebp,esp
83 7d 08 03
8049165 cmp    DWORD PTR [ebp+0x8],0x3
74 35
8049169 je    80491a0 <decidir+0x3e>
83 7d 08 03
804916b cmp    DWORD PTR [ebp+0x8],0x3
7f 36
804916f jg    80491a7 <decidir+0x45>
83 7d 08 02
8049171 cmp    DWORD PTR [ebp+0x8],0x2
74 22
8049175 je    8049199 <decidir+0x37>
83 7d 08 02
8049177 cmp    DWORD PTR [ebp+0x8],0x2
7f 2a
804917b jg    80491a7 <decidir+0x45>
83 7d 08 00
804917d cmp    DWORD PTR [ebp+0x8],0x0
74 08
8049181 je    804918b <decidir+0x29>
83 7d 08 01
8049183 cmp    DWORD PTR [ebp+0x8],0x1
74 09
8049187 je    8049192 <decidir+0x30>
eb 1c
8049189 jmp    80491a7 <decidir+0x45>
b8 02 00 00 00
```

No hay mucho que discutir, una cascada de comparaciones de valores respecto a uno en concreto (el parámetro pasado) y sus respectivas comparaciones.

Lo que es probable es que el cuerpo de los casos posea más código y las comparaciones estén menos agrupadas entre sí.

Hay que tener cuidado porque, siempre, el switch puede imitarse con un conjunto de "if-else-if":

```

int decidir(int valor) {
    if (valor == 0) {
        return 2;
    } else if (valor == 1) {
        return 4;
    } else if (valor == 2) {
        return 8;
    } else if (valor == 3) {
        return 16;
    }
}

int main()
{
    decidir(3);
    return 0;
}

```

Assembly Address	Assembly Instruction	Comment
804909f	90	
8049162	nop	
8049163	decidir:	
8049165	push ebp	
8049166	mov esp,ebp	
8049167	b8 08 00 00	
8049168	cmp DWORD PTR [ebp+0x8],0x0	
8049169	75 07	
8049170	jne 8049172 <decidir+0x10>	
8049171	b8 02 00 00 00	
8049172	mov eax,0x2	
8049173	eb 27	
8049174	jmp 8049199 <decidir+0x37>	
8049175	b8 0d 08 01	
8049176	cmp DWORD PTR [ebp+0x8],0x1	
8049177	75 07	
8049178	jne 804917f <decidir+0x1d>	
8049179	b8 04 00 00 00	
8049180	mov eax,0x4	
8049181	eb 1a	
8049182	jmp 8049199 <decidir+0x37>	
8049183	b8 0d 08 02	
8049184	cmp DWORD PTR [ebp+0x8],0x2	
8049185	75 07	
8049186	jne 804918c <decidir+0x2a>	
8049187	b8 08 00 00 00	
8049188	mov eax,0x8	
8049189	eb 0d	
8049190	jmp 8049199 <decidir+0x37>	
8049191	b8 0d 08 03	
8049192	cmp DWORD PTR [ebp+0x8],0x3	
8049193	75 07	
8049194	jne 8049199 <decidir+0x37>	
8049195	b8 10 00 00 00	
8049196	mov eax,0x10	
8049197	eb 00	
8049198	jmp 8049199 <decidir+0x37>	
8049199	-:	

8.4. DETECCIÓN DE ESTRUCTURAS DE DATOS

8.4.1. ESTRUCTURAS

Un tipo de dato especialmente común en C y C++ (en su caso, el más complejo “objeto”) son las estructuras. Como sabemos una estructura se compone de varios miembros o variables de diverso tipo (incluso otras estructuras).

En ensamblador, una estructura aparece como un puntero al inicio del primer miembro y luego se va desplazando según la longitud del tipo del miembro referenciado. Con un ejemplo, se verá más claro:

```
#include <stdio.h>
#include <stdlib.h>

struct ejemplo {
    int valor;
    char *cadena;
    int array[3];
};

void procesar(struct ejemplo *e) {
    e->valor = 1;
    e->cadena = "hola";
    e->array[0] = 1;
    e->array[1] = 2;
    e->array[2] = 3;
}

int main()
{
    struct ejemplo *e = (struct ejemplo*)malloc(sizeof(struct ejemplo));
    procesar(e);
    printf(e->cadena);
    free(e);
    return 0;
}
```

A la izquierda se ve un pequeño programa que define una estructura muy simple con tres miembros, un entero, un puntero a una cadena y un array de tres enteros. Además, una función que rellena la estructura que ha de ser pasada por referencia.

A la derecha vemos la manipulación de la estructura pasada a la función “procesar” en ensamblador. Como vemos, se hace característico el puntero al comienzo de la estructura, **[EBP+0x8]**, que se carga en **EAX**, y a partir de ahí, se producen unos desplazamientos del tipo **[EAX+tamaño_del_tipo]**, para ir referenciando a los distintos miembros.

Las clases y objetos no son muy diferentes, ya que en realidad son estructuras con punteros a funciones y tablas de salto⁶⁸ para producir la herencia, polimorfismo, etc. Al final, el compilador ha de traducir a ensamblador y una clase o una estructura solo puede ser representada por los objetos más básicos de la programación.

8.4.2. ARRAYS

El array será similar a la estructura, pero con una clara característica: Al ser de tipo homogéneo (la estructura sería de tipos heterogéneos). De esa consecuencia se deriva que los desplazamientos serán todos ellos del mismo tamaño, sin variar.

Los arrays además tienen una característica. Al conocerse o derivarse su tamaño o longitud, suelen ser parte de una iteración; es decir, estar dentro de esta para que puedan ser procesados.

Abajo un programa que ilustra esa consecuencia. Tipo fijo, longitud derivada e iteración para procesarlo:

⁶⁸ https://en.wikipedia.org/wiki/Virtual_method_table

164

MODULO 3 – ANÁLISIS DE CÓDIGO FUENTE

<https://campusciberseguridad.com>

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4
5 void procesar(int *e, size_t len) {
6     for (int i = 0; i < len ; ++i) {
7         e[i] = i* 10;
8     }
9 }
10
11 int main()
12 {
13     int e[] = {1,2,3,4,5};
14     size_t len = sizeof(e)/sizeof(int);
15     procesar(e, len);
16     for (int i = 0; i < len ; ++i) {
17         printf("valor de e[%i]: %i\n", i, e[i]);
18     }
19     return 0;
20 }
```

```

eb 22
804917f jmp    80491a3 <procesar+0x31>
Bb 45 fc
8049181 mov    eax,DWORD PTR [ebp-0x4]
8d 14 85 00 00 00
8049184 lea    edx,[eax*4+0x0]
Bb 45 08
804918b mov    eax,DWORD PTR [ebp+0x8]
Bb 55 02
804918e lea    ecx,[edx+eax*1]
Bb 45 fc
8049191 mov    edx,DWORD PTR [ebp-0x4]
89 d0
8049194 mov    eax,edx
c1 e0 02
8049196 shl    eax,0x2
01 d0
8049199 add    eax,edx
01 c0
804919b add    eax,eax
89 01
804919d mov    DWORD PTR [ecx],eax
83 45 fc 01
804919f add    DWORD PTR [ebp-0x4],0x1
8b 45 fc
80491a3 mov    eax,DWORD PTR [ebp-0x4]
39 45 0c
80491a6 cmp    DWORD PTR [ebp+0xc],eax
77 d6
80491a9 ja    8049181 <procesar+0xf>
90
```

Algo muy importante que se debe aprender en C/C++ es que el nombre de un array es equivalente a la dirección de memoria de su primer elemento. De hecho, los arrays son simple azúcar sintáctica para no tener que trabajar con aritmética de punteros.

Es decir, se usa:

```
array[0] = ...
array[1] = ...
```

En vez de:

```
*array = ...
*(array+1) = ...
```

Observemos una modificación del programa anterior de la estructura que contenía un array. Fijaros en el modo en el que se está accediendo a los elementos del array:

The screenshot shows a debugger interface with two panes. The left pane displays the C source code:

```

#include <stdio.h>
#include <stdlib.h>

struct ejemplo {
    int valor;
    char *cadena;
    int array[3];
};

void procesar(struct ejemplo *e) {
    e->valor = 1;
    e->cadena = "hola\n";
    *(e->array) = 1;
    *(e->array + 1) = 2;
    *(e->array + 2) = 3;
}

int main() {
    struct ejemplo *e = (struct ejemplo *)malloc(sizeof(struct
procesar(e);
    printf(e->cadena);
    printf("%i\n", e->array[0]);
    printf("%i\n", e->array[1]);
    printf("%i\n", e->array[2]);
    free(e);
    return 0;
}

```

The right pane shows the assembly output and the command-line interface:

```

ASM generation compiler returned: 0
Execution build compiler returned:
Program returned: 0
hola
1
2
3

```

A vertical list of numbers from 1 to 27 is visible on the far right.

8.5. ¿POR QUÉ ES ÚTIL ESTA INFORMACIÓN?

El ensamblador es un lenguaje sencillo. Compuesto por instrucciones muy básicas. Por ello mismo, al no poseer de estructuras de alto nivel, se produce un código muy vertical y lleno de saltos.

Por consiguiente, debemos entrenar el ojo para detectar construcciones de más alto nivel que nos ayude a comprender el código. Las herramientas nos van a ayudar con los saltos, ya que no es complicado para ellas representarlos.

Algunas herramientas nos pueden facilitar también la detección de funciones, estructuras, etc. Pero no podemos poner la mano en el fuego por los resultados que estas arrojen, además de que debemos, como analistas, comprender con un simple vistazo que está ocurriendo.

8.6. ¿QUÉ TENGO QUE HABER APRENDIDO?

- Detectar funciones mediante el análisis de sus prólogos y epílogos.
- Detectar estructuras de iteración y control de flujo.
- Observar que parámetros toma la función analizada y sus variables locales.

9. ANÁLISIS DE CÓDIGO FUENTE EN LENGUAJES DE ALTO NIVEL

Llegados a este punto, debemos saber identificar el lenguaje o lenguajes usados en un proyecto de análisis. Obtener un desensamblado e incluso la decompilación de este en determinados casos (principalmente, cuando estamos estudiando un binario o librería). Sabemos detectar las estructuras básicas, clasificar las funciones, identificar los módulos empleados, etc.

Tampoco debemos olvidar las herramientas, que serán las que nos van a facilitar el trabajo de análisis. Una vez disponemos de todos los elementos necesarios, tan solo nos queda armarnos de paciencia y un buen blog de notas al que acudir, para reflejar nuestras anotaciones y comentarios.

Vamos a suponer que queremos estudiar el código fuente de **Mirai**⁶⁹, una botnet cuyo código fuente fue liberado y es público. Veremos que podemos sacar en claro, como funcionan determinados componentes, etc.

ATENCIÓN: No compiles y mucho menos ejecutes el código de este proyecto. Al fin y al cabo, se trata de un malware y esta asignatura no requiere ningún tipo de análisis dinámico. Tan solo se estudiará el código fuente de forma estática.

El código de Mirai está disponible en el siguiente repositorio de Github:

<https://github.com/jgamblin/Mirai-Source-Code>
(commit: 3273043e1ef9c0bb41bd9fc5317f7b797a2a94)

No descaguéis el código de otra fuente ni otro origen. En caso de no estar disponible, contactad con el profesor de la asignatura.

Podéis clonar el código con:

\$ git clone <https://github.com/jgamblin/Mirai-Source-Code>

9.1. DESCRIPCIÓN DEL PROYECTO MIRAI

El código de Mirai se divide en varias partes, un bot escrito en lenguaje C, un panel de control escrito en Go, scripts en bash para realizar la compilación cruzada de los bots (Mirai ataca a varias plataformas, luego necesita que su código sea multiplataforma).

De hecho, para poder generar binarios en las múltiples arquitecturas que afecta, necesita descargar varias versiones de **ulibc**⁷⁰, una librería estándar de C especialmente diseñada para dispositivos IoT.

Mirai no usa un programa de construcción tipo **make**, sino que contiene scripts para ese cometido.

⁶⁹ [https://en.wikipedia.org/wiki/Mirai_\(malware\)](https://en.wikipedia.org/wiki/Mirai_(malware))

⁷⁰ <https://uclibc-ng.org/>

En realidad, no posee dependencias salvo por el caso de la librería estándar.

Llama la atención que para almacenaje de datos necesite una instalación completa de **mysql**, tanto servidor como cliente. La base de datos es usada para el C2 (Command and Control), es decir, el panel de control de Mirai; programado en Go.

9.2. OBJETIVO

El proyecto es grande, tanto que podríamos escribir un libro entero solo comentando el funcionamiento que vamos encontrando a lo largo del código. Vamos a centrarnos en una sola funcionalidad, la metodología empleada podemos aplicarla a otras partes.

En concreto, vamos a analizar y estudiar un componente o funcionalidad que el propio autor de Mirai comenta en los documentos que posee el propio proyecto:

Infrastructure Overview

- To establish connection to CNC, bots resolve a domain ([resolv.c](#) / [resolv.h](#)) and connect to that IP address
- Bots brute telnet using an advanced SYN scanner that is around 80x faster than the one in qbot, and uses almost 20x When finding bruted result, bot resolves another domain and reports it. This is chained to a separate server to automa

Vamos a ver como realiza la conexión con el CNC (o C2), especialmente, en la funcionalidad de resolver un dominio hacia una IP.

9.3. HERRAMIENTAS EMPLEADAS

- Máquina virtual con Kali Linux (sin conexión a Internet)
- Visual Studio Code
- Git
- Ripgrep (<https://github.com/BurntSushi/ripgrep>)
- Fd (<https://github.com/sharkdp/fd>)
- CyberChef (<https://gchq.github.io/CyberChef/>)

9.4. MÓDULOS EMPLEADOS Y REFERENCIAS

El propio autor nos da una pista sobre donde localizar la funcionalidad: los archivos **resolv.c** y su cabecera con las definiciones, **resolv.h**.

Con una simple búsqueda con ‘fd’ encontramos dichos archivos:

```
→ Mirai-Source-Code git:(master) fd resolv
mirai/bot/resolv.c
mirai/bot/resolv.h
```

Ahora buscamos que archivos podrían tener "resolv.h" como dependencia, es decir, sus definiciones para posteriormente poder usarlas en el código. Usamos "RipGrep":

```
+ Mirai-Source-Code git:(master) rg resolv.h
mirai/bot/resolv.c
16:#include "resolv.h"

mirai/bot/main.c
25:#include "resolv.h"

mirai/bot/scanner.c
28:#include "resolv.h"

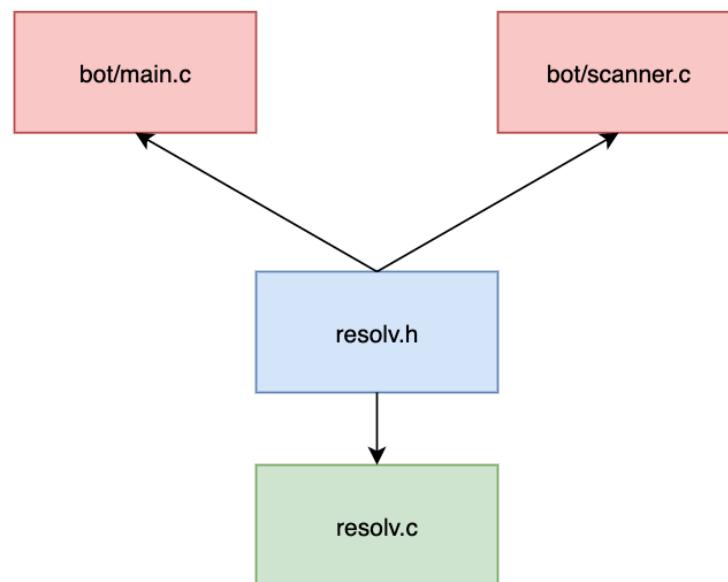
ForumPost.txt
46:- To establish connection to CNC, bots resolve a domain (resolv.c/resolv.h) and connect to that IP address

ForumPost.md
84: (['resolv.c'](mirai/bot/resolv.c)/['resolv.h'](mirai/bot/resolv.h)) and
+ Mirai-Source-Code git:(master) |
```

Obviamos los no relacionados con el código y el propio "resolv.c", tenemos dos archivos que usan las definiciones de resolv.h:

- **mirai/bot/main.c**
- **mirai/bot/scanner.c**

Esto nos deja con la siguiente estructura:



Ahora veremos que está definiendo (o exportando) el archivo "resolv.h":

```

3   #include "includes.h"
4
5   struct resolv_entries {
6       uint8_t addrs_len;
7       ipv4_t *addrs;
8   };
9
10  void resolv_domain_to_hostname(char *, char *);
11  struct resolv_entries *resolv_lookup(char *);
12  void resolv_entries_free(struct resolv_entries *);
13

```

Bien, tenemos una estructura sencilla, con un entero sin signo y un puntero a un tipo denominado “`ipv4_t`”. Los tipos de esta estructura vendrán declarados en algún archivo de “`includes.h`”; de momento no nos interesa.

Declara tres funciones:

- `resolv_domain_to_hostname`
- `resolv_lookup`
- `resolv_entries_free`

Ahora, con “RipGrep”, vamos a ver donde se están usando exactamente:

`resolv_domain_to_hostname`

```

→ Mirai-Source-Code git:(master) rg resolv_domain_to_hostname
mirai/bot/resolv.c
21:void resolv_domain_to_hostname(char *dst_hostname, char *src_domain)
74:    resolv_domain_to_hostname(qname, domain);

```

Solo un uso (obviamos la re-declaración), además es interno, es decir, solo es usado por `resolv.c`. Esta función podría haberse declarado “static”.

`resolv_lookup`

```

mirai/bot/main.c
363:     entries = resolv_lookup(table_retrieve_val(TABLE_CNC_DOMAIN, NULL));

mirai/bot/resolv.c
67:struct resolv_entries *resolv_lookup(char *domain)

mirai/bot/resolv.h
11:struct resolv_entries *resolv_lookup(char *);

mirai/bot/scanner.c
921:     entries = resolv_lookup(table_retrieve_val(TABLE_SCAN_CB_DOMAIN, NULL));
→ Mirai-Source-Code git:(master) █

```

De nuevo, obviamos las declaraciones. Se usa dos veces. En bot/main.c y bot/scanner.c

resolv_entries_free

```

mirai/bot/scanner.c
932:     resolv_entries_free(entries);

mirai/bot/resolv.h
12:void resolv_entries_free(struct resolv_entries *);

mirai/bot/resolv.c
225:     resolv_entries_free(entries);
230:void resolv_entries_free(struct resolv_entries *entries)

mirai/bot/main.c
373:     resolv_entries_free(entries);
+ Mirai-Source-Code git:(master) ┌─[root]

```

Por ultimo, esta función es usada dos veces coincidiendo en los mismos archivos que la función anterior. Además, era predecible, ese “**free**” en el nombre de la función deja muchas pistas: suelen ser funciones para liberar recursos, sobre todo memoria del montículo (o **heap**) obtenida a través de alguna función tipo **malloc** y familia.

9.5. ANALIZANDO EL CONTEXTO DE LAS FUNCIONES IMPLICADAS

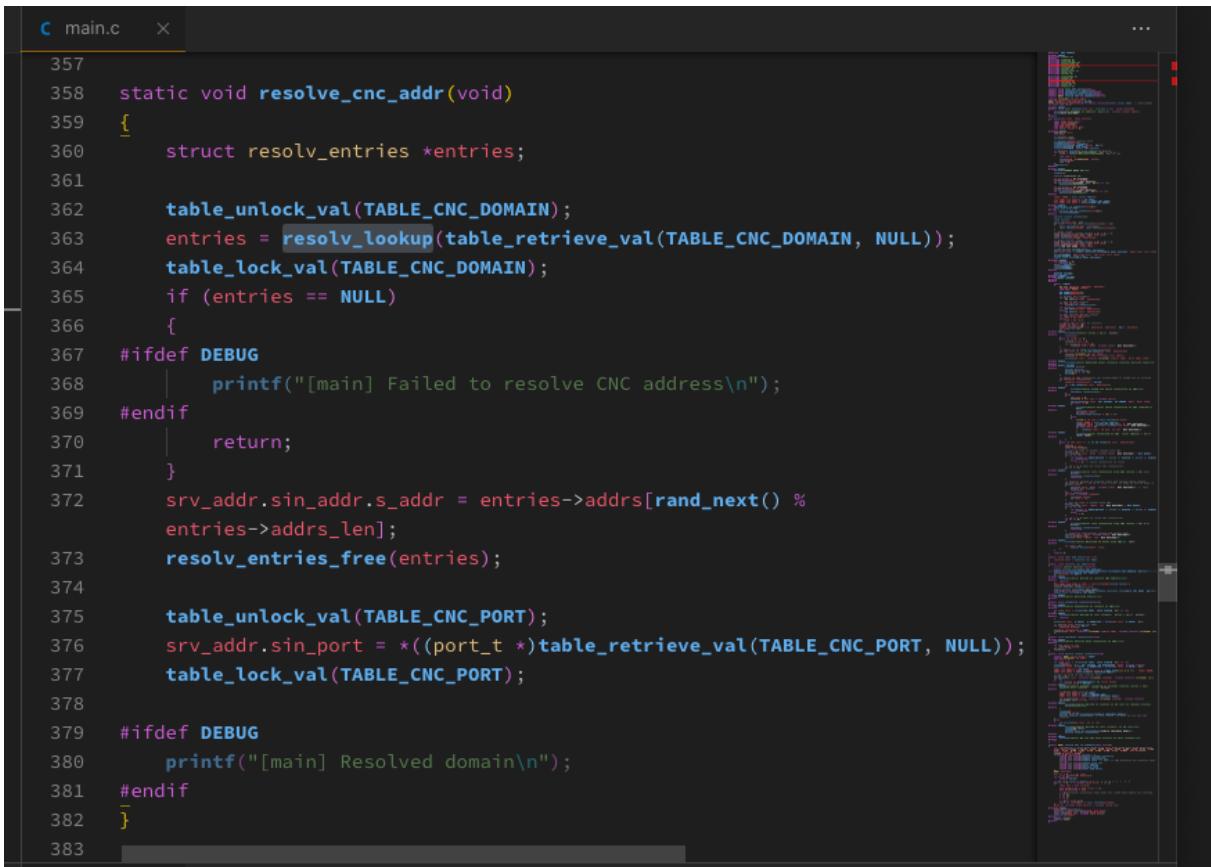
Veamos, ya sabemos que “**resolv_domain_to_hostname**” se usa de forma interna, por lo que esperaremos a analizar en detalle cuando lleguemos al archivo “resolv.c”.

Vamos a ver donde se usa “**resolv_lookup**”. Es importante analizar el **contexto**, ya que nos informa de como se está usando, que valores se le pasa, etc. Ya por lo pronto, debemos ver en la firma de la función que parámetros recoge y que devuelve:

```
struct resolv_entries *resolv_lookup(char *);
```

Es decir, toma un puntero a una cadena y devuelve un puntero a una estructura “**resolv_entries**” que, como hemos visto, está definida en “**resolv.h**”. Dado que no toma como parámetro ningún tipo “**resolv_entries**” ya podemos sospechar que la crea y reserva memoria dentro de “**resolv_lookup**” a partir del parámetro pasado. Lo veremos más tarde.

Ahora toca ver uno de los usos que hemos detectado, en el archivo “bot/main.c”:



```

C main.c  X
357
358     static void resolve_cnc_addr(void)
359     {
360         struct resolv_entries *entries;
361
362         table_unlock_val(TABLE_CNC_DOMAIN);
363         entries = resolv_lookup(table_retrieve_val(TABLE_CNC_DOMAIN, NULL));
364         table_lock_val(TABLE_CNC_DOMAIN);
365         if (entries == NULL)
366         {
367             #ifdef DEBUG
368                 printf("[main] Failed to resolve CNC address\n");
369             #endif
370             return;
371         }
372         srv_addr.sin_addr.s_addr = entries->addrs[rand_next() %
373             entries->addrs_len];
373         resolv_entries_free(entries);
374
375         table_unlock_val(TABLE_CNC_PORT);
376         srv_addr.sin_port = *((port_t *)table_retrieve_val(TABLE_CNC_PORT, NULL));
377         table_lock_val(TABLE_CNC_PORT);
378
379         #ifdef DEBUG
380             printf("[main] Resolved domain\n");
381         #endif
382     }
383

```

Su uso se da en la línea 363 y su valor lo recoge la variable “entries”. Más abajo, en la línea 373 podemos ver como se libera esa estructura que nos devuelve “resolv_lookup”. Ya se ha usado y entonces se liberan sus recursos.

¿Para qué se usa “entries”? Como vemos, y podemos sospechar, en la línea 372 se toma un valor de la estructura, en concreto del campo “addrs” que es un array de tipos “ipv4_t”:

```

4
5     struct resolv_entries {
6         uint8_t addrs_len;
7         ipv4_t *addrs;
8     };
9

```

Es fácil deducir que ese array contiene direcciones IPv4. Además, toma una de ellas de una manera curiosa. Hay que fijarse en la forma de indexar el array:

```

370         return;
371     }
372     srv_addr.sin_addr.s_addr = entries->addrs[rand_next() % entries->addrs_len]; Jerry Gamblin, 4 years ago • Trying to Sh
373     resolv_entries_free(entries);
374

```

Es un módulo entre un entero aleatorio y la cantidad de entradas que hay en “entries”, almacenada como campo en la propia estructura, en “addrs_len”. Eso evita que el índice salga del array y provoque un error (siempre y cuando “addrs_len” cumpla, claro)

Por si acaso, “ipv4_t” está definida así:

```

14
15     typedef uint32_t ipv4_t; Jerry Gamblin, 4 years ago

```

Es un entero sin signo de 32 bits (4 bytes). Coincide con una dirección IPv4 a que está compuesta por cuatro octetos y es posible almacenar estos valores en un número de 32 bits⁷¹.

Bien, pues ya tenemos una pista bastante buena de que hace la función “resolv_lookup”. Toma un parámetro y devuelve una estructura que contiene enteros que son direcciones IPv4.

¿Qué juego tiene el parámetro? Pues según la firma este parámetro es un puntero a una cadena. Resulta curioso, toma una cadena y devuelve direcciones IP...

Veamos como se le pasa ese parámetro:

```

table_unlock_val(TABLE_CNC_DOMAIN),
entries = resolv_lookup(table_retrieve_val(TABLE_CNC_DOMAIN, NULL));
table_lock_val(TABLE_CNC_DOMAIN);

```

El “char *” es la salida de una función “table_retrieve_val” que toma dos parámetros definidos por macro. Viendo las definiciones de estos (las podéis buscar con RipGrep como ejercicio) son: 3 y 0.

Buscamos “table_retrieve_val” y vemos que podría estar haciendo:

⁷¹ <https://rm-rf.es/convertir-direcciones-ip-decimal-a-binario-y-viceversa/>

```

108 char *table_retrieve_val(int id, int *len) Jerry Gamblin, 4 years ago • Tr
109 {
110     struct table_value *val = &table[id];
111
112 #ifdef DEBUG
113     if (val->locked)
114     {
115         printf("[table] Tried to access table.%d but it is locked\n", id);
116         return NULL;
117     }
118 #endif
119
120     if (len != NULL)
121         *len = (int)val->val_len;
122     return val->val;
123 }
```

Vale, ya sabemos que en nuestro caso es llamada con 3 y 0. Además, devuelve (línea 122) el miembro “val” de la estructura “table_value”, que a su vez, toma el valor de una variable presumiblemente global o de archivo: “table”.

Esta es la definición de “table_value”:

```

Jerry Gamblin, 4 years ago | 1 author (Jerry Gamblin)
6 struct table_value { Jerry Gamblin, 4 years ago • Tryin
7     char *val;
8     uint16_t val_len;
9 #ifdef DEBUG
10    BOOL locked;
11 #endif
12 };
13
```

Ya sabemos que es una estructura que guarda una cadena y un valor que podría ser su longitud. Es decir, las cadenas no van a terminar con el carácter de finalización de cadena clásico de C, el ‘\0’.

Ahora veamos, ese “&table[id]” que en nuestro caso sería “&table[3]”.

Se declara aquí:

```

12
13     uint32_t table_key = 0xdeadbeef;
14     struct table_value [table[TABLE_MAX_KEYS];
15
```

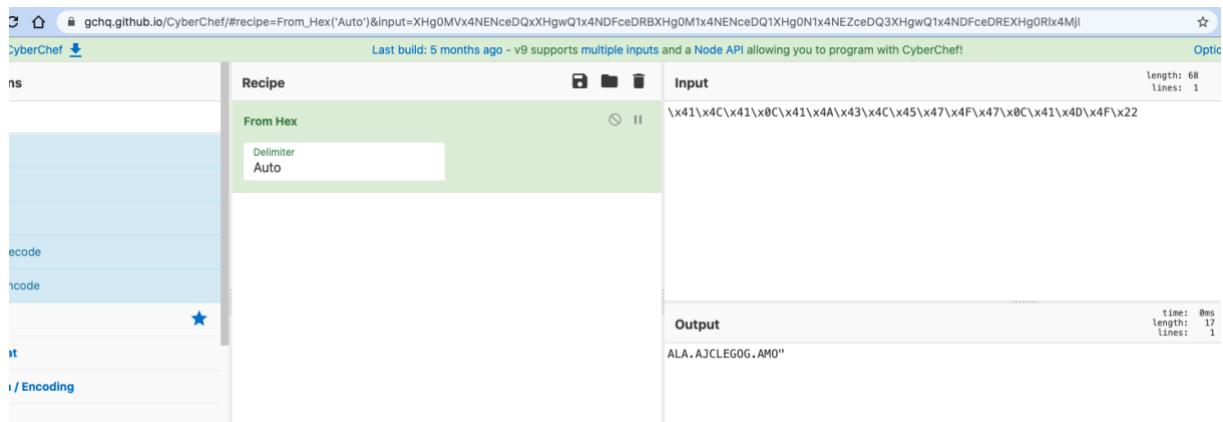
“TABLE_MAX_KEYS” es una macro definida a 52, el máximo de la tabla.

Esa tabla ha de ser llenada con cadenas y se van a indexar en orden. El orden viene definido en macros, al menos la que vimos originalmente como “TABLE_CNC_DOMAIN”. Hacemos una búsqueda para ver el uso de esa macro y sospechosamente vemos en el mismo archivo donde está “table” lo siguiente:

```

16  void table_init(void)
17  {
18      add_entry(TABLE_CNC_DOMAIN,
19      "\x41\x4C\x41\x0C\x41\x4A\x43\x4C\x45\x47\x4F\x47\x0C\x41\x4D\x4F\x22",
20      30); // cnc.changeme.com Jerry Gamblin, 4 years ago • Trying to Shri
      add_entry(TABLE_CNC_PORT, "\x22\x35", 2); // 23
  
```

Ahí lo tenemos, la cadena y su longitud, 30 caracteres en hexadecimal. Vemos un comentario “cnc.changeme.com”, pero es simplemente un aviso del creador de Mirai para que se cambie el dominio del C2. Aislemos la cadena y decodifiquemos su contenido. Usaremos **CyberChef**⁷².



The screenshot shows the CyberChef interface. In the 'Input' field, the hex value `\x41\x4C\x41\x0C\x41\x4A\x43\x4C\x45\x47\x4F\x47\x0C\x41\x4D\x4F\x22` is pasted. The 'From Hex' tab is selected under the 'Recipe' dropdown. The output is displayed in the 'Output' field as `ALA.AJCLEGOG.AMO`.

Bien, nos da la cadena: **ALA.AJCLEGOG.AMO**” Es evidente que es un dominio, pero está cifrado para evitar un análisis trivial de cadenas sobre el binario, de este modo, solo las descifra cuando es necesario.

9.6. ANALIZANDO LAS FUNCIONES IMPLICADAS

Ahora ya sabemos como se están usando, su contexto. Vayamos al corazón de estas funciones para hallar su verdadero comportamiento.

En primer lugar, despejemos la incógnita de la función que sospechábamos, se usaba simplemente para liberar recursos, memoria del montículo (heap):

⁷² <https://gchq.github.io/CyberChef/>

```
void resolv_entries_free(struct resolv_entries *entries)
{
    if (entries == NULL)
        return;
    if (entries->addrs != NULL)
        free(entries->addrs);
    free(entries);
}
```

Listo, “`resolv_entries_free`” no tiene más. Entra como parámetro un puntero a una estructura “`resolv_entries`”, que hemos visto, comprueba que no es NULL (indicaría que ya no apunta a una dirección del montículo) y pasa a liberar los recursos del miembro o campo “`addrs`” y el de la propia estructura por ese mismo orden.

Ahora tenemos por delante dos funciones más:

- `resolv_lookup`
- `resolv_domain_to_hostname`

Sus firmas son, respectivamente:

```
struct resolv_entries *resolv_lookup(char *);
```

```
void resolv_domain_to_hostname(char *, char *);
```

De la primera ya conocemos su uso, porque es llamada externamente y hemos visto su contexto. Toma una cadena (que está codificada) y devuelve una estructura que contiene un array de direcciones IPv4 en formato de entero sin signo; fácilmente traducible a octetos como hemos visto.

La segunda no devuelve nada, pero toma dos punteros. Por el nombre, y la gran pista que es que no devuelva nada, parecería que toma dos punteros porque lee de uno, procesa, y guarda el resultado en el otro puntero. ¿Será eso?

Veamos su definición:

```

21 void resolv_domain_to_hostname(char *dst_hostname, char *src_domain)
22 {
23     int len = util_strlen(src_domain) + 1;
24     char *lbl = dst_hostname, *dst_pos = dst_hostname + 1;
25     uint8_t curr_len = 0;
26
27     while (len-- > 0)
28     {
29         char c = *src_domain++;
30
31         if (c == '.' || c == 0)
32         {
33             *lbl = curr_len;
34             lbl = dst_pos++;
35             curr_len = 0;
36         }
37         else
38         {
39             curr_len++;
40             *dst_pos++ = c;
41         }
42     }
43     *dst_pos = 0;
44 }
```

- 1.- Comienza tomando en “len” la longitud de la cadena apuntada por “src_domain” + 1.
- 2.- Inicializa dos punteros. Uno al comienzo de “dst_domain” y otro justo la dirección posterior. Es decir, el carácter inicial y el siguiente.
- 3.- Inicializa “curr_len” a cero. Es una variable que usará (viendo el código) como contador.
- 4.- Entra en un bucle “while”, estructura iterativa que lo hará “len” veces, es decir, ya sabemos que va a recorrer el equivalente a lo que mide la cadena “src_domain”. Es decir, o va a leerla o va a escribirla. Si observamos con detenimiento el código vemos que tan solo lee de “src_domain”, es decir podría haber sido un “const char*” porque no escribe en ella.
- 5.- Línea 29, toma un carácter de la cadena apuntada por “src_domain”.
- 6.- ¿El carácter es un ‘.’ o un ‘0’?
 - 7.- Si lo es, guarda el valor actual de “curr_len” allí donde apunte “lbl”, que recordemos apunta a una posición en “dst_hostname”.
 - 8.- Mueve el puntero “lbl” a la posición donde esté “dst_pos” y avanza en una posición este último.
 - 9.- Reinicia “curr_len” a 0.
- 10.- El carácter ‘c’ no es ni ‘.’ ni ‘0’
 - 11.- Suma 1 a “curr_len”
 - 12.- Escribe el carácter allí donde apunte “dst_pos” y avanza el puntero una posición.
- 13.- Línea 43, fuera ya del bucle “while” finalmente escribe un ‘0’ al final de “dst_pos” que recordemos apunta a “dst_hostname”.

Teníamos razón, “src_domain” se usa de lectura y guarda el resultado de la función en “dst_hostname”. Es una función hecha para mutar un objeto pasado como parámetro.

¿Pero que hace realmente la función? Inserta un número y luego el número de caracteres que le sigue y que coinciden antes del final de cadena ‘0’ o de un ‘.’, es decir, un fragmento del dominio.

Vamos a extraer el fragmento de la función, lo modificamos para que funcione de forma aislada y en un programa ad-hoc y haremos una prueba:

```

3 #include <string.h>
4
5 void resolv_domain_to_hostname(char *dst_hostname, char *src_domain) {
6     int len = strlen(src_domain) + 1;
7     char *lbl = dst_hostname, *dst_pos = dst_hostname + 1;
8     uint8_t curr_len = 0;
9
10    while (len-- > 0) {
11        char c = *src_domain++;
12
13        if (c == '.' || c == 0) {
14            *lbl = curr_len;
15            printf("%u", curr_len);
16            lbl = dst_pos++;
17            curr_len = 0;
18        } else {
19            curr_len++;
20            *dst_pos++ = c;
21            printf("%c", c);
22        }
23    }
24    *dst_pos = 0;
25    printf("\n");
26 }
27
28 char *cad = "www.google.es";
29
30 int main() {
31     char buf[2048];
32     resolv_domain_to_hostname(buf, cad);
33     return 0;
34 }
```

Ejecutamos:

```

→ master_ACF ./a.out
www3google6es2
→ master_ACF _
```

¿Qué es eso? Esto -> <https://www.rfc-editor.org/rfc/rfc1035.html#section-4.1.2>

Es decir, es una **función auxiliar**, que nos permite pasar un dominio a un formato necesario para realizar consultas DNS. Ya hemos desvelado el misterio de la función “resolv_domain_to_hostname”.

Vamos a pasar a **resolv_lookup**. Prescindimos de las líneas que están bajo la macro de debug y plegamos el código del “while”:

```
You, 11 minutes ago | 2 authors (Jerry Gamblin and others)
struct resolv_entries *resolv_lookup(char *domain)
{
    struct resolv_entries *entries = calloc(1, sizeof(struct resolv_entries));
    char query[2048], response[2048];
    struct dnshdr *dnsh = (struct dnshdr *)query;
    char *qname = (char *)(dnsh + 1);

    resolv_domain_to_hostname(qname, domain);

    struct dns_question *dnst = (struct dns_question *) (qname + util_strlen(qname) + 1);
    struct sockaddr_in addr = {0};
    int query_len = sizeof(struct dnshdr) + util_strlen(qname) + 1 + sizeof(struct dns_question);
    int tries = 0, fd = -1, i = 0;
    uint16_t dns_id = rand_next() % 0xffff;

    util_zero(&addr, sizeof(struct sockaddr_in));
    addr.sin_family = AF_INET;
    addr.sin_addr.s_addr = INET_ADDR(8, 8, 8, 8);
    addr.sin_port = htons(53);

    // Set up the dns query
    dnsh->id = dns_id;
    dnsh->opts = htons(1 << 8); // Recursion desired
    dnsh->qdcount = htons(1);
    dnst->qtype = htons(PROTO_DNS_QTYPE_A);
    dnst->qclass = htons(PROTO_DNS_QCLASS_IP);

    while (tries++ < 5) -
        close(fd); Jerry Gamblin, 4 years ago • Trying to Shrink Size

    if (entries->addrs_len > 0)
        return entries;
    else
    {
        resolv_entries_free(entries);
        return NULL;
    }
}
```

¿Qué vemos? Algo realmente apasionante. Mirai no usa librerías DNS ni el resovedor del sistema, más que nada porque probablemente ni tan siquiera tenga (recordad que los afectados son dispositivos IoT). Prepara una consulta DNS y la lanza usando sockets UDP. La función en si soporta demasiadas responsabilidades, es decir, hace muchas cosas que podrían ser fraccionadas en funciones más pequeñas, pero es lo que tenemos.

Como vemos, son funciones y estructuras de preparación de una consulta DNS a través de sockets, podemos abstraernos de algunas funciones y centrarnos en el nexo de comportamiento de la función. Pero antes veremos como prepara la consulta.

En esta parte prepara los bufés que necesita para enviar el paquete DNS request de consulta:

```
struct resolv_entries *entries = calloc(1, sizeof(struct resolv_entries));
char query[2048], response[2048];
struct dnshdr *dnsh = (struct dnshdr *)query;
char *qname = (char *)(dnsh + 1);
```

Fijaros en ese “qname”, que apunta a una parte concreta de la estructura “dnsh”, que como su propio nombre indica es la cabecera de la petición DNS.

73	
74	resolv_domain_to_hostname(qname, domain);
75	

A continuación, utiliza la función auxiliar que acabamos de estudiar, para convertir un dominio en el

formato adecuado para la consulta DNS (ver enlace a la RFC1035 que hemos puesto arriba).

Más subestructuras necesarias para la consulta, el cuerpo de la consulta, el socket que se va a abrir, el número de intentos y un ‘id’ aleatorio:

```

15
76     struct dns_question *dnst = (struct dns_question *) (qname + util_strlen(qname) + 1);
77     struct sockaddr_in addr = {0};
78     int query_len = sizeof(struct dnshdr) + util_strlen(qname) + 1 + sizeof(struct dns_question);
79     int tries = 0, fd = -1, i = 0;
80     uint16_t dns_id = rand_next() % 0xffff;
81

```

Inicialización de los valores adecuados sobre las estructuras de arriba:

```

81
82     util_zero(&addr, sizeof(struct sockaddr_in));
83     addr.sin_family = AF_INET;
84     addr.sin_addr.s_addr = INET_ADDR(8, 8, 8, 8);      You, 9 hours ago • Uncommitted changes
85     addr.sin_port = htons(53);
86
87     // Set up the dns query
88     dnsh->id = dns_id;
89     dnsh->opts = htons(1 << 8); // Recursion desired
90     dnsh->qdcount = htons(1);
91     dnst->qtype = htons(PROTO_DNS_QTYPE_A);
92     dnst->qclass = htons(PROTO_DNS_QCLASS_IP);
93

```

Nótese como usa el conocido DNS 8.8.8.8 de Google y el correspondiente puerto 53 que manda usar el protocolo DNS por defecto.

Ahora, la función entra en un bucle “while” que ejecutará 5 veces:

```

} > while (tries++ < 5) ...

```

Esta parte, primera sección del bucle “while”, se encarga de testear que ha sido posible obtener un socket, conectar al servidor y enviar la petición; “socket”, “connect” y “send” respectivamente. Los “if” crean una pausa para asegurarse que el sistema está preparado para la siguiente llamada. Son dispositivos con poca capacidad de cómputo y tampoco hay prisas.

El siguiente bloque es, básicamente, una espera por la respuesta (o ninguna respuesta) del servidor DNS interrogado:

```

fcntl(F_SETFL, fd, O_NONBLOCK | fcntl(F_GETFL, fd, 0));
FD_ZERO(&fdset);
FD_SET(fd, &fdset);

timeo.tv_sec = 5;
timeo.tv_usec = 0;
nfds = select(fd + 1, &fdset, NULL, NULL, &timeo);

if (nfds == -1)
{
    break;
}
else if (nfds == 0)
{
    continue;
}
else if (FD_ISSET(fd, &fdset))
{

```

Si se tiene éxito, se ejecutará el “if” correspondiente a la macro FD_ISSET, ya que los “if” anteriores miran el valor de retorno de “select”. Si este es “0” indica un timeout de la operación y volverá a intentarlo de nuevo en la siguiente iteración. Si es “-1” indica un error, no hay mucho que hacer y simplemente se repliega, libera recursos y devolverá el NULL del final de la función.

Vamos a suponer que tiene éxito, seguimos por el tercer “if”. La parte que sigue es la obtención de la respuesta y creación de las estructuras necesarias para su interpretación (parseo):

```

int ret = recvfrom(fd, response, sizeof(response), MSG_NOSIGNAL, NULL, NULL);
char *name;
struct dnsans *dnsa;
uint16_t ancount;
int stop;

if (ret < (sizeof(struct dnshdr) + util_strlen(qname) + 1 + sizeof(struct dns_question)))
{
    continue;

dns = (struct dnshdr *)response;
qname = (char *)(dns + 1);
dnst = (struct dns_question *) (qname + util_strlen(qname) + 1);
name = (char *) (dnst + 1);

if (dns->id != dns_id)
    continue;
if (dns->ancount == 0)
    continue;

ancount = ntohs(dns->ancount);

```

Finalmente, itera sobre la respuesta, pero nos interesa especialmente una parte esencial de esa iteración:

```

if (ntohs(r_data->data_len) == 4)
{
    uint32_t *p;
    uint8_t tmp_buf[4];
    for (i = 0; i < 4; i++)
        tmp_buf[i] = name[i];

    p = (uint32_t *)tmp_buf;      Jerry Gamblin, 4 years ago • Trying to Shrink Size
    entries->addrs = realloc(entries->addrs, (entries->addrs_len + 1) * sizeof
    (ipv4_t));
    entries->addrs[entries->addrs_len++] = (*p);
}
  
```

Está obteniendo las direcciones IP de la respuesta (que pueden ser varias) y las está añadiendo a la estructura que devolverá “resolv_entries”, que como sabemos de su definición posee un array de direcciones IPv4 en formato entero y su longitud.

Hay partes interesantes, como la vuelta a reasignar memoria cada vez que mete un nuevo registro en la estructura, cuando curiosamente, solo almacena enteros del tipo uint32_t. Un compromiso entre el gasto de cómputo en recalcular fragmentos del montículo y no malgastar memoria reservando un array para varias entradas:

```

entries->addrs = realloc(entries->addrs, (entries->addrs_len + 1) * sizeof
(ipv4_t));
entries->addrs[entries->addrs_len++] = (*p);
  
```

Finalmente, libera recursos y devuelve las entradas “entries” que ha recibido del servidor DNS como respuesta. O lo que es lo mismo, ha encontrado la IP de uno o varios C2 para el bot.

```

203 if (entries->addrs_len > 0)
204     return entries;
205 else
206 {
207     resolv_entries_free(entries);
208     return NULL;
209 }
210 }
  
```

9.7. CONCLUSIÓN

Hemos visto al detalle como funciona un módulo de un malware. Un componente que se encarga de gestionar la resolución de nombres. Sabemos como lo hace, a que DNS fijado interroga, como gestiona las entradas, como implementa las peticiones y respuestas, etc.

Esto solo es la punta del iceberg. Si navegáis por el código veréis multitud de cosas interesantes y preguntas que podríamos arrojar, siendo la más evidente: ¿Cómo descifra las cadenas donde supuestamente van los dominios del C2?

La respuesta es clave para, por ejemplo, producir reglas IDS que permitan bloquear peticiones salientes a esos dominios, salvando a una organización de que sus dispositivos infectados obtengan órdenes de los C2.

9.8. ¿POR QUÉ ES ÚTIL ESTA INFORMACIÓN?

Hemos visto como un análisis detallado responde a preguntas que pueden hacernos: ¿Cómo funciona este malware? ¿Cómo podemos pararlo? etc. Un análisis concienzudo puede ayudar a entender como funciona a la perfección y proveernos de conocimientos para paliar una amenaza concreta.

La ingeniería inversa es leer código e interpretar su intención. Es atar cabos entre funciones, seguir el hilo de las bifurcaciones, estar atentos a los cambios iterativos de los bucles e interrelacionar módulos para crear, a partir de piezas más pequeñas, el puzzle completo.

9.9. ¿QUÉ TENGO QUE HABER APRENDIDO?

- A comprender la tarea de un analista ya sea sobre malware, búsqueda de vulnerabilidades o reversing para entender como funcionan las cosas.
- A entender que análisis significa separar en partes un todo para obtener un conocimiento sobre este, observando y estudiando sus piezas.
- A ver el código no solo como algo escrito para un ordenador. El código es para declarar una intención y nuestro objetivo es llegar a descubrirla.

10. APÉNDICE A: GUÍA BÁSICA DE ENSAMBLADOR

A continuación, se da una introducción al ensamblador de la arquitectura Intel x86. Como sabemos, el ensamblador es la lengua franca del reversing y este no es único, sino que cada arquitectura posee su propio conjunto de mnemotécnicos, registros, direccionamientos y reglas que, aunque pueden tener un nexo común, difieren entre ellas. Es más, incluso el propio ensamblador de Intel puede presentarse en distintos formatos: ATT&T y el propio de Intel. Aquí usaremos el formato Intel, más cercano al reversing de la plataforma Windows.

Aunque estudiemos el ensamblador de una sola arquitectura, los conceptos son similares en casi todas. No es complicado, una vez nos sentimos cómodos en una arquitectura y sintaxis concreta, dominar el ensamblador de otra arquitectura. De ahí que solo mostremos o, mejor dicho, nos centremos, en una sola en particular.

No presentaremos aquí la visión o perspectiva de “programar en ensamblador”. El reversing proporciona un “desensamblado”, que puede ser o no el producto de un compilador. Programar en ensamblador es una cosa e interpretar un desensamblado otra. En particular, los ensambladores modernos proveen de macros, funciones, etc, mientras que el desensamblado es una “traducción” del código máquina; que es lo que al final procesará el (valga la redundancia) microprocesador.

Tampoco es esta una referencia completa, sino una introducción muy sucinta que ayudará a superar los primeros pasos en este lenguaje tan básico a las personas que no hayan tenido contacto con él.

10.1. REGISTROS

10.1.1. REGISTROS DE LA CPU

Acceder a la memoria RAM es lento. Muy lento. Para eso se diseñaron las memorias caché y las estrategias asociadas a las mismas para renovar su contenido intentando bajar la tasa de fallos todo lo que les es posible.

Debido a que es acceder a memoria es lento, la CPU posee un espacio dentro de ella para alojar unas pocas variables. A esos espacios los denominamos **registros**. Son pocos. 8 registros de propósito general en la arquitectura de 32 bits y 16 en la de 64 bits. Además, el registro de puntero a instrucción o *instruction pointer*, 6 registros de segmento y el EFLAGS (en x86) o RFLAGS (en x64).

Cada registro, posee una longitud acorde a la arquitectura: en x86 serán de 32 bits y en x64 son de 64 bits. Además, acceder a un registro es bastante fácil, solo debemos referirnos a su nombre y el ensamblador lo traducirá en el *opcode*⁷³ adecuado para ejecutar la instrucción de forma oportuna.

Por ejemplo:

ADD EAX, 1 ; Suma 1 al registro EAX

⁷³ <https://en.wikipedia.org/wiki/Opcode>

NOTA: Existen muchos más registros, en particular aquellos asociados con la aritmética de punto flotante y aquellos registros usados como extensiones al estándar de Intel, como: SSE, AVX o MMX. Estos registros y extensiones aparecieron cuando la informática entró de lleno en la era multimedia y las capacidades de las CPU para este tipo de cálculos se vieron desbordadas.

10.1.2. REGISTROS DE PROPÓSITO GENERAL

Como hemos comentado, son 8 en x86 y 16 en x64. La denominación viene de la época de las CPU de Intel de 16 bits, probablemente podrías enumerar algunos: EAX, EBX, ECX, EDX...

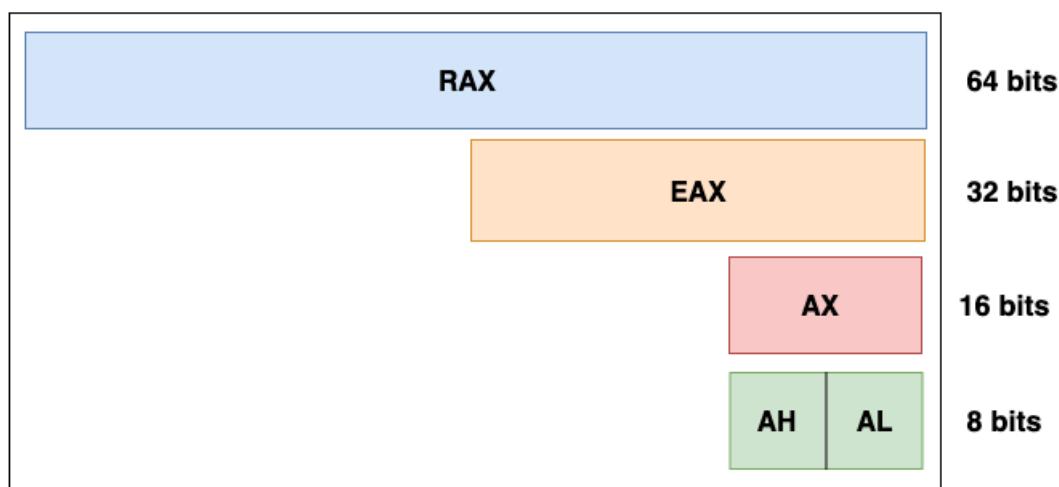
Vamos a explicar algo sobre la denominación. El registro **EAX** proviene de cuando en la arquitectura 8086 se denominaba **AX**, en virtud de su empleo más común: *Accumulator* o acumulador. Cuando surgió la arquitectura de 32 bits, el registro pasó a “extender” esos 16 bits a 32 bits por lo que pasó, a su vez, a denominarse: *Extended Accumulator Register*.

A su vez, cuando se pasó a la arquitectura de 64 bits, se pasó a denominar **RAX** siendo la ‘E’ sustituida por la ‘R’ de *register* o registro. Por lo tanto, cuando vemos a los registros denominados con la letra ‘E’ nos estamos refiriendo a los 32 bits de ese registro, cuando nos referimos a los registros con la letra ‘R’, entonces queremos acceder a sus 64 bits (naturalmente, solo disponible en la arquitectura de 64 bits)

¿Cómo nos referimos a los registros si queremos acceder solo a una parte? Bien, si queremos acceder a los primeros 16 bits, retiramos la primera letra de su nombre. Así, **EAX** pasa a ser **AX** y la CPU operará sobre los 16 bits (de derecha a izquierda).

Curiosamente, podemos, a su vez, dividir ese registro **AX** de 16 bits en otras dos partes: **AH** y **AL**. Así, si queremos acceder a los primeros 8 bits (de nuevo, de derecha a izquierda) entonces ponemos **AL**. Y si en cambio, queremos acceder a los otros 8 bits, lo referimos por **AH**. Siendo la ‘L’, low o bajo y ‘H’, high o alto (parte baja, parte alta).

Visualmente queda así (tomando a RAX como ejemplo):



Este acceso a bits “altos” o “bajos” (u más significativo, menos significativo) os sonará con mucha frecuencia. Los valores en la memoria van agrupados en bytes. Suponed el valor de un entero que es 0x1234. Eso son dos bytes: 0x12 y 0x34. Cuando nos referimos al byte **más significativo** es 0x12 y el **menos significativo** sería el 0x34.

Otro concepto que se debe integrar es el de “endianess”. Tomando el entero anterior, 0x1234, en plataformas **Little endian** el orden en memoria es 0x34 0x12. Es decir, encuentras primero el byte 0x34 y luego el 0x12. En plataformas **big endian** la disposición en memoria es tal cual: 0x12 0x34. Los registros de propósito general, como su nombre indican, pueden valer para casi cualquier cosa que se necesite con limitaciones.

Respecto al rol de los registros, ya hemos comentado que al ser de propósito general pueden poseer ciertos significados, pero estos dependen de los registros. Vamos a ver, en primer lugar, aquellos que poseen un rol fijo:

EBP (Base Pointer): Este registro se usa como base de la pila, donde creamos el marco de pila de la función. A partir de este eje, diferenciamos argumentos de variables locales, por ejemplo.

ESP (Stack Pointer): Apunta a la cima de la pila.

Los siguientes seis registros poseen roles variables, indicamos algunos usos clásicos:

EAX (Accumulator): Se denomina acumulador por su uso en operaciones aritméticas. También es usado como registro en el que se suele depositar el valor de retorno de una función.

EBX (Base): Lo de base (no confundir con el EBP) le viene de su uso en apuntar a la base de un segmento de memoria (ver luego los registros de segmentos) y a partir de ahí, mediante una operación aritmética, desplazarse sobre dicho segmento hasta alcanzar la dirección deseada.

ECX (Counter): Usado como contador para los bucles.

EDX (Data): Usado como parte de las operaciones de multiplicación y división.

ESI (Source Index): Tanto ESI como EDI se usan conjuntamente para operaciones, por ejemplo, de copia de arrays.

EDI (Destination Index): Ver ESI.

Recordemos: usos clásicos. Nada impide (de hecho, suele ser así) al compilador usar los registros a su discreción.

A estos 8 registro comentados, la arquitectura de 64 bits le suma otros 8 registros denominados de forma numérica de **r8 a r15**. Son de propósito general y esta ampliación de registros hace que el compilador pueda generar código que disminuya el uso de la pila, lo que redundaría en una mayor velocidad.

10.1.3. REGISTROS DE SEGMENTO

En primer lugar, estos registros salvo excepciones no son usados en la práctica hoy día. Su origen estaba en la forma de referenciar memoria, la cual se dividía en segmentos cuyas bases eran apuntadas por dichos registros. Así, para referirnos a una dirección en memoria, esta se componía del segmento (su registro) y un desplazamiento dentro del mismo.

Los registros de segmentos son:

SS (Stack segment),
CS (Code segment),
DS (Data segment),
ES (Extra segment),
FS (extra-extra segment),
GS (extra-extra-extra segment)

Es posible ver en uso los dos últimos (FS y GS) relacionados con la programación de hilos (*threads*). En particular, sirven aun para que un hilo pueda acceder a su TLS (*Threat Local Storage*), que es la zona de memoria exclusiva para almacenamiento local de ese hilo.

Por cierto, las letras 'F' y 'G' no poseen ningún significado especial, simplemente son las que siguen, alfabéticamente, a la 'E'.

10.1.4. EFLAGS O RFLAGS

Es un registro especial, booleano. Es así porque cada uno de los bits de este registro mapea sobre un significado (no se aprovechan todos, es decir, no hay 32 o 64 usos).

Algunos de los importantes son: **carry flag**, **parity flag**, **zero flag**, **sign flag**, **overflow flag**.

Son los denominados, respectivamente: acarreo, paridad, cero, signo y desbordamiento.

En inglés se les dice “flag”, en español se les suele decir “bit de”, por ejemplo: “bit de desbordamiento”.

El funcionamiento es muy simple. Por ejemplo, en una instrucción de comparación, tal como:

CMP EAX, ECX
JNZ <dir.salto>

La primera instrucción realiza una comparación de los valores que se hallan en los registros EAX y ECX.

Si EAX fuese igual a ECX no tendrían ninguna diferencia, es decir **tendrían 0 diferencias**.

Con 0 diferencias se activará el “flag zero” o “registro cero”. ¿Cómo se activa? **Se pondrá a ‘1’**.

JNZ es **JUMP if NOT ZERO**, es decir, saltará a <dir.salto> si el bit cero no se activa. Sería el equivalente en código a:

```
if (eax != exc) ...
```

10.1.5. PUNTERO DE INSTRUCCIÓN

El RIP, EIP o IP. Es el puntero donde se aloja la dirección de aquello que está ejecutando la CPU. Es decir, apunta a una dirección que contiene código a ejecutar. Es así de simple. Si paramos la ejecución de un programa mediante un depurador, para saber donde estamos solo tenemos que observar el EIP (bueno, y la traza de la pila que también nos ofrece gran cantidad de información) y dirigirnos a esa dirección para ver el código.

10.2. DIRECCIONAMIENTO

El direccionamiento es la forma en la que nos referimos a un valor o dirección de memoria. Cuando realizamos una operación en ensamblador, disponemos de un opcode (código de operación) y uno o dos operandos (en algunas arquitecturas este número puede variar).

Esos operandos pueden ser valores, registros, direcciones de memoria e incluso punteros a una dirección de memoria (base) más una operación aritmética de desplazamiento sobre dicha base.

A continuación, vamos a ver los direccionamientos más comunes:

10.2.1. DIRECCIONAMIENTO INMEDIATO

Es cuando ponemos el valor directamente en el operando, no referenciamos ni registros ni memoria. Un ejemplo:

ADD EAX, 1

Sumamos 1 al valor que ya hay en EAX. Como vemos es **inmediato** porque no necesitamos

referenciar nada.

10.2.2. DIRECCIONAMIENTO A REGISTRO

Hacemos referencia a un registro de la CPU. Por ejemplo:

MOV EAX, EBX

Esto hace que EAX valga aquello que contiene EBX (EAX = EBX). Es decir, nos dirigimos al registro EBX.

10.2.3. DIRECCIONAMIENTO DIRECTO (O, ABSOLUTO)

Reconocible por los corchetes y un único operando entre ellos.

MOV EAX, [0x3443FBCA]

MOV ECX, [EAX]

Toma, no el valor sino trata el objeto entre [] como una dirección a la que acudir a recoger el valor que asignará al primer operando. Como podemos observar, los corchetes denotan “el valor al que apunta” en vez de “el valor que contiene”, lo que nos lleva a pensar en como se manejan en lenguaje C los punteros.

10.2.4. DIRECCIONAMIENTO INDIRECTO

Idéntico al anterior, pero con una operación aritmética. Extremadamente común en el código. Recordad como se referencian las variables locales y parámetros de una función en la pila:

MOV EAX, [EBP+4]

Hace que EAX valga el valor que esté depositado cuatro bytes más arriba del puntero a la base de la pila EBP. Es decir, toma como dirección base el valor que haya en EBP y le suma cuatro para acceder al valor referenciado.

10.3. OPCODES

Un *opcode* es una instrucción. El nombre viene de *operation-code*. A cada instrucción se le asigna un código, que viene a ser un valor numérico habitualmente expresado en hexadecimal. Imaginad una larga, liguísima tabla de instrucciones y a cada uno de ellos se le asigna un valor numérico.

Existen multitud de opcodes o instrucciones. Un documento que debéis tener a mano siempre es el famoso manual de Intel⁷⁴, en el que viene toda la documentación acerca de las CPU de este

⁷⁴ <https://software.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-sdm-combined-volumes-1-2a-2b-2c-2d-3a-3b-3c-3d-and-4.html>

fabricante. Es la biblia del ensamblador o mejor dicho de la arquitectura.

Habitualmente, cuando queremos interpretar el código objeto (lo que haría el desensamblador), un shellcode o parchear un binario, acudimos a recursos de este tipo (manuales) o tablas que resumen el comentado manual⁷⁵.

Un opcode puede tener operandos, ya lo hemos comentado anteriormente. Estos pueden ser registros, direcciones o valores e incluso expresiones aritméticas de desplazamiento. El tipo y número de operandos viene especificado por el opcode en cuestión.

Por ejemplo, la instrucción MOV en si misma no posee un solo código, sino varios. Dependiendo del tipo de MOV puede tener varios códigos. Un ejemplo:

<code>00007f0f:32f7b159</code>	<code>48 8b 78 08</code>	<code>mov rdi, [rax+8]</code>
<code>00007f0f:32f7b15d</code>	<code>48 8b 05 64 d9 02 00</code>	<code>mov rax, [rel 0x7f0f32fa8ac8]</code>
<code>00007f0f:32f7b164</code>	<code>4c 8b 05 65 da 02 00</code>	<code>mov r8, [rel 0x7f0f32fa8bd0]</code>
<code>00007f0f:32f7b16b</code>	<code>48 8b 48 08</code>	<code>mov rcx, [rax+8]</code>
<code>00007f0f:32f7b16f</code>	<code>48 89 fb</code>	<code>mov rbx, rdi</code>
<code>00007f0f:32f7b172</code>	<code>48 89 f8</code>	<code>mov rax, rdi</code>
<code>00007f0f:32f7b175</code>	<code>48 8d 34 0f</code>	<code>lea rsi, [rdi+rcx]</code>

Vemos dos instrucciones MOV contiguas en el recuadro rojo. ¿Qué las diferencia? Una es un MOV (registro, registro+desplazamiento) y la otra es (registro, registro).

Algunos opcodes son muy típicos y pronto podréis memorizarlos, por ejemplo, el opcode de NOP el 0x90. Este opcode es muy característico para realizar padding o relleno. De hecho, es común escuchar hablar de “nopear” cierta cantidad de bytes.

Por cierto, NOP viene de NO-OPERATION, es decir, no hace nada.

10.4. INSTRUCCIONES COMUNES

Vamos a ver un grupo de instrucciones comunes, agrupadas por funcionalidad.

10.4.1. OPERADORES ARITMÉTICOS

INC Y DEC

Respectivamente, incremento y decremento a 1. Se ven mucho en bucles y hace lo que se espera de ellos, por ejemplo.

MOV EAX, 1
INC EAX

⁷⁵ <http://ref.x86asm.net/index.html>

Ahora EAX vale 2.

ADD, SUB

Suma y resta respectivamente. Suma o resta lo que esté a la derecha afectando al operador de la izquierda.

SUB EAX, 1
ADD ECX, EAX

Resta 1 a EAX y luego suma lo que valga EAX en ECX.

MUL Y DIV

Multiplicación y división respectivamente.

Ejemplo:

MUL 2
DIV EBX

Multiplica o divide el valor que se halle en EAX por el valor o registro o dirección y almacena dicho valor en EAX. También puede usar como dividendo conjuntamente los registros EDX:EAX si el dividendo es un número de mayor a 32 bits.

Poseen variantes para operar sobre tipos de datos sin signo: **IMUL** e **IDIV**.

10.4.2. OPERADORES DE COMPARACIÓN, LÓGICOS Y DE DESPLAZAMIENTO

Son varios, los más reseñables:

XOR, AND, NOT, OR

Son, como su propio nombre indica operadores lógicos. Funcionan exactamente igual a como esperaríamos en cualquier lenguaje de programación. Admiten dos operandos sobre los que hacer la operación lógica, salvo **NOT** que funciona, como es de esperar también, sobre un solo operando. El resultado es almacenado en el primer operando.

Ejemplo:

XOR EAX, ECX

Realizará la operación xor entre EAX y ECX, almacenando el resultado en EAX.

TEST, CMP, NEG

El operador **TEST** realiza un **AND**, pero no modifica el primer operando. Es usado muy a menudo como operador para realizar comparaciones respecto al primer operando.

El operador **CMP**, en realidad opera restando el primer operando con el segundo y descartando el resultado. Lo interesante es que si utiliza dicho resultado para manipular los bits del **EFLAGS**. Hay que estar atentos al comportamiento del **EFLAGS** cuando operamos con esta instrucción.

NEG realiza la negación en complemento a dos del valor de un registro o dirección.

SHL, SHR

Realizan un desplazamiento a 'L' izquierda o 'R' derecha, tantas posiciones como pongamos en el segundo operando. No se trata de una rotación, los bits que se van desplazando son ceros:

MOV EAX, 2
SHL EAX, 2

Sería equivalente a (en binario EAX):

0b10 << 2 == 0b1000

El lector atento habrá podido deducir que, efectivamente, SHL multiplica por 2 y SHR divide por esa misma cantidad.

10.4.3. OPERADORES DE SALTO

Tenemos saltos de dos tipos: condicionales e incondicionales. Comenzaremos con este último, el incondicional. Solo posee una instrucción muy aclaratoria: **JMP**, de *jump* o salto.

Una vez el procesador se topa con una instrucción **JMP** salta a la dirección que se halle en el operando.

El caso distinto lo producen los saltos condicionales: hay numerosos operadores que se basan en inspeccionar uno o dos bits del **EFLAGS**, tras una instrucción de comparación entre dos operandos.

Es decir, primero nos encontramos con una instrucción que compara de forma lógica dos operandos, por ejemplo, **CMP op1, op2** o **TEST op1, op2**:

CMP EAX, EBX

y a continuación, vendrá la instrucción de salto condicional.

Como hemos comentado, son bastantes. Afortunadamente, todas empiezan por J, por lo que son bastante identificables. Ejemplos, supongamos que hacemos **CMP EAX, EBX**, veamos unos cuantos:

JE ;	Salta si son iguales
JZ ;	Salta si la comparación da 0
JNE ;	Salta si no son iguales
JNZ ;	Salta si la comparación no da 0
JA ;	Salta si EAX es mayor que EBX
JNBE ;	Salta si EAX no es menor o igual que EBX
JL ;	Salta si EAX es menor que EBX
JNGE ;	Salta si EAX no es mayor o igual a EBX

Para ver una tabla comprensiva visitar nota⁷⁶.

10.4.4. OPERADORES DE PILA Y FUNCIONES

Los operadores de pila se han visto de forma extensiva en el apartado de la memoria del material del módulo:

PUSH Y POP

Respectivamente, PUSH mete el valor apuntado por el operando en la pila o, al contrario, POP saca el valor al que esté apuntando el ESP hacia el operando, que casi siempre será un registro.

CALL

Básicamente, cambia el **EIP (instruction pointer)** a la dirección de la función a la que estemos llamando, con lo cual el procesador comienza a ejecutar código a partir de ahí. Antes de efectuar el salto meterá la dirección siguiente (pasada la instrucción CALL) como dirección de retorno.

Es decir, sintéticamente hace algo similar a:

EIP ->	PUSH EIP+8
EIP+4 ->	CALL función
EIP+8 ->	...siguiente instrucción...

Decimos sintéticamente porque ese PUSH de la dirección de retorno no se ve en el código, solo en

⁷⁶ <http://www.unixwiz.net/techtips/x86-jumps.html>

ejecución veremos la dirección de retorno (el EIP+8) entrar en la pila.

LEAVE Y RET

Se usan cuando se retorna de una función o procedimiento. En concreto, LEAVE lo que hace es restaurar el espacio de pila que la función saliente ha utilizado para su cometido y lo hace con un simple movimiento, algo similar a:

MOV ESP, EBP
POP EBP

Es decir, “baja” el puntero a la cima de la pila a donde estaba el marco de pila anterior (ver apartado de la pila en los materiales del módulo). Es como limpiar la zona de camping en la que uno ha pasado la tarde de picnic.

RET, por su parte, hace una especie de POP de la dirección que CALL guardó que no es otra que la dirección de retorno guardada en la pila y que ahora ira a parar a EIP para proseguir la ejecución.

Especialmente cuidado con el LEAVE, que muchas veces no está presente y la restauración se hace tal y como se ha indicado, con el MOV y el POP para deshacernos de dicho valor en la pila y que el siguiente al que apunta el ESP (ya restaurado) sea la dirección de retorno que RET va a poner en EIP.

INT

La instrucción INT provoca una interrupción en el procesador. Esto está directamente relacionado con las llamadas al sistema, que no se ejecutan en el proceso de usuario sino en el del kernel o núcleo.

En Linux 32 bits, por ejemplo, es común ver la instrucción

INT 0x80

cuando se está llamando a una *syscall* o llamada de sistema.

10.5. LEER EL DESENSAMBLADO

La lectura de código estático de un desensamblado suele estar acompañada de herramientas de análisis tipo IDAPro, Ghidra o radare2. De otra forma, es fácil perder el hilo o tener que volver sobre nuestros pasos una y otra vez.

Con el tiempo, el ojo entrenado sabe distinguir donde está el prologo de una función o el epílogo con solo mirar un grupo de instrucciones. O, algo común, distinguir estructuras de datos, flujos y bucles.

No se trata por lo tanto de programar en ensamblador sino de tener soltura a la hora de interpretar bloques de instrucciones donde se sospecha “esta ocurriendo algo”.

En el análisis de malware, donde no está disponible el código en el 99% de ocasiones, es fundamental, dado que todo se orienta alrededor del desensamblado y se avanza con el continuo ciclo de ejecuciones y breakpoints para tratar de arrancar el comportamiento de un ejemplar de malware.

En la búsqueda de bugs, aunque dispongamos del código, es habitual tener que compilar dicho código fuente con los símbolos de depuración e intentar ver su comportamiento en un análisis dinámico. De nuevo, manejarnos con el ensamblador (y el depurador, que no es algo trivial) será fundamental. Es decir, encontramos un bug prometedor viendo código, pero probamos nuestras premisas delante de un depurador.

Ya lo dijimos: el ensamblador es la lengua franca del reversing. Aprender dicha lengua y manejarnos con soltura es indispensable.