5.2.—Estructura de pila (Stack Frame)

Debido a que IDA es una herramienta de análisis de lenguaje de bajo nivel, algunas de sus características y vistas utilizan detalles familiarizados con los lenguajes compilados de bajo nivel, algunas de las cuales se centran en especificaciones del lenguaje máquina generado y otras en la administración de la memoria utilizada por el programa con el lenguaje de alto nivel. Por esto mismo y a fin de comprender mejor algunas de las vistas de IDA, será necesario de vez en cuando explicar un poco de teoría relacionada con los programas compilados, a fin de que tenga sentido lo que IDA nos muestra.

Uno de los conceptos relacionados con los lenguajes de bajo nivel es la estructura de pila o "**Stack Frame**", yo prefiero llamarle estructura de pila ya que la idea de almacenamiento de pila la prefiero utilizar para el "**heap**", por lo tanto y para no crear controversias desde ahora nos referiremos a él en inglés **Stack Frame**. El stack frame como ya apuntamos son bloques de memoria, distribuidos en la **pila (stack)** de un programa en ejecución, el cual está dedicado a especificar la forma en que se invocará a una función. Los programadores normalmente agrupan las declaraciones ejecutables en unidades llamadas funciones, también pueden llamarse declaraciones, subrutinas o métodos, según sea el caso del lenguaje utilizado. En cualquier caso es considerada una buena práctica de programación construir los programas como unidades funcionales.

Cuando una función no se está ejecutando, normalmente necesita muy poca o ninguna memoria. En contraposición cuando una función es llamada, podrá necesitar memoria por distintas razones. Primero, si el llamador de una función quiere pasarle información en forma de parámetros, argumentos, a la función. Dichos parámetros necesitarán ser guardados en alguna ubicación para que la función los pueda encontrar. Segundo, la función puede necesitar temporalmente espacio para guardar información mientras realiza el proceso de su tarea. Dicho espacio temporal a menudo es distribuido por el programador cuando declara las variables locales utilizadas en la función, pero que no se podrá acceder a ellas una vez se haya completado la tarea de dicha función.

Debido a lo expuesto antes, los compiladores utilizan el **stack frame**, también llamado **activation records**, para colocar o sacar los parámetros de una función y sus variables locales tal como lo ha querido el programador. Los pasos de dicha acción son los siguientes; el compilador inserta código para que coloque los parámetros de una función en el stack frame antes de transferirle el control de ejecución a la propia función, una vez realizado, el compilador inserta código para que se distribuya la memoria suficiente para albergar las variables locales de la función. Estas dos acciones dan como resultado la construcción de los stack frame, un elemento a añadir el cual se guarda también en el stack frame es la dirección en la cual la función deberá retornar. El resultado de utilizar los stack frames es que nos posibilita la recursión, ya que cada llamada recursiva a una función nos viene dada en el stack frame, separando limpiamente cada llamada realizada por su predecesor. Vamos a detallar los pasos que detallan las operaciones que tienen lugar cuando se llama a una función:

1. El llamador coloca cualquier parámetro requerido por la función existente en las ubicaciones llamadas, las cuales son dictadas por los acuerdos de llamada, **Calling Conventions** hablaremos de ellas a continuación, empleadas por la función llamada. Esta operación da como resultado un cambio en el puntero de pila si dichos parámetros son colocados en tiempo de ejecución en la pila.

2. El llamador transfiere el control a la función existente llamada. Esta acción se realiza normalmente con instrucciones tipo en **x86 CALL** o en **MIPS JAL**. Normalmente se guarda una dirección de retorno en la pila del programa o en un registro CPU.

3. Si es necesario, la función llamada realiza los pasos necesarios para configurar un **frame pointer** (puntero a la estructura). ("Un frame pointer es un registro que apunta a una ubicación dentro del stack frame. Normalmente las variables en el stack frame están referenciadas por su distancia relativa que es desde su ubicación al punto apuntado por el frame pointer"). Y guarda cualquier valor de registro que el llamador espera obtener, para que permanezcan inalterados.

4. La función llamada distribuye el espacio necesario para cualquiera de las variables locales que se requieran. Esto se realiza para ajustar el puntero de pila del programa de manera que se reserve el espacio necesario para ellas en la pila en tiempo de ejecución.

5. La función llamada ejecuta dichas operaciones, generando potencialmente un resultado. En el transcurso de estas operaciones, la función llamada puede acceder a los parámetros pasados a ella. Si la función retorna un resultado, el resultado es a menudo colocado en un registro específico o en registros que el llamador puede examinar una vez que la función haya retornado valor.

6. Una vez que la función ha completado estas operaciones, cualquier espacio de la pila reservado para las locales variables es limpiado. Esta acción se realiza para deshacer lo realizado en el **paso 4**.

7. Cualquier registro cuyo valor fue guardado, **paso 3**, por orden del llamador son restaurados a sus valores originales. Esto también incluye la restauración del registro del **frame pointer** del llamador.

8. La función llamada retorna el control al llamador. Normalmente las instrucciones para realizarlo son en **x86 RET** y en **MIPS JR**. Dependiendo de los acuerdos de llamada utilizados, estas operaciones también pueden utilizarse para borrar uno o más parámetros de la pila de programa.

9. Una vez que el llamador retoma el control, puede necesitar quitar parámetros de la pila de programa. En tales casos puede ser necesario un reajuste de la pila para restaurar el puntero de pila al valor que tenía antes del **paso 1**.

Los pasos **3 y 4** se realizan siempre así para la entrada de una función, el conjunto de estos pasos son llamado **prólogo de la función**. De forma similar los pasos del **6 al 8** son ejecutados al final de una función y reciben el nombre de **epílogo de la función**. Por lo tanto exceptuando el paso 5, el cual representa el cuerpo de la función, Todas estas operaciones constituyen la realización de una llamada a una función.

5.2.1.—Acuerdos de llamada (Calling Conventions)

Una vez hemos comprendido básicamente que es el **stack frame**, podemos finalizar viendo exactamente como están estructurados. Los siguientes ejemplos que veremos están referenciados con la arquitectura **x86** y asimismo asociados a compiladores de x86 como **Microsoft Visual C/C++ o GNU gcc/g++**. Uno de los pasos más importantes en la creación del stack frame se refiere a la colocación de los parámetros de la función en la pila por la función de llamada. La función de llamada debe guardar los parámetros de la misma forma que la función a la que se está llamando espera encontrarlos; si no es así nos creará problemas graves. Las funciones advierten la manera en que esperan recibir sus argumentos seleccionando y realizando unos acuerdos específicos de llamada.

Un acuerdo de llamada dicta exactamente como un llamador deberá colocar cualquier parámetro que la función requiera. Dichos acuerdos pueden necesitar colocar parámetros en registros específicos, en la pila del programa o en ambos registros y pila. Es igual de importante, indicar quien es el responsable de quitar los parámetros de la pila una vez pasados y ejecutados por la función. Algunos acuerdos de llamada dictan que el llamador es el responsable de quitar los parámetros colocados en la pila, mientras que otros acuerdos de llamada dictan que la función llamada deberá encargarse de limpiar los parámetros de la pila. Realizar los acuerdos de llamada correctamente es esencial para mantener la integridad del puntero de pila del programa.

5.2.1.1 Acuerdo de llamada en C

Los acuerdos de llamada por defecto utilizados en la mayoría de los compiladores C para arquitectura x86 es llamada la **C calling convention**. El modificador **_cdecl** puede verse en los programas **C/C++** como éste fuerza a utilizar a los compiladores el acuerdo de llamada C, sobrescribiendo el acuerdo de llamada por defecto. Desde ahora nos referiremos a este acuerdo de llamada con el nombre de **cdecl**. El acuerdo de llamada **cdecl** especifica que el llamador coloque los parámetros de una función en la pila en el orden de derecha a izquierda y el llamador sea el encargado de quitar los parámetros de la pila después de que la función llamada se haya realizado.

Uno de los resultados de colocar los parámetros en la pila de derecha a izquierda es que el parámetro de la función más a la izquierda (el primero) siempre estará en la parte superior de la pila cuando la función sea llamada. Esto hace que el primer parámetro, por muchos que tenga la función, sea fácil de encontrar con lo cual hace que **cdecl** sea ideal para utilizarla con funciones que toman un número de argumentos variable, como puede ser **printf.**

El requisito de que el llamador de la función quite los parámetros de la pila, significa a menudo mirar a las instrucciones que realizan el ajuste del puntero de pila del programa inmediatamente después del retorno de la función llamada. En el caso de las funciones que no acepten un número de argumentos variable, el llamador es el apropiado para hacer los ajustes, ya que el llamador sabe exactamente cuántos argumentos ha escogido para pasárselos a la función, por lo tanto puede efectuar fácilmente el ajuste correcto. En contrapartida la función llamada nunca sabe por adelantado cuántos parámetros puede recibir y por lo tanto le sería difícil realizar el ajuste necesario de la pila.

En los siguientes ejemplos consideraremos la llamada a una función con el prototipo siguiente:

void ejemplo_cdecl (int w, int x, int y, int z);

Por defecto esta función utilizara el acuerdo de llamada **cdecl**, esperando el paso de cuatro parámetros empujados a la pila de derecha a izquierda y con el requisito de que el llamador limpie los parámetros de la pila. Un compilador podría generar un código para llamar a esta función, como este:

ejemplo_cdecl (1, 2, 3, 4); //programa llamadas ejemplo_cdecl

```
push 4                  ; empuja al stack el parámetro z
push 3                  ; empuja al stack el parámetro y
push 2                  ; empuja al stack el parámetro x
push 1                  ; empuja al stack el parámetro w
call   ejemplo_cdecl    ; llamada a la función
add    esp,16           ; ajusta esp a su valor anterior
```

Las cuatro operaciones **push** iniciales dan como resultado un cambio de 16 bytes, en una arquitectura de 32-bit **4 * sizeof (int)**, al puntero de pila del programa (**ESP**), lo cual es deshecho con el retorno de **ejemplo_cdecl**. Si la función ejemplo_cdecl es llamada 50 veces, cada llamada será seguida por un ajuste similar al de antes. El siguiente ejemplo también realiza el mismo acuerdo de llamada a **cdecl** pero elimina la necesidad de que el llamador borre explícitamente los parámetros de la pila después de cada llamada a ejemplo_cdecl

```
ejemplo_cdecl (1, 2, 3, 4);      //programa llamada  ejemplo_cdecl
mov  [esp+12], 4        ;mueve parámetro z a la cuarta posición del stack
mov  [esp+8], 3         ;mueve parámetro y a la tercera posición del stack
mov  [esp+4], 2         ;mueve parámetro x a segunda posición del stack
mov  [esp], 1           ;mueve parámetro w parte superior del stack
call   ejemplo_cdecl    ;llamada a la función
```

En este ejemplo el compilador a predistribuido el espacio de almacenamiento para los parámetros que serán pasados a **ejemplo_cdecl** en la parte superior de la pila durante el **prólogo** de la función. Cuando los parámetros para ejemplo_cdecl son colocados en la pila, no se produce ningún cambio en el puntero de pila, lo cual elimina la necesidad de reajustar el puntero de pila cuando la llamada a ejemplo_cdecl se complete. Los compiladores de **GNU, gcc y g++,** utilizan estás técnicas para colocar los parámetros de función en el **stack**. Observa que en cualquiera de los dos métodos el resultado del **stack pointer** apunta al argumento más a la izquierda cuando la función es llamada.

5.2.1.2.—Acuerdo de llamada Standard

El término **standard** es inapropiado ya que es un nombre creado por Microsoft para referirse a su propio acuerdo de llamada el cual utiliza el modificador **_stdcall**, la declaración de una función, sería de la siguiente forma:

void _stdcall ejemplo_stdcall (int w, int x, int y);

Al igual que el anterior modificador, a partir de ahora nos referiremos a este acuerdo de llamada como **stdcall**.

Al igual que con **cdecl, stdcall** requiere que los parámetros de la función sean colocados en la pila en el orden de derecha a izquierda. La diferencia en usar **stdcall** es que la función llamada es la responsable de limpiar los parámetros de la pila cuando la función ha finalizado. Para que una función pueda realizar lo anterior, la función necesita saber exactamente cuantos parámetros hay en la pila. Esto sólo es posible para funciones que tengan un número fijo de parámetros. En consecuencia las funciones de argumentos variables como **printf** no pueden utilizar **stdcall**. La función **ejemplo_stdcall**, espera tres parámetros como números enteros, que ocuparán un total de **12 bytes,** en arquitectura de 32-bit, en la pila **(3 * sizeof (int**)). Un compilador x86 puede utilizar la instrucción **RET** de una forma especial para sacar simultáneamente de la pila, la dirección de retorno de la parte superior de la pila y sumar **12** al puntero de pila limpiando los parámetros de la función. En el caso **ejemplo_stdcall**, podemos ver la siguiente instrucción para retornar al llamador:

```
ret 12                      ;retorna al llamador y limpia 12 bytes del stack
```

La ventaja principal por utilizar **stdcall** es la eliminación de todo el código necesario para limpiar los parámetros del **stack** después de cada llamada de una función, lo cual da como resultado la premisa programa corto, programa rápido. Como acuerdo Microsoft utiliza **stdcall** para todas las funciones de parámetros fijos exportadas de archivos de librería compartida (**DLL**). Este es un punto importante a recordar si lo que intentas es generar prototipos de funciones o reemplazar binarios compatibles con cualquier componente de librería compartida.

5.2.1.3.—Acuerdo de llamada para x86 fastcall

Una variación de **stdcall**, es el acuerdo de llamada **fastcall**. Esta pasa hasta dos parámetros a los registros CPU antes que a la pila del programa. Los compiladores Microsoft Visual C/C++ y los GNU gcc/g++ (versión 3.4 y posteriores) reconocen el modificador **fastcall** en las declaraciones de función. Cuando una **fastcall** se especifica, el primero de los dos parámetros pasados a la función será colocado en los registros **ECX y EDX**, respectivamente. Todos los parámetros restantes serán colocados en la pila con el orden de derecha a izquierda igual que en las **stdcall**. También al igual que en **stdcall**, en **fastcall** las funciones son las responsables de quitar los parámetros de la pila cuando se retorna al llamador. La siguiente declaración te muestra como se utiliza el modificador **fastcall**.

```
void fastcall ejemplo_fastcall (int w, int x, int y, int z);
```

Un compilador podría generar el código siguiente para llamar a **ejemplo_fastcall**:

```
ejemplo_fastcall (1, 2, 3, 4)       //programa llamadas ejemplo_fastcall
push 4                      ;mueve parámetro z a segunda posición de la pila
push 3                      ;mueve parámetro y parte superior de la pila
mov  edx, 2                 ;mueve parámetro x a EDX
mov  ecx, 1                 ;mueve parámetro w a ECX
call   ejemplo_fastcall     ;llamada a la función
```

Observa que no se requiere ningún ajuste de retorno después de la llamada, esto se debe a que **ejemplo_fastcall** es el responsable de limpiar los parámetros **y** y **z** de la pila al retornar al llamador. Es importante comprender lo siguiente, como que dos de los argumentos han sido pasados a los registros, la función llamada sólo necesitará limpiar **8 bytes** de la pila, aunque existan cuatro argumentos a la función.

5.2.1.4.—Acuerdos de llamada C++

Las funciones **nonstatic** en las clases C++ difieren de las funciones estándares en que estás deben hacerse disponibles con el puntero **this**, el cual apunta al objeto utilizado para invocar la función. La dirección del objeto utilizado para invocar la función tiene que ser suministrado por el llamador y por lo tanto proporcionado como parámetro al llamar a las funciones **nonstatic**. La norma del lenguaje C++ no especifica cómo **this** debe ser pasado a las funciones nonstatic, así pues no nos debe sorprender que distintos compiladores utilicen distintas técnicas al pasar **this**.

Microsoft Visual C++ nos proporciona el acuerdo de llamada **thiscall**, el cual pasa a **this** al registro **ECX** y requiere que la función **nonstatic** limpie los parámetros de la pila como **stdcal**l. El compilador de **GNU g++** trata a **this** como el primer parámetro implicado a cualquier función **nonstatic** y se comporta en todos los aspectos como **cdecl**. Así pues un código compilado por **g++, this** es colocado en la parte superior de la pila antes de la llamada a la función **nonstatic**, y el llamador es el responsable de quitar los parámetros, siempre que al menos exista uno, de la pila cada vez que la función retorna. Más características del compilador **C++** las iremos viendo más adelante.

5.2.1.5.—Otros acuerdos de llamada

La explicación completa de todos los acuerdos de llamada requeriría quizás un libro entero para exponerlas bien. Dichos acuerdos de llamada son específicos a cada idioma, compilador y CPU, por lo tanto dependerá de tí buscar la adecuada según el compilador que utilices si éste es poco conocido. No obstante algunas situaciones merecen una mención especial, como puede ser: código optimizado, código en lenguaje ensamblador personalizado y las **system calls**.

Cuando una función es exportada para utilizarla otros programadores, como por ejemplo funciones de una librería, es importante que realicen acuerdos de llamada conocidos de modo que a dichos programadores les sea fácil interactuar con dichas funciones. Por otra parte si una función está destinada para uso interno del programa, entonces el acuerdo de llamada para esta función sólo es conocida por las funciones del programa. En tales casos, optimizando los compiladores pueden elegir acuerdos de llamada alternativos a fin de generar el código rápidamente. En los procesos en que esto puede ocurrir, utilizaremos la opción **/GL con Microsoft Visual C**++ y el nombre clave **regparm con GNU gcc/g++.**

Si el programador domina el lenguaje ensamblador, lo que hace es tomar el control completo de los parámetros que serán pasados a cualquier función que se vaya a crear, y a menos que desee que dichas funciones puedan ser utilizadas por otros programadores, tiene la libertad absoluta de pasar los parámetros de la forma que desee. Como consecuencia, tendremos que tener cuidado cuando analicemos código

ensamblado personalizado. Dicho código contiene a menudo rutinas de ofuscación y conjuntos de órdenes programadas.

Una **system call** es una función de llamada especial utilizada para requerir un servicio del sistema operativo. Las llamadas de sistema efectúan un estado de transición entre el **modo usuario** y el **modo kernel** para que el kernel del sistema operativo pueda atender la solicitud del usuario. La forma en que se inician estas llamadas varía según el sistema operativo y la CPU¨. Por ejemplo, en **Linux x86** las **system calls** se inician utilizando la instrucción **int 0x80**, mientras que otros sistemas operativos **x86** utilizan la instrucción **sysenter**. En cualquier sistema x86, Linux es una excepción, los parámetros para las llamadas de sistema son colocados en la pila en tiempo de ejecución, y un número de la **system call** es colocado en el registro **EAX** inmediatamente antes de inicializar la system call. Las llamadas de sistema en Linux aceptan sus parámetros en registros específicos y ocasionalmente en la pila del programa cuando existen más parámetros que registros habilitados.

5.2.2.—Esquema de variable local

A diferencia de los acuerdos de llamada que dictan la forma en que los parámetros serán pasados a la función, no existe ningún tipo de acuerdo que dicte el esquema de las variables locales de una función. La primera tarea de un compilador es mirar y computar la cantidad de espacio necesario que requerirán las variables locales de una función. La segunda tarea es determinar si esas variables se deben distribuir en los registros CPU o en la pila del programa. La forma exacta en que se realicen estas distribuciones es irrelevante para ambas acciones de llamada de una función y a cualquier otra función que se pueda llamar a su vez. En la mayoría de ocasiones es imposible determinar un esquema de variables locales de una función basándose en el estudio del código fuente de dicha función.

5.2.3.—Unos ejemplos de Stack Frame

Consideremos la siguiente función compilada en una máquina basada en 32-bit x86:

```
void tora (int j, int k);   //Una función para llamar
void demo_stackframe (int a, int b, int c)
{
        int x;
        char buffer [64];
        int y;
        int z;
        // cuerpo de la función a llamar
        tora (z, y)
}
```

La cantidad mínima de espacio requerido por las variables locales en la pila es de **76 bytes** ¿Por qué? Veamos tenemos tres enteros **a,b,c** de **4-byte** y un **buffer** de **64-byte, 4*3=12, 12+64 = 76**. Esta función puede utilizar o **stdcall** o **cdecl**, y su **stack frame** será el mismo. La figura abajo, muestra un posible desarrollo de un stack frame para una invocación de la función **demo_stackframe**, asumiendo que no se utiliza ningún **frame pointer**, así pues el puntero de pila **ESP**, servirá como **frame pointer**. Esta estructura podría ser la preparación de la entrada a **demo_stackframe** con una línea como **prólogo**:

sub esp,76 ;distribuirá el espacio suficiente para todas las variables locales

La siguiente columna de **Offset** indica la **dirección base + desplazamiento** requeridos para referenciar cualquiera de las variables locales o parámetros en el **stack frame**.

	Variables	Offset	
Esp→	Z	[esp]	|
	Y	[esp + 4]	|
	buffer	[esp + 8]	|→ variables loc.
	X	[esp + 72]	|
	eip guardado	[esp + 76]	
	a	[esp + 80]	|
	b	[esp + 84]	|→ parámetros
	c	[esp + 88]	|

Generar las funciones que utilizan el puntero de pila para computar todas las referencias variables requiere un poco más de esfuerzo por parte del compilador, debido a que el puntero de pila cambia frecuentemente y el compilador debe asegurarse de proporcionar los **Offset** apropiados todas las veces que se referencia cualquier variable en el **stack frame**. Consideremos la llamada realizada a **tora** en la función **demo_stackframe** el código será el siguiente:

```
push    dword [esp + 4]   ;empuja a la pila a y
push    dword [esp + 4]   ;empuja a la pila a z
call    tora              ;llama a función
add     esp, 8            ;cdecl requiere al llamador para limpiar parámetros
```

El primer **push** introduce la variable local **y** con el Offset mostrado en figura arriba. A primera vista podría parecer que el segundo **push** referencia incorrectamente a la variable local **y** una segunda vez. Sin embargo, debido a que estamos tratando con una estructura basada en **ESP** y el primer **push** modifica el **ESP**, todos los Offset de la figura arriba deberán ajustarse temporalmente cada vez que el **ESP** cambie. Después de haber realizado el primer **push**, el nuevo Offset para la variable local **z** se convierte en **[esp + 4]** la cual está referenciada correctamente por el segundo **push**. Cuando examinemos funciones cuyas variables estén referenciadas en el **stack frame** utilizando el **stack pointer**, deberemos tener cuidado en observar todos los cambios del puntero de pila y ajustar todos los Offset de las variables siguientes de acuerdo con dichos cambios. Una de las ventajas de utilizar el puntero de pila para referenciar todas las variables del **stack frame** es que el resto de registros quedan disponibles para otros propósitos.

Una vez que **demo_stackframe** se ha completado, necesita retornar al llamador. Por lo tanto para finalizar se utilizará la instrucción **ret**, la cual sacará la dirección de retorno deseada de la parte superior de la pila y se pasará al registro que apunte a la pila, en este caso, **EIP**. Además antes de que la dirección de retorno sea sacada, las variables locales necesitan ser limpiadas de la parte superior de la pila para que el puntero de pila apunte

correctamente a la dirección de retorno guardada cuando la instrucción **ret** sea ejecutada. Para nuestra función en particular el **epílogo** sería:

```
add     esp,76    ;ajuste de esp para que apunte a la dirección de retorno
ret               ; retorna al llamador
```

A expensas de utilizar un registro como **frame pointer** y algunas líneas de código para configurarlo en la entrada a una función, la tarea de calcular los Offset de las variables locales puede realizarse fácilmente. En los programas x86, el registro **EBP** es normalmente utilizado como puntero al **stack frame**. Por defecto, la mayoría de los compiladores generan el código utilizando un **frame pointer**, sin embargo existen opciones para indicar que puntero de pila deberá ser utilizado. Por ejemplo, **GNU gcc/g++**, ofrece la opción de compilación **–fomit-frame-pointer**, la cual genera funciones que no dependen de un registro **frame pointer** fijo.

A fin de ver el **stack frame** para **demo_stackframe** utilizando un **frame pointer** fijo, necesitamos considerar este código de **prólogo** nuevo:

```
push    ebp        ; guarda el valor del llamador en ebp
mov     ebp, esp   ; hace que ebp apunte al valor de registro guardado
sub     esp, 76    ; distribuye el espacio necesario para las variables locales
```

La instrucción **push ebp** guarda el valor actual de **ebp** utilizado por el llamador. Los acuerdos de llamada **cdecl y stdcall** permiten a una función modificar los registros **EAX, ECX y EDX** pero para esto se requiere que la función deje inalterados todos los otros registros. Por lo tanto si deseamos utilizar **EBP** como **frame pointer**, debemos guardar el valor actual de **EBP** antes de que lo cambiemos y deberemos restaurar el mismo valor de **EBP** antes que retornemos al llamador. Si es necesario salvar cualquier otro registro que le interese al llamador, por ejemplo **ESI o EDI**, los compiladores pueden optar por; guardarlo al mismo tiempo que es guardado **EBP**, o guardar las variables locales que hayan sido distribuidas. Así pues no existe ninguna norma en la ubicación de los registros guardados en un **stack frame**.

Una vez que **EBP** ha sido guardado, puede ser cambiado para que apunte a la ubicación actual de la pila. Esto se realiza con la instrucción **mov ebp,esp**, la cual copia el valor actual del puntero de pila en **EBP**. Finalmente, como el **stack frame** no está basado en **EBP**, el espacio para las variables locales se distribuye con la instrucción **sub esp,76**. El resultado del esquema del **stack frame** es el siguiente:

	Variables	Offset	
esp →	z	[ebp – 76]	→ variables loc.
	y	[ebp – 72]	
	buffer	[ebp – 68]	
	x	[ebp – 4]	
ebp→	Ebp guardado	[ebp]	registro(s) guardado
	Eip guardado	[ebp + 4]	→ parámetros
	a	[ebp + 8]	
	b	[ebp + 12]	
	c	[ebp + 16]	

Con un **frame pointer** dedicado, todos los Offset de las variables son calculados relativamente al registro **frame pointer**. Es frecuente, pero no necesario, que los Offset positivos (suma) se utilizan para acceder a los parámetros de la función, mientras que si son negativos (resta) se utilizan para acceder a las variables locales. Utilizando un **frame pointer** dedicado, el puntero de pila tiene la libertad de cambiar sin que afecte a los Offset de cualquier variable de la estructura. La llamada a la función **tora** puede ahora ejecutarse de la siguiente forma:

```
push   dword [ebp – 72]     ;empuja a la pila a y
push   dword [ebp – 76]     ;empuja a la pila a z
call   tora                 ;llama a la función
add    esp, 8               ;cdecl requiere al llamador para limpiar parámetros
```

El hecho de que el puntero de pila haya cambiado después del primer **push**, no tiene ningún efecto en el siguiente **push** para que este acceda a la variable local **z**.

Para finalizar el ejemplo, la utilización de un **frame pointer** necesita un **epílogo** ligeramente distinto una vez se ha completado la función, debido a que el frame pointer del llamador debe ser restaurado antes de retornar. Las variables locales deben ser limpiadas de la pila antes de que el valor antiguo del frame pointer sea recuperado, eso se realiza fácilmente por el hecho de que el frame pointer actual apunta al frame pointer antiguo. En los programas x86 que utilizan a EBP como frame pointer, el código siguiente representa un típico epílogo de función:

```
mov    esp, ebp     ;limpia las variables locales restableciendo esp
pop    ebp          ;restaura el valor del llamador a ebp
ret                 ;saca la dirección de retorno al llamador
```

Esta operación es tan común en la arquitectura x86 que nos ofrece una instrucción, **leave**, que realiza la misma tarea anterior.

```
leave               ;copia ebp a esp y coloca el valor a ebp
ret                 ;saca la dirección de retorno al llamador
```

Aunque los nombres de los registros y las instrucciones utilizadas difieran para otras arquitecturas de procesador, el proceso básico de construcción del **stack frame** es el mismo. Independientemente de la arquitectura, lo que nos interesa con todo este desarrollo teórico, es familiarizarnos con las secuencias típicas de **prólogo** y **epílogo** de modo que podamos pasar rápidamente a analizar otro código más interesante en las funciones.

6.2.4.—Vistas de la pila en IDA

El stack frame es desde luego un concepto en tiempo de ejecución; un stack frame no puede existir sin una pila y un programa ejecutándose. Pero aunque esto es verdadero no significa que deba ignorarse el concepto de stack frame cuando se está realizando un análisis estático con herramientas como IDA. Todo el código necesario para preparar el stack frame de cada función está presente en el binario. Con el análisis cuidadoso de este código, podemos tomar una comprensión detallada de la estructura del stack frame de cualquier función aún cuando la función no se esté ejecutando. En realidad, algunos de los análisis más sofisticados que realiza IDA son para determinar el esquema del

stack frame de cada función que IDA desensambla. Durante el análisis inicial, IDA monitoriza líneas de código muy largas para controlar el comportamiento del puntero de pila sobre el curso de una función tomando nota de cualquier operación **push** o **pop** conjuntamente con cualquier operación aritmética que pueda cambiar el puntero pila, ya sea sumando o restando valores constantes. La primera meta de este análisis es determinar el tamaño exacto del espació distribuido para las variables locales de una función. Lo siguiente será determinar si se utiliza un **frame pointer** dedicado para una función dada, reconociendo por ejemplo una secuencia(**push ebp** o **mov ebp, esp**), y también reconocer todas las referencias de memoria a las variables en el stack frame de una función. Por ejemplo, si IDA anota la siguiente instrucción del cuerpo de la función demo_stackframe,

mov eax, [ebp + 8]

comprendería que el primer argumento a la función, **a** en este caso, se cargará en **EAX**, figura abajo

	Variables	Offset	
esp →	z	[ebp – 76]	
	y	[ebp – 72]	→ variables loc.
	buffer	[ebp – 68]	
	x	[ebp – 4]	
ebp→	Ebp guardado	[ebp]	registro(s) guardado
	Eip guardado	[ebp + 4]	
	a	[ebp + 8]	→ parámetros
	b	[ebp + 12]	
	c	[ebp + 16]	

A través de un análisis cuidadoso de la estructura del stack frame, IDA puede distinguir entre las referencias de memoria que acceden a los parámetros de la función, aquellas que están por debajo de la dirección de retorno guardada, y las referencias que acceden a las variables locales, aquellas que están por encima de la dirección de retorno guardada. Además IDA determina cuales son las ubicaciones de memoria referenciadas directamente en el stack frame. Por ejemplo, el stack frame de la figura arriba, tiene un tamaño de 96 bytes, y existen sólo siete variables que están referenciadas, cuatro como locales y tres como parámetros.

Para comprender el comportamiento de una función, a menudo basta con comprender los tipos de datos que manipula la función. Cuando leemos un listado de desensamblado, una de las primeras opciones que tendremos, para comprender la manipulación de datos de una función será ver el corte del stack frame de la función. IDA nos proporciona dos vistas del stack frame en cualquier función: **una vista resumida** y **una vista detallada**. Para que comprendamos estas dos vistas, nos referiremos a la siguiente versión de la función demo_stackframe, la cual ha sido compilada utilizando gcc.

void demo_stackframe (int a, int b, int c)
{
int x = c;
char buffer [64];
int y = b;
int z = 10;
buffer [0] = 'A';
tora (z, y);
}

En este ejemplo hemos dado valores iniciales a las variables **y** y **z** para impedir que el compilador se queje diciéndonos, cuando se realice la llamada a **tora**, que no están inicializadas. Además, hemos guardado un carácter **'A'** como primer elemento del conjunto **buffer**, y hemos decidido no inicializar la variable local **x.** Ahora sí, vamos a ver el desensamblado en IDA y a interpretarlo. El archivo demo_stackframe.exe se adjunta al escrito para poder cargarlo en Ida.

```
IDA View-A
.text:00401090
.text:00401090 ; =============== S U B R O U T I N E =======================================
.text:00401090
.text:00401090 ; Attributes: bp-based frame
.text:00401090
Program control flow 01090 sub_401090      proc near               ; CODE XREF: sub_4010C1+41↓p
.text:00401090
.text:00401090 var_78          = dword ptr -78h
.text:00401090 var_74          = dword ptr -74h
.text:00401090 var_60          = dword ptr -60h
.text:00401090 var_5C          = dword ptr -5Ch
.text:00401090 var_58          = byte ptr -58h
.text:00401090 var_C           = dword ptr -0Ch
.text:00401090 arg_4           = dword ptr  0Ch
.text:00401090 arg_8           = dword ptr  10h
.text:00401090
.text:00401090                 push    ebp
.text:00401091                 mov     ebp, esp
.text:00401093                 sub     esp, 78h
.text:00401096                 mov     eax, [ebp+arg_8]
.text:00401099                 mov     [ebp+var_C], eax
.text:0040109C                 mov     eax, [ebp+arg_4]
.text:0040109F                 mov     [ebp+var_5C], eax
.text:004010A2                 mov     [ebp+var_60], 0Ah
.text:004010A9                 mov     [ebp+var_58], 41h
.text:004010AD                 mov     eax, [ebp+var_5C]
.text:004010B0                 mov     [esp+78h+var_74], eax
.text:004010B4                 mov     eax, [ebp+var_60]
.text:004010B7                 mov     [esp+78h+var_78], eax
.text:004010BA                 call    tora
.text:004010BF                 leave
.text:004010C0                 retn
.text:004010C0 sub_401090      endp
00000490    00401090: sub_401090
```

En este pequeño listado existen muchos puntos de información, pero como nos estamos iniciando ahora, empezaremos por comprender lo explicado anteriormente. He renombrado la llamada a **tora** con su nombre, ya veremos como hacerlo más adelante.

Empezaremos observando la línea siguiente:

```
.text:00401090 ; Attributes: bp-based frame
```

Nos informa de que IDA cree que esta función utiliza el registro **EBP** como **frame pointer** basándose en el análisis del **prólogo** de la función. La siguiente línea

```
.text:00401093                 sub     esp, 78h
```

nos muestra que el compilador ha distribuido un espacio de**120 bytes, 78 en hexa**, en el **stack frame** para las variables locales. En el espacio incluiremos los **8 bytes** de los dos

parámetros pasados a **tora**, lo cual se realiza en las líneas mostradas. En **4010B0** se

```
.text:004010B0                 mov     [esp+78h+var_74], eax
```

```
.text:004010B7                 mov     [esp+78h+var_78], eax
```

le pasa el primer parámetro, **var_74**, y en **4010B7** el segundo parámetro, **var_78**.

Sigamos, aún nos queda mucho para rellenar el espacio de **76 bytes** que estimamos antes y además demostraremos como los compiladores rellenan el espacio de las variables locales con bytes extras para conseguir un alineamiento particular en el **stack frame**. Sigamos, al inicio, esta parte del listado, IDA nos proporciona un resumen de la

```
.text:00401090 var_78          = dword ptr -78h
.text:00401090 var_74          = dword ptr -74h
.text:00401090 var_60          = dword ptr -60h
.text:00401090 var_5C          = dword ptr -5Ch
.text:00401090 var_58          = byte ptr -58h
.text:00401090 var_C           = dword ptr -0Ch
.text:00401090 arg_4           = dword ptr  0Ch
.text:00401090 arg_8           = dword ptr  10h
```

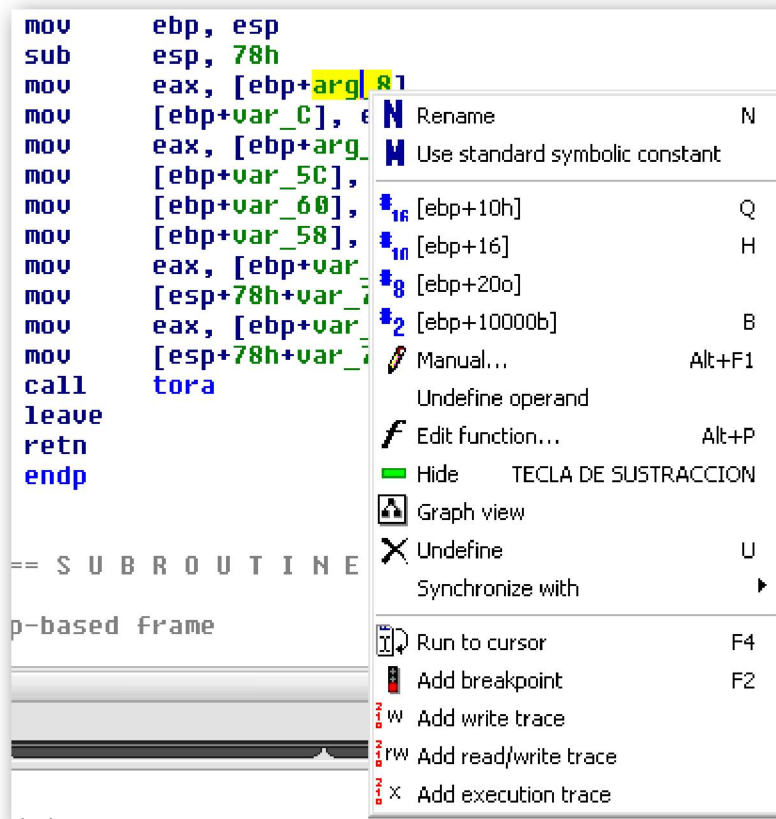
vista de la pila donde lista cada variable referenciada directamente dentro del stack frame, conjuntamente con su tamaño (**byte, Word, dword**) y su distancia con respecto (**Offset**) al **frame pointer**.

IDA asigna nombres a las variables basándose en su ubicación relativa respecto a la dirección de retorno guardada. Los nombres de las variables locales tienen como prefijo, derivado de su nombre genérico, **var_** al cual se le añade un sufijo hexadecimal que indica la distancia, en bytes, con respecto a las variables posicionadas por encima del **frame pointer** guardado. La variable local **var_C**, es este caso, es de **4-byte (dword)** dicha variable está posicionada **12 bytes** por encima del frame pointer guardado **([ebp – 0Ch]**). Los nombres de los parámetros de la función se generan utilizando el prefijo **arg_** combinado con el sufijo hexadecimal el cual representa la distancia relativa al parámetro más alto. De esta forma el parámetro más alto de **4 bytes** podría nombrarse **arg_0**, mientras que los parámetros sucesivos podrían nombrarse arg_**4, arg_8, arg_C** y sucesivamente. En nuestro ejemplo **arg_0** no está listado porque la función no utiliza el parámetro **a**. Esto se produce porque IDA no consigue localizar la referencia en memoria de **ebp + 8**, ubicación del primer parámetro, por lo tanto **arg_0** no es listado en la vista del resumen de pila. Si hacemos un repaso rápido del resumen de la vista de pila nos revela que existen algunas ubicaciones de pila que IDA no ha podido nombrar esto es debido a que dichas ubicaciones no están referenciadas directamente en el código del programa.

Observación: IDA solamente generará automáticamente nombres, para aquellas variables de la pila las cuales estén directamente referenciadas en una función.

Una diferencia importante entre el listado de desensamblado de IDAy el análisis de stack frame que realizamos nosotros anteriormente, es el hecho de que en el listado de desensamblado no vemos referencias de memoria similares a **[ebp – 12]**. En vez de eso IDA ha reemplazado todos los Offset constantes con nombres simbólicos

correspondientes a los símbolos del resumen de la vista de pila y sus Offset relativos al puntero del stack frame. Esto se corresponde con la meta de IDA para generar un desensamblado de más alto nivel. Es muy fácil trabajar con nombres simbólicos de constantes numéricas. De hecho, como veremos posteriormente, IDA nos permite cambiar cualquier nombre de cualquier variable de la pila por el que nosotros deseemos, haciéndonos muy fácil el poder recordarlos. El resumen de la vista de pila es como un plano de los nombres generados por IDA para saber sus correspondientes Offset en el stack frame. Por ejemplo, donde aparezca la referencia de memoria **[ebp + arg_8]** en el desensamblador, puede utilizarse **[ebp + 10] o [ebp + 16].** Si prefieres los Offset numéricos, IDA te los puede mostrar. Para que los muestre haces click izquierdo sobre **arg_8** y te mostrará un menú contextual, figura abajo, el cual contiene varias opciones para cambiar el formato de la información mostrada.



En nuestro ejemplo, como disponemos del código fuente para compararlo, podemos combinar los nombres generados por IDA con los nombres reales de la fuente original utilizando una gran variedad de pistas en el desensamblado. Hagámoslo.

1. En primer lugar, **demo_stackframe** toma tres parámetros: **a, b** y **c**. Estos corresponden a las variables **arg_0, arg_4** y **arg_8** respectivamente, sin embargo como ya hemos dicho **arg_0** no está en el desensamblado porque nunca es referenciada.
2. La variable local **x** es inicializada con el parámetro **c**. Por lo tanto **var_C** corresponderá a **x** desde que es inicializada por **arg_8** en la línea

```
.text:00401096                 mov     eax, [ebp+arg_8]
.text:00401099                 mov     [ebp+var_C], eax
```

3. De forma similar, la variable local **y** es inicializada con el parámetro **b**. Por lo tanto **var_5C** corresponderá a **y** desde que es inicializada por **arg_4** en la línea

```
.text:0040109C                 mov     eax, [ebp+arg_4]
.text:0040109F                 mov     [ebp+var_5C], eax
```

4. Igualmente la variable local **z** corresponde a **var_60** desde que es inicializada con el valor **10** en la línea

```
.text:004010A2                 mov     [ebp+var_60], 0Ah
```

5. El conjunto de caracteres **buffer**, de 64-byte, comienza en **var_58** desde que **buffer[0]** es inicializado con una **A**, ASCII 0x41, en la línea

```
.text:004010A9                 mov     [ebp+var_58], 41h
```

6. Los dos argumentos para la llamada a **tora** son movidos a la pila, en las líneas

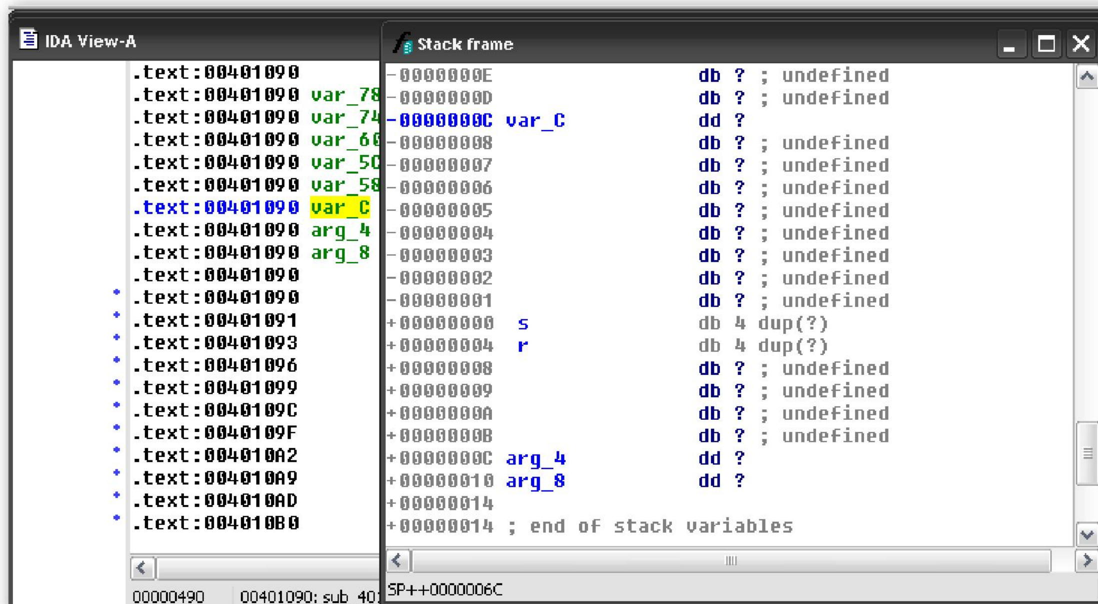
```
.text:004010B0                 mov     [esp+78h+var_74], eax
```

```
.text:004010B7                 mov     [esp+78h+var_78], eax
```

antes de ser empujados en la pila. Esto es normal en las actuales versiones de gcc, versiones 3.4 y posteriores. Lo que parecen ser las variables locales **var_74** y **var_78** son en realidad ubicaciones de la pila reservadas por el compilador para la colocación de parámetros de la función antes de llamar a la función.

Puedes percatarte que la sintaxis utilizada para referenciar **var_78** y **var_74** difiere de la sintaxis utilizada para referenciar otras variables, como **var_60**. IDA para referenciar a **var_60** utiliza la sintaxis **ebp + var_60**. Examinando la vista resumida de la pila, comprenderemos que esto iguala a **ebp – 60h**. En cambio la sintaxis para referirse a **var_78** parece más complicada si la comparamos con las otras: **esp + 78h + var_78**. La diferencia menos sutil es el hecho de que **var_78** está referenciado relativamente a **ESP** en lugar de **EBP**, sin embargo todos los nombres simbólicos se representan con Offset del puntero de estructura **EBP**. Como consecuencia, IDA realiza un truco con el fin de poder utilizar el nombre simbólico **var_78**. La dirección **esp + var_78** será incorrecta al querer igualar con **esp – 78h**. Para poder utilizar el nombre simbólico de la variante, se debe añadir una constante **+78h** para que matemáticamente sea correcto, quedando por lo tanto: **[esp + 78h – 78h]**, lo cual reduce a **[esp]** correctamente. Este truco de bits es importante recordarlo, debido a que también se utilizará cuando una función no utilice un **frame pointer** y todas las variables estén referenciadas relativamente al puntero de pila. En dichos casos IDA genera Offset variables relativos a la dirección de retorno guardada antes de que sea guardado el **frame pointer**, requiriendo el mismo tipo de ajuste realizado antes, para cada referencia relativa al puntero de pila.

En la segunda vista que nos proporciona IDA del **stack frame**, es la **vista detallada** en la cual cada byte distribuido en el stack frame es individualmente explicado. A esta vista detallada se puede acceder realizando doble click en cualquier nombre de variable asociada con el stack frame que estemos estudiando. Realizando doble click en **var_C** en el listado anterior nos abrirá una vista del stack frame como se puede ver en figura abajo, como ya sabemos la tecla **ESC** cierra la ventana.



Como vemos debido a que esta vista explica cada byte en el stack frame, ocupa mucho más espacio que la vista resumida, la cual sólo referencia las variables. Observa que a los bytes no referenciados directamente dentro de la función no se les asigna nombre. Por ejemplo, el parámetro **a**, correspondiente a **arg_0**, o está porque nunca se referenciará en **demo_stackframe**. Sin ninguna referencia en memoria al analizar, IDA opta no hacer nada con dichos bytes del stack frame, los cuales ocupan los Offset **+00000008** hasta **+0000000B**. Por otro lado vemos que **arg_4** ha sido referenciado, línea abajo, en el listado de desensamblado, donde su contenido será cargado al

```
.text:0040109C                 mov     eax, [ebp+arg_4]
.text:0040109F                 mov     [ebp+var_5C], eax
```

registro EAX de 32-bit. Basándose en el hecho de que 32 bits de datos serán movidos, IDA será capaz de intuir que **arg_4** vale la cantidad de **4-bytes** y lo etiqueta como tal, **db** define un **byte** guardado, **dw** define dos bytes guardados un **Word** y **dd** define 4 bytes guardados un **double Word**.

Dos valores especiales mostrados en la figura anterior son '**s**' y '**r**', fíjate que cada uno tiene un espacio en blanco al inicio. Estas pseudo variables de IDA representan: la '**r**' es la dirección de retorno guardada y la '**s**' es el valor o valores guardados del o de los registros, en este ejmplo '**s**' representa solamente a **EBP**. Estos valores se incluyen en la vista detallada de stack frame para completar la explicación de los bytes del mismo.

Esta vista del stack frame permite realizar una mirada al interior del trabajo realizado por los compiladores. En, figura abajo vemos claramente que el compilador ha insertado

```
.text:00401090
.text:00401090 var_78          = dword ptr -78h
.text:00401090 var_74          = dword ptr -74h
.text:00401090 var_60          = dword ptr -60h
.text:00401090 var_5C          = dword ptr -5Ch
.text:00401090 var_58          = byte ptr -58h
.text:00401090 var_C           = dword ptr -0Ch
.text:00401090 arg_4           = dword ptr  0Ch
.text:00401090 arg_8           = dword ptr  10h
.text:00401090
```

8 bytes extras entre el frame pointer '**s**' y la variable local **x (var_C)**. Estos bytes ocupan los Offset del -**00000001** hasta -**00000008** en el stack frame. Además, con una pequeña ejecución matemática en el Offset asociado con cada variable listada en el resumen de pila nos revela que el compilador ha distribuido **76 bytes**, en vez de **64 bytes** ,como indica el código fuente, al **buffer** de caracteres en **var_58**. La colocación de estos bytes extras se realiza en la mayoría de casos como relleno para conseguir una alineación y normalmente la presencia de dichos bytes no repercuten en el comportamiento del programa. Bien hasta aquí de momento sobre stack frame, más adelante retomaremos el tema para usos más complejos con conjuntos (arrays) y estructuras.

Performance Bigundill@