15.2.3 Tipos de dato útiles del SDK

Las Api de Ida definen un número de clases C++ diseñadas para cualquier componente típico existente en archivos ejecutables. El SDK contiene clases para describir funciones, secciones de programa, estructuras de datos, instrucciones en lenguaje ensamblador y operandos dentro de cada instrucción. Otras clases están definidas para ejecutar las herramientas que utiliza ida para la manipulación del proceso de desensamblado. Las clases que se encuentran en esta última categoría definen las características de la base de datos, del módulo de carga, del módulo de procesador, del módulo plug-in y de la sintaxis de ensamblado que se utilizará para cada instrucción.

Algunas de las clases de propósito general más comunes las describiremos a continuación. No explicaremos las clases que son más específicas a los módulos de carga, procesador y plug-in ya que estas las explicaremos en sus partes correspondientes más adelante. Nuestra meta ahora es explicar las clases, sus propósitos y ciertos elementos dato importante de cada clase. Las funciones útiles para manipular cada clase serán descritas en el próximo apartado 15.2.4-- "Funciones SDK de uso común".

area_t (area.hpp)
Esta **struct** describe un rango de direcciones y es la clase base de otras distintas clases. La struct contiene dos elementos dato, **startEA** (inclusivo) y **endEA** (exclusivo), los cuales definen los límites del rango de las direcciones. Las funciones con dichos elementos se definen para calcular el tamaño del rango de direcciones y poder ejecutar comparaciones entre dos áreas.

func_t (funcs.hpp)
Esta clase es una subclase de **area_t**. Los campos de datos añadidos a la clase se utilizan para registrar los atributos binarios de la función, tales como si la función utiliza un **frame pointer** o no y describir las **variables locales y argumentos** de las funciones. Como optimización, algunos compiladores parten funciones en distintas áreas no contiguas dentro de un binario. IDA a estas áreas les da el nombre de **chunks** o **tails**. La clase **func_t** también es utilizada para describir dichos trozos de funciones.

segment_t (segment.hpp)
La **clase segment_t** es otra subclase de **area_t**. Los campos de datos describen el nombre del segmento, sus permisos y su efecto en el segmento (leer, escribir y ejecutar), el tipo del segmento (code, data, etc) y el número de bit utilizados en una dirección del segmento (16,32 o 64).

idc_value_t (expr.hpp)
Esta clase describe los contenidos de un valor IDC, que pueda contener en algún momento un valor cadena, entero o punto flotante. El tipo es utilizado extensamente cuando interactuamos con funciones IDC desde dentro de un módulo compilado.

idainfo (ida.hpp)
Esta **struct** está llena de descripciones de las características de una base de datos cargada. Una variable global de nombre **inf** y de tipo **idainfo**, es declarada en **ida.hpp**. Los campos dentro de esta estructura describen el nombre del módulo de procesador que se está utilizando, el tipo de archivo cargado como **f_PE** o **f_MACHO** vía **filetype_t**, el "**entry point**" del programa **begin_EA**, la dirección mínima dentro del binario **minEA**, la dirección máxima del programa **maxEA**, los **endianness** (formato en

el que se almacenan los datos) del procesador actual **mf** y un número de configuración analizada desde **ida.cfg**.

struc_t (struct.hpp)
Esta clase describe el esquema de datos estructurados dentro de un desensamblado. Es utilizada para describir estructuras dentro de la ventana **Structures** así como para describir la composición del **stack frame** de una función. Una **struct_t** contiene los **flags** que describen los atributos de la estructura y también contiene un conjunto de los elementos de la estructura.

member_t (struct.hpp)
Esta clase describe un elemento tipo dato estructurado. Los campos de datos incluidos describen el offset del byte, tanto de inicio como final, del elemento dentro de la estructura matriz.

op_t (ua.hpp)
Esta clase describe un operando dentro de una instrucción desensamblada. La clase contiene un campo con base cero para almacenar el número del operando (**n**), un campo de tipo de operando (**type**) y otros campos cuyo significado variará dependiendo del tipo de operando. El campo **type** es habilitado a uno por las constantes **optype_t** definidas en **ua.hpp** y describe el tipo de operando o el modo de direccionamiento utilizado por el operando.

insn_t (ua.hpp)
Esta clase contiene la información para describir una instrucción desensamblada. Los campos dentro de la clase describen la dirección de la instrucción dentro del desensamblado (**ea**), el tipo de la instrucción (**itype**), la longitud de la instrucción en byte (**size**) y un conjunto de seis posibles valores de operando (**operands**) del tipo **op_t** (ida limita cada instrucción para un máximo de seis operandos). El campo **itype** es habilitado por el módulo de procesador. Para los módulos de procesador estándares de IDA, el campo **itype** está habilitado a uno por las constantes definidas en **allins.hpp**. Cuando se utiliza un módulo de procesador no estándar, tiene que existir una lista de valores potenciales para **itype** los cuales deberán de obtenerse desde el módulo desarrollado. Digamos que generalmente el campo **itype** no tiene ninguna relación con el opcode binario de la instrucción.

El listado anterior no es en ningún modo una guía definitiva de todos los tipos de dato utilizados en SDK. Solamente es una propuesta como introducción en algunas de las clases más utilizadas y los campos accedidos normalmente dentro de esas clases.

15.2.4.—Funciones SDK de uso común

Aunque el SDK esté programado utilizando C++ y defina distintas clases de C++, en muchos casos el SDK favorece a las funciones de estilo C para ejecutar la manipulación de objetos dentro de una base de datos. En la mayor parte de API, es común encontrar funciones que requieren un puntero a un objeto que se utiliza para encontrar el elemento del objeto a manipular de la manera que deseemos.

En el siguiente resumen, cubriremos las funciones API que proporcionan una funcionalidad similar a algunas de las funciones IDC explicadas en el escrito 14.

Desafortunadamente las funciones que ejecutan tareas idénticas, tienen distintos nombres para IDC y para la API.

15.2.4.1.—Acceso básico a la base de datos

Las siguientes funciones, declaradas en **bytes.hpp**, proporcionan el acceso a los **byte**, **word** y **dword** dentro de una base de datos.

uchar get_byte(ea_t addr) Lee el valor del **byte** de la dirección virtual **addr**.
ushort get_word(ea_t addr) Lee el valor del **word** de la dirección virtual **addr**.
ulong get_long(ea_t addr) Lee el valor **dword** de la dirección virtual **addr**.
get_many_bytes(ea_t addr, void *buffer; ssize_t len) Copia **len byte** de **addr** al **buffer** suministrado.
patch_byte(ea_t addr, ulonglong val) Coloca un valor **byte** en la dirección virtual **addr**.
patch_word(long addr, ulonglong val) Coloca un valor **word** en la dirección virtual **addr**.
patch_long(long addr, ulonglong val) Coloca un valor **dword** en dirección virtual **addr**.
patch_many_bytes(ea_t addr, const void *buffer, size_t len) Modifica los valores de la base de datos desde **addr** con **len byte** que se encuentran en el **buffer** suministrado.
ulong get_original_byte(ea_t addr) Lee el valor del byte **original**, antes de modificarlo, de la dirección virtual **addr**.
ulonglong get_original_word(ea_t addr) Lee el valor del word **original** de **addr**.
ulonglong get_original_long(ea_t addr) Lee el valor dword **original** de **addr**.
bool isLoaded(ea_t addr) Retorna **true** si **addr** contiene un dato válido y **false** si no lo es.

Existen otras funciones para acceder a tamaños alternativos de datos. Digamos que las funciones **get_original_XXX** toman el primer valor original, que no es necesariamente el valor de una dirección antes de modificarla. Consideremos el caso, cuando un valor byte es modificado dos veces; mientras se ha trabajado con él ha cambiado tres veces de valor. Después de la segunda modificación, ambos valores el actual y el original son accesibles, pero no hay forma de obtener el segundo valor, que se ha colocado con la primera modificación.

15.2.4.2.—Funciones de interconexión con el usuario

La interacción de ida con el usuario es manejada por una función **dispatcher** llamada **callui**. Se realizan distintas peticiones de servicios de interconexión para pasar una petición de usuario, una de ellas es la constante **ui_notification**_t, a **callui** conjuntamente con todos los parámetros requeridos para dicha petición. Los parámetros requeridos para cada tipo de petición están especificados en **kernwin.hpp**. Afortunadamente, muchas funciones de las que necesitamos que oculten muchos detalles de la utilización de **callui**, también están definidas en **kernwin.hpp**. Algunas de estas funciones se describen seguidamente:

msg(char *format, …) Imprime un mensaje formateado en la ventana de mensajes. Esta función es análoga a la función en C **printf** y acepta el formato de cadenas **printf**
warning(char *format, …) Muestra un mensaje formateado en un diálogo.
char *askstr(int hist, char *default, char *format, …) Muestra un diálogo de entrada preguntando al usuario si quiere entrar una cadena. El parámetro **hist** dicta cómo será rellenado el listado historial del diálogo y habilita a uno las constantes **HIST_xxx** definidas en **kernwin.hpp**. El **format** y cualquier parámetro adicional son utilizados para realizar una cadena indicadora.

char *askfile_c(int dosave, char *default, char *prompt, …) Muestra un diálogo de guardar archivo (**dosave= 1**) o abrir archivo (**dosave= 0**), inicialmente muestra el directorio y la extensión de archivo especificada por defecto (como **E:\\windows*.exe**)
Retorna el nombre del archivo seleccionado o **NULL** si el diálogo es cancelado.
askin_c(int default, char *prompt, …) Se le indica al usuario una pregunta de **yes** o **no**, como respuesta por defecto (**1 = yes, 0 = no, - 1 = cancel**). Retorna un entero que representa la respuesta seleccionada.
askUsingForm_c(const char *form, …) El parámetro **form** es la especificación de una cadena ASCII de un diálogo y sus elementos de entrada asociados. Esta función puede ser utilizada para construir elementos de enlace con el usuario a medida, cuando ninguna de las funciones del SDK nos cubra nuestras necesidades. El formato de la cadena **form** se detalla en **kernwin.hpp**.
get_screen_ea( ) Retorna la dirección virtual de la posición actual del cursor.
jumpto(ea_t addr) Salta a la dirección especificada en la ventana de desensamblado.

Con la API disponemos de muchas otras capacidades de enlace con el usuario que únicamente utilizando las disponibles en los scripts IDC, una de ellas puede ser la capacidad para crear a nuestra medida diálogos de listados para su selección de una o varias columnas. Los usuarios que estén interesados en dichas capacidades deberán consultar **kernwin.hpp** y en particular las funciones **choose** y **choose2**.

15.2.4.3.—Manipulación de los nombres (Names) en la base de datos

Las siguientes funciones nos permitirán trabajar con las ubicaciones nombradas dentro de una base de datos:

get_name(ea_t from, ea_t addr, char *namebuf, size_t maxsize) Retorna el nombre asociado con **addr**. Retorna una cadena vacía si la ubicación no ha sido nombrada. Esta función proporciona acceso a los nombres locales cuando **from** es cualquier dirección de la función que contenga **addr**. El nombre es copiado a un buffer de salida proporcionado.
set_name(ea_t addr, char *name, int flags) Asigna el nombre dado a la dirección dada. El nombre es creado con los atributos especificados en los **flags** de "**bitmask**". Los valores posibles de las banderas se describen en **name.hpp**.
get_name_ea(ea_t funcaddr, char *localname) Busca el nombre local dado dentro de la función dada por **funcaddr**. Retorna **BADADDR (-1)** si no existe el nombre dado dentro de la función especificada.

15.2.4.4.— Manipulación de funciones

Las funciones API para acceder a la información sobre el desensamblado de funciones están declaradas en **funcs.hpp**. Las funciones para acceder a la información del **stack frame** están declaradas en **frame.hpp**. Algunas de las funciones más utilizadas se describen seguidamente:

func_t *get_func(ea_t addr) Retorna un puntero a un objeto **func_t** el cual describe el contenido de la función indicada por la dirección.
size_t get_func_qty( ) Retorna el número de funciones presentes en la base de datos.
func_t *getn_func(size_t n) Retorna un puntero a un objeto **func_t** el cual representa la enésima función en la base de datos donde **n** está entre **cero** (incluido) y **get_func_qty ( )** (excluido).
func_t *get_next_func(ea_t addr) Retorna un puntero a un objeto **func_t** el cual describe la siguiente función de la dirección especificada.

get_func_name(ea_t addr, char *name, size_t namesize) Copia el nombre de la función indicada en la dirección **ea** un buffer **name** proporcionado.
struct_t *get_frame(ea_t addr) Retorna un puntero a un objeto **struct_t** el cual describe el **stack frame** de la función que contenga la dirección indicada.

15.2.4.5.—Manipulación de estructuras

La clase **struct_t** es utilizada para acceder al **stack frame** de una función así como a los tipos de dato estructurados definidos dentro de las librerías tipo. Alguna de las funciones básicas para interactuar con las estructuras y con sus elementos asociados las describiremos seguidamente. Muchas de estas funciones utilizan un dato tipo **ID (tid_t)**. El API incluye funciones para el mapeado de una **struct_t** a un **tid_t** asociado y viceversa. Observemos que, ambas clases, la **struct_t** y la **member_t** contienen un elemento dato **tid_t**, de esa forma obtener la información del tipo **ID** es fácil si ya tenemos un puntero válido a un objeto **struct_t** o **member_t**.

tid_t get_struc_id(char *name) Hace una búsqueda del tipo **ID** de una estructura dada por **name**.
struct_t *get_struc(tid_t id) Obtiene un puntero a **struc_t** representando la estructura especificada por el tipo **ID** dado.
asize_t get_struc_size(struc_t *s) Retorna el tamaño en byte de la estructura dada.
member_t *get_member(struc_t *s, asize_t offset) Retorna un puntero a un objeto **member_t** este describe el elemento de la estructura el cual reside en el **offset** especificado, dentro de la estructura dada.
member_t *get_member_by_name(struc_t *s, char *name) Retorna un puntero a un objeto **member_t** el cual describe el elemento de la estructura identificado por **name**.
tid_t add_struc(uval_t index, char *name, bool is_union=false) Añade una nueva estructura con nombre dado por **name** dentro de la lista de estructuras estandares. La estructura es también añadida a la ventana **Structures** con el **index** dado. Si **index** es **BADADDR**, la estructura es añadida como última estructura en la ventana **Structures**.
add_struc_member(struc_t *s, char *name, ea_t Offset, flags_t flags, typeinfo_t, asize_t size) Añade un nuevo elemento con el nombre dado en **name** a la estructura dada. El elemento es añadido dentro de la estructura al **offset** indicado o añadido al final de la estructura si **offset** es igual a **BADADDR**. El parámetro **flags** describe el tipo de dato del elemento nuevo. Los **flags** válidos se definen utilizando las constantes **FF_XXX** descritas en **bytes.hpp**. El parámetro **info** proporciona información adicional de los tipos de dato complejos; para tipos de dato primitivos se habilitará a **NULL**. El tipo de dato **typeinfo_t** está definido en **nalt.hpp**. El parámetro **size** especifica el número de byte ocupados por el elemento nuevo.

15.2.4.6.—Manipulación de segmento

La clase **segment_t** almacena la información relacionada a los distintos segmentos dentro de una base de datos, como **.text** y **.data**, y listado en **View> Open Subviews> Segments window**. Recordemos que IDA utiliza el término **segments** para referirse a las **sections** de distintos formatos de archivo ejecutable tales como **PE** y **ELF**. Las siguientes funciones proporcionan el acceso básico a los objetos **segment_t**. Las funciones restantes que tratan sobre la clase **segment_t** están declaradas en **segment.hpp**.

segment_t *getset(ea_t addr) Retorna un puntero al objeto **segment_t** el cual contiene la dirección dada.

segment_t *ida_export get_segm_by_name(char *name) Retorna un puntero al objeto **segment_t** con el nombre dado.
add_segm(ea_t para, ea_t start, ea_t end, char *name, char *sclass) Crea un segmento nuevo en la base de datos actual. Los límites del segmento se especifican con los parámetros de dirección **start** (incluido) y **end** (excluido), mientras que el nombre del segmento se especifica con el parámetro **name**. La clase del segmento describe el tipo de segmento que se está creando. Las clases predefinidas son **CODE** y **DATA**. Un listado completo de las clases predefinidas se puede hallar en **segment.hpp**. El parámetro **para** describe la dirección base de la sección cuando las direcciones segmentadas (**seg:offset**) se están usando, en cuyo caso **start** y **end** serán interpretados como offset en vez de direcciones virtuales. Cuando no se utilizan direcciones segmentadas, o todos los segmentos son cero, este parámetro debería estar puesto a cero
add_segm_ex(segment_t *s, char *name, char *sclass, int flags) Método alternativo para crear segmentos. Los campos deberán habilitarse de manera que reflejen el rango de direcciones del segmento. El segmento es nombrado y con tipo, de acuerdo a los parámetros **name** y **class**. El parámetro **flags** deberá habilitar a uno los valores **ADDSEG_XXX** definidos en **segment.hpp**.
int get_segm_qty( ) Retorna el número de secciones presentes dentro de la base de datos
segment_t *getnseg(int n) Retorna un puntero a un objeto **segment_t** conteniendo información con respecto a la enésima sección del programa en la base de datos.
int set_segm_name(segment_t *s, char *name, …) Cambia el nombre del segmento dado. El nombre está formado por **name** como una cadena formateada e incorporando cualquier parámetro adicional requerido por dicha cadena.
get_segm_name(ea_t addr, char *name, size_t namesize) Copia el nombre del segmento contenido en la dirección dada al buffer proporcionado **name**. Observemos que **name** puede ser filtrado para reemplazar caracteres que ida considere inválidos, caracteres no especificados como **NameChars** en **ida.cfg**, con un carácter **dummy**. Normalmente un guión bajo en ida.cfg es especificado como SubstChar.
get_segm_name(segment_t *s, char *name, size_t namesize) Copia el nombre potencialmente filtrado del segmento dado en el buffer **name** proporcionado.
get_true_segm_name(segment_t *s, char *name, size_t namesize) Copia exactamente el nombre del segmento dado en el buffer **name** proporcionado, sin filtrar ningún carácter.

Una de las funciones **add_segm** debe utilizarse para crear el segmento actual. Simplemente declarar e inicializar un objeto **segment_t**, no crea en realidad un segmento dentro de la base de datos. Esto es cierto con todas las clases relacionadas como **func_t** y **struc_t**. Estas clases meramente proporcionan los medios convenientes para acceder a los atributos de la entidad de una base de datos. Las funciones apropiadas para crear, modificar o eliminar objetos de la base de datos actual deben ser utilizadas de forma que los cambios sean persistentes en la base de datos.

15.2.4.7.—Referencias cruzadas de código

Las funciones y la enumeración de constantes para utilizar con las referencias cruzadas de código, están definidas en **xref.hpp**. Algunas de ellas las vamos a describir:

get_first_cref_from(ea_t from) Retorna la primera ubicación a la cual la dirección dada transfiere el control. Retorna **BADADDR (-1)** si la dirección dada no se refiere a otras direcciones.
get_next_cref_from(ea_t from, ea_t current) Retorna la siguiente ubicación de la dirección dada (**from**) que transfiera el control, suponiendo que **current** haya sido retornado

previamente por una llamada a **get_first_ref_from** o **get_next_cref_from**. Retorna **BADADDR (-1)** si no existen más referencias cruzadas.
get_first_cref_to(ea_t to) Retorna la primera ubicación que transfiere el control a la dirección dada. Retorna **BADADDR (-1)** si no existen referencias a la dirección dada.
get_next_cref_to(ea_t to, ea_t current) Retorna la siguiente ubicación que transfiere el control a la dirección dada (**to**) suponiendo que **current** ha sido retornado previamente por una llamada a **get_first_cref_to** o **get_next_cref_to**. Retorna **BADADDR** si no existen más referencias cruzadas a la ubicación dada.

15.2.4.8.—Referencias cruzadas de datos
Las funciones para acceder a la información de referencias cruzadas de datos, también declaradas en **xref.hpp**, son muy similares a las funciones utilizadas para acceder a la información de las referencias cruzadas de código. Se describen a continuación:

get_first_dref_from(ea_t from) Retorna la primera ubicación a la cual referencia un valor dato la dirección dada. Retorna **BADADDR (-1)** si la dirección dada no tiene referencia con otras direcciones.
get_next_dref_from(ea_t from, ea_t current) Retorna la siguiente ubicación a la cual la dirección dada (**from**) referencia un valor dato (**current**) el cual ha sido retornado previamente por una llamada a **get_first_dref_from** o **get_next_dref_from**. Retorna **BADADDR** si no existen más referencias cruzadas.
get_first_dref_to(ea_t to) Retorna la primera ubicación la cual referencia como dato a la dirección dada. Retorna **BADADDR (-1)** si no existe ninguna referencia a la dirección dada.
get_next_dref_to(ea_t to, ea_t current) Retorna la siguiente ubicación referenciada a la dirección dada (**to**) como valor dato, dado por **current** el cual previamente ha sido retornado por una llamada a **get_first_dref_to** o **get_next_dref_to**. Retorna **BADADDR** si no existen más referencias cruzadas a la ubicación dada.

El SDK no contiene función equivalente a la función IDC **XrefType**. Pero una variable llamada **lastXR** está declarada en **xref.hpp**; sin embargo, no es exportada. Si necesitamos determinar exactamente el tipo de una referencia cruzada, deberemos iterar por las referencias cruzadas utilizando una estructura **xrefblk_t**. Esta estructura **xrefblk_t** se describe en la sección 15.2.5.3 “Enumerar referencias cruzadas”.

Performance Bigundill@