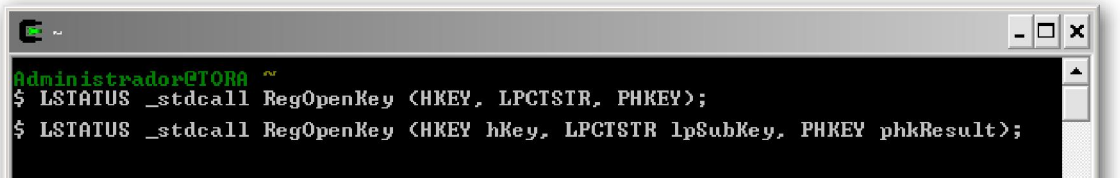
12.1.— Aumentar la información de las funciones para IDA

El conocimiento que tiene IDA respecto a las funciones le viene dado desde dos fuentes distintas: Una los archivos librería **(.til)** y otra los archivos de utilidades **IDS (.ids)**. Durante la fase de análisis inicial, IDA utiliza la información almacenada en estos archivos para mejorar la exactitud del desensamblado y hacer el desensamblado más legible. Esto se logra con la incorporación de los nombres y los tipos de los parámetros de la función y los comentarios asociados a las distintas funciones de librería.

En el **escrito 7** explicamos que los archivos de librería son un mecanismo por el cual IDA guarda el complejo esquema de las estructuras de datos. Dichos archivos son también el medio mediante el cual IDA registra la información respecto al acuerdo de llamada y la secuencia de parámetros. Ida utiliza la información de firma de la función de varias maneras. Primero cuando un binario utiliza librerías compartidas, IDA no tiene forma de conocer el acuerdo de llamada que se utilizará por las funciones en esas librerías. En tales casos, IDA intentará hacer coincidir las funciones de librería con las firmas asociadas a un archivo de tipo de librería. Si una de las firmas coincide, IDA puede interpretar el acuerdo de llamada utilizado por la función y realizar los ajustes necesarios al puntero de pila (recordemos, en **stdcall** las funciones ejecutan su propia limpieza de la pila). El segundo uso de las firmas de la función, es anotar los parámetros que serán pasados a la función con comentarios que denoten exactamente el parámetro que se pasará a la pila antes de llamar a la función. La cantidad de información presente en dicho comentario dependerá de cómo esté mostrada la información en la firma de la función, la cual no es más que la que IDA fue capaz de analizar. En el ejemplo siguiente las dos firmas son declaraciones C válidas, aunque la segunda proporciona más información de la función, ya que nos proporciona el nombre formal del parámetro y los tipos de dato.



Las librerías tipo de IDA contienen las informaciones de firma de una gran cantidad de funciones **API** comunes, incluidas una gran parte de las **API Windows**. Veamos un desensamblado por defecto de una llamada a la función **RegOpenKeyExA**:

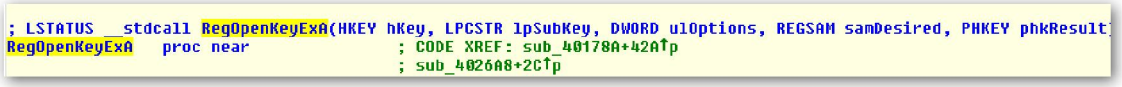
```
.text:00401BA1                lea     ecx, [esp+0C68Ch+hKey]
.text:00401BA5                push    ecx             ; phkResult
.text:00401BA6                push    1               ; samDesired
.text:00401BA8                push    0               ; ulOptions
.text:00401BAA                push    offset SubKey   ; "Software\\Microsoft\\Windows\\CurrentVersi"...
.text:00401BAF                push    80000002h       ; hKey
.text:00401BB4                call    RegOpenKeyExA
```

Observemos como IDA nos ha añadido comentarios en el lado derecho, figura abajo, indicando en cada instrucción qué parámetro será empujado a la pila hasta realizarse la llamada a **RegOpenKeyExA**. Cuando los nombres formales están disponibles en la firma de la función, IDA intenta nombrar automáticamente las variables que correspon-

```
.text:00401BA5                push    ecx             ; phkResult
.text:00401BA6                push    1               ; samDesired
.text:00401BA8                push    0               ; ulOptions
```

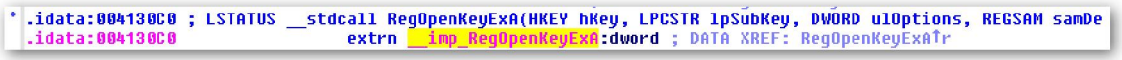
-dan a los parámetros específicos. En dos casos del ejemplo anterior, puede verse que

IDA ha nombrado a la variable local como **hKey** y a la variable global como **SubKey** basándose en sus correspondencias con los parámetros formales del prototipo de **RegOpenKeyExA**. Si el prototipo de función analizada hubiera contenido solamente la información de tipo pero no los nombres formales de los parámetros, entonces los comentarios existentes en el ejemplo sólo serían el nombre de los tipos de dato de los correspondientes argumentos y no los nombres de parámetros. En el caso del parámetro **lpsubkey**, el nombre del parámetro no se muestra como un comentario ya que el parámetro apunta a una variable cadena, y el contenido de la cadena puede ser mostrado utilizando comentario repetitivo. Para finalizar observemos que IDA ha reconocido a **RegOpenKeyExA** como una función **stdcall** y automáticamente ha ajustado el



stack pointer, ver figura abajo, para una vez realizado **RegOpenKeyExA** pueda retornarse el flujo de ejecución con la pila ajustada.

Toda esta información se extrae de las firmas de funciones, que IDA también muestra como comentarios dentro del desensamblado en la ubicación apropiada que es la **tabla de importación**, como podemos ver seguidamente:



El comentario que muestra la función prototipo se ha extraído de un archivo **.til** de IDA el cual contiene la información de las funciones **API Windows**.

¿Bajo qué circunstancias podremos desear crear nuestras propias firmas de función? Siempre que nos encontremos con un binario que esté enlazado, dinámicamente o estáticamente, a una librería de la cual IDA no tenga la información de las funciones prototipo. Por lo cual se tendrá que generar la información de tipo para todas las funciones contenidas en dicha librería y además hacer que IDA las anote automáticamente en el desensamblado. En el **escrito 7**, vimos cómo añadir información de las funciones prototipos a un archivo **.til**, haciendo que IDA analizara una o más funciones prototipo con la acción **File > Load File > Parse C Header File**. Desafortunadamente, como previamente ya explicamos, esta es actualmente la única forma de añadir contenido a un archivo **.til**, y dicho contenido sólo permanece asociado a la base de datos en la cual será analizado. Desde el momento en que los archivos **.til** son archivados dentro de archivos **.idb** cuando la base de datos es cerrada, la única forma de extraer la información de firmas de las funciones analizadas es copiar el archivo **.til** de la base de datos, mientras esta se encuentre abierta en IDA. Los siguientes pasos indican el proceso para crear una librería tipo de funciones prototipo:

1. Cargamos el ejecutable en una nueva base de datos. El ejecutable que escojamos no tiene importancia, lo único que nos interesa es acceder a la capacidad que tiene IDA para analizar archivos **C**. Como ejemplo elegiremos el **CRACKME.EXE**.

2. Analicemos los encabezados del archivo **C** que contiene los prototipos de las funciones que queremos incorporar. Esto puede requerir alguna transformación de los encabezados del archivo, como eliminar tipos de dato no reglamentarios como **uchar** o **dword**, para que IDA pueda analizarlos correctamente. La información del proceso de análisis será incorporado en un archivo **.til**. En nuestro caso, es el archivo llamado **CRACKME.til**.

3. Antes de cerrar la base de datos, copiamos el archivo **.til** en la carpeta **til** de IDA. Si queremos podemos renombrar el nuevo archivo con el nombre de la biblioteca que representa, por ejemplo en nuestro caso podría ser **mssdk.til**. En este momento el archivo **.til** estará disponible para poderlo utilizar en cada base de datos utilizando el atajo de teclado **INSERT** en la ventana **Loaded Type Libraries** o con la acción **View > Open Subviews > Type Libraries.**

4. La base de datos utilizada para analizar los encabezados de archivo ahora podemos cerrarla y opcionalmente guardarla.

Todos estos pasos son perfectos cuando tenemos el acceso al código fuente lo cual permite a IDA analizarlo con todos sus nombres. Pero por desgracia no siempre es así, si no tenemos acceso al código fuente y queremos un desensamblado de alta calidad ¿Cómo podremos indicarle a IDA para que lo haga sin el código fuente? Este es precisamente el propósito de las **utilidades IDS** o **idsutils**. Las utilidades IDS son un conjunto de tres programas de utilidades utilizadas para crear archivos **.ids**. Bien, lo primero que haremos será estudiar qué es un archivo **.ids** y luego veremos como crear nuestros propios archivos **.ids**.

12.1.1.—Archivos IDS

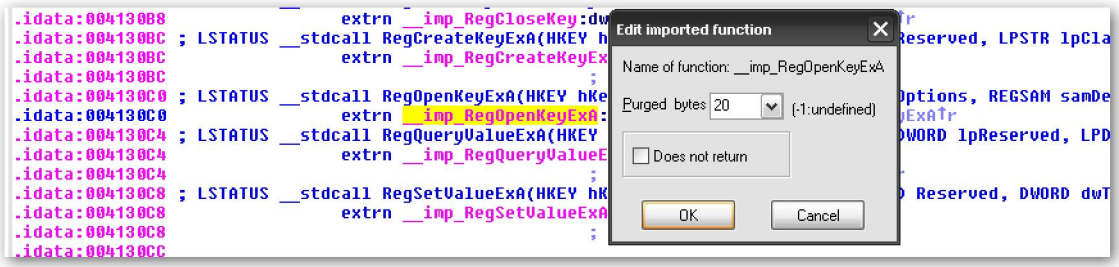
IDA utiliza el archivo **.ids** para complementar el reconocimiento de las funciones de librería. Un archivo .ids describe el contenido de una librería compartida listando cualquier función exportada contenida dentro de la librería. La información detallada de cada función incluye el nombre de la función y su número ordinal asociado, ("Un número ordinal es un índice entero asociado a cada función exportada. La utilización de ordinales permite localizar a una función en la tabla utilizando un entero en vez de realizar una comparación de cadenas, más lenta, con el nombre de la función"), si la función utiliza **stdcall**, y si es así, cuántos bytes limpiará de la pila cuando la función retorne, y si los comentarios opcionales se mostrarán cuando la función sea referenciada dentro del desensamblado. En la práctica, los archivos **.ids** realmente se comprimen en archivos **.idt**, los cuales contienen descripciones textuales de cada función de librería.

Cuando se carga por primera vez un ejecutable en una base de datos, IDA determina de que archivos de librería depende. Para cada librería compartida, IDA busca su correspondiente archivo **.ids** en la carpeta **ids** para así obtener las descripciones de cualquier función que el ejecutable referencia. Es importante comprender lo siguiente, los archivos **.ids** no necesariamente contendrán la información de firma de la función.

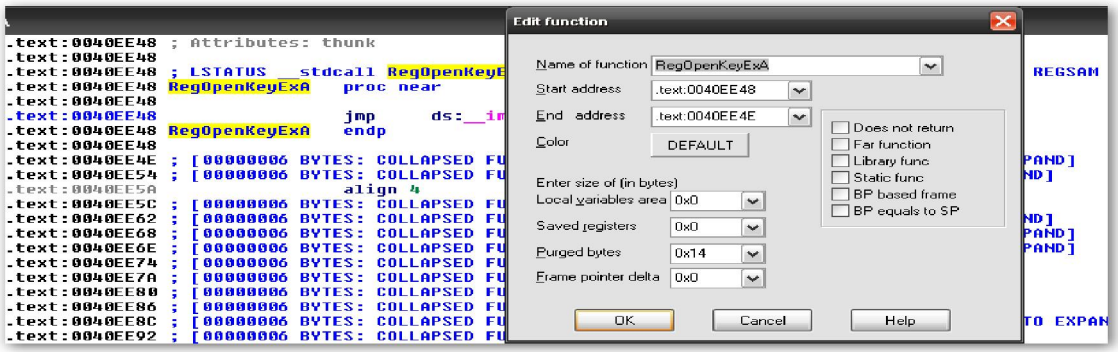
Por lo tanto IDA no puede proporcionar el análisis de los parámetros de la función basándose exclusivamente en los archivos **.ids**. Sin embargo si puede hacer el cálculo preciso del **stack pointer** cuando un **.ids** contenga la información correcta concerniente a los acuerdos de llamada empleados por las funciones y el número de bytes que la función limpiará de la pila. En las situaciones donde una **DLL** exporta nombres modificados, Ida puede ser capaz de inferir una firma de los parámetros de la función a partir del nombre modificado, en cuyo caso esta información estará disponible cuando el archivo **.ids** sea cargado. La sintaxis de los archivos .ids la describiremos cuando estudiemos como crear dichos archivos.

Eliminación manual de bytes

Las funciones de librería que utilizan el acuerdo de llamada **stdcall** pueden causar problemas con el análisis que hace IDA del **stack pointer**. Si carecemos de la información de archivos como tipo de librería o .ids, IDA no tiene forma de conocer ninguna de las funciones importadas que utiilicen **stdcall**. Esto significa que IDA no será capaz de seguir la pista correctamente del comportamiento del **stack pointer** a través de las llamadas a funciones ya que no tiene información sobre el acuerdo de llamada. Más allá de conocer si una función utiliza **stdcall**, IDA debe también conocer exactamente cómo será limpiado cualquier byte de la pila cuando la función se haya completado y retorne. Careciendo de la información de acuerdo de llamada, IDA intentará automáticamente determinar si una función utiliza **stdcall** utilizando una método de análisis matemático conocido con el nombre de **simplex method**. El segundo método depende de nuestra intervención manual sobre IDA. La siguiente figura muestra el diálogo de edición utilizado para las funciones importadas y en el cual podemos de forma especial eliminar los bytes que creamos pertinentes.



Podrás acceder a dicho diálogo desplazándote hasta la entrada de la función en la tabla de importación y una vez allí editar la función con la acción **Edit > Functions > Edit Function** o con el atajo de teclado **ALT-P**. Observemos que la funcionalidad limitada de este particular diálogo, es opuesto al de edición de una función mostrado a continuación.



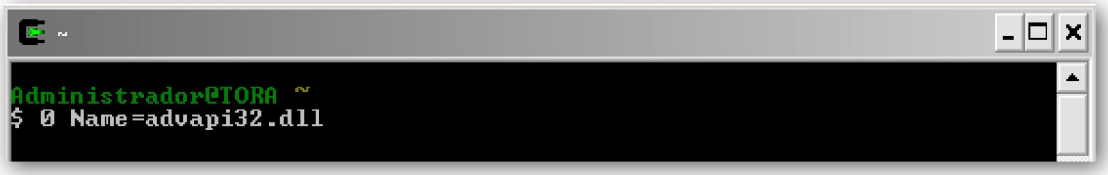
Debido a que ésta es una entrada de una función importada, IDA no tiene acceso al cuerpo compilado de la función y por lo tanto a ninguna información asociada con respecto al **stack frame** de la función y sin ninguna evidencia de que la función utilice el acuerdo de llamada **stdcall**. Al carecer de dicha información, IDA habilita el campo **Purged bytes** con el valor **-1**, indicando que no sabe cuantos bytes se limpiarán de la pila cuando la función retorne. Lo que hace IDA en tales casos, es entrar el valor correcto del número de bytes limpiados e incorporará la información suministrada en el análisis del **stack pointer** dondequiera que sea llamada la función asociada a este. En los casos en que IDA es consciente del comportamiento de la función, como hemos visto en el diálogo de la función importada, el campo **Purged bytes** ya está rellenado con la cantidad de bytes. Sepamos que este campo nunca se rellenará como resultado de un análisis **simplex method**.

12.1.2.—Crear archivos IDS

Las utilidades de IDA **idsutils** se utilizan para crear archivos **.ids**. Dichas utilidades incluyen dos librerías de análisis, **dll2idt.exe** para extraer información de DLL de Windows y **ar2idt.exe** para extraer información de librerías AR. En ambos casos nos dará como salida un archivo texto **.idt** el cual contendrá una línea por cada función exportada la cual nos combina el número ordinal de la función con el nombre de dicha función. La sintaxis de los archivos **.idt** es muy fácil, dicha sintaxis se describe en el **readme.txt** que está incluido en **idsutils**. La mayoría de líneas de un archivo **.idt** se utilizan para describir una función exportada de acuerdo con el siguiente esquema:

** Una entrada de exportación empieza con un número positivo. Este número representa el número ordinal de la función exportada.

** Al número ordinal le sigue un espacio y seguidamente una directiva **Name** en la forma **Name=función**, por ejemplo, **Name=RegOpenKeyExA**. Si se utiliza el valor especial **0** como número ordinal, entonces la directiva **Name** se utiliza para especificar el nombre de la librería descrita en el fichero **.idt** actual de esta forma:

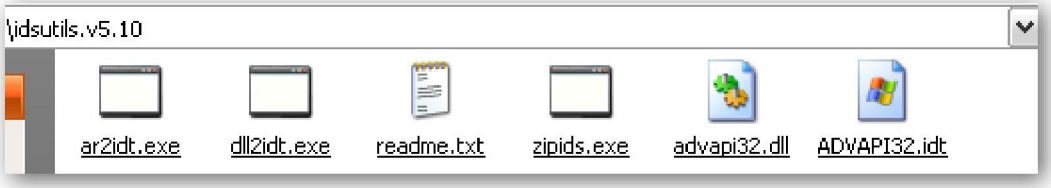


Una directiva opcional que se puede utilizar para especificar que una función utiliza el acuerdo de llamada **stdcall es, **Pascal** y además indicar cuántos bytes de la pila limpia la función en su retorno. Veamos un ejemplo:

** Otra opción de directiva es **Comment** la cual puede se puede añadir a una entrada de exportación para especificar un comentario que será mostrado junto con la función en

cada referencia a la función dentro del desensamblado. Una entrada de exportación completa tendría una apariencia como la siguiente:

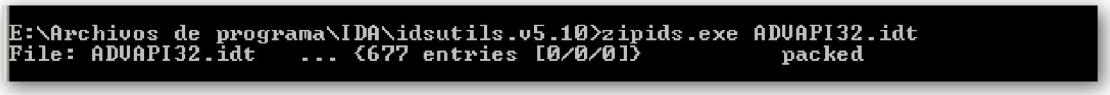
Otras directivas opcionales están explicadas en el **readme.txt** de **idsutils**. El propósito de las utilidades de análisis **idsutils** es automatizar, en lo posible, la creación de archivos **.idt**. El primer paso para la creación de un archivo .idt es obtener una copia de la librería la cual queremos analizar; cuando la tenemos la analizamos utilizando la utilidad de análisis apropiada. Si lo que queremos es crear un archivo **.idt** de la librería **advapi32.dll** relacionada con **Windows**, deberemos utilizar la siguiente orden:



Una vez realizado el análisis, en nuestro caso nos proporcionará un archivo llamado **ADVAPI32.idt**. La diferencia en que el nombre nos salga con mayúsculas es debido a que **dll2idt.exe** deriva el nombre del archivo de salida basándose en la información contenida dentro de la propia **DLL**. Las primeras líneas del archivo **.idt** resultante son las siguientes:

Observemos que para los analizadores, no es posible determinar si una función utiliza **stdcall**, y si la utiliza tampoco cuántos bytes se limpiarán de la pila. Para añadir cualquier directiva como **Pascal** o **Comment** debe realizarse manualmente utilizando el editor de texto antes de crear el archivo final **.ids**. El último paso para crear nuestro

archivo **.ids** es utilizar la utilidad **zipids.exe** la cual nos lo comprime en un archivo **.idt** y para finalizar copiar nuestro archivo **.ids** resultante a la carpeta **ids**.



Una vez realizados todos estos pasos, IDA cargará **ADVAPI32.ids** cada vez que un binario enlace con **advapi32.dll**. Si elegimos no copiar el nuevo archivo **.ids** en la carpeta **ids,** podremos cargarlo cada vez que queramos realizando la acción **File > Load File > IDS File.**

Otra acción que nos permite realizar los archivos .ids es poder vincular un archivo **.ids** con un archivo específico tipo **.sig o .til**. Cuando elegimos un archivo .ids. IDA utiliza un archivo de configuración IDS llamado **idsnames**, lo encontraremos en **Archivos de programa\IDA\idsnames**. Este archivo de texto contiene líneas que nos muestran lo siguiente:

** La combinación entre el nombre de la librería compartida y el nombre del archivo .ids. Esto permite a IDA localizar el archivo .ids correcto cuando el nombre de la librería compartida no se traduce correctamente como MS-DOS 8.3.

** La combinación entre un archivo .ids con un archivo .til. En tales casos, IDA automáticamente carga el .til especificado, siempre que se cargue el .ids especificado.

** La combinación entre un archivo .sig y un archivo .ids. En este caso, IDA carga el .ids indicado con cualquier archivo .sig aplicado al desensamblado.

En un apartado más adelante, veremos una alternativa para utilizar las librerías de análisis proporcionadas por idsutils dentro de un script, con lo cual conseguiremos perfeccionar el análisis de las funciones con IDA y poder generar archivos .idt mucho más completos y descriptivos.

Performance Bigundill@