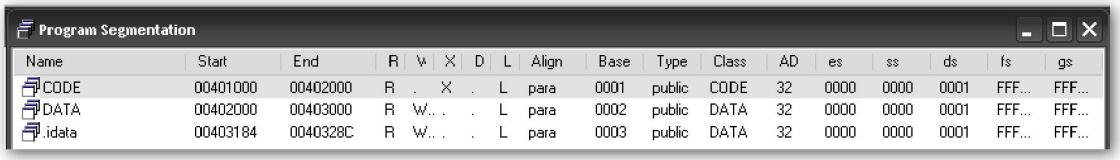
4.3.—Ventanas terciarias de IDA

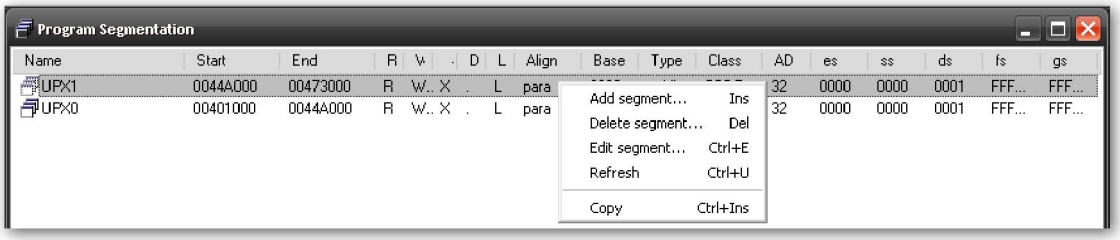
Las últimas vistas que vamos a estudiar son las que no están abiertas por defecto en IDA. Cada una de dichas ventanas se pueden habilitar realizando la acción **View > Open Subviews**, esto es así porque tienden a proporcionar información que no se necesita de inmediato.

4.3.1.—Ventana Segments

La ventana **Segments** muestra un listado resumido de los segmentos presentes en el archivo binario. Ten en cuenta que a lo que IDA llama **segmentos**, lo llamamos normalmente **secciones** cuando nos referimos a un binario. No confundamos el término segmentos cuando nos referimos a segmentos de memoria relacionados con la CPU la cual ejecuta una memoria de arquitectura segmentada. La información mostrada en la ventana se compone del nombre del segmento, direcciones de principio y final y banderas permitidas. Las direcciones de inicio y final representan el rango de dirección virtual en la cual cada sección del programa serán mapeadas al ejecutarse. La figura es la vista **Segments** del archivo CRACKME.EXE:



Si nos encontráramos en el caso de un archivo empaquetado, nos mostraría secciones que no son normales, por ejemplo un binario empaquetado con UPX nos mostraría la siguiente imagen:



Vemos que los dos segmentos de código son **grabables (W)** y **ejecutables (X)**, esto nos indica la posibilidad de código auto modificable, hablaremos de ello más adelante. De hecho que IDA sepa el tamaño de una sección no quiere decir que sepa su contenido. Esto es así ya que los segmentos ocupan menos espacio en disco que en memoria. En estos casos IDA muestra los valores de las partes de la sección que ha determinado llenar el archivo desde el disco. El resto de espacio de la sección será rellenado con signos de interrogación.

Si hacemos doble click en cualquier entrada de la ventana, nos mostrará el principio de la sección seleccionada en la ventana de desensamblado. Hacer click derecho en una entrada nos proporciona un menú contextual con el cual puedes añadir un segmento nuevo, eliminar un segmento o editar las propiedades de los segmentos existentes, figura arriba. Estas características son particularmente útiles para realizar ingeniería inversa de un archivo con secciones no estandars las cuales no han sido detectadas por el cargador de IDA.

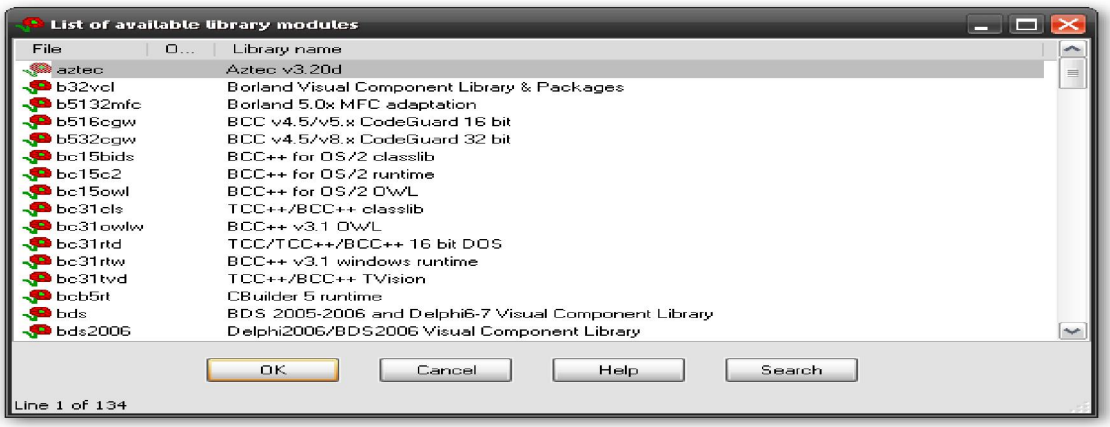
También en la línea de comandos IDA puedes ejecutar **objdump (-h), readelf (-S) y dumpbin (/HEADERS).**

4.3.2.—Ventana Signatures

IDA utiliza una extensa librería de firmas para identificar bloques conocidos de código. Las firmas se utilizan para identificar el compilador utilizado en las secuencias de inicio y un intento de determinar el compilador que se ha utilizado para construir el binario tratado. Las firmas también se utilizan para categorizar las funciones como funciones de librería conocidas insertadas por el compilador o como funciones añadidas al binario como resultado de un enlace estático. Cuando IDA identifica las funciones de librería para nosotros, podemos dedicar más esfuerzos en el código que este no ha reconocido, probablemente una de la más importante para realizar ingeniería inversa es la **printf.**

La ventana **Signatures** es utilizada para listar las firmas que IDA ha verificado al abrir el archivo binario. En, figura abajo, nos indica que se ha aplicado la firma **vcseh** del directorio **\Archivos de Programa\IDA\sigs** y que no ha encontrado ninguna función de librería.

En al menos dos casos, querrás saber como aplicar firmas adicionales a tus binarios. El primer caso será cuando IDA no logra reconocer el compilador que se ha utilizado para construir el binario, con lo cual se verá incapacitado para escoger las firmas apropiadas a aplicar. Dado este caso podemos forzar a IDA para que aplique una o más firmas basándonos en el análisis preliminar. La segunda situación supone crear tus propias firmas para librerías que IDA no tenga sus firmas. Para realizar lo comentado **DataRescue** nos proporciona unas herramientas para generar firmas a nuestra necesidad y que IDA pueda verificar con su motor de firmas. Cómo generar las firmas lo trataremos también más adelante. No obstante si ya quieres aplicar distintas firmas, pulsa la tecla **INSERT** o click derecho en la ventana **Signatures** y se te ofrecerá la opción **Apply new signature**, con lo cual podrás elegir la que gustes de la lista de todas las firmas conocidas por IDA en su instalación.



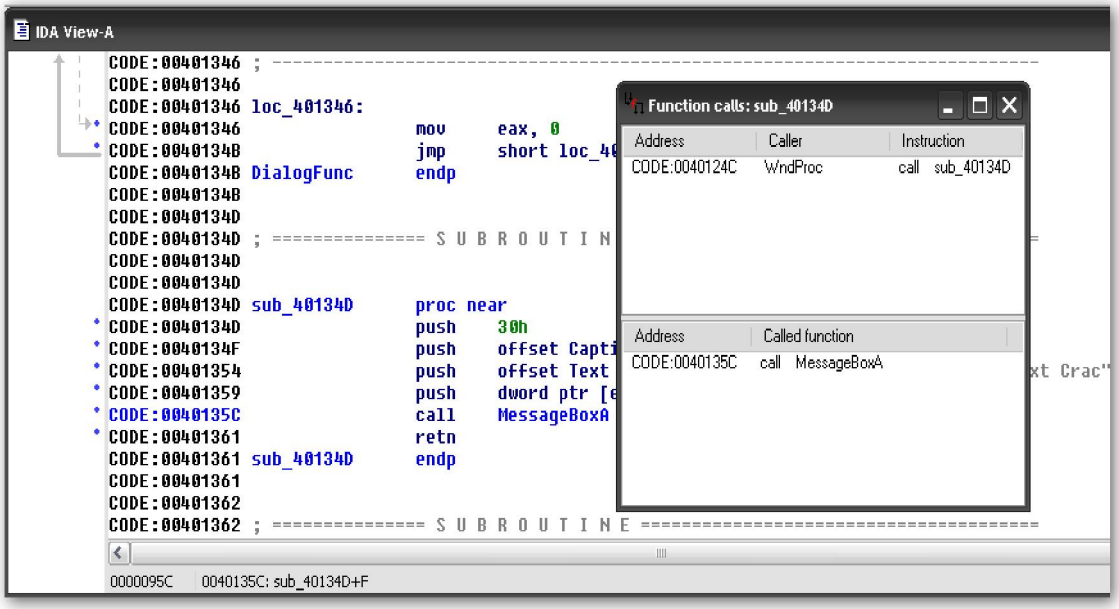
4.3.3.—Ventana Type Libraries

La ventana **Type Libraries** representa los tipos de datos predefinidos y los prototipos de funciones tomados de los encabezados de los archivos compilados con los compiladores más conocidos. Al procesar los **headers** de los archivos, IDA comprende los tipos de datos que son esperados por las funciones de las librerías más comunes y las puede anotar en los desensamblados. Similarmente también comprende el tamaño y esquema de las estructuras de datos más complejas. Toda esta información se almacena en archivos **TIL** en el directorio **IDA/til** y se aplica cada vez que un binario es analizado. Al igual que con las firmas, IDA debe ser capaz de deducir primero las librerías que un programa utiliza antes de que pueda escoger un conjunto apropiado de archivos **TIL** y cargarlos. Puedes decirle a IDA que cargue librerías adicionales pulsando la tecla **INSERT** o haciendo click derecho dentro de la ventana **Type libraries** y escoger la librería a cargar. Las **Type libraries** las estudiaremos con más detalle.

4.3.4.—Ventana Function Calls

En cualquier programa, una función puede llamar o ser llamada por otras funciones. De hecho es simple construir una gráfica que nos muestre la relación entre los llamadores y los llamados. Dicha gráfica tiene el nombre de **function call graph** o **function call tree**, cómo IDA crea estas gráficas lo trataremos más adelante. En ocasiones no nos interesará ver todo el gráfico de llamadas de un programa; en vez de esto nos interesará sólo conocer las llamadas más cercanas de una función dada. Para nuestros propósitos llamaremos **Y** a una cercana de **X** si **Y** llama directamente a **X** o **X** si llama directamente a **Y**.

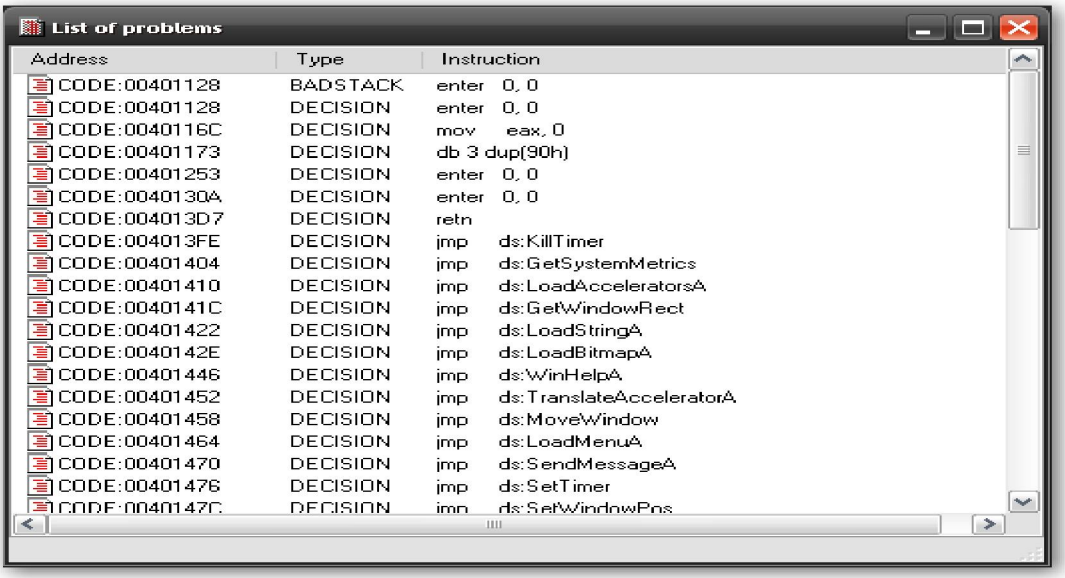
La ventana **Function Calls** nos proporciona la respuesta a la pregunta de la llamada cercana. Cuando abrimos la ventana Function Calls, IDA determina las llamadas cercanas de la función en la que el cursor está posicionado y genera una vista como la figura:



En la figura vemos que la función **0040134D** es llamada solamente una vez en **WndProc** y la función hace sólo una llamada a **MessageBoxA**. Si hacemos doble click en cualquier línea de la ventana inmediatamente saltaremos a la vista de desensamblado en la posición de la llamada seleccionada ya sea llamada o llamador. De momento también decir que el mecanismo que mantiene y genera la ventana **Functions Calls** son las referencias cruzadas (**xrefs**) las cuales las trataremos con más detalle.

4.3.5.—Ventana Problems

La ventana **Problems** es la forma en que IDA nos informa de cualquier dificultad con la que se haya encontrado el desensamblar un binario y lo que ha realizado para superar dichas dificultades. En algunos casos, podemos ser capaces de manipular el desensamblado para ayudar a IDA a superar un problema, pero en otros no. Puedes encontrarte con problemas aún siendo un binario muy sencillo. En muchos casos optar en ignorar el problema no es una estrategia mala. Fíjate en algunos de los problemas superados en el CRACKME.EXE:



Cada problema tiene las siguientes características, primero nos proporciona la dirección en donde ocurrió el problema, segundo el tipo de problema encontrado y tercero la instrucción en la ubicación del problema. En el **IDA help > Problems List** tenemos un listado completo de los tipos de problemas y su posible forma de solucionarlos.

Performance Bigundill@