14.0.—Realizar scripts con IDC

Como ya sabemos ninguna aplicación está realizada al gusto de cada uno, ni a las necesidades específicas que deseamos. Esto para la "flor y nata" de programadores no es problema ya que adecuan las aplicaciones a su gusto. No obstante IDA nos proporciona esta opción integrando un motor de scripts el cual permite al usuario realizar un gran control programático sobre las acciones de IDA.

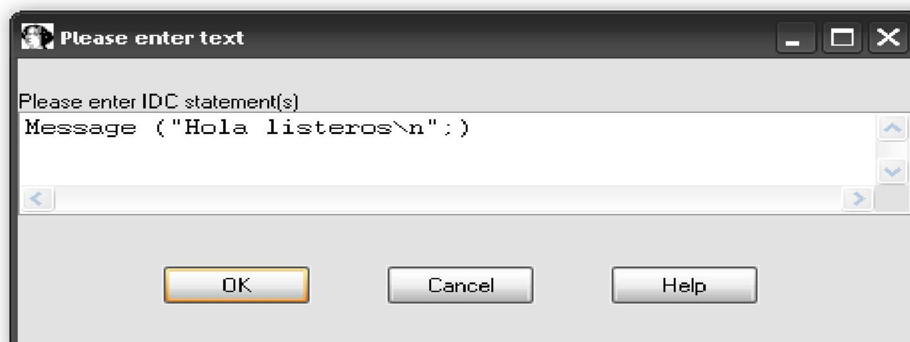
Las utilizaciones potenciales de los scripts son infinitos pueden actuar desde una sola línea del programa, hasta automatizar tareas para ejecutar complejas funciones de análisis. Desde un punto de vista de automatización, los scripts de IDA se pueden considerar como macros, mientras que desde el punto de vista de análisis los scripts sirven como lenguaje de consulta proporcionándonos acceso programático a los contenidos de la base de datos de IDA. El lenguaje de script en IDA recibe el nombre de **IDC**, quizás porque su sintaxis tiene un gran parecido a la de **C**. En el resto de este escrito estudiaremos la programación muy básica y la ejecución de scripts IDC, así como algunas funciones útiles disponibles para los programadores de IDC.

14.1.—Ejecución básica de un script

Antes de introducirnos en el lenguaje IDC, es necesario comprender la forma más sencilla de ejecutar los scripts IDC. Para acceder al motor de scripts, IDA nos proporciona dos opciones **File > IDC file** y **File > IDC command**. Si seleccionamos **File > IDC file** indicamos que queremos ejecutar un programa IDC, y se nos muestra un diálogo donde podemos seleccionar el script a ejecutar. Cada vez que ejecutemos un nuevo programa IDC, el programa se añade a la ventana proporcionándonos acceso para editarlo o reejecutarlo. La siguiente figura muestra el diálogo **Recent IDC scripts**.

El botón de la izquierda, que muestra una hoja de papel en blanco, abre el script para editarlo utilizando el editor especificado en **Options > General** y la solapa **Misc**. El botón derecho, mostrando un engranaje como icono, se utiliza para ejecutar el script.

Como alternativa para ejecutar un script, podemos elegir abrir una consola de órdenes IDC realizando la acción **File > IDC command**, la figura siguiente nos muestra el resultado, la cual es útil en situaciones en donde sólo deseamos ejecutar unas cuantas declaraciones IDC pero no queremos crear un archivo script.



Existen ciertas restricciones en cuanto a los tipos de declaraciones que pueden introducirse en dicho diálogo, pero éste es muy útil en los casos en donde crear un archivo script completo no es necesario.

Como última opción para ejecutar fácilmente órdenes IDC es utilizar la opción **Command line**. Esta opción sólo está habilitada en la versión GUI, y se habilita poniendo la opción **DISPLAY_COMMAND_LINE** en **YES** en el archivo **idagui.cfg**. Una vez se ha habilitado, deberemos hacer click derecho en el área de herramientas de IDA y activar la herramienta **Command line**. La siguiente figura muestra la línea de órdenes que aparece en la esquina inferior izquierda justo debajo de la ventana de mensajes.

Aunque la línea de órdenes contiene una sola declaración, podemos introducir declaraciones IDC múltiples separadas cada una por un punto y coma. Un problema de la línea de órdenes es que no podemos pegar si hemos copiado algo de otro lugar. Sin embargo la lista de órdenes recientes es accesible vía las teclas de flechas. Si frecuentemente necesitamos ejecutar scripts IDC muy cortos tendríamos que considerar habilitar la línea de órdenes IDC.

14.2.—El lenguaje IDC

A diferencia de otros aspectos de IDA, en la ayuda de IDA existe una gran cantidad de ayuda disponible respecto al lenguaje IDC. El apartado **IDC language**, trata la sintaxis básica IDC y tiene un índice de las funciones IDC, el cual proporciona un exhaustivo listado de funciones para utilizarlas por los programadores IDC.

El lenguaje IDC utiliza la mayor parte de los elementos sintácticos de C. Debido a dicha similitud, describiremos a IDC desde un punto de vista de lenguaje C y nos centraremos en las diferencias entre los dos lenguajes. IDC reconoce el estilo C de comentarios multilinea utilizando /* */ y la de línea de comentarios utilizando //.

14.2.1.—Variables IDC

En el lenguaje IDC las variables no tienen ningún tipo explícito. En IDC se utilizan tres tipos de dato: valores enteros (tipo long), cadenas (strings) y punto flotante, siendo los más utilizados son los enteros y cadenas. Las cadenas son tratadas en IDC como dato tipo nativo, y no hay necesidad de tener presente el espacio requerido para almacenar una cadena o si una cadena termina en nulo o no.

Todas las variables deben de ser declaradas antes de utilizarse. IDC sólo soporta variables locales, las globales no están soportadas. Todas las declaraciones de variables deben de hacerse antes de la primera declaración dentro de una función. No es posible inicializar una variable al mismo tiempo que es declarada. La palabra clave **auto** es utilizada para introducir la declaración de una variable. Los siguientes ejemplos muestran una declaración de variable correcta e incorrecta.

```
auto addr, reg, val;    //correcto
auto count = 0;         //incorrecto, la inicialización no es permitada en declaraciones
```

Observemos que varias variables pueden ser declaradas en una sola declaración y

que todas las declaraciones en IDC deben terminar utilizando un punto y como como en C. IDC no soporta como en C los arrays, punteros o tipos de datos complejos como **structs** o **unions**.

14.2.2.—Expresiones IDC

Con algunas excepciones, IDC soporta todos los operadores de C tanto aritméticos como lógicos, incluidos los operadores ternarios **(? :)**. La composición de operadores de asignación de forma **op = (+=, *=, >>= y semejantes)** no son soportados, ni tampoco la operación **coma**. Todos los operandos enteros son tratados como valor con signo. Esto afecta a la comparación de enteros (los cuales son siempre con signo) y al operador >>, el cual siempre ejecuta un desplazamiento aritmético con una respuesta dependiendo del signo del bit. Si necesitamos un desplazamiento lógico, deberemos poner en práctica el enmascaramiento del bit superior del resultado, como podemos ver en el ejemplo:

```
resultado = (x >> 1) & 0x7fffffff; //coloca el bit más significativo a cero
```

Debido a que las cadenas son un tipo nativo en IDC, ciertas operaciones con cadenas toman un significado distinto que en C. La asignación de una cadena operando a una cadena variable da como resultado una operación de copia de cadena; así no necesitamos copiar cadenas o duplicar funciones como en C con **strcpy** y **strdup**. La suma de dos cadenas operandos da como resultado la concatenación de los dos operandos; así **"Hola"** + **"listeros"** nos da **"Holalisteros"**; por lo tanto no necesitamos la función de C **strcat**.

14.2.3.—Declaraciones IDC

Al igual que en C, las declaraciones simples se terminan con un punto y coma. La declaración **switch** de C no es soportada por IDC. Cuando utilicemos **for** para bucles, tengamos en cuenta que IDC no soporta la asignación de operadores compuestos, lo cual afecta si queremos hacer un contador de más uno, veámoslo:

```
auto i;
for (i = 0; i < 10; i += 2) {}          //incorrecto, += no es soportado
for (i = 0; i < 10; i = i + 2) {}       //correcto
```

Para las declaraciones compuestas, IDC utiliza los mismos cierres **{}** que en C al igual que la misma semántica y sintaxis. Dentro de un bloque encerrado, está permitido declarar nuevas variables tipo **long**, pero al igual que la declaración de variables, éstas tienen que ser la primera declaración dentro del bloque. Sin embargo, IDC no impone el alcance de dichas variables nuevas, con lo cual pueden ser referenciadas más allá del bloque en donde se han declarado. Consideremos el siguiente ejemplo:

```
if (1) {                    //siempre verdadero
    auto x;
    x = 10;
}
else {                      //nunca se ejecuta
    auto Y;
    y = 3;
}
Message ("x = %d\n", x);// x continua accesible después de haberse terminado el bloque
Message ("y = %d\n", y);// IDC permite esto aunque no se ejecute
```

Las declaraciones de salida, la función **Message** es la análoga a **printf** en C, nos informará que **x = 10** e **y = 0**. Suponiendo que IDC no impone estrictamente el alcance de **x**, no nos tiene que sorprender que se nos permita imprimir el valor de **x**.

Lo que si es sorprendente es que **y** sea accesible a todo, siempre y cuando el bloque en donde se ha declarado **y** se ejecute. Simplemente es una cosa extraña de IDC. Observemos que si imponemos el alcance de una variable dentro de una función, la variable declarada dentro de esa función permanecerá inaccesible a cualquier otra función.

14.2.4.—Funciones IDC

IDC sólo soporta las funciones definidas por el usuario en programas que se encuentren como archivos **.idc**. Las funciones definidas por el usuario no son soportadas a través de la opción **IDC command**. La sintaxis de IDC para declarar funciones definidas por el usuario, difiere la mayor parte del lenguaje C. La palabra clave **static** se utiliza para introducir una función definida, y la lista de parámetros de la función consiste solamente en la lista de los nombres de los parámetros separados por una coma. El siguiente listado detalla una estructura básica de una función definida por el usuario:

```
static mi_funcion (x, y, z) {
        // primero declarar cualquier variable local
        auto a, b, c;
        // añadir declaraciones para definir el comportamiento de la función

        // ...
}
```

Todos los parámetros de función son llamados estrictamente por valor. IDC no proporciona la llamada de parámetros por referencia, ni soporta punteros de ningún tipo. Tengamos en cuenta que una declaración de función nunca indica si la función explícitamente retorna un valor o el tipo de valor retornado cuando dicha función produce un resultado.

Cuando queramos retornar un valor de una función, utilizaremos la declaración **return** para retornar el valor deseado. Está permitido retornar distintos tipos de dato desde distintos flujos de ejecución dentro de la función. En otras palabras, una función en algunos casos podrá retornar una **string**, en otros casos la misma función podrá retornar un **entero**. Al igual que en C, la utilización de **return** dentro de la función es opcional. Sin embargo, en contraposición a C, cualquier función que no retorna un valor explícitamente, implícitamente retorna el valor **cero**.

14.2.5.—Programas IDC

Para todos los script de aplicaciones que requieran de unas cuantas declaraciones IDC, es probable que queramos crear un archivo programa IDC. Entre otras cosas, guardar nuestros scripts como programas nos da en cierta medida la duración y la portabilidad.

Los archivos programa IDC se requieren para poder utilizar las funciones definidas por nosotros. Como mínimo, debemos definir una función llamada **main** que no toma ningún argumento. El único otro requisito es que nuestro **main** contenga una directiva de preprocesamiento que incluya el archivo **idc.idc**. El siguiente listado detalla los componentes mínimos de un archivo programa IDC:

```
#include <idc.idc>      //directiva include obligada
//declaramos las funciones que se requieran
static main () {
        //colocar algo aqui que realice lo que deseamos
}
```

Un archivo programa IDC, reconoce las siguientes directivas C de preprocesado:

#include <archivo>
Incluye el nombre “archivo” en el archivo actual.

#define <nombre> [valor opcional]
Crea una macro llamada “nombre”, opcionalmente se le asigna el valor especificado

#ifdef <nombre>
Verifica la existencia de la macro “nombre” y opcionalmente seguidamente procesa cualquier declaración si la macro “nombre” existe.

#else
Opcionalmente utilizada en conjunción con un #ifdef, proporciona una alternativa para procesar declaraciones en el supuesto de que la macro “nombre” no exista.

#endif
Requerida para la finalización de un bloque #ifdef o #ifdef/#else.

#undef <nombre>
Elimina la macro “nombre”

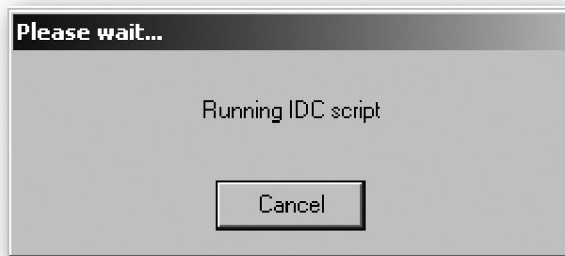
Cualquier intención de utilizar las directivas dentro del **IDC command** nos dará como resultado un error de sintaxis.

14.2.6.—Como se tratan los errores en IDC
No se pueden alabar las capacidades de IDC para presentar informes de error. Pero, existen dos tipos de error con los cuales nos podemos encontrar al ejecutar scripts IDC: **errores de análisis** y **errores de ejecución**.

Los **errores de análisis** son aquellos que impiden que el programa se ejecute, estos son tales como errores sintácticos, referencias a variables indefinidas y suministrar un número incorrecto de argumentos a una función. Durante la fase de análisis, IDC reporta lo que encuentra en el primer análisis. En algunos casos, los mensajes de error identifican correctamente la ubicación y el tipo de error (**Hola_listeros.idc,20: Missing semicolon**), pero en otros casos el mensaje de error no nos indica casi nada (**Syntax error near: <END>**). Si utilizamos el **IDC command** los mensajes de error proporcionados son realmente muy pobres. Solamente es indicado el primer error encontrado durante el análisis. Esto quiere decir que en un script con 15 errores de sintaxis tendremos que ejecutarlo 15 veces para tener información de cada error.

Los **errores de ejecución** son generalmente menores que los de análisis. Dichos errores causan que el script se cierre inmediatamente. Un ejemplo de error de ejecución es el provocado al intentar llamar a una función no definida, lo cual no es

detectado cuando se realiza el análisis inicial del script. Otro problema está cuando un script tarda mucho tiempo en ejecutarse. Una vez que el script se ha iniciado, no existe ninguna forma fácil para finalizarlo si ha realizado un bucle infinito o simplemente si está más tiempo del previsto para ejecutarse. Una vez que el script se ejecuta en unos dos o tres segundos IDA nos muestra el diálogo:



Este diálogo es el único medio, mediante el cual podemos finalizar un script que no funciona correctamente.

La depuración es otro de los puntos débiles de IDC. Aparte de poderlo realizar con las declaraciones de salida, no hay ora forma de depurar un script IDC.

14.2.6.—Almacenamiento permanente de datos en IDC

Como dijimos anteriormente, IDC no soporta conjuntos (arrays) en el sentido tradicional de declarar un bloque de almacenamiento y utilizando un subíndice acceder individualmente a elementos dentro de dicho bloque. Sin embargo en la documentación de IDA sobre scripting se menciona algo llamado **global persistent arrays**. Los global arrays de IDC es mejor llamarlos nombrado de objetos permanentes. Los objetos sólo se encuentran como conjuntos esparcidos. Los **global array** son guardados dentro de la base de datos de IDA y persisten en cada invocación del script en las sesiones de IDA. Los datos son guardados en los global arrays especificando un índice y un valor del dato, y serán guardados en el conjunto especificado por otro índice. Cada elemento en un conjunto puede tener simultáneamente un valor entero o un valor string. Los global arrays de IDC no proporcionan ningún medio para poder almacenar valores de punto flotante.

Observación: Para los curiosos, el mecanismo interno de IDA para almacenar conjuntos permanentes tiene el nombre de **netnode**. Las funciones de manipulación de conjuntos ya descritas nos proporcionan una idea abstracta sobre los **netnodes**, pero la explicación del acceso a los datos del **netnode** se encuentra en el **IDA SDK**, del cual hablaremos junto con los **netnodes** más adelante.

Toda interacción con los global arrays se realiza a través del uso de funciones IDC dedicadas a la manipulación de conjuntos. Seguidamente describimos dichas funciones:

long CreateArray(string name)

Esta función crea un objeto permanente con el nombre especificado. El valor de retorno es un manejador (handle) entero requerido para futuros accesos al conjunto. Si el objeto nombrado ya existe, el valor de retorno es -1.

long GetArrayId(string name)
Una vez que un conjunto ha sido creado, el acceso al conjunto se debe realizar a través del manejador (handle) entero, el cual se puede obtener al buscar el nombre del conjunto. El valor retornado por esta función es un manejador entero para utilizarse en todas las futuras interacciones con el conjunto. Si el conjunto nombrado no existe, el valor de retorno es -1.

long SetArrayLong (long id, long idx, long value);
Guarda un entero **value** en el conjunto referido por **id** en la posición especificada por **idx.** El valor de retorno es 1 se se realiza o 0 si falla. La operación fallará si el **id** del array es inválido.

long SetArrayString (long id, long idx, string str);
Guarda una cadena **string** en el conjunto referido por **id** y en la posición especificada por **idx**. El valor de retorno es 1 si se realiza o 0 si falla. La operación fallará si el **id** del conjunto es invalido.

string or long GetArrayElement (long tag, long id, long idx);
Mientras que las anteriores funciones se utilizan para guardar datos dependiendo del tipo de dato a guardar, esta es sólo una función para recuperar datos de un conjunto. Esta función recupera un valor entero o un valor string especificado en el índice **idx** del conjunto especificado en **id**. Lo que determina si el valor recuperado será un entero o una string es el parámetro **tag** el cual debe ser una de las dos constantes siguientes **AR_LONG** para recuperar un entero o **AR_STR** para recuperar una cadena.

long DelArrayElement (long tag, long id, long idx);
Borra los contenidos de la ubicación especificada en el conjunto especificado. El valor de **tag** determina si será borrado el valor entero o el valor cadena asociado al índice especificado.

void Delete Array (long id);
Borra el conjunto referenciado por **id** y todos los contenidos asociados a él. Una vez que el conjunto ha sido creado, continuará existiendo, aún después de que el script finalice, mientras no se haga una llamada a **DeleteArray** la cual borrará el conjunto de la base de datos en donde se creó.

long RenameArray (long id, string newname);
Renombra el conjunto referenciado por **id** por **newname**. Retorna 1 si se realiza o 0 si la operación falla.

Los posibles usos de los global arrays pueden ser, acercar variables globales, acercar tipos de dato complejos y proporcionar almacenamiento permanente a través de invocaciones en el script. Las variables globales de un script son simuladas creando un global array cuando se inicia el script y almacenando los valores globales en el conjunto. Estos valores globales están divididos entre los que para pedir acceso a las funciones tienen que pasarle el manejador del conjunto (**handle array**) o los que para acceder a una función necesitan realizar una búsqueda del nombre del conjunto deseado.

Los valores almacenados en un **IDC global array** persisten para toda la vida de la base de datos en que el script ha sido ejecutado. Podemos verificar la existencia de un conjunto examinando el valor de retorno de la función **CreateArray**. Si los valores almacenados en un conjunto sólo son aplicables a la invocación específica de un script, entonces dicho conjunto debe borrarse antes de que termine el script. Borrar el conjunto nos asegura que ningún valor global creado por la ejecución del script sea utilizado en la próxima utilización de dicho script.

Performance Bigundill@