16.6.—Opciones de diseño para pantalla de usuario del plug-in

Nuestra pretensión no es hacer ninguna guía de programación de pantallas de usuario. Sin embargo en muchas ocasiones el plug-in necesitará funcionar según lo que el usuario de IDA le indique para recabar la información que queremos nos muestre. Bien, pues además de las API **askXXX** mencionadas en el escrito 15, disponemos de unas cuantas funciones más complejas para interactuar con el usuario realizándolas a través de la IDA API. Para los que se aventuren en esta tarea hay que recordarles que los plugin se programan para la versión **Windows GUI de IDA**, para que tengan acceso completo a las funciones de pantalla de usuario disponibles en la **Windows API**. Por eso mismo es posible utilizar cualquier elemento de interconexión gráfica dentro de nuestros plugin.

16.6.1.—Construir elementos de enlace (pantalla) con el SDK

Mas allá de las funciones de enlace del SDK askXXX, tenemos "cosas" en el SDK para conseguir construir elementos de enlace con el usuario mucho más elaboradas. Una de las razones por las que se proporcionó al SDK un enlace de programación muy genérico fue el poder proporcionar un elemento GUI a un usuario y poder aceptar la entrada de un usuario.

16.6.1.1.—Utilizar los diálogos de elección (Choose) del SDK

La primera de las dos funciones de las cuales hablaremos es el grupo llamado **choose** y **choose2**. Cada una de estas funciones a través de distintas constantes utilizadas para controlar sus características, están declaradas en **kernwin.hpp**. El propósito de cada función es mostrar una lista de elementos dato al usuario preguntando a éste cual o cuales quiere escoger de la lista. Dichas funciones son capaces de mostrar cualquier tipo de dato en virtud de lo que se requiera, para especificar el formato de las funciones que son llamadas y generar cada línea de texto mostrada en la ventana de selección. La diferencia entre las dos funciones está en que **choose** muestra un listado de una sola columna, mientras que **choose2** es capaz de mostrar un listado multicolumna. En los siguientes ejemplos veremos los formatos más simples de ellas, las cuales dependen de algunos parámetros por defecto. Si queremos explorar todo el rango posible de capacidades de **choose y choose2**, tendremos que consultar **kernwin.hpp**.

Para mostrar una simple columna de información al usuario, la forma más simple de la función choose queda reducida a lo siguiente, una vez que los parámetros por defecto son omitidos:

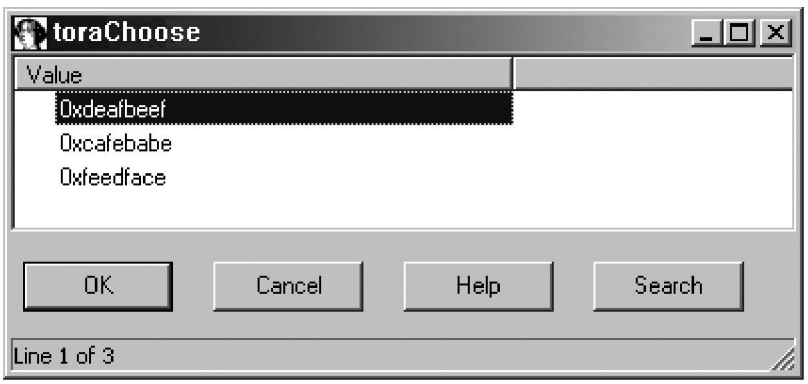
```
ulong choose (void *obj, int width, ulong (idaapi *sizer) (void *obj),
              char * (idaapi *getline) (void *obj, ulong n, char *buf),
              const char *title);
```

Aquí, El parámetro **obj** es un puntero al bloque de datos que será mostrado, y **width** es el ancho deseado de la columna que se utilizará en la ventana de selección. El parámetro **sizer** es un puntero a una función la cual es capaz de analizar los datos apuntados por **obj** y retornar el número de líneas necesarias para mostrar dichos datos. El parámetro **getline** es un puntero a una función que puede generar la cadena de caracteres representando a un elemento seleccionado desde **obj**. Observemos que el puntero **obj** puede apuntar a cualquier tipo de dato ya que la función sizer puede analizar dichos datos para determinar el número de líneas necesarias para mostrarlos y que la función **getline** puede ubicar un elemento dato específico utilizando un índice entero y generar una cadena de caracteres representando dicho elemento dato. El parámetro **title**

especifica la cadena título utilizada en el diálogo de selección generado. La función **choose** retorna un índice numérico **(1..n)** del elemento seleccionado por el usuario o **cero** si el diálogo es cancelado por el usuario. El código del siguiente ejemplo, se ha extraído de un plugin para mostrar el uso de la función choose.

```
#include <kernwin.hpp>

//Los datos que seran mostrados
int data[] = {0xdeafbeef, 0xcafebabe, 0xfeedface, 0};

//esta función obliga a obj a apuntar a cero cuando finalice el conjunto
//de enteros sin el cero.
ulong idaapi tora_sizer(void *obj) {
  int *p = (int*)obj;
  int count = 0;
  while (*p++) count++;
  return count;
}

/*
 * En esta funcion obj es obligado a apuntar a un conjunto de enteros
 * n indica que linea (1..n) mostrada debe ser formateada.
 * Si n es cero, se pide la línea de encabezado.
 * buf es un puntero al buffer de salida para el dato formateado.Nuestra
 * salida no deberá sobrepasar la anchura especificada en la llamada a choose.
 */
char * idaapi tora_getline(void *obj, ulong n, char *buf) {
  int *p = (int*)obj;
  if (n == 0) { //Caso del encabezado
    qstrncpy(buf, "Value", strlen("Value") + 1);
  }
  else { //Caso de datos
    qsnprintf(buf, 32, "0x%08.8x", p[n - 1]);
  }
  return buf;
}

void idaapi run(int arg) {
  int choice = choose(data, 16, tora_sizer, tora_getline, "toraChoose");
  msg("El usuario ha elegido %d\n", choose);
}
```

Activando el plugin del ejemplo nos da como resultado un diálogo de selección como



La función **choose2** nos ofrece la posibilidad de multicolumna en el diálogo de selección. Veamos la versión más simple de dicha función:

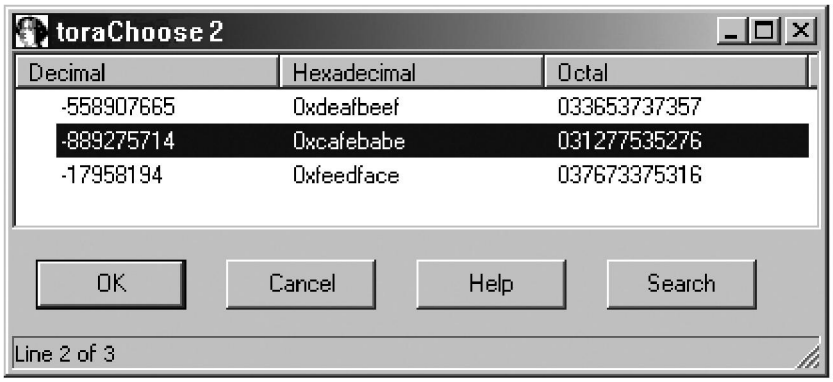
```
ulong choose2 (void *obj), int ncol, const int *widths, ulong (idaapi *sizer) (void *obj),
                Void (idaapi *getline) (void *obj), ulong n, char* const *cells),
                Const char *title);
```

Podemos observar unas cuantas diferencias entre choose2 y choose. En primer lugar, el parámetro **ncol** especifica el número de columnas a mostrar, mientras que el parámetro **widths** es un conjunto de enteros el cual especifica el ancho de cada columna. El formato de la función **getline** cambia un poco en choose2. Debido a que el diálogo choose2 puede contener varias columnas, la función **getline** debe de proporcionar los datos de cada columna con una sola línea. El siguiente ejemplo de código muestra el uso de choose2 en un plugin de ejemplo.

```
#include <kernwin.hpp>

//Los datos que se mostraran
int data[] = {0xdeafbeef, 0xcafebabe, 0xfeedface, 0};
//El ancho de cada columna
int widths[] = {16, 16, 16};
//Encabezado de cada columna
char *headers[] = {"Decimal", "Hexadecimal", "Octal"};
//El formato de cadena de cada columna
char *formats[] = {"%d", "0x%x", "0%o"};

//Esta funcion obliga a obj a apuntar a cero cuando el conjunto de enteros
//sin cero haya terminado.
ulong idaapi tora_sizer(void *obj) {
  int *p = (int*)obj;
  int count = 0;
  while (*p++) count++;
  return count;
}

/*
 * En esta función obj es obligado a apuntar a un conjunto de enteros
 * n indica que linea de (1..n) se está formateando.
 * Si n es cero, se requiera la linea de encabezado.
 * cells es un puntero a un conjunto de punteros de caracter. Este conjunto
 *      contiene un puntero para cada columna del diálogo. La salida de
 *      cada columna no podrá exceder la anchura especificada en el
 *      conjunto widths.
 */
void idaapi tora_getline_2(void *obj, ulong n, char* const *cells) {
  int *p = (int*)obj;
  if (n == 0) {
    for (int i = 0; i < 3; i++) {
      qstrncpy(cells[i], headers[i], widths[i]);
    }
  }
  else {
    for (int i = 0; i < 3; i++) {
      qsnprintf(cells[i], widths[i], formats[i], p[n - 1]);
    }
  }
}
```

```
void run(int arg) {
  int choice = choose2(data, 3, widths, tora_sizer, tora_getline_2,
                "toraChoose2");
  msg("La elección es %d\n", choose);
}
```

El diálogo multicolumna generado con el código anterior es el siguiente



Se pueden utilizar de forma más compleja tanto la función choose como choose2. Cada función es capaz de crear diálogos no modal o modal (“un diálogo modal es un diálogo que debe ser cerrado por el usuario antes de continuar obrando recíprocamente con la aplicación matriz del diálogo. Abrir y guardar archivos son ejemplos comunes de diálogos modales. Dichos diálogos son utilizados normalmente cuando una aplicación requiere la información de un usuario antes de que la aplicación pueda continuar su ejecución. Por otra parte, los diálogos no modales permiten al usuario continuar obrando recíprocamente con la aplicación matriz mientras que el diálogo permanezca abierto”), y cada función puede generar diálogos que nos permita la selección de varios elementos a la vez. También, cada función acepta distintos parámetros adicionales los cuales nos permiten notificar distintos eventos que ocurran, dentro del diálogo. Cuando utilicemos dichas funciones para crear diálogos no modales, el resultado es una ventana cliente MDI (Las **Windows Multiple Document Interface (MDI)** permite que múltiples ventanas hijo (cliente) sean contraídas dentro de una sola ventana contenedora. Todas las ventanas **subview** de IDA son creadas como ventanas **MDI** (cliente del escritorio de IDA) con su propia solapa añadida en donde se muestran las otras solapas de las ventanas de IDA, como la ventana **Names**. De hecho la ventana Names de IDA está ejecutada utilizando **choose2.** Para más información sobre las capacidades de choose y choose2, ver **kernwin.hpp**.

16.6.1.2.—Crear formularios a medida con el SDK

Para poder crear elementos de conexión con el usuario más complejos, el SDK nos proporciona la función **AskUsingForm_c**. El prototipo de dicha función es la siguiente:

```
int AskUsignForm_c (const char *form, …);
```

La función parece bastante simple, sin embargo es la más compleja entre las funciones de pantalla de usuario disponibles en el SDK. Esta complejidad es debida a la naturaleza del argumento **form,** el cual es utilizado para especificar el esquema de varios elementos de la pantalla de usuario dentro del diálogo. **AskUsignForm_c** es similar a **printf** en la que el argumento **form** es esencialmente un formato de cadena que describe el esquema de los distintos elementos de entrada. Donde el formato de cadena de **printf**

utiliza un formato de salida especificado, este es reemplazado con procesos de elementos de entrada cuando el formulario es mostrado. **AskUsingForm_c** reconoce un completo conjunto de diferentes campos de salida como printf. Estos campos específicos están detallados en **kernwin.hpp** al igual que toda la documentación completa de AskUsignForm_t. El formato básico de un campo del formulario es el siguiente:

<#hint text#label:type:width:swidth:@hlp [ ]>

Los componentes individuales del campo de formulario específico se describen seguidamente en la siguiente lista:

#hint text# Este elemento es opcional. Si está presente el texto entre #--# se mostrará, sin los signos, como aviso de herramienta cuando el ratón pase por encima del campo de entrada asociado.

label Se muestra un texto estático como una etiqueta a la izquierda del campo de entrada asociado. En el caso de un campo button, este será el texto del botón.

type Un simple carácter indicando el tipo del campo de formulario especificado. Los tipos de los campos de formulario se describen en el siguiente listado.

width Máximo número de caracteres aceptado en el campo de entrada asociado. En el caso de un campo button, este campo especifica un entero, código de identificación del botón para distinguirlo de otro.

swidth Muestra la anchura del campo de entrada.

@hlp [ ] Este campo es descrito en kernwin.hpp como “the number of help screen from the IDA.HLP file”. El contenido de este archivo es dictado por Hex-Rays, por lo tanto es improbable que se utilice en la mayoría de los casos. Lo substituimos con dos puntos para que sea ignorado.

Los caracteres utilizados por el campo **type** especifica qué tipo de campo de entrada se generará cuando el diálogo se ejecute. Cada tipo de campo de formulario requiere un parámetro asociado en la parte de argumentos variable de la lista de parámetros de AskUsignForm_c. Los tipos específicos del campo de formulario y su parámetro asociado se muestran en el siguiente listado, tomados de kernwin.hpp:

```
Input field types                va_list parameter
-----------------                -----------------

A - ascii string                 char* at least MAXSTR size
S - segment                      sel_t*
N - hex number, C notation       uval_t*
n - signed hex number, C notation    sval_t*
L - default base (usually hex) number,  ulonglong*
   C notation
l - default base (usually hex) number,  longlong*
   signed C notation
M - hex number, no "0x" prefix       uval_t*
D - decimal number               sval_t*
O - octal number, C notation         sval_t*
```

Y - binary number, "0b" prefix sval_t*
H - char value, C notation sval_t*
$ - address ea_t*
I - ident char* at least MAXNAMELEN size
B - button formcb_t button callback function
K - color button bgcolor_t*
C - checkbox ushort* bit mask of checked boxes
R - radiobutton ushort* number of selected radiobutton

Todos los campos numéricos interpretan la entrada del usuario como una expresión IDC que se analiza y evalúa cuando el usuario hace clic sobre el botón OK del diálogo. Todos los campos requieren un argumento puntero para utilizarse tanto en la entrada como en la salida de datos. Cuando el formulario se genera por primera vez, los valores iniciales de todos los campos de formulario tienen sus punteros asociados. El puntero argumento asociado a un campo botón es la dirección de una función la cual será llamada si el botón asociado a esta es pulsado. La función **formcb_t** se define como sigue.

```
// callback for buttons
typedef void (idaapi *formcb_t) (TView *fields [ ], int code);
```

El argumento **code** al callback del botón representa el valor asociado al código (width) cuando el botón es pulsado. Al utilizar una declaración interruptor al verificar este código, podemos utilizar una simple función para procesar algunos botones distintos.

La sintaxis para especificar controles para un radio button o checkbox difieren un poco con respecto al formato de otros tipos de campos formulario. Estos campos utilizan el siguiente formato:

```
<#item hint#label:type>
```

Los radio button y checkbox se pueden agrupar por listas según sus especificaciones y denotan el final de cada lista utilizando el siguiente formato especial (un > extra al final).

```
<#item hint#label:type>>
```

Un grupo de radio button o checkbox serán enmarcados para realzar el grupo. Podemos poner título al agrupamiento utilizando un formato especial cuando especificamos el primer elemento en el grupo, como podemos ver:

```
<#item hint#title#box hint#label:type>
```

Si queremos tener un título de caja pero no queremos utilizar ningún indicador, podemos omitirlos con el siguiente formato:

```
<##title##label:type>
```

Una vez aquí lo mejor será mostrar un ejemplo de un diálogo construido con AskUsingForm_c.

Los formatos de cadena creados para los diálogos **AskUsingForm_c** se realizan en líneas individuales que especifican cada aspecto deseado del diálogo. Además del campo de formulario específico, la **format string** puede contener texto estático para mostrar en el diálogo. También puede contener un título para el diálogo, el cual debe estar seguido de dos retornos de carro, y una o más directivas de características como **STARTITEM** el cual especifica el índice del campo del formulario el cual es activado inicialmente cuando el diálogo es mostrado por primera vez. Las format string para crear el diálogo anterior es el siguiente:

```
char *dialog =
 "STARTITEM 0\n"          //El primer elemento toma el foco de entrada
 "Este es el titulo\n\n"  //seguido por 2 nuevas lineas
 "Esto es texto estatico\n"
 "<String:A:32:32::>\n"     //Un campo de entrada ASCII, necesita char[MAXSTR]
 "<Decimal:D:10:10::>\n"  //Un campo de entrada decimal, sval_t*
 "<#No leading 0x#Hex:M:8:10::>\n"  //Un campo de entrada Hex con chivato, uval_t*
 "<Button:B:::>\n"                  //Un campo button, formcb_t
 "<##Radio Buttons##Radio 1:R>\n"   //Un radio button con título de enmarcado
 "<Radio 2:R>>\n"                    //Ultimo radio button en el grupo
                                    //ushort* numero de radio seleccionado
 "<##Check Boxes##Check 1:C>\n"   //Un campo checkbox con título de enmarcado
 "<Check 2:C>>\n";                //Ultimo checkbox en el grupo
                                    //ushort* bitmask de checks
```

Para formatear la especificación del diálogo que tenemos, hay que hacerlo cada elemento por línea, hemos de intentar de hacerlo lo más fácilmente posible consiguiendo combinar cada especificación del campo con la figura. Como vimos en la figura anterior todo el texto y los campos den entrada numéricos aparecen en sus controles adecuados. Con la intención de ahorrar tiempo, IDA rellena cada lista con los valores entrados recientemente los cuales concuerden con el tipo asociado al campo de entrada. El siguiente código de plug-in puede utilizarse para mostrar el diálogo anterior y procesar cualquier resultado:

```
void idaapi button_func(TView *fields[], int code)
{
  msg("Se ha pulsado el boton!\n");
}

void idaapi run(int arg)
{
  char input[MAXSTR];
  sval_t dec = 0;
  uval_t hex = 0xdeadbeef;
  ushort radio = 1;      //botón selccionado inicialmente 1
  ushort checkmask = 3;  //ambos checkbox seleccionados inicialmente
  qstrncpy(input, "valor inicial", sizeof(input));
  if (AskUsing Form_c(dialog, input, &dec, &hex,
                button_func, &radio, &checkmask) == 1) {
    msg("La cadena entrada ha sido: %s\n", input);
    msg("Decimal: %d, Hex %x\n", dec, hex);
    msg("Radio button %d seleccionado\n", radio);
    for (int n = 0; checkmask; n++) {
      if (checkmask & 1) {
        msg("Checkbox %d seleccionado\n", n);
      }
      checkmask >>= 1;
    }
  }
}
```

Observemos que cuando se procesan los resultados de radio button y checkbox, el primer botón de cada grupo es considerado botón cero.

La función AskUsingForm_c proporciona un considerable potencial para diseñar los elementos de la pantalla de usuario para nuestros plug-in. El ejemplo que hemos explicado sólo explora algunas capacidades de la función, otras están especificadas en **kernwin.hhp**.

16.6.1.3.—Otras técnicas para generar pantallas de usuario

Muchos programadores se han roto los “cuernos” con el problema de crear pantallas de usuario para sus plug-in. El autor de **mIDA** plug-in de Tenable Security desarrolla un acercamiento alternativo para crear ventanas cliente MDI. En los foros de IDA se pueden hallar ejecuciones realizadas con mIDA y sus correspondientes códigos mostrando los ejemplos.

El **ida-x86emu** plug-in utiliza otros tipos de acercamiento para las pantallas de usuario. Este plug-in de hecho depende de un manejador a la ventana principal de IDA el cual se puede obtener utilizando el siguiente código SDK:

```
HWND mainWindiw = (HWND) callui (ui_get_hwnd) .vptr;
```

Utilizando la ventana principal de IDA como padre, ida-x86emu no atenta sobre el funcionamiento del espacio de trabajo de IDA. Todos los diálogos de enlace son generados utilizando un editor de recursos Windows, y todas las interacciones del usuario son manipuladas directamente por funciones Windows API. La utilización del editor gráfico conjuntamente con llamadas directas a funciones nativas API Windows nos proporciona el mayor potencial de capacidades para generar pantallas de usuario a

expensas de una complejidad añadida y el conocimiento adicional requerido para procesar mensajes de Windows y trabajar con funciones de bajo nivel.

Tanto el ida-x86emu plug-in como el mIDA plug-in lo estudiaremos en partes posteriores.

Performance Bigundill@