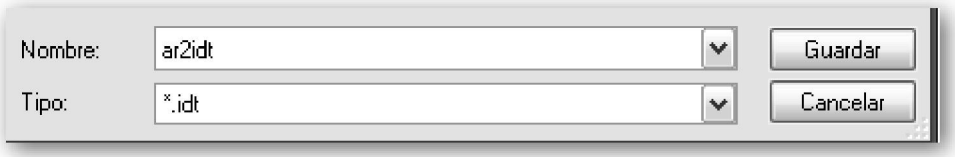
14.5.4.—Enumerar funciones exportadas

En el escrito 12 hablamos sobre la utilización de **idsutils** para generar archivos **.ids**, los cuales describen los contenidos de las librerías compartidas. Recordemos que el primer paso para generar un archivo **.ids** era generar un archivo **.idt**, el cual es un archivo texto que contiene las descripciones de cada función exportada contenida en la librería. El lenguaje IDC contiene funciones para iterar a través de las funciones que son exportadas de una librería compartida. El siguiente script, cuando se ejecuta nos genera un archivo **.idt** después de abrir ida una librería compartida:

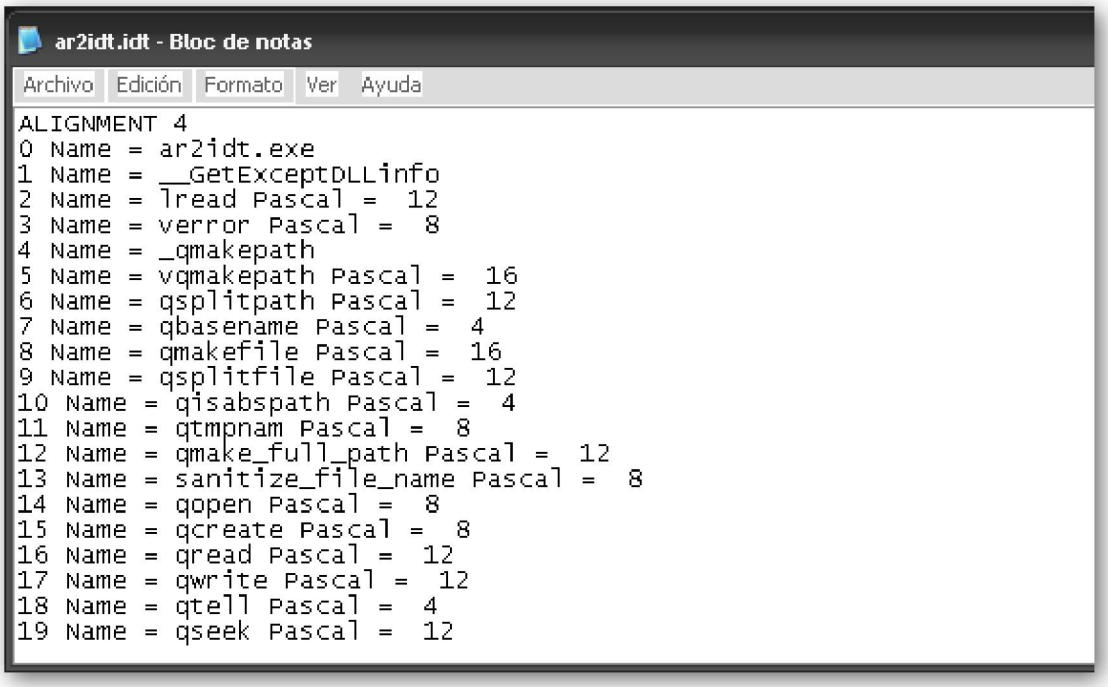
```
#include <idc.idc>
static main ( )   {
   auto puntoEntrada, i, ordinal, direccion, nombre, purgados, archivo, archivoDest;
   archivo = AskFile (1, ".idt", "Selecciona donde guardar el archivo IDT");
   archivoDest = fopen (archivo, "w");
   puntoEntrada = GetEntryPointQty ( );
   fprintf (archivoDest, "ALIGNMENT 4\n");
   fprintf (archivoDest, "0 Name = %s\n", GetInputFile ( ));
   for (i = 0; i < puntoEntrada; i++)   {
         ordinal = GetEntryOrdinal (i);
         if (ordinal == 0) continue;
         direccion = GetEntryPoint (ordinal);
         if (ordinal == direccion)   {
            continue; //punto de entrada no tiene ordinal
         }
         nombre = Name (direccion);
         fprintf (archivoDest, "%d Name = %s", ordinal, nombre);
         purgados = GetFunctionAttr (direccion, FUNCATTR_ARGSIZE);
         if (purgados > 0)   {
            fprintf (archivoDest, " Pascal =  %d", purgados);
         }
         fprintf (archivoDest, "\n");
   }
}
```

La salida del script es guardada en un archivo elegido por el usuario. Las funciones nuevas que se han utilizado en este script son **GetEntryPointQty**, la cual retorna el número de símbolos exportados de la librería; **Get EntryOrdinal**, la cual retorna un número ordinal, un índice de la tabla de exportación de la librería; **GetEntryPoint**, la cual retorna la dirección asociada con una función exportada la cual es definida por el número ordinal; y **GetInputFile**, la cual retorna el nombre del archivo que ha sido cargado en IDA.

Veamos el resultado de dicho script en el archivo **ar2idt.exe**, antes como siempre, hemos tenido que guardar nuestro script **enumFunExp.idc** en la carpeta **idc** de ida. Seguidamente cargamos el archivo y elegimos el script con **File > IDC file**. Cuando lo ejecutamos lo primero que nos pide es que demos un nombre al archivo **.idt** y el lugar dónde le queremos guardar. También hemos editado una cabecera del diálogo la cual nos indica dónde queremos guardar el archivo.



Esta acción nos crea un archivo **ar2idt.idt**, si lo abrimos con un editor de texto, podemos ver lo siguiente:

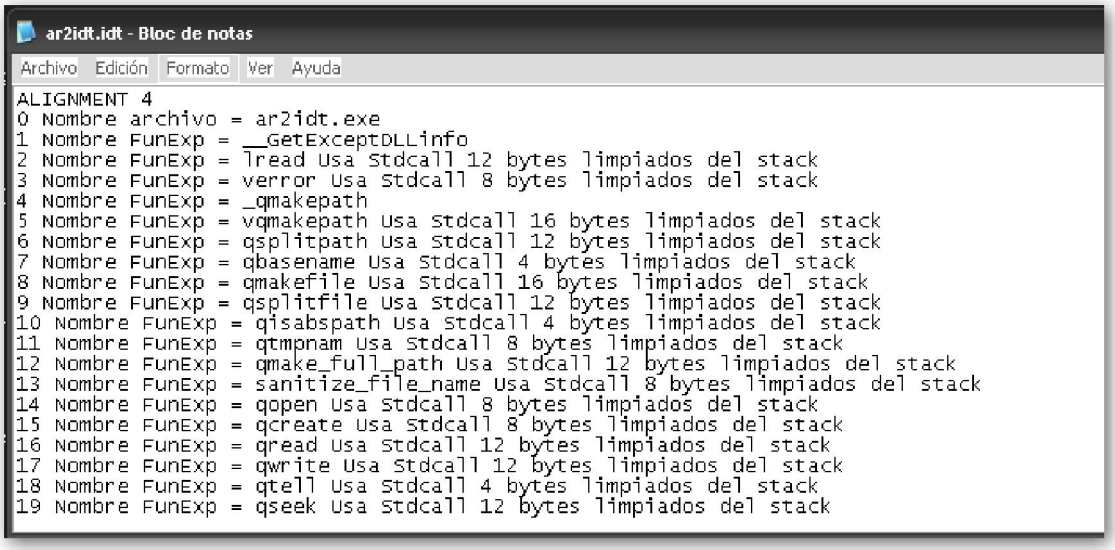


Ahora ya tenemos nuestro archivo .idt para poder crear el archivo .ids.

Por otro lado si solamente queremos tener una relación de las funciones importadas si estas utilizan la llamada Stdcall y si la utiliza saber cuantos byte serán limpiados de la pila en su retorno, lo podemos modificar de esta forma.

```
#include <idc.idc>
static main ( )   {
   auto puntoEntrada, i, ordinal, direccion, nombre, purgados, archivo, archivoDest;
   archivo = AskFile (1, ".txt", "Selecciona donde guadar el archivo TXT");
   archivoDest = fopen (archivo, "w");
   puntoEntrada = GetEntryPointQty ( );
   fprintf (archivoDest, "ALIGNMENT 4\n");
   fprintf (archivoDest, "0 Nombre archivo = %s\n", GetInputFile ( ));
   for (i = 0; i < puntoEntrada; i++)   {
         ordinal = GetEntryOrdinal (i);
         if (ordinal == 0) continue;
         direccion = GetEntryPoint (ordinal);
         if (ordinal == direccion)   {
            continue; //punto de entrada no tiene ordinal
         }
         nombre = Name (direccion);
         fprintf (archivoDest, "%d Nombre FunExp = %s", ordinal, nombre);
         purgados = GetFunctionAttr (direccion, FUNCATTR_ARGSIZE);
         if (purgados > 0)   {
            fprintf (archivoDest, " Usa Stdcall  %d bytes limpiados del stack", purgados);
         }
         Fprintf (archivoDest, "\n");
    }
}
```

Cuando lo ejecutemos nos creara un archivo txt con los siguientes resultados.



Performance Bigundill@

14.5.5.—Hallar y etiquetar los argumentos de una función

Las versiones de gcc posteriores a la 3.4 utilizan las declaraciones **mov** en vez de **push** ,en declaraciones binarias x86, para colocar los argumentos de una función en la pila antes de llamar a esta. Esto produce algunos problemas de análisis en IDA, debido a que el motor de análisis depende de las declaraciones **push** encontradas para apuntar con precisión las ubicaciones en las cuales serán colocados los argumentos para la llamada a una función. El siguiente listado muestra un desensamblado de IDA cuando los parámetros son colocados en la pila:

```
.text:0040FA83                push    ebx             ; n
.text:0040FA84                mov     eax, [ebp+ptr]
.text:0040FA87                push    eax             ; ptr
.text:0040FA88                mov     edx, [ebp+arg_0]
.text:0040FA8B                mov     ecx, [edx+4]
.text:0040FA8E                push    ecx             ; stream
.text:0040FA8F                call    qfread
```

Observemos los comentarios que ida ha colocado en el margen derecho. Estos comentarios son posibles sólo cuando ida reconoce los parámetros que serán colocados en la pila y cuando ida conoce la firma de la función que es llamada.

Por otro lado cuando son utilizadas declaraciones tipo **mov** para colocar parámetros en la pila, el resultado del desensamblado es mucho menos informativo, ver figura.

```
.text:004010EB                mov     [esp+18h+var_10], 3
.text:004010F3                mov     [esp+18h+var_14], 2
.text:004010FB                mov     [esp+18h+var_18], 1
.text:00401102                call    sub_401090
```

Como podemos ver en los casos de declaración **mov**, utilizados para colocar los parámetros de la función no se ha podido colocar automáticamente ningún comentario.

El siguiente script es capaz de restaurar información en el desensamblado, reconociendo automáticamente las instrucciones que preparan los parámetros para la llamada a la función:

```
#include <idc.idc>
static main() {
  auto direccion, operando, final, idx;
  auto funcion_banderas, tipo, valor, busqueda;
  busqueda = SEARCH_DOWN | SEARCH_NEXT;
  direccion = GetFunctionAttr(ScreenEA(), FUNCATTR_START);
  funcion_banderas = GetFunctionFlags(direccion);
  if (funcion_banderas & FUNC_FRAME) {  //¿Es una estructura basada en ebp?
    final = GetFunctionAttr(direccion, FUNCATTR_END);
    for (; direccion < final && direccion != BADADDR; direccion = FindCode(direccion, busqueda))
{
      tipo = GetOpType(direccion, 0);
      if (tipo == 3) {  //¿Es un operando indirecto a un registro?
        if (GetOperandValue(direccion, 0) == 4) {   //¿Es el registro esp?
          MakeComm(direccion, "arg_0");  //[esp] identico a arg_0
        }
```

```
    }
    else if (tipo == 4) {  //¿Es un operando de registro + desplazamiento?
      idx = strstr(GetOpnd(direccion, 0), "[esp"); //¿Es el registro esp?
      if (idx != -1) {
        valor = GetOperandValue(direccion, 0);   //toma el desplazamiento
        MakeComm(direccion, form("arg_%d", valor));  //añade un comentario
      }
    }
  }
 }
}
```

Dicho script sólo funciona en estructuras basadas en EBP y depende del hecho en que cuando los parámetros son movidos a la pila antes de la llamada a la función, **gcc** genera las referencias de memoria relativas a esp. El script itera por todas las instrucciones dentro de una función y para cada instrucción que escribe a una ubicación de memoria utilizando **esp** como registro, el script determina la posición en la pila y añade un comentario indicando el parámetro que se está moviendo. La función **GetFunctionFlags** nos proporciona acceso a distintas banderas asociadas a una función, de la misma forma que si la función estuviera basada en **ebp**. Ejecutar el script en el ejemplo de la figura anterior nos lo convertiría a esta vista:

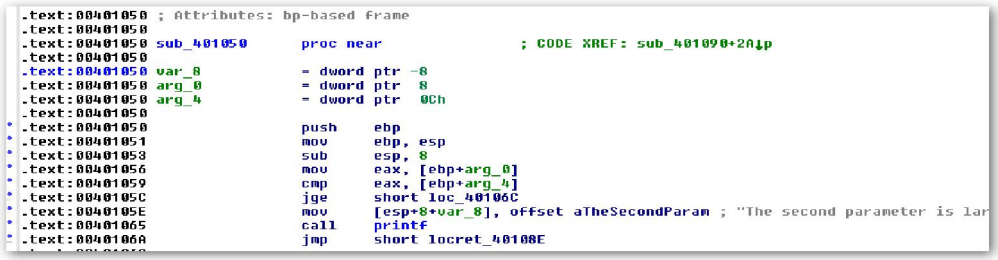
Los comentarios agregados no son particularmente informativos. Sin embargo ahora podemos, con un solo vistazo, saber que las tres declaraciones **mov** se utilizan para colocar parámetros en la pila, lo cual es un adelanto. Si completamos más nuestro script explorando algunas capacidades de IDC, podemos obtener un script el cual nos proporciona casi la misma información que IDA cuando reconoce perfectamente los parámetros.

```
#include <idc.idc>
static getArgCount(funcion) {
  auto tipo, idx, contador;
  tipo = GetType(funcion);
  if (tipo != "") {
    if (strstr(tipo, "()") != -1) return 0;
    if (strstr(tipo, "( )") != -1) return 0;
    if (strstr(tipo, "(void)") != -1) return 0;
    idx = strstr(tipo, "(");
    if (idx != -1) {
      contador = 1;
      while (strstr(tipo, ",") != -1) {
        idx = strstr(tipo, ",");
        contador++;
        tipo = substr(tipo, idx + 1, -1);
      }
      return contador;
    }
  }
  return -1;
}
static tomaArg(funcion, n, nargumentos) {
```

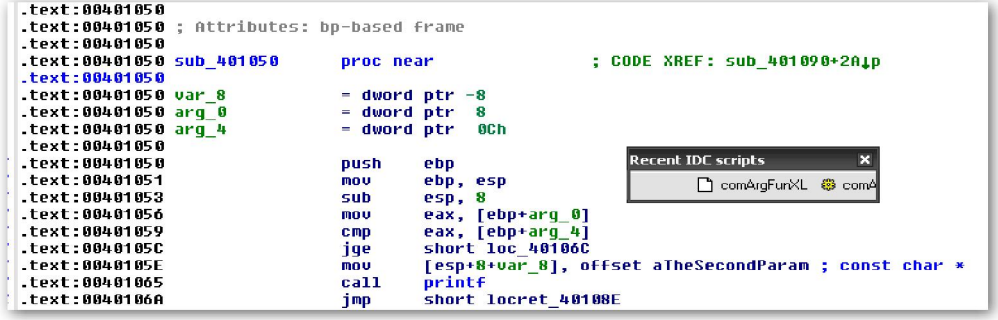
```
  auto tipo, idx, contador;
  tipo = GetType(funcion);
  if (tipo != "") {
    if (strstr(tipo, "()") != -1) return "";
    if (strstr(tipo, "( )") != -1) return "";
    if (strstr(tipo, "(void)") != -1) return "";
    idx = strstr(tipo, "(");
    if (idx != -1) {
      contador = 1;
      do {
        tipo = substr(tipo, idx + 1, -1);
        Message("%d/%d: %s\n", contador, nargumentos, tipo);
        idx = strstr(tipo, ",");
        if (contador == n) {
          if (idx == -1) {
            idx = strstr(tipo, ")");
          }
          return substr(tipo, 0, idx);
        }
        idx = strstr(tipo, ",");
        contador++;
      } while (contador <= nargumentos);
    }
  }
  return "";
}
static toma_arg(ea, n) {
  auto operando, objetivo, flujo, final, nargumentos;
  final = GetFunctionAttr(ea, FUNCATTR_END);
  while (ea < final && ea != BADADDR) {
    objetivo = Rfirst0(ea);
    if (objetivo != BADADDR) {
      flujo = XrefType();
      if (flujo == fl_CF || flujo == fl_CN) {
        Message("llamada encontrada en %x, el objetivo es %x\n", ea, objetivo);
        nargumentos = getArgCount(objetivo);
        Message("argumentos contados = %d\n", nargumentos);
        if (nargumentos == -1) {
          return "";
        }
        if (n <= nargumentos) {
          return tomaArg(objetivo, n, nargumentos);
        }
      }
    }
    ea = FindCode(ea, SEARCH_DOWN | SEARCH_NEXT);
  }
  return "";
}
static main() {
  auto funcion, ea, comentario, operando, maximo, argumento, idx;
  auto funcion_banderas, tipo, valor, call_loc;
  funcion = GetFunctionAttr(ScreenEA(), FUNCATTR_START);
  funcion_banderas = GetFunctionFlags(funcion);
  if (funcion_banderas & FUNC_FRAME) {
    maximo = GetFunctionAttr(funcion, FUNCATTR_END);
    for (ea = funcion; ea < maximo && ea != BADADDR; ea = FindCode(ea, SEARCH_DOWN |
SEARCH_NEXT)) {
      tipo = GetOpType(ea, 0);
      if (tipo == 3) {
```

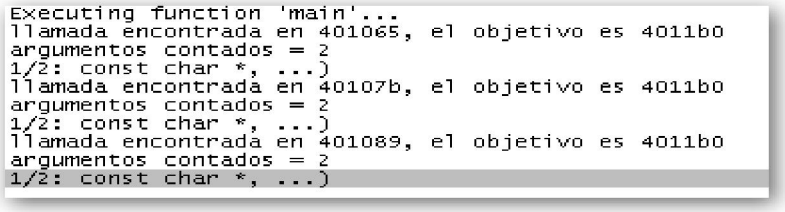
```
        //base + indice
        if (GetOperandValue(ea, 0) == 4) {  //esp
          argumento = toma_arg(ea, 1);
          if (argumento != "") {
            comentario = argumento;
          }
          else {
            comentario = "arg_0";
          }
          MakeComm(ea, comentario);
        }
      }
      else if (tipo == 4) {
        //base + desplazamiento + indice
        operando = GetOpnd(ea, 0);
        idx = strstr(operando, "[esp");
        if (idx != -1) {
          valor = GetOperandValue(ea, 0);
          argumento = toma_arg(ea, valor / 4 + 1);
          if (argumento != "") {
            comentario = argumento;
          }
          else {
            comentario = form("arg_%d", valor);
          }
          MakeComm(ea, comentario);
        }
      }
    }
  }
}
```

Veamos que información nos proporciona, la primera figura nos muestra una función sin aplicarle el script:



En esta segunda figura le hemos aplicado el script y vemos la información que nos proporciona en la ventana de desensamblado y en la de mensajes de ida:





Performance Bigundill@