15.2.5,-- Técnicas de iteración utilizando IDA API

Usando la IDA API, tenemos distintas formas para iterar por distintos objetos de una base de datos. En los ejemplos siguientes aprenderemos algunas de las técnicas de iteración más comunes:

15.2.5.1.—Enumerar funciones

La primera técnica de iteración a través de las funciones dentro de una base de datos es idéntica a la forma en que ejecutamos dicha tarea utilizando IDC:

```
for (func_t *f = get_next_func(0); f != NULL; f = get_next_func(f->startEA))
{
  char fname[1024];
  get_func_name(f->startEA, fname, sizeof(fname));
  msg("%08x: %s\n", f->startEA, fname);
}
```

Como alternativa, simplemente podemos iterar por las funciones utilizando sus índices numerados, veámoslo en el siguiente ejemplo:

```
for (int idx = 0; idx < get_func_qty(); idx++)
{
  char fname[1024];
  func_t *f = getn_func(idx);
  get_func_name(f->startEA, fname, sizeof(fname));
  msg("%08x: %s\n", f->startEA, fname);
}
```

Para finalizar, también podemos trabajar a un nivel más inferior y utilizaremos una estructura dato llamada **areacb_t**, también conocida como **area control block**, definida en **area.hpp**. Los bloques de área de control son utilizados para mantener los listados de objetos **area_t**. Una **areacb_t** global llamada **funcs** es exportada (**funcs.hpp**) como parte de la IDA API. Utilizando la clase **areacb_t**, el ejemplo anterior se podría reescribir de la siguiente forma:

```
int a = funcs.get_next_area(0);
while (a != -1)
{
  char fname[1024];
        func_t *f = (func_t*)funcs.getn_area(a);  // getn_area retorna un area_t
  get_func_name(f->startEA, fname, sizeof(fname));
  msg("%08x: %s\n", f->startEA, fname);
        a = funcs.get_next_area(f->startEA);
}
```

En este ejemplo, los elementos de función **get_next_area** (**int a = funcs.get_next_area(0);** y **a = funcs.get_next_area(f->startEA);**) son utilizados repetidamente para obtener los valores índices de cada área del bloque de control **funcs**. Se obtiene un puntero a cada área encontrada **func_t**, supliendo cada valor índice al elemento función **getn_area** (**func_t *f = (func_t*)funcs.getn_area(a);**). Distintas variables globales **areacb_t** están declaradas en el SDK, incluida la **segs**, la cual es un área de control que contiene los punteros **segment_t** para cada sección del binario.

15.2.5.2.— Enumerar elementos de estructura

En el SDK, el **stack frame** se modela utilizando las capacidades de la clase **struct_t**. El ejemplo siguiente utiliza la iteración de elementos de estructura como medio para imprimir el contenido de un stack frame.

```
func_t *func = get_func(get_screen_ea());  //toma la función en la que está ubicada el cursor
msg("El tamano de la variable local es %d\n", func->frsize);
msg("Tamano de los regs guardados es %d\n", func->frregs);
struc_t *frame = get_frame(func);          //toma el puntero al stack frame
if (frame) {
  size_t ret_addr = func->frsize + func->frregs;  //offset a la dirección de retorno
  for (size_t m = 0; m < frame->memqty; m++) {    //bucle a traves de los elementos
    char fname[1024];
    get_member_name(frame->members[m].id, fname, sizeof(fname));
    if (frame->members[m].soff < func->frsize) {
      msg("Variable local ");
    }
    else if (frame->members[m].soff > ret_addr) {
      msg("Parametro ");
    }
    msg("%s esta en el frame offset %x\n", fname, frame->members[m].soff);
    if (frame->members[m].soff == ret_addr) {
      msg("%s es la direccion de retorno\n", fname);
    }
  }
}
```

Este ejemplo resume el stack frame de una función utilizando la información de la función dada por el objeto **func_t** y asociándola a la representación del stack frame por **struct_t.** Los campos **frsize** y **frregs** especifican respectivamente, el tamaño de la variable local en el stack frame y el número de byte dedicados a guardar los registros. La dirección de retorno se puede encontrar dentro de la estructura frame de variables locales y de los registros guardados. Dentro de la misma estructura frame, el campo **memqty** especifica el número de elementos definidos en la estructura del frame, lo cual también corresponde al tamaño del conjunto de **members**. Se utiliza un bucle para recuperar el nombre de cada elemento y determinar si el elemento es una variable local o un argumento basándose en el offset de inicio (**soff**) dentro de la estructura del frame.

15.2.5.3.—Enumerar referencias cruzadas

En el escrito 14 vimos cómo era posible enumerar referencias cruzadas utilizando scripts IDC. En el SDK existen las mismas capacidades, sin embargo su forma es algo diferente. Como ejemplo retomaremos la tarea de listar todas las llamadas a una función en particular.

```
void listado_llamadores(char *bad_func) {
  char name_buf[MAXNAMELEN];
  ea_t func = get_name_ea(BADADDR, bad_func);
  if (func == BADADDR) {
    warning("Lo siento, %s no se encuentra en la base de datos", bad_func);
  }
  else {
    for (ea_t addr = get_first_cref_to(func); addr != BADADDR;
         addr = get_next_cref_to(func, addr)) {
      char *name = get_func_name(addr, name_buf, sizeof(name_buf));
      if (name) {
        msg("%s es llamada desde 0x%x en %s\n", bad_func, addr, name);
```

```
      }
      else {
        msg("%s es llamada desde 0x%x\n", bad_func, addr);
      }
    }
  }
}
```

La razón de que esta función no trabaje igual que en IDC, es que no hay forma para determinar el tipo de referencia cruzada retornada en cada iteración del bucle, recordemos que en SDK no existe la función IDC **XrefType**. En este caso deberíamos verificar para cada referencia cruzada a la función dada si es una llamada de referencia cruzada del tipo **fl_CN** o **fl_CF**.

Cuando necesitemos determinar dentro del SDK el tipo de una referencia cruzada, deberemos utilizar una forma de iteración de referencias cruzadas alternativa facilitada por la estructura **xrefblk_t**, la cual está descrita en **xref.hpp**. El esquema básico de una **xrefblk_t** se muestra en el siguiente ejemplo. Para más detalles, ver **xref.hpp**.

```
struct xrefblk_t {
  ea_t from;     // la direccion a referenciar - rellenada por first_to(),next_to()
  ea_t to;       // la direccion referenciada - rellenada por first_from(),next_from()
  uchar iscode;  // 1-es referencia de código; 0-es referencia de dato
  uchar type;    // tipo de la última referencia retornada
  uchar user;    // 1-xref definida por el usuario, 0-definida por ida

  //rellena el campo "to" con la primera direccion referida en "from".
 bool first_from(ea_t from, int flags);
  //rellena el campo "to" con la siguiente dirección referida en "from".
  //Esta funcion asume una llamada previa a first_from.
 bool next_from(void);
  //rellena elcampo "from" con la primera dirección que refiere a "to".
 bool first_to(ea_t to,int flags);
  //rellena el campo "from" con la siguiente direccion que refiere a "to".
  //Esta funcion asume una llamada previa a first_to.
 bool next_to(void);
}
```

Las funciones elemento de **xrefblk_t** **(bool first_from(ea_t from, int flags);** y **bool first_to(ea_t to,int flags);)** son utilizadas para inicializar la estructura y (**bool next_from(void);** y **bool next_to(void);)** para ejecutar la iteración, mientras que los elementos dato se utilizan para acceder a la información de la última referencia cruzada recuperada. El valor **flags** requerido por las funciones **first_from** y **first_to** dictan el tipo de referencia cruzada que se debe retornar. Los valores admitidos por el parámetro **flags** son los siguientes, (sacados de **xref.hpp**):

```
#define XREF_ALL      0x00        // retorna todas las referencias
#define XREF_FAR      0x01        // no retorna las referencias cruzadas de flujo ordinario
#define XREF_DATA     0x02        // solamente retorna las referencias de dato
```

Observemos que ningún valor del flag restringe el retorno de referencias de código. Si estamos interesados en las referencias de código, debemos comparar el campo **xrefblk_t type** con los tipos de referencia cruzada específicos, tal como **fl_JN**, o verificar el campo **iscode** para determinar si la última referencia cruzada retornada es una referencia cruzada de código.

La siguiente versión modificada de **listado_llamadores**, muestra el uso de una estructura de iteración **xrefblk_t**

```
void listado_llamadores(char *bad_func) {
  char name_buf[MAXNAMELEN];
  ea_t func = get_name_ea(BADADDR, bad_func);
  if (func == BADADDR) {
    warning("Lo siento, %s no existe en la base de datos", bad_func);
  }
  else {
    xrefblk_t xr;
    for (bool ok = xr.first_to(func, XREF_ALL); ok; ok = xr.next_to()) {
      if (xr.type != fl_CN && xr.type != fl_CF) continue;
      char *name = get_func_name(xr.from, name_buf, sizeof(name_buf));
      if (name) {
        msg("%s es llamada desde 0x%x en %s\n", bad_func, xr.from, name);
      }
      else {
        msg("%s es llamada desde 0x%x\n", bad_func, xr.from);
      }
    }
  }
}
```

Utilizando una **xrefblk_t** , ahora tenemos la oportunidad de examinar en (**if (xr.type != fl_CN && xr.type != fl_CF) continue;)** el tipo de cada referencia cruzada retornada por el bucle y decidir si es la que nos interesa o no. En este ejemplo sencillamente ignora cualquier referencia cruzada que no esté relacionada con una llamada a función. No utilizamos el elemento **iscode** de **xrefblk_t** debido a que **iscode** es **true** en los saltos y en las referencias cruzadas de flujo ordinario, además de las llamadas a referencias cruzadas. Así pues **iscode** no nos da la garantía de que la referencia cruzada actual está relacionada con una llamada de función.

Performance Bigundill@