18.3.3.-- El analizador

En este momento y como ya hemos rellenado bastante la estructura **LPH** podemos empezar a pensar en la primera parte de un módulo de procesador que será la que ejecuta el analizador. En los ejemplos de módulos procesador, el analizador normalmente es ejecutado por una función llamada **ana** (no obstante, la podemos nombrar como queramos) la cual se encuentra en un archivo nombrado **ana.cpp**. El prototipo de dicha función es muy simple, como podemos ver a continuación:

```
int idaapi ana(void); //analiza una instrucción y retorna el largo de dicha instruccion
```

Deberemos inicializar el elemento **u_ana** en LPH con un puntero a nuestra función analizador. La tarea del analizador es analizar una instrucción, rellenando la variable global **cmd** con información respecto a dicha instrucción, y retornando el largo de la instrucción. El analizador no realizará ningún cambio en la base de datos.

La variable **cmd** es un proceso global de un objeto **insn_t.** La clase **insn_t**, definida en **ua.hpp**, es utilizada para describir una sola instrucción de la base de datos. Dicha declaración se muestra a continuación:

```
class insn_t {

 public:

  ea_t cs; // Actual segmento base. Habilitado por el kernel.

  ea_t ip; // Direccion virtual de instruccion (dentro del segmento). Habilitado por kernel

  ea_t ea; // Direccion lineal de una instruccion. Habilitada por kernel

  ushort itype; // valor numerico de instruccion (no opcode). El proc habilita estos en ana

  ushort size;  // Tamano de instrucción en byte. El proc habilita estos en ana

  union {       // campo dependiente del procesador. El proc puede habilitar esta

    ushort auxpref;

    struct {

      uchar low;

      uchar high;

    } auxpref_chars;

  };

  char segpref;     // Campo dependiente del procesador.  El proc puede habilitarla

  char insnpref;    // Campo dependiente del procesdor.  El proc puede habilitarla

  op_t Operands[6]; // instrucion informativa de operando.  El proc lo habilita en ana

  char flags;       // instrucción de flags.  El proc puede habilitarla

 };
```

Antes de llamar a nuestra función analizador, el kernel de IDA (corazón de IDA) rellena los tres primeros campos del objeto **cmd** con la dirección segmentada y linear de la instrucción. Una vez rellenas estas, rellenar el resto es tarea del analizador. Los campos esenciales a rellenar por el

analizador son **itype**, **size** y **Operands**, los cuales podemos encontrar en el procedimiento anterior. El campo **itype** debe habilitarse a **1** como numeración de instrucción, como habíamos comentado anteriromente. El campo **size** debe habilitarse al tamaño total de la instrucción y deberá utilizarse como el valor retornado de la instrucción. Si la instrucción no puede analizarse, el analizador deberá retornar un tamaño **cero**. Finalmente, una instrucción puede tener como máximo hasta seis operandos, y el analizador deberá rellenar la información respecto a cada operando utilizado por la instrucción.

La función analizador a menudo es ejecutada utilizando una declaración de conmutación. Normalmente el primer paso del analizador es buscar uno o más byte (dependiendo del procesador) de la instrucción y utilizar estos byte como una variable de verificación de conmutación. El SDK nos proporciona funciones especiales para utilizar en el analizador, con el propósito de obtener los byte de la cadena de instrucción.

Estas funciones son las mostradas a continuación:

```
//lee un byte desde ubicacion actual de instruccion

uchar ua_next_byte(void);

//lee dos byte desde la ubicación actual de instruccion

ushort ua_next_word(void);

//lee cuatro byte desde la ubicación actual de instruccion

ulong ua_next_long(void);

//lee ocho byte desde la ubicación actual de instruccion

ulonglong ua_next_qword(void);
```

La **ubicación actual de instrucción** es inicialmente el mismo valor contenido en **cmd.ip**. Cada llamada a una de las funcioes **ua_next_xxx** tiene el efecto secundario de incrementar **cmd.size** de acuerdo al número de byte requeridos por la función ua_next_xxx que será llamada (1, 2, 4 o 8). Los byte obtenidos se tendrán que descodificar para asignar el valor numerado apropiado a la instrucción dentro del campo **itype,** determinar el número y tipo de cualquier operando requerido por la instrucción y determinar la longitud total de la instrucción. Mientras se va ejecutando el proceso de descodificación, pueden ser requeridos byte adicionales hasta el punto en que se haya obtenido toda la cadena completa de la instrucción actual. Mientras se esté utilizando la función **ua_next_xxx**, se nos irá actualizando automáticamente **cmd.size**, eliminando de esta forma la necesidad de recordar cuántos byte se han requerido para la instrucción actual. Desde un punto de vista de lenguaje de nivel alto, el analizador tiene una cierta similitud con la fase de descodificación de la CPU, ya que esta para descodificar las instrucciones las va tomando instrucción a instrucción. Trasladando el proceso a la realidad, la descodificación de instrucciones tiende a ser fácil para los procesadores con instrucciones de tamaño fijo, como puede ser con las arquitecturas estilo **RISC**, mientras que la descodificación se complica en procesadores cuyas instrucciones sean de longitud variable, como ocurre en el **x86**.

Utilizando los byte obtenidos, el analizador deberá inicializar un elemento en el conjunto **cmd.Operands** para cada operando utilizado por la instrucción. Los operandos de las instrucciones están representados por procesos de la clase **op_t**, los cuales están definidos en **ua.hpp**, veamos unos ejemplos:

```
class op_t {

public:
```

```
  char n;  //numero operandos (0,1,2).Kernel los habilita no cambia
  optype_t type; //tipo operando.Habilitado en ana.Valores en ua.hpp

  // offset de operando relativo a la primera instruccion
  char offb;  //Proc lo habilita en ana,habilitado a 0 si desconocido
  // offset a segunda parte operando (si existe) relativo a la  primera
instruccion

  char offo;  //Proc lo habilita en ana,habilitado a 0 si desconocido
  uchar flags; //Proc lo habilita en ana.Para valores ver ua.hpp

  char dtyp; //Especifica tipo dato operando.Habilitado en ana. Para valores ver
ua.hpp

  //La siguiente struc unions guarda otra informacion del operando
  union {
    ushort reg;    // numero de registro para tipo o_reg
    ushort phrase; // numero de registro phrase para tipos o_phrase  y o_displ
                   // definir numero de phrases como nos apetezca
  };

  union {          // valor del oprerando para el tipo o_imm o
    uval_t value;  // outer displacement (o_displ+OF_OUTER_DISP)
    struct {       // Acceso utilizado a la mitad de los valores
        ushort low;
        ushort high;
    } value_shorts;
  };
  union {   //direccion virtual usada o apuntada por el operando
    ea_t addr;  //para tipos (o_mem,o_displ,o_far,o_near)
    struct {    // acceso utilizado para la mitad de addr
        ushort low;
        ushort high;
    } addr_shorts;
  };
```

```
  //campos dependientes del procesador,usar como queramos.Habilitados
      //en ana
  union {
    ea_t specval;
    struct {
        ushort low;
        ushort high;
    } specval_shorts;
  };
  char specflag1, specflag2, specflag3, specflag4;
};
```

La configuración de un operando se inicia con la habilitación del campo **type** en una de las constantes numeradas **optype_t** definidas en **ua.hpp**. El **type** de un operando, describe la fuente y el destino del dato del operando. En otras palabras, el campo **type** describe el modo de direccionamiento empleado por el operando. Los tipos de aperando incluyen a los siguientes **o_reg**, significa que el operando es el contenido de un registro, **o_mem**, siginifica que el operando es una dirección de memoria conocida en el momento de compilación y **o_imm**, significa que el operando son los datos inmediatos.

El campo **dtype** especifica el tamaño del dato del operando. Este campo debería habilitar a **1** los valores **dt_xxx** especificados en **ua.hpp**.

Las siguientes instrucciones x86 muestran la correspondencia de algunos de los tipos dato principales utilizados como operandos:

```
 mov  eax, 0x31337          ; o_reg(dt_dword), o_imm(dt_dword)
push word ptr [ebp - 12]   ; o_displ(dt_word)
mov [0x08049130], bl       ; o_mem(dt_byte), o_reg(dt_byte)
movzx eax, ax              ; o_reg(dt_dword), o_reg(dt_word)
ret                        ; o_void(dt_void)
```

La forma en que son utilizadas las distintas **struct unions** dentro de un **op_t,** está dictada por el valor del campo **type**. Por ejemplo, cuando un operando tipo es **o_imm,** el valor del dato inmediato deberá ser almacenado en el campo **value**, y cuando el operando tipo es **o_reg**, el número de registro (del conjunto numerado de constantes registro) deberá ser guardado en el campo **reg**. Los detalles completos de en dónde hay que almacenar cada parte de un instrucción se puede encontrar en **ua.hpp**.

Observemos que ninguno de los campos dentro de un **op_t** describe si el operando se está utilizando como fuente o como destino del dato. De hecho, no es tarea del analizador determinar tales cosas. Las banderas reglamentarias especificadas en el conjunto de nombres de instrucción son utilizadas, en una fase posterior por el procesador, para determinar exactamente cómo se está utilizando un operando.

Algunos de los campos dentro de las clases **insn_t** y **op_t** se describen como dependientes del procesador, lo cual significa que podemos utilizar dichos campos para cualquier propósito que deseemos. Dichos campos a menudo son utilizados para almacenar la información que no se ajusta limpiamente en otros campos dentro de estas clases. Los campos dependientes del procesador son también un mecanismo conveniente para información a lo largo de fases posteriores del procesador de modo que dichas fases no necesitan duplicar el trabajo del analizador.

Una vez que ya sabemos lo que realiza el analizador, podemos crear un mínimo analizador para el byte code Python, debido a que es muy directo. Los opcodes Python tienen un byte de longitud. Los opcodes con numeración menor de 90 no tienen operandos, mientras que los opcodes mayor o igual a 90 tienen operando de dos byte. Nuestro analizador básico sería como se muestra a continuación:

```
#define HAVE_ARGUMENT 90

int idaapi py_ana(void) {

   cmd.itype = ua_next_byte();//opcodes ARE itypes(actualiza cmd.size)

   if (cmd.itype >= PYTHON_LAST) return 0; //instruccion invalida

   if (Instructions[cmd.itype].name == NULL) return 0; //ins.invalida

   if (cmd.itype < HAVE_ARGUMENT) { //no hay operandos

      cmd.Op1.type = o_void;//Op1 macro de Operand[0] (ver ua.hpp)

      cmd.Op1.dtyp = dt_void;

   }

   else {   //instrucion debe tener un dato operando de dos byte

      if (flags[cmd.itype] & (HAS_JREL | HAS_JABS)) {

         cmd.Op1.type = o_near;//operando referido a ubicación codigo

      }

      else {

         cmd.Op1.type = o_mem;//operando referido a memoria(como sea)

      }

      cmd.Op1.offb = 1;//operando offset instrucción es de 1 byte

      cmd.Op1.dtyp = dt_dword;//En Python sin tamaño, cogemos algo


      cmd.Op1.value = ua_next_word(); //toma un operando word (actualiza
cmd.size)

      cmd.auxpref = flags[cmd.itype]; //almacena flags para futuras fases


      if (flags[cmd.itype] & HAS_JREL) {

         //calcula el salto relativo del objetivo

         cmd.Op1.addr = cmd.ea + cmd.size + cmd.Op1.value;

      }
```

```
      else if (flags[cmd.itype] & HAS_JABS) {

         cmd.Op1.addr = cmd.Op1.value;//guarda direcciones absolutas

      }

      else if (flags[cmd.itype] & HAS_CALL) {

         //En Python el objetivo,operando indicado, de la llamada esta en el
stack

         //como algunos argumentos están en el stack, almacenamos estos para
posteriores fases

         cmd.Op1.specflag1 = cmd.Op1.value & 0xFF;   //parametros de posición

         cmd.Op1.specflag2 = (cmd.Op1.value >> 8) & 0xFF; //parametros de
teclado

      }

   }

   return cmd.size;

}
```

Para el módulo procesador Python hemos elegido crear un conjunto de banderas adicional, uno por instrucción, para suplir (en casos de duplicación) las características determinadas para cada instrucción. Las banderas definidas para utilizar en nuestro conjunto **flags** son **HAS_JREL, HAS_JABS** y **HAS_CALL**. Estas banderas se utilizarán respectivamente para indicar si un operando de una instrucción representa un offset relativo a un salto, un salto absoluto a un objetivo o la descripción de una función call stack. Explicar cada detalle de la fase de análisis es dificil sin introducirnos en las operaciones del interprete de Python, por lo tanto llegaremos hasta aquí con el analizador y recordaremos qué tareas realiza el analizador para diseccionar una instrucción:

1.-- El analizador toma el siguiente byte de instrucción dentro de la cadena de la instrucción y determina si el byte es un opcode válido de Python.

2.-- Si la instrucción no tiene operandos, **cmd.Operand [0]** (cmd.Op1) es inicializado como **o_void**.

3.-- Si la orden tiene un operando, **cmd.Operand [0]** es inicializado para que refleje el tipo del operando. Algunos campos específicos al procesador se utilizan para almacenar dicha información para futuras fases del módulo procesador.

4.-- La longitud de la instrucción es retornada al llamador.

Los conjuntos de instrucciones más sofisticadas, requerirán fases de análisis más complejas, Sin embargo, en conjunto, el comportamiento del analizador puede generalizarse de la siguiente forma:

1.-- Lee bastantes byte de la cadena de instrucción para determinar si la instrucción es válida y comprobar dicha instrucción con una de las constantes tipo instrucciones numeradas, luego es almacenada en **cmd. itype**. Esta operación es a menudo ejecutada utilizando una larga declaración de conmutación para categorizar los opcodes de instrucción.

2.-- Lee todos los byte adicionales necesarios para determinar correctamente el número de operandos requeridos por la instrucción, los modos de direccionamiento utilizados por esos operandos y los componentes de cada operando (registros y datos inmediatos). Estos datos se utilizan para rellenar elementos del conjunto **cmd.Operands**. Esta operación se puede descomponer

en distintas funciones de descodificación de los operandos.

3.-- Retorna la longitud total de la instrucción y de los operandos.

En el sentido estricto de la palabra, una vez que una instrucción ha sido diseccionada, IDA tiene la suficiente información para generar una representación en lenguaje ensamblador de dicha instrucción. A fin de generar las referencias cruzadas, facilitar el proceso recursivo descendente y controlar el comportamiento del puntero de pila del programa, IDA deberá obtener detalles adicionales con respecto al comportamiento de cada instrucción. Está será la tarea de la fase de emulación de un módulo de procesador IDA la cual veremos en el próximo escrito.

Performance Bigundill@