18.0 – Modulos de procesador de IDA

El último tipo de módulos de Ida que pueden construirse con el SDK son los módulos de procesador, los cuales son los más complejos de los estudiados.

Dichos módulos son los responsables de todas las operaciones de desensamblado que tienen lugar en IDA. Mas allá de la simple conversión de los opcodes del lenguaje máquina en sus equivalentes de lenguaje ensamblador, los módulos de procesador son también los responsables de tareas como creación de funciones, generación de referencias cruzadas y del seguimiento del puntero de pila.

Es obvio que la programación de un módulo de procesador se realizará para poder ejecutar ingeniería inversa sobre un binario del cual no exista ningún módulo de procesador.Entre otras cosas, dicho binário puede ser la imagen de un firmware adjuntado a un microcontrolador o imágenes ejecutables de algún dispositivo. Otra utilización, en menor medida, podría ser desensamblar las instrucciones de una máquina virtual adjuntada dentro de una ejecutable ofuscado. En tales casos un módulo de procesador, como el pc x86, sólo nos podría ayudar a comprender la propia máquina virtual, pero no nos proporcionaría ninguna ayuda para desensamblar el byte code de la máquina virtual.

En un paper posteado en OpenRCE.org, Rolf Rolles nos muestra la aplicación de un módulo de procesador. En el apéndice B de dicho paper, Rolf comparte la creación de un módulo de procesador, uno de los pocos documentos disponibles con respecto a este tema. Referirse a "**Defeating HyperUnpackMe2 With an IDA Processor Module**".

En estos escritos siguientes, intentaremos desmitificar un poco el tema de módulos de procesador. Esto lo realizaremos programando un módulo de procesador el cual desensamblará byte code Python (Realizado por Chris Eagle). Es necesario comprender que sin un módulo de carga Python, será imposible ejecutar un desensamblado automatizado de un archivo compilado **.pyc**. Faltándonos dicho cargador tendremos que realizar la carga del archivo como binario y seleccionar el módulo de procesador Python, identificar el punto de inicio de una función y convertir los byte mostrados a código Python con la acción **Edit > Code**.

18.1.-- Byte code Python

Python es un lenguaje de programación interpretado orientado al objeto, a menudo usado para realizar scripts, de forma similar a Perl. Los archivos fuente Python normalmente son guardados con extensión **py**. Siempre que es ejecutado un script Python, el intérprete de Python compila el código fuente a una representación interna llamada byte code Python,y finalmente dicho byte code es interpretado por una máquina virtual. Este proceso es análogo a la manera en que el código fuente de Java es compilado a byte code de Java, y que finalmente es ejecutado por una máquina virtual Java. La diferencia principal es que los usuarios de Java deben compilar explícitamente su fuente Java en byte code Java, mientras que el código fuente de Python se convierte implícitamente a byte code Python cada vez que un usuario elige ejecutar un script Python. Referirse a <http://www.python.org/>

Para poder acceder a una lista completa de instrucciones byte code Python y ver sus significados referirse a <http://docs.python.org/lib/bytecodes.html> . También mirar el **opcode.h** en la distribución de Python para poder ver los mnemóticos del byte code y sus opcodes equivalentes.

Una vez compilado, y para evitar la acción repetida de interpretar byte code. El interprete Python

guarda la representación del byte code de un archivo fuente Python en un archivo **.pyc** para cargarlo directamente y realizar su subsecuente ejecución. Como los usuarios no crean explícitamente el archivo .pyc, el intrprete de Python de forma automática lo crea para cada módulo importado por otro módulo fuente Python. Por similitud los archivos **pyc** de Python son como los archivos **class** de Java.

Dado que el interprete de Python no requiere necesariamente el código fuente, es posible distribuir partes de un proyecto Python en byte code en vez de código fuente. En dichos casos, es muy útil ejecutar ingeniería inversa sobre el byte code para poder averiguar qué realiza dicho código, lo mismo que se realiza con una distribución binaria. Esto último será el propósito de nuestro módulo de procesador Python, poder aplicar ingeniería inversa sobre el byte code Python.

18.2.-- El intérprete Python

Conocer un poco el intérprete de Python nos será útil para programar el módulo procesador Python. El intérprete Python ejecuta una máquina virtual con **stack** capaz de ejecutar byte code Python. Una máquina virtual basada en stack significa que dicha máquina no tiene registros a parte del puntero a instrucción y del puntero a pila. La mayor parte de las instrucciones byte code Python manipulan de alguna forma la pila, leyendo, escribiendo o examinando el contenido de dicho stack. Por ejemplo la instrucción **BINARY_ADD**, quita dos elementos de la pila del intérprete, los junta y coloca el valor de su resultado en la parte superior de la pila del interprete.

Desde un punto de vista de esquema de instrucciones, el byte code es relativamente simple de comprender. Todas las instrucciones consisten en operandos de un byte, un cero o dos byte. El ejemplo que mostramos en este escrito sobre el módulo de procesador, no incluye necesariamente el conocimiento previo del byte code Python. En los casos en donde sea necesario su conocimiento, nos tomaremos la molestia de explicarlo suficientemente para su comprensión. La meta principal de este escrito es proporcionar una comprensión básica sobre los módulos de procesador de IDA y algunas consideraciones para crearlos. El byte code Python no es más que una herramienta utilizada para facilitarnos nuestra meta.

18.3.-- Programar un módulo procesador

Como ya hemos apuntado anteriormente las explicaciones sobre la programaciòn de módulos de procesador IDA son escasas. Aparte de leer los archivos include del SDK, los archivos fuente de los módulos procesador en el SDK y el readme.txt del SDK son los únicos lugares en dónde se pueden encontrar detalles sobre los módulos de procesador.

Aclaremos también que las referencias específicas sobre los nombres de archivos del módulo de procesador definidas en el README, de hecho no lo son. Estos, sin embargo, son los nombres utilizados en los ejemplos incluidos en el SDK, y son también los nombres con referencia en los scripts incluidos en dichos ejemplos. Por lo tanto podemos utilizar los nombres de archivo que queramos en la construcción de nuestro módulo de procesador, siempre y cuando estén en concordancia con los de los scripts.

Para referirnos a los archivos específicos del procesador y trasladarlas al módulo de procesador, lo haremos con tres componentes lógicos: un analizador, un emulador de instrucción y un generador de salida. Durante la creación de nuestro módulo procesador Python trataremos con cada una de dichas partes.

Podemos encontrar distintos ejemplos de módulos de procesador en **<SDKDIR>\module**. Uno de

los procesadores mas simples para leer de “cabo a rabo” es el procesador z8. Si lo que queremos es programar nuestro propio modulo procesador, una forma de familiarizarse, recomendado por Ilfak en el archivo README, es copiar un modulo procesador ya existente y modificarlo según nuestras necesidades.

18.3.1.-- La estructura processor_t

Al igual que en los módulos plugin y de carga, los de procesador exportan también una sola cosa. Para los de procesador es la estructura **processor_t** llamada **LPH**. Esta estructura es exportada automáticamente si incluimos **<SDKDIR>\module\idaidp.hpp**, que incluye a su vez muchos otros archivos header SDK requeridos normalmente por los módulos de procesador. Una de las razones por las que programar un módulo procesador sea complicado es que dicha estructura contiene 56 campos distintos los cuales hay que inicializar, asímismo 26 de dichos campos son punteros a funciones y uno de los campos es un puntero a un conjunto de una o más estructuras de punteros los cuales cada uno apunta a un distinto tipo de estructura (**asm_t**) la cual contiene 59 campos más que requieren su inicialización. ¿Facilito verdad? Una de las principales pegas en programar un módulo procesador gira alrededor de la inicialización de todos estos datos estáticos, lo cual nos puede llevar a errores debido al gran número de campos dentro de cada estructura. Esta es una de la razones por las que Ilfak recomienda utilizar un procesador ya creado, para basarse en él en la construcción de una nuevo.

Debido a la complejidad de dichas estructuras de datos, no enumeraremos todos los campos posibles y sus usos. En vez de esto, explicaremos los campos principales y para más detalles tendremos que referirnos al **idp.hpp**. El órden en que relacionaremos dichos campos de **processor_t** no tiene ninguna relación con el órden que estos son declarados dentro de la estructura **processor_t**.

18.3.2.-- Inicialización básica de la estructura LPH

Antes de adentrarnos sobre los aspectos del comportamiento de nuestro módulo procesador, existen distintos requisitos respecto a datos estáticos que debemos tener en cuenta. Debido a que vamos a construir un módulo de desensamblado, necesitaremos crear una lista con cada mnemotecnio de lenguaje ensamblador que utilicemos en nuestro módulo procesador. Esta lista se crea con la forma de un conjunto de estructuras **instruct_t** (definido en **idp.hpp**) y colocarla normalmente en un archivo con nombre **ins.cpp**. Como mostraremos a continuación, **instruct_t** es una estructura la cual tiene dos propósitos. En primer lugar, proporcionar una tabla de búsqueda de las instrucciones mnemonicas y en segundo lugar, describir ciertas características básicas de cada instrucción.

```
struct instruc_t {

  const char *name;  //instruccion mnemonico

  ulong feature;     //bitwise OR de la bandera CF_xxx flags definido
                en idp.hpp

};
```

El campo **feature** se utiliza para indicar comportamientos tales como si la instrucción lee o escribe alguno de sus operandos y cómo la ejecución contiua una vez se ha ejecutado la instrucción (salto, llamada). La bandera **CF** en **CF_xxx** representa la característica dispuesta. El campo **feature** básicamente maneja los conceptos de control de flujo y referencias cruzadas. Algunas de las banderas de disposición de características más interesantes son las siguientes:

CF_STOP

La instrucción no pasa el control a la instrucción siguiente. Como ejemplos podríamos incluir los saltos absolutos e instrucciones de retorno de función.

CF_CHGn

La instrucción modifica el operando **n,** donde n está en el rango de 1 a 6.

CF_USEn

La instrucción utiliza el operando **n,** donde n está en el rango de 1 a 6, y lo utiliza con el significado "**reads**" o "**refers to**"(pero no lo modifica, ver CF_CHGn) a una ubicación en memoria.

CF_CALL

La instrucción llama a una función

No es preciso listar las instrucciones en un orden determinado. De hecho, no existe ninguna necesidad de ordenar las instrucciones de acuerdo a sus opcodes binarios asociados. Las primeras y últimas líneas del conjunto de instrucciones de nuestro ejemplo se muestran a continuación:

```
instruc_t Instructions[] = {
    {"STOP_CODE", CF_STOP},   /* 0 */
    {"POP_TOP", 0},           /* 1 */
    {"ROT_TWO", 0},           /* 2 */
    {"ROT_THREE", 0},         /* 3 */
    {"DUP_TOP", 0},           /* 4 */
    {"ROT_FOUR", 0},          /* 5 */
    {NULL, 0},                /* 6 */
    ...
    {"CALL_FUNCTION_VAR_KW", CF_CALL}, /* 142 */
    {"EXTENDED_ARG", 0}               /* 143 */
 };
```

En nuestro ejemplo, debido a la sencillez del byte code Python, mantendremos una correspondencia de uno a uno entre las instrucciones y el byte code. Observemos que ciertas instrucciones deben de actuar como relleno cuando un opcode no está definido, como es el caso del **opcode 6**.

Un conjunto enumerado de constantes asociadas se define en **ins.hpp** proporcionando una equivalencia entre enteros y instrucciones correspondientes como se muestra a continuación:

```
enum python_opcodes {
   STOP_CODE = 0,
   POP_TOP = 1,    //quita elemento superior del stack
   ROT_TWO = 2,    //intercambia los dos elementos superiores de stack
   ROT_THREE = 3,  //mueve el elemento superior debajo de los
```

```
            elementos 2ndo y 3tro

   DUP_TOP = 4,    //duplica el elemento superior de la pila

   ROT_FOUR = 5,   //mueve el elemento superior de la pila por debajo
            del 2ndo, 3tro y 4to elementos

   NOP = 9,        //no realiza nada



   ...

   CALL_FUNCTION_VAR_KW = 142,

   EXTENDED_ARG = 143,

   PYTHON_LAST = 144
};
```

En este caso hemos elegido asignar explícitamente un valor a cada numerador, primero para clarificar y segundo debido a que hemos elegido utilizar los opcodes reales de Python como nuestros índices de instrucción. También se ha añadido al final de la lista una constante (**PYTHON_LAST**) para proporcionar una referencia fácil. Con una lista de instrucciones y sus correspondientes equivalencias en números enteros, tenemos lo necesario para inicializar tres campos de **LHP** (nuestra estructura global processor_t). Estos tres campos son los siguientes:

```
int instruc_start;   // entero codigo primera instruccion

int instruc_end;     // entero codigo ultima instruccion + 1

instruc_t *instruc;  // Conjunto (array) de instrucciones
```

Debemos de inicializar estos campos con **STOP_CODE, PYTHON_LAST** y **Instructions**, respectivamente. Juntos, estos campos habilitan a un módulo procesador para que busque rápidamente en el desensamblado los mnemónicos de cualquier instrucción.

Para la mayoría de módulos procesador, también necesitaremos definir un conjunto de nombres de registros y un conjunto de constantes numeradas asociadas a ellos para poder referirnos a dichos registros. Si estuviéramos programando un módulo procesador para x86, podríamos iniciarlo con algo parecido a lo siguiente, para no alargarlo nos centraremos en los registros básicos:

```
static char *RegNames[] = {

   "eax", "ebx", "ecx", "edx", "edi", "esi", "ebp", "esp",

   "ax", "bx", "cx", "dx", "di", "si", "bp", "sp",

   "al", "ah", "bl", "bh",  "cl", "ch", "dl", "dh",

   "cs", "ds", "es", "fs", "gs"
};
```

El conjunto **RegNames** es a menudo declarado en un archivo nombrado **reg.cpp**. Este archivo es también en donde se declara, en nuestro ejemplo, **LHP** lo cual habilita a **RegNames** para declararlo en modo estático. La numeración asociada a los registros podría ser declarada en un archivo header, normalmente llamado como el procesador (en este caso **x86.hpp**), de la siguiente forma:

```
enum x86_regs {
```

```
   r_eax, r_ebx, r_ecx, r_edx, r_edi, r_esi, r_ebp, r_esp,
   r_ax, r_bx, r_cx, r_dx, r_di, r_si, r_bp, r_sp,
   r_al, r_ah, r_bl, r_bh,  r_cl, r_ch, r_dl, r_dh,
   r_cs, r_ds, r_es, r_fs, r_gs
};
```

Tenemos que asegurarnos que entre el conjunto de nombres de registro y el conjunto de constantes existe la apropiada correspondencia entre ellos, ya que ambos permitirán al modulo procesador buscar rápidamente los nombres de registro cuando se formateen los operando de una instrucción. Estas dos declaraciones de datos se utilizan para inicializar otros campos en LPH:

```
int   regsNum;            // numero total de registros
char  **regNames;         // array de nombres de registro
```

Estos dos campos son a menudo inicializados como **qnumber (RegNames)** y **RegNames**, respectivamente, donde **qnumber** es una macro definida en **pro.h** la cual calcula el número de elementos en un conjunto distribuido en modo estático.

Siempre se requiere en un módulo procesador IDA la especificación de la información sobre los registros de segmento aunque el procesador no los utilice. Como el x86 utiliza los registros de segmento, este ejemplo es fácil de configurar. Dentro de **processor_t** los registros de segmento son configurados en los siguientes campos:

```
 // informacion registro segmento (utiliza registros virtuales CS y DS  //si el
procesador no tiene registros de segmento):
   int   regFirstSreg;        // numero del primer registro segmento
   int   regLastSreg;         // numero del ultimo registro segmento
   int   segreg_size;         // tamano en byte de un registro seg.

 // Si el procesador no utilza registros de segmento. Deberemos  //definir 2
registros virtuales de segmento para CS y DS.
 // los llamaremos como rVcs y rVds.
   int   regCodeSreg;         // numero del registro CS
   int   regDataSreg;         // numero del registro DS
```

Para inicializar nuestro hipotético módulo procesador x86, los cinco campos previos pueden inicializarse de la siguiente forma:

```
r_cs, r_gs, 2, r_cs, r_ds
```

Repasemos las explicaciones y observemos los registros de segmento. IDA requiere siempre la información sobre segmentos aunque el procesador no los utilice. Volviendo a nuestro ejemplo Python, no tendremos mucha dificultad en preparar un mapeado de registros debido a que el interprete Python tiene una arquitectura basada en la pila y no existen registros, pero sí necesitaremos solucionar el asunto de los registros de segmento. La forma de realizarlo será

componer los nombres y numerarlos con valores para que representen un conjunto mínimo de registros de segmento (código y dato). Básicamente estamos falsificando los registros de segmento para que IDA lo acepte y no nos ponga pegas. De esa forma aunque IDA los pida, nosotros simplemente los ignoraremos en nuestro módulo procesador. Para nuestro procesador Python, haremos lo siguiente:

```
//en reg.cpp

static char *RegNames = { "cs", "ds" };


//en python.hpp

enum py_registers { rVcs, rVds };
```

Con estas declaraciones en su lugar apropiado, podemos retornar para inicializar los campos apropiados dentro de LPH utilizando la siguiente secuencia de valores:

```
rVcs, rVds, 0, rVcs, rVds
```

Antes de adentrarnos en la ejecución de cada comportamiento en el módulo procesador Python, nos tomaremos el tiempo necesario para el resto de inicialización concerniente a LPH. Los primeros cinco campos de **processor_t** se describen a continuación:

```
int version; // debera ser IDP_INTERFACE_VERSION

int id;     // IDP id, un valor PLFM_xxx o auto asignado > 0x8000

ulong flag; // Caracteristicas processor, bitwise OR de valores PR_xxx

int cnbits; // Numero de bits en un byte de segmentos code (normal 8)

int dnbits; // Numero de bits en un byte de segmentos data (normal 8)
```

El campo **version** nos tiene que ser familiar ya que también se exige en los módulos plugin y cargador. Para la mayoría de modulos procesador, el campo **id** deberá ser un valor auto asignado mayor que **0x8000**. El campo **flag** describe distintas características del módulo procesador según la combinación con las banderas **PR_xxx** definidas en **idp.hpp**. Para el procesador Python, optaremos especificar sólo **PR_RNAMESOK**, lo que permite a los nombres de registro utilizarlos como ubicaciones nombradas (lo cual es perfecto ya que no disponemos de ningún registro) y **PRN_DEC**, lo cual habilita por defecto mostrar el número en formato decimal. Los dos campos restantes **cnbits** y **dnbits** cada uno se habilita a **8.**

Performance Bigundill@