16.0—Arquitectura de IDA Plug-in

En el curso de los siguientes escritos, trataremos los tipos de módulos que se pueden construir utilizando el IDA SDK. Aunque nunca hayamos pensado en crear nuestro propio plug-in para ida, la comprensión básica de ellos mejorará en mucho nuestra experiencia en el uso de él, la mayoría de software programado para utilizarlo con ida se distribuye en forma de plugins. En este escrito, iniciaremos la exploración de módulos ida hablando de los plugins, de cómo construirlos, instalarlos y configurarlos.

Los plugins normalmente están asociados con atajos de teclado o un elemento de menú y sólo son accesibles después de haber abierto una base de datos. Individualmente pueden ser de uso general o utilizados para distintos tipos de archivo binario y arquitecturas de procesador, pueden ser muy especializados, diseñados para utilizarlo sólo con un tipo de archivo o procesador específico. En todos los casos su virtud es estar compilado como módulo, los plugins tienen acceso completo a la IDA API y generalmente pueden ejecutar tareas mucho más complejas que las realizadas con scripts.

16.1.— Programar un plug-in

Todos los módulos de ida, incluido plug-in, son ejecutados como componentes de librería compartida apropiada a la plataforma en la cual dicho plug-in tiene que ejecutarse. Según la arquitectura modular de ida, los módulos no requieren exportar ninguna función. En vez de eso, cada tipo de módulo deberá exportar una variable de una clase específica. En el caso de plug-in, esta clase es llamada **plugin_t** y está definida en el archivo **loader.hpp** del SDK.

Para poder entender cómo crear un plug-in, primero deberemos entender la clase **plugin_t** y sus componentes de campos dato (la clase no tiene elementos función). El esquema de la clase **plugin_t** la vemos seguidamente juntamente con los comentarios tomados de **loader.hpp**:

```
class plugin_t {
public:
  int version;        // Deberá ser igual a IDP_INTERFACE_VERSION
  int flags;          // Características del plugin
  int (idaapi* init)(void); // Inicializa el plugin
  void (idaapi* term)(void);   // Finaliza el plugin. Esta función será llamada
                       // cuando el plugin sea descargado. Quizá sea NULL.
  void (idaapi* run)(int arg); // Invoca al plugin
  char *comment;              // Comentario acerca del plugin
  char *help;         // Multilineas de ayuda respecto al plugin
  char *wanted_name;    // El nombre adjudicado al plugin
  char *wanted_hotkey;  // El atajo de teclado adjudicado para ejecutar el plugin
};
```

Cada plug-in deberá exportar un objeto **plugin_t** llamado **PLUGIN**. La exportación del objeto **PLUGIN** es manejada por **loader.hpp**, el cual es el responsable de declarar e inicializar el objeto actual. Una vez que hayamos creado con éxito nuestro plugin dependerá de **loader.hpp** que se inicialice correctamente dicho objeto, seguidamente describimos el propósito de cada elemento:

version
Este elemento indica el número de versión de la API la cual se ha utilizado para construir el plug-in. Normalmente habilitado por la constante **IDP_INTERFACE_VERSION**, declarada en **idp.hpp**. El valor de dicha constante no ha cambiado desde que la API se normalizó con el SDK versión 4.9. Originalmente con este campo se impedía que los plug-in creados con versiones anteriores del SDK pudieran ser cargados en versiones posteriores de ida.

flags
Este campo contiene distintas banderas indicando a ida cómo deberá manejar el plugin en distintas situaciones. El **flags** se habilitará utilizando una combinación de bits de las constantes **PLUGIN_XXX** definidas en **loader.hpp**. Para algunos plugin, asignando cero a este campo será suficiente. Para saber el significado de cada bit bandera, tendremos que referirnos a **loader.hpp**.

init
Este es el primer puntero a una función de los tres que contiene la clase **plugin_t**. Este particular elemento es un puntero a la función de inicialización del plug-in. La función no toma parámetros y retorna un **int**. Ida llama a esta función para ofrecer la posibilidad de carga de nuestro plugin. La inicialización de plug-in la estudiaremos en la sección 16.1.2.

term
Este elemento es otro puntero a una función. Ida llama a la función asociada cuando nuestro plug-in es descargado. La función no toma argumentos y no retorna ningún valor. El propósito de esta función es ejecutar cualquier tarea de limpieza (quitar memoria, cerrar handles, guardar estados y otros) requerida por nuestro plug-in antes de que ida lo descargue. Este campo se puede habilitar como **NULL** si no hay que ejecutar ninguna acción cuando nuestro plug-in es descargado.

run
Este elemento puntero a una función será llamado siempre que un usuario active (por atajo de teclado, elemento de menú o invocación de IDC) nuestro plug-in. Esta función es el corazón de cualquier plug-in, es aquí donde el usuario inicia el comportamiento que tiene que ejecutar el plug-in. Al comparar scripts con plug-in, esta es la función más parecida entre ambos. La función recibe un parámetro entero (hablaremos de él en la sección 16.1.4) y no retorna nada.

comment
Este elemento es un puntero a una cadena de caracteres la cual sirve como comentario del plug-in. No es utilizado directamente por IDA y puede habilitarse como **NULL**.

help
Este elemento es un puntero a una cadena de caracteres la cual habilita multilíneas de ayuda. No es utilizada directamente por ida y puede habilitarse como **NULL**.

wanted_name
Este elemento es un puntero a una cadena de caracteres la cual alberga el nombre del plug-in. Cuando se carga un plug-in, esta cadena es añadida al menú **Edit> Plugins**

como medio para ejecutarlo. No es necesario que el nombre sea único, pero si no es así será más difícil identificarlo en la lista del menú.

wanted_hotkey
Este elemento es un puntero a la cadena de caracteres que alberga el nombre del atajo de teclado (como "Alt-F8") con el cual ida tendrá que asociar a nuestro plugin. Aquí también no es necesario que sea único para el plugin; sin embargo; si el valor no es único, el atajo de teclado será asociado con el último plugin utilizado. En la sección 16.4 hablaremos de la forma en que los usuarios pueden sobrescribir el valor de **wanted_hotkey**.

Un ejemplo de inicialización de un objeto **plugin_t** lo vemos seguidamente:

```
int idaapi tora_plugin_init(void);
void idaapi tora_plugin_term(void);
void idaapi tora_plugin_run(int arg);

char tora_comment[] = "Esto es un ejemplo de un plugin";
char tora_name[] = "Tora";
char tora_hotkey = "Alt-F9";

plugin_t PLUGIN {
  IDP_INTERFACE_VERSION, 0, tora_plugin_init,  tora_plugin_term,
   tora_plugin_run, tora_comment, NULL, tora_name, tora_hotkey
};
```

Los punteros de función en la clase **plugin_t** permiten a IDA localizar las funciones requeridas en el plugin sin el requisito de exportar estas funciones o escoger nombres específicos para ellas.

16.1.1.— El ciclo vital del plug-in

Una sesión de IDA se inicia con la ejecución de la misma aplicación, la cual empieza cargando y autoanalizando un nuevo archivo binario o una base de datos existente antes de pararse y esperar una acción del usuario. Durante este proceso, existen tres puntos distintos en los que IDA ofrece la posibilidad de cargar un plugin:

1.- Un plugin puede cargarse en el arranque de IDA aunque se esté o no cargando una base de datos. Cargándolo de esta forma está controlado con la presencia del bit de **PLUGIN_FIX** habilitado en **PLUGIN.flags**.

2.- Un plugin puede cargarse inmediatamente después de la carga del módulo de procesador y seguirá cargado hasta que el módulo de procesador sea descargado. La relación de un plugin con un módulo de procesador está controlada por el bit de PLUGIN_PROC habilitado en **PLUGIN.flags**.

3.- En ausencia del bit bandera, mencionados anteriormente, IDA ofrece la oportunidad de cargar un plugin cada vez que se abre una base de datos en IDA.

Ida nos permite cargar el plugin con la llamada **PLUGIN.init**. Cuando llame a la función **INIT** debe determinar si el plugin está diseñado para cargarlo con el estado actual de IDA. El significado de "**estado actual**" varía dependiendo de la opción de carga aplicada de entre las tres situaciones anteriores. Ejemplos de estado interesantes

pueden ser según el tipo de archivo a cargar (un plugin puede estar diseñado específicamente para archivos PE) o por el tipo de procesador (un plugin puede estar diseñado exclusivamente para utilizarlo con binarios x86).
Para poder indicar a IDA nuestros deseos, **PLUGIN.init** deberá retornar uno de los siguientes valores definidos en **loader.hpp**.

PLUGIN_SKIP Retornando este valor señala que el plugin no debe ser cargado.
PLUGIN_OK Retornando este valor instruimos a IDA para permitir el uso del plugin en la base de datos actual. Ida carga el plugin cuando el usuario activa el plugin utilizando la acción del menú o un atajo de teclado.
PLUGIN_KEEP Retornando este valor instruimos a IDA para que permita el uso del plugin en la base de datos actual y mantenga el plugin cargado en memoria.

Una vez que el plugin ha sido cargado, puede ser activado de dos formas distintas. El método más frecuente para activar un plugin es a través de la acción del usuario con el menú contextual o con un atajo de teclado. Cada vez que un plugin es activado de esta forma, ida pasa el control al plugin llamando a **PLUGIN.run**. Un método alternativo para la activación de un plugin es enganchar el plugin al sistema de notificación de eventos de IDA. En tal caso, un plugin debe mostrar interés por uno o más tipo de eventos de IDA y registrar una función de rellamada la cual será llamada por ida cuando ocurra el evento en el cual el plugin esté interesado.

Cuando llega el momento de descargar un plugin, ida llama a **PLUGIN.term** (asumiendo que no es NULL). Las circunstancias bajo las que un plugin es descargado varían de acuerdo a los bit habilitados en **PLUGIN.flags**. Los plugin que no especifican ningún bit bandera son cargados según el valor retornado por **PLUGIN.init**. Estos tipos de plugin son descargados cuando la base de datos para la cual se ha cargado se cierra.

Cuando un puglin especifica el bit bandera **PLUGIN_UNL**, este es descargado después de cada llamada a **PLUGIN.run**. Dichos plugin se deben recargar (resultado de una llamada a **PLUGIN.init**) con la consecuente activación. Cuando especifica el bit bandera **PLUGIN_PROC** es descargado cuando el módulo de procesador para el cual ha sido cargado, se descarga. Los módulos de procesador son descargados, siempre que se cierra una base de datos. Y finalmente los plugin que especifican el bit bandera **PLUGIN_FIX** sólo cuando el propio ida se cierra.

16.1.2.—Inicialización del plug-in

Los plugin son inicializados en dos fases. La inicialización estática de un plugin se realiza en el momento de compilación, mientras que la inicialización dinámica se realiza en el momento de su carga de la forma que lo indique la ejecución de **PLUGIN.init**. Como ya hemos dicho antes, cuando el campo **PLUGIN.flags** es inicializado en la compilación, dicta las distintas características de un plugin.

Cuando IDA es ejecutado, el campo **PLUGIN.flags** de cada plugin, en **<IDADIR>\plugins** es examinado. Las llamadas de ida a **PLUGIN.init** de cada plugin, le especifican la bandera **PLUGIN_FIX**. Dicha bandera es cargada antes de cualquier otro módulo de ida, para que tenga la oportunidad de notificar cada evento del que IDA sea capaz de generar, incluidas las notificaciones generadas por los módulos de carga y módulos de procesador. La función **PLUGIN.init** de tales plugin generalmente debe retornar o **PLUGIN_OK** o **PLUGIN_KEEP**, pero si es cargado en el arranque sólo

retornará **PLUGIN_SKIP**. Sin embargo si nuestro plugin está diseñado para ejecutar una tarea de inicialización una vez arranque ida, consideraremos la ejecución en la función **INIT** con lo cual retornará **PLUGIN_SKIP** indicando que el plugin no se necesita.

Cada vez que un módulo de procesador es cargado, ida prueba la bandera **PLUGIN_PROC** en cada plugin y las llamadas **PLUGIN.init** disponibles para cada plugin en donde esté habilitado **PLUGIN_PROC**. La bandera **PLUGIN_PROC** permite a los plugin responder a las notificaciones creadas por los módulos de procesador y con esto suplir los comportamientos de esos módulos. La función **PLUGIN.init** para dichos módulos tiene acceso al objeto global, **ph** de **processor_t**, para examinarlo, utilizarlo y determinar si hay que saltar o retener el plugin. Por ejemplo un plugin diseñado específicamente para usarlo con un módulo de procesador **MIPS** deberá retornar **PLUGIN_SKIP** si se está cargando el módulo de procesador x86, como podemos ver seguidamente:

```
void idaapi mips ( ) {
        if (ph.id != PLFM_MIPS) return PLUGIN_SKIP;
        else return PLUGIN_OK; //o, alternativamente PLUGIN_KEEP
}
```

Finalmente, cada vez que es cargada o creada una base de datos, la función **PLUGIN.init** de cada plugin que no haya sido cargado es llamada para determinar si el plugin debe ser cargado o no. En este punto cada plugin puede utilizar distintos criterios para determinar se ida debe retenerlo o no. Ejemplos de plugin especializados pueden ser los que ofrecen características específicas para ciertos tipos de archivo (ELF, PE, Mach-O, etc.), tipos de procesador o tipos de compilador.

A pesar de todo, cuando un plugin decide retornar **PLUGIN_OK** o **PLUGIN_KEEP**, la función **PLUGIN.init** deberá de asegurarse de cualquier acción de inicialización necesaria para asegurar la capacidad del plugin de ejecutar la propiedad por la que ha sido activado eventualmente. Cada recurso pedido por **PLUGIN.init**, deberá ser proporcionado por **PLUGIN.term**, una de las principales diferencias entre **PLUGIN_OK** y **PLUGIN_KEEP** es que el segundo impide que un plugin sea cargado y descargado repetidamente reduciendo así la necesidad de colocarlo o descolocarlo y recolocar recursos cuando el plugin especifique **PLUGIN_OK**. Por regla general **PLUGIN.init** debe retornar **PLUGIN_KEEP** cuando las futuras invocaciones del plugin dependan de los estados acumulados durante las previas invocaciones de este. Una forma de realizar dicha tarea es guardar cualquier información de estado en la base de datos abierta en ida utilizando un mecanismo de almacenamiento persistente como pueden ser los **netnodes**. Utilizando dicha técnica las invocaciones siguientes del plugin pueden localizar y utilizar los datos almacenados por las invocaciones anteriores de él. Este método tiene la ventaja de proporcionar el almacenamiento persistente no sólo a través de las invocaciones del plugin sino también a través de las sesiones de ida.

Para los plugin en los cuales la invocación es completamente independiente de cualquier invocación anterior, a menudo es adecuado que **PLUGIN.init** retorne **PLUGIN_OK**, con lo cual tenemos la ventaja de reducir la memoria de IDA reduciendo los módulos cargados en memoria en un momento dado.

16.1.3.—Notificación de eventos

Mientras que los plugin normalmente son activados con las acciones (**Edit>Plugins**) o a través de **atajos de teclado**, las capacidades de notificación de eventos ofrecen una alternativa para activar el plugin.

Cuando queramos que nuestro plugin nos notifique eventos específicos que tienen lugar dentro de IDA, deberemos registrar una función de rellamada expresa en los específicos tipos de eventos. La función **hook_to_notification_point** es utilizada para informar a ida; primero que estamos interesados en los eventos de una clase en particular y segundo que ida deberá llamar a la función que le indiquemos cada vez que ocurra el evento en la clase indicada. Un ejemplo de la utilización de hook_to_notification_point para registrar los eventos que nos interesan de una base de datos lo vemos seguidamente:

```
//typedef for event hooking callback functions (from loader.hpp)
typedef int idaapi hook_cb_t(void *user_data, int notification_code, va_list va);
//prototype for  hook_to_notification_point (from loader.hpp)
bool hook_to_notification_point(hook_type_t hook_type,
                        hook_cb_t *callback,
                        void *user_data);
int idaapi idabook_plugin_init() {
  //Example call to  hook_to_notification_point
  hook_to_notification_point(HT_IDB, idabook_database_cb, NULL);
}
```

Existen cuatro categorías de notificaciones: Notificaciones de procesador (**idp_notify** en **idp.hpp**, **HT_IDP**), notificaciones de interacción de usuario (**ui_notification_t** en **kernwin.hpp**, **HT_UI**), eventos de depuración (**dbg_notification_t** en **dbg.hpp**, **HT_DBG**) y eventos de base de datos (**idp_event_t** en **idp.hpp**, **HT_IDB**). Dentro de cada categoría de eventos existen distintas notificaciones codificadas con las cuales recibirán notificaciones eventos específicos. Ejemplos de notificaciones de base de datos (**HT_IDB**) son **idb_event::byte_patched**, lo cual indica que ha sido modificado un byte de la base de datos y **idb_event::cmt_changed**, para indicar que un comentario ordinario o repetitivo ha sido cambiado. Cada vez que ocurre un evento, ida invoca a cada función de rellamada registrada, pasando el código especifico de la notificación de evento y cualquier parámetro adicional específico al código de notificación. Los parámetros pasados para cada código de notificación están detallados en los archivos header del SDK que definen cada código de notificación.
Continuando con el ejemplo anterior, definimos una función de rellamada para manejar los eventos de la base de datos:

```
int tora_database_cb(void *user_data, int notification_code, va_list va) {
  ea_t addr;
  ulong original, actual;
  switch (notification_code) {
    case idb_event::byte_patched:
      direccion = va_arg(va, ea_t);
      actual = get_byte(addr);
      original = get_original_byte(addr);
      msg("%x sera modificada con %x.  Valor original era %x\n",
          direccion, actual, original);
      break;
  }
}
```

Este particular ejemplo reconoce sólo el mensaje de notificación **byte_patched**, el cual imprime la dirección del byte modificado, el nuevo valor del byte y el valor original del byte. Las funciones de rellamada de la notificación utilizan la lista de argumentos de variables C++, **va_list**, para proporcionar acceso a un número variable de argumentos, dependiendo de cómo ha sido enviado el código de notificación a la función. El número y tipo de argumentos proporcionados para cada código de notificación están especificados en los archivos **header** en donde cada código de notificación está definido. El código de notificación **byte_patched** está definido en **loader.hpp** y recibe un argumento del tipo **ea_t** en esta **va_list**. La macro C++ **va_arg** debería utilizarse para recuperar los argumentos sucesivos de una **va_list**. La dirección del byte modificado, en el ejemplo anterior es recuperada de **va_list** en **direccion = va_arg(va, ea_t);** .

Un ejemplo para desenganchar los eventos de notificación de una base de datos es el siguiente:

```
void idaapi tora_plugin_term ( ) {
        unhook_from_notification (HT_IDB, tora_database_cb, NULL);
}
```

Todos los buenos plugin deberían desenganchar todas las notificaciones siempre que sean descargados. Este es uno de los propósitos propuestos por la función **PLUGIN.term**.

16.1.4.—Ejecución de un plug-in

Hasta ahora hemos hablado de distintos procesos en los que ida llama a funciones pertenecientes a un plugin. Las operaciones de carga y descarga de un plugin son el resultado de las llamadas a **PLUGIN.init** y **PLUGIN.term**, respectivamente. La activación de un plugin por el usuario se realiza con el menú **Edit>Plugins** o con **atajos de teclado** asociados al plugin todo esto es resultado de la llamada a **PLUGIN.run**. Finalmente, las funciones de rellamada registradas por un plugin, serán llamadas en respuesta a distintos eventos que ocurran dentro de IDA.

Para conocer cómo va a ejecutarse un plugin, es importante comprender unos cuantos hechos esenciales. Las funciones del plugin son invocadas por el bucle principal de procesamiento de eventos de ida. Mientras un plugin se esté ejecutando, ida no puede procesar eventos, incluidos las tareas de análisis pendientes, o la interconexión con el usuario. Por lo tanto es importante que nuestro plugin ejecute su tarea lo más rápidamente posible y retorne el control a IDA. De otra forma ida estará completamente pausado y no existirá ninguna forma de recobrar su control. En otras palabras una vez esté ejecutándose nuestro plugin no habrá ninguna manera de salir de él. Deberemos esperar a que termine o cerrar el proceso de IDA. En este último caso si tenemos una base de datos abierta, puede ocurrir que se corrompa y no pueda ser reparada por ida. El SDK nos proporciona tres funciones que pueden utilizarse para intentar reparar dicha situación. La función **show_wait_box** puede ser llamada para mostrar un diálogo con el mensaje **Please wait…** conjuntamente con un botón **Cancel**.

Verificando periódicamente si el usuario ha pulsado dicho botón, llamando a la función **wasBreak**. La ventaja de esto es que cuando **wasBreak** es llamado, ida tendrá la oportunidad de actualizar la interconexión con el usuario y permitir a nuestro plugin la

oportunidad de decidir si debe parar el proceso que está realizando. En este caso, debe llamar a **hide_wait_box** para quitar el diálogo de espera.

No existe ningún mecanismo para sincronizar el acceso a las muchas variables globales utilizadas por ida, ni existe mecanismo alguno para asegurar la integridad de las transacciones de base de datos. En otros términos, si creamos un nuevo hilo y utilizamos las funciones SDK dentro de este hilo para modificar la base de datos, podríamos corromper dicha base de datos, ya que ida podría entrar en conflicto entre sus modificaciones y el intento de las nuestras.

Teniendo en cuenta dichas limitaciones, para la mayoría de los plugin, la gran parte de trabajo realizado por el plugin se ejecutará dentro de **PLUGIN.run**. Realizándolo en nuestro objeto inicializado **PLUGIN**, una mínima ejecución de **PLUGIN.run** puede parecerse a lo siguiente:

```
void idaapi tora_plugin_run (int arg)   {
        msg (“plugin tora activado¡\n”);
```

Cada plugin tiene a su disposición las **API IDA y C++.** Pueden disponerse capacidades adicionales si vinculamos nuestro plugin con librerías específicas en plataformas apropiadas. Por ejemplo, la API de Windows estará disponible para plugins desarrollados para ejecutarse con versiones de Ida Windows. Para hacer algo más interesante que imprimir un mensaje en la ventana **message**, necesitaremos comprender cómo realizar la tarea deseada utilizando las funciones disponibles del **IDA SDK**. Tomando el código siguiente, por ejemplo, podríamos desarrollar la siguiente función:

```
void idaapi extended_plugin_run(int arg) {
  func_t *func = get_func(get_screen_ea());  //toma la funcion en la ubicacion actual del cursor
  msg("El tamano de la variable local es %d\n", func->frsize);
  msg("El tamano de los registros 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 direccion 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 is at 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);
      }
    }
  }
}
```

Utilizando esta función tenemos el corazón de un plugin el cual nos vuelca la información del stack frame de la función seleccionada, cada vez que se active dicho plugin.

Performance Bigundill@