13.1.— El vergonzoso menú “Patch Program”

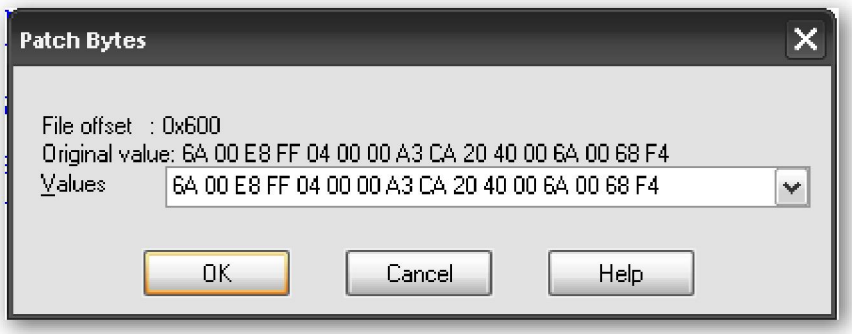
Esta opción fue ya mencionada en el escrito 10, se encontraba en **Edit > Patch Program**, es una característica oculta en la versión GUI de IDA. Recordemos que se habilitaba editando el archivo de configuración **idagui.cfg** (en la versión de cónsola está habilitado por defecto). La siguiente figura muestra los opciones del submenú de **Patch Program.**

Cada elemento del menú nos da a entender que podremos modificar el binario de distintas formas. De hecho nos proporciona la opción de modificar la base de datos de tres formas distintas. En realidad, los tres elementos de este menú son de todos, los que hacen más distinción entre la base de datos de IDA y el binario del cual se ha creado. Una vez que la base de datos es creada, IDA nunca referencia al binario original. Resumiendo dado su comportamiento real, el menú de dichas opciones se tendría que llamar más bien **Patch Database**.

Sin embargo, no está todo perdido, las opciones del menú de la figura anterior nos proporcionan la forma más fácil para poder observar el efecto de todos los cambios que hagamos y eventualmente utilizar dicha información para parchear el binario original.

13.1.1.—Cambiar bytes individualmente en la base de datos

La acción de menú **Edit > Patch Program > Change Byte** se utiliza para editar uno o más valores de byte de una base de datos de IDA. La siguiente figura muestra el diálogo con la relación de los bytes editados.

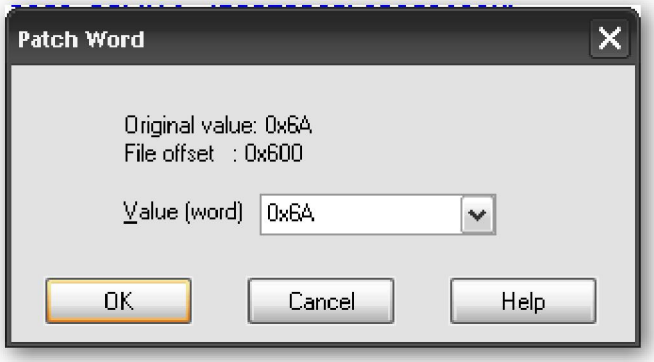


El diálogo muestra el valor de los primeros 16 bytes a partir de la posición actual del cursor. Podemos cambiar alguno o todos los bytes mostrados, pero no podemos realizar ningún cambio más allá del décimo sexto byte sin cerrar el diálogo, si lo cerramos colocaremos el cursor en una nueva posición y lo reabriremos. Observemos que el diálogo nos muestra el valor offset del archivo, de los bytes que podemos cambiar. Este valor refleja el offset hexadecimal en donde están ubicados los bytes dentro del archivo binario original. El valor no refleja la dirección virtual de donde están ubicados los bytes en la base de datos actual. El hecho de que IDA nos conserve en la base de datos la información del offset del archivo original para cada byte, es lo que nos permitirá desarrollar un parche para el binario original. Para finalizar digamos que a pesar de todos los cambios que hayamos realizado en los bytes de la base de datos, el campo en el diálogo **Original value**, siempre nos mostrará el valor original cargado en la base de datos de los bytes. No existe ninguna característica automática para rehacer los cambios

a los valores originales del byte, sin embargo podríamos crear un script para IDA para realizar la tarea.

13.1.2.—Cambiar en la base de datos un "Word"

Menos útil que la capacidad de parchear bytes, tenemos la opción de parchear un **Word**. La siguiente figura nos muestra el diálogo para parchear un Word (2-byte) de una vez.

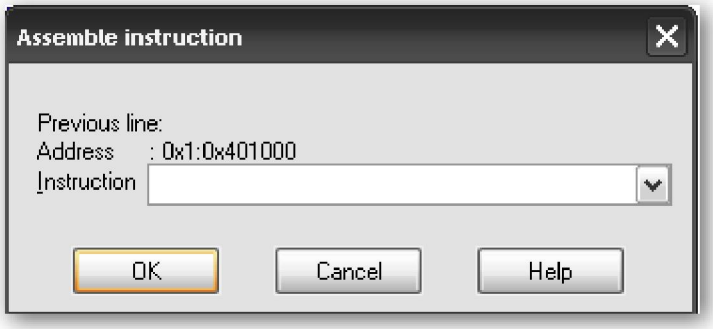


Al igual que el diálogo de parcheado de bytes, se muestra el offset del archivo y no la dirección virtual del word que se está modificando. Una observación importante es que el valor de los bytes del word se muestran en el orden natural del procesador. Por ejemplo, en un desensamblado para **x86** los **word** son tratados como valores **little-endian**, mientras que en un desensamblado **MIPS**, los **word** son tratados como valores **big-endian**. Tengámoslo en cuenta cuando introduzcamos valores nuevos al word. También como en el parcheado de byte, el campo **Original value** siempre muestra el valor inicial cargado del archivo binario original, sin tener en cuenta todos los posibles cambios que le hayamos realizado.

13.1.3.—Utlizar el diálogo Assemble

Quizás la característica más importante del menú **Patch Program** es la opción **Assemble**, llegamos a ella con **Edit > Patch Program > Assemble**. Por desgracia esta capacidad no se puede utilizar con todos los tipos de procesadores, y está ligada a la presencia de la capacidad interna de ensamblado dentro del módulo de procesador que se utilice. Por ejemplo, en el módulo de procesador x86 se reconoce dicha capacidad, mientras que en el módulo de procesador MIPS no se reconoce dicha característica. Cuando el ensamblado no se permite, recibiremos un mensaje de error como **"Sorry, this processor module doesn't support the assembler".**

La opción **Assemble** permite introducir declaraciones ensamblador, las cuales son ensambladas utilizando un ensamblador interno. Los bytes de la instrucción introducida se escriben en la ubicación actual de la pantalla. La siguiente figura muestra el diálogo **Assemble instruction** utilizado para introducir nuestra instrucción deseada.



Cada vez sólo podemos entrar una instrucción en el campo **Instruction**. El componente ensamblador de IDA para el módulo de procesador x86 acepta la misma sintaxis utilizada para x86 en el listado de desensamblado. Cuando hagamos click en **OK**, o presionamos **ENTER**, nuestra instrucción es ensamblada y los bytes correspondientes a dicha instrucción son introducidos en la base de datos empezando desde la dirección virtual mostrada en el campo **Address**. El ensamblador interno de IDA nos permite utilizar los nombres de símbolos en nuestras instrucciones siempre y cuando dichos nombres existan dentro del programa. Una sintaxis como **mov [ebp+var_4], eax y call sub_401896** son correctas, y el ensamblador resolverá correctamente las referencias a los símbolos.

Una vez hayamos introducido una instrucción, el diálogo permanece abierto y listo en espera de una siguiente instrucción, la cual será ubicada en la dirección virtual inmediatamente después de la instrucción introducida previamente. Mientras no añadamos más instrucciones, el diálogo mostrará en el campo **Previous line** la instrucción previamente introducida.

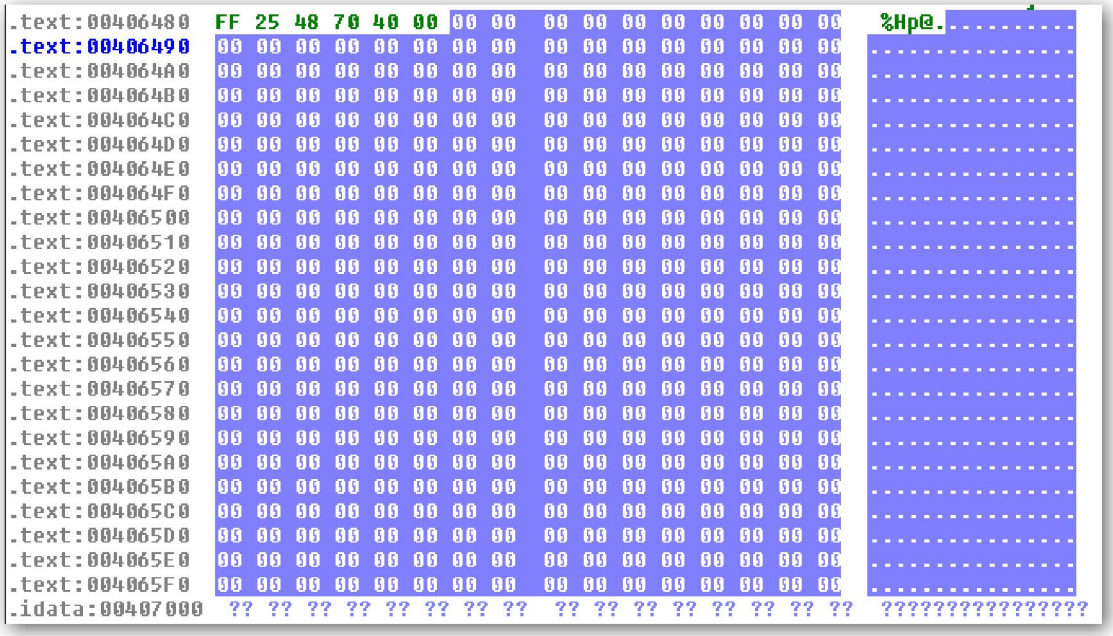
Al introducir nuevas instrucciones, debemos tener en cuenta la alineación de la instrucción, especialmente cuando la instrucción introducida tenga una longitud distinta a la que se reemplaza. Cuando la nueva instrucción es más corta que la que se reemplaza, necesitas rellenar el exceso de bytes de la instrucción antigua, una posible opción es la inserción de instrucciones **NOP**. Cuando una nueva instrucción es más larga que la reemplazada, IDA sobrescribirá los bytes necesarios de la siguiente instrucción para que quepa la introducida. Esto quizá, puede o no, ser lo que queramos, por lo tanto antes de entrar instrucciones de ensamblado nuevas hay que planificar los bytes que serán modificados y cómo serán modificados. Una forma fácil de ver como quedará el ensamblado es utilizar un editor de texto en modo sobrescribir. No existe ninguna forma directa para insertar nuevas instrucciones sin sobrescribir las ya existentes.

Es importante recordar que las capacidades de IDA para parchear bases de datos se limitan a pequeños parcheados, que encajen fácilmente en el espacio existente dentro de la base de datos. Si tenemos un parche con necesidad de mucho espacio, necesitaremos localizar alguno dentro del binario original que no sea utilizado por él. Dicho espacio a menudo son rellenos, insertados por los compiladores para alinear las secciones de un archivo. Por ejemplo, en muchos archivos **Windows PE**, cada sección del programa deberá de empezar con **offset** del archivo los cuales sean múltiplos de **512 bytes**. Cuando una sección no ocupa un espacio múltiplo de 512 bytes, dicha sección debe ser rellenada dentro del archivo con el fin de mantener dicho límite para la siguiente sección. Las siguientes líneas de un desensamblado de archivo PE muestran esta situación:

```
.text:00406480 ; [00000006 BYTES: COLLAPSED FUNCTION RtlUnwind. PRESS KEYPAD "+" TO EXPAND]
.text:00406486                 align 200h
.text:00406486 _text           ends
.text:00406486
.idata:00407000 ; Section 2. (virtual address 00007000)
```

En este caso IDA está utilizando la directiva **align** para indicar que ésta sección se rellenará con un máximo de 512 bytes (200h) y que el inicio será en la dirección **text : 00406486** y el final del relleno será el siguiente múltiplo de 512 bytes, que es la

dirección **text : 00406600**. El área de relleno normalmente es llenada con ceros por el compilador, lo cual como podemos ver en la **hex view** se ve claramente:



En este binario el espacio que será rellenado es de exactamente de **378 (406600 – 406486 = 17A) bytes**, el cual se rellenará con ceros hasta el final de la sección **.text**. Aquí podríamos parchear una función para que saltará a esta área del binario y ejecutar las instrucciones que insertáramos y una vez ejecutadas retornara a la función original.

Observemos que la siguiente sección del binario, la sección **.idata**, en realidad no empieza hasta la dirección **idata : 00407000**. Este es un resultado de la alineación de la memoria, no de la alineación del archivo, debido a que las secciones de un PE deben empezar en límites de 4Kb (una página de memoria). En teoría debería ser posible inyectar un parche de datos de **2560 bytes** entre los límites de memoria de **00406600 – 00407000**. La dificultad para realizarlo estriba en el hecho de que ningún byte correspondiente a estos límites de memoria está presente dentro del **disk image** del ejecutable. Para poder utilizar dicho espacio necesitaríamos sobrescribir más de una parte del archivo original. Primero necesitaríamos insertar un bloque de datos de **2.048 byte** entre el final de la sección **.text** y el inicio de la sección **.idata**. Luego, necesitaríamos ajustar el tamaño de la sección **.text** en los encabezados del archivo PE. Finalmente, necesitaríamos ajustar la ubicación de **.idata** y todas las siguientes secciones en los encabezados PE, para reflejar en todas las secciones el movimiento de los **2.048 bytes** introducidos en el archivo. Estos cambios nos pueden parecer muy complicados, pero sólo requieren un poco de atención en los cálculos y tener conocimiento del formato de un archivo PE.

Performance Bigundill@