IDA de “cabo a rabo”

1.-- Preparando el camino

Para cualquiera que quiera iniciarse en el mundo de la ingeniería inversa, debe saber lo importante que es desarrollar las habilidades de programación pertinentes a esta ciencia.. Lo ideal sería gustarte tanto el código hasta el punto de que comieras, durmieras y respiraras con él. Por otro lado si cuando ojeas cualquier código te da “repelús”, esto quiere decir que la ingeniería inversa no es para ti. En mi caso, el realizar ingeniería inversa es un desafío como el que espera cada día el crucigrama del periódico para solucionarlo. Es un desafío a problemas particularmente difíciles.

La idea de estos escritos será, ayudar a otros y a mí mismo en el inicio o profundización ,según el caso de cada cual, de realizar ingeniería inversa utilizando la herramienta IDA.

Como bien dice el título, IDA de “cabo a rabo” y como dicho programa es el mejor desensamblador de hoy día. Vamos a iniciarnos con una pequeña explicación de ¿qué es un desensamblado?. Como la intención no es solamente explicar el manual de IDA Pro, sino también utilizar a éste para referirnos a técnicas de ingeniería inversa las cuales analizarán distintos análisis de aplicaciones.

1.1--Introducción a la teoría del desensamblado

Cualquiera que alguna vez haya estudiado algún lenguaje de programación, lo más seguro es que haya aprendido las distintas generaciones de lenguajes, no obstante haremos un breve resumen para refrescar la memoria.

Lenguajes de primera generación

Son los lenguajes de más bajo nivel, normalmente consisten en ceros y unos, estos no son legibles para una persona normal. En este nivel de lenguaje cualquier acción es difícil de entender y de seguir los datos de las instrucciones. Estos lenguajes también son llamados **lenguajes máquina** y en otros casos **bytecode** mientras que a los programas en dicho lenguaje se les llaman **binarios**.

Lenguajes de segunda generación

Estos son los llamados **lenguajes ensamblador**, es un paso intermedio del lenguaje máquina y normalmente combina pautas de bit específicas o códigos de operaciones (**opcodes**), estos son cortas secuencias de caracteres memorizadas llamados **mnemonics**. Normalmente, estos mnemonics ayudan a los programadores a asociar la acción que éste realizará en el programa. El ensamblador, es la herramienta que utilizan los programadores para trasladar el lenguaje ensamblador a lenguaje máquina para que pueda ejecutarse la aplicación.

Lenguajes de tercera generación

Estos lenguajes son un paso más avanzado hacia los lenguajes con capacidad de expresión y a los lenguajes naturales nuestros. Estos proporcionan nombres clave y construcciones que los programadores utilizarán para construir bloques de programación en la aplicación. Normalmente estos lenguajes son de plataforma independiente, con lo cual sus características son utilizadas únicamente en sistemas operativos específicos. Algunos ejemplos son FORTRAN, COBOL, C, PYTHON,

JAVA. Los programadores normalmente utilizan compiladores para trasladar dichos programas a lenguaje ensamblado, lenguaje máquina o bytecode.

1.1.1-- ¿Qué podemos decir del desensamblado?

Tomando un modelo tradicional de programación los compiladores, ensambladores y enlazadores se utilizan, por separado o en combinación, para crear programas ejecutables. Para realizar ingeniería inversa, utilizamos herramientas para deshacer los procesos de ensamblado y compilación. No nos tiene que sorprender que dichas herramientas se llamen **desensambladores y descompiladores** ya que esta es su función. Un desensamblador deshace el proceso de ensamblado con lo cual nos dará como resultado lenguaje ensamblado. Por otro lado los descompiladores nos proporcionan como resultado un lenguaje de alto nivel del ensamblado o lenguaje máquina. El poder descompilar los programas para tener en nuestras manos el código fuente es una muy buena razón para un ingeniero inverso. Las siguientes razones son algunas de las que hacen difíciles dicha descompilación:

El proceso de compilación se pierde

En el lenguaje máquina no existen variables ni nombres de funciones, y la información del tipo de las variables sólo pueden ser determinadas por los datos explícitos usados antes de las declaraciones. Cuando observas un dato de 32 bits que será transferido, es difícil determinar si esos 32 bits representan un entero, un valor 32 bits punto flotante o un puntero de 32 bits.

La compilación se puede realizar de muchas formas

Esto significa que un código fuente puede traducirse a lenguaje ensamblado de muchas maneras y asimismo el lenguaje máquina puede traducirse a código fuente de otras muchas formas, la consecuencia es que al descompilar dicho código compilado nos puede proporcionar código fuente distinto.

Los descompiladores dependen mucho del lenguaje y las librerías utilizadas. Procesar un binario con un compilador Delphi y descompilarlo con un descompilador de C, nos puede proporcionar resultados extraños. De la misma forma, descompilando un binario Windows con un descompilador que no conozca las API de programación de Windows, no nos servirá de nada.

Por lo tanto necesitamos una capacidad de desensamblado casi perfecta para descompilar un binario. Cualquier error u omisión en la fase de desensamblado, lo propagaremos a lo largo de todo el código descompilado. Por esta razón estudiaremos el descompilador más sofisticado que existe hoy día en el mercado el **IDA Pro** de DataRescue y posteriormente de Hex-Rays.

1.1.2-- ¿Por qué hay que desensamblar?

El propósito de los desensambladores es facilitar la comprensión de los programas cuando no disponemos de su código fuente. Algunas de las situaciones donde podemos necesitar el desensamblado son:

** Análisis de malware
** Análisis de vulnerabilidades en software de código cerrado.

** Análisis de la interoperabilidad en software de código cerrado.
** Análisis del código compilado generado para validar su correcta ejecución.
** Mostrar las instrucciones del programa mientras depuramos.

Vamos a realizar una pequeña explicación de cada situación.

Análisis de Malware
A menos que estemos estudiando un gusano basado en un script, los autores de malware raramente nos proporcionan el código fuente de sus creaciones. Con un código fuente deficiente es casi imposible saber exactamente como se comporta un malware. Las dos técnicas principales para el análisis del malware son su análisis dinámico y su análisis estático. El análisis dinámico presupone ejecutar dicho malware dentro de un entorno controlado (entorno virtual), para poder registrar cada acción notable de su comportamiento utilizando cualquier instrumento del sistema. En contrapartida, el análisis estático es intentar comprender el comportamiento del programa sólo leyendo su código, que en el caso del malware, generalmente consiste en un listado del desensamblado.

Análisis de vulnerabilidades
Para simplificar su comprensión, el proceso de una auditoría de seguridad lo dividiremos en tres procesos: descubrir la vulnerabilidad, analizar la vulnerabilidad y programar su explotación. Los mismos pasos se realizarán teniendo o no el código fuente de la aplicación, no obstante la dificultad aumenta si sólo tenemos el binario. El primer paso es descubrir una condición potencialmente explotable en un programa. Esto se realiza utilizando técnicas dinámicas como “fuzzing” (ver Wiki), por otro lado también se puede realizar con su análisis estático del código pero con mucho más trabajo. Una vez se ha encontrado el problema, hay que realizar otro análisis para ver si este es explotable y si es así en que condiciones.

Los listados de desensamblado proporcionan el nivel de conocimiento necesario para comprender exactamente como el compilador ha elegido distribuir las variables del programa. Por ejemplo, será útil para conocer que un conjunto de caracteres de 70 bytes declarados por el programador, será redondeado a 80 bytes distribuidos por el compilador. El listado también proporciona el único medio para determinar como el compilador ha distribuido las variables globales en las funciones. Comprender la relación espacial existente entre las variables es esencial para desarrollar las explotaciones. Finalmente decir que utilizando un desensamblador y un depurador en conjunto, se puede desarrollar una explotación.

Interoperabilidad del software
Cuando una aplicación es entregada con solamente su binario, es muy complicado crear cualquier software que pueda ínter operar con él o proporcionar algún plugin que pueda interactuar. Un ejemplo común es el driver que se suministra para un hardware específico y que sólo funciona para una determinada plataforma, en este punto podemos utilizar la ingeniería inversa para desarrollar los drivers necesarios para que soporten dicho hardware. En estos casos la única forma de realizarlo es el análisis estático del código del firmware para conseguir realizar nuestro driver.

Validación de la compilación
El propósito de un compilador o ensamblador es generar lenguaje máquina, los desensambladores se utilizan para verificar que dicha compilación está en concordancia con las especificaciones del diseño de programación. Además de poder realizar correcciones, también se utilizan para optimizar la compilación desde un punto de vista de seguridad comprobando que el mismo compilador no haya comprometido dicha seguridad insertando "back doors" en el código generado.

Mostrar la depuración
Quizás el uso más común de los desensambladores es generar listados de instrucciones en los depuradores. Por desgracia los ensambladores incluidos en los depuradores suelen ser poco sofisticados, una excepción es el del OllyDbg como ya sabemos muy notable. Generalmente son incapaces de desensamblar un batch y se detienen cuando no pueden determinar los límites de una función. Esta es una de las razones de utilizar un depurador en conjunción de un desensamblador de calidad, para obtener una situación del contexto en la depuración.

1.1.3-- ¿Cómo desensamblar?
Ahora que ya hemos visto los motivos para desensamblar, vamos a ver como funciona realmente dicho proceso. Consideremos una tarea normal de un desensamblador: Tomemos 100 KB, de código de datos, convertidos en lenguaje ensamblado para mostrar al usuario. Podemos pedirle al desensamblador cosas concretas como localizar funciones, reconocer saltos redireccionados e identificar variables locales, poniéndole el trabajo más difícil al desensamblador.

Para poder realizar nuestras peticiones, cualquier desensamblador necesita poder elegir un algoritmo para desplazarse dentro del archivo proporcionado. La calidad del listado de desensamblado es directamente proporcional a la calidad del algoritmo utilizado. Vamos a estudiar dos algoritmos fundamentales para realizar el desensamblado del código máquina. Al igual que su estudio, también mostraremos sus deficiencias a fin de estar preparados para las situaciones en que fallen. Comprendiendo las limitaciones de un desensamblador, seremos capaces de mejorar manualmente la calidad de salida del listado de desensamblado.

Algoritmo básico de desensamblado
Para empezar, desarrollemos un sencillo algoritmo con el cual se acepte la entrada de lenguaje máquina y nos produzca una salida en lenguaje ensamblado. Haciéndolo, obtendremos la comprensión de las suposiciones, acciones y retos en los que se sustenta un desensamblado automatizado.

Paso1
El primer paso de un proceso de desensamblado es identificar la parte de código a desensamblar. Esto no es tan fácil como pueda parecer. Normalmente las instrucciones están mezcladas con los datos y es muy importante poder separar unas de otras. En la mayoría de los casos, se realiza el desensamblado de un archivo ejecutable, este archivo tiene un formato común como puede ser el **Portable Executable (PE)** utilizado en Windows o el **Executable and Linking Format (ELF)** utilizado en muchos sistemas con base UNIX. Estos formatos contienen mecanismos, encabezados en jerarquia, para localizar en el código del archivo sus **secciones** y sus **entry points** que contengan código.

Paso2
Una vez tengamos la dirección inicial de una instrucción, el próximo paso es leer el valor que contiene dicha dirección, este valor es el llamado **Offset** del archivo, y repasar la tabla para hacer coincidir los valores de los **opcodes** binarios con los **mnemonics** del lenguaje ensamblado. Dependiendo de la complejidad de las instrucciones desensambladas, este proceso puede ser sencillo o puede suponer realizar varias operaciones para llegar a la comprensión del comportamiento de la instrucción y determinar los operandos requeridos para realizar dicha instrucción. Para el conjunto de instrucciones con instrucciones de longitud variable, como el Intelx86, se necesitarán instrucciones de bytes adicionales para desensamblar una instrucción sencilla.

Paso3
Una vez que una instrucción ha sido comprendida y cualquier operando necesario descifrado, su equivalente en lenguaje ensamblado es formateado y sacado como parte del listado de desensamblado. Esta salida se puede mostrar con distinta sintaxis de lenguaje ensamblado. Por ejemplo los dos formatos predominantes del lenguaje ensamblado son el formato **Intel** y el formato **AT&T.**

> Sintaxis del ensamblado X86: AT&T VS INTEL
> Aunque los dos son lenguajes de segunda generación, la sintaxis de los dos varía notablemente respecto a variables, constantes y acceso a los segmentos de registro y tamaño de la instrucción respecto a la dirección y offset. La sintaxis AT&T se distingue por colocar el símbolo **%** delante de todos los nombres de registro y utilizar el símbolo **$** como prefijo de las constantes literales (operando inmediato) y que el orden de los operandos es, el operando fuente es el de la izquierda y el operando destino es el de la derecha. Utilizando la sintaxis AT&T la suma de 4 a eax sería: **add $0x4, %eax**.
>
> En sintaxis Intel no se requieren los prefijos de registro ni literales y el orden de los operandos son inversos. La misma instrucción de sumar 4 a eax, tendría la forma de **add eax, 0x4**. La sintaxis Intel se utiliza en MASM, TASM y NASM.

Paso4
Una vez obtenemos la salida de la instrucción, necesitamos tomar la siguiente instrucción y repetimos el proceso anterior, hasta que tengamos desensamblada cada instrucción del archivo.

Existen distintos tipos de algoritmos; para determinar donde empezar a desensamblar, para escoger la instrucción siguiente a desensamblar, cómo distinguir entre código y datos y cómo determinar cuando ha sido desensamblada la última instrucción. Los dos algoritmos predominantes en el desensamblado son el **barrido lineal** y la **descendencia recursiva.**

1.2-- Desensamblado de barrido lineal

El algoritmo de desensamblado de barrido lineal, es una forma muy directa para localizar las instrucciones a desensamblar: Donde acaba una instrucción empieza la otra. En consecuencia lo más difícil es decidir donde empezar a desensamblar. La solución normal es desensamblar cualquier sección que contenga código, especificadas en condiciones normales en los encabezados (headers) del archivo, y que represente

instrucciones de lenguaje máquina. El desensamblado se inicia en el primer byte de una sección de código y se va desplazando de forma lineal, desensamblando una instrucción después de la otra hasta finalizar la sección seleccionada. En este no se hace hincapié en explicar el control del flujo de ejecución de las distintas desviaciones. El desensamblado lineal va a "piñón fijo" una instrucción detrás de la otra y hasta el final.

Durante el proceso de desensamblado, se puede mantener un puntero al inicio de la instrucción actual que será desensamblada. Como parte del proceso de desensamblado, se calcula la longitud de cada instrucción y se utiliza para determinar la localización de la siguiente instrucción a desensamblar.

La ventaja principal del algoritmo de barrido lineal es que proporciona una cobertura total de las secciones de código de un programa. Una de sus desventajas es que no puede comprender los datos que se producirán con el código. Esto lo podemos ver en el listado 1-1, este muestra una función desensamblada con barrido lineal. Esta función contiene una declaración desviación (salto) y el compilador utilizado ha decidido que el salto se ejecutará utilizando un salto redireccionado (**jump table**). Además también ha colocado un salto redireccionado a él mismo. La declaración **jmp** en **00401250**, referencia una dirección redireccionada que se inicia con el valor **00401257**. Por desgracia el desensamblador trata a la dirección **00401257** como si fuera una instrucción e incorrectamente genera su representación en lenguaje ensamblador.

Listado 1-1 desensamblado linear

```
40123f:    55                      push   ebp
401240:    8b ec                   mov    ebp,esp
401242:    33 c0                   xor    eax,eax
401244:    8b 55 08                mov    edx,DWORD PTR [ebp+8]
401247:    83 fa 0c                cmp    edx,0xc
40124a:    0f 87 90 00 00 00       ja     0x4012e0
401250:    ff 24 95 57 12 40 00    jmp    DWORD PTR [edx*4+0x401257]
401257:    e0 12                   loopne 0x40126b
401259:    40                      inc    eax
40125a:    00 8b 12 40 00 90       add    BYTE PTR [ebx-0x6fffbfee],cl
401260:    12 40 00                adc    al,BYTE PTR [eax]
401263:    95                      xchg   ebp,eax
401264:    12 40 00                adc    al,BYTE PTR [eax]
401267:    9a 12 40 00 a2 12 40    call   0x4012:0xa2004012
40126e:    00 aa 12 40 00 b2       add    BYTE PTR [edx-0x4dffbfee],ch
401274:    12 40 00                adc    al,BYTE PTR [eax]
401277:    ba 12 40 00 c2          mov    edx,0xc2004012
40127c:    12 40 00                adc    al,BYTE PTR [eax]
40127f:    ca 12 40                lret   0x4012
401282:    00 d2                   add    dl,dl
401284:    12 40 00                adc    al,BYTE PTR [eax]
401287:    da 12                   ficom  DWORD PTR [edx]
401289:    40                      inc    eax
40128a:    00 8b 45 0c eb 50       add    BYTE PTR [ebx+0x50eb0c45],cl
401290:    8b 45 10                mov    eax,DWORD PTR [ebp+16]
401293:    eb 4b          jmp   0x4012e0
```

Si examinamos los valores de grupos de 4-byte sucesivos empezando en el **00401257**, vemos que cada uno representará un puntero a una dirección cercana que en realidad será el destino de los distintos saltos (**004012e0, 0040128b, 00401290, …**). Así pues la instrucción **loopne** en **00401257**, no es ninguna instrucción como tal, sino que indica un fallo del algoritmo de barrido lineal provocado para poder distinguir correctamente los datos adjuntados al código.

El barrido lineal es utilizado por desensambladores como; el depurador **GNU (gdb)**, el depurador **WinDbg** de Microsoft y la utilidad **objdump**.

1.3-- Desensamblado recursivo descendiente

El recursivo descendiente localiza las instrucciones de distinta forma que el lineal. Este se centra en el concepto de flujo de control de ejecución, lo cual determina si una instrucción debe desensamblarse o no, basándose en si está referenciada por otra instrucción. Para comprenderlo, nos ayudará mucho clasificar las instrucciones según en como éstas afecten al puntero de instrucción CPU.

Instrucciones de flujo de ejecución secuenciales

Las instrucciones de flujo consecutivas pasan inmediatamente su ejecución a la instrucción siguiente. Un ejemplo de instrucciones secuenciales son instrucciones aritméticas simples tal como **add**; transferir instrucciones del registro a memoria, tal como **mov**; y operaciones de manipulación de pila como **push** y **pop**. Para dichas instrucciones, el desensamblador procederá con un barrido lineal.

Instrucciones de desviación de flujo condicionales

Las instrucciones de desvío condicionales, como la instrucción **jnz**, nos ofrecen dos posibles rutas de ejecución. Si la condición evaluada es cierta, la desviación se produce y el puntero a la instrucción debería cambiar reflejando el objetivo marcado por el desvío. Sin embargo, si la condición es falsa, la ejecución continua en modo lineal y el barrido lineal desensamblará la siguiente instrucción. Como normalmente en un contexto de desensamblado estático, es imposible determinar el resultado de una verificación condicional, el algoritmo de descendencia recursiva desensambla ambas rutas de ejecución, añadiendo las instrucciones de los dos posibles objetivos a una lista de direcciones para poderlas desensamblar más tarde.

Instrucciones de desviación de flujo incondicionales

Los desvíos incondicionales no siguen el modelo de flujo lineal y por lo tanto son tratados de otra forma por el algoritmo recursivo descendiente. Al igual que con las instrucciones de flujo consecutivas, la ejecución puede seguir hacia una sola instrucción; sin embargo no necesariamente necesita seguir en este momento la instrucción de desvío. En realidad como hemos visto en el ejemplo listado 1-1, no existe ningún requisito para que una instrucción siga en ese momento una desviación incondicional. Por lo tanto no hay razón para desensamblar los bytes que existan en el desvío incondicional.

Un desensamblado recursivo descendiente intentará determinar el objetivo del salto incondicional y añadir la dirección del destino a la lista de direcciones que aún tienen que ser exploradas. Por desgracia ciertas desviaciones incondicionales pueden causar problemas en el desensamblado recursivo descendiente. Esto sucede cuando el objetivo de un salto dependa de un valor en tiempo de ejecución, ya que no es posible determinar su destino utilizando el análisis estático. La instrucción **jmp eax** nos muestra este problema. El registro **eax** contiene un valor que sólo sabremos cuando el programa se esté ejecutando. Como durante el análisis estático **eax** no tiene ningún valor de podremos determinar el objetivo de la instrucción **jmp**, en consecuencia no podremos determinar dónde continuar el proceso de desensamblado.

Instrucciones de llamada a funciones (call)
Las instrucciones de llamada a funciones operan de manera similar a las instrucciones de saltos incondicionales, incluido el no poder determinar su objetivo si la instrucción es del tipo **call eax**, pero con la particularidad que normalmente el flujo de ejecución se retorna a la instrucción inmediatamente posterior a la instrucción de llamada, una vez completada la ejecución de la función. En este sentido son similares a las instrucciones de desvío condicional que generan dos rutas de ejecución. La dirección objetivo de la instrucción **call** es añadida a una lista de desensamblado postergado, mientras que la siguiente instrucción del **call** es desensamblada similarmente al barrido lineal.

La descendencia recursiva puede fallar si el programa no se comporta como es de esperar al retornar de las funciones llamadas. Por ejemplo el código de una función puede manipular deliberadamente la dirección de retorno de esa función de modo que los resultados de control al finalizar esta son distintos a los esperados por el desensamblador. Veamos un ejemplo simple de retorno incorrecto, vemos que la función **foo** añade una dirección de retorno distinta a la de retorno real del llamador.

```
foo                 proc near
  FF 04 24          inc     dword ptr [esp] ;incrementa la dirección de retorno guardada
  C3                retn
foo                 endp
; ---------------------------------------
bar:
  E8 F7 FF FF FF    call    foo


 05 89 45 F8 90           add     eax, 90F84589h
```

Esto nos da como resultado que cuando retorna no tomará la siguiente instrucción, que es **add**, del **call foo**, el desensamblado real sería el siguiente:

```
foo                 proc near
  FF 04 24          inc     dword ptr [esp]
  C3                retn
foo                 endp
; ---------------------------------------
bar:
  E8 F7 FF FF FF    call    foo
  05                db    5 ;antes era el primer byte de la instrucción add
  89 45 F8          mov   [ebp-8], eax ;instrucción correspondiente a los bytes
  90                nop
```

Este listado más claro muestra el flujo de ejecución real del programa en que la función **foo** retorna a la instrucción **mov**. Es importante comprender que un desensamblado de barrido lineal tampoco logrará desensamblar correctamente este código, pero por otras razones distintas.

Instrucciones de retorno
En algunos casos el algoritmo de descendencia recursiva se queda sin rutas para continuar. Una función de retorno, por ejemplo **ret**, no ofrece ninguna información acerca de la instrucción que se ejecutará después. Si el programa se esta ejecutando, se puede tomar una dirección de la parte superior del **stack** y la ejecución se retomará en dicha dirección. Los desensambladores no pueden acceder al **stack**. En ese momento el desensamblador se para y el desensamblado recursivo de descendencia retorna a la lista de direcciones que han sido separadas para un desensamblado posterior. **Entonces toma una dirección de la lista y el proceso de desensamblado continua desde esta dirección.** Este es el proceso recursivo el cual da nombre al algoritmo.

La principal ventaja del algoritmo recursivo descendiente es la habilidad de distinguir los **valores de código** de los **valores de datos**. Al ser un algoritmo basado en el flujo de control de ejecución, es menos probable el desensamblado incorrecto tanto de datos como de código. Su principal desventaja es la incapacidad para seguir el código de las rutas indirectas, tales como **jumps** y **calls**, los cuales utilizan las tablas de punteros para mirar a sus direcciones objetivo. Sin embargo si le añadimos algunas heurísticas para identificar los punteros al código, los desensamblados recursivos descendientes pueden proporcionarnos una completa cobertura del código y un reconocimiento excelente entre el código y los datos. El listado 1-2 nos muestra una salida de un desensamblado de descendencia recursivo utilizada en la misma declaración de desviación mostrada antes en el listado 1-1

Listado 1-2 Desensamblado descendiente recursivo

```
0040123F   push ebp
00401240   mov  ebp, esp
00401242   xor  eax, eax
00401244   mov  edx, [ebp+arg_0]
00401247   cmp  edx, 0Ch             ; 13 desviaciones posibles
0040124A   ja   loc_4012E0           ; salto por defecto
0040124A                             ; saltará a 00401250 en el caso 0
00401250   jmp  ds:off_401257[edx*4] ; salto de desviación
00401250 ; ---------------------------------------------------
00401257 off_401257:
00401257   dd offset loc_4012E0  ; DATA XREF: sub_40123F+11r
00401257   dd offset loc_40128B  ; jump table para las declaraciones de desviación
00401257   dd offset loc_401290
00401257   dd offset loc_401295
00401257   dd offset loc_40129A
00401257   dd offset loc_4012A2
00401257   dd offset loc_4012AA
00401257   dd offset loc_4012B2
00401257   dd offset loc_4012BA
00401257   dd offset loc_4012C2
00401257   dd offset loc_4012CA
00401257   dd offset loc_4012D2
00401257   dd offset loc_4012DA
0040128B ; ---------------------------------------------------
0040128B
0040128B loc_40128B:             ; CODE XREF: sub_40123F+11j
0040128B                         ; DATA XREF: sub_40123F:off_401257o
0040128B   mov eax, [ebp+arg_4] ; saltará a 00401250 caso 1
0040128E   jmp short loc_4012E0 ; salto por defecto
0040128E                         ; saltará a 00401250 caso 0
```

Performance Bigundill@