

# Técnicas avanzadas de ofuscación de código ejecutable x86

Oscar Delgado

S21sec  
odelgado@s21sec.com

## Resumen

Mucho se ha escrito y trabajado sobre la ofuscación de código fuente, hasta el punto de que esta técnica es imprescindible en cualquier código malicioso moderno. Englobadas dentro de las categorías de polimorfismo, metamorfismo, oligomorfismo u ofuscación del punto de entrada (EPO) podemos encontrar multitud de técnicas. Sin embargo, apenas hay trabajo hecho sobre la ofuscación de código ejecutable. Es decir, dado un ejecutable para I86, del que no disponemos de código fuente, ¿es posible ofuscar el mismo para dificultar al máximo su desensamblado? En este artículo analizamos y presentamos algunas técnicas novedosas para la ofuscación de llamadas a procedimientos (instrucción CALL) y se presenta una serie de ataques efectivos contra el, probablemente, desensamblador más conocido: el IDA Pro.

**Palabras clave:** Ofuscación de código, desensambladores, IDA.

## 1. Introducción

Desde su eclosión a finales de los 80, los virus, troyanos y demás códigos maliciosos han introducido técnicas pioneras en protección y mutación de código a través de la *ofuscación de código fuente*. Técnicas que luego han sido recogidas y utilizadas por el “lado bueno” para protección de programas, rutinas anticopia, marcas de agua, etc... Curiosamente, sin embargo, se ha hecho muy poco trabajo en la ofuscación de código ejecutable. En este artículo estudiaremos esta aproximación y analizaremos las posibilidades que ofrece.

### 1.1. ¿Qué entendemos por ofuscación de código ejecutable?

La *ofuscación* de código ejecutable consiste en utilizar técnicas dirigidas a dificultar en la mayor medida posible la tarea de *desensamblado* y *comprensión* de un binario dado.

En función del objetivo podemos, por tanto, establecer dos grandes meta-categorías:

- técnicas cuya finalidad es complicar, e incluso impedir, un *desensamblado* correcto.
- técnicas que tratan de, una vez obtenido el código fuente del programa, dificultar en lo posible la *comprensión* del mismo.

Las técnicas del primer grupo, potencialmente muy poderosas, son las que van a recibir nuestra atención en este trabajo.

## 2. Técnicas de confusión al desensamblado

El desensamblado consiste en recuperar la secuencia original de instrucciones máquina del programa en una forma inteligible y cómoda para un ser humano (típicamente, en lenguaje ensamblador). Podemos distinguir, básicamente, dos tipos de desensambladores: estáticos y dinámicos. El primero recupera y examina el listado de instrucciones del programa analizando de forma pasiva el código, mientras que el segundo lo ejecuta y monitoriza con alguna herramienta externa (habitualmente, un depurador).

En este artículo nos centraremos en los desensambladores estáticos que, a su vez, pueden ser clasificados en lineales y recursivos según la estrategia que utilicen en su funcionamiento.

### 2.1. Engañando a los desensambladores lineales

Los desensambladores lineales son aquellos que utilizan un método simple de funcionamiento: se sitúan en el primer byte ejecutable del fichero y procesan cada instrucción que van encontrando desde allí hasta el final de la sección de código. Este método es

utilizado, por ejemplo, por la conocida utilidad GNU *objdump*, el desensamblador de NASM y algunas herramientas de optimización para las plataformas Alpha e I32.

### 2.1.1. Autosincronización en el desensamblado

Antes de empezar a explicar ninguna técnica, es necesario entender el fenómeno que se esconde tras este nombre tan rimbombante, producto de la estructura de algún conjunto de instrucciones, sobre todo de la plataforma I32. Cuando se produce un error de desensamblado, por ejemplo porque se estén tratando datos o porque el código no esté alineado<sup>1</sup>, finalmente el proceso acaba resincronizándose rápidamente con el flujo real de instrucciones. El efecto puede observarse en la figura 1, obtenida con el desensamblador de NASM, donde se muestra a la izquierda la secuencia correcta de instrucciones y a la derecha el desensamblado que se obtiene con un desplazamiento erróneo de 1, 2 y 3 bytes respectivamente. Como puede observarse, todos los flujos se resincronizan en un máximo de 4 instrucciones.

Obviamente, el número de instrucciones erróneamente desensambladas depende de la distribución particular de instrucciones del programa. En la práctica, sin embargo, hemos observado que la resincronización se produce rápidamente, siendo raros los casos en los que se producen más de 4 instrucciones incorrectamente desensambladas. Este efecto deberá ser tenido en cuenta a la hora de aplicar técnicas de confusión al desensamblado, como la que analizamos en el siguiente apartado.

### 2.1.2. Inserción de código basura

Esta técnica consiste, básicamente, en producir errores de desensamblado introduciendo bytes sin sentido en lugares donde el desensamblador espera código. Para que esto funcione, estos bytes “basura” deben cumplir dos condiciones:

- Para producir errores reales y que éstos se propaguen, los bytes deben formar instrucciones parciales y no completas.
- Por supuesto, para que el ejecutable siga funcionando correctamente, estos bytes no deben ser alcanzables en tiempo de ejecución.

Como hemos dichos, estos bytes no pueden

<sup>1</sup> El alineamiento consiste en colocar los datos y código de un programa en direcciones de memoria cuyo acceso es “fácil” y eficiente por parte de la CPU.

insertarse en cualquier lugar del código, sino en aquellas zonas donde no exista un flujo de ejecución que las atraviese. En la práctica, las *zonas candidatas* son las inmediatamente posteriores a un salto incondicional o a una instrucción de retorno de función (ya que la ejecución no va a continuar tras ellas).

En el siguiente ejemplo podemos ver un ejemplo práctico. El primer listado es un programa sencillo: básicamente es una llamada a una función que suma 6 al valor almacenado previamente en el registro EDX. El resultado, 9, es impreso por pantalla:

```
00000000 66BA03000000    mov edx,0x3
00000006 E80300        call 0xb
00000009 EB05          jmp short 0x10
0000000B 6683C206      add edx, 0x6
0000000F C3           ret
00000010 B402          mov ah,0x2
00000012 6683C230      add edx, 0x30
00000016 CD21          int 0x21
00000018 CD20          int 0x20
```

Listado 1. Programa sin ofuscar

Las zonas sombreadas indican las zonas candidatas: aquellas instrucciones delante de las cuales se puede insertar código basura. El resultado de aplicar esta transformación es el siguiente:

```
00000000 66BA03000000    mov edx,0x3
00000006 E80300          call 0xc
00000009 EB07          jmp short 0x12
0000000B 836683C2      and word [bp-
0x7d], byte -0x3e
0000000F 06           push es
00000010 C3           ret
00000011 83B4026683    xor word
[si+0x6602],byte -0x7d
00000016 C230CD      ret 0xcd30
00000019 21CD          and bp,cx
0000001B 20           db 0x20
```

Listado 2. Programa ofuscado

En este caso, las instrucciones sombreadas son aquellas que han sido incorrectamente desensambladas. No hay que olvidar que la semántica del programa no ha cambiado y que el resultado de su ejecución sigue siendo exactamente el mismo, a pesar de que su “sintaxis” o su aspecto ha cambiado considerablemente. Si definimos el *factor de confusión* como el tanto por ciento de instrucciones incorrectamente interpretadas,

$$FC = C/I*100$$

donde C es el número de instrucciones correctas e I el de incorrectas, en el ejemplo anterior tenemos un FC del 44.4%; es decir, aproximadamente 4 de cada 10 instrucciones no son desensambladas de forma adecuada. En cualquier caso, en programas más

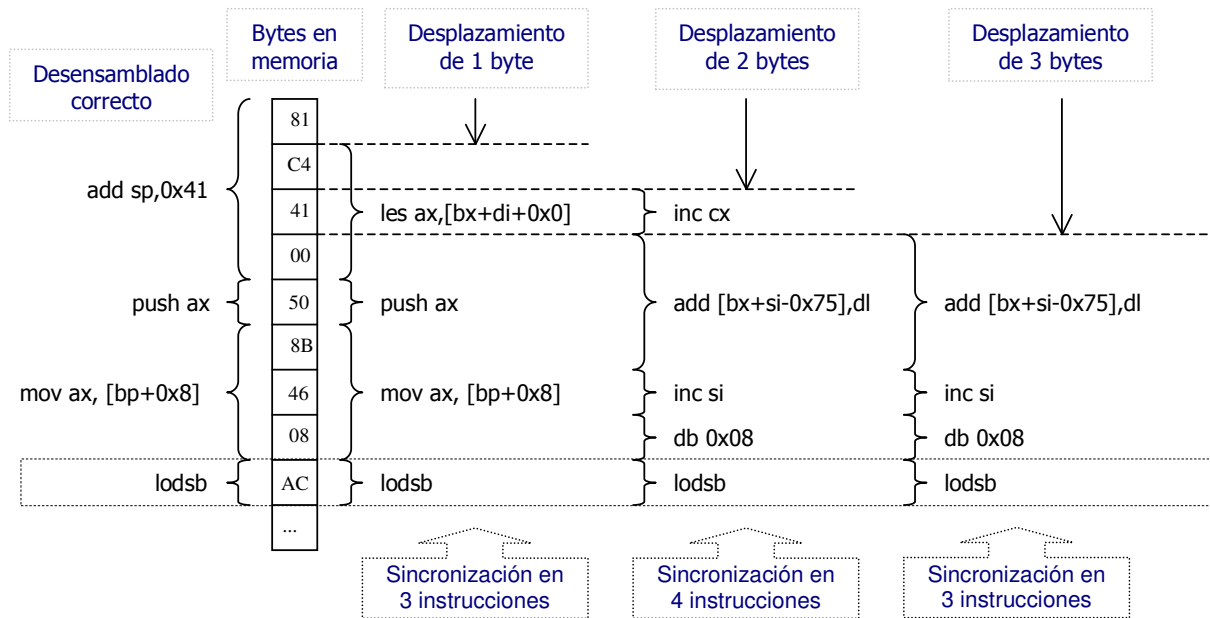


Figura 1. Un ejemplo de desensamblado autosincronizante

realistas (y de mayor longitud) no son habituales FC tan altos: en nuestras pruebas los valores típicos estaban alrededor del 15%.

El siguiente paso tras introducir las instrucciones “basura” es actualizar las referencias a éstas, de manera que la ejecución del código siga siendo correcta y no exista un flujo de ejecución que llegue hasta ellas. Estas modificaciones aparecen recuadradas en el listado<sup>2</sup>.

El primero de los requisitos para los bytes “basura” es que formen instrucciones parciales. Esto quiere decir que debemos utilizar los primeros  $k$  bytes del código de operación de una instrucción cualquiera. Pero el valor concreto sí es importante, debido al efecto de autosincronización que estudiamos con anterioridad. Dependiendo de los valores, la sincronización se producirá antes o después. Una forma de retardar lo máximo posible este efecto es utilizar una instrucción larga (como un XOR de 6 bytes) e ir insertando los  $k$  primeros bytes, variando  $k$  desde 1 hasta 6. Para algún valor de  $k$ , la autosincronización se producirá más tarde (raramente más allá de las 4 instrucciones).

La principal razón de que el FC típico se sitúe alrededor del 15% se debe a la primera de las

restricciones que los bytes basura deben cumplir: el bloque precedente debe acabar en un salto incondicional. Hemos encontrado que, en un programa típico compilado con las opciones de optimización habituales, las zonas candidatas suelen situarse cada 30 instrucciones aproximadamente. Esta distancia, junto con el efecto de autosincronización, provoca que el flujo de ejecución consiga sincronizarse antes de encontrar la siguiente zona de “basura”. Para incrementar el número de estas zonas, podemos utilizar una técnica como la siguiente. La idea es invertir el sentido de los saltos condicionales, de manera que convirtamos las instrucciones de este tipo en una combinación de un salto condicional y uno incondicional. Por ejemplo, imaginemos el siguiente salto:

```
jz 0x776
```

El truco consiste en sustituir esta instrucción por:

```
jnz L
jmp 0x776
L:...
```

Claramente, la semántica no ha cambiado. Pero ahora hemos introducido un salto incondicional, que podremos utilizar para insertar instrucciones basura. De esta forma, podemos reducir la distancia típica de las zonas candidatas a 12 instrucciones y subir el FC al 40% aproximadamente.

<sup>2</sup> Para estudiar estas técnicas los autores han desarrollado una aplicación, aún en fase beta, que aplica estas transformaciones de forma automática.

## 2.2. Algo más difícil: a por IDA Pro

Los desensambladores recursivos, a diferencia de los lineales, poseen “inteligencia”: siguen los posibles flujos de ejecución y, por tanto, pueden desembarazarse fácilmente de datos introducidos en la sección de código y de las técnicas que acabamos de ver. Aunque esta habilidad también puede volverse en su contra y ser explotada.

Concretamente, cuando encuentra una transferencia de control IDA continúa el desensamblado en aquellas posiciones que probablemente sean el objetivo del salto. Así, asume que las estructuras de transferencias de control comunes (saltos, llamadas a funciones, etc...) se comportan siempre de igual manera y “razonablemente”. Por ejemplo, se supone que la llamada a una función salta a una posición de memoria y luego continúa la ejecución en la línea siguiente a la instrucción CALL. Pero, ¿qué ocurre si conseguimos que los saltos no se compartan como se espera que lo hagan?. Vamos a verlo.

### 2.2.1. Funciones salto

La idea de esta técnica es sencilla: se trata de conseguir que instrucciones en las que confía el desensamblador se comporten de forma “extraña”. Para ello podemos utilizar las llamadas *funciones salto*. Una función de este tipo es una función que, cuando es llamada desde una posición  $a_i$ , transfiere el control a una posición  $b_i$  correspondiente. Así, dada una función salto  $f$ , podemos sustituir los saltos incondicionales de un programa:

```
a1 : jmp b1
...
a2 : jmp b2
...
an : jmp bn
```

por llamadas a la función salto:

```
a1 : call f
...
a2 : call f
...
an : call f
```

Cuando es llamada, la función salto determina la posición  $b_i$  a la que debe saltar basándose en la posición  $a_i$  desde donde es llamada. Además, es

importante que esto lo lleve a cabo de tal forma que la ejecución continúe en  $b_i$  tal y como continuaría si hubiera llegado allí a través de un salto incondicional.

Podemos pensar en dos propósitos para estas funciones:

- El primero es claro: comprender y seguir el flujo de ejecución de un programa que las utilice es mucho más complicado. Sólo la imaginación (y el rendimiento) limita lo enrevesadas que pueden escribirse estas funciones.
- Pero el objetivo principal es otro: provocar errores de desensamblado introduciendo bytes basura inmediatamente después de cada *call f*, de forma similar a como hicimos en la sección 2.1.2.

La implementación de estas funciones puede llevarse a cabo de distintas formas. Una primera estrategia podría utilizar una tabla para determinar la dirección de salto. Pero, evidentemente, esto es fácil de “romper”. Otra aproximación, más sofisticada, sería pasar el desplazamiento entre la instrucción siguiente al CALL y el objetivo  $b_i$  como argumento en la pila a la función salto. Ésta sólo tendría entonces que sumar este desplazamiento a su dirección de retorno almacenada en la pila, que sería la dirección de  $b_i$ . El código que para realizar esto en I32 podría ser algo parecido a esto:

```
xchg eax, [esp]      (I1)
pushf                (I2)
add [esp+8], eax     (I3)
popf                 (I4)
pop eax              (I5)
ret                  (I6)
```

La instrucción 1 coloca en el registro *eax* la dirección de retorno de la función salto y, al mismo tiempo, salva el valor de éste en la pila. I2, por su parte, guarda el valor de los *flags* en la pila, que son modificados por la instrucción ADD, para luego restaurarlos en I4. I3 lleva a cabo la suma del desplazamiento pasado como argumento y de la dirección de retorno y deja el resultado en la pila. I5 restaura el valor de *eax* previamente almacenado en la pila por I1. I6 desapila la dirección de  $b_i$  y salta a su posición de memoria.

<pre> ... 08048350 68 11 00 00 00    push dword 0xc 08048355 E8 01 00 00 00    call 0x0804835B  0804835A 86 87 04 24 9C 01  xchg al, [edi+0x19c2404] 08048360 44                inc esp 08048361 24 08            and al, 0x8 08048363 9D                popf 08048364 58                pop eax 08048365 C3                ret  08048366 E9 01 00 00 00    jmp 0x36c 0804836B 86 B8 01 00 00 00  xchg bh, [eax+0x1] 08048371 BB 00 00 00 00    mov ebx, 0x0 08048376 CD 80            int 0x80 </pre>	<pre> ... 08048350 68 11 00 00 00    push 0Ch 08048355 E8 01 00 00 00    call \$+6  0804835A 86 87 04 24 9C 01  xchg al, [edi+19C2404h] 08048360 44                inc esp 08048361 24 08            and al, 8 08048363 9D                popf 08048364 58                pop eax 08048365 C3                retn  08048366 E9 01 00 00 00    jmp loc_804836C 0804836B 86                db 86h 08048371 B8 01 00 00 00    mov eax, 1 08048376 BB 00 00 00 00    mov ebx, 0 08048376 CD 80            int 80h </pre>
--	--

Desensamblado con NASM (FC=34%)

Desensamblado con IDA Pro (FC=21%)

Figura 2. Efectos de esta técnica en NASM e IDA Pro

Es importante destacar que ninguna instrucción de la función salto debe modificar el valor de los *flags* ni de ningún registro, pues esto podría provocar un mal funcionamiento en el programa ofuscado.

Vamos a comprobar ahora como esta técnica simple funciona muy bien en un desensamblador comercial como IDA Pro. Partiendo del listado siguiente, sin ofuscar, aplicamos las técnicas que acabamos de describir, inserción de código basura y funciones salto. El resultado se muestra en la figura 2, donde el byte insertado es 86h, recuadrado.

```

;equivalentes a un 'jmp 0x08048365'
08048350 6810000000    push dword 0x10
08048355 E800000000    call 0x0804835A

;función salto
0804835A 870424        xchg eax, [esp]
0804835D 9C            pushf
0804835E 01442408     add [esp+0x8], eax
08048362 9D            popf
08048363 58            pop eax
08048364 C3            ret

;salto una instrucción más allá
08048365 E900000000    jmp 0x0804836A

;termina el programa
0804836A B801000000    mov eax, 0x1
0804836F BB00000000    mov ebx, 0x0
08048374 CD80            int 0x80

```

Listado 3. Listado original sin ofuscar

Podemos destacar un par de resultados:

- Como era de esperar, el desensamblador lineal no resuelve correctamente los bytes basura y las funciones salto, siendo confundido por

ambas técnicas. IDA, sin embargo, es capaz de detectar y corregir la primera.

- Sin embargo, IDA desensambla incorrectamente las instrucciones entre las posiciones 0804835A y 08048361, las siguientes a la llamada *call*. Como hemos visto, esto es debido a su confianza a que esta instrucción siempre retornará y que, por tanto, tras ella siempre encontrará código ejecutable válido.

De nuevo, el efecto de autosincronización evita que las instrucciones “basura” se propaguen más allá de, en este caso, 3 instrucciones. Sin embargo, aún no hemos agotado todas las posibilidades: todavía nos queda un as en la manga...

### 2.2.2. Modificación de instrucciones CALL

En nuestra búsqueda de más candidatos donde insertar instrucciones basura, podemos utilizar una variación del esquema de funciones saltos que hemos visto. De esta forma, podríamos utilizar también instrucciones CALL “normales”, y no aquellas que hemos creado artificialmente sustituyendo a instrucciones JMP como en el punto anterior, para insertar basura tras ellas.

Evidentemente, de momento no podemos insertar basura directamente detrás de la instrucción CALL, pues violaríamos una de las condiciones que impusimos en la sección 2.1.2 (a parte de que obtendríamos un bonito mensaje de *Segmentation Fault* o *Illegal Instruction*). Pero podemos “maquillar” estas instrucciones creando una función *wrapper* que reciba dos parámetros: la dirección de la función

original a la que hay que saltar y la cantidad de basura que se ha insertado tras la instrucción CALL. De esta forma, una llamada CALL normal se transformaría en:

```

call f                (I1)
instruccionSiguiente (I2)

push offset f        (I1')
push 2               (I2')
call <wrapper_function> (I3')
<1 byte basura>     (I4')
<1 byte basura>     (I5')
instruccionSiguiente (I6')

```

El código de la función *wrapper* sería similar a la función salto que ya hemos visto. Recuperaría de la pila los argumentos, realizando una llamada CALL (que no podría ofuscarse) a la función original. Tras volver de esta llamada, sumaría la cantidad de bytes basura a la dirección de retorno original para llegar hasta la instrucción I6', evitando la zona inútil entre I4' e I5'.

Esta técnica tiene un coste evidente en incremento de tamaño de código. Este incremento (en bytes) es de  $7 + n^{\circ} \text{ bytes\_basura}$  por cada instrucción CALL, que no es excesivo si consideramos que, en un programa normal, el número de instrucciones de este tipo supone aproximadamente el 8% del número total.

La ventaja de esta técnica es que resulta difícil de detectar. Si se hace una elección cuidadosa de las instrucciones basura que se insertan tras las llamadas CALL, es difícil determinar si se trata de basura o no. Un hecho que podría dar una pista es que todas las instrucciones CALL saltan a la misma posición (la de la función salto). Para mitigar esto, pueden crearse varias funciones saltos, de distintas complejidades, ubicadas en posiciones diferentes, y utilizarlas de forma aleatoria.

### 2.2.3. Predicados opacos

Otra de las suposiciones hechas por IDA que podemos explotar es el hecho de que un salto condicional tiene dos posibles objetivos. “Disfrazando” un salto condicional de forma que, en tiempo de ejecución, siempre salte al mismo lugar tenemos, en la práctica, un salto incondicional. De esta forma, podemos aprovechar dos situaciones: un salto que nunca es realizado y el flujo de ejecución siempre continúa a través de él y un salto que siempre es realizado y la ejecución nunca continúa.

Este concepto de utilizar condiciones que siempre se evalúan a un valor verdadero o falso no es nuevo. Se

conocen como predicados opacos (*opaque predicates*), y han sido ampliamente estudiados en otros campos [Cohen]. Lo novedoso en este caso es aplicarlos al problema que estamos estudiando.

Una vez que hemos conseguido transformar un salto incondicional en uno condicional con el uso de un predicado opaco, podemos aplicar la técnica de inserción de basura que hemos ido desarrollando a lo largo del artículo. El resultado es que tenemos una posición, bien el objetivo del salto, bien la instrucción siguiente a la condición, dependiendo si el predicado es siempre verdadero o siempre falso, que parece ser una continuación legítima de la ejecución tras el salto condicional pero que, en realidad, no lo es.

Mucho se ha escrito y estudiado sobre los predicados opacos, principalmente para uso en ofuscación de código fuente. Su uso es también frecuente en los ofuscadores comerciales de código (Java habitualmente), aunque suelen incluir transformaciones tan simples como la siguiente<sup>3</sup>:

```

int v, a=5, b=6;

v = a+b;
if (v>5){
  <<La ejecución siempre continúa por aquí>>
}else{
  <<Nunca se alcanza en tiempo de ejecución>>
  <<Es posible insertar basura aquí>>
}

```

Otros, sin embargo, son más complicados. ¿Podría decir, a simple vista, si el siguiente predicado es siempre verdadero o falso?:

$$((q + q^2) \bmod 2) = 0$$

Bastan unas cuantas pruebas para llegar a la conclusión de que el resultado es siempre verdadero. Pero, ¿podría hacer lo mismo analizando el código máquina desensamblado que lleva a cabo la operación, sin saber de antemano que es un predicado opaco?.

#### *Ejemplo de predicado opaco en código ejecutable*

El siguiente código en C implementa el predicado opaco anterior:

```

#include <stdio.h>
#include <math.h>

int main(void){

```

<sup>3</sup> De hecho, este predicado es eliminado por el compilador gcc si se utiliza optimización (opción O3). Sin optimización sí se mantiene.

```

double v, q=3;

v = fmod(q+pow (q, 2), 2);

if (v>5){
    printf("true");
}else{
    printf("false");
}
return 1;
}

```

Y éste es el desensamblado de parte del “if” y de las ramas TRUE y FALSE:

```

0040102D jnz      short loc_401042
0040102F push     offset aTrue
00401034 call    _printf
00401039 add     esp, 4
0040103C mov     eax, 1
00401041 retn
00401042 loc_401042:
00401042 push     offset aFalse
00401047 call    _printf
0040104C add     esp, 4
0040104F mov     eax, 1
00401054 retn

```

} Rama TRUE

} Rama FALSE

La rama TRUE nunca se va a alcanzar, así que puede incluirse cualquier código en él. Incluyendo simplemente un byte 83h (parte del código de operación de la instrucción SUB) en la posición 00401041 puede conseguirse que IDA desensamble incorrectamente toda la rama FALSE:

```

0040102D jnz      short near ptr loc_401041+1
0040102F push     offset unk_40A068
00401034 call    _printf
00401039 add     esp, 4
0040103C mov     eax, 1
00401041 loc_401041:
00401041 sub     dword ptr [eax+30h], 0FFFFFFA0h
00401045 inc     eax
00401046 add     al, ch
00401048 adc     al, 0
0040104A dd     0C4830000h
0040104E dd     1B804h, 90C30000h, 2 dup(90909090h)

```

} Rama TRUE

} Rama FALSE

Algunas consideraciones sobre el ejemplo que acabamos de ver:

1. La cantidad de bytes basura que se pueden insertar no tiene, en principio, límite. La rama puede hacerse de cualquier longitud deseada.
2. En las secciones muertas no es necesario incluir códigos de operación parciales. Pueden ser instrucciones válidas, inválidas, con sentido o sin él, pues se tiene la seguridad de que la ejecución nunca llegará allí. Eso sí, para que IDA desensamble incorrectamente también la rama FALSE (o TRUE), puede incluirse un código de operación

(preferiblemente alguno de 3 o 4 bytes) justo en el byte anterior a esa rama, de manera que la autosincronización se retarde lo más posible.

3. Esta técnica, como cualquier otra, no es infalible. Puede detectarse analizando el salto de la posición 0040102D. Se observa que su objetivo es una posición no alineada, lo que puede dar una pista sobre que el desensamblado podría no ser correcto. Una vez detectado esto, el listado correcto puede obtenerse utilizando los comandos *Undefine* y *MakeCode* de IDA desde la posición correcta, 00401042.

### 2.2.5. Saltos dinámicos

Esta técnica, utilizada desde hace tiempo en virus, consiste en ocultar el destino real de un salto incondicional haciendo que éste sea calculado en tiempo de ejecución. Es decir, se trata de pasar de instrucciones del tipo

```
jmp 0x24048ae5
```

a una como

```
jmp dword ptr [100050F0+eax*4]
```

Evidentemente, la segunda es mucho más difícil de entender y seguir en un desensamblado estático. El cálculo de la dirección puede hacerse arbitrariamente complejo. Hay mucha y buena literatura al respecto, como [Zombie], por lo que no entraremos en detalles aquí.

Esta técnica no es estrictamente del tipo que hemos estado viendo (técnicas nuevas de modificación de código ejecutable), pero se ha incluido aquí porque puede ser llevada a cabo por BEA.

### 2.2.6. Binary Executable Anti-disassembler

BEA (*Binary Executable Anti-disassembler*) es la herramienta desarrollada por los autores como prueba de concepto y estudio de las técnicas presentadas en este artículo. Se encuentra aún en una fase beta, muy preliminar, pero funcional. Si el desarrollo continúa o se detecta interés en la comunidad, se liberará bajo licencia GPL.

### 2.2.7. Combinación de técnicas

Por supuesto, todas estas técnicas se pueden

combinar para mejorar su eficacia individual. Por ejemplo, la segunda rama de un predicado opaco (que, imaginemos, es la que nunca se toma) podría ofuscarse insertando basura y la primera con un salto dinámico.

El resultado es que, para un desensamblador, el código inválido resulte indistinguible del válido, y no cuenta con ninguna pista (excepto la pericia humana) para encontrarlo.

### 3. Análisis final

#### 3.1. Impacto en tamaño y rendimiento

Estas técnicas, evidentemente, tienen un incremento en el tamaño y velocidad de ejecución del binario ofuscado. A continuación se muestra el resultado de aplicar las técnicas de inserción de basura y modificación de instrucciones CALL.

	Aplicación	Líneas candidatas	Líneas totales	+Δ tamaño	Media
UNIX	gcc	1877	29001	6'5%	+8'4%
	perl	27308	320545	8'5%	
	ethereal	33482	326724	10'2%	
Window	winhlp32.exe	10718	99009	10'8%	+8'7%
	explorer.exe	6138	102728	5'9%	
	wininet.dll	21150	218355	9'6%	

Tabla 1. Incremento de tamaño de los ejecutables ofuscados

Como vemos, el incremento de tamaño se sitúa alrededor de un 9%, porcentaje bajo que nos parece asumible.

#### 3.1. Impacto en velocidad de ejecución

En este caso, medimos el incremento en la velocidad de ejecución debido a la ofuscación. Teniendo en cuenta que la función salto mostrada en la sección 2.2.1 ejecuta en exactamente 0'1 microsegundos<sup>4</sup>, tenemos los siguientes tiempos:

	Aplicación	Funciones salto		Modificación instrucc. CALL		Inc. total
		Número func.	+Δ ms	Número instrucc.	+Δ ms	
UNIX	Gcc	735	0,066	1017	0'091	0,157
	Perl	11763	1,058	13199	1'187	2,245
	Ethereal	6388	0'574	24967	2'247	2,821
Window	winhlp32.exe	2409	0,216	7193	0,647	0,863
	explorer.exe	1565	0,140	3630	0,326	0,466
	wininet.dll	6007	0,540	12396	1,115	1,655

<sup>4</sup> Resultados obtenidos en una CPU Pentium IV con reloj de 2,4Ghz y 512 Mb de RAM.

Tabla 2. Incremento de la velocidad de ejecución

En este caso, el incremento en tiempo de ejecución puede considerarse despreciable y, en ningún caso, apreciable por el usuario.

## 4. Resumen

Las técnicas de ofuscación y deofuscación no han dejado de evolucionar y mejorar en los últimos años. En este artículo hemos presentado una clase de técnicas novedosas dirigidas a binarios ejecutables efectivas en gran medida contra IDA y todos los desensambladores lineales.

La siguiente tabla muestra un resumen de estas técnicas y su efectividad contra distintos desensambladores (✓ = técnica efectiva, ✗ = técnica inefectiva):

	Objdump	NASM	IDA
Bytes basura	✓	✓	✗
Modificación instrucciones CALL	✓	✓	✓
Predicados opacos	✓	✓	✓

Tabla 3. Efecto de las técnicas en cada desensamblador

## Agradecimientos

A Giovanni Vigna por sus valiosos comentarios y a todos mis chic@s del SMS por aguantarme.

## Referencias

- [Collberg 97] C. Collberg, C. Thomborson, D. Low. *A taxonomy of obfuscating transformations*, 1997.
- [Cohen 92] F. B. Cohen. *Operating system protection through program evolution*, 1992. <http://all.net/books/IP/evolve.html>.
- [Lifshits] Yury Lifshits. *Introduction to Program Obfuscation*. <http://logic.pdmi.ras.ru/~yural>
- [Roo et al.] Arjan de Roo, Leon van den Oord. *Stealthy obfuscation techniques: misleading the pirates*.
- [Zombie] Automated reverse engineering: Mistfall engine. <http://vx.netlux.org/lib/vzo21.html>