U23 - Shellcode

Twix

Chaos Computer Club Cologne e.V.
http://koeln.ccc.de

2016-11-28

Überblick

Was ist Shellcode?

- Shellcode ist eine Sequenz von Befehlen, die eine Shell öffnet.
- Wir haben eine Möglichkeit gefunden, Daten in einen ausführbaren Bereich eines Programms einzuschleusen und auszuführen.

Beispiel eines verwundbaren Programms

```
1  int main(int argc, char **argv)
2  {
3          char buffer[100];
4          strcpy(buffer,argv[1]);
5          return 0;
6  }
```

- Warum ist dieses Programm anfällig?
  - strcpy() prüft Bufferlänge nicht
  - $\Rightarrow$ Rücksprungadresse überschreiben
- Welche Vorbedingungen brauchen wir?
  - ausführbarer Stack
  - bekanntes Adresslayout

Erzeugen von Shellcode

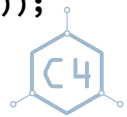
1. Anweisungen in Assembler schreiben
2. Anweisungen assemblieren
3. Assemblierten Code aus der Objektdatei kopieren
4. In escapte Darstellung überführen

Python ist gut geeignet um diesen Vorgang zu scripten. Auch Makefiles vereinfachen das.

```
$ as -o example.elf example.S
$ objcopy -O binary
↪  example.elf example.bin
$ hexdump -ve '1/1 "\\x%.2x"'
↪  example.bin
\xc8\x00\x00\x00\x31\xc0\xc9\xc3
```

Testen von Shellcode

```
1  #include <string.h>
2  #include <sys/mman.h>
3  const char shellcode[] = "\xc8\x00\x00\x00\x31\xc0\xc9\xc3";
4  int main(void) {
5          void (*executable_memory)(void) = mmap(
6                  NULL, sizeof(shellcode),
7                  PROT_READ | PROT_WRITE | PROT_EXEC,
8                  MAP_ANONYMOUS | MAP_PRIVATE, 0, 0
9          );
10         memcpy(executable_memory, shellcode, sizeof(shellcode));
11         executable_memory();
12         return 0;
13 }
```



Was sind Syscalls?

- Einige Aktionen darf nur das Betriebssystem ausführen, z.B. die Hardware steuern, Input/Output
- Unsere Software läuft in der Regel im Userspace, muss also das Betriebssystem ansprechen
- Das geschieht mittels sogenannten Syscalls
- Syscalls lösen einen Softwareinterrupt aus, der vom Betriebssystem gefangen und behandelt wird

Linux Syscall Calling Convention

- Jedem Syscall ist eine Nummer zugeordnet, die auf i386 in eax abgelegt wird
- Die Argumente der Syscalls liegen jeweils in einem bestimmtem Register
- Der Rückgabewert liegt in eax, wenn die Kontrolle an das Programm zurückgegeben wird
- Auf i386 wird der Syscall durch die Instruktion `int 0x80` ausgelöst
- Kernel stellt Register wiederher

Linux Syscall Calling Convention

Syscall Nr.	Arg. 1	Arg. 2	Arg. 3	Arg. 4	Arg. 5	Arg. 6	Rückgabewert
eax	ebx	ecx	edx	esi	edi	ebp	eax

Der write() Syscall

```
ssize_t write(int fd, const void *buf, size_t count);
```

- Syscall mit der Nummer 4
- write() schreibt ein char-Array ‘buf’ der Länge ‘count’ in einen Datenstrom, referenziert über eine Dateidiskriptornummer ‘fd’
- Immer vorhanden sind die Datenströme stdin(Standardeingabe) 0, stdout(Standardausgabe) 1 und stderr(Standardfehlerausgabe) 2
- Rückgabewert: Anzahl der Bytes die geschrieben wurden, im Fehlerfall -1

Beispiel mit write()

```
1  .data
2  hlo_str:
3  .string
     ↪  "Hello
     ↪  World!\n"
4  .text
5  .global main
6  main:
7  enter $0,$0
8  push %ebx
9  mov $13, %edx
10 mov $hlo_str,
     ↪  %ecx
11 mov $1, %ebx
12 mov $4, %eax
13 int $0x80
14 pop %ebx
15 leave
16 ret
```

- Länge des Strings(13) ⇒ edx
- Pointer auf den String ⇒ ecx
- Datenstrom(stdout,1) ⇒ ebx
- Syscallnummer(write,4) ⇒ eax
- Softwareinterrupt Nummer 0x80 auslösen

Der read() Syscall

```
ssize_t read(int fd, void *buf, size_t count);
```

- Syscall mit der Nummer 3
- Liest von einem Datenstrom ‘fd‘ in ein char-Array ‘buf‘ bis zu ‘count‘ Bytes
- Rückgabewert: Anzahl gelesener Bytes, im Fehlerfall -1

Beispiel mit read() und write() und Nutzung des Rückgabewerts

```
1  .text
2  .global main
3  main:
4  enter $0,$0
5  push %ebx
6  sub $32, %esp
7  mov $32, %edx
8  mov %esp, %ecx
9  mov $0, %ebx
10 mov $3, %eax
11 int $0x80
12 mov %eax, %edx
13 mov $1, %ebx
14 mov $4, %eax
15 int $0x80
16 pop %ebx
17 leave
18 ret
```

Aufgaben

- Ohne die Libc zu benutzen: Schreibt ein Programm, dass nach einem Namen fragt und dann “Hallo <name>” ausgibt.

Der fork() Syscall

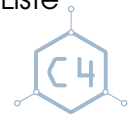
```
pid_t fork(void);
```

- Syscall mit der Nummer 2
- fork() "kopiert" den aktuellen Prozess, erzeugt also einen neuen Childprocess
- Rückgabewert: 0 im Childprocess, die PID des neuen Prozesses im Parentprocess, -1 im Fehlerfall

Der execve() Syscall

```
int execve(const char *filename, char *const argv[], char *const
envp[]);
```

- Syscall mit der Nummer 11
- execve() ersetzt das Programm des aktuell laufenden Prozesses durch ein anderes Programm
- ‘filename‘ ist der Dateiname des zu ladenen Programmes, ‘argv‘ die Null-terminierte Liste der Argumente und ‘envp‘ die Null-terminierte Liste mit den Environmentvariablen
- Rückgabewert: -1 im Fehlerfall



Der exit() Syscall

```
void _exit(int status);
```

- Syscall mit der Nummer 1
- Beendet das laufende Programm mit dem Rückgabewert ‘status’

Der open() Syscall

```
int open(const char *pathname, int flags);
```

- Syscall mit der Nummer 5
- Öffnet eine Datei ‘pathname’
- Rückgabewert: Nummer des Filediskriptors im Erfolgsfall, -1 im Fehlerfall

Der close() Syscall

```
int close(int fd);
```

- Syscall mit der Nummer 6
- Schließt eine geöffnete Datei mit dem Filediskriptor ‘fd’
- Rückgabewert: 0 im Erfolgsfall, -1 im Fehlerfall

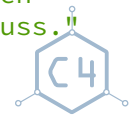
Wie kommt man eigentlich an die Adresse von einem String?

- Wo liegt denn hier das Problem?
- Ideen?

Lösung

- Wir wissen wo der String relativ im Shellcode liegt ⇒ relativer Jump vor den String
- Dann ein call zurück, die Adresse des Strings wird auf den Stack gepusht
- Mit einem Pop kommt die Adresse des Strings in ein Register

```
1  .text
2  jmp my_string
3  back_to_my_code:
4  pop %eax
5  ... do stuff
6  my_string:
7  call back_to_my_code
8  .string "Mein String, an den
    ↪  ich unbedingt kommen muss."
```



Beispiel: Hello World!

```
1  .text
2  enter $0, $0
3  push %ebx
4  jmp hello_str
5  begin:
6  pop %ecx
7  mov $13, %edx
8  mov $1, %ebx
9  mov $4, %eax
10 int $0x80
11 pop %ebx
12 leave
13 ret
14 hello_str:
15 call begin
16 .string "Hello World!\n"
```

Aufgaben

1. Compiliert das Hello World Beispiel und führt es in der Testbench aus.
2. Schreibt einen Shellcode der eine Shell öffnet, also z.B. /bin/sh ausführt, und führt ihn in der Testbench aus.

Das Beispiel vom Beginn

```
1 int main(int argc, char **argv)
2 {
3         char buffer[100];
4         strcpy(buffer,
   ↪  argv[1]);
5         return 0;
6 }
```

- Welches Problem haben wir hier?
- Würde euer Shellcode von vorhin hier einfach funktionieren?
- Ideen?

Erster Schritt: Nullbyte freie Instruktionen

- Einige Instruktionen oder deren Argumente enthalten Nullbytes. z.B. `enter $0, $0` oder `mov $0, %eax`.
- Statt `enter $0, $0` ⇒ `push %ebp` und `mov %esp, %ebp` benutzen
- Statt `mov $0, %eax` ⇒ `xor %eax, %eax` benutzen

Zweiter Schritt: Nullbyte freie Strings

- Ein Nullbyte haben wir immer noch: den Nulldelimiter von unserem String
- Das stört, wenn wir dahinter z.B. noch eine Rücksprungadresse überschreiben
- Die Lösung ist einfach: Nulldelimiter durch einen Platzhalter ersetzen, während der Ausführung wieder durch Null ersetzen

Beispiel: Nullbyte freie Strings

```
1  push %ebp
2  mov %esp, %ebp
3  push %ebx
4  jmp filename_str
5  begin:
6  pop %ebx
7  xor %eax, %eax
8  xor %ecx, %ecx
9  movb %cl, 8(%ebx)
10 movb $0100, %cl
11 movb $5, %al
12 int $0x80
13 pop %ebx
14 leave
15 ret
16 filename_str:
17 call begin
18 .string "test.txt"
```

NOP Slides

- Oft schwer, exakt eine Adresse zu treffen
- Abhilfe können NOP Slides schaffen: Eine Folge von 1 Byte langen NOP Anweisungen vor den eigentlichen Shellcode schreiben
- Opcode von NOP: 0x90
- Wird interpretiert als `xchg %eax, %eax`

Aufgaben

1. Macht euren Shellcode Nullbyte frei.
2. Probiert euren Shellcode an der Bufferoverflow Challenge mit NOP Slide vom letzten Termin aus.