

Praktikumsbericht: Einführung in Python

Marcus Ganske, 36603

Lukas Krieg, 53506

6. Juni 2017



Hochschule Aalen

Inhaltsverzeichnis

1	Einleitung	3
2	Aufgabe 1: Codeanalyse & Reverse Engineering	4
2.1	Codeanalyse	4
2.2	Reverse Engineering mit gdb	6
2.2.1	Wie werden die Übergabeparameter an die Funktion execve() übergeben?	6
2.2.2	Wie und an welcher Stelle werden die übergebenen Parameter im Speicher abgelegt?	8
2.3	Welche Daten befinden sich beim Aufruf von execve() im Stack Frame?	10

1 Einleitung

```
Code/  
├─ Aufgabe1.c  
├─ shellcode.asm  
├─ shellcode.bin  
├─ shellcode.o  
└─ shellcode.py
```

Abbildung 1: Aufbau des Ordners Code mit allen Dateien

2 Aufgabe 1: Codeanalyse & Reverse Engineering

In diesem Kapitel wird auf Aufgabe 1 des Praktikums eingegangen. Es sollte der folgende Code implementiert und analysiert werden.

```
1 include <unistd.h>
2 int main() {
3     char *cmd[] = {"/bin/sh", "-c", "ls-l", (char*)0 };
4     int ret;
5     ret = execve(cmd[0], cmd, NULL);
6     return ret;
7 }
```

Der Quellcode des C Programms wurde in der Datei unistd.c implementiert. Diese Datei befindet sich im verzeichnis Code.

2.1 Codeanalyse

Beschreiben sie die Funktionsweise des Codes

Das C-Programm initialisiert in Zeile 3 ein String Array mit Variablennamen cmd. Der Inhalt der Strings ist:

```
1 "/bin/sh"
2 "-c"
3 "ls_l-l"
4 (char *)0 // NULL Pointer Value von Typ char.
```

Danach wird in Zeile 4 ein integer mit Variablennamen "ret" initialisiert. Die Funktion execve wird aufgerufen mit den Übergabeparametern.

```
1 cmd[0]    -> "/bin/sh"
2 cmd       -> "/bin/sh", "-c", "ls_l-l", NULL Pointer Value
3 NULL
```

Funktionsaufruf Execve

Die Funktion Execve führt Programme aus, auf welche mit einem Dateipfad gezeigt wird. In diesem Fall handelt es sich um das erste Übergabeparameter cmd[0] worin sich der Pfad "/bin/bash" befindet. Es wird eine Shell gestartet.

Das zweite Übergabeparameter von Execve muss ein Array aus Argumentstrings sein die an das neue Programm übergeben werden. In diesem Fall:

```
1 cmd = {"/bin/sh", "-c", "ls -l", (char*)0}
```

das erste Argument `"/bin/sh"` gibt den Interpreter an das zweite Argument `"-c"` bedeutet, dass das folgende command ausgeführt werden soll das dritte Argument `"ls -l"` wird durch die option `"-c"` ausgeführt das vierte Argument `(char*) 0` ist ein Nullpointer, der benötigt wird um das Ende des Arrays für `execve` ersichtlich zu machen

Das dritte Übergabeparameter von `execve` ist `"NULL"`. Hier wird ein Stringarray mit zusätzlichen Variablen erwartet, die an das neue Programm weitergereicht werden können. Dieses Parameter ist genau wie das Zweite nullterminiert. Deswegen wird hier einfach `NULL` übergeben.

Der Returnwert wird in die integervariable `"ret"` geschrieben. Bei Erfolg gibt es bei `execve` allerdings keinen Rückgabewert. Bei einem Fehler wird allerdings `-1` zurückgegeben.

2.2 Reverse Engineering mit gdb

2.2.1 Wie werden die Übergabeparameter an die Funktion `execve()` übergeben?

Die Übergabeparameter werden in die Register `rdx,rdi,rsi` gespeichert

```
0x0000555555554758 <+72>:    mov     edx,0x0
0x000055555555475d <+77>:    mov     rsi,rcx
0x0000555555554760 <+80>:    mov     rdi,rax
=> 0x0000555555554763 <+83>:    call    0x5555555545c8
0x0000555555554768 <+88>:    mov     DWORD PTR [rbp-0x34],eax
0x000055555555476b <+91>:    mov     eax,DWORD PTR [rbp-0x34]
0x000055555555476e <+94>:    mov     rdx,QWORD PTR [rbp-0x8]
0x0000555555554772 <+98>:    xor     rdx,QWORD PTR fs:0x28
0x000055555555477b <+107>:   je      0x555555554782 <main+114>
0x000055555555477d <+109>:   call    0x5555555545c0
0x0000555555554782 <+114>:   leave
0x0000555555554783 <+115>:   ret
End of assembler dump.
(gdb) info registers
rax                0x555555554814      93824992233492
rbx                0x0              0
rcx                0x7fffffffde20    140737488346656
rdx                0x0              0
rsi                0x7fffffffde20    140737488346656
rdi                0x555555554814      93824992233492
rbp                0x7fffffffde50     0x7fffffffde50
rsp                0x7fffffffde10     0x7fffffffde10
r8                 0x555555554800      93824992233472
r9                 0x7ffff7de8bd0     140737351945168
r10                0x0              0
r11                0x1              1
r12                0x5555555545c0      93824992233000
```

Abbildung 2: Aufgabe 1a Inhalt Register vor Call `Execve()`

Im Register rdi befindet sich durch Zeile

```
1 0x000055555554760 <+80>:    mov rdi, rax
```

die Adresse 0x55555554814 in der sich der String `"/bin/sh"` befindet. (siehe Abbildung 4)

```
1 0x00005555555475d <+77>:    mov rsi, rcx
```

Im Register rsi befindet sich die Adresse des ersten Strings unseres Stringarrays `cmd`. Die Adresse lautet 0x7fffffffde20. Diese Adresse liegt im Stack. Der Stack sieht vor dem Aufruf der Funktion folgendermaßen aus.

```
=> 0x000055555554763 <+83>:    call 0x555555545c8
0x000055555554768 <+88>:    mov  DWORD PTR [rbp-0x34],eax
0x00005555555476b <+91>:    mov  eax,DWORD PTR [rbp-0x34]
0x00005555555476e <+94>:    mov  rdx,QWORD PTR [rbp-0x8]
0x000055555554772 <+98>:    xor  rdx,QWORD PTR fs:0x28
0x00005555555477b <+107>:   je   0x55555554782 <main+114>
0x00005555555477d <+109>:   call 0x555555545c0
0x000055555554782 <+114>:   leave
0x000055555554783 <+115>:   ret
End of assembler dump.
(gdb) x/10a $rsp
0x7fffffffde10: 0x1      0x555555547dd <__libc_csu_init+77>
0x7fffffffde20: 0x55555554814 0x5555555481c
0x7fffffffde30: 0x5555555481f 0x0
0x7fffffffde40: 0x7fffffffdf30 0x1951c62b86755e00
0x7fffffffde50: 0x55555554790 <__libc_csu_init> 0x7ffff7a313f1 <__libc_start_main+241>
```

Abbildung 3: Aufgabe 1a Inhalt Stack vor Aufruf `execve()`

```
1 0x7fffffffde20: 0x55555554814 -> "/bin/sh"
2 0x7fffffffde28: 0x5555555481c -> "-c"
3 0x7fffffffde30: 0x5555555481f -> "ls -l"
4 0x7fffffffde38: 0x0
```

```
(gdb) x/s 0x55555554814
0x55555554814: "/bin/sh"
(gdb) x/s 0x5555555481c
0x5555555481c: "-c"
(gdb) x/s 0x5555555481f
0x5555555481f: "ls -l"
(gdb)
```

Abbildung 4: Speicherort der Strings

Somit wird über den Register rsi das Stringarrays `cmd` übergeben.

```
1 0x0000555555554758 <+72>:    mov  eax, 0x0
```

Ist ein NULL, welches das dritte Übergabeparameter von `execve` ist.

2.2.2 Wie und an welcher Stelle werden die übergebenen Parameter im Speicher abgelegt?

Zunächst werden Pointer für alle Strings auf den Stack geschrieben um dann vor Funktionsaufruf in den entsprechenden Registern gespeichert zu werden.

Dump of assembler code for function `main`:

```
1 0x0000555555554710 <+0>: push  rbp
2 0x0000555555554711 <+1>: mov   rbp, rsp
3 0x0000555555554714 <+4>: sub   rsp, 0x40
4 0x0000555555554718 <+8>: mov   rax, QWORD PTR fs:0x28
5 0x0000555555554721 <+17>: mov   QWORD PTR [rbp-0x8], rax
6 0x0000555555554725 <+21>: xor   eax, eax
7 0x0000555555554727 <+23>: lea   rax, [rip+0xe6]
8 0x000055555555472e <+30>: mov   QWORD PTR [rbp-0x30], rax
9 0x0000555555554732 <+34>: lea   rax, [rip+0xe3]
10 0x0000555555554739 <+41>: mov   QWORD PTR [rbp-0x28], rax
11 0x000055555555473d <+45>: lea   rax, [rip+0xdb]
12 0x0000555555554744 <+52>: mov   QWORD PTR [rbp-0x20], rax
13 0x0000555555554748 <+56>: mov   QWORD PTR [rbp-0x18], 0x0
14 0x0000555555554750 <+64>: mov   rax, QWORD PTR [rbp-0x30]
15 0x0000555555554754 <+68>: lea   rcx, [rbp-0x30]
16 0x0000555555554758 <+72>: mov   edx, 0x0
17 0x000055555555475d <+77>: mov   rsi, rcx
18 0x0000555555554760 <+80>: mov   rdi, rax
19 0x0000555555554763 <+83>: call  0x5555555545c8
20 0x0000555555554768 <+88>: mov   DWORD PTR [rbp-0x34], eax
21 0x000055555555476b <+91>: mov   eax, DWORD PTR [rbp-0x34]
22 0x000055555555476e <+94>: mov   rdx, QWORD PTR [rbp-0x8]
23 0x0000555555554772 <+98>: xor   rdx, QWORD PTR fs:0x28
24 0x000055555555477b <+107>: je    0x555555554782 <main+114>
25 0x000055555555477d <+109>: call  0x5555555545c0
26 0x0000555555554782 <+114>: leave
27 0x0000555555554783 <+115>: ret
```


Siehe Abbildung 3 für den Inhalt des Stacks, dieser reicht von der Adresse des Registers rbp(0x7fffffffde50) bis rsp(0x7fffffffde10) +23 Pointer von `"/bin/sh"` wird in Register rax geschrieben
+30 Pointer von `"/bin/sh"` wird aus rax auf den Stack Addr: rbp-0x30 geschrieben
+34 Pointer von `"-c"` wird in rax geschrieben
+41 Pointer von `"-c"` wird aus Register rax auf den Stack Addr: rbp-0x28 geschrieben
+45 Pointer von `"ls -l"` wird in Register rax geschrieben
+52 Pointer von `"ls -l"` wird aus Register rax auf den Stack Addr: rbp-0x20 geschrieben
+56 Die folgenden 8 Byte auf dem Stack werden mit 0 beschrieben.
+64 Der Pointer auf den String `"/bin/sh"` wird aus dem Stack in den Register rax geschrieben.
+68 Die Adresse rbp-0x30 wird in den rsi geschrieben, diese Adresse zeigt auf den Stack und somit auf den Beginn des Stringarrays.
+72 0 wird in Register rdx gespeichert.
+77 Pointer der auf den Beginn des Stringarrays im Stack zeigt wird in rsi gespeichert.
+80 Pointer auf den String `"bin/sh"` wird in Register rdi gespeichert.

Im Stack werden die Stringadressen als QWORD Pointer gespeichert, d.h. 8byte pointer =64bit

2.3 Welche Daten befinden sich beim Aufruf von `execve()` im Stack Frame?

Im Stack befindet sich die Rücksprungadresse auf die mainfunktion

```
0x7fffffffde08: 0x555555554768 <main+88>
```

Abbildung 5: neuer Stackframe beim aufruf von `execve()`

```
1 (gdb) x/a $rsp
2 0x7fffffffde08: 0x555555554768 <main+88>
```

0x555555554768 ist die Adresse die der rip nach ausführen der funktion `execve` wieder aufnehmen muss, dies ist der nächste Befehl nach Aufruf der Funktion `execve()` in der Mainfunktion

```
1 0x0000555555554763 <+83>: call 0x5555555545c8
2 0x0000555555554768 <+88>: mov  DWORD PTR [rbp-0x34], eax
3 0x000055555555476b <+91>: mov  eax, DWORD PTR [rbp-0x34]
4 0x000055555555476e <+94>: mov  rdx, QWORD PTR [rbp-0x8]
5 0x0000555555554772 <+98>: xor  rdx, QWORD PTR fs:0x28
6 0x000055555555477b <+107>: je   0x555555554782 <main+114>
7 0x000055555555477d <+109>: call 0x5555555545c0
8 0x0000555555554782 <+114>: leave
9 0x0000555555554783 <+115>: ret
```