

Malware-Analyse und Reverse Engineering

6: Ausnutzen von Buffer Overflows

27.4.2017

Prof. Dr. Michael Engel

Basierend auf Unterlagen von Bart Coppens

<https://www.bartcoppens.be/>
MARE 06 – Ausnutzen von Buffer Overflows



Überblick

Themen:

- Buffer overflows:
 - Ausnutzen: Returnadressen (Wdh.)
 - Return into libc
 - Return-Oriented Programming

Angriffsmethoden bei ausführbarem Stack

Ausführbarer Stack

- Code kann direkt auf dem Stack abgelegt werden
- „Mitliefern“ von Code für Malware-Funktionalität

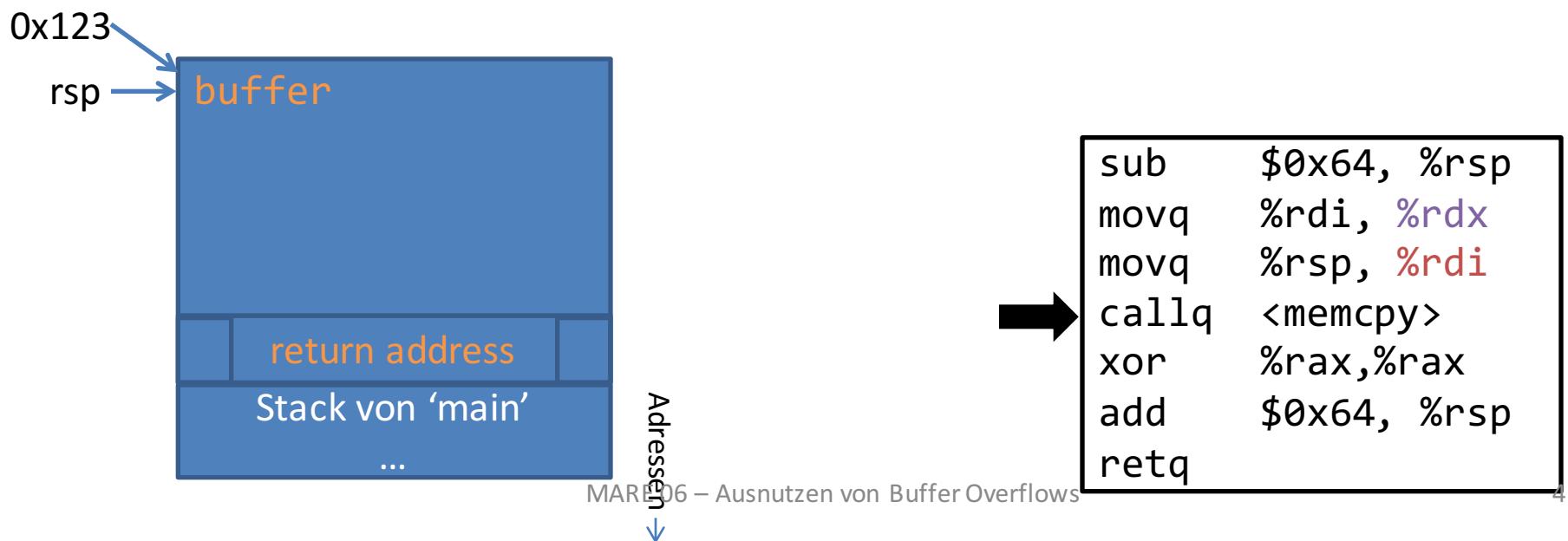
Abwehrmaßnahme

- Abschalten der Möglichkeit, Code auf beschreibbaren Speicherseiten auszuführen (in Hardware)

Ausführbarer Stack

```
int silly_function(int len, char* src) {
    char buffer[100];
    memcpy(buffer, src, len); ← function(%rdi, %rsi, %rdx)
    return 0;
}
```

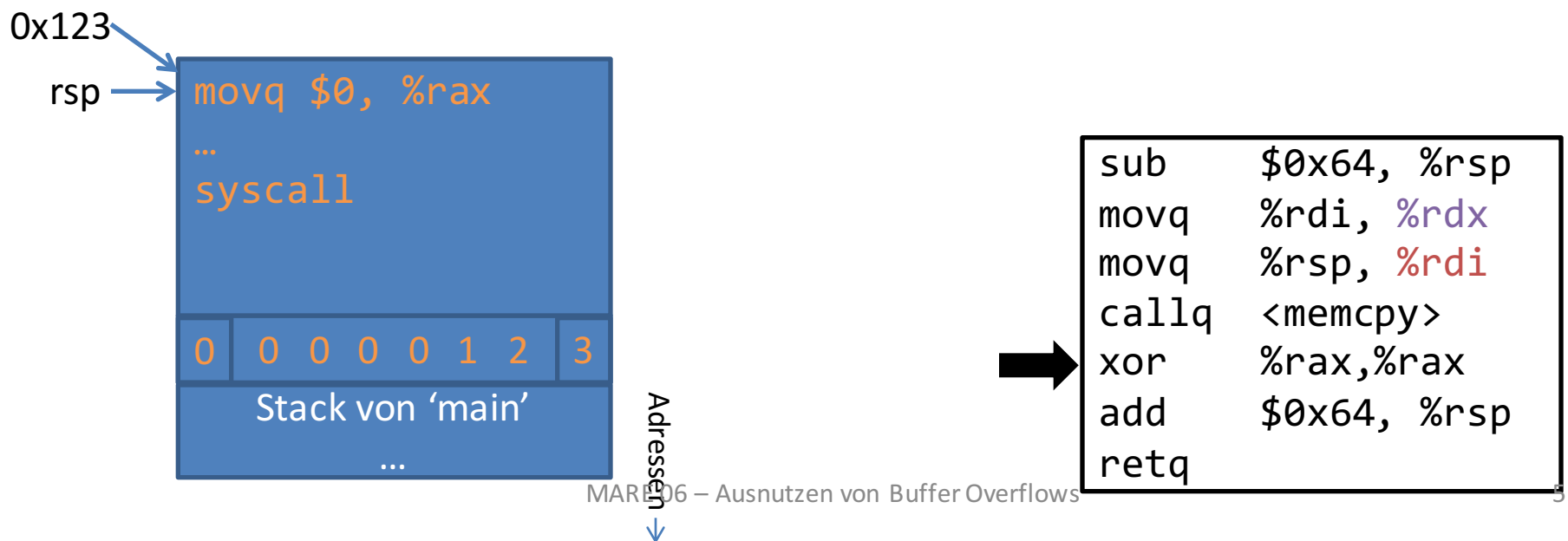
```
int main() {
    return silly_function(108, "\x48\xc7...\x23\x01\x00...");
}
```



Ausführbarer Stack

```
int silly_function(int len, char* src) {
    char buffer[100];
    memcpy(buffer, src, len); ← function(%rdi, %rsi, %rdx)
    return 0;
}
```

```
int main() {
    return silly_function(108, "\x48\xc7...\x23\x01\x00...");
}
```



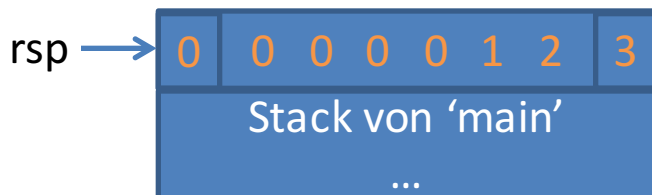
Ausführbarer Stack

```
int silly_function(int len, char* src) {
    char buffer[100];
    memcpy(buffer, src, len); ← function(%rdi, %rsi, %rdx)
    return 0;
}
```

```
int main() {
    return silly_function(108, "\x48\xc7...\x23\x01\x00...");
}
```

0x123 →

```
movq $0, %rax
...
syscall
```



Adressen
↓

```
sub    $0x64, %rsp
movq   %rdi, %rdx
movq   %rsp, %rdi
callq  <memcpy>
xor     %rax,%rax
add     $0x64, %rsp
retq
```

Ausführbarer Stack

```
int silly_function(int len, char* src) {  
    char buffer[100];  
    memcpy(buffer, src, len);  
    return 0;  
}
```

← function(%rdi, %rsi, %rdx)

```
int main() {  
    return silly_function(108, "\x48\xc7...\x23\x01\x00...");  
}
```

0x123 →

```
movq $0, %rax  
...  
syscall
```

rsp →

Stack von 'main'
...

Adressen
↓

```
sub    $0x64, %rsp  
movq   %rdi, %rdx  
movq   %rsp, %rdi  
callq  <memcpy>  
xor     %rax, %rax  
add     $0x64, %rsp  
retq
```

Ausführbarer Stack

```
int silly_function(int len, char* src) {
    char buffer[100];
    memcpy(buffer, src, len); ← function(%rdi, %rsi, %rdx)
    return 0;
}
```

```
int main() {
    return silly_function(108, "\x48\xc7...\x23\x01\x00...");
}
```

0x123 →

→ `movq $0, %rax`

...

`syscall`

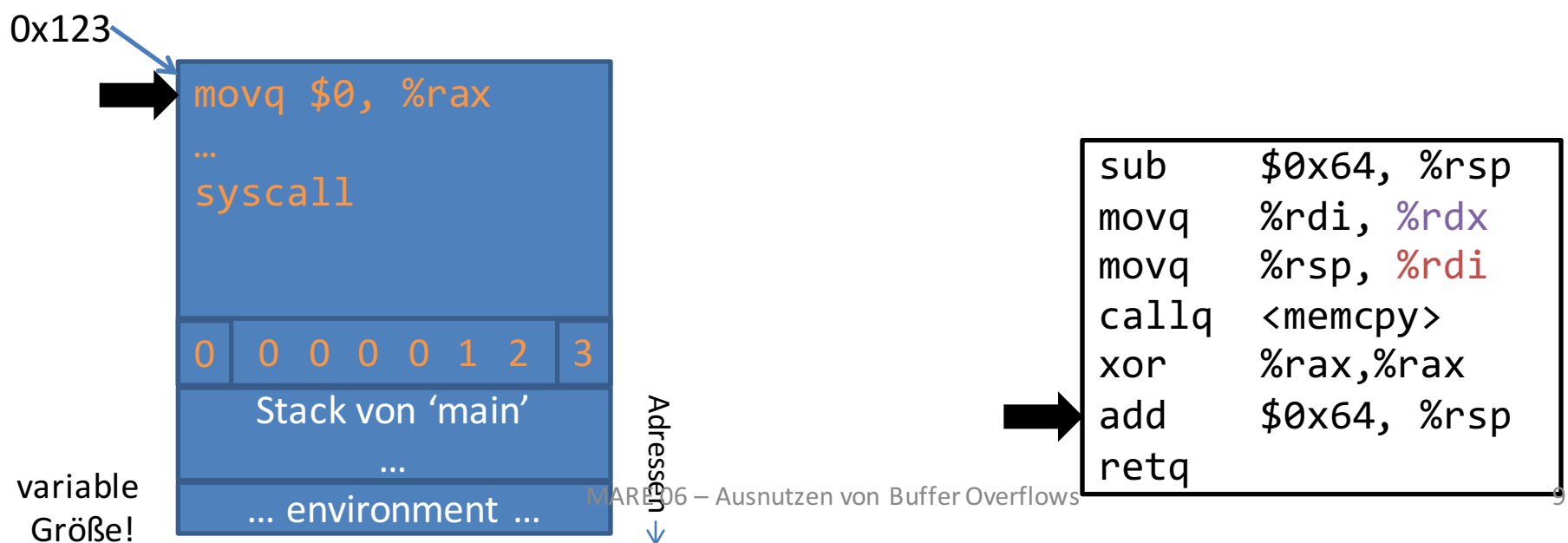
sub	\$0x64, %rsp
movq	%rdi, %rdx
movq	%rsp, %rdi
callq	<memcpy>
xor	%rax,%rax
add	\$0x64, %rsp
retq	



Ausführbarer Stack

```
int silly_function(int len, char* src) {
    char buffer[100];
    memcpy(buffer, src, len); ← function(%rdi, %rsi, %rdx)
    return 0;
}
```

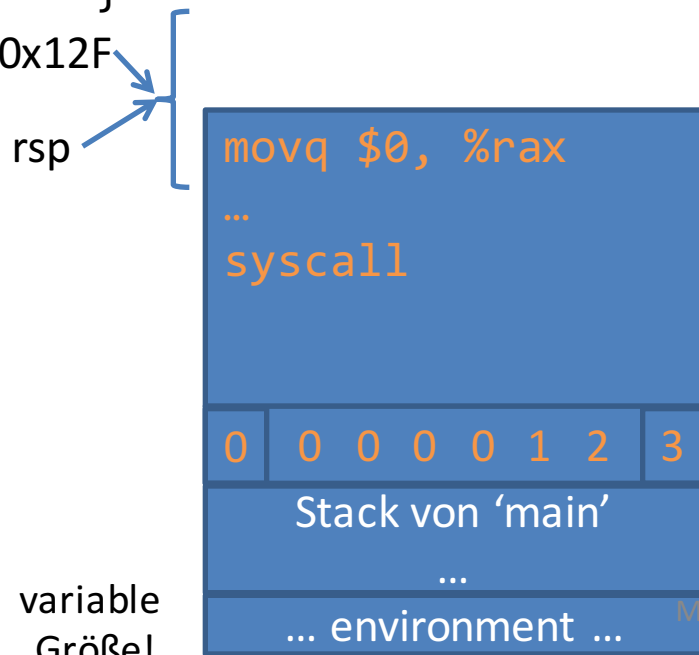
```
int main() {
    return silly_function(108, "\x48\xc7...\x23\x01\x00...");
}
```



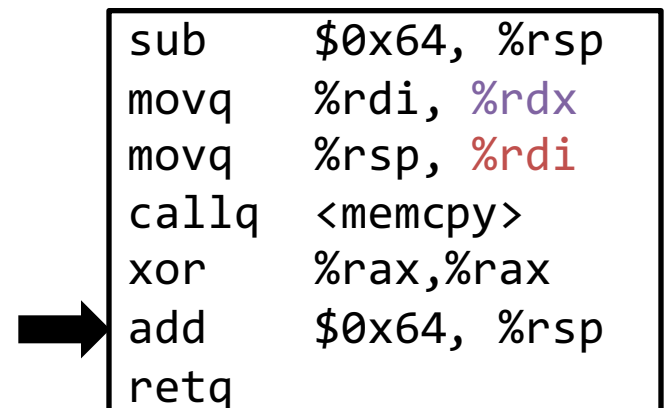
Ausführbarer Stack

```
int silly_function(int len, char* src) {
    char buffer[100];
    memcpy(buffer, src, len); ← function(%rdi, %rsi, %rdx)
    return 0;
}
```

```
int main() {
    return silly_function(108, "\x48\xc7...\x23\x01\x00...");
}
```



Adressen
↓



```
sub    $0x64, %rsp
movq   %rdi, %rdx
movq   %rsp, %rdi
callq  <memcpy>
xor     %rax,%rax
add     $0x64, %rsp
retq
```

Ausführbarer Stack

```
int silly_function(int len, char* src) {
    char buffer[100];
    memcpy(buffer, src, len);
    return 0;
}
```

← function(%rdi, %rsi, %rdx)

```
int main() {
    return silly_function(108, "\x48\xc7...\x23\x01\x00...");
}
```

0x12F →

```
movq $0, %rax
...
syscall
```



Adressen ↓

```
sub    $0x64, %rsp
movq   %rdi, %rdx
movq   %rsp, %rdi
callq  <memcpy>
xor     %rax,%rax
add     $0x64, %rsp
retq
```

Ausführbarer Stack

```
int silly_function(int len, char* src) {
    char buffer[100];
    memcpy(buffer, src, len);
    return 0;
}
```

← function(%rdi, %rsi, %rdx)

```
int main() {
    return silly_function(108, "\x48\xc7...\x23\x01\x00...");
}
```

0x12F → **garbage instructions**

movq \$0, %rax
...
syscall



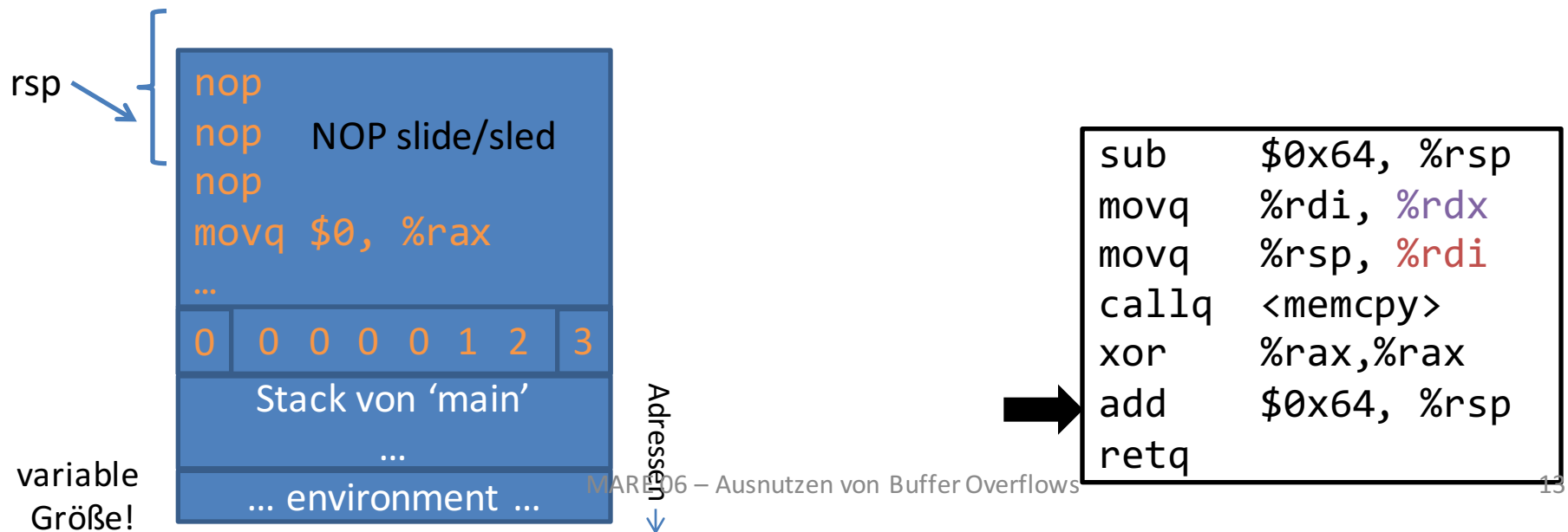
Adressen ↓

```
sub    $0x64, %rsp
movq   %rdi, %rdx
movq   %rsp, %rdi
callq  <memcpy>
xor     %rax,%rax
add     $0x64, %rsp
retq
```

Ausführbarer Stack

```
int silly_function(int len, char* src) {
    char buffer[100];
    memcpy(buffer, src, len); ← function(%rdi, %rsi, %rdx)
    return 0;
}
```


```
int main() {
    return silly_function(108, "\x48\xc7...\x23\x01\x00...");
}
```



Ausführbarer Stack

```
int silly_function(int len, char* src) {
    char buffer[100];
    memcpy(buffer, src, len); ← function(%rdi, %rsi, %rdx)
    return 0;
}
```

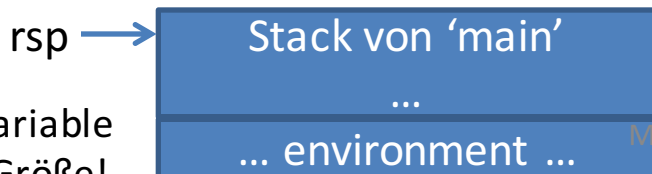
```
int main() {
    return silly_function(108, "\x48\xc7...\x23\x01\x00...");
}
```



 nop
 nop
 nop
 movq \$0, %rax
 ...

NOP slide/sled

```
sub    $0x64, %rsp
movq   %rdi, %rdx
movq   %rsp, %rdi
callq  <memcpy>
xor     %rax,%rax
add     $0x64, %rsp
retq
```

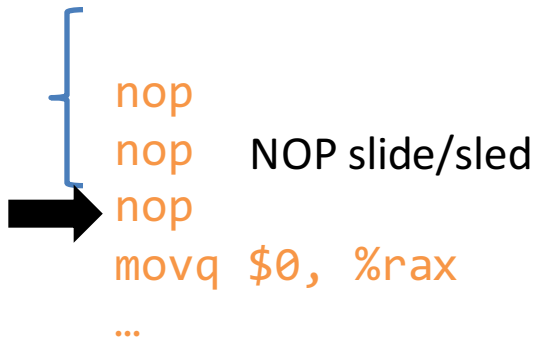


Adressen ↓

Ausführbarer Stack

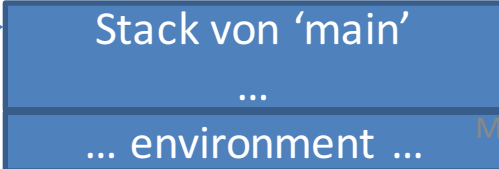
```
int silly_function(int len, char* src) {
    char buffer[100];
    memcpy(buffer, src, len); ← function(%rdi, %rsi, %rdx)
    return 0;
}
```

```
int main() {
    return silly_function(108, "\x48\xc7...\x23\x01\x00...");
}
```



 nop
 nop NOP slide/sled
 nop
 movq \$0, %rax
 ...

```
sub    $0x64, %rsp
movq   %rdi, %rdx
movq   %rsp, %rdi
callq  <memcpy>
xor     %rax,%rax
add     $0x64, %rsp
retq
```

rsp → 

 variable
 Größe!

Adressen
↓

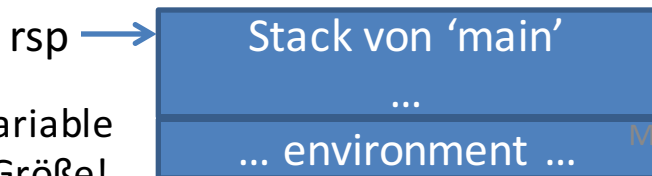
Ausführbarer Stack

```
int silly_function(int len, char* src) {
    char buffer[100];
    memcpy(buffer, src, len); ← function(%rdi, %rsi, %rdx)
    return 0;
}
```

```
int main() {
    return silly_function(108, "\x48\xc7...\x23\x01\x00...");
}
```

{
 nop
 nop NOP slide/sled
 nop
 → movq \$0, %rax
 ...

```
sub    $0x64, %rsp
movq   %rdi, %rdx
movq   %rsp, %rdi
callq  <memcpy>
xor     %rax,%rax
add     $0x64, %rsp
retq
```



Adressen ↓

Angriffsmethoden bei *nicht* ausführbarem Stack (1)



Nicht ausführbarer Stack

- Ausführen von Code im Stackbereich nicht mehr möglich
- Aber: Aufruf existierender Funktionen, z.B. aus der libc
 - oder weiteren geladenen shared libraries
- Nützlich, wenn “brauchbare” Funktion in libc usw. enthalten

Nicht ausführbarer Stack

```
int silly_function(int len, char* src) {  
    char buffer[100];  
    memcpy(buffer, src, len);  
    return 0;  
}
```

```
int main() {  
    return silly_function(108, "\x48\xc7...\x23\x01\x00...");  
}
```

  `movq $0, %rax`
...
`syscall`

**Speicherseiten des Stacks
als "nicht ausführbar"
(DEP = W^X) markiert!**

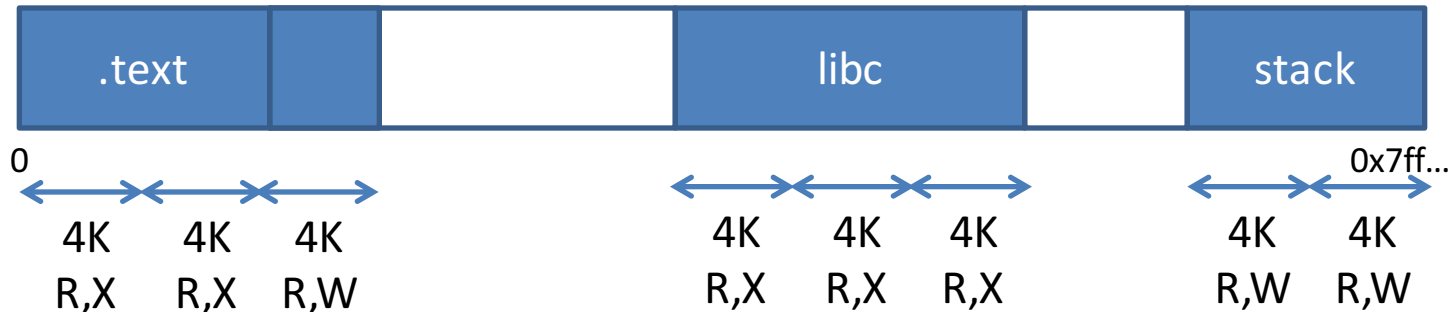
**⇒ Prozessor erzeugt Exception
⇒ Betriebssystem bricht
den Prozess ab!**

`rsp` →  Stack von 'main'
...

Adressen
↓

```
sub    $0x64, %rsp  
movq   %rdi, %rdx  
movq   %rsp, %rdi  
callq  <memcpy>  
xor     %rax, %rax  
add     $0x64, %rsp  
retq
```

Nicht ausführbarer Stack



Eigener Code so nicht mitlieferbar

Aber: anderer Code ist in .text-Segment des Programms und der shared libraries enthalten: z.B. Aufruf von Systemfunktionen

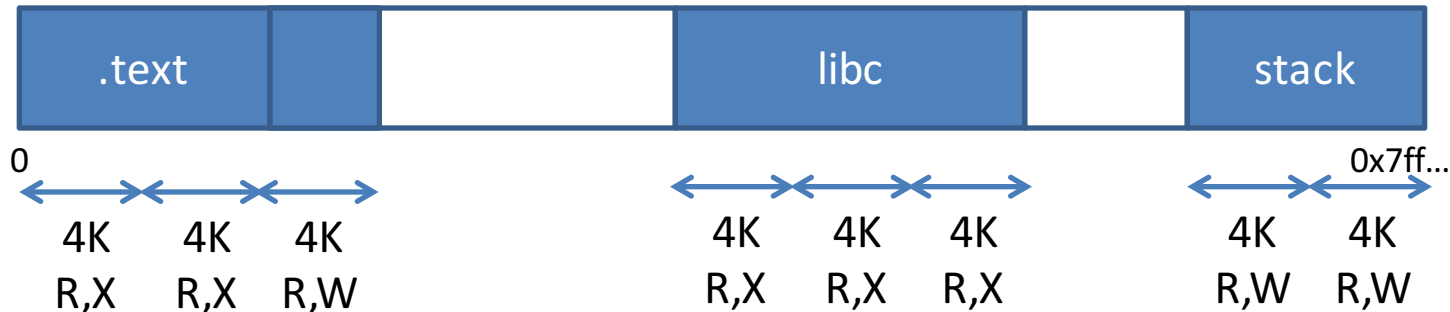
⚡ → `movq $0, %rax`
`... syscall`

rsp → **Stack von 'main'**
`...`

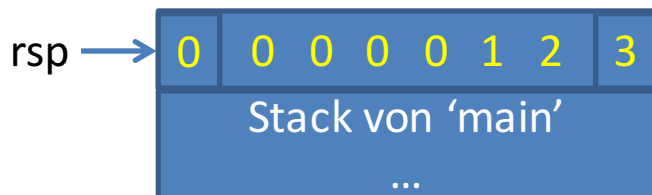
Adressen ↓

```
sub    $0x64, %rsp
movq   %rdi, %rdx
movq   %rsp, %rdi
callq  <memcpy>
xor     %rax,%rax
add     $0x64, %rsp
retq
```

Nicht ausführbarer Stack



Anderer Code in .text-Segment
=> Anspringen durch Überschreiben der return-Adresse auf dem Stack mit Adresse von „nützlicher“ Funktion



```
sub    $0x64, %rsp
movq   %rdi, %rdx
movq   %rsp, %rdi
callq  <memcpy>
xor     %rax,%rax
add     $0x64, %rsp
retq
```

Nützliche libc-Funktion: `exit()`

Dokumentation zu libc-Funktionen mit: `$ man 3 exit`

- “3” ist “Kapitelnummer” im Unix-Handbuch für libc-Funktionen
- libc-Funktion `exit` beendet das aufrufende Programm regulär

```
EXIT(3)                                Linux Programmer's Manual                                EXIT(3)

NAME      top

    exit - cause normal process termination

SYNOPSIS  top

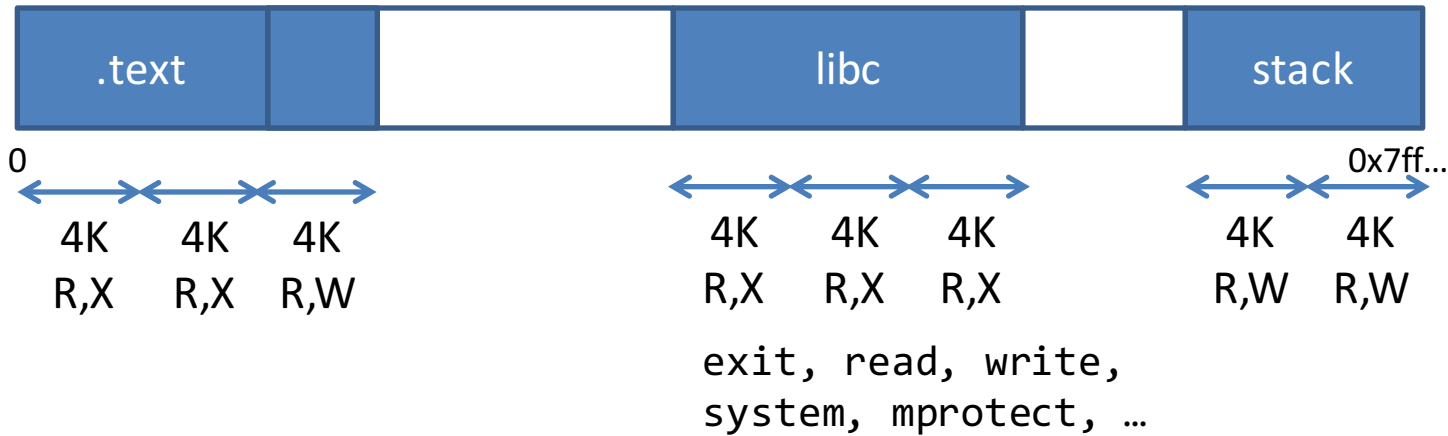
    #include <stdlib.h>

    void exit(int status);

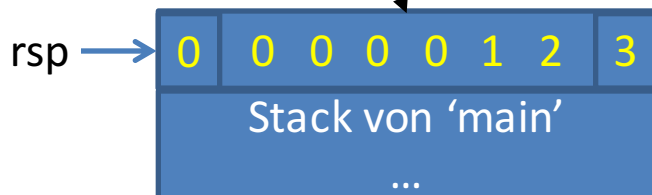
DESCRIPTION  top

    The exit() function causes normal process termination and the value of status & 0377 is returned to the parent (see wait(2)).
```

Return zur libc



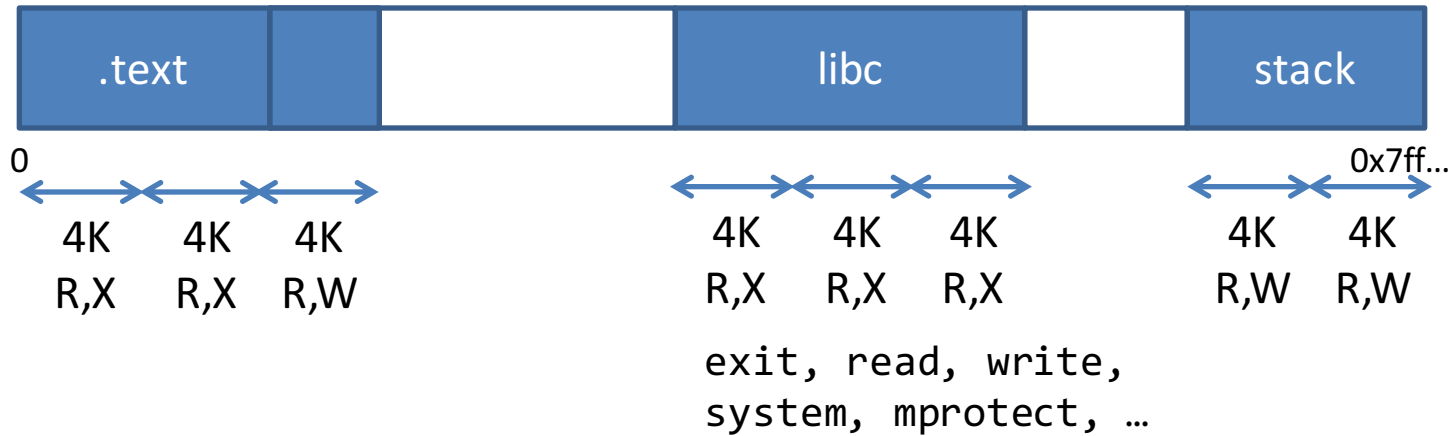
Returnadresse mit
Adresse von exit()
überschrieben



Beispiel für nützliche Funktion: `exit()`
=> Programm wird beendet

```
sub    $0x64, %rsp
movq   %rdi, %rdx
movq   %rsp, %rdi
callq  <memcpy>
xor     %rax, %rax
add     $0x64, %rsp
retq
```

Return zur libc



Anspringen von `exit()`
=> Programm wird beendet



Adressen
↓

```
sub    $0x64, %rsp
movq   %rdi, %rdx
movq   %rsp, %rdi
callq  <memcpy>
xor     %rax,%rax
add     $0x64, %rsp
retq
```

Nützliche libc-Funktion: system()

Nur Programm beenden ist langweilig 😊

- Wir wollen *andere Programme aufrufen*, z.B. eine Shell
- libc-Funktion system – Aufruf eines Programms mit Parametern

NAME [top](#)

system - execute a shell command

SYNOPSIS [top](#)

```
#include <stdlib.h>
```

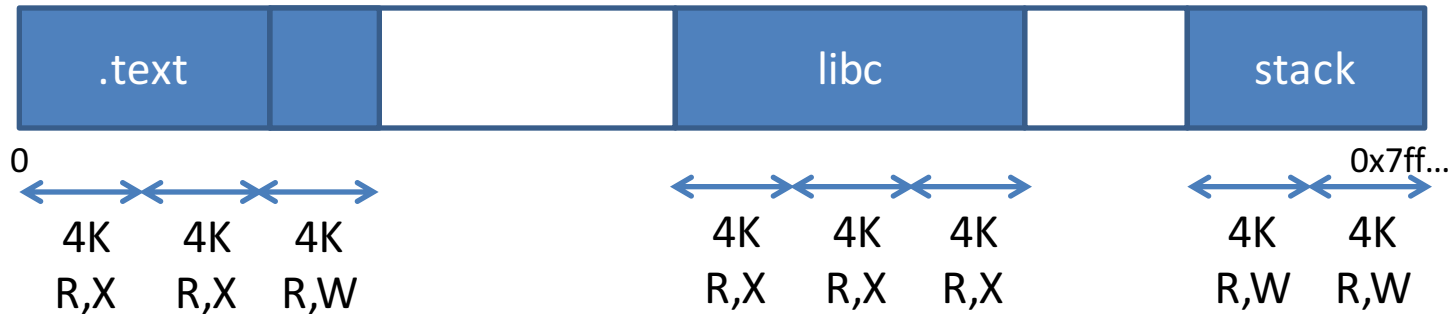
```
int system(const char *command);
```

DESCRIPTION [top](#)

The **system()** library function uses `fork(2)` to create a child process that executes the shell command specified in `command` using `execl(3)` as follows:

```
execl("/bin/sh", "sh", "-c", command, (char *) 0);
```


Return zur libc



```
int system(const char *command);
```

exit, read, write,
system, mprotect, ...

Wie kann der Parameter „command“ an den Aufruf von system übergeben werden?

Beispiel für nützliche Funktion: **system()**
=> Returnadresse überschreiben

rsp →

Adresse von system

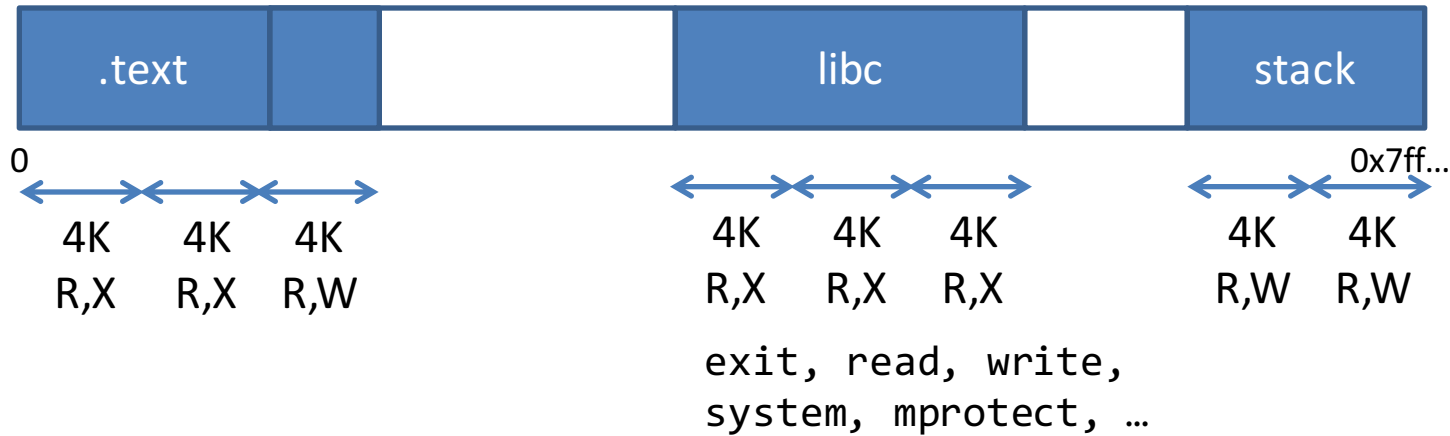
Stack von 'main'

...

Adressen ↓

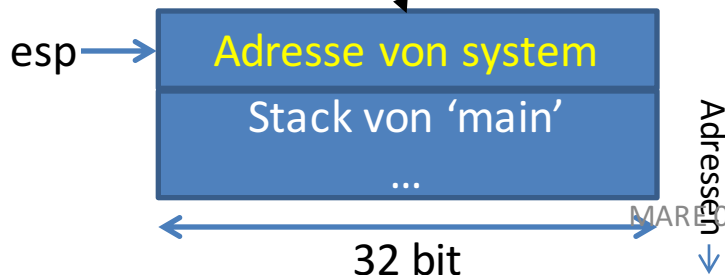
```
sub    $0x64, %rsp
movq   %rdi, %rdx
movq   %rsp, %rdi
callq  <memcpy>
xor     %rax,%rax
add     $0x64, %rsp
retq
```

Return zur libc



Im 32-Bit-Modus:
Adresse des Parameters
auf dem Stack übergeben

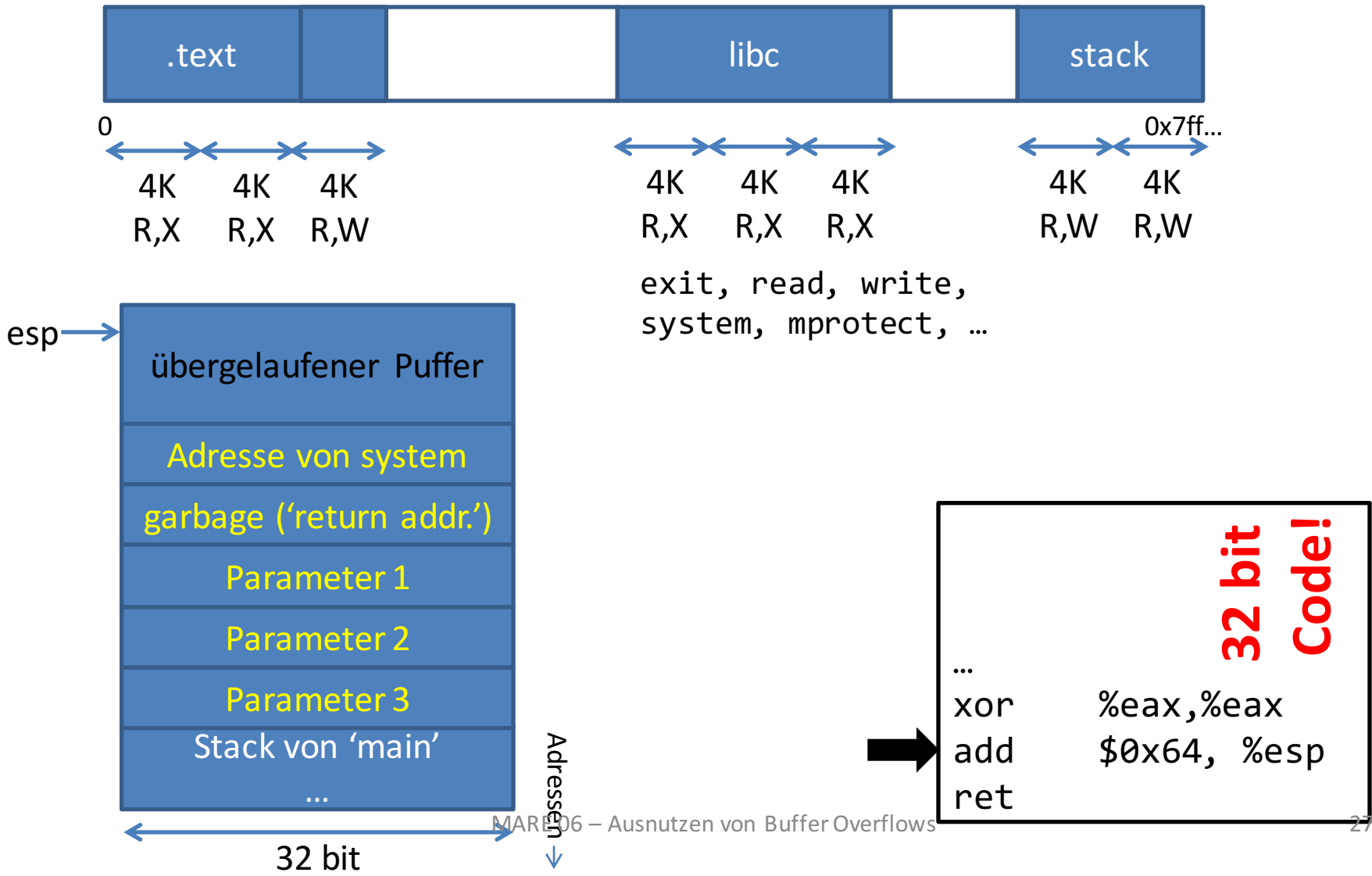
```
int system(const char *command);
```



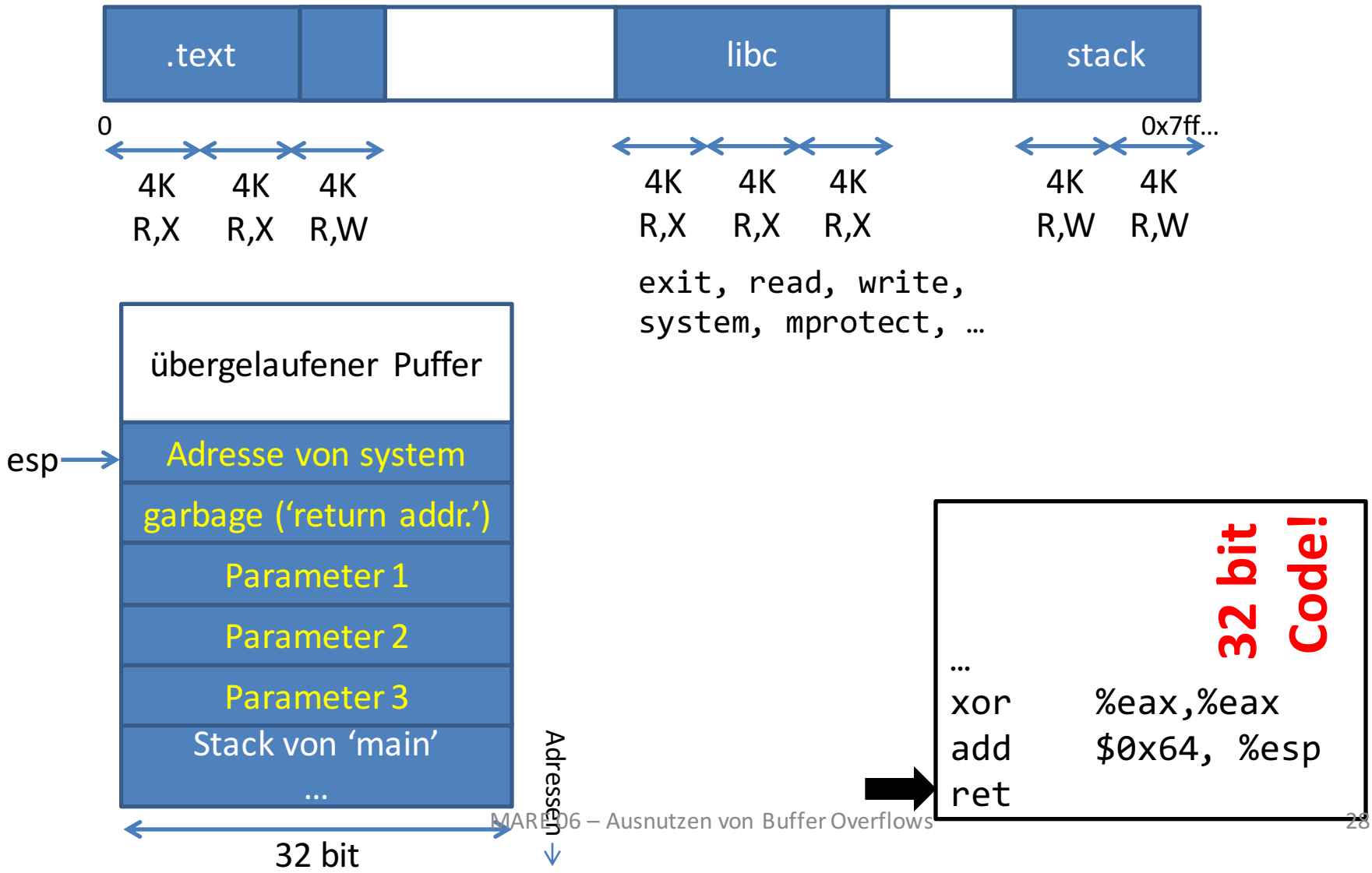
32 bit Code!

```
..
xor    %eax,%eax
add    $0x64, %esp
retq
```

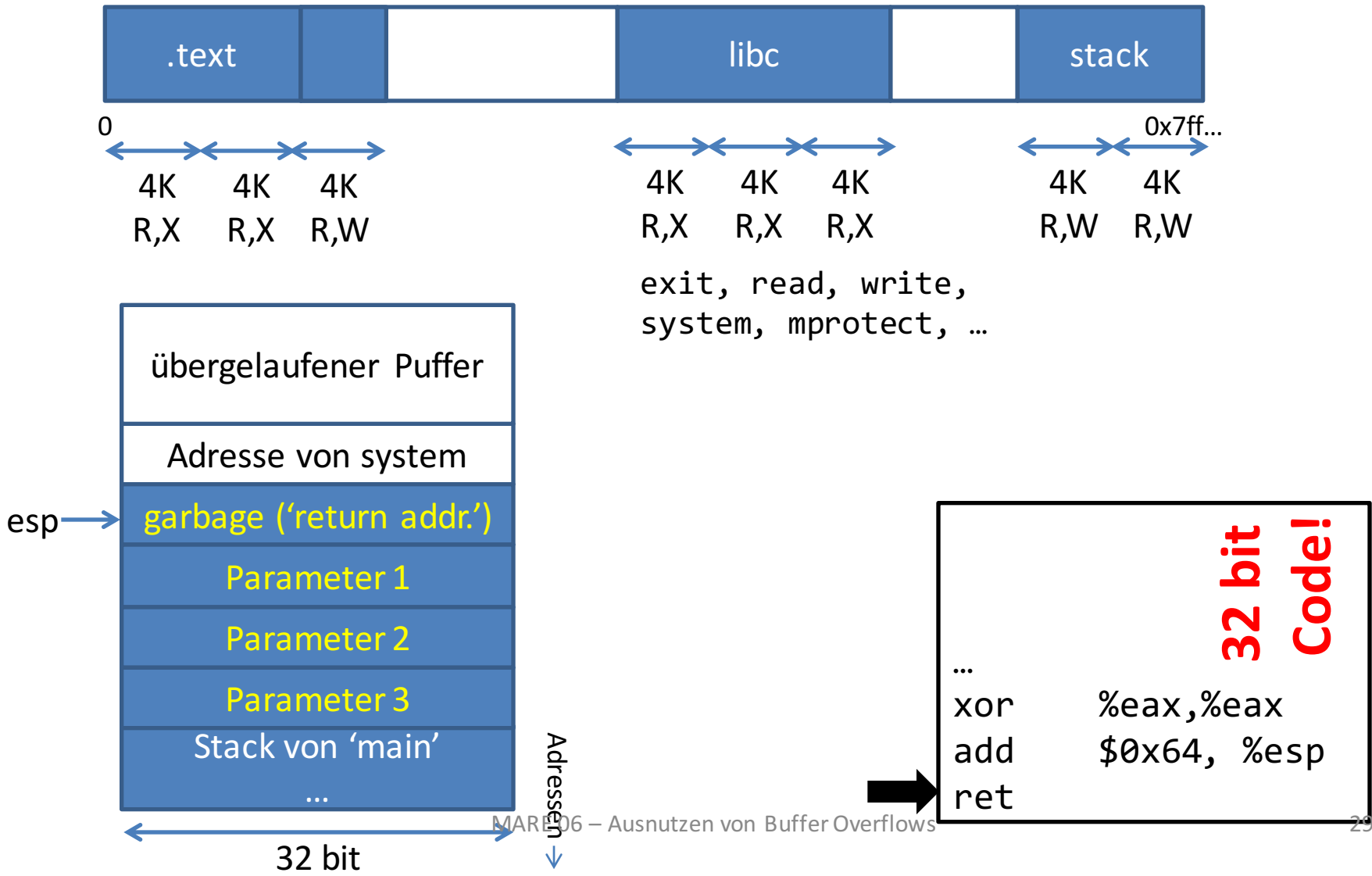
Return zur libc



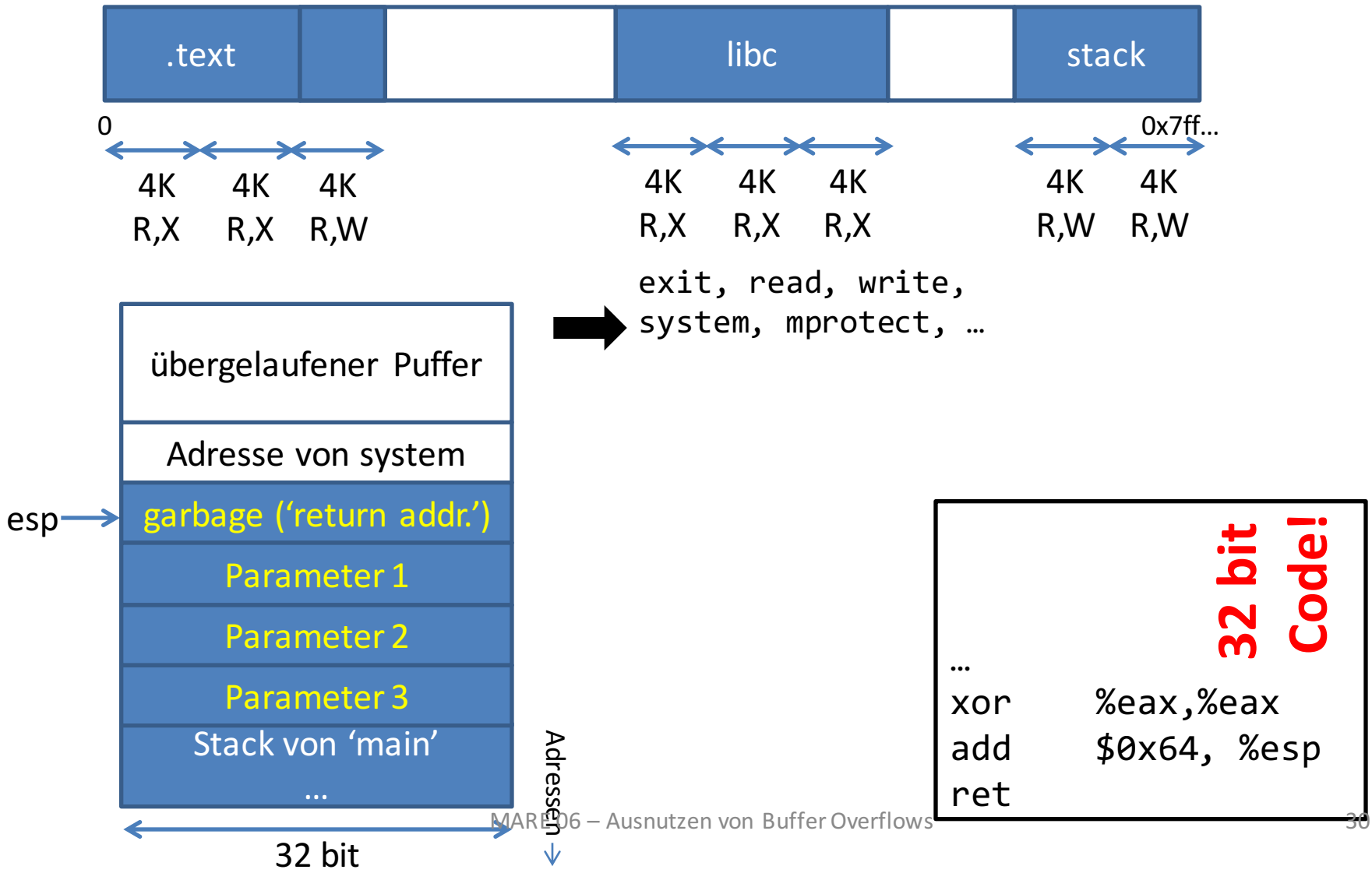
Return zur libc



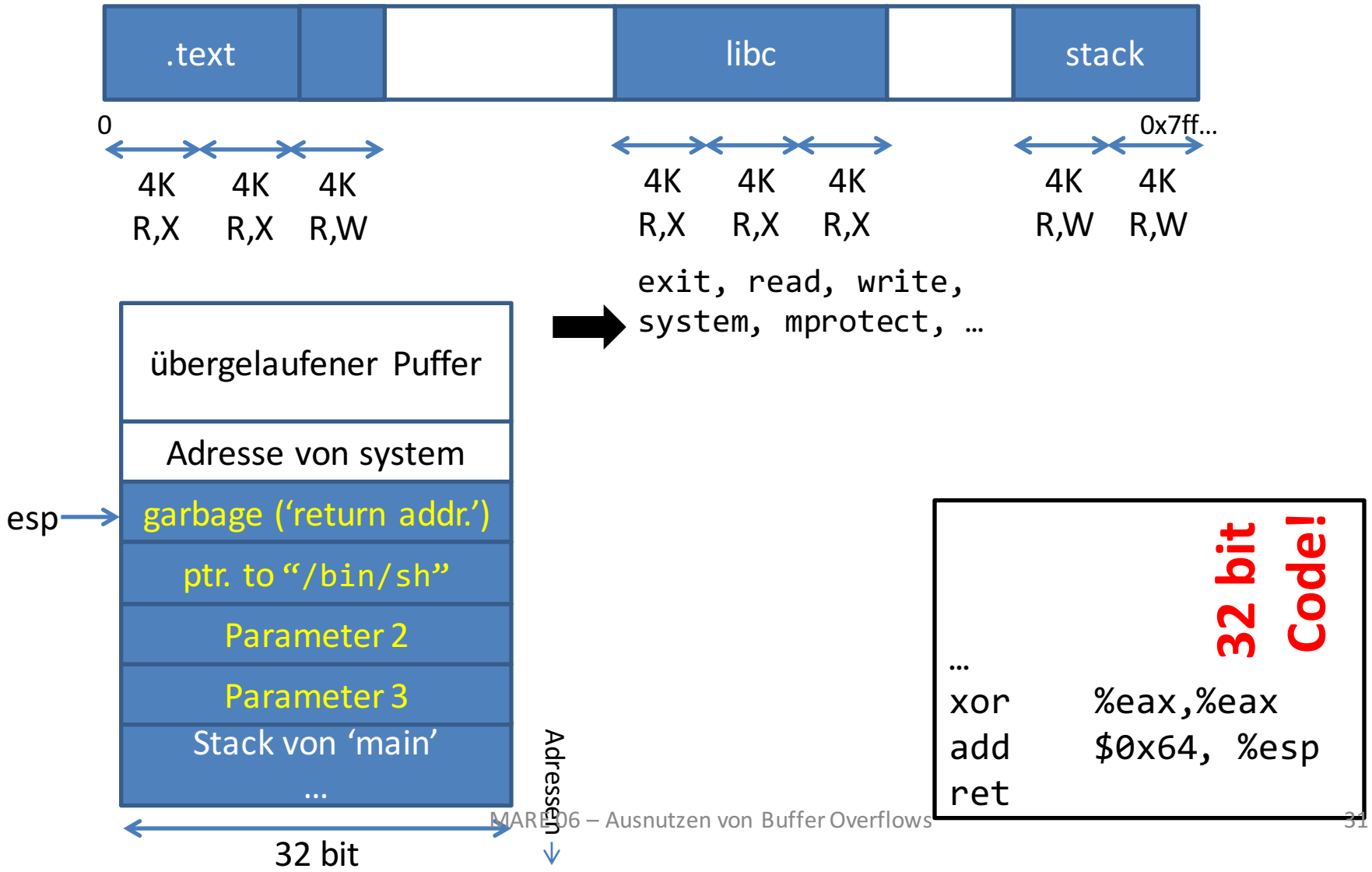
Return zur libc



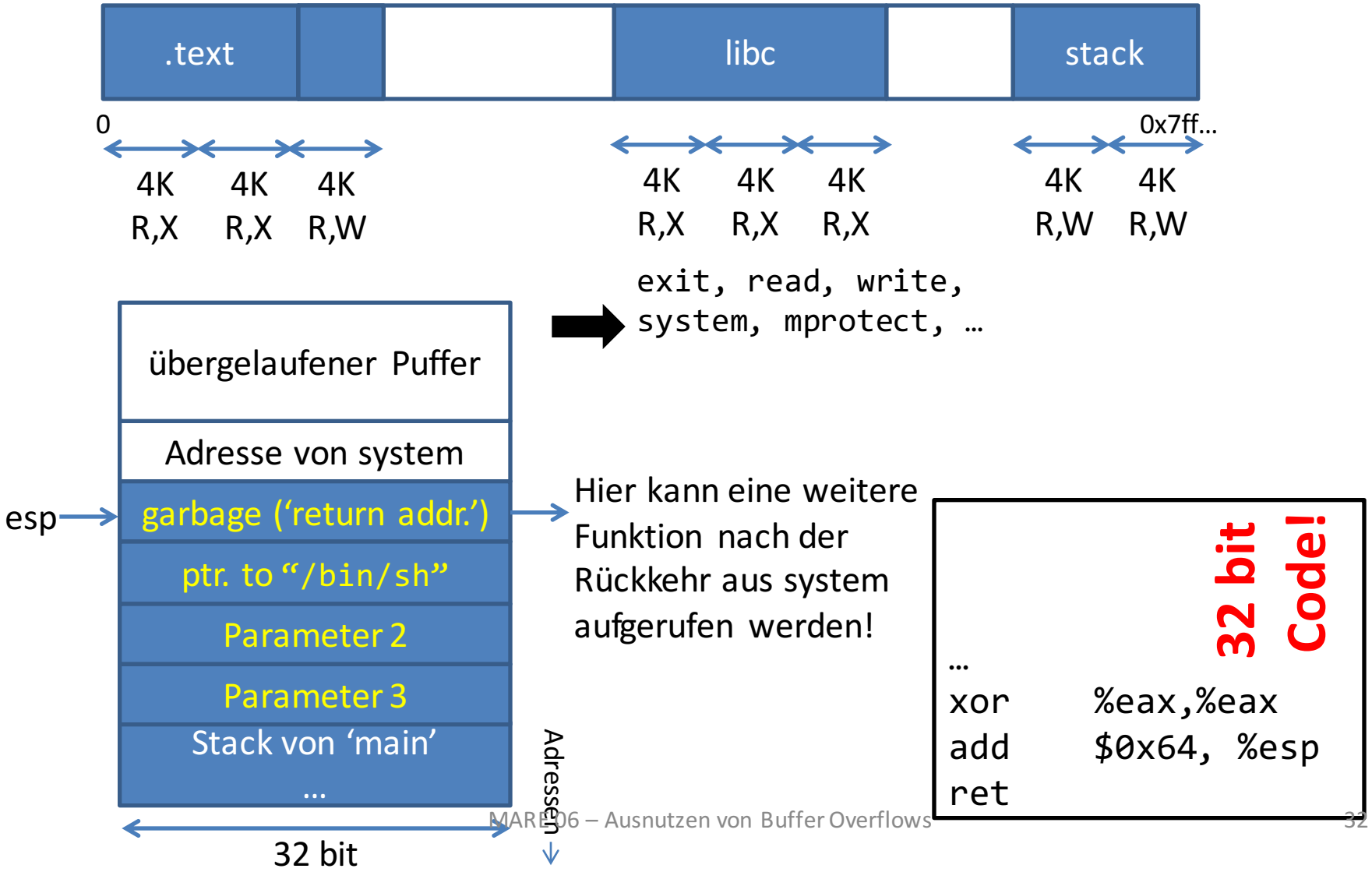
Return zur libc



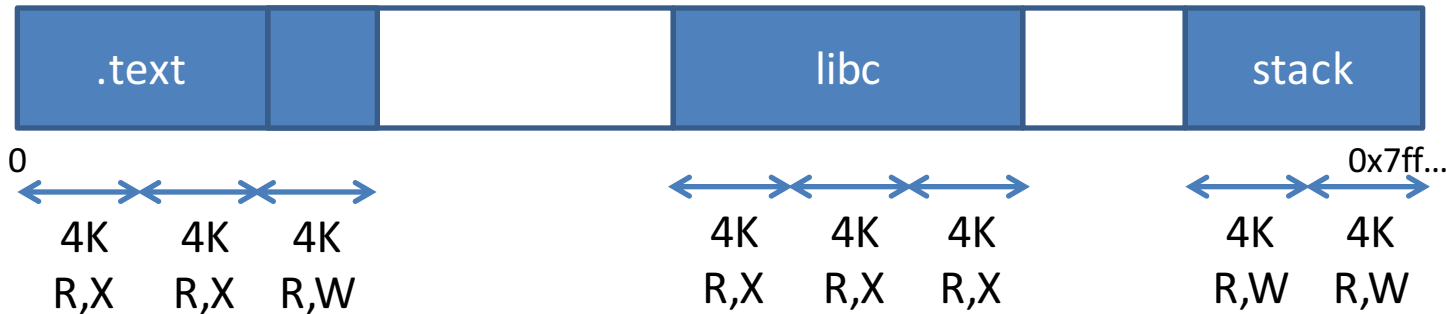
Return zur libc



Return zur libc



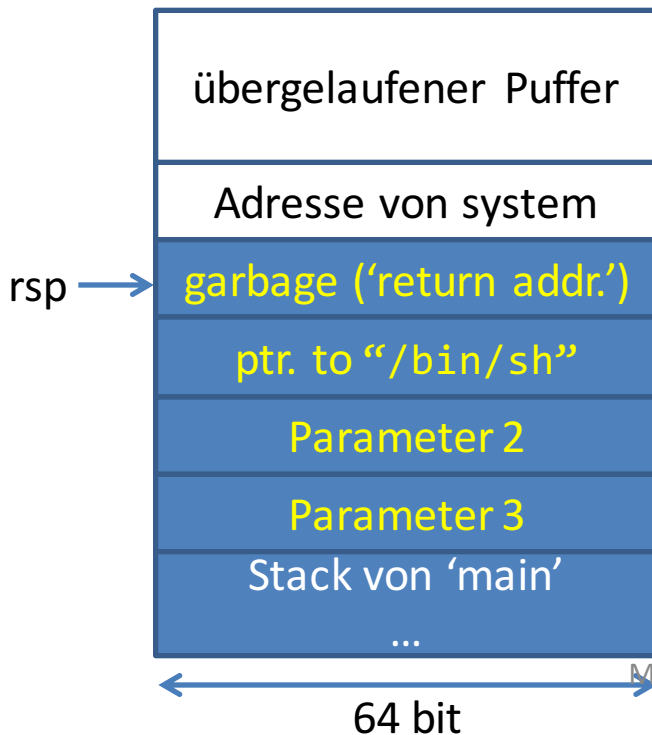
Return zur libc



exit, read, write,
system, mprotect, ...

Im 64-Bit-Modus: Parameterübergabe in Registern!

`fun(%rdi,%rsi,%rdx)`



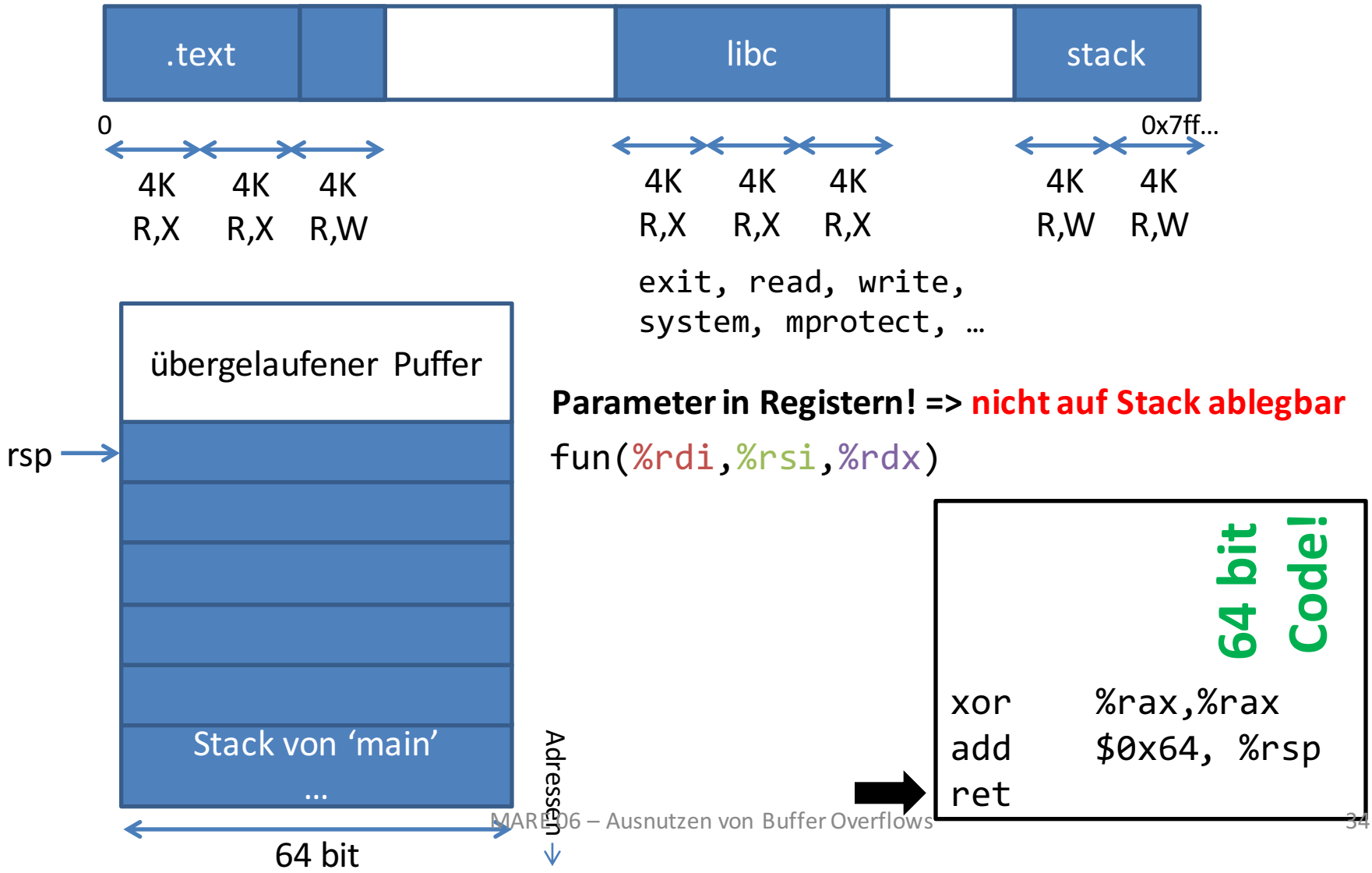
64 bit
Code!

```

...
xor    %rax,%rax
add    $0x64, %rsp
ret

```

Return zur libc



Angriffsmethoden bei *nicht* ausführbarem Stack (3)

Nicht ausführbarer Stack

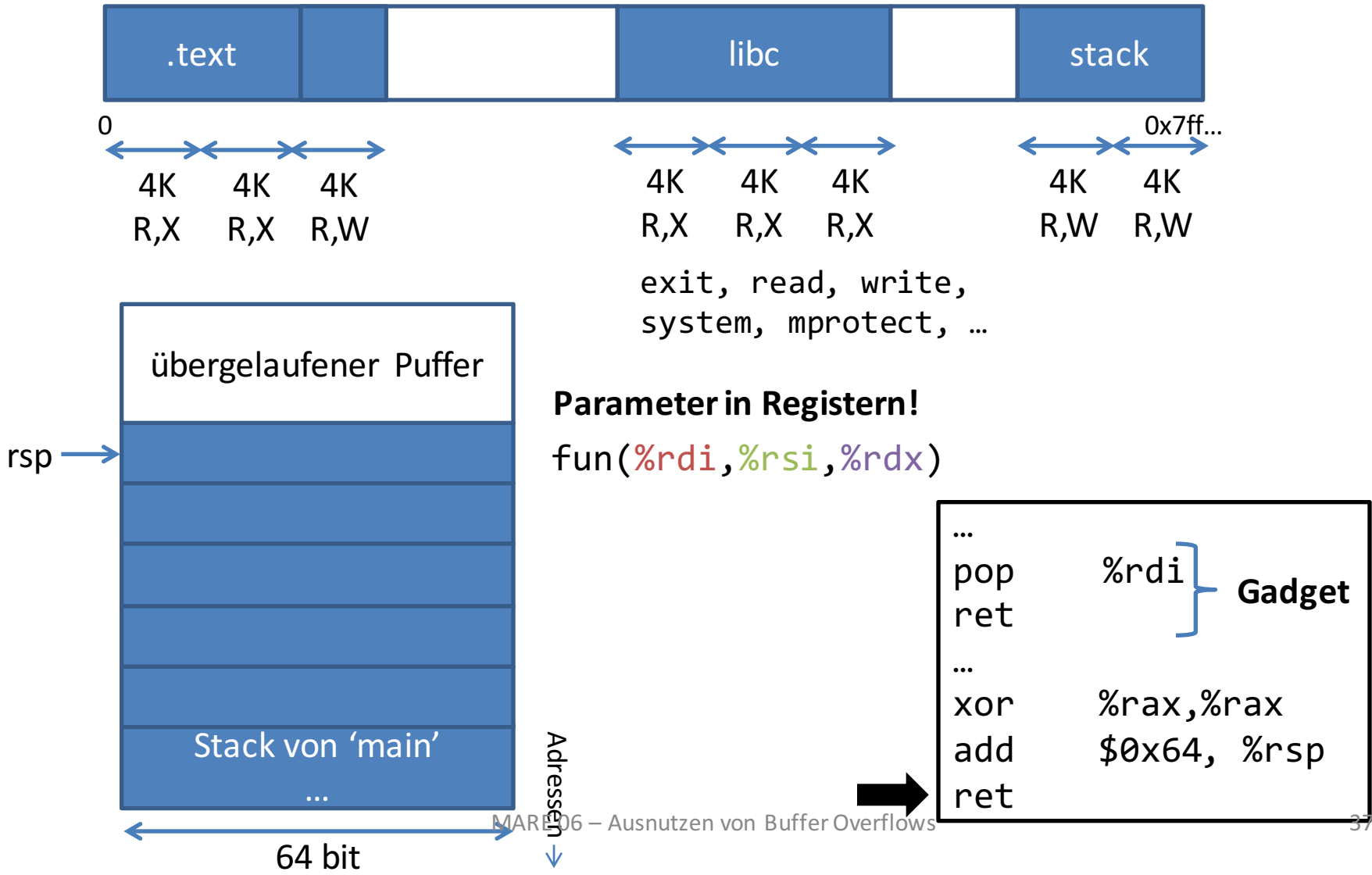
- Ausführen von Code im Stackbereich nicht mehr möglich
 - Kein ausführbarer Code „mitlieferbar“
 - Aber nützliche Funktionen in libc enthalten
- Bei 64-Bit-Code: Parameter in Registern übergeben
 - Parameter nicht auf dem Stack ablegbar
 - Kein Code auf Stack ablegbar, der Register vor Return laden kann
- *Wie können Register mit nützlichen Werten geladen werden?*
 - **Return-Oriented Programming**

Return-Oriented Programming

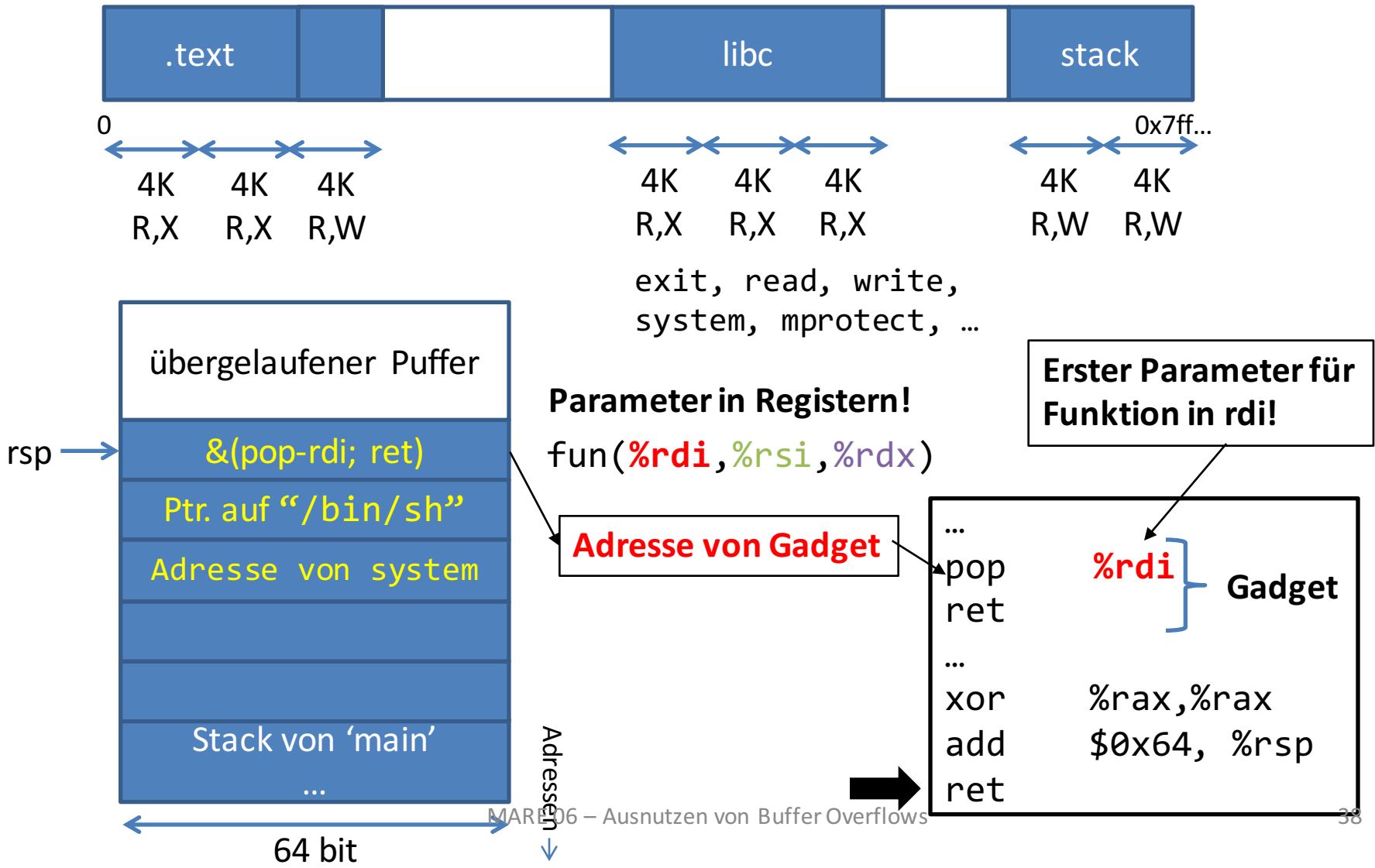
- *Wie können Register mit nützlichen Werten geladen werden?*
 - Suchen von passenden Bytefolgen im Text-Segment von Programm und shared Libraries mit nützlicher Teilfunktionalität: sog. **“Gadgets”**
 - Bytefolgen stellen kurze Folgen von Maschineninstruktionen dar
 - Hier: Laden eines Registers vom Stack: **pop**
 - Jeweils “Return”-Instruktion am Ende
 - **Return-Oriented Programming**
=> Abfolge durch Stackinhalt kontrollierbar
 - Damit: “Zusammenstückeln” von Parameter-Lade-Befehlen



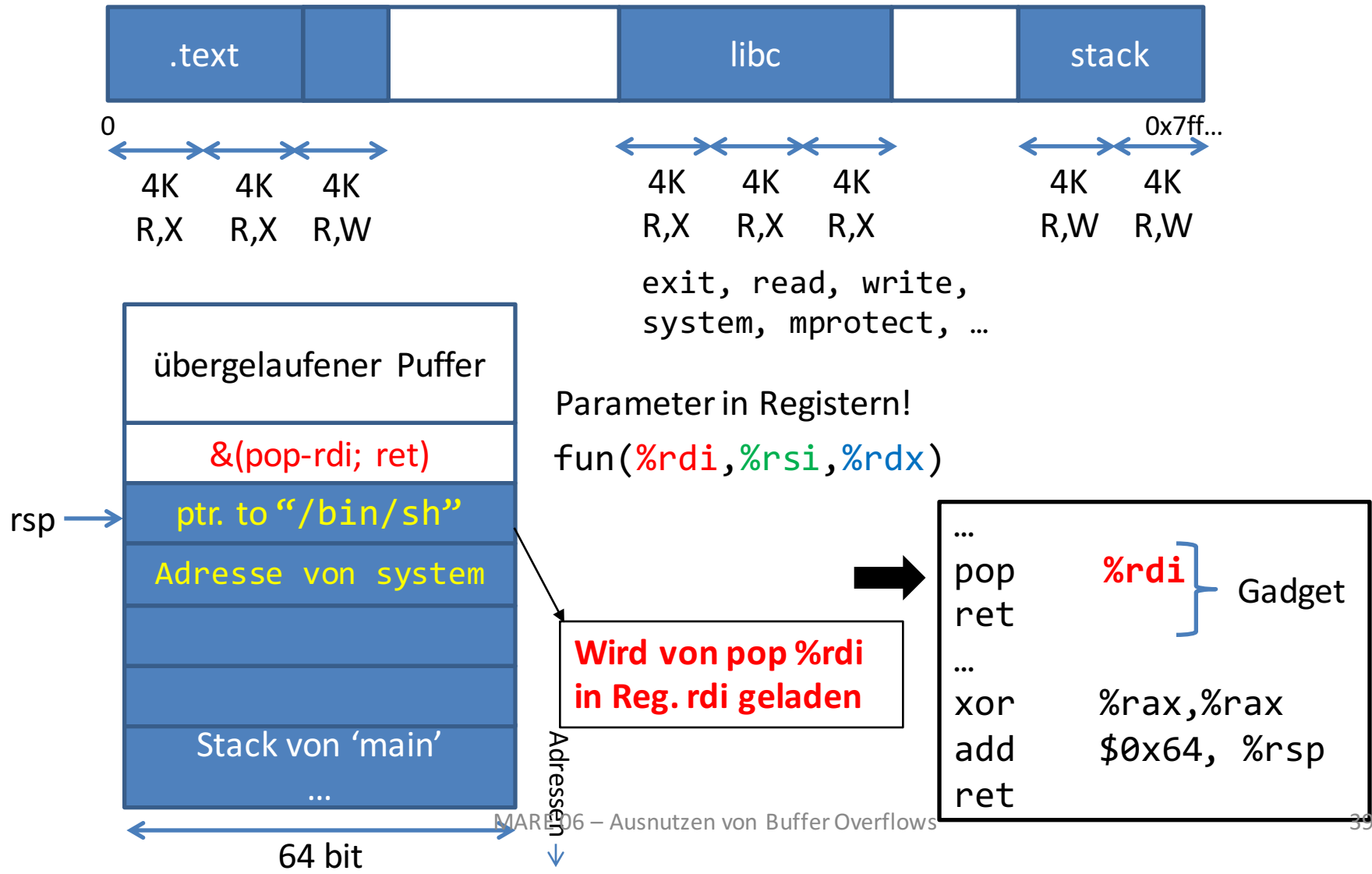
Return-Oriented Programming



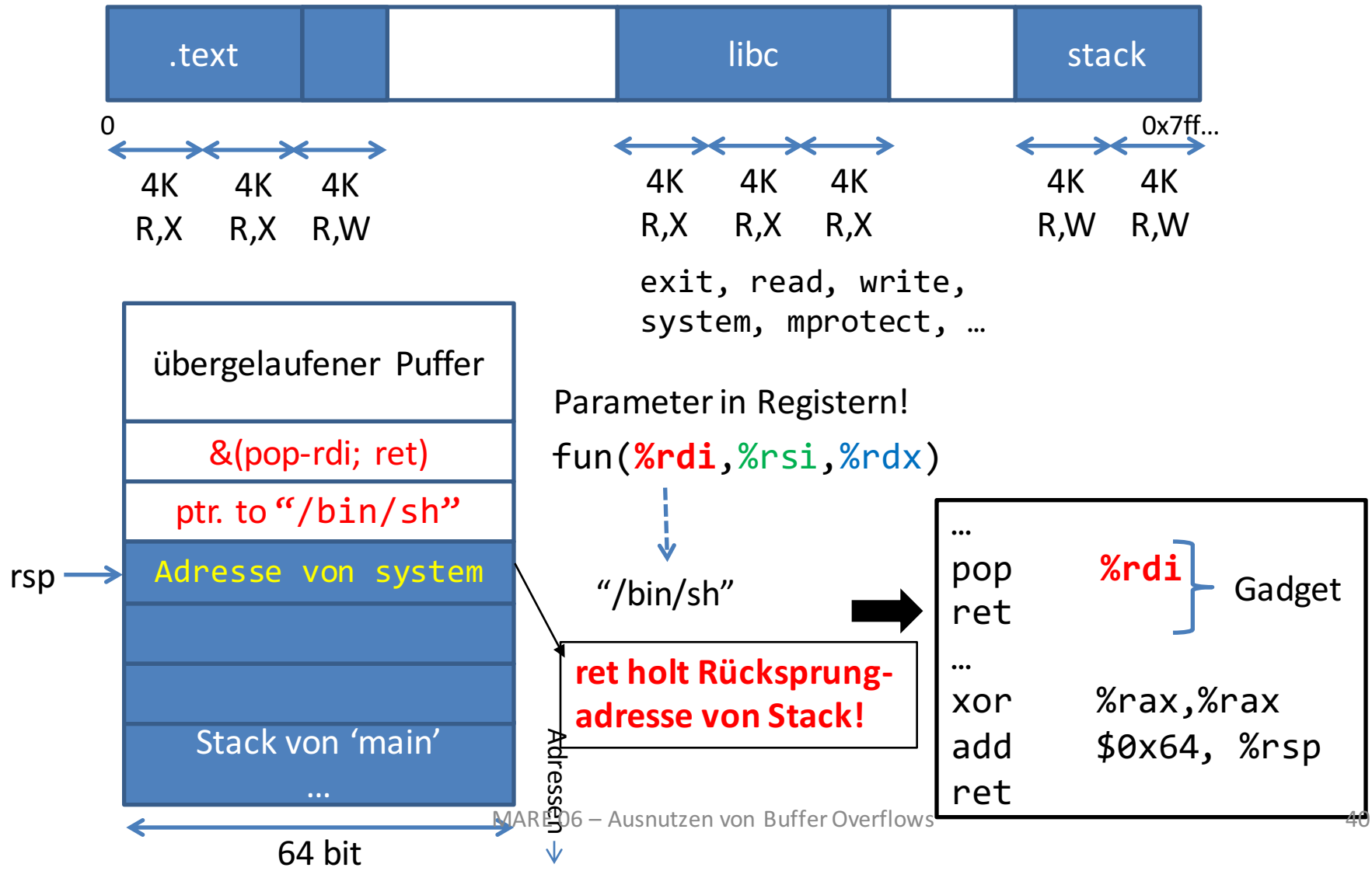
Return-Oriented Programming



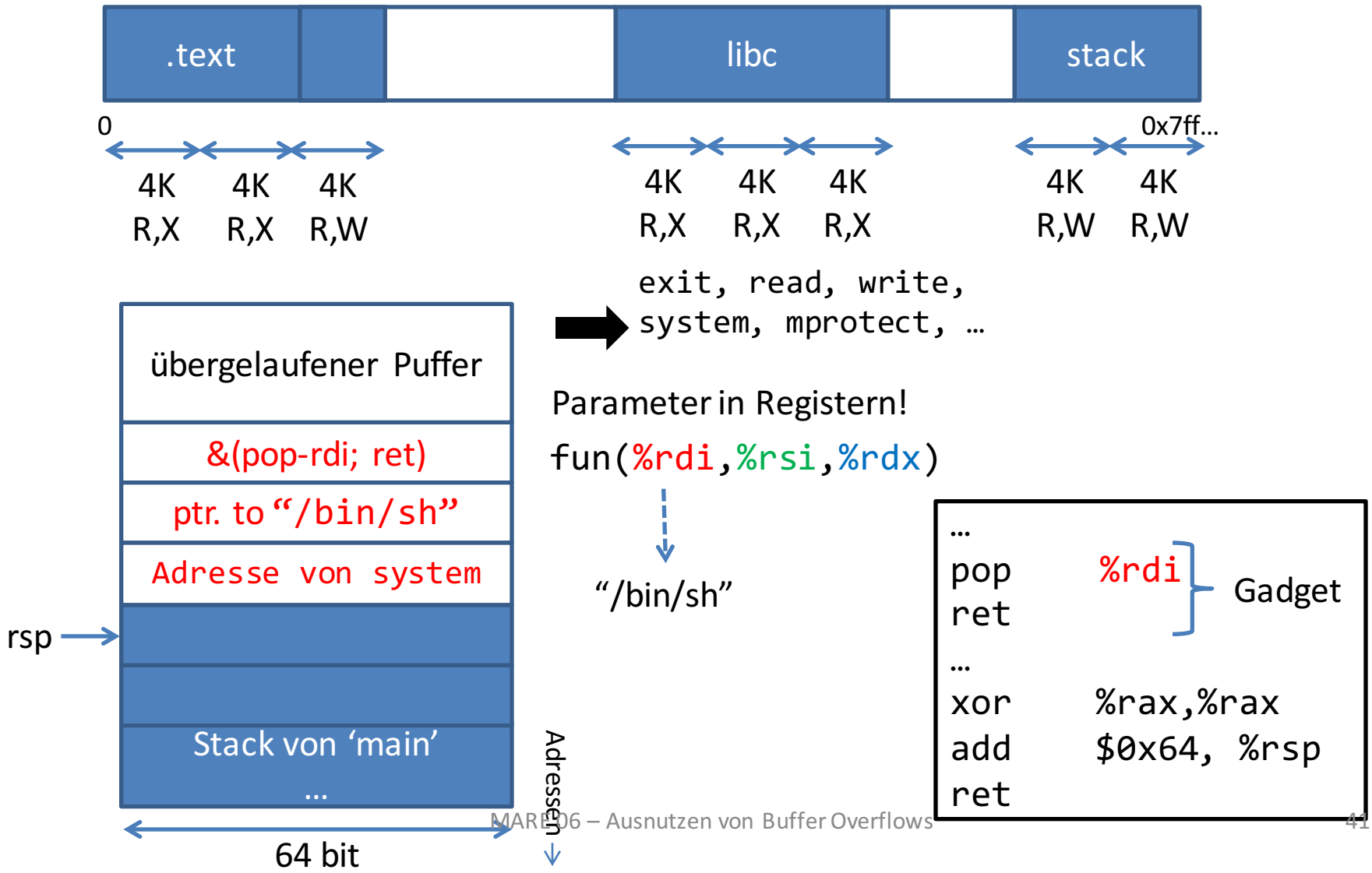
Return-Oriented Programming



Return-Oriented Programming



Return-Oriented Programming

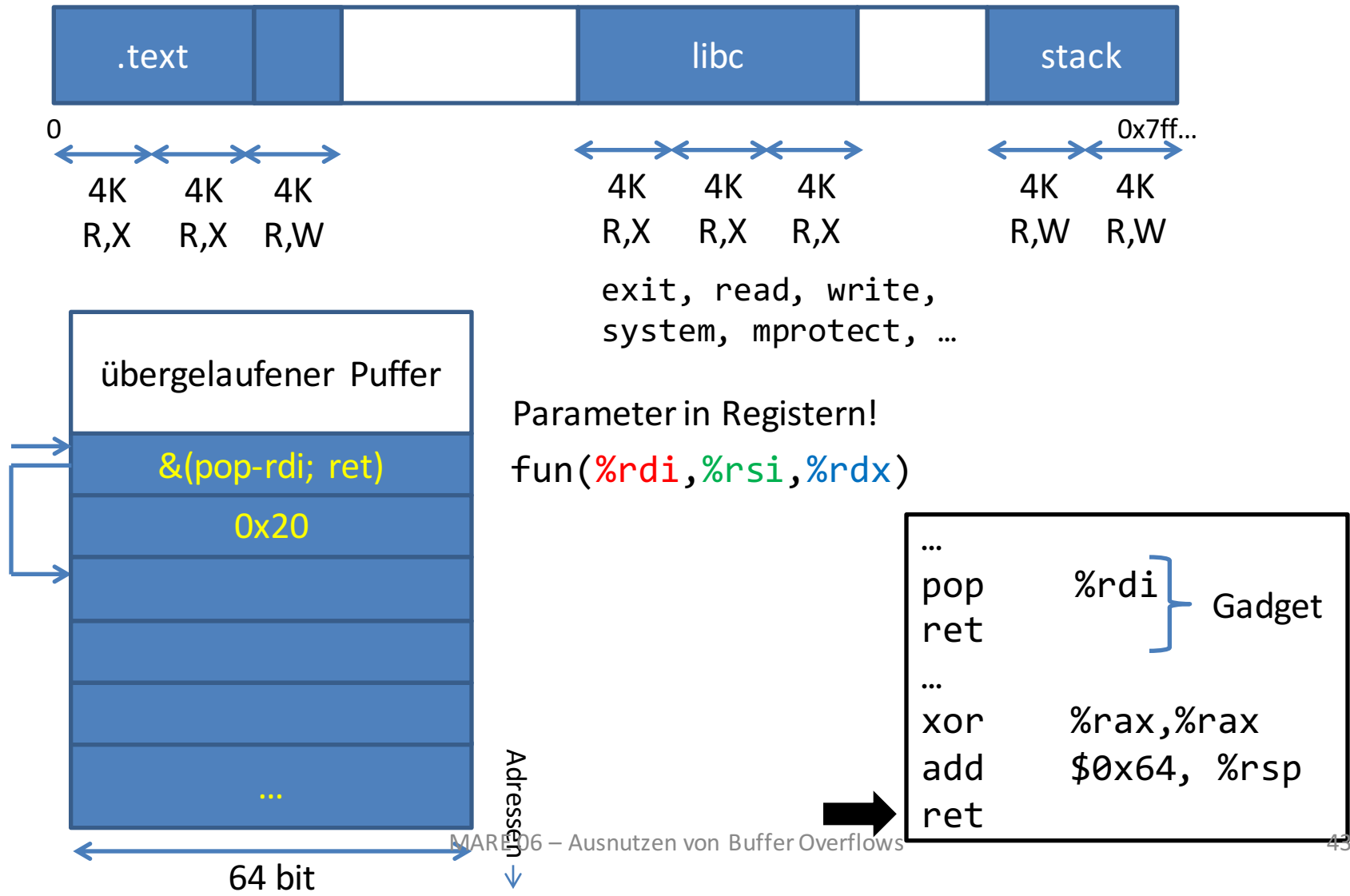


Verketteten von Gadgets

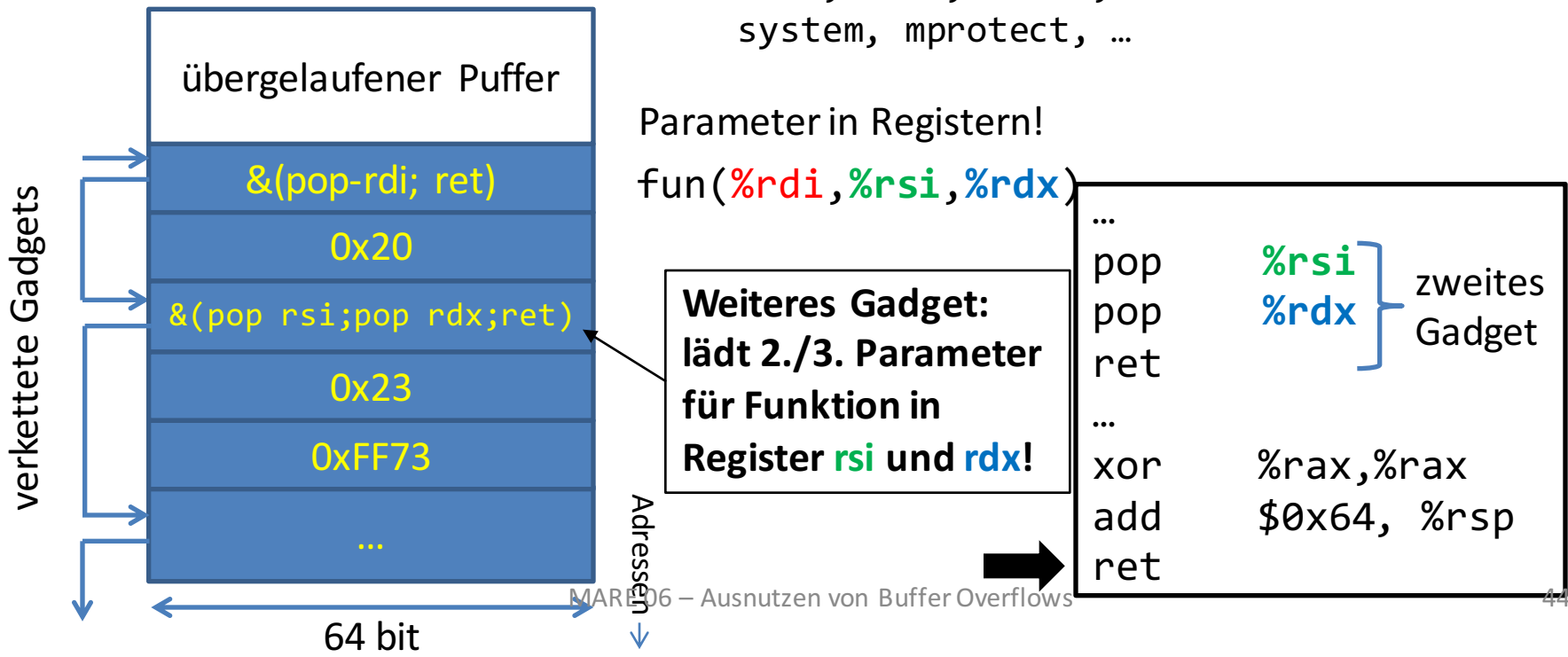
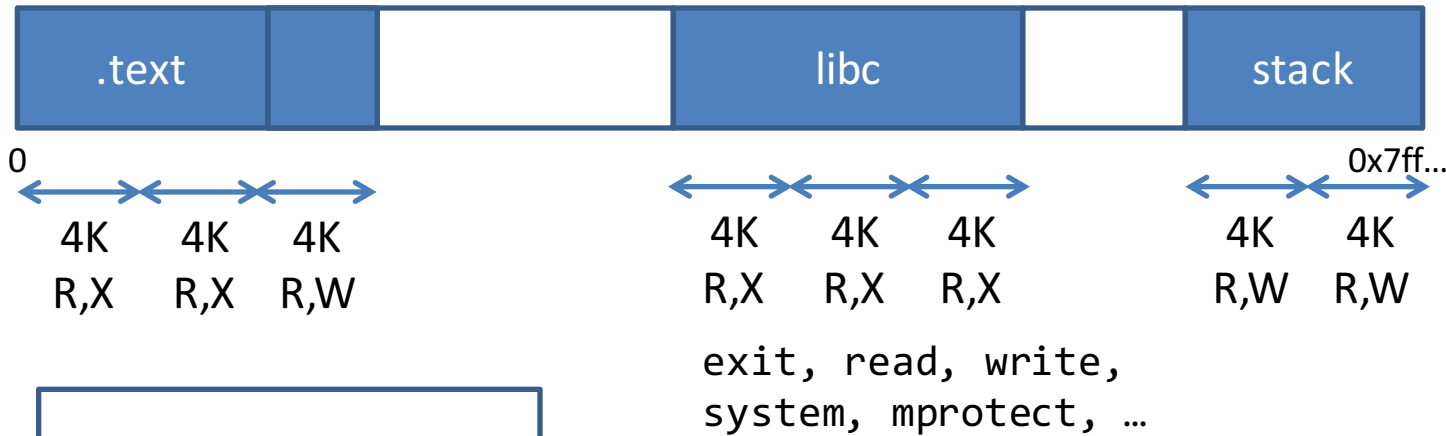
- *Bisher: **ein** Register mit nützlichem Wert geladen...*
 - ...danach: Aufruf der libc-Funktion
- Wie können Funktionen mit mehreren Parametern aufgerufen werden?
 - Nach ret aus Gadget: nächste Adresse wird vom Stack gelesen
 - Also: Adresse von weiterem Gadget auf Stack legen...



Return-Oriented Programming



Return-Oriented Programming



Finden von Gadgets (1)

- Wie kann man Gadgets im .text-Segment finden?
 - Assembler-Instruktionen liegen ja nicht in (menschen-)lesbarer Form (also: als Mnemonics) im Programmspeicher...
 - Suchen nach **Opcodes** der zugehörigen Maschineninstruktionen:

```
0000000000000000 <foo1>:
```

0:	5f	pop	%rdi
1:	c3	retq	

```
0000000000000002 <foo2>:
```

2:	5e	pop	%rsi
3:	5a	pop	%rdx
4:	c3	retq	



Finden von Gadgets (2)

- Suchen nach **Opcodes** der zugehörigen Maschineninstruktionen
 - Oft: Keine passenden Opcodes vorhanden!
- Aber: Instruktionen mit längerer Codierung im Speicher
 - Einzelne Bytes dieser Instruktionen entsprechen benötigten Opcodes!

0x5c

“pop %rsi”

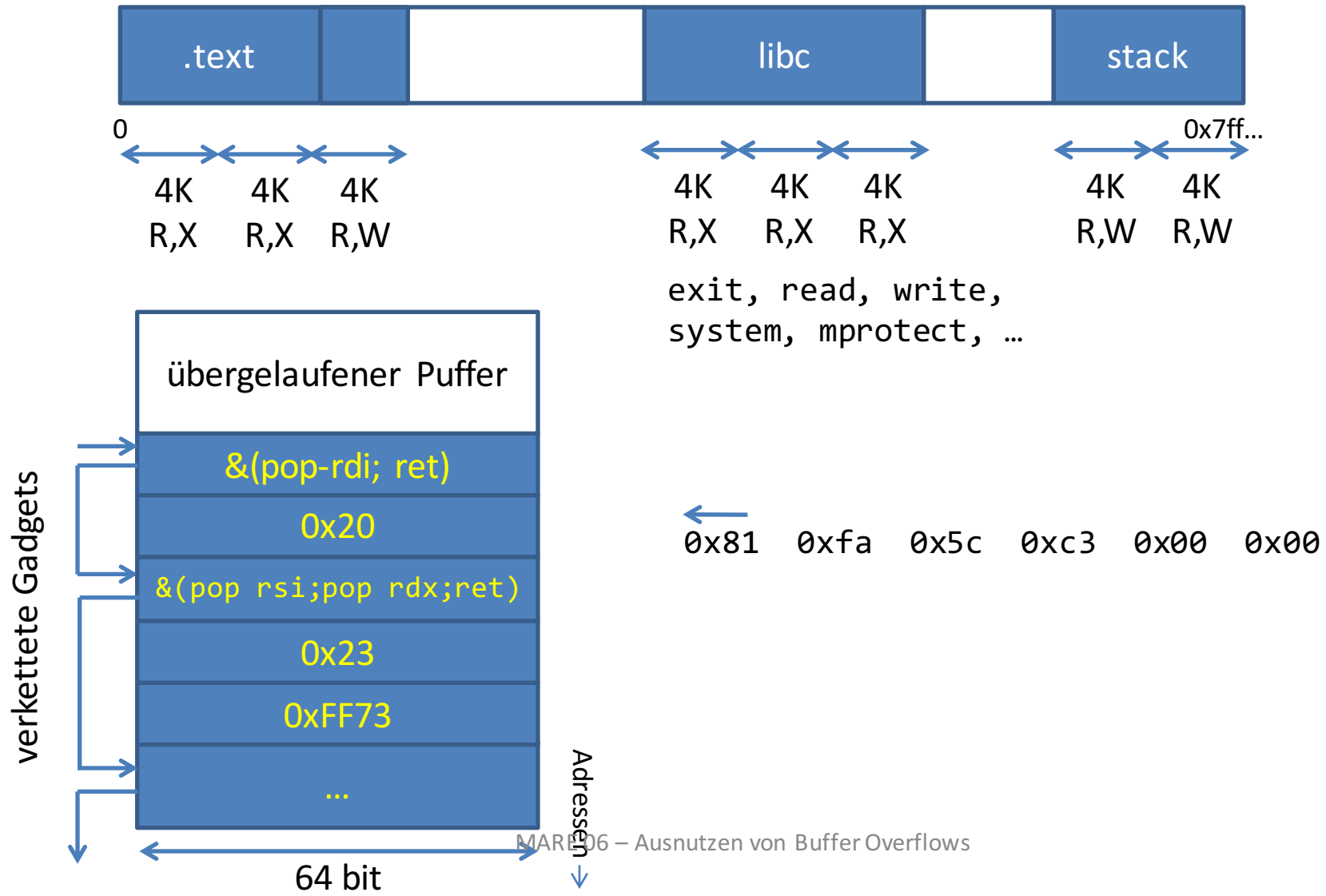
0xc3

“ret”

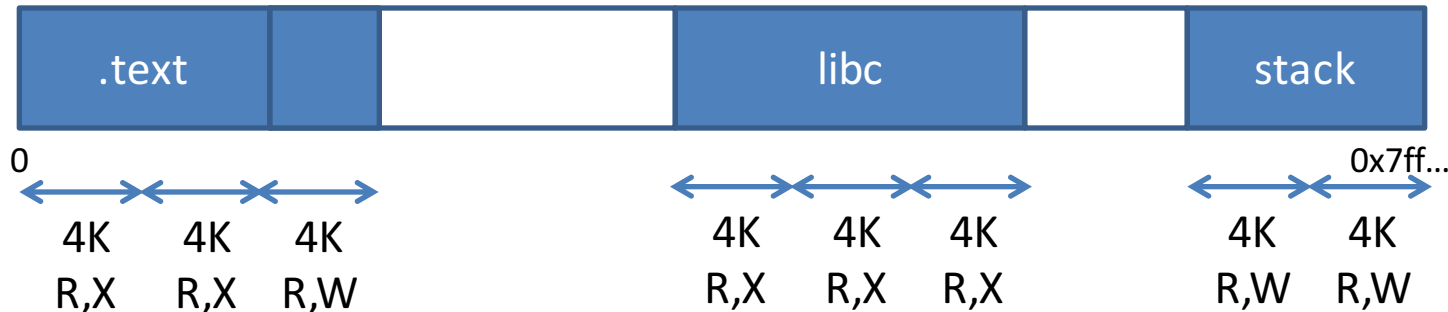
```
0000000000000000 <foo1>:  
0: 81 fa 5c c3 00 00      cmp    $0xc35c,%edx
```



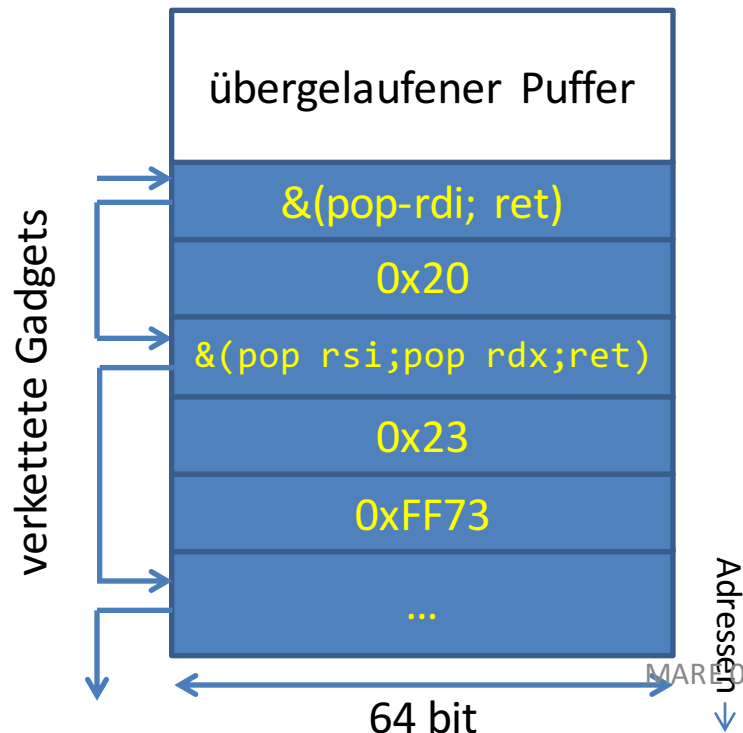
Return-Oriented Programming



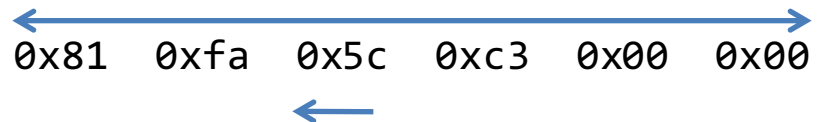
Return-Oriented Programming



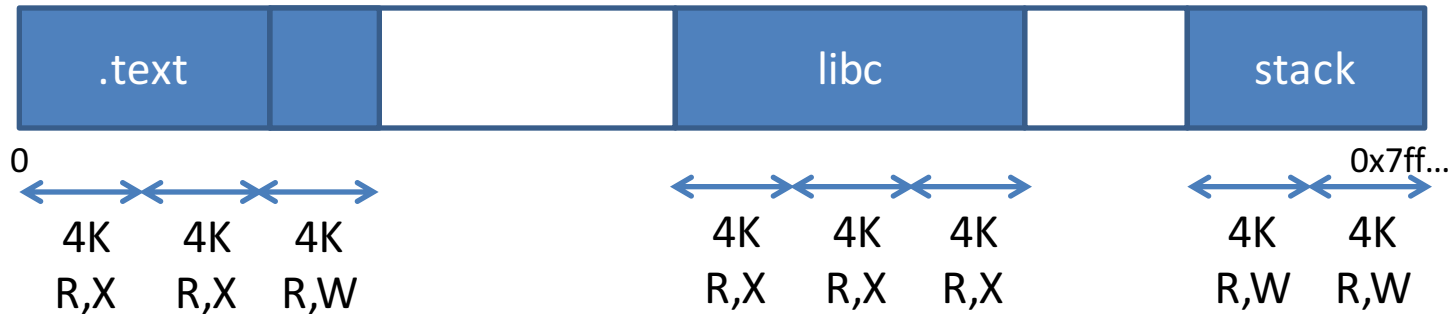
exit, read, write,
system, mprotect, ...



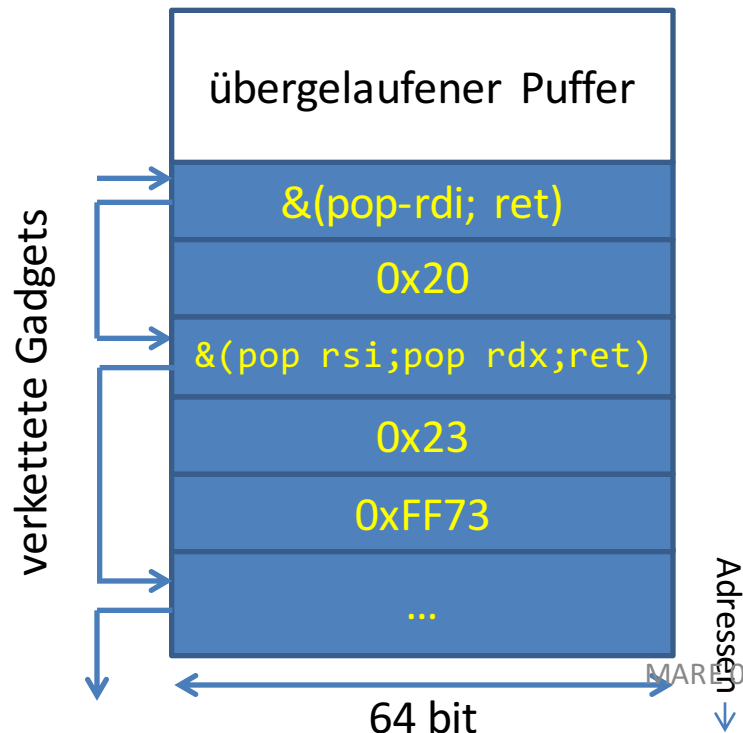
cmp \$0xc35c, %edx



Return-Oriented Programming



exit, read, write,
system, mprotect, ...

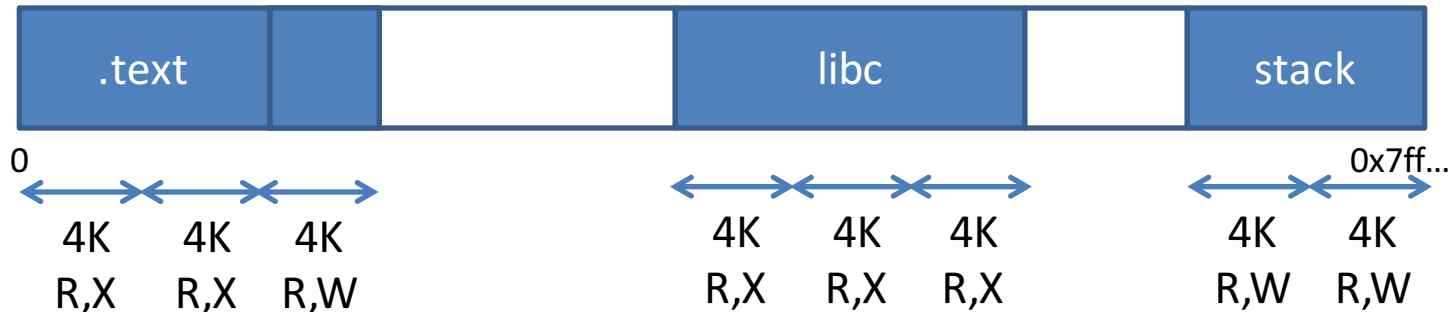


cmp \$0xc35c, %edx

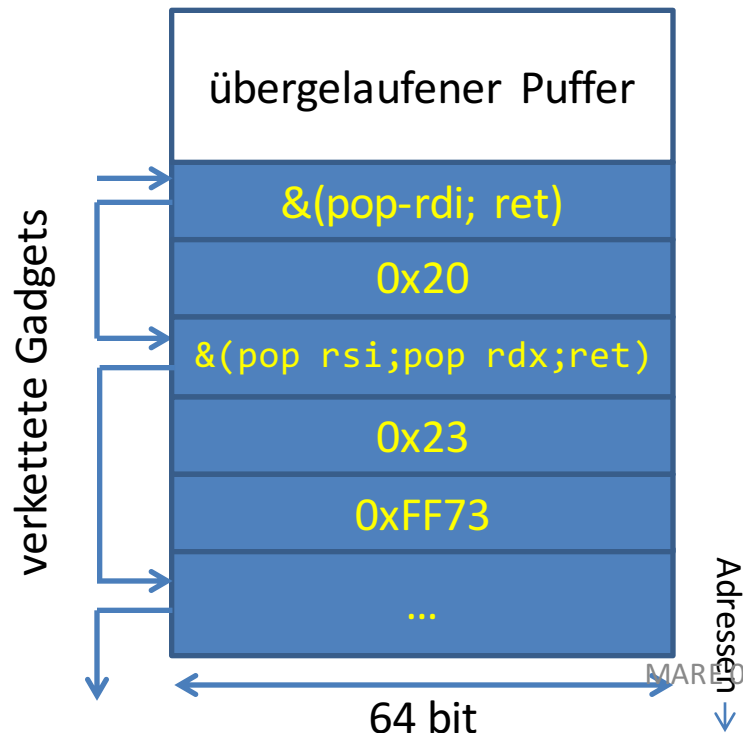
0x81 0xfa 0x5c 0xc3 0x00 0x00

pop %rsp

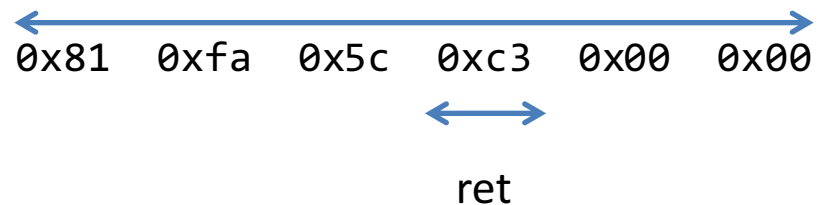
Return-Oriented Programming



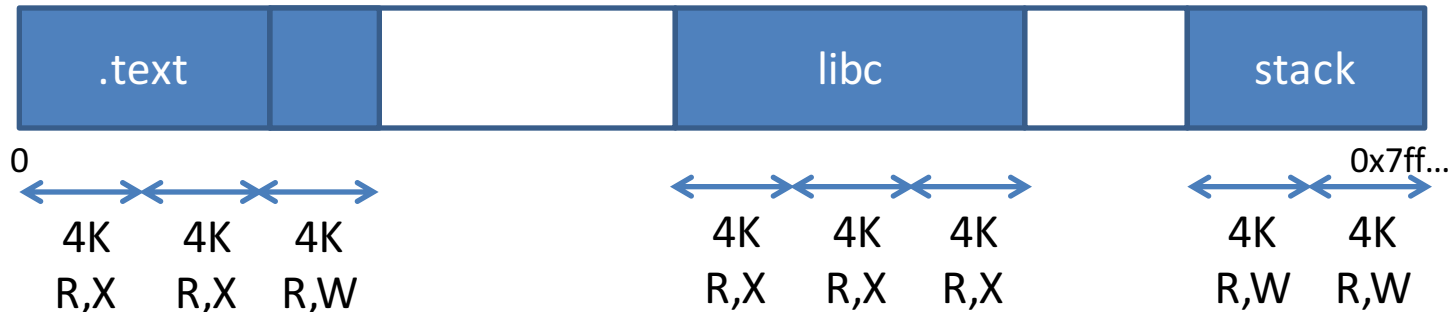
exit, read, write,
system, mprotect, ...



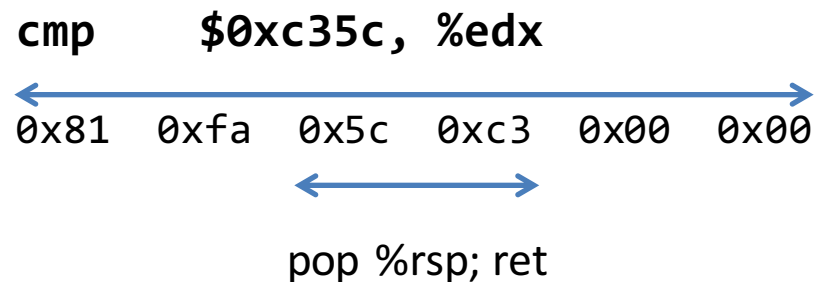
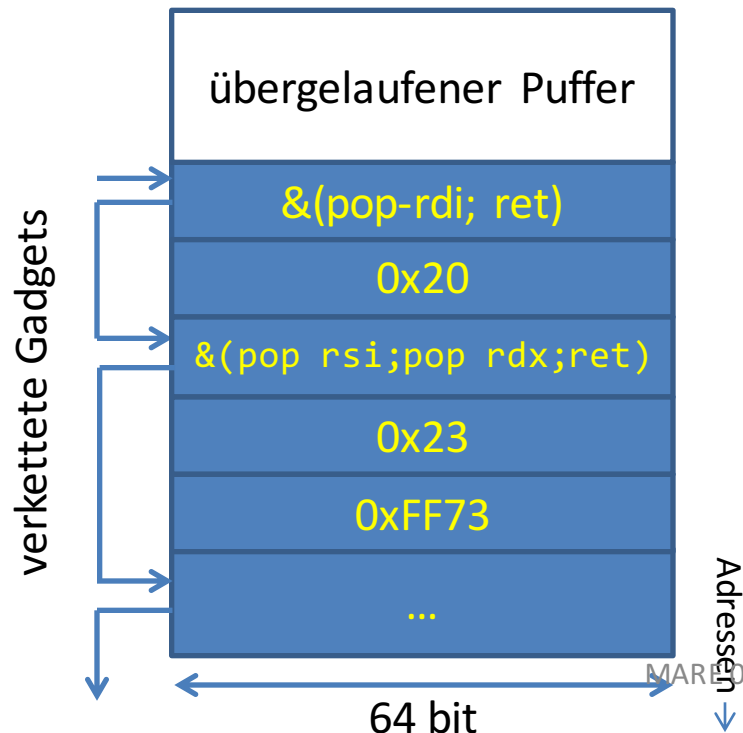
cmp \$0xc35c, %edx



Return-Oriented Programming



exit, read, write,
system, mprotect, ...

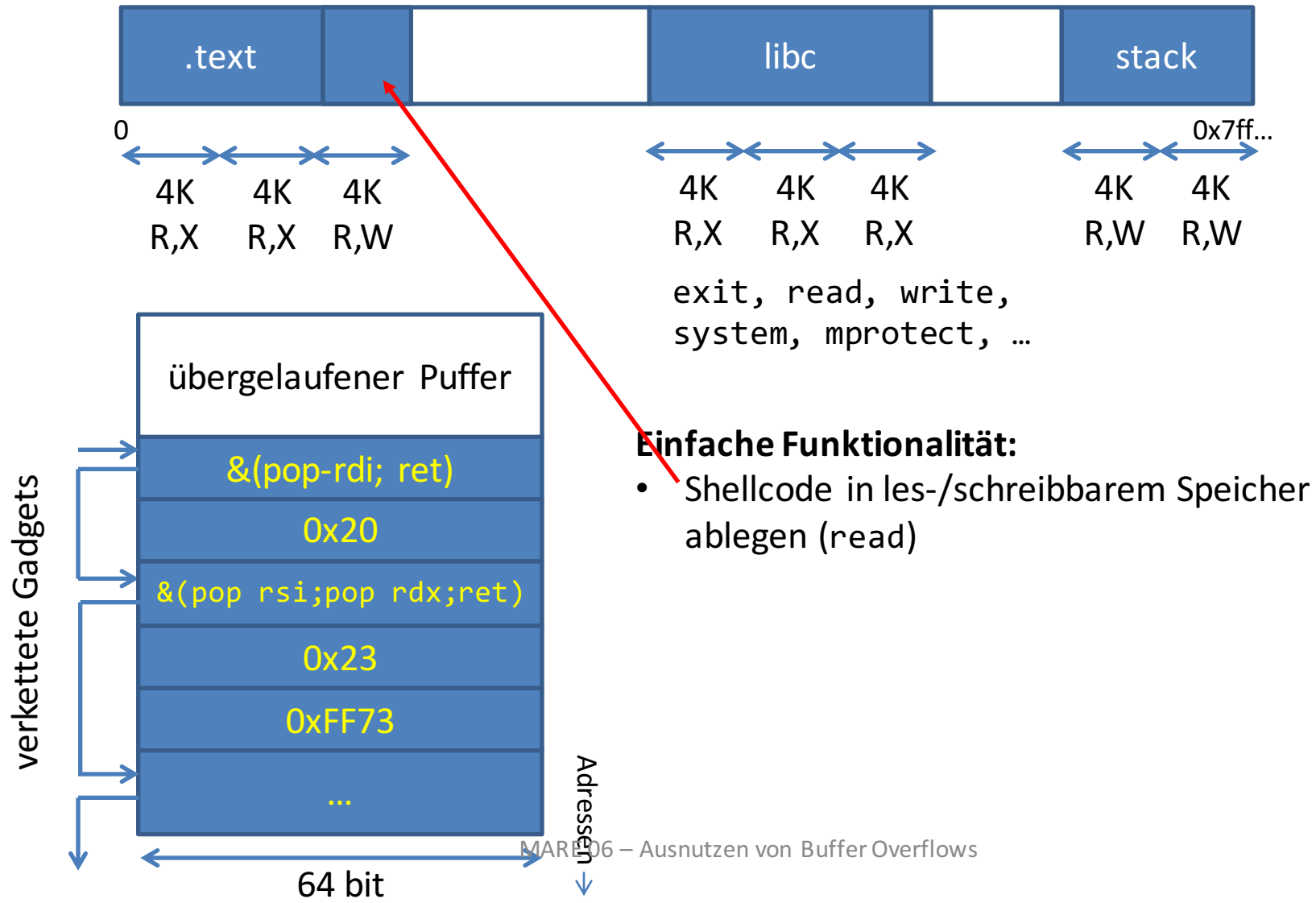


Arten von Gadgets

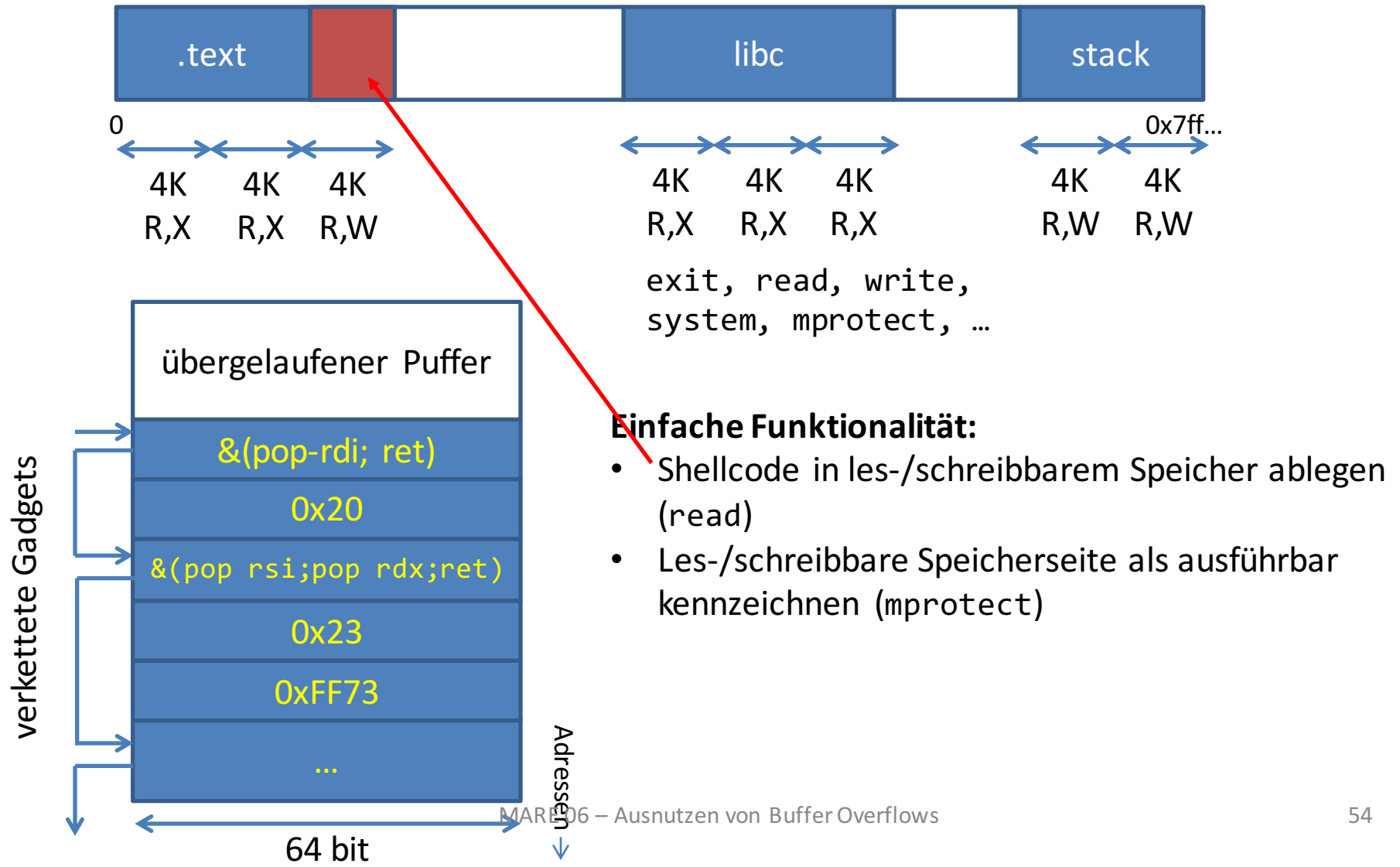
Pop-Instruktionen sind nicht die einzigen nützlichen Gadgets

- Write-anywhere gadgets
`mov %rdi, (%rsi); ret`
- Turing-vollständige Menge von Gadgets erzeugen komplexen Shellcode (resultiert in Aufruf von libc oder Kernel)
- Oft reicht eine einfache Funktionalität:
 - Shellcode in les-/schreibbarem Speicher ablegen (read)
 - Les-/schreibbare Speicherseite als ausführbar kennzeichnen (mprotect)

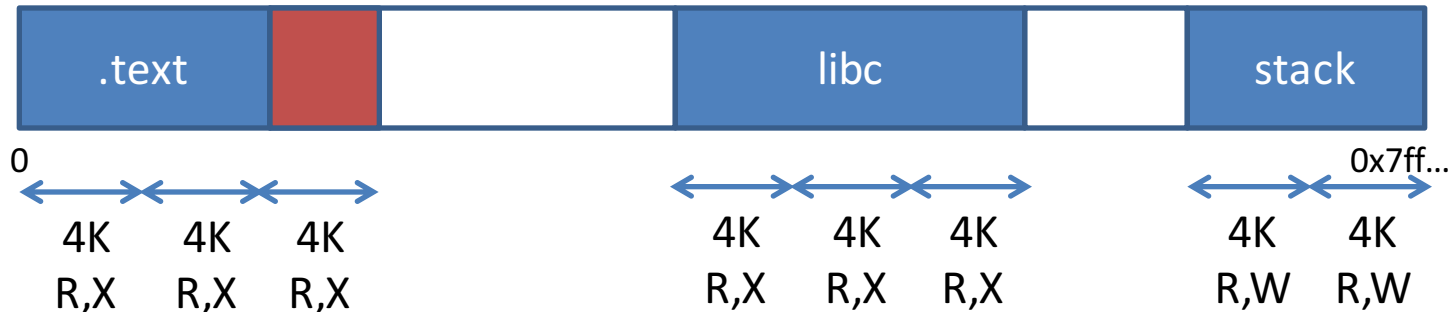
Return-Oriented Programming



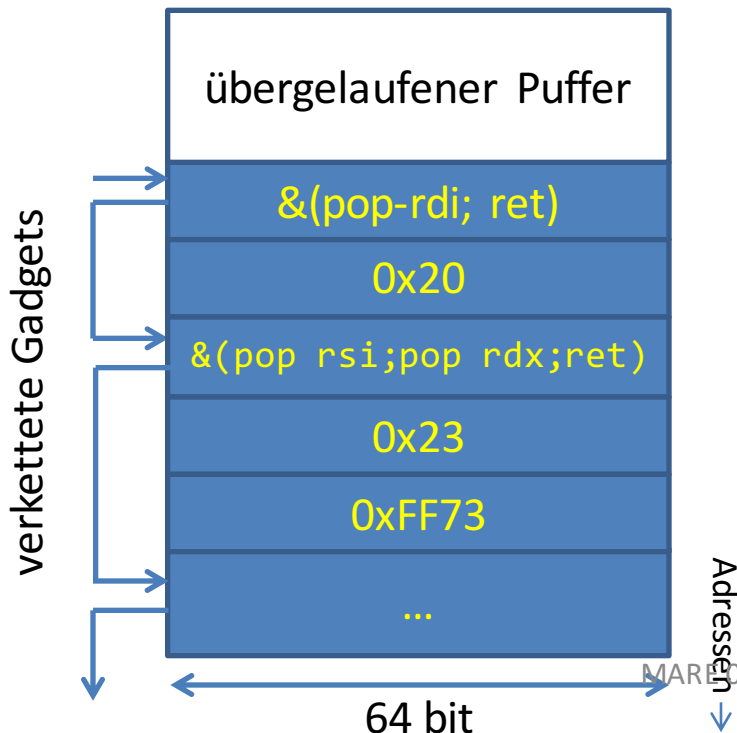
Return-Oriented Programming



Return-Oriented Programming



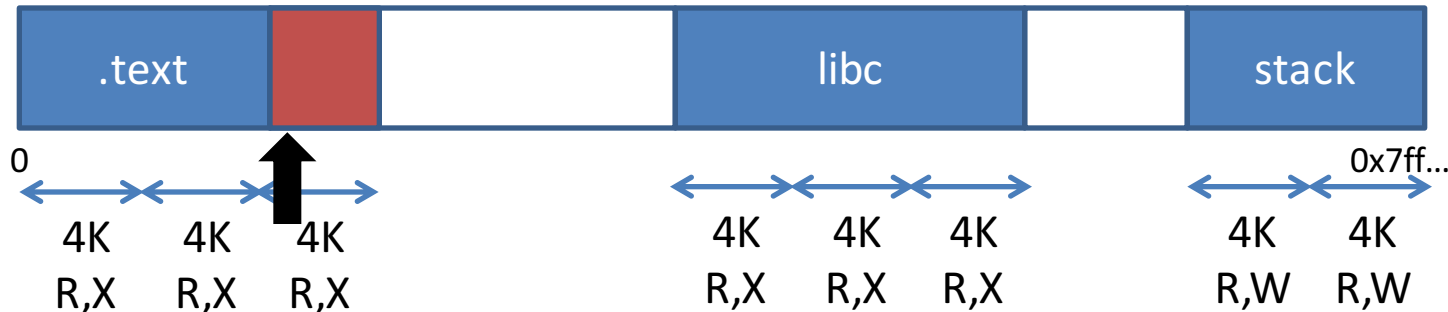
exit, read, write,
system, mprotect, ...



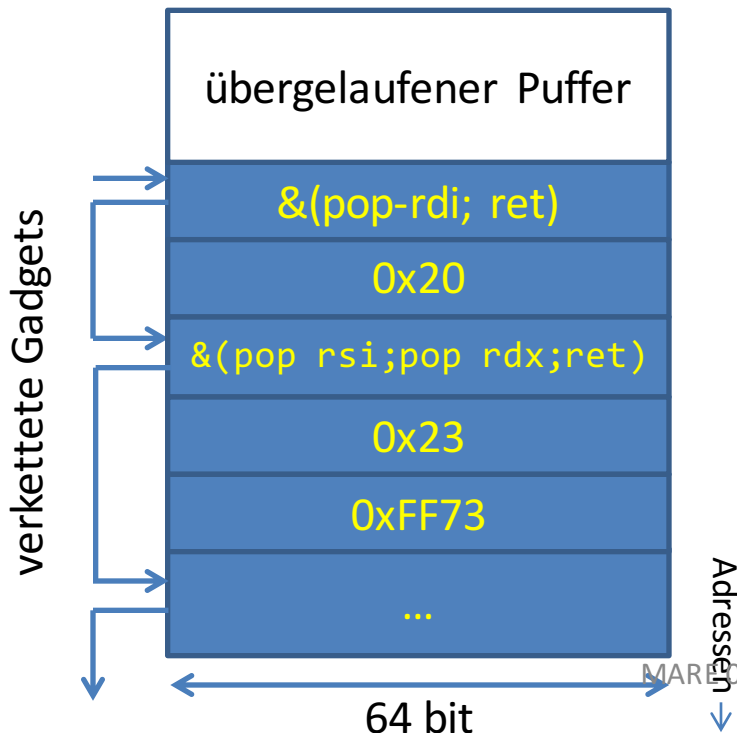
Einfache Funktionalität:

- Shellcode in les-/schreibbarem Speicher ablegen (read)
- Les-/schreibbare Speicherseite als ausführbar kennzeichnen (mprotect)
- 'Rücksprung' zum jetzt ausführbaren Shellcode

Return-Oriented Programming



exit, read, write,
system, mprotect, ...



Einfache Funktionalität:

- Shellcode in les-/schreibbarem Speicher ablegen (read)
- Les-/schreibbare Speicherseite als ausführbar kennzeichnen (mprotect)
- 'Rücksprung' zum jetzt ausführbaren Shellcode

Fazit

Erste Schutzmaßnahme ist überwunden...

- Return-to-libc nützlich, um Systemfunktionen aufzurufen
- Parameter für Funktionen bei 64-Bit x86 nicht auf Stack übergebbar (sondern in Registern)
- Return-Oriented Programming (ROP) hilft, Parameter vom Stack in Register zu schreiben
- Komplexere ROP-Programme ermöglichen das Ausführen von nachgeladenem Shellcode aus ursprünglich nicht ausführbaren Speicherseiten
- ROP kann Turing-vollständig sein (siehe Literatur)

Literatur

ROP

- Shacham et al., „*Return-Oriented Programming: Exploits Without Code Injection*“, https://www.blackhat.com/presentations/bh-usa-08/Shacham/BH_US_08_Shacham_Return_Oriented_Programming.pdf
- Nicholas Carlini and David Wagner, „*ROP is still dangerous: breaking modern defenses*“, In Proceedings of the 23rd USENIX Conference on Security (SEC'14) <https://www.usenix.org/node/184508>
- Turing-vollständiger ROP-Compiler: <https://github.com/pakt/ropc>
<http://css.csail.mit.edu/6.858/2017/projects/je25365-ve25411.pdf>
- Andrei Homescu et al., „*Microgadgets: size does matter in turing-complete return-oriented programming*“, In Proceedings of the 6th USENIX conference on Offensive Technologies (WOOT'12) <https://www.usenix.org/system/files/conference/woot12/woot12-final9.pdf>

Shellcode

- Heasman et al., „*The Shellcoder's Handbook: Discovering and Exploiting Security Holes*“, 2nd Edition, Wiley 2007, ISBN-13 978-0470080238