Malware-Analyse und Reverse Engineering

5: Buffer Overflows

20.4.2017

Prof. Dr. Michael Engel

Basierend auf Unterlagen von Bart Coppens

https://www.bartcoppens.be/

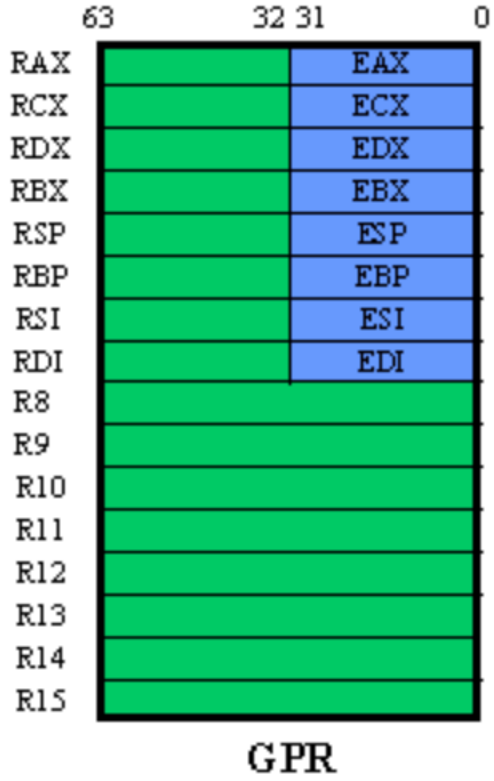
Überblick

Themen:

- Nochmal Stack frames und Funktionsaufrufe
- Implementierung von Buffern in x86-Assembler
- Buffer overflows:
  - Ursache
  - Ausnutzen: Returnadressen, Return into libc, Return-oriented programming
  - Schutz: Stack canaries, DEP (W^X), ...

Funktionsaufruf: 32- vs. 64-bit x86 (1)

HOCHSCHULE COBURG



Die 64-Bit-Variante der x86-Architektur besitzt 8 weitere (General Purpose) Register

- x86-32: eax, ebx, ecx, edx, ebp, esp, esi, edi
- x86-64: **r**ax, **r**bx, **r**cx, **r**dx, **r**bp, **r**sp, **r**si, **r**di, *r8, r9, r10, r11, r12, r13, r1*
  - „r“-Prefix: 64 bit breite Register

- **Folge:**
  - Bei 64 bit sind genügend freie Register vorhanden, um Parameter zu übergeben
  - Erinnerung: Übergabe bei 32-bit x86 auf dem Stack!

Funktionsaufruf: 32- vs. 64-bit (2)

HOCHSCHULE COBURG

Assemblercode für Funktionsaufruf:

```
long myfunc(long a, long b, long c, long d,
            long e, long f, long g, long h)
{   long xx = a * b * c * d * e * f * g * h;
    long yy = a + b + c + d + e + f + g + h;
    long zz = utilfunc(xx, yy, xx % yy);
    return zz + 20;
}

Aufruf: myfunc(1,2,3,4,5,6,7,8);
```

32 bit

```
main: pushl   %ebp
      movl    %esp, %ebp
      pushl   $8
      pushl   $7
      pushl   $6
      pushl   $5
      pushl   $4
      pushl   $3
      pushl   $2
      pushl   $1
      call    myfunc
      addl    $32, %esp
```

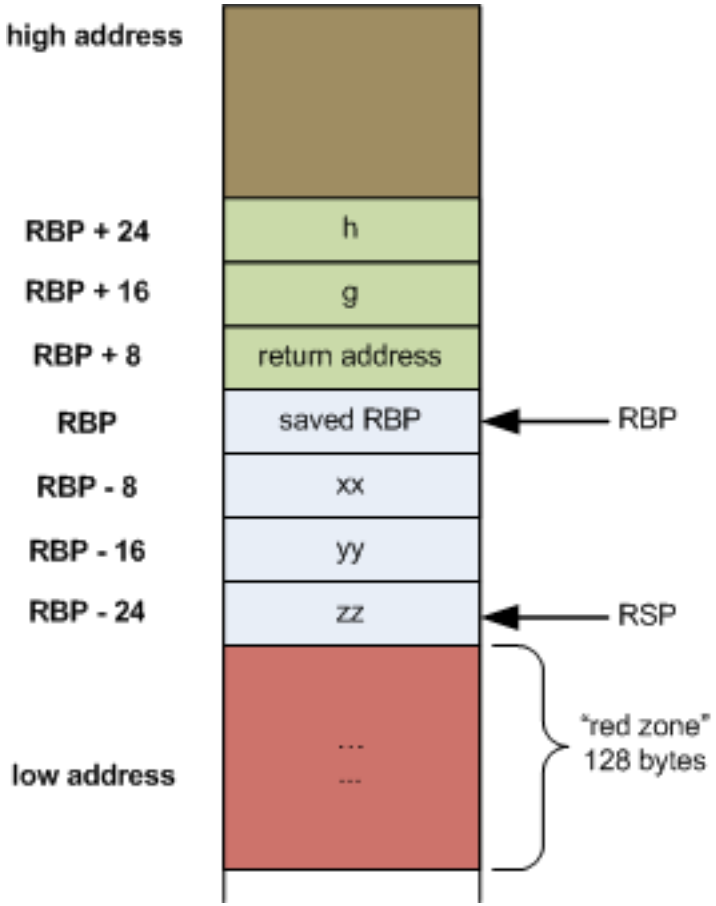
Weitere Parameter auf Stack

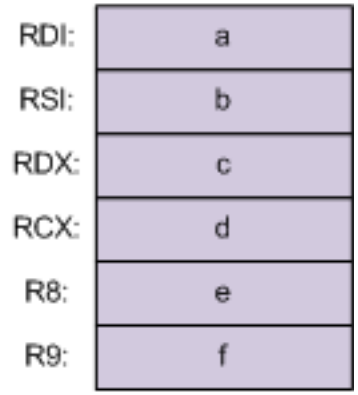
Parameter 1-6 in Reg. edi, esi, edx, ecx, r8, r9

64 bit

```
main: pushq   %rbp
      movq    %rsp, %rbp
      pushq   $8
      pushq   $7
      movl    $6, %r9d
      movl    $5, %r8d
      movl    $4, %ecx
      movl    $3, %edx
      movl    $2, %esi
      movl    $1, %edi
      call    myfunc
      addq    $16, %rsp
```

Aufbau Stackframes bei x86-64 (1)



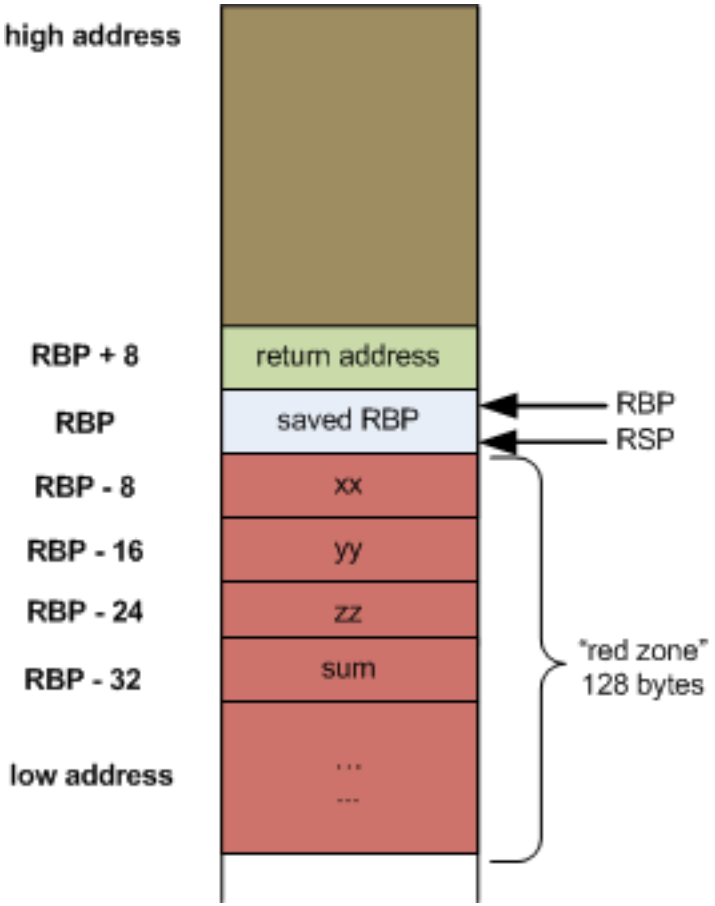


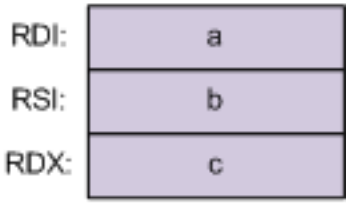
„Red zone“ kann lokale Variable enthalten, ohne RSP anzupassen

- Nützlich nur für „leaf functions“

```
long myfunc(long a, long b, long c, long d,
            long e, long f, long g, long h)
{   long xx = a * b * c * d * e * f * g * h;
    long yy = a + b + c + d + e + f + g + h;
    long zz = utilfunc(xx, yy, xx % yy);
    return zz + 20;
}

Aufruf: myfunc(1,2,3,4,5,6,7,8);
```

Aufbau Stackframes bei x86-64 (2)





„utilfunc“ ist eine leaf function

- Ruft keine weiteren Funktionen auf
- Nutzt „red zone“

```
long utilfunc(long a, long b, long c)
{
    long xx = a + 2;
    long yy = b + 3;
    long zz = c + 4;
    long sum = xx + yy + zz;

    return xx * yy * zz + sum;
}
```

Notwendigkeit des Base Pointers

Compiler-erzeugter Code benötigt nicht unbedingt Base Pointer

- Compiler kann Offsets vom SP zum Variablenzugriff berechnen
- gcc-Option "-fomit-frame-pointer"/"-f**no**-omit-frame-pointer"
  - Standard: kein Basepointer, wenn Optimierungen (-Ox) aktiv

Parameter -fno**-omit-frame-pointer:**

64 bit **mit** rbp

```
myfunc: pushq   %rbp
        movq    %rsp, %rbp
        subq    $80, %rsp
        movq    %rdi, -40(%rbp)
        movq    %rsi, -48(%rbp)
        movq    %rdx, -56(%rbp)
        movq    %rcx, -64(%rbp)
        movq    %r8, -72(%rbp)
        movq    %r9, -80(%rbp)
```

Parameter -fomit-frame-pointer:

64 bit **ohne** rbp

```
myfunc: subq    $80, %rsp
        movq    %rdi, 40(%rsp)
        movq    %rsi, 32(%rsp)
        movq    %rdx, 24(%rsp)
        movq    %rcx, 16(%rsp)
        movq    %r8, 8(%rsp)
        movq    %r9, (%rsp)
```

Speichern der übergebenen Parameter in lokalen Variablen

C und Probleme mit dem Stackaufbau

C verwendet aus Effizienzgründen keine Überprüfung von Arraygrenzen bei Lese-/Schreibzugriffen

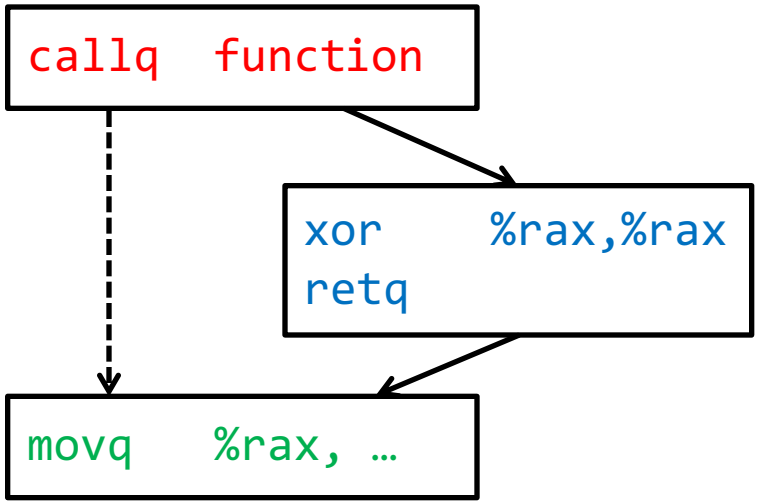
- Strings sind nur Arrays von char, mit Nullbyte terminiert
- Folge: Zugriffe über das Ende von Arrays hinaus möglich

Viele unsichere libc-Funktionen prüfen nicht auf Anzahl eingegebener Zeichen, z.B.:

- strcpy (char *dest, const char *src)
- strcat (char *dest, const char *src)
- gets (char *s)
- scanf (const char *format, …)

Normales Programmverhalten

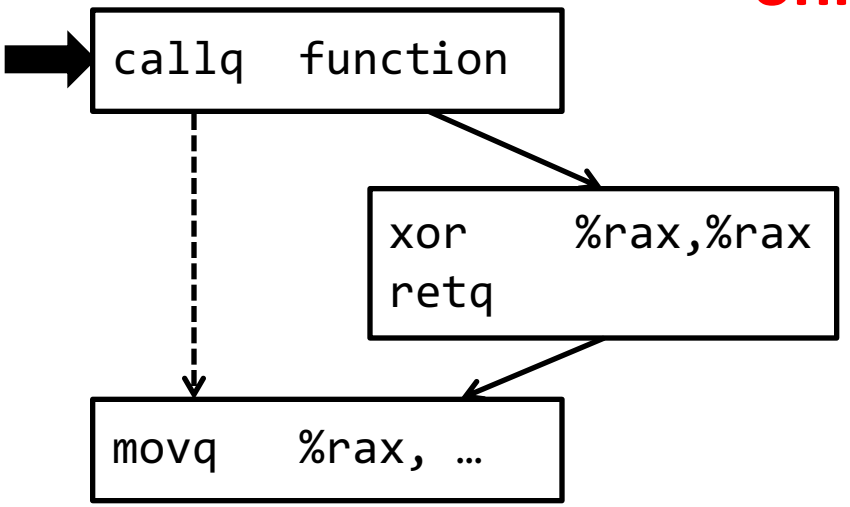
```
int function() {
  return 0;
}

…
int i = function();
…
```

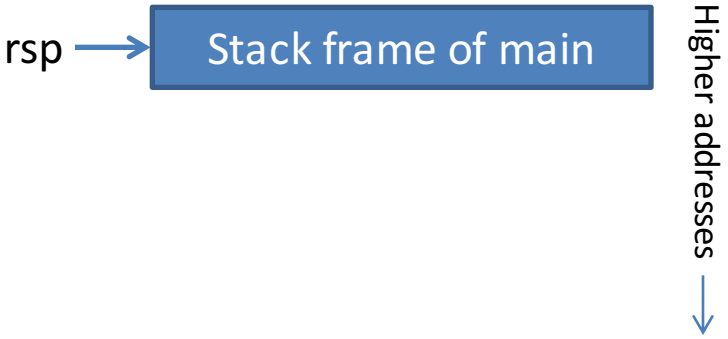


Normales Programmverhalten

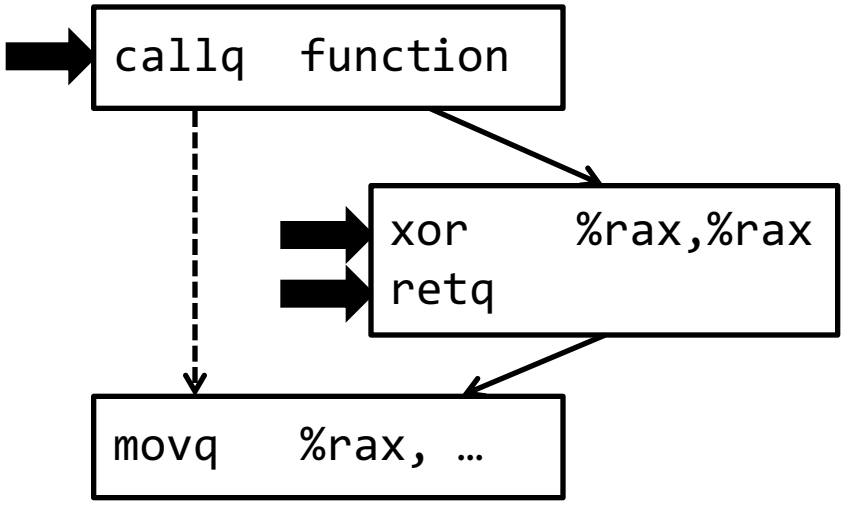
64 bit-Code

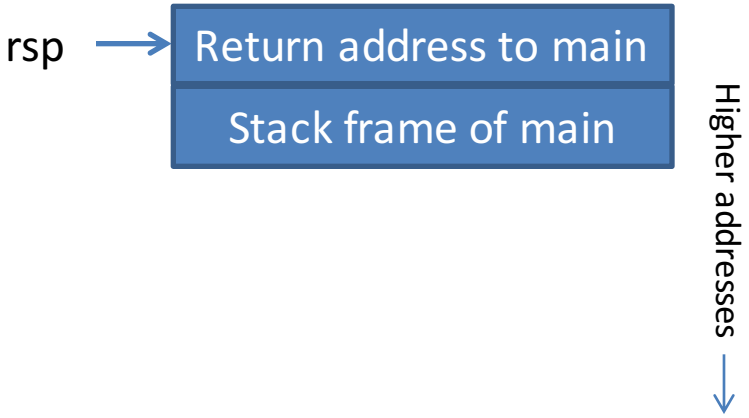
ohne rbp



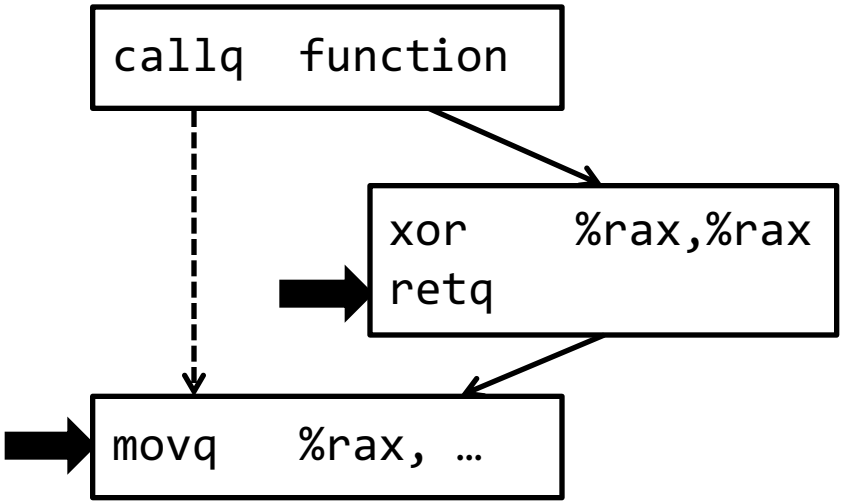


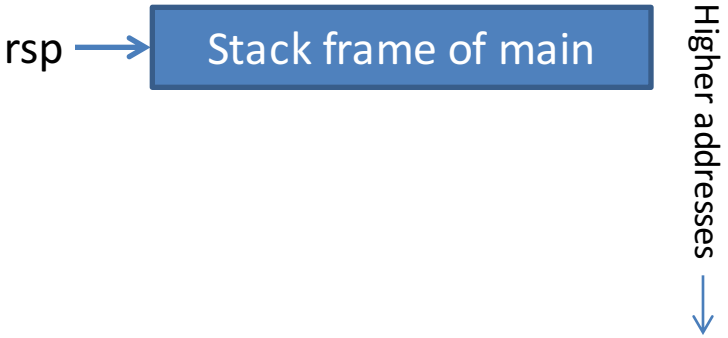
Normales Programmverhalten





Normales Programmverhalten





Lokale Variablen und der Stack

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

int main() {
 return silly_function(50, "Hello\n");
}
```

function(%rdi, %rsi, %rdx)

Wir übergeben eine Länge von 50 Zeichen und einen kürzeren String

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

Lokale Variablen und der Stack

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

int main() {
 return silly_function(50, "Hello\n");
}
```

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

%rdi ist Adresse von “buffer”, der lokal 100 Bytes auf dem Stack belegt (ab %esp)

Übergabekonvention x86-64:

Parameter 1-6 in Registern
edi, esi, edx, ecx, r8, r9;
der evtl. Rest auf dem Stack.

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

Lokale Variablen und der Stack

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

int main() {
 return silly_function(50, "Hello\n");
}
```

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

%rsi ist Adresse von “src”, diese wurde schon als 2. Parameter in %rsi an die Funktion übergeben und muss daher nicht umkopiert werden

Übergabekonvention x86-64:

Parameter 1-6 in Registern
edi, esi, edx, ecx, r8, r9;
der evtl. Rest auf dem Stack.

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

Lokale Variablen und der Stack

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

int main() {
 return silly_function(50, "Hello\n");
}
```

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

%rdx enthält die Länge “len”, diese wurde als 1. Parameter in %rdi an die Funktion übergeben und muss daher für memcpy in den 3. Parameter %rdx umkopiert werden.

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

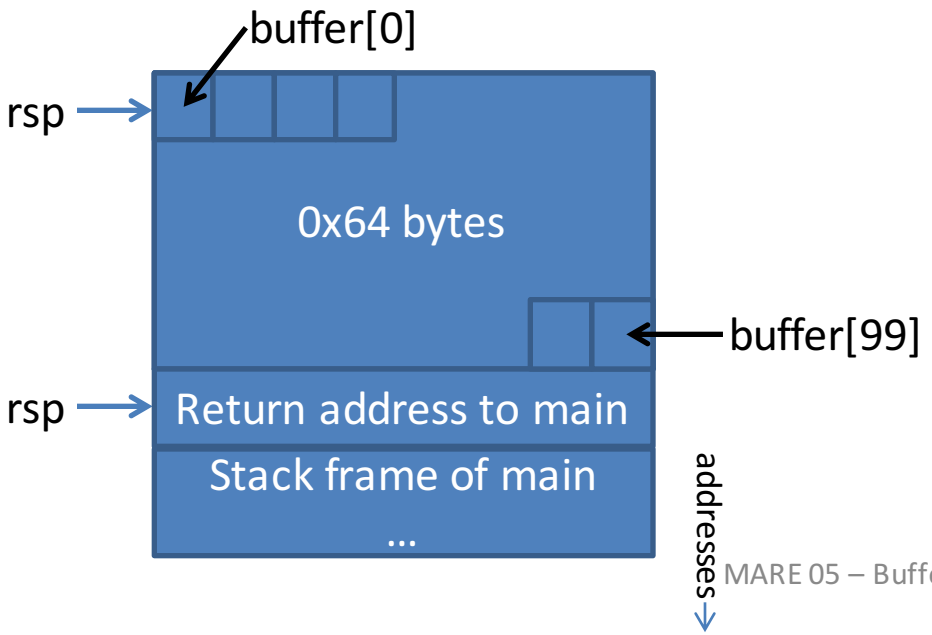
Übergabekonvention x86-64:

Parameter 1-6 in Registern
edi, esi, edx, ecx, r8, r9;
der evtl. Rest auf dem Stack.

Lokale Variablen und der Stack

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

int main() {
 return silly_function(50, "Hello\n");
}
```

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



Der Compiler erzeugt Code, der 0x64 (=100 dez.) Bytes von %rsp subtrahiert, um Platz für buffer zu schaffen.

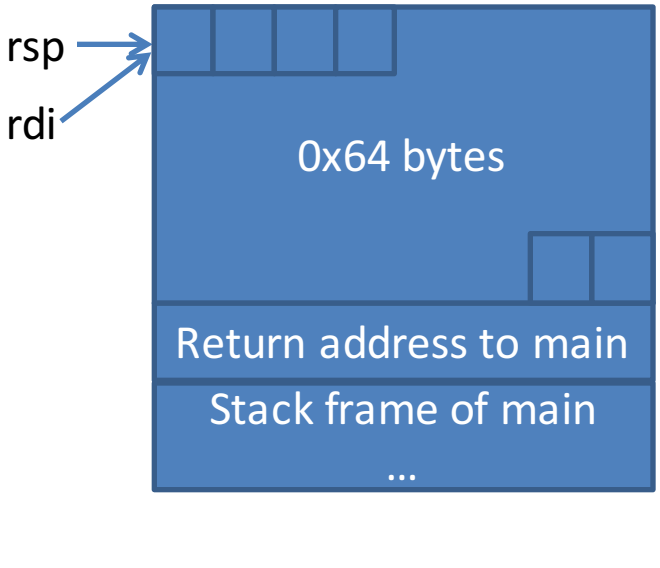
```
sub     $0x64, %rsp
movq    %rdi, %rdx
movq    %rsp, %rdi
callq   <memcpy>
xor     %rax,%rax
add     $0x64, %rsp
retq
```

Lokale Variablen und der Stack

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

int main() {
 return silly_function(50, "Hello\n");
}
```

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

%rsp zeigt auf den Anfang von "buffer", wird zur Verwendung als 1. Parameter von "malloc" in %rdi kopiert.

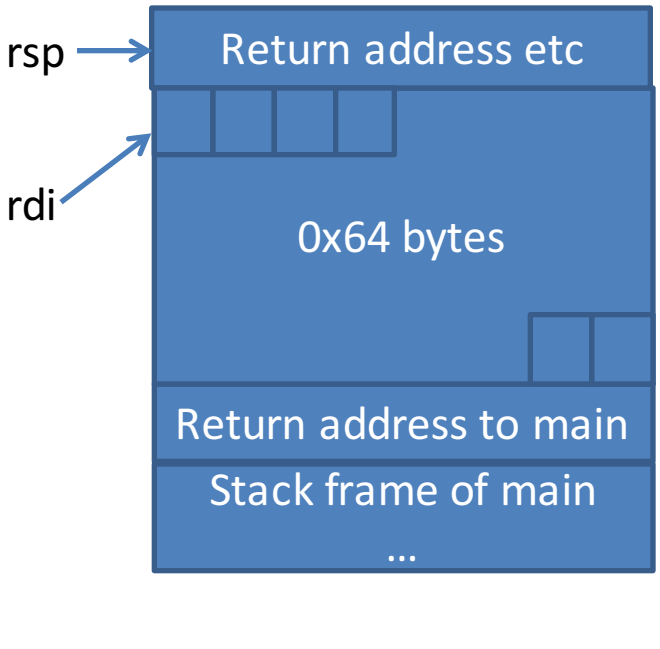



```
sub    $0x64, %rsp
movq   %rdi, %rdx
movq   %rsp, %rdi
callq  <memcpy>
xor    %rax,%rax
add    $0x64, %rsp
retq
```

Lokale Variablen und der Stack

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

int main() {
 return silly_function(50, "Hello\n");
}
```

function(%rdi, %rsi, %rdx)



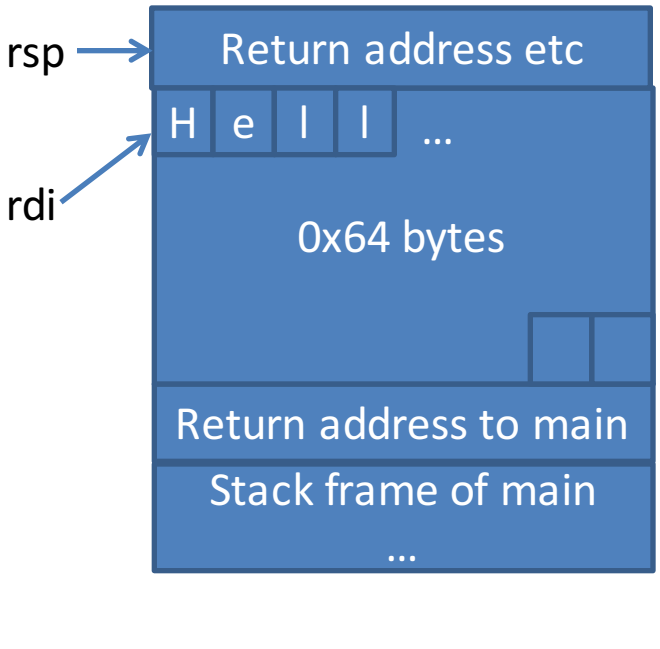
Aufruf der Funktion "memcpy" legt Rücksprung-adresse auf den Stack

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

Lokale Variablen und der Stack

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

int main() {
 return silly_function(50, "Hello\n");
}
```

function(%rdi, %rsi, %rdx)



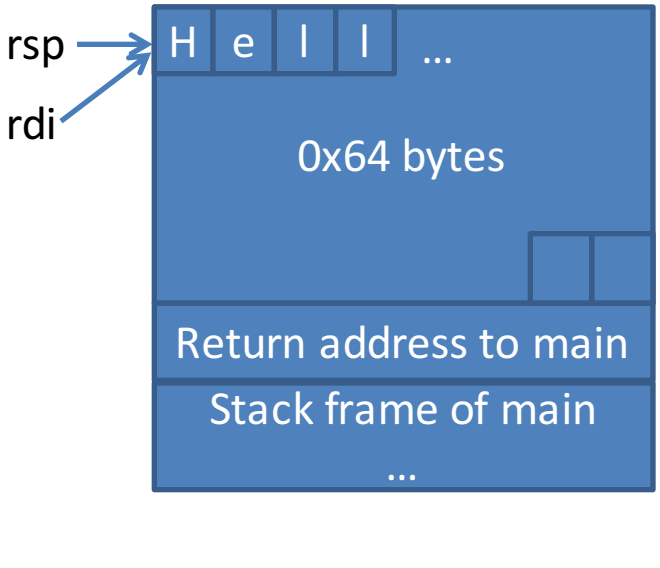
"memcpy" kopiert Speicher von "src" nach "buffer" (%rdx Bytes)

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

Lokale Variablen und der Stack

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

int main() {
 return silly_function(50, "Hello\n");
}
```

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



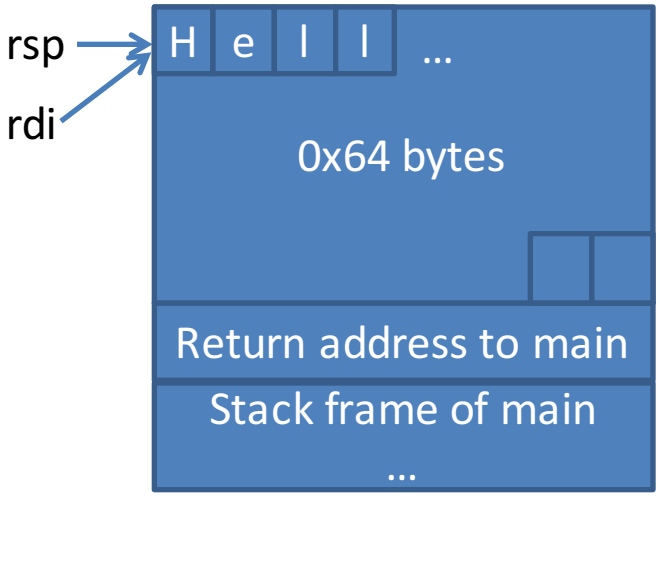
"memcpy" ist zurückgekehrt, Returnadresse wurde von Stack geholt. Returnwert in %rax wird auf 0 gesetzt

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

Lokale Variablen und der Stack

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

int main() {
 return silly_function(50, "Hello\n");
}
```

function(%rdi, %rsi, %rdx)



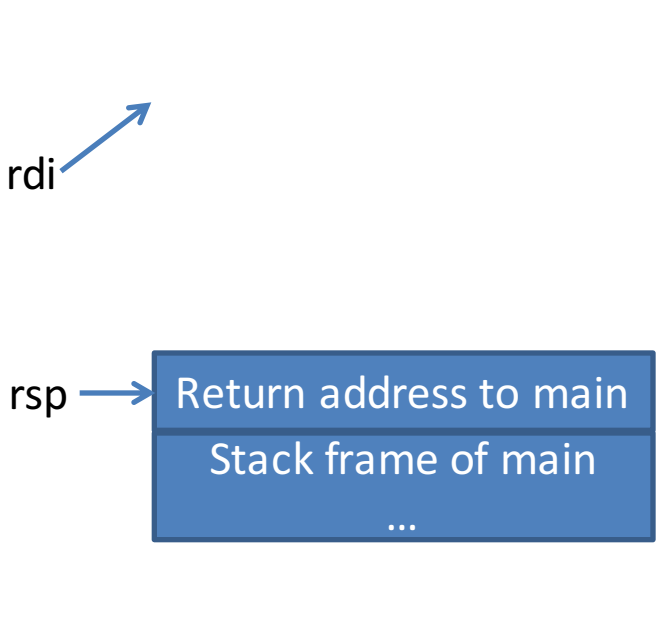
Vor dem Return fügt der Compiler Code ein, um den lokalen Stackframe ("buffer") zu entfernen (%rsp += 0x64)

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

Lokale Variablen und der Stack

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

int main() {
 return silly_function(50, "Hello\n");
}
```

function(%rdi, %rsi, %rdx)



Stack ist aufgeräumt...

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

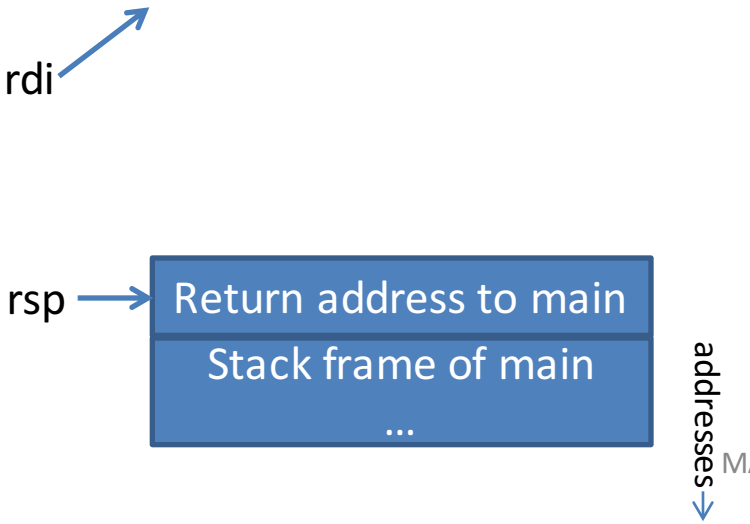
Lokale Variablen und der Stack

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

int main() {
 return silly_function(50, "Hello\n");
}
```

function(%rdi, %rsi, %rdx)

Rücksprung zu main entfernt
Returnadresse vom Stack

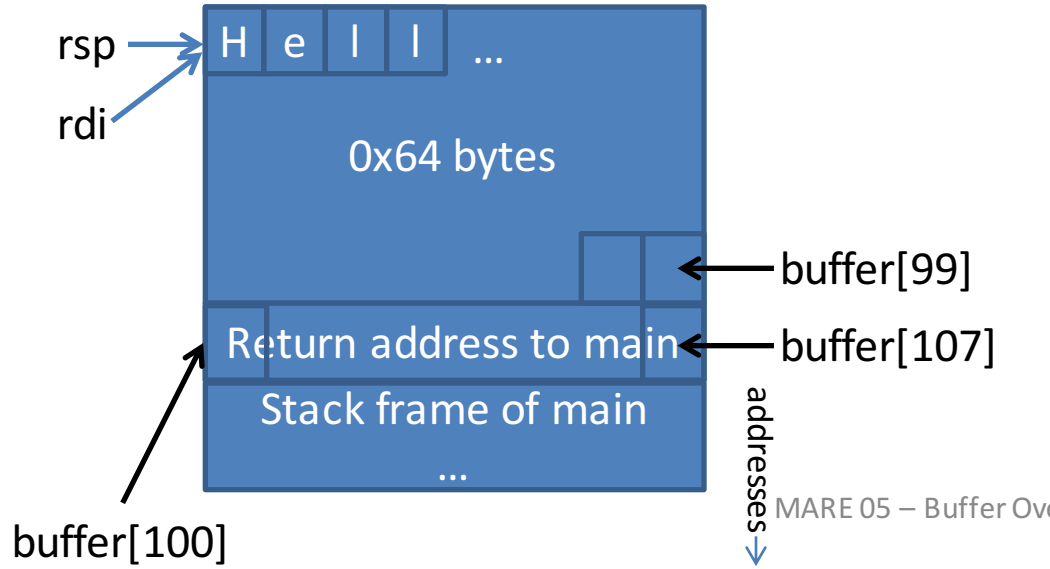
```
sub     $0x64, %rsp
movq    %rdi, %rdx
movq    %rsp, %rdi
callq   <memcpy>
xor     %rax,%rax
add     $0x64, %rsp
retq
```



Lokale Variablen und der Stack

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

int main() {
 return silly_function(50, "Hello\n");
}
```

function(%rdi, %rsi, %rdx)




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

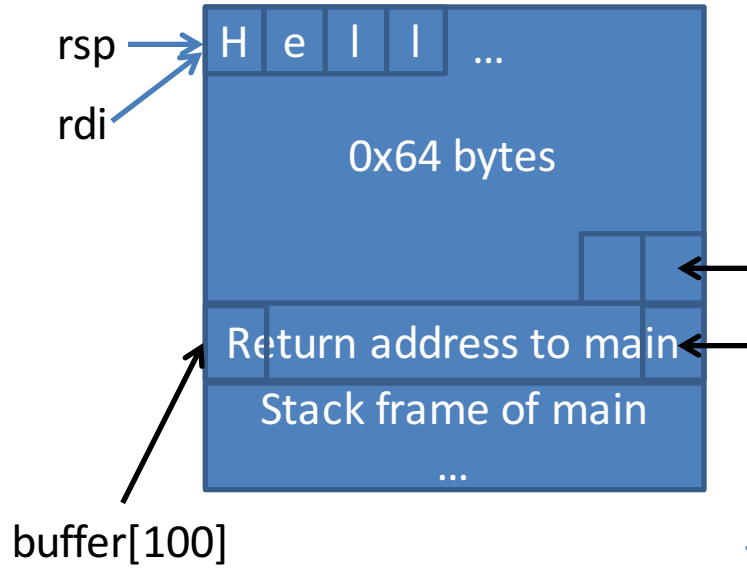
Lokale Variablen und der Stack

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

int main() {
 return silly_function(108, "Hello…ABCDEFGH");
}
```

function(%rdi, %rsi, %rdx)

Wir übergeben jetzt eine Länge von **108** Zeichen und einen entsprechend langen String

memcpy kennt die Größe von "buffer" nicht und wird angewiesen, 108 Byte zu kopieren



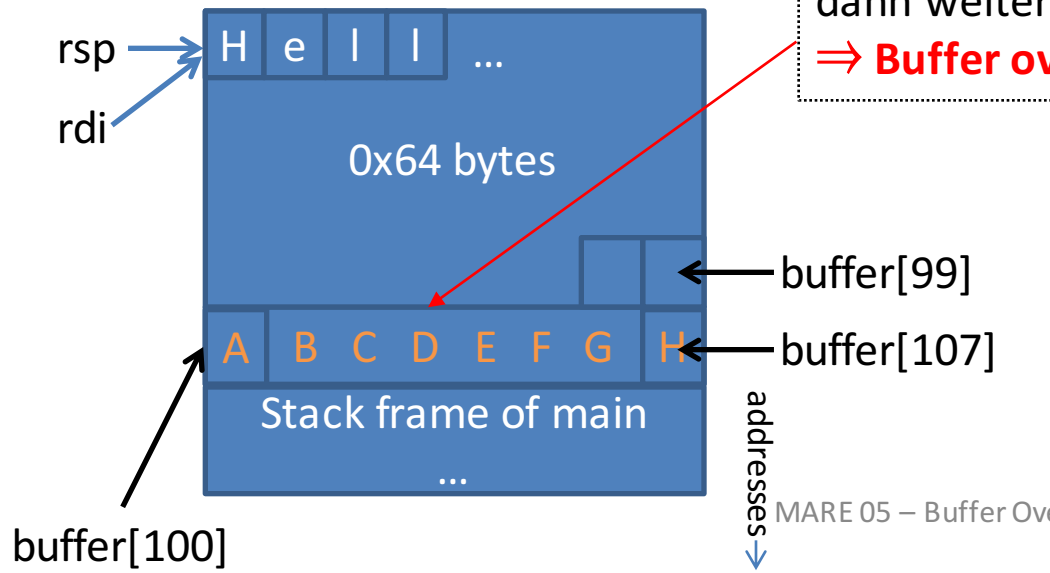

```
sub     $0x64, %rsp
movq    %rdi, %rdx
movq    %rsp, %rdi
callq   <memcpy>
xor     %rax,%rax
add     $0x64, %rsp
retq
```

Lokale Variablen und der Stack

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

int main() {
 return silly_function(108, "Hello…ABCDEFGH");
}
```

function(%rdi, %rsi, %rdx)

memcpy schreibtbuffer[0]…buffer[99] und dann weiter buffer[100]…buffer[107]!
⇒ **Buffer overflow!**



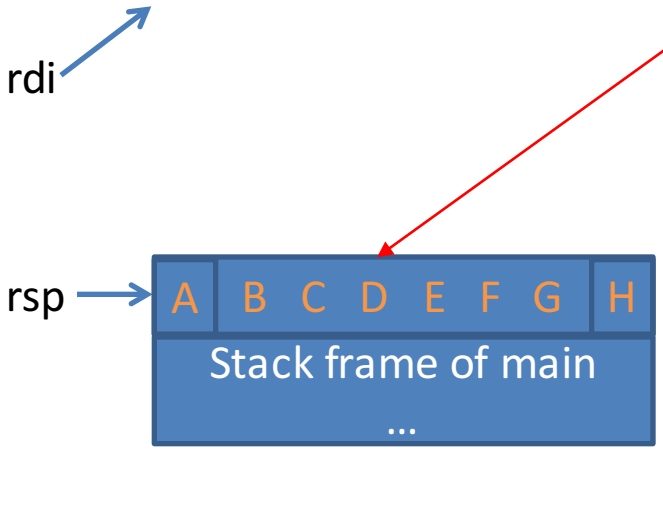

```
sub     $0x64, %rsp
movq    %rdi, %rdx
movq    %rsp, %rdi
callq   <memcpy>
xor     %rax,%rax
add     $0x64, %rsp
retq
```

Lokale Variablen und der Stack

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

int main() {
 return silly_function(108, "Hello…ABCDEFGH");
}
```

function(%rdi, %rsi, %rdx)

buffer wurde vom Stack entfernt (aber natürlich nur 100 Bytes), %rsp zeigt auf die Rücksprungadresse...




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

Lokale Variablen und der Stack

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

int main() {
 return silly_function(108, "Hello…ABCDEFGH");
}
```

function(%rdi, %rsi, %rdx)

An Stelle der Rücksprungadresse stehen die letzten 8 kopieren Bytes: „ABCDEFGH“ = **0x4142434445464748**

⇒ Segmentation fault bei retq!

rdi

rsp

addresses

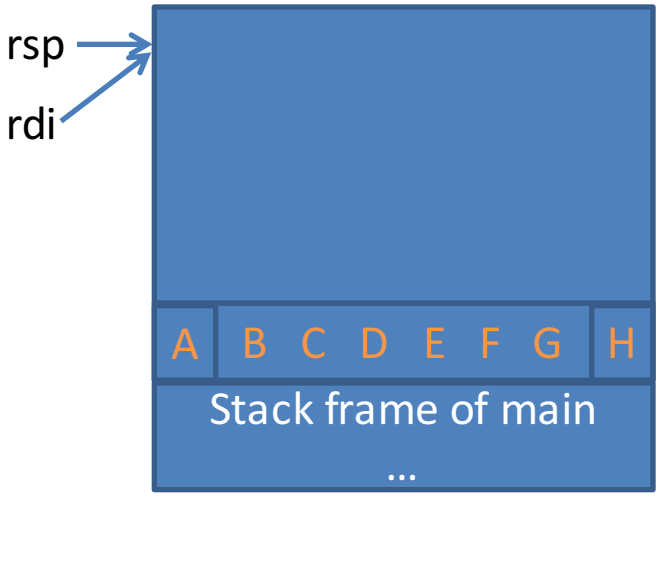
```
sub     $0x64, %rsp
movq    %rdi, %rdx
movq    %rsp, %rdi
callq   <memcpy>
xor     %rax,%rax
add     $0x64, %rsp
retq
```

Lokale Variablen und der Stack

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

int main() {
 return silly_function(108, "Hello…ABCDEFGH");
}
```

function(%rdi, %rsi, %rdx)

Wir übergeben hier eine Länge von **108** Zeichen und einen String, der **Text** enthält! ⇒ **Crash**!



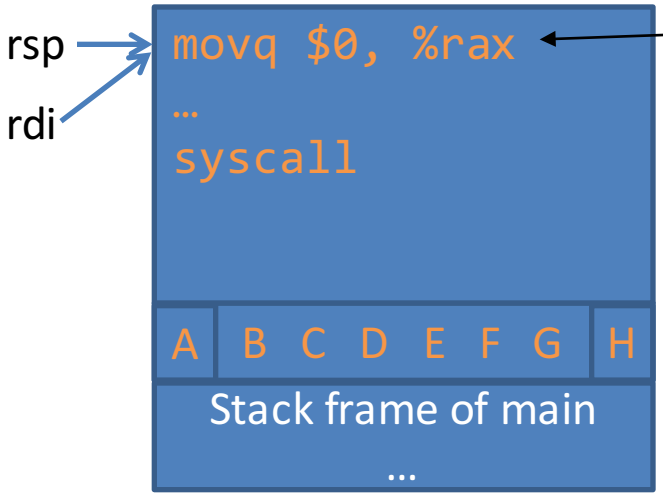

```
sub     $0x64, %rsp
movq    %rdi, %rdx
movq    %rsp, %rdi
callq   <memcpy>
xor     %rax,%rax
add     $0x64, %rsp
retq
```

Lokale Variablen und der 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…ABCDEFGH");
}
```

function(%rdi, %rsi, %rdx)

Wir übergeben **jetzt** eine Länge von **108** Zeichen und einen **String, der Maschinencode enthält!**



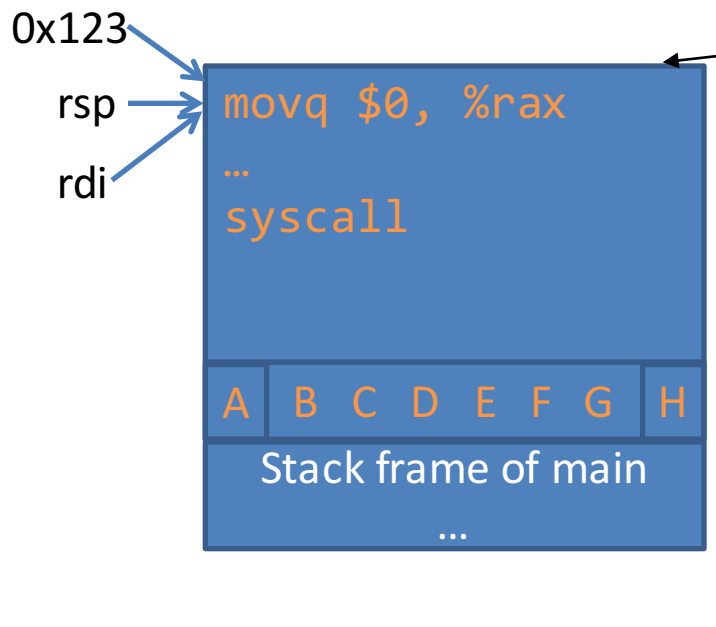

```
sub     $0x64, %rsp
movq    %rdi, %rdx
movq    %rsp, %rdi
callq   <memcpy>
xor     %rax,%rax
add     $0x64, %rsp
retq
```

Lokale Variablen und der 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…ABCDEFGH");
}
```

function(%rdi, %rsi, %rdx)

Der Maschinencode (=Startadresse von “buffer”) liegt hier ab Adresse 0x123



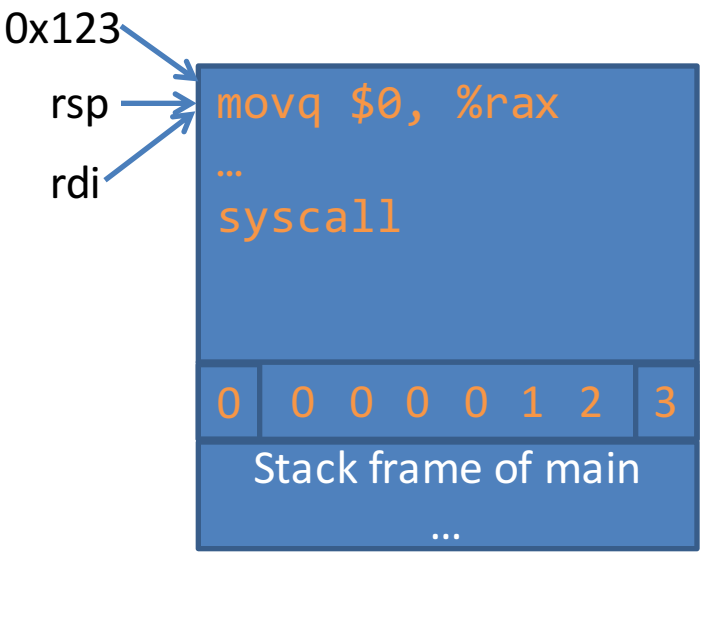

```
sub     $0x64, %rsp
movq    %rdi, %rdx
movq    %rsp, %rdi
callq   <memcpy>
xor     %rax,%rax
add     $0x64, %rsp
retq
```

Lokale Variablen und der 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…");
}
```

function(%rdi, %rsi, %rdx)

Ersetzen von Rücksprungadresse “ABCDEF” durch **Adresse des Maschinencodes!** (little endian!)



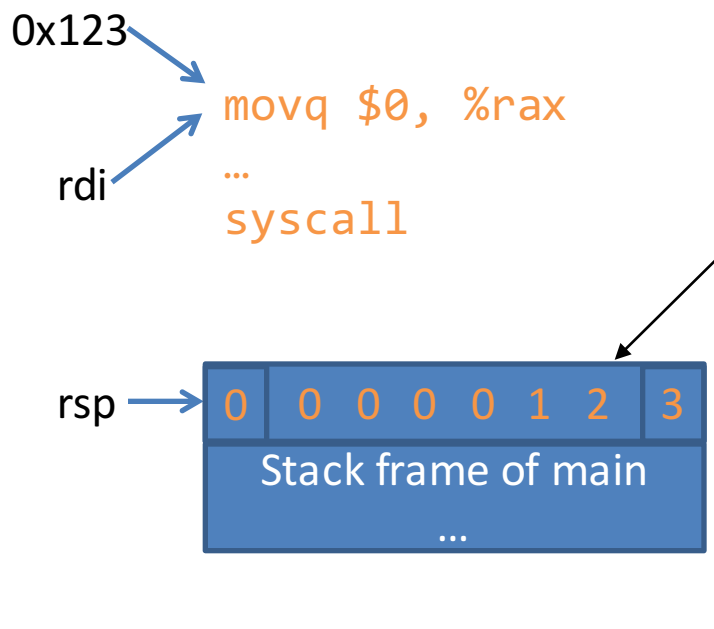

```
sub     $0x64, %rsp
movq    %rdi, %rdx
movq    %rsp, %rdi
callq   <memcpy>
xor     %rax,%rax
add     $0x64, %rsp
retq
```

Lokale Variablen und der 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…230100…");
}
```

function(%rdi, %rsi, %rdx)

Rücksprung aus der Funktion springt nach 0x123. “buffer” ist nicht mehr im Stackframe, aber **Werte von buffer noch im Speicher**!



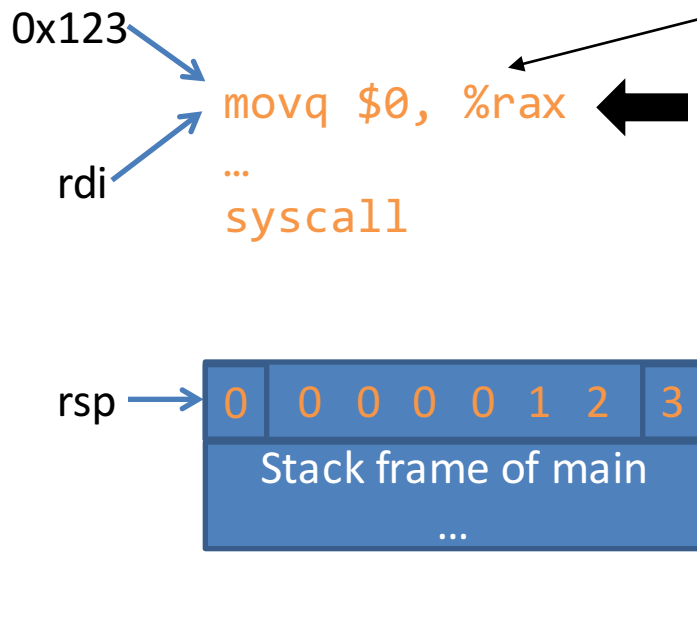

```
sub     $0x64, %rsp
movq    %rdi, %rdx
movq    %rsp, %rdi
callq   <memcpy>
xor     %rax,%rax
add     $0x64, %rsp
retq
```

Lokale Variablen und der 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…230100…");
}
```

function(%rdi, %rsi, %rdx)

Ausführung des im übergebenen String enthaltenen Maschinencode nach "Rück"-sprung an Adresse 0x123!




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

Angriffsmethoden

Ausführbarer Stack

- Platzieren von Code im Stackbereich, wie gerade gesehen
- Wie kann man die Adresse auf dem Stack finden?

Nicht ausführbarer Stack

- Platzieren von Code im Stackbereich nicht mehr möglich
- Aber Aufruf anderer Funktionen, z.B. aus der libc

Ausführbarer Stack

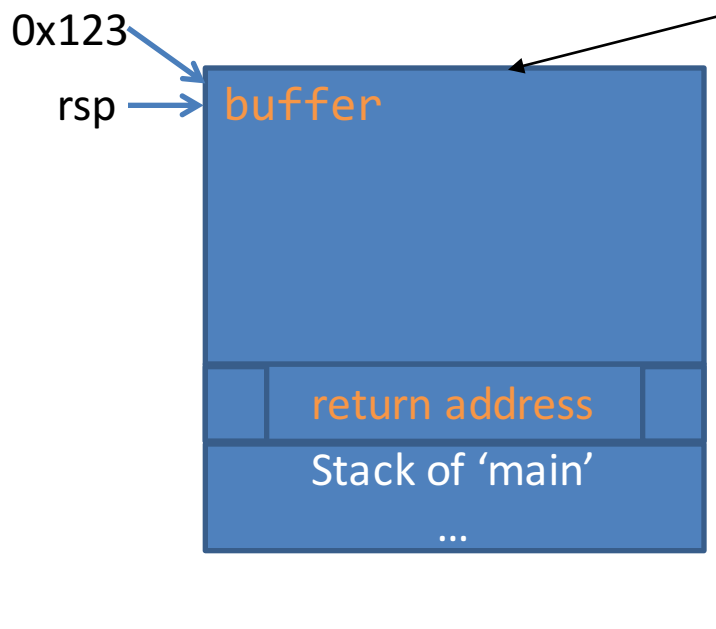
- Platzieren von Code im Stackbereich

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…");
}
```

function(%rdi, %rsi, %rdx)

Woher kennen wir die Adresse 0x123 von “buffer”?



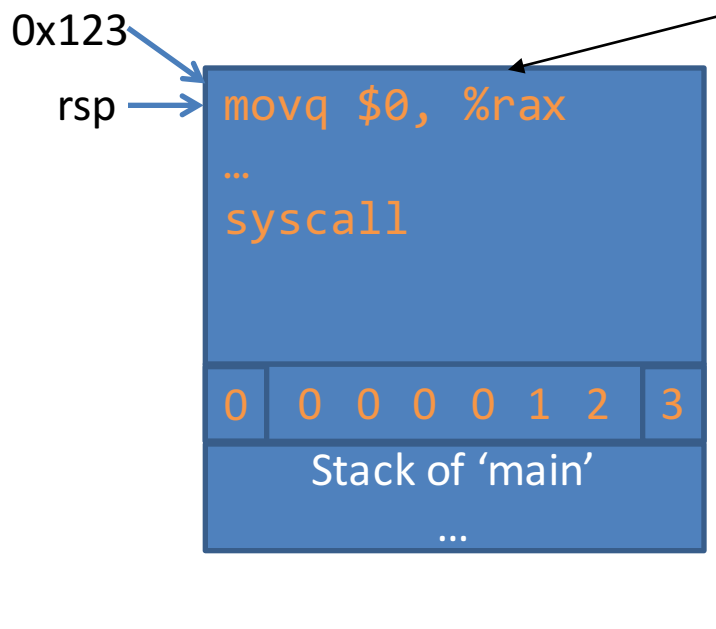

```
sub     $0x64, %rsp
movq    %rdi, %rdx
movq    %rsp, %rdi
callq   <memcpy>
xor     %rax,%rax
add     $0x64, %rsp
retq
```

Ausführbarer Stack

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

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

function(%rdi, %rsi, %rdx)

memcpy kopiert Maschinencode an den Anfang von “buffer”




```
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;
}

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

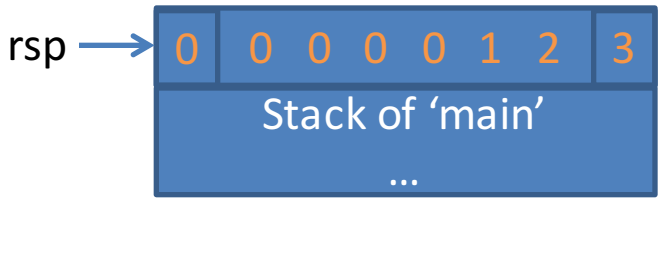
function(%rdi, %rsi, %rdx)

Wenn “buffer” bei 0x123 anfängt
⇒ Code wird ausgeführt

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);
 return 0;
}

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

function(%rdi, %rsi, %rdx)

Der Stack liegt aber nicht immer an der selben Adresse (oberhalb des top of Stack liegen z.B. Environment-Variable!)

0x123

```
movq $0, %rax
…
syscall
```

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

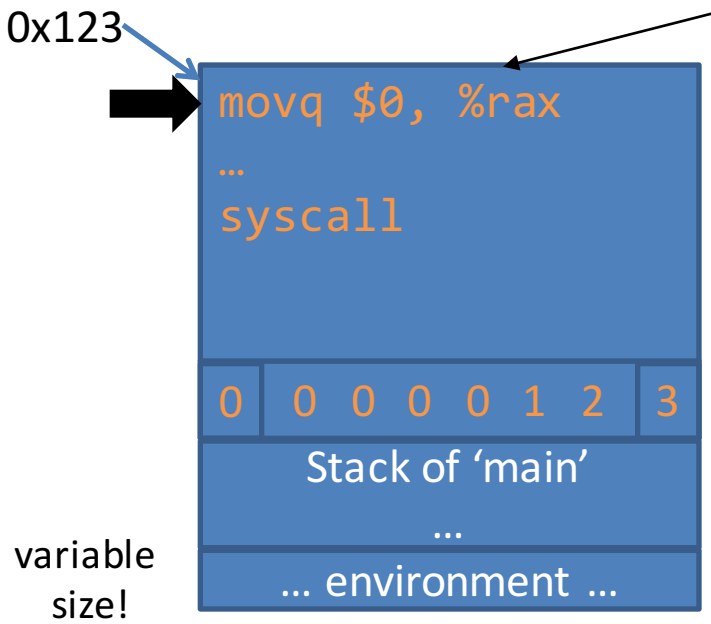
variable size!

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…");
}
```

function(%rdi, %rsi, %rdx)

Bisher: buffer an 0x123, funktioniert!

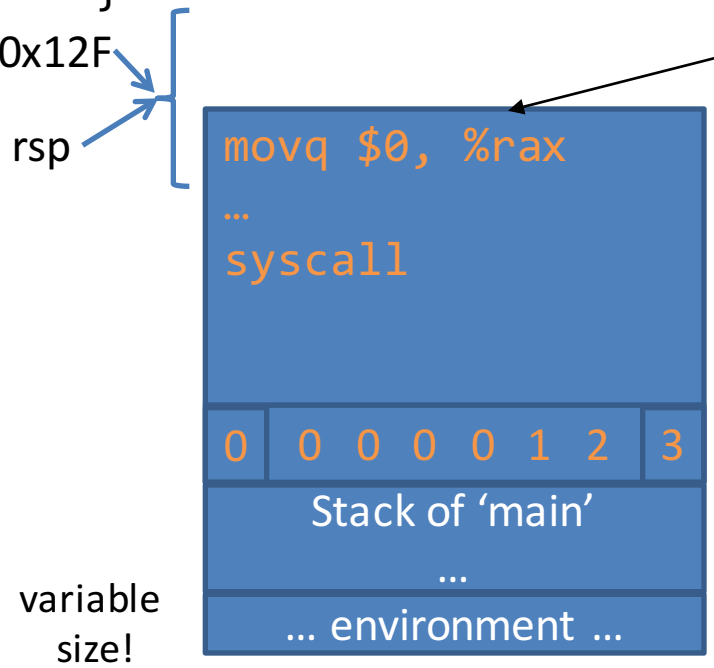



```
sub     $0x64, %rsp
movq    %rdi, %rdx
movq    %rsp, %rdi
callq   <memcpy>
xor     %rax,%rax
add     $0x64, %rsp
retq
```

Ausführbarer Stack

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

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

function(%rdi, %rsi, %rdx)



Der Stack liegt aber nicht immer an der selben Adresse (oberhalb des top of Stack liegen z.B. Environment-Variable!)

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

Ausführbarer Stack

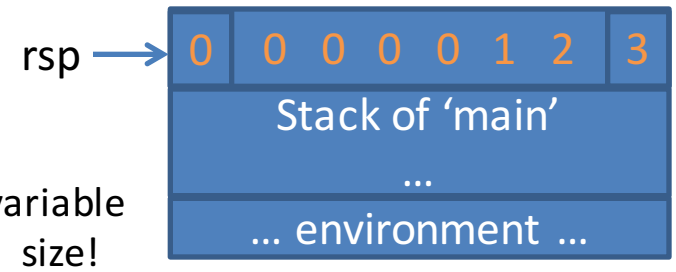
```
int silly_function(int len, char* src) {
 char buffer[100];
 memcpy(buffer, src, len);
 return 0;
}

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

function(%rdi, %rsi, %rdx)

Jetzt: buffer an 0x12F, was passiert?

0x12F

```
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);
 return 0;
}

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

function(%rdi, %rsi, %rdx)

0x12F

(garbage instructions)

```
movq $0, %rax
…
syscall
```

Jetzt: buffer an 0x12F, was passiert?

Sprung nach 0x123

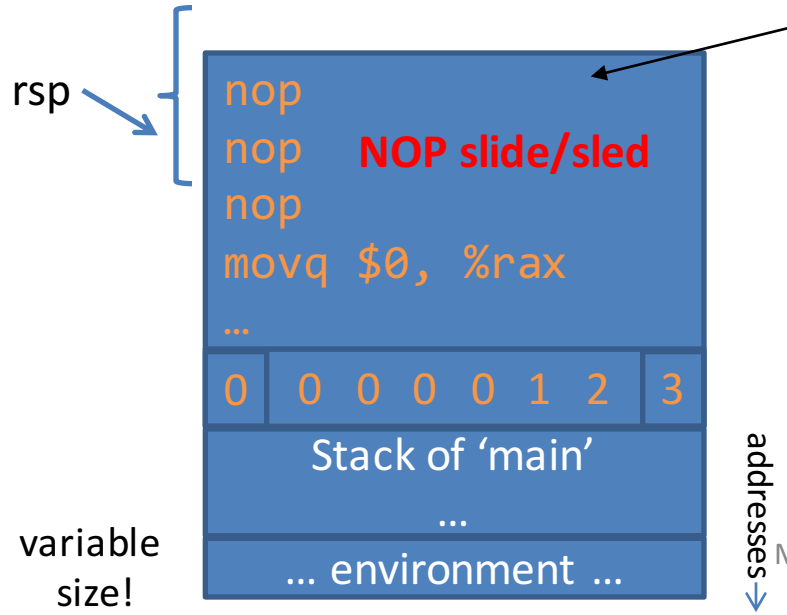
Dort stehen zufällige Daten!

```
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;
}

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

function(%rdi, %rsi, %rdx)



Zum movq kommen wir evtl. nicht mehr, da vorher Crash (z.B. ungültige Instruktion)! Vorher eine Reihe **"NOP"-Instruktionen einfügen**, die eventuellen **Adressversatz kompensieren**

```
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;
}

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

function(%rdi, %rsi, %rdx)

```
nop
nop
nop
movq $0, %rax
…
```

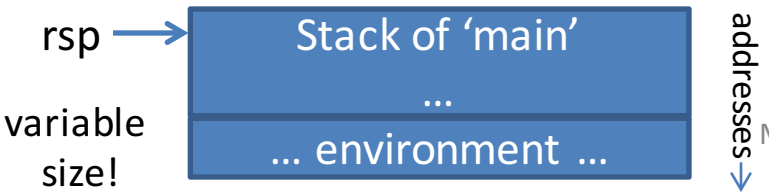
NOP slide/sled

"NOP"-Instruktionen, die eventuellen Adressversatz kompensieren.

Opcode von NOP = 0x90 (nur 1 Byte)

An jeder Adresse vor eigentlichem Code steht eine gültige (NOP-) Instruktion

```
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;
}

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

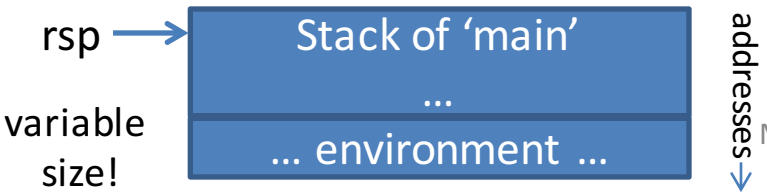
function(%rdi, %rsi, %rdx)

Ein Byte Versatz? Kein Problem, noch ein NOP!

```
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
```



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…");
}
```

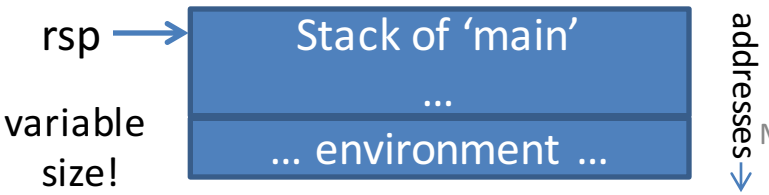
function(%rdi, %rsi, %rdx)

Nach endlich vielen NOPs **kommt unser eigentlicher Code**

```
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
```



Nicht ausführbarer Stack

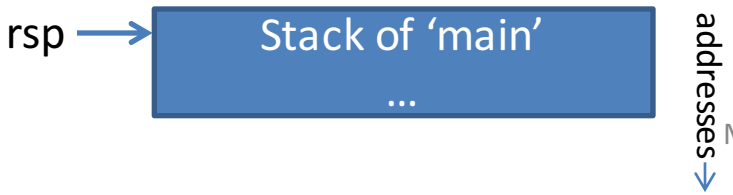
```
int silly_function(int len, char* src) {
 char buffer[100];
 memcpy(buffer, src, len);
 return 0;
}

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

function(%rdi, %rsi, %rdx)

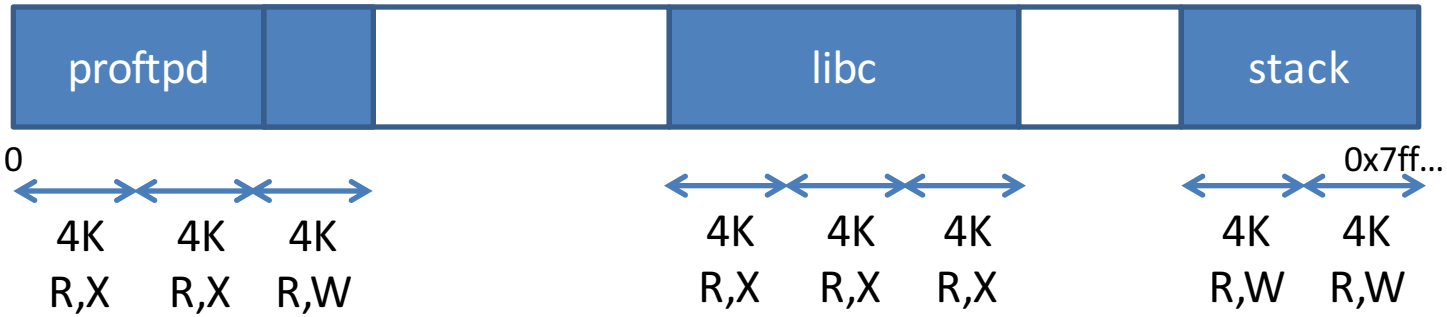
```
movq $0, %rax
…
syscall
```

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

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



Nicht ausführbarer Stack

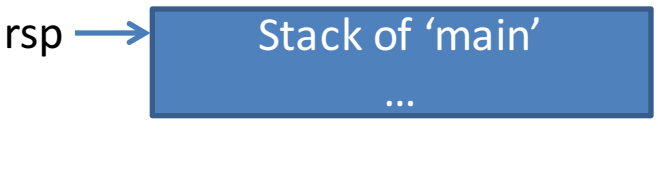


```
movq $0, %rax
…
syscall
```

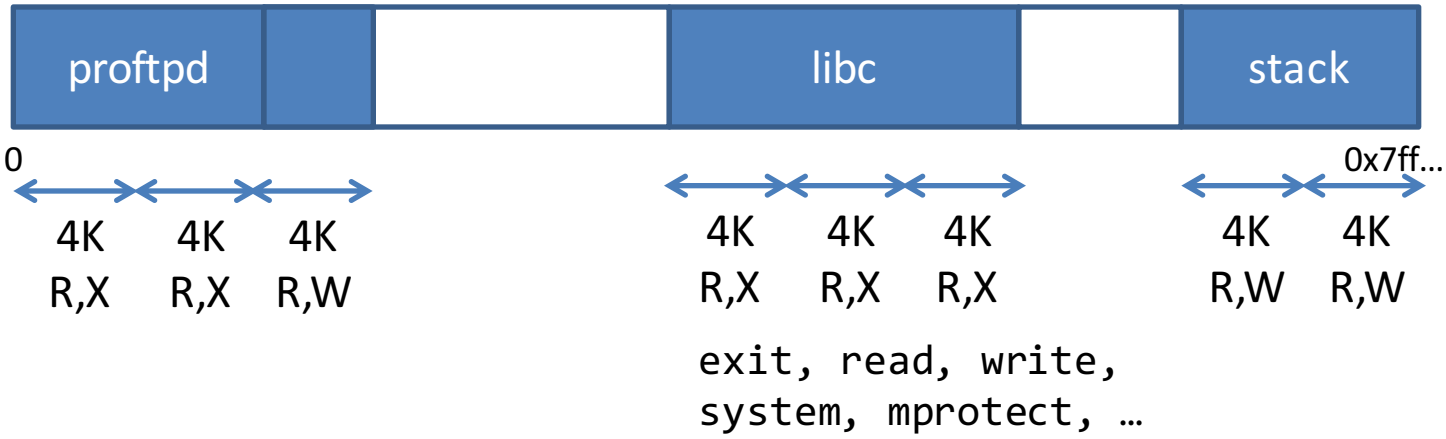
Code aus Buffer overflow-String ist nicht mehr ausführbar.

Aber: Existierende Funktionen können angesprungen werden

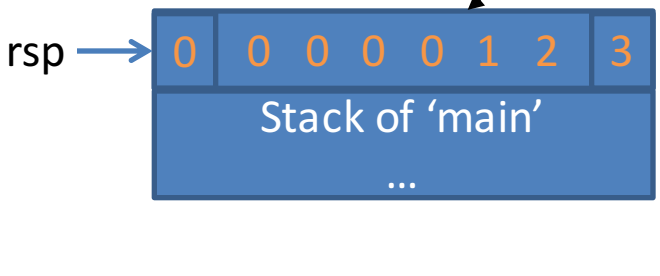
```
sub     $0x64, %rsp
movq    %rdi, %rdx
movq    %rsp, %rdi
callq   <memcpy>
xor     %rax,%rax
add     $0x64, %rsp
retq
```



Return zur libc

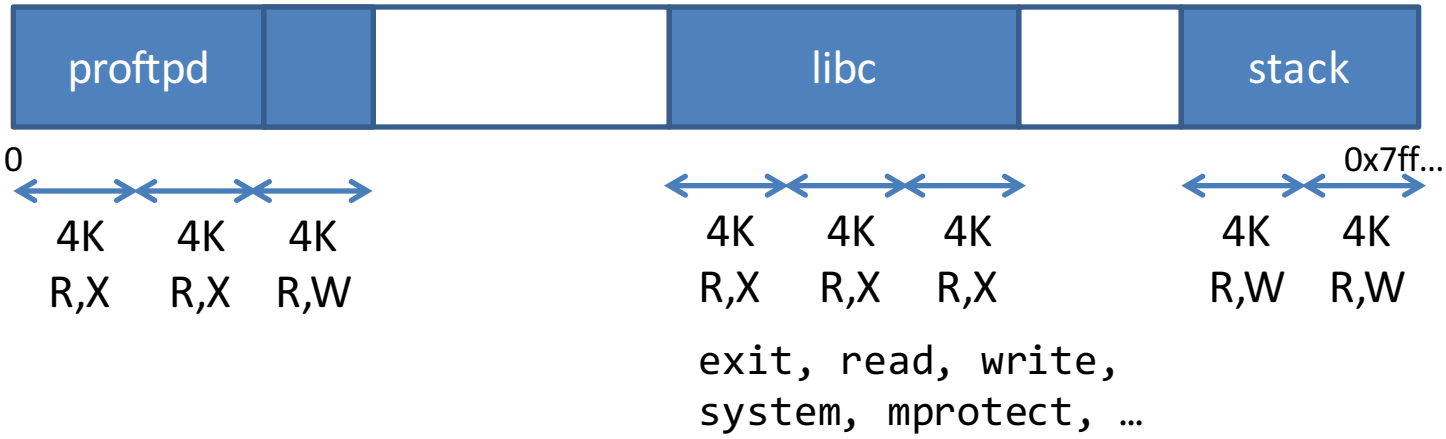


Ersetzen der Adresse 0x123 durch Adresse einer existierenden Funktion, z.B. in der libc




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

Return zur libc



Ersetzen der Adresse 0x123 durch Adresse einer existierenden Funktion, z.B. in der libc

Hier: „exit“ (wenig nützlich…)


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

Für Experimente...

Abschalten von Stack Canaries (Details nächste Woche):

- gcc-Option „**-fno-stack-protector**“

Abschalten von ASLR (auch nächste Woche):

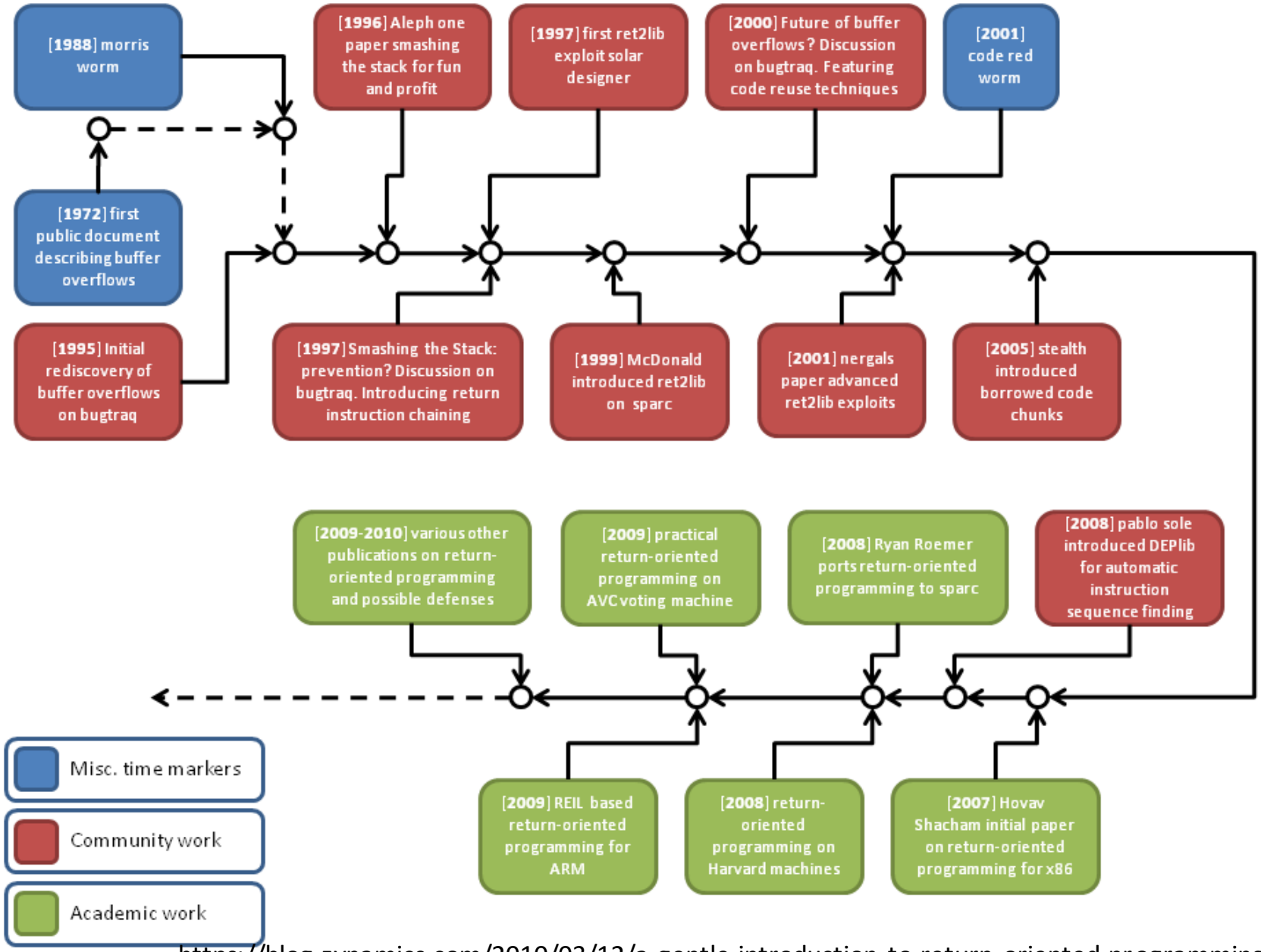
- Neue bash mit deaktivierter ASLR aufrufen:
  $ **setarch `uname -m` -R /bin/bash**

Abschalten von Data Execution Protection für den Stack:

- gcc-Option „**-z execstack**”

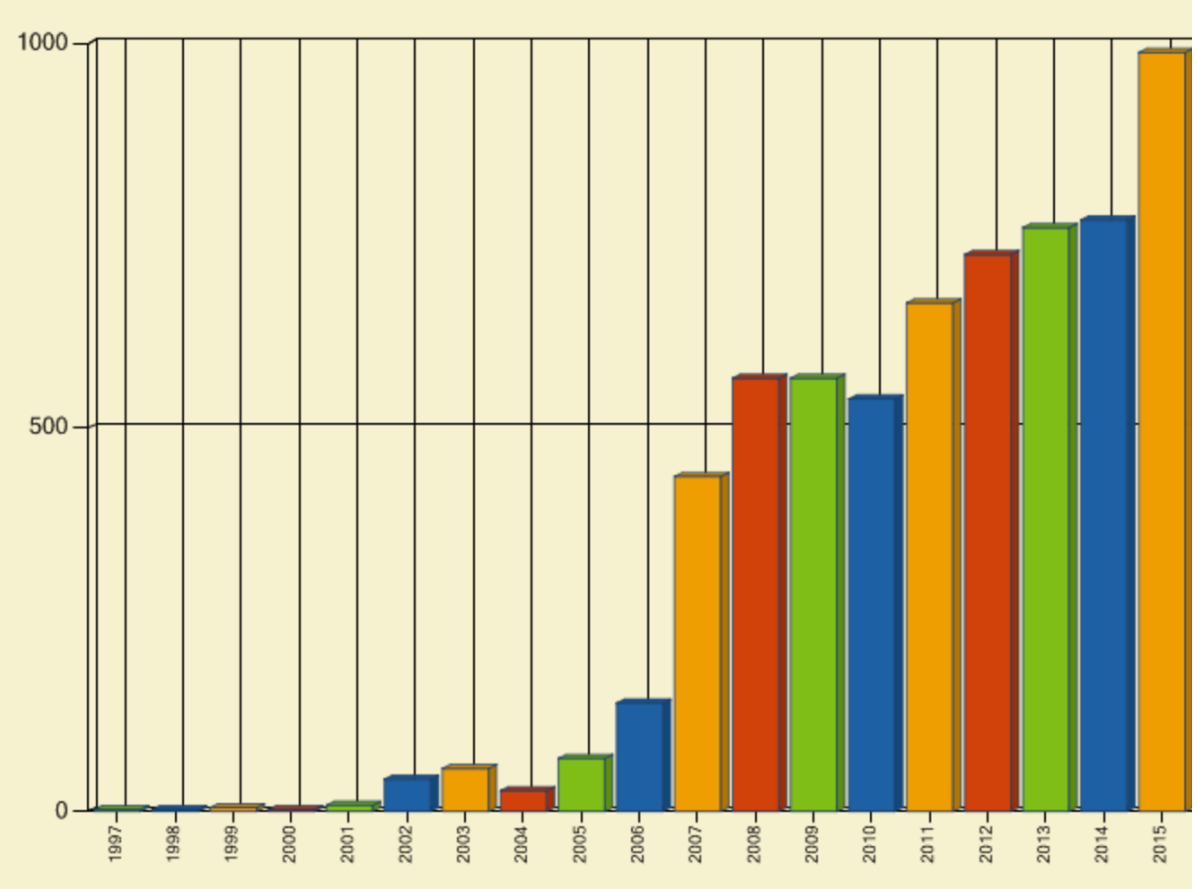
Historie von Bufferoverflows

HOCHSCHULE COBURG



https://blog.zynamics.com/2010/03/12/a-gentle-introduction-to-return-oriented-programming/

Anzahl ausgenutzter Bufferoverflows



[web.nvd.nist.gov]

Fazit

Bufferoverflows sind eine der häufigsten Ursachen für Sicherheitsprobleme

- Schon seit Langem ausgenutzt, immer noch großes Problem
  - Angriff über injizierten Code
- Gegenmaßnahme: Data Execution Protection (DEP)
  - Verwendung von „return to libc“

Literatur

Bufferoverflows

- Aleph One, *„Smashing The Stack For Fun And Profit“,* PHRACK49, http://insecure.org/stf/smashstack.html
- Heffner, *„Smashing The Modern Stack For Fun And Profit“, https://www.ethicalhacker.net/columns/heffner/smashing-the-modern-stack-for-fun-and-profit*
- Cowan et al., *„Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade“*, SANS 2000
- Pinkus and Baker, *„Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overruns“*, IEEE Security & Privacy (Volume: 2, Issue: 4, July-Aug. 2004)

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*

Shellcode

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