Übungen zur Vorlesung Systemsicherheit

Address Space Layout Randomization

Tilo Müller, Reinhard Tartler, Michael Gernoth

Lehrstuhl Informatik 1 + 4

19. Januar 2011

Intro

ASLR

ASLR | What is ASLR?

- ASLR = Address Space Layout Randomization, also die zufällige Anordnung des virtuellen Addressraums
- ASLR = Buffer Overflow Prävention auf Betriebssystem-Ebene. Alternativen:
  - Hardware-Ebene: NX-bit (non-executable pages)
    - Intel: XD = execute disable
    - ARM: XN = execute never
    - AMD: EVP = enhanced virus protection

    Bsp: MS DEP (data execution prevention) basiert auf dem NX-bit
  - Compiler-Ebene: Canaries (StackGuard, Stack Smashing Protector etc.)
- Default seit Linux 2.6.12 (2005; Windows seit Vista Beta 2)
- Welche Segmente werden randomisiert?
  - text?
  - data, bss?
  - heap, stack?

  Theoretisch können alle randomisiert werden. Für Linux-2.6.32 / gcc-4.3.2 (CIP) wird nur der *Stack* standardmäßig randomisiert.

ASLR | Example

aslr-demo.c

```
#include <stdio.h>
#include <malloc.h>
long data = 16;
long bss;
int main() {
        long stack;
        long *heap = malloc(8);
        printf("Text : %08x\n",&main);
        printf("Data : %08x\n",&data);
        printf("BSS  : %08x\n",&bss);
        printf("Heap : %08x\n",heap);
        printf("Stack: %08x\n",&stack);
}
```

Compile & run

```
> gcc -o aslr-demo aslr-demo.c ; ./aslr-demo
Text : 080483d4
Data : 08049674
BSS  : 08049680
Heap : 0804a008
Stack: bfaf706c
```

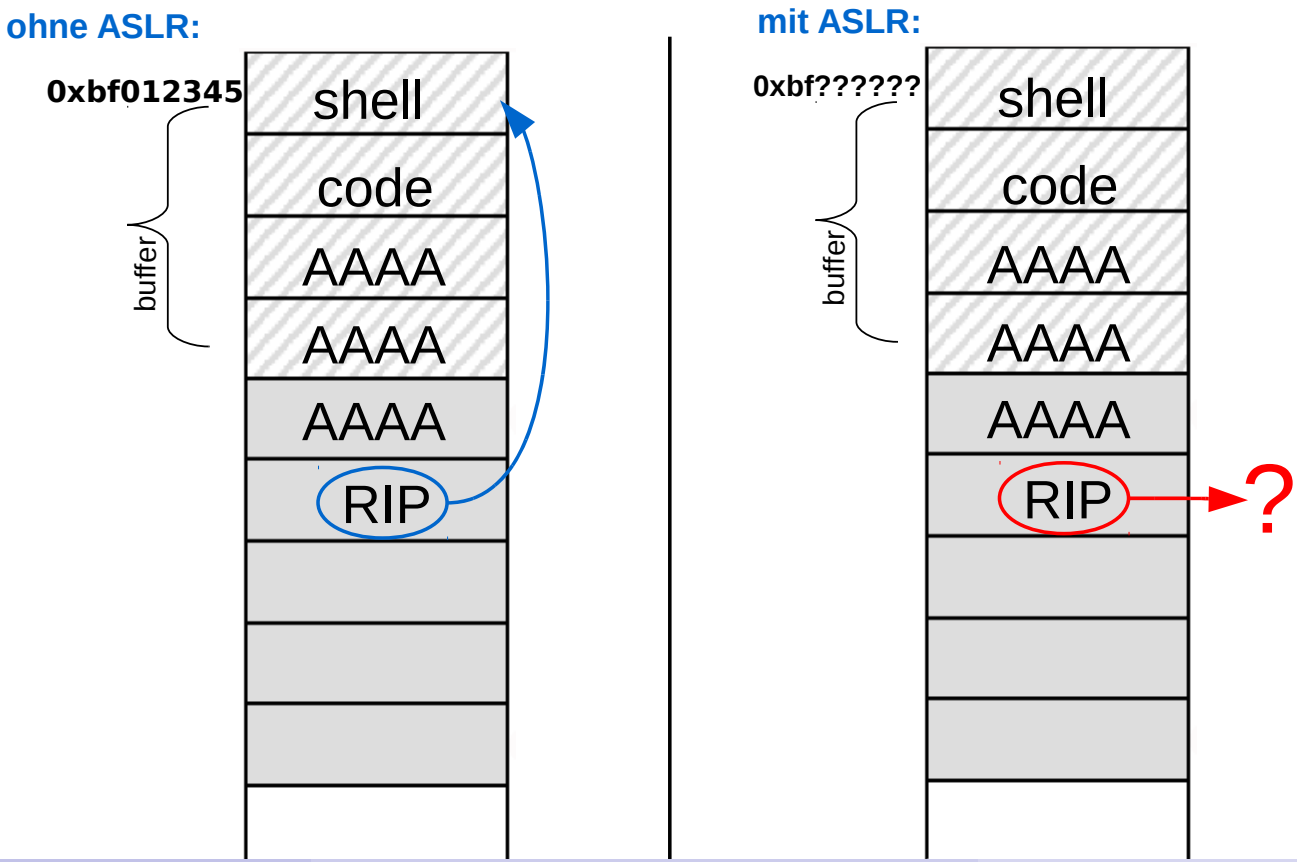
```
> ./aslr-demo
Text : 080483d4
Data : 08049674
BSS  : 08049680
Heap : 0804a008
Stack: bfc4a82c
```

```
./aslr-demo
Text : 080483d4
Data : 08049674
BSS  : 08049680
Heap : 0804a008
Stack: bf92c3ec
```

Beobachtung: 24 der 32-bit Stack-Adressen werden randomisiert (0xbf.....)

ASLR | Consequences

- Shellcode wird i.d.R. auf dem Stack abgelegt.
- Durch ASLR sind die Adressen des Stacks nicht mehr voraussagbar.
- Dadurch ist es unmöglich direkt in den Shellcode zu springen.



ASLR | Counter measures

Gegenmaßnahmen:

- Bruteforce / gut Glück
  (in Verbindung mit NOP slides, s.u.)
- Stack Adressen vorher in Erfahrung bringen
  (bspw. über `/proc/<pid>/stat`; für jeden lesbar)
- Shellcode in nicht-randomisierten Segmenten ablegen
  (d.h. in data, bss oder heap; Achtung: *Stack* overflow weiterhin nötig)
- Return-to-libc bzw. Return-oriented Programming
- *"Stack Juggling Methods"*:
  - ret2ret (A 4.3)
  - ret2esp (A 4.4)
  - ret2pop, ret2eax, ...

ASLR | Bruteforce

Auf 32-bit Systemen ist pures Bruteforce nicht ganz hoffnungslos:

- Chance richtige Adresse zu treffen: $1 : 2^{24}$
- Angriffe die im Mittel nötig sind: $2^{23} = 8388608$
- Angenommen 8 Angriffe pro Sekunde: nur 291 Stunden

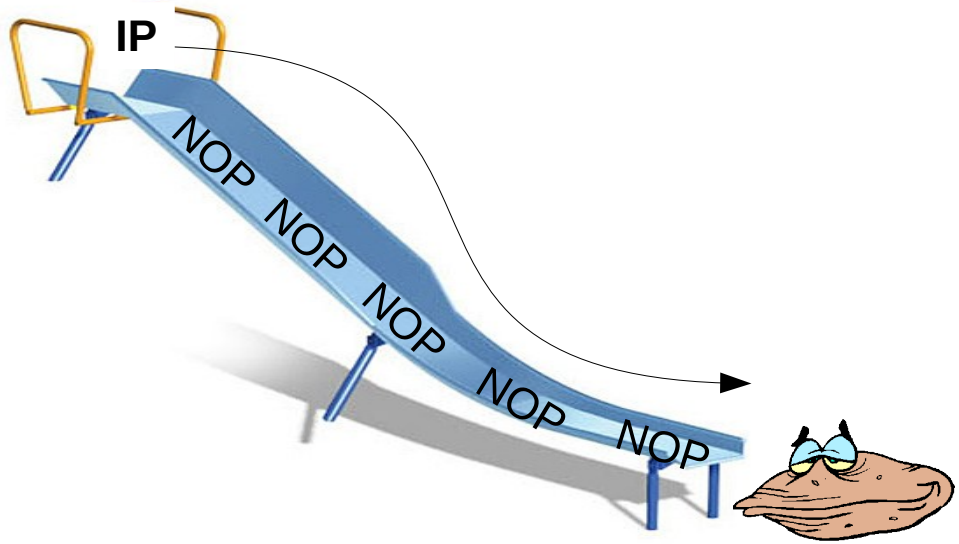
Chance erheblich verbessern: durch *NOP slides*

- NOP = No Operation
- x86 assembler opcode: `0x90`
  - Alternativ selber basteln:
    - `xchg eax, eax`
    - `mov edi,edi`
    - `sub esi,0`
    - `not ebx; not ebx`
    - ...

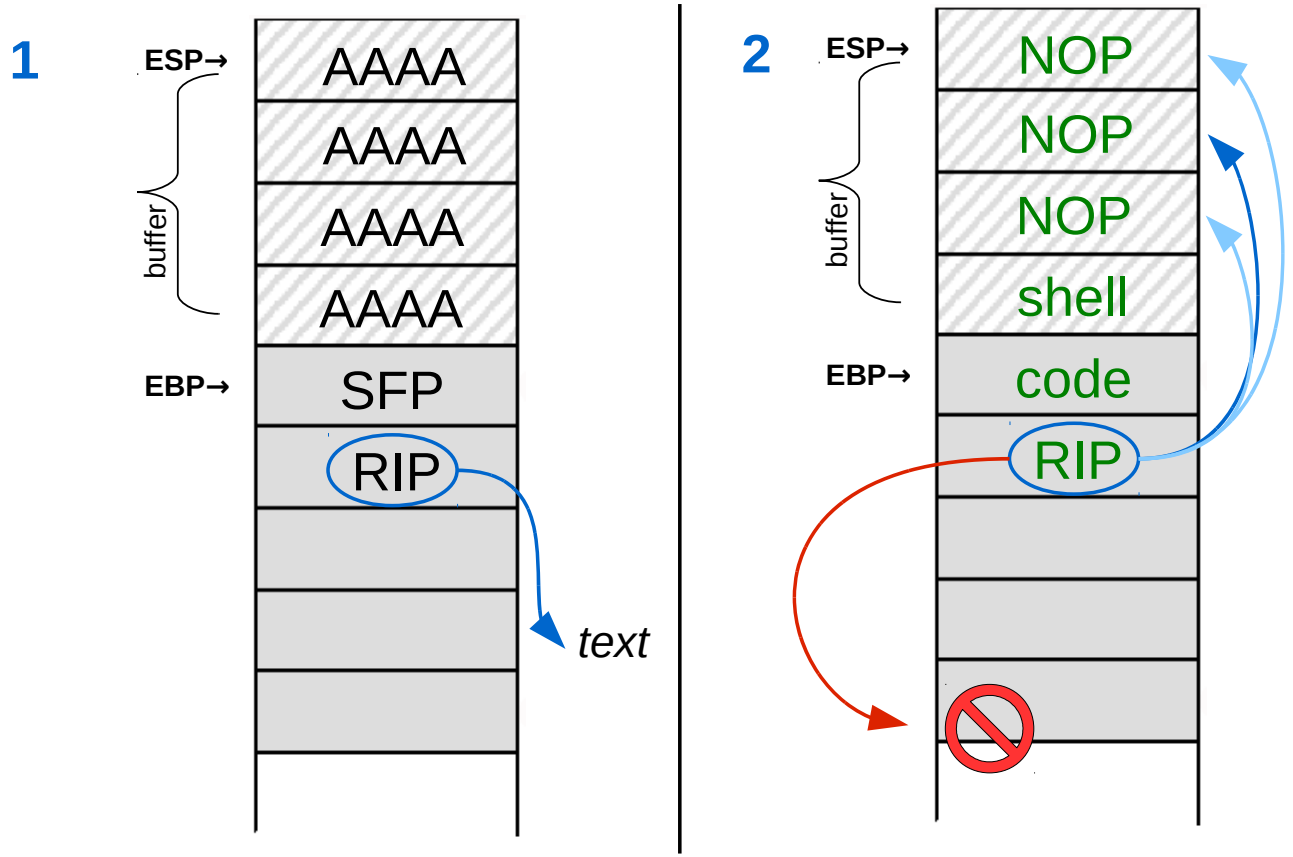
ASLR | NOP slide

Grundidee:

- Vor dem Shellcode einen möglichst großen NOP Bereich anlegen.
- Angreifer springt auf gut Glück *irgendwo* in den NOP Bereich.
- Der IP rutscht den NOPs entlang bis in den Shellcode.



ASLR | NOP slide



Aufgabe 4.3

ret2ret

ret2ret

- Klar: Vom Programm erzeugte Stackpointer kennen die korrekten Stack-Adressen.
- Idee: Missbrauche das Wissen dieser Stackpointer um in den Shellcode zu springen.

stackpointer.c

```
void main() {
        int no = 1;
        int* ptr = &no;
}
```

Compile and analyse

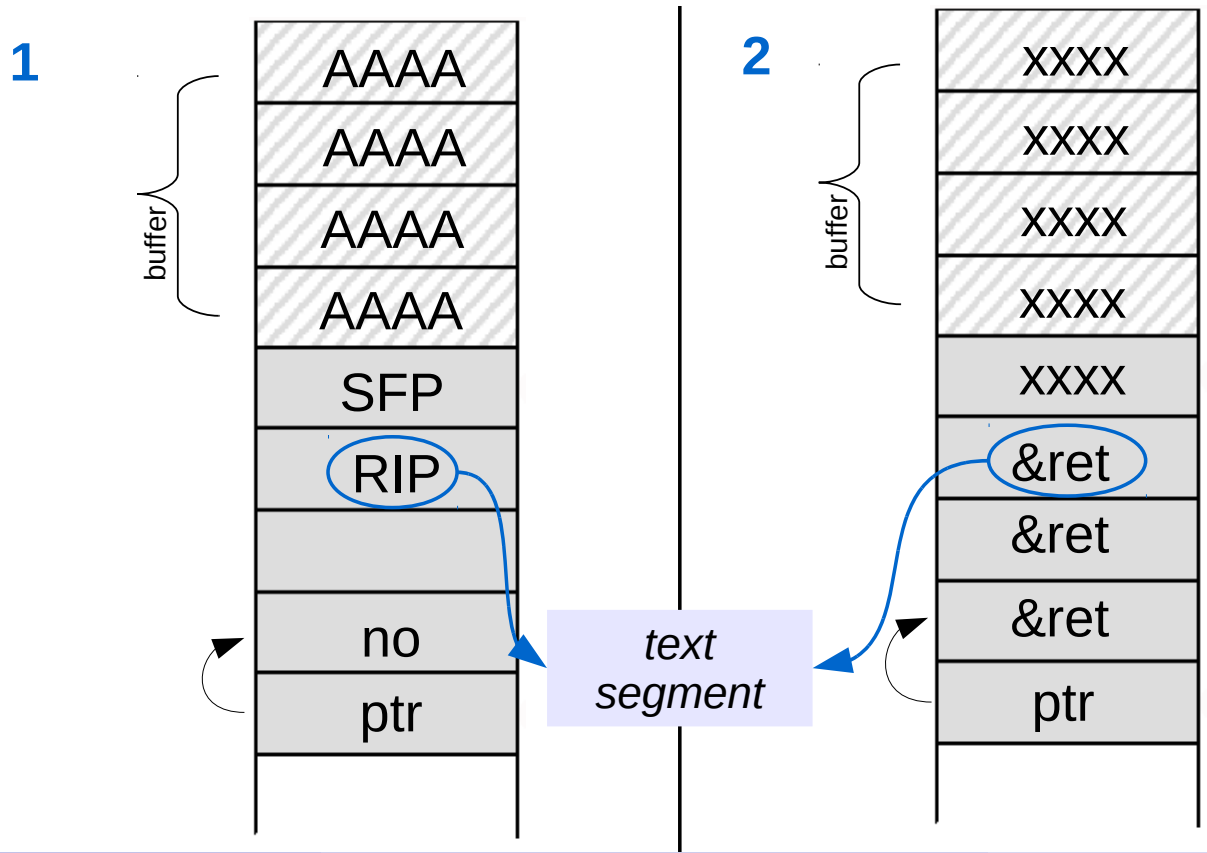
```
> gcc -g -o stackpointer stackpointer.c
(gdb) break 4
(gdb) run
(gdb) x/4x $esp
0xbfdb9e34: 0x0804954c 0xbfdb9e58 0x00000001 0xbfdb9e3c
```

→ An Speicherstelle 0xbfdb9e40 (*ptr) steht der Wert 0xbfdb9e3c (&no).

ret2ret

Solch ein Stackpointer steht aber nicht an der Stelle des RIP.
Wie also kann man ihn als Rücksprungadresse verwenden??

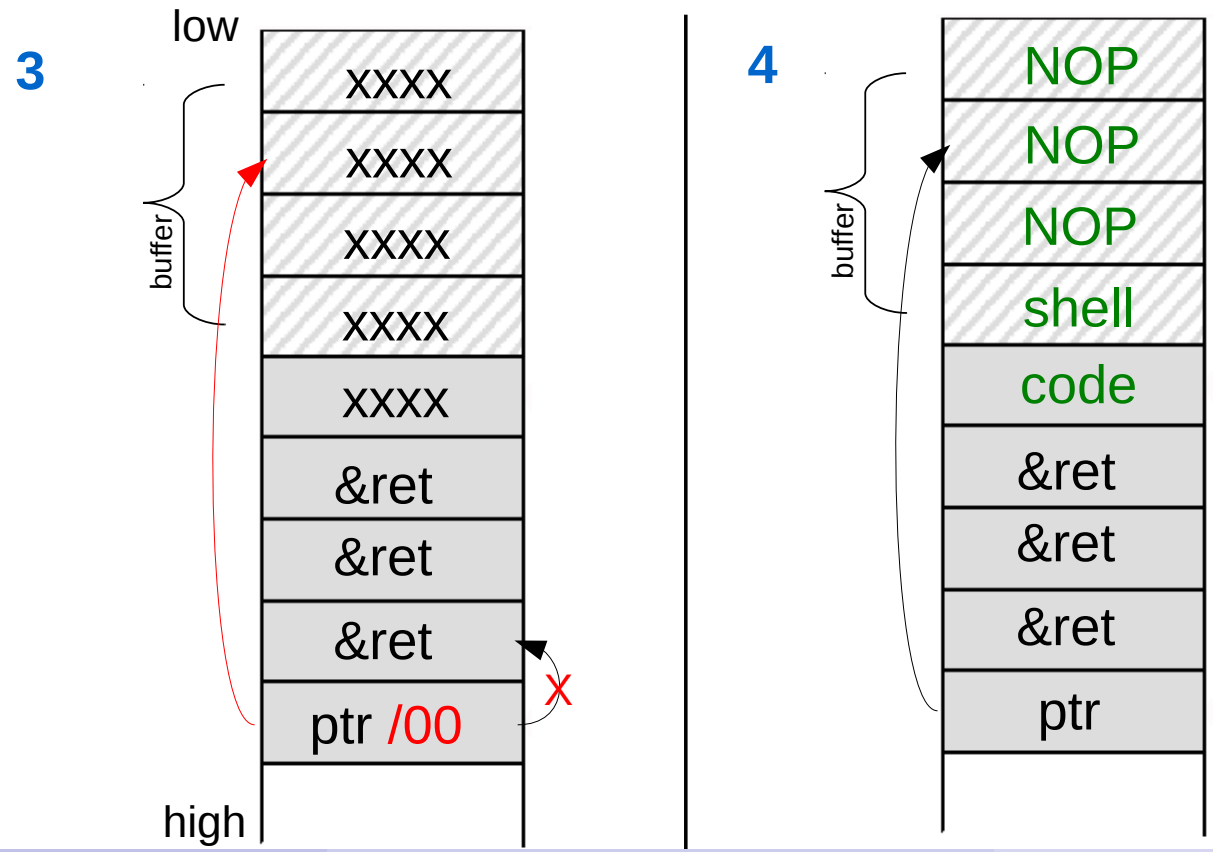
Idee: return chain



ret2ret

Stackpointer zeigt nur auf Stelle die zuvor mit `&ret` überschrieben wurde. Wie soll man eigenen Shellcode zur Ausführung bringen??

Idee: Den Stackpointer leicht umbiegen, d.h. least significant byte = 0.



ret2ret | Exercise

vulnerable.c

```
#include <string.h>
void function(char* str) {
        char buf[256];
        strcpy(buf, str);
}
int main(int argc, char** argv) {
        int no = 1;
        int *ptr = &no;
        function(argv[1]);
}
```

Hinweise:

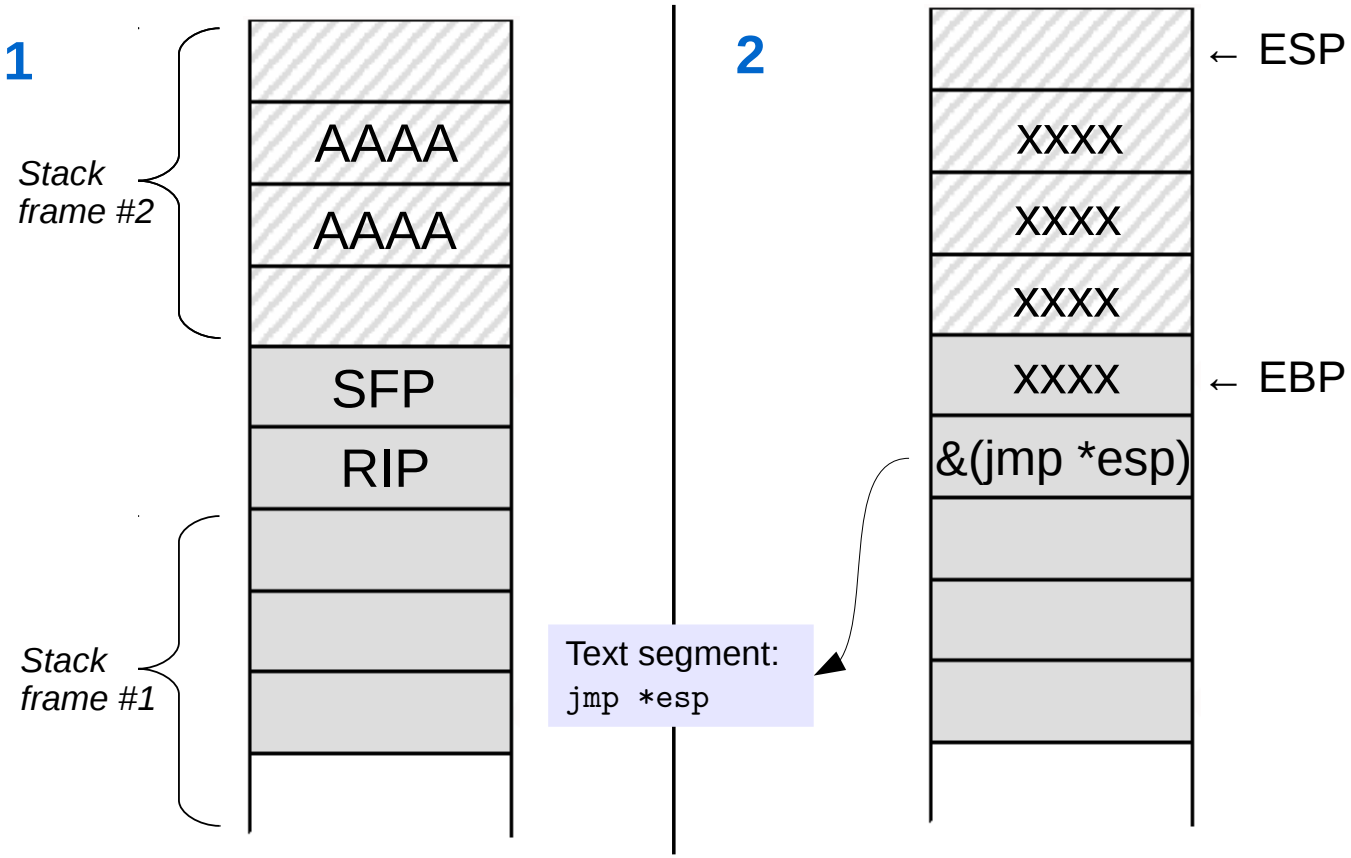
- Platzieren Sie NOP Slide und Shellcode in `buf`.
- Schreiben Sie nach dem Shellcode über die Grenzen von `buf` hinaus.
- Füllen Sie den Platz zwischen RIP und `ptr` durch eine ret2ret Kette.
- Biegen Sie `ptr` so um, dass er ungefähr auf `buf` zeigt.
  Achten Sie dabei darauf, dass `strcpy` *als letzte Operation sowieso schon ein Nullbyte kopiert (Ende eines Strings).*
- Es kann 2-3 Anläufe benötigen bis Ihr Exploit funktioniert.

Aufgabe 4.4

ret2esp

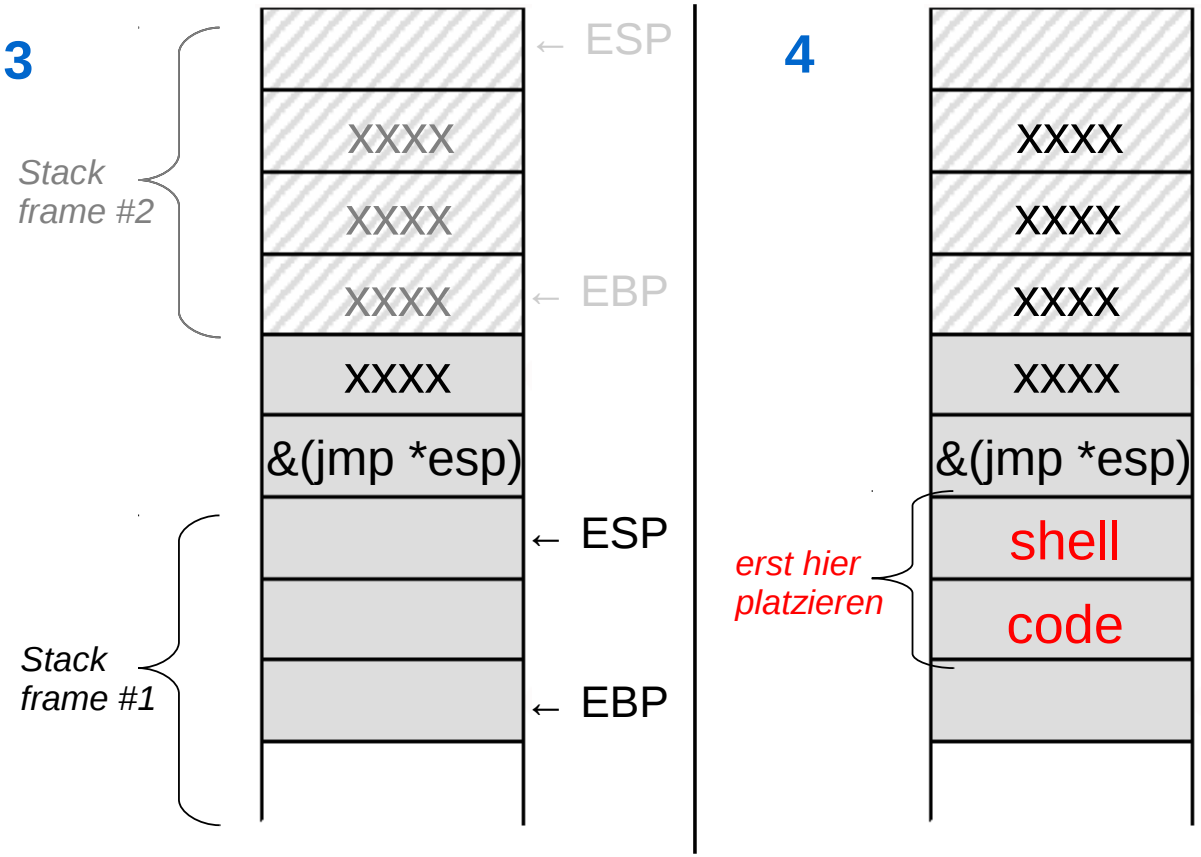
ret2esp

Ziel des ESP ist voraussagbar und zeigt auf den Stack.
→ Überschreibe RIP mit der Adresse einer `jmp *esp` Instruktion.



ret2esp

Achtung: `jmp *esp` wird erst nach dem Frame-Abbau ausgeführt.
$\rightarrow$ der Shellcode muss am darunterliegenden Frame beginnen.



ret2esp

Problem: `jmp *esp` ist eine sehr seltene Instruktion; sie wird von keinem C Compiler generiert und ist daher im Textsegment kaum anzutreffen.

Idee: Interpretiere beliebige Daten als Code, d.h. durchsuche das ganze Binary nach der Bytefolge 0xe4ff (Opcode für `jmp *esp`). So größer das Programm, desto größer die Chance, dass diese Bytefolge irgendwie produziert wurde.

Eine 2-Byte-Folge zufällig anzutreffen ist nicht soo unwahrscheinlich:

Beispiel

```
> hexdump /usr/bin/* | grep e4ff | wc -l
  1183
```

ret2esp | Exercise

vulnerable.c

```
#innclude <string.h>
void function(char* str) {
        char buf[256];
        strcpy(buf,str);
}
int main(int argc, char** argv) {
        int j=58623;
        function(argv[1]);
}
```

Hinweise:

- Zufälliger Weise gilt $e4ff_{hex} = 58623_{dec}$.
- Bestimmen Sie die Adresse von `jmp *esp`.
- Füllen Sie den Puffer mit beliebigen Daten, lassen Sie ihn überlaufen und überschreiben Sie den RIP mit der Adresse von `jmp *esp`.
- Schreiben Sie den Shellcode hinter die Grenzen von `buf`, in den Stackframe von `main`.

GNU Debugger (GDB)

GDB | What is GDB?

- GNU Projekt Debugger, von Richard Stallman initiiert.
- Hauptsächlich für C (aber auch C++, D, Fortran, ...)
- Viele Architekturen (x86 / x86-64, auch ARM, Power PC, Sparc, ...)
- Typische Eigenschaften eines Debuggers:
  - Analyse eines Programms zur Laufzeit
  - Breakpoints, Watchpoints
  - Single Stepping, Pause, Continue
  - Register auslesen und manipulieren
  - Speicherinhalte ausgeben und manipulieren
  - Textsegmente disassemblen
  - ...

GDB | Start

debugme.c

```
#include <string.h>
int main(int argc, char** argv) {
        char buf[16];
        int i = 1;
        memset(buf,'A',16);
        strcpy(buf, argv[1]);
        return i;
}
```

GCC Option -g produziert Debugging Informationen für GDB:

Compile and open

```
> gcc -g -o debugme debugme.c
> gdb debugme
GNU gdb 6.8-debian
Copyright (C) 2008 Free Software Foundation, Inc
(gdb) run abc
Program exited with code 01.
(gdb) run abcdefghijklmnopqrstuvqxyz
Program received signal SIGSEGV, Segmentation fault.
```

GDB | Breakpoints

C breakpoints

```
(gdb) list
1   #include <string.h>
2   int main(int argc, char** argv) {
3       char buf[16];
4       int i = 1;
5       memset(buf,'A',16);
6       strcpy(buf, argv[1]);
7       return i;
8   }
(gdb) break 6
(gdb) run abcdefghijklmnopqrstuvqxyz
Breakpoint 1, main (argc=2, argv=0xbfcbf864) at debugme.c:6
```

Assembler breakpoints

```
(gdb) disass main
0x080483d4 <main+0>: lea    0x4(%esp),%ecx
0x080483d8 <main+4>: and    $0xfffffff0,%esp
0x080483db <main+7>: pushl  -0x4(%ecx)
0x080483de <main+10>: push   %ebp
...
(gdb) break *0x080483de
(gdb) run abc
Breakpoint 2, 0x080483de in main (argc=1285532, argv=0xb78bdb38) at debugme.c:2
```

GDB | Single Stepping

Normal fortfahren

```
(gdb) continue
```

Eine C Instruktion

```
(gdb) step
```

Eine C Instruktion ohne in Subroutines zu gehen

```
(gdb) next
```

Eine Assembler Instruktion

```
(gdb) stepi
```

GDB | Register

Alle Register auslesen

```
(gdb) info registers
eax            0xbfb1442c -1078901716
ecx            0x0 0
edx            0x10 16
ebx            0xbfb14460 -1078901664
esp            0xbfb14410 0xbfb14410
ebp            0xbfb14448 0xbfb14448
esi            0x8048440 134513728
edi            0x8048320 134513440
eip            0x804840a 0x804840a <main+54>
```

Einzelne Register auslesen

```
(gdb) print $esp
$1 = (void *) 0xbfb14410
(gdb) print $eip
$2 = (void (*)()) 0x804840a <main+54>
```

Register setzen

```
(gdb) set $eip = 0x080483d4
```

GDB | Speicherinhalte

Stack hexadezimal

```
(gdb) x/16x $esp
0xbfb14410: 0xbfb1442c 0x00000041 0x00000010 0xb77d417e
0xbfb14420: 0xb78a3ff4 0x080495dc 0xbfb14438 0x41414141
0xbfb14430: 0x41414141 0x41414141 0x41414141 0x00000001
0xbfb14440: 0xbfb14460 0xb78a3ff4 0xbfb144b8 0xb7780455
(gdb) print $ebp
$6 = (void *) 0xbfb14448
```

Variablen: Adresse und Inhalt ausgeben

```
(gdb) print buf
$7 = 'A' <repeats 16 times>
(gdb) print &buf
$8 = (char (*)[16]) 0xbfb1442c
(gdb) print i
$9 = 1
(gdb) print &i
$10 = (int *) 0xbfb1443c
```

Stack als String

```
(gdb) x/s  0xbffa53ec
0xbffa53ec:  "abcdefghijklmnopqrstuvwxyz"
```

Speicherstelle setzen

```
(gdb) set *0xbfe2b920 = 0x0
```

GDB | Disassemblen

Textsegment: `disass`

```
(gdb) disass main
0x080483d4 <main+0>: lea    0x4(%esp),%ecx
0x080483d8 <main+4>: and    $0xfffffff0,%esp
0x080483db <main+7>: pushl  -0x4(%ecx)
0x080483de <main+10>: push   %ebp
0x080483df <main+11>: mov    %esp,%ebp
0x080483e1 <main+13>: push   %ebx
0x080483e2 <main+14>: push   %ecx
...
```

Datensegmente (bspw. Stack): `x/i`

```
(gdb) x/4i  0xbffa53ec
0xbffa53ec: popa
0xbffa53ed: bound  %esp,0x64(%ebx)
0xbffa53f0: gs
0xbffa53f1: addr16 pushw $0x6a69
```

GDB | Help

Hier konnte nur eine kurze Einführung der wichtigsten Kommandos gegeben werden.
Aber GDB ist gut dokumentiert:

- Offizielle Doku: http://sourceware.org/gdb/current/onlinedocs/gdb/
- Eingebaute Hilfe: `(gdb) help`, bspw. `help stack` oder `help all`
- Meist genutzten Befehle: `man gdb`
- Viele 3rd Party Tutorials ($\rightarrow$ Google)
- ...

Ende