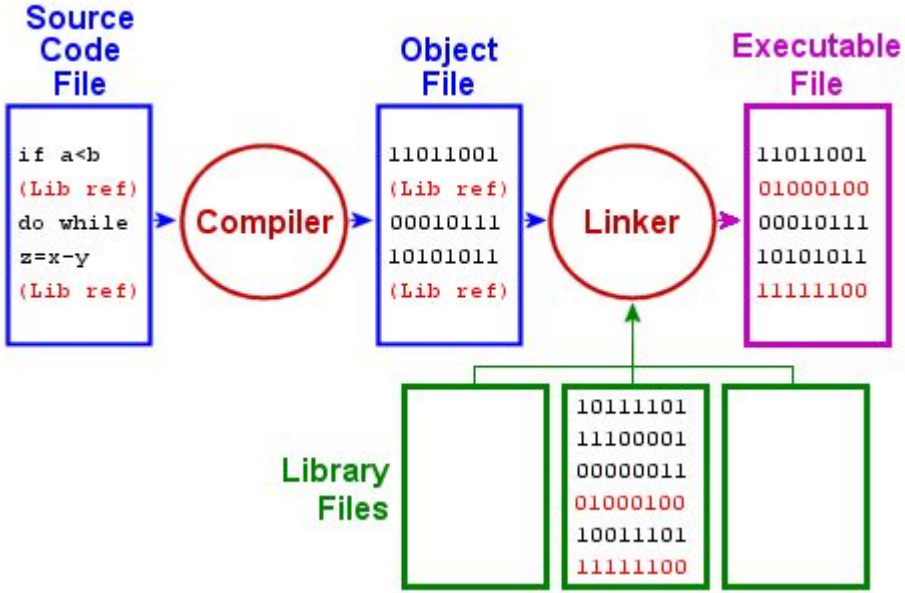
Assembler-Crashkurs

Kompilierung eines Programmes

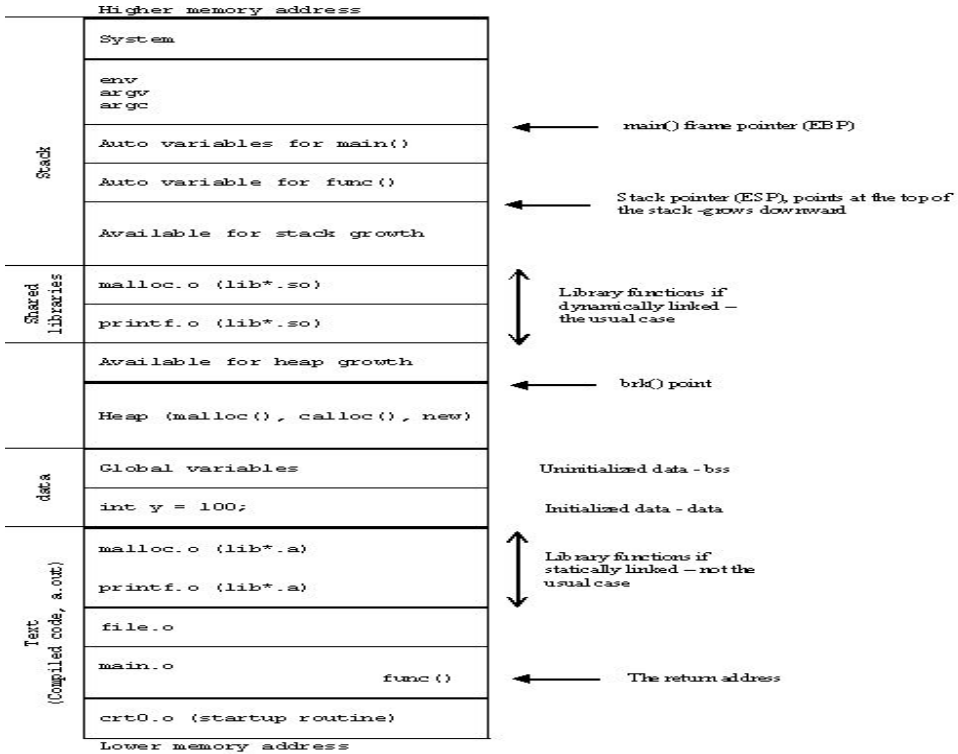
Mnemonics, ld, GCC, AS

Wie entsteht eine Binary?

1. Wir schreiben Source-Code
2. Wir Kompilieren diesen Source-Code (es entsteht Object-Code/Assembler)
3. Wir linken das Programm → es entsteht eine ELF-Binary



ELF-Binary Segmente



HEAP vs. Stack

STACK

Zusammenhängender Speicherbereich.

Enthält die lokalen Variablen und Argumente einer Funktion.

Im Normalfall: Deutlich kleiner als der Heap

Aufräumen “standartisiert”

Pro Thread eigener Stack

HEAP:

Dynamische allozierte Datenstrukturen mit “längerer” Laufzeit (z.B Struct’s und Objekte einer Klasse)

Der Heap hat meist eine eigene Verwaltungsstruktur (z.B. B-Baum)

Bei C/C++ müssen diese explizit wieder freigegeben werden (free()).

Andere Sprachen haben meist eine eigene Garbage-Collection

Pro Prozess: nur 1 Heap

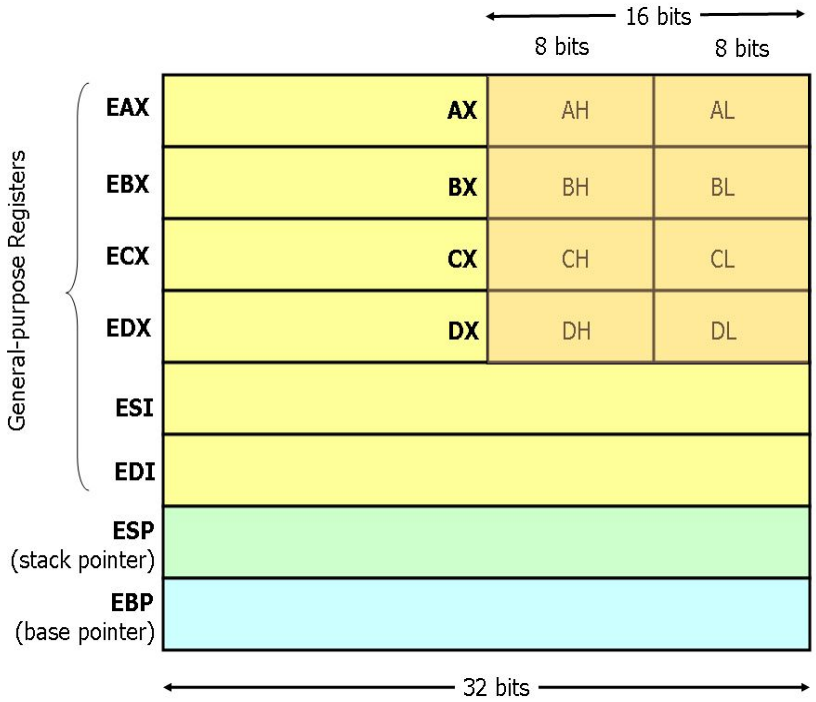
x86

Wir haben bei jeder CPU einen Instruction Pointer (EIP bei x86)

Dieser zeigt auf eine Speicheraddresse die dann hoch iteriert wird. Diese ist vom Programmierer nur indirekt modifizierbar (durch jmp,ret,call)

EBP: zeigt auf die höchste Adresse im aktuellen Stackframe
ESP: zeigt auf die niedrigste Adresse im aktuellen Stackframe

```
Mehrere General-Purpose-Register:
EAX: beinhaltet immer Return-Werte
ECX: Schleifenzähler, Zähler im Allgemeinen
EDX: Arithmetik
EDI, ESI - Stream Operationen
```



Assembler-Befehle (die wichtigsten)

MOV EAX, EBX	Bewege etwas von EBX nach EAX
XOR EAX, EAX	XOR die beiden Register und packe es in EAX
SUB/ADD/MUL EAX	Arithmetik
PUSH EBX	Packe Wert von EBX auf den Stack
POP EBX	Nehme Wert von Stack und lege in EBX
JMP .foo	Springe zu Label .foo
Call foo	Pushe EIP/RIP+4 auf Stack und Springe zu Foo
RET	Nehme Wert von Stack und Springe dorthin

Stack

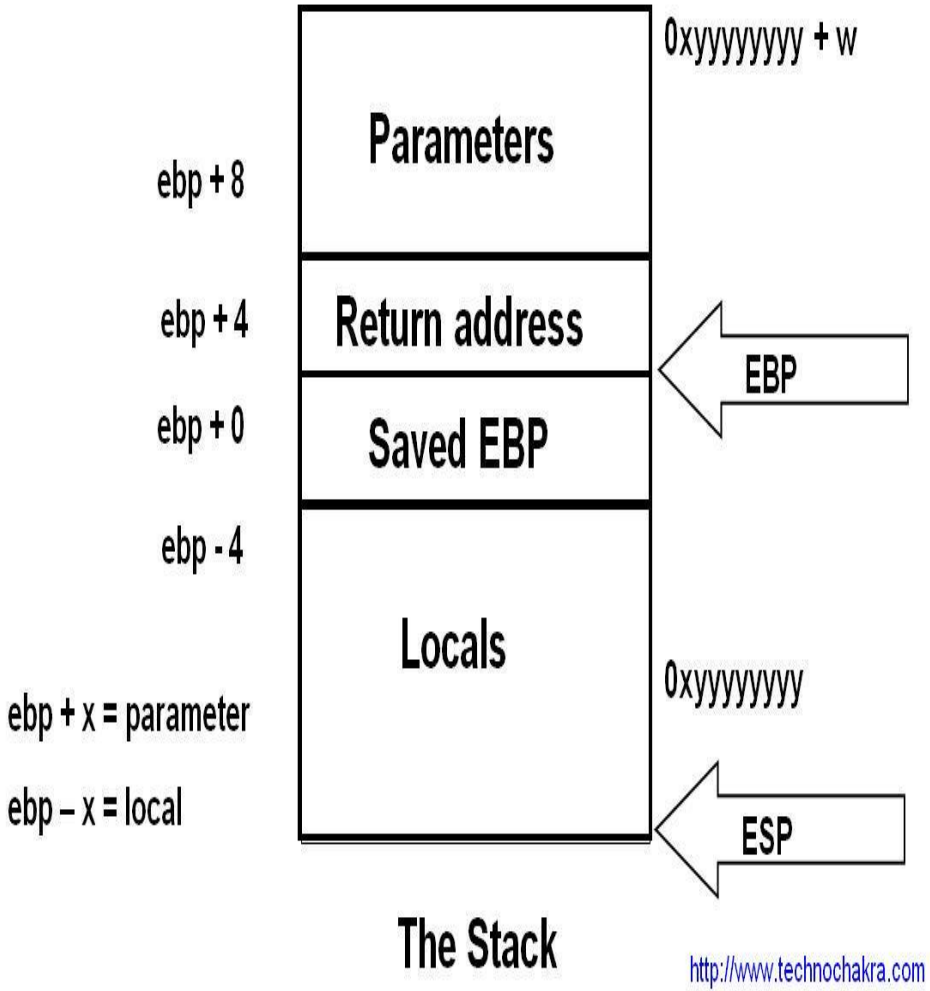
Wächst von Oben nach unten.

EBP zeigt damit immer auf die “kleinste-Adresse”-Adresse

ESP zeigt immer auf die neueste Adresse des aktuellen Stack-Frames.

Daumenregel:

Pro Funktion wird ein Stack-Frame aufgemacht



Konventionen

BIG-Endian, Little-Endian

Calling-Conventions

CDECL: Jedes Argument wird von Rechts nach Links über den Stack gepusht	`/* foo(int a, int b, int c)*/`
	
	`push c;`
	
	`push b;`
	
	`push a;`
	
	`call foo;`

Syscall (maximal 6 Argumente), Aufruf per int80 über die Interrupt-Table.
 	
`Syscall# P1 P2 P3 P4 P5 P6`
 	
`Eax ebx ecx edx esi edi ebp`
	`//exit(0);`
	
	`Mov eax,1;`
	
	`Mov ebx,0;`
	
	`Int 80h;`

Big vs. Little-Endian

X86 nutzt ein Little-Endian-Format, dass heißt das Most-Signifcant-Bit(MSB) steht Rechts.

Aus 0x12345678 wird: ‘/x78/x56/x34/x12’

Radare2

Was ist radare2

- Ursprünglich als Tool zur forensischen Untersuchung von File-Systemen gedacht
- Hat sich stück für stück zu einem Reverse-Engineering-Framework gewandelt (… wenn man gut ist, ungefähr mit IDA-Pro vergleichbar)
- Idee: Alles über die Kommandozeile. Viele Subprogramme
- Kommandos sind selbstdokumentierend
- Referenzen werden mit @ gemacht (vergleichbar mit *):
- ps @ 0x1337d00d
    - 0x1337d00d: “this is a 1337 string”

Wichtigste Kommandos

Kommando	Was passiert?
`R2 -AA <Programm>`	Programm wird direkt analysiert
`r2 -d <Programm>`	Programm wird im Debug Modus geöffnet
`[0x00000000]> i`	Zeigt Informationen an
`[0x00000000]> afl`	Listet alle Funktionen
`[0x00000000]> s <SPEICHERADRESSE>`	Gehe zu einer Speicheraddresse
`[0x00000000]> px[dwq] @ <SPEICHERADRESSE>`	Print Hex String (PXS)
`[0x00000000]> db <SPEICHERADRESSE>`	Debugger-Breakpoint bei <SPEICHERADRESSE>
`[0x00000000]>dc`	Debug Continue, geht bis<Breakpoint oder EOF>
`[0x00000000]>ds/S`	steps

GDB

Gnu Debbugger

1. Standard-Debugger wenn man unter Linux C programmiert
2. In der Standardausführung relativ “beschränkt” und kryptisch zu bedienen, deswegen über Python-API Erweiterungen:
   a. GEF
   b. Pwndbg
   c. PEDA
3. Tun ungefähr alle das gleiche an Grundfeatures:
   a. Variablen “Guessing” aus dem Code (“MAGIE!”)
   b. Teilweise ROP Erkennungsfeatures (pwndbg ←-->RADARE2 Integration)
   c. Eigentlich wichtiges: zeigt Stack und Register während des Debuggings an
   d.

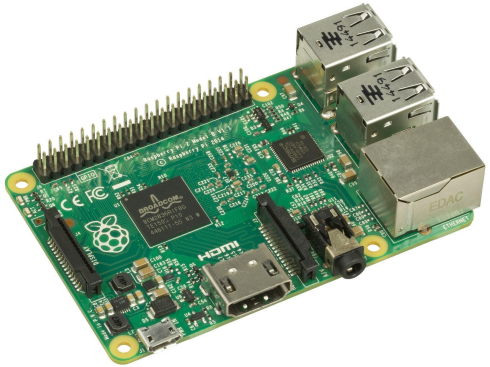
Wichtigste Befehle

BEFEHL	Was passiert
`I file`	Zeigt die Sektionen
`I func`	Zeigt alle Funktionen an
`b <func+offset> oder *Speicheradresse`	Setzt einen Breakpoint
`r (optional) <<< INPUT`	Lässt das Programm laufen
`s`	Single step
`n`	Over step
`fin`	Lauf bis man aus Funktion ist
`Del <BREAKPOINT-Nummer>`	Löscht breakpoint
`! <CMD>`	Führt Shell-Kommando aus
`p [<symbolname>|<expr>]`	Print

ARM

Warum ARM?

- Sehr beliebt, besonders im Embedded-Bereich
  - Geringer Verbrauch, hohe Leistung
- Smartphone, IoT, Spielekonsolen, Netzwerk-Hardware, …



CISC vs RISC

Complex Instruction Set Computing

Reduced Instruction Set Computing

RISC

- Simple aber vielseitige Instruktionen
- Weniger Taktzyklen je Instruktion
- Load-Store Architektur
- Mehr General Purpose Registers

ARM Architektur

- Viele Varianten, direkt von ARM oder lizenziert
  - (ARM1176JZF-S, Cortex-R, Cortex-A57)
- Alternativer Instruktionsatz: Thumb-Mode
- Big- und Little-Endian (Standard ist Little-Endian)
- Bedingte Ausführung (BLT, ADDLT)
- Heterogene Multi-Core Konfiguration möglich (big.LITTLE)

ARM Mnemonics

```
MNEMONIC{S}{condition} {Rd}, Operand1, Operand2
```

{S} - An optional suffix. If S is specified, the condition flags are updated on the result of the operation
{condition} - Condition that is needed to be met in order for the instruction to be executed
{Rd} - Register (destination) for storing the result of the instruction
Operand1 - First operand. Either a register or an immediate value
Operand2 - Second (flexible) operand. Can be an immediate value (number) or a register with an optional shift

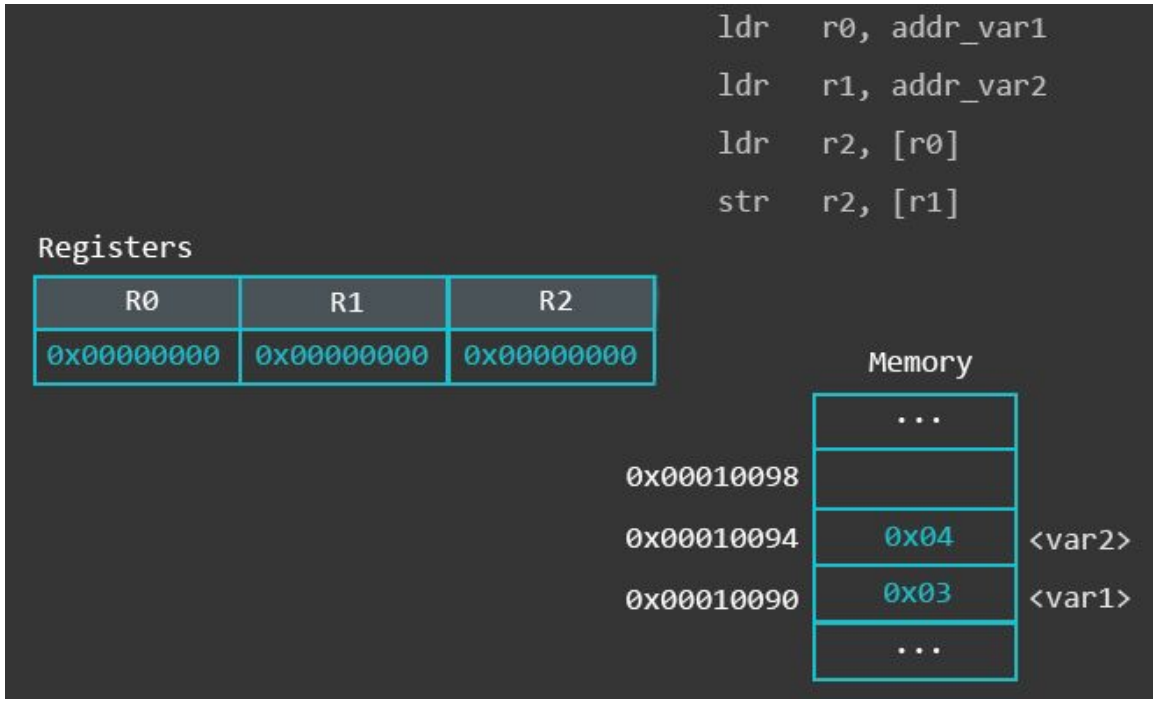
ARM Assembly kann auch exotisch:

Quelle: https://azeria-labs.com/arm-instruction-set-part-3/

Wichtigste Befehle

`LDR R0 ADDR`	Lade Wert aus ADDR in R0
`STR R0 ADDR`	Speichere Wert aus R0 nach ADDR
`MOV R0 R1`	Kopiere Wert aus R1 nach R0
`BL ADDR`	Branch with Link
`LSL R0 R1 2`	Speichere R1<<2 in R0
`ADD,SUB,MUL,ORR,EOR`	...

Load / Store



Quelle: azeria-labs.com

ARM Exoten

`STR R2, [R1, R2, LSL#2]`	Speichere Wert aus R2 nach R1+ (R2<<2)

Register in ARM und x86

ARM	Description	x86
R0	General Purpose	EAX
R1-R5	General Purpose	EBX, ECX, EDX, ESI, EDI
R6-R10	General Purpose	–
R11 (FP)	Frame Pointer	EBP
R12	Intra Procedural Call	–
R13 (SP)	Stack Pointer	ESP
R14 (LR)	Link Register	–
R15 (PC)	<- Program Counter / Instruction Pointer ->	EIP
CPSR	Current Program State Register/Flags	EFLAGS

Quelle: azeria-labs.com

But wait, there’s more!

https://azeria-labs.com/writing-arm-assembly-part-1/

http://infocenter.arm.com