Spezielle Techniken und Technologien der Informatik

Reverse Engineering

Robert Baumgartl

18. März 2023

Reverse Engineering

Unter **Reverse Engineering** versteht man die Analyse von Systemen mit dem Ziel, ihren inneren Aufbau, ihre Struktur zu erkennen und ihre Arbeitsweise zu verstehen. Zwei typische Anwendungsfälle sind

- die Rekonstruktion des Quellcodes von Programmen, die nur als Binärabbild vorliegen, z.B. Schadsoftware,
- die Analyse von Kommunikationsprotokollen proprietärer Software[1] zur Herstellung von Interoperabilität zu anderer, meist freier, Software.

In der Lehrveranstaltung werden wir dazu geeignete Techniken und Werkzeuge kennenlernen. Als Übungsplattform nutzen wir die Intel-Architektur unter Linux.

Empfehlenswerte Literatur

- Bruce Dang, Alexandre Gazet und Elias Bachaalany. *Practical Reverse Engineering*. Wiley, 2014
- Eldad Eilam. *Reversing: Secrets of Reverse Engineering*. Wiley, 2005

[1] Beispielsweise ist das der populären Software *Samba* zugrundeliegende Protokoll SMB/CIFS per Reverse Engineering aufgeklärt worden.

Weitere Informationsquellen

- Intel 64 and IA-32 Architectures Software Developer's Manual (3 Volumes, knapp 4000 S.); frei verfügbar: http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html
- Einführung in die AT&T-Assembler-Syntax für IA-32 https://en.wikibooks.org/wiki/X86_Assembly/GAS_Syntax
- Informationen zum Gnu Debugger `gdb` http://www.gnu.org/software/gdb/documentation/

Wozu?

Sicherheitsbezogene Arbeiten

- Analyse bösartiger Software (z. B. Kryptotrojaner)
- Analyse von Applikationen und Betriebssystemen mit dem Ziel, Schwachstellen zu finden
- Analyse kryptografischer Algorithmen und deren Implementierung (Vortrag[2] von Philippe Oechslin auf dem 26C3)
- Unterwanderung von DRM

Softwareentwicklung

- Herstellung der Interoperabilität zu proprietärer Software bzw. Protokollen (.doc-Format, NTFS-Treiber in Linux, Samba, „Hacking“ der Sony Playstation 3)
- Entwicklung konkurrierender Software
- Test der Robustheit und allgemeinen Qualität von Software

[2] https://events.ccc.de/congress/2009/Fahrplan/events/3703.en.html

Rechtliche Aspekte

- unterschiedliche nationale Gesetzgebung
- in Dt. grundsätzlich zunächst erlaubt
- Ausnahme: Umgehung von Kopierschutzmechanismen aka „Cracking“ (privat legal, gewerblich illegal – §95a UrhG)
- Lizenz verbietet das Reverse Engineering häufig
- Analyse von Protokollen ist grundsätzlich erlaubt
- USA: DMCA (vgl. Sony vs. George Hotz, der *PS3-Hack*)

Beschränkung

HLL[3]: C, weil

- leicht zu disassemblieren, da maschinennah,
- BS und die meisten Applikationen (und Schadcode) in C implementiert wurden und werden,
- alle Teilnehmer C könnten müssten ☺.

ISA[4]: IA-32 = Intel-Prozessoren im 32-Bit-Betrieb, weil

- in den Laboren diese Prozessoren verbaut sind,
- Befehlssatz gut zu verstehen; viel Literatur.
- (IMHO) leichter zu verstehen, als 64-Bit-Assembler

BS: Linux, weil

- offene und freie Werkzeuge existieren,
- ich offene und freie Software bevorzuge.

---

[3] **H**igh **L**evel **L**anguage – Höhere Programmiersprache

[4] **I**nstruction **S**et **A**rchitecture - der Befehlssatz und die Architektur des Prozessors

Werkzeuge

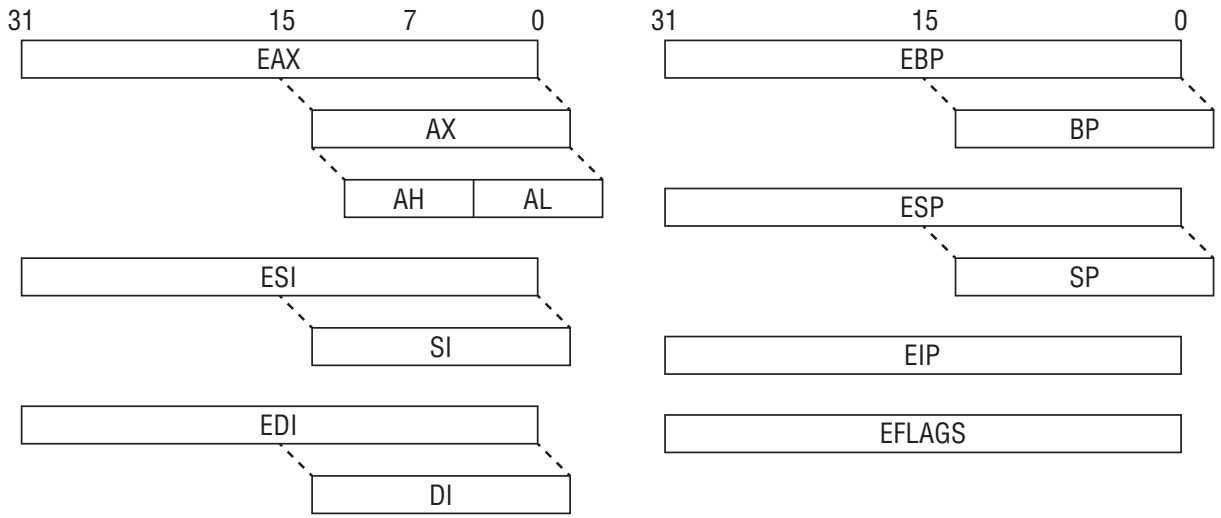
1. Disassembler
   - überführt (maschinenlesbares) Binary in (menschenlesbaren) Assemblercode
   - `objdump -d`
   - `gdb`
2. Decompiler
   - überführt (maschinenlesbares) Binary in Hochsprachencode (bzw. versucht dies)
   - z. B. Hex-Rays IDAPro (kommerziell)
   - Ghidra als Open-Source-Alternative (https://ghidra-sre.org/)
3. Debugger
   - zur schrittweisen Ausführung des zu analysierenden Codes, Speicherinspektion usw.
   - Klassiker: `gdb`

Intel-Syntax vs. AT&T-Syntax

- ins dest,src vs. ins src,dest
- nur AT&T: an Befehl angehängtes `l`, `w` oder `b` – verrät Länge des Operanden (32, 16 oder 8 Bit, respektive)
- register vs. %register
- Speicheroperanden mit [] vs. Speicheroperanden mit () indexiert
- Konstanten („Immediate-Werte“) wird bei AT&T ein ‘$’ vorangestellt

vgl. https://imada.sdu.dk/~kslarsen/dm546/Material/IntelnATT.htm

- 8 Universalregister á 32 Bit: EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP
- Befehlszähler EIP
- Flagregister EFLAGS
- Datentypen: 8 Bit (*Byte*), 16 Bit (*Word*), 32 Bit (*Double Word*), 64 Bit (*Quad Word*, nur EDX:EAX)

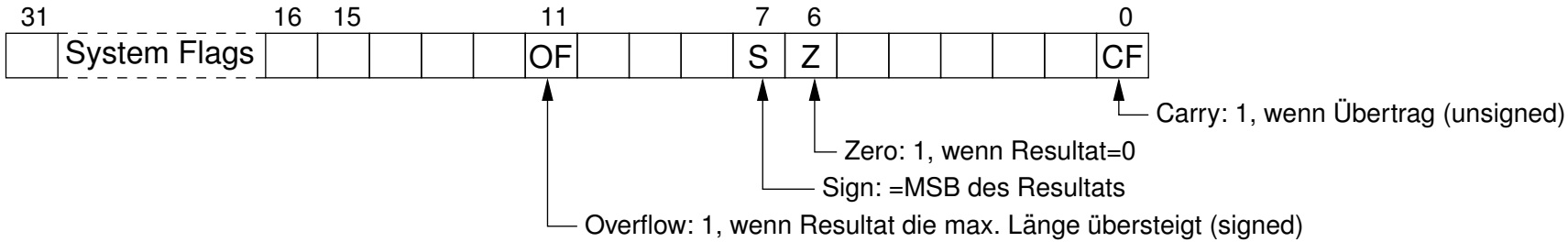


Register	Semantik
EAX	Resultatregister
ECX	Zählerregister in Schleifen
ESP	Stack Pointer (TOS)
EBP	zeigt auf Beginn des aktuellen Stackframes
ESI	Quelladresse bei Blockoperationen
EDI	Zieladresse bei Blockoperationen
EIP	Adresse der als nächstes auszuführenden Instruktion

Viele weitere Register existieren, haben aber Spezialbedeutung z. B. für Gleitkommaarithmetik, Control Registers CRx, Model Specific Registers (MSR), Multimedia Extensions (MMX), Streaming SIMD Extensions (SSE) . . .

IA-32 CPU

EFLAGS



CONDITIONAL CODE	ENGLISH DESCRIPTION	MACHINE DESCRIPTION
B/NAE	Below/Neither Above nor Equal. Used for unsigned operations.	CF=1
NB/AE	Not Below/Above or Equal. Used for unsigned operations.	CF=0
E/Z	Equal/Zero	ZF=1
NE/NZ	Not Equal/Not Zero	ZF=0
L	Less than/Neither Greater nor Equal. Used for signed operations.	(SF ^ OF) = 1
GE/NL	Greater or Equal/Not Less than. Used for signed operations.	(SF ^ OF) = 0
G/NLE	Greater/Not Less nor Equal. Used for signed operations.	((SF ^ OF) | ZF) = 0

Die wichtigsten Instruktionen

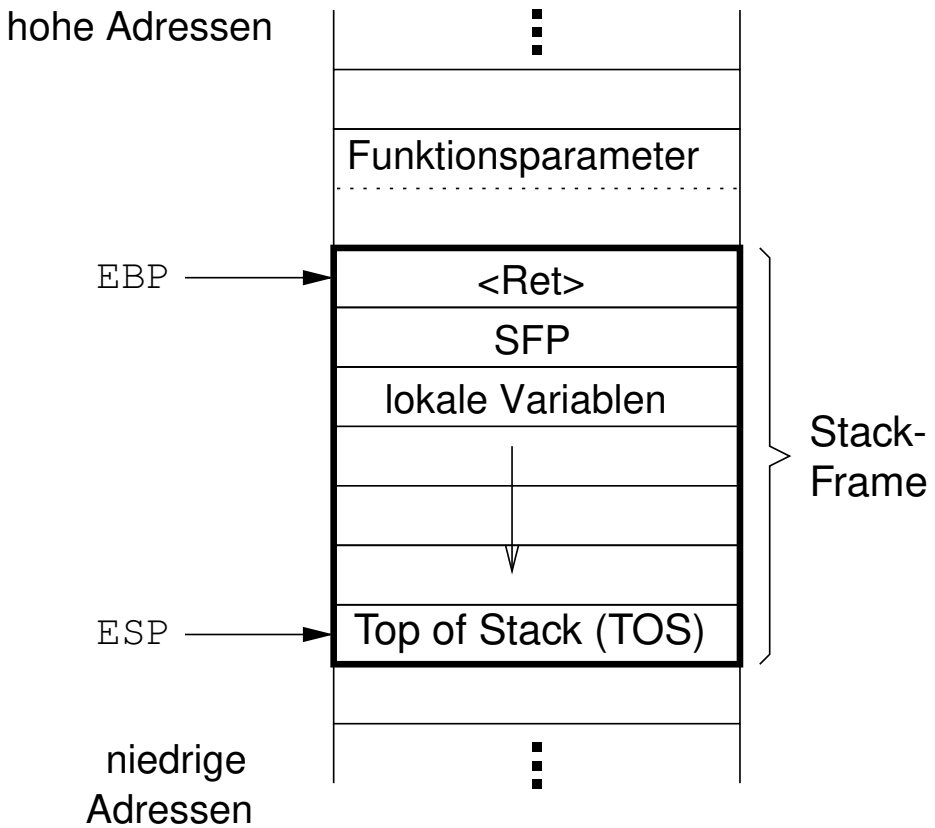
- Kopieren von Daten Speicher $\leftrightarrow$ Register und Register $\leftrightarrow$ Register: `mov`
- $\sim$ mit Vorzeichenerweiterung: `movsxb`, `movsxw`; mit Nullerweiterung: `movzb`, `movzw`
- Laden einer Adresse (nicht: eines Wertes): `lea` (“Load Effective Address”)
- arithmetische Operationen: `mul`, `add`, `sub`, `inc`, `dec`
- logische Operationen: `or`, `and`, `shr`, `shl`, `xor`
- Stackoperationen und Funktionsaufrufe: `push`, `pop`, `call`, `ret`
- Sprünge/bedingte Sprünge mit Vergleichen: `jmp`, `j`*cc* mit `cmp` und `test`

Der Stack

- Bereich des Speichers
- wächst in Richtung kleinerer Adressen
- LIFO- (Last In First Out) Prinzip
- für Daten mit begrenzter Gültigkeitsdauer
  - Parameter einer Funktion
  - Rückkehradresse beim Sprung in eine Funktion (CALL)
  - lokale Variablen einer Funktion
  - temporäre Kopien von Registerinhalten (Save/Restore)
  - nutzerdefinierte DS $\rightarrow$ alloca()
- organisiert in Frames, die jeweils durch die Register `EBP` und `ESP` begrenzt werden
- `ESP` zeigt stets auf oberstes Element des Stacks (Top of Stack, TOS)
- Instruktionen: `pop` und `push`

Der Stack

Aufbau eines Stackframes



`push` und `pop`

`push`

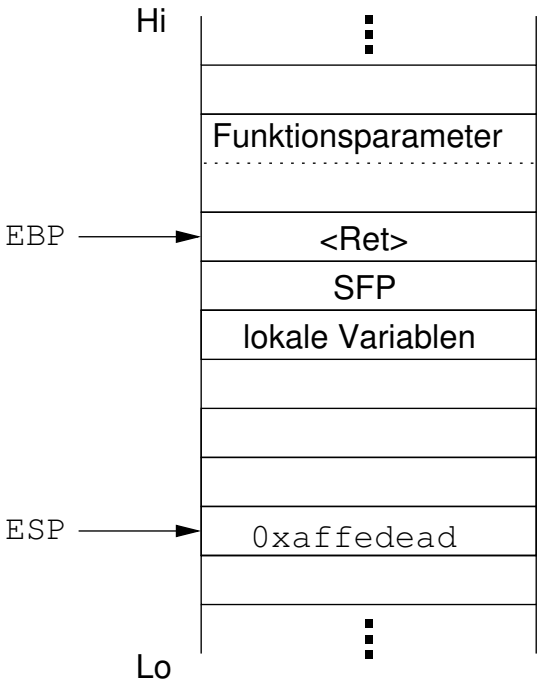
- dekrementiert **zuerst** `ESP` (gewöhnlich -4, da 32-Bit-Worte abgelegt werden)
- legt **danach** den Operanden auf die Adresse, auf die `ESP` verweist

`pop`

- schreibt **zuerst** das Datum, auf das `ESP` verweist, in den übergebenen Operanden
- inkrementiert **danach** den Stackpointer `ESP` (typisch: +4)
- Operand ist in beiden Fällen ein Register oder eine Speicherstelle

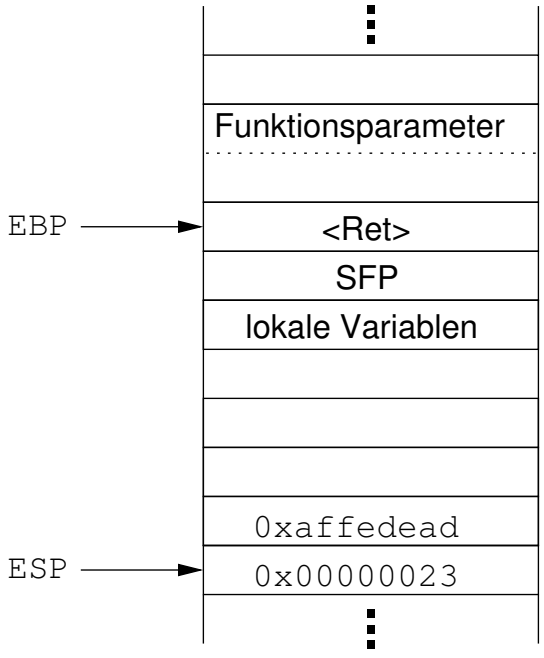
`push` und `pop`

Ablauf bei einer `push`-Instruktion



a) vor `push`-Operation

Register EAX enthält Wert 0x23.
ausgeführte Instruktion:
`pushl %eax`



b) nach `push`-Operation

Call-Return-Mechanismus

`call <address>`

- legt aktuelle Adresse auf den Stack (d. h., `push %eip`)
- `<address>` $\rightarrow$ `%eip`
- $\rightsquigarrow$ Sprung zu `<address>`

`ret`

- holt bei `call` gespeicherte Adresse zurück (d. h., `pop %eip`)
- $\rightsquigarrow$ Rück„Sprung“ zu auf call folgender Adresse
- Voraussetzung: Stack ist konsistent

Calling Conventions

aka „Wie werden Parameter und Resultate beim Funktionsaufruf kommuniziert?“

- In welcher Reihenfolge werden Parameter übergeben?
- Wie werden Parameter übergeben (in Registern oder über den Stack)?
- Wie wird das Funktionsergebnis zurückgeliefert?
- Welche Registerwerte dürfen durch die Funktion nicht zerstört werden (müssen ggf. innerhalb der Funktion gesichert und restauriert werden)?

	CDECL	STDCALL	FASTCALL
Parameters	Pushed on the stack from right-to-left. Caller must clean up the stack after the call.	Same as `CDECL` except that the callee must clean the stack.	First two parameters are passed in `ECX` and `EDX`. The rest are on the stack.
Return value	Stored in `EAX`.	Stored in `EAX`.	Stored in `EAX`.
Non-volatile registers	`EBP`, `ESP`, `EBX`, `ESI`, `EDI`.	`EBP`, `ESP`, `EBX`, `ESI`, `EDI`.	`EBP`, `ESP`, `EBX`, `ESI`, `EDI`.

Abbildung: Die wichtigsten Calling Conventions der IA-32[5]

[5] Bruce Dang, Alexandre Gazet und Elias Bachaalany. *Practical Reverse Engineering*. Wiley, 2014, S. 16.

Elementare Konstrukte in C

- Systemrufe (Linux)
- if-else-Statement
- Schleifen
- Funktionsaufrufe
- `switch-case`-Statement ($\rightarrow$ Praktikum)

Wiederholung BS1: Systemrufe

- sind Dienste, die das Betriebssystem mitbringt
- werden im Adressraum des Kernels erbracht (Umschaltung Usermode $\leftrightarrow$ Kernelmode nötig)
- können daher nicht durch *Call-Return*-Mechanismus erbracht werden
- stattdessen: Nutzung eines Interrupts oder `sysenter`- und `sysleave`-Instruktionen
- Beispiele: `fork()`, `open()`, `read()`, `socket()` usw.

Aufrufkonvention Systemruf Linux (32Bit)

- jeder Syscall hat eine eineindeutige ID (Systemrufnummer; z. B. `exit()`: 1, `fork()`: 2, `read()`: 3 usw.
- Systemrufnummer in eax
- Argumente in ebx, ecx, edx, esi, edi (in dieser Reihenfolge)
- Systemeintritt durch `int 0x80`
- (Systemdienst wird im Kernelmode ausgeführt)
- Resultatwert in eax
- Systemaustritt mittels `iret`

Systemrufnummern mit Parametern:
http://docs.cs.up.ac.za/programming/asm/derick_tut/syscalls.html

Systemrufe

„Hello, world“ in Assemblercode

Listing: `hello32.s`

```
# hello32.s
# "Hello, world!" in IA-32 assembler program
#
  .text
  .global _start

_start:
  movl $4,%eax
  movl $1,%ebx
  movl $msg,%ecx
  movl $0xe,%edx
  int $0x80
  movl $1,%eax
  movl $42,%ebx
  int $0x80

  .data
msg:
  .ascii "Hello, world!\n\00"
```

- `gcc -m32 -nostdlib -o hello32 hello32.s`

Systemrufe

Beispiel

Listing: `helloasm.S`

```
  .text
  .global _start
_start:
  movl $0, %eax
  xorl %ebx, %ebx /* 0 -> %ebx */
  xorl %edx, %edx
  jmp string      /* push string addr */
code:
  pop %ecx        /* ecx <-- string addr */
  movb $01, %bl   /* filedesc, stdout */
  movb $15, %dl   /* string lgth */
  movb $04, %al   /* 'write(stdout, addr, lgth)' */
  int $0x80
  decb %bl
  movb $01,%al    /* 'exit(0)' */
  int $0x80
string:
  call code
  .ascii "Hello, world!\x0a\x00"
```

Systemrufe

`helloasm.S`, analysiert

if-else-Statement

Listing: `ifelse.c`

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

int main(int argc, char*
      argv[])
{
  int a, b=1;

  if (b==42) {
    a=23;
  }
  else {
    a=42;
  }
  exit(0);
}
```

Listing: Disassemblat zu `ifelse.c`

```
(gdb) disas main
Dump of assembler code for function main:
   0x080483fb <+0>:     lea    0x4(%esp),%ecx
   0x080483ff <+4>:     and    $0xfffffff0,%esp
   0x08048402 <+7>:     pushl  -0x4(%ecx)
   0x08048405 <+10>:    push   %ebp
   0x08048406 <+11>:    mov    %esp,%ebp
   0x08048408 <+13>:    push   %ecx
   0x08048409 <+14>:    sub    $0x14,%esp
   0x0804840c <+17>:    movl   $0x1,-0xc(%ebp)
   0x08048413 <+24>:    cmpl   $0x2a,-0xc(%ebp)
   0x08048417 <+28>:    jne    0x8048422 <main
         +39>
   0x08048419 <+30>:    movl   $0x17,-0x10(%ebp)
   0x08048420 <+37>:    jmp    0x8048429 <main
         +46>
   0x08048422 <+39>:    movl   $0x2a,-0x10(%ebp)
   0x08048429 <+46>:    sub    $0xc,%esp
   0x0804842c <+49>:    push   $0x0
   0x0804842e <+51>:    call   0x80482e0 <
         exit@plt>
End of assembler dump.
```

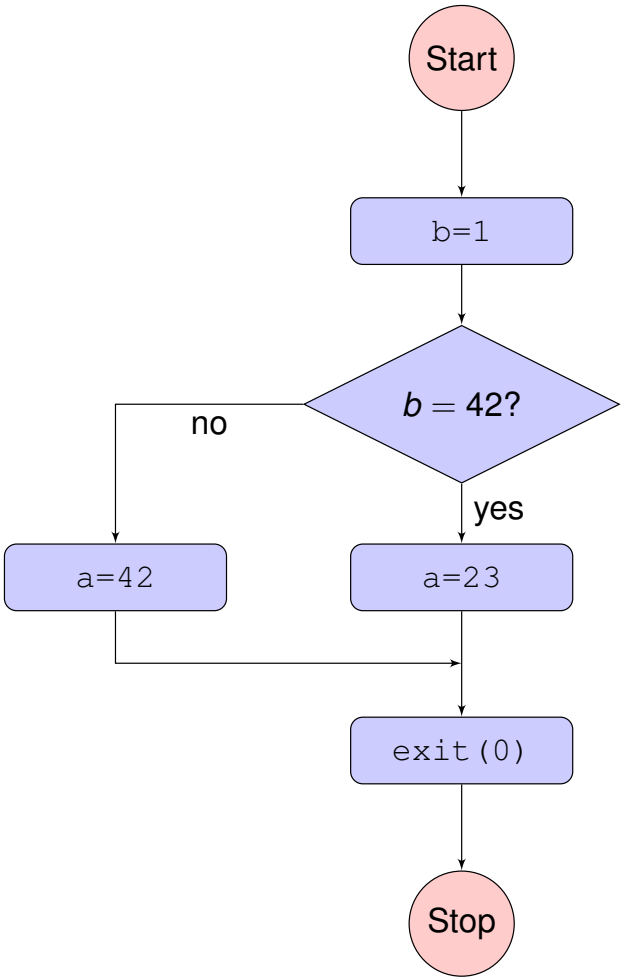
if-else-Statement

Listing: Disassemblat zu `ifelse.c`

```
(gdb) disas main
Dump of assembler code for function main:
   0x080483fb <+0>:     lea    0x4(%esp),%ecx
   0x080483ff <+4>:     and    $0xfffffff0,%esp
   0x08048402 <+7>:     pushl  -0x4(%ecx)
   0x08048405 <+10>:    push   %ebp
   0x08048406 <+11>:    mov    %esp,%ebp
   0x08048408 <+13>:    push   %ecx
   0x08048409 <+14>:    sub    $0x14,%esp
   0x0804840c <+17>:    movl   $0x1,-0xc(%ebp)
   0x08048413 <+24>:    cmpl   $0x2a,-0xc(%ebp)
   0x08048417 <+28>:    jne    0x8048422 <main+39>
   0x08048419 <+30>:    movl   $0x17,-0x10(%ebp)
   0x08048420 <+37>:    jmp    0x8048429 <main+46>
   0x08048422 <+39>:    movl   $0x2a,-0x10(%ebp)
   0x08048429 <+46>:    sub    $0xc,%esp
   0x0804842c <+49>:    push   $0x0
   0x0804842e <+51>:    call   0x80482e0 <exit@plt>
End of assembler dump.
```

`if-else`-Konstrukt

PAP für den Assemblerquelltext



Schleifen

`for`-Konstrukt

Listing: `loop1.c`

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

int main(int argc, char *argv[])
{
  int c;

  for(c=0; c<23; c++) {
    printf("%d\n", c);
  }
  exit(0);
}
```

Schleifen

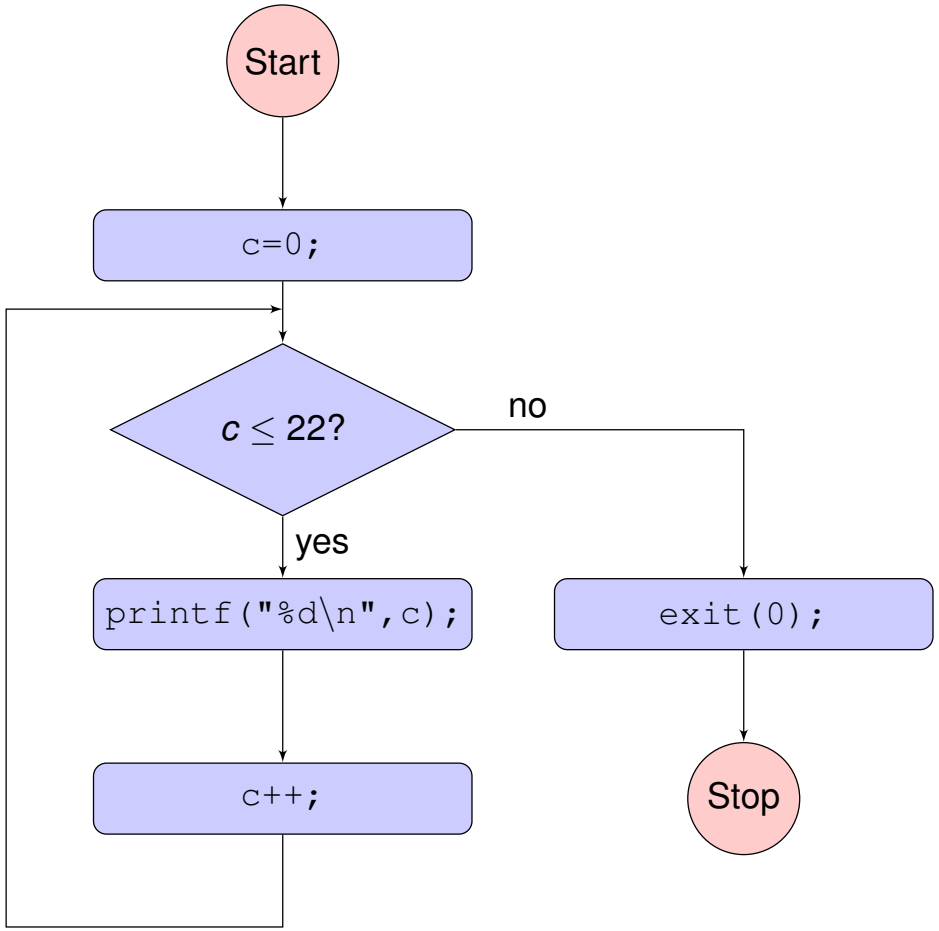
`for`-Konstrukt

Listing: Disassemblat zu `loop1.c`

```
(gdb) disas main
Dump of assembler code for function main:
   0x0804842b <+0>:     lea    0x4(%esp),%ecx
   0x0804842f <+4>:     and    $0xfffffff0,%esp
   0x08048432 <+7>:     pushl  -0x4(%ecx)
   0x08048435 <+10>:    push   %ebp
   0x08048436 <+11>:    mov    %esp,%ebp
   0x08048438 <+13>:    push   %ecx
   0x08048439 <+14>:    sub    $0x14,%esp
   0x0804843c <+17>:    movl   $0x0,-0xc(%ebp)
   0x08048443 <+24>:    jmp    0x804845c <main+49>
   0x08048445 <+26>:    sub    $0x8,%esp
   0x08048448 <+29>:    pushl  -0xc(%ebp)
   0x0804844b <+32>:    push   $0x8048500
   0x08048450 <+37>:    call   0x80482f0 <printf@plt>
   0x08048455 <+42>:    add    $0x10,%esp
   0x08048458 <+45>:    addl   $0x1,-0xc(%ebp)
   0x0804845c <+49>:    cmpl   $0x16,-0xc(%ebp)
   0x08048460 <+53>:    jle    0x8048445 <main+26>
   0x08048462 <+55>:    sub    $0xc,%esp
   0x08048465 <+58>:    push   $0x0
   0x08048467 <+60>:    call   0x8048310 <exit@plt>
```

`for`-Konstrukt

PAP für den Assemblerquelltext



Listing: `loop2.c`

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

int main(int argc, char *argv[])
{
  int c;

  c = 0;
  goto label1;
 label2:
  printf("%d\n", c);
  c++;
 label1:
  if (c<=22)
    goto label2;
  exit(0);
}
```

- Analyse Assemblat?

Schleifen

„Verschlüsselung“ von Zeichenketten

Listing: array.c

```c
#include <stdio.h>

int main(void)
{
  int c;
  char pass[] = "1ENZRJZ_B";
  char key[]  = "a$=)%z(;c";

  for (c=0; c<10; c++) {
    pass[c] ^= key[c];
  }
  printf("%s\n", pass);
  return 23;
}
```

```
0x000011b5 movl   $0x5a4e4531,-0x16(%ebp)
0x000011bc movl   $0x5f5a4a52,-0x12(%ebp)
0x000011c3 movw   $0x42,-0xe(%ebp)
0x000011c9 movl   $0x293d2461,-0x20(%ebp)
0x000011d0 movl   $0x3b287a25,-0x1c(%ebp)
0x000011d7 movw   $0x63,-0x18(%ebp)
0x000011dd movl   $0x0,-0xc(%ebp)
0x000011e4 jmp    0x120c <main+115>
0x000011e6 lea    -0x16(%ebp),%ecx
0x000011e9 mov    -0xc(%ebp),%edx
0x000011ec add    %ecx,%edx
0x000011ee movzbl (%edx),%ecx
0x000011f1 lea    -0x20(%ebp),%ebx
0x000011f4 mov    -0xc(%ebp),%edx
0x000011f7 add    %ebx,%edx
0x000011f9 movzbl (%edx),%edx
0x000011fc xor    %edx,%ecx
0x000011fe lea    -0x16(%ebp),%ebx
0x00001201 mov    -0xc(%ebp),%edx
0x00001204 add    %ebx,%edx
0x00001206 mov    %cl,(%edx)
0x00001208 addl   $0x1,-0xc(%ebp)
0x0000120c cmpl   $0x9,-0xc(%ebp)
0x00001210 jle    0x11e6 <main+77>
0x00001212 sub    $0xc,%esp
0x00001215 lea    -0x16(%ebp),%edx
0x00001218 push   %edx
0x00001219 mov    %eax,%ebx
0x0000121b call   0x1030 <puts@plt>
```

Schleifen: `array.c`

Analyse Assembler-Code

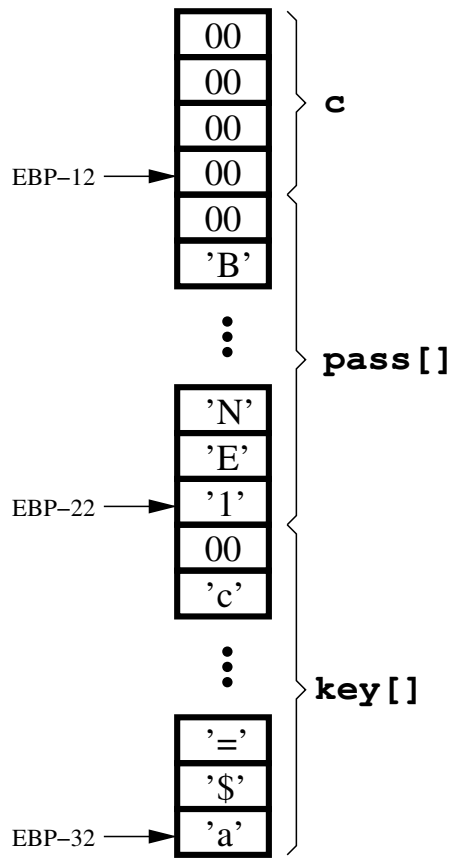
```
0x000011b5 movl   $0x5a4e4531,-0x16(%ebp)
0x000011bc movl   $0x5f5a4a52,-0x12(%ebp)
0x000011c3 movw   $0x42,-0xe(%ebp)
0x000011c9 movl   $0x293d2461,-0x20(%ebp)
0x000011d0 movl   $0x3b287a25,-0x1c(%ebp)
0x000011d7 movw   $0x63,-0x18(%ebp)
0x000011dd movl   $0x0,-0xc(%ebp)          ; 0 -> c
0x000011e4 jmp    0x120c <main+115>
0x000011e6 lea    -0x16(%ebp),%ecx         ; &pass -> ECX
0x000011e9 mov    -0xc(%ebp),%edx          ; c -> EDX
0x000011ec add    %ecx,%edx
0x000011ee movzbl (%edx),%ecx              ; *(pass+c) -> ECX
0x000011f1 lea    -0x20(%ebp),%ebx         ; &key -> EBX
0x000011f4 mov    -0xc(%ebp),%edx          ; c -> EDX
0x000011f7 add    %ebx,%edx
0x000011f9 movzbl (%edx),%edx              ; *(key+c) -> EDX
0x000011fc xor    %edx,%ecx                ; ECX ^ EDX -> ECX
0x000011fe lea    -0x16(%ebp),%ebx         ; &pass -> EBX
0x00001201 mov    -0xc(%ebp),%edx          ; c -> EDX
0x00001204 add    %ebx,%edx
0x00001206 mov    %cl,(%edx)               ; CL -> *(pass+c)
0x00001208 addl   $0x1,-0xc(%ebp)          ; c++
0x0000120c cmpl   $0x9,-0xc(%ebp)
0x00001210 jle    0x11e6 <main+77>
0x00001212 sub    $0xc,%esp
0x00001215 lea    -0x16(%ebp),%edx
0x00001218 push   %edx
0x00001219 mov    %eax,%ebx
0x0000121b call   0x1030 <puts@plt>        ; printf(pass)
```

Init pass

Init key

Sprung, wenn c \le 9

Stacklayout nach Init:



Funktionsaufrufe

Beispiel

Listing: `mul.c`

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

int mul(int x, int y)
{
  int res;

  res = x * y;
  return res;
}

int main(int argc, char* argv[])
{
  int a=23, b=42, erg;

  erg = mul(a, b);
  printf("Ergebnis: %d\n", erg);

  exit(0);
}
```

Funktionsaufrufe

Listing: Disassemblat zu `mul.c`

```
(gdb) disas main
   ...
   0x0804844b <+11>: mov    %esp,%ebp             ; neuer Stackframe
   0x0804844d <+13>: push   %ecx
   0x0804844e <+14>: sub    $0x14,%esp            ; Platz fuer lokale Variablen
   0x08048451 <+17>: movl   $0x17,-0xc(%ebp)      ; a=23
   0x08048458 <+24>: movl   $0x2a,-0x10(%ebp)     ; b=42
   0x0804845f <+31>: pushl  -0x10(%ebp)           ; push b
   0x08048462 <+34>: pushl  -0xc(%ebp)            ; push a
   0x08048465 <+37>: call   0x804842b <mul>
   0x0804846a <+42>: add    $0x8,%esp             ; Parameter vom Stack
   0x0804846d <+45>: mov    %eax,-0x14(%ebp)      ; erg=mul(a,b);
   0x08048470 <+48>: sub    $0x8,%esp
   0x08048473 <+51>: pushl  -0x14(%ebp)           ; push erg
   0x08048476 <+54>: push   $0x8048520            ; push &"Ergebnis: %d\n"
   0x0804847b <+59>: call   0x80482f0 <printf@plt>
   0x08048480 <+64>: add    $0x10,%esp            ; Parameter vom Stack
   0x08048483 <+67>: sub    $0xc,%esp
   0x08048486 <+70>: push   $0x0                  ; push 0
   0x08048488 <+72>: call   0x8048310 <exit@plt>  ; exit(0)
(gdb) disas mul
   0x0804842b <+0>:  push   %ebp                  ; akt. Stackframe sichern
   0x0804842c <+1>:  mov    %esp,%ebp             ; neuer Stackframe
   0x0804842e <+3>:  sub    $0x10,%esp            ; Platz fuer lokale Variablen
   0x08048431 <+6>:  mov    0x8(%ebp),%eax        ; eax:=a (Parameter)
   0x08048434 <+9>:  imul   0xc(%ebp),%eax        ; eax:=a*b
   0x08048438 <+13>: mov    %eax,-0x4(%ebp)       ; result=a*b
   0x0804843b <+16>: mov    -0x4(%ebp),%eax       ; Resultat in eax
   0x0804843e <+19>: leave                        ; esp:=ebp; pop ebp
   0x0804843f <+20>: ret                          ; pop eip
```

Funktionsaufrufe

Parameter- und Ergebniskommunikation

- Funktionsparameter per Stack übergeben (Parameterliste wird von rechts nach links auf den Stack ‘gepusht’)
- Parameter-Bereich wird auf Vielfaches von 16 Byte aufgefüllt
- Resultat einer Funktion in Register EAX geliefert
- rufende Instanz räumt Parameter nach Rückkehr vom Stack (Caller Cleanup)
- $\rightsquigarrow$ CDECL-Konvention genutzt

Struktur einer Funktion

1. Prolog:
   - aktuellen Framepointer auf Stack sichern (`push %ebp`)
   - neuen Framepointer setzen (`mov %esp,%ebp`)
   - Platz für lokale Variablen schaffen (`sub` imm16`,%esp`)
2. Funktionskörper
3. Epilog
   - aktuellen Frame zerstören (`mov %ebp,%esp`)
   - gesicherten Framepointer restaurieren (`pop %ebp`)
   - beides zusammen ist die `leave`-Instruktion
   - Fortsetzung an auf Funktionsaufruf folgender Adresse (pop eip aka `ret`)

Bibliotheksfunktionen

Beispiel `printf()`

Listing: printf.c

```
int main(void)
{
  int c;
  c = printf("Huhu: %d %ld %lld %s!\n", 42, 41L, (1LL<<40), "zweiundvierzig");
  return c;
}
```

Listing: Zugehöriges 32-Bit-Assemblat (Ausschnitt)

```
 0x565561ab <+18>:  call   0x565561eb <__x86.get_pc_thunk.ax>
 0x565561b0 <+23>:  add    $0x2e50,%eax
 0x565561b5 <+28>:  sub    $0x8,%esp
 0x565561b8 <+31>:  lea    -0x1ff8(%eax),%edx
 0x565561be <+37>:  push   %edx
 0x565561bf <+38>:  push   $0x100
 0x565561c4 <+43>:  push   $0x0
 0x565561c6 <+45>:  push   $0x29
 0x565561c8 <+47>:  push   $0x2a
 0x565561ca <+49>:  lea    -0x1fe9(%eax),%edx
 0x565561d0 <+55>:  push   %edx
 0x565561d1 <+56>:  mov    %eax,%ebx
 0x565561d3 <+58>:  call   0x56556030 <printf@plt>
 0x565561d8 <+63>:  add    $0x20,%esp
 0x565561db <+66>:  mov    %eax,-0xc(%ebp)
 0x565561de <+69>:  mov    -0xc(%ebp),%eax
```

Bibliotheksfunktionen

Erkenntnisse:

- Parameter werden über den Stack an Bibliotheksfunktionen übergeben (wie bei „eigenen“ Funktionen)
- Übergabe der Parameter in umgekehrter Reihenfolge (beginnend beim *letzten*)
- Zeichenketten als Referenzen übergeben
- 64-Bit-Werte (`long long`) werden als 2 separate 32-Bit-Worte übergeben
- Aufruf der Funktion ist einfache `call`-Instruktion
- Resultatwert im Register EAX zurückgeliefert

64-Bit-Binaries

Unterschiede zum 32-Bit-Mode

- laaange Adressen
- Register nun 64 Bit

(Viele) neue Instruktionen:

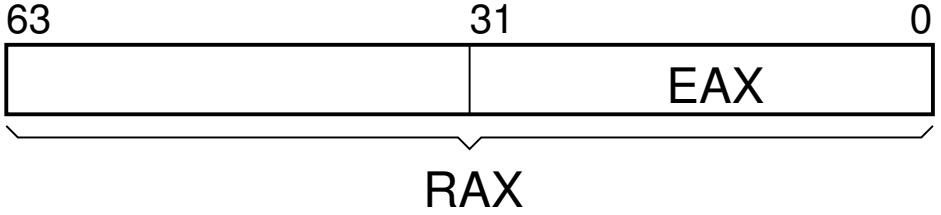
- `movabs` – `mov` mit 64-Bit-Operand
- `cltq` – *convert long to quad* = Vorzeichenerweiterung von EAX in RAX

Systemcall Convention:

- Nummer des Syscalls in RAX (differieren zur IA-32!)
- Argumente in RDI, RSI, RDX, RCX, R8, R9 (in dieser Reihenfolge)
- Kerneleintritt mit `syscall`-Instruktion
- Resultat in RAX

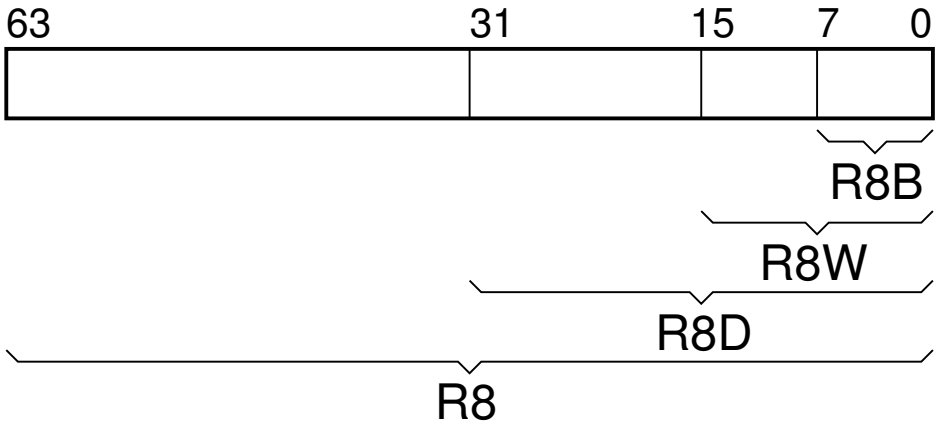
Registermodell im 64-Bit-Modus

1. Erweiterung der Universalregister von 32 → 64 Bit



(analog EBX→RBX, ECX→RCX, EDX→RDX, ESI→RSI, EDI→RDI, ESP→RSP, EBP→RBP)

2. 8 zusätzliche 64-Bit-Universalregister R8 . . . R15



3. Spezialregister 32 → 64 Bit
   - Instruction Pointer EIP → RIP
   - Flag Register EFLAGS → RFLAGS

hello64.s

```
# hello64.s
# "Hello, world!" as x86_64 assembler program
# gcc -no-pie -nostdlib -o hello64 hello64.s

  .text
  .global _start

_start:
  movq $1,%rax
  movq $1,%rdi
  movq $msg,%rsi
  movq $0xe,%rdx
  syscall
  movq $60,%rax
  movq $42,%rdi
  syscall

  .data
msg:
  .ascii "Hello, world!\n\00"
```

Schleifen

`array.c` als 64-Bit-Programm

```
0x0000000000001135 <+0>: push   %rbp
0x0000000000001136 <+1>: mov    %rsp,%rbp
0x0000000000001139 <+4>: sub    $0x20,%rsp
0x000000000000113d <+8>: movabs $0x5f5a4a525a4e4531,%rax
0x0000000000001147 <+18>:  mov    %rax,-0xe(%rbp)
0x000000000000114b <+22>:  movw   $0x42,-0x6(%rbp)
0x0000000000001151 <+28>:  movabs $0x3b287a25293d2461,%rax
0x000000000000115b <+38>:  mov    %rax,-0x18(%rbp)
0x000000000000115f <+42>:  movw   $0x63,-0x10(%rbp)
0x0000000000001165 <+48>:  movl   $0x0,-0x4(%rbp)
0x000000000000116c <+55>:  jmp    0x1191 <main+92>
0x000000000000116e <+57>:  mov    -0x4(%rbp),%eax
0x0000000000001171 <+60>:  cltq
0x0000000000001173 <+62>:  movzbl -0xe(%rbp,%rax,1),%edx
0x0000000000001178 <+67>:  mov    -0x4(%rbp),%eax
0x000000000000117b <+70>:  cltq
0x000000000000117d <+72>:  movzbl -0x18(%rbp,%rax,1),%eax
0x0000000000001182 <+77>:  xor    %eax,%edx
0x0000000000001184 <+79>:  mov    -0x4(%rbp),%eax
0x0000000000001187 <+82>:  cltq
0x0000000000001189 <+84>:  mov    %dl,-0xe(%rbp,%rax,1)
0x000000000000118d <+88>:  addl   $0x1,-0x4(%rbp)
0x0000000000001191 <+92>:  cmpl   $0x9,-0x4(%rbp)
0x0000000000001195 <+96>:  jle    0x116e <main+57>
0x0000000000001197 <+98>:  lea    -0xe(%rbp),%rax
0x000000000000119b <+102>: mov    %rax,%rdi
0x000000000000119e <+105>: call   0x1030 <puts@plt>
0x00000000000011a3 <+110>: mov    $0x17,%eax
0x00000000000011a8 <+115>: leave
```

Obfuscation

- = Techniken, die das Reverse Engineering künstlich erschweren
- z. B. Malware, Schutz geistigen Eigentums, **DRM**
- datenbasiert vs. steuerungsbasiert
- Quellcode oder Maschinencode
- vgl. International Obfuscated C Code Contest (IOCCC, https://www.ioccc.org/)

Listing: „Best One Liner“ 1987

```
main() { printf(&unix["\021%six\012\0"],(unix)["have"]+"fun"-0x60);}
```

Obfuscation auf Hochsprachen-Ebene

https://picheta.me/obfuscator

```
int main(int argc, char *argv[])
{
  int c;

  for(c=0; c<23; c++) {
    printf("%d\n", c);
  }
  exit(0);
}
```

```
int main(int o_d766f30742ab7352b8831a949b9ed64f,char *
     o_1c1547fba9bddeb4aba9dfd604216317[]){int
     o_eda61ef6e79df0dfd5da5c6519e7bc90;for (o_eda61ef6e79df0dfd5da5c6519e7bc90
     = (0x0000000000000000 + 0x0000000000000200 + 0x0000000000000800 - 0
     x0000000000000A00);(o_eda61ef6e79df0dfd5da5c6519e7bc90 < (0
     x000000000000002E + 0x0000000000000217 + 0x0000000000000817 - 0
     x0000000000000A45)) & !!(o_eda61ef6e79df0dfd5da5c6519e7bc90 < (0
     x000000000000002E + 0x0000000000000217 + 0x0000000000000817 - 0
     x0000000000000A45));o_eda61ef6e79df0dfd5da5c6519e7bc90++){
     o_d673411ad95df1898ce339f8f5da3eb3("\x25""d\012",
     o_eda61ef6e79df0dfd5da5c6519e7bc90);};o_88a74c468add110255c6a4c0257cc290((0
     x0000000000000000 + 0x0000000000000200 + 0x0000000000000800 - 0
     x0000000000000A00));};
```

Constant Unfolding

```
push $0xf9cbe47a
addl $0x6341b86, (%esp)
popl %ebx
```

- äquivalent zu `movl $0,%ebx`
- unnötige Berechnungen
- Umkehroperation zur Compileroptimierung *Constant Folding*

Dead Code Insertion

```
movl $0x24,%esi
xor %ebx,%ebx
movl $0x02,%eax
addl $0x12,%esi
addl $0x01,%eax
movl %esi,%ecx
movl %eax,%ebx
xorl %ebx,%esi
incl %eax
movl %eax,%edx
addl $-3,%ebx
```

- Code, dessen Ergebnis nirgends verwendet wird

Ausgangsmuster: `push`-Operation mit Register

```
pushl reg32
```

Ersetzung mit einem der folgenden Muster:

Listing: Muster #1

```
pushl imm32
movl reg32,(%esp)
```

Listing: Muster #2

```
lea (%esp-4),%esp
movl reg32,(%esp)
```

Listing: Muster #3

```
sub 4,%esp
movl reg32,(%esp)
```

Listing: Muster #4

```
add -4,%esp
movl reg32,(%esp)
```

TODO

- Never try to analyze everything!
- Üben, üben, üben!
- komplexeren Code analysieren: Division u. a.
- IDAPro, Radare2 erlernen
- Capture-the-Flag, Crackmes einbeziehen
  `https://ctflearn.com` o. ä.
- Übergang zu ‘praxisnäherem’ Schadcode
- kurze Auswertung (Interesse, Schwierigkeit, Verständlichkeit)
- bei Interesse: komplexere Aufgabenstellung (Praktikum, Abschlussarbeit) in diesem Kontext möglich und nützlich