Malware-Analyse und Reverse Engineering

2: Compiler und Dateiformate

23.3.2017

Prof. Dr. Michael Engel

Überblick (heute)

Themen:

- Übersetzung von C/C++-Programmen und Binärformate
  - Compiliervorgang, Linken, Statische und dynamische Libraries
  - Werkzeuge zur Analyse von Binärdateien

Übersetzung eines (C-)Programms

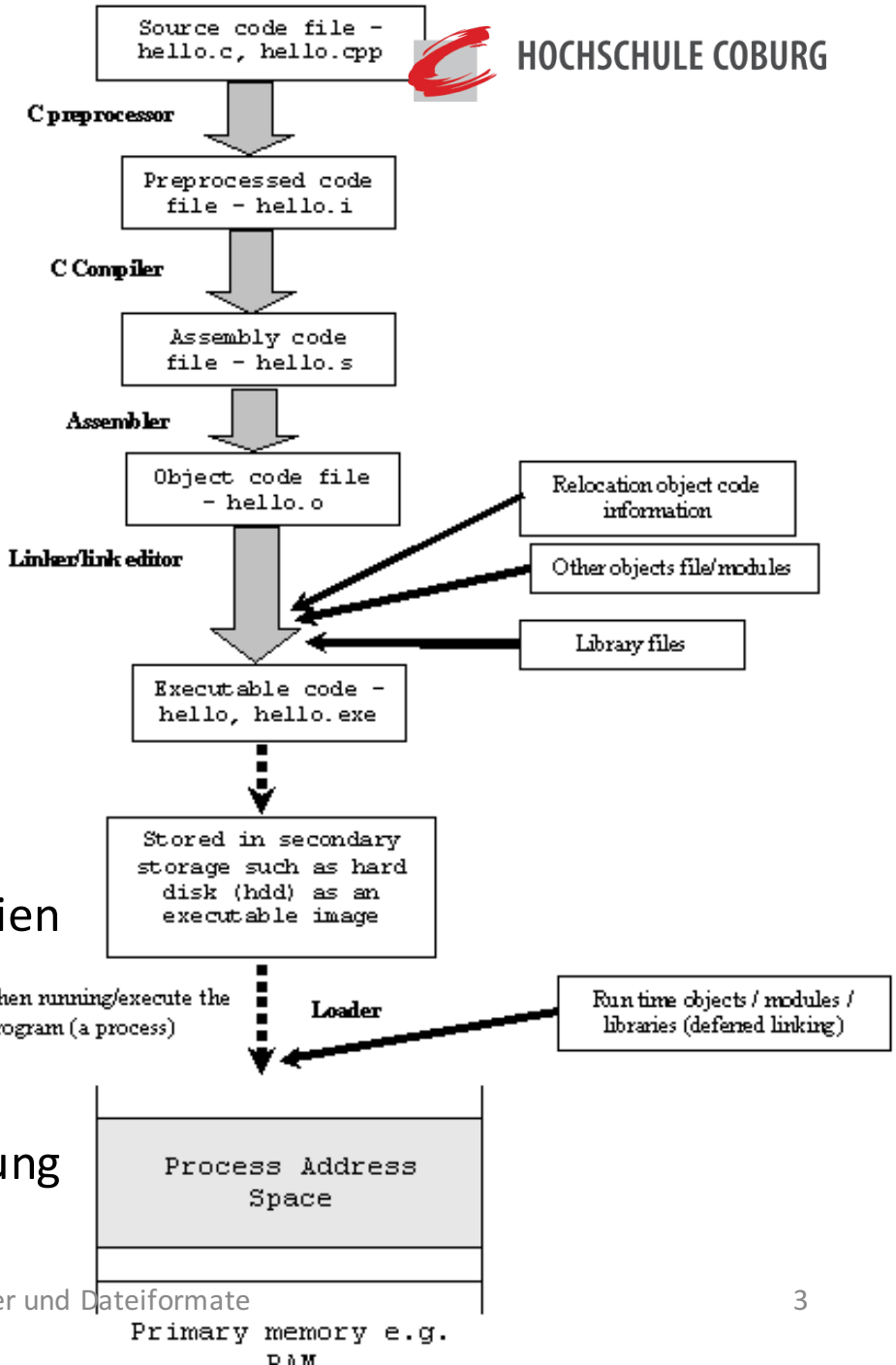
Präprozessor
- Expandiert #include und Makros

Compiler
- Erzeugt Assembler-Quellcode aus C-Sourcecode

Assembler
- Erzeugt Objektcode aus Assembler-Quellcode

Linker
- Fügt (ein oder) mehrere Objektdateien (+ evtl. Libraries) zu ausführbarer Datei (oder Library) zusammen

Loader
- Lädt ausführbare Datei zur Ausführung in Hauptspeicher



Beispiel: Von .c nach .o

```
#include <stdio.h>
static void display(int i, int *ptr);

int main(void)
{
      int x = 5;
      int *xptr = &x;
      printf("In main() program:\n");
      printf("x value is %d and is stored at address %p.\n", x, &x);
      printf("xptr pointer points to address %p which holds a value of %d.\n",
           xptr, *xptr);
      display(x, xptr);
      return 0;
}

void display(int y, int *yptr)
{
      char var[7] = "ABCDEF";
      printf("In display() function:\n");
      printf("y value is %d and is stored at address %p.\n", y, &y);
      printf("yptr pointer points to address %p which holds a value of %d.\n",
           yptr, *yptr);
}
```

Beispiel: Von .c nach .o (2)

```
$ gcc -c testprog1.c
```

Compileraufruf: Übersetzt Source .c nach Objektdatei .o

```
$ ls -l
total 8
-rw-r--r-- 1 me me  629 Mar 22 13:15 testprog1.c
-rw-r--r-- 1 me me 1412 Mar 22 13:16 testprog1.o
```

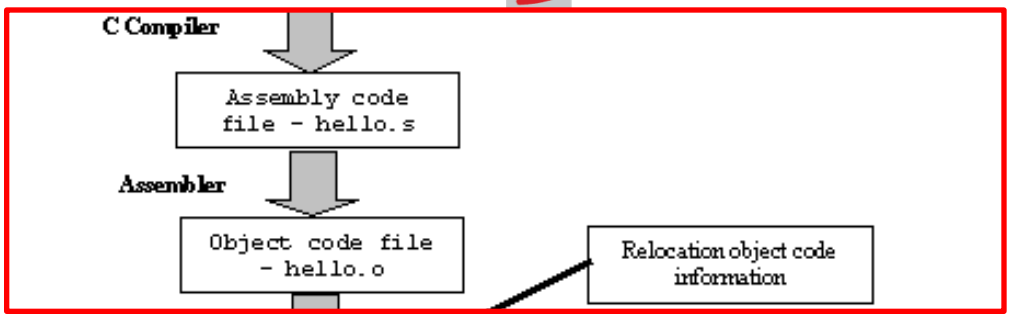
Was steckt in der Objektdatei?

```
$ file testprog1.o
testprog1.o: ELF 32-bit LSB relocatable, intel 80386, version 1 (SYSV), not stripped
```

Inhalt des .o als Hex und ASCII anzeigen

```
$ hexdump -C testprog1.o
00000000  7f 45 4c 46 01 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010  01 00 03 00 01 00 00 00  00 00 00 00 00 00 00 00  |................|
00000020  84 02 00 00 00 00 00 00  34 00 00 00 00 00 28 00  |........4.....(.|
00000030  0b 00 08 00 8d 4c 24 04  83 e4 f0 ff 71 fc 55 89  |.....L$.....q.U.|
[…usw]
```

Mooooment mal!



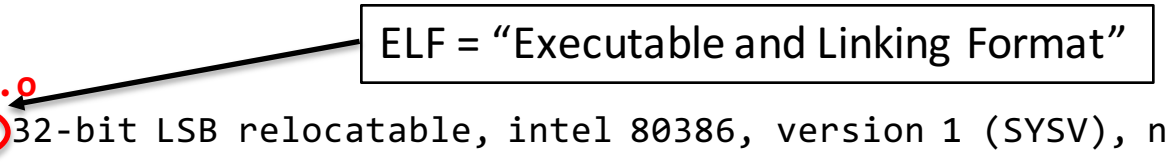
Wo ist unser Assemblerquellcode?

- Wird nur als Zwischendatei temporär erzeugt
- Option „**-S**“ beim Compileraufruf erzeugt .s-Datei explizit:

```
$ gcc -S testprog1.c
$ ls -1
testprog1.c
testprog1.o
testprog1.s
```

```
        .file   "testprog1.c"
        .section        .rodata
.LC0:
        .string "In main() program:"
        .align 4
.LC1:
        .string "x value is %d and is stored at address %p.\n"
        .align 4
.LC2:
        .string "xptr pointer points to address %p which holds a value of %d.\n"
        .text
.globl main
        .type   main, @function
main:
        leal    4(%esp), %ecx
        andl    $-16, %esp
        pushl   -4(%ecx)
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %ecx
        subl    $36, %esp
        movl    $5, -12(%ebp)
        leal    -12(%ebp), %eax
```

ELF-Interna (1)




```
$ file testprog1.o
testprog1.o: ELF 32-bit LSB relocatable, intel 80386, version 1 (SYSV), not stripped

$ man elf
[...]
The ELF header is described by the type Elf32_Ehdr or Elf64_Ehdr:
#define EI_NIDENT 16
typedef struct {
    unsigned char e_ident[EI_NIDENT];
    uint16_t      e_type;
    uint16_t      e_machine;
    uint32_t      e_version;
    ElfN_Addr     e_entry;
    ElfN_Off      e_phoff;
    ElfN_Off      e_shoff;
    uint32_t      e_flags;
    uint16_t      e_ehsize;
    uint16_t      e_phentsize;
    uint16_t      e_phnum;
    uint16_t      e_shentsize;
    uint16_t      e_shnum;
    uint16_t      e_shstrndx;
} ElfN_Ehdr;
```

ELF-Interna (2)

```
$ hexdump -C testprog1.o
00000000  7f 45 4c 46 01 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010  01 00 03 00 01 00 00 00  00 00 00 00 00 00 00 00  |................|
00000020  84 02 00 00 00 00 00 00  34 00 00 00 00 00 28 00  |........4.....(.|
00000030  0b 00 08 00 8d 4c 24 04  83 e4 f0 ff 71 fc 55 89  |.....L$.....q.U.|
[…usw]
```

```
00000000  7f 45 4c 46 ; ELF „Magic“
00000004  01
00000005  01
00000006  01
00000007  00  00 00 00 00 00 00 00 00
```

/usr/include/elf.h:

```
#define ELFMAG          "\177ELF"
#define EI_CLASS        4       /* File class byte index */
#define ELFCLASSNONE    0       /* Invalid class */
#define ELFCLASS32      1       /* 32-bit objects */
#define ELFCLASS64      2       /* 64-bit objects */
#define ELFCLASSNUM     3
```

Achtung:
\177 ist *oktal!*

```
#define EI_NIDENT 16
typedef struct {
    unsigned char e_ident[EI_NIDENT];
    uint16_t      e_type;
    uint16_t      e_machine;
    uint32_t      e_version;
    ElfN_Addr     e_entry;
    ElfN_Off      e_phoff;
    ElfN_Off      e_shoff;
    uint32_t      e_flags;
    uint16_t      e_ehsize;
    uint16_t      e_phentsize;
    uint16_t      e_phnum;
    uint16_t      e_shentsize;
    uint16_t      e_shnum;
    uint16_t      e_shstrndx;
} ElfN_Ehdr;
```

ELF-Interna (3)

```
00000000  7f 45 4c 46 ; ELF „Magic“
00000004  01
00000005  01
00000006  01
00000007  00
00000008  00 00 00 00 00 00 00 00
```

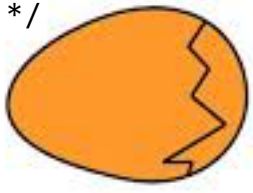
/usr/include/elf.h:

```
#define ELFMAG          "\177ELF"
#define EI_CLASS        4    /* File class byte index */
#define ELFCLASSNONE    0    /* Invalid class */
#define ELFCLASS32      1    /* 32-bit objects */
#define ELFCLASS64      2    /* 64-bit objects */
#define ELFCLASSNUM     3

#define EI_DATA         5    /* Data encoding byte index */
#define ELFDATANONE     0    /* Invalid data encoding */
#define ELFDATA2LSB     1    /* 2's complement, little endian */
#define ELFDATA2MSB     2    /* 2's complement, big endian */
```

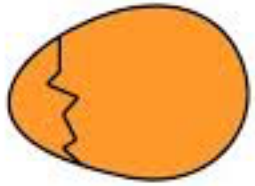
Exkurs: Endianness

/usr/include/elf.h:

```
#define EI_DATA         5    /* Data encoding byte index */
#define ELFDATANONE     0    /* Invalid data encoding */
#define ELFDATA2LSB     1    /* 2's complement, little endian */
#define ELFDATA2MSB     2    /* 2's complement, big endian */
```



BIG ENDIAN - The way people always broke their eggs in the Lilliput land



LITTLE ENDIAN - The way the king then ordered the people to break their eggs

Endianness oder Byte-Reihenfolge (byte order)

- Reihenfolge, in der die Bytes eines Datentyps mit > 8 Bit Länge im Speicher abgelegt werden
- Beispiel:
  Dezimalzahl 123456789 = Hexadezimal **0x075BCD15**

Little Endian: Least significant byte zuerst (an niedrigster Adresse)

00000000 **15**
00000001 **CD**
00000002 **5B**
00000003 **07**

⇔

Big Endian: Most significant byte zuerst:

00000000 **07**
00000001 **5B**
00000002 **CD**
00000003 **15**

Byte-Pfriemeleien...

Das ganze Semester Hex-Dumps anschauen? Igitt!

- Nein – es gibt Werkzeuge, die das Leben erleichtern

- Tools für Binärformat-Erzeugung und –Analyse
  - z.B. GNU binutils
- Tools und Libraries zum Arbeiten mit Objektdateien
  - readelf, libelf
- Disassembler
  - GNU objdump, IDA Pro, OllyDbg
- Decompiler und statische Analysatoren

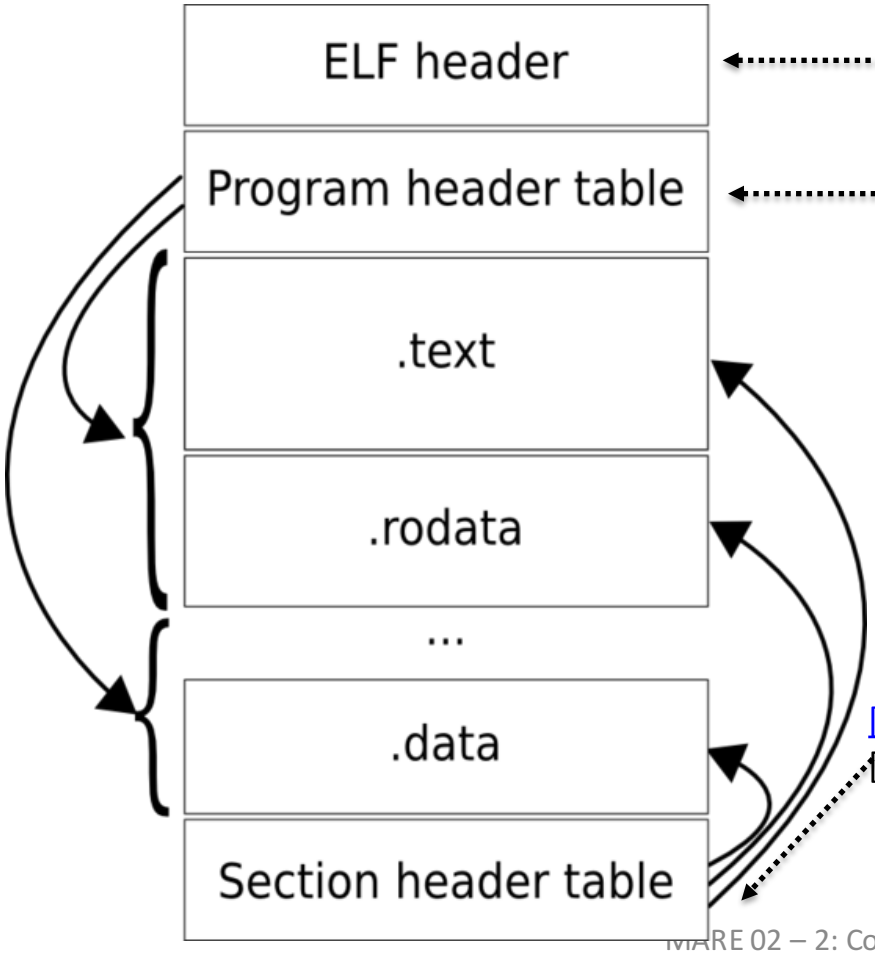
Beispiel: readelf

```
$ readelf testprog1.o
Usage: readelf <option(s)> elf-file(s)
 Display information about the contents of ELF format files
 Options are:
  -a --all              Equivalent to: -h -l -S -s -r -d -V -A -I
  -h --file-header      Display the ELF file header
[...]
```

```
$ readelf -h testprog1.o
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              REL (Relocatable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x0
  Start of program headers:          0 (bytes into file)
  Start of section headers:          644 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           0 (bytes)
  Number of program headers:         0
  Size of section headers:           40 (bytes)
  Number of section headers:         11
  Section header string table index: 8
```

Struktur von ELF-Objektdateien

ELF-Dateisystemformat standardisiert in [1] und [2]



Header mit allgemeinen Informationen (siehe vorherige Folien)

Information über enthaltene Segmente

Tabelle der Sektionsheader, beschreibt einzelne Segmente

[1] *System V Application Binary Interface* Edition 4.1 (1997-03-18)
[2] Tool Interface Standard (TIS)
Executable and Linking Format (ELF) Specification Version 1.2 (May 1995)

ELF-Sektionen

```
$ readelf -S testprog1.o
There are 11 section headers, starting at offset 0x284:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        00000000 000034 0000db 00  AX  0   0  4
  [ 2] .rel.text         REL             00000000 000524 000060 08      9   1  4
  [ 3] .data             PROGBITS        00000000 000110 000000 00  WA  0   0  4
  [ 4] .bss              NOBITS          00000000 000110 000000 00  WA  0   0  4
  [ 5] .rodata           PROGBITS        00000000 000110 000102 00   A  0   0  4
  [ 6] .comment          PROGBITS        00000000 000212 00001f 00      0   0  1
  [ 7] .note.GNU-stack   PROGBITS        00000000 000231 000000 00      0   0  1
  [ 8] .shstrtab         STRTAB          00000000 000231 000051 00      0   0  1
  [ 9] .symtab           SYMTAB          00000000 00043c 0000c0 10     10   9  4
  [10] .strtab           STRTAB          00000000 0004fc 000026 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
```

ELF-Sektionen (2)

Unterteilung des Binärcodes in einzelne Abschnitte

Die wichtigsten Arten von Sektionen:

Sektion	Funktion
Text (.text)	Maschinencode (Instruktionen) und Einsprungadresse
Nur-lese-Daten (.rodata)	Vorinitialisierte Konstanten
Lese-/Schreib-Daten (.rwdata)	Vorinitialisierte Variable
Base Storage Segment (.bss)	Uninitialisierte Daten
Symbole (.symtab)	Adressen zu symbolischen Namen

ELF-Sektionen (3)

Was kommt wohin in der Objektdatei?

$ **gcc –c foo.c**
$ **readelf –S foo.o**

```
There are 11 section headers, starting at offset 0xd8:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        00000000 000034 00002c 00  AX  0   0  4
  [ 2] .rel.text         REL             00000000 000364 000020 08      9   1  4
  [ 3] .data             PROGBITS        00000000 000060 000004 00  WA  0   0  4
  [ 4] .bss              NOBITS          00000000 000064 000000 00  WA  0   0  4
  [ 5] .rodata           PROGBITS        00000000 000064 000004 00   A  0   0  4
```

Beispielprogramm foo.c:

```
const int a = 42;
int b = 23;
int c;

int main(void) {
  c = a + b;
  return c;
}
```

Sektion	Funktion
Text (.text)	Maschinencode (Instruktionen) und Einsprungadresse
Nur-lese-Daten (.rodata)	Vorinitialisierte Konstanten
Lese-/Schreib-Daten (.data)	Vorinitialisierte Variable
Base Storage Segment (.bss)	Uninitialisierte Daten
Symbole (.symtab)	Adressen zu symbolischen Namen

Jeweils eine Variable zu 4 Byte = sizeof(int) in .data und .rodata

ELF-Sektionen (4)

Offsets in ELF .o-Datei
Keine Speicheradressen!

```
There are 11 section headers, starting at offset 0xd8:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        00000000 000034 00002c 00  AX  0   0  4
  [ 2] .rel.text         REL             00000000 000364 000020 08      9   1  4
  [ 3] .data             PROGBITS        00000000 000060 000004 00  WA  0   0  4
  [ 4] .bss              NOBITS          00000000 000064 000000 00  WA  0   0  4
  [ 5] .rodata           PROGBITS        00000000 000064 000004 00   A  0   0  4
```

```
00000000  7f 45 4c 46 01 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010  01 00 03 00 01 00 00 00  00 00 00 00 00 00 00 00  |................|
00000020  d8 00 00 00 00 00 00 00  34 00 00 00 00 00 28 00  |........4.....(.|
00000030  0b 00 08 00 8d 4c 24 04  83 e4 f0 ff 71 fc 55 89  |.....L$.....q.U.|
00000040  e5 51 8b 15 00 00 00 00  a1 00 00 00 00 8d 04 02  |.Q..............|
00000050  a3 00 00 00 00 a1 00 00  00 00 59 5d 8d 61 fc c3  |..........Y].a..|
00000060  17 00 00 00 2a 00 00 00  00 47 43 43 3a 20 28 44  |....*....GCC: (D|
00000070  65 62 69 61 6e 20 34 2e  33 2e 32 2d 31 2e 31 29  |ebian 4.3.2-1.1)|
```

An Adresse:

0x60: 17 00 00 00 => 0x00000017 = 23_{10}

0x64: 2a 00 00 00 => 0x0000002a = 42_{10}

ELF-Sektionen (5)

Beispielprogramm foo.c:

```
const int a = 42;
int b = 23;
int c;

int main(void) {
  c = a + b;
  return c;
}
```

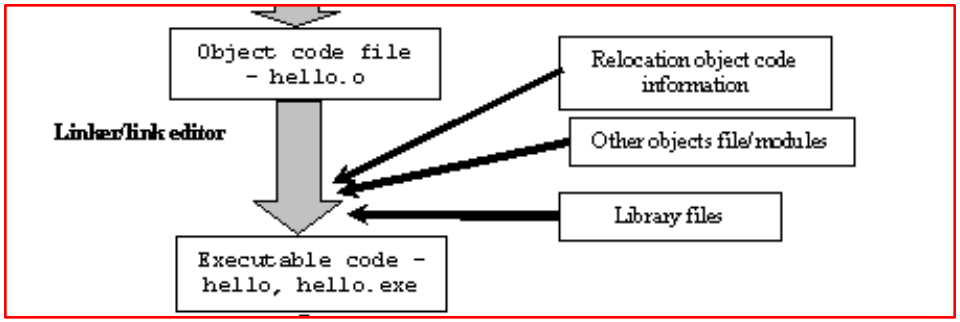
Im Assemblercode:

$ **gcc –S foo.c**
$ **cat foo.s**

```
    .file   "foo.c"
.globl a
    .section .rodata
    .align 4
    .type   a, @object
    .size   a, 4
a:
    .long   42
.globl b
    .data
    .align 4
    .type   b, @object
    .size   b, 4
b:
    .long   23
```

```
    .text
.globl main
    .type   main, @function
main:
    leal    4(%esp), %ecx
    andl    $-16, %esp
    pushl   -4(%ecx)
    pushl   %ebp
    movl    %esp, %ebp
    pushl   %ecx
    movl    a, %edx
    movl    b, %eax
    leal    (%edx,%eax), %eax
    movl    %eax, c
    movl    c, %eax
    popl    %ecx
    popl    %ebp
    leal    -4(%ecx), %esp
    ret
    .size   main, .-main
    .comm   c,4,4
    .ident  "GCC: (Debian 4.3.2-1.1) 4.3.2"
    .section .note.GNU-stack,"",@progbits
```

Objektdatei – und dann?



.o-Objektdateien können nicht direkt ausgeführt werden!

- Wichtige Komponenten eines ausführbaren Programms fehlen
  - crt0 – Startup
  - Initialisierung
    - Variablen in .bss werden mit „0“ initialisiert)
    - Für C++: Aufruf von Konstruktoren
  - Sprung zu „main“-Funktion und Parameterübergabe (argc, argv, envp)
  - Libraries, z.B. libc (C-Standardfunktionen), müssen hinzugefügt werden

- **Adressen von Variablen und Funktionen sind nicht aufgelöst**
  - Eine der Hauptaufgaben des Linkers

Symbole und Adressen in .o-Dateien

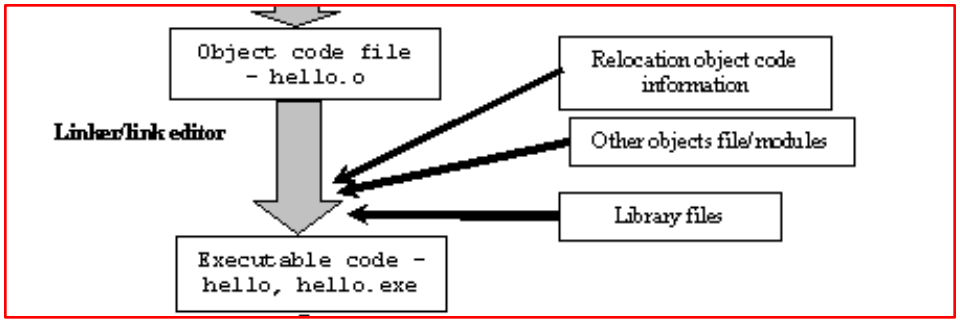
Sektion	Funktion
Symbole (.symtab)	Adressen zu symbolischen Namen

Adressen von Funktionen und (globalen) Variablen müssen bekannt sein

- Symboltabelle: ordnet symbolischen Namen (Funktions-/Variablennamen) Adressen zu:
- In .o-Datei sind Adressen auf „0“ gesetzt

```
$ readelf -s foo.o

Symbol table '.symtab' contains 12 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00000000     0 FILE    LOCAL  DEFAULT  ABS foo.c
     2: 00000000     0 SECTION LOCAL  DEFAULT    1
     3: 00000000     0 SECTION LOCAL  DEFAULT    3
     4: 00000000     0 SECTION LOCAL  DEFAULT    4
     5: 00000000     0 SECTION LOCAL  DEFAULT    5
     6: 00000000     0 SECTION LOCAL  DEFAULT    7
     7: 00000000     0 SECTION LOCAL  DEFAULT    6
     8: 00000000     4 OBJECT  GLOBAL DEFAULT    5 a
     9: 00000000     4 OBJECT  GLOBAL DEFAULT    3 b
    10: 00000000    44 FUNC    GLOBAL DEFAULT    1 main
    11: 00000004     4 OBJECT  GLOBAL DEFAULT  COM c
```

Linker und Relokationen



Linker erhält Liste aller zum Bauen eines Programms erforderlichen Objektdateien

- .o-Dateien (von Compiler oder Assembler erzeugt): Objektdateien
  - Können auch von C++, Fortran, ... stammen!
- .a-Dateien: „Archive“ von .o-Dateien
  - Statische Libraries
- .so-Dateien: „shared object“-Dateien (nächste Woche mehr dazu)
  - Dynamische Libraries (Windows: DLL „Dynamic Loadable Library“)

Linker ordnet Text- und Datensegmente der einzelnen .o-Dateien in Adressraum zum späteren Laden durch das Betriebssystem

- Dabei werden Referenzen auf Symbole (Variablen, Funktionen) aufgelöst
- Angeleitet durch Linkerskript (Konfigurationsdatei)

Linker und Symbolauflösung (1)

Sektion	Funktion
Symbole (.symtab)	Adressen zu symbolischen Namen

Symboltabelle des gelinkten Programms

- Enthält auch die Symbole aller dazu gelinkten Libraries – unübersichtlich…

```
$ gcc –o foo foo.o

$ readelf -s foo

Symbol table '.symtab' contains 76 entries:
Num:    Value  Size Type    Bind   Vis      Ndx Name
[…]
 64: 08048460     4 OBJECT  GLOBAL DEFAULT   15 a
[…]
 69: 0804956c     4 OBJECT  GLOBAL DEFAULT   23 b
[…]
 73: 08048374    44 FUNC    GLOBAL DEFAULT   13 main
 74: 08048254     0 FUNC    GLOBAL DEFAULT   11 _init
 75: 08049578     4 OBJECT  GLOBAL DEFAULT   24 c
```

“gcc” ist nicht der Compiler an sich, sondern ein Frontend, das bei Bedarf Präprozessor, Compiler, Linker… aufruft. Hier wird foo.o (+Libraries) zum ausführbaren Programm foo gelinkt!

Linker und Symbolauflösung (2)

HOCHSCHULE COBURG

Sektion	Funktion
Text (.text)	Maschinencode (Instruktionen) und Einsprungadresse
Nur-lese-Daten (.rodata)	Vorinitialisierte Konstanten
Lese-/Schreib-Daten (.data)	Vorinitialisierte Variable
Base Storage Segment (.bss)	Uninitialisierte Daten
Symbole (.symtab)	Adressen zu symbolischen Namen

Symboltabelle des gelinkten Programms

- Alternativ: „nm“ (für „names“) aus den GNU binutils

```
$ gcc -o foo foo.o
$ nm foo
[…]
08048254 T _init
080482c0 T _start
08048460 R a
0804956c D b
08049578 B c
[…]
08048374 T main
```

Abk	Sektion
T	.text
R	.rodata
D	.data
B	.bss

…zum Vergleich:

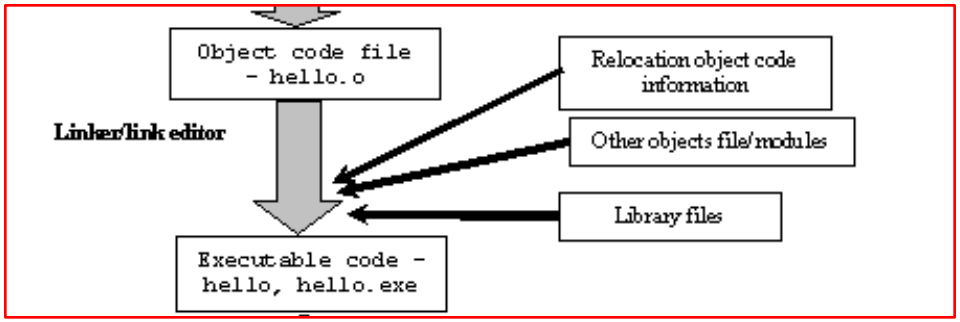
```
$ readelf -s foo
Symbol table '.symtab' contains 76 entries:
Num:    Value  Size Type    Bind   Vis      Ndx Name
[…]
 64: 08048460     4 OBJECT  GLOBAL DEFAULT   15 a
 69: 0804956c     4 OBJECT  GLOBAL DEFAULT   23 b
 73: 08048374    44 FUNC    GLOBAL DEFAULT   13 main
 74: 08048254     0 FUNC    GLOBAL DEFAULT   11 _init
 75: 08049578     4 OBJECT  GLOBAL DEFAULT   24 c
```

Linker und Relokationen: .o-Dateien

```
$ objdump -d foo.o
```

foo.o – Objektdatei

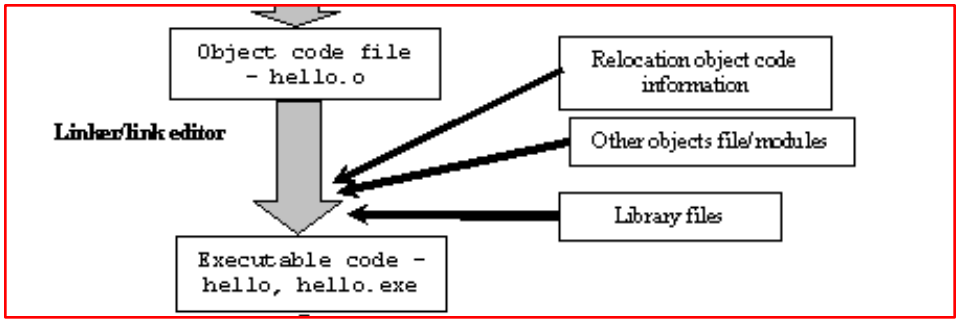
```
foo.o:     file format elf32-i386

Disassembly of section .text:

00000000 <main>:
   0:   8d 4c 24 04             lea    0x4(%esp),%ecx
   4:   83 e4 f0                and    $0xfffffff0,%esp
   7:   ff 71 fc                pushl  -0x4(%ecx)
   a:   55                      push   %ebp
   b:   89 e5                   mov    %esp,%ebp
   d:   51                      push   %ecx
   e:   8b 15 00 00 00 00       mov    0x0,%edx
  14:   a1 00 00 00 00          mov    0x0,%eax
  19:   8d 04 02                lea    (%edx,%eax,1),%eax
  1c:   a3 00 00 00 00          mov    %eax,0x0
  21:   a1 00 00 00 00          mov    0x0,%eax
  26:   59                      pop    %ecx
  27:   5d                      pop    %ebp
  28:   8d 61 fc                lea    -0x4(%ecx),%esp
  2b:   c3                      ret
```



Adressen von Variablen und Funktionen sind nicht aufgelöst (=0)!

...den Assemblercode muss man (jetzt noch) nicht verstehen...

Linker und Relokationen: Relokationstabelle



Relokationstabelle

- Enthält zu jeder in .o-Datei mit „0“ initialisierten Adresse Information über die relative Lage der Adresse in .text-Sektion (Relocation Offset) und die Länge der Adresse (R_386_32 = 32 Bit)

```
$ readelf -r foo.o

Relocation section '.rel.text' at offset 0x364 contains 4 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
00000010  00000801 R_386_32          00000000   a
00000015  00000901 R_386_32          00000000   b
0000001d  00000b01 R_386_32          00000004   c
00000022  00000b01 R_386_32          00000004   c
```

Linker und Relokationen: Auflösung

Offsets innerhalb des Text-Segments

```
Relocation section '.rel.text' at offset 0x364 contains 4 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
00000010  00000801 R_386_32          00000000   a
00000015  00000901 R_386_32          00000000   b
0000001d  00000b01 R_386_32          00000004   c
00000022  00000b01 R_386_32          00000004   c
```

```
00000000 <main>:
   0:    8d 4c 24 04             lea    0x4(%esp),%ecx
   4:    83 e4 f0                and    $0xfffffff0,%esp
   7:    ff 71 fc                pushl  -0x4(%ecx)
   a:    55                      push   %ebp
   b:    89 e5                   mov    %esp,%ebp
   d:    51                      push   %ecx
   e:    8b 15 00 00 00 00       mov    0x0,%edx
  14:    a1 00 00 00 00          mov    0x0,%eax
  19:    8d 04 02                lea    (%edx,%eax,1),%eax
  1c:    a3 00 00 00 00          mov    %eax,0x0
  21:    a1 00 00 00 00          mov    0x0,%eax
  26:    59                      pop    %ecx
  27:    5d                      pop    %ebp
  28:    8d 61 fc                lea    -0x4(%ecx),%esp
  2b:    c3                      ret
```

Aufgelöste Relokationen in Executable

```
$ objdump -d foo
```

foo – ausführbare Datei!

```
foo:     file format elf32-i386

Disassembly of section .text:

08048374 <main>:
 8048374: 8d 4c 24 04            lea    0x4(%esp),%ecx
 8048378: 83 e4 f0               and    $0xfffffff0,%esp
 804837b: ff 71 fc               pushl  -0x4(%ecx)
 804837e: 55                     push   %ebp
 804837f: 89 e5                  mov    %esp,%ebp
 8048381: 51                     push   %ecx
 8048382: 8b 15 60 84 04 08      mov    0x8048460,%edx
 8048388: a1 6c 95 04 08         mov    0x804956c,%eax
 804838d: 8d 04 02               lea    (%edx,%eax,1),%eax
 8048390: a3 78 95 04 08         mov    %eax,0x8049578
 8048395: a1 78 95 04 08         mov    0x8049578,%eax
 804839a: 59                     pop    %ecx
 804839b: 5d                     pop    %ebp
 804839c: 8d 61 fc               lea    -0x4(%ecx),%esp
 804839f: c3                     ret
```

Adressen von Variablen und Funktionen sind jetzt aufgelöst (!=0)!

Aufgelöste Relokationen vs. „nm“

```
08048374 <main>:
 8048374: 8d 4c 24 04          lea    0x4(%esp),%ecx
 8048378: 83 e4 f0             and    $0xfffffff0,%esp
 804837b: ff 71 fc             pushl  -0x4(%ecx)
 804837e: 55                   push   %ebp
 804837f: 89 e5                mov    %esp,%ebp
 8048381: 51                   push   %ecx
 8048382: 8b 15 60 84 04 08    mov    0x8048460,%edx
 8048388: a1 6c 95 04 08       mov    0x804956c,%eax
 804838d: 8d 04 02             lea    (%edx,%eax,1),%eax
 8048390: a3 78 95 04 08       mov    %eax,0x8049578
 8048395: a1 78 95 04 08       mov    0x8049578,%eax
 804839a: 59                   pop    %ecx
 804839b: 5d                   pop    %ebp
 804839c: 8d 61 fc             lea    -0x4(%ecx),%esp
 804839f: c3                   ret
```

```
; Variable a
; ...b

; und c!
```

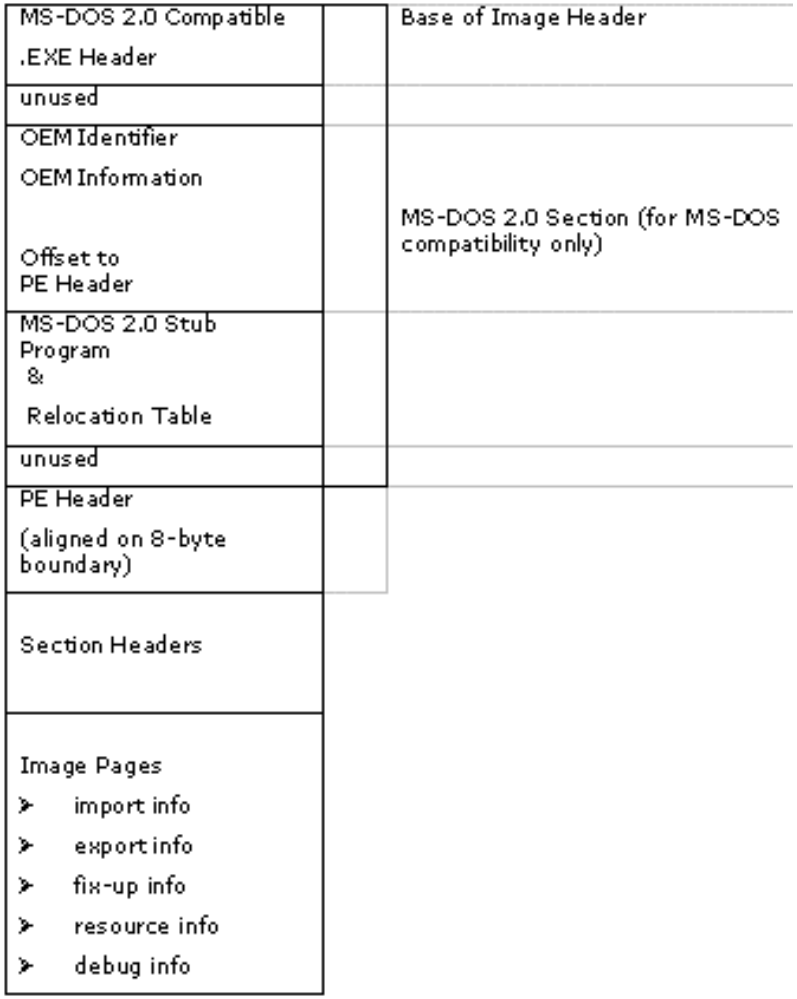
```
$ gcc -o foo foo.o
$ nm foo
[…]
08048254 T _init
080482c0 T _start
08048460 R a
0804956c D b
08049578 B c
[…]
08048374 T main
```

Zuordnung von Adressen in ausführbarer Datei zu Variablennamen mit Hilfe von “nm”:
Erstes Reverse Engineering!

Und wie sieht das bei Windows aus?

Ausführbares Format „PE“
(Portable Executable)

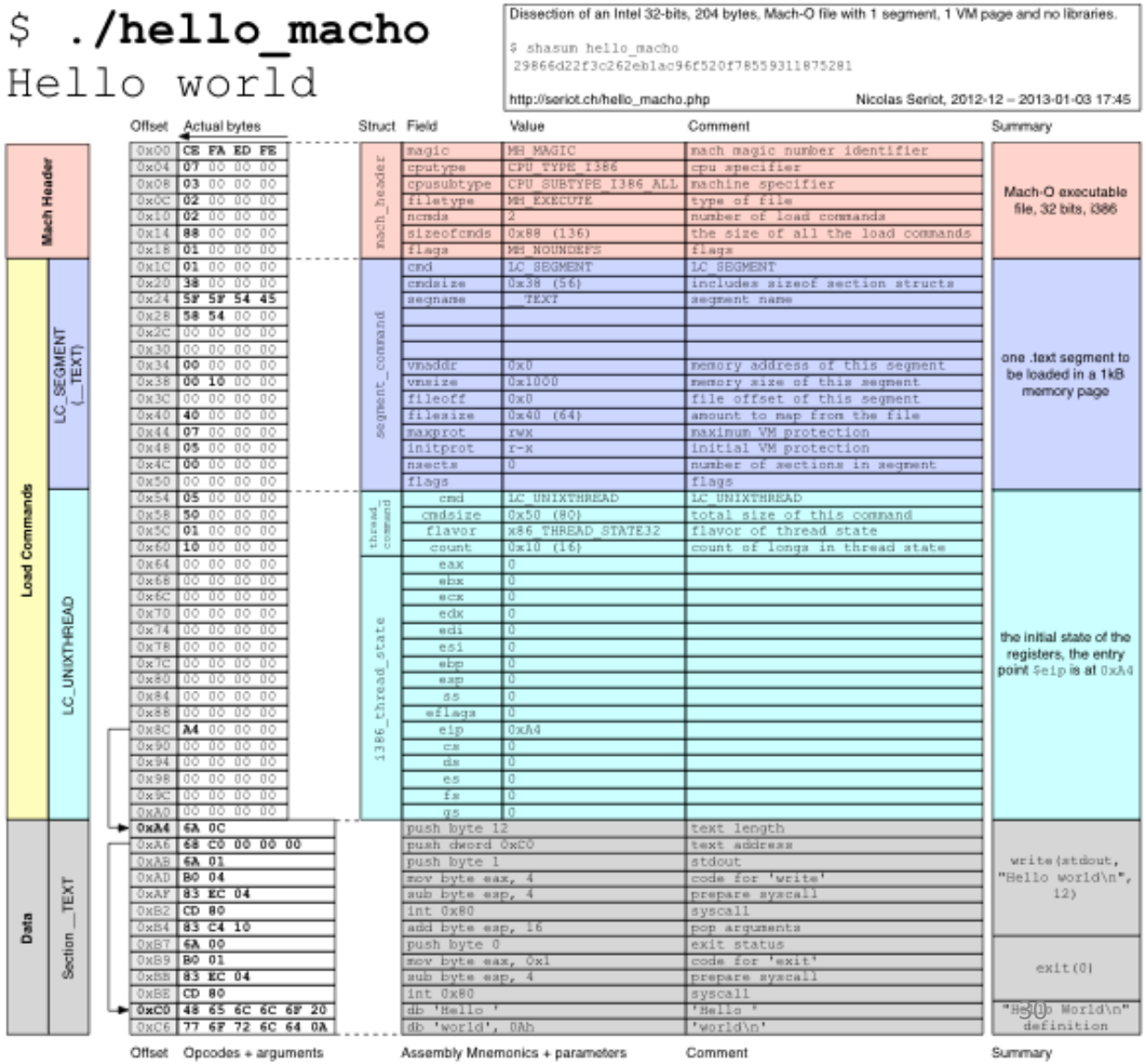
- Abgeleitet von altem Unix-Format „COFF“ (Common Object File Format)
- Für alle Fälle etwas Kompatibilität zu MS-DOS...



Und bei MacOS X?

Mach-O-Format (Mach-Object)

- Besonderheit: Multi-Architecture Binaries
- Objektcode für verschiedene Prozessorarchi-tekturen (z.B. x64, x86, ppc) in einer ausführbaren Datei



Fazit

Übersetzung und Linken

- Compiler-Toolflow
  - Viele einzelne Schritte, Zwischenergebnisse analysierbar
- Binärformate
  - Einblick in ELF: mehr als Binärcode von Instruktionen
- Aufgaben des Linkers
  - Symbolzuordnung, Startadresse, Libraries
- Rest der Welt
  - Windows, MacOS X

Literatur

ELF

- TIS Committee, „*Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification*“, Version 1.2
  Online unter https://www.uclibc.org/docs/elf.pdf

Windows PE

- Microsoft Corp., „*Microsoft Portable Executable and Common Object File Format Specification*“, Revision 6, Februar 1999
  Online unter http://www.osdever.net/documents/PECOFF.pdf
- PE-Format Poster:
  http://www.openrce.org/reference_library/files/reference/PE%20Format.pdf

Mach-O:

- Apple, Inc., „*Mach-O Runtime Architecture*“, 2004

Linker

- John R. Levine, „*Linkers and Loaders*“, MKP 1999, ISBN 1-55860-496-0
  Online (beta) unter https://www.iecc.com/linker/