CS 6V81-05: System Security and Malicious Code Analysis
Revealing Internals of Linkers

Zhiqiang Lin

Department of Computer Science
University of Texas at Dallas

March 26\textsuperscript{th}, 2012
Outline

1. Background
2. ELF Linking
3. Static Linking
4. Dynamic Linking
5. Summary
Outline

1. Background
2. ELF Linking
3. Static Linking
4. Dynamic Linking
5. Summary
In computer science, a linker or link editor is a program that takes one or more objects generated by a compiler and combines them into a single executable program.
**Definition**

In computer science, a linker or link editor is a program that takes one or more objects generated by a compiler and combines them into a single executable program.
Linker

**Definition**

In computer science, a linker or link editor is a program that takes one or more objects generated by a compiler and combines them into a single executable program.

**Various Stages**

1. Linking can be done at compile time.
2. at load time (by loaders)
3. at run time (by application programs)
Compilation vs. Linking

Compilation

Compilation refers to the processing of source code files (.c, .cc, or .cpp) and the creation of an 'object' file. This step doesn’t create anything the user can actually run. Instead, the compiler merely produces the machine language instructions that correspond to the source code file that was compiled.
Compilation vs. Linking

Compilation
Compilation refers to the processing of source code files (.c, .cc, or .cpp) and the creation of an ’object’ file. This step doesn’t create anything the user can actually run. Instead, the compiler merely produces the machine language instructions that correspond to the source code file that was compiled.

Linking
Linking refers to the creation of a single executable file from multiple object files. In this step, it is common that the linker will complain about undefined functions (commonly, main itself).
Why Linkers?

Modularity

- Program can be written as a collection of smaller source files, rather than one monolithic mass.

- Can build libraries of common functions (more on this later)
  - e.g., Math library, standard C library
A library is a collection of pre-written function calls.
A library is a collection of pre-written function calls.

Using existing libraries can save a programmer a lot of time. (e.g., the printf() in the C library)
A library is a collection of pre-written function calls.

Using existing libraries can save a programmer a lot of time. (e.g., the printf() in the C library)

However, if a library is not shared among many programs, but instead a copy is included in every program that uses it, the size of each linked program will be large.

Also, a lot of disk and memory space will be wasted by these programs that uses non-shared libraries.
Non-shared v.s. Shared Library

- A library is a collection of pre-written function calls.
- Using existing libraries can save a programmer a lot of time. (e.g., the printf() in the C library)
- However, if a library is not shared among many programs, but instead a copy is included in every program that uses it, the size of each linked program will be large.
- Also, a lot of disk and memory space will be wasted by these programs that uses non-shared libraries.
- This motivates the concept of shared library.
Program with shared library

**Figure 9.1** • Program with shared libraries.
Why Linkers? (cont’d)

**Efficiency**

- **Time: Separate compilation**
  - Change one source file, compile, and then relink.
  - No need to recompile other source files.

- **Space: Libraries**
  - Common functions can be aggregated into a single file...
  - Yet executable files and running memory images contain only code for the functions they actually use.
Outline

1. Background
2. ELF Linking
3. Static Linking
4. Dynamic Linking
5. Summary
Elf Linking

**Figure 4.9** ELF linking.
At link time, the linker searches through libraries to find modules that resolve undefined external symbols.
How Does It Work?

- At link time, the linker searches through libraries to find modules that resolve undefined external symbols.
- Rather than copying the contents of the module into the output file, the linker makes a note of what library the module come from and puts a list of the libraries in the executable.
How Does It Work?

- At link time, the linker searches through libraries to find modules that resolve undefined external symbols.
- Rather than copying the contents of the module into the output file, the linker makes a note of what library the module come from and puts a list of the libraries in the executable.
- When the program is loaded, startup code finds those libraries and maps them into the program’s address space before the program starts.
Elf Linking Overview

Goal: Merges object files

- merges multiple relocatable (.o) object files into a single executable object file that can be loaded and executed by the loader.
Elf Linking Overview

Goal: Merges object files
- merges multiple relocatable (.o) object files into a single executable object file that can be loaded and executed by the loader.

Step I: Resolves external references
- as part of the merging process, resolves external references.
  - external reference: reference to a symbol defined in another object file
Elf Linking Overview

Step II: Relocates symbols

- relocates symbols from their relative locations in the .o files to new absolute positions in the executable.
- updates all references to these symbols to reflect their new positions.
  - references can be in either code or data
    - code: a(); /* ref to symbol a */
    - data: int *xp=&x; /* ref to symbol x */
  - because of this modifying, linking is sometimes called link editing.
Example C Program

```c
main.c
1 int buf[2] = {1, 2};
2
3 int main()
4 {
5   swap();
6   return 0;
7 }

swap.c
1 extern int buf[];
2
3 int *bufp0 = &buf[0];
4 static int *bufp1;
5
6 void swap()
7 {
8   int temp;
9
10  bufp1 = &buf[1];
11  temp = *bufp0;
12  *bufp0 = *bufp1;
13  *bufp1 = temp;
14 }
```
1. Background

2. ELF Linking

3. Static Linking

4. Dynamic Linking

5. Summary
Static Linking

Programs are translated and linked using a compiler driver:

```
unix> gcc -O2 -g -o p main.c swap.c
unix> ./p
```

Source files

Separately compiled relocatable object files

Fully linked executable object file
(contains code and data for all functions defined in main.c and swap.c)
Step I: Symbol resolution

- Programs define and reference symbols (variables and functions):
  
  - `void swap() {...} /* define symbol swap */`
  - `swap(); /* reference symbol a */`
  - `int *xp = &x; /* define symbol xp, reference x */`

- Symbol definitions are stored (by compiler) in symbol table.
  
  - Symbol table is an array of structs
  - Each entry includes name, size, and location of symbol.

- Linker associates each symbol reference with exactly one symbol definition.
Step II: Relocation

- Merges separate code and data sections into single sections
- Relocates symbols from their relative locations in the .o files to their final absolute memory locations in the executable.
- Updates all references to these symbols to reflect their new positions.
Recall: Three Kinds of Object Files

- Relocatable object file (.o file)
  - Contains code and data in a form that can be combined with other relocatable object files to form executable object file.
  - Each .o file is produced from exactly one source (.c) file

- Executable object file (a.out file)
  - Contains code and data in a form that can be copied directly into memory and then executed.

- Shared object file (.so file)
  - Special type of relocatable object file that can be loaded into memory and linked dynamically, at either load time or run-time.
  - Called Dynamic Link Libraries (DLLs) by Windows
ELF object file format

- Elf header
  - magic number, type (.o, exec, .so), machine, byte ordering, etc.
- Program header table
  - page size, virtual addresses for memory segments (sections), segment sizes.
- .text section
  - code
- .data section
  - initialized (static) data
- .bss section
  - uninitialized (static) data
  - “Block Started by Symbol”
  - “Better Save Space”
  - has section header but occupies no space
ELF object file format

- .symtab section
  - symbol table
  - procedure and static variable names
  - section names
- .rel.text section
  - relocation info for .text section
  - addresses of instructions that will need to be modified in the executable
  - instructions for modifying.
- .rel.data section
  - relocation info for .data section
  - addresses of pointer data that will need to be modified in the merged executable
- .debug section
  - info for symbolic debugging (gcc -g)
Global symbols
- Symbols defined by module m that can be referenced by other modules.
- E.g.: non-static C functions and non-static global variables.

External symbols
- Global symbols that are referenced by module m but defined by some other module.

Local symbols
- Symbols that are defined and referenced exclusively by module m.
- E.g.: C functions and variables defined with the static attribute.
- Local linker symbols are not local program variables.
Resolving Symbols

```c
int buf[2] = {1, 2};
int main()
{
    swap();
    return 0;
}
main.c
```

```c
extern int buf[];
int *bufp0 = &buf[0];
static int *bufp1;
void swap()
{
    int temp;
    bufp1 = &buf[1];
    temp = *bufp0;
    *bufp0 = *bufp1;
    *bufp1 = temp;
}
swap.c
```

Global

External

Local

Linker knows nothing of temp
Relocating Code and Data

Relocatable Object Files

- System code
- System data

main.o
- main()
- int buf[2]={1,2}

swap.o
- swap()
- int *bufp0=&buf[0]
- static int *bufp1
- int buf[2]={1,2}
- int *bufp0=&buf[0]

Executable Object File

- Headers
- System code
- main()
- swap()
- More system code
- System data
- int buf[2]={1,2}
- int *bufp0=&buf[0]
- int *bufp1

Even though private to swap, requires allocation in .bss
Relocation Info (main)

main.c

```c
int buf[2] = {1,2};
int main()
{
    swap();
    return 0;
}
```

main.o

```
00000000 <main>:
  0:   8d 4c 24 04      lea    0x4(%esp),%ecx
  4:   83 e4 f0         and    $0xfffffffff0,%esp
  7:   ff 71 fc         pushl   0xfffffffff0(%ecx)
  a:   55               push   %ebp
  b:   89 e5             mov     %esp,%ebp
  d:   51               push   %ecx
  e:   83 ec 04         sub     $0x4,%esp
 11:   e8 fc ff ff ff    call    12 <main+0x12>
 12:   R_386_PC32 swap
 16:   83 c4 04         add     $0x4,%esp
 19:   31 c0             xor     %eax,%eax
 1b:   59               pop      %ecx
 1c:   5d               pop      %ebp
 1d:   8d 61 fc         lea     0xfffffffff0(%ecx),%esp
 20:   c3               ret
```

Disassembly of section .data:

```
00000000 <buf>:
  0:   01 00 00 00 02 00 00 00
```
Relocs in i386 in static linking

**R_386_32**

Simply deposit the **absolute memory address** of "symbol" into a dword

**R_386_PC32**

Determine the distance from this memory location to the "symbol", then add it to the value currently at this dword; deposit the result back into the dword
Relocation Info (swap, .text)

swap.c

```c
extern int buf[];
int *bufp0 = &buf[0];
static int *bufp1;

void swap()
{
    int temp;
    bufp1 = &buf[1];
temp = *bufp0;
*bufp0 = *bufp1;
*bufp1 = temp;
}
```

swap.o

Disassembly of section .text:

```
00000000 <swap>:
  0: 8b 15 00 00 00 00 mov 0x0,%edx
  2: R_386_32 buf
  6: a1 04 00 00 00 mov 0x4,%eax
  7: R_386_32 buf
 b: 55 push %ebp
 c: 89 e5 mov %esp,%ebp
 e: c7 05 00 00 00 00 00 04 movl $0x4,0x0
 15: 00 00 00
  10: R_386_32 .bss
  14: R_386_32 buf
 18: 8b 08 mov (%eax),%ecx
 1a: 89 10 mov %edx,(%eax)
 1c: 5d pop %ebp
 1d: 89 0d 04 00 00 00 mov %ecx,0x4
 1f: R_386_32 buf
 23: c3 ret
```
Relocation Info (swap, .data)

swap.c

extern int buf[];

int *bufp0 = &buf[0];
static int *bufp1;

void swap()
{
    int temp;

    bufp1 = &buf[1];
    temp = *bufp0;
    *bufp0 = *bufp1;
    *bufp1 = temp;
}

Disassembly of section .data:

00000000 <bufp0>:
    0:   00 00 00 00

    0: R_386_32 buf
Executable Before/After Relocation (.text)

0000000 <main>:
  ... e: 83 ec 04    sub  $0x4,%esp
  11: e8 fc ff ff ff  call  12 <main+0x12>
    12: R_386_PC32 swap
  16: 83 c4 04     add  $0x4,%esp
  ... 0x8048396 + 0x1a = 0x80483b0

08048380 <main>:
  8048380:  8d 4c 24 04           lea  0x4(%esp),%ecx
  8048384:  83 e4 f0              and  $0xffffffff0,%esp
  8048387:  ff 71 fc              pushl 0xfffffffffc(%ecx)
  804838a:  55                  push  %ebp
  804838b:  89 e5               mov   %esp,%ebp
  804838d:  51                  push  %ecx
  804838e:  83 ec 04              sub  $0x4,%esp
  8048391:  e8 1a 00 00 00          call  80483b0 <swap>
  8048396:  83 c4 04              add  $0x4,%esp
  8048399:  31 c0                xor  %eax,%eax
  804839b:  59                  pop   %ecx
  804839c:  5d                  pop   %ebp
  804839d:  8d 61 fc              lea  0xfffffffffc(%ecx),%esp
  80483a0:  c3                  ret
Executable Before/After Relocation (.text)

<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>NumberOfOperands</th>
<th>Operand1</th>
<th>Operand2</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>8b 15 00 00 00 00</td>
<td>mov</td>
<td>0x0, %edx</td>
<td></td>
</tr>
<tr>
<td>00000002</td>
<td>R_386_32</td>
<td>buf</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000006</td>
<td>a1 04 00 00 00</td>
<td>mov</td>
<td>0x4, %eax</td>
<td></td>
</tr>
<tr>
<td>00000007</td>
<td>R_386_32</td>
<td>buf</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000000C</td>
<td>c7 05 00 00 00 00 04</td>
<td>movl</td>
<td>$0x4,0x0</td>
<td></td>
</tr>
<tr>
<td>00000017</td>
<td>R_386_32</td>
<td>.bss</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000001C</td>
<td>R_386_32</td>
<td>buf</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000021</td>
<td>89 0d 04 00 00 00</td>
<td>mov</td>
<td>%ecx,0x4</td>
<td></td>
</tr>
<tr>
<td>00000027</td>
<td>R_386_32</td>
<td>buf</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000002C</td>
<td>c3</td>
<td>ret</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

080483b0 <swap>:

<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>NumberOfOperands</th>
<th>Operand1</th>
<th>Operand2</th>
</tr>
</thead>
<tbody>
<tr>
<td>080483b0</td>
<td>8b 15 20 96 04 08</td>
<td>mov</td>
<td>0x8049620, %edx</td>
<td></td>
</tr>
<tr>
<td>080483b6</td>
<td>a1 24 96 04 08</td>
<td>mov</td>
<td>0x8049624, %eax</td>
<td></td>
</tr>
<tr>
<td>080483bb</td>
<td>55</td>
<td>push</td>
<td>%ebp</td>
<td></td>
</tr>
<tr>
<td>080483bc</td>
<td>89 e5</td>
<td>mov</td>
<td>%esp,%ebp</td>
<td></td>
</tr>
<tr>
<td>080483be</td>
<td>c7 05 30 96 04 08 24</td>
<td>movl</td>
<td>$0x8049624,0x8049630</td>
<td></td>
</tr>
<tr>
<td>080483c5</td>
<td>96 04 08</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>080483c8</td>
<td>8b 08</td>
<td>mov</td>
<td>(%eax),%ecx</td>
<td></td>
</tr>
<tr>
<td>080483ca</td>
<td>89 10</td>
<td>mov</td>
<td>%edx, (%eax)</td>
<td></td>
</tr>
<tr>
<td>080483cc</td>
<td>5d</td>
<td>pop</td>
<td>%ebp</td>
<td></td>
</tr>
<tr>
<td>080483cd</td>
<td>89 0d 24 96 04 08</td>
<td>mov</td>
<td>%ecx,0x8049624</td>
<td></td>
</tr>
<tr>
<td>080483d3</td>
<td>c3</td>
<td>ret</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Disassembly of section .data:

08049620 <buf>:
  8049620:       01 00 00 00 02 00 00 00

08049628 <bufp0>:
  8049628:       20 96 04 08
Program symbols are either strong or weak

- **Strong**: procedures and initialized globals
- **Weak**: uninitialized globals

Carnegie Mellon
25

```c
int foo=5;
p1() {
}
```

```c
int foo;
p2() {
}
p1.c p2.c
```

- **Strong**: procedures and initialized globals
- **Weak**: uninitialized globals

```c
int foo=5;
p1() {
}
```

```c
int foo;
p2() {
}
p1.c p2.c
```
Linker’s Symbol Rules

- Rule 1: Multiple strong symbols are not allowed
  - Each item can be defined only once
  - Otherwise: Linker error

- Rule 2: Given a strong symbol and multiple weak symbol, choose the strong symbol
  - References to the weak symbol resolve to the strong symbol

- Rule 3: If there are multiple weak symbols, pick an arbitrary one
  - Can override this with `gcc -fno-common`
Symbols are lexical entities that name functions and variables.

Each symbol has a value (typically a memory address).

Code consists of symbol definitions and references.

References can be either local or external.
Linker Puzzles

- **int x;**
  - p1() {}
  - p1() {}

  Link time error: two strong symbols (p1)

- **int x;**
  - int x;
  - p1() {}
  - p2() {}

  References to x will refer to the same uninitialized int. Is this what you really want?

- **int x;**
  - double x;
  - int y;
  - p1() {}
  - p2() {}

  Writes to x in p2 might overwrite y!
  Evil!

- **int x=7;**
  - int y=5;
  - p1() {}
  - p2() {}

  Writes to x in p2 will overwrite y!
  Nasty!

- **int x=7;**
  - int x;
  - p1() {}
  - p2() {}

  References to x will refer to the same initialized variable.

**Nightmare scenario:** two identical weak structs, compiled by different compilers with different alignment rules.
Role of .h Files

c1.c

```c
#include "global.h"

int f() {
    return g+1;
}
```

c2.c

```c
#include <stdio.h>
#include "global.h"

int main() {
    if (!init)
        g = 37;
    int t = f();
    printf("Calling f yields %d\n", t);
    return 0;
}
```

global.h

```c
#ifdef INITIALIZE
    int g = 23;
    static int init = 1;
#else
    int g;
    static int init = 0;
#endif
```
Running Preprocessor

#include "global.h"

int f() {
  return g+1;
}

global.h

#include causes C preprocessor to insert file verbatim

-DINITIALIZE

no initialization
Role of .h Files (cont'd)

What happens:
gcc -o p c1.c c2.c
  ??
gcc -o p c1.c c2.c \\ -DINITIALIZE
  ??

- c1.c

```c
#include "global.h"

int f() {
    return g+1;
}
```

- global.h

```c
#include <stdio.h>
#include "global.h"

int main() {
    if (!init)
        g = 37;
    int t = f();
    printf("Calling f yields %d\n", t);
    return 0;
}
```

- c2.c

```c
#include <stdio.h>
#include "global.h"

int f() {
    return g+1;
}
```

```c
#include <stdio.h>
#include "global.h"

int main() {
    if (!init)
        g = 37;
    int t = f();
    printf("Calling f yields %d\n", t);
    return 0;
}
```
Global Variables

- Avoid if you can
- Otherwise
  - Use **static** if you can
  - Initialize if you define a global variable
  - Use **extern** if you use external global variable
How to package functions commonly used by programmers?

- Math, I/O, memory management, string manipulation, etc.

Awkward, given the linker framework so far:

- **Option 1**: Put all functions into a single source file
  - Programmers link big object file into their programs
  - Space and time inefficient

- **Option 2**: Put each function in a separate source file
  - Programmers explicitly link appropriate binaries into their programs
  - More efficient, but burdensome on the programmer
Solution: Static Libraries

- **Static libraries** (.a archive files)
  - Concatenate related relocatable object files into a single file with an index (called an archive).
  - Enhance linker so that it tries to resolve unresolved external references by looking for the symbols in one or more archives.
  - If an archive member file resolves reference, link it into the executable.
Creating Static Libraries

Archiver allows incremental updates
Recompile function that changes and replace .o file in archive.
Commonly Used Libraries

- libc.a (the C standard library)
  - 8 MB archive of 1392 object files.
  - I/O, memory allocation, signal handling, string handling, data and time, random numbers, integer math

- libm.a (the C math library)
  - 1 MB archive of 401 object files.
  - Floating point math (sin, cos, tan, log, exp, sqrt, ...)

% ar -t /usr/lib/libc.a | sort
... fork.o ...
fprintf.o fpu_control.o fputc.o freopen.o fscanf.o fseek.o fseek.o fstab.o ...

% ar -t /usr/lib/libm.a | sort
... e_acos.o e_acosf.o e_acosh.o e_acoshf.o e_acoshl.o e_acosl.o e_asin.o e_asinf.o e_asinl.o e_asinl.o ...

Linking with Static Libraries

Translators (cpp, cc1, as)

main2.c vector.h

Archiver (ar)

libvector.a libc.a

Addvec.o multvec.o

Linker (ld)

main2.o

addvec.o

printf.o and any other modules called by printf.o

Relocatable object files

Fully linked executable object file

Static libraries
Using Static Libraries

Linker’s algorithm for resolving external references:

- Scan .o files and .a files in the command line order.
- During the scan, keep a list of the current unresolved references.
- As each new .o or .a file, obj, is encountered, try to resolve each unresolved reference in the list against the symbols defined in obj.
- If any entries in the unresolved list at end of scan, then error.

Problem:

- Command line order matters!
- Moral: put libraries at the end of the command line.
Loading Executable Object Files

Executable Object File

- ELF header
- Program header table (required for executables)
- .init section
- .text section
- .rodata section
- .data section
- .bss section
- .symtab
- .debug
- .line
- .strtab

Section header table (required for relocatables)

Memory outside 32-bit address space

Kernel virtual memory
- User stack (created at runtime)
- Memory-mapped region for shared libraries
- Run-time heap (created by malloc)
- Read/write segment (.data, .bss)
- Read-only segment (.init, .text, .rodata)

Unused

Loaded from the executable file

Memory - mapped region for shared libraries

User stack (created at runtime)

Run-time heap (created by malloc)

User stack

Run-time heap

Memory - mapped region for shared libraries

Memory - mapped region for shared libraries
1. Background
2. ELF Linking
3. Static Linking
4. Dynamic Linking
5. Summary
Shared Libraries

- Static libraries have the following disadvantages:
  - Duplication in the stored executables (every function need std libc)
  - Duplication in the running executables
  - Minor bug fixes of system libraries require each application to explicitly relink

- Modern solution: Shared Libraries
  - Object files that contain code and data that are loaded and linked into an application dynamically, at either load-time or run-time
  - Also called: dynamic link libraries, DLLs, .so files
Dynamic linking can occur when executable is first loaded and run (load-time linking).

- Common case for Linux, handled automatically by the dynamic linker (`ld-linux.so`).
- Standard C library (`libc.so`) usually dynamically linked.

Dynamic linking can also occur after program has begun (run-time linking).

- In Linux, this is done by calls to the `dlopen()` interface.
  - Distributing software.
  - High-performance web servers.
  - Runtime library interpositioning.

Shared library routines can be shared by multiple processes.

- More on this when we learn about virtual memory
Dynamic Linking at Load-time

Translators (cpp, cc1, as)

main2.c  vector.h

Translators (cpp, cc1, as)

main2.o

Translators (cpp, cc1, as)

linker (ld)

Translators (cpp, cc1, as)

p2

Translators (cpp, cc1, as)

Loader (execve)

Translators (cpp, cc1, as)

Dynamic linker (ld-linux.so)

Translators (cpp, cc1, as)

Relocatable object file

Partially linked executable object file

Fully linked executable in memory

unix> gcc -shared -o libvector.so \
     addvec.c multvec.c

Translator:

addvec.c multvec.c

Relocation and symbol table info

libc.so
libvector.so

Code and data

libc.so
libvector.so
Examples of Dynamic Linking

```c
#cat hello1.c
main()
{
    greet();
}
```

```c
#cat english.c
greet()
{
    printf("Hello World");
}
```

```c
#gcc -fPIC -c english.c
#gcc -shared -o libenglish.so english.o
#gcc -c hello1.c
#gcc -o hello1 hello1.o -L. -lenglish

# LD_LIBRARY_PATH=. ./hello1
Hello World
```
## Trace it using objdump

```
softos-server-debian:~/elf-test$ objdump -s -j .dynstr hello1

hello1: file type: elf32-i386

.dynstr:
  8048240 006c6962 656e676c 6973682e 736f005f .libenglish.so._
  8048250 44594e41 4d494300 5f696e69 74006772 DYNAMIC._init.gr
eet._fini._GLOBALA
  8048260 65657400 5f66696e 69005f47 4c4f4241 L_OFFSET_TABLE_.
  8048270 4c5f4f46 5f556e6400 5f5f6c69 _stdin_used.__li
  8048280 62635f73 74617274 005f656e 6400474c _bss_start.
  8048290 005f494f 5f737464696e5f73 74617274 _IO__gmon_start
  80482a0 5f73746962656e676c6973682e736f2e36 .libc.so.6._IO
  80482b0 5f5f006c 6973682e 736f005f 494f 5f73746962656e676c6973682e736f2e36 _IO__gmon_start
  80482c0 62635f73 74617274 005f656e 6400474c _bss_start.
  80482d0 005f656e 6400474c 49424352 322e3000 _GLIBC_2.0.
```

---

**Background**

- ELF Linking
- Static Linking
- Dynamic Linking

**Summary**

- Trace it using objdump

---
Let’s see again on library creating

Static Library (How to Create)

```
vi hello-1.c
gcc -c hello-1.c
ar -r hello-1.a hello-1.o
-> get hello-1.c
-> get hello-1.o
-> get hello-1.a (static library)
```

Static Library (How to Use)

```
vi main-1.c
gcc -c main-1.c
gcc -o main-1 main-1.o hello-1.a
./main-1
-> get main-1.c
-> get main-1.o
-> get main-1
-> execute main-1
```
Let's see again on library creating

### Dynamic Library (How to Create)

<table>
<thead>
<tr>
<th>Command</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>vi hello-2.c</td>
<td>get hello-2.c</td>
</tr>
<tr>
<td>gcc -c hello-2.c</td>
<td>get hello-2.o</td>
</tr>
<tr>
<td>gcc -shared -o libhello-2.so hello-2.o</td>
<td>get libhello-2.so (dynamic library)</td>
</tr>
</tbody>
</table>

### Dynamic Library (How to Use)

<table>
<thead>
<tr>
<th>Command</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>vi main-2.c</td>
<td>get main-2.c</td>
</tr>
<tr>
<td>gcc -c main-2.c</td>
<td>get main-2.o</td>
</tr>
<tr>
<td>gcc -o main-2 main-2.o -L -lhello-2</td>
<td>get main-2</td>
</tr>
<tr>
<td>export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:.</td>
<td></td>
</tr>
<tr>
<td>./main-2</td>
<td>execute main-2</td>
</tr>
</tbody>
</table>
Dynamic Linking at Run-time

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

int x[2] = {1, 2};
int y[2] = {3, 4};
int z[2];

int main()
{
    void *handle;
    void (*addvec)(int *, int *, int *, int);
    char *error;

    /* dynamically load the shared lib that contains addvec() */
    handle = dlopen("./libvector.so", RTLD_LAZY);
    if (!handle) {
        fprintf(stderr, "%s\n", dlerror());
        exit(1);
    }
```
Dynamic Linking at Run-time

.../* get a pointer to the addvec() function we just loaded */addvec = dlsym(handle, "addvec");if ((error = dlerror()) != NULL) {
    fprintf(stderr, "%s\n", error);
    exit(1);
}

/* Now we can call addvec() just like any other function */addvec(x, y, z, 2);
printf("z = [%d %d]\n", z[0], z[1]);

/* unload the shared library */if (dlclose(handle) < 0) {
    fprintf(stderr, "%s\n", dlerror());
    exit(1);
}return 0;
Outline

1. Background
2. ELF Linking
3. Static Linking
4. Dynamic Linking
5. Summary
NAME

ld - The GNU linker

SYNOPSIS

ld [options] objfile ...

DESCRIPTION

ld combines a number of object and archive files, relocates their data and ties up symbol references. Usually the last step in compiling a program is to run ld.

Linker is one of the key system components
References

- http://www.linuxjournal.com/article/6463
- Linker and Loader