ELF binaries and everything before main() starts

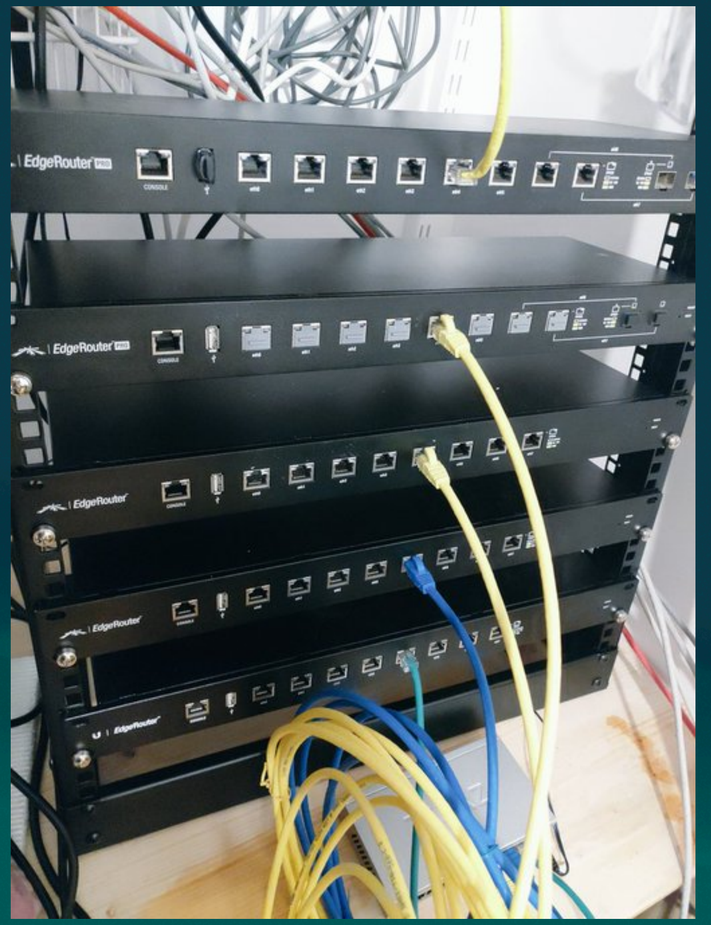
jj@deadzoft.org 2023-05-18

$ whoami

jj@deadzoft.org 2023-05-18

$ whoami

- Started with OpenBSD 2.2 in the 90s



jj@deadzoft.org 2023-05-18

$ whoami

- Started with OpenBSD 2.2 in the 90s
- Did some release builds for OpenBSD-amiga (m68k)

jj@deadzoft.org 2023-05-18

$ whoami

- Started with OpenBSD 2.2 in the 90s
- Did some release builds for OpenBSD-amiga (m68k)
- Off an on as jj@openbsd.org (currently off)

jj@deadzoft.org 2023-05-18

$ whoami

- Started with OpenBSD 2.2 in the 90s
- Did some release builds for OpenBSD-amiga (m68k)
- Off an on as jj@openbsd.org (currently off)
- Host various services as *.eu.openbsd.org

jj@deadzoft.org 2023-05-18

$ whoami

- Started with OpenBSD 2.2 in the 90s
- Did some release builds for OpenBSD-amiga (m68k)
- Off an on as jj@openbsd.org (currently off)
- Host various services as *.eu.openbsd.org
- “IcePic” on IRC

jj@deadzoft.org 2023-05-18

$ whoami

- Started with OpenBSD 2.2 in the 90s
- Did some release builds for OpenBSD-amiga (m68k)
- Off an on as jj@openbsd.org (currently off)
- Host various services as *.eu.openbsd.org
- “IcePic” on IRC
- Collects octeon/mips64 based computers

jj@deadzoft.org 2023-05-18

Contents

- A Simple(tm) Program
- Object files
- Shared libraries
- ELF sections
- Loading binaries into RAM
- The role of ld.so
- GOT / PLT
- OpenBSD recent changes

What will not be here

- Long lists of code or pictures of ELF-code C structs
- How Rust,Golang,Zig,… binaries work

A Simple(tm) Program

```
> mg hello.c
> cat hello.c
#include "stdio.h"

int main() {
        printf("Hello world!\n");
        return 0;
}
> make hello
cc     hello.c   -o hello
> ./hello
Hello world!
> ls -l hello
-rwxr-xr-x  1 jj  staff  49424 Jul 28 12:43 hello
>
```

A Simple(tm) Program

A Simple(tm) Program

- Building your C Program:

A Simple(tm) Program

- Building your C Program:
  - C source to assembler (.s)

A Simple(tm) Program

- Building your C Program:
  - C source to assembler (.s)
  - Assembler source to object (.o) file

A Simple(tm) Program

- Building your C Program:
  - C source to assembler (.s)
  - Assembler source to object (.o) file
  - Linker joins one or more object files to a binary

A Simple(tm) Program

- Building your C Program:
  - C source to assembler (.s)
  - Assembler source to object (.o) file
  - Linker joins one or more object files to a binary
- Prg.c -> Prg.s -> Prg.o -> Prg

A Simple(tm) Program

- Building your C Program:
  - C source to assembler (.s)
  - Assembler source to object (.o) file
  - Linker joins one or more object files to a binary
- Prg.c -> Prg.s -> Prg.o -> Prg
- (cc —save-temps)

Object files

Object files

- Building your C Program:

Object files

- Building your C Program:
  - clang and gcc often do all in one step

Object files

- Building your C Program:
  - clang and gcc often do all in one step
  - Takes long time for huge source files

Object files

- Building your C Program:
  - clang and gcc often do all in one step
  - Takes long time for huge source files
  - Split sources into separate files leading to multiple .o object files linked into one binary

Object files

- Building your C Program:
  - clang and gcc often do all in one step
  - Takes long time for huge source files
  - Split sources into separate files leading to multiple .o object files linked into one binary
  - a.c -> a.o
    b.c -> b.o
    ld a.o b.o -o ./c-prg

Object files

Object files

- Building your C Program:

Object files

- Building your C Program:
  - The linking is the interesting step for us

Object files

- Building your C Program:
  - The linking is the interesting step for us
  - Last chance for certain optimizations (LTO) or joining similar sections into one

Object files

- Building your C Program:
  - The linking is the interesting step for us
  - Last chance for certain optimizations (LTO) or joining similar sections into one
  - Takes more than 4G RAM for linking browsers with full debug info

Object Files

Object Files

- Building your C Program:

Object Files

- Building your C Program:
  - Each object file lists which symbols it provides and which ones it requires

Object Files

- Building your C Program:
  - Each object file lists which symbols it provides and which ones it requires
  - C++ programs and objects “mangle” their functions (methods) to include information about what data types they accept and return

Object Files

Object Files

- Building your C Program:

Object Files

- Building your C Program:
  - int add_to_d(a,b)
    { int c; extern int d; return c=d+a+b; }

Object Files

- Building your C Program:
  - int add_to_d(a,b)
    { int c; extern int d; return c=d+a+b; }
  - will require “d” and provide “add_to_d” but not a,b or c. Those are just anonymous ints in the code

Object Files

Object Files

- Building your C Program:

Object Files

- Building your C Program:
  - Use objdump -t to list what an object file or a program requires and provides.

Object Files

- Building your C Program:
  - Use objdump -t to list what an object file or a program requires and provides.
  - nm(1) also works, readelf(1) and many other utilities

Object Files

Object Files

- Building your C Program:

Object Files

- Building your C Program:
  - Adding 3rd party library named libabc:

Object Files

- Building your C Program:
  - Adding 3rd party library named libabc:
  - #include “abc-lib.h”

Object Files

- Building your C Program:
  - Adding 3rd party library named libabc:
  - #include “abc-lib.h”
  - Calling abc(ABC_OK,myint); from your program

Object Files

- Building your C Program:
  - Adding 3rd party library named libabc:
  - #include “abc-lib.h”
  - Calling abc(ABC_OK,myint); from your program
  - Your object file/program now requires the “abc” symbol

Object Files

- Building your C Program:
  - Adding 3rd party library named libabc:
  - #include “abc-lib.h”
  - Calling abc(ABC_OK,myint); from your program
  - Your object file/program now requires the “abc” symbol
  - Perhaps an abc.c -> abc.o exists?

Static libraries

Static libraries

- Building your C Program:

Static libraries

- Building your C Program:
  - Use cc -static -labc

Static libraries

- Building your C Program:
  - Use cc -static -labc
  - Static libraries are precompiled objects.

Static libraries

- Building your C Program:
  - Use cc -static -labc
  - Static libraries are precompiled objects.
  - a.o b.o c.o -> /usr/lib/libabc.a

Static libraries

- Building your C Program:
  - Use cc -static -labc
  - Static libraries are precompiled objects.
  - a.o b.o c.o -> /usr/lib/libabc.a
  - Makes your binary large, can’t update abc()

Static libraries

- Building your C Program:
  - Use cc -static -labc
  - Static libraries are precompiled objects.
  - a.o b.o c.o -> /usr/lib/libabc.a
  - Makes your binary large, can’t update abc()
  - Vendoring lib sources is boring

Shared Libraries

Shared Libraries

- Building your C Program:

Shared Libraries

- Building your C Program:
  - If you have /usr/lib/libabc.so(.12.3.4) you can get the abc symbol from it. Use cc -labc

Shared Libraries

- Building your C Program:
  - If you have /usr/lib/libabc.so(.12.3.4) you can get the abc symbol from it. Use cc -labc
  - But OS and exec*() calls need to make sure libabc.so is in memory when your program is done loading

Shared Libraries

- Building your C Program:
  - If you have /usr/lib/libabc.so(.12.3.4) you can get the abc symbol from it. Use cc -labc
  - But OS and exec*() calls need to make sure libabc.so is in memory when your program is done loading
  - Linker does checks, runtime fail if lib is missing

Shared Libraries

- Building your C Program:
  - If you have /usr/lib/libabc.so(.12.3.4) you can get the abc symbol from it. Use cc -labc
  - But OS and exec*() calls need to make sure libabc.so is in memory when your program is done loading
  - Linker does checks, runtime fail if lib is missing
  - Dynamic lib might in turn ALSO require symbols

A Less Simple(tm) Program

```
> mg hello.c
> cat hello.c
#include "stdio.h"

int flurb;

void print() {
        int blaha;
        printf("blaha: %d flurb: %d\n", blaha, flurb);
}

int main() {
        printf("Starting\n");
        print();
        return 0;
}
> make hello
cc -O2 -pipe    -o hello hello.c
> ./hello
Starting
blaha: 1114820608 flurb: 0
> ./hello
Starting
blaha: 325099520 flurb: 0
> ./hello
Starting
blaha: -1199149056 flurb: 0
>
```

A Less Simple(tm) Program

```
> mg hello.c
> cat hello.c
#include "stdio.h"

int flurb;

void print() {
        int blaha;
        printf("blaha: %d flurb: %d\n", blaha, flurb);
}

int main() {
        printf("Starting\n");
        print();
        return 0;
}
> make hello
cc -O2 -pipe    -o hello hello.c
> ./hello
Starting
blaha: 1114820608 flurb: 0
> ./hello
Starting
blaha: 325099520 flurb: 0
> ./hello
Starting
blaha: -1199149056 flurb: 0
>
```

A Less Simple(tm) Program

```
> objdump -t hello |tail -18
0000000000002ba0 l     O .openbsd.randomdata    0000000000000008 .hidden __retguard_1773
0000000000002b88 l     O .openbsd.randomdata    0000000000000008 .hidden __guard_local
00000000000016b0 g       .text  0000000000000000 __start
00000000000016b0 g       .text  0000000000000000 _start
0000000000000000         F *UND*  0000000000000000 _csu_finish
0000000000001a20 g     F .text  0000000000000050 main
0000000000000000         F *UND*  0000000000000000 exit
0000000000001a80 g     F .fini  0000000000000000 __fini
0000000000001820  w    F .text  0000000000000020 __register_frame_info
0000000000000000  w      *UND*  0000000000000000 _Jv_RegisterClasses
0000000000000000         F *UND*  0000000000000000 atexit
0000000000001a00 g     F .text  0000000000000019 print
0000000000003db0 g     O .bss   0000000000000004 flurb
0000000000000000         F *UND*  0000000000000000 printf
0000000000000000         F *UND*  0000000000000000 puts
0000000000003db4 g       .bss   0000000000000000 _end


>
```

A Less Simple(tm) Program

```
> objdump -t hello |tail -18
0000000000002ba0 l     O .openbsd.randomdata    0000000000000008 .hidden __retguard_1773
0000000000002b88 l     O .openbsd.randomdata    0000000000000008 .hidden __guard_local
00000000000016b0 g       .text  0000000000000000 __start
00000000000016b0 g       .text  0000000000000000 _start
0000000000000000         F *UND*  0000000000000000 _csu_finish
0000000000001a20 g     F .text  0000000000000050 main
0000000000000000         F *UND*  0000000000000000 exit
0000000000001a80 g     F .fini  0000000000000000 __fini
0000000000001820  w    F .text  0000000000000020 __register_frame_info
0000000000000000  w      *UND*  0000000000000000 _Jv_RegisterClasses
0000000000000000         F *UND*  0000000000000000 atexit
0000000000001a00 g     F .text  0000000000000019 print
0000000000003db0 g     O .bss   0000000000000004 flurb
0000000000000000         F *UND*  0000000000000000 printf
0000000000000000         F *UND*  0000000000000000 puts
0000000000003db4 g       .bss   0000000000000000 _end


>
```

A Less Simple(tm) Program

```
> mg hello.c
> cat hello.c
#include "stdio.h"

int flurb;

void print() {
        int blaha;
        printf("blaha: %d flurb: %d\n", blaha, flurb);
}

int main() {
        printf("Starting\n");
        print();
        return 0;
}
> make hello
cc -O2 -pipe    -o hello hello.c
> ./hello
Starting
blaha: 1114820608 flurb: 0
> ./hello
Starting
blaha: 325099520 flurb: 0
> ./hello
Starting
blaha: -1199149056 flurb: 0
>
```

ELF sections

ELF sections

- ELF binaries

ELF sections

- ELF binaries
  - A header listing number and length of sections

ELF sections

- ELF binaries
  - A header listing number and length of sections
  - Each section with its own content, length and type

ELF sections

- ELF binaries
  - A header listing number and length of sections
  - Each section with its own content, length and type
  - Lots of optional sections, like debug info

ELF sections

- ELF binaries
  - A header listing number and length of sections
  - Each section with its own content, length and type
  - Lots of optional sections, like debug info
  - Kernel skips loading optional sections when executing

ELF sections

- ELF binaries
  - A header listing number and length of sections
  - Each section with its own content, length and type
  - Lots of optional sections, like debug info
  - Kernel skips loading optional sections when executing
  - strip(1) removes all “unused” sections

ELF sections

ELF sections

- ELF binaries with debug symbols

ELF sections

- ELF binaries with debug symbols
  - Debug info could simply be labels and function names

ELF sections

- ELF binaries with debug symbols
  - Debug info could simply be labels and function names
  - But it can also be the whole source file

ELF sections

- ELF binaries with debug symbols
  - Debug info could simply be labels and function names
  - But it can also be the whole source file
  - Allows for godbolt style line-by-line assembler dumps/debugger listings

ELF sections

ELF sections

- ELF binaries

ELF sections

- ELF binaries
  - Text - Where the code goes. Readonly in mem

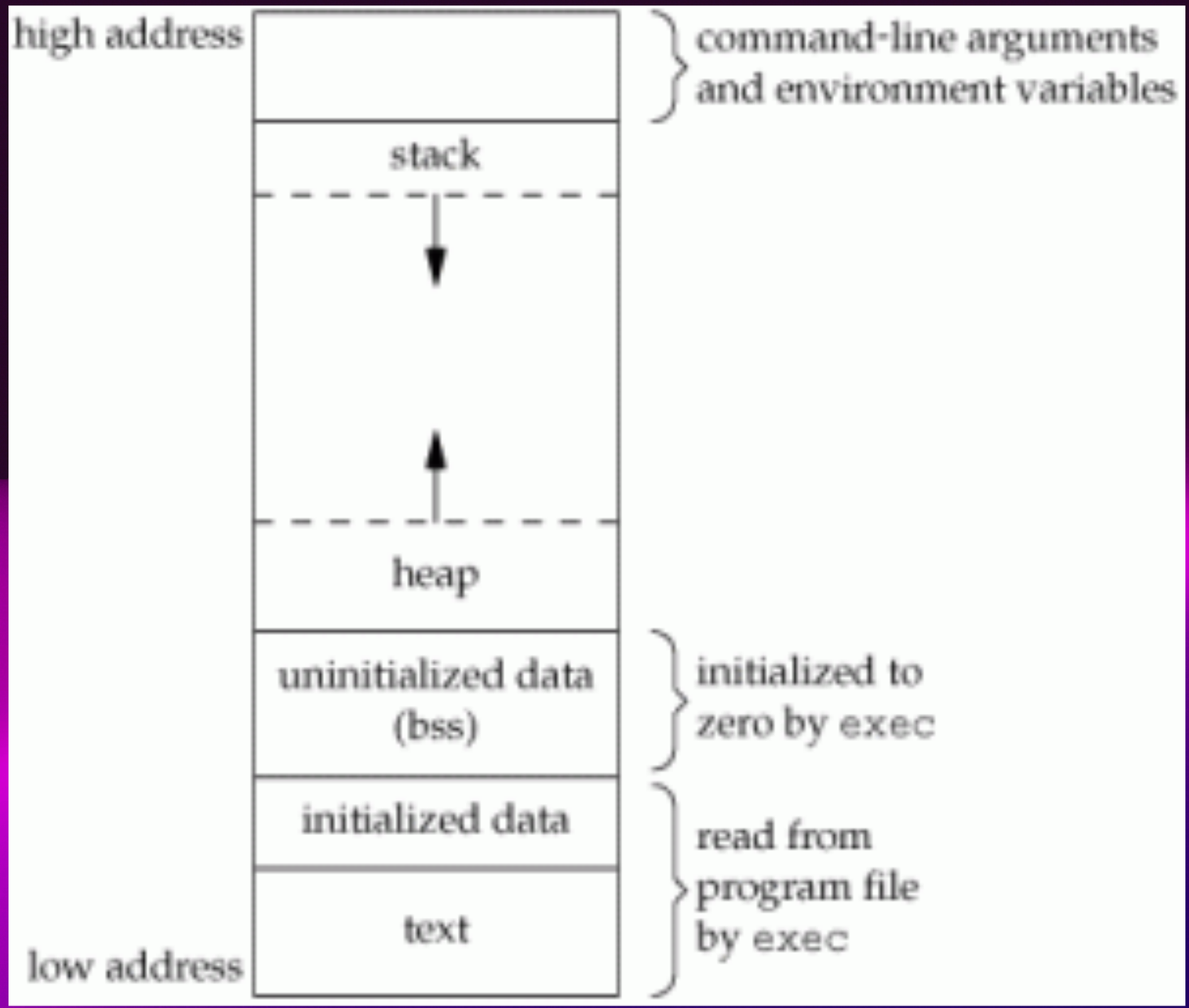
ELF sections

- ELF binaries
  - Text - Where the code goes. Readonly in mem
  - Data - All strings and variables with non-zero content

ELF sections

- ELF binaries
  - Text - Where the code goes. Readonly in mem
  - Data - All strings and variables with non-zero content
  - BSS - For all zero-filled variables and structs. The BSS section only has a size, then variables point into this calloc(3)ed space

ELF sections



ELF sections

```
fido$ size hello
text    data    bss     dec     hex
2172    472     92      2736    ab0
```

ELF sections

- ELF binaries

```
fido$ size hello
text    data    bss     dec     hex
2172    472     92      2736    ab0
```

ELF sections

- ELF binaries
  - Text - Where the code goes. Readonly in mem

```
fido$ size hello
text    data    bss     dec     hex
2172    472     92      2736    ab0
```

ELF sections

- ELF binaries
  - Text - Where the code goes. Readonly in mem
  - Data - All strings and variables with non-zero content

```
fido$ size hello
text    data    bss     dec     hex
2172    472     92      2736    ab0
```

ELF sections

- ELF binaries
  - Text - Where the code goes. Readonly in mem
  - Data - All strings and variables with non-zero content
  - BSS - For all zero-filled variables and structs

```
fido$ size hello
text    data    bss     dec     hex
2172    472     92      2736    ab0
```

ELF sections

ELF sections

- ELF binaries

ELF sections

- ELF binaries
  - Ok, a few more sections than just text, data & bss.

ELF sections

- ELF binaries
  - Ok, a few more sections than just text, data & bss.

```
fido$ size hello
text    data    bss     dec     hex
2172    472     92      2736    ab0
fido$ llvm-objdump -h   hello

hello:  file format elf64-x86-64

Sections:
Idx Name                  Size     VMA                 Type
  0                       00000000 0000000000000000
  1 .interp               00000013 00000000000002e0 DATA
  2 .note.openbsd.ident   00000018 00000000000002f4
  3 .dynsym               000000c0 0000000000000310
  4 .gnu.hash             00000020 00000000000003d0
  5 .hash                 00000048 00000000000003f0
  6 .dynstr               0000004b 0000000000000438
  7 .rela.dyn             00000030 0000000000000488
  8 .rela.plt             00000090 00000000000004b8
  9 .rodata               0000001e 0000000000000548 DATA
 10 .eh_frame_hdr         0000003c 0000000000000568 DATA
 11 .eh_frame             000000fc 00000000000005a8 DATA
 12 .text                 000003bc 00000000000016b0 TEXT
 13 .init                 0000000e 0000000000001a70 TEXT
 14 .fini                 0000000e 0000000000001a80 TEXT
 15 .plt                  000000f0 0000000000001a90 TEXT
 16 .openbsd.randomdata   00000038 0000000000002b80 DATA
 17 .jcr                  00000008 0000000000002bb8 DATA
 18 .ctors                00000010 0000000000002bc0 DATA
 19 .dtors                00000010 0000000000002bd0 DATA
 20 .dynamic              00000120 0000000000002be0
 21 .got                  00000010 0000000000002d00 DATA
 22 .got.plt              00000048 0000000000002d10 DATA
 23 .bss                  0000005c 0000000000003d58 BSS
 24 .comment              00000013 0000000000000000
 25 .symtab               00000330 0000000000000000
 26 .shstrtab             000000ee 0000000000000000
 27 .strtab               000001a9 0000000000000000
```

ELF sections

ELF sections

- ELF binaries

ELF sections

- ELF binaries
  - Text - Where the code goes.

ELF sections

- ELF binaries
  - Text - Where the code goes.
  - Text sections list for which CPU and arch they belong

ELF sections

- ELF binaries
  - Text - Where the code goes.
  - Text sections list for which CPU and arch they belong
  - Means you can make “fat binaries” with several CPU/Arch code sections that all reference same variables in data and BSS sections

ELF sections

- Data Section(s)

A Less Simple(tm) Program

```
> mg hello.c
> cat hello.c
#include "stdio.h"

int flurb;

void print() {
        int blaha;
        printf("blaha: %d flurb: %d\n", blaha, flurb);
}

int main() {
        printf("Starting\n");
        print();
        return 0;
}
> make hello
cc -O2 -pipe    -o hello hello.c
> ./hello
Starting
blaha: 1114820608 flurb: 0
> ./hello
Starting
blaha: 325099520 flurb: 0
> ./hello
Starting
blaha: -1199149056 flurb: 0
>
```

A Less Simple(tm) Program

```
[> mg hello.c
[> cat hello.c
#include "stdio.h"

int flurb;

void print() {
        int blaha;
        printf("blaha: %d flurb: %d\n", blaha, flurb);
}

int main() {
        printf("Starting\n");
        print();
        return 0;
}
[> make hello
cc -O2 -pipe    -o hello hello.c
[> ./hello
Starting
blaha: 1114820608 flurb: 0
[> ./hello
Starting
blaha: 325099520 flurb: 0
[> ./hello
Starting
blaha: -1199149056 flurb: 0
>
```

A Less Simple(tm) Program

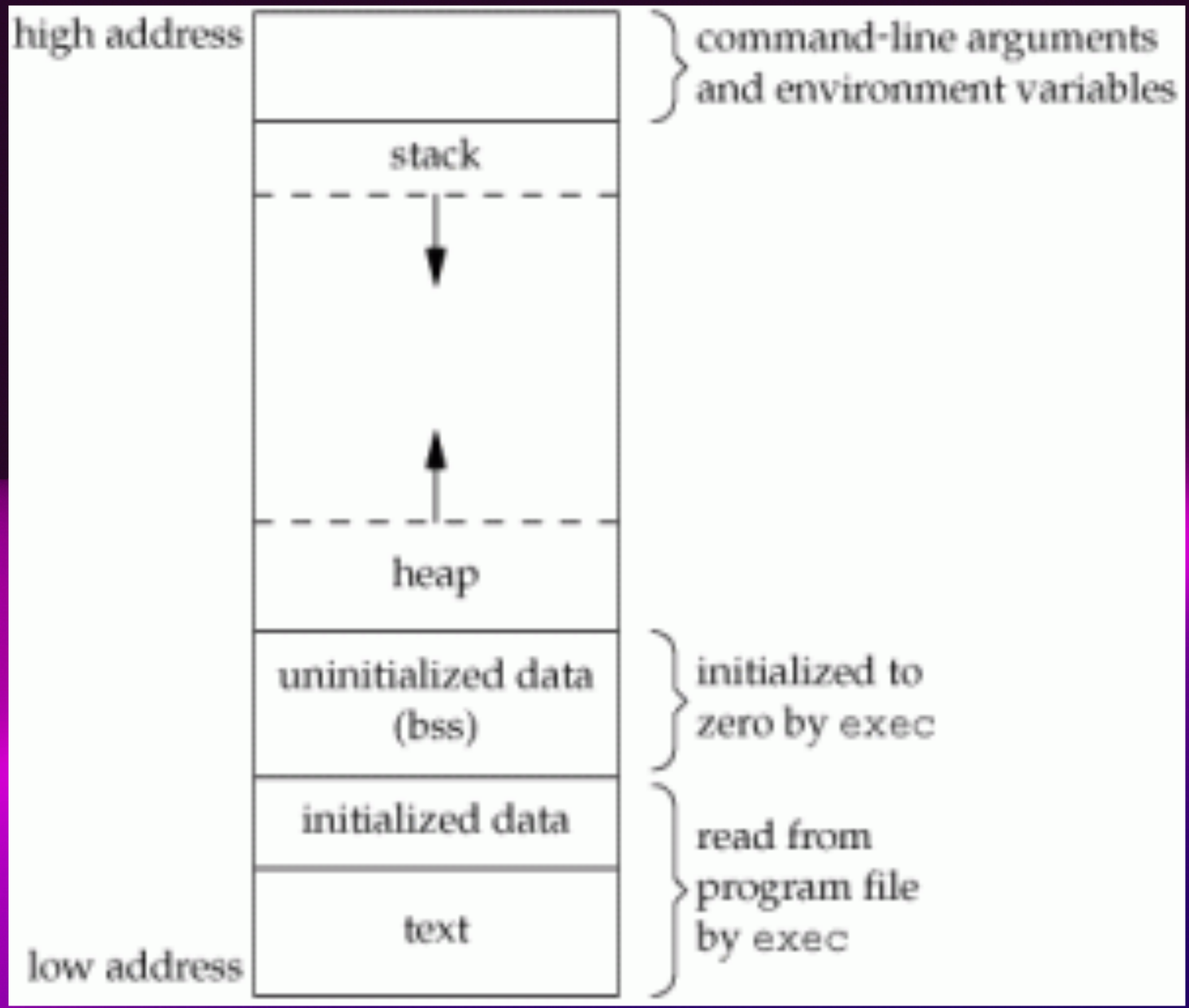
```
[> mg hello.c
[> cat hello.c
#include "stdio.h"

int flurb;

void print() {
        int blaha;
        printf("blaha: %d flurb: %d\n", blaha, flurb);
}

int main() {
        printf("Starting\n");
        print();
        return 0;
}
[> make hello
cc -O2 -pipe    -o hello hello.c
[> ./hello
Starting
blaha: 1114820608 flurb: 0
[> ./hello
Starting
blaha: 325099520 flurb: 0
[> ./hello
Starting
blaha: -1199149056 flurb: 0
>
```

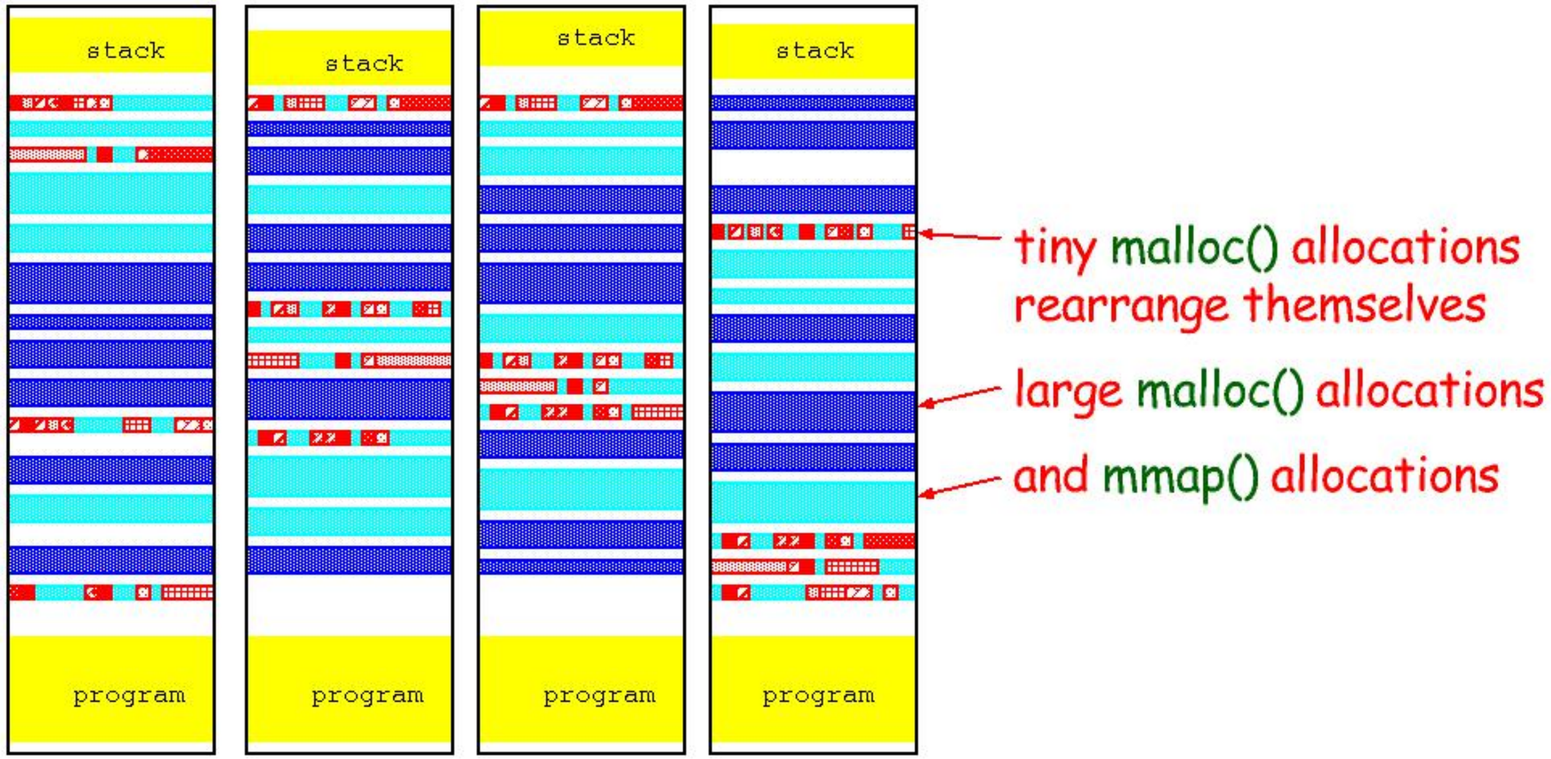
A Simple(tm) Program



A Simple(tm) Program

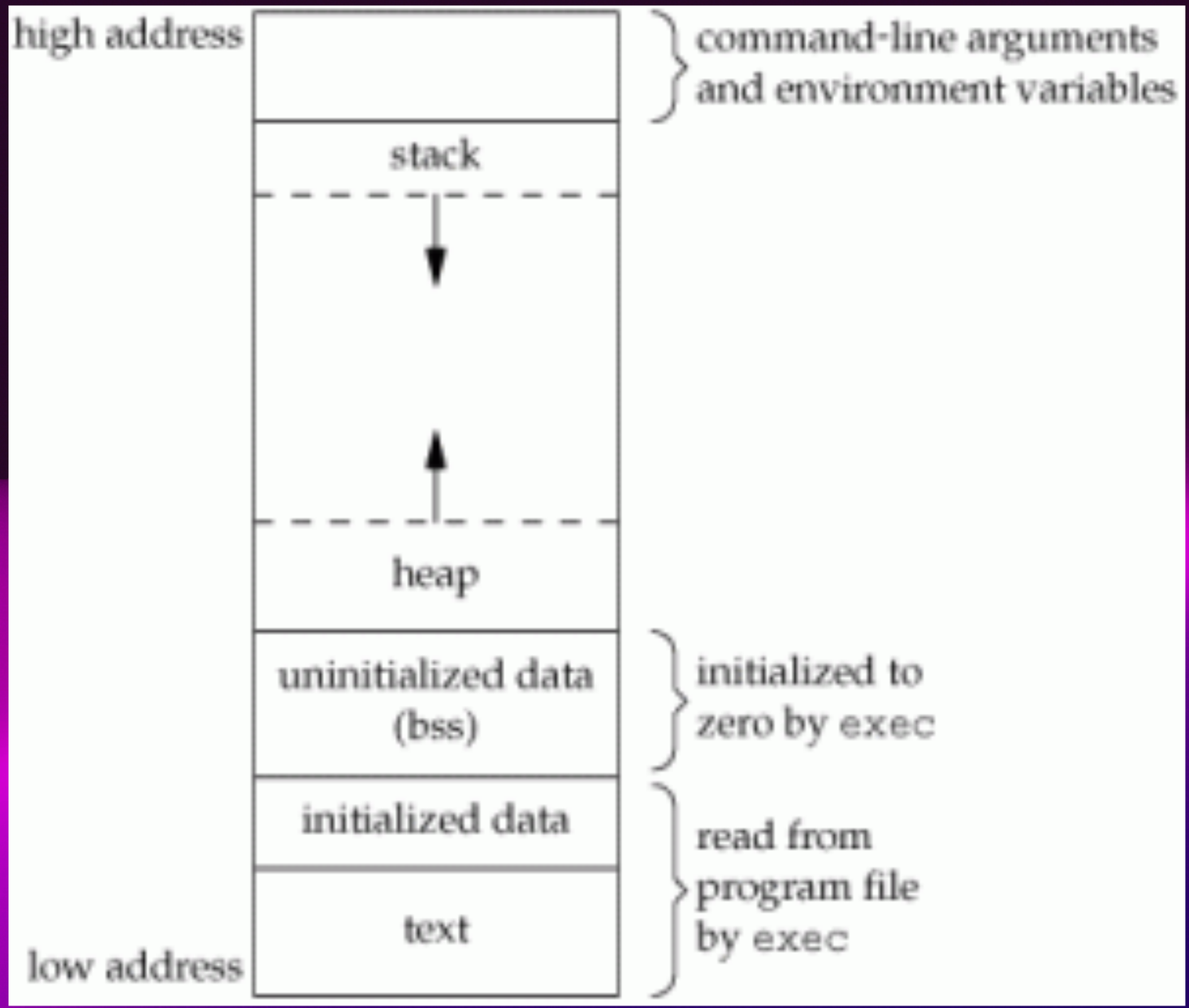
Randomized allocations..

Of course, each time you run the program the allocations change.



Note: Not showing the effects of many other changes, like shared library randomization, etc, etc

Loading into RAM



Loading into RAM

Loading into RAM

- Should be simple, set up pmap for new PID, open(“binary”); read(…); jump into main().

Loading into RAM

- Should be simple, set up pmap for new PID, open(“binary”); read(…); jump into main().
  - Get a lock on the program file

Loading into RAM

- Should be simple, set up pmap for new PID, open(“binary”); read(…); jump into main().
  - Get a lock on the program file
  - mmap() the code from file into pmap RAM.

Loading into RAM

- Should be simple, set up pmap for new PID, open(“binary”); read(…); jump into main().
  - Get a lock on the program file
  - mmap() the code from file into pmap RAM.
  - As CPU runs code, mmap:ed memory gets filled in from file content one page at a time

Loading into RAM

- Should be simple, set up pmap for new PID, open(“binary”); read(…); jump into main().
  - Get a lock on the program file
  - mmap() the code from file into pmap RAM.
  - As CPU runs code, mmap:ed memory gets filled in from file content one page at a time
  - Readonly Text section means forks and parallel instances can share code RAM

Loading into RAM

Loading into RAM

- Get a lock on the program file inode

Loading into RAM

- Get a lock on the program file inode
  - Means filesystem will not allow delete until last binary exits

Loading into RAM

- Get a lock on the program file inode
  - Means filesystem will not allow delete until last binary exits
  - rm will only remove directory entry; inode and contents still on disk

Loading into RAM

- Get a lock on the program file inode
  - Means filesystem will not allow delete until last binary exits
  - rm will only remove directory entry; inode and contents still on disk
- fstat(1) can list open files

Loading into RAM

- Get a lock on the program file inode
  - Means filesystem will not allow delete until last binary exits
  - rm will only remove directory entry; inode and contents still on disk
- fstat(1) can list open files
- Linux with /proc/<PID>/exe could perhaps recreate a new binary

Loading into RAM

```
jj       less      1087 text /usr        52358  -r-xr-xr-x     r   156432
jj       less      1087   wd /home       16996  drwxr-xr-x     r      512
jj       less      1087    0 pipe 0x0 state:
jj       less      1087    1 /           26303  crw--w----    rw    ttyp3
jj       less      1087    2 /           26303  crw--w----    rw    ttyp3
jj       fstat    24155 text /usr        52319  -r-xr-xr-x     r    23288
jj       fstat    24155   wd /home       16996  drwxr-xr-x     r      512
jj       fstat    24155    0 /           26303  crw--w----    rw    ttyp3
jj       fstat    24155    1 pipe 0x0 state:
jj       fstat    24155    2 /           26303  crw--w----    rw    ttyp3
jj       ksh      70405 text /           52956  -r-xr-xr-x     r   607112
jj       ksh      70405   wd /home       16996  drwxr-xr-x     r      512
jj       ksh      70405    1 /           26303  crw--w----    rw    ttyp3
jj       ksh      70405    2 /           26303  crw--w----    rw    ttyp3
jj       ksh      70405   10 /           26984  crw-rw-rw-  rwep     tty
jj       ksh      70405   11 /           26303  crw--w----   rwe    ttyp3
jj       sshd     15960 text /usr       208176  -r-xr-xr-x     r   964176
jj       sshd     15960   wd /               2  drwxr-xr-x     r     1024
jj       sshd     15960    0 /           26987  crw-rw-rw-    rw     null
jj       sshd     15960    1 /           26987  crw-rw-rw-    rw     null
jj       sshd     15960    2 /           26987  crw-rw-rw-    rw     null
```

The role of ld.so

The role of ld.so

- ld.so is “the interpreter” of all dynamically linked programs

The role of ld.so

- ld.so is “the interpreter” of all dynamically linked programs
- Just like #!/bin/bash is the interpreter for *.sh and #!/usr/bin/python3 for the *.py programs

The role of ld.so

- ld.so is “the interpreter” of all dynamically linked programs
- Just like #!/bin/bash is the interpreter for *.sh and #!/usr/bin/python3 for the *.py programs
- Only static binaries need no interpreter

The role of ld.so

- ld.so is “the interpreter” of all dynamically linked programs
- Just like #!/bin/bash is the interpreter for *.sh and #!/usr/bin/python3 for the *.py programs
- Only static binaries need no interpreter
  - Some of the tasks done by ld.so are in crt0.o for static binaries

The role of ld.so

The role of ld.so

- ELF sections list required symbols (functions) and suggests names of libs to provide them

The role of ld.so

- ELF sections list required symbols (functions) and suggests names of libs to provide them
- Libc supplies some weak symbols so other libraries can override them; perhaps like zlib, could give transparent gunzip functionality to any program

The role of ld.so

- ELF sections list required symbols (functions) and suggests names of libs to provide them
- Libc supplies some weak symbols so other libraries can override them; perhaps like zlib, could give transparent gunzip functionality to any program
- LD_RUN_PATH, LD_PRELOAD, LD_LIBRARY_PATH overrides OS defaults

The role of ld.so

- ELF sections list required symbols (functions) and suggests names of libs to provide them
- Libc supplies some weak symbols so other libraries can override them; perhaps like zlib, could give transparent gunzip functionality to any program
- LD_RUN_PATH, LD_PRELOAD, LD_LIBRARY_PATH overrides OS defaults
- ldd(1) to test/show what would be loaded

The role of ld.so

- ldd “half-runs” the binary and lists what gets picked up by ld.so along the way

```
> ldd ./hello
./hello:
        Start              End                Type  Open Ref GrpRef Name
        000000a8661bc000 000000a8661c0000 exe   1    0   0      ./hello
        000000ab43d6d000 000000ab43e61000 rlib  0    1   0      /usr/lib/libc.so.96.1
        000000aad833a000 000000aad833a000 ld.so 0    1   0      /usr/libexec/ld.so
> ldd ./hello
./hello:
        Start              End                Type  Open Ref GrpRef Name
        00000aa8e46f7000 00000aa8e46fb000 exe   1    0   0      ./hello
        00000aab7a560000 00000aab7a654000 rlib  0    1   0      /usr/lib/libc.so.96.1
        00000aabcae98000 00000aabcae98000 ld.so 0    1   0      /usr/libexec/ld.so
> ldd ./hello
./hello:
        Start              End                Type  Open Ref GrpRef Name
        0000052f21adc000 0000052f21ae0000 exe   1    0   0      ./hello
        000005314c82d000 000005314c921000 rlib  0    1   0      /usr/lib/libc.so.96.1
        000005318e9fa000 000005318e9fa000 ld.so 0    1   0      /usr/libexec/ld.so
>
```

The role of ld.so

- ldd “half-runs” the binary and lists what gets picked up by ld.so along the way

```
> ldd ./hello
./hello:
        Start             End               Type  Open Ref GrpRef Name
        000000a8661bc000 000000a8661c0000 exe   1    0   0      ./hello
        000000ab43d6d000 000000ab43e61000 rlib  0    1   0      /usr/lib/libc.so.96.1
        000000aad833a000 000000aad833a000 ld.so 0    1   0      /usr/libexec/ld.so
> ldd ./hello
./hello:
        Start             End               Type  Open Ref GrpRef Name
        00000aa8e46f7000 00000aa8e46fb000 exe   1    0   0      ./hello
        00000aab7a560000 00000aab7a654000 rlib  0    1   0      /usr/lib/libc.so.96.1
        00000aabcae98000 00000aabcae98000 ld.so 0    1   0      /usr/libexec/ld.so
> ldd ./hello
./hello:
        Start             End               Type  Open Ref GrpRef Name
        0000052f21adc000 0000052f21ae0000 exe   1    0   0      ./hello
        000005314c82d000 000005314c921000 rlib  0    1   0      /usr/lib/libc.so.96.1
        000005318e9fa000 000005318e9fa000 ld.so 0    1   0      /usr/libexec/ld.so
>
```

The role of ld.so

The role of ld.so

- ldd was the first program in OpenBSD base to use execpromises from pledge(2)

The role of ld.so

- ldd was the first program in OpenBSD base to use execpromises from pledge(2)
- Having a very limited set for the half-run program ldd runs limits the security impact a lot

The role of ld.so

- ldd was the first program in OpenBSD base to use execpromises from pledge(2)
- Having a very limited set for the half-run program ldd runs limits the security impact a lot
- Probably the best case currently for execpromises

The role of ld.so

The role of ld.so

- ld.so will pick the highest numbered libabc.so it can find

The role of ld.so

- ld.so will pick the highest numbered libabc.so it can find
- Old libraries may actually not hurt

The role of ld.so

- ld.so will pick the highest numbered libabc.so it can find
- Old libraries may actually not hurt
- Your program should not hard-code minor versions

The role of ld.so

- ld.so will pick the highest numbered libabc.so it can find
- Old libraries may actually not hurt
- Your program should not hard-code minor versions
- /usr/local/lib/libabc.so -> libabc.so.12.3

The role of ld.so

- ld.so will pick the highest numbered libabc.so it can find
- Old libraries may actually not hurt
- Your program should not hard-code minor versions
- /usr/local/lib/libabc.so -> libabc.so.12.3
- libpng16.so.16 ; compile with -lpng16

The role of ld.so

The role of ld.so

- Slightly less common

The role of ld.so

- Slightly less common
  - Using handle=dlopen(“somelib”) to open a library during runtime of your binary

The role of ld.so

- Slightly less common
  - Using handle=dlopen(“somelib”) to open a library during runtime of your binary
  - Then call somefunc=dlsym(handle,”somefunc”) to get the pointer to a named function

The role of ld.so

- Slightly less common
  - Using handle=dlopen(“somelib”) to open a library during runtime of your binary
  - Then call somefunc=dlsym(handle,”somefunc”) to get the pointer to a named function
  - Call (*somefunc)(a, b); as usual

The role of ld.so

- Slightly less common
  - Using handle=dlopen(“somelib”) to open a library during runtime of your binary
  - Then call somefunc=dlsym(handle,”somefunc”) to get the pointer to a named function
  - Call (*somefunc)(a, b); as usual
  - For plugins and similar hot loadable code

The role of ld.so

The role of ld.so

- Cached results of ldconfig(8) at /var/run/ld.so.hints to quickly find correct library

The role of ld.so

- Cached results of ldconfig(8) at /var/run/ld.so.hints to quickly find correct library
- Generated at every boot

The role of ld.so

- Cached results of ldconfig(8) at /var/run/ld.so.hints to quickly find correct library
- Generated at every boot
- setuid programs only use ld.so.hints

The role of ld.so

- Cached results of ldconfig(8) at /var/run/ld.so.hints to quickly find correct library
- Generated at every boot
- setuid programs only use ld.so.hints
- Bad if hints file is empty/outdated

The role of ld.so

- Cached results of ldconfig(8) at /var/run/ld.so.hints to quickly find correct library
- Generated at every boot
- setuid programs only use ld.so.hints
- Bad if hints file is empty/outdated
- Double check ldconfig(8) manpage

The role of ld.so

The role of ld.so

- Shared libraries are ELF just like binaries, but without main()

The role of ld.so

- Shared libraries are ELF
  just like binaries, but
  without main()
- So alike, that programs
  can dlopen() other programs

The role of ld.so

- Shared libraries are ELF
  just like binaries, but
  without main()
- So alike, that programs
  can dlopen() other programs
- Some programs even
  dlopen() themselves!

The role of ld.so

- Shared libraries are ELF just like binaries, but without main()
- So alike, that programs can dlopen() other programs
- Some programs even dlopen() themselves!

The role of ld.so

The role of ld.so

- More stages while your program is starting and ending

The role of ld.so

- More stages while your program is starting and ending
  - atexit(3) might be commonly known

The role of ld.so

- More stages while your program is starting and ending
  - atexit(3) might be commonly known
  - Each call to atexit() registers a subroutine to call before real exit. They run in reverse order.

The role of ld.so

The role of ld.so

- More stages while your program is starting and ending

The role of ld.so

- More stages while your program is starting and ending
  - constructor and destructor stages.

The role of ld.so

- More stages while your program is starting and ending
  - constructor and destructor stages.
  - destructor runs after your atexit() calls

The role of ld.so

- More stages while your program is starting and ending
  - constructor and destructor stages.
  - destructor runs after your atexit() calls
  - void __attribute__ ((constructor)) myconstructor() { … }

The role of ld.so

The role of ld.so

- More stages while your program is starting and ending

The role of ld.so

- More stages while your program is starting and ending
  - There are also preinit, init and fini stages.

The role of ld.so

- More stages while your program is starting and ending
  - There are also preinit, init and fini stages.
  - In my obsd tests, only preinit worked, __init and __fini already defined in libc.

The role of ld.so

- More stages while your program is starting and ending
  - There are also preinit, init and fini stages.
  - In my obsd tests, only preinit worked, __init and __fini already defined in libc.
  - --allow-multiple-definition “solves” this

The role of ld.so

- More stages while your program is starting and ending
  - There are also preinit, init and fini stages.
  - In my obsd tests, only preinit worked, __init and __fini already defined in libc.
  - --allow-multiple-definition “solves” this
  - The final order of all hooks is:

The role of ld.so

The role of ld.so

- More stages while your program is starting and ending

The role of ld.so

- More stages while your program is starting and ending
  - preinit

The role of ld.so

- More stages while your program is starting and ending
  - preinit
  - constructor

The role of ld.so

- More stages while your program is starting and ending
  - preinit
  - constructor
  - init

The role of ld.so

- More stages while your program is starting and ending
  - preinit
  - constructor
  - init
  - main

The role of ld.so

- More stages while your program is starting and ending
  - preinit
  - constructor
  - init
  - main
  - atexit

The role of ld.so

- More stages while your program is starting and ending
  - preinit
  - constructor
  - init
  - main
  - atexit
  - fini

The role of ld.so

- More stages while your program is starting and ending
  - preinit
  - constructor
  - init
  - main
  - atexit
  - fini
  - destructor

The role of ld.so

The role of ld.so

- Even more stages while your program is starting and ending

The role of ld.so

- Even more stages while your program is starting and ending
  - constructor
    This one has 65536 possible levels

The role of ld.so

- Even more stages while your program is starting and ending
  - constructor
    This one has 65536 possible levels
  - Levels <=100 reserved for not-you

GOT / PLT

GOT / PLT

- GOT - Global Offset Table

GOT / PLT

- GOT - Global Offset Table
- A list of relative offsets into segments so relocated programs can find program-internal addresses

GOT / PLT

- GOT - Global Offset Table
- A list of relative offsets into segments so relocated programs can find program-internal addresses
- Needed by static binaries too

GOT / PLT

- GOT - Global Offset Table
- A list of relative offsets into segments so relocated programs can find program-internal addresses
- Needed by static binaries too
- How you divide BSS into the ints, longs, structs the linker placed there

GOT / PLT

GOT / PLT

- PLT - Program Link Table

GOT / PLT

- PLT - Program Link Table
- List of jumps to shared library symbols like printf() and puts()

GOT / PLT

- PLT - Program Link Table
- List of jumps to shared library symbols like printf() and puts()
- Often lazy resolving - Wacky code rewrite

GOT / PLT

- PLT - Program Link Table
- List of jumps to shared library symbols like printf() and puts()
- Often lazy resolving - Wacky code rewrite
- Requires some tricks due to code reentrancy

GOT / PLT

- PLT - Program Link Table
- List of jumps to shared library symbols like printf() and puts()
- Often lazy resolving - Wacky code rewrite
- Requires some tricks due to code reentrancy
- Threads, signal handlers and so on

GOT / PLT

GOT / PLT

- PLT - Program Link Table

GOT / PLT

- PLT - Program Link Table
- Each jump subsection contains NOPs around the actual call to have ld.so resolve the current symbol

GOT / PLT

- PLT - Program Link Table
- Each jump subsection contains NOPs around the actual call to have ld.so resolve the current symbol
- Filled in backwards, overwriting the jmp instruction last

GOT / PLT

- PLT - Program Link Table
- Each jump subsection contains NOPs around the actual call to have ld.so resolve the current symbol
- Filled in backwards, overwriting the jmp instruction last
- If we race, symbol resolves happen twice. Ok worst case

GOT / PLT

GOT / PLT

- PLT - Program Link Table

GOT / PLT

- PLT - Program Link Table
- Program calls printf() -> ld.so turns this into

GOT / PLT

- PLT - Program Link Table
- Program calls printf() -> ld.so turns this into
  - Call our own PLT Entry #2

GOT / PLT

- PLT - Program Link Table
- Program calls printf() -> ld.so turns this into
  - Call our own PLT Entry #2
  - First time, resolve entry #2 to pointer to where printf() is defined in libc GOT then call it

GOT / PLT

- PLT - Program Link Table
- Program calls printf() -> ld.so turns this into
  - Call our own PLT Entry #2
  - First time, resolve entry #2 to pointer to where printf() is defined in libc GOT then call it
  - Change PLT entry #2 to point to libc GOT for printf()

GOT / PLT

- PLT - Program Link Table
- Program calls printf() -> ld.so turns this into
  - Call our own PLT Entry #2
  - First time, resolve entry #2 to pointer to where printf() is defined in libc GOT then call it
  - Change PLT entry #2 to point to libc GOT for printf()
  - Next call(s) go directly from PLT#2 to libc-GOT

OpenBSD recent changes

OpenBSD recent changes

- kbind - ok not very recent

OpenBSD recent changes

- kbind - ok not very recent
- libc only syscalls

OpenBSD recent changes

- kbind - ok not very recent
- libc only syscalls
- mimmutable

OpenBSD recent changes

- kbind - ok not very recent
- libc only syscalls
- mimmutable
- pinsyscall

OpenBSD recent changes

- kbind - ok not very recent
- libc only syscalls
- mimmutable
- pinsyscall
- xonly code regions

OpenBSD recent changes

OpenBSD recent changes

- kbind(2)

OpenBSD recent changes

- kbind(2)
  - Allows for lazy symbol bindings

OpenBSD recent changes

- kbind(2)
  - Allows for lazy symbol bindings
  - OpenBSD would prefer non-lazy to keep PLT readonly, kbind(2) works around RO.

OpenBSD recent changes

- kbind(2)
  - Allows for lazy symbol bindings
  - OpenBSD would prefer non-lazy to keep PLT readonly, kbind(2) works around RO.
  - First use from ld.so registers location and a cookie, later uses must come from same place and with same cookie

OpenBSD recent changes

OpenBSD recent changes

- Syscalls via libc only

OpenBSD recent changes

- Syscalls via libc only
  - Kernel enforces syscalls from designated locations

OpenBSD recent changes

- Syscalls via libc only
  - Kernel enforces syscalls from designated locations
  - Previously, syscalls could not be made from writeable memory

OpenBSD recent changes

- Syscalls via libc only
  - Kernel enforces syscalls from designated locations
  - Previously, syscalls could not be made from writeable memory
  - Requires specific support from golang compiler

OpenBSD recent changes

OpenBSD recent changes

- mimmutable(2)

OpenBSD recent changes

- mimmutable(2)
  - Makes sure the memory range permissions can never change.

OpenBSD recent changes

- mimmutable(2)
  - Makes sure the memory range permissions can never change.
  - The memory may change, but not the permissions (RW -> RO exception possible)

OpenBSD recent changes

- mimmutable(2)
  - Makes sure the memory range permissions can never change.
  - The memory may change, but not the permissions (RW -> RO exception possible)
  - Lots of ELF sections in libc, ld.so and crt0.o becomes immutable by default

OpenBSD recent changes

OpenBSD recent changes

- pinsyscall(2)

OpenBSD recent changes

- pinsyscall(2)
  - Marks the only spot where the pinned syscalls can be made from - sigabort if wrong

OpenBSD recent changes

- pinsyscall(2)
  - Marks the only spot where the pinned syscalls can be made from - sigabort if wrong
  - Currently (7.3) only execve()

OpenBSD recent changes

OpenBSD recent changes

- Xonly code regions

OpenBSD recent changes

- Xonly code regions
  - With some trickery, one can check/detect reads made while executing -vs- reading code segment

OpenBSD recent changes

- Xonly code regions
  - With some trickery, one can check/detect reads made while executing -vs- reading code segment
  - Have to move data out from inline asm code

OpenBSD recent changes

- Xonly code regions
  - With some trickery, one can check/detect reads made while executing -vs- reading code segment
  - Have to move data out from inline asm code
  - Potential defense against Blind ROP

OpenSSH - CVE-2023-38408

OpenSSH - CVE-2023-38408

- Only works if you use ssh-agent and “ssh -A evil-host” intending to jump to dest-host instead of “ssh -J evil-host dest-host”

OpenSSH - CVE-2023-38408

- Only works if you use ssh-agent and “ssh -A evil-host” intending to jump to dest-host instead of “ssh -J evil-host dest-host”
- The “evil-host” gets to pretend that dest-host wants you to try huge amount of auth protocols

OpenSSH - CVE-2023-38408

- Only works if you use ssh-agent and “ssh -A evil-host” intending to jump to dest-host instead of “ssh -J evil-host dest-host”
- The “evil-host” gets to pretend that dest-host wants you to try huge amount of auth protocols
- Each protocol will mean ssh-agent uses dlopen() to load library appropriate for the auth tried

OpenSSH - CVE-2023-38408

- Only works if you use ssh-agent and “ssh -A evil-host” intending to jump to dest-host instead of “ssh -J evil-host dest-host”
- The “evil-host” gets to pretend that dest-host wants you to try huge amount of auth protocols
- Each protocol will mean ssh-agent uses dlopen() to load library appropriate for the auth tried
- Load bad libs -> *boom*

OpenSSH - CVE-2023-38408

- Only works if you use ssh-agent and “ssh -A evil-host” intending to jump to dest-host instead of “ssh -J evil-host dest-host”
- The “evil-host” gets to pretend that dest-host wants you to try huge amount of auth protocols
- Each protocol will mean ssh-agent uses dlopen() to load library appropriate for the auth tried
- Load bad libs -> *boom*
- Linux seemingly have tons of bad libs

ELF links

https://s3.inet6.se/links.html

Questions?