

# Background on Binary Analysis

Introduction to Security (184.783, 192.082)  
S&P Research Division

slides are adapted from “CyberChallenge.IT Software Security I” by Lorenzo Veronese

```

x0040095a]> pdf
;-- main:
(fcn) sym.main.133
; var int local_118h @ rbp-0x118
; var int local_110h @ rbp-0x110
; var int local_104h @ rbp-0x104
; var int local_100h @ rbp-0x100
; var int local_1h @ rbp-0x1
; DATA XREF from 0x0040090d (entry0)
0x0040095a 55 push rbp
0x0040095b 4889e5 mov rbp, rsp
0x0040095e 4881ec200100 sub rsp, 0x120
0x00400965 89bdfcfeffff mov dword [rbp - local_104h], edi
0x0040096b 4889b5f0fefff mov qword [rbp - local_110h], rsi
0x00400972 488995e8fefff mov qword [rbp - local_110h], rdx
0x00400979 b800000000 mov eax, 0
0x0040097e e863ffffff call sym.banner
0x00400983 bf170b4000 mov edi, str.Enter_Password_ ; "Enter Password: " @ 0x400b17
0x00400988 b800000000 mov eax, 0
0x0040098d 8b40ffff lea rax, [sym.imp.printf]
0x00400993 488b00ffff lea rax, [rbp - local_100h]
0x00400999 89c0 mov rsi, rax
0x0040099c bf280b4000 mov edi, str._255s_ ; "%255s" @ 0x400b20
0x004009a1 b800000000 mov eax, 0
0x004009a6 e825feffff call sym.imp.__isoc99_scanf
0x004009ab c645ffff mov byte [rbp - local_1h], 0
0x004009af 488d8500ffff lea rax, qword [rbp - local_100h]
0x004009b6 4889c7 mov rdi, rax
0x004009b9 e861ffffff call sym.checkPassword
0x004009be 84c0 test al, al
0x004009c0 740c je 0x4009ce
0x004009c2 bf2e0b4000 mov edi, str.Password_accepted_ ; "Password accepted!" @ 0x400b2e
0x004009c7 e8c4fdffff call sym.imp.puts
0x004009cc eb0a jmp 0x4009db
; JMP XREF from 0x004009c0 (sym.main)
0x004009ce bf410b4000 mov edi, str.Wrong_ ; "Wrong!" @ 0x400b41
0x004009d3 e8b8fdffff call sym.imp.puts
; JMP XREF from 0x004009cc (sym.main)

```

# Compilation

# From C Code to Executables

Compiling a C program is a multi-stage process composed by 4 steps

- **preprocessing**
- **compilation**
- **assembly**
- **linking**

see first 10 minutes of <https://www.youtube.com/watch?v=8PrOp9tOPvQ>

# Preprocessing

In the first phase, **preprocessor** commands (in C they start with '#') are interpreted

```
#include <stdio.h>
#define MESSAGE "Hello world!"
int main() {
    printf(MESSAGE);
    return 0;
}
```

gcc -E hello.c

```
# 2 "hello.c" 2
int main() {
    printf("Hello world!");
    return 0;
}
```

# Compilation

In the second phase, preprocessed code is translated into **assembly instructions**

```
#include <stdio.h>
#define MESSAGE "Hello world!"
int main() {
    printf(MESSAGE);
    return 0;
}
```

gcc -s hello.c

```
# 2 "hello.c" 2
# 5 "hello.c"
int main() {
    printf("Hello world!");
    return 0;
}
```

```
main:
.LFB0:
.cfi_startproc
pushq  %rbp
.cfi_def_cfa_offset 16
.cfi_offset 6, -16
movq  %rsp, %rbp
.cfi_def_cfa_register 6
movl  $.LC0, %edi
movl  $0, %eax
call  printf
movl  $0, %eax
popq  %rbp
.cfi_def_cfa 7, 8
ret
```

# Assembly

In the assembly phase assembly instructions are translated into **machine or object code**

```
#include <stdio.h>
#define MESSAGE "Hello world!"
int main() {
    printf(MESSAGE);
    return 0;
}
```

gcc -c hello.c

```
# 2 "hello.c" 2
# 5 "hello.c"
int main() {
    printf("Hello world!");
    return 0;
}
```

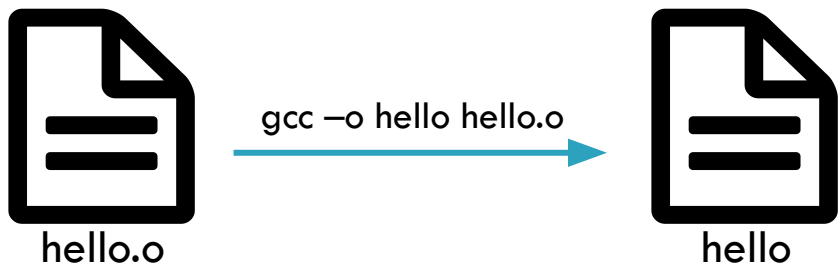
```
main:
.LFB0:
.cfi_startproc
pushq %rbp
.cfi_def_cfa_offset 16
.cfi_offset 6, -16
movq %rsp, %rbp
.cfi_def_cfa_register 6
movl $.LC0, %edi
movl $0, %eax
call printf
movl $0, %eax
popq %rbp
.cfi_def_cfa 7, 8
ret
```



hello.o

# Linking

- In the last phase (multiple) object files are combined in a single executable
- In the generated file references (links) to the used library are added.



Two approaches can be used in the linking phase

## Static Link

- Binaries are self-contained and do not depend on any external libraries

## Dynamic Link

- Binaries rely on system libraries that are loaded when needed
- Mechanisms are needed to dynamically relocate code

# Executable and Linkable Format

The Executable and Linkable Format (ELF) is a common file format for object files

There are three types of object files

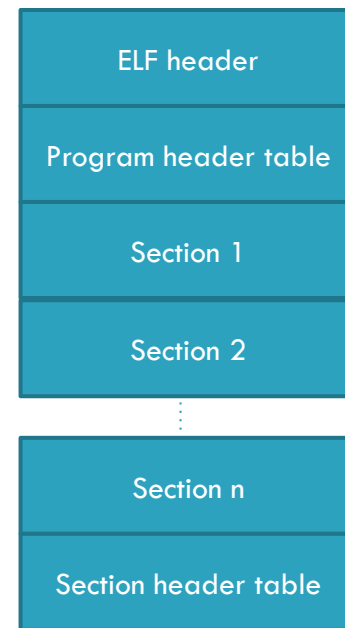
- **Relocatable file** containing code and data that can be linked with other object files to create an executable or a shared object file
- **Executable files** holding a program suitable for execution
- **Shared object files** that can be
  - linked with other relocatable and shared object files to obtain another object file
  - used by a **dynamic linker** together with other executable files and object files to create a **process image**



# Executable and Linkable Format

Any ELF file is composed by

- **ELF header** describing the file content
- **Program header table** providing informations on how to create a process image
- sequence of **Sections** containing what is needed for linking (instructions, data, symbol table, relocation information, ...)
- **Section header table** with a description of previous sections



# ELF: Relevant Sections

- .text** contains the executable instructions of a program
- .bss** contains uninitialised data that contribute to the program's memory image
- .data** contain initialized data that contribute to the program's memory image
- .data1**
- .rodata** are similar to .data and .data1, but refer to read only data
- .rodata1**
- .symtab** contains the program's symbol table
- .dynamic** provides linking information