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

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=8PrOp9t0PyQ

Preprocessing

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

```c
#include <stdio.h>

#define MESSAGE "Hello world!"

int main() {
    printf(MESSAGE);
    return 0;
}
```

gcc –E hello.c

```c
# 2 "hello.c" 2


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

Compilation

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

```c
#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**

```c
#include <stdio.h>

#define MESSAGE "Hello world!"

int main() {
    printf(MESSAGE);
    return 0;
}
```

gcc –c hello.c

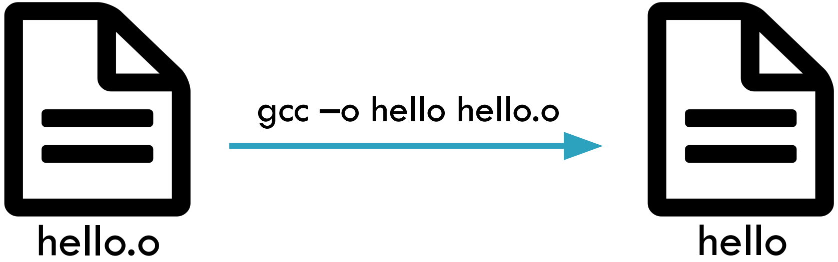
```c
# 2 "hello.c" 2


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

```asm
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

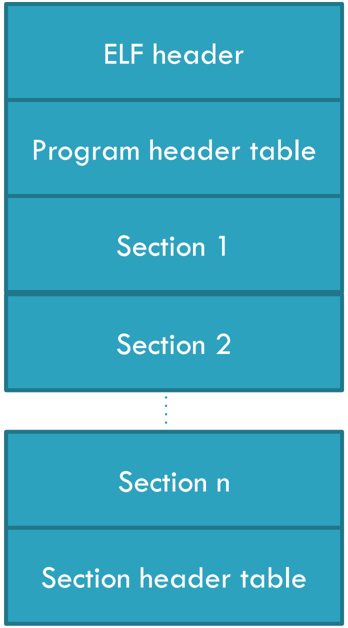
There are three types of object files

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

x86 Assembly Crash Course

The x86(-64) Assembly Language

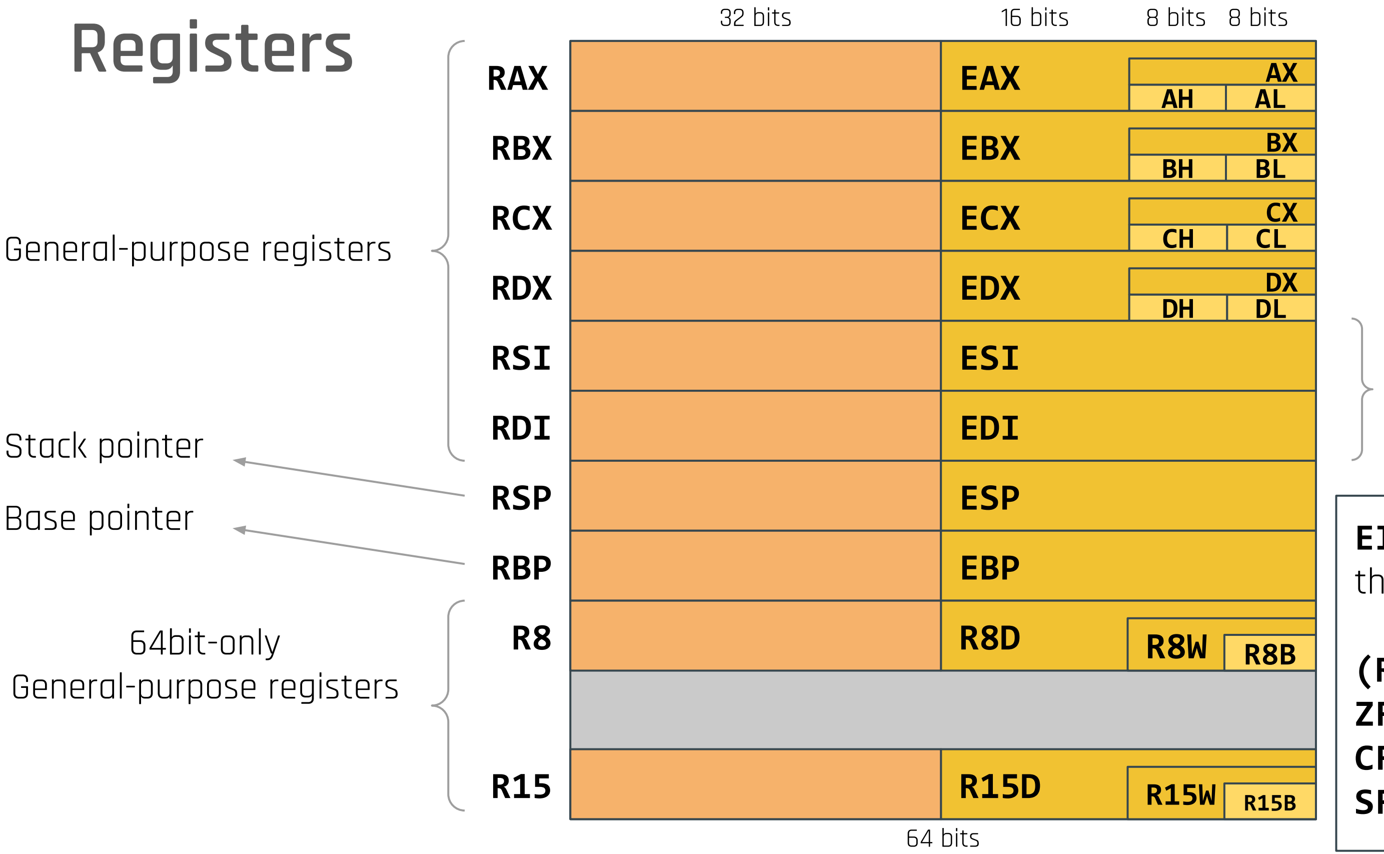
Assembly language makes machine code more readable

- depends on computer architecture (eg. x86 vs. ARM)
- subtle differences between 32- and 64-bit x86

Intel Syntax:	AT&T Syntax:
`command <destination>, <source>`	`command <source>, <destination>`
Example	**Example**
`mov eax, 5`	`mov $5, %eax`
more readable and explicit	default of GNU tools

see http://www.cs.virginia.edu/~evans/cs216/guides/x86.html

Registers



src and dst
Indexes (arrays, string copying and parameters)

EIP: Instruction pointer, points to the next instruction

(R/E)FLAGS: status flags
ZF - zero flag, when result is zero
CF - carry flag, result too large
SF - sign flag, result is negative

Memory Layout

A flat sequence of **bytes** identified by **addresses**

- 32-bit address `0xffffd728`
- 64-bit address `0x00007fffffffe5f0`

Types do not exists in memory! Bytes can be interpreted in different ways depending on our abstractions

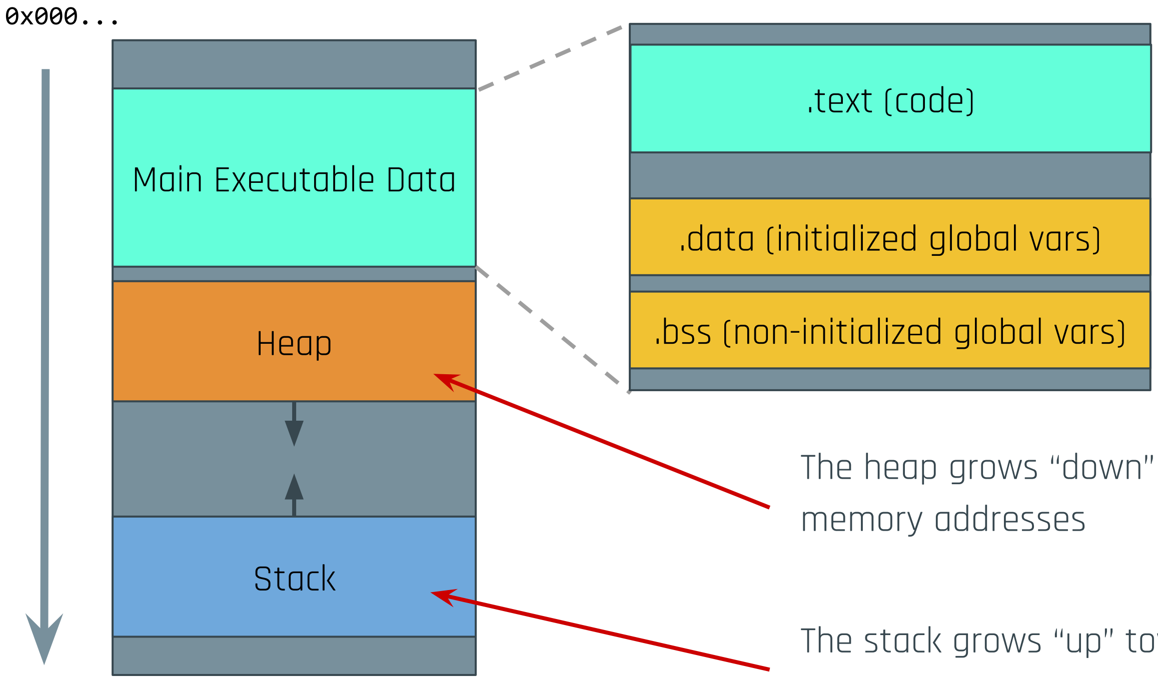
Integers (and pointers) are stored as little-endian words

- `78 56 34 12` ⟷ `0x12345678`

	+0	+1	+2	+3	+4	+5	+6	+7
+0	2f	6c	69	62	36	34	2f	6c
+8	64	2d	6c	69	6e	75	78	2d
+16	78	38	36	2d	36	34	2e	73
+24	6f	2e	32	00	04	00	00	00
+32	00	00	00	00	00	00	00	00
+40	00	00	00	00	00	00	00	00
+48	00	00	00	00	00	00	00	00
+56	00	00	00	00	00	00	00	00
+64	00	00	00	00	00	00	00	00

see https://en.wikipedia.org/wiki/Endianness

Memory Layout



The heap grows “down” towards high memory addresses

The stack grows “up” towards low memory addresses

Stack and Calling Convention

Stack: region of memory where local variables are stored

- Supports push and pop operations
- Grows towards lower memory addresses → pushed values have lower addresses
- When a function is called, a **stack frame** is set up
  - RBP/EBP contains the address of the base of the current stack frame
  - RSP/ESP contains the address of the top element of the stack

32 bits

Every function call pushes all its arguments to the stack

64 bits

Every function call stores the first 6 arguments in **RDI**, **RSI**, **RDX**, **RCX**, **R8**, **R9**, and pushes extra arguments on the stack

Return value in **EAX/RAX**

see https://www.youtube.com/watch?v=akCce7vSSfw

Instructions

`mov <dst>, <src>`

moves the <src> value to <dst>

`add <dst>, <src>`

adds the value in <src> to <dst>

`sub <dst>, <src>`

subtracts the value in <src> from <dst>

`and <dst>, <src>`

performs a **logical AND** between <src> and <dst>, placing the result in <dst>

`push <target>`

pushes the value in <target> to the stack

`pop <target>`

pops a value from the stack into <target>

`cmp <dst>, <src>`

compares <src> with <dst>. This is done by subtracting <src> from <dst> and updating flags that can be checked by subsequent conditional operations

Instructions

`call <address>`

calls the function at <address>. Before jumping to the function, the address of the next instruction is pushed to the stack in order to be able to return

`ret`

pops the return address and **returns** control to it

`leave`

restores the stack frame (rsp←rbp and old rbp is popped)

`jle <target>`

jumps to the address in <target> if the previously compared <src> was **less than or equal** to <dst>. The test is done on the flags set by `cmp`

`jge <target>`

jumps to the address in <target> if the previously compared <src> was **greater than or equal** to <dst>. The test is done on the flags set by `cmp`

Instructions

`jmp <target>`

jumps to the address in <target>. Copies target address into the RIP/EIP register

`lea <dst>, <src>`

stands for "load effective address": loads the address of <src> into <dst>

`int <value>`

generates software interrupt<value>. This is commonly used to invoke system calls

`nop`

no-operation, does nothing

NOTE multiple nops directly after each other are called a nop-slide or a nop-sled

Addressing modes

Register direct

```
mov eax, ebx
```

moves the content of ebx into eax

Register indirect

```
mov DWORD PTR [eax], ebx
```

moves the content of ebx into the memory location at the address in eax. used for array-addressing

NOTE DWORD PTR is like a type: indicates a 32-bit double word pointer

Immediate

```
mov eax, 3
```

move the value 3 into eax

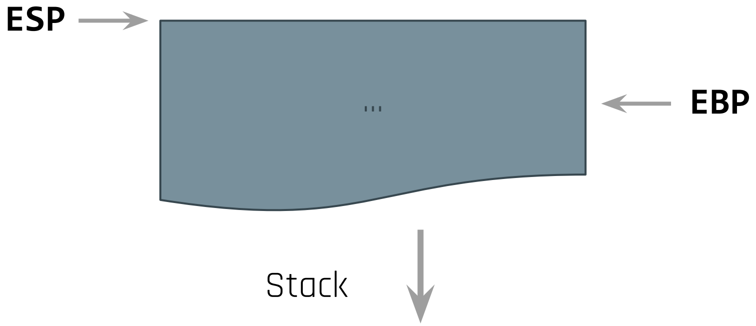
Memory Direct

```
mov eax, [0x1234]
```

move value at address 0x1234 into eax

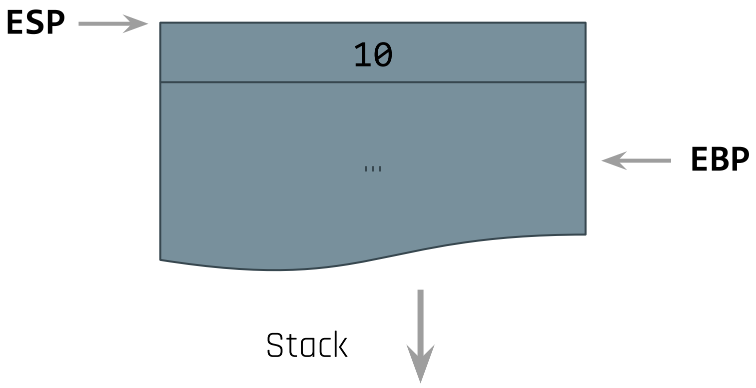
see https://en.wikipedia.org/wiki/Addressing_mode

```
func(10);          push 10
                   call func    /* push next inst. addr */
                                /* jmp func */
```



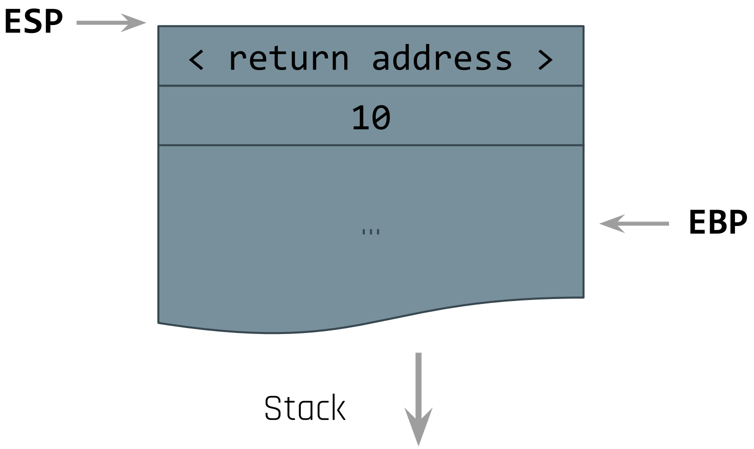
Function calls (32bit)

```
func(10);   push 10
            call func   /* push next inst. addr */
                        /* jmp func */
```



Function calls (32bit)

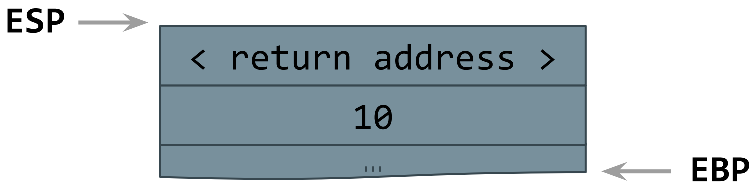
```
func(10);          push 10
                   call func    /* push next inst. addr */
                                /* jmp func */
```



Function calls (32bit)

```c
int func (int x) {
  int a = 0;
  int b = x;
  ...
}
```

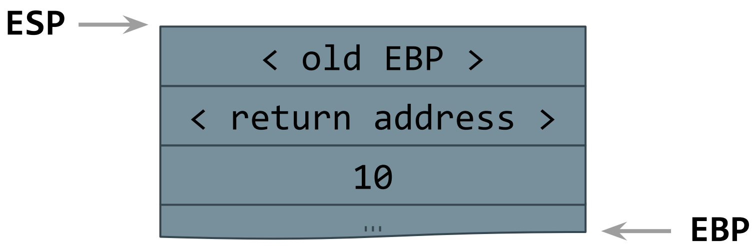
```asm
push ebp
mov ebp, esp
sub esp, 8
mov DWORD PTR [ebp - 4], 0
mov eax, DWORD PTR [ebp + 8]
mov DWORD PTR [ebp - 8], eax
...
```



Function calls (32bit)

```c
int func (int x) {
  int a = 0;
  int b = x;
  ...
}
```

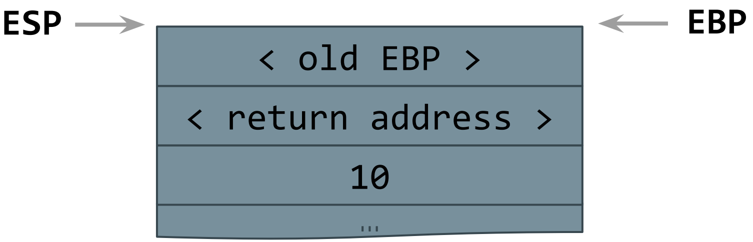
```asm
push ebp
mov ebp, esp
sub esp, 8
mov DWORD PTR [ebp - 4], 0
mov eax, DWORD PTR [ebp + 8]
mov DWORD PTR [ebp - 8], eax
...
```



Function calls (32bit)

```c
int func (int x) {
  int a = 0;
  int b = x;
  ...
}
```

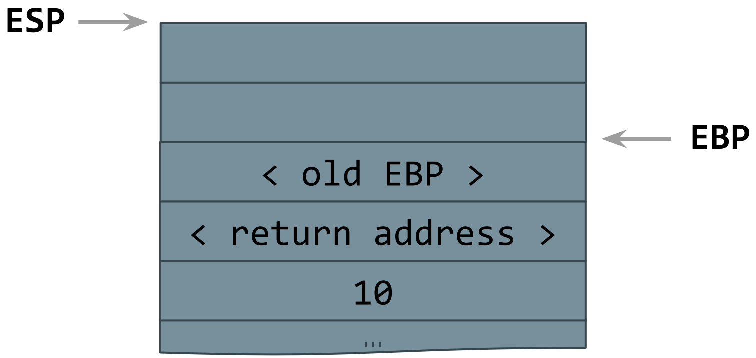
```asm
push ebp
mov ebp, esp
sub esp, 8
mov DWORD PTR [ebp - 4], 0
mov eax, DWORD PTR [ebp + 8]
mov DWORD PTR [ebp - 8], eax
...
```



Function calls (32bit)

```c
int func (int x) {
  int a = 0;
  int b = x;
  ...
}
```

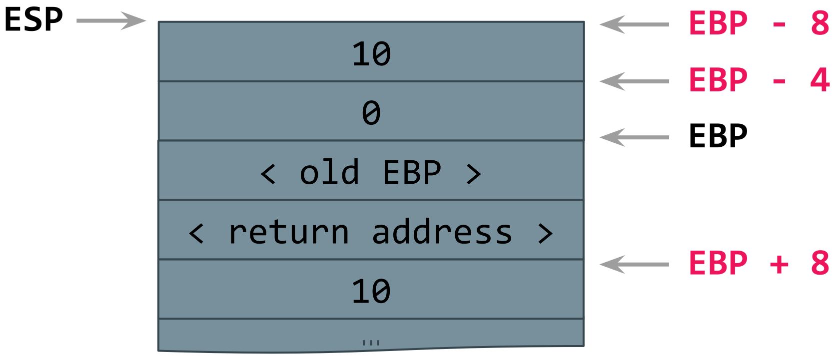
```asm
push ebp
mov ebp, esp
sub esp, 8
mov DWORD PTR [ebp - 4], 0
mov eax, DWORD PTR [ebp + 8]
mov DWORD PTR [ebp - 8], eax
...
```



Function calls (32bit)

```c
int func (int x) {
  int a = 0;
  int b = x;
  ...
}
```

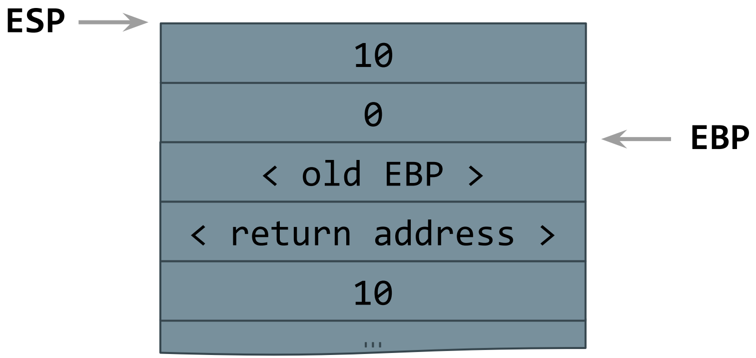
```asm
push ebp
mov ebp, esp
sub esp, 8
mov DWORD PTR [ebp - 4], 0
mov eax, DWORD PTR [ebp + 8]
mov DWORD PTR [ebp - 8], eax
...
```



Function calls (32bit)

```
int func (int x) {
  int a = 0;
  int b = x;
  ...

}
```

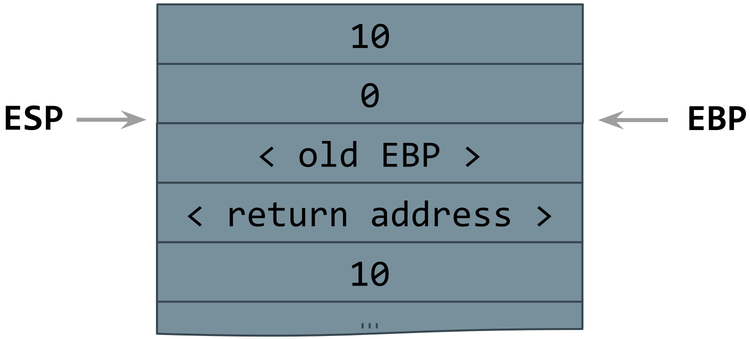
```
...
mov esp, ebp
pop ebp
ret
```



Function calls (32bit)

```c
int func (int x) {
  int a = 0;
  int b = x;
  ...

}
```

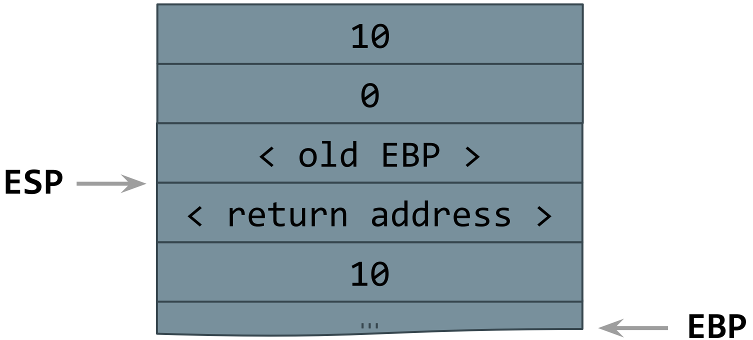
```asm
...
mov esp, ebp
pop ebp
ret
```



Function calls (32bit)

```
int func (int x) {
  int a = 0;
  int b = x;
  ...

}
```

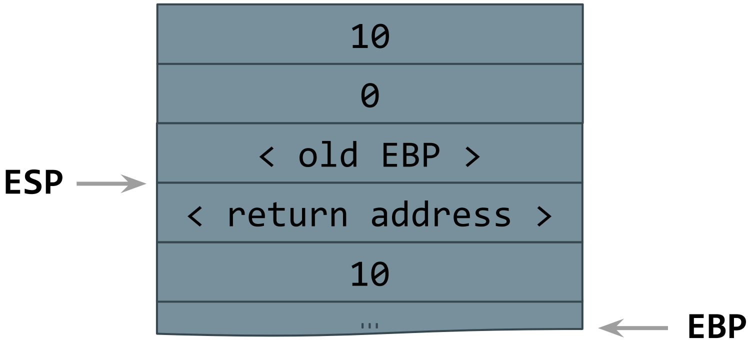
```
...
mov esp, ebp
pop ebp
ret
```



Function calls (32bit)

```c
int func (int x) {
  int a = 0;
  int b = x;
  ...

}
```

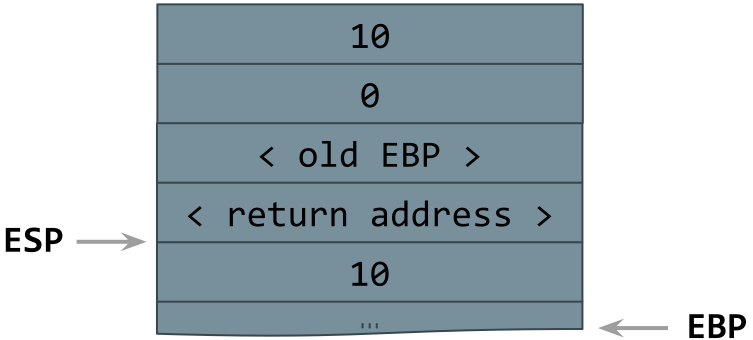
```
...
mov esp, ebp
pop ebp          /* or leave */
ret
```



Function calls (32bit)

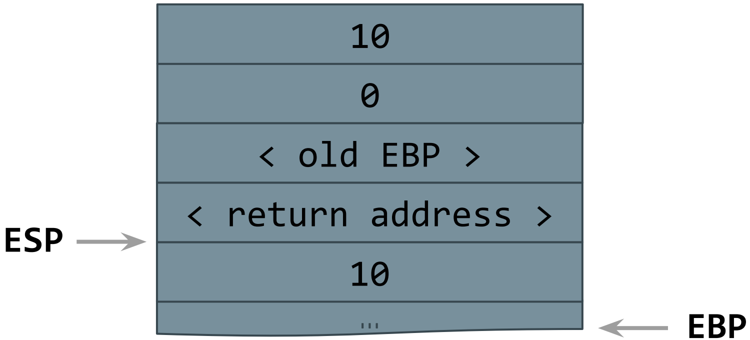
```c
int func (int x) {
  int a = 0;
  int b = x;
  ...

}
```

```
...
mov esp, ebp
pop ebp          /* or leave */
ret
```



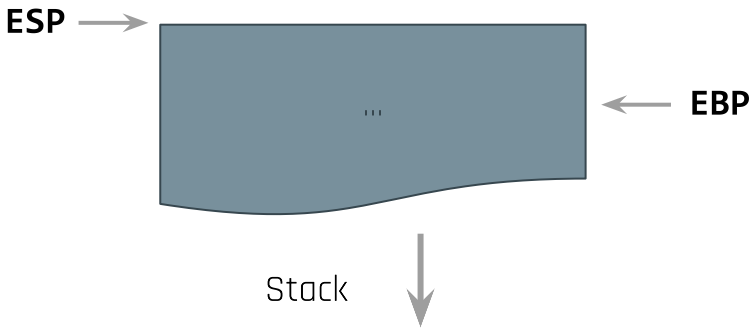
Function calls (32bit)

```
func(10);        push 10
                 call func    /* push next inst. addr */
                 ...          /* jmp func */
```



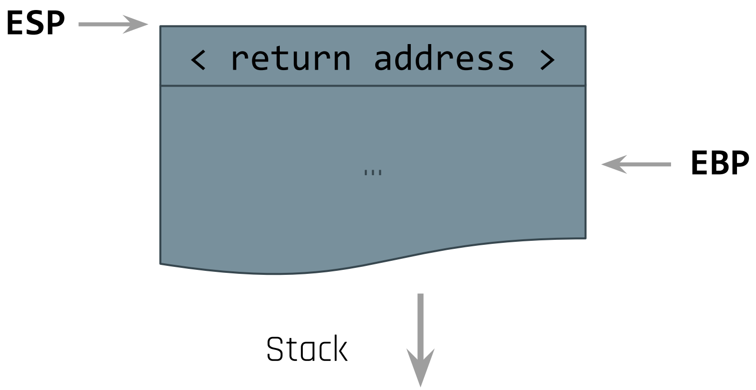
Function calls (32bit)

```
func(10);      mov rdi, 10
               call func    /* push next inst. addr */
                            /* jmp func */
```



Function calls (64bit)

```
func(10);           mov rdi, 10
                    call func    /* push next inst. addr */
                                 /* jmp func */
```



Function calls (64bit)

Binary Analysis

Gathering Information from Binary Files

Several tools are available to extract information from an **ELF** file

- `objdump` Displays informations about object files
- `readelf` Displays informations about ELF files
- `strings` Displays strings and printable characters in a file
- `file` Determines file type and displays some general info
- `ldd` Displays shared object dependencies

These tools can be used to gather information from binaries **without executing them**: they **statically** inspect the structure of the file

Disassembly

```
$ objdump -d {{path_to_your_binary}}
0804843b <main>:
 804843b:    8d 4c 24 04     lea ecx,[esp+0x4]
 804843f:    83 e4 f0        and esp,0xfffffff0
 8048442:    ff 71 fc        push   DWORD PTR [ecx-0x4]
 8048445:    55              push   ebp
 ...
```

Addresses (may be relative / relocatable addresses)

The actual **machine code** as bytes. Note that commands may have **different lengths**

Assembly in x86 Intel Syntax

see https://tldr.ostera.io/objdump

Static vs. Dynamic Analysis

Programs can be analysed in two ways

- **Static analysis**
  by inspecting the assembly we try to understand the program logic (tools can infer the program control flow effectively)

- **Dynamic analysis**
  the program is run with **debuggers** (on virtual or real processors) to observe its dynamic behaviour (for example, malware executed in sandboxes)

Usually the two techniques complement each other

Several **dynamic analysis tools** are available

- **gdb** The GNU project debugger
- **strace** Trace system calls and signals
- **ltrace** Trace library calls

see https://wizardzines.com/zines/strace/

Example Time!

An example is located at `/challenges/secret_handshake/hi` on the testbed server

With everything you know so far, you can answer these questions *without executing the program itself:*

- Is it an 32 bit or 64 bit executable?
- What libraries are linked?
- What is the entry point address?
- What are the most likely messages it will print when you execute the program?

Disassembly

Given a binary file we can use a disassembler to extract from a binary file info about the executed code

- This can be done with **`objdump`**:

```
objdump -M intel -ds /exercises/secret_handshake/hi > hi.s
```

  Here we ask **`objdump`** to produce the assembly code (**`-d`**) and display sections (**`-s`**) in Intel syntax (**`-M intel`**) and put the result in the file **`hi.s`**

Disassembly

Scrolling through the main function disassembly in the disassembly of the `main` function we see calls to `<__x86.get_pc_thunk.bx>`, `<printf@plt>`, `<__isoc99_scanf@plt>`, `<puts@plt>` and `<handshake_ok>`

We graciously ignore the first one, **`printf`**, **`scanf`** and **`puts`** are all part of the standard library but **`handshake_ok`** is defined just above the main function

```
000012fe <handshake_ok>:
    12fe:    f3 0f 1e fb             endbr32
    1302:    55                      push   ebp
    1303:    89 e5                   mov    ebp,esp
    1305:    53                      push   ebx
    1306:    83 ec 04                sub    esp,0x4
    1309:    e8 f0 00 00 00          call   13fe <__x86.get_pc_thunk.ax>
    130e:    05 b6 2c 00 00          add    eax,0x2cb6
    1313:    8b 55 08                mov    edx,DWORD PTR [ebp+0x8]
    1316:    3b 55 0c                cmp    edx,DWORD PTR [ebp+0xc]
    1319:    75 1b                   jne    1336 <handshake_ok+0x38>
    131b:    83 ec 0c                sub    esp,0xc
    131e:    8d 90 74 e0 ff ff       lea    edx,[eax-0x1f8c]
    1324:    52                      push   edx
    1325:    89 c3                   mov    ebx,eax
    1327:    e8 c4 fd ff ff          call   10f0 <puts@plt>
    132c:    83 c4 10                add    esp,0x10
    132f:    b8 01 00 00 00          mov    eax,0x1
    1334:    eb 19                   jmp    134f <handshake_ok+0x51>
    1336:    83 ec 0c                sub    esp,0xc
    1339:    8d 90 a0 e0 ff ff       lea    edx,[eax-0x1f60]
    133f:    52                      push   edx
    1340:    89 c3                   mov    ebx,eax
    1342:    e8 a9 fd ff ff          call   10f0 <puts@plt>
    1347:    83 c4 10                add    esp,0x10
    134a:    b8 00 00 00 00          mov    eax,0x0
    134f:    8b 5d fc                mov    ebx,DWORD PTR [ebp-0x4]
    1352:    c9                      leave
    1353:    c3                      ret
```

Disassembly

function call setup

load an argument (from ebp+0x8) into edx

compare edx with another argument (at ebp+0xc)

jump if not equal, aka. decide on the result

```
000012fe <handshake_ok>:
    12fe:   f3 0f 1e fb             endbr32
    1302:   55                      push   ebp
    1303:   89 e5                   mov    ebp,esp
    1305:   53                      push   ebx
    1306:   83 ec 04                sub    esp,0x4
    1309:   e8 f0 00 00 00          call   13fe <__x86.get_pc_thunk.ax>
    130e:   05 b6 2c 00 00          add    eax,0x2cb6
    1313:   8b 55 08                mov    edx,DWORD PTR [ebp+0x8]
    1316:   3b 55 0c                cmp    edx,DWORD PTR [ebp+0xc]
    1319:   75 1b                   jne    1336 <handshake_ok+0x38>
    131b:   83 ec 0c                sub    esp,0xc
    131e:   8d 90 74 e0 ff ff       lea    edx,[eax-0x1f8c]
    1324:   52                      push   edx
    1325:   89 c3                   mov    ebx,eax
    1327:   e8 c4 fd ff ff          call   10f0 <puts@plt>
    132c:   83 c4 10                add    esp,0x10
    132f:   b8 01 00 00 00          mov    eax,0x1
    1334:   eb 19                   jmp    134f <handshake_ok+0x51>
    1336:   83 ec 0c                sub    esp,0xc
    1339:   8d 90 a0 e0 ff ff       lea    edx,[eax-0x1f60]
    133f:   52                      push   edx
    1340:   89 c3                   mov    ebx,eax
    1342:   e8 a9 fd ff ff          call   10f0 <puts@plt>
    1347:   83 c4 10                add    esp,0x10
    134a:   b8 00 00 00 00          mov    eax,0x0
    134f:   8b 5d fc                mov    ebx,DWORD PTR [ebp-0x4]
    1352:   c9                      leave
    1353:   c3                      ret
```

The arguments are compared, but what are they?

Disassembly

```
 13d3:    50                        push   eax
 13d4:    ff 75 f4                  push   DWORD PTR [ebp-0xc]
 13d7:    e8 22 ff ff ff            call   12fe <handshake_ok>
```

```
00001354 <main>:
    1354:   f3 0f 1e fb             endbr32
    1358:   8d 4c 24 04             lea    ecx,[esp+0x4]
    135c:   83 e4 f0                and    esp,0xfffffff0
    135f:   ff 71 fc                push   DWORD PTR [ecx-0x4]
    1362:   55                      push   ebp
    1363:   89 e5                   mov    ebp,esp
    1365:   53                      push   ebx
    1366:   51                      push   ecx
    1367:   83 ec 10                sub    esp,0x10
    136a:   e8 01 fe ff ff          call   1170 <__x86.get_pc_thunk.bx>
    136f:   81 c3 55 2c 00 00       add    ebx,0x2c55
    1375:   c7 45 f4 41 41 41 41    mov    DWORD PTR [ebp-0xc],0x41414141
    137c:   c7 45 ec 00 00 00 00    mov    DWORD PTR [ebp-0x14],0x0
```

Back to in `<main>`:

`eax` and `[ebp-0xc]` are passed to `<handshake_ok>`

...and `[epb-0xc]` is initialize with **0x**41414141!

Might this be the number to make the handshake work? Try it!

Debugging

Static analysis is cool, but sometimes it helps to mess with a program while it runs. Luckily, the binary `/challenges/secret_handshake/hi` has been compiled with debug symbols, making it easy to work with a debugger like `gdb`

- Load the binary into the debugger with:

  ```
  gdb /exercises/secret_handshake/hi
  ```

  After a header you are presented with a prompt, like the terminal

Debugging

commands are given to gdb like in a terminal, here's what we need for this challenge:

- help - displays the build-in help if you want to know more
- quit - exits gdb. the most important, best when you were successful!
- run - runs the loaded program as if you did from the command line
- break - sets a breakpoint at a function or address. when the program runs and the breakpoint is reached, the execution is frozen and you get the gdb interface again
- continue - resumes execution
- info - displays all kinds of info, most common is info registers to show the registers
- print - powerful command to display and format data
- set - to change settings of gdb or set values inside the program memory

Debugging

- gdb also can disassemble, but we skip to the fun bit
- set a breakpoint
- run the program
- input a number
- inspect values
- change value
- continue the execution

```
stud4@testbed:~$ gdb /exercises/secret handshake/hi
GNU gdb (Ubuntu 8.1-0ubuntu3.2) 8.1.0.20180409-git
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86 64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from /exercises/secret handshake/hi...done.
(gdb) break handshake ok
Breakpoint 1 at 0x672: file secret handshake.c, line 6.
(gdb) run
Starting program: /exercises/secret handshake/hi
Hey there, what's the secret number?: 1234

Breakpoint 1, handshake ok (secret_number=1094795585, input_number=1234)
    at secret handshake.c:6
6               if(secret_number == input_number) {
(gdb) print input_number
$1 = 1234
(gdb) set variable secret_number = 1234
(gdb) print secret_number
$2 = 1234
(gdb) continue
Continuing.
```

SPOILER ;)

Tips!

- gdb does not take over the permissions of the target program - if you try to use gdb to mess with the challenge programs it won't work like in this example ;)
- Start early! "Something that seems incomprehensible at 23:00 will probably still be baffling at 02:00, but less so the next morning."
- programs like cutter can help reversing complicated programs, but since they have a steep learning curve, we keep the challenges simple so they can be attacked with the tools mentioned.