CyberX

REVERSING

The human view

When you write programs in C (or any other compiled language) you tipically operate with:

- Keywords (if, else, while, …)
- Typed variables (int, float, double)
- Functions

You write the program, compile it to get an executable, then launch the executable

The computer view

- The **microprocessor** is a multipurpose, programmable device that accepts data as input, processes it according to *instructions* stored in its *memory*, and provides results as output. It has internal memory.
- Microprocessors operate on numbers and symbols represented in the binary numeral system. Instructions are commands to give to processor to perform some operation.
- We focus on Intel x86 Architecture
  - IA-32
  - X86_64

Microprocessors talks binary code

- It’s boring and error prone!
- Assembly = language that associates symbols (for humans) to binary sequences (for computers)

GCC Compiler

Example: gcc compiler

- is the C compiler included in the *GNU Compiler Collection*
- is available in the main operating systems
  - Standard compiler in all the UNIX/Linux distributions

Detailed documentation is available at the following link:

https://gcc.gnu.org/onlinedocs/

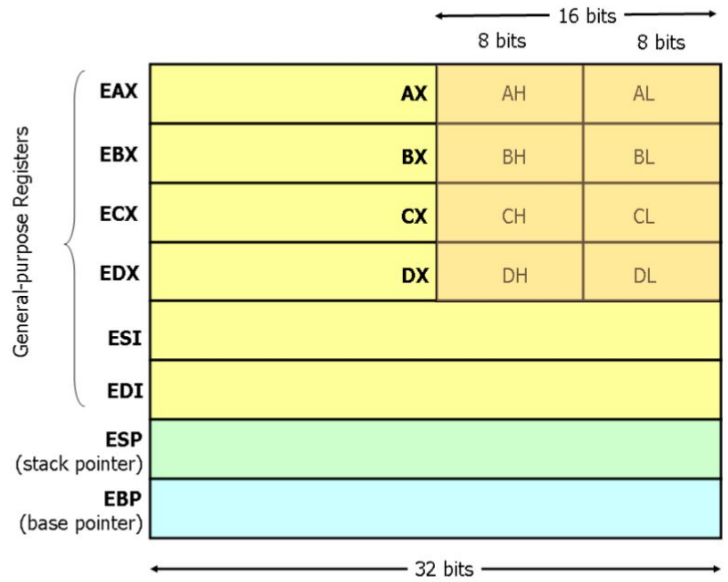
Static vs Dynamic Linking

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

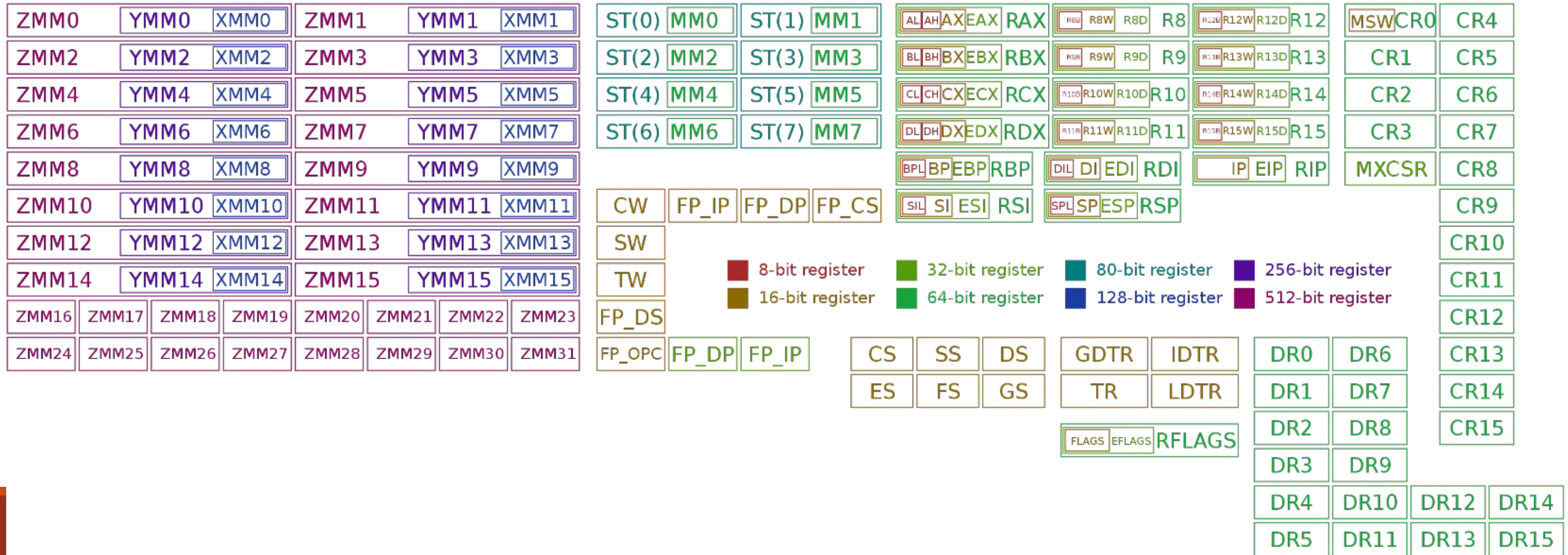
The computer view: registers

- Registers are **very fast** memory locations on board of the processor, intended to perform computations.
- Intel IA-32 architecture
  - 6 general purpose 32-bit registers (EAX, EBX, ECX, EDX, ESI, EDI)
  - 3 special 32-bit registers
    - ESP = stack pointer
    - EBP = base pointer
    - EIP = instruction pointer
      - Contains the address to the next instruction to be executed!



X86-64 Registers

X86-64 architecture provides a larger set of registers:



Syntax comparison

There are two types of syntax

- AT&T
- Intel

AT&T

- Register naming: **%eax**
- op src dest
- Operand suffix for size
  - movl (%ebx),%eax
- Immediates: $42
- Memory access: offset%(reg)

Intel

- Register naming: **eax**
- op dst src
- No suffix, but long syntax (e.g. word move)
  - mov eax, dword ptr [ebx]
- Immediates: 42
- Memory access: [reg+offset]

https://www.imada.sdu.dk/~kslarsen/dm546/Material/IntelnATT.htm

ASM instructions

- Data movement
  - mov <dst>,<src> [direct memory-to-memory movements not allowed]
    - mov eax, dword ptr [ebx+ecx*4]
      - int32 eax = *(ebx+ecx*4)
  - push <what>
  - pop <where>
- Arithmetic
  - add,sub,inc,dec,imul,idiv,and,or,xor,…
- Flow
  - jmp <label/addr>
  - Jcondition <label/addr>
  - call <label/addr>
  - ret

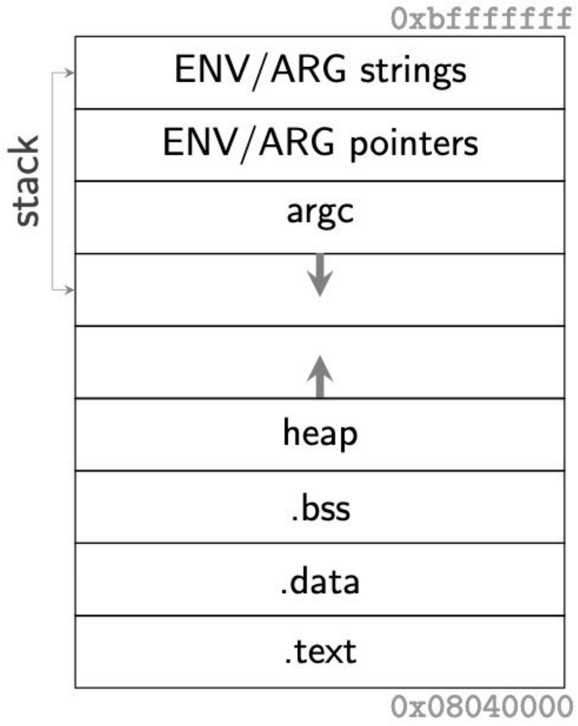
https://www.cs.virginia.edu/~evans/cs216/guides/x86.html

The memory

- Memory = (?) big array of bytes
- Each element of the array has an **address**
- 32-bit architecture 32-bit registers + 32-bit addresses
  - -> max memory that can be addressed = 2^{32} bytes = 4.294.967.296 bytes = 4 GigaBytes
- 64-bit architecture 64-bit registers + 64-bit addresses
  - max memory = a lot!!!

The memory

The memory of each **process** is mainly divided in 4 areas:



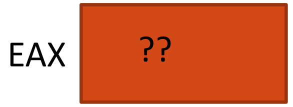
Text	Contains the code to be executed
Data	Global and static variables
Heap	Dynamic memory: this will be allocated on-demand by malloc()
Stack	Static memory: this memory is pre-allocated and its size is fixed (8M), is used for local variables and function calls

The memory - stack

The stack

- Data structure with two operations:
  - push add an element on top (decrease stack pointer ESP register)
  - pop remove the top element (increase stack pointer ESP register)
- Grows to small addresses

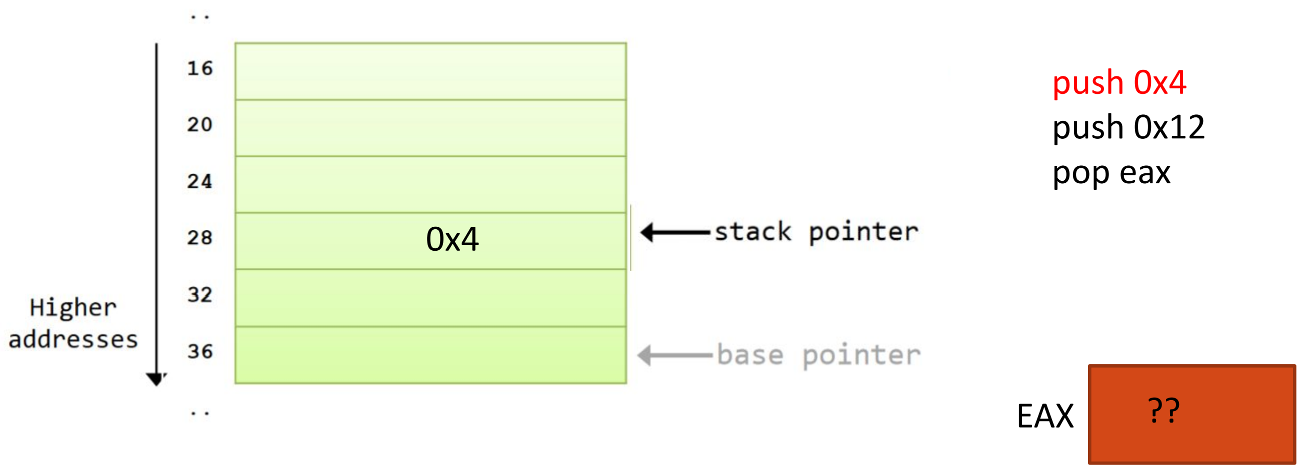
push 0x4
push 0x12
pop eax



The memory - stack

The stack

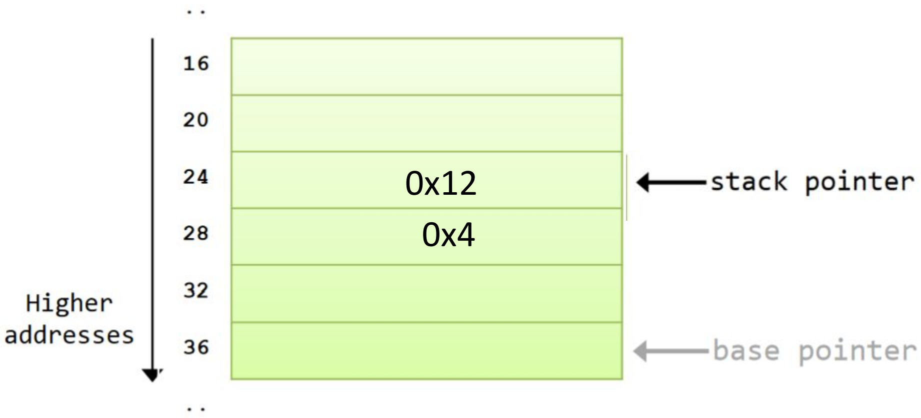
- Data structure with two operations:
  - push add an element on top (decrease stack pointer ESP register)
  - pop remove the top element (increase stack pointer ESP register)
- Grows to small addresses



The memory - stack

The stack

- Data structure with two operations:
  - push add an element on top (decrease stack pointer ESP register)
  - pop remove the top element (increase stack pointer ESP register)
- Grows to small addresses



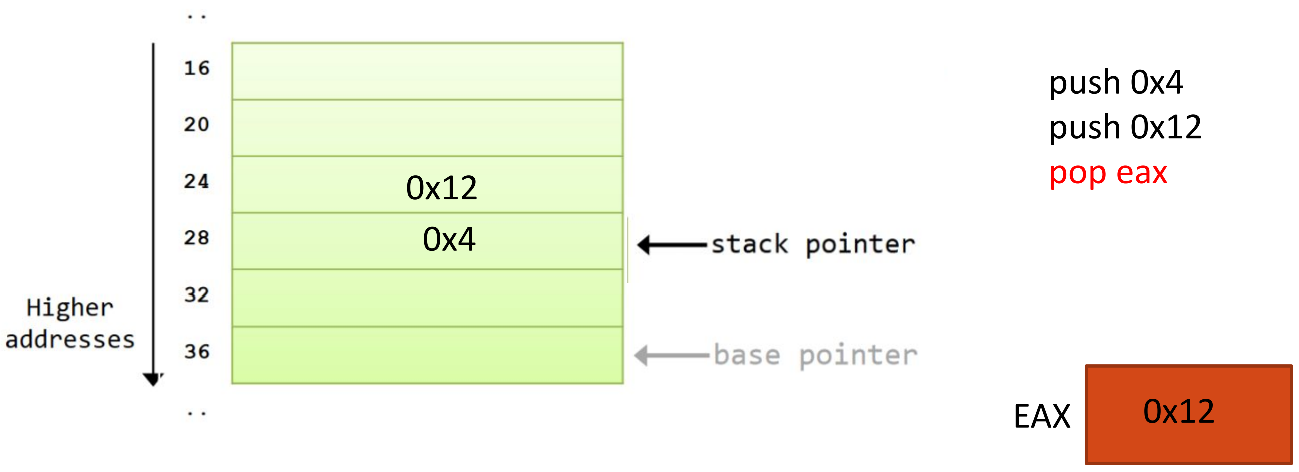
push 0x4
push 0x12
pop eax

EAX ??

The memory - stack

The stack

- Data structure with two operations:
  - push add an element on top (decrease stack pointer ESP register)
  - pop remove the top element (increase stack pointer ESP register)
- Grows to small addresses



Function Calling and Stack Frame

- Subroutines at the assembly level make contracts, known as **calling conventions**
- These contracts describe
  - How arguments are passed
  - Who is responsible to clean up the stack
  - Where to put return values
- The calling convention is part of the **ABI** (Application Binary Interface)
  - The *ABI* defines how machine code should behave when accessing data structures or subroutines
- Adhering to an ABI is usually the job of the compiler
  - gcc □ **cdecl** convention

Function calling

- **cdecl** convention
  - The **caller** pushes parameters on the stack in reverse order (right to left)
  - The **caller** cleans up the stack after the called function returns
  - The **callee**'s return value is put into the EAX register

Convention	Stack cleanup	Parameter passing
cdecl	Caller	Pushes parameters on the stack in reverse order (right to left)
fastcall	Callee	First two in registers the rest on the stack in reverse order
stdcall	Callee	Pushes parameters on the stack in reverse order
thiscall	Callee	First param in ECX (usually this) the rest on the stack in reverse order

Function Calling and Stack Frame

- Before transfering the control to the called function, the caller
  - prepares the parameters on the stack according to the convention
  - Saves the return address on the stack (i.e. the address of the next instruction to execute when the function returns)

- the **callee**’s code is surrounded by the so called function **prologue** and **epilogue**
  - Every function has its own **stack frame** (=own area in the stack, identified by EBP and ESP values) in order to
    - allocate local variables
    - do computations
    - re-transfer control to the caller
  - The stack frame of a function should never be accessed by another function
  - The stack frame is created/destroyed thanks to the prologue/epilogue

Function calling - steps

1. The caller pushes the parameters on the stack (via push)
2. The caller ”calls” the function and saves the return address on the stack
   - *call <func>* is equivalent to
     1. push <next instruction addr>
     2. jmp <func>
3. The callee executes the **prologue**
   1. *push EBP* (preserves stack frame of caller!)
   2. *mov ebp, esp* (EBP □ ESP, creates new stack frame)
5. The callee executes
6. The callee executes the **epilogue**
   1. *mov esp, ebp* (ESP □ EBP, recover caller stack frame)
   2. *pop EBP* (recover caller stack frame)
   3. *ret* = pop EIP (return to caller)

= *leave*

The heap

Memory allocation and de-allocation in the stack is very fast

- However, this memory cannot be used after a function returns

The heap is used to store dynamically allocated data that outlive function calls:

- This area is under programmer's responsibility

Memory management functions

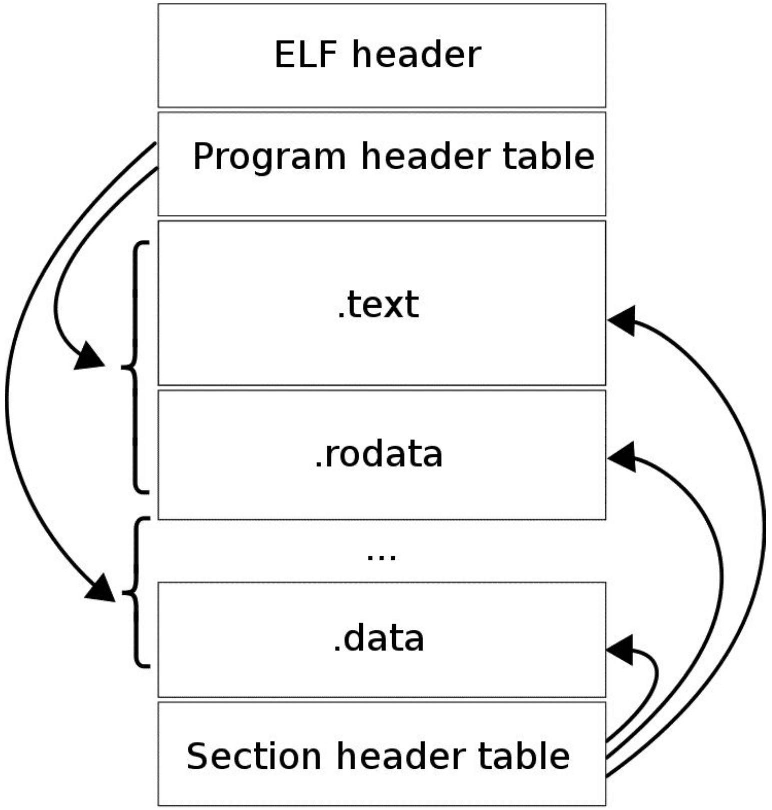
Basic *C* functions for memory management are:

- *malloc(int),* given an integer *n* allocates an area of *n* (continuous) byes and returns a pointer to that area
- *free(void*)*, deallocates the memory associated with a pointer

ELF

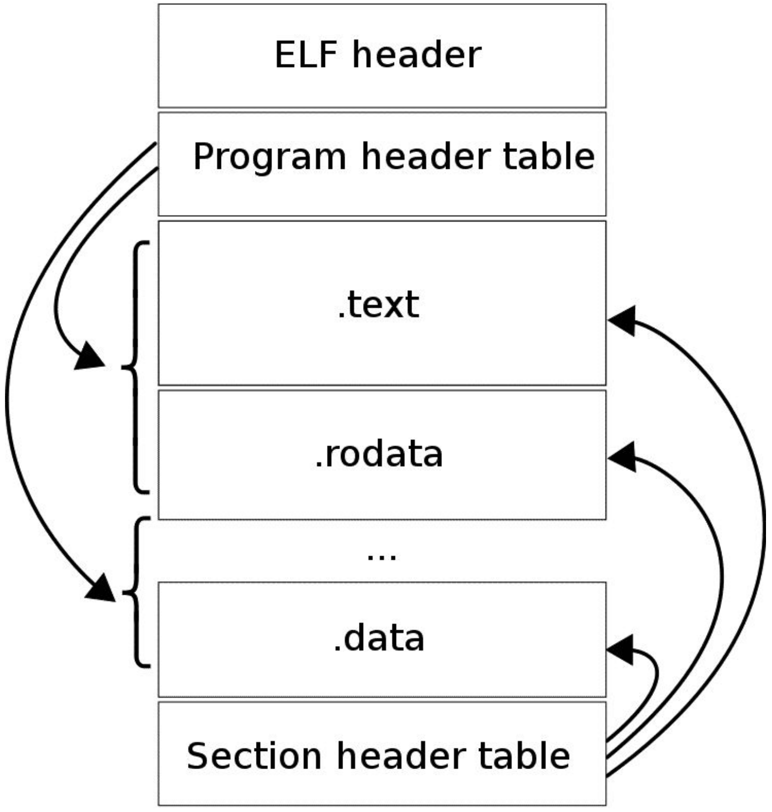
- ELF = Executable and Linkable Format
  - It’s the standard format for executable files under Linux
  - `man elf`

- ELF header: 32 bytes of general ELF information
  - ABI, CPU, elf class, entry point, …
  - `readelf -h elf_file`
- Program header: segments used at run-time
  - .text, .data, .rodata, .bss, …
  - `readelf -l elf_file`
- Section header: lists all sections
  - `readelf -S elf_file`
- Data



ELF Sections

- .text
  - Executable code [R-X]
- .data
  - Initialized data [RW-]
- .rodata
  - Initialized data [R--]
- .bss
  - Uninitialized data [RW-]
- …



Reversing a program

- Reversing = understand what the program does, without having its source code
  - launch the program, play with it, see how it behaves, …
  - somehow get the assembly code of the program, and reconstruct its behavior

- A program is usually reversed with 2 different analysis
  - Static analysis
  - Dynamic analysis

- Two are the main classes of tools to be used:
  - Disassemblers/Decompilers [e.g. r2, objdump, cutter, IDA, …]
    - For static analysis!
  - Debuggers [e.g. gdb, r2, IDA, …]
    - For dynamic analysys!

Static Analysis vs Dynamic Analysis

- *Static analysis* describes the process of analyzing the code or structure of a program to determine its function.
  - The program itself is not run at this time.
- *Dynamic analysis* is any examination performed after executing the program

Basic Static Analysis

- General ELF information

```
$ file elf_file
elf_file: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked, interpreter
/lib/ld-linux.so.2, BuildID[sha1]=3a19049c7e91345d6ce8d40dc0ae2d2a31516409, for GNU/Linux 3.2.0, not
stripped
```

- Strings

```
$ strings elf_file
td

td
  /
/lib/ld-linux.so.2
-*1Qd
libc.so.6
_IO_stdin_used
printf
__libc_start_main
```

Basic Static Analysis

- ELF header/sections analysis
  - `readelf`

- Disassemble (get ASM code)
  - `objdump`
    - Used to dump an object file (sequence of bytes)
  - `r2`
  - …

- After disassembling a binary file
  - Find functions
  - See how functions interact
  - …
  - Analyze behavior of the program

Static analysis - example

- We will analyze
  - the program behaviour
    - assuming we don’t know the source code
  - How function calling works

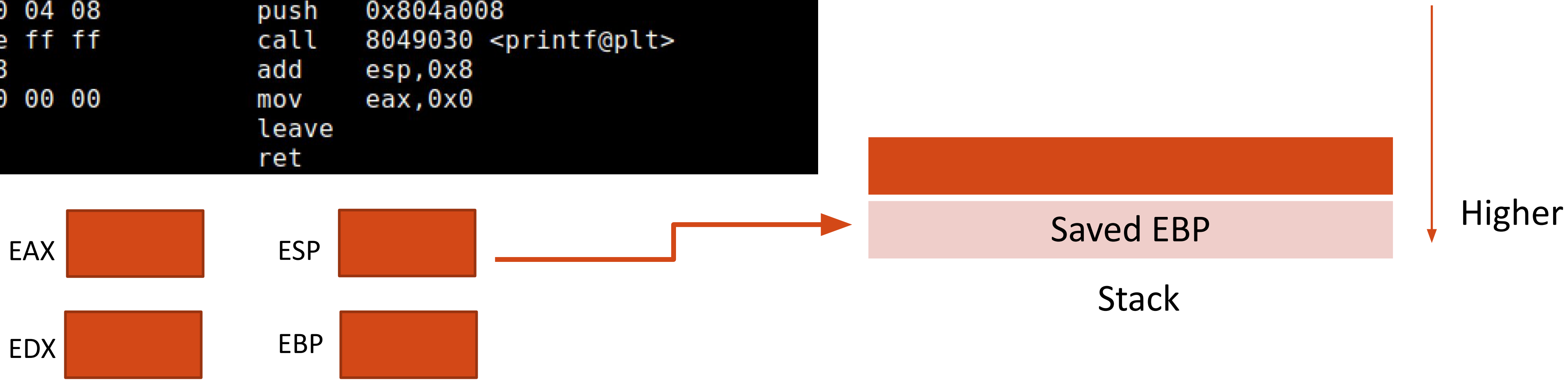
```c
#include <stdio.h>

int sum(int a, int b)
{
    int c = a+b;
    return c;
}

int main(int argc, char *argv[])
{
    int a = sum(2,3);
    printf("%d\n", a);
}
```

- Compile with:
  - *gcc -m32 -mpreferred-stack-boundary=2 -no-pie -fno-pic -O0 sum.c -o sum*
- Get asm with
  - *objdump -M intel -D ./sum*

Static analysis - example

```
08049178 <main>:
 8049178:       55                      push   ebp
 8049179:       89 e5                   mov    ebp,esp
 804917b:       83 ec 04                sub    esp,0x4
 804917e:       6a 03                   push   0x3
 8049180:       6a 02                   push   0x2
 8049182:       e8 db ff ff ff          call   8049162 <sum>
 8049187:       83 c4 08                add    esp,0x8
 804918a:       89 45 fc                mov    DWORD PTR [ebp-0x4],eax
 804918d:       ff 75 fc                push   DWORD PTR [ebp-0x4]
 8049190:       68 08 a0 04 08          push   0x804a008
 8049195:       e8 96 fe ff ff          call   8049030 <printf@plt>
 804919a:       83 c4 08                add    esp,0x8
 804919d:       b8 00 00 00 00          mov    eax,0x0
 80491a2:       c9                      leave
 80491a3:       c3                      ret
```

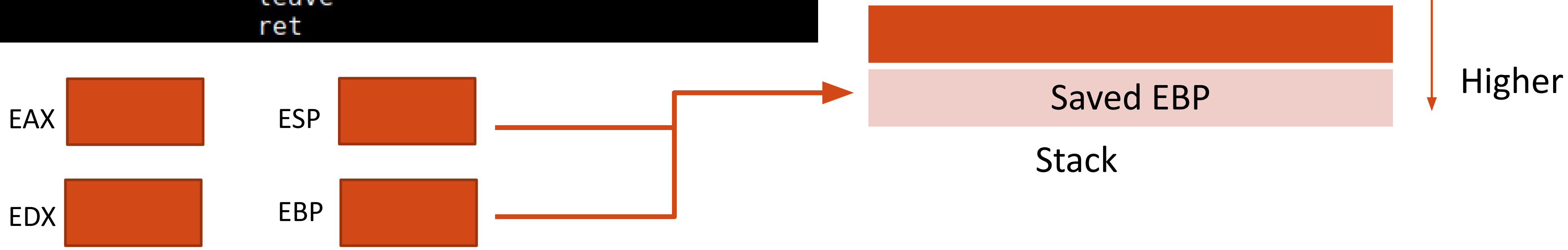
prologue



Static analysis - example

```
08049178 <main>:
 8049178:       55                      push   ebp
 8049179:       89 e5                   mov    ebp,esp
 804917b:       83 ec 04                sub    esp,0x4
 804917e:       6a 03                   push   0x3
 8049180:       6a 02                   push   0x2
 8049182:       e8 db ff ff ff          call   8049162 <sum>
 8049187:       83 c4 08                add    esp,0x8
 804918a:       89 45 fc                mov    DWORD PTR [ebp-0x4],eax
 804918d:       ff 75 fc                push   DWORD PTR [ebp-0x4]
 8049190:       68 08 a0 04 08          push   0x804a008
 8049195:       e8 96 fe ff ff          call   8049030 <printf@plt>
 804919a:       83 c4 08                add    esp,0x8
 804919d:       b8 00 00 00 00          mov    eax,0x0
 80491a2:       c9                      leave
 80491a3:       c3                      ret
```

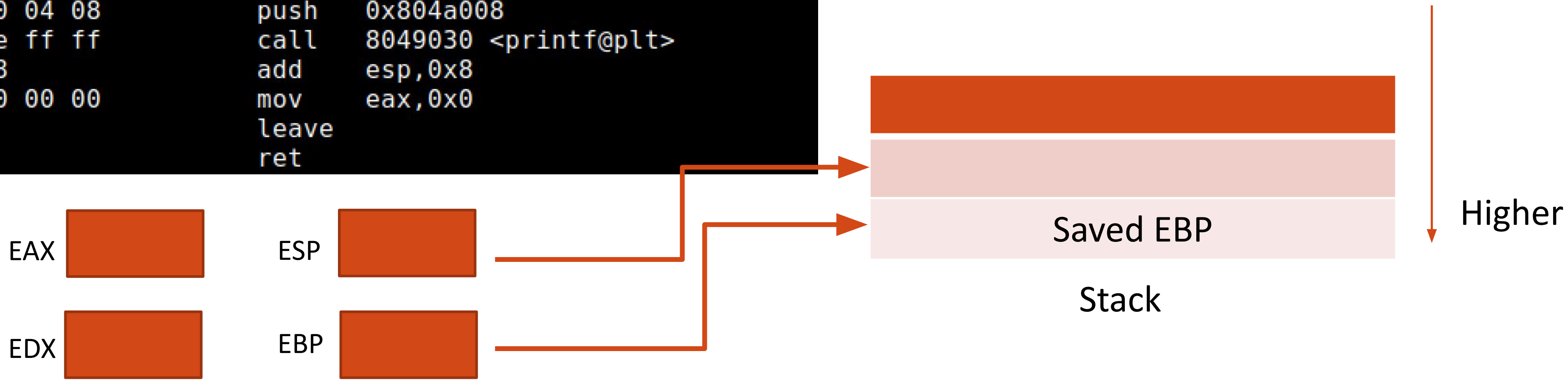
prologue



Static analysis - example

```
08049178 <main>:
 8049178:       55                      push   ebp
 8049179:       89 e5                   mov    ebp,esp
 804917b:       83 ec 04                sub    esp,0x4
 804917e:       6a 03                   push   0x3
 8049180:       6a 02                   push   0x2
 8049182:       e8 db ff ff ff          call   8049162 <sum>
 8049187:       83 c4 08                add    esp,0x8
 804918a:       89 45 fc                mov    DWORD PTR [ebp-0x4],eax
 804918d:       ff 75 fc                push   DWORD PTR [ebp-0x4]
 8049190:       68 08 a0 04 08          push   0x804a008
 8049195:       e8 96 fe ff ff          call   8049030 <printf@plt>
 804919a:       83 c4 08                add    esp,0x8
 804919d:       b8 00 00 00 00          mov    eax,0x0
 80491a2:       c9                      leave
 80491a3:       c3                      ret
```

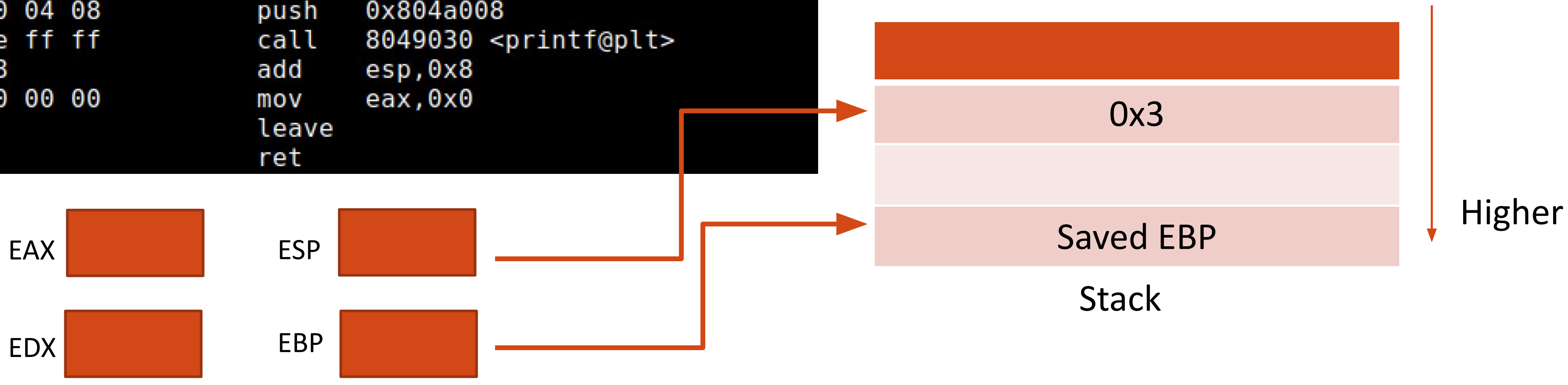
reserve 4 bytes



Static analysis - example

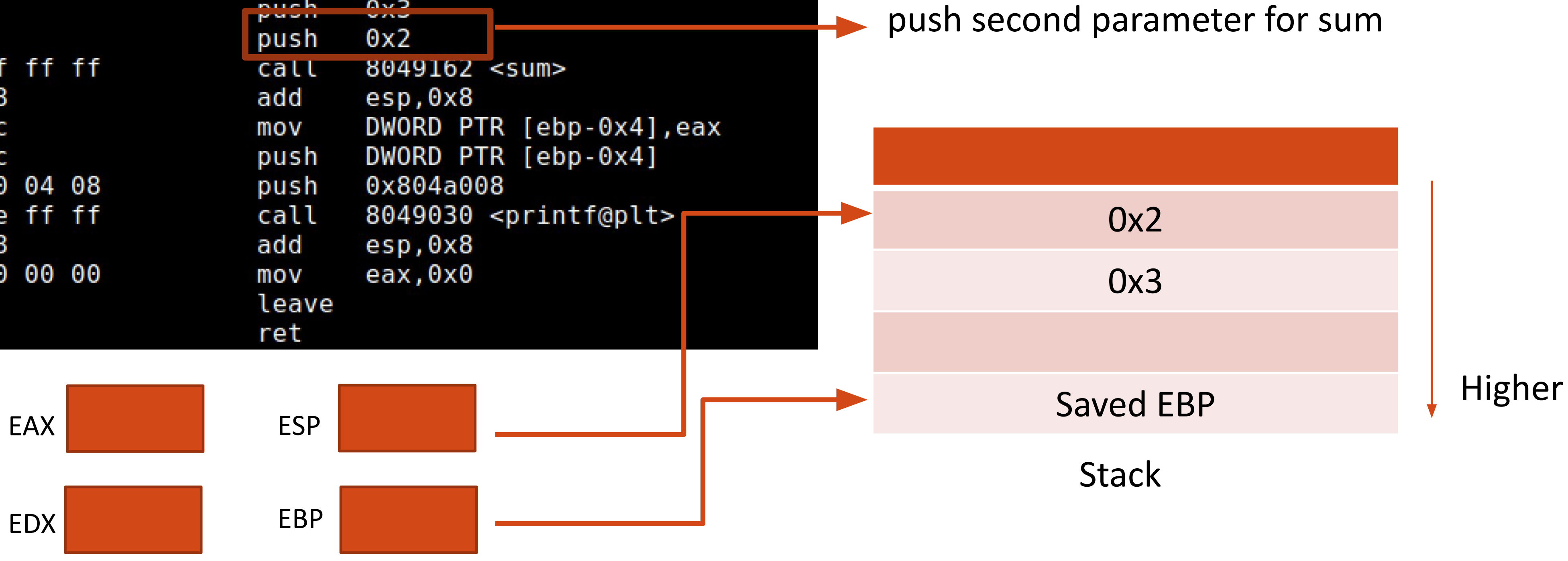
```
08049178 <main>:
 8049178:       55                      push   ebp
 8049179:       89 e5                   mov    ebp,esp
 804917b:       83 ec 04                sub    esp,0x4
 804917e:       6a 03                   push   0x3
 8049180:       6a 02                   push   0x2
 8049182:       e8 db ff ff ff          call   8049162 <sum>
 8049187:       83 c4 08                add    esp,0x8
 804918a:       89 45 fc                mov    DWORD PTR [ebp-0x4],eax
 804918d:       ff 75 fc                push   DWORD PTR [ebp-0x4]
 8049190:       68 08 a0 04 08          push   0x804a008
 8049195:       e8 96 fe ff ff          call   8049030 <printf@plt>
 804919a:       83 c4 08                add    esp,0x8
 804919d:       b8 00 00 00 00          mov    eax,0x0
 80491a2:       c9                      leave
 80491a3:       c3                      ret
```

push first parameter for sum



Static analysis - example

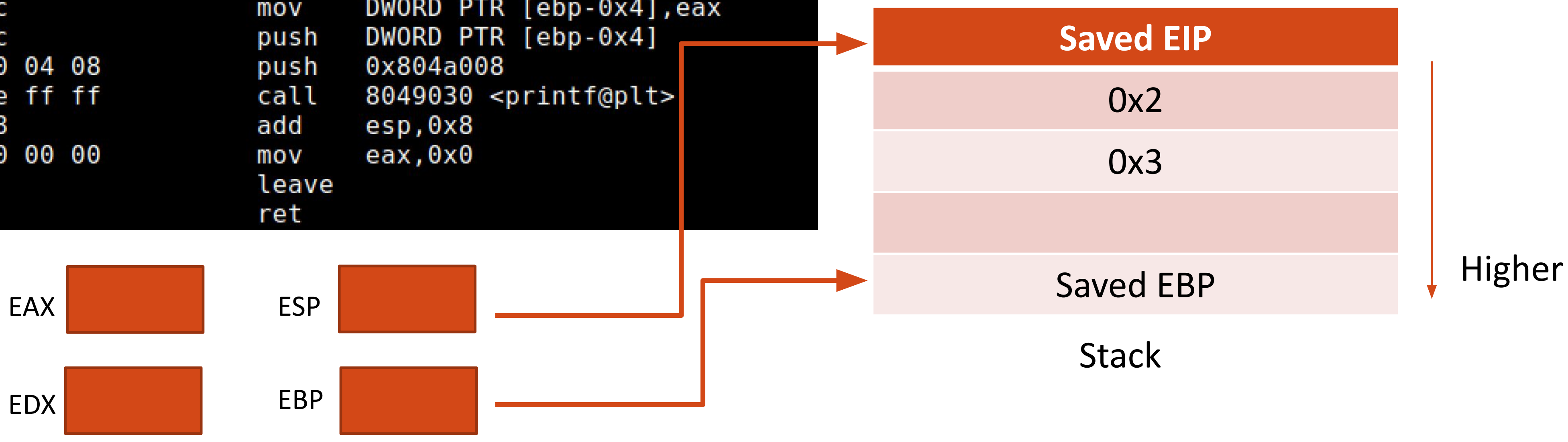
```
08049178 <main>:
 8049178:       55                      push   ebp
 8049179:       89 e5                   mov    ebp,esp
 804917b:       83 ec 04                sub    esp,0x4
 804917e:       6a 03                   push   0x3
 8049180:       6a 02                   push   0x2
 8049182:       e8 db ff ff ff          call   8049162 <sum>
 8049187:       83 c4 08                add    esp,0x8
 804918a:       89 45 fc                mov    DWORD PTR [ebp-0x4],eax
 804918d:       ff 75 fc                push   DWORD PTR [ebp-0x4]
 8049190:       68 08 a0 04 08          push   0x804a008
 8049195:       e8 96 fe ff ff          call   8049030 <printf@plt>
 804919a:       83 c4 08                add    esp,0x8
 804919d:       b8 00 00 00 00          mov    eax,0x0
 80491a2:       c9                      leave
 80491a3:       c3                      ret
```



Static analysis - example

```
08049178 <main>:
 8049178:       55                      push   ebp
 8049179:       89 e5                   mov    ebp,esp
 804917b:       83 ec 04                sub    esp,0x4
 804917e:       6a 03                   push   0x3
 8049180:       6a 02                   push   0x2
 8049182:       e8 db ff ff ff          call   8049162 <sum>
 8049187:       83 c4 08                add    esp,0x8
 804918a:       89 45 fc                mov    DWORD PTR [ebp-0x4],eax
 804918d:       ff 75 fc                push   DWORD PTR [ebp-0x4]
 8049190:       68 08 a0 04 08          push   0x804a008
 8049195:       e8 96 fe ff ff          call   8049030 <printf@plt>
 804919a:       83 c4 08                add    esp,0x8
 804919d:       b8 00 00 00 00          mov    eax,0x0
 80491a2:       c9                      leave
 80491a3:       c3                      ret
```

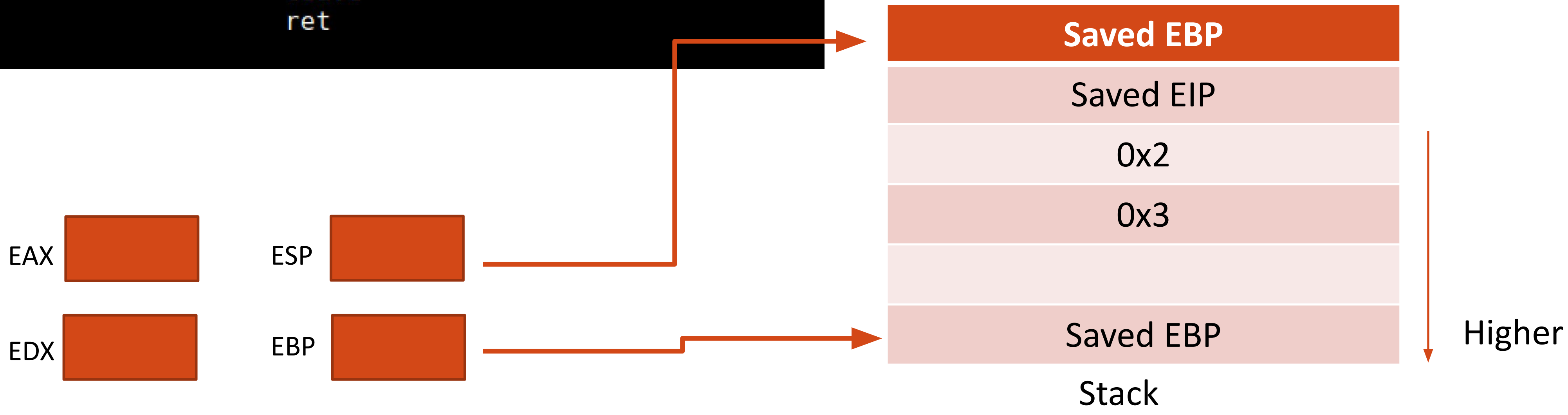
- save EIP
- transfer control



Static analysis - example

```
08049162 <sum>:
 8049162:       55                      push   ebp
 8049163:       89 e5                   mov    ebp,esp
 8049165:       83 ec 04                sub    esp,0x4
 8049168:       8b 55 08                mov    edx,DWORD PTR [ebp+0x8]
 804916b:       8b 45 0c                mov    eax,DWORD PTR [ebp+0xc]
 804916e:       01 d0                   add    eax,edx
 8049170:       89 45 fc                mov    DWORD PTR [ebp-0x4],eax
 8049173:       8b 45 fc                mov    eax,DWORD PTR [ebp-0x4]
 8049176:       c9                      leave
 8049177:       c3                      ret
```

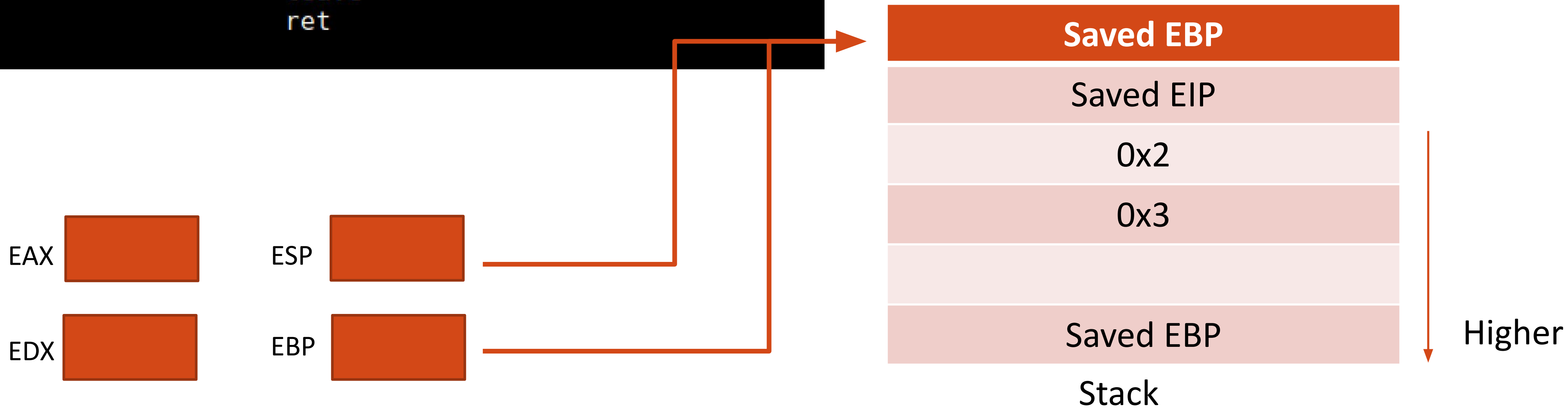
prologue



Static analysis - example

```
08049162 <sum>:
 8049162:       55                      push   ebp
 8049163:       89 e5                   mov    ebp,esp
 8049165:       83 ec 04                sub    esp,0x4
 8049168:       8b 55 08                mov    edx,DWORD PTR [ebp+0x8]
 804916b:       8b 45 0c                mov    eax,DWORD PTR [ebp+0xc]
 804916e:       01 d0                   add    eax,edx
 8049170:       89 45 fc                mov    DWORD PTR [ebp-0x4],eax
 8049173:       8b 45 fc                mov    eax,DWORD PTR [ebp-0x4]
 8049176:       c9                      leave
 8049177:       c3                      ret
```

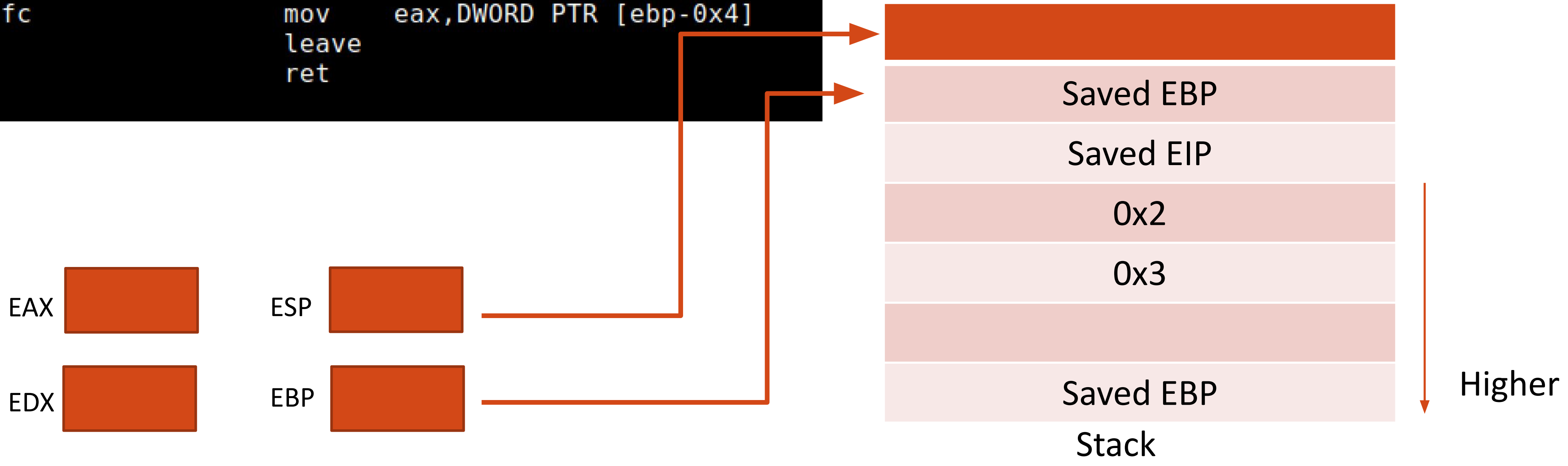
prologue



Static analysis - example

```
08049162 <sum>:
 8049162:       55                      push   ebp
 8049163:       89 e5                   mov    ebp,esp
 8049165:       83 ec 04                sub    esp,0x4
 8049168:       8b 55 08                mov    edx,DWORD PTR [ebp+0x8]
 804916b:       8b 45 0c                mov    eax,DWORD PTR [ebp+0xc]
 804916e:       01 d0                   add    eax,edx
 8049170:       89 45 fc                mov    DWORD PTR [ebp-0x4],eax
 8049173:       8b 45 fc                mov    eax,DWORD PTR [ebp-0x4]
 8049176:       c9                      leave
 8049177:       c3                      ret
```

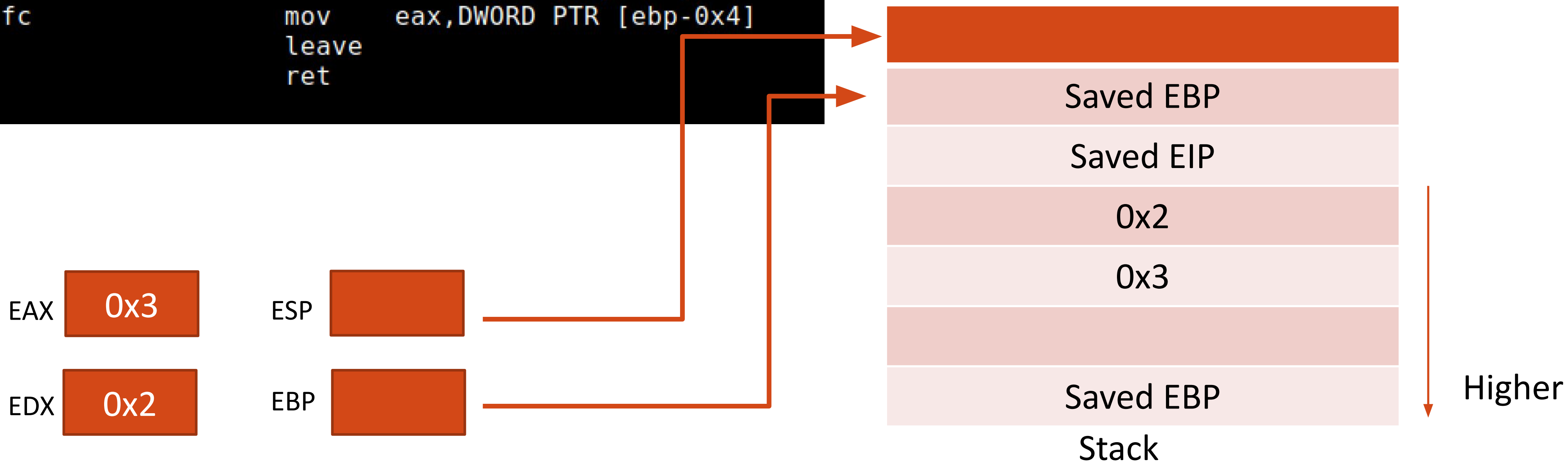
local var c



Static analysis - example

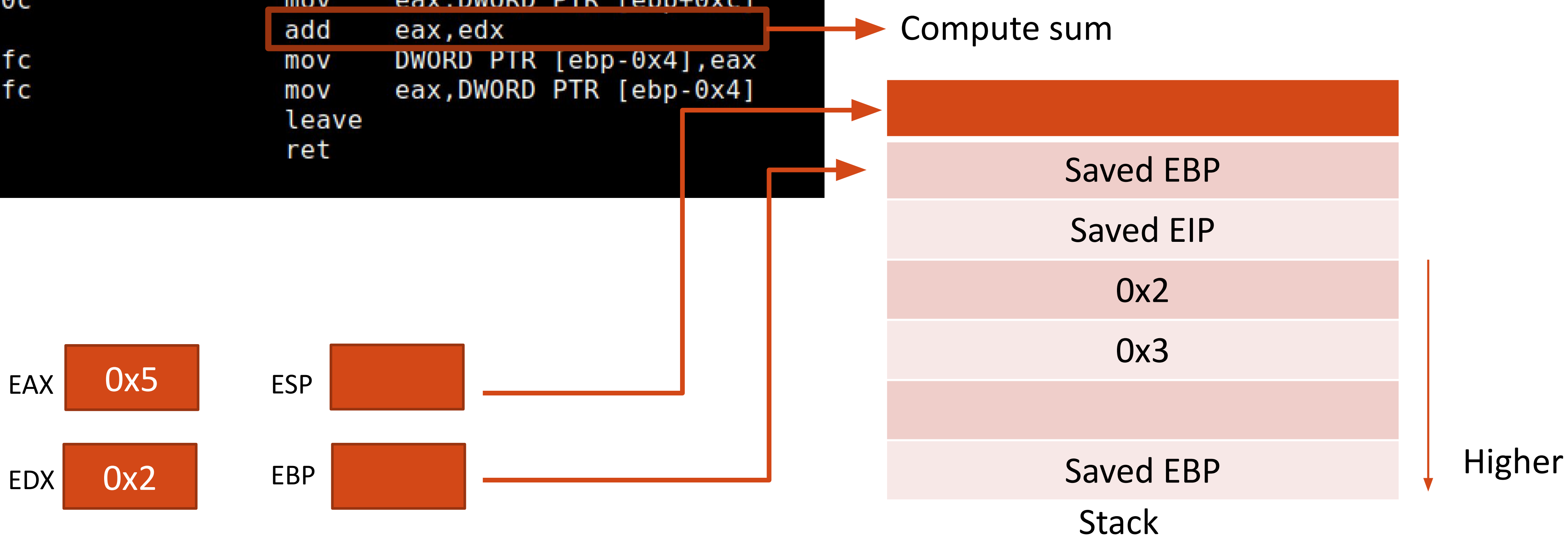
```
08049162 <sum>:
 8049162:       55                      push   ebp
 8049163:       89 e5                   mov    ebp,esp
 8049165:       83 ec 04                sub    esp,0x4
 8049168:       8b 55 08                mov    edx,DWORD PTR [ebp+0x8]
 804916b:       8b 45 0c                mov    eax,DWORD PTR [ebp+0xc]
 804916e:       01 d0                   add    eax,edx
 8049170:       89 45 fc                mov    DWORD PTR [ebp-0x4],eax
 8049173:       8b 45 fc                mov    eax,DWORD PTR [ebp-0x4]
 8049176:       c9                      leave
 8049177:       c3                      ret
```

Get parameters (32bit)



Static analysis - example

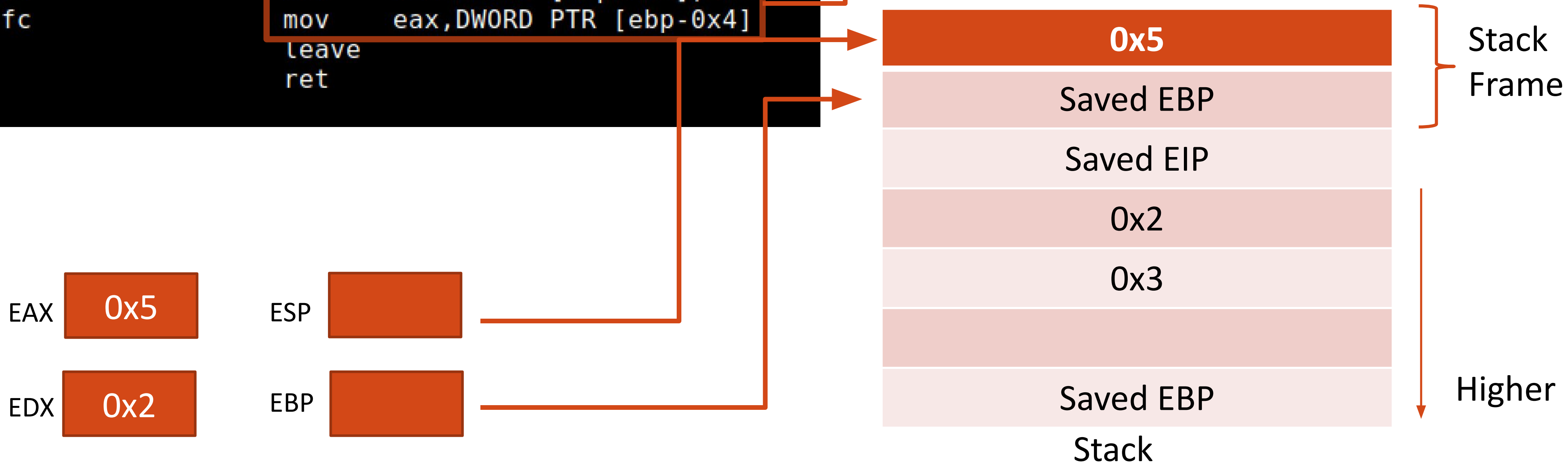
```
08049162 <sum>:
 8049162:       55                      push   ebp
 8049163:       89 e5                   mov    ebp,esp
 8049165:       83 ec 04                sub    esp,0x4
 8049168:       8b 55 08                mov    edx,DWORD PTR [ebp+0x8]
 804916b:       8b 45 0c                mov    eax,DWORD PTR [ebp+0xc]
 804916e:       01 d0                   add    eax,edx
 8049170:       89 45 fc                mov    DWORD PTR [ebp-0x4],eax
 8049173:       8b 45 fc                mov    eax,DWORD PTR [ebp-0x4]
 8049176:       c9                      leave
 8049177:       c3                      ret
```



Static analysis - example

```
08049162 <sum>:
 8049162:       55                      push   ebp
 8049163:       89 e5                   mov    ebp,esp
 8049165:       83 ec 04                sub    esp,0x4
 8049168:       8b 55 08                mov    edx,DWORD PTR [ebp+0x8]
 804916b:       8b 45 0c                mov    eax,DWORD PTR [ebp+0xc]
 804916e:       01 d0                   add    eax,edx
 8049170:       89 45 fc                mov    DWORD PTR [ebp-0x4],eax
 8049173:       8b 45 fc                mov    eax,DWORD PTR [ebp-0x4]
 8049176:       c9                      leave
 8049177:       c3                      ret
```

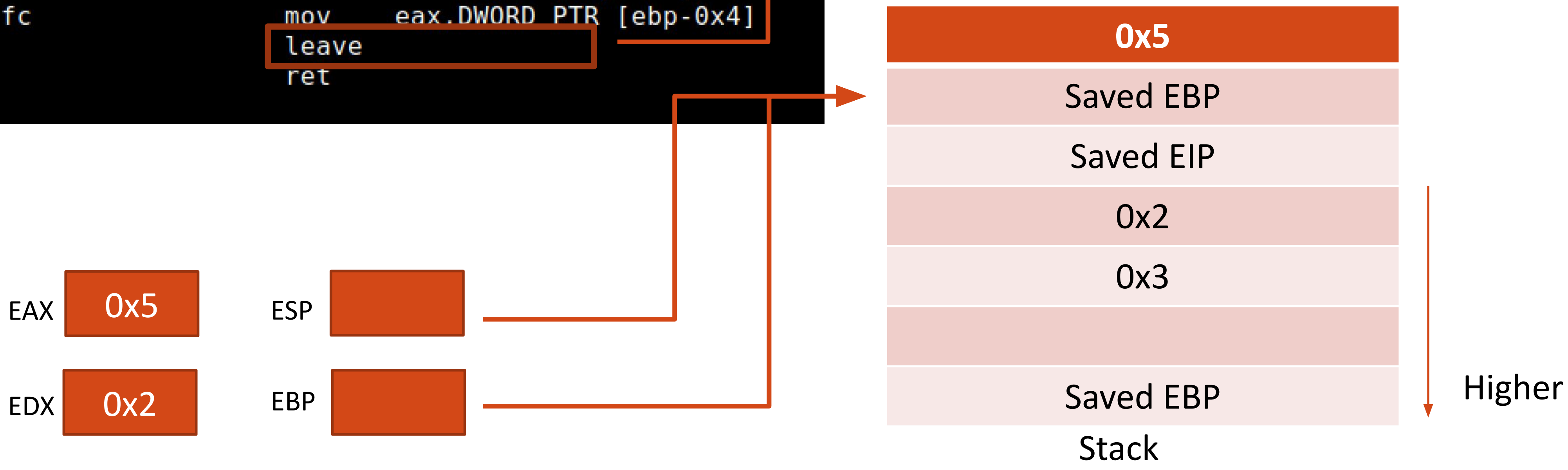
- Save result in local var c
- Move c into eax



Static analysis - example

```
08049162 <sum>:
 8049162:       55                   push   ebp
 8049163:       89 e5                mov    ebp,esp
 8049165:       83 ec 04             sub    esp,0x4
 8049168:       8b 55 08             mov    edx,DWORD PTR [ebp+0x8]
 804916b:       8b 45 0c             mov    eax,DWORD PTR [ebp+0xc]
 804916e:       01 d0                add    eax,edx
 8049170:       89 45 fc             mov    DWORD PTR [ebp-0x4],eax
 8049173:       8b 45 fc             mov    eax,DWORD PTR [ebp-0x4]
 8049176:       c9                   leave
 8049177:       c3                   ret
```

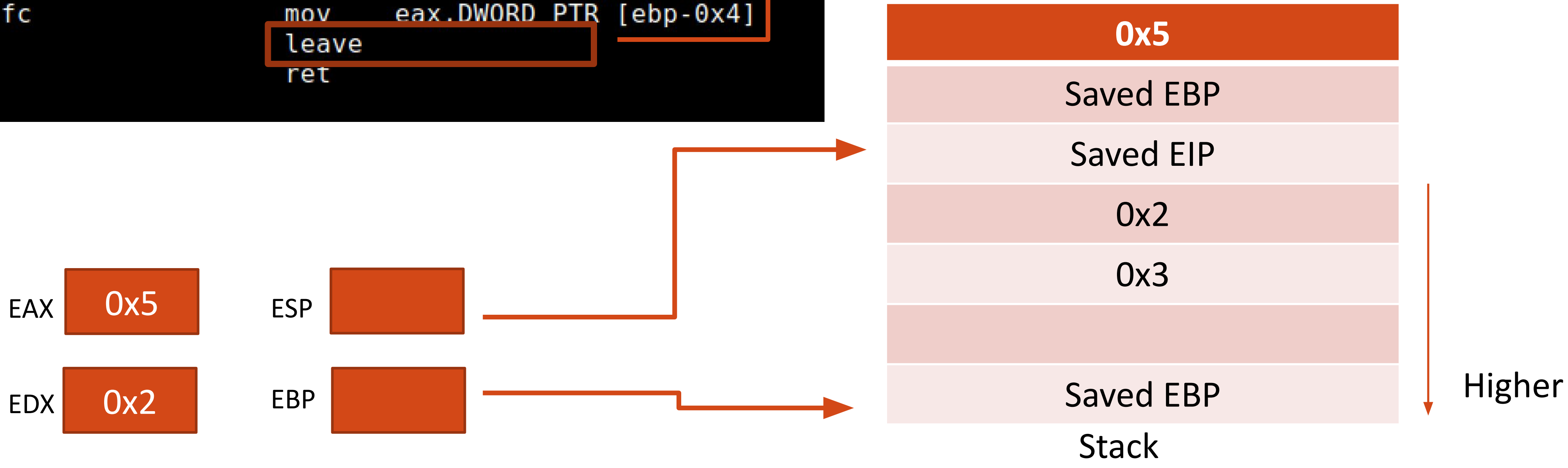
- mov esp, ebp
- pop ebp



Static analysis - example

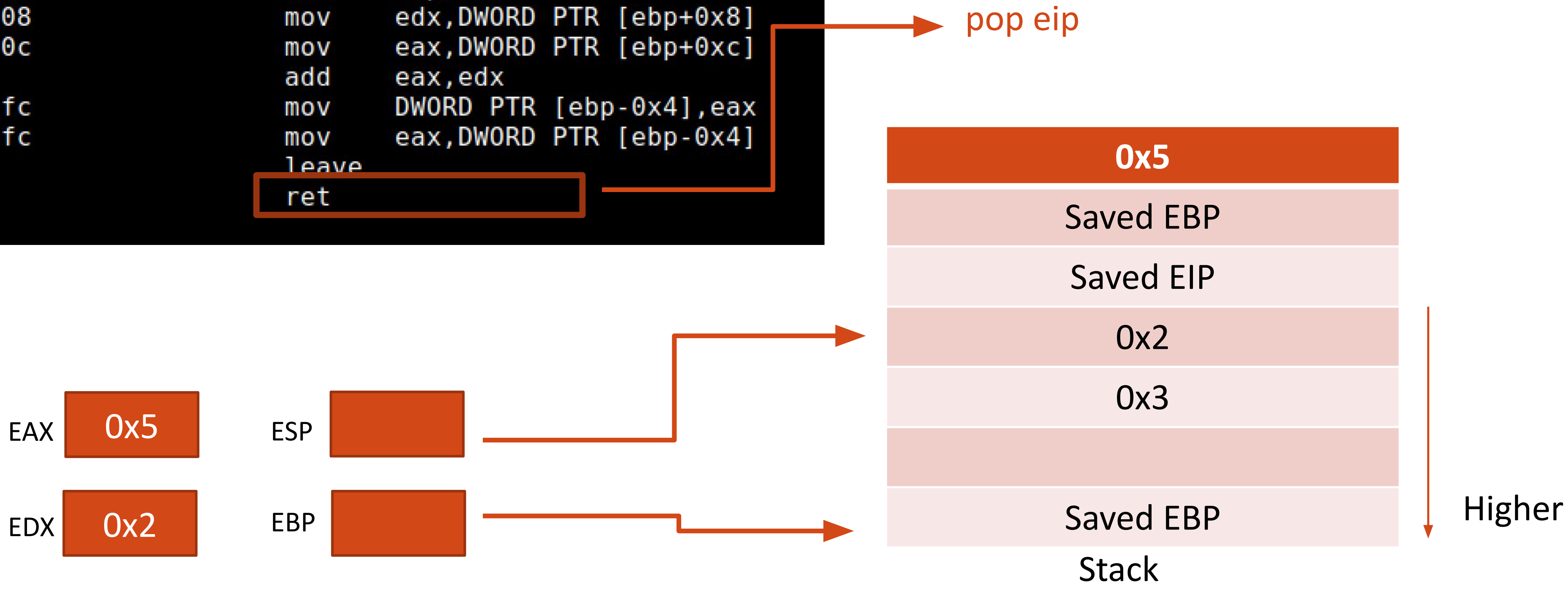
```
08049162 <sum>:
 8049162:       55                      push   ebp
 8049163:       89 e5                   mov    ebp,esp
 8049165:       83 ec 04                sub    esp,0x4
 8049168:       8b 55 08                mov    edx,DWORD PTR [ebp+0x8]
 804916b:       8b 45 0c                mov    eax,DWORD PTR [ebp+0xc]
 804916e:       01 d0                   add    eax,edx
 8049170:       89 45 fc                mov    DWORD PTR [ebp-0x4],eax
 8049173:       8b 45 fc                mov    eax,DWORD PTR [ebp-0x4]
 8049176:       c9                      leave
 8049177:       c3                      ret
```

- mov esp, ebp
- pop ebp



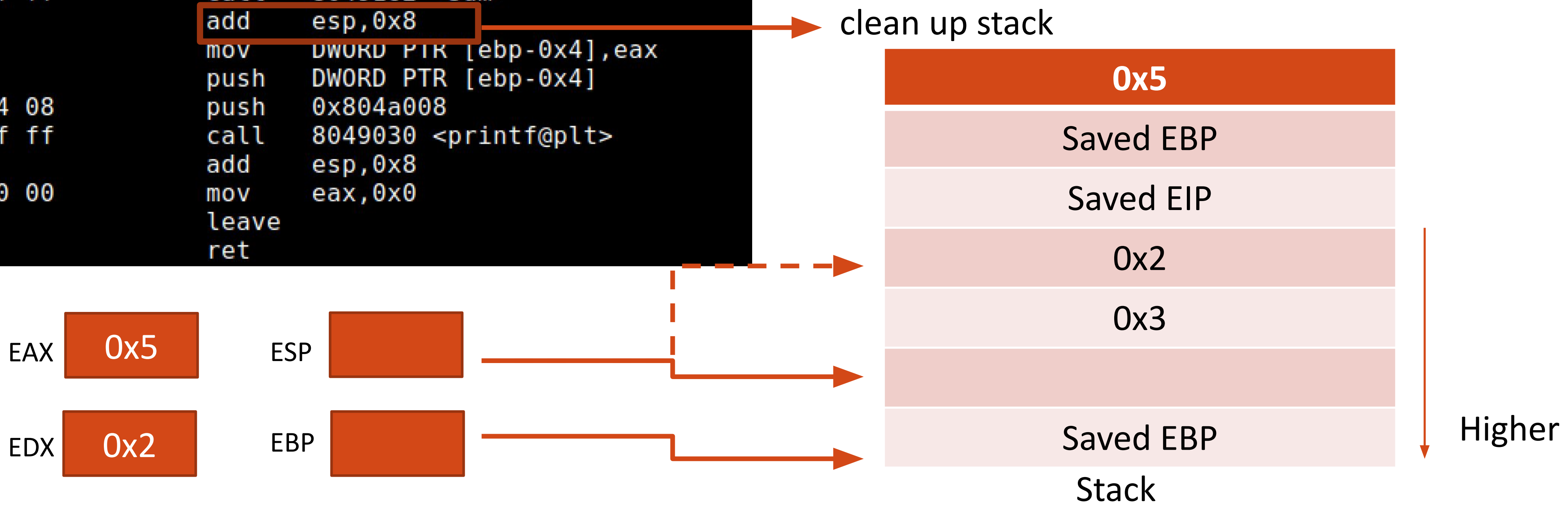
Static analysis - example

```
08049162 <sum>:
 8049162:       55                      push   ebp
 8049163:       89 e5                   mov    ebp,esp
 8049165:       83 ec 04                sub    esp,0x4
 8049168:       8b 55 08                mov    edx,DWORD PTR [ebp+0x8]
 804916b:       8b 45 0c                mov    eax,DWORD PTR [ebp+0xc]
 804916e:       01 d0                   add    eax,edx
 8049170:       89 45 fc                mov    DWORD PTR [ebp-0x4],eax
 8049173:       8b 45 fc                mov    eax,DWORD PTR [ebp-0x4]
 8049176:       c9                      leave
 8049177:       c3                      ret
```



Static analysis - example

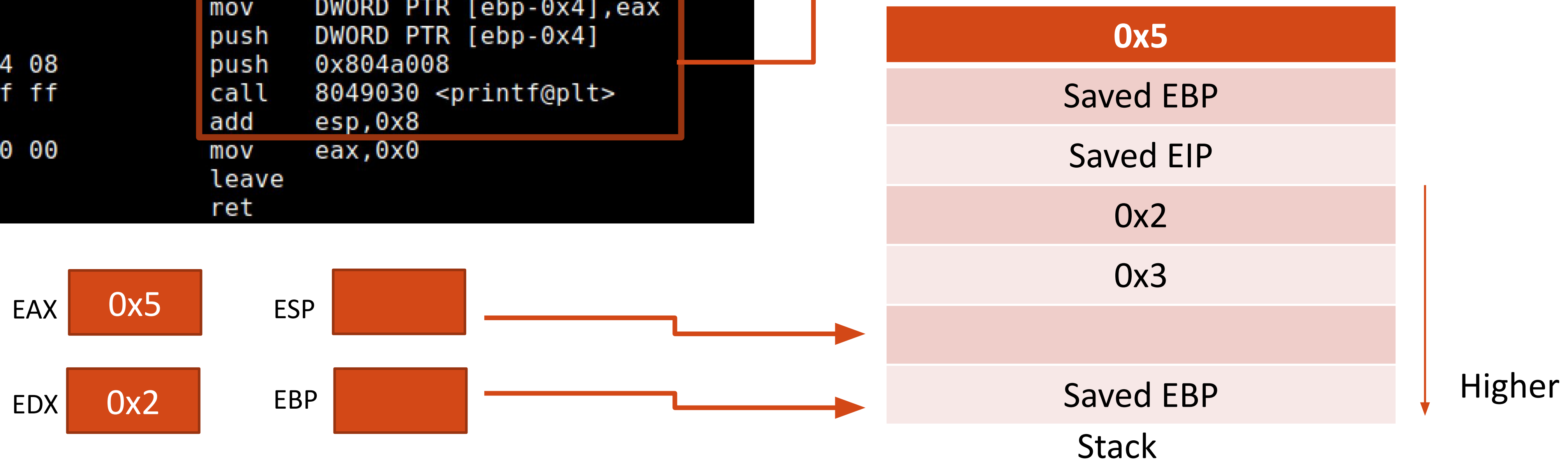
```
08049178 <main>:
 8049178:       55                      push   ebp
 8049179:       89 e5                   mov    ebp,esp
 804917b:       83 ec 04                sub    esp,0x4
 804917e:       6a 03                   push   0x3
 8049180:       6a 02                   push   0x2
 8049182:       e8 db ff ff ff          call   8049162 <sum>
 8049187:       83 c4 08                add    esp,0x8
 804918a:       89 45 fc                mov    DWORD PTR [ebp-0x4],eax
 804918d:       ff 75 fc                push   DWORD PTR [ebp-0x4]
 8049190:       68 08 a0 04 08          push   0x804a008
 8049195:       e8 96 fe ff ff          call   8049030 <printf@plt>
 804919a:       83 c4 08                add    esp,0x8
 804919d:       b8 00 00 00 00          mov    eax,0x0
 80491a2:       c9                      leave
 80491a3:       c3                      ret
```



Static analysis - example

```
08049178 <main>:
 8049178:       55                      push   ebp
 8049179:       89 e5                   mov    ebp,esp
 804917b:       83 ec 04                sub    esp,0x4
 804917e:       6a 03                   push   0x3
 8049180:       6a 02                   push   0x2
 8049182:       e8 db ff ff ff          call   8049162 <sum>
 8049187:       83 c4 08                add    esp,0x8
 804918a:       89 45 fc                mov    DWORD PTR [ebp-0x4],eax
 804918d:       ff 75 fc                push   DWORD PTR [ebp-0x4]
 8049190:       68 08 a0 04 08          push   0x804a008
 8049195:       e8 96 fe ff ff          call   8049030 <printf@plt>
 804919a:       83 c4 08                add    esp,0x8
 804919d:       b8 00 00 00 00          mov    eax,0x0
 80491a2:       c9                      leave
 80491a3:       c3                      ret
```

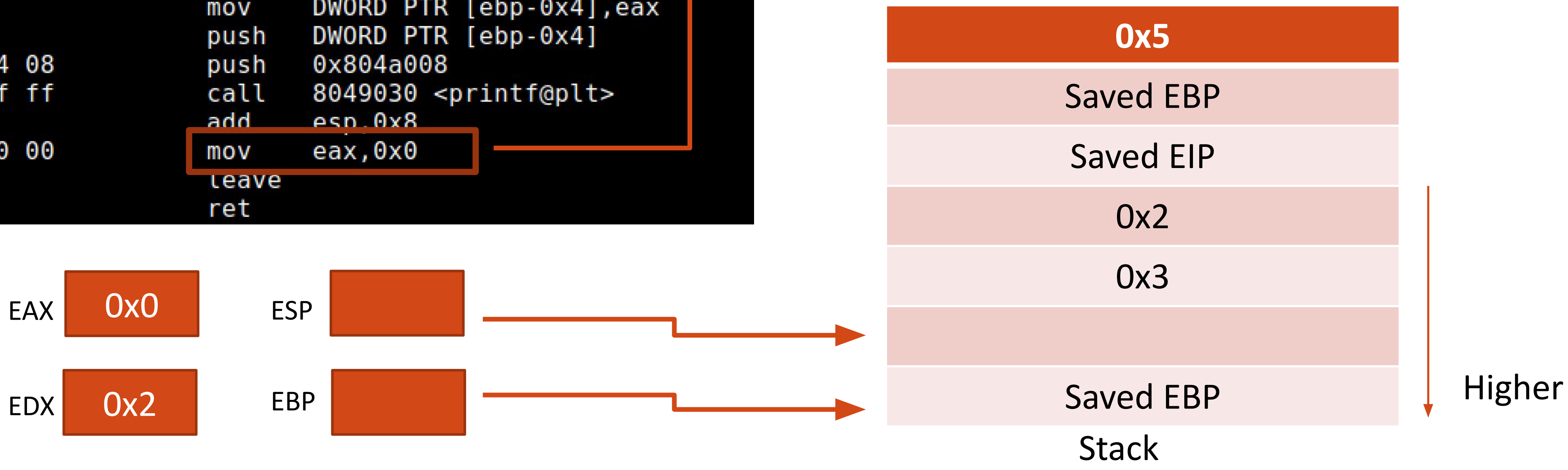
Exercise: try to understand how printf is called



Static analysis - example

```
08049178 <main>:
 8049178:       55                      push   ebp
 8049179:       89 e5                   mov    ebp,esp
 804917b:       83 ec 04                sub    esp,0x4
 804917e:       6a 03                   push   0x3
 8049180:       6a 02                   push   0x2
 8049182:       e8 db ff ff ff          call   8049162 <sum>
 8049187:       83 c4 08                add    esp,0x8
 804918a:       89 45 fc                mov    DWORD PTR [ebp-0x4],eax
 804918d:       ff 75 fc                push   DWORD PTR [ebp-0x4]
 8049190:       68 08 a0 04 08          push   0x804a008
 8049195:       e8 96 fe ff ff          call   8049030 <printf@plt>
 804919a:       83 c4 08                add    esp,0x8
 804919d:       b8 00 00 00 00          mov    eax,0x0
 80491a2:       c9                      leave
 80491a3:       c3                      ret
```

main returns 0



Calling conventions - x86

Function calling/leaving:

- call
  - push EIP [needed to return to caller's address when the called function returns]
  - jmp to function address
- ret
  - pop EIP

Function prologue/epilogue:

- Needed to create/destroy a function stack frame
- push EBP, EBP ESP [prologue]
- ESP EBP, pop EBP [epilogue]

Function parameters:

- pushed on the stack in reverse order by the caller
- called function goes outside its stack frame in order to get its parameters

Calling conventions - amd64

X86_64 changes:

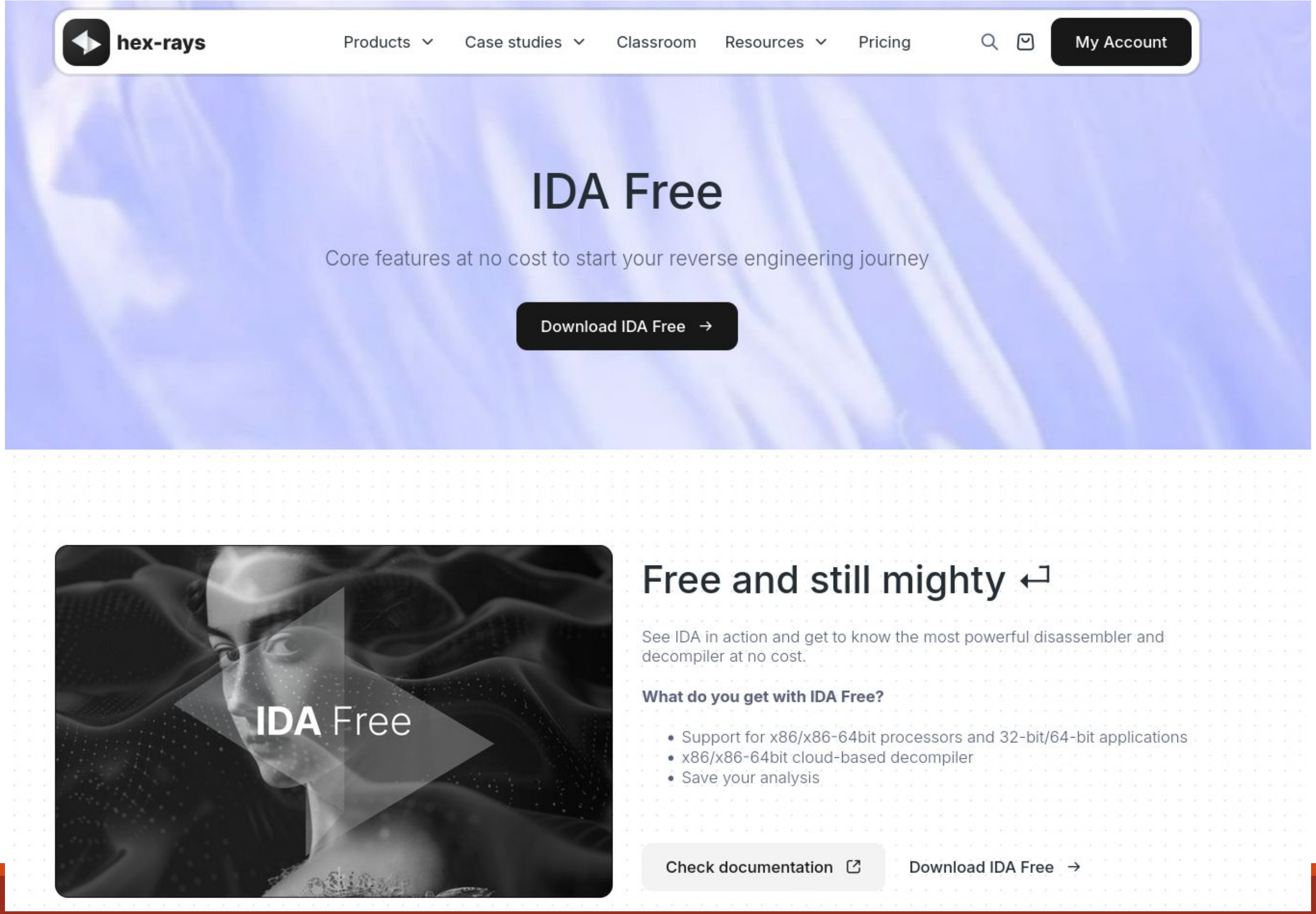
- General purpose registers and IP/BP/SP have been expanded to 64-bit
  - RAX, RBX, RCX, RDX, RSI, RDI
  - RIP, RBP, RSP
- Additional registers have been provided
  - R8 to R15
- Pointers are 8-bytes wide
- Push/pop on the stack are 8-bytes wide
- Maximum canonical address size of 0x00007FFFFFFFFFFF
- Parameters to functions are passed through registers
  - RDI, RSI, RCX, RDX, R8, R9
  - Other parameters are passed on the stack

Basic Dynamic Analysis

- Execute the program
- Trace library calls
  - `ltrace <elf_name>`
- Trace system calls
  - `strace <elf_name>`

- Debug the program
  - `gdb <elf_name>`
  - `gdb -p <pid>`

Tools - IDA



hex-rays
Products
Case studies
Classroom
Resources
Pricing
My Account
IDA Free
Core features at no cost to start your reverse engineering journey
Download IDA Free
IDA Free
Free and still mighty
See IDA in action and get to know the most powerful disassembler and decompiler at no cost.
What do you get with IDA Free?
Support for x86/x86-64bit processors and 32-bit/64-bit applications
x86/x86-64bit cloud-based decompiler
Save your analysis
Check documentation
Download IDA Free

Tools - GDB

GDB can do four main kinds of things (plus other things in support of these) to help you catch bugs in the act:

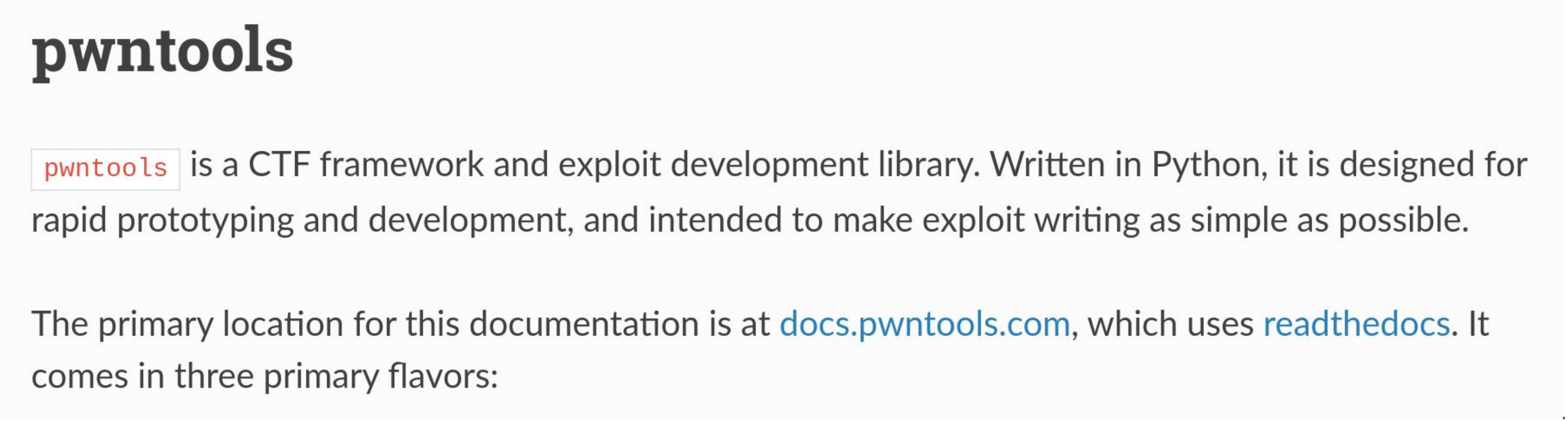
- Start your program, specifying anything that might affect its behavior.
- Make your program stop on specified conditions.
- Examine what has happened, when your program has stopped.
- Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another.

Tools - GDB Enhanced Features

- GEF consists of a set of commands that extends GDB with additional features for *dynamic analysis* and *exploit development*.
- GEF is based on GDB Python API
- Main GEF features:
  - Embedded hexdump view
  - Automatic dereferencing of *data* and *registers*
  - Heap analysis
  - Display ELF information
- Detailed GEF documentation is available at

https://gef.readthedocs.io/en/master/

Tools - Pwntools



pwntools

`pwntools` is a CTF framework and exploit development library. Written in Python, it is designed for rapid prototyping and development, and intended to make exploit writing as simple as possible.

The primary location for this documentation is at docs.pwntools.com, which uses readthedocs. It comes in three primary flavors:

https://docs.pwntools.com/en/stable/

Tools - dnspy

dnSpy - Latest release

dnSpy is a debugger and .NET assembly editor. You can use it to edit and debug assemblies even if you don't have any source code available. Main features:

- Debug .NET and Unity assemblies
- Edit .NET and Unity assemblies
- Light and dark themes

See below for more features

https://github.com/dnSpy/dnSpy

Tools - jadx

JADX

build passing | contributors 117 | downloads 2.4M | downloads@latest 299k | release v1.5.0 | maven-central v1.5.0 | Java 11+ | license apache

jadx - Dex to Java decompiler

Command line and GUI tools for producing Java source code from Android Dex and Apk files

https://github.com/skylot/jadx

Tools - bindiff

zynamics BinDiff

BinDiff uses a unique graph-theoretical approach to compare executables by identifying identical and similar functions

Description | Use Cases | Screenshots

Description

BinDiff is a comparison tool for binary files, that assists vulnerability researchers and engineers to quickly find differences and similarities in disassembled code.

With BinDiff you can identify and isolate fixes for vulnerabilities in vendor-supplied patches. You can also port symbols and comments between disassemblies of multiple versions of the same binary or use BinDiff to gather evidence for code theft or patent infringement.

https://www.zynamics.com/bindiff.html

Tools - z3

Z3 API in Python

Z3 is a high performance theorem prover developed at Microsoft Research. Z3 is used in many applications such as: software/hardware verification and testing, constraint solving, analysis of hybrid systems, security, biology (in si
analysis), and geometrical problems.

This tutorial demonstrates the main capabilities of Z3Py: the Z3 API in Python. No Python background is needed to read this tutorial. However, it is useful to learn Python (a fun language!) at some point, and there are many excelle
resources for doing so (Python Tutorial).

The Z3 distribution also contains the **C**, **.Net** and **OCaml** APIs. The source code of Z3Py is available in the Z3 distribution, feel free to modify it to meet your needs. The source code also demonstrates how to use new features in Z3
cool front-ends for Z3 include Scala^Z3 and SBV.

Please send feedback, comments and/or corrections to leonardo@microsoft.com. Your comments are very valuable.

https://ericpony.github.io/z3py-tutorial/guide-examples.htm

DEMO