Microsoft

The magic behind it!

# Debugging the debugger

Isadora Sophia and Henrique Silveira

March 2019

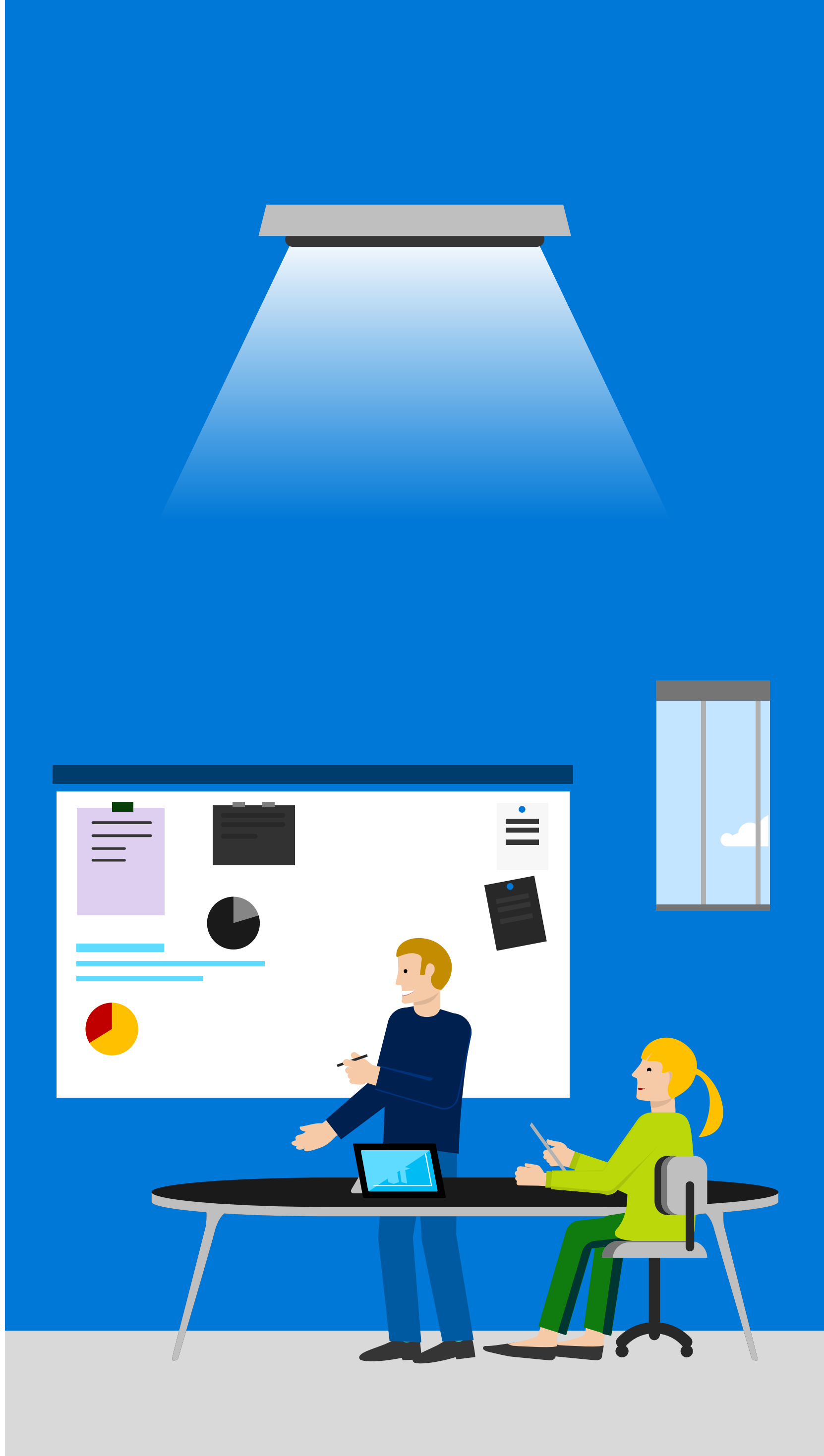
# Agenda

1 What is debugging?

2 From code to bits

3 The magic behind it

4 Debugger secrets



# What is debugging?

0800 arctan started {1.2700 9.037 847 025

1000 " stopped - arctan ✓ } 9.037 846 995 correct

~~1.982647000~~

13" sec (032) MP - MC ~~2.130476415 (-3~~) 4.615925059(-2)

(033) PRO 2 2.130476415

correct 2.130676415

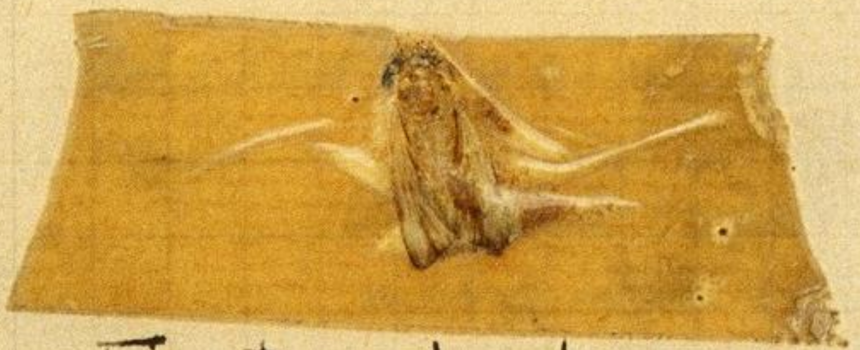
Relays 6-2 in 033 failed special speed test

In Relay " 10,000 test.

Relays changed

1100 Started Cosine Tape (Sine check)

1525 Started Mult + Adder Test.

1545  Relay #70 Panel F

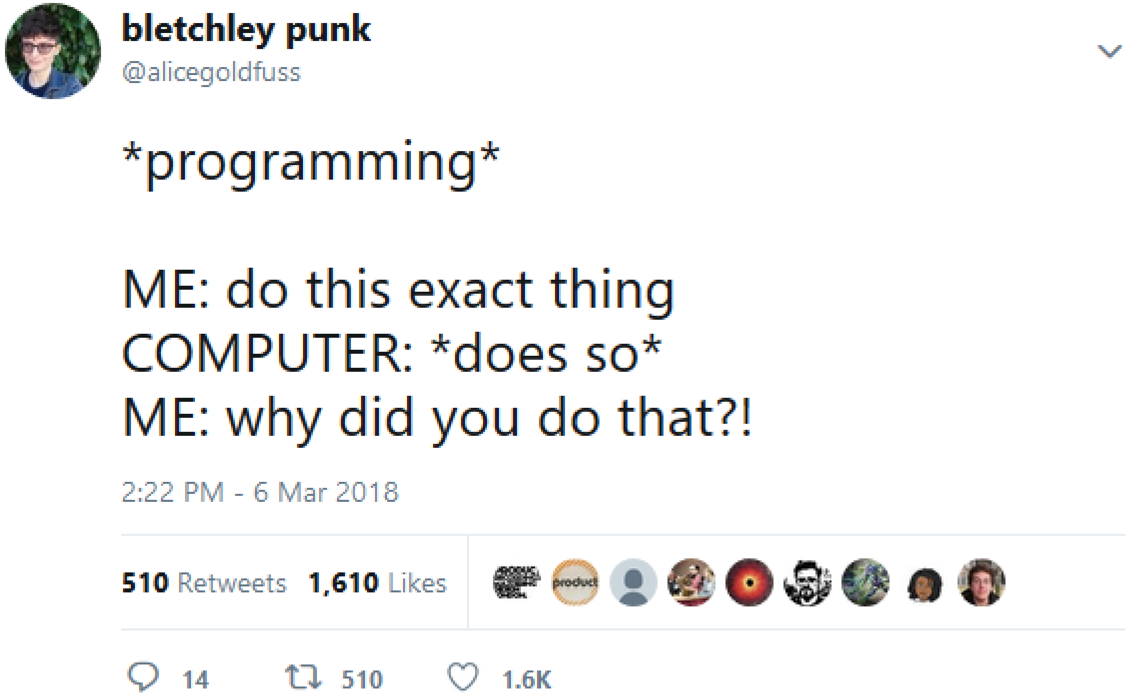
(moth) in relay.

First actual case of bug being found.

~~1600~~ 1630 arctangent started.

1700 closed down.

# …bugs are not charming



**bletchley punk**
@alicegoldfuss

*programming*

ME: do this exact thing
COMPUTER: *does so*
ME: why did you do that?!

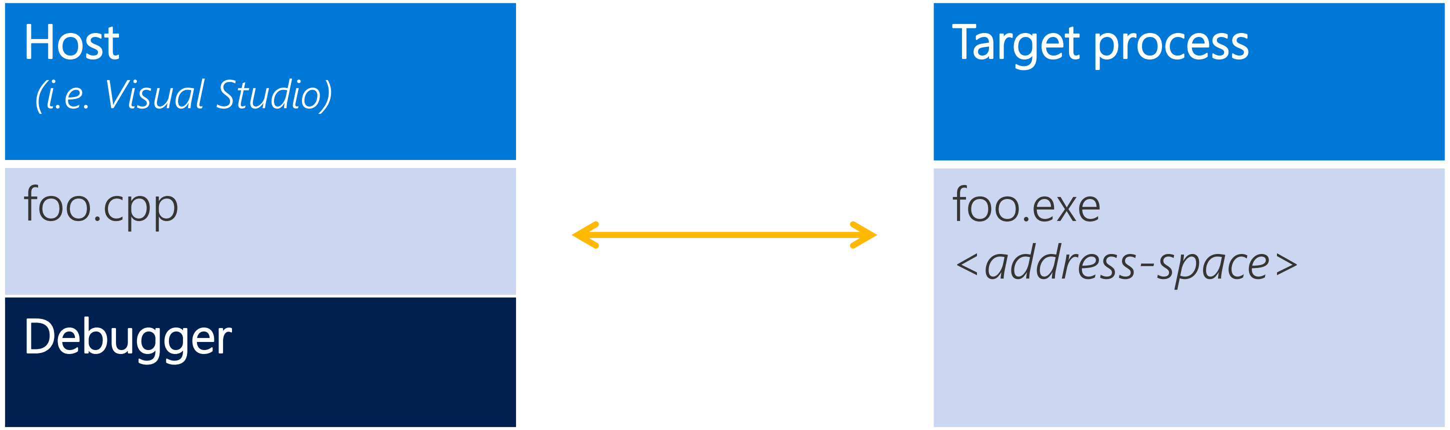
2:22 PM - 6 Mar 2018

**510** Retweets **1,610** Likes

14 510 1.6K

“a debugger is an application that is used to test and debug other applications.”

-Wikipedia



Host

*(i.e. Visual Studio)*

foo.cpp

Debugger

Target process

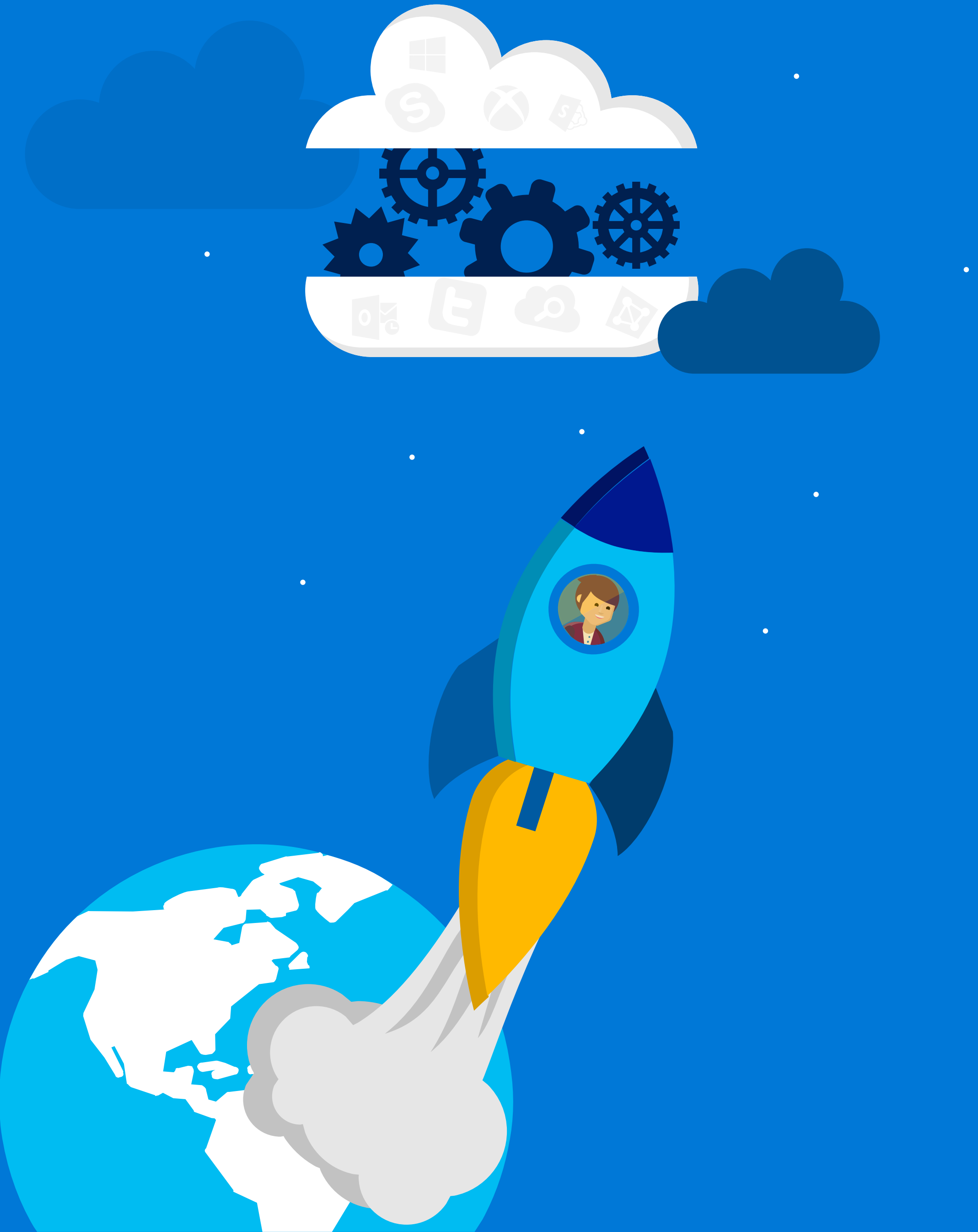
foo.exe

*<address-space>*

# Segmentation fault 101

Demo

# From code to bits



## Code it

```
int foo()
{
    return 42;
}
```

## Build it

```
$ gcc ./foo.c -g -o foo
```

*It targets…*

Operating System

Windows, Linux…

Architecture

x86, x64, ARM…

## Debug

Choose your weapon:

- Visual Studio
- gdb
- lldb
- windbg
- ++!

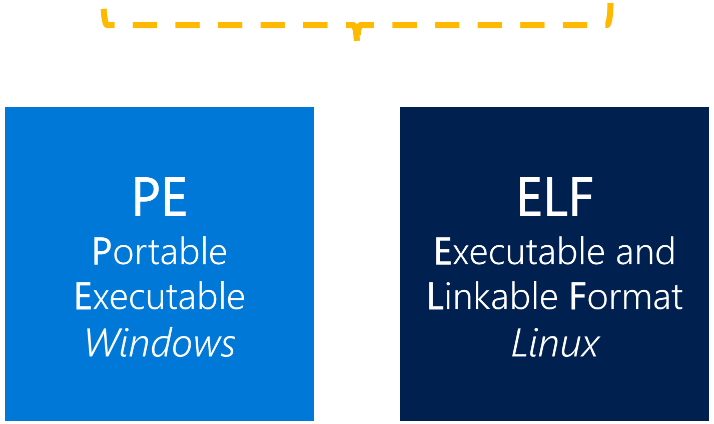
## Execute

```
$ ./foo
```

# Building it

Compiler does all the hard work for you!

It turns your code into an executable file.



**PE**
Portable
Executable
*Windows*

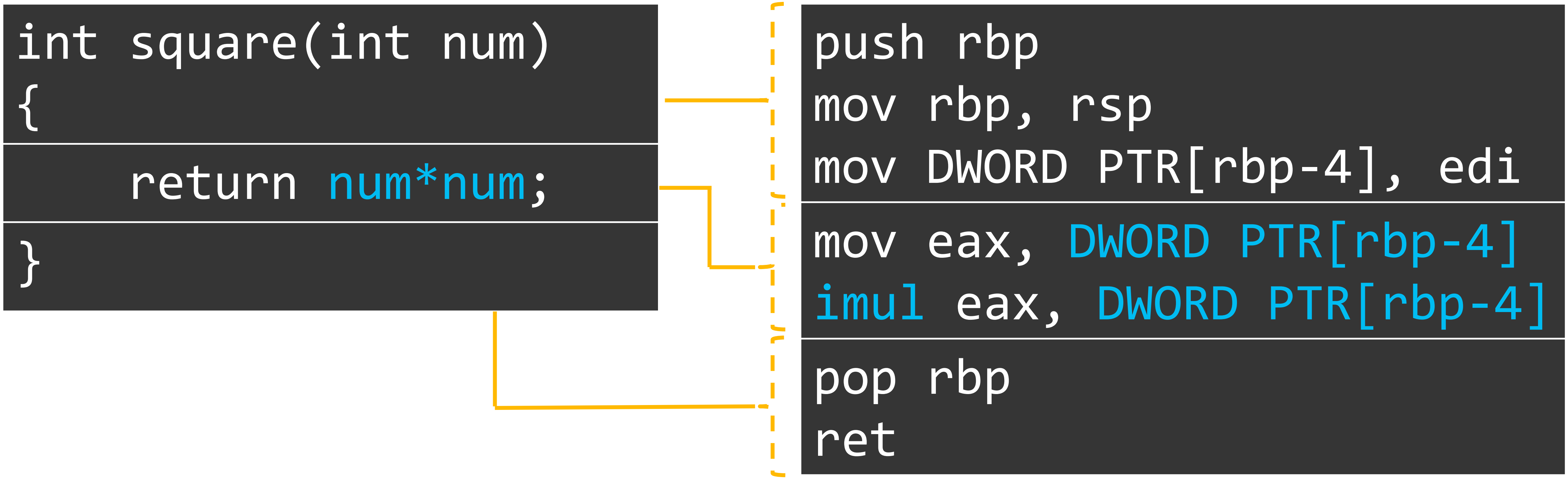
**ELF**
Executable and
Linkable Format
*Linux*

.exe

.elf

*+ target architecture*

# Where is my code?

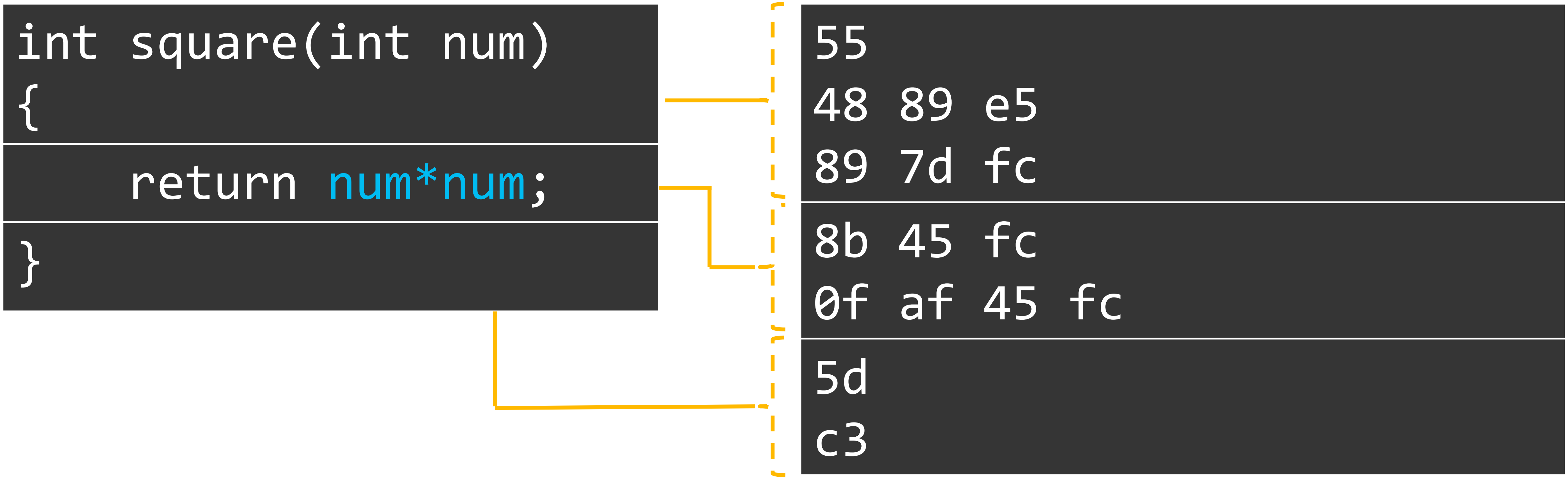


```c
int square(int num)
{
    return num*num;
}
```

```asm
push rbp
mov rbp, rsp
mov DWORD PTR[rbp-4], edi
mov eax, DWORD PTR[rbp-4]
imul eax, DWORD PTR[rbp-4]
pop rbp
ret
```

**x86_64 code. Check out https://godbolt.org/!*

# Where is my code?



```
int square(int num)
{
```

```
    return num*num;
```

```
}
```

```
55
48 89 e5
89 7d fc
```

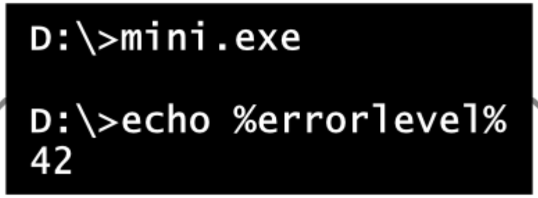
```
8b 45 fc
0f af 45 fc
```

```
5d
c3
```

**x86_64 code. Check out https://godbolt.org/!*

# Portable Executable

ANGE ALBERTINI
http://www.corkami.com



```
D:\>mini.exe

D:\>echo %errorlevel%
42
```

```
      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
000: .M .Z

030:                                     40 00 00 00
040: .P .E 00 00 4C 01
050:                   02 00 0B 01
060:                         40 01 00 00
070:             00 00 40 00 01 00 00 00 01 00 00 00
080:                         04 00
090: 60 01 00 00 40 01 00 00             03 00

140: B8 2A 00 00 00 C3
```

## MINI.EXE

## DOS HEADER

IT'S A BINARY

| FIELDS | VALUES |
|---|---|
| e_magic | MZ |
| e_lfanew | 0x40 → PE Header |

## PE HEADER

IT'S A 'MODERN' BINARY

| FIELDS | VALUES |
|---|---|
| → Signature | PE\0\0 |
| Machine | 0x14C [intel 386] |
| Characteristics | 2 [executable] |

## OPTIONAL HEADER

EXECUTABLE INFORMATION

| FIELDS | VALUES |
|---|---|
| Magic | 0x10B [32b] |
| AddressOfEntryPoint | 0x140 |
| ImageBase | 0x400000 |
| SectionAlignment | 1 |
| FileAlignment | 1 |
| MajorSubsystemVersion | 4 [NT 4 or later] |
| SizeOfImage | 0x160 |
| SizeOfHeaders | 0x140 |
| Subsystem | 3 [CLI] |

## CODE

X86 ASSEMBLY

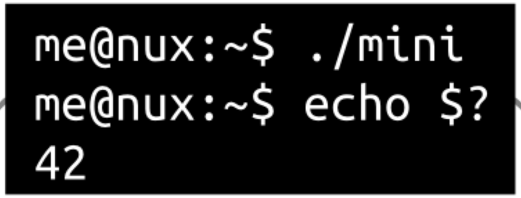
```
mov eax, 42
retn
```

EQUIVALENT C CODE

```
return 42;
```

# Executable and Linkable Format

ANGE ALBERTINI
http://www.corkami.com



```
me@nux:~$ ./mini
me@nux:~$ echo $?
42
```

```
     0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
00: 7F .E .L .F 01 01 01
10: 02 00 03 00 01 00 00 00 60 00 00 08 40 00 00 00
20:                         34 00 20 00 01 00

40: 01 00 00 00 00 00 00 00 00 00 00 08 00 00 00 08
50: 70 00 00 00 70 00 00 00 05 00 00 00

60: BB 2A 00 00 00 B8 01 00 00 00 CD 80
```

# MINI

## ELF HEADER

IDENTIFY AS AN ELF TYPE
SPECIFY THE ARCHITECTURE

| FIELDS | VALUES |
|---|---|
| e_ident | |
| EI_MAG | 0x7F, "ELF" |
| EI_CLASS, EI_DATA | $1^{\text{ELFCLASS32}}$, $1^{\text{ELFDATA2LSB}}$ |
| EI_VERSION | $1^{\text{EV\_CURRENT}}$ |
| e_type | $2^{\text{ET\_EXEC}}$ |
| e_machine | $3^{\text{EM\_386}}$ |
| e_version | $1^{\text{EV\_CURRENT}}$ |
| e_entry | 0x8000060 |
| e_phoff | 0x0000040 |
| e_ehsize | 0x0034 |
| e_phentsize | 0x0020 |
| e_phnum | 0001 |

## PROGRAM HEADER TABLE

EXECUTION INFORMATION

| FIELDS | VALUES |
|---|---|
| p_type | $1^{\text{PT\_LOAD}}$ |
| p_offset | 0 |
| p_vaddr | 0x8000000 |
| p_paddr | 0x8000000 |
| p_filesz | 0x0000070 |
| p_memsz | 0x0000070 |
| p_flags | $5^{\text{PF\_R}\vert\text{PF\_X}}$ |

## CODE

X86 ASSEMBLY

```
mov ebx, 42
mov eax, 1    SC_EXIT
int 80h
```

EQUIVALENT C CODE

```
return 42;
```

# Debug

For debugging, we need even more information.

The code must be compiled in debug mode*, so it also outputs debug information (e.g. symbols).

PDB
Program Database
*Windows*

.pdb

DWARF
Debugging with Attributed Record Formats
*Linux*

.dwarf

+ *others*

**check which flag you will need for your favorite compiler*

# Symbols are all you need!

## Address ranges ⇔ Source line

We have no idea how an instruction maps to a source line with just an executable file. The debug information allow us a map between each of the address ranges with its specified source line.

## Types used in the program

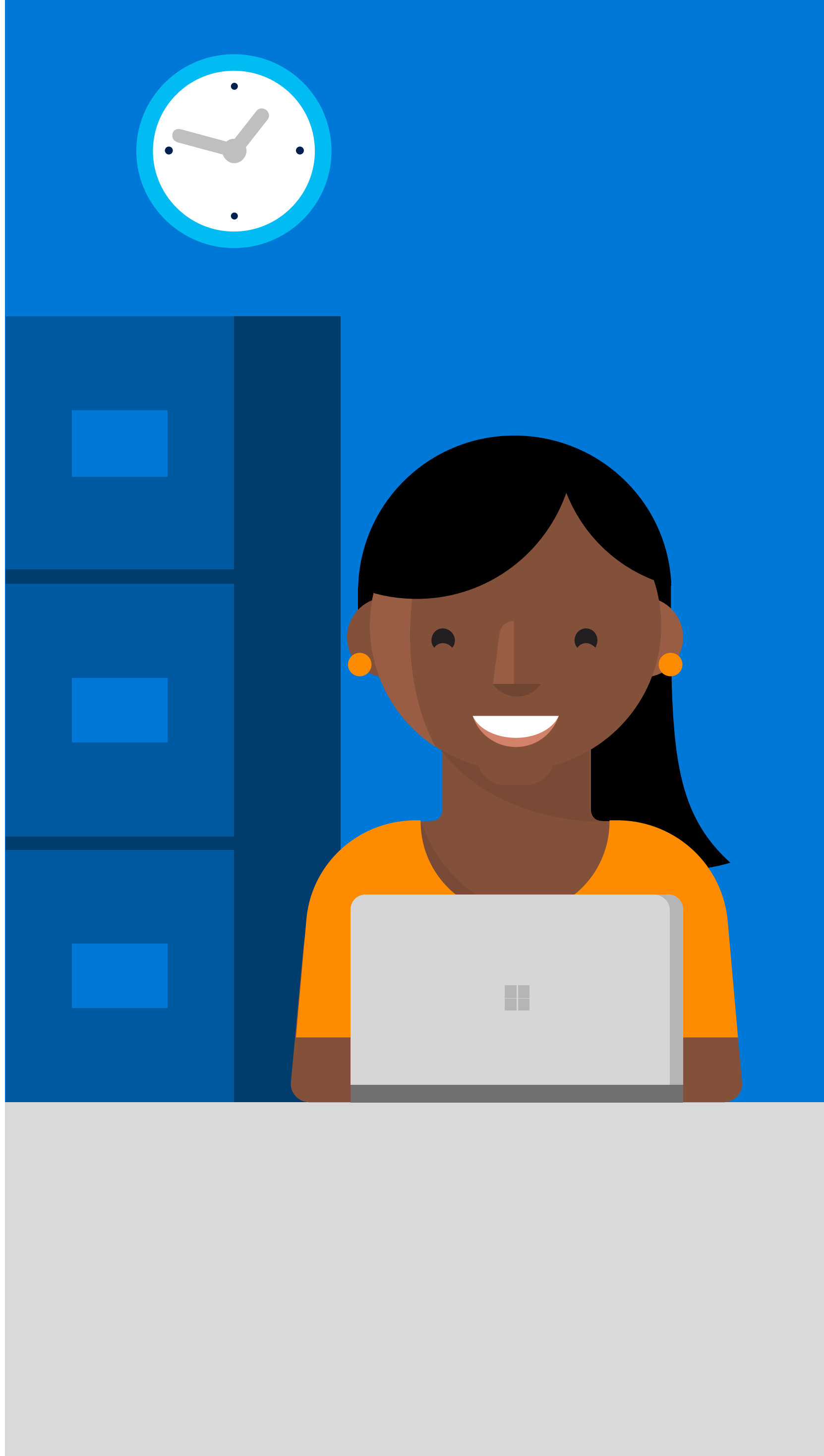
Tell us all the types referenced in the program. Very useful for expression evaluators.

## Variables

Tells us how to find each of the variables created in the program. Static, local or global variables, they are all here.

## Call frame

Tell us about the method call frames. When a method is called, a call frame is created on the program stack. Call frames also allows us to retrieve the call stack, which are super useful! They basically tell us how we got in the current method.



# More about symbols

Symbols are binary files, so it's very hard to read them by yourself.

Use specific tools if you are curious about their contents, such as:

```
$ cvdump.exe file.pdb
```

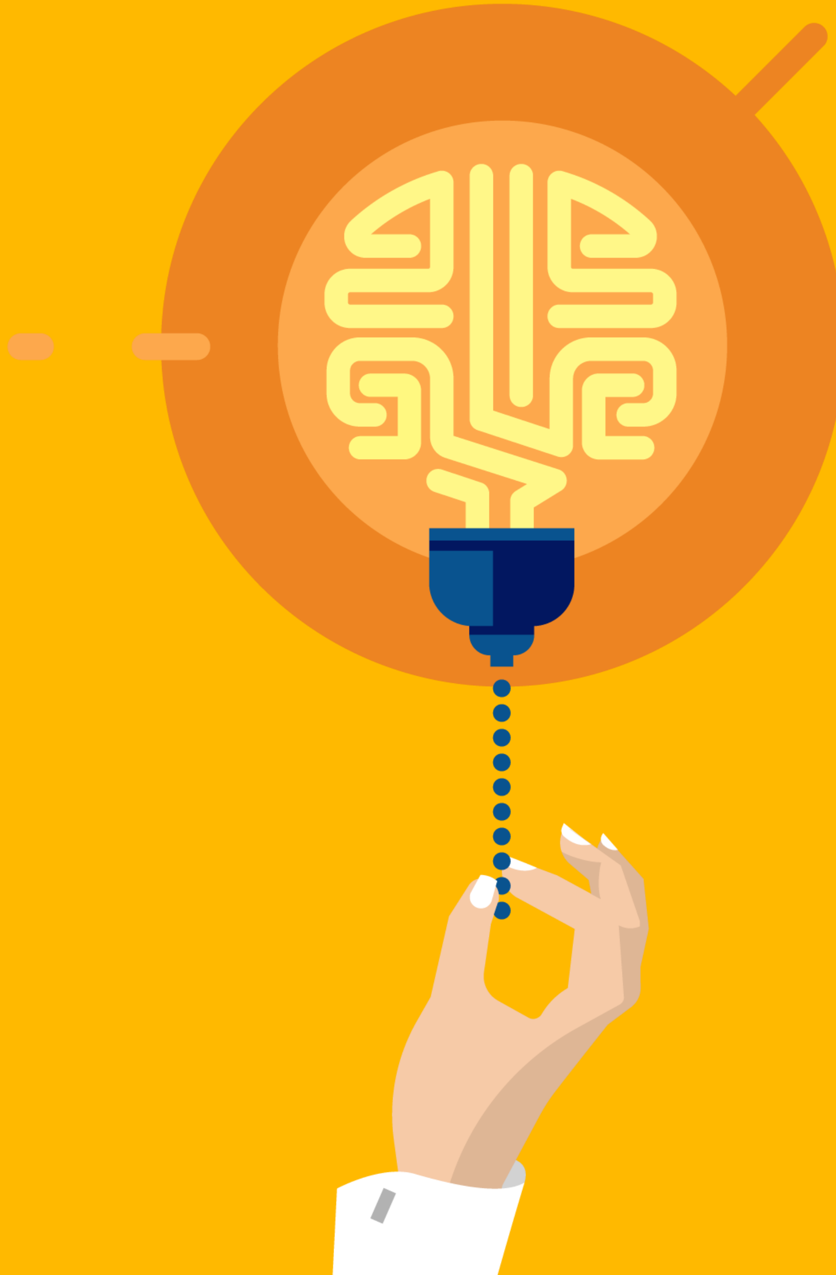
(https://github.com/Microsoft/microsoft-pdb)

```
$ dwarfdump file.dwarf
```

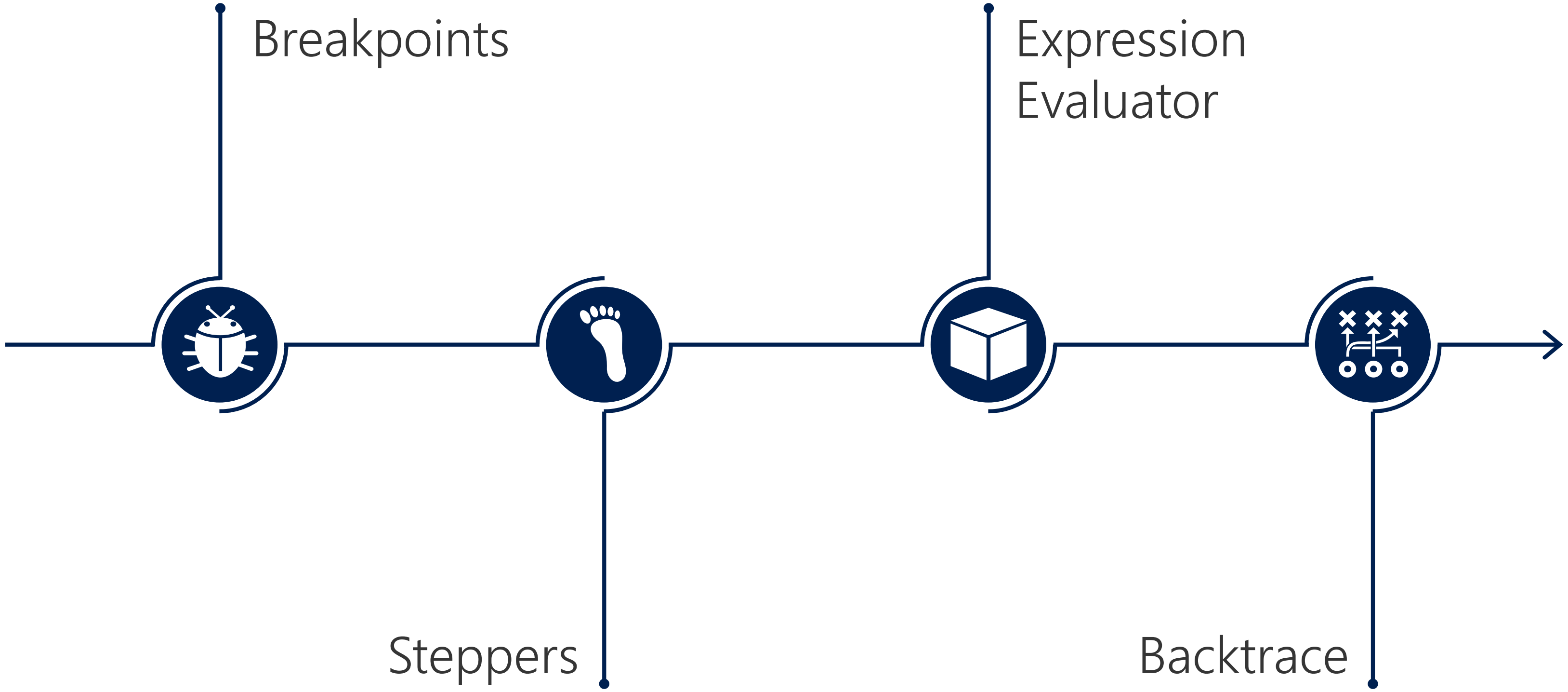
# Dumping stuff!

Demo

# Behind the magic…



# Debugger concepts



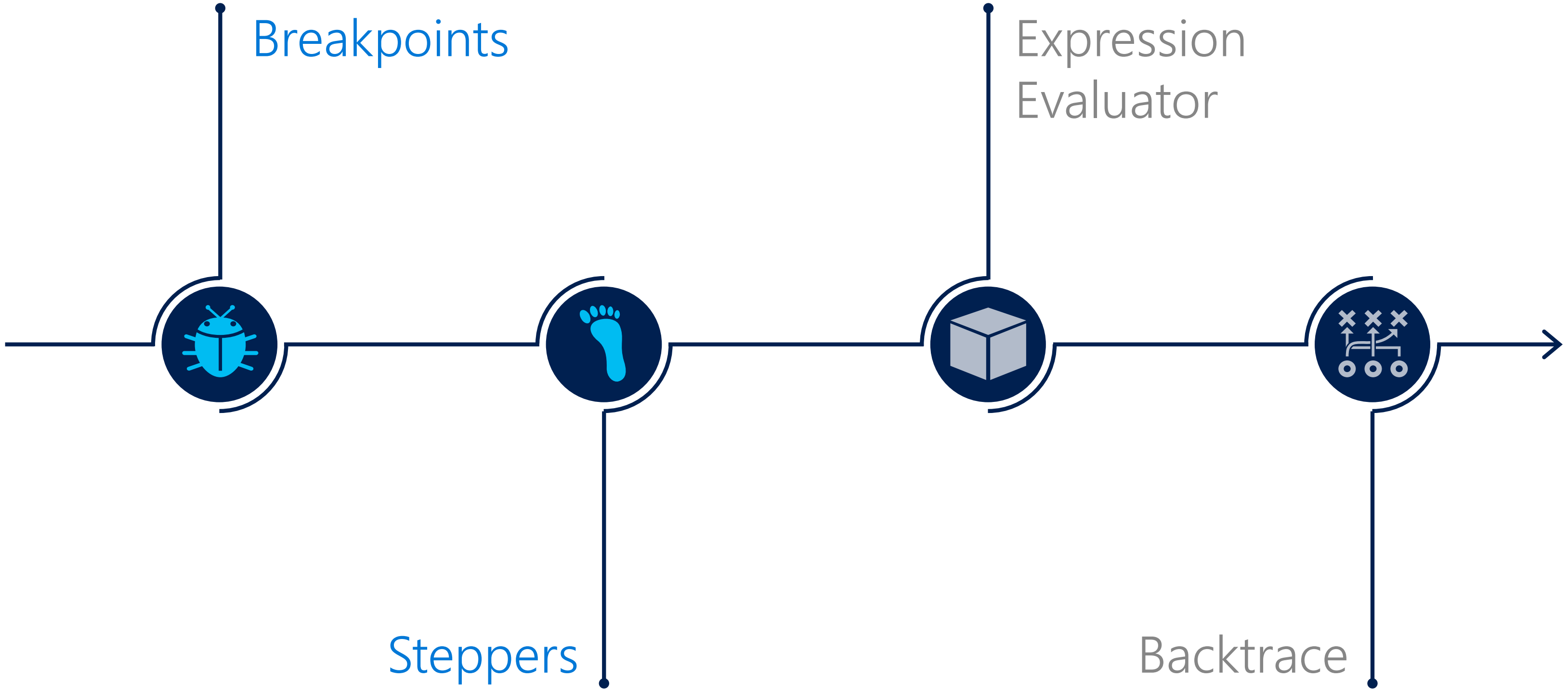
Breakpoints

Steppers

Expression Evaluator

Backtrace

# Debugger concepts



Breakpoints

Steppers

Expression Evaluator

Backtrace

# Breakpoints

Allow us to stop at any given point in our program.

There are two different kinds of breakpoints:

Hardware Breakpoints

VS.

Software Breakpoints

# Hardware Breakpoints

- **“Dr.”** - **D**ebug **R**egister
  - Rely on *x86* debug registers!
- Can break on…
  - Read,
  - Write,
  - Execute.
- Limited in number
  - Up to 4 breakpoints

**DR0-DR3**

Linear addresses of up to **4** breakpoints

**DR4-DR5**

Reserved

**DR4/DR6** – *Debug Status Register*

Determine which debug conditions have occurred

**DR5/DR7** – *Debug Control Register*

Type of breakpoint

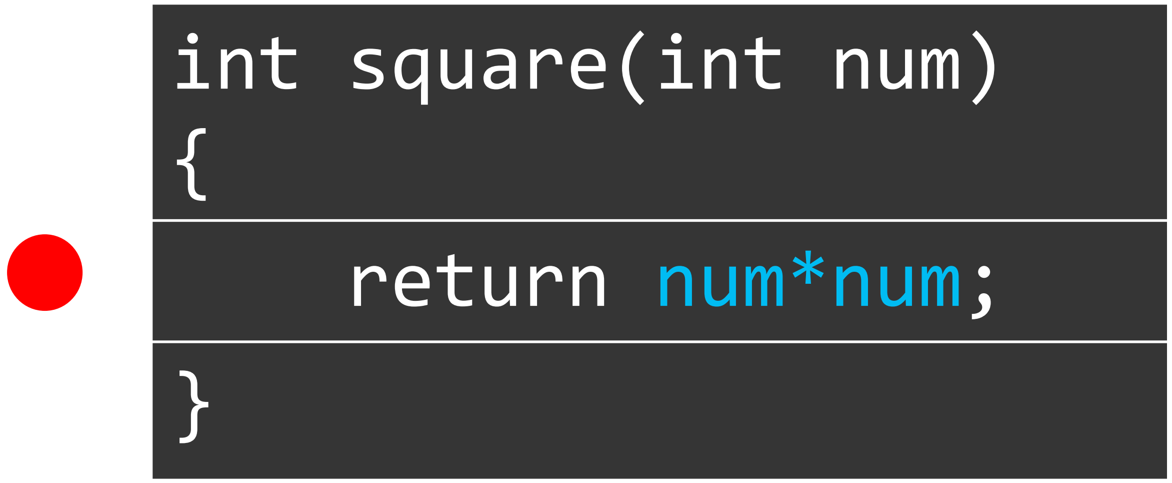
# Memory corruption simulation

Demo

# Software Breakpoints

- INT 3
  - *x86* instruction
  - Opcode is **0xCC**
  - Breakpoint trap
- Unlimited in number
- Debugger modifies the running code to introduce breaking instructions

# Software Breakpoints



```
int square(int num)
{
    return num*num;
}
```

# Software Breakpoints

```
  55
  48 89 e5
  89 7d fc
● 8b 45 fc
  0f af 45 fc
  5d
  c3
```

```
push rbp
mov rbp, rsp
mov DWORD PTR[rbp-4], edi
mov eax, DWORD PTR[rbp-4]
imul eax, DWORD PTR[rbp-4]
pop rbp
ret
```

# Software Breakpoints

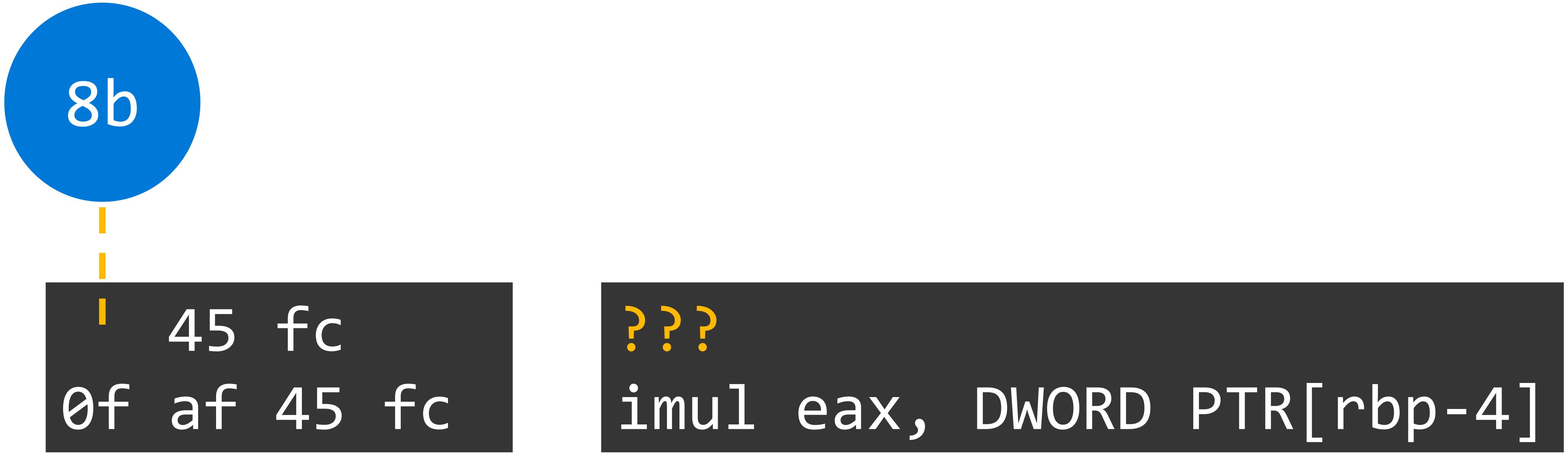
```
8b 45 fc
0f af 45 fc
```

```
mov eax, DWORD PTR[rbp-4]
imul eax, DWORD PTR[rbp-4]
```

**Debugger**

| | |
|---|---|
| Status | Loading... |

# Software Breakpoints



```
   45 fc
0f af 45 fc
```

```
???
imul eax, DWORD PTR[rbp-4]
```

| Debugger | |
|---|---|
| Status | Loading... |

# Software Breakpoints

```
   45 fc
0f af 45 fc
```

```
???
imul eax, DWORD PTR[rbp-4]
```

Debugger

Status Loading...

8b

# Software Breakpoints

```
cc 45 fc
0f af 45 fc
```

```
int3
imul eax, DWORD PTR[rbp-4]
```

Debugger

Status Loading...

8b

# Software Breakpoints

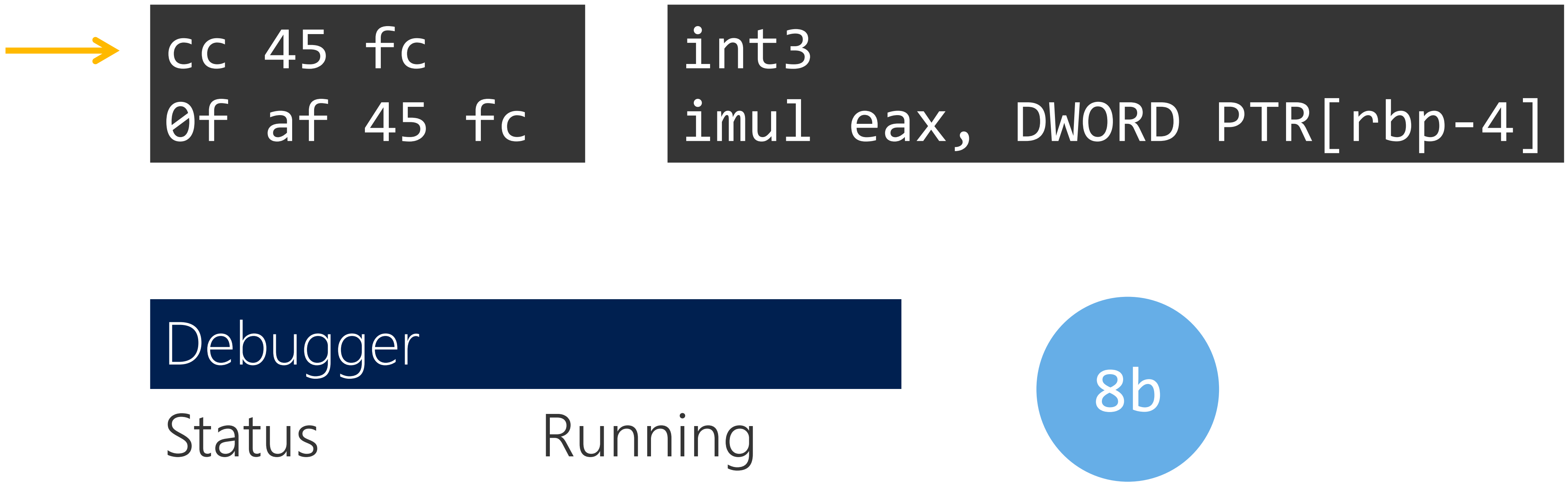
```
cc 45 fc
0f af 45 fc
```

```
int3
imul eax, DWORD PTR[rbp-4]
```

| Debugger | |
|---|---|
| Status | Running |

8b

# Software Breakpoints



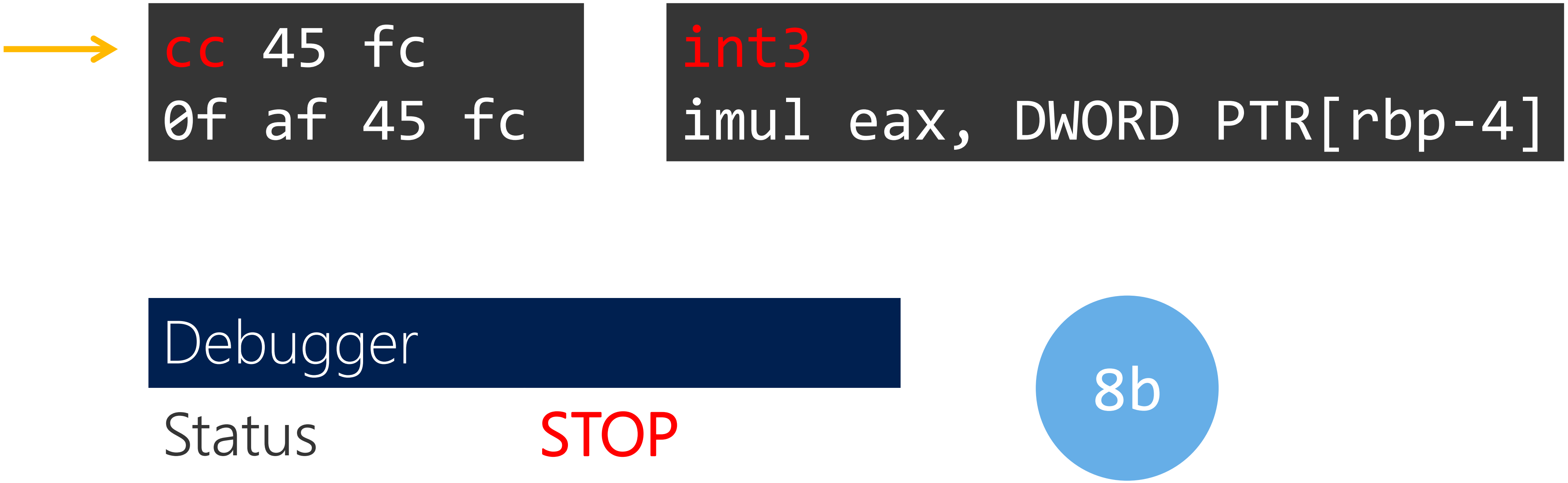
```
cc 45 fc
0f af 45 fc
```

```
int3
imul eax, DWORD PTR[rbp-4]
```

Debugger

Status Running

# Software Breakpoints



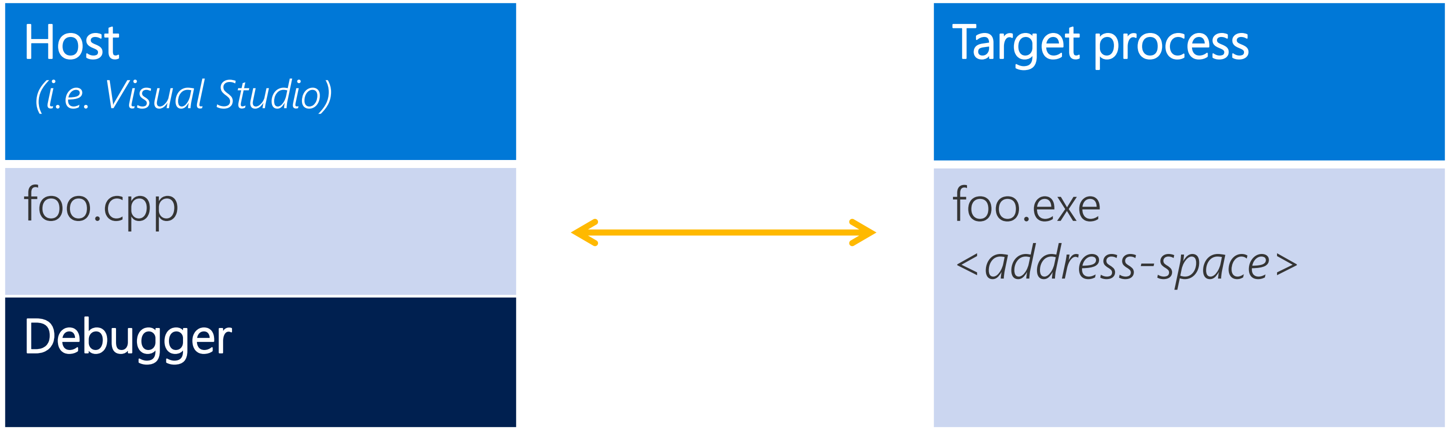
```
cc 45 fc
0f af 45 fc
```

```
int3
imul eax, DWORD PTR[rbp-4]
```

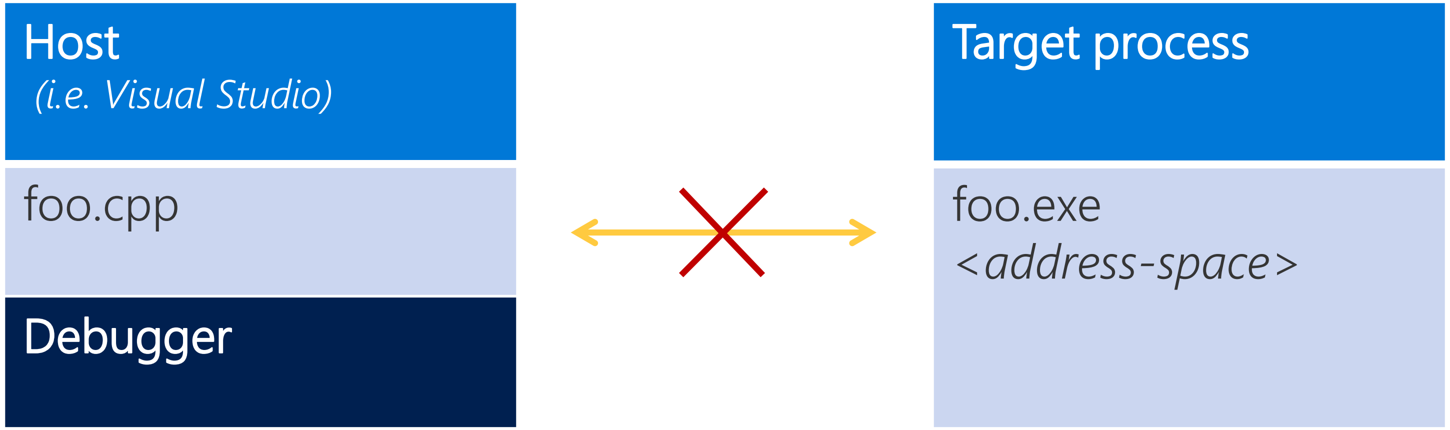
Debugger

Status STOP

# Software Interrupt



# Software Interrupt



Host

*(i.e. Visual Studio)*

foo.cpp

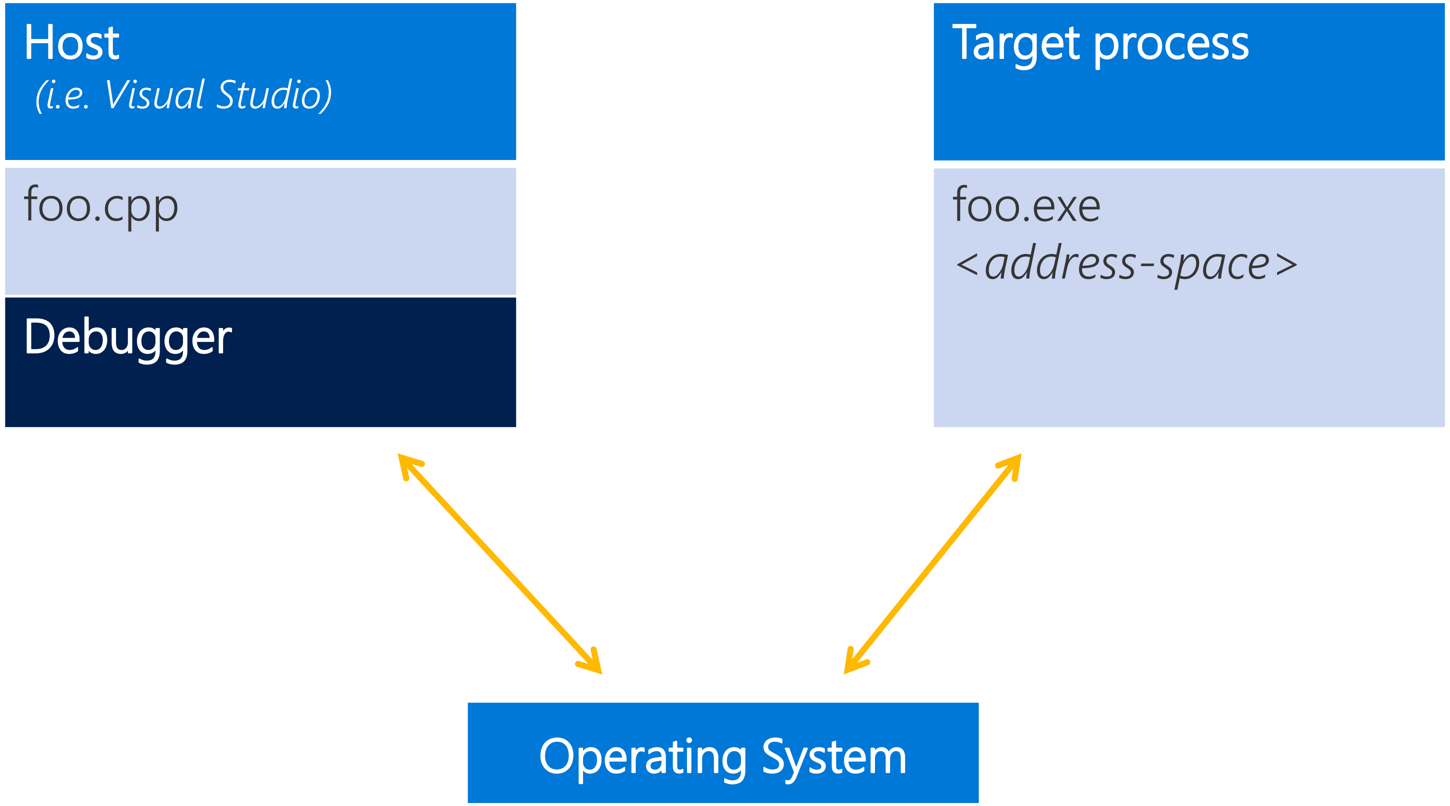
Debugger

Target process

foo.exe

*<address-space>*

# Software Interrupt



# Exceptions

- Everything is an exception!
- **Faults**
  - Happens before the CPU can execute the instruction
    - e.g. divide by zero
- **Traps**
  - Happens after or during the execution of an instruction
    - e.g. breakpoints, system overflow
- **Aborts**
  - Operation is no longer possible
    - e.g. killing a process

# Beyond a breakpoint

Demo

# Steppers

Step

- ...an instruction
- ...out
- ...in
- ...over

# Step an instruction

```
push rbp
mov rbp, rsp
mov DWORD PTR[rbp-4], edi
int3
imul eax, DWORD PTR[rbp-4]
pop rbp
ret
```

1. Put back the original instruction bytes

# Step an instruction

```
push rbp
mov rbp, rsp
mov DWORD PTR[rbp-4], edi

mov eax, DWORD PTR[rbp-4]
imul eax, DWORD PTR[rbp-4]

pop rbp
ret
```

1. Put back the original instruction bytes

# Step an instruction

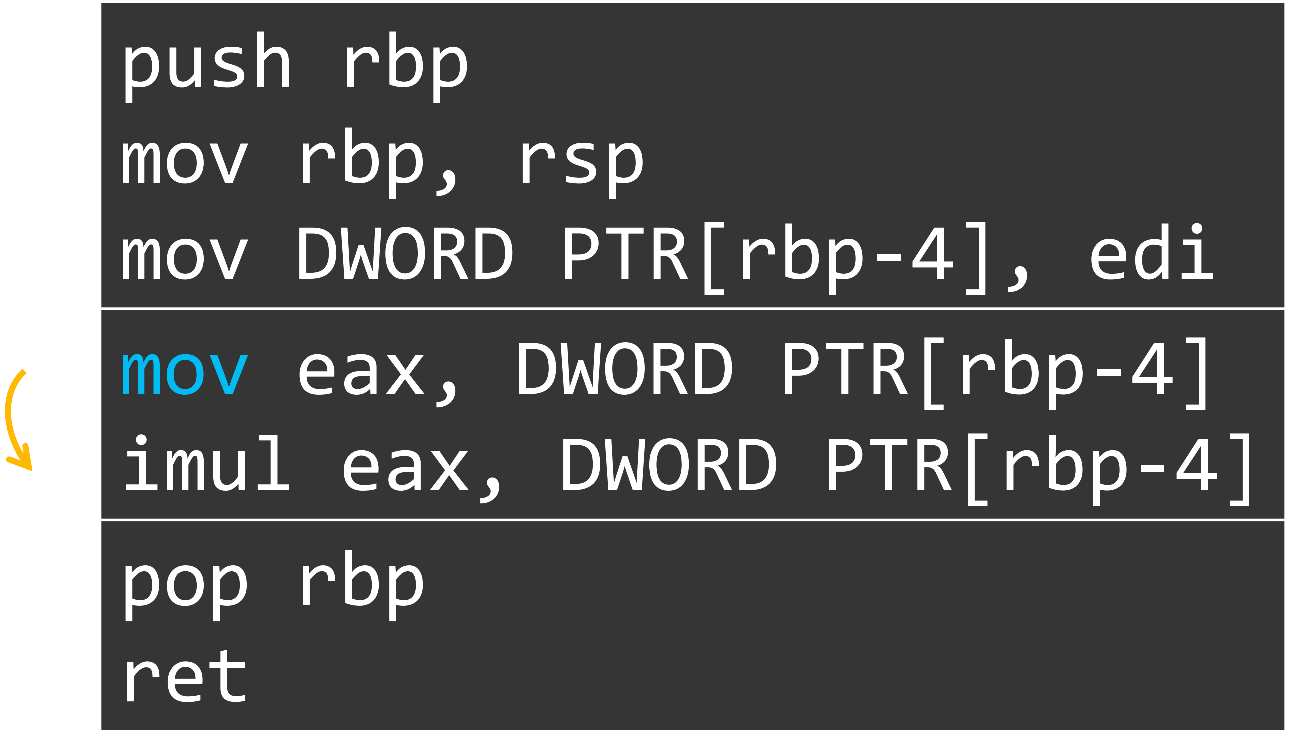
```
push rbp
mov rbp, rsp
mov DWORD PTR[rbp-4], edi

mov eax, DWORD PTR[rbp-4]
imul eax, DWORD PTR[rbp-4]

pop rbp
ret
```

1. Put back the original instruction bytes
2. Manage any other threads (if needed)

# Step an instruction



```
push rbp
mov rbp, rsp
mov DWORD PTR[rbp-4], edi
mov eax, DWORD PTR[rbp-4]
imul eax, DWORD PTR[rbp-4]
pop rbp
ret
```

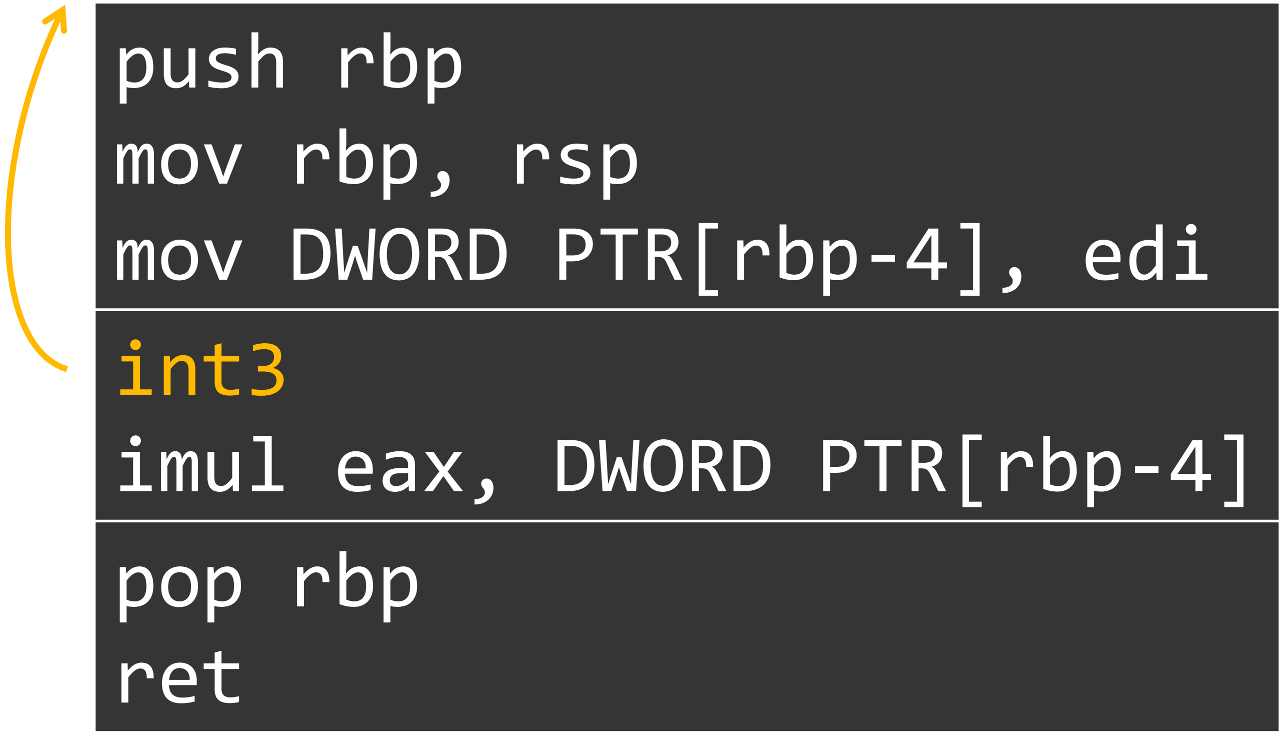
1. Put back the original instruction bytes
2. Manage any other threads (if needed)
3. Step on next instruction
   - Enable the CPU single-step trap flag

# Step an instruction

```
push rbp
mov rbp, rsp
mov DWORD PTR[rbp-4], edi
mov eax, DWORD PTR[rbp-4]
imul eax, DWORD PTR[rbp-4]
pop rbp
ret
```

1. Put back the original instruction bytes
2. Manage any other threads (if needed)
3. Step on next instruction
   - Enable the CPU single-step trap flag

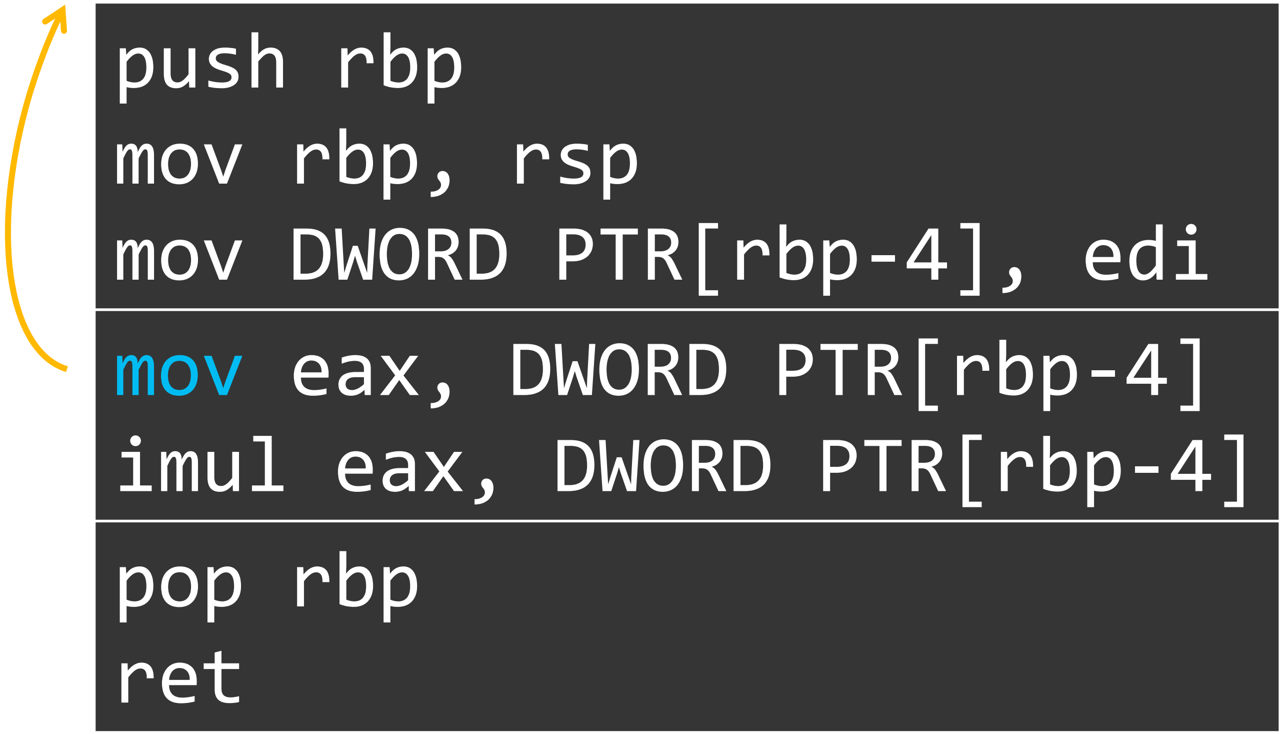
# Step out



```
push rbp
mov rbp, rsp
mov DWORD PTR[rbp-4], edi
int3
imul eax, DWORD PTR[rbp-4]
pop rbp
ret
```

1. Put back the original instruction bytes

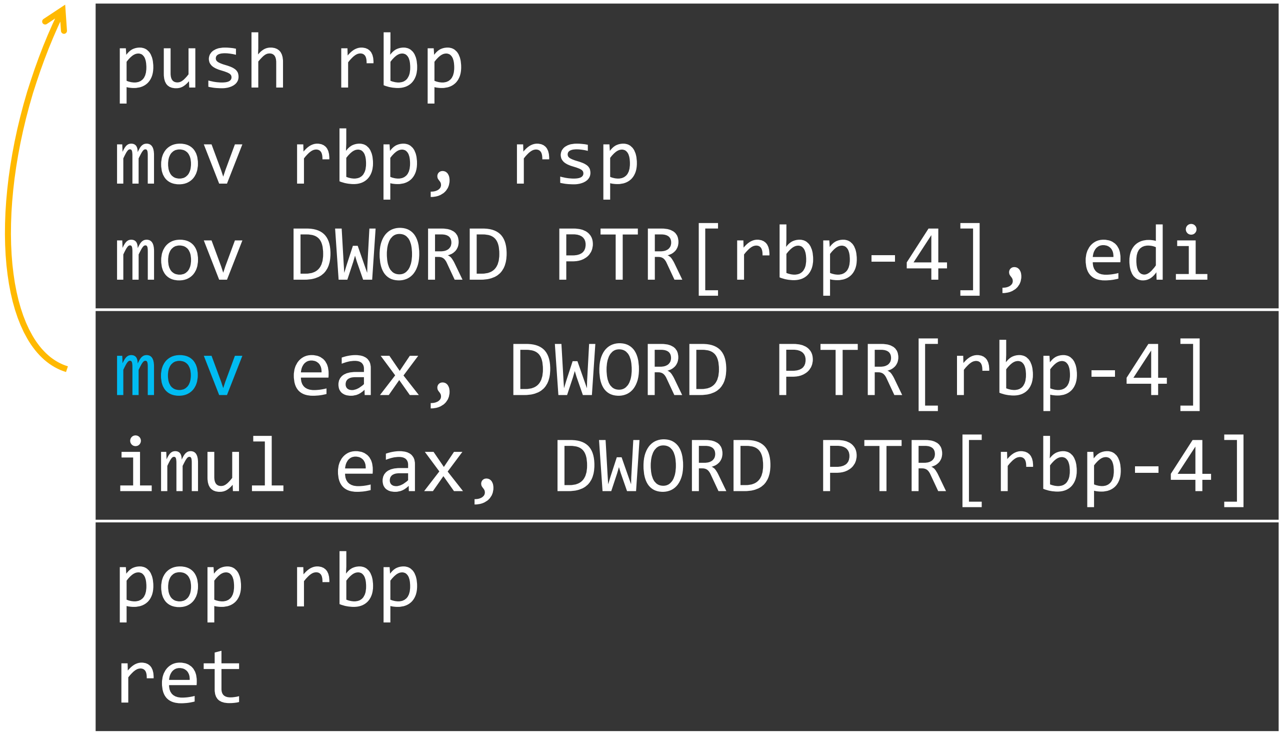
# Step out



```
push rbp
mov rbp, rsp
mov DWORD PTR[rbp-4], edi
mov eax, DWORD PTR[rbp-4]
imul eax, DWORD PTR[rbp-4]
pop rbp
ret
```

1. Put back the original instruction bytes

# Step out



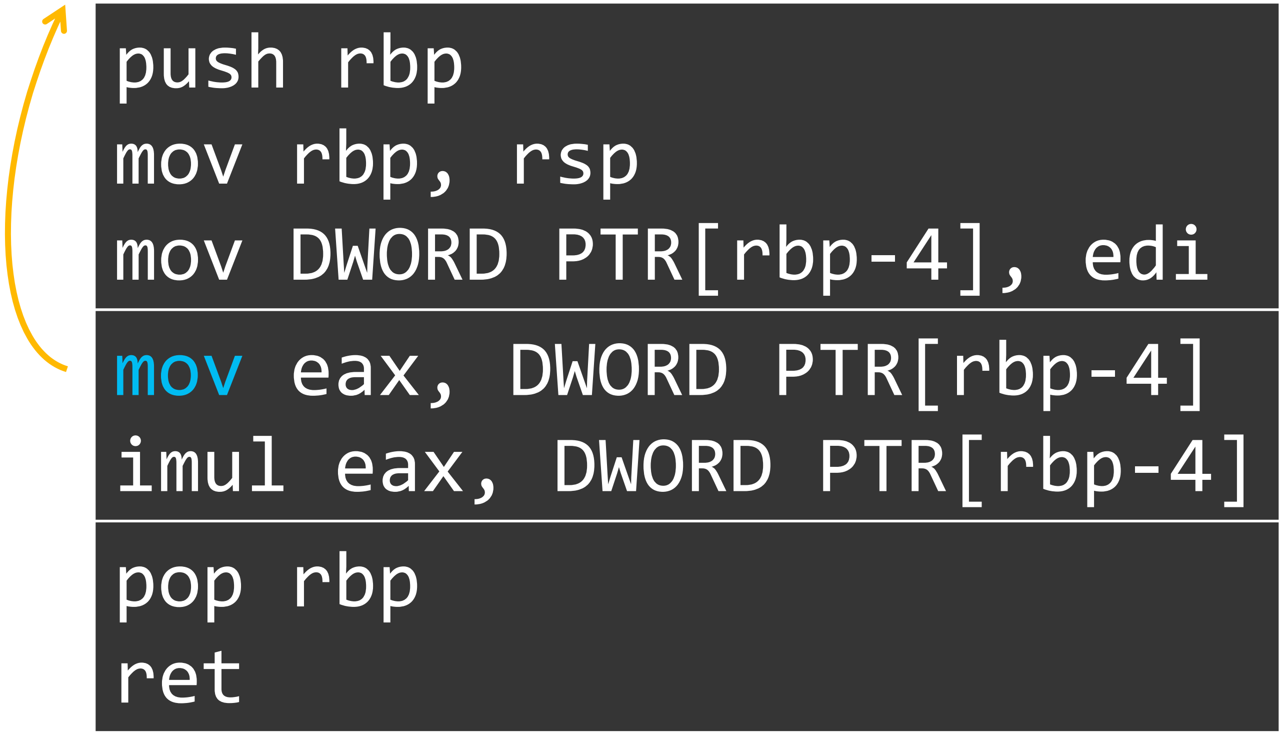
```
push rbp
mov rbp, rsp
mov DWORD PTR[rbp-4], edi

mov eax, DWORD PTR[rbp-4]
imul eax, DWORD PTR[rbp-4]

pop rbp
ret
```

1. Put back the original instruction bytes
2. Manage any other threads (if needed)

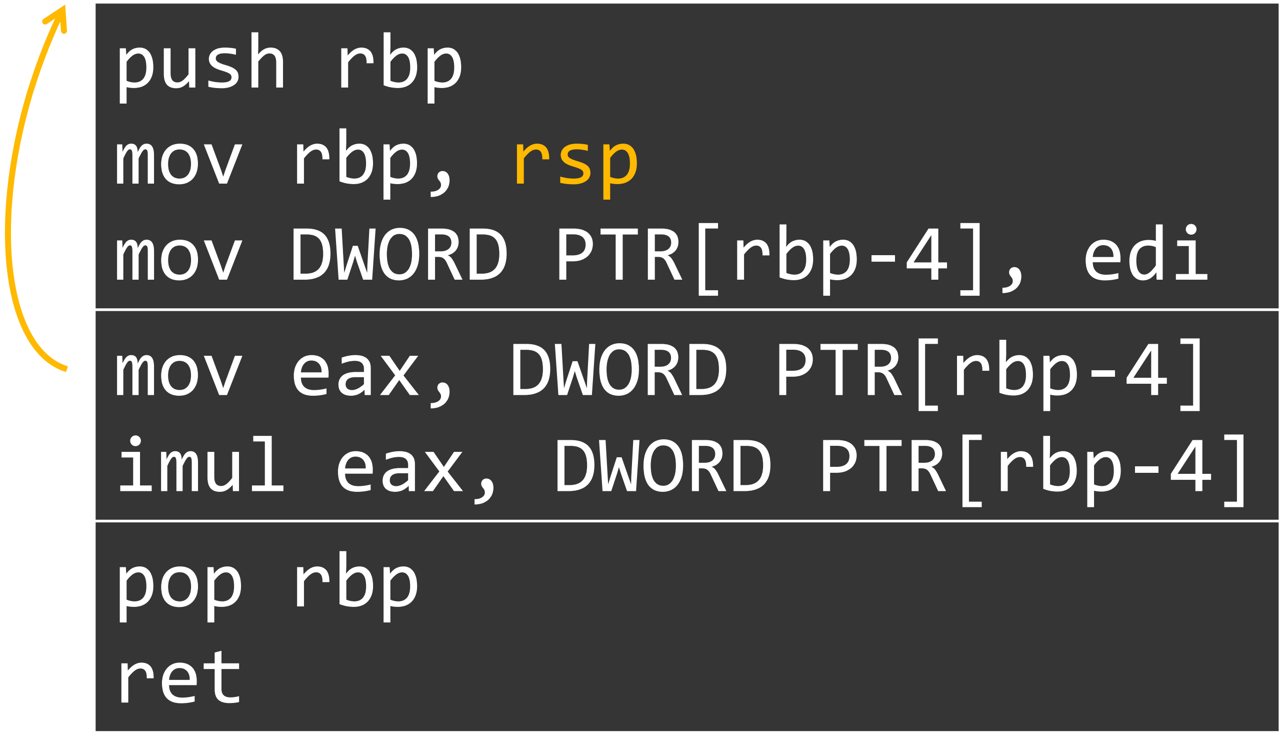
# Step out



```
push rbp
mov rbp, rsp
mov DWORD PTR[rbp-4], edi
mov eax, DWORD PTR[rbp-4]
imul eax, DWORD PTR[rbp-4]
pop rbp
ret
```

1. Put back the original instruction bytes
2. Manage any other threads (if needed)
3. Set breakpoint on return address

# Step out



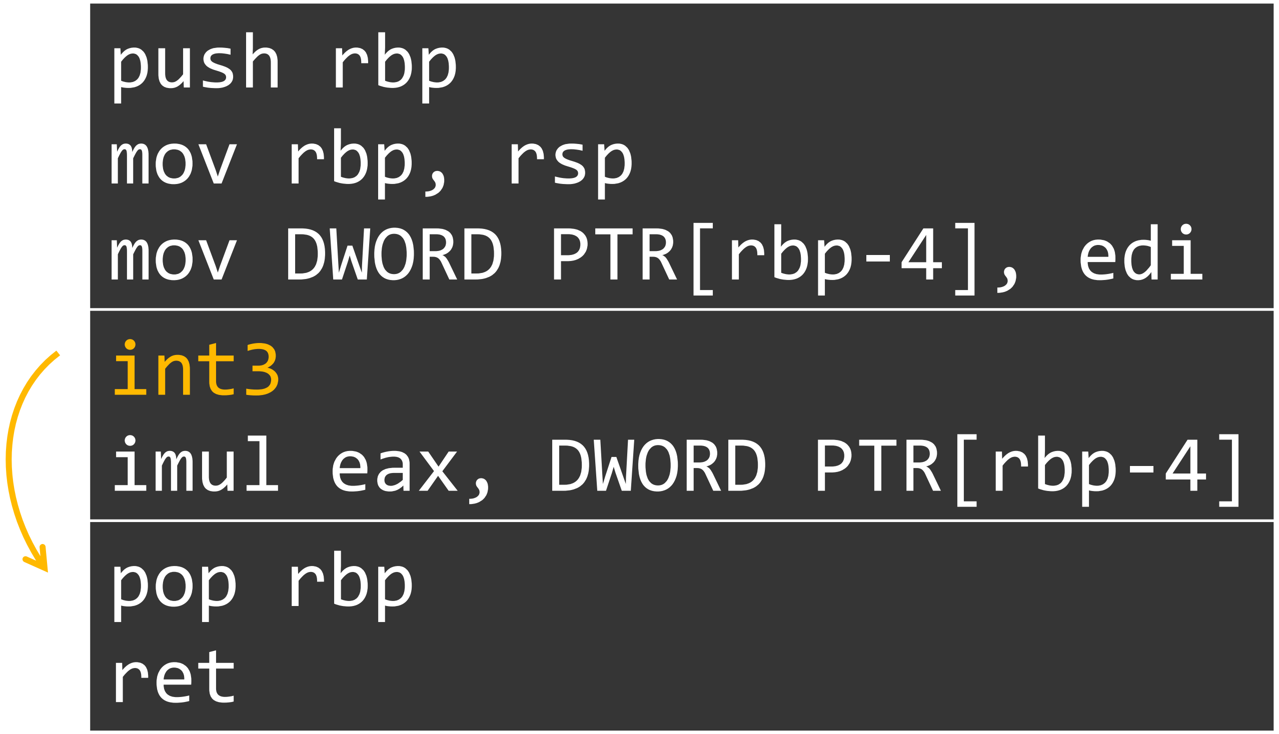
```
push rbp
mov rbp, rsp
mov DWORD PTR[rbp-4], edi
```

```
mov eax, DWORD PTR[rbp-4]
imul eax, DWORD PTR[rbp-4]
```

```
pop rbp
ret
```

1. Put back the original instruction bytes
2. Manage any other threads (if needed)
3. Set breakpoint on return address

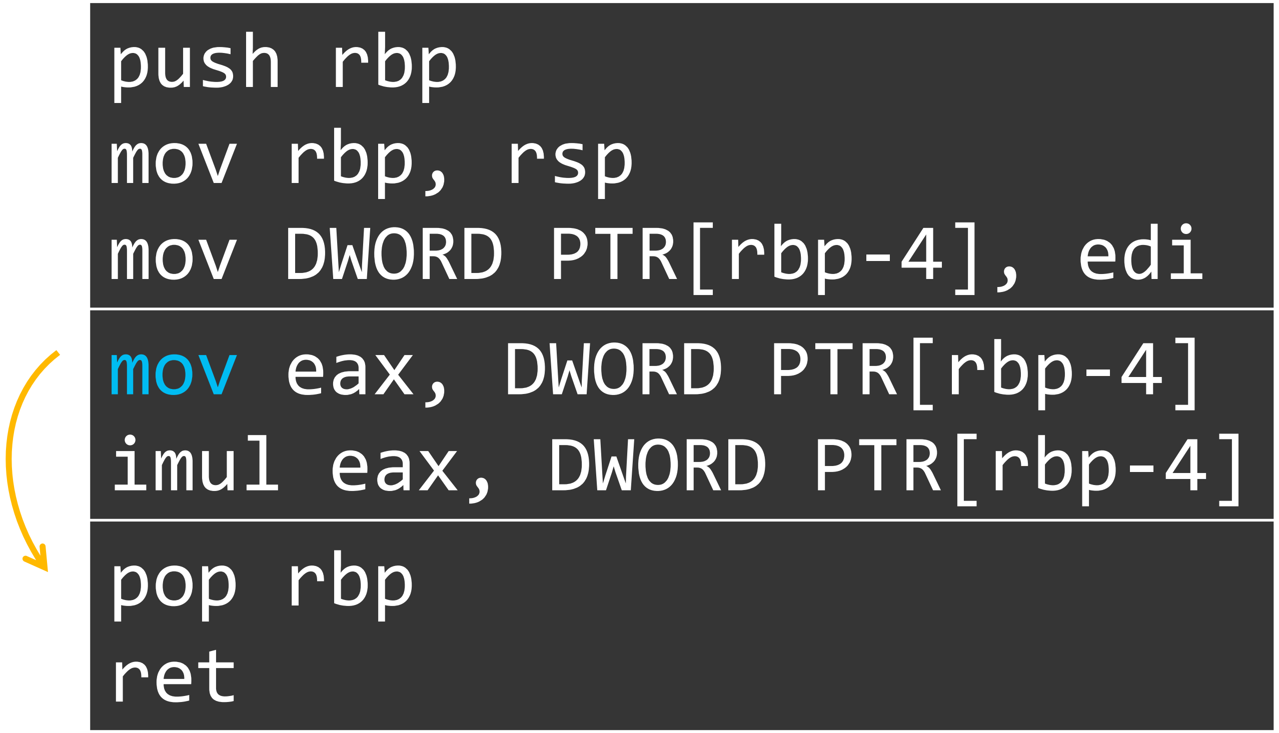
# Step over



```
push rbp
mov rbp, rsp
mov DWORD PTR[rbp-4], edi
int3
imul eax, DWORD PTR[rbp-4]
pop rbp
ret
```

1. Put back the original instruction bytes

# Step over



```
push rbp
mov rbp, rsp
mov DWORD PTR[rbp-4], edi
mov eax, DWORD PTR[rbp-4]
imul eax, DWORD PTR[rbp-4]
pop rbp
ret
```

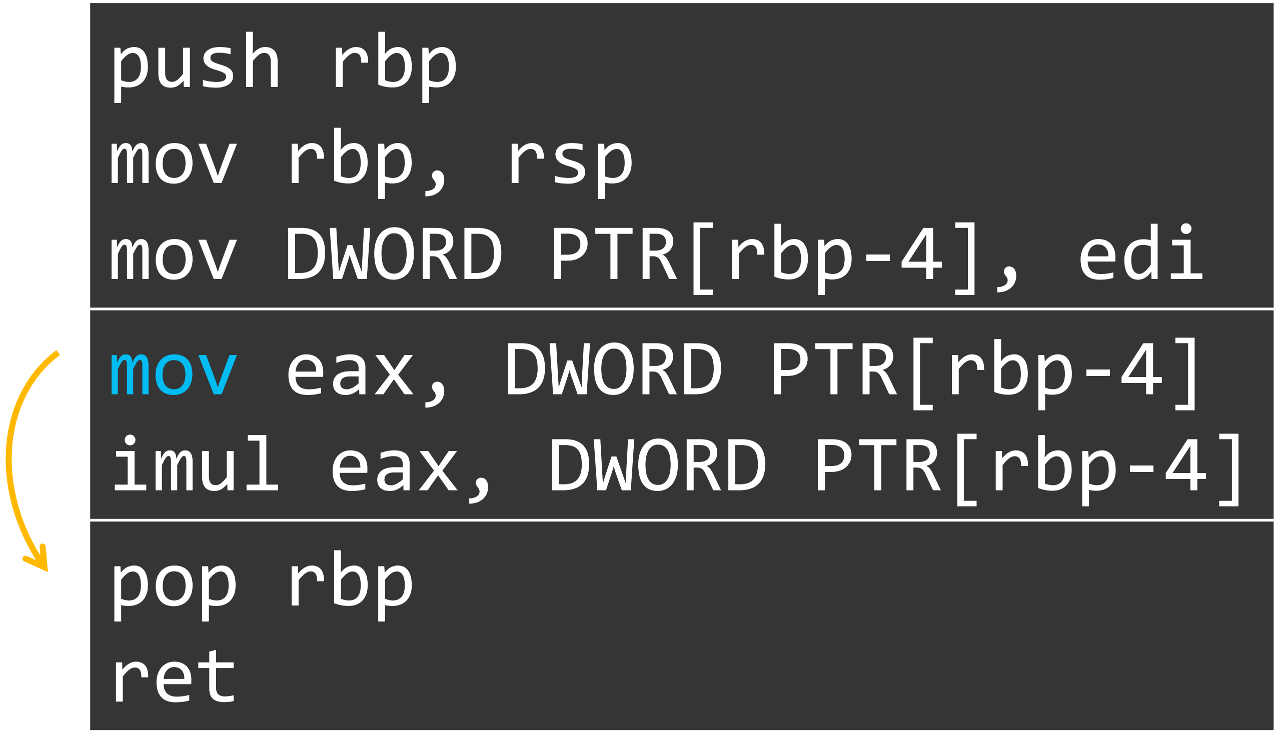
1. Put back the original instruction bytes

# Step over

```
push rbp
mov rbp, rsp
mov DWORD PTR[rbp-4], edi
mov eax, DWORD PTR[rbp-4]
imul eax, DWORD PTR[rbp-4]
pop rbp
ret
```

1. Put back the original instruction bytes
2. Manage any other threads (if needed)

# Step over



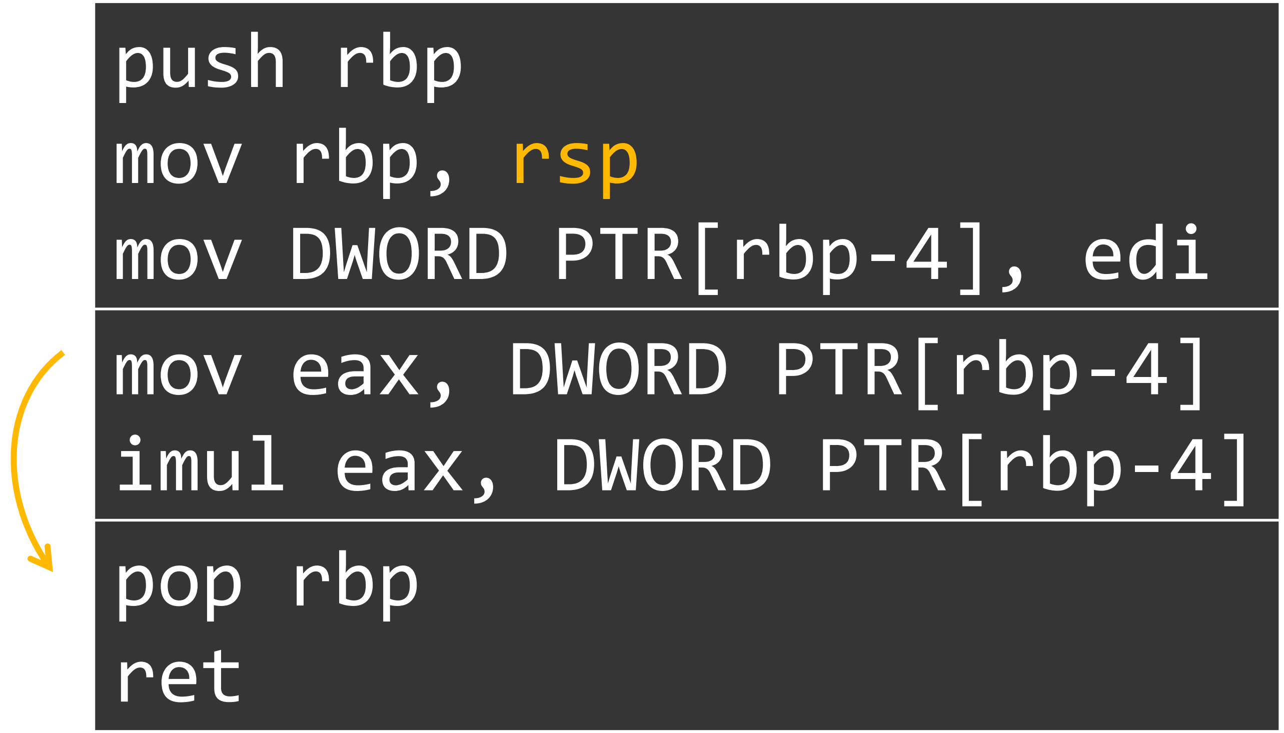
```
push rbp
mov rbp, rsp
mov DWORD PTR[rbp-4], edi

mov eax, DWORD PTR[rbp-4]
imul eax, DWORD PTR[rbp-4]

pop rbp
ret
```

1. Put back the original instruction bytes
2. Manage any other threads (if needed)
3. Set breakpoint on return address
   - In case the current instruction is actually returning out

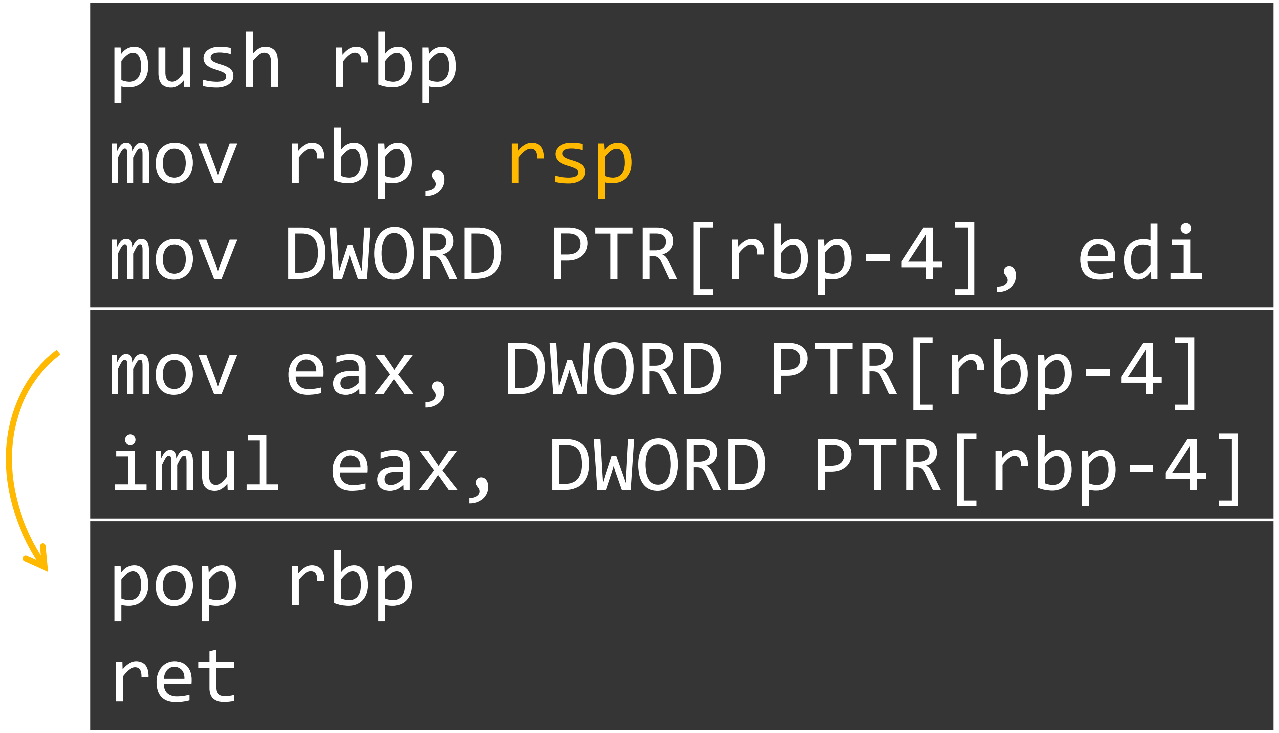
# Step over



```
push rbp
mov rbp, rsp
mov DWORD PTR[rbp-4], edi
mov eax, DWORD PTR[rbp-4]
imul eax, DWORD PTR[rbp-4]
pop rbp
ret
```

1. Put back the original instruction bytes
2. Manage any other threads (if needed)
3. Set breakpoint on return address
   - In case the current instruction is actually returning out

# Step over



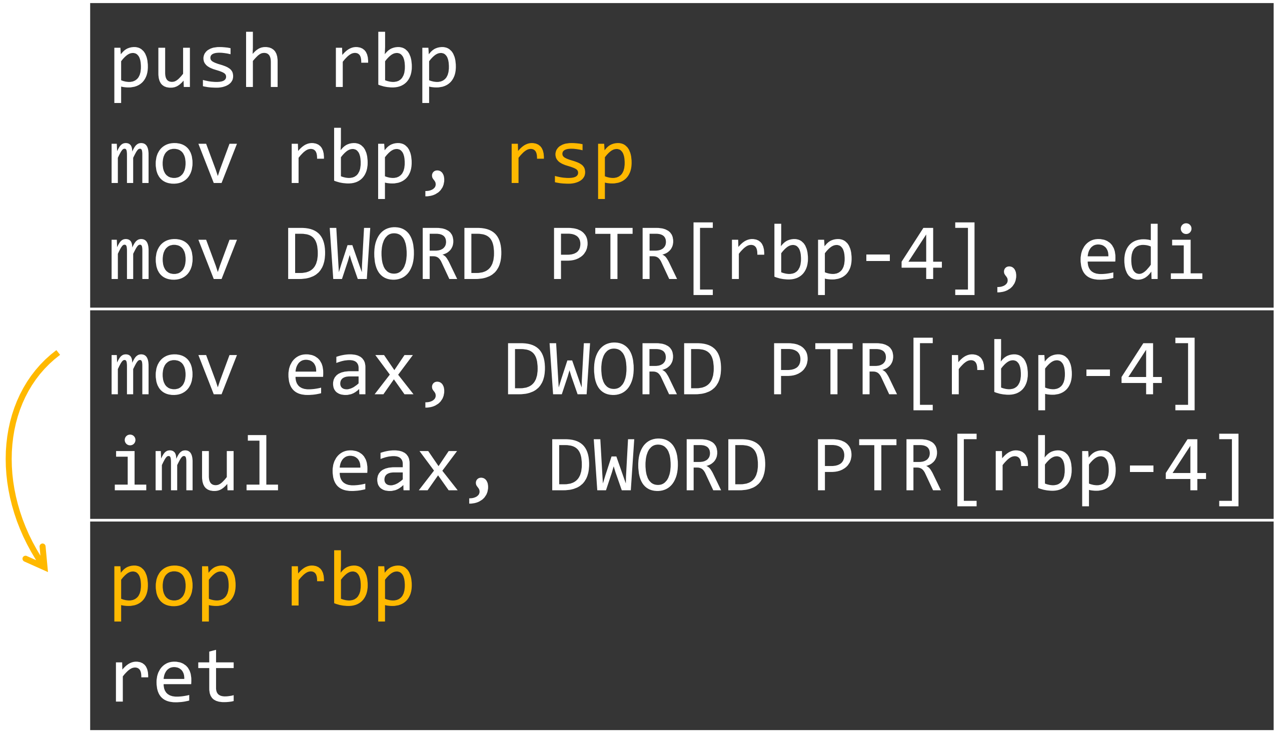
```
push rbp
mov rbp, rsp
mov DWORD PTR[rbp-4], edi
```

```
mov eax, DWORD PTR[rbp-4]
imul eax, DWORD PTR[rbp-4]
```

```
pop rbp
ret
```

1. Put back the original instruction bytes
2. Manage any other threads (if needed)
3. Set breakpoint on return address
   - In case the current instruction is actually returning out
4. Set breakpoint on next instruction
   - Within the function

# Step over



```
push rbp
mov rbp, rsp
mov DWORD PTR[rbp-4], edi
mov eax, DWORD PTR[rbp-4]
imul eax, DWORD PTR[rbp-4]
pop rbp
ret
```

1. Put back the original instruction bytes
2. Manage any other threads (if needed)
3. Set breakpoint on return address
   - In case the current instruction is actually returning out
4. Set breakpoint on next instruction
   - Within the function

# Step in

```
push rbp
mov rbp, rsp
mov DWORD PTR[rbp-4], edi
mov eax, DWORD PTR[rbp-4]
imul eax, DWORD PTR[rbp-4]
pop rbp
ret
```

1. Put back the original instruction bytes
2. Manage any other threads (if needed)
3. Set breakpoint on return address
   - In case the current instruction is actually returning out
4. Set breakpoint on next instruction
   - Within the function or callee

# Everything is out of order

Demo

# what is next?

## Stack unwinding

- Retrieve the call stack from your call frame

## Expression evaluator

- Evaluate methods or variables values at any given point

## Flow analysis

- Understand the target’s execution flow
- Find out what the next instruction is

…& a lot more

# references!

1. CppCon 2018: Simon Brand “How C++ Debuggers Work”
2. How debuggers work, by Eli Bendersky
3. Writing a Linux Debugger, by Simon Brand
4. GoingNative 28: The VS Debugger: How It Works + Tips and Tricks
5. Supercharge your Debugging in Visual Studio
6. Corkami – Reverse Engineering and visual documentations
7. Godbolt – Compiler Explorer

http://github.com/isadorasophia/debugger-demo

# Debugger secrets

Demo