ADVANCED DYNAMIC ANALYSIS

MALWARE ANALYSIS AND INCIDENT FORENSICS

M.Sc. in Cyber Security

MALWARE ANALYSIS

M.Sc. in Engineering in Computer Science

A.Y. 2025/2026

SAPIENZA
UNIVERSITÀ DI ROMA

CIS SAPIENZA
CYBER INTELLIGENCE AND INFORMATION SECURITY

MALWARE UNDER THE HOOD

- Basic dynamic analysis focuses on *externally observable* behavior
- Advanced static analysis focuses on malware internals
- Both can fall short for different reasons
  - malware evasion (e.g., analysis tools and virtual machines)
  - trigger-based behavior (e.g., packet containing a command)
  - code obfuscation and packing
  - anti-disassembly, self-modifying code

DEBUGGING

- A debugger supports the analysis of execution in different ways
  - instruction-level
  - registers and memory contents
  - function calls
  - threads and exceptions
- Ability to **inspect** and control execution provides critical insights
- Analysts can use a debugger also to **alter** the execution

DEBUGGING IN USER SPACE

- Two ways to a debugging session
  - load a program from the debugger
  - attach to a running process
- Desirable features
  - Inspect and alter registers (including EIP) and memory
  - Capture control flow transfers and specific instructions
  - Control threads (pause, resume)
  - Modify code

FOLLOWING INSTRUCTIONS

- A debugger can follow one thread at a time
- Stepping lets you execute instructions in a controlled way
  - **Single-stepping** steps through the instructions one at a time, exactly as how the CPU sees them arriving. To be used selectively on narrow portions.
- Function calls and returns are milestones in the execution
  - **Stepping into** a call instruction means that the debugger will perform single-stepping over the instructions of the called function
  - **Stepping over** a call instruction means that single-stepping mode will be resumed only when the called function returns (if it ever does so)
  - When **stepping out** of a function, the debugger suspends single-step execution and will resume it only once execution leaves the function

BREAKPOINTS

Breakpoints let the program run until a point of interest is reached

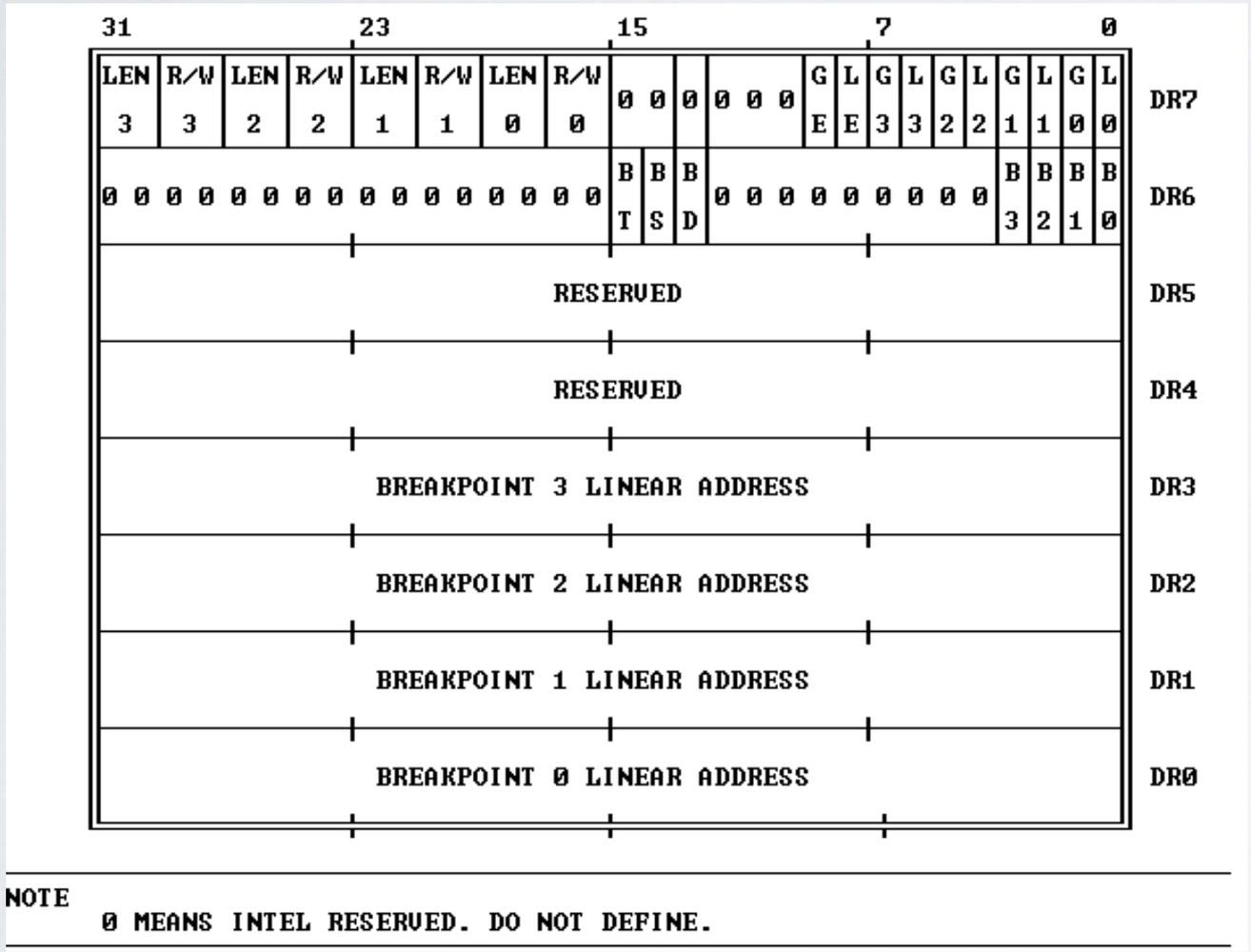
- **Instruction breakpoints** pause execution once EIP reaches the address of interest
- **Memory breakpoints** are more general and trap accesses to arbitrary memory addresses
  - on read/write access
  - on write access only
  - on execute (like an instruction breakpoint)
- **Conditional breakpoints** suspend execution only when an additional condition is matched (e.g., stop at address 0x401130 only when ECX=8)

IMPLEMENTING BREAKPOINTS

- Instruction breakpoints can be implemented via software or using hardware assistance
- A **software breakpoint** replaces the first byte of the instruction at the given address with `INT 3` (opcode `0xCC`)
  - The CPU generates a software interrupt and the debugger captures the event
  - Analysts can insert as many software breakpoints as they want
- Software breakpoints are conspicuous and malware may look for them
  - perform a checksum of `.text`
  - look for `0xCC` in its code

IMPLEMENTING BREAKPOINTS

- A **hardware breakpoint** does not alter code, but leverages the CPU debug registers to trap execution automatically



- Limited number (4 on x86)
- Harder to detect

IMPLEMENTING BREAKPOINTS

- Conditional breakpoints evaluate a user-provided condition before entering single-stepping mode
  - Very useful on loops (e.g., break only after N iterations)
  - Useful also for function calls (e.g., argument-related condition)
  - Careful: they slow down execution also when they don't enter single-stepping
- Memory breakpoints can be software or hardware-based
  - Hardware: up to 4 bytes (dword size) per registered breakpoint
  - Software: they change memory protection attributes (not 100% reliable)
  - Very useful when dealing for example with packed malware

EXCEPTIONS

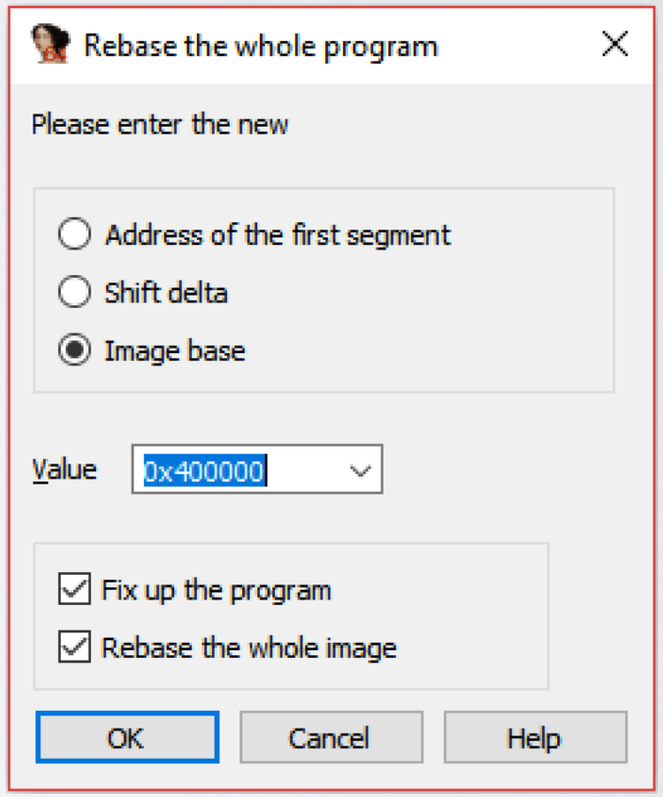
- Exceptions are a mechanism for supporting alternative control flow (e.g., to recover from errors)
  - From the CPU perspective, they are an interrupt (just like the `int` instruction)
  - Debuggers use them to regain control of the execution
    - Instruction/memory breakpoints
    - Single-stepping when trap flag TF in EFLAGS is set
- To handle an exception, a corresponding handler must have been registered
  - Benign programs use them to handle error situations (e.g., division by zero)
  - Malware analysts use them to hinder analysis (for instance, if the debugger catches an exception meant for the malware, it can give away its presence)

EXCEPTIONS

- Exceptions are passed to the debugger as they occur. A debugger can catch an exception up to **two** times:
  - Some APIs behave differently when executing under a debugger, raising exceptions to inform the user
  - *First-chance* exceptions can be **passed** to the program when they are unrelated to debugging (e.g., floating point or division-by-zero error)
  - When the program cannot recover from the exception (e.g., there is no corresponding registered handler), the debugger receives it again
  - *Second-chance* exceptions require intervention or the program will crash
  - Malware often uses exceptions as an anti-debug technique

DEBUGGING AND ASLR

- With ASLR, changing in addresses can slow down debugging
  - For programs compiled with /DYNAMICBASE Windows randomizes the base address for .text at every execution. Hence, the addresses that you see in the disassembly may not match what you see in a debugger
  - IDA offers manual loading for a PE and, better, a **rebase** option for the current session (Edit->Segments->Rebase Program)
  - Alternatively, you can modify the PE header using CFF Explorer (OptionalHeader->Dll Characteristics->DLL can move), but some Windows 10+ settings may override this



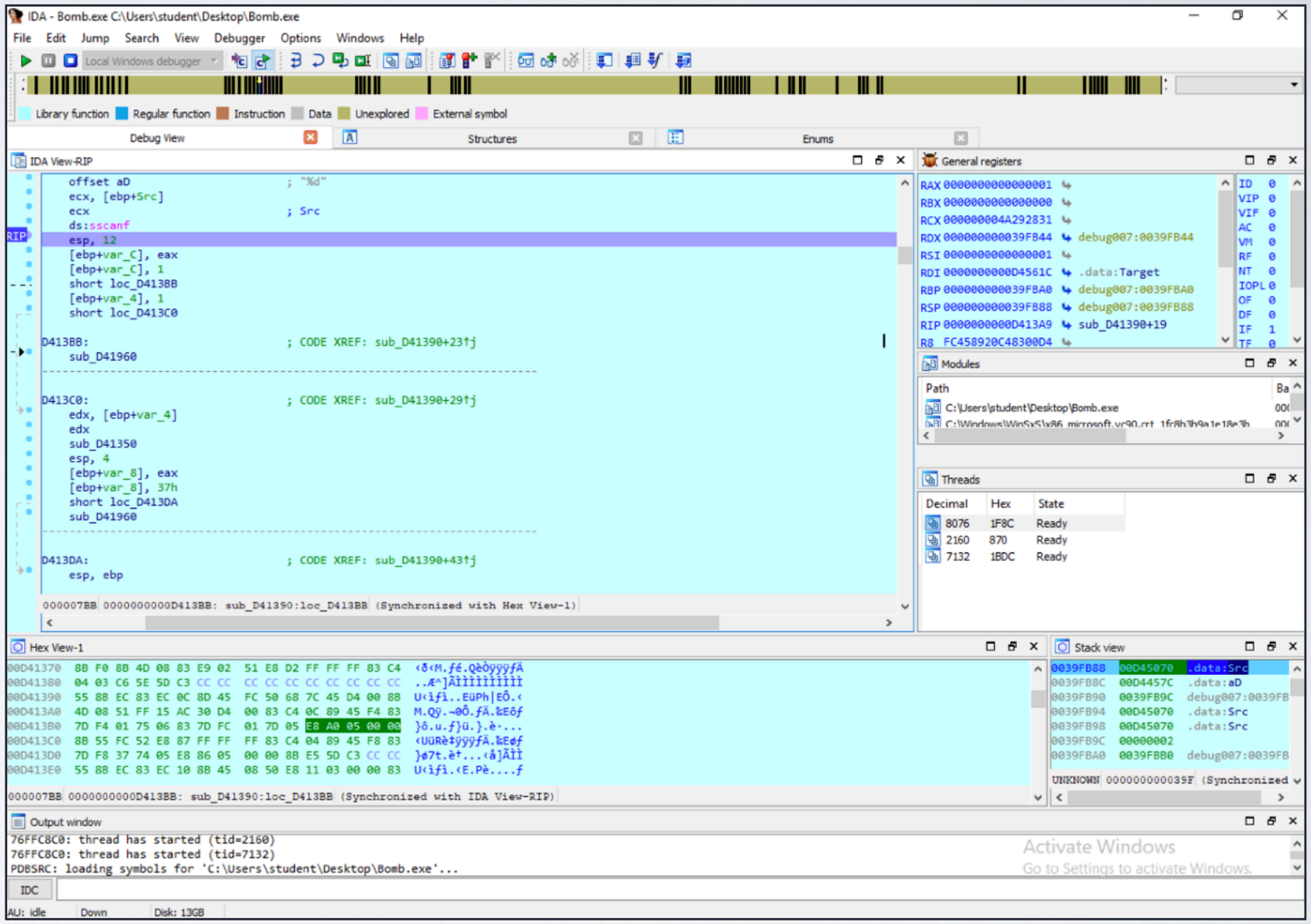
POPULAR DEBUGGERS

- Many debuggers exist. Some are particularly well suited for malware analysis and reverse engineering
  - WinDbg
  - IDA Pro and **IDA Freeware**
  - OllyDbg, x64dbg
  - Immunity Debugger
  - radare2 with plugins

IDA'S DEBUGGER

- Recent versions of IDA Freeware include a debugger
  - Lacks integration with third-party tools like Scylla and some advanced scripting features

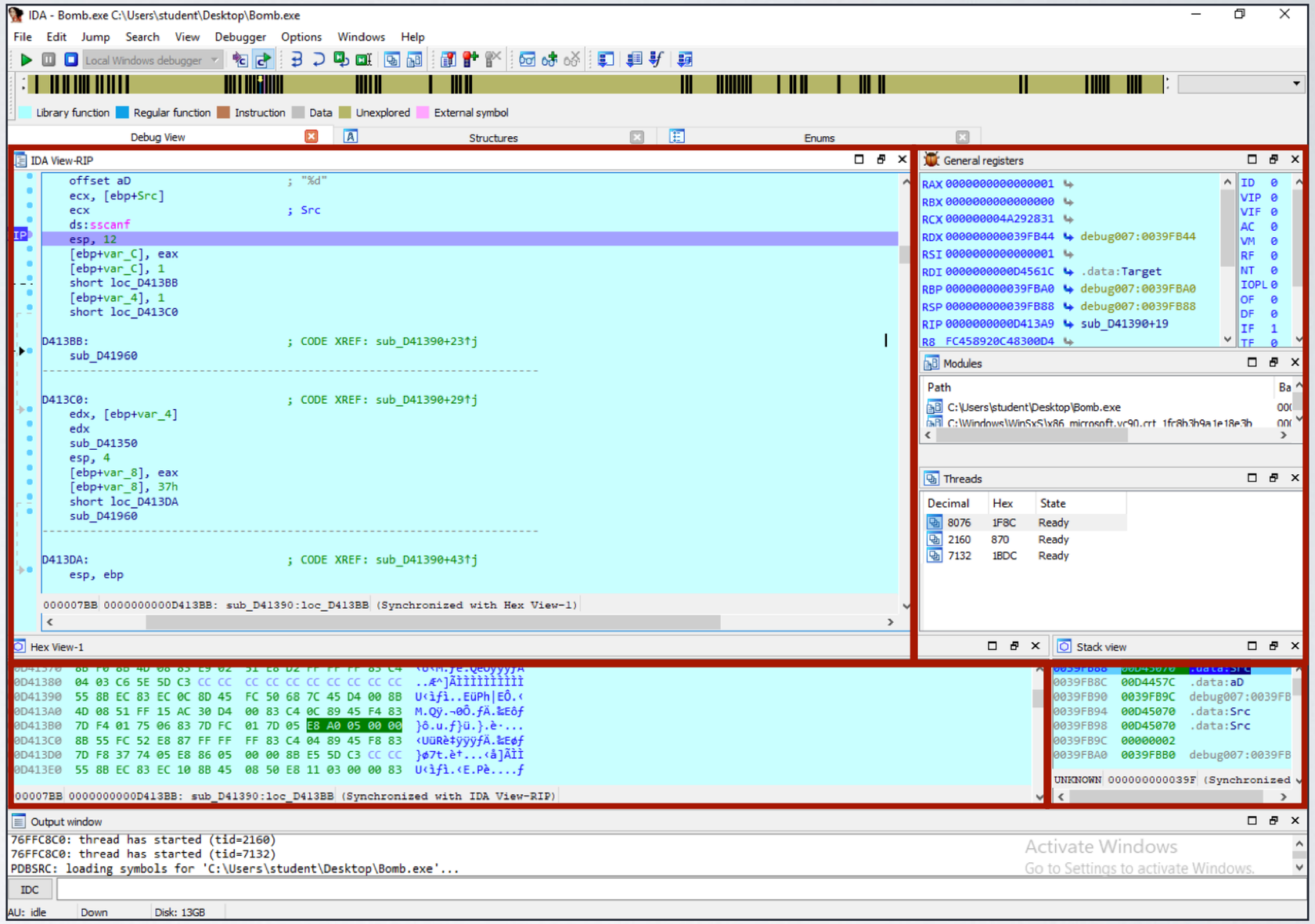
    (so you may still consider using x64dbg sometimes!)



MAIN INTERFACE

Four quadrants

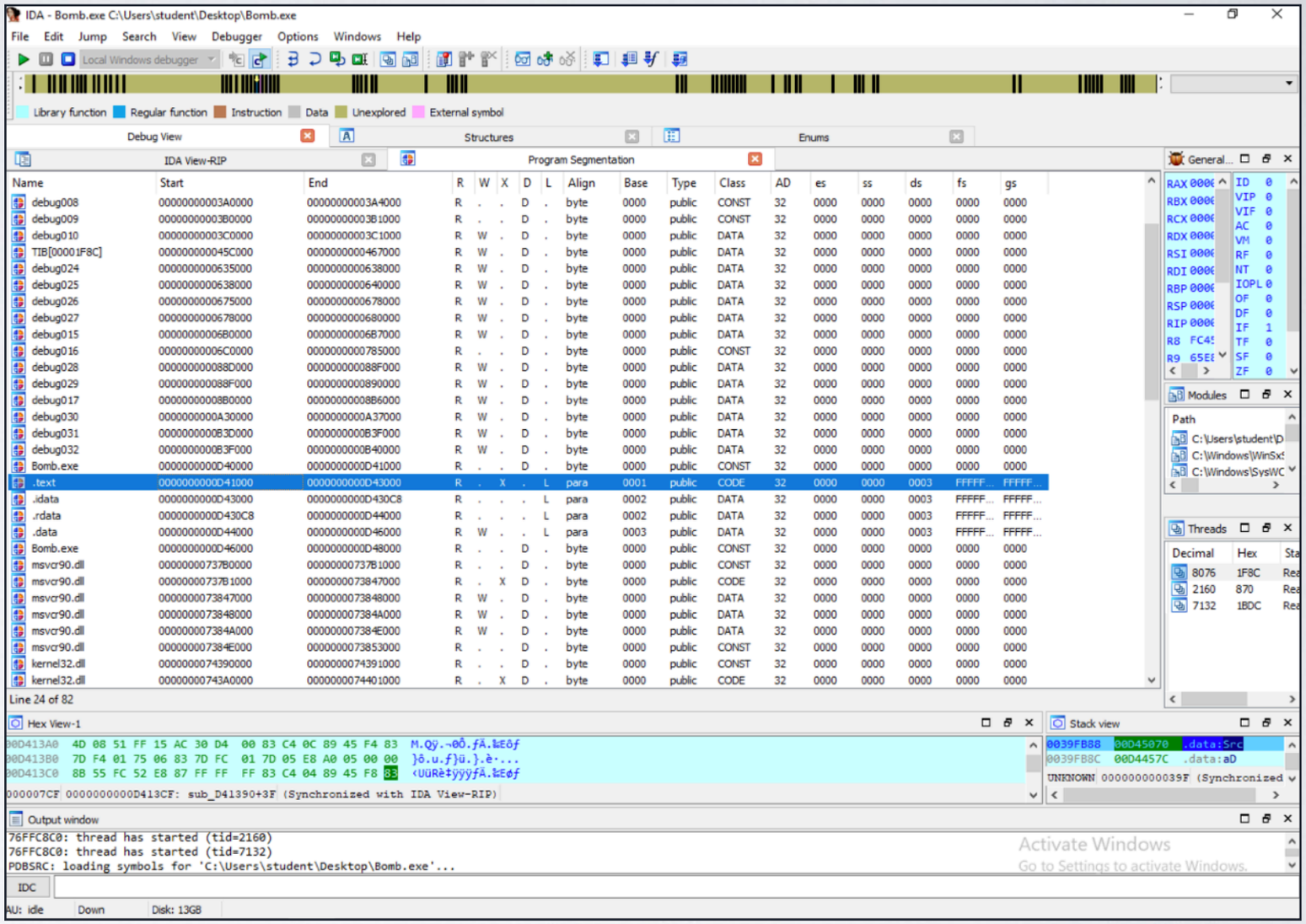
1. Disassembler window
2. Registers window
3. Stack window
4. Memory dumps window



MEMORY MAP

Layout information can be very valuable in the analysis

- Writable+executable regions may hint at packing or self-modifying code
- A changed protection for a region may suggest that as well
- Lists load address for DLLs
- Clicking on a base address will update the disassembler (code) or the memory dump (data) view

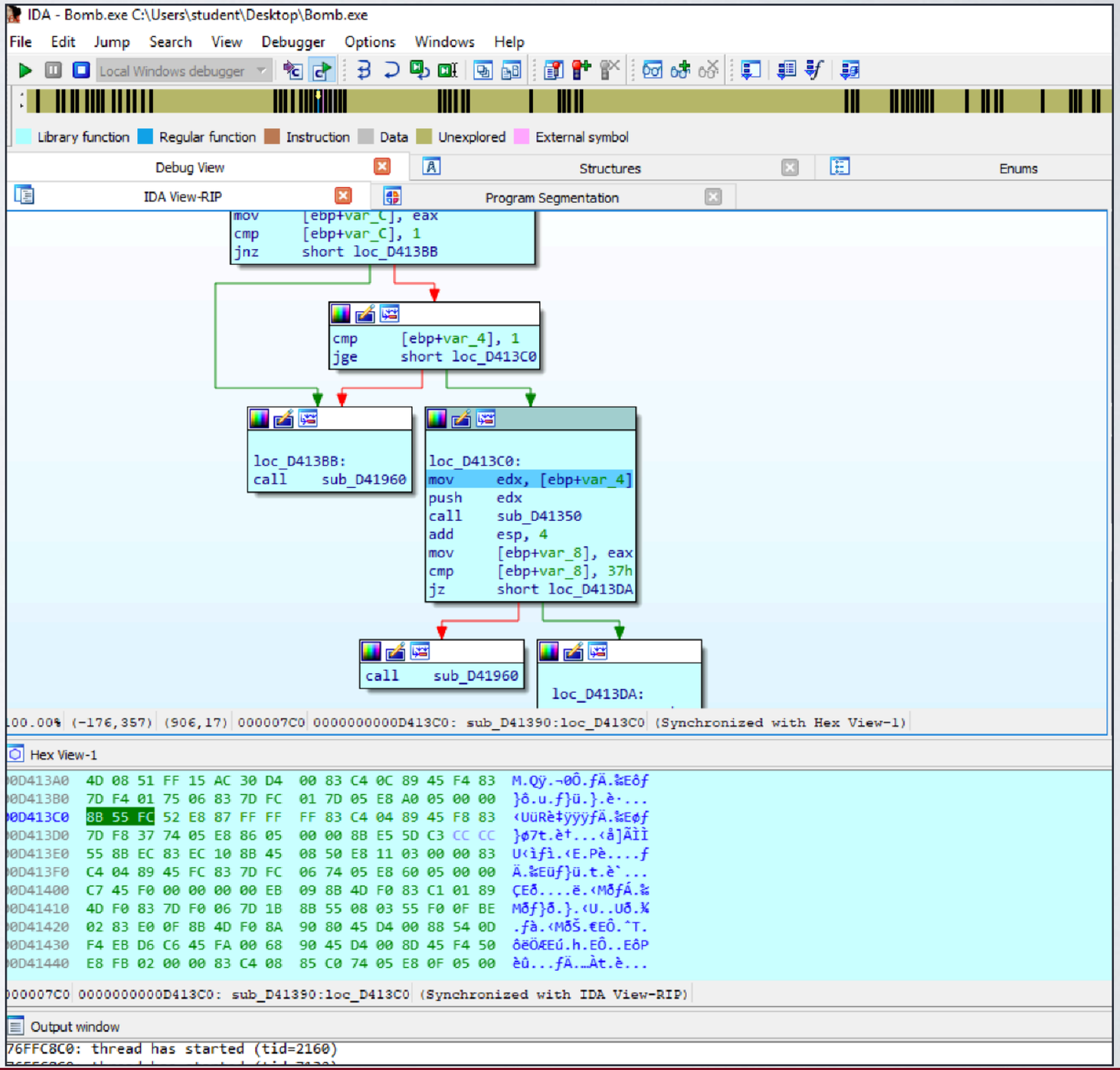


PANES

- IDA offers several panes besides CPU and Memory Map
  - Graph
  - Breakpoints
  - Call stack
  - SEH (Exception handling)
  - Symbols
  - Threads
  - and others…

GRAPH PANE

- By right-clicking or pressing SPACE on an instruction address in the Disassembler window, we can toggle between the classic IDA view and the “flat” disassembly one



CALL STACK AND THREADS

- The Call Stack pane shows the sequence of routine calls currently active on the stack, while the Threads pane lets the analyst control the state of each thread

IDA View-RIP | Call Stack | Program Segmentation

Address	Module	Function
0000000000D413C0	Bomb.exe	int __cded sub_D41390(char *Src)
0000000000D41127	Bomb.exe	sub_D41000+127
0000000000D41BA4	Bomb.exe	strlen+0x168
00000000743A8482	kernel32.dll	kernel32_BaseThreadInitThunk+22
0000000077012FE8	ntdll.dll	0000000077012FE8
0000000077012FB5	ntdll.dll	0000000077012FB5

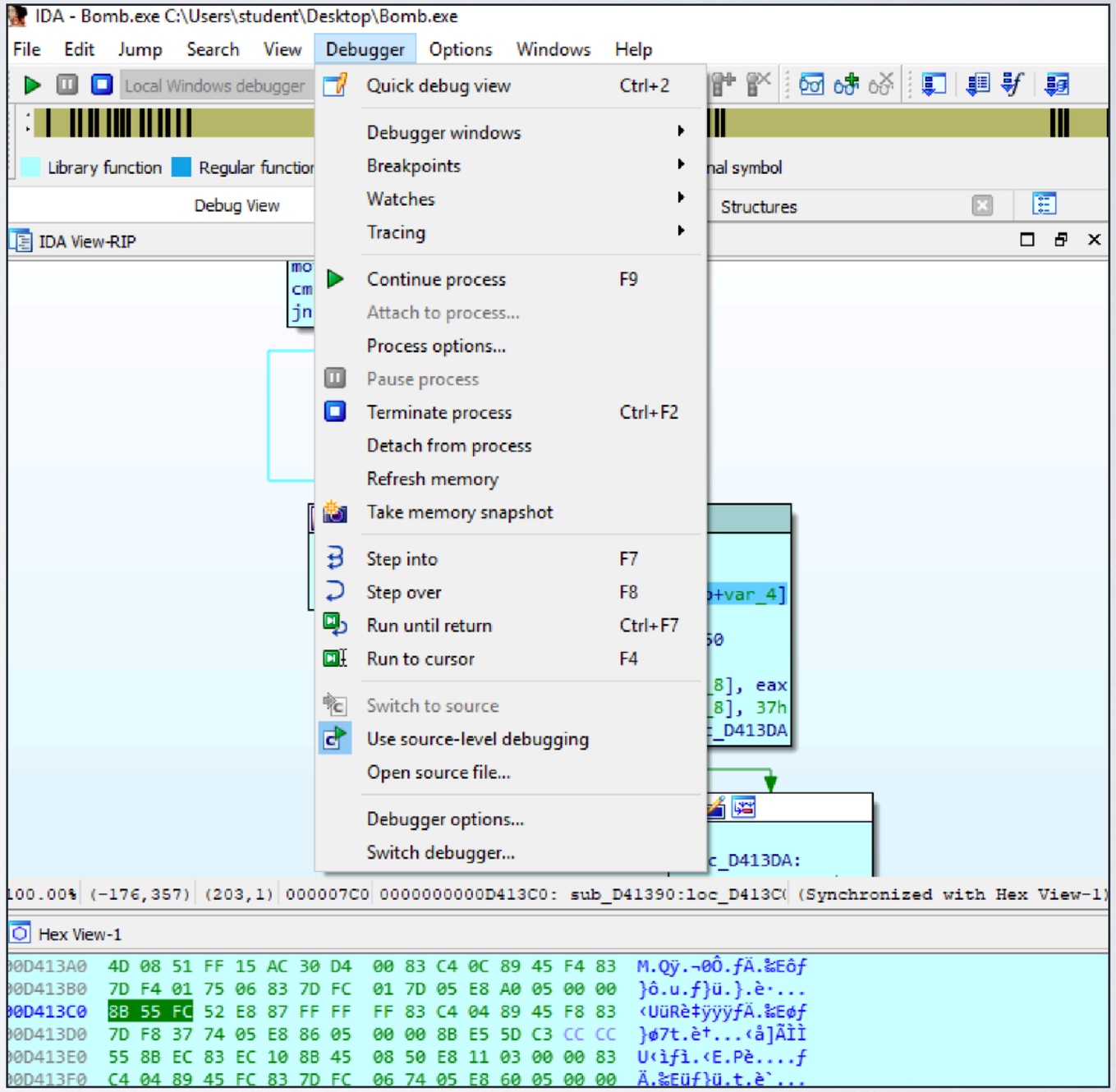
Threads

Decimal	Hex	State
8076	1F8C	Ready
2160	870	Ready
7132	1BDC	Ready

MANAGING STEPPING

Fine-grained control of the execution

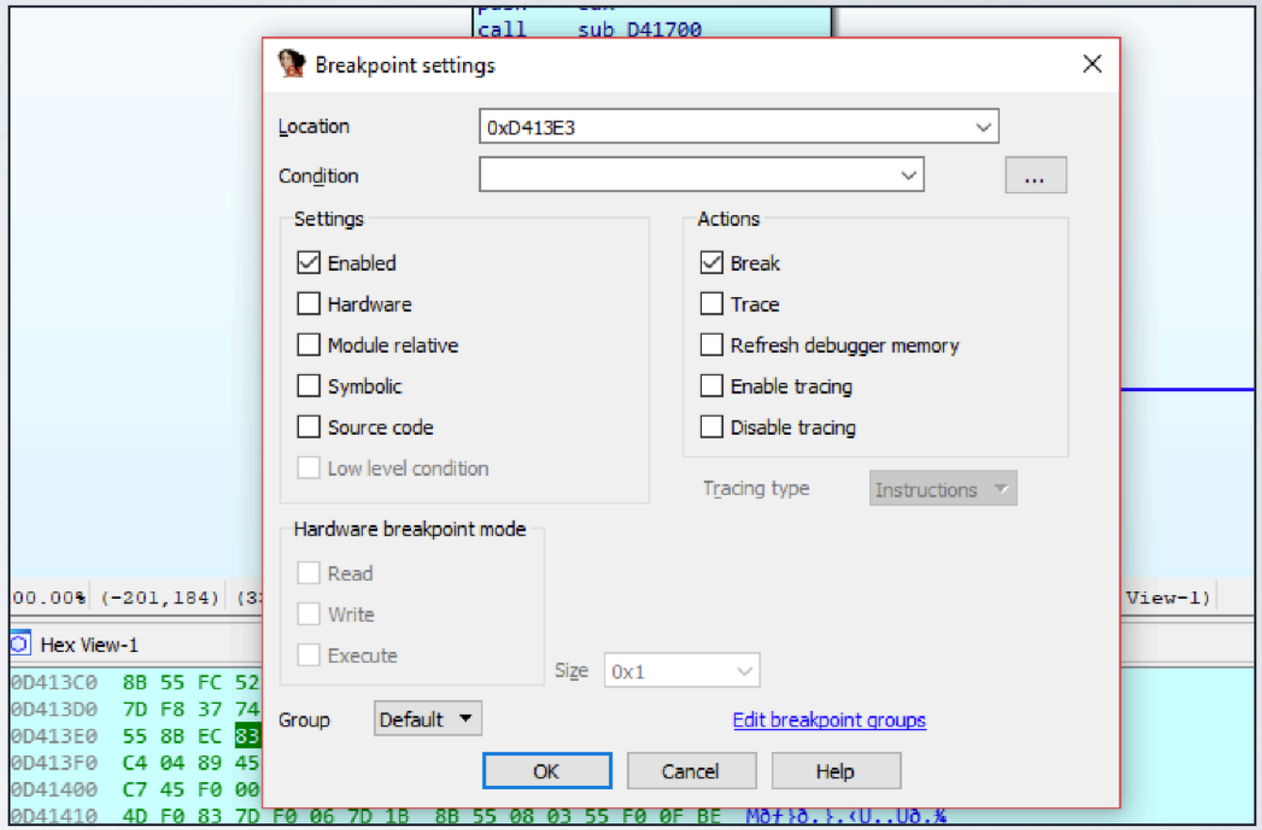
- Run until a breakpoint is hit
- Step into and step over for single-step
- Execute till return
- Restart the execution

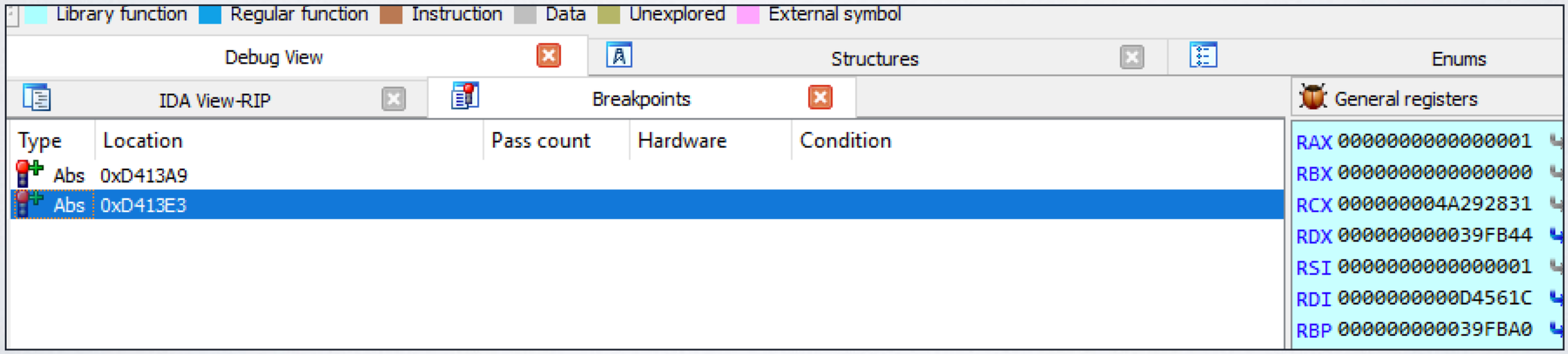


CONTROLLING BREAKPOINTS

Intuitive instruction breakpoints

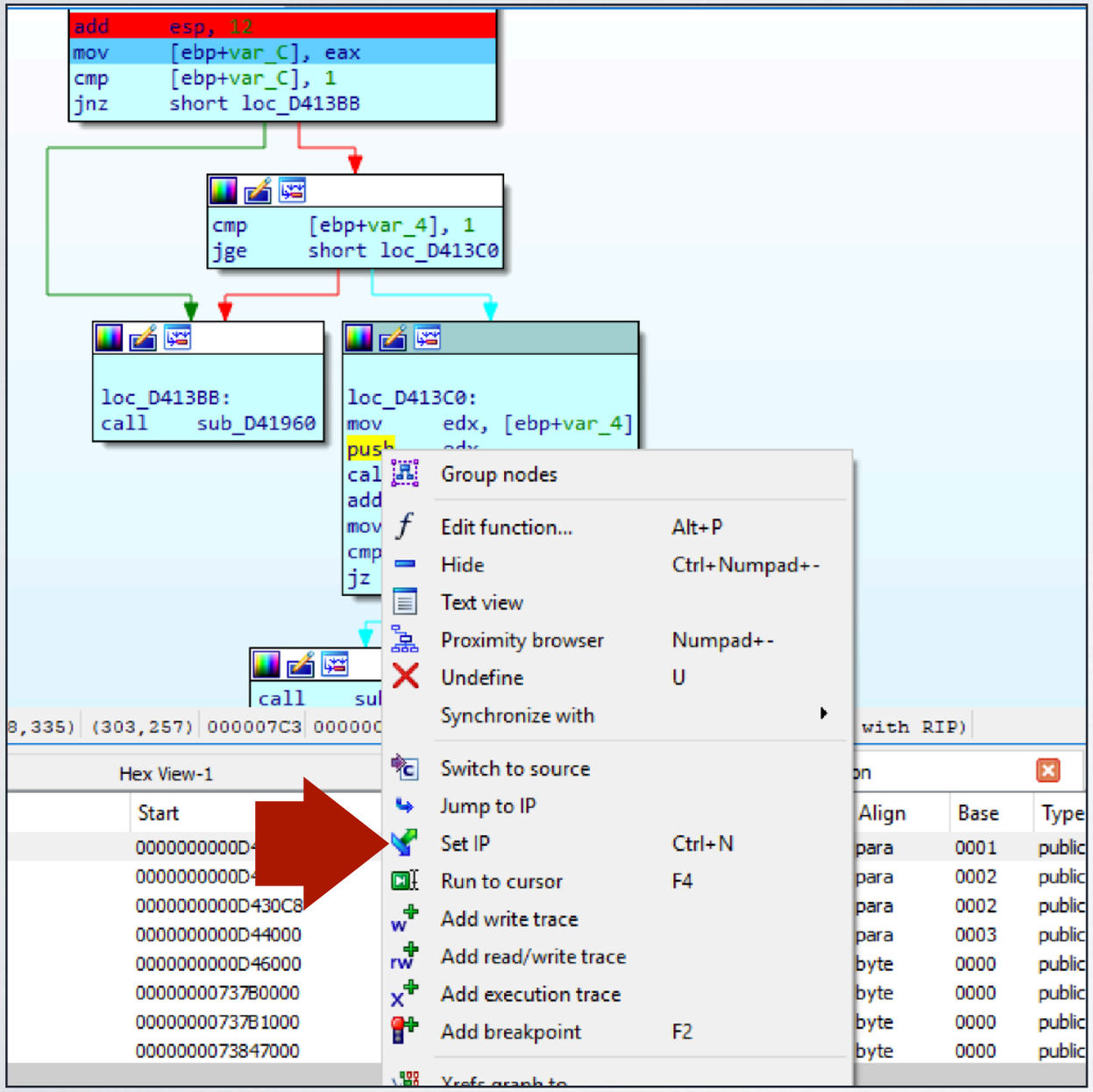
- Toggle software breakpoints (on/off): F2
- Can later enable **hardware** assistance
- Breakpoints can be controlled from pane (e.g., enable/disable/remove all)
- **Hit count** for each breakpoint
- Memory breakpoints are similar

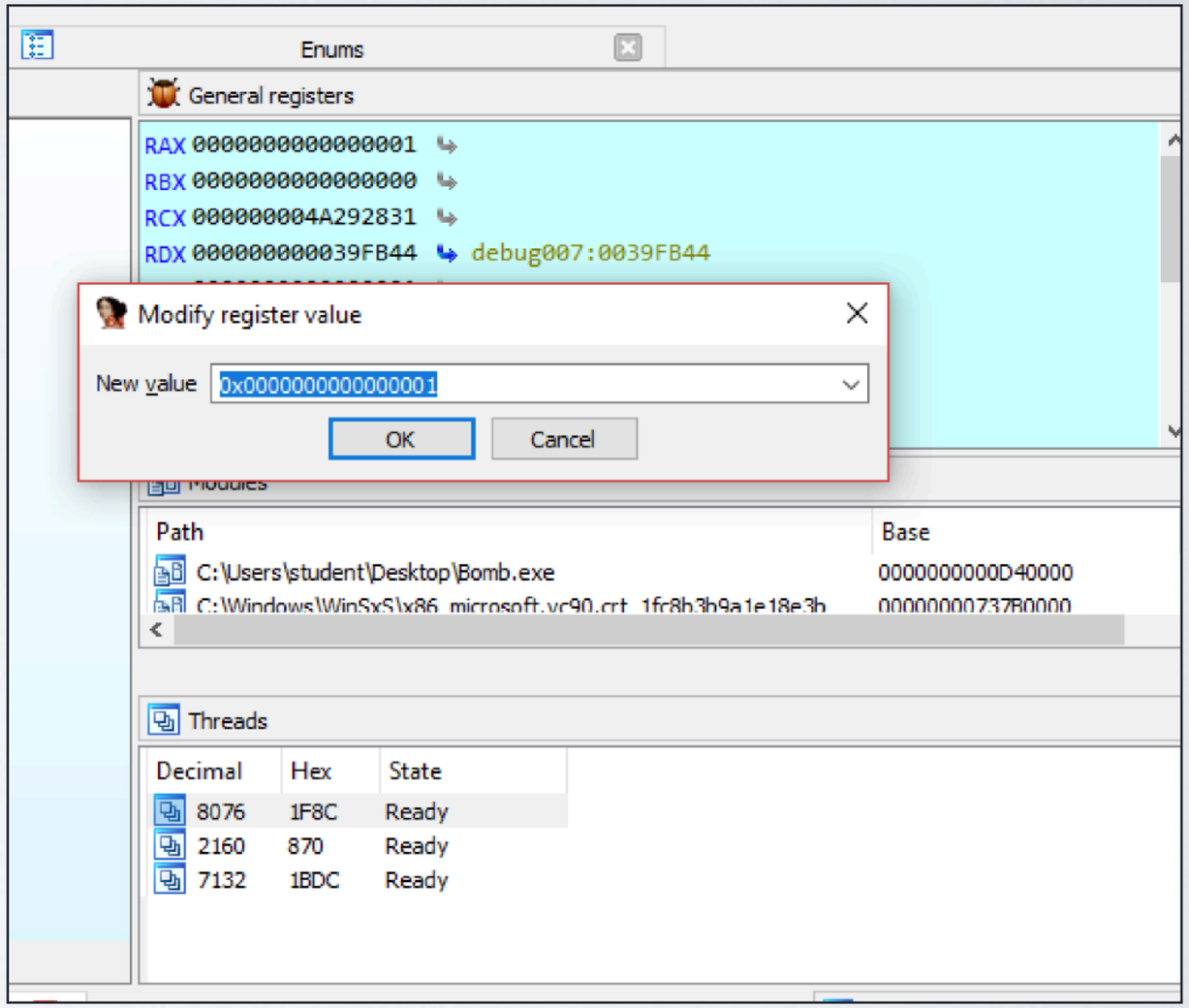




MODIFYING REGISTERS

We can edit general-purpose registers (e.g., ECX) or change the instruction pointer

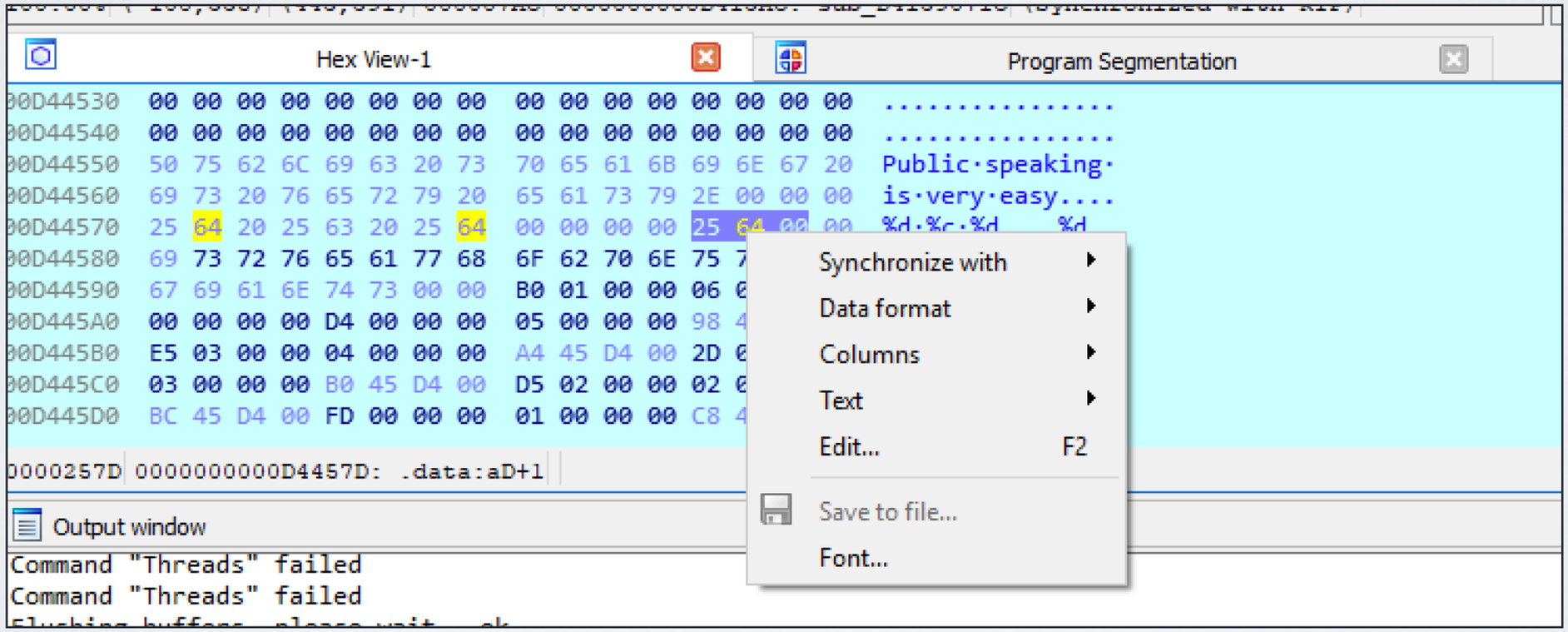




MODIFYING MEMORY

We can alter memory easily: we can follow an address on stack first (e.g., to dereference a pointer) or directly on dump (e.g., ASCII data), then modify the corresponding data.

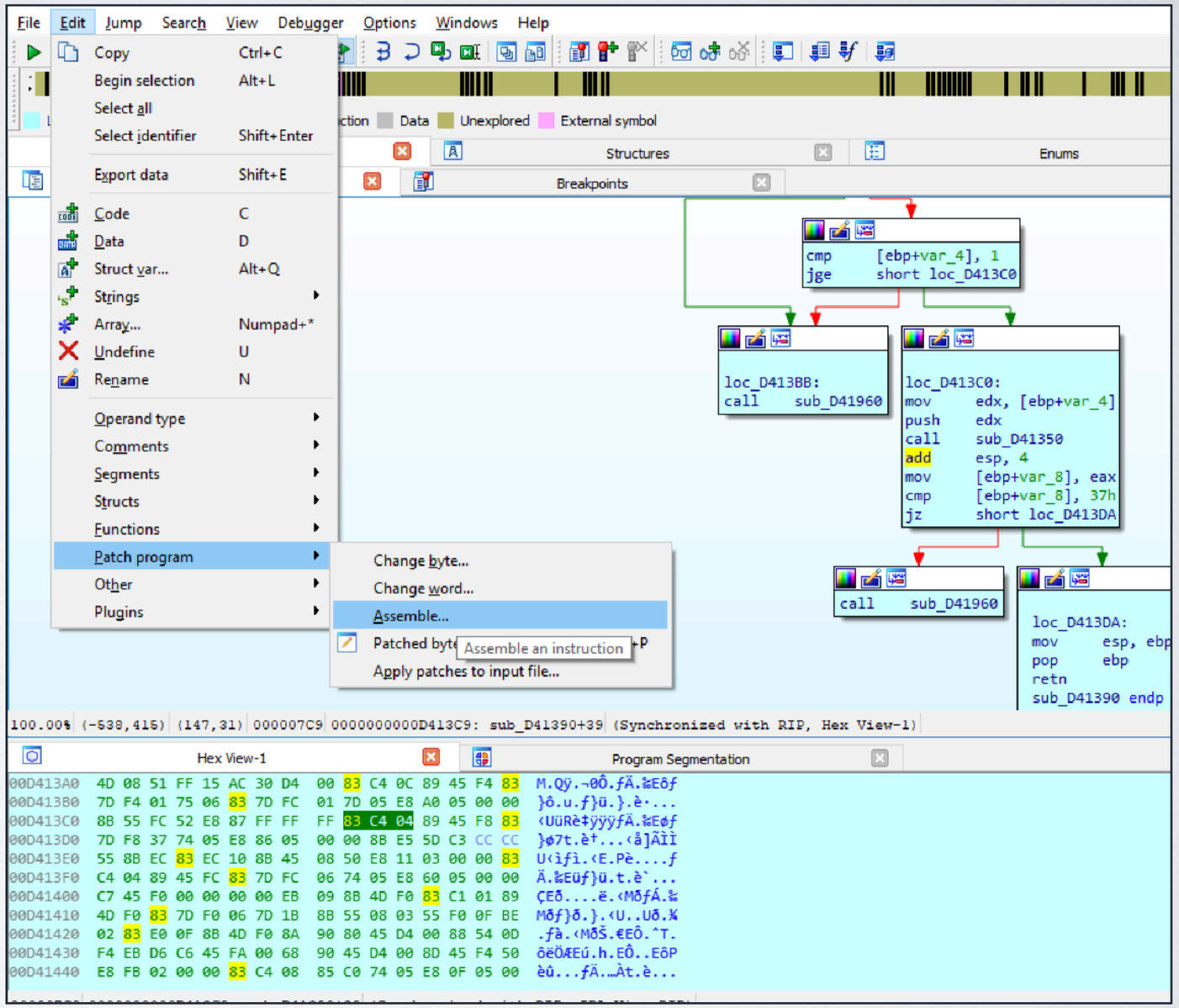
Experience will tell you every time!



MODIFYING CODE

We can also modify code, provided that there is enough space to host the desired new instruction(s).

- Changes by default do not survive restarts. But they can be imported/ exported/made permanent using patches (File->Patches)
- Beware of possible integrity checks
- Ensure stack pointer validity w.r.t. frame layout when deleting a function call!



COMMON IDA SHORTCUTS

- Learn shortcuts to use IDA more efficiently
  - Escape - go back to previous position in current view
  - Ctrl+Enter - go forward to next position in current view
  - Space - switch between text and graph mode
  - G - jump to address in current view
  - F2
    - (during static code analysis) set a breakpoint at the current instruction
    - (during debugging) enable edit mode to change selected value, then F2 again to apply changes
  - X - show cross referencing info for selected function entrypoint
  - N - rename selected reference
- https://docs.hex-rays.com/user-guide/configuration/shortcuts
  (some commands may be available in Pro version only)