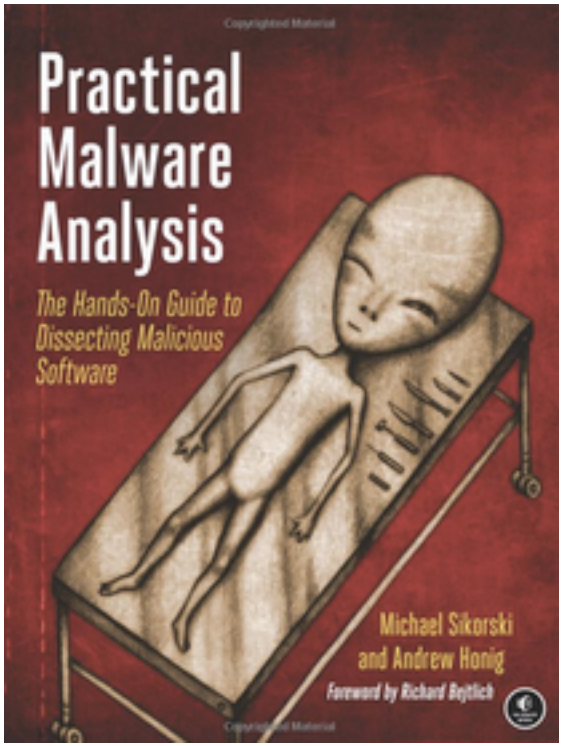
Practical Malware Analysis

Ch 9: OllyDbg



Updated 9-14-21

History

- OllyDbg was developed more than a decade ago
- First used to crack software and to develop exploits
- The OllyDbg 1.1 source code was purchased by Immunity and rebranded as Immunity Debugger
- The two products are very similar

Loading Malware

Ways to Debug Malware

- You can load EXEs or DLLs directly into OllyDbg
- If the malware is already running, you can attach OllyDbg to the running process

Opening an EXE

- File, Open
- Add command-line arguments if needed
- OllyDbg will stop at the entry point, WinMain, if it can be determined
- Otherwise it will break at the entry point defined in the PE Header
  - Configurable in Options, Debugging Options

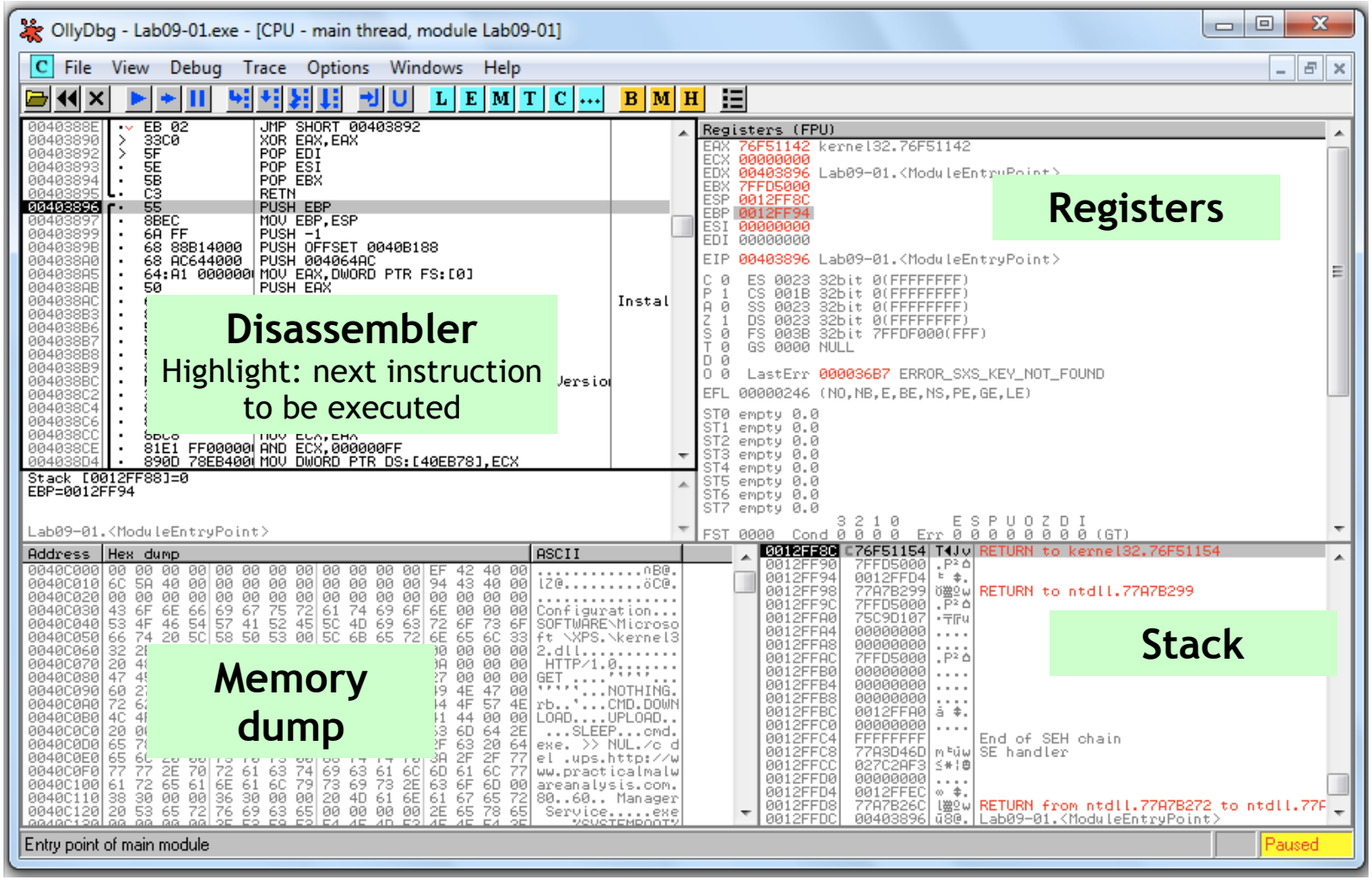
Attaching to a Running Process

- File, Attach
- OllyDbg breaks in and pauses the program and all threads
  - If you catch it in DLL, set a breakpoint on access to the entire code section to get to the interesting code

Reloading a File

- Ctrl+F2 reloads the current executable
- F2 sets a breakpoint

The OllyDbg Interface


OllyDbg - Lab09-01.exe - [CPU - main thread, module Lab09-01]
File View Debug Trace Options Windows Help
Registers
Disassembler
Highlight: next instruction to be executed
Memory dump
Stack
Entry point of main module
Paused

Modifying Data

- Disassembler window
  - Press **spacebar**
- Registers or Stack
  - Right-click, modify
- Memory dump
  - Right-click, Binary, Edit
  - Ctrl+G to go to a memory location
  - Right-click a memory address in another pane and click "Follow in dump"

Memory Map

View, Memory Map

Memory map

Address	Size	Owner	Section	Contains	Type	Access	Initial	Mapped as
00010000	00010000				Map	RW	RW	
00020000	00010000				Map	RW	RW	
0012D000	00001000				Priv	RW Gua	RW Gua	
0012E000	00002000			Stack of main thr	Priv	RW	RW	
00130000	00004000				Map	R	R	
00140000	00001000				Priv	RW	RW	
00150000	00067000				Map	R	R	\Device\HarddiskVolume1\Windows\System32\locale.nls
001C0000	00001000				Priv	RW	RW	
001D0000	00001000				Priv	RW	RW	
00240000	00003000				Priv	RW	RW	
002A0000	00008000				Priv	RW	RW	
00400000	00001000	Lab09-01		PE header	Img	R	RWE Cop	
00401000	0000A000	Lab09-01	.text	Code	Img	R E	RWE Cop	
0040B000	00001000	Lab09-01	.rdata	Imports	Img	R	RWE Cop	
0040C000	00005000	Lab09-01	.data	Data	Img	RW Cop	RWE Cop	
00420000	00005000				Map	R	R	
004E0000	00003000				Map	R	R	
004F0000	00101000			GDI handles	Map	R	R	
00600000	0008B000				Map	R	R	
75C60000	00001000	KERNELBA		PE header	Img	R	RWE Cop	
75C61000	00044000	KERNELBA			Img	R E	RWE Cop	
75CA5000	00002000	KERNELBA			Img	RW	RWE Cop	
75CA7000	00004000	KERNELBA			Img	R	RWE Cop	
75EB0000	00001000	NSI		PE header	Img	R	RWE Cop	
75EB1000	00002000	NSI			Img	R E	RWE Cop	
75EB3000	00001000	NSI			Img	RW	RWE Cop	
75EB4000	00002000	NSI			Img	R	RWE Cop	
75EC0000	00001000	SHELL32		PE header	Img	R	RWE Cop	
75EC1000	003C8000	SHELL32			Img	R E	RWE Cop	
76289000	00007000	SHELL32			Img	RW Cop	RWE Cop	
76290000	00879000	SHELL32			Img	R	RWE Cop	
76B10000	00001000	USER32		PE header	Img	R	RWE Cop	
76B11000	00068000	USER32			Img	R E	RWE Cop	
76B79000	00001000	USER32			Img	RW	RWE Cop	
76B7A000	0005F000	USER32			Img	R	RWE Cop	
76BE0000	00001000	sechost		PE header	Img	R	RWE Cop	
76BE1000	00013000	sechost			Img	R E	RWE Cop	
76BF4000	00003000	sechost			Img	RW Cop	RWE Cop	
76BF7000	00002000	sechost			Img	R	RWE Cop	

- EXE and DLLs are identified
- Double-click any row to show a memory dump
- Right-click, View in Disassembler

Rebasing

- Rebasing occurs when a module is not loaded at its preferred *base address*
- PE files have a preferred base address
  - The *image base* in the PE header
  - Usually the file is loaded at that address
  - Most EXEs are designed to be loaded at 0x00400000
- EXEs that support Address Space Layout Randomization (ASLR) will often be relocated

DLL Rebasing

- DLLs are more commonly relocated
  - Because a single application may import many DLLs
  - Windows DLLs have different base addresses to avoid this
  - Third-party DLLs often have the same preferred base address

Absolute v. Relative Addresses

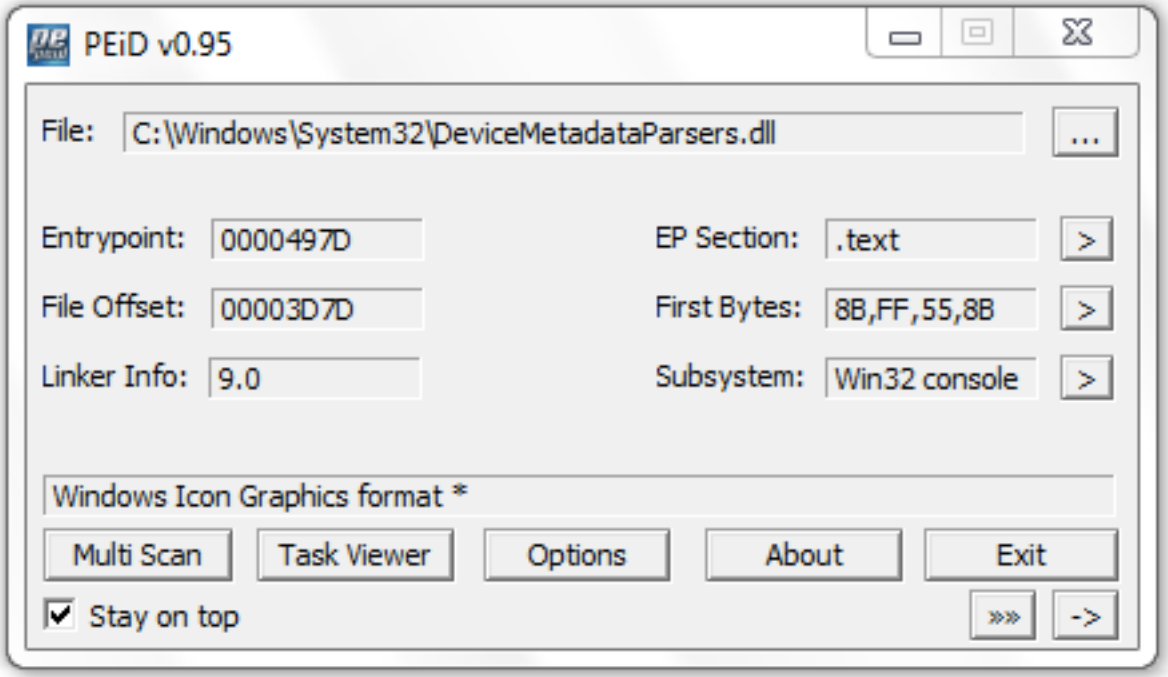
Example 10-1. Assembly code that requires relocation

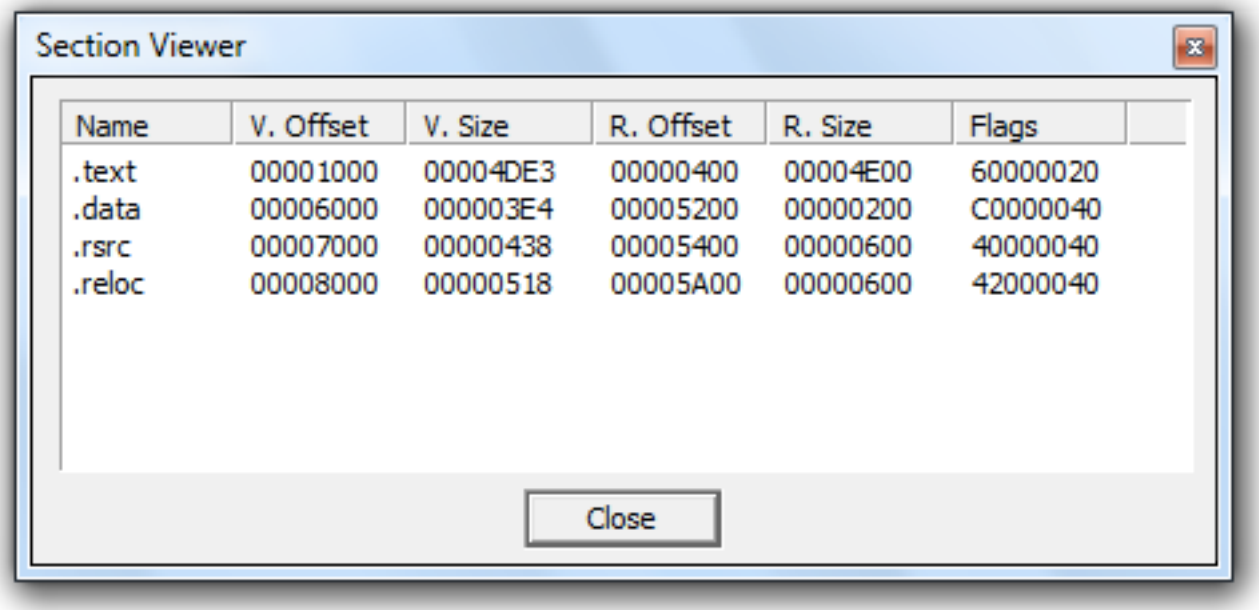
```
00401203        mov eax, [ebp+var_8]
00401206        cmp [ebp+var_4], 0
0040120a        jnz loc_0040120
0040120c       1mov eax, dword_40CF60
```

- The first 3 instructions will work fine if relocated because they use *relative addresses*
- The last one has an *absolute address* that will be wrong if the code is relocated

Fix-up Locations

- Most DLLS have a list of fix-up locations in the `.reloc` section of the PE header
  - These are instructions that must be changed when code is relocated
- DLLs are loaded after the EXE and in any order
- You cannot predict where DLLs will be located in memory if they are rebased
- Example .reloc section on next slide





Name	V. Offset	V. Size	R. Offset	R. Size	Flags
.text	00001000	00004DE3	00000400	00004E00	60000020
.data	00006000	000003E4	00005200	00000200	C0000040
.rsrc	00007000	00000438	00005400	00000600	40000040
.reloc	00008000	00000518	00005A00	00000600	42000040

DLL Rebasing

- DLLS can have their `.reloc` removed
  - Such a DLL cannot be relocated
  - Must load at its preferred base address
- Relocating DLLs is bad for performance
  - Adds to load time
  - So good programmers specify non-default base addresses when compiling DLLs

Example of DLL Rebasing Olly Memory Map

- DLL-A and DLL-B prefer location 0x100000000

00340000	00001000	DLL-B		PE header	Imag	R	RWE
00341000	00009000	DLL-B	.text	code	Imag	R	RWE
0034A000	00002000	DLL-B	.rdata	imports,exp	Imag	R	RWE
0034C000	00003000	DLL-B	.data	data	Imag	R	RWE
0034F000	00001000	DLL-B	.rsrc	resources	Imag	R	RWE
00350000	00001000	DLL-B	.reloc	relocations	Imag	R	RWE
00400000	00001000	EXE-1		PE header	Imag	R	RWE
00401000	00010000	EXE-1	.textbss	code	Imag	R	RWE
00411000	00004000	EXE-1	.text	SFX	Imag	R	RWE
00415000	00002000	EXE-1	.rdata		Imag	R	RWE
00417000	00001000	EXE-1	.data	data	Imag	R	RWE
00418000	00001000	EXE-1	.idata	imports	Imag	R	RWE
00419000	00001000	EXE-1	.rsrc	resources	Imag	R	RWE
10000000	00001000	DLL-A		PE header	Imag	R	RWE
10001000	00009000	DLL-A	.text	code	Imag	R	RWE
1000A000	00002000	DLL-A	.rdata	imports,exp	Imag	R	RWE
1000C000	00003000	DLL-A	.data	data	Imag	R	RWE
1000F000	00001000	DLL-A	.rsrc	resources	Imag	R	RWE
10010000	00001000	DLL-A	.reloc	relocations	Imag	R	RWE

Figure 10-5. DLL-B is relocated into a different memory address from its requested location

IDA Pro

- IDA Pro is not attached to a real running process
- It doesn't know about rebasing
- If you use OllyDbg and IDA Pro at the same time, you may get different results
  - To avoid this, use the "Manual Load" option in IDA Pro
  - Specify the virtual base address manually

Viewing Threads and Stacks

- View, Threads
- Right-click a thread to "Open in CPU", kill it, etc.

Threads

Ord	Ident	Window's title	Last error	Entry	TIB	Suspend	Priority	User time	System time
Main	00000F34	Cisco Packet Trac	ERROR_SUCCESS (00		7FFDF000	0.	Normal	1.1544 s	0.2964 s
2.	00000488		ERROR_SUCCESS (00	7029345E	7FFDE000	0.	Normal	0.0000 s	0.0000 s
3.	000007C4		ERROR_SUCCESS (00	76E6EB16	7FFDD000	0.	Normal	0.0000 s	0.0000 s
4.	00000414		ERROR_SUCCESS (00	76E6D34E	7FFDC000	0.	Normal	0.0000 s	0.0000 s
5.	00000A80		ERROR_SUCCESS (00	76E6D34E	7FFDB000	0.	Normal	0.0000 s	0.0000 s
6.	0000093C		ERROR_SUCCESS (00	768DC89D	7FFDA000	0.	Normal	0.0000 s	0.0000 s
7.	000008C8		ERROR_SUCCESS (00	749E6F14	7FFD9000	0.	High	0.0000 s	0.0000 s

Each Thread Has its Own Stack

- Visible in Memory Map

Memory map

Address	Size	Owner	Section	Contains	Type	Access	Initial
05050000	00800000				Priv	RW	RW
05850000	00A80000				Priv	RW	RW
06820000	003FC000				Map	R	R
06D1D000	00002000				Priv	RW Gua	RW Gua
06D1F000	00001000			Stack of thread 2. (00000488)	Priv	RW	RW
06E1D000	00002000				Priv	RW Gua	RW Gua
06E1F000	00001000			Stack of thread 3. (000007C4)	Priv	RW	RW
06F10000	00BBD000				Priv	RW	RW
07AD0000	006B5000				Priv	RW	RW
0828D000	00002000				Priv	RW Gua	RW Gua
0828F000	00001000			Stack of thread 4. (00000414)	Priv	RW	RW
0838D000	00002000				Priv	RW Gua	RW Gua
0838F000	00001000			Stack of thread 5. (00000A80)	Priv	RW	RW
0848C000	00002000				Priv	RW Gua	RW Gua
0848E000	00002000			Stack of thread 6. (0000093C)	Priv	RW	RW
0858D000	00002000				Priv	RW Gua	RW Gua
0858F000	00001000			Stack of thread 7. (000008C8)	Priv	RW	RW
08630000	00019000				Priv	RW	RW
08670000	0021F000				Map	RW	RW
088B0000	01C57000				Priv	RW	RW
0A510000	001E6000				Priv	RW	RW

ASLR is Fading

- Address Space Layout Randomization
  - *"ASLR is fundamentally flawed in sandboxed environments such as JavaScript and future defenses should not rely on randomized virtual addresses as a building block."*
- https://www.theregister.com/2021/02/26/chrome_aslr_bypass/

Executing Code

Table 10-1. OllyDbg Code-Execution Options

Function	Menu	Hotkey	Button
Run/Play	Debug ▸ Run	F9	
Pause	Debug ▸ Pause	F12	
Run to selection	Breakpoint ▸ Run to Selection	F4	
Run until return	Debug ▸ Execute till Return	CTRL-F9	
Run until user code	Debug ▸ Execute till User Code	ALT-F9	
Single-step/step-into	Debug ▸ Step Into	F7	
Step-over	Debug ▸ Step Over	F8	

Run and Pause

- You could Run a program and click Pause when it's where you want it to be
- But that's sloppy and might leave you somewhere uninteresting, such as inside library code
- Setting breakpoints is much better

Run and Run to Selection

- Run is useful to resume execution after hitting a breakpoint
- Run to Selection will execute until just before the selected instruction is executed
  - If the selection is never executed, it will run indefinitely

Execute till Return

- Pauses execution until just before the current function is set to return
- Can be useful if you want to finish the current function and stop
- But if the function never ends, the program will continue to run indefinitely

Execute till User Code

- Useful if you get lost in library code during debugging
- Program will continue to run until it hit compiled malware code
  - Typically the `.text` section

Stepping Through Code

- F7 -- Single-step (also called step-into)
- F8 -- Step-over
  - Stepping-over means all the code is executed, but you don't see it happen
- Some malware is designed to fool you, by calling routines and never returning, so stepping over will miss the most important part

Kahoot!

9a

Breakpoints

Types of Breakpoints

- Software breakpoints
- Hardware breakpoints
- Conditional breakpoints
- Breakpoints on memory

- F2 – Add or remove a breakpoint

Viewing Active Breakpoints

- View, Breakpoints, or click B icon on toolbar

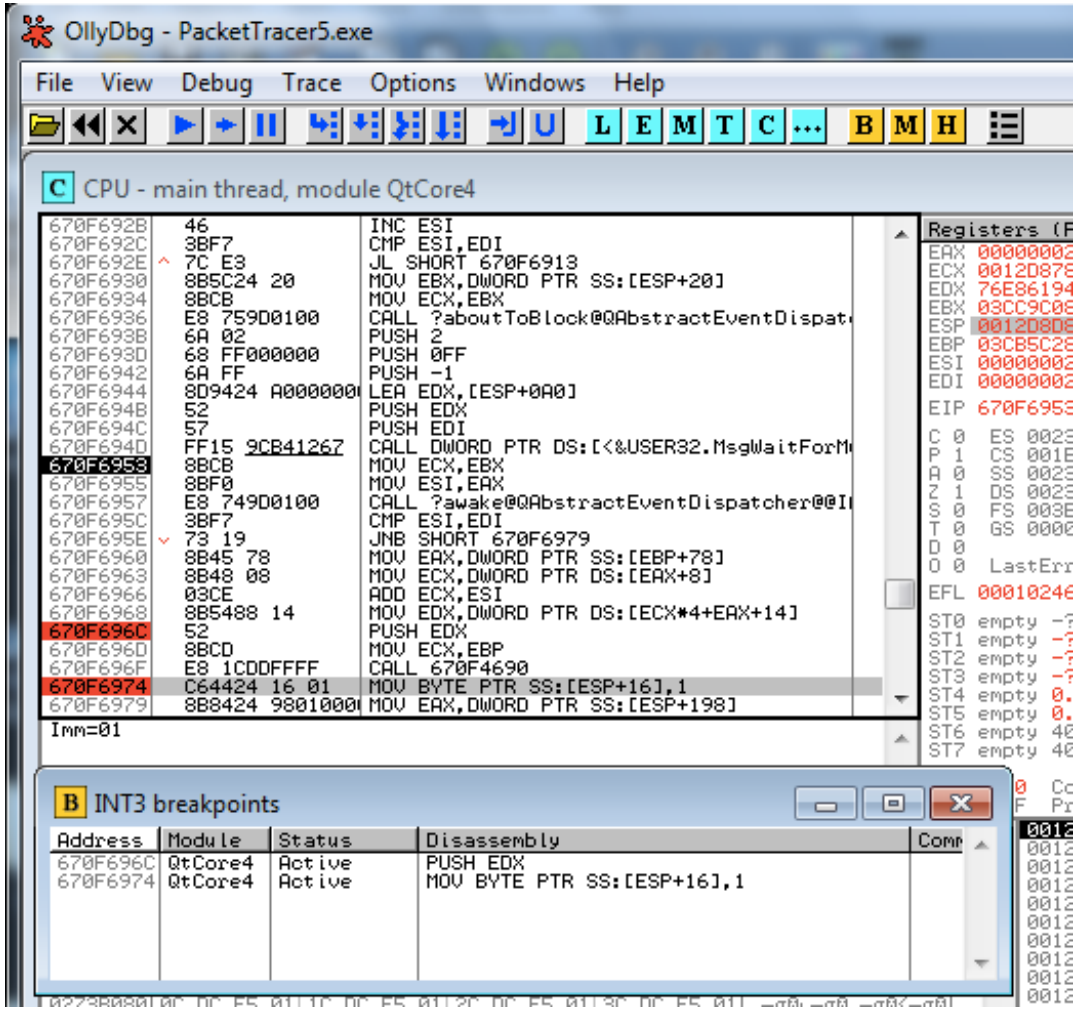


Table 10-2. OllyDbg Breakpoint Options

Function	Right-click menu selection	Hotkey
Software breakpoint	Breakpoint ▸ Toggle	F2
Conditional breakpoint	Breakpoint ▸ Conditional	SHIFT-F2
Hardware breakpoint	Breakpoint ▸ Hardware, on Execution	
Memory breakpoint on access (read, write, or execute)	Breakpoint ▸ Memory, on Access	F2 (select memory)
Memory breakpoint on write	Breakpoint ▸ Memory, on Write	

Saving Breakpoints

- When you close OllyDbg, it saves your breakpoints
- If you open the same file again, the breakpoints are still available

Software Breakpoints

- Useful for string decoders
- Malware authors often obfuscate strings
  - With a **string decoder** that is called before each string is used

Example 10-2. A string decoding breakpoint

```
push offset "4NNpTNHLKIXoPm7iBhUAjvRKNaUVBlr"
call String_Decoder
...
push offset "ugKLdNlLT6emldCeZi72mUjieuBqdfZ"
call String_Decoder
...
```

String Decoders

- Put a breakpoint at the end of the decoder routine
- The string becomes readable on the stack Each time you press Play in OllyDbg, the program will execute and will break when a string is decoded for use
- This method will only reveal strings as they are used

Conditional Breakpoints

- Breaks only when a condition is true
- Ex: Poison Ivy backdoor
  - Poison Ivy allocates memory to house the shellcode it receives from Command and Control (C&C) servers
  - Most memory allocations are for other purposes and uninteresting
  - Set a conditional breakpoint at the VirtualAlloc function in Kernel32.dll

Normal Breakpoint

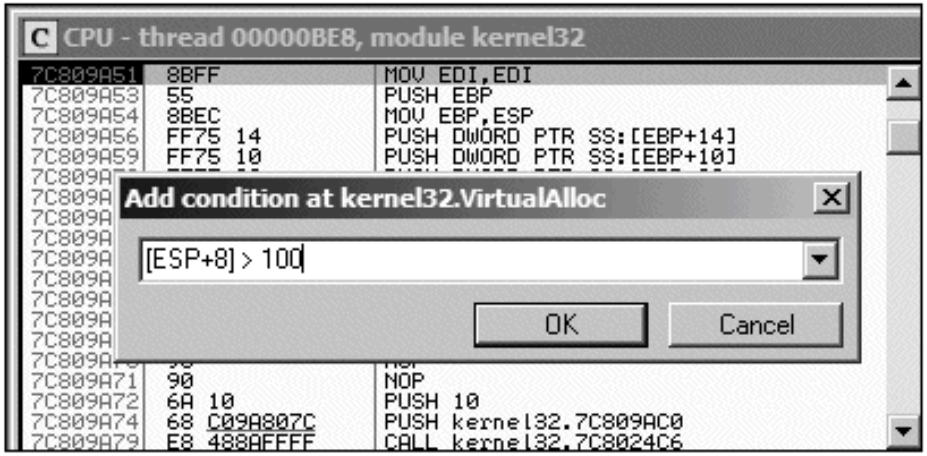
- Put a standard breakpoint at the start of the VirtualAlloc function
- Here's the stack when it hits, showing five items:
  - Return address
  - 4 parameters (Address, Size, AllocationType, Protect)

```
00C3FDB0  0095007C  CALL to VirtualAlloc from 00950079
00C3FDB4  00000000  Address = NULL
00C3FDB8  00000029  Size = 29 (41.)
00C3FDBC  00001000  AllocationType = MEM_COMMIT
00C3FDC0  00000040  Protect = PAGE_EXECUTE_READWRITE
```

Figure 10-7. Stack window at the start of `VirtualAlloc`

Conditional Breakpoint

1. Right-click in the disassembler window on the first instruction of the function, and select **Breakpoint ▶ Conditional**. This brings up a dialog asking for the conditional expression.
2. Set the expression and click **OK**. In this example, use **`[ESP+8]>100`**.
3. Click **Play** and wait for the code to break.



Hardware Breakpoints

- Don't alter code, stack, or any target resource
- Don't slow down execution
- But you can only set 4 at a time
- Click **Breakpoint**, "**Hardware**, **on Execution**"
- You can set OllyDbg to use hardware breakpoints by default in Debugging Options
  - Useful if malware uses anti-debugging techniques

Memory Breakpoints

- Code breaks on access to specified memory location
- OllyDbg supports software and hardware memory breakpoints
- Can break on read, write, execute, or any access
- Right-click memory location, click **Breakpoint**, "**Memory, on Access**"

Memory Breakpoints

- You can only set one memory breakpoint at a time
- OllyDbg implements memory breakpoints by changing the attributes of memory blocks
- This technique is not reliable and has considerable overhead
- Use memory breakpoints sparingly

When is a DLL Used?

1. Bring up the Memory Map window and right-click the DLL's `.text` section (the section that contains the program's executable code).
2. Select **Set Memory Breakpoint on Access**.
3. Press F9 or click the play button to resume execution.

The program should break when execution ends up in the DLL's `.text` section.

Kahoot!

9b

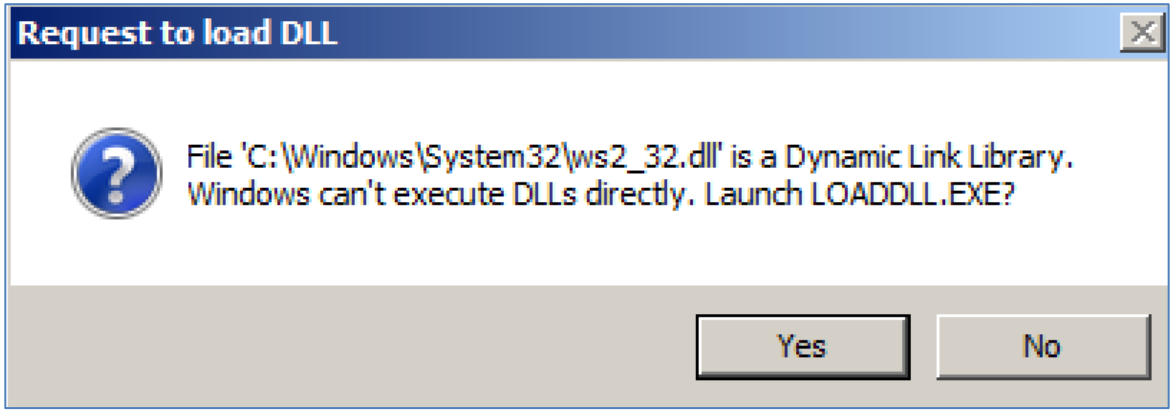
Loading DLLs

loaddll.exe

- DLLs cannot be executed directly
- OllyDbg uses a dummy loaddll.exe program to load them
- Breaks at the DLL entry point DLLMain once the DLL is loaded
- Press Play to run DLLMain and initialize the DLL for use

Demo

- Get OllyDbg 1.10, NOT 2.00 or 2.01
  - Link Ch 9a
- Use Win 2016 Server, 64 bit
- In OllyDbg, open
  - C:\Windows\SysWOW64\ws2_32.dll
- Click **Yes** at this box



Demo: Calling DLL Exports

- Click **Debug**, **Call DLL Export** – it fails because DLLMain has not yet been run
- Reload the DLL (**Ctrl+F2**), click **Run** button once
- Click **Debug**, **Call DLL Export** – now it works
  - Image on next slide

OllyDbg - ws2_32.dll - [CPU - main thread, module ws2_32]

File View Debug Plugins Options Window Help

```
75BEC750 $ 8BFF          MOV EDI,EDI
75BEC752 . 55            PUSH EBP
75BEC753 . 8BEC          MOV EBP,ESP
75BEC755 . 8B55 08       MOV EDX,DWORD PT
75BEC758 . 8BC2          MOV EAX,EDX
75BEC75A . 8BCA          MOV ECX,EDX
75BEC75C . 25 00FF0000   AND EAX,0FF00
75BEC761 . C1E1 10       SHL ECX,10
75BEC764 . 0BC1          OR EAX,ECX
75BEC766 . 8BCA          MOV ECX,EDX
75BEC768 . 81E1 0000FF00 AND ECX,0FF0000
75BEC76E . C1EA 10       SHR EDX,10
75BEC771 . 0BCA          OR ECX,EDX
75BEC773 . C1E0 08       SHL EAX,8
75BEC776 . C1E9 08       SHR ECX,8
75BEC779 . 0BC1          OR EAX,ECX
75BEC77B . 5D            POP EBP
75BEC77C . C2 0400       RETN 4
75BEC77F   CC            INT3
75BEC780   CC            INT3
75BEC781   CC            INT3
75BEC782   CC            INT3
75BEC783   CC            INT3
75BEC784   CC            INT3
75BEC785   CC            INT3
75BEC786   CC            INT3
75BEC787   CC            INT3
Local calls from 75BDB5C8, 75BE7FEB
```

Address	Hex dump	ASCII
75C19000	F0 49 BE 75 20 95 C1 75	ðI¾u •Áu

Call export in ws2_32.dll

Export: 75BEC750 ntohl

Pure function (no side effects)
Number of arguments: 1.
Valid stack frame

☑ 0 (no arguments)

Follow in Disassembler

☑ 1 7f0001 (Pushed last)

```
007F0001 00 00 00 00 00 00 00 00 ........
007F0009 00 00 00 00 00 00 00 00 ........
007F0011 00 00 00 00 00 00 00 00 ........
```

☐ 2 0
☐ 3 0
☐ 4 0
☐ 5 0
☐ 6 0
☐ 7 0
☐ 8 0
☐ 9 0
☐ 10 0 (Pushed first)

Value of registers:

	Before call	After call
EAX	0	01007F00
ECX	0	00007F00
EDX	0	0000007F
EBX	0	00000000
ESI	0	00000000
EDI	0	00000000

Done

☐ Hide on call

Call

☐ Pause after call

Close

Demo: Running ntohl

- Converts a 32-bit number from network to host byte order
- Click argument 1, type in 7f000001
  - 127.0.0.1 in "network" byte order
- Click "Follow in Disassembler" to see the code
- Click "Call" to run the function
- Answer in EAX

Tracing

Tracing

- Powerful debugging technique
- Records detailed execution information
- Types of Tracing
  - Standard Back Trace
  - Call Stack Trace
  - Run Trace

Standard Back Trace

- You move through the disassembler with the Step Into and Step Over buttons
- OllyDbg is recording your movement
- Use minus key on keyboard to see previous instructions
  - But you won't see previous register values
- Plus key takes you forward
  - If you used Step Over, you cannot go back and decide to step into

Call Stack Trace

- Views the execution path to a given function
- Click **View**, **Call Stack**
- Displays the sequence of calls to reach your current location

Demo from PMA 401

- Simple guessing game
- Wrong answer produces an insult

```
C:\Users\Administrator\Documents\easy\new>00000.exe
Launch codes?
1
I think my dog figured this out before you.

C:\Users\Administrator\Documents\easy\new>
```

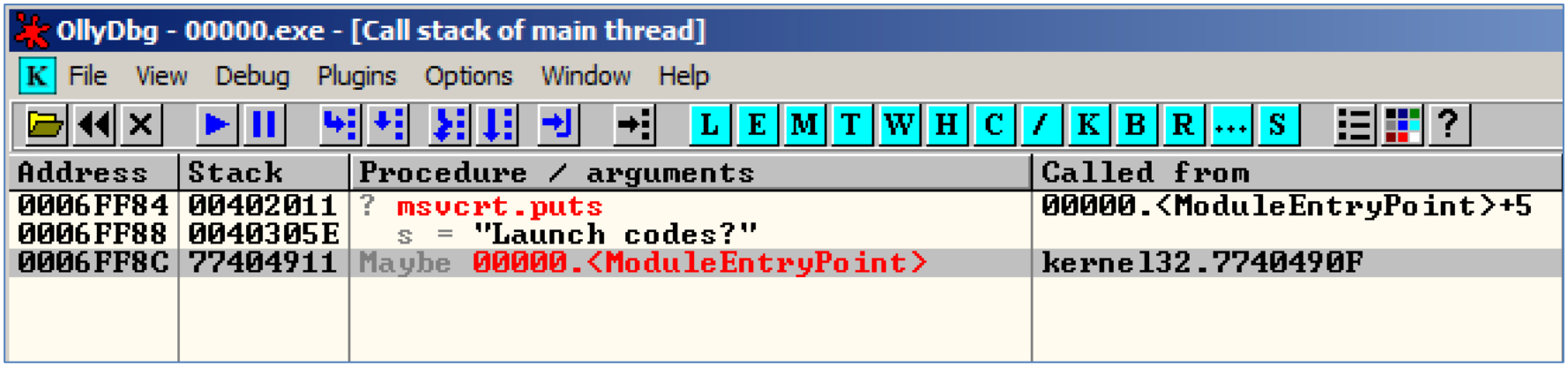
Entire main() in OllyDbg

```
OllyDbg - 00000.exe - [CPU - main thread, module 00000]
C File View Debug Plugins Options Window Help

00402006  $ 68 5E304000   PUSH 00000.0040305E          s = "Launch codes?"
0040200B  . FF15 44104000 CALL DWORD PTR DS:[<&msvcrt.puts>]   puts
00402011  . 58            POP EAX
00402012  . 68 6C304000   PUSH 00000.0040306C
00402017  . 68 04304000   PUSH 00000.00403004          format = "%d"
0040201C    FF15 48104000 CALL DWORD PTR DS:[<&msvcrt.scanf>]   msvcrt.scanf
00402022  . 83C4 08       ADD ESP,8
00402025  . A1 00304000   MOV EAX,DWORD PTR DS:[403000]
0040202A  . B9 EDA7A8A1   MOV ECX,A1A8A7ED
0040202F  . E8 CFFFFFFF   CALL 00000.00402003
00402034  . 3B05 6C304000 CMP EAX,DWORD PTR DS:[40306C]
0040203A    75 1E         JNZ SHORT 00000.0040205A
0040203C  . 8A0D 07304000 MOV CL,BYTE PTR DS:[403007]
00402042  . D3F8          SAR EAX,CL
00402044  . 25 FF000000   AND EAX,0FF
00402049  . 50            PUSH EAX                    <%c>
0040204A  . 68 34304000   PUSH 00000.00403034         format = "Wow you got it. Here is the result: (%c)"
0040204F  . FF15 4C104000 CALL DWORD PTR DS:[<&msvcrt.printf>]   printf
00402055  . 83C4 08       ADD ESP,8
00402058  . EB 0C         JMP SHORT 00000.00402066
0040205A  > 68 08304000   PUSH 00000.00403008         s = "I think my dog figured this out before you."
0040205F  . FF15 44104000 CALL DWORD PTR DS:[<&msvcrt.puts>]   puts
00402065  . 58            POP EAX
00402066  > C3            RETN
00402067  . 01C8          ADD EAX,ECX
00402069  . C3            RETN
```

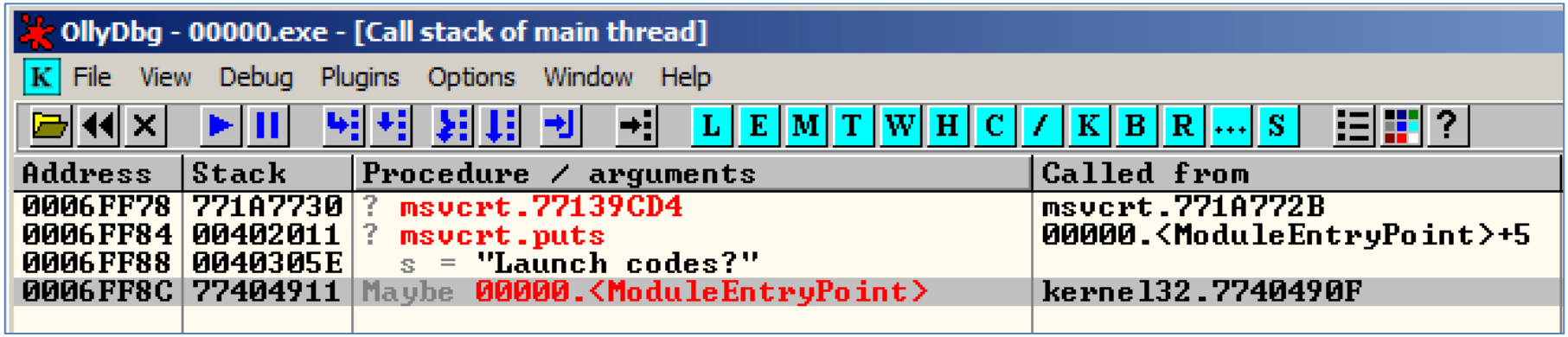
Step into puts

- Press **F7** twice
- Click **View**, **Call Stack**



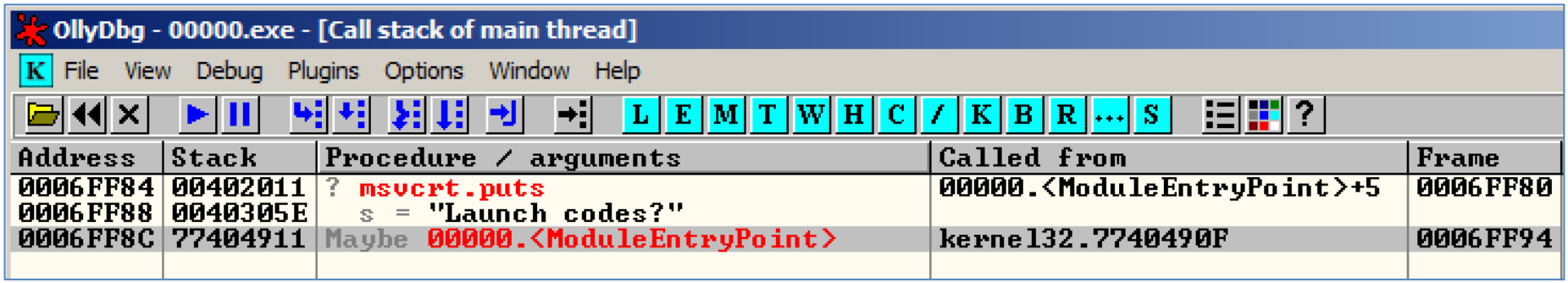
Step into again

- Click **View**, **CPU**
- Press **F7** three times
- Click **View**, **Call Stack**
- New function appears at top



Return

- Click **View**, **CPU**
- Press **F7** 22 times, until the RETN and execute it
- Click **View**, **Call Stack**



A Deeper Call Stack

OllyDbg - Lab09-01.exe

File View Debug Plugins Options Window Help

CPU - thread 00000F20, module CFGMGR32

75C95FF7	6A 18	PUSH 18
75C95FF9	68 7060C975	PUSH CFGMGR32.75C96070
75C95FFE	E8 29B2FFFF	CALL CFGMGR32.75C9122C
75C96003	33FF	XOR EDI,EDI
75C96005	897D E4	MOV DWORD PTR SS:[EBP-1C],EDI
75C96008	897D E0	MOV DWORD PTR SS:[EBP-20],EDI

Registers (FPU)

EAX 000000C0
ECX 001B1170
EDX 2E3FA1D5
EBX 001B49E8
ESP 0177F704

Call stack of thread 00000F20

Address	Stack	Procedure / arguments	Called from	Frame
0177F704	77AC43FC	Includes ntdll.KiFastSystemCallRet	ntdll.77AC43FA	0177
0177F708	75F50346	ntdll.ZwAlpcConnectPort	RPCRT4.75F50340	0177
0177F7D4	75F4F51E	RPCRT4.75F501D0	RPCRT4.75F4F519	0177
0177F804	75F4F3FE	RPCRT4.75F4F418	RPCRT4.75F4F3F9	0177
0177F838	75F3846D	RPCRT4.75F4F266	RPCRT4.75F38468	0177
0177F888	75F4BC1B	Includes RPCRT4.75F3846D	RPCRT4.75F4BC18	0177
0177F8AC	75F49D6D	RPCRT4.I_RpcGetBufferWithObject	RPCRT4.75F49D68	0177
0177F8BC	75F4A041	RPCRT4.I_RpcGetBuffer	RPCRT4.75F4A03C	0177
0177F8CC	75FA5718	Includes RPCRT4.75F4A041	RPCRT4.75FA5712	0177
0177FCF0	75C960B6	? <JMP.&RPCRT4.NdrClientCall2>	CFGMGR32.75C960B1	0177
0177FD08	75C96055	CFGMGR32.75C9609D	CFGMGR32.75C96050	0177
0177FD5C	762E0356	? CFGMGR32.CM_Get_Device_Interface_	SHELL32.762E0350	0177
0177FD98	762E02ED	SHELL32.762E0326	SHELL32.762E02E8	0177
0177FDA8	7709B6CF	Includes SHELL32.762E02ED	SHLWAPI.7709B6CD	0177
0177FDB8	77AAB338	Includes SHLWAPI.7709B6CF	ntdll.77AAB335	0177

Process terminated, exit code 0

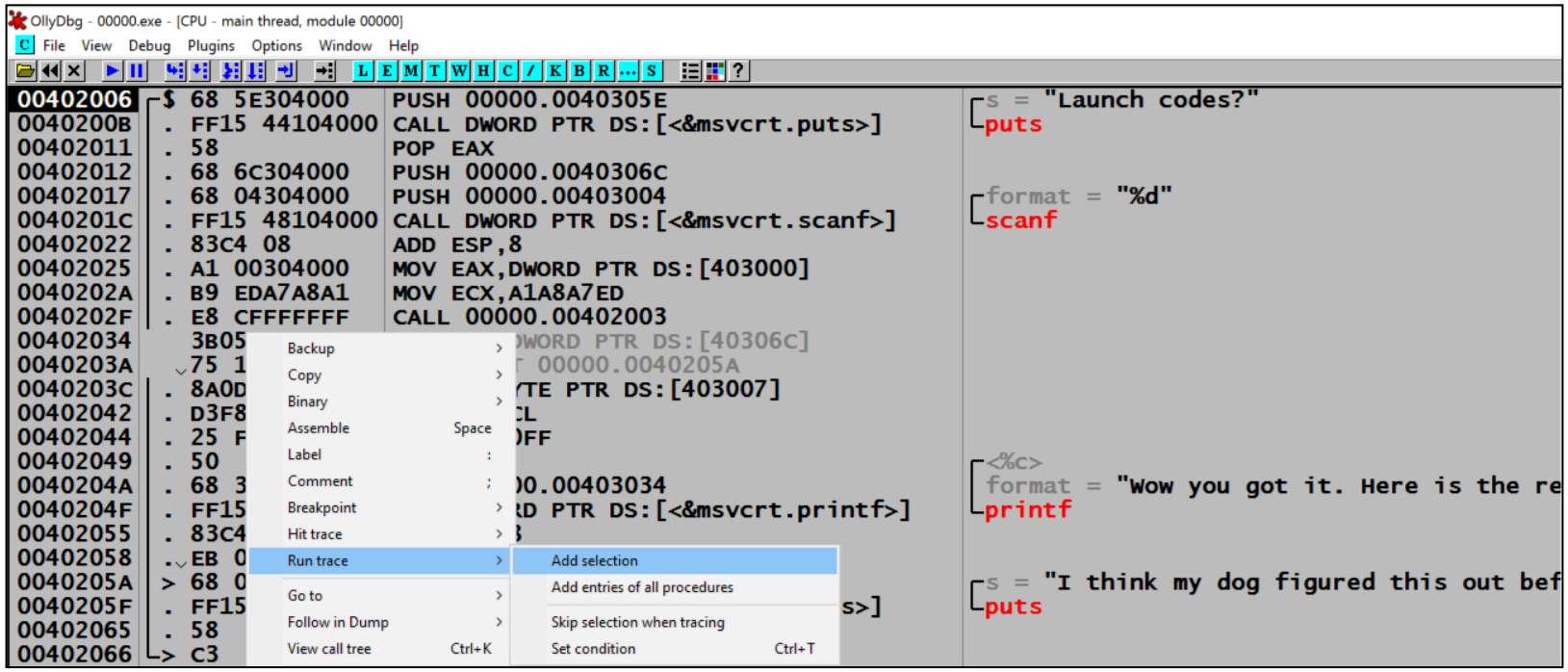
Terminated

Run Trace

- Code runs, and OllyDbg saves every executed instruction and all changes to registers and flags
- Highlight code, right-click, **Run Trace**, **Add Selection**
- After code executes, **View**, **Run Trace**
  - To see instructions that were executed
  - + and - keys to step forward and backwards

Demo: Run Trace of 00000.exe

- Highlight code, right-click, **Run Trace**, **Add Selection**



Demo: Run Trace of 00000.exe

- Run code
- Step back with - and forward with +

```
OllyDbg - 00000.exe - [CPU - Run trace 4. steps back, module 00000]
C File View Debug Plugins Options Window Help

00402006  $ 68 5E304000   PUSH 00000.0040305E                 s = "Launch codes?"
0040200B  . FF15 44104000 CALL DWORD PTR DS:[<&msvcrt.puts>]   puts
00402011  . 58            POP EAX
00402012  . 68 6C304000   PUSH 00000.0040306C
00402017  . 68 04304000   PUSH 00000.00403004                 format = "%d"
0040201C  . FF15 48104000 CALL DWORD PTR DS:[<&msvcrt.scanf>]  scanf
00402022  . 83C4 08       ADD ESP,8
00402025  . A1 00304000   MOV EAX,DWORD PTR DS:[403000]
0040202A  . B9 EDA7A8A1   MOV ECX,A1A8A7ED
0040202F  . E8 CFFFFFFF   CALL 00000.00402003
00402034    3B05 6C304000 CMP EAX,DWORD PTR DS:[40306C]
0040203A    75 1E         JNZ SHORT 00000.0040205A
0040203C  . 8A0D 07304000 MOV CL,BYTE PTR DS:[403007]
00402042  . D3F8          SAR EAX,CL
00402044  . 25 FF000000   AND EAX,0FF
00402049  . 50            PUSH EAX                           <%c>
0040204A  . 68 34304000   PUSH 00000.00403034                format = "Wow you got it. Here is the re
0040204F  . FF15 4C104000 CALL DWORD PTR DS:[<&msvcrt.printf>]  printf
00402055  . 83C4 08       ADD ESP,8
00402058  . EB 0C         JMP SHORT 00000.00402066
0040205A  > 68 08304000   PUSH 00000.00403008                 s = "I think my dog figured this out bef
0040205F  . FF15 44104000 CALL DWORD PTR DS:[<&msvcrt.puts>]   puts
00402065  . 58            POP EAX
00402066  > C3            RETN
00402067  . 01C8          ADD EAX,ECX
00402069  . C3            RETN
```

Trace Into and Trace Over

- Buttons below "Options"
- Easier to use than Add Selection
- If you don't set breakpoints, OllyDbg will attempt to trace the entire program, which could take a long time and a lot of memory

Debug, Set Condition

- Traces until a condition hits
- This condition catches Poison Ivy shellcode, which places code in dynamically allocated memory below 0x400000

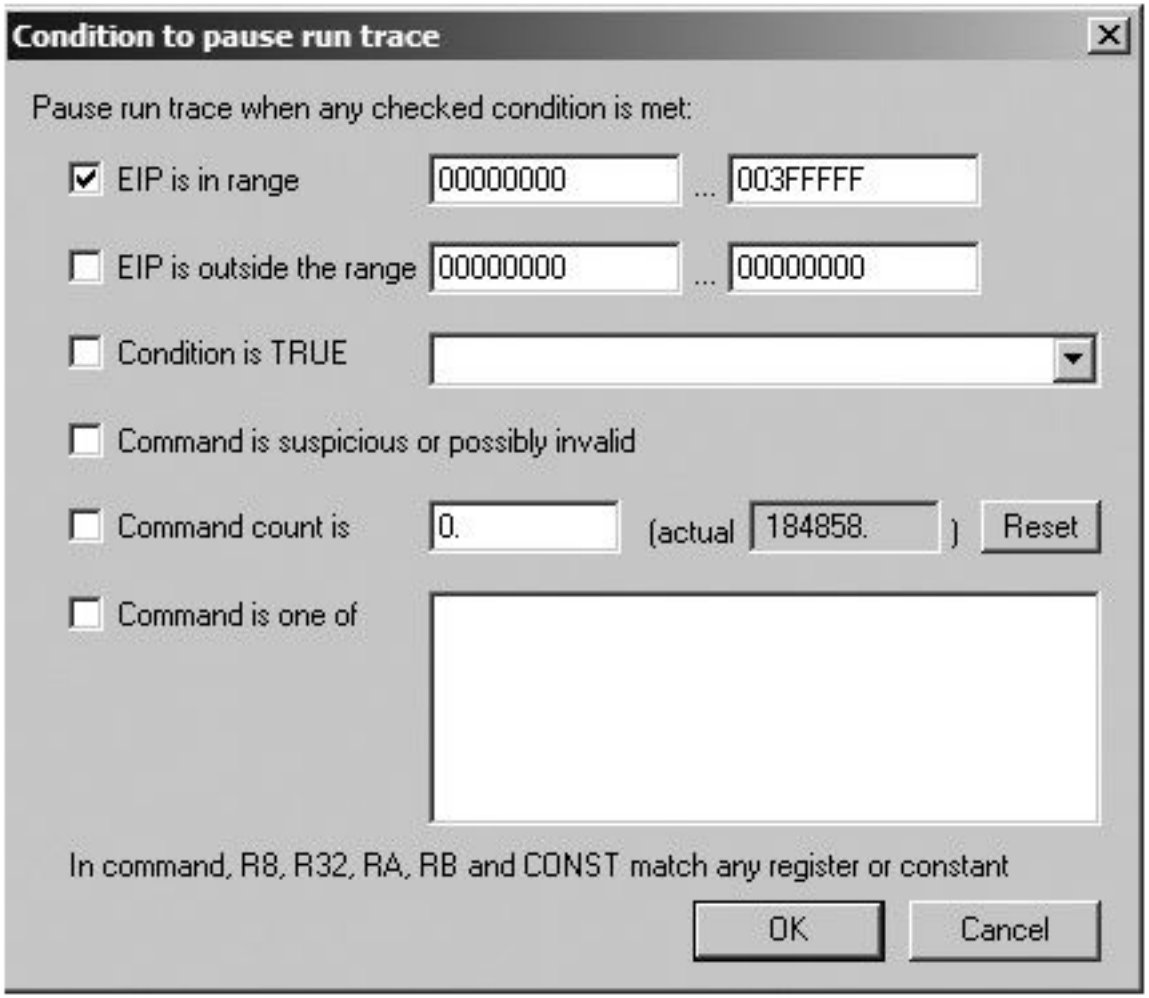


Figure 10-11. Conditional tracing

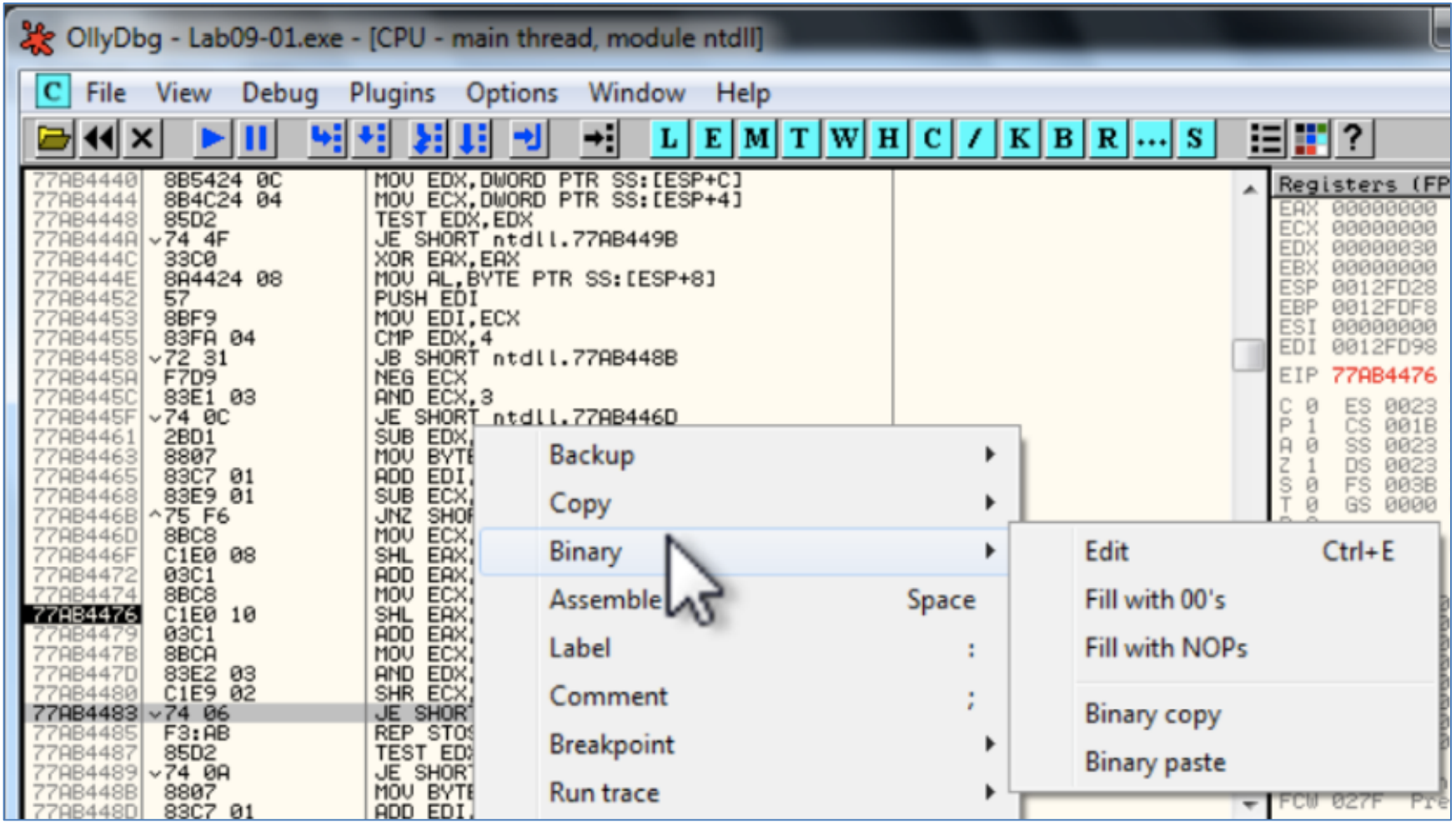
Exception Handling

When an Exception Occurs

- OllyDbg will stop the program
- You have these options to pass the exception into the program:
  - Shift+F7 Step into exception
  - Shift+F8: Step over exception
  - Shift+F9: Run exception handler
- Often you just ignore all exceptions in malware analysis
  - We aren't trying to fix problems in code

Patching

Binary Edit


OllyDbg - Lab09-01.exe - [CPU - main thread, module ntdll]
File View Debug Plugins Options Window Help
Backup
Copy
Binary
Assemble Space
Label :
Comment ;
Breakpoint
Run trace
Edit Ctrl+E
Fill with 00's
Fill with NOPs
Binary copy
Binary paste
Registers (FP
EAX 00000000
ECX 00000000
EDX 00000030
EBX 00000000
ESP 0012FD28
EBP 0012FDF8
ESI 00000000
EDI 0012FD98
EIP 77AB4476

Fill

- Fill with 00
- Fill with NOP (0x90)
  - Used to skip instructions
  - e.g. to force a branch

Saving Patched Code

- Right-click disassembler window after patching
  - Copy To Executable, All Modifications, Save File
  - Copy All
- Right-click in new window
  - Save File

Analyzing Shellcode

Undocumented technique

Easy Way to Analyze Shellcode

- Copy shellcode from a hex editor to clipboard
- Within memory map, select a region of type "Priv" (Private memory)
- Double-click rows in memory map to show a hex dump
  - Find a region of hundreds of consecutive zeroes
- Right-click chosen region in Memory Map, Set Access, Full Access (to clear NX bit)

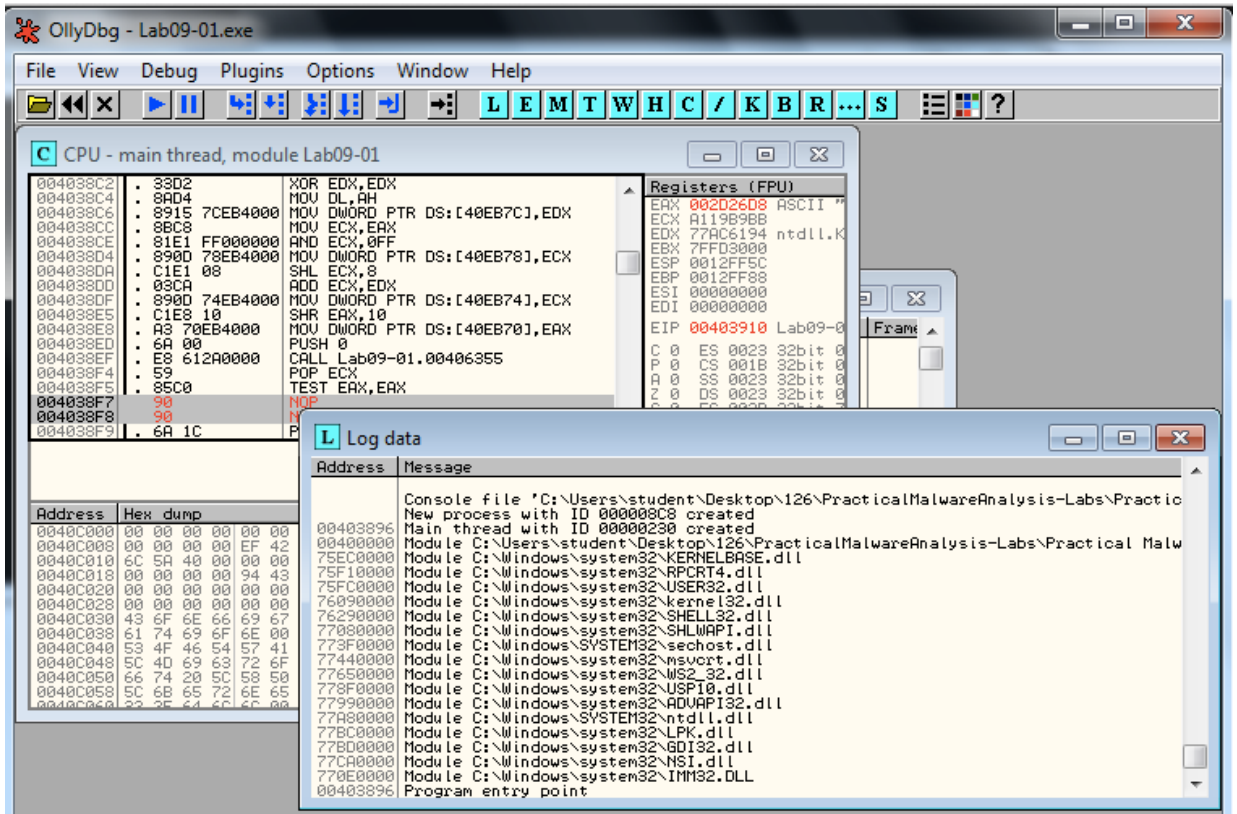
Analyzing Shellcode

- Highlight a region of zeroes, Binary, Binary Paste
- Set EIP to location of shellcode
  - Right-click first instruction, New Origin Here

Assistance Features

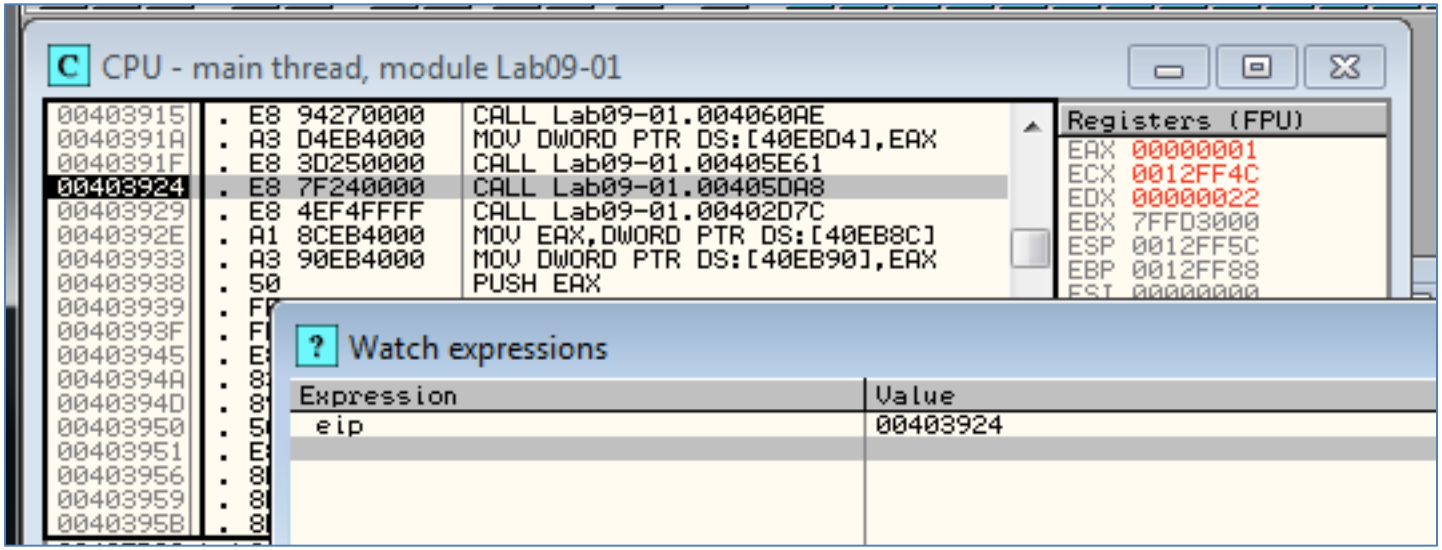
Log

- View, Log
  - Shows steps to reach here



Watches Window

- View, Watches
  - Watch the value of an expression
  - Press SPACEBAR to set expression
  - OllyDbg Help, Contents
    - Instructions for Evaluation of Expressions



Labeling

- Label subroutines and loops
  - Right-click an address, Label

Plug-ins

Recommended Plugins

- OllyDump
  - Dumps debugged process to a PE file
  - Used for unpacking
- Hide Debugger
  - Hides OllyDbg from debugger detection
- Command Line
  - Control OllyDbg from the command line
  - Simpler to just use WinDbg
- Bookmarks
  - Included by default in OllyDbg
  - Bookmarks memory locations

Scriptable Debugging

Immunity Debugger (ImmDbg)

- Unlike OllyDbg, ImmDbg employs Python scripts and has an easy-to-use API
- Scripts are located in the PyCommands subdirectory under the install directory of ImmDbg
- Easy to create custom scripts for ImmDbg

Kahoot!

9c