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
Don't Use OllyDbg 2!
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
Disassembler
Highlight: next instruction to be executed

Registers
Memory dump
Stack
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
- 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    mov 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
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

<table>
<thead>
<tr>
<th>Address</th>
<th>Section</th>
<th>DLL-B</th>
<th>PE header</th>
<th>Imag</th>
<th>R</th>
<th>RWE</th>
</tr>
</thead>
<tbody>
<tr>
<td>00340000</td>
<td>.text</td>
<td>DLL-B</td>
<td>code</td>
<td>Imag</td>
<td>R</td>
<td>RWE</td>
</tr>
<tr>
<td>00341000</td>
<td>.rdata</td>
<td>DLL-B</td>
<td>imports,exp</td>
<td>Imag</td>
<td>R</td>
<td>RWE</td>
</tr>
<tr>
<td>00340000</td>
<td>.data</td>
<td>DLL-B</td>
<td>data</td>
<td>Imag</td>
<td>R</td>
<td>RWE</td>
</tr>
<tr>
<td>0034C000</td>
<td>.rsrc</td>
<td>DLL-B</td>
<td>resources</td>
<td>Imag</td>
<td>R</td>
<td>RWE</td>
</tr>
<tr>
<td>00350000</td>
<td>.reloc</td>
<td>DLL-B</td>
<td>relocations</td>
<td>Imag</td>
<td>R</td>
<td>RWE</td>
</tr>
</tbody>
</table>

**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.
Each Thread Has its Own Stack

- Visible in Memory Map
Executing Code
<table>
<thead>
<tr>
<th>Function</th>
<th>Menu</th>
<th>Hotkey</th>
<th>Button</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run/Play</td>
<td>Debug ▶ Run</td>
<td>F9</td>
<td></td>
</tr>
<tr>
<td>Pause</td>
<td>Debug ▶ Pause</td>
<td>F12</td>
<td></td>
</tr>
<tr>
<td>Run to selection</td>
<td>Breakpoint ▶ Run to Selection</td>
<td>F4</td>
<td></td>
</tr>
<tr>
<td>Run until return</td>
<td>Debug ▶ Execute till Return</td>
<td>CTRL-F9</td>
<td></td>
</tr>
<tr>
<td>Run until user code</td>
<td>Debug ▶ Execute till User Code</td>
<td>ALT-F9</td>
<td></td>
</tr>
<tr>
<td>Single-step/step-into</td>
<td>Debug ▶ Step Into</td>
<td>F7</td>
<td></td>
</tr>
<tr>
<td>Step-over</td>
<td>Debug ▶ Step Over</td>
<td>F8</td>
<td></td>
</tr>
</tbody>
</table>
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
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>
<thead>
<tr>
<th>Function</th>
<th>Right-click menu selection</th>
<th>Hotkey</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software breakpoint</td>
<td>Breakpoint ▶ Toggle</td>
<td>F2</td>
</tr>
<tr>
<td>Conditional breakpoint</td>
<td>Breakpoint ▶ Conditional</td>
<td>SHIFT-F2</td>
</tr>
<tr>
<td>Hardware breakpoint</td>
<td>Breakpoint ▶ Hardware, on Execution</td>
<td></td>
</tr>
<tr>
<td>Memory breakpoint on access</td>
<td>Breakpoint ▶ Memory, on Access</td>
<td>F2 (select memory)</td>
</tr>
<tr>
<td>(read, write, or execute)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory breakpoint on write</td>
<td>Breakpoint ▶ Memory, on Write</td>
<td></td>
</tr>
</tbody>
</table>
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 \textbf{string decoder} that is called before each string is used

\begin{quote}
\textit{Example 10-2. A string decoding breakpoint}

\begin{verbatim}
push offset "4NNpTNHLKIXoPm7iBhUAjvRKNaUVBlr"
call String_Decoder
...
push offset "ugKLDnLlLT6emldCeZi72mUjieuBqdfZ"
call String_Decoder
...
\end{verbatim}
\end{quote}
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)

![Stack window at the start of VirtualAlloc](image.png)

*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.
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 2008 Server
• In OllyDbg, open c:\windows\system32\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
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
Don't Use OllyDbg 2!
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 EasyCTF 2017

• Simple guessing game
• Wrong answer produces an insult
Entire main() in OllyDbg
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 until the RETN and execute it
- Click View, Call Stack
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 ntohl

- Click Debug, Call DLL Export
- Click argument 1, type in 7f000001 – 127.0.0.1 in "network" byte order
- Click "Follow in Disassembler" to see the code
- Highlight code, right-click, Run Trace, Add Selection
Demo: Run Trace of ntohl

• Click Debug, Call DLL Export
• Click Call
• Code is now marked with a red bar
• Indicating that it can be played back
• Step back with - and forward with +
Demo: Run Trace of ntohs

- Click Debug, Call DLL Export
- 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
<table>
<thead>
<tr>
<th>Back</th>
<th>Thread</th>
<th>Module</th>
<th>Address</th>
<th>Command</th>
<th>Modified registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>Main</td>
<td>Lab09-01</td>
<td>004038C4</td>
<td>MOV DL, AH</td>
<td>EDX=00000002</td>
</tr>
<tr>
<td>8</td>
<td>Main</td>
<td>Lab09-01</td>
<td>004038C6</td>
<td>MOU DWORD PTR DS:[40EB7C], EDX</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Main</td>
<td>Lab09-01</td>
<td>004038CC</td>
<td>MOU ECX, EAX</td>
<td>ECX=23F00206</td>
</tr>
<tr>
<td>6</td>
<td>Main</td>
<td>Lab09-01</td>
<td>004038CE</td>
<td>AND ECX, 0FF</td>
<td>ECX=00000006</td>
</tr>
<tr>
<td>5</td>
<td>Main</td>
<td>Lab09-01</td>
<td>004038D4</td>
<td>MOU DWORD PTR DS:[40EB78], ECX</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Main</td>
<td>Lab09-01</td>
<td>004038DA</td>
<td>SHL ECX, 8</td>
<td>ECX=00006000</td>
</tr>
<tr>
<td>3</td>
<td>Main</td>
<td>Lab09-01</td>
<td>004038DD</td>
<td>ADD ECX, EDX</td>
<td>ECX=00006002</td>
</tr>
<tr>
<td>2</td>
<td>Main</td>
<td>Lab09-01</td>
<td>004038DF</td>
<td>MOU DWORD PTR DS:[40EB74], ECX</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Main</td>
<td>Lab09-01</td>
<td>004038E5</td>
<td>SHR EAX, 10</td>
<td>EAX=000023F0</td>
</tr>
<tr>
<td>0</td>
<td>Main</td>
<td>Lab09-01</td>
<td>004038E8</td>
<td>MOU DWORD PTR DS:[40EB70], EAX</td>
<td></td>
</tr>
</tbody>
</table>
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
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 pas 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