Practical Malware Analysis

CH 5: Advanced Dynamic Analysis

HTU

CYBER SECURITY CLUB

Previous session wrap-up

Dynamic analysis: Running malware in safe environment to study the behavior of the malware and how it works.

Real machines:
- No internet connection, some parts of the malware won’t work.
- Difficult to remove malware → resetting machine is required.
+ Some malware detects VMs, so it’ll run its native behaviour.

VMs:
- Sometimes it’s detected by malware, thus it won’t 100% work.
+ Most common method, easy to setup and reset.

Previous session wrap-up

Why dynamic analysis?

You don’t have to deal with:

- obfuscation
- packing
- advanced techniques.

Important tools:

- ProcMon
- Process explorer
- Process hacker
- RegShot
- INetSim
- WireShark

Dynamic analysis activities:

- Monitoring processes
- Tracking DLLs
- Detecting malicious documents.
- Registry before & after.
- Simulating a network and inspecting traffic

Debugging

Debuggers vs Disassemblers

- A disassembler like IDA Pro shows the state of the program just before execution begins

- Debuggers show
  - Every memory location
  - Registers
  - Argument to every function
  - At any point during processing

And it lets you change those values in real time

Two debuggers

ollydbg:

- Most popular
- user-mode debugging only.
- IDA Pro has a built-in debugger, but harder than OllyDBG

Windbg:

- Supports kernel-mode debugging.

Two debuggers

ollydbg:

- Most popular
- user-mode debugging only.
- IDA Pro has a built-in debugger, but harder than OllyDBG

Windbg:

- Supports kernel-mode debugging.

Source-level VS Assembly-level debuggers

Source-level:
- Built into development platform.
- Breakpoints are based on line of code.
- Can step through the program one line at a time.

Assembly-level:
- Operates on assembly code (instructions)
- Breakpoints are based on instructions instead of lines of code.
- Mostly used by malware anslysts

User-mode vs Kernel-mode

User-mode

Debugger always runs on the same system as the code is analyzed on.

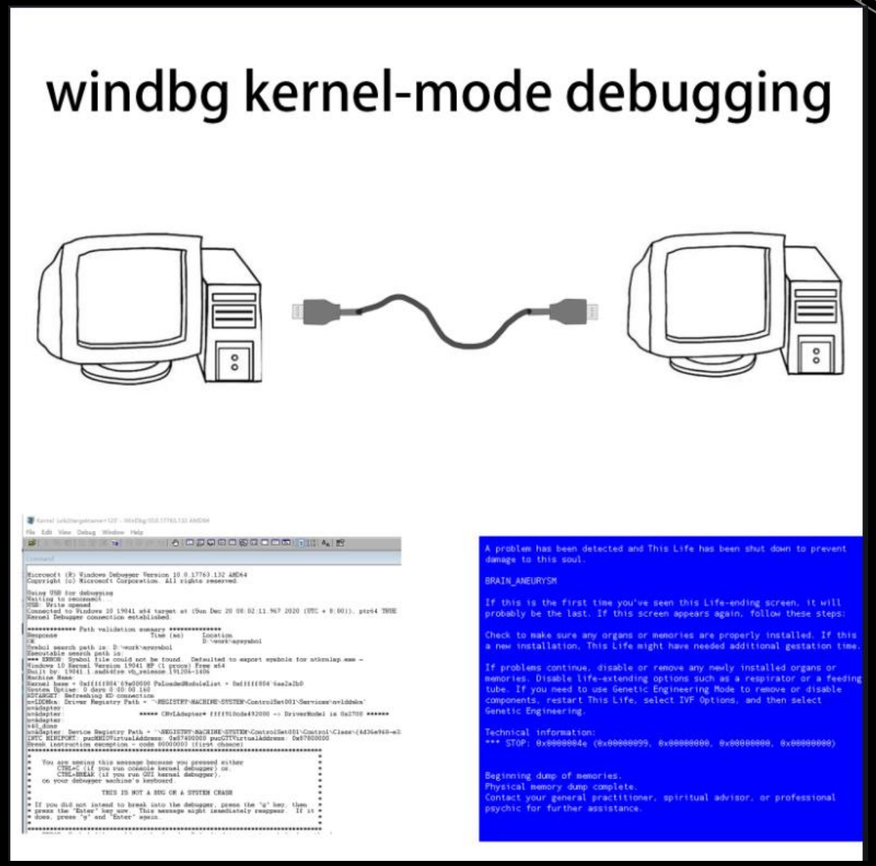
Separated from other executables by the OS

Debugs a single executable.

Kernel mode debugging (old way)

The two computers are connected, one runs debugger, the other runs the malware.

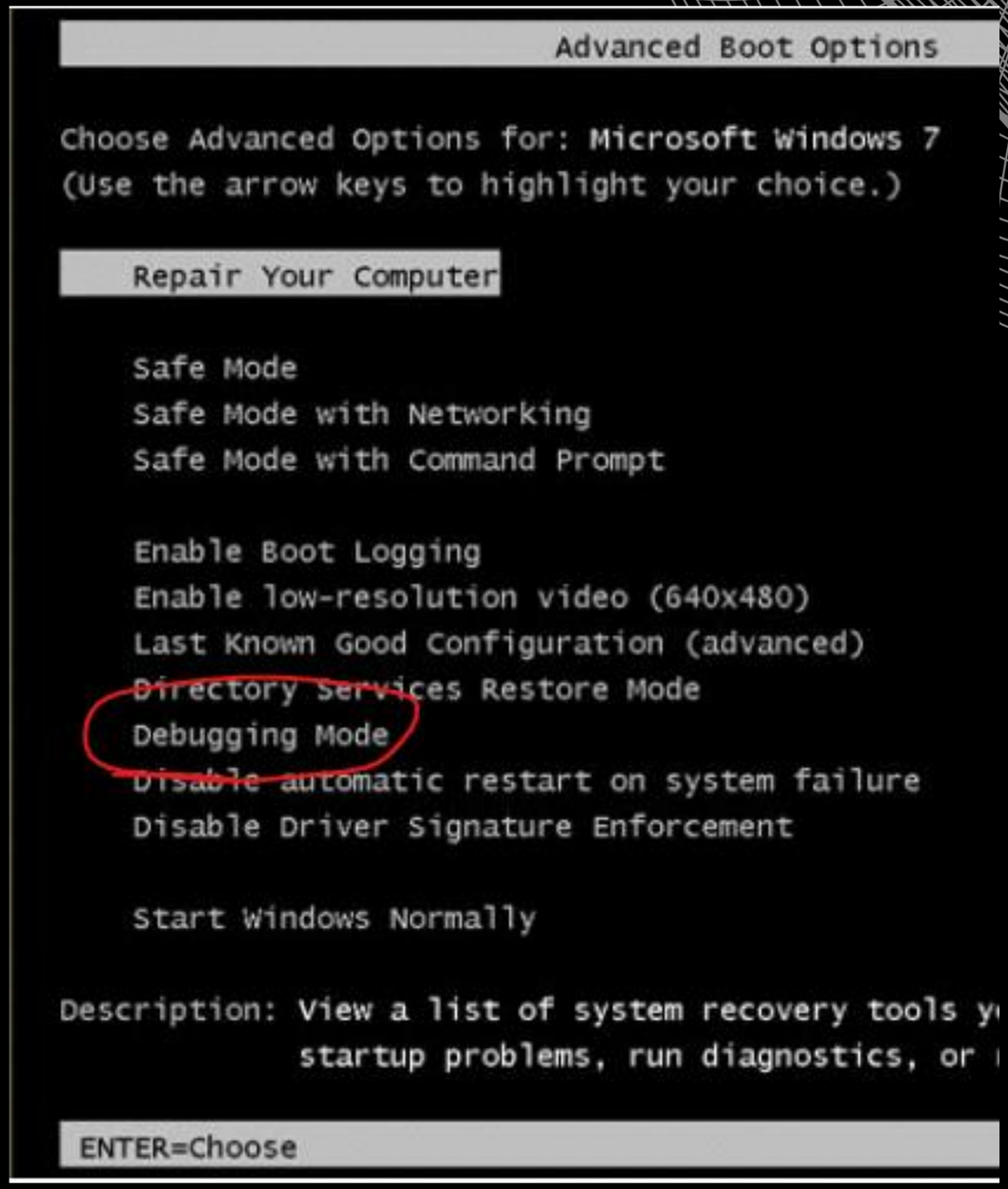
- Requires two computers
- Breakpoints stop the whole system, (not just the program).



Kernel-mode debugging (new way)

- LiveKD tool No other computer is required. This tool has some limitations.

- On windows 7, you can enter debugging mode (kernel) when you press f8 during startup.



How to debug using a debugger ☺
Two ways

- Start malware with the debugger:
  The debugger will stop the program after its entry point.

- Attach a debugger to a program that is already running:
  Stop all the threads of the program and load it into the debugger. This is useful when you're a debugging a process that is affected by malware.

Example

This code encrypts the string with XOR.

Single stepping is to stop after each Instruction.

```
mov     edi, DWORD_00406904
mov     ecx, 0x0d
LOC_040106B2
xor     [edi], 0x9C
inc     edi
loopw   LOC_040106B2
...
DWORD:00406904:   F8FDF3D01
```

```
D0F3FDF8 D0F5FEEE FDEEE5DD 9C (.............)
4CF3FDF8 D0F5FEEE FDEEE5DD 9C (L.............)
4C6FFDF8 D0F5FEEE FDEEE5DD 9C (Lo............)
4C6F61F8 D0F5FEEE FDEEE5DD 9C (Loa...........)
. . . SNIP . . .
4C6F6164 4C696272 61727941 00 (LoadLibraryA.)
```

Step-over VS Step-into

- Stepping over is to finish the subroutine (function) as a single instruction, no pauses until the subroutine is finished.

- Stepping into is to stop at each instruction of the subroutine.

Breakpoint usage

We can't tell what this program does. Set a breakpoint at the call and see what's in eax.

```
00401008    mov     ecx, [ebp+arg_0]
0040100B    mov     eax, [edx]
0040100D    call    eax
```

This program encrypts data before sending it, pause the program before it encrypts the data, and see what data it's encrypting.

```
004810D0  sub     esp, 0CCh
004810D6  mov     eax, dword_403000
004810DB  xor     eax, esp
004810DD  mov     [esp+0CCh+var_4], eax
004810E4  lea     eax, [esp+0CCh+buf]
004810E7  call    GetData
004810EC  lea     eax, [esp+0CCh+buf]
004810EF  call    EncryptData
004810F4  mov     ecx, s
004810FA  push    0                   ; flags
004810FC  push    0C8h                ; len
00481101  lea     eax, [esp+0D4h+buf]
00481105  push    eax                 ; buf
00481106  push    ecx                 ; s
00481107  call    ds:Send
```

Types of Breakpoints

Software execution

- default

- Overwrites the first byte of instruction with 0xcc (int 3 opcode) to raise exception to give control to the debugger.

Anti-debugging

- Sometimes anti-debugging malware checks this opcode (0xCC) in the memory and modifies it, and the breakpoint fails.

- Sometimes other code reads memory containing the breakpoint, and it will read 0xCC instead of original byte.

- Code that verifies integrity will notice the discrepancy

Hardware execution breakpoints

- Instead of opcode 0xCC, the DR (debug register) is used.

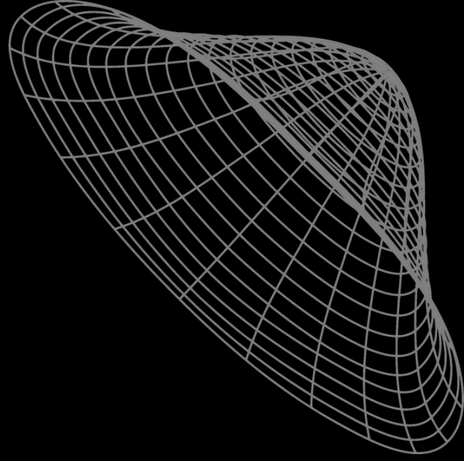
- This causes a breakpoint prior to any mov instruction that would change the contents of debug register.

- Does not detect other instructions, however.

Conditional breakpoints

- Same as software breakpoints but breaks when a condition is true.
- takes much longer than ordinary instructions.
- Sometimes so much that it never finishes.

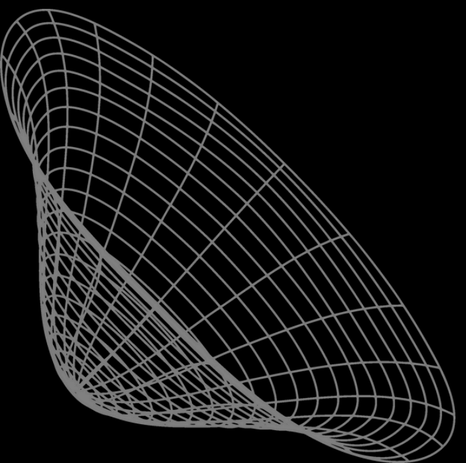
For example: breakpoint on GetProcAddress function.



Exceptions

Used by debuggers to gain control of a running program.

- Generated by breakpoints.
- Also caused by invalid memory access.
- Other exceptions, such as division by 0, etc.



Common Exceptions

Examples of common exceptions:

- int 3 (opcode 0xCC, for software breakpoint.)
- Single-stepping: execute an instruction, generate exception.
- Memory-access violation
- Violation of other privilege rules:
  For example, executing a kernel-mode instruction while the code is in user-mode.

Other debugging activities (skipping a function)

While debugging, we change the current memory of the debugged code, by changing value of the instruction pointer, or the code itself.

You can skip a function by setting a breakpoint at the call, then changing the instruction pointer to the instruction after that function. (which isn’t always safe for the program execution)

Other debugging activities (Testing a function)

You can run a function directly, without waiting for the main code to use it.

- You have to set the parameters
- This will destroy the program’s attack.
- The program won’t run properly when the function completes.

OllyDbg

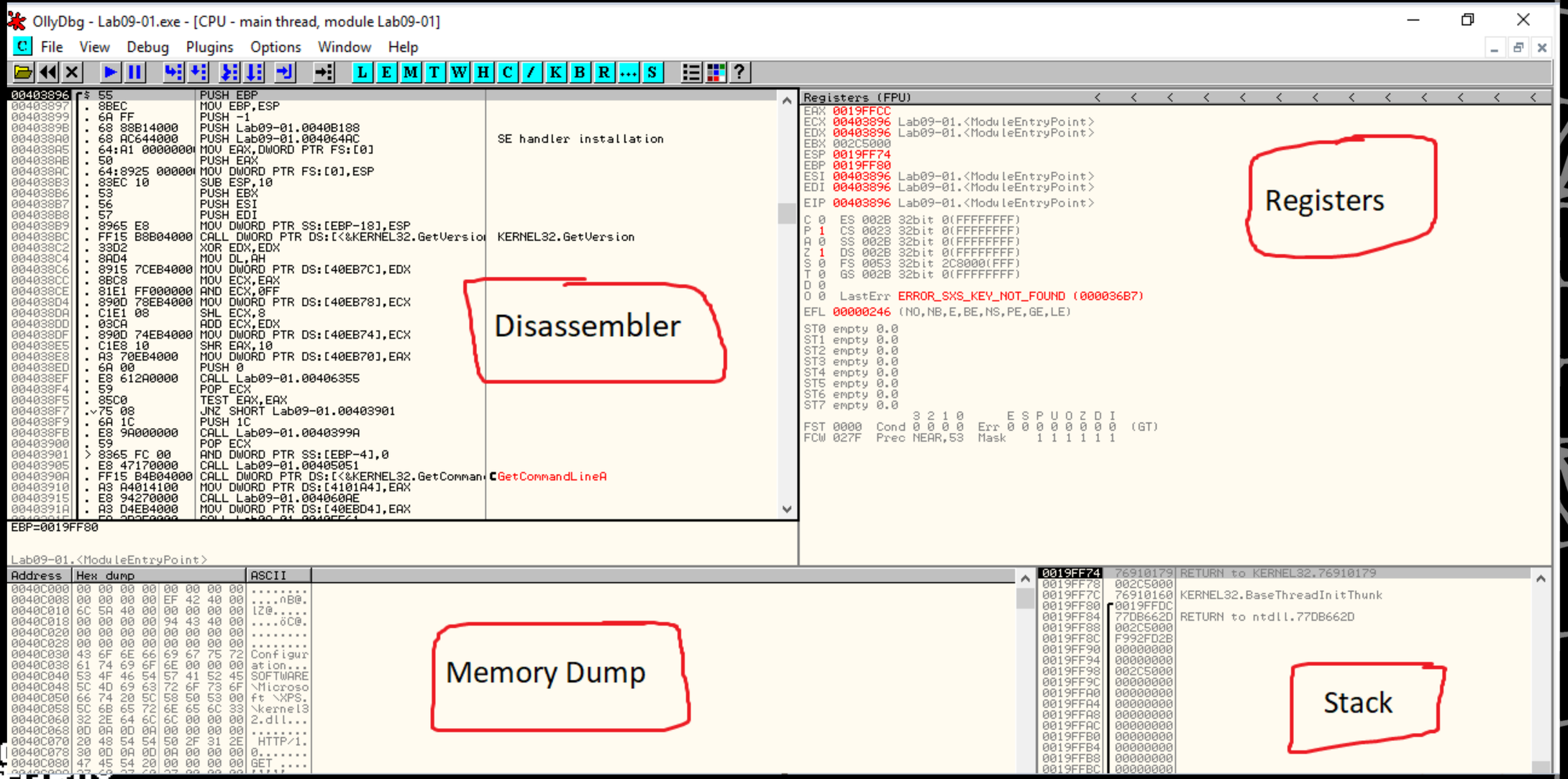
Loading malware

- You can load EXEs or DLLs into OllyDbg
- If the malware is already dunning, you can attach OllyDbg to it.

In case of opening the malware with OllyDbg, it'll stop at the entry point (WinMain, the main function if found), otherwise, it'll break at the entry point defined in PE header.

In the case of attaching OllyDbg to a process, it'll stop all the threads of the process.

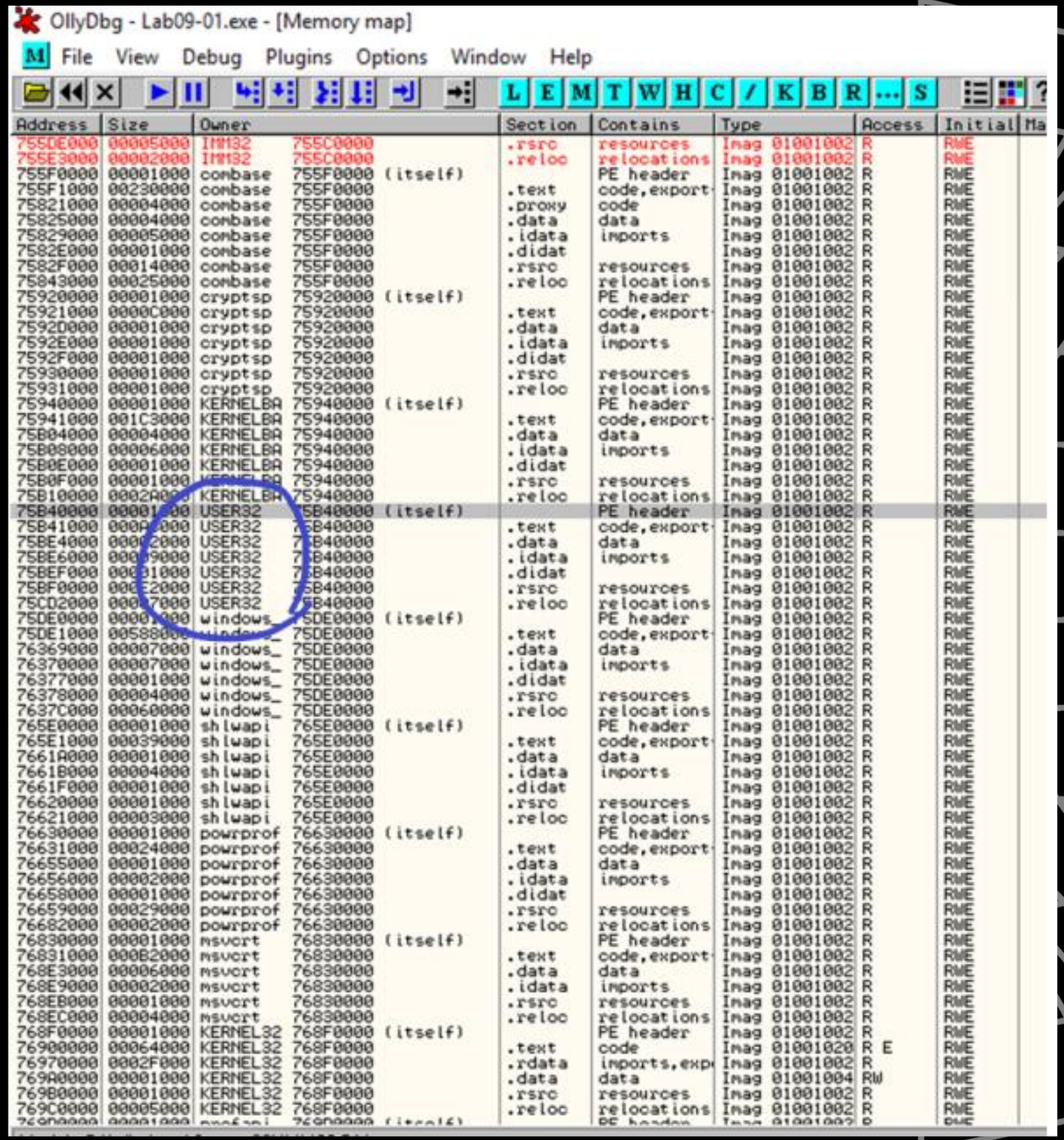
OllyDbg Interface



Memory dump

You can see which DLLs are identified, double click any row to see its memory.

You can also show the instructions in disassembler



Rebasing

- Rebasing is when a module isn’t loaded at its preferred base address.

- PEs have a preferred base address. Usually, the files are loaded at that address, which often equals 0x00400000.

- DLLs are commonly relocated, because a single EXE can import many DLLs. The list of DLLs locations are in the .reloc section of the PE header.

Absolute vs Relative Address

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

The first 3 instructions will run fine, since they use a relative address. → [ebp+var_8]

The last instruction won't always work, as it uses an absolute address to access the exact memory location.

Executing malware

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	

Notes on executing code

- Setting breakpoints is much better than just Run and Pause.
- You can execute till return
- Execute till User Code → Will execute till it hits the compiled malware code
- Single-stepping: step-into or step-over
- Sometimes step-over will miss important functions, so the code will run forever.

Breakpoints

We’ve already explained breakpoints and their types, you can refer to this pdf to have a detailed explanation on how to set each type.

Loading DLLs

- DLLs aren’t executed directly or alone.
- OllyDbg uses a dummy loaddll.exe to load them.
- breaks at the DLL entry point DLLMain once DLL is loaded.

Tracing

Tracing is recording detailed execution information.

- 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

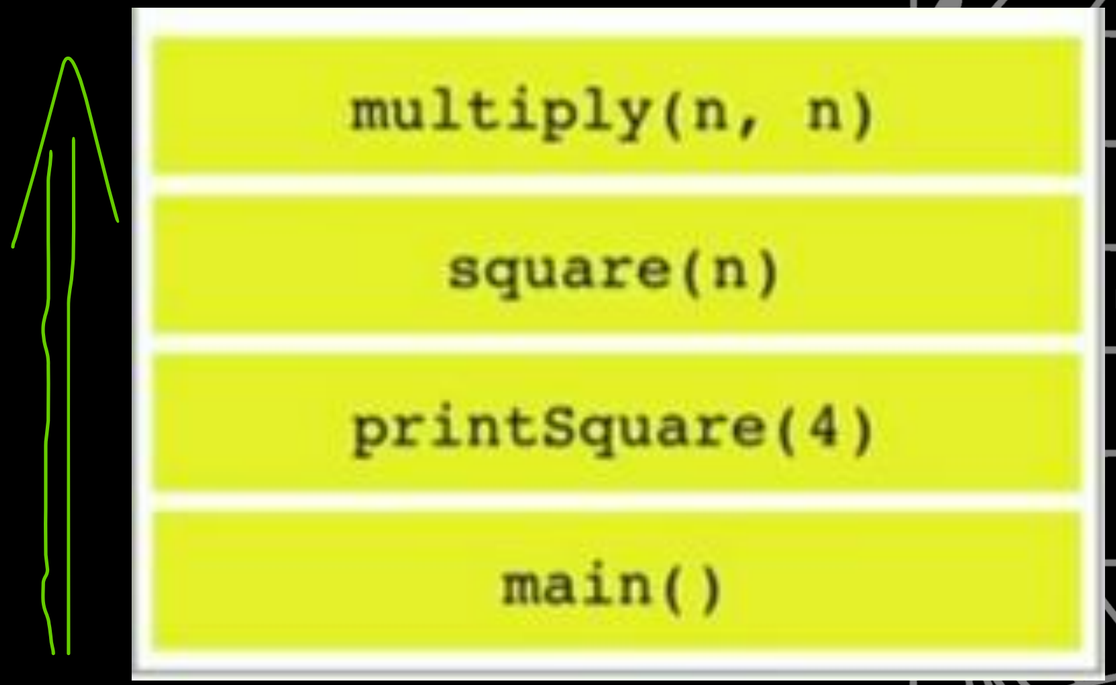
- Use minus key on keyboard to see previous instructions, But you won't see previous register values

– If you used Step Over, you cannot go back and decide to step into

Call Stack Trace

- Views execution path to a given function.

- Displays the sequence of calls to reach your current location, which is the stack of calls (hence the name).



Run Trace

- Code runs, OllyDbg saves every executed instruction and all changes to Registers and flags.

- You can go back and forth, if you go back, you could see the previous values of registers.

Handling exceptions

- When an exception occurs, OllyDbg will stop, and you have to :
    - Step into exception
    - Step over exception (most common and practical)
    - Run exception handler

Patching

- A good way to skip some instructions in the binary.

- You edit the instructions, change them to 0x90, which is NOP instruction (No-Operation)

- This will skip instructions or force a branch

Drivers

- Drivers to hardware are like DLLs to EXEs. They make hardware communicate with the kernel indirectly.

- Example: USB Flash Drive. When it’s plugged in, a device object is created.

- Application → OS → Device Driver → hardware.

- Multiple devices can use the same Driver (just like multiple EXEs can use the same DLL.)

Loading drivers

- Drivers are loaded in the kernel.

- DriverEntry procedure is called (like DLLMain)

- Callback objects and functions can be used by OS (like DLL Exports)

- Difference is that DLLs Exports are for user-mode programs, while Driver Callback functions will be a part of kernel-mode programs which interact with the kernel.

Kernel and user-mode calls

You can see how the program is using a malicious driver, even if it doesn't include malicious DLL.

Ntoskrnl.exe has core OS functions, so malware will often import functions from one it so it can manipulate the kernel.

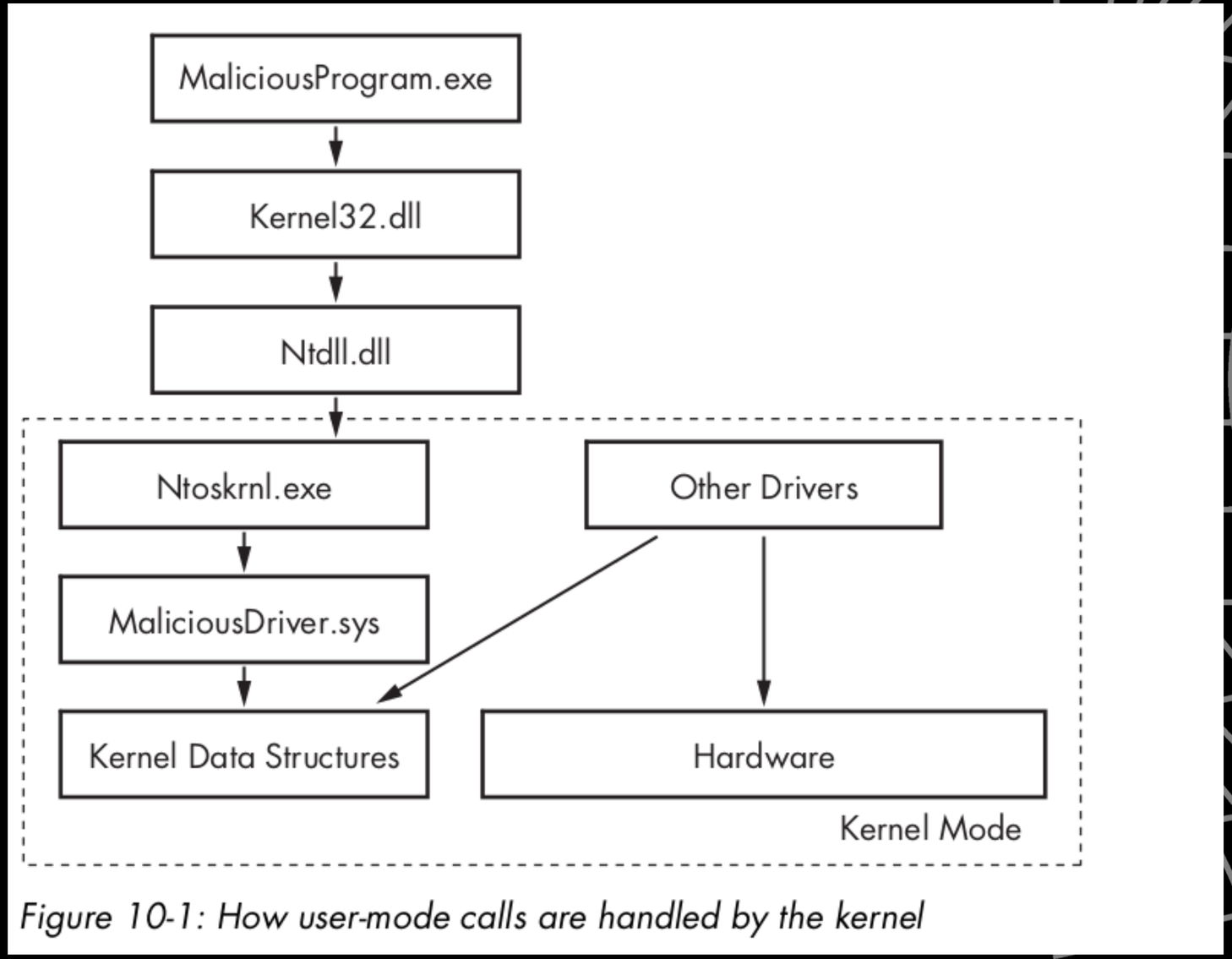


Figure 10-1: How user-mode calls are handled by the kernel

Practical Malware Analysis

CH 6: Malware behavior

HTU

CYBER SECURITY CLUB

Downloaders and launchers

They form the first stages of malware.

- Downloader often uses URLDownloadtoFileA, followed by WinExec.

- Launcher (loader) will install the malware and installs malware in unexpected places lie .rsrc section of PE.

Backdoor

- A backdoor will provide a persistent way to remotely access a victim machine.
- Most common.
- Communication is usually through http/s port or other common ports to prevent detection.
- Often manipulates registry, creates directories, searches files, etc.

Reverse shell

- Infected machine calls out to attacker, asking for commands to execute.
- On Windows, this is done by calling CreateProcess and manipulating STARTUPINFO structure. Then creating a socket to a remote machine and bind the socket to stdin/stdout/stderr for cmd.exe.

RATs
(Remote Administration Tools)

- Carried on remotely managing a group of compromised computers.

- Used in targeted attacks. The structure is client-server.

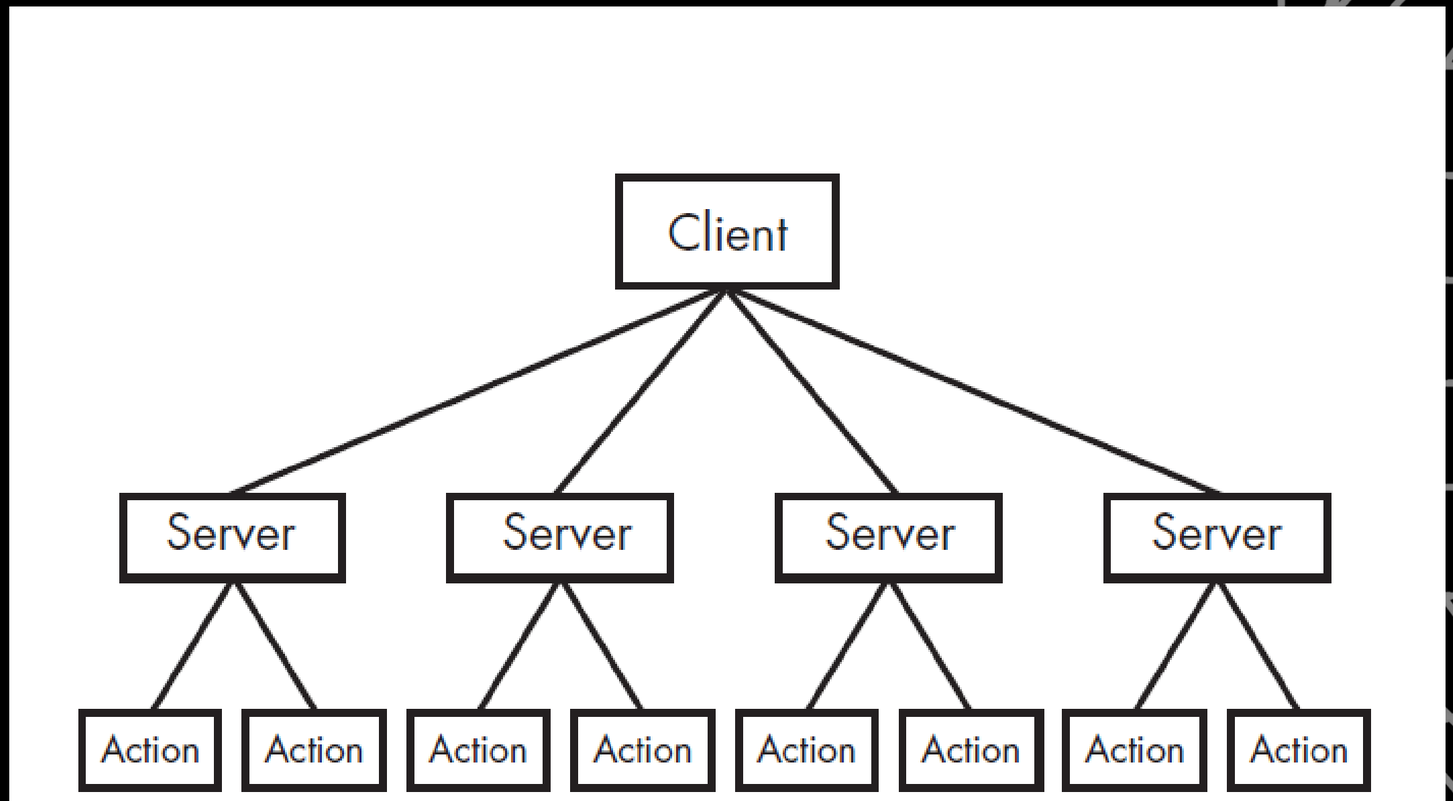


Figure 11-1: RAT network structure

Botnets

- A botnet is like a RAT, but it’s for mass attacks.

- It has much bigger number of compromised machines than RATs. Compromised machines are called bots/zombies.

- All bots are controlled at once, while in RAT the victims are controlled one by one.

Credential stealers

- Dump stored data, such as password hashes, cached cookies, etc.

- Keyloggers.

Keyloggers (kernel-based)

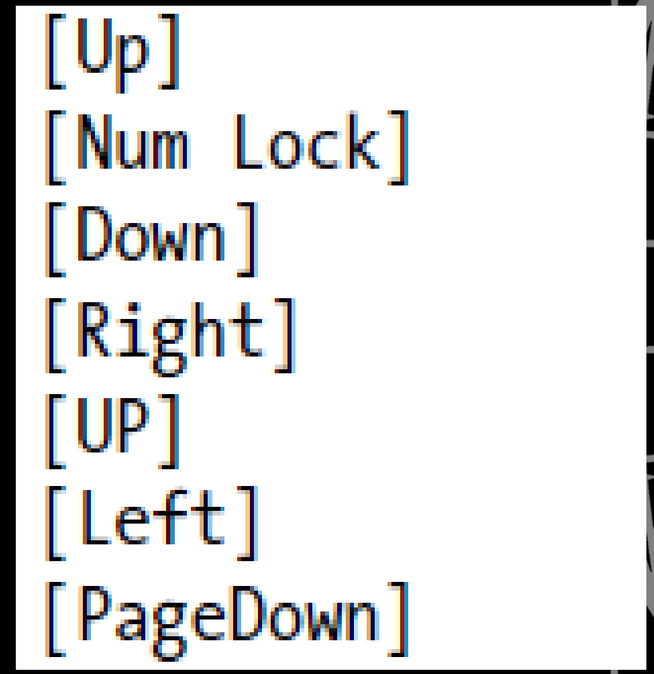
- Built as keyboard driver.
- Bypasses user-space protection.
- Difficult to detect using user-space programs.

Keyloggers (user-mode)

- Uses Windows API along with hooking.

- Uses SetWindowsHookEx function to notify the malware each time a key is pressed.

- When running strings on a keylogger, you might see terms like Num Lock, Up, Down.



Downloaders and launchers

They form the first stages of malware.

- Downloader often uses URLDownloadtoFileA, followed by WinExec.

- Launcher (loader) will install the malware and installs malware in unexpected places lie .rsrc section of PE.

Persistence

- Making your malware stay in the victim machine for a long time.

- Done through modifying registry, Trojanizing binaries, or DLL Load-Order hijacking.

Registry modification

- A very common registry that malware often modifies is HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Run

- This registry is set to run on each boot.

- Registry actions can be detected using ProcMon when running the malware.

Trojanizing binaries

- Malware patches bytes of a system binary so it'll execute the malware next time that binary is loaded

- DLLs are common targets
- Modifies entry function so it jumps to the malware inserted in empty portion of the binary. Then it continues executing the binary normally.

Trojanized binaries

Original code	Trojanized code

`DllEntryPoint(HINSTANCE hinstDLL,`
`  DWORD fdwReason, LPVOID lpReserved)`
 	
`mov  edi, edi`
`push  ebp`
`mov  ebp, esp`
`push  ebx`
`mov  ebx, [ebp+8]`
`push  esi`
`mov  esi, [ebp+0Ch]`
	`DllEntryPoint(HINSTANCE hinstDLL,`
	`  DWORD fdwReason, LPVOID lpReserved)`
	
	`jmp  DllEntryPoint_0`

DLL Load-Order Hijacking

- The registry key KnownDLLs has a list of specific known DLL locations.
- Overrides the search order for listed DLLs, makes them load faster. And prevents load-order hijacking.

- Load-order hijacking can be used on binaries in directories other then system32 that load DLLs in system32 and aren’t protected by KnownDLLs.

DLL Load-Order Hijacking

For example, explorer.exe. This binary is in \Windows. Loads ntshrui.dll from System32, but ntshrui.dd isn’t a known DLLs.

Default search is performed, so a malicious ntshrui.dll can be loaded instead.

Privilege escalation

- Metasploit has many privilege escalation exploits.

- privilege escalation can also be done using DLL order-load hijacking.

Covert Malware launching

Malware used to be visible in windows task manager. So malware authors try to blend their malware into normal windows landscape.

As we discussed before, a malware launcher sets itself or another piece of code for future execution.

Malware launching

The malicious piece of code is often compressed.

Resource extraction uses API functions such as FindResource, LoadResource, and SizeofResource.

Common techniques are process injection and DLL injection.

Process injection

- Injecting code to an already running process.

- Can be DLL injection or direct injection.

- Common API calls:
  - **VirtualAllocEx** to allocate space in remote process’s memory
  - **WrtieProcessMemory** to write to it.

Process injection

For example, launcher wants internet access to download code, but process-specific firewall won’t let it access the internet.

We can inject that loader to a process (e.g. IExplorer) that already has internet access.

DLL injection

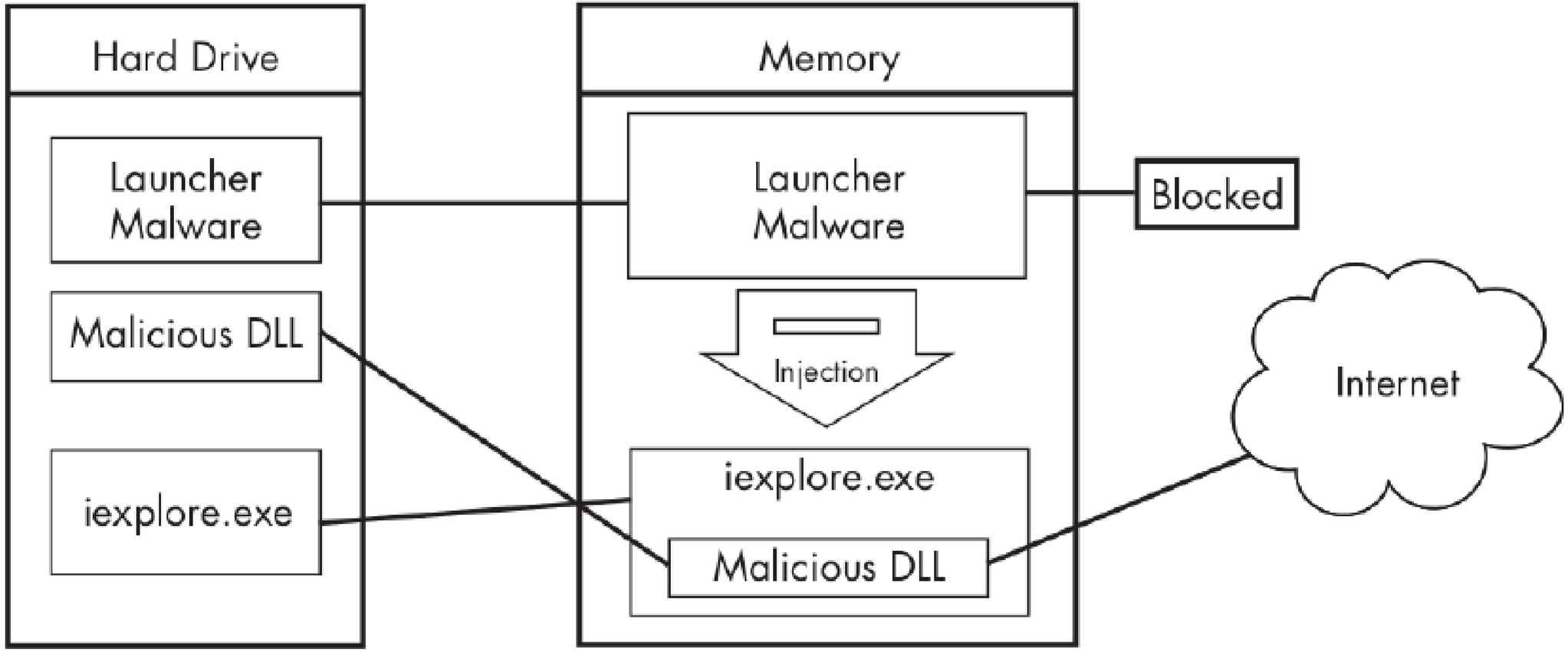


Figure 12-1: DLL injection—the launcher malware cannot access the Internet until it injects into iexplore.exe.

DLL Injection pseudocode

```
hVictimProcess = OpenProcess(PROCESS_ALL_ACCESS, 0, victimProcessID ❶);

pNameInVictimProcess = VirtualAllocEx(hVictimProcess,...,sizeof(maliciousLibraryName),...,...);
WriteProcessMemory(hVictimProcess,...,maliciousLibraryName, sizeof(maliciousLibraryName),...);
GetModuleHandle("Kernel32.dll");
GetProcAddress(...,"LoadLibraryA");
❷ CreateRemoteThread(hVictimProcess,...,...,LoadLibraryAddress,pNameInVictimProcess,...,...);
```

Listing 12-1: C Pseudocode for DLL injection

Direct injection

- Injects code directly into remote process.

- No DLLs, difficult to write.

- Requires a lot of testing to make sure it doesn’t mess with the flow of the native process.

Process replacement

- Instead of writing a process inside a remote process memory, we replace the whole memory of that process.

- Avoids the risk of crashing a process with process injection.

- Commonly replaces **svchost.exe**

Suspended state

In a suspended state, the process is loaded into memory, but primary thread is suspended (so malware can overwrite it before the code runs)

```
CreateProcess(...,"svchost.exe",...,CREATE_SUSPEND,...);
ZwUnmapViewOfSection(...);
VirtualAllocEx(...,ImageBase,SizeOfImage,...);
WriteProcessMemory(...,headers,...);
for (i=0; i < NumberOfSections; i++) {
 ❶ WriteProcessMemory(...,section,...);
}
SetThreadContext();
...
ResumeThread();
```

Listing 12-3: C pseudocode for process replacement

Direct injection

- Injects code directly into remote process.

- No DLLs, difficult to write.

- Requires a lot of testing to make sure it doesn’t mess with the flow of the native process.

Hook Injection

- Windows hooks intercept messages destined for applications.

- Malicious hooks:
  - Ensure that a malicious code will run whenever a message is intercepted.
  - Ensure that a DLL will be loaded in a victim process’s memory.

Hook injection

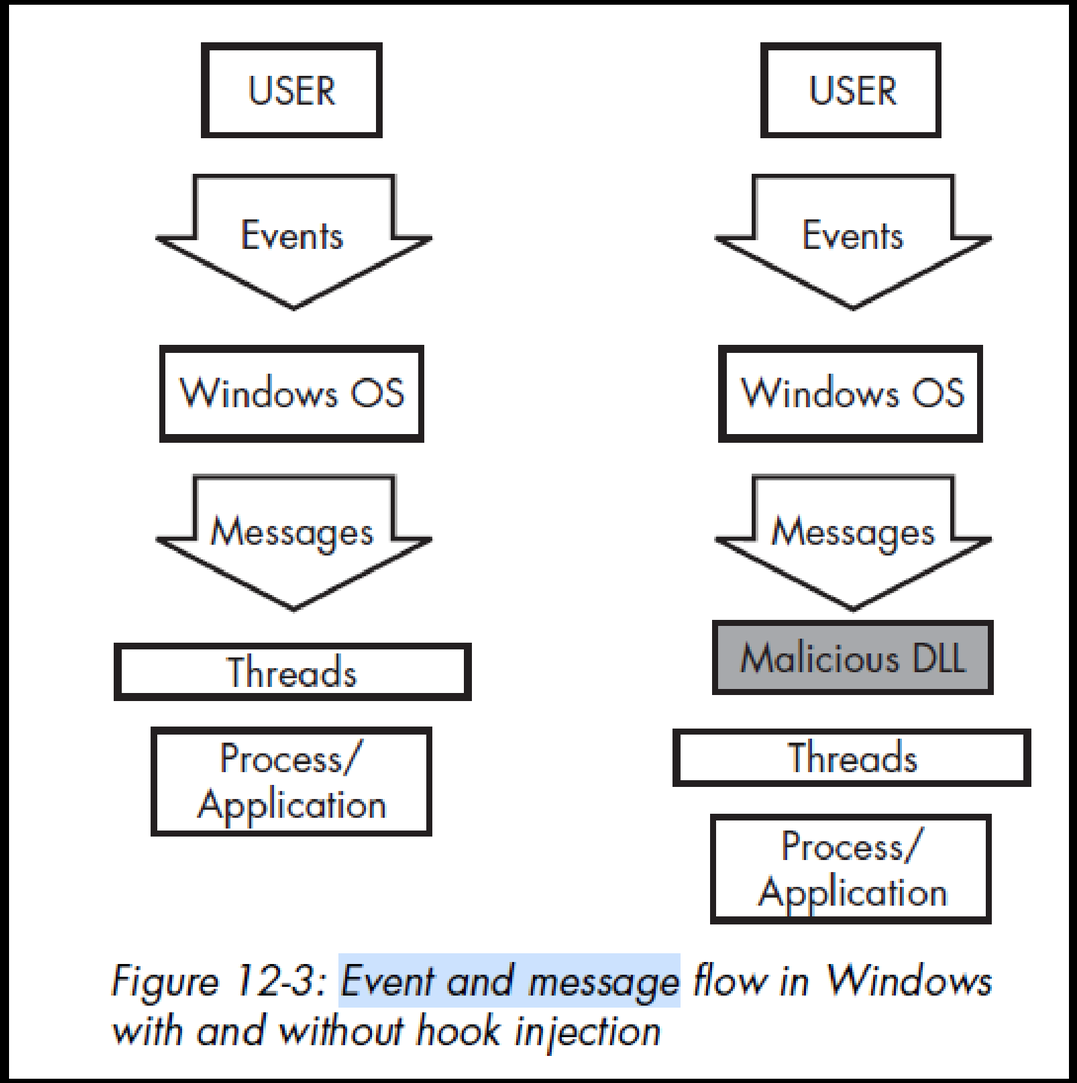


Figure 12-3: Event and message flow in Windows with and without hook injection

Keyloggers using hooks

In keyloggers, **WH_KEYBOARD** and **WH_KEYBOARD_DLL procedures are used.**

We can use **SetWindowsHookEx** for remote windows hooking.

Parameters:

- **idHook** → Type of hook procedure.
- **lpfn** → pointer to hook procedure
- **hMod** → Handle to DLL or local module containing hook procedure.
- dwThreadId → Thread to associate hook with.

Encoding

Malware uses encoding for some reasons:
- Hide some information (e.g. C2 domains)
- Save information for staging file
- Store strings needed by malware.
- Hide suspicious strings.

Encryption is also used, and it’s much harder to decrypt than decoding encoded strings.

Common encodings

- Ceasar cipher: Moving each letter by a certain number.
For example, CMD.EXE → FPG.HAH.

- XOR: Bitwise operation. Uses a key to encrypt data.
  For example, malware ^ 0xa → gkf}kxo
NOTE: XOR loops can be found in IDA Pro.

- Base64: Includes [A-Za-z0-9\+\/]. Refer to it here

Encryption

- Much harder to break than encoding. Depends on the encryption algorithm

- For example, AES, SSL, etc.

- Cryptographic libraries required.
- Sometimes the code becomes less portable
- Symmetric encryption requires a way to hide the key, otherwise it’s easy to decrypt.

Entropy

Entropy is the randomness of an executable.

High entropy probably means that there’s encrypted content.

Can be found using IDA Pro Entropy Plugin.

Note that sometimes multiple methods of encryption can be applied in the same malware.

Entropy

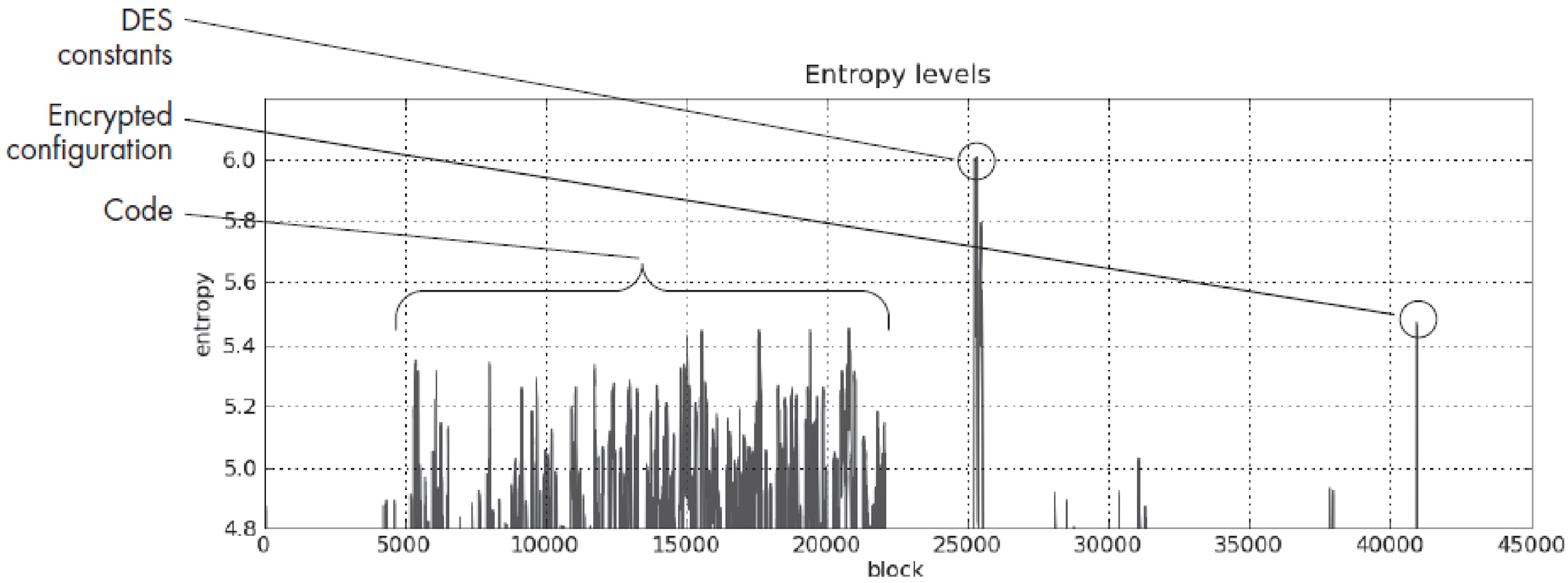
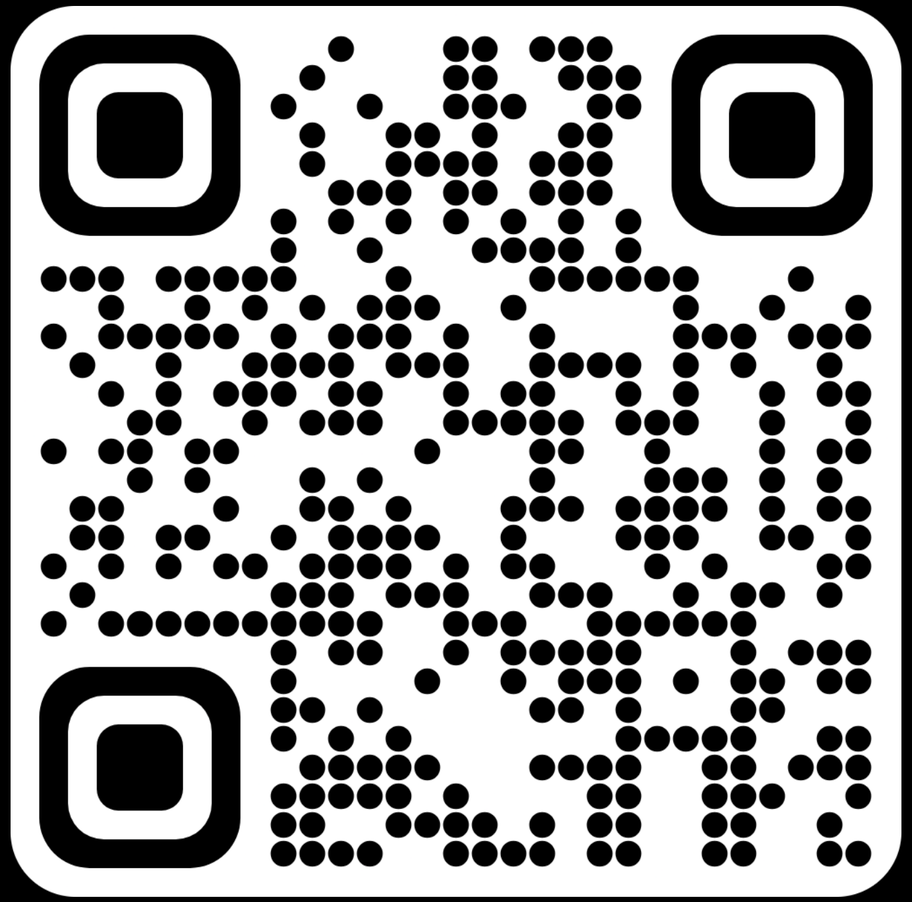


Figure 13-13: Entropy graph for a malicious executable

HTU

Quiz



CYBER SECURITY CLUB

THANK YOU

CYBER SECURITY CLUB

HTU