School of
Engineering and Computer Science
Te Kura Mātai Pūkaha, Pūrorohiko

CYBR 473 T1 2023

Malware and Reverse Engineering

Debugging

Chapter 8: "*Practical Malware Analysis: The Hands-on Guide to Dissecting Malicious Software*", Michael Sikorski and Andrew Honig, 2012

1897
VICTORIA UNIVERSITY OF
WELLINGTON
TE HERENGA WAKA

Disassemblers vs. Debuggers

- A **disassembler** like IDA Pro shows the state of the program just **before execution** begins
- **Debuggers** show
  - Every memory location
  - Register
  - Argument to every function
  - **At any point** during the processing
  - And let you **change** them

Two Debuggers

- OllyDbg
  - Most popular for malware analysis
  - **User-mode** debugging only
  - IDA Pro has a built-in debugger, but it's not as easy to use or powerful as Ollydbg

- Windbg
  - Supports **kernel-mode** debugging

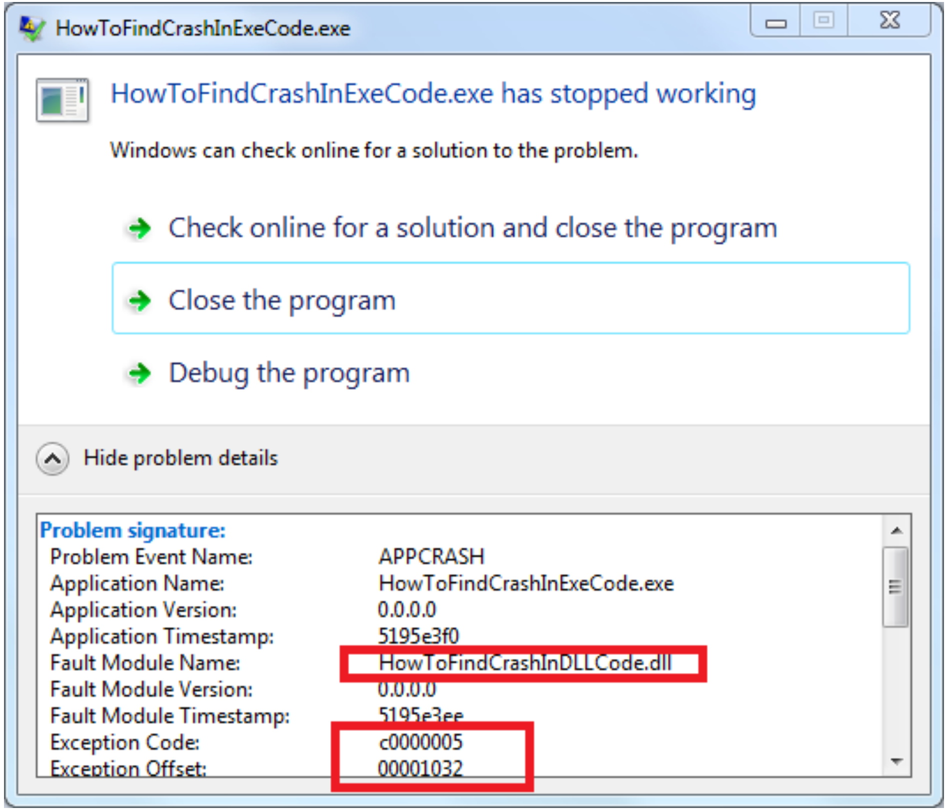
Source-Level vs. Assembly-Level Debuggers

- **Source-level** debugger
  - Usually built into the development platform
  - Can set breakpoints (which stop at lines of code)
  - Can step through the program one line at a time

- **Assembly-level** debuggers (**low-level**)
  - Operate on **assembly code** rather than source code
  - Malware analysts are usually forced to use them because they <u>don't have source code</u>

Windows Crashes

- When an app crashes, Windows may offer to open it in a debugger
- Usually it uses Windbg



KERNEL-MODE VS. USER-MODE DEBUGGING

User Mode Debugging

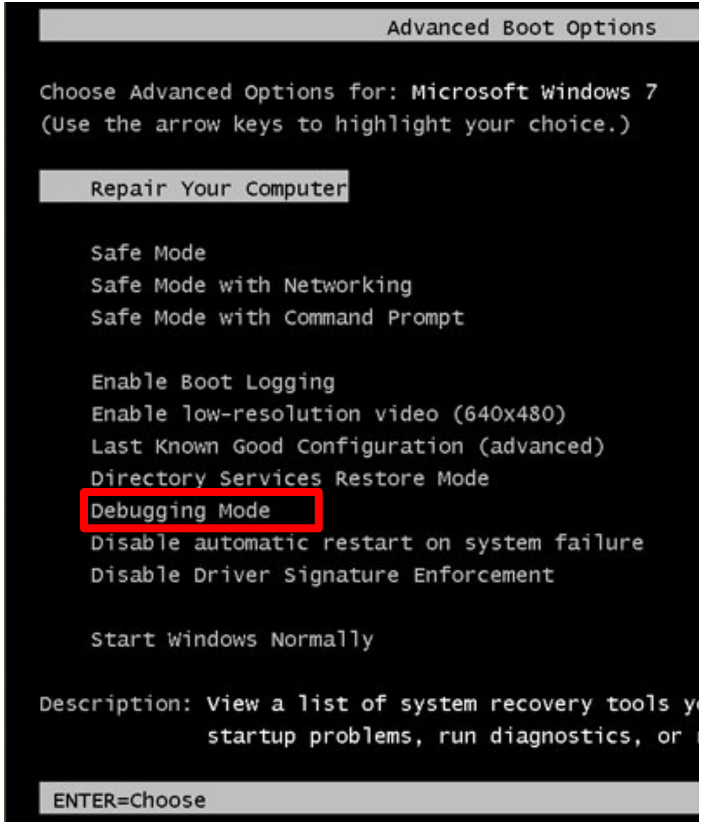
- Debugger **runs on the same system** as the code being analysed
- Debugging a **single executable**
- **Separated** from other executables by the OS

Kernel Mode Debugging (*The Old Way*)

- Requires **two computers** because there is **only one kernel** per computer
- If the kernel is at a breakpoint, the system stops
- One computer runs the **code** being debugged
- Other computer runs the **debugger**
- OS must be configured to allow kernel debugging
- Two machines must be **connected**

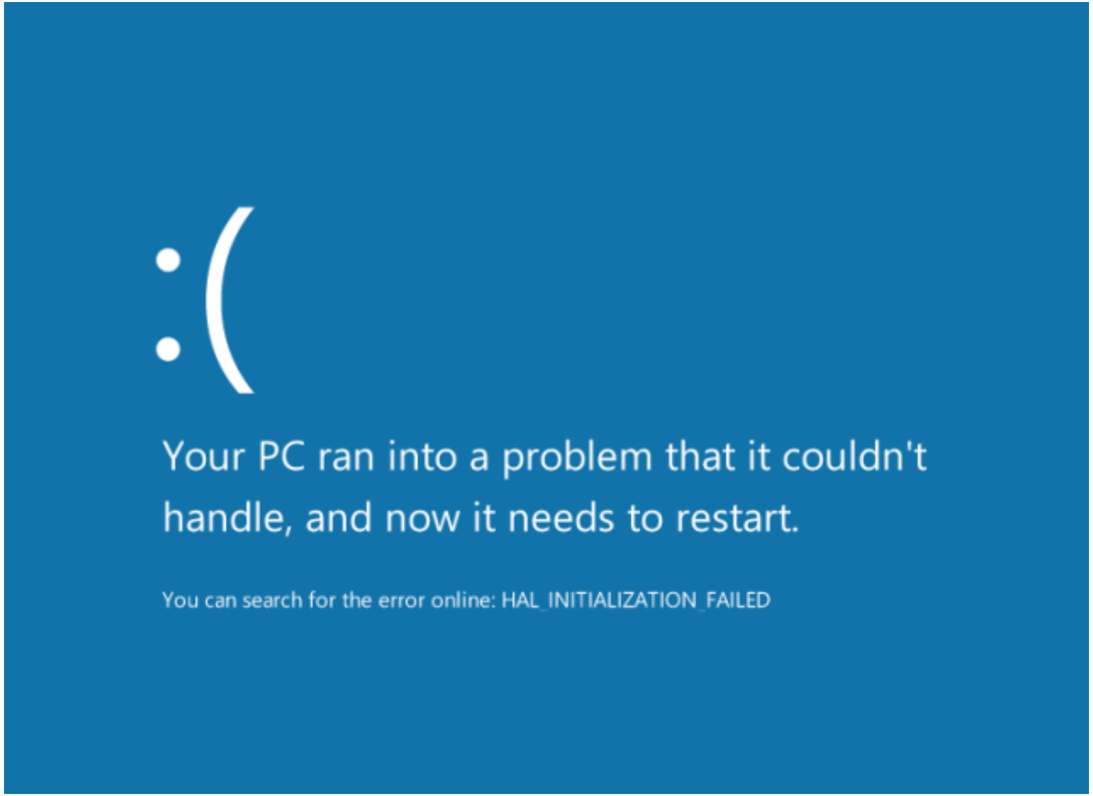
Windows 7 Advanced Boot Options

- Press F8 during startup
- “Debugging Mode”



Side-Effect of Debug Mode

- PrntScn key causes BSOD
- Use Shift+PrntScn instead



Kernel Mode Debugging (*The New Way*)

- Mark Russinovich's Livekd tool allows you to debug the kernel with only one computer!
- MUCH easier :)
- Tool has some limitations

How LiveKd Works

After giving it some thought, I realized that I could fool the debuggers into thinking that they were looking at a crash dump file by implementing a file system filter driver that presented a "virtual" crash dump file debuggers could open. Since a crash dump file is simply a file header followed by the contents of physical memory, the driver could satisfy reads of the virtual dump file with the contents of physical memory, which the driver could easily read from the \Device\Physical Memory section object the memory manager creates. A couple of weeks later, LiveKd was born.

USING A DEBUGGER

Two Ways

- Start the program with the debugger
  - It stops running immediately prior to the execution of its entry point
- Attach a debugger to a program that is already running
  - All its threads are paused
  - Useful to debug a process that is affected by malware

Single-Stepping

- Simple, but slow

- Don't get bogged down in details

Example

- This code decodes the string with XOR

```
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.)
```

Stepping-over vs. Stepping-into

- Single step executes one instruction
- **Step-over** call instructions
  - Completes the call and returns without pausing
  - Decreases the amount of code you need to analyse
  - Might miss important functionality, especially if the function never returns
- **Step-into** a call
  - Moves into the function and stops at its first command

Pausing Execution with Breakpoints

- A program that is paused at a **breakpoint** is called **broken**

- Example
  - You can't tell where this call is going
  - 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
```

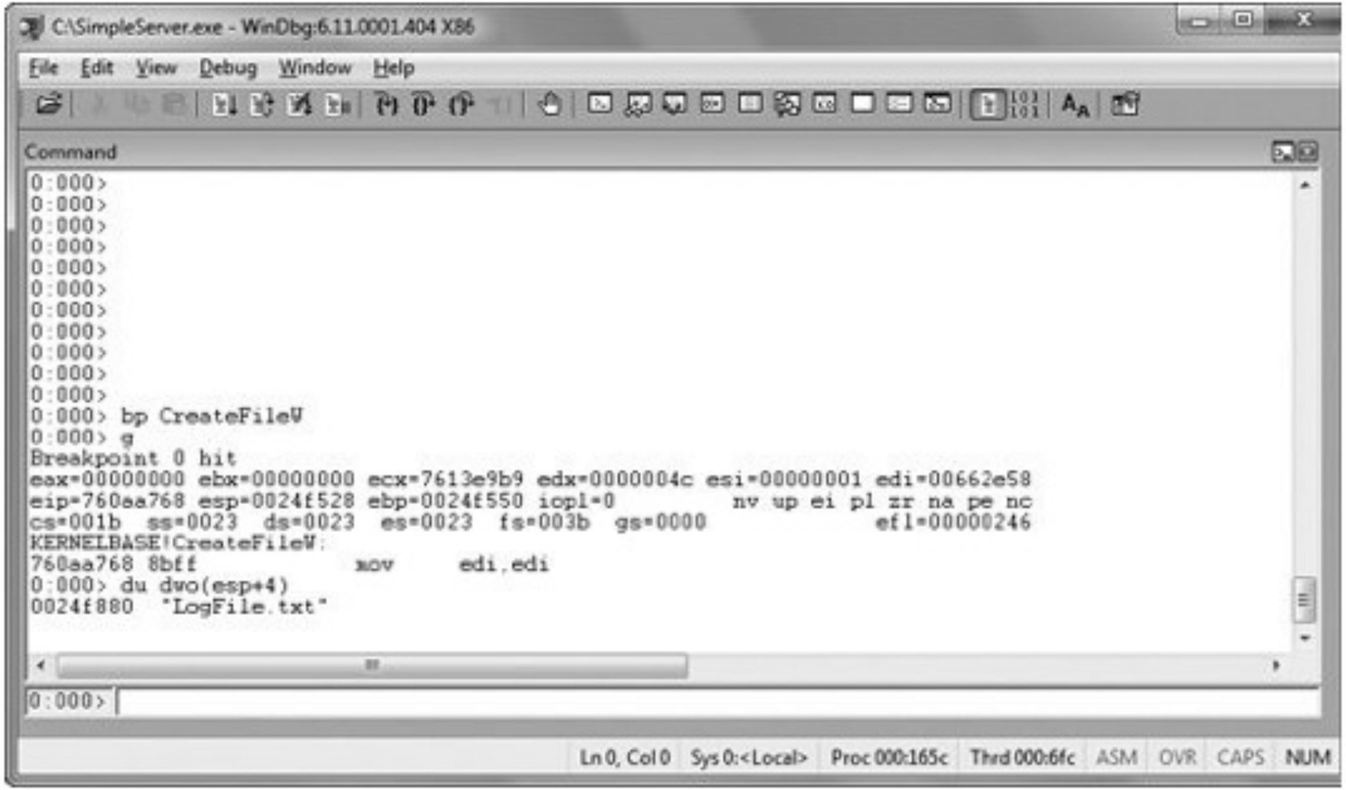
Using a Debugger to Determine a Filename

- This code calculates a **filename** and then creates the file

- Set a breakpoint at `CreateFileW` and look at the stack to see the filename

```
0040100B  xor     eax, esp
0040100D  mov     [esp+0D0h+var_4], eax
00401014  mov     eax, edx
00401016  mov     [esp+0D0h+NumberOfBytesWritten], 0
0040101D  add     eax, 0FFFFFFFEh
00401020  mov     cx, [eax+2]
00401024  add     eax, 2
00401027  test    cx, cx
0040102A  jnz     short loc_401020
0040102C  mov     ecx, dword ptr ds:a_txt ; ".txt"
00401032  push    0                ; hTemplateFile
00401034  push    0                ; dwFlagsAndAttributes
00401036  push    2                ; dwCreationDisposition
00401038  mov     [eax], ecx
0040103A  mov     ecx, dword ptr ds:a_txt+4
00401040  push    0                ; lpSecurityAttributes
00401042  push    0                ; dwShareMode
00401044  mov     [eax+4], ecx
00401047  mov     cx, word ptr ds:a_txt+8
0040104E  push    0                ; dwDesiredAccess
00401050  push    edx              ; lpFileName
00401051  mov     [eax+8], cx
00401055  call    CreateFileW ; CreateFileW(x,x,x,x,x,x,x)
```

WinDbg



C:\SimpleServer.exe - WinDbg:6.11.0001.404 X86
File Edit View Debug Window Help
Command
0:000> bp CreateFileW
0:000> g
Breakpoint 0 hit
eax=00000000 ebx=00000000 ecx=7613e9b9 edx=0000004c esi=00000001 edi=00662e58
eip=760aa768 esp=0024f528 ebp=0024f550 iopl=0 nv up ei pl zr na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000246
KERNELBASE!CreateFileW:
760aa768 8bff mov edi,edi
0:000> du dwo(esp+4)
0024f880 "LogFile.txt"
Ln 0, Col 0 Sys 0:<Local> Proc 000:165c Thrd 000:6fc ASM OVR CAPS NUM

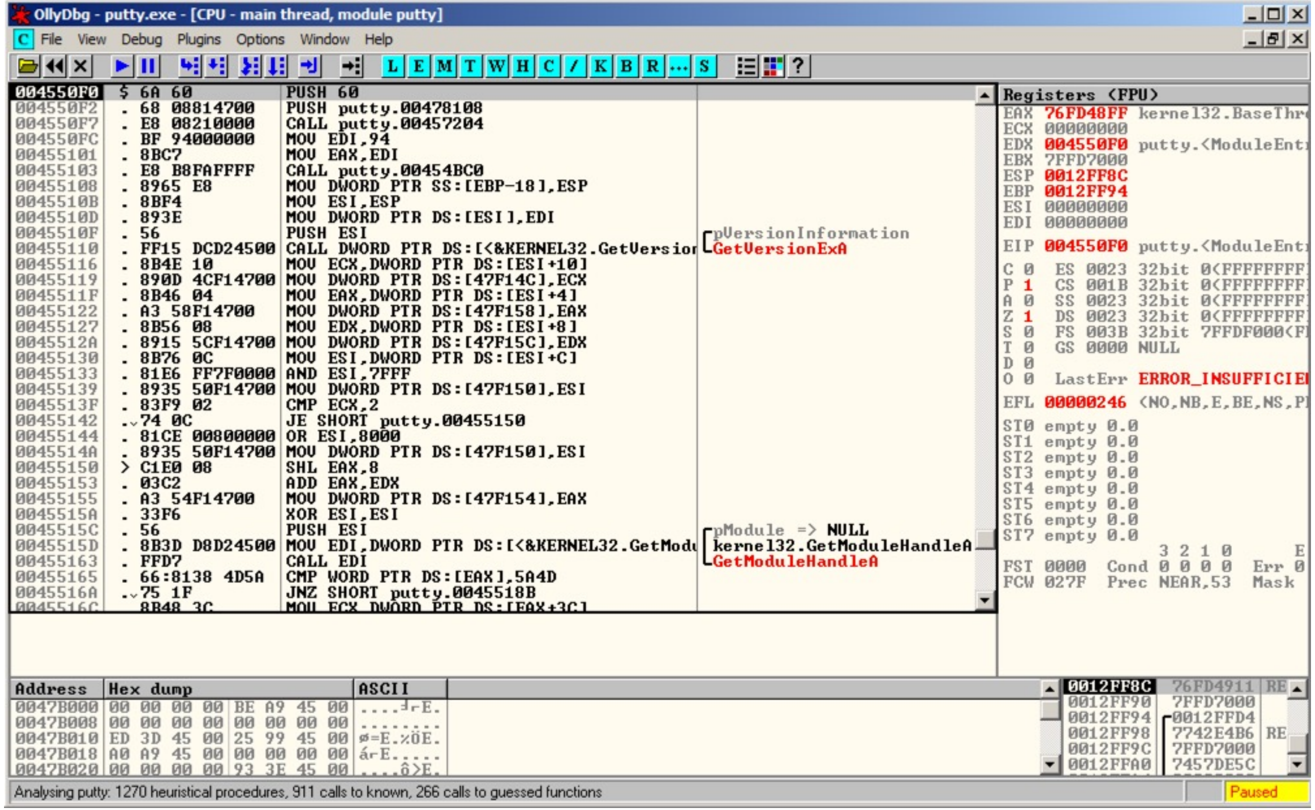
Encrypted Data

- Suppose malware sends encrypted network data

- Set a breakpoint before the data is encrypted and view it

```
004010D0  sub    esp, 0CCh
004010D6  mov    eax, dword_403000
004010DB  xor    eax, esp
004010DD  mov    [esp+0CCh+var_4], eax
004010E4  lea    eax, [esp+0CCh+buf]
004010E7  call   GetData
004010EC  lea    eax, [esp+0CCh+buf]
004010EF  call   EncryptData
004010F4  mov    ecx, s
004010FA  push   0                  ; flags
004010FC  push   0C8h               ; len
00401101  lea    eax, [esp+0D4h+buf]
00401105  push   eax                ; buf
00401106  push   ecx                ; s
00401107  call   ds:Send
```

OllyDbg



OllyDbg - putty.exe - [CPU - main thread, module putty]
File View Debug Plugins Options Window Help
L E M T W H C / K B R ... S
004550F0 $ 6A 60 PUSH 60
004550F2 . 68 08814700 PUSH putty.00478108
004550F7 . E8 08210000 CALL putty.00457204
004550FC . BF 94000000 MOV EDI,94
00455101 . 8BC7 MOV EAX,EDI
00455103 . E8 B8FAFFFF CALL putty.00454BC0
00455108 . 8965 E8 MOV DWORD PTR SS:[EBP-18],ESP
0045510B . 8BF4 MOV ESI,ESP
0045510D . 893E MOV DWORD PTR DS:[ESI],EDI
0045510F . 56 PUSH ESI
00455110 . FF15 DCD24500 CALL DWORD PTR DS:[<&KERNEL32.GetVersior
00455116 . 8B4E 10 MOV ECX,DWORD PTR DS:[ESI+10]
00455119 . 890D 4CF14700 MOV DWORD PTR DS:[47F14C],ECX
0045511F . 8B46 04 MOV EAX,DWORD PTR DS:[ESI+4]
00455122 . A3 58F14700 MOV DWORD PTR DS:[47F158],EAX
00455127 . 8B56 08 MOV EDX,DWORD PTR DS:[ESI+8]
0045512A . 8915 5CF14700 MOV DWORD PTR DS:[47F15C],EDX
00455130 . 8B76 0C MOV ESI,DWORD PTR DS:[ESI+C]
00455133 . 81E6 FF7F0000 AND ESI,7FFF
00455139 . 8935 50F14700 MOV DWORD PTR DS:[47F150],ESI
0045513F . 83F9 02 CMP ECX,2
00455142 .v 74 0C JE SHORT putty.00455150
00455144 . 81CE 00800000 OR ESI,8000
0045514A . 8935 50F14700 MOV DWORD PTR DS:[47F150],ESI
00455150 > C1E0 08 SHL EAX,8
00455153 . 03C2 ADD EAX,EDX
00455155 . A3 54F14700 MOV DWORD PTR DS:[47F154],EAX
0045515A . 33F6 XOR ESI,ESI
0045515C . 56 PUSH ESI
0045515D . 8B3D D8D24500 MOV EDI,DWORD PTR DS:[<&KERNEL32.GetModu
00455163 . FFD7 CALL EDI
00455165 . 66:8138 4D5A CMP WORD PTR DS:[EAX],5A4D
0045516A .v 75 1F JNZ SHORT putty.0045518B
pVersionInformation
GetVersionExA
pModule => NULL
kernel32.GetModuleHandleA
GetModuleHandleA
Registers (FPU)
EAX 76FD48FF kernel32.BaseThr
ECX 00000000
EDX 004550F0 putty.<ModuleEnt
EBX 7FFD7000
ESP 0012FF8C
EBP 0012FF94
ESI 00000000
EDI 00000000
EIP 004550F0 putty.<ModuleEnt
C 0 ES 0023 32bit 0(FFFFFFFF
P 1 CS 001B 32bit 0(FFFFFFFF
A 0 SS 0023 32bit 0(FFFFFFFF
Z 1 DS 0023 32bit 0(FFFFFFFF
S 0 FS 003B 32bit 7FFDF000(F
T 0 GS 0000 NULL
D 0
O 0 LastErr ERROR_INSUFFICIE
EFL 00000246 (NO,NB,E,BE,NS,P
ST0 empty 0.0
ST1 empty 0.0
ST2 empty 0.0
ST3 empty 0.0
ST4 empty 0.0
ST5 empty 0.0
ST6 empty 0.0
ST7 empty 0.0
FST 0000 Cond 0 0 0 0 Err 0
FCW 027F Prec NEAR,53 Mask
Address Hex dump ASCII
0047B000 00 00 00 00 BE A9 45 00
0047B008 00 00 00 00 00 00 00 00
0047B010 ED 3D 45 00 25 99 45 00
0047B018 A0 A9 45 00 00 00 00 00
0047B020 00 00 00 00 93 3E 45 00
0012FF8C 76FD4911
0012FF90 7FFD7000
0012FF94 0012FFD4
0012FF98 7742E4B6
0012FF9C 7FFD7000
0012FFA0 7457DE5C
Analysing putty: 1270 heuristical procedures, 911 calls to known, 266 calls to guessed functions
Paused

Types of Breakpoints

- **Software** execution

- **Hardware** execution

- **Conditional**

Software Execution Breakpoints

- The **default** option for most debuggers
- Debugger **overwrites the first byte** of the instruction with **`0xCC`**
  - The instruction for **`INT 3`**
  - An interrupt designed for use with debuggers
  - When the breakpoint is executed, the OS generates an **exception** and transfers control to the **debugger**

Memory Contents at a Breakpoint

- There's a breakpoint at the push instruction
- Debugger says it's `0x55`, but it's really `0xCC`

Disassembly view | **Memory dump**

```
00401130 55                 1push  ebp               00401130 2CC 8B EC 83
00401131 8B EC                mov   ebp, esp          00401134  E4 F8 81 EC
00401133 83 E4 F8             and   esp, 0FFFFFFF8h   00401138  A4 03 00 00
00401136 81 EC A4 03 00 00    sub   esp, 3A4h         0040113C  A1 00 30 40
0040113C A1 00 30 40 00       mov   eax, dword_403000 00401140  00
```

When Software Execution Breakpoints Fail

- If the `0xCC` byte is **changed** during code execution, the breakpoint won't occur
- If other code reads the memory containing the breakpoint, it will read `0xCC` instead of the original byte
- Code that **verifies integrity** will notice the discrepancy

Hardware Execution Breakpoints

- Uses **four** hardware **Debug Registers**
  - **DR0** through **DR3** – **addresses** of breakpoints
  - **DR7** stores **control** information
- The address to stop at is in a register
- Can break on access or execution
  - Can set to break on **read**, **write**, or **both**
- No change in code bytes
- Running code can **change** the **DR** registers, to interfere with debuggers
- General Detect flag in **DR7**
  - Causes a breakpoint prior to any mov instruction that would change the contents of a Debug Register
  - Does not detect other instructions, however

Conditional Breakpoints

- Breaks only if a **condition** is **true**
  - Ex: Set a breakpoint on the `GetProcAddress` function
  - Only if the parameter being passed in is **RegSetValue**
- Implemented as **software breakpoints**
  - The **debugger** always receives the break
  - If the condition is **not met**, it resumes execution *without alerting the user*
- Conditional breakpoints **take much longer** than ordinary instructions
- A conditional breakpoint on a frequently-accessed instruction can slow a program down
- Sometimes so much that it never finishes

EXCEPTIONS

Exceptions

- Used by **debuggers** to gain control of a running program
- **Breakpoints** generate exceptions
- Exceptions are also caused by
  - **Invalid memory access**
  - **Division by zero**
  - Other **conditions**
- First- and second-chance exceptions
  - When a exception occurs while a debugger is attached
    - The program **stops executing**
    - The debugger is given **first chance** at control
    - Debugger can either **handle the exception**, or **pass it** on to the program
    - If it's passed on, the program's exception handler takes it

Second Chance

- If the application **doesn't** handle the exception
- The debugger is given a **second chance** to handle it
    - This means the program would have **crashed** if the debugger were not attached
- In malware analysis, **first-chance** exceptions can usually be **ignored**
- **Second-chance** exceptions cannot be ignored
    - They usually mean that the malware doesn't like the environment in which it is running

Common Exceptions

- `INT 3` (Software breakpoint)
- **Single-stepping** in a debugger is implemented as an exception
  - If the **trap flag** in the flags register is set,
  - The processor executes one instruction and then generates an exception
- **Memory-access violation** exception
  - Code tries to access a location that it cannot access, either because the address is invalid or because of access-control protections
- **Violating Privilege Rules**
  - Attempt to execute **privileged instruction** with **outside privileged mode**
  - In other words, attempt to execute a **kernel mode** instruction in **user mode**
  - Or, attempt to execute **Ring 0** instruction from **Ring 3**

List of Exceptions

The following chart lists the exceptions that can be generated by the Intel 80286, 80386, 80486, and Pentium processors:

```
Exception | Description
(dec/hex) |
----------|------------------------------------------------
  0  00h  | Divide error:
          | Occurs during a DIV or an IDIV instruction when the
          | divisor is zero or a quotient overflow occurs.

  1  01h  | Single-step/debug exception:
          | Occurs for any of a number of conditions:
          | - Instruction address breakpoint fault
          | - Data address breakpoint trap
          | - General detect fault
          | - Single-step trap
          | - Task-switch breakpoint trap

  2  02h  | Nonmaskable interrupt:
          | Occurs because of a nonmaskable hardware interrupt.

  3  03h  | Breakpoint:
          | Occurs when the processor encounters an INT 3 instruction.
```

MODIFYING EXECUTION WITH A DEBUGGER

Skipping a Function

- You can change **control flags**, the instruction pointer, or the code itself

- You could avoid a function call by setting a breakpoint where at the **`call`**, and then changing the instruction pointer to the instruction after it
  - This may cause the program to <u>crash</u> or <u>malfunction</u>

Testing a Function

- You could run a function **directly**, without waiting for the main code to use it
  - You will have to set the parameters
  - This destroys a program's stack
  - The program won't run properly when the function completes

MODIFYING PROGRAM EXECUTION IN PRACTICE

Example (A Real Virus)

- Operation depends on language setting of a computer
  - Simplified Chinese
    - Uninstalls itself & does no harm
  - English
    - Display pop-up "Your luck's no good"
  - Japanese or Indonesian
    - Overwrite the hard drive with random data

Break at 1; Change Return Value

```
00411349    call    GetSystemDefaultLCID
0041134F   1mov     [ebp+var_4], eax
00411352    cmp     [ebp+var_4], 409h            409 = English
00411359    jnz     short loc_411360
0041135B    call    sub_411037
00411360    cmp     [ebp+var_4], 411h            411 = Japanese
00411367    jz      short loc_411372
00411369    cmp     [ebp+var_4], 421h            421 = Indonesian
00411370    jnz     short loc_411377
00411372    call    sub_41100F
00411377    cmp     [ebp+var_4], 0C04h           C04 = Chinese
0041137E    jnz     short loc_411385
00411380    call    sub_41100A
```

END OF LECTURE. THANK YOU.