Reverse Engineering Malware
Dynamic Analysis of Binary Malware I

F-Secure

Dynamic analysis

- Code is executed on physical machine or emulator
- Covers code that is actually executed
- Code execution is analyzed
- Analysis at various levels
  - CPU instructions
  - CPU exceptions (interrupts, page faults etc.)
  - CPU memory access
  - OS system calls
  - OS API's
  - OS high-level activity (filesystem, registry etc.)
  - Network activity

Dynamic vs. static analysis

- Problems with static analysis
  - Cost of reversing
  - Code obfuscation
  - Coverage
- Problems with dynamic analysis
  - Execution path depends on environment
  - Analysis logic visibility
  - Performance (emulators)
  - Scalability (hardware)

Dynamic analysis tools and techniques

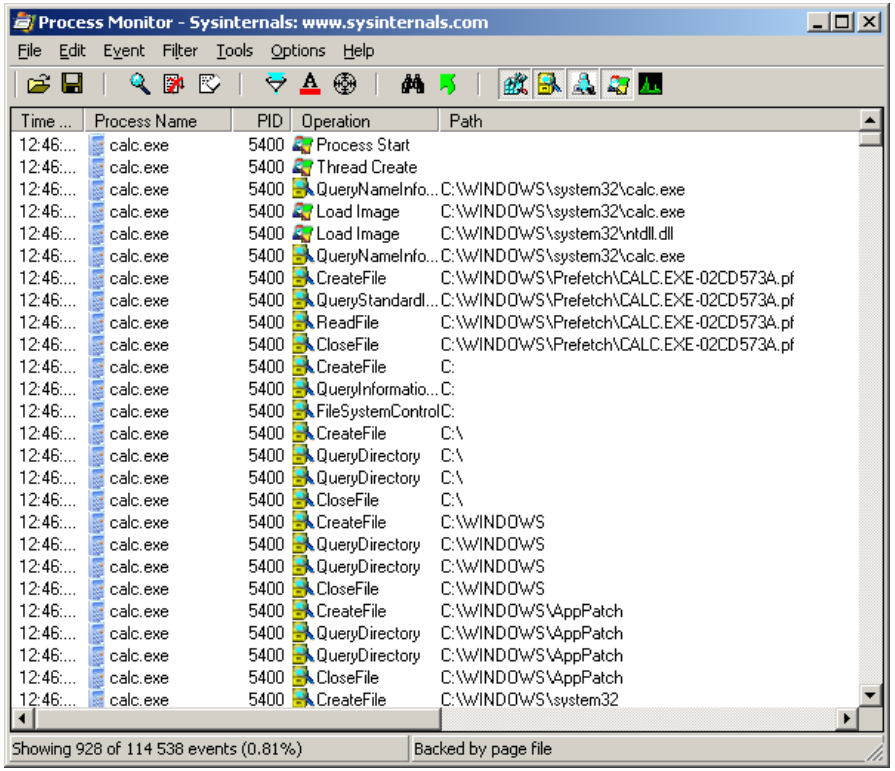
- Tracing and logging utilities (Dynamic Analysis I)
- Debuggers (DA I)
- Emulators (DA II)
- Instrumentation frameworks (DA II)
- Memory forensics (DA II)

Tracing and logging utilities

- Basic troubleshooting and system administration utilities can also be used in malware analysis
- Lots of interesting action can be logged: network, filesystem, registry
- Most of the utilities are non-intrusive

Utilities: Procmon

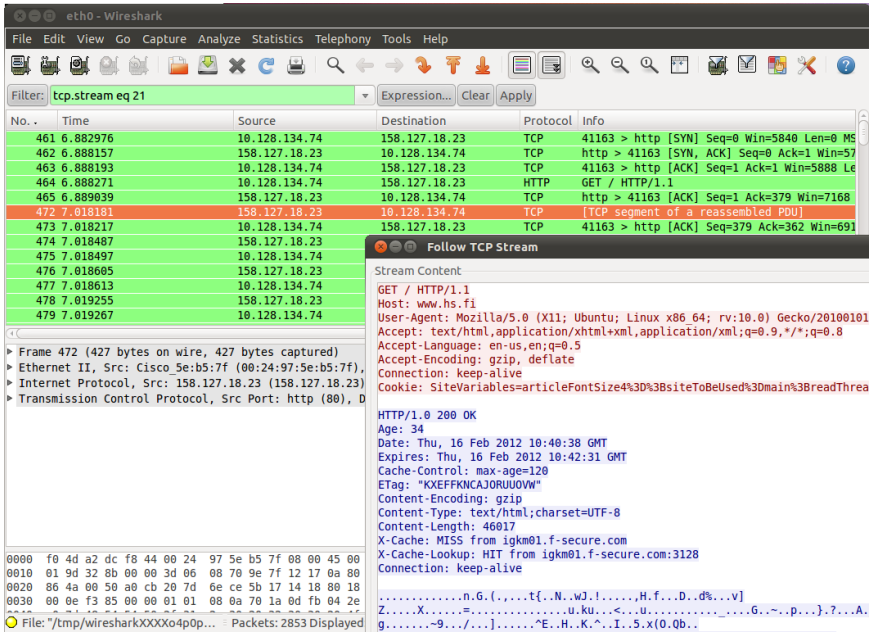
- Procmon (from Sysinternals) is a light-weight tool for dynamic analysis
- A flexible process monitor
  - File system
  - Registry
  - Process/thread activity
- Rich filtering possibilities



F-Secure

Utilities: Wireshark

- Free, open-source packet filter and analyzer
- Originally known as Ethereal
- Lots of supported protocol analyzers
- Expressive filtering
- Plugin support



F-Secure

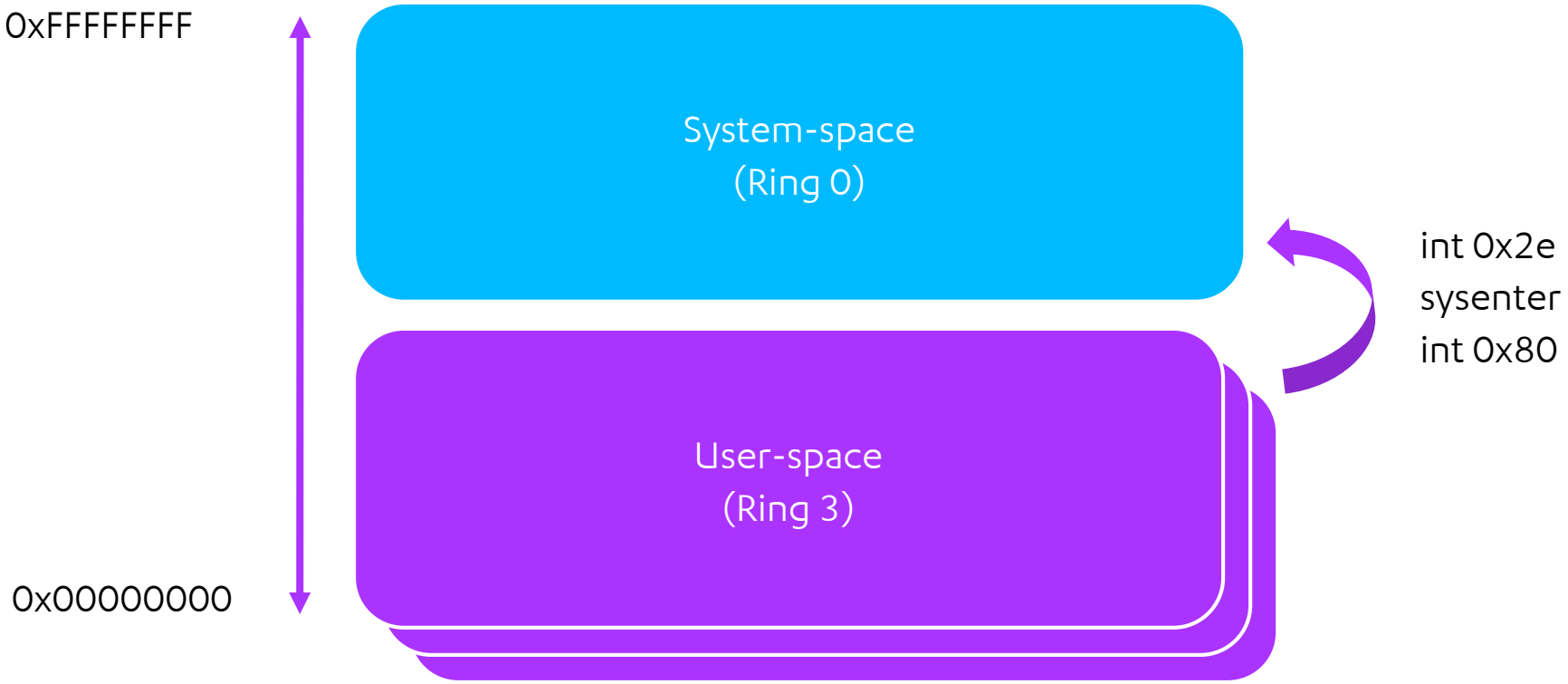
Debugging and debuggers

- Wikipedia:
  - Debugging is a methodological process of finding and reducing the number of bugs, or defects (…)
  - A debugger is a computer program that is used to test and debug other programs
- Our purpose is different
  - The debugger is just a tool to analyze the behavior of unknown applications

Common Debugger Features

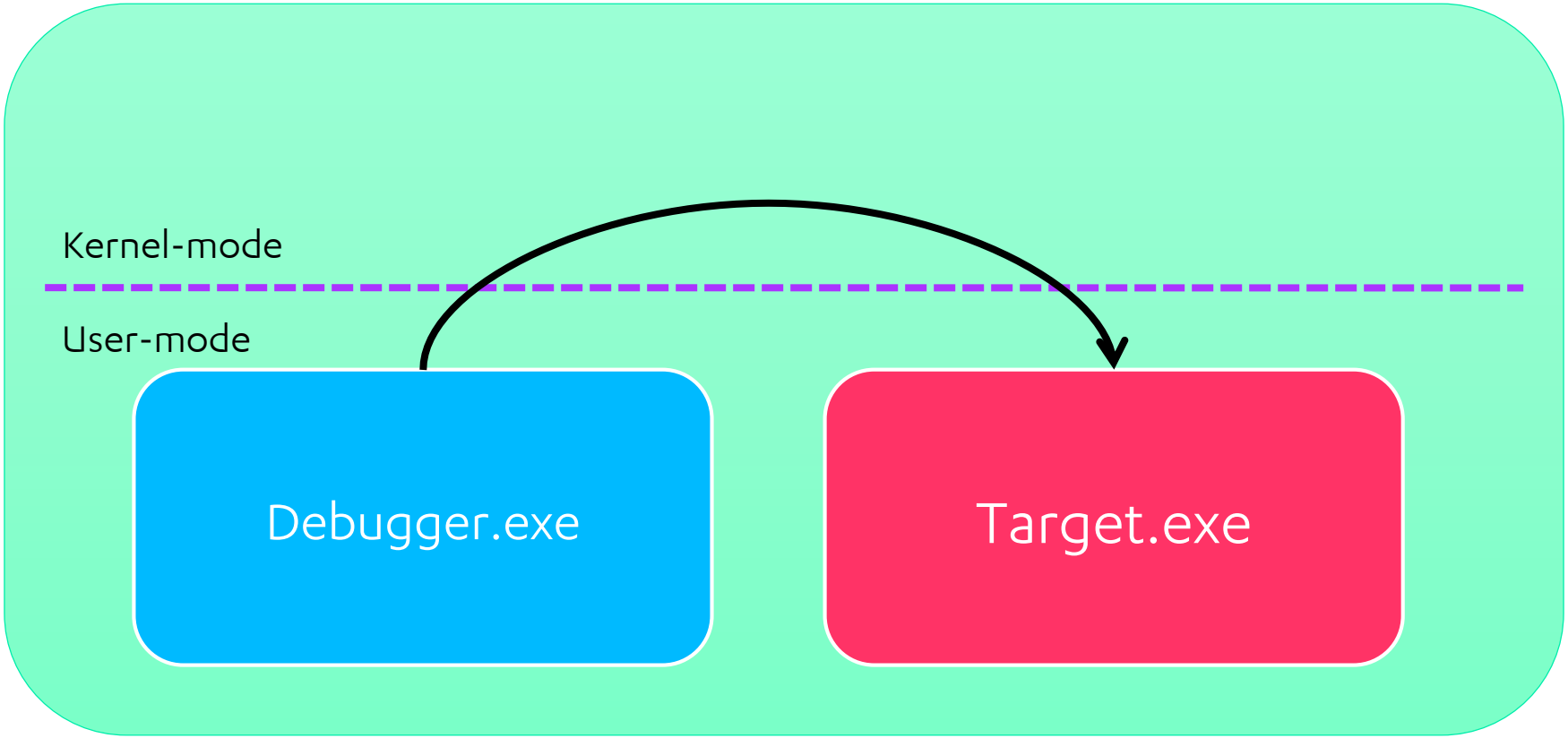
- Create a new process or attach to an existing process
- Control the target process execution
- Set breakpoints
- Read and write memory
- Read and write registers and CPU flags
- View the call stack
- View a disassembly of the code
- (View source code)

Kernel-mode & user-mode memory

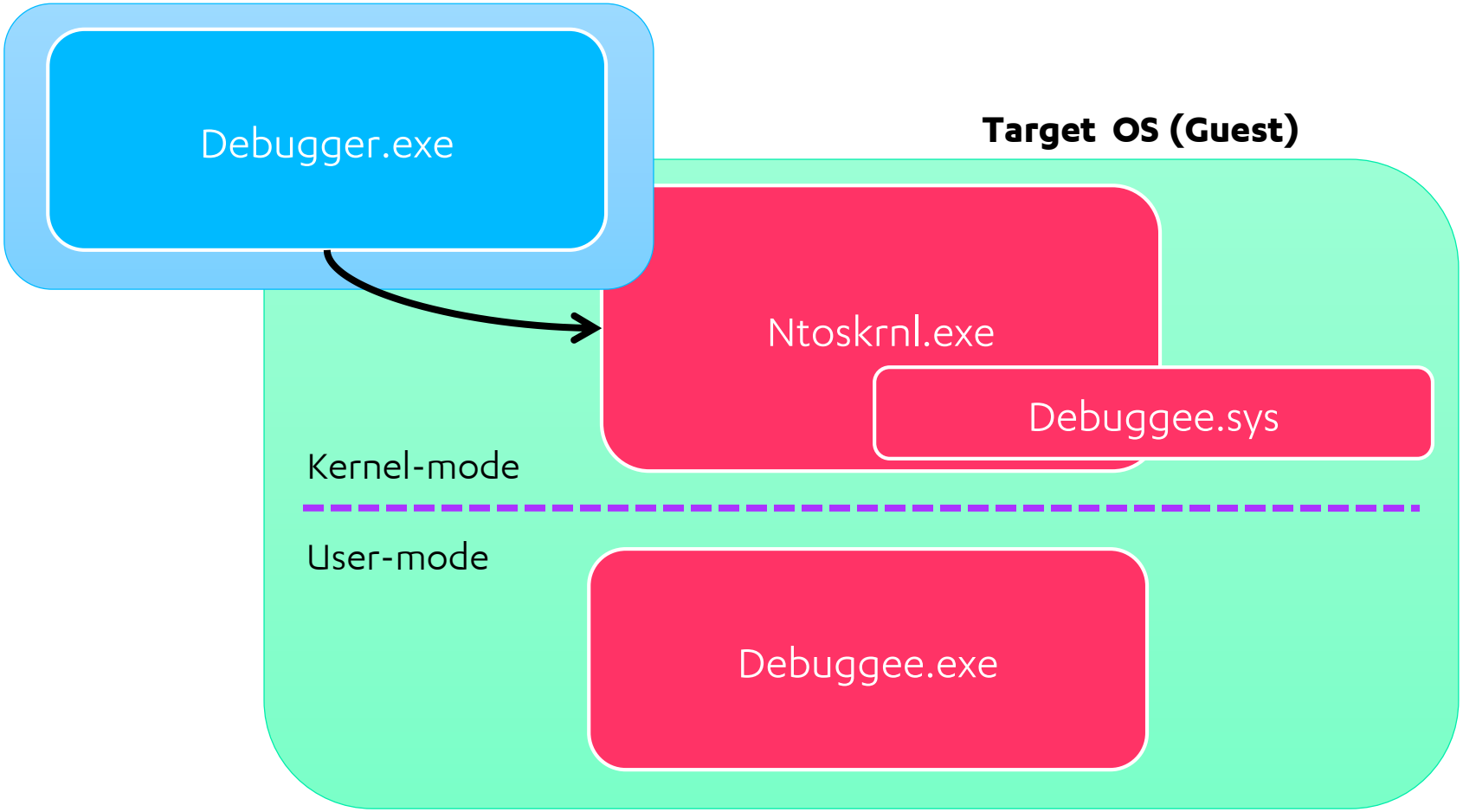


F-Secure

User-mode Debugging

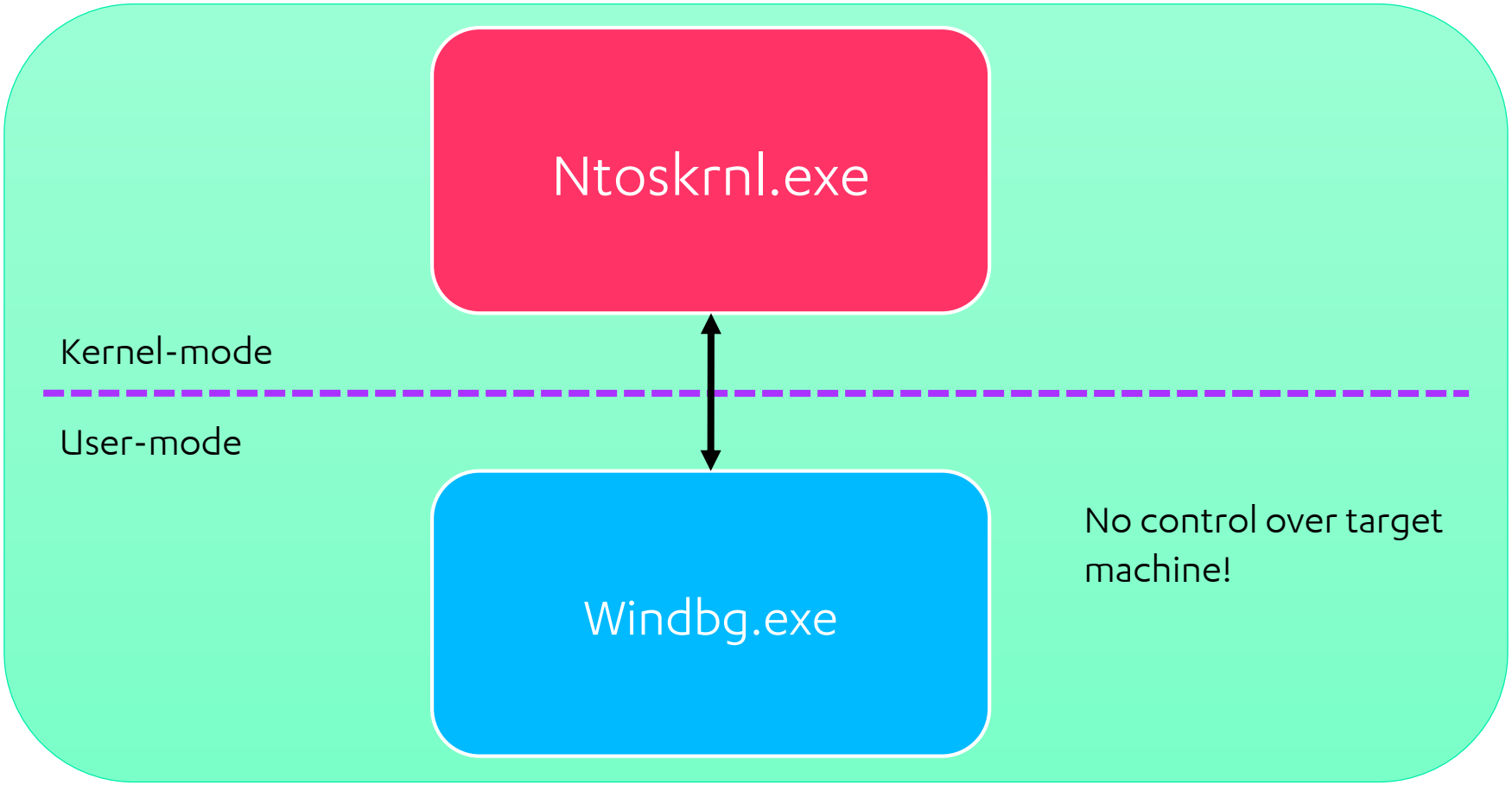


Remote Kernel-mode Debugging



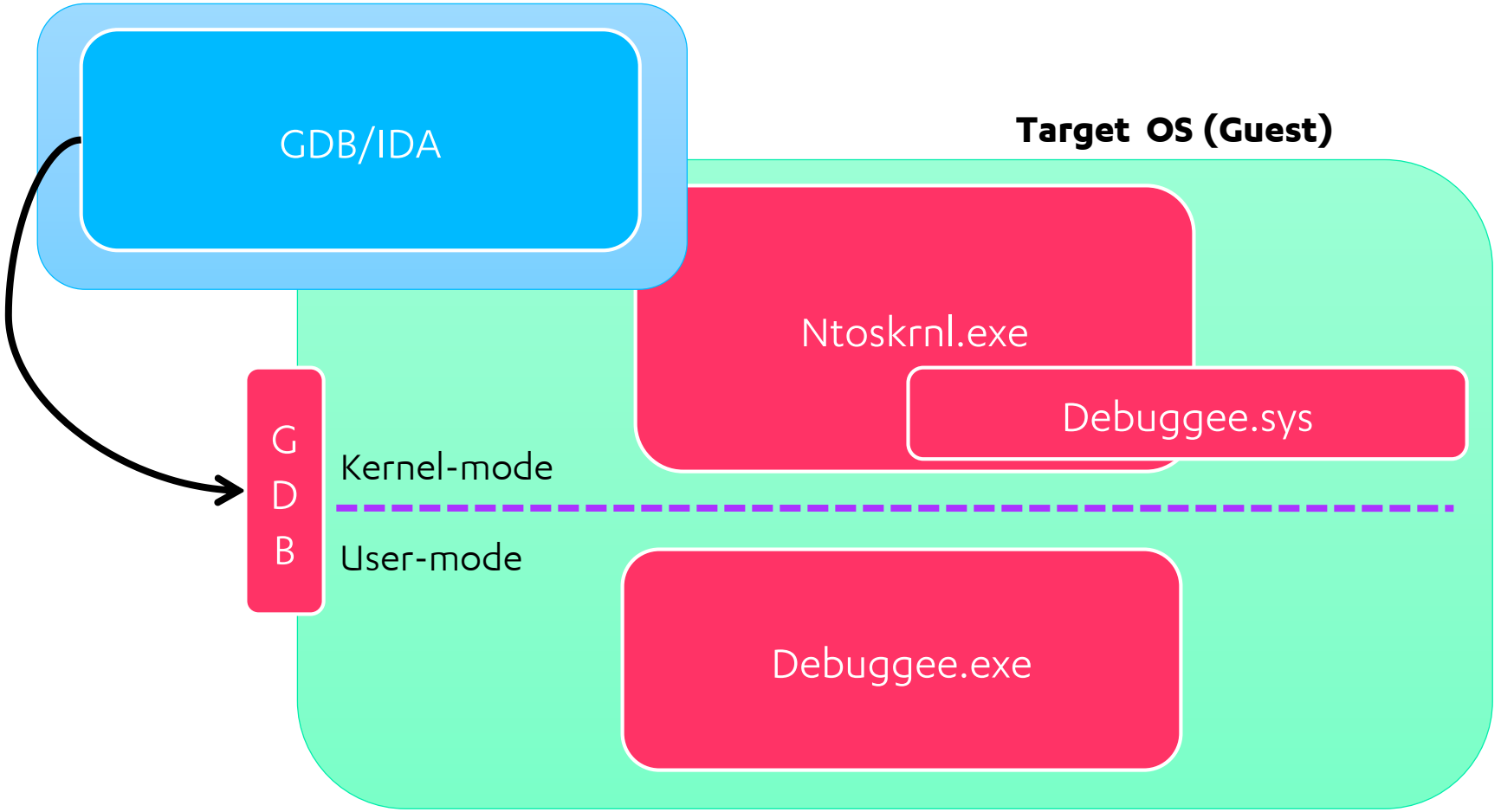
F-Secure

Local Kernel-mode Debugging



F-Secure

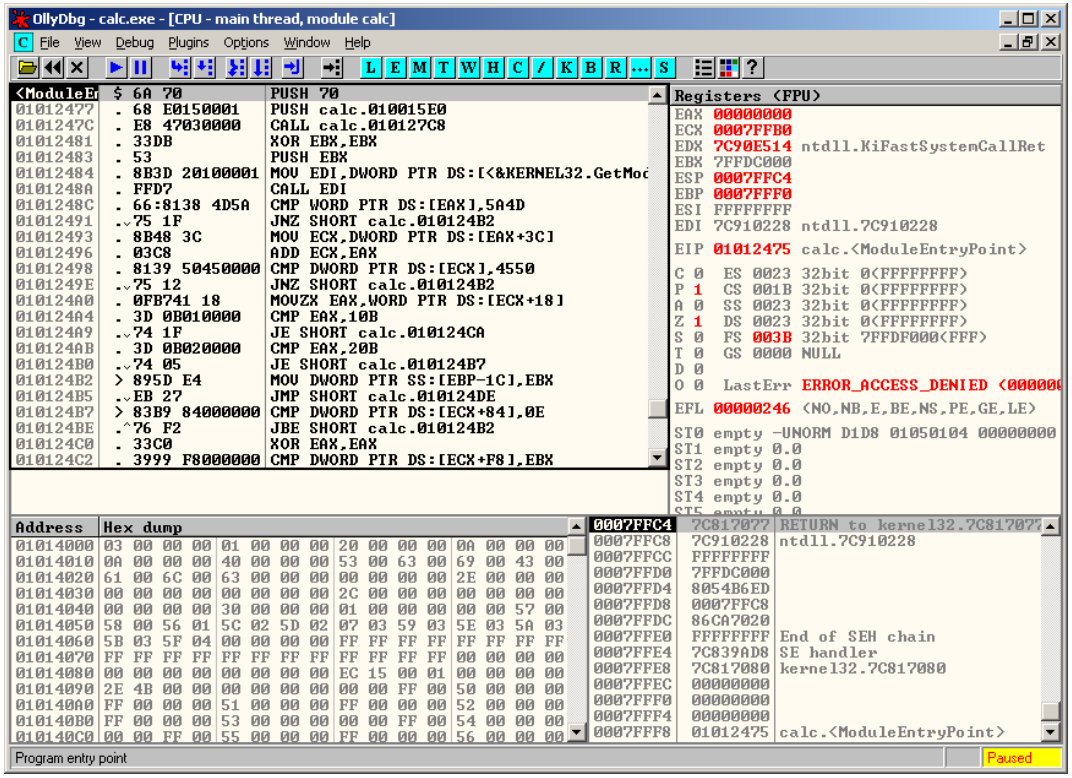
GDB "Hardware Debugging"



F-Secure

Debuggers: OllyDbg

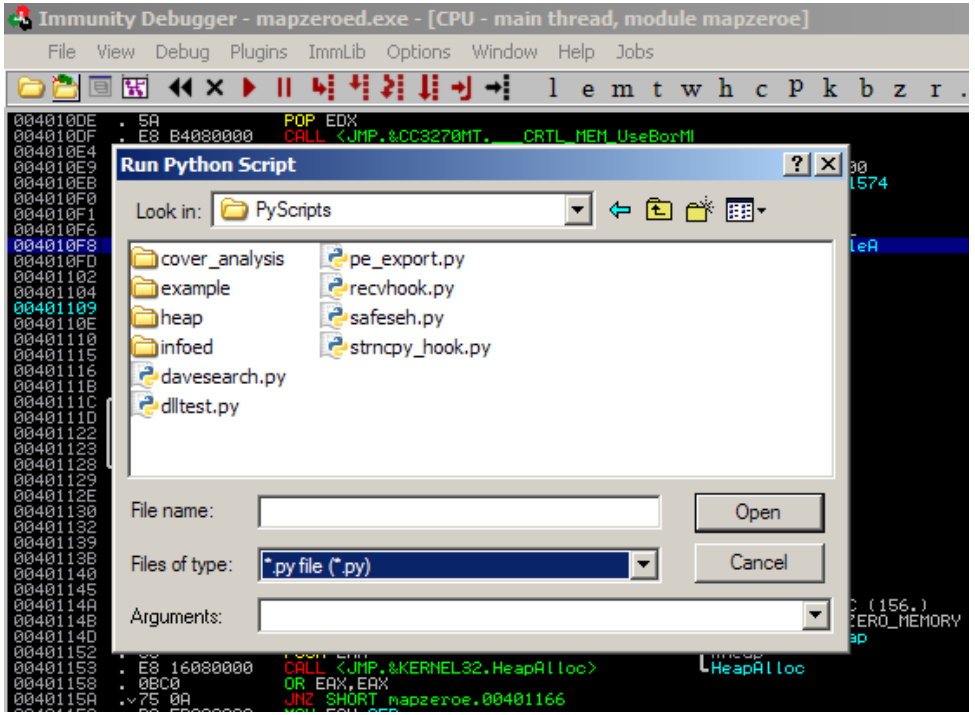
- Graphical debugger for 32-bit Windows (64-bit coming)
- Only for user-mode debugging
- Designed for working without source code
- Lots of useful plugins



F-Secure

Debuggers: Immunity Debugger

- Similar to OllyDbg, but adds several nice features such as Python scripting



Debuggers: GDB

- Free, open-source (GPL) source-level debugger
- Multiple targets (x86, AMD64, ARM, PPC etc.)
- Local and remote, user -and kernel-mode (Linux KGDB extension)
- Console program
- Graphical frontends: DDD, IDA
- Not really good for binaries

```
Temporary breakpoint 1 at 0x824c
Starting program: /home/turkja/research/src/hello

Temporary breakpoint 1, 0x0000824c in main ()
(gdb) info registers
r0             0x1      1
r1             0xbed66834       3201722420
r2             0xbed6683c       3201722428
r3             0x824c   33356
r4             0x898c   35212
r5             0x0      0
r6             0x8948   35144
r7             0xbed66805       3201722373
r8             0x0      0
r9             0x0      0
r10            0x0      0
r11            0x0      0
r12            0xbed66754       3201722196
sp             0xbed666d8       0xbed666d8
lr             0x842c   33836
pc             0x824c   0x824c <main>
fps            0x1001000        16781312
cpsr           0x60000010       1610612752
(gdb) x /5i $pc
0x824c <main>:  push    {r11, lr}
0x8250 <main+4>:        add     r11, sp, #4
0x8254 <main+8>:        ldr     r0, [pc, #28]   ; 0x8278 <main+44>
0x8258 <main+12>:       bl      0x9060 <puts>
0x825c <main+16>:       ldr     r0, [pc, #24]   ; 0x827c <main+48>
(gdb)
```

Debuggers: IDA remote debugger

- Small debugger server installed on a target machine
- IDA as a graphical frontend
- At the moment only good graphical Linux debugger
- Targets: Windows 32/64-bit, Linux x86 32/64-bit, OSX 32/64-bit, ARM Linux, Android
- Cross-functional: debug for example Windows binaries on Linux

Windows debug API

- Most Windows debuggers are based on the Debug API
  - Implemented by dbghelp.dll
- Interesting functions
  - DebugActiveProcess() to attach to an existing process
  - WaitForDebugEvent() to get events
  - DebugBreakProcess() to break into a running debuggee

Debug loop

```
while (TRUE) {
  WaitForDebugEvent(event, timeout);
  switch (event->dwDebugEventCode) {
        case EXCEPTION_DEBUG_EVENT:
                switch (event->u.Exception.ExceptionRecord.ExceptionCode)
  {
                        case EXCEPTION_ACCESS_VIOLATION:
                        case EXCEPTION_BREAKPOINT:
                        (...)
                }
        case LOAD_DLL_DEBUG_EVENT:
        (...)
}
```

Context

- The current state of a thread is described by a CONTEXT structure
- Passed to debug events and exception handlers
- Contains all registers and flags
- CPU-specific

```
lkd> dt nt!_CONTEXT
   +0x000 ContextFlags     : Uint4B
   +0x004 Dr0              : Uint4B
…
   +0x08c SegGs            : Uint4B
   +0x090 SegFs            : Uint4B
…
   +0x09c Edi              : Uint4B
   +0x0a0 Esi              : Uint4B
…
   +0x0b8 Eip              : Uint4B
   +0x0bc SegCs            : Uint4B
   +0x0c0 EFlags           : Uint4B
   +0x0c4 Esp              : Uint4B
```

x86 memory segmentation

- Segment registers (CS, DS, SS, ES, FS, GS) point to descriptor table
- Descriptor table entry referenced by a register defines a **segment descriptor**
- Segment descriptor translates *logical address* to a *linear address*
- The segment descriptor contains the following fields:
  - A segment base address
  - The segment limit which specifies the segment size
  - Access rights byte containing the protection mechanism information
  - Control bits
- Logical address examples: **DS:[0x00401121], FS:[0]**
- Linear address examples: **[0x00401121], [0x7FFE0000]**
- Segmentation not used anymore in modern operating systems (except for a special purposes, for example Windows exception handling)

TEB & PEB

- TEB = Thread Environment Block
  - Container for thread-specific things like the exception handler list, stack pointer, ...
  - Windows uses the fs segment to store it (offset 0x18 has pointer to self)
    - mov eax, fs:[0x18]
- PEB = Process Environment Block
  - Container for process-specific things like the list of loaded modules
  - TEB has a pointer to PEB at offset 0x30
- Important when reversing code that
  - Enumerates loaded modules (Peb.Ldr)
  - Checks for an attached debugger (PEB.BeingDebugged)
  - Installs an exception handler (TEB.NtTib.ExceptionList)

Example: Checking For a Debugger

```
; Call IsDebuggerPresent()
call [IsDebuggerPresent]
test eax, eax
; Do the same by checking PEB
mov eax, large fs:18h ; Offset 18h has self-pointer to TEB
mov eax, [eax+30h] ; Offset 30h has pointer to PEB
movzx eax, byte ptr [eax+2] ; PEB.BeingDebugged
test eax, eax
```

Example: Installing an Exception Handler

```
; Install a SEH exception handler
push offset_my_handler ; pointer to our handler
push fs:[0] ; pointer to old exception record
mov fs:[0], esp ; update TEB.NtTib.ExceptionList
```

Exceptions

- Exceptions are to software what interrupts are to CPU
- An event that occurs during execution of a program that requires execution of code outside the normal execution flow
- Windows exceptions match roughly to CPU exceptions, examples:
  - **EXCEPTION_INT_DIVIDE_BY_ZERO** Devide by zero (0)
  - **EXCEPTION_SINGLE_STEP** Debug (1)
  - **EXCEPTION_BREAKPOINT** Breakpoint (3)
  - **EXCEPTION_ACCESS_VIOLATION** Page fault (14)

Exception example

- What happens when this code executes in user-mode?

```
0042D9B0 xor eax,eax
0042D9B2 push eax
0042D9B3 call dword ptr [myfunc]
0042D9B6 mov ecx,80494678h
0042D9BB mov dword ptr [ecx],eax
0042D9BD push eax
0042D9BE call dword ptr [myfunc2]
```

Handling an Exception (Windows XP on x86)

1. CPU does address translation for 80494678h and sees the supervisor-bit set for this page of virtual memory. A page fault exception (#PF) is raised
   - See "IA-32 Intel Architecture Software Developer's Manual, Volume 3A" for details for exceptions and interrupts on x86
2. The page fault handler in the kernel, through the Interrupt Descriptor Table (IDT), gets control. It passes control to the exception dispatcher.
3. Since the exception happened in user-mode, the dispatcher looks for a user-mode debugger listening to a debug port.
4. The user-mode debugger gets a "first-chance" exception notification.
5. If the user-mode debugger does not handle the exception, the context is adjusted so that the user-mode exception dispatcher will run next.

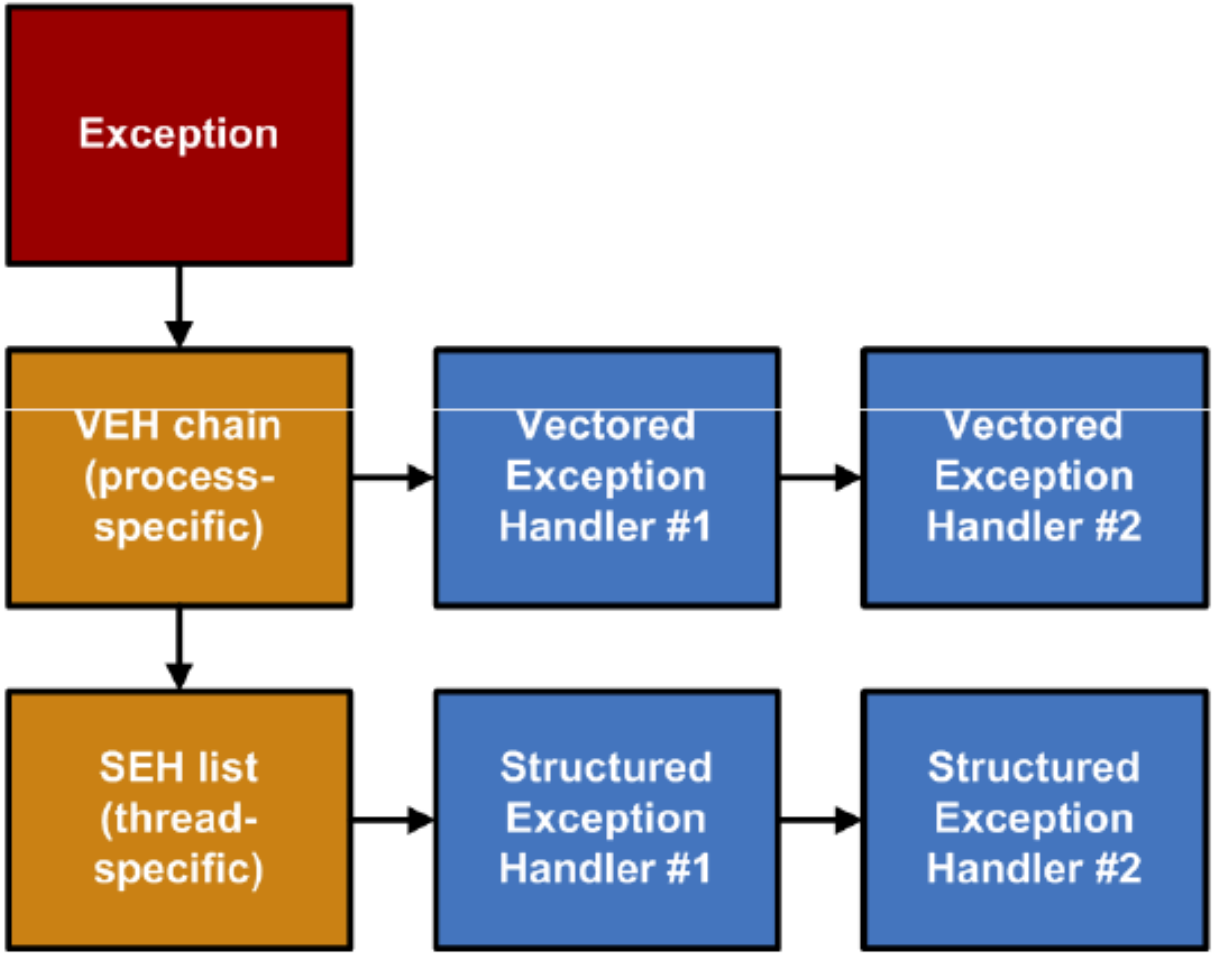
Handling an Exception (Continued)

6. The user-mode dispatcher looks for any installed vectored exception handlers (VEH) and calls them
7. If none of the handlers were prepared to handle the exception, a chain of structured exception handlers (SEH) is also called
8. If the exception is still not handled, it's re-raised and execution goes back to the kernel exception dispatcher
9. The user-mode debugger is sent a "second-chance" exception notification.

Handling an Exception in Application Code

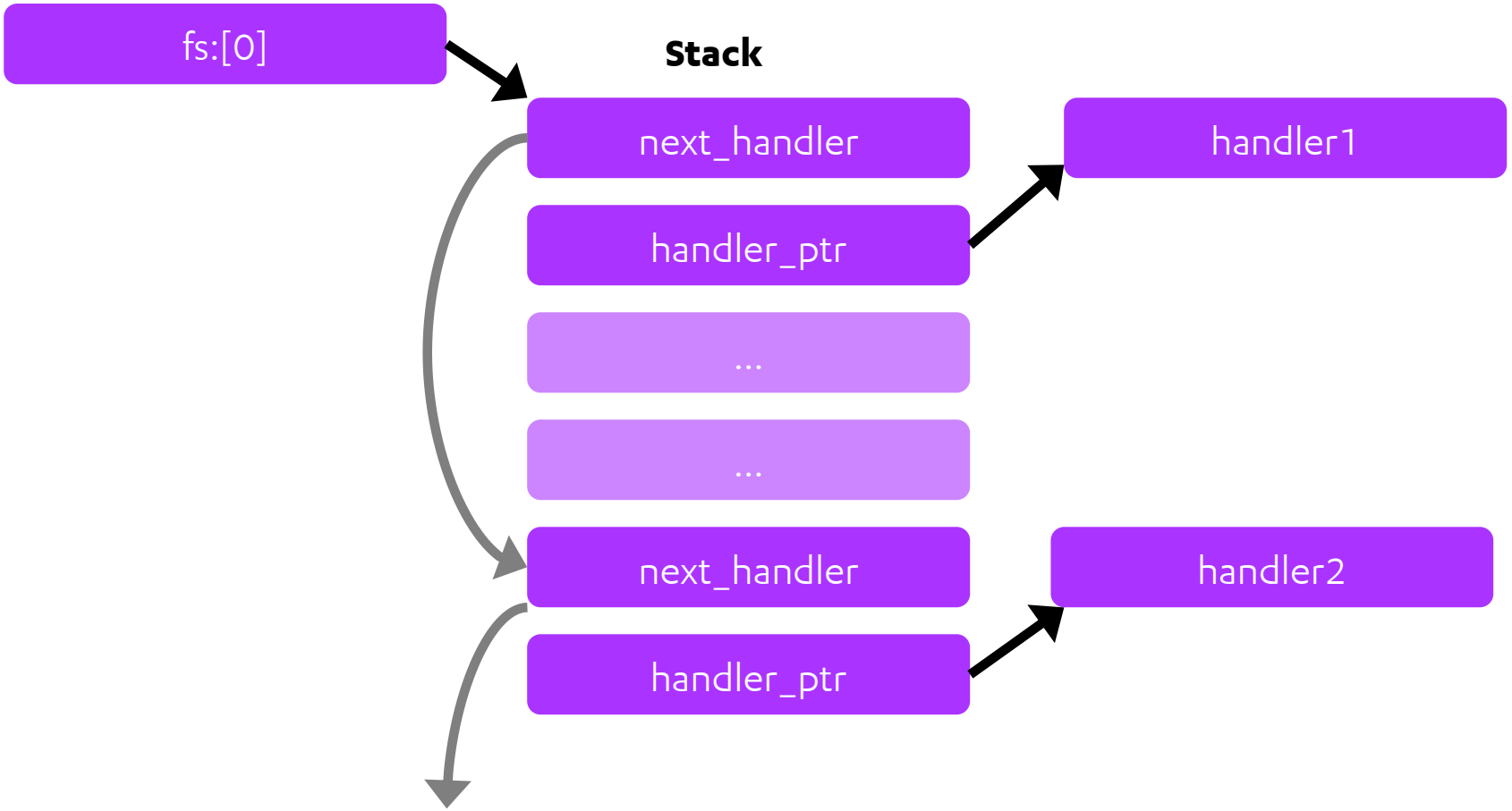
- Structured Exception Handling (SEH)
  - Operating system service for applications to provide mechanism for handling exceptions
  - In code: __try/__except/__finally
  - Exceptions are handled by the thread that caused the exception
  - Many handlers can be registered to a stack-based handler chain
- Vectored Exception Handling (VEH)
  - Expands SEH
  - Not frame-based
  - VEH exception handlers take precedence over SEH chain
  - See AddVectoredExceptionHandler() in MSDN

VEH and SEH



F-Secure

SEH Chain



F-Secure

Debugger Features: Single Stepping

- Single stepping means executing the application one instruction at a time
  - A very typical debugger feature
- Usually implemented using EFLAGS.TF (Trace Flag)
- When TF=1, the processor generates a debug exception for each executed instruction

Debugger Features: SW Breakpoints

- Used to break the execution of the target process at a specific address
- Typically implemented using INT 3
  - Debugger writes a byte with value 0xCC (opcode for INT 3) to the memory address
  - Note: usually the debugger makes this transparent to the user, so the modification is not visible in memory view
- Good:
  - No limitation to the amount of software breakpoints
- Bad:
  - Modifies the actual code bytes
  - Cannot break on reads or write addresses, just execution

Debugger Features: HW Breakpoints

- The CPU debug registers provide support for up to 4 hardware breakpoints
- DR0-3 store the linear addresses to be monitored
- DR7 configures the type of event
  - Break on execution, break on read, break on read/write
  - Length of data item to be monitored (1, 2 or 4 bytes)
- Good:
  - Does not modify code bytes
- Bad:
  - Limited number of breakpoints
  - Limited length of monitored data item (often you would like to break on a range of bytes)
  - On Windows, target can read and change the debug register contents through exception handlers

Debugger Features: Reading and Writing Memory

- Debugger must be able read and write the virtual memory space of the debuggee
- Done through normal Windows API functions
  - ReadProcessMemory()
  - WriteProcessMemory()

Debugger Features: Initial Breakpoint

- Initial breakpoint = first time the debugger gets control of the target
- OllyDbg has three options for the initial breakpoint
  - System breakpoint
    - Loader breaks into debugger **before any application code is run**
  - Entrypoint of main module
    - First break is at the entrypoint as defined by the main module PE header
  - WinMain (if known)
    - Attempts to skip compiler-generated stub and break at high-level main
- With anything else than system breakpoint, **application code can run before you get control**!
  - See PE/COFF specification and TLS callbacks
  - Support for TLS callbacks in Ollydbg 2.0

Why Debug Malware?

- Faster to execute and step through code than just read it
  - Especially for beginners it's more convenient to "see what the code does"
- Dealing with runtime packers
- A good, free debugger is sometimes all you need
  - They all have a disassembler
  - Ollydbg has pretty good code analysis features
- Also a matter of preference
  - Sometimes a combination of static and dynamic analysis is good
    - Browse through the application in a good interactive disassembler
    - When you've spotted the interesting parts, you can see how they are called and what they do in a debugger
    - Tip: use plugin and MAP files to transfer names from IDA to OllyDBG

Note on Debugging and Security

- We are now moving from **reading unknown code into executing it!**
  - Even if you are very careful, eventually your debuggee will escape
- If you ever debug potentially malicious applications, you need a safe environment
  - A machine you don't care about (a virtual machine running on anything important is not good enough…)
  - No Internet connectivity (or very limited)
  - Be extra careful with any portable media

Debugging Applications vs. Debugging Malware

- When debugging normal applications, you typically have symbols and/or source code
  - Obviously not the case for malware
- Normal applications don't actively prevent debugging
  - Malware plays a lot of tricks to avoid dynamic analysis
- Most common reason to debug a normal application: analyze a bug
  - Most common reason to debug malware: analyze functionality

Requirements for the tools are different!

Anti-Debugging

- Anti-debugging is used to prevent debugging an application or make it less convenient
  - Attempt to prevent a debugger from being attached
  - Attempt to detect an attached debugger and
    - Exit
    - Crash the application
    - Behave differently
    - …
  - Make debugging difficult by clearing breakpoints, causing ”noise” with exceptions, jumping to the middle of exported functions to avoid breakpoints, …

Anti-Debugging Techniques

- Documented API's to check if a debugger is active
  - IsDebuggerPresent()
  - CheckRemoteDebuggerPresent()
- Debugger-specific tricks
  - Checking for objects created by the debugger
    - Registry keys
    - Files
    - Devices
    - Windows
    - Remote process memory scanning

Anti-Debugging Techniques

- Checking data set in the process by the debugger
  - PEB!IsDebugged
  - PEB!NtGlobalFlags
- Scanning for software breakpoints (0xCC)
- Detecting through timing key points of execution
  - See *rdtsc instruction*
- Detecting virtual machines *)
  - Processes, file system, registry: VMWare tools service, registry settings
  - Memory: look for ”VMWare”, IDT location
  - Hardware: virtual hardware
  - CPU: non-standard opcodes, non-standard behaviour of existing opcodes
  - Lots and lots more…

*) http://handlers.sans.org/tliston/ThwartingVMDetection_Liston_Skoudis.pdf

Anti-Debugging Techniques

- Playing tricks with exceptions
  - Flooding with exceptions
  - Disabling hardware breakpoints through exception handlers
- Self-debugging
  - Create a child process that attempts to debug the parent
  - Split the execution into parent and child (debuggee), which communicate through exceptions
- Other miscellanenous:
  - NtQueryInformationProcess() with ProcessDebugPort
  - NtSetInformationThread() with ThreadHideFromDebugger

Anti-Debugging: Example 1

```
; Check from Process Environment Block (PEB)
; if a debugger is attached
mov eax, dword ptr fs:[18h]    ; self-pointer to TEB
mov eax, [eax+30h]             ; pointer to PEB
movzx eax, byte ptr [eax+2]    ; PEB.BeingDebugged
test eax, eax
```

Anti-Debugging: Example 2

```
push offset handler
push dword ptr fs:[0]
mov fs:[0],esp
xor eax, eax
div eax             ; exception
pop fs:[0]
add esp, 4
; continue execution
;...
```

```
handler:
mov ecx, [esp+0Ch]
add dword ptr [ecx+0B8h], 2 ; skip div
mov dword ptr [ecx+04h], 0  ; clean dr0
mov dword ptr [ecx+08h], 0  ; clean dr1
mov dword ptr [ecx+0Ch], 0  ; clean dr2
mov dword ptr [ecx+10h], 0  ; clean dr3
mov dword ptr [ecx+14h], 0  ; clean dr6
mov dword ptr [ecx+18h], 0  ; clean dr7
xor eax, eax
ret
```

Anti-Debugging: Example 3

```
.text:004042F7                push 0
.text:004042F9                call dword ptr [eax]          ; eax = msvcrt!_CIacos
.text:004042FB                mov edx, eax                  ; eax = 0x00321EA8
.text:004042FD                imul edx, 10000h              ; edx = 0x1EA80000
...
.text:004042D8                push 0E1A8A200h
.text:004042DD                pop esi
.text:004042DE                add esi, edx          ; debugger present: 0x0050A200 (r)
.text:004042E0                mov edi, esi          ; not present: 0x0040A200 (rw)
.text:004042E2
.text:004042E2      loc_4042E2:
.text:004042E2                lodsd
.text:004042E3                xor eax, 0C2EA41h
.text:004042E8                stosd                 ; access violation if debugger present
.text:004042E9                loop loc_4042E2
```

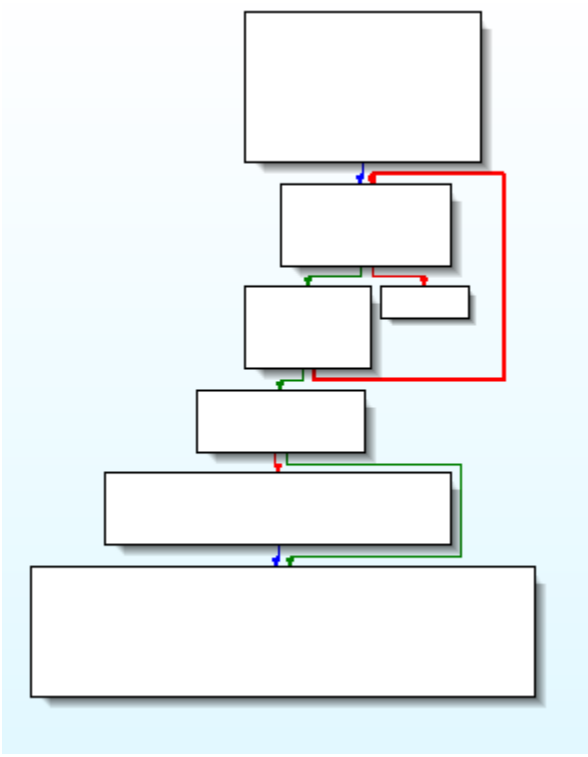
Source: https://www.openrce.org/blog/view/1043/SpyShredder_Malware_Spammed_on_OpenRCE (Rolf Rolles)

Example 3 Explained

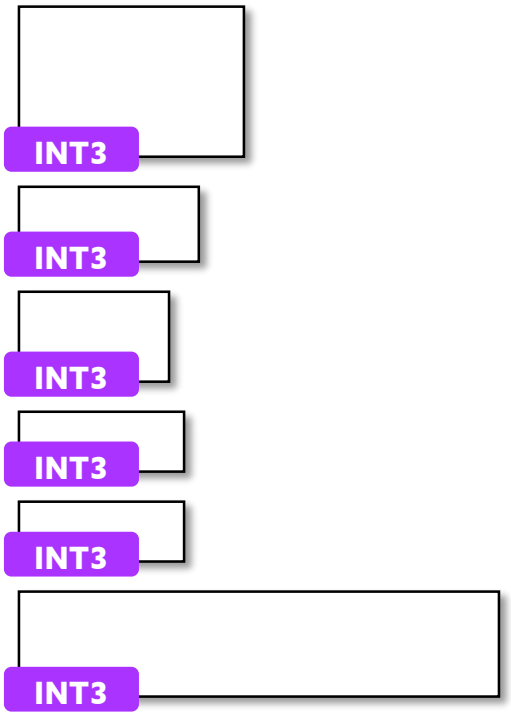
- msvcrt!_Ciacos calculates the arccos of the input
  - Return value in floating point register, not eax!
- After the call to _Ciacos,
  - Eax = 0x00321EA8 if a debugger is present
  - Eax = 0x00321E98 if a debugger is not present
- The value in eax is left there by the _Ciacos function as a side-effect
  - It comes indirectly from an earlier call to calloc()
  - The difference of 0x10 bytes in the pointers is caused by the debugger enabling debug heap settings!

Anti-Debugging: Example 4

Function in original application

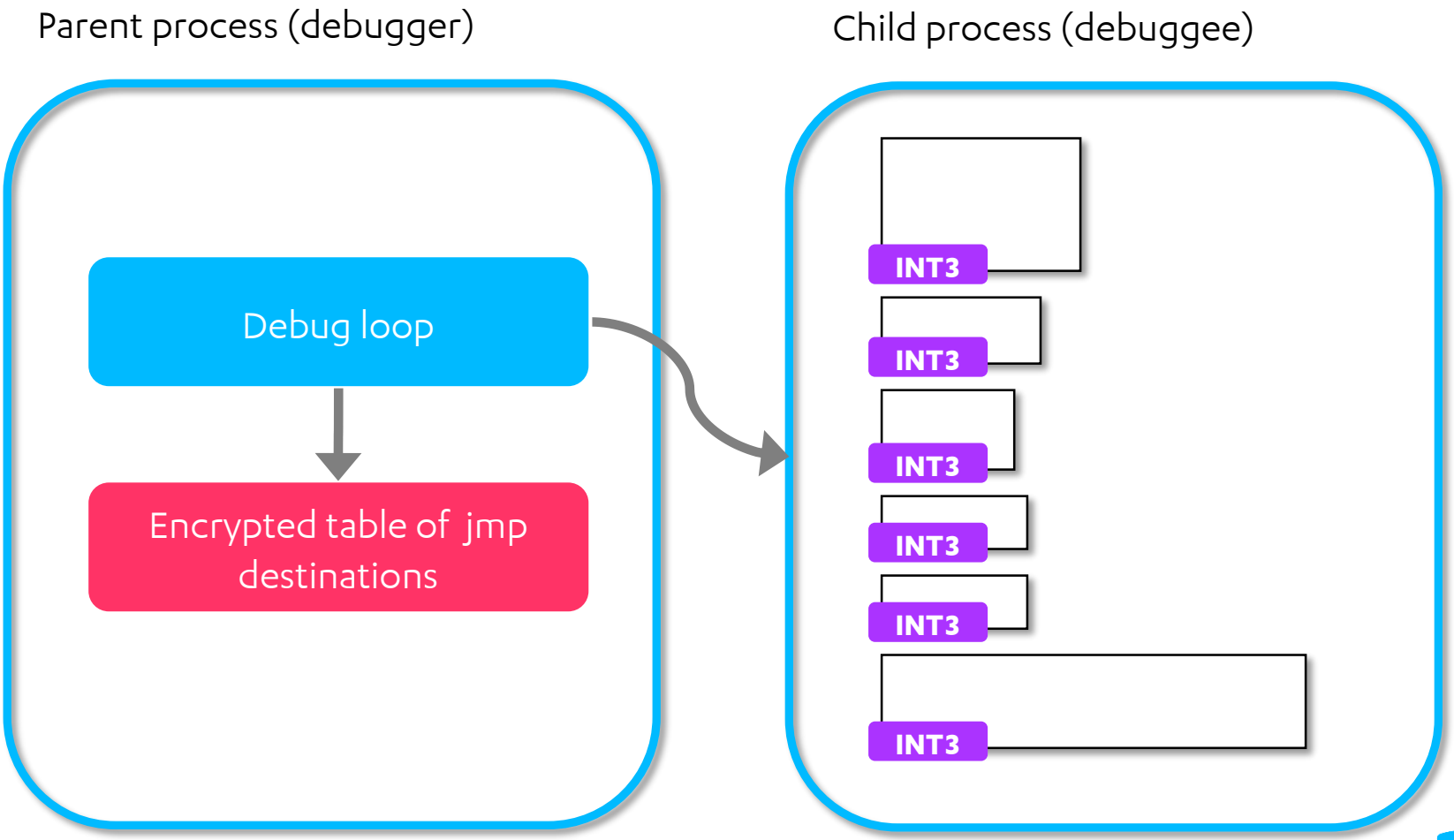


Control flow (jumps) replaced with interrupts



F-Secure

Anti-Debugging: Example 4 (continued)



F-Secure

Getting Around Anti-Debugging

- Make the debugger less visible to the target
  - Clear out bits from PEB
  - Disable setting of debug heap flags
  - OllyDbg has extensions to automate this
- Depending on the anti-debugging techniques used, change your methods
  - If hardware breakpoints are cleared, try software breakpoints
  - Attach to the process after it has unpacked itself, but before it exits
- Step through the most problematic parts of code and work around manually
  - Tedious and time-consuming

Resources

- Ollydbg
  - http://www.ollydbg.de
- Debugging Tools for Windows (Windbg)
  - www.microsoft.com/whdc/devtools/debugging/default.mspx
- Structured Exception Handling, Vectored Exception Handling
  - http://www.microsoft.com/msj/0197/exception/exception.aspx
  - http://msdn.microsoft.com/msdnmag/issues/01/09/hood/
- Windows Anti-Debug Reference (N. Falliere)
  - http://www.securityfocus.com/infocus/1893
- P. Szor, The Art of Computer Virus Research and Defense
  - Chapter 15.4.4 – Dynamic Analysis Techniques
  - Chapter 6.2.7 – Antidebugging

SWITCH ON FREEDOM