Reverse Engineering

6. Debugging and Anti-Debugging

Ing. Tomáš Zahradnický, EUR ING, Ph.D.
Ing. Martin Jirkal
Ing. Josef Kokeš, Ph.D.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Czech Technical University in Prague
Faculty of Information Technology
Department of Information Security

Version September 2, 2023

Table of Contents

Debugger

Debugger

A debugger or debugging tool is an application used to find bugs during an application's runtime. Application code can be examined at any time at the source level or at the assembly language level.

- Reverse engineers use debuggers to:
  - Watch the execution flow.
  - Get a better understanding of a specific part of the application.
  - Get answers to the engineer's questions.
    - What are the API's arguments?
    - Where exactly is the application crashing?
    - Is the algorithm working as expected?
  - Bypass obfuscation/encryption.
  - Verify theories about the application.

Debugger Types

- A **user-mode debugger** can debug applications in the user-mode.
- A **kernel debugger** can debug the kernel and the kernel drivers.
- A **source-level debugger** can set breakpoints in application according to its source code. Most commonly it is integrated with the development tool's IDE.
- A **low-level** or **assembly-level debugger** works with the assembly language.

Debugger Features

A debugger:

- Runs the application.
- Displays the current content of variables/registers.
- Stops the application on request. Allows single-stepping through the code.
- Provides symbol name resolution.
- Performs reverse debugging (go back in the execution flow).
- Reads/writes program memory.
- Facilitates remote debugging (over a serial line or ethernet).

Debuggers

- Commercial
  - Hex-Rays Interactive Disassembler Pro (IDA Pro)
  - Hopper
  - Visual DuxDebugger
  - † Syser, † SoftICE
- Free (user-level)
  - WinDBG
  - OllyDBG
  - Immunity Debugger
  - x64dbg
  - Eclipse
  - GDB
  - LLDB
  - Microsoft Visual Studio Debugger
- Free (kernel-level)
  - WinDBG

OllyDBG

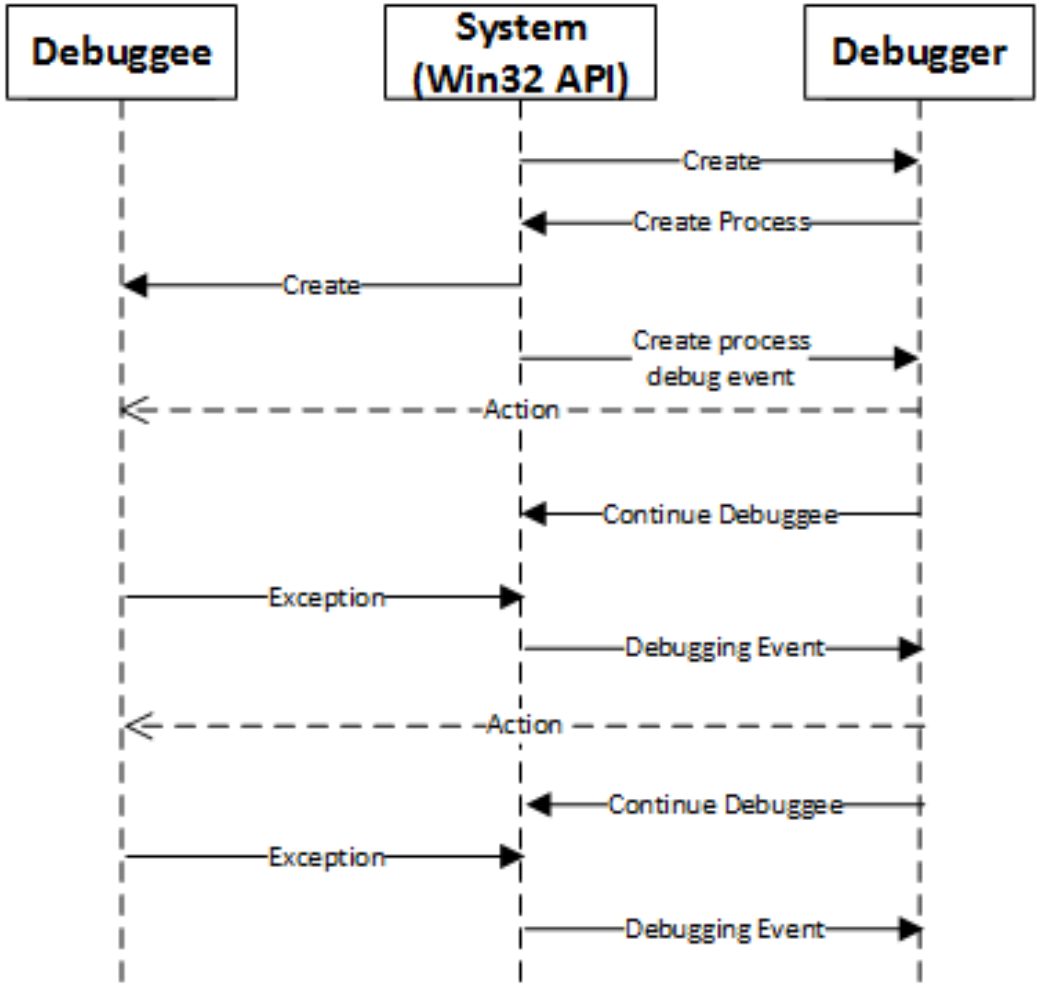
- Free.
- Commonly used in reverse engineering for dynamic analysis.
- Features:
  - Code analysis — traces registers, recognizes procedures, loops, API calls, switches, tables, constants, and strings.
  - Can load and debug DLLs directly.
  - Object file scanning — locates routines from object files and libraries.
  - Allows user-defined labels, comments, and function descriptions.
  - Understands debugging information in the Borland format.
  - Saves patches between sessions, writes them back to the executable file and updates fixups.
  - Open for 3^{rd} party plugins, mainly for version 1.x. There is only a fraction of version 1.x plug-ins for version 2.x.

How Debugging Works

Debuggee

A process that is being debugged.

1. The debugger either creates a new debuggee or attaches to an already running one.
2. The debugger processes debugging events in a debugging loop.



Debugger Connection

- Non-invasive
  - The debugger does not attach itself to the application.
  - All threads of the application are suspended to read the program's state (registers, memory, etc.).
  - Only limited control over the target application.
- Invasive
  - An application can only have one debugger attached.
  - The attached debugger is receiving all debugger events for that application. These include:
    - `CREATE_PROCESS_DEBUG_EVENT`
    - `CREATE_THREAD_DEBUG_EVENT`
    - `EXCEPTION_DEBUG_EVENT`
    - `EXIT_PROCESS_DEBUG_EVENT`
    - `EXIT_THREAD_DEBUG_EVENT`
    - `LOAD_DLL_DEBUG_EVENT`
    - `OUTPUT_DEBUG_STRING_EVENT`
    - `UNLOAD_DLL_DEBUG_EVENT`
    - `RIP_EVENT`

Debugging Events I

CREATE_PROCESS_DEBUG_EVENT

Generated whenever a new process is created in a process being debugged or whenever the debugger begins debugging an already active process. The system generates this debugging event before the process begins to execute in user mode and before the system generates any other debugging events for the new process.

EXIT_PROCESS_DEBUG_EVENT

Generated whenever the last thread in a process being debugged exits. This debugging event occurs immediately after the system unloads the process's DLLs and updates the process's exit code.

Debugging Events II

CREATE_THREAD_DEBUG_EVENT

Generated whenever a new thread is created in a process being debugged or whenever the debugger begins debugging an already active process. This debugging event is generated before the new thread begins to execute in user mode.

EXIT_THREAD_DEBUG_EVENT

Generated whenever a thread that is part of a process being debugged exits. The system generates this debugging event immediately after it updates the thread's exit code.

Debugging Events III

LOAD_DLL_DEBUG_EVENT

Generated whenever a process being debugged loads a DLL. This debugging event occurs when the system loader resolves links to a DLL or when the debugged process uses the `LoadLibrary` function. This debugging event only occurs when the system attaches the DLL to the virtual address space of a process.

UNLOAD_DLL_DEBUG_EVENT

Generated whenever a process being debugged unloads a DLL by using the `FreeLibrary` function. This debugging event only occurs when the DLL is actually unloaded from a process's address space (that is, when the DLL's usage count is zero).

Debugging Events IV

EXCEPTION_DEBUG_EVENT

Generated whenever an exception occurs in the process being debugged. Possible exceptions include attempting to access inaccessible memory, executing breakpoint instructions, attempting to divide by zero, or any other exception available in the Structured Exception Handling.

OUTPUT_DEBUG_STRING_EVENT

Generated when a process being debugged uses the `OutputDebugString` function.

RIP_EVENT

Generated when a system error occurs during debugging. This is caused by a bug in the debugger, not a debuggee! The process being debugged may or may not survive, depending on the severity of the error.

Debugger Connection by Process Creation

Code to Run a Debuggee

```
DWORD dwError = ERROR_SUCCESS;
STARTUPINFO si;
PROCESS_INFORMATION pi;

ZeroMemory( &si, sizeof(si) );
si.cb = sizeof(si);
ZeroMemory( &pi, sizeof(pi) );

dwError = CreateProcess( DebuggeePath, NULL, NULL, NULL, FALSE,
                         DEBUG_ONLY_THIS_PROCESS, NULL, NULL, &si, &pi );
```

- The process is created suspended.
- The `CREATE_PROCESS_DEBUG_EVENT` event is sent to the debugger for processing.

Debugger Connection by Attaching

An API to Attach a Process

```
BOOL WINAPI DebugActiveProcess( DWORD dwProcessId )
```

- The debugger must have the appropriate permissions.
  - Each user can debug her own processes without additional privileges.
  - The `SeDebugPrivilege` privilege is needed only for **other** users' processes! This includes `Read/WriteProcessMemory` APIs.
- Debugs an active process as if it were created with the `CreateProcess` API and the `DEBUG_ONLY_THIS_PROCESS` flag.
- All debuggee threads will be suspended by the system.
- The debugger receives a `LOAD_DLL_DEBUG_EVENT` event for each module loaded into the debuggee.
- The debugger receives the `CREATE_PROCESS_DEBUG_EVENT` event from the first thread of the debuggee.
- The operating system resumes suspended threads.

Debugger Loop

```
typedef struct _DEBUG_EVENT {
  DWORD dwDebugEventCode;
  DWORD dwProcessId;
  DWORD dwThreadId;
  union {
    EXCEPTION_DEBUG_INFO Exception;
    CREATE_THREAD_DEBUG_INFO CreateThread;
    CREATE_PROCESS_DEBUG_INFO CreateProcessInfo;
    EXIT_THREAD_DEBUG_INFO ExitThread;
    EXIT_PROCESS_DEBUG_INFO ExitProcess;
    LOAD_DLL_DEBUG_INFO LoadDll;
    UNLOAD_DLL_DEBUG_INFO UnloadDll;
    OUTPUT_DEBUG_STRING_INFO DebugString;
    RIP_INFO RipInfo;
  } u;
} DEBUG_EVENT, *LPDEBUG_EVENT;

DEBUG_EVENT debug_event = {0,};

for(;;)
{
  if(!WaitForDebugEvent(&debug_event, INFINITE))
    return;

  // process debugging event

  ContinueDebugEvent(debug_event.dwProcessId, debug_event.dwThreadId, DBG_CONTINUE);
}
```

Communicating with the Debuggee I

Memory Functions

```
BOOL WINAPI WriteProcessMemory(
  _In_ HANDLE hProcess,
  _In_ LPVOID lpBaseAddress,
  _In_ LPCVOID lpBuffer,
  _In_ SIZE_T nSize,
  _Out_ SIZE_T *lpNumberOfBytesWritten
);

BOOL WINAPI ReadProcessMemory(
  _In_ HANDLE hProcess,
  _In_ LPCVOID lpBaseAddress,
  _Out_ LPVOID lpBuffer,
  _In_ SIZE_T nSize,
  _Out_ SIZE_T *lpNumberOfBytesRead
);
```

- Read/Write process memory.
- Debuggers use these functions to read/write memory and to put software breakpoints into the code.
- Debuggers work with the application's image loaded into memory, **they do not change the executable on disk**.

Communicating with the Debuggee II

Thread Functions

```
BOOL WINAPI GetThreadContext(
  _In_ HANDLE hThread,
  _Inout_ LPCONTEXT lpContext
);

BOOL WINAPI SetThreadContext(
  _In_ HANDLE hThread,
  _In_ const CONTEXT *lpContext
);
```

- The `CONTEXT` structure contains the current state of all registers.
- The `CONTEXT` structure is specific for the architecture of the executable being debugged (i.e. different for i386 and x86_64).
- Debuggers use these functions to change the `EIP` and to obtain register values.

Software Breakpoints

- A software interrupt (or exception) is an event caused by software, notifying the kernel that the normal instruction flow of the program must be changed (an abnormal condition occurred).
- A pointer to the (top-level) exception handling function is stored as the first variable of the Thread Information Block (TIB) and can be accessed through FS:[0].
- The exception chain triggers and standard handler search and invocation is performed.
- If there is no exception handler to handle the exception, the default Windows mechanism is used to handle the exception.
- Most debuggers use the `int3` instruction (opcode `0xCC`) for breakpoints. The same result can be achieved with the `int 3` instruction (opcode `0xCD03`): both will raise a Breakpoint exception.

Setting a Breakpoint

```
BOOL bSuccess;
BYTE cInstruction;
BYTE OriginalInstruction;
DWORD dwReadBytes;

bSuccess = ReadProcessMemory( hProcess, (void*)Breakpoint_Address,
                              &cInstruction, 1, &dwReadBytes );

OriginalInstruction = cInstruction;
cInstruction = 0xCC;

// Rewrite the actual instruction with 0xCC
bSuccess = WriteProcessMemory( hProcess, (void*)Breakpoint_Address,
                               &cInstruction, 1, &dwReadBytes );

FlushInstructionCache( hProcess, (void*)Breakpoint_Address, 1);
```

1. Read 1 byte from the `Breakpoint_Address` memory location and remember what we read.
2. Rewrite the first byte of that instruction with `0xCC`.
3. Flush the instruction cache.
4. Continue debugging.

Handling a One-Time Breakpoints

Handling a One-Time Breakpoint

```
CONTEXT lcContext;
DWORD dwWriteSize;

lcContext.ContextFlags = CONTEXT_ALL;
GetThreadContext( hThread, &lcContext );
lcContext.Eip --;
SetThreadContext( hThread, &lcContext );
WriteProcessMemory( hProcess, Breakpoint_Address,
                    &OriginalInstruction, 1, &dwWriteSize );
FlushInstructionCache( hProcess, StartAddress, 1 );
```

1. Get the thread `CONTEXT` structure.
2. Go back 1 byte code (to the breakpoint address) by decreasing the `EIP`.
3. Apply the new context.
4. Restore the original instruction.
5. Continue debugging.

Note that the breakpoint is removed!

Handling a Persistent Breakpoint

Stopping After One Instruction

```
CONTEXT lcContext;
DWORD dwWriteSize;

lcContext.ContextFlags = CONTEXT_ALL;

GetThreadContext( hThread, &lcContext );
lcContext.EFlags |= 0x100; // Set the Trap flag for EXCEPTION_SINGLE_STEP
SetThreadContext( m_cProcessInfo.hThread, &lcContext );
```

1. Get thread `CONTEXT` and save it.
2. Restore the original instruction.
3. Decrease `EIP` to return to that instruction.
4. Set the Trap flag in `EFL` for single-step exception.
5. Apply the `CONTEXT` structure with `SetThreadContext`.
6. Restore the original instruction.
7. Continue debugging.
8. After the first instruction, `STATUS_BREAKPOINT` occurs.
9. Restore the breakpoint.

Hardware Breakpoints

- The Intel x86 architecture contains 6 debug registers.
- `DR0`–`DR3` can each contain a linear address of a hardware breakpoint.
- `DR6` (Debug Status) tells the application which debug action occurred.
- `DR7` (Debug Control) contains flags:
  - Locally enabled hardware breakpoint.
  - Globally enabled hardware breakpoint.
  - Break on data execute.
  - Break on data write.
  - Break on data access (write or read).
  - Size of the memory being watched (1 B, 2 B, 4 B, or 8 B).

Hardware Breakpoints vs Software Breakpoints

- Software breakpoints
  - The default kind of breakpoint in most debuggers.
  - An unlimited number of breakpoints in a program.
  - Can detect memory execution.
  - Modify the memory $\rightarrow$ can be easily detected.
- Hardware breakpoints
  - Only a limited number of breakpoints is supported (depends on the processor, 4 for Intel x86).
  - Can detect memory access and execution.
  - Do not modify program memory $\rightarrow$ more difficult to detect.
  - Supported by most debuggers.
  - Both reading and writing of the debug registers is a privileged operation.

Tracing

- Most debuggers allow instruction tracing and function tracing. They use the Trap flag from the flags registers to facilitate it.
- This feature monitors the exact execution of one control flow.
- An **instruction trace** is a recording of executed instructions.
- A **function trace** is a recording of executed calls.
- Tracing slows down the thread execution significantly because the debugger must stop after each instruction.

Kernel Debugging Introduction

- "I am creating a kernel driver and my computer is crashing all the time."
  - "Of course. The kernel is not as friendly as the user space is. A mistake is often followed with a BSOD. Read the BSOD message, analyze dumps created by the BSOD, or simply debug the driver."
- "What if I stop the program to examine the machine's status?"
  - "You will stop the kernel $\rightarrow$ the computer will freeze."
- "How can I debug it then? Creating an application without debugging is quite hard!"
  - "You need a second computer to control the debugging!"
- "Which application should I use?"
  - "WinDBG is the best."

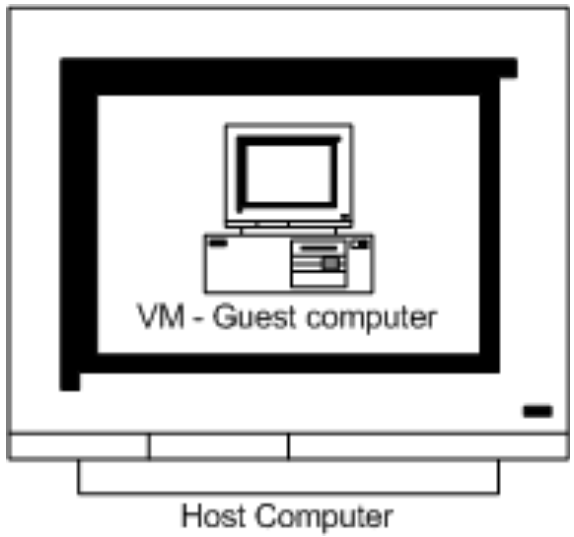
Kernel Debugging Setup – Local Debugging

- Start WinDBG, press `CTRL+K` and choose Local.
- You can debug the computer that is running your kernel.
- You cannot use all commands. Execution commands (go, step etc.), dumps, breakpoints, registers, and stack views are all inaccessible.
- You can read/write memory.
- Kernel is running $\rightarrow$ all information can change in time.
- You CAN crash the system.

Kernel Debugging Setup – Remote Computer

- Debug a computer using TCP/IP, IEEE 1394, COM, or USB.
- The most reliable and precise solution.
- Long crash recovery.

Kernel Debugging Setup – Virtual Computer



- Debug a computer in a virtual environment. Named pipes or virtual networks are used for "remote" debugging. This can depend on the virtualization software.
- Can be quite difficult to setup.
- The virtual computer behaves almost the same as a real computer. In 99.9% of the cases it will give the same results.
- Machine states can be stored and restored → much faster crash recovery.

Kernel Debugging – Application Start Breakpoint Trick I

- Get the file address of the entrypoint (The Raw Entrypoint).
  $\mathrm{EP}_{Raw} = \mathrm{EP}_{RVA} - \mathrm{Segment}_{RVA} + \mathrm{Segment}_{Raw}$
- Change the filebyte at EP_{Raw} to `0xCC` (breakpoint). Store the byte you are overwriting!
- Do not forget to update the CRC and the digital signature of the modified file, otherwise Windows will not accept it.
- Run the changed driver. The kernel should stop at your breakpoint.
- Change the byte by using the debugger to the old value.
  In WinDBG: `eb [ENTRYPOINT_RVA] [OLD_VALUE]`
- Another possibility is to use the `0xEB 0xFE` (`JMP -2`) instruction, which will create an infinite loop by jumping to itself.

Kernel Debugging – Application Start Breakpoint Trick II

- Find an undocumented function `IopLoadDriver()`.
- Find operand `call [edi + xx]`. On Windows XP SP3 it should be located at offset `0x66A` from the function's start.
- Create a breakpoint there and run the debugger. On the debugged machine, execute the driver. The execution will stop one call before entry point.

Set Breakpoint in WinDBG

```
bp nt!IopLoadDriver + 0x66a
```

Anti-Debugging

Anti-Debugging

Anti-Debugging is an application protection. It tries to detect an attached debugger, interfere with its execution, or escape from the debugger.

- The presence of the debugger can be detected by various methods, from calling a simple API to detecting behavior usual for a debugger.
- The debugger can be crashed by abusing specific debugger vulnerabilities.
- The application can try to escape from the debugger by moving to another thread or process.

Using Windows API to Detect a Debugger

- Basic debugger detection can be performed using Windows API functions.
  - **IsDebuggerPresent** — returns the `BeingDebugged` field of the Process Environment Block (PEB) structure.
  - **CheckRemoteDebuggerPresent** — the same as `IsDebuggerPresent` but can check a different process too.
  - **NtQueryInformationProcess** — a native API function from `ntdll.dll`. Can return multiple values. For example can return `ProcessDebugPort` that will be set if the process is being debugged.
  - **OutputDebugString** — then use `GetLastError` function to detect if a process is being debugged.
- Anti-Anti-Debugging techniques:
  - Modify the program flow.
  - Hook the API to return the correct values.

The BeingDebugged Flag

- IsDebuggerPresent checks the PEB for the BeingDebugged flag. Doing the same manually can be tricky to detect and defeat:

i386 ASM Code

```
// Read PEB lin. addr. from TIB
mov eax, dword ptr fs:[30h]
// Read PEB.BeingDebugged
movzx ebx, byte ptr [eax+2]
// Test for a non-zero
test ebx, ebx
jz NoDebuggerDetected
```

x86_64 ASM Code

```
// Read PEB lin. addr. from TIB
mov rax, qword ptr gs:[60h]
// Read PEB.BeingDebugged
movzx rbx, byte ptr [rax+2]
// Test for a non-zero
test rbx, rbx
jz NoDebuggerDetected
```

PEB structure in C

```
typedef struct _PEB {
  BYTE                          Reserved1[2];
  BYTE                          BeingDebugged;
  BYTE                          Reserved2[1];
  PVOID                         Reserved3[2];
  PPEB_LDR_DATA                 Ldr;
  PRTL_USER_PROCESS_PARAMETERS  ProcessParameters;
  BYTE                          Reserved4[104];
  PVOID                         Reserved5[52];
  PPS_POST_PROCESS_INIT_ROUTINE PostProcessInitRoutine;
  BYTE                          Reserved6[128];
  PVOID                         Reserved7[1];
  ULONG                         SessionId;
} PEB, *PPEB;
```

- An Anti-Anti-Debugging technique: Reset BeingDebugged.

The ProcessHeap Flag

- Microsoft does not reveal a lot of internal structures, but many structures are well known on the Internet.
- The `PEB.ProcessHeap` field contains a structure that can be used to detect a debugger.
- The offset `0x10` of the `ProcessHeap` header is called `ForceFlags` field flag and is set if the heap was created by a debugger.

ASM code for Windows XP

```
mov eax, dword ptr fs:[30h]
mov eax, byte ptr [eax+18h]
cmp dword ptr ds:[eax+10h], 0
jne DebuggerDetected
```

- Anti-Anti-Debugging techniques:
  - Change `ProcessHeap` flags or start the debugger with heap disabled (`windbg -hd`).

The NTGlobalFlag Flag

- Another well known but officially undocumented flag is the NTGlobalFlag. A process created by the debugger has the FLG_HEAP_ENABLE_TAIL_CHECK, FLG_HEAP_ENABLE_FREE_CHECK, and FLG_HEAP_VALIDATE_PARAMETERS bits set.
- A usual approach is to check the NTGlobalFlag at offset 0x68 against the value 0x70. This approach is not reliable, as more constants may be set.

ASM code

```
mov eax, dword ptr fs:[30h]
cmp dword ptr ds:[eax+68h], 70h
jz DebuggerDetected
```

- Anti-Anti-Debugging techniques:
  - Change the NTGlobalFlag.

Detecting a Debugger by Scanning Processes

- This debugger check can be bypassed by changing the debugger's file name.

WinDBG Detection Code

```
#include <windows.h>
#include <tlhelp32.h>

BOOL isDebugged(void)
{
  HANDLE hProcessSnapshot;
  HANDLE hProcess;
  PROCESSENTRY32 pe32;
  DWORD dwPriorityClass;

  hProcessSnapshot = CreateToolhelp32Snapshot( TH32CS_SNAPPROCESS, 0 );
  pe32.dwSize = sizeof( PROCESSENTRY32 );

  if( !Process32First( hProcessSnapshot, &pe32 ) )
    return( FALSE );

  do {
    if (wcsicmp(pe32.szExeFile, L"windbg.exe") == 0 )
      return ( TRUE );
  } while( Process32Next( hProcessSnapshot, &pe32 ) );

  CloseHandle( hProcessSnap );
  return( FALSE );
}
```

Detecting a Debugger Through Timing

- Check whether functions are executed in normal time (milliseconds) or whether their execution is being delayed.
- This method can expose if an application is being single-stepped or stopped between the measured instructions.
- The second method is to trigger an exception. Naturally it is processed quickly, but debugger a normally awaits user interaction.

rdtsc example

```
rdtsc
xor ecx,ecx
add ecx,eax
rdtsc
sub eax,ecx
cmp eax 0xFFF
jb NoDebugger
```

GetTickCount

```
ULONGLONG a = GetTickCount64();
ImportantFunction();
ULONGLONG b = GetTickCount64();
delta = b - a;
if ( delta > 30 ) {
  /*A debugger detected*/
} else {
  /*No debugger detected*/
}
```

- Anti-Anti-Debugging techniques:
  - Hook timing methods.

Other Debugger Detection Methods

- Application can scan itself for `int3` (0xCC) bytes to find software breakpoints.
- Advanced applications perform code checksums, comparing actual and expected values, to find if the code was modified.

Interfering with Debugging

- Exceptions require user input. Inserting hundreds of exceptions with an empty handler does not lower the performance much but it will annoy the analyst enough to turn the notifications off. Then it is a good time to hide an important code into one of these exception functions.
- The program can contain an `int3` instructions to confuse the debugger to think that it is its breakpoint. Without a debugger the `STATUS_BREAKPOINT` exception would be raised, but with a debugger there is no exception raised and next byte is executed. The debugged application will then follow a different execution flow.
  - Some debuggers will even crash on such an unknown debugger!
- Every debugger has a known set of exploits that can crash it. For example, OllyDBG v1.1 is known to crash after executing `OutputDebugString("%s%s%s%s%s%s%s%s%s%s")`.
- Attach yourself to your own process. Only 1 debugger can be attached at a time.

Debugger Evasion

- Inject important code into a different process.
- Store important code in an exception handler.
- Place important code before/after `main` (e.g. the `initterm` functions explained in Lecture 2).
- Place important code before `start`: Even before calling the entry-point, the operating system may execute so-called TLS[1] callbacks which take care of initialization of TLS variables. If the entire malicious code executes here, the analyst – and sometimes event the debugger – will not even notice it!

[1] thread-local storage

Other Anti-Debugging Methods

- Many more anti-debugging methods exists. A very good compilation of anti-debugging techniques can be found in `http://pferrie.host22.com/papers/antidebug.pdf`.
- Note that for basic and some advanced common anti-debugging tricks, there are multiple anti-anti-debugging plugins, such as IDA Stealth for IDA Pro or Phant0m for OllyDBG.

Bibliography


Ajay Vijayvargiya: *Writing a basic Windows debugger*, January 2015, http://www.codeproject.com/Articles/43682/Writing-a-basic-Windows-debugger.


Microsoft: *MSDN*, January 2015, https://msdn.microsoft.com/.


Peter Ferrie: *The "Ultimate" Anti-Debugging Reference*, April 2011, http://pferrie.host22.com/papers/antidebug.pdf.