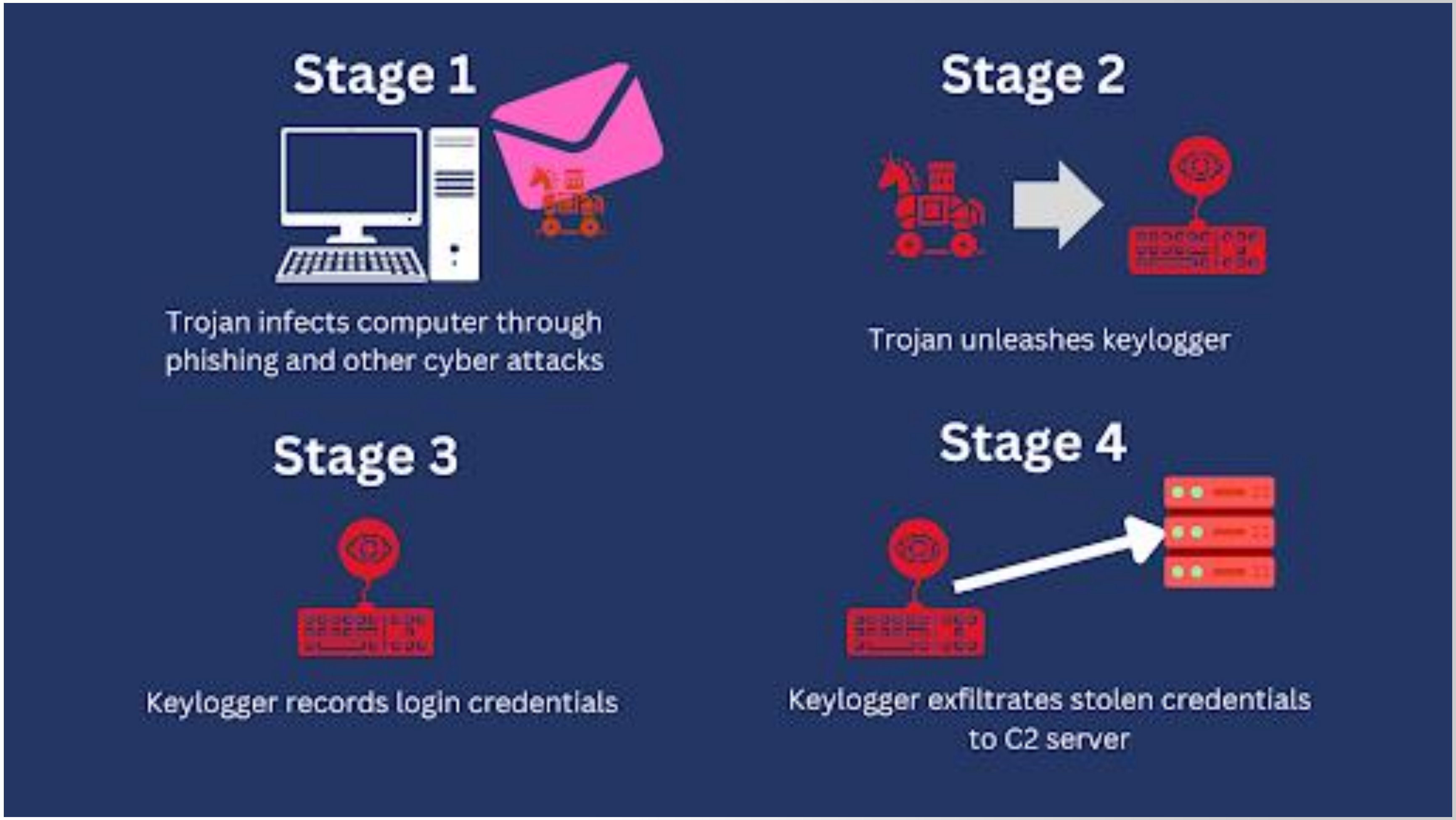
Class 10

CSC 471 Modern Malware Analysis
Windows API Hooks

Si Chen (schen@wcupa.edu)

A keylogger is a type of malicious software that **records every keystroke you make on your computer**. Keyloggers are a type of spyware — malware designed to spy on victims. Because they can capture everything you type, keyloggers are one of the most invasive forms of malware.

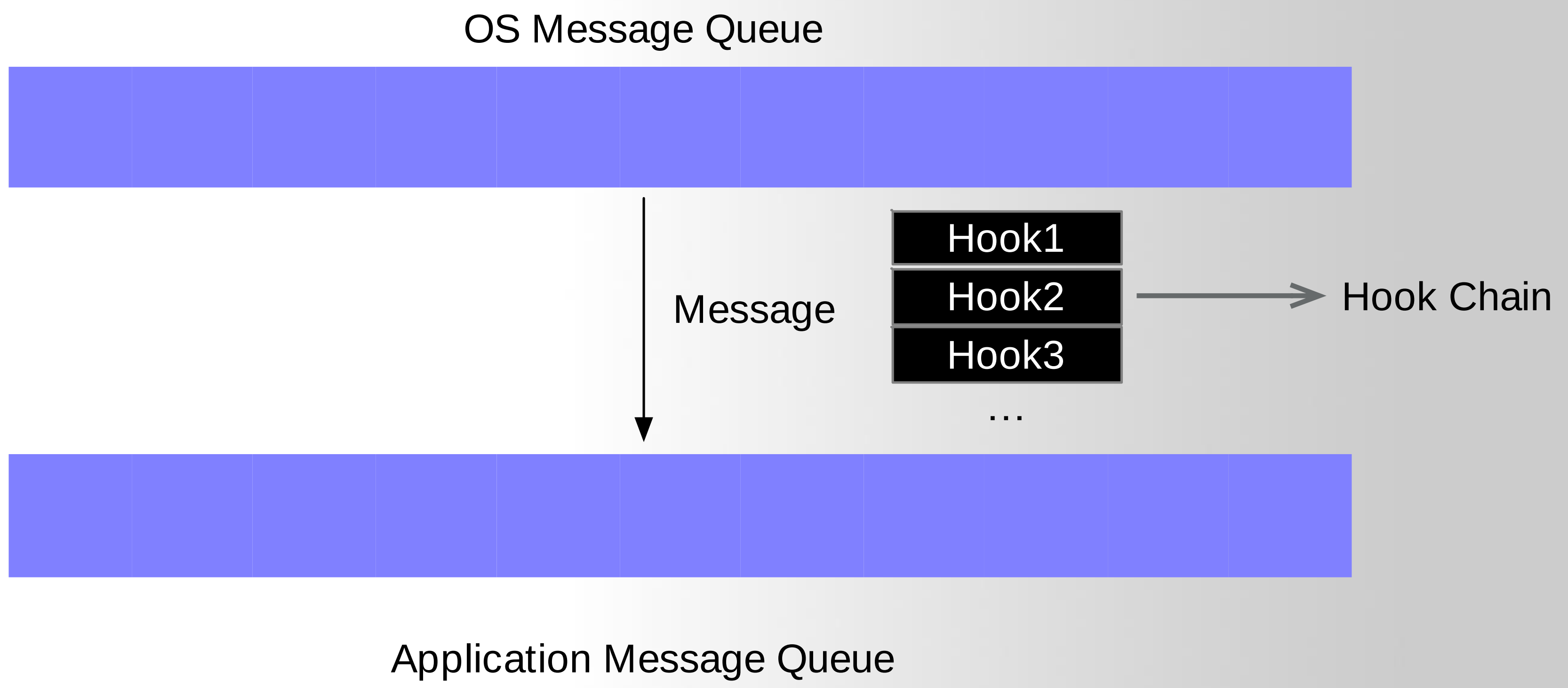

Stage 1
Trojan infects computer through phishing and other cyber attacks
Stage 2
Trojan unleashes keylogger
Stage 3
Keylogger records login credentials
Stage 4
Keylogger exfiltrates stolen credentials to C2 server

Message Hooks

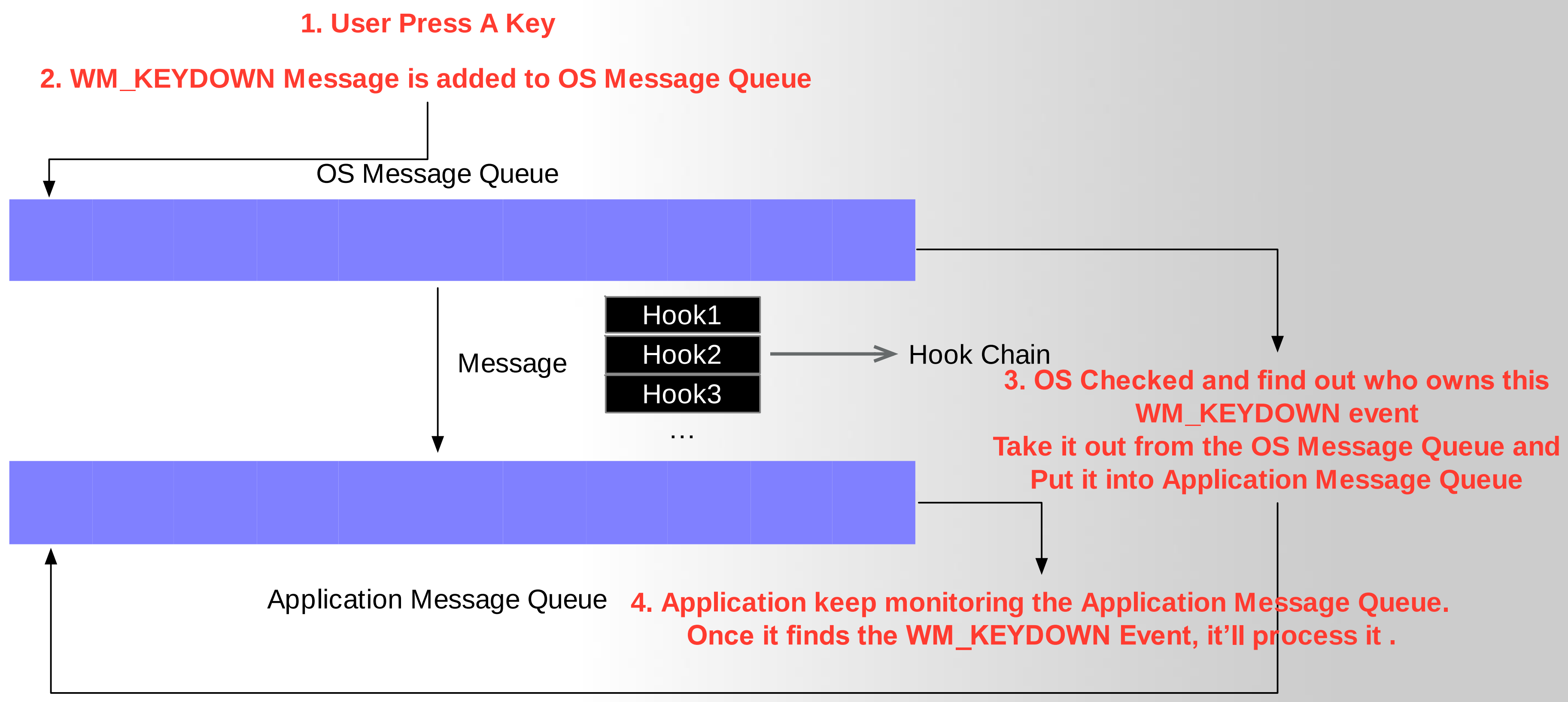
Message Hook Example

- Try HookMain.exe
- Download Hook.zip from our course website, unzip it (password: infected)

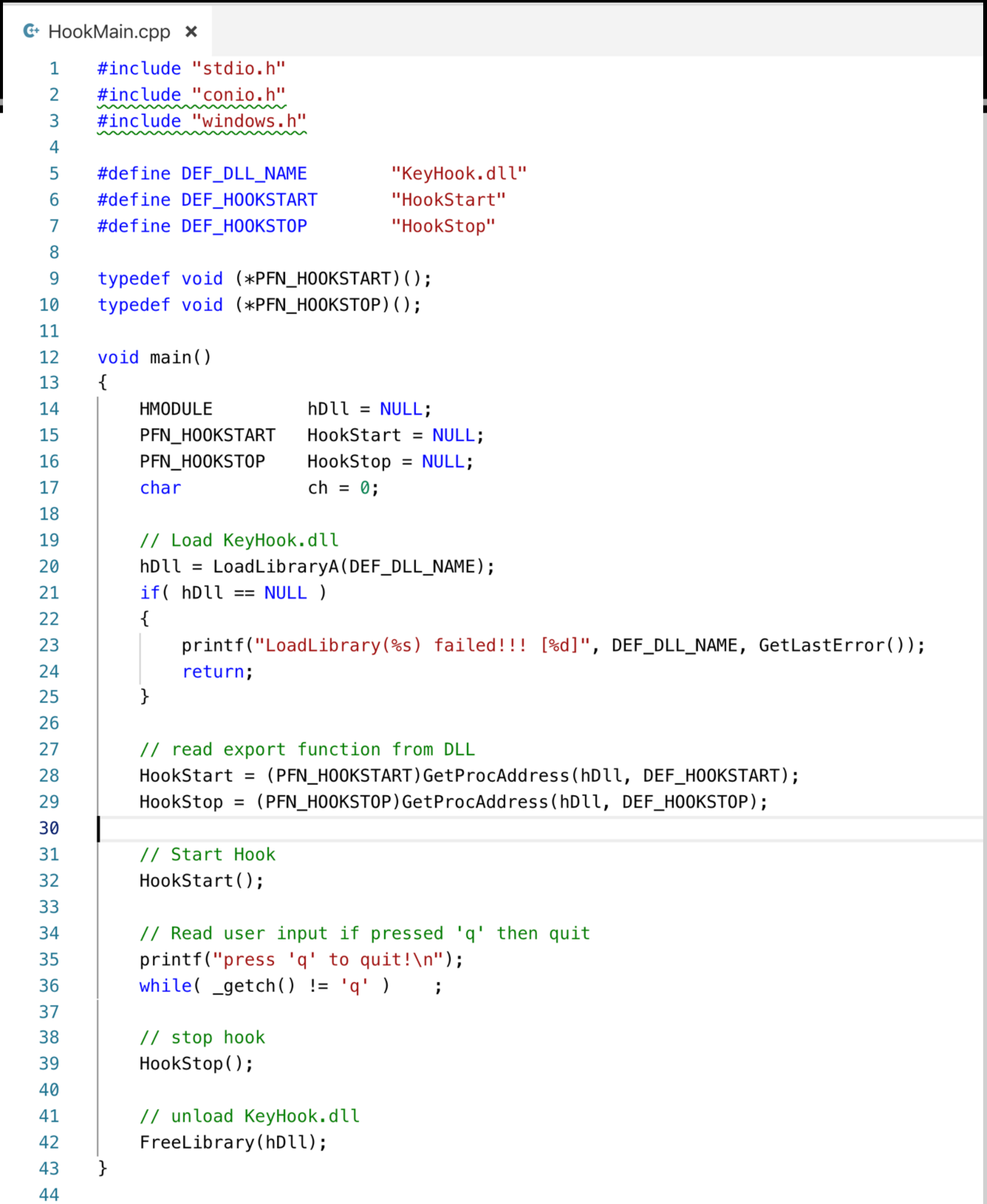
Message Hook



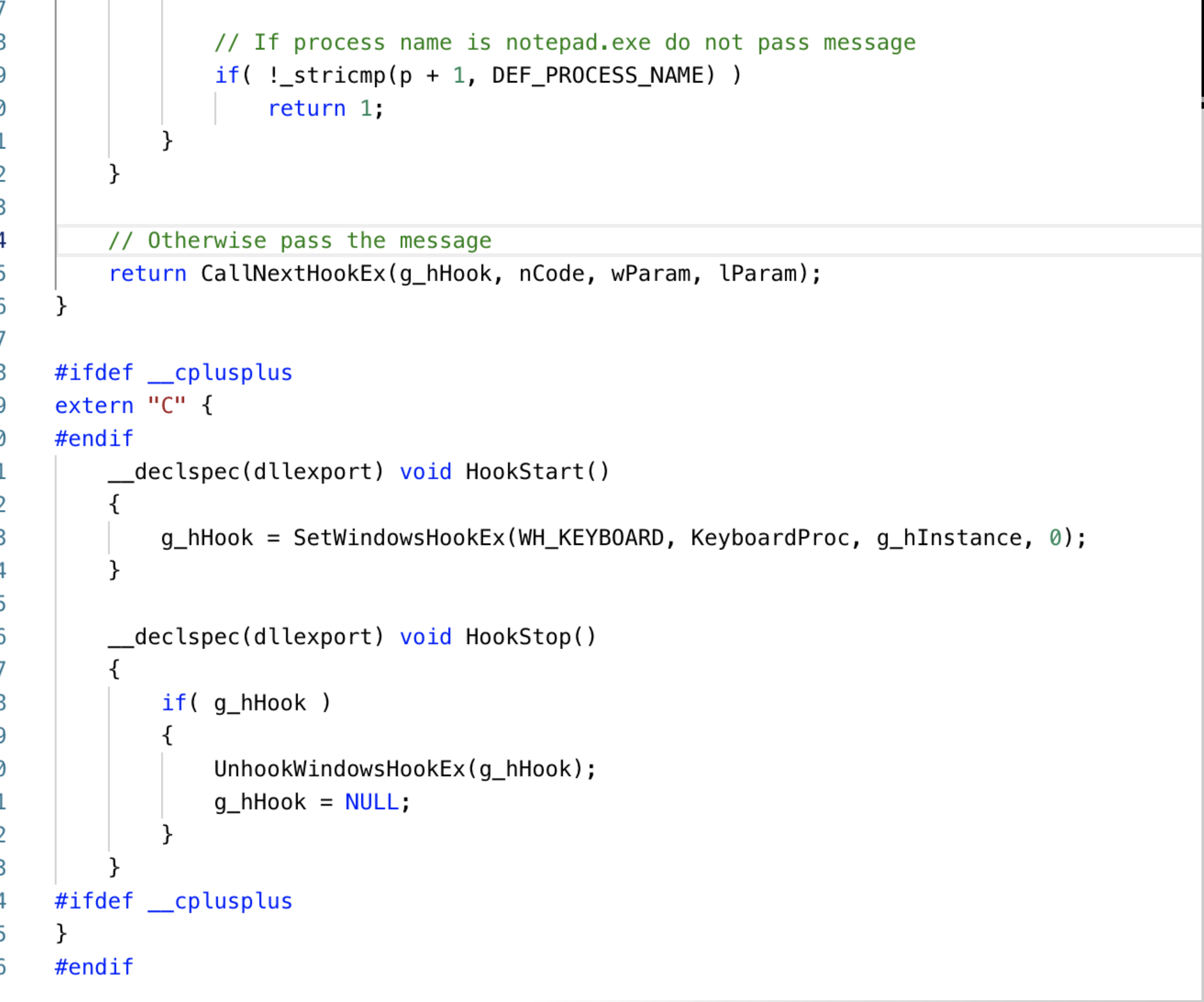
Message Hook



1. User Press A Key
2. WM_KEYDOWN Message is added to OS Message Queue
OS Message Queue
Message
Hook1
Hook2
Hook3
…
Hook Chain
3. OS Checked and find out who owns this WM_KEYDOWN event
Take it out from the OS Message Queue and Put it into Application Message Queue
Application Message Queue
4. Application keep monitoring the Application Message Queue. Once it finds the WM_KEYDOWN Event, it'll process it .

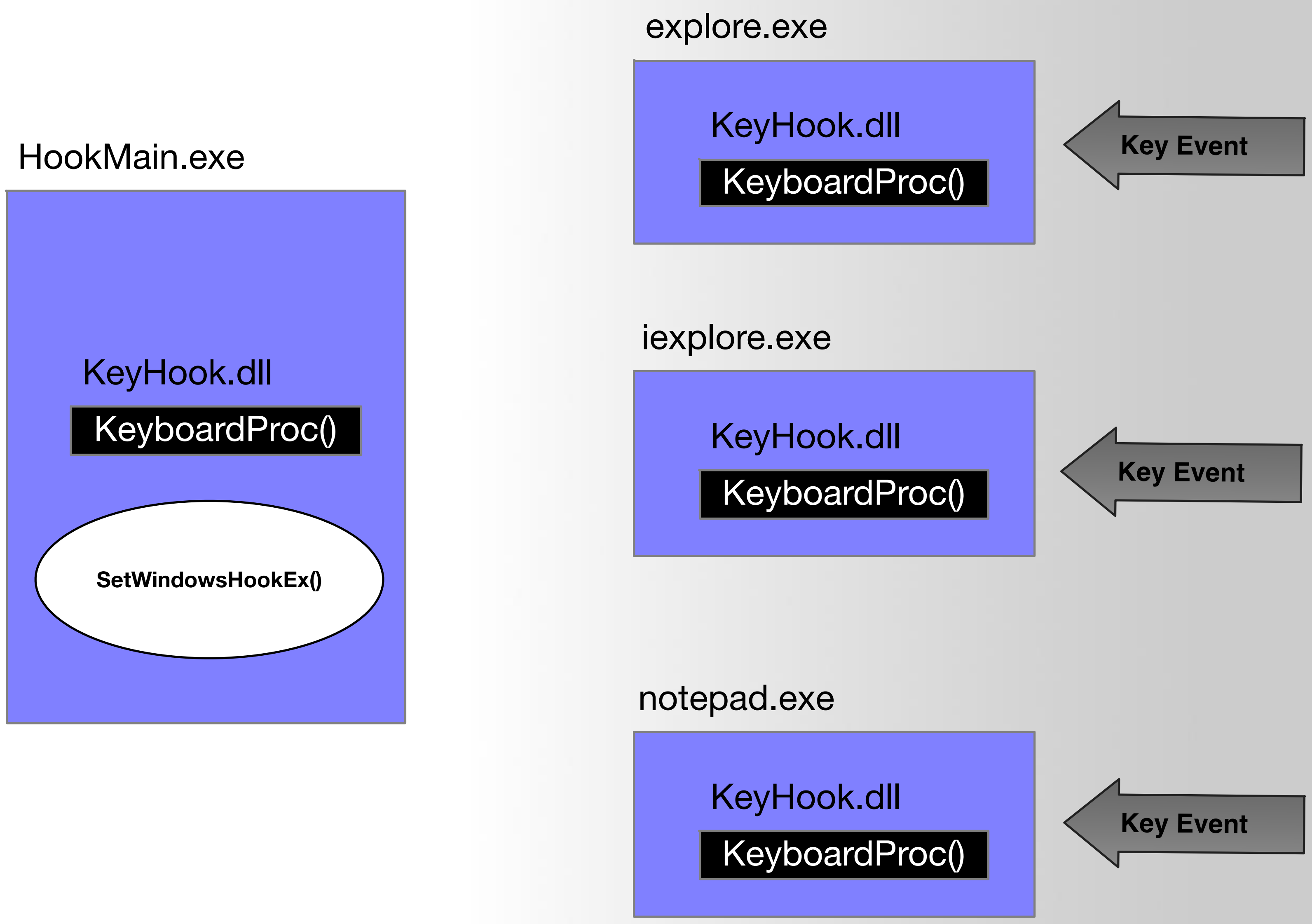

HookMain.cpp

```
#include "stdio.h"
#include "conio.h"
#include "windows.h"

#define DEF_DLL_NAME        "KeyHook.dll"
#define DEF_HOOKSTART       "HookStart"
#define DEF_HOOKSTOP        "HookStop"

typedef void (*PFN_HOOKSTART)();
typedef void (*PFN_HOOKSTOP)();

void main()
{
    HMODULE         hDll = NULL;
    PFN_HOOKSTART   HookStart = NULL;
    PFN_HOOKSTOP    HookStop = NULL;
    char            ch = 0;

    // Load KeyHook.dll
    hDll = LoadLibraryA(DEF_DLL_NAME);
    if( hDll == NULL )
    {
        printf("LoadLibrary(%s) failed!!! [%d]", DEF_DLL_NAME, GetLastError());
        return;
    }

    // read export function from DLL
    HookStart = (PFN_HOOKSTART)GetProcAddress(hDll, DEF_HOOKSTART);
    HookStop = (PFN_HOOKSTOP)GetProcAddress(hDll, DEF_HOOKSTOP);

    // Start Hook
    HookStart();

    // Read user input if pressed 'q' then quit
    printf("press 'q' to quit!\n");
    while( _getch() != 'q' )    ;

    // stop hook
    HookStop();

    // unload KeyHook.dll
    FreeLibrary(hDll);
}

```



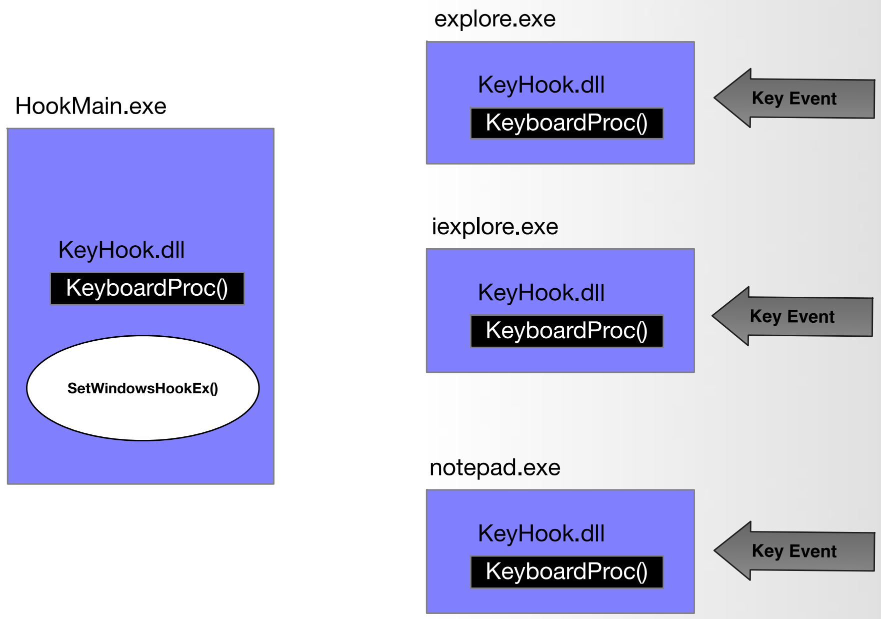
```
            // If process name is notepad.exe do not pass message
            if( !_stricmp(p + 1, DEF_PROCESS_NAME) )
                return 1;
        }
    }

    // Otherwise pass the message
    return CallNextHookEx(g_hHook, nCode, wParam, lParam);
}

#ifdef __cplusplus
extern "C" {
#endif
    __declspec(dllexport) void HookStart()
    {
        g_hHook = SetWindowsHookEx(WH_KEYBOARD, KeyboardProc, g_hInstance, 0);
    }

    __declspec(dllexport) void HookStop()
    {
        if( g_hHook )
        {
            UnhookWindowsHookEx(g_hHook);
            g_hHook = NULL;
        }
    }
#ifdef __cplusplus
}
#endif
```

Review – Message Hook



API Hook Tech Map

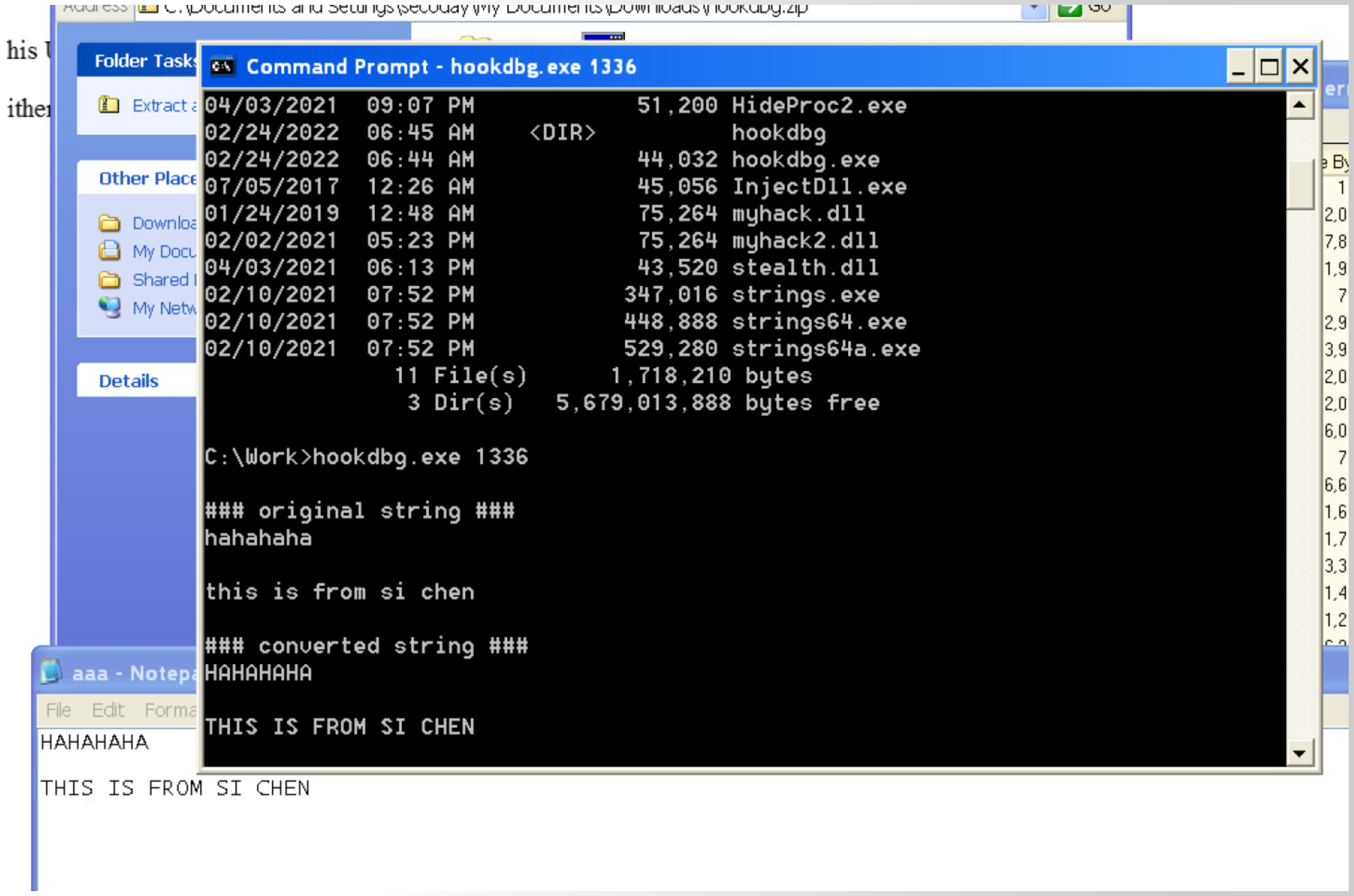
Method	Target	Location	Tech		API
Dynamic	**Process/Memory** **00000000** **- 7FFFFFFF**	**1) IAT** **2) Code** **3) EAT**	**Interactive Debug**		DebugActiveProcess GetThreadContext SetThreadContext
			Standalone Injection	**Independent Code**	CreateRemoteThread
				Dll File	Resistry (AppInit_DLLs) BHO (IE only)
					SetWindowsHookEx CreateRemoteThread



API Hooks

Hookdbg.exe

- API hook for Notepad WriteFile() function

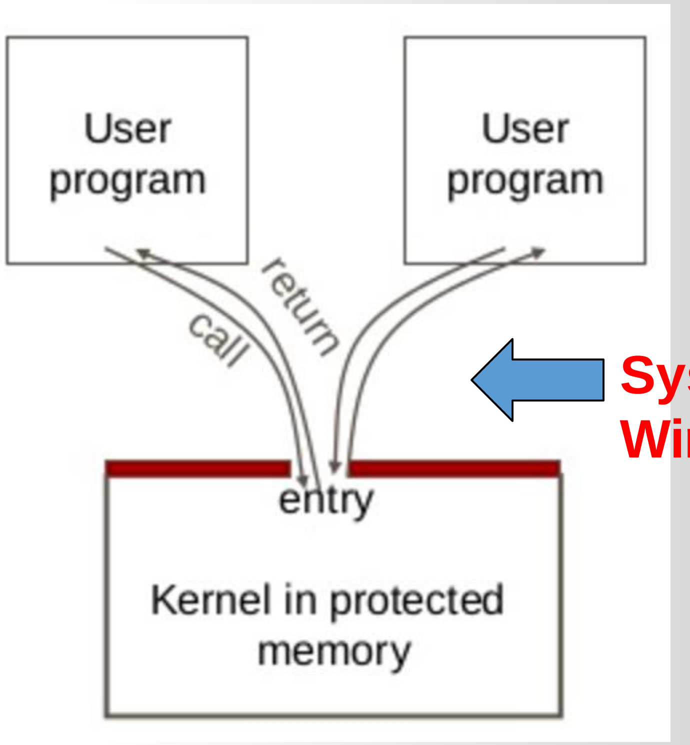


```
04/03/2021  09:07 PM            51,200 HideProc2.exe
02/24/2022  06:45 AM    <DIR>          hookdbg
02/24/2022  06:44 AM            44,032 hookdbg.exe
07/05/2017  12:26 AM            45,056 InjectDll.exe
01/24/2019  12:48 AM            75,264 myhack.dll
02/02/2021  05:23 PM            75,264 myhack2.dll
04/03/2021  06:13 PM            43,520 stealth.dll
02/10/2021  07:52 PM           347,016 strings.exe
02/10/2021  07:52 PM           448,888 strings64.exe
02/10/2021  07:52 PM           529,280 strings64a.exe
              11 File(s)      1,718,210 bytes
               3 Dir(s)   5,679,013,888 bytes free

C:\Work>hookdbg.exe 1336

### original string ###
hahahaha

this is from si chen

### converted string ###
HAHAHAHA

THIS IS FROM SI CHEN
```

```
HAHAHAHA

THIS IS FROM SI CHEN
```

System Call & WinAPI

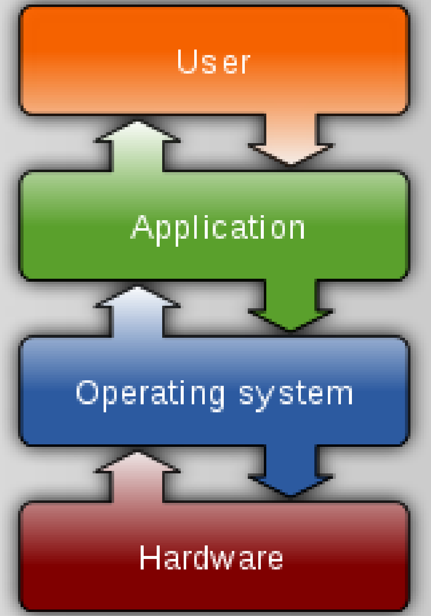
- User code can be arbitrary
- User code cannot modify kernel memory
- The call mechanism switches code to kernel mode



System Call → WinAPI (Windows)

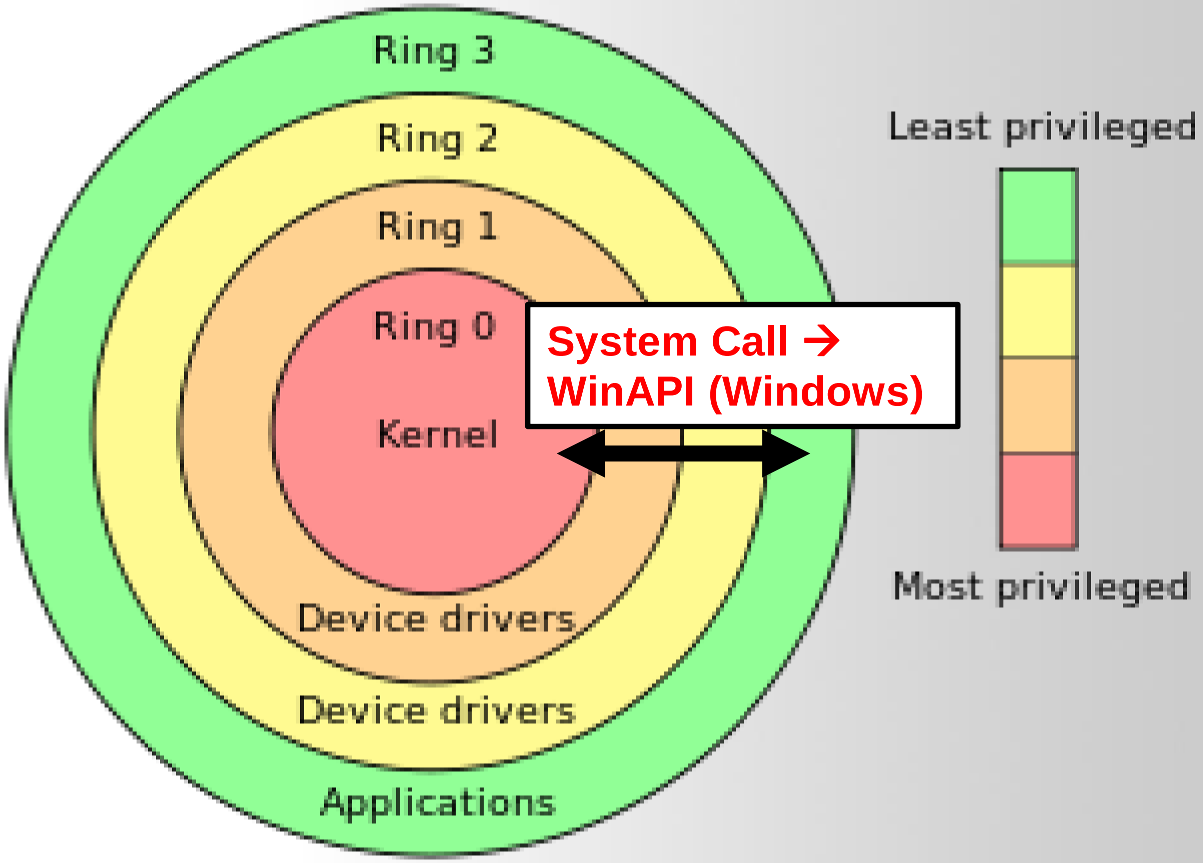
What is System Call?

- Let an application to access system resources.
- OS provide an interface (**System call**) for the application
- It usually use the technique called "interrupt vector"
  - Linux use **0x80**
  - Windows use **SYSENTER**



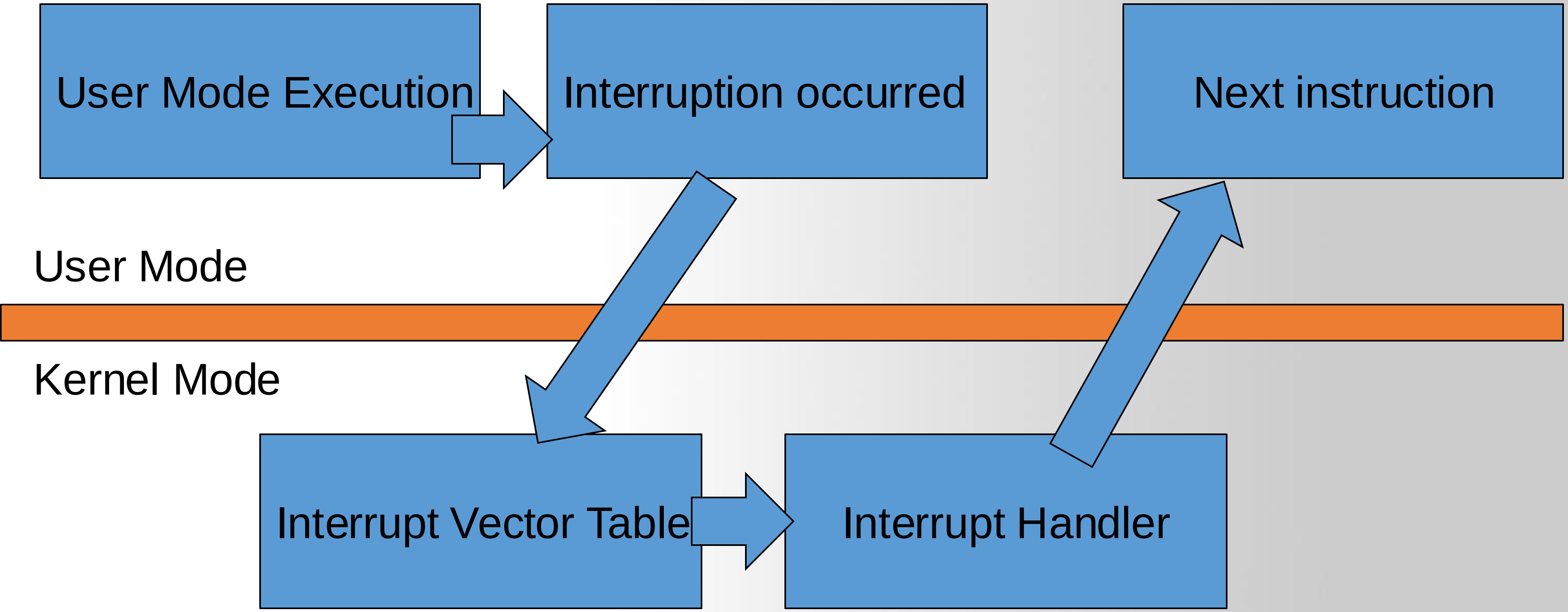
In system programming, an **interrupt** is a signal to the processor emitted by hardware or software indicating an event that needs immediate attention.

The “Ring”



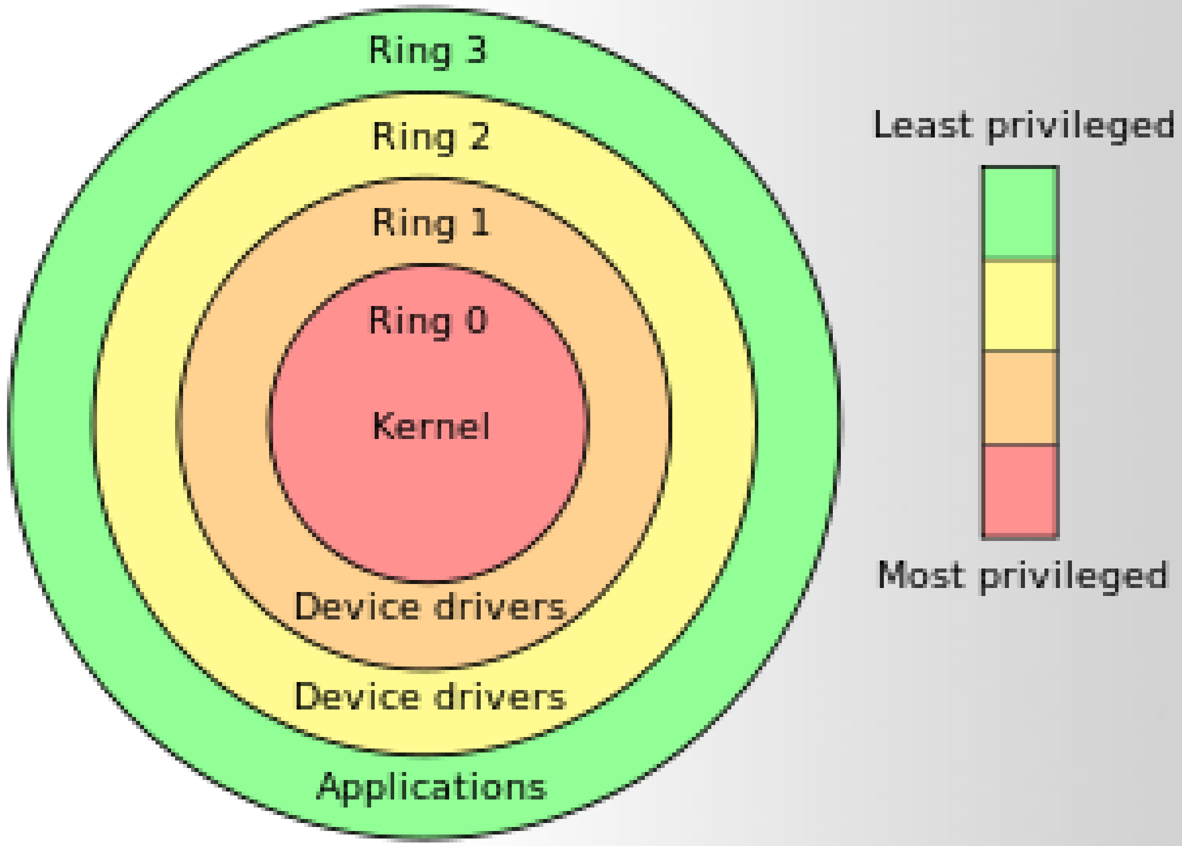
System Call → WinAPI (Windows)

CPU Interrupt

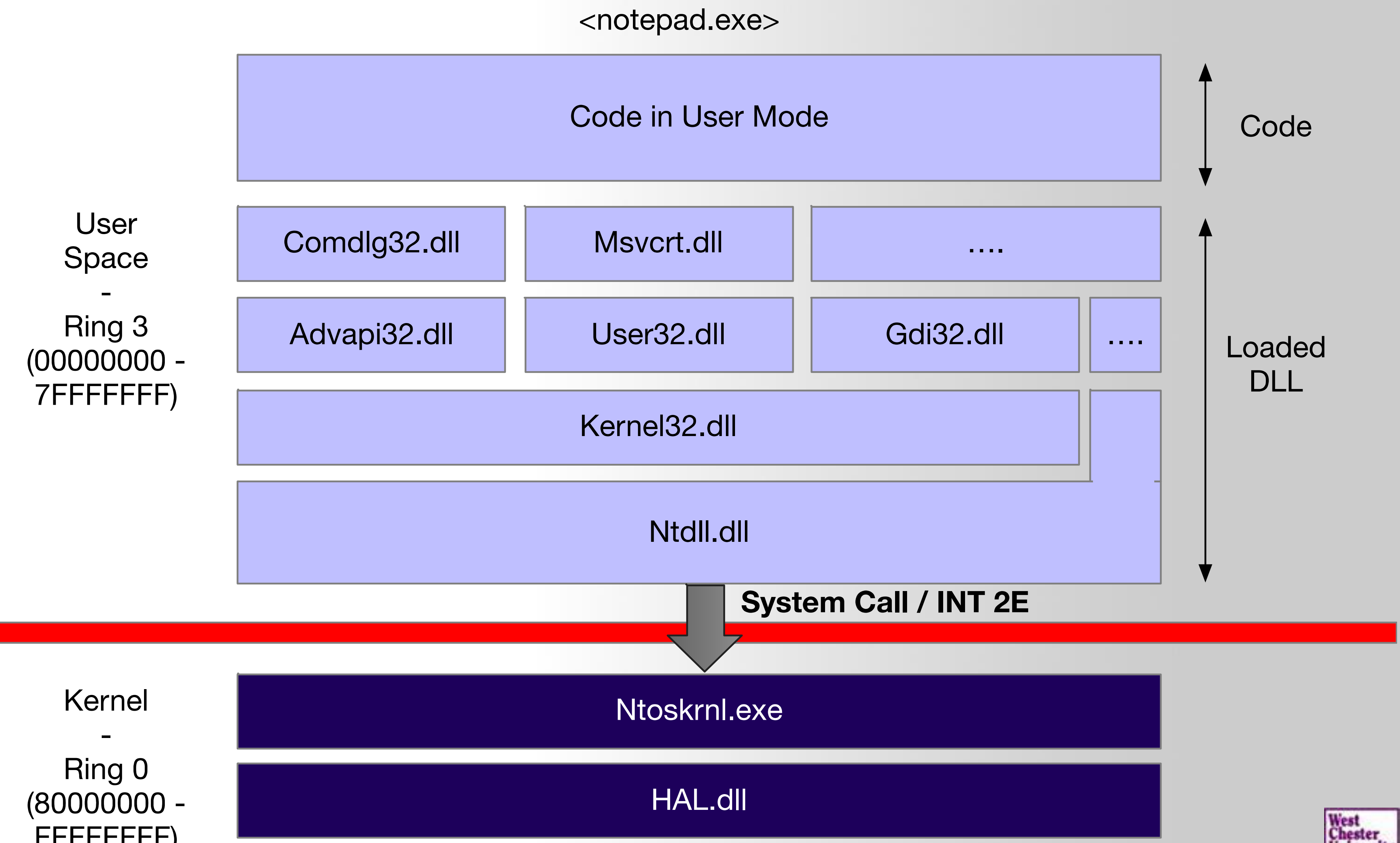


Windows System Call and API

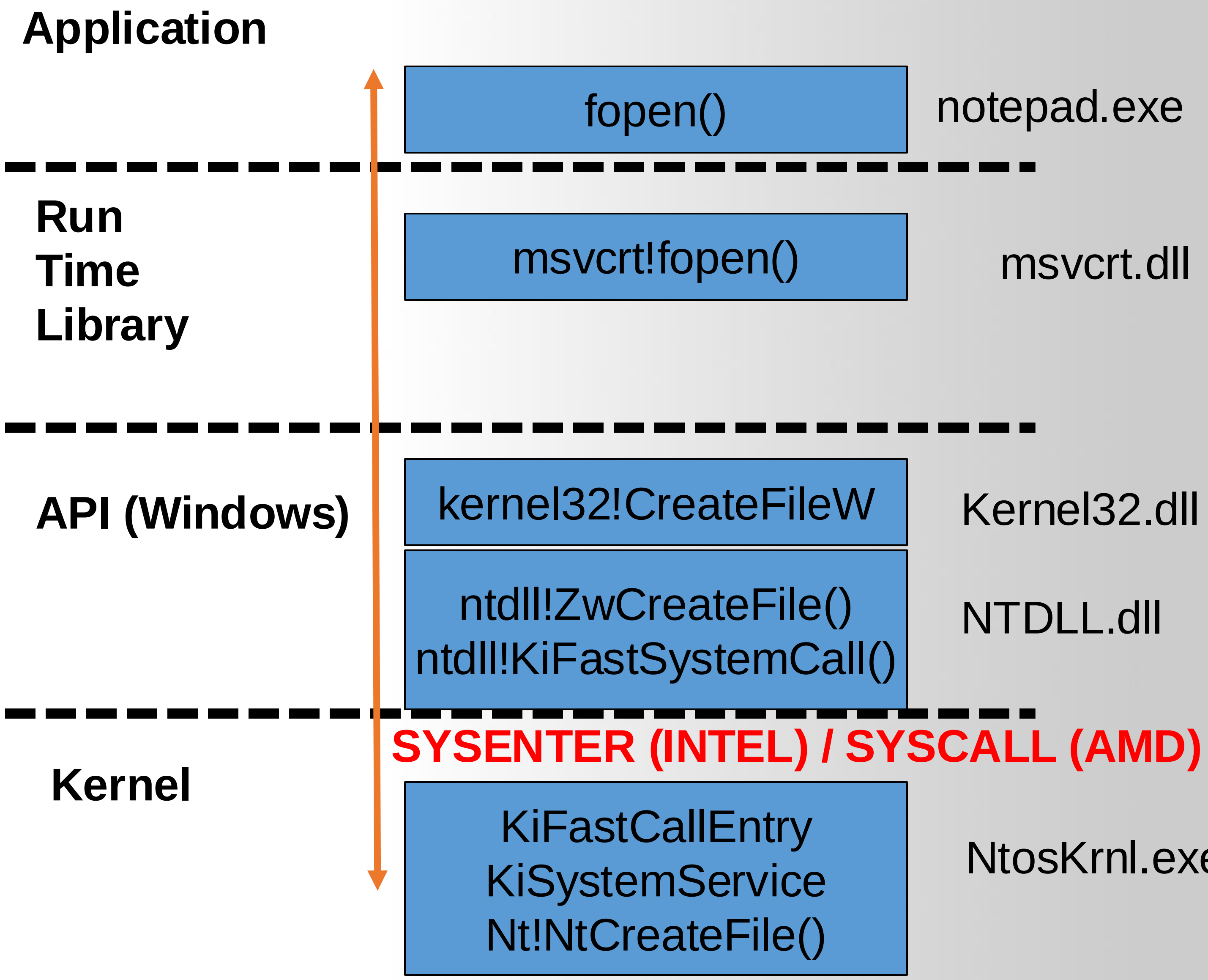
- The **Win32 API** is a layer that **runs in user mode** (ring 3).
- **Only API calls that use kernel resources** (CreateThread, VirtualAlloc, etc) will call into the "real" operating system (ntdll.dll) and trap into **ring 0** with a software interrupt (int 0x2e).



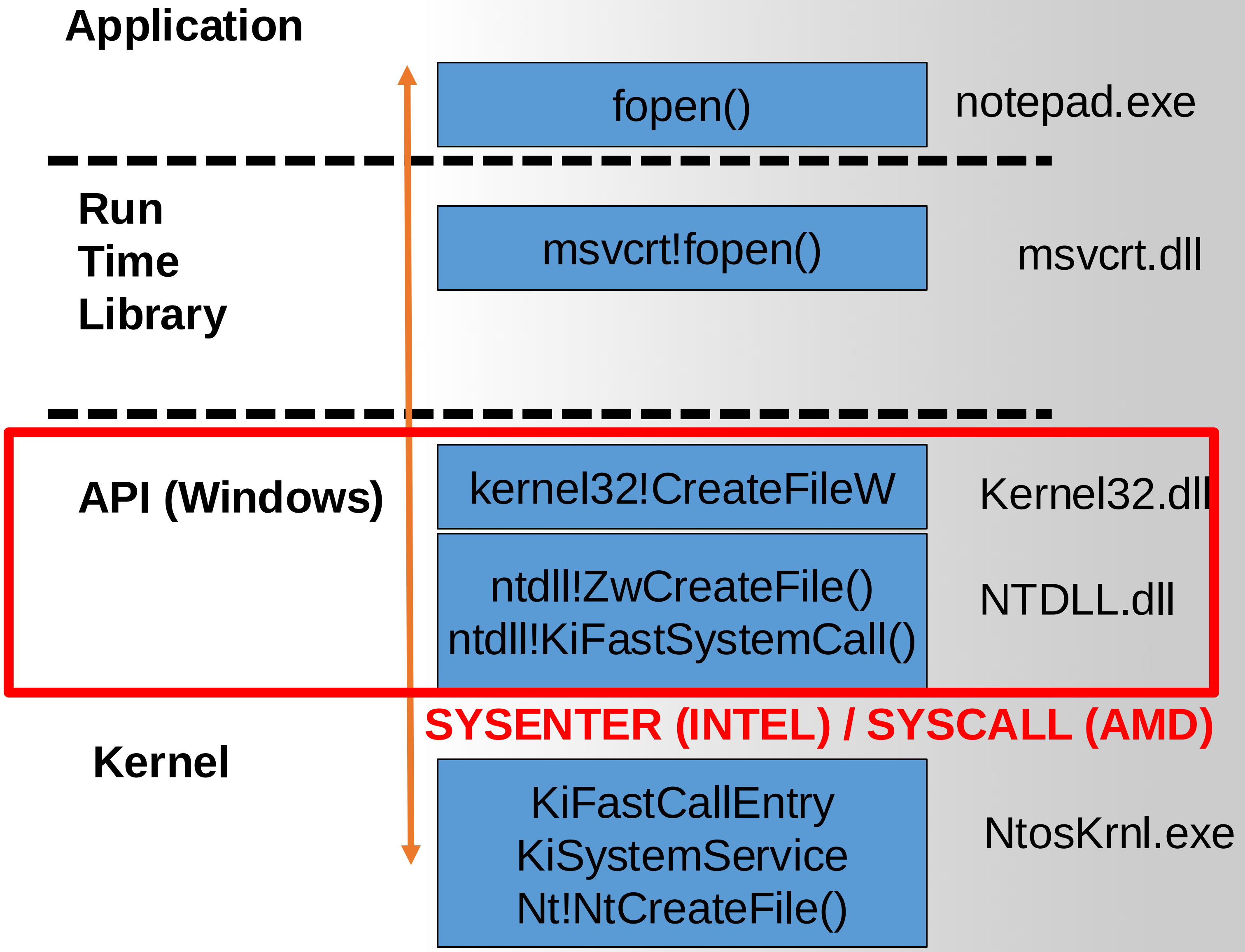
User Mode and Kernel



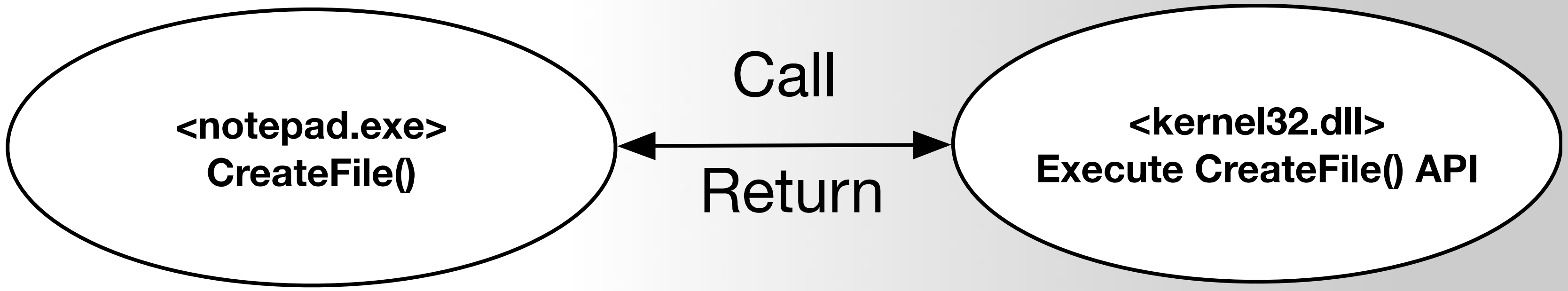
Open a file in Notepad



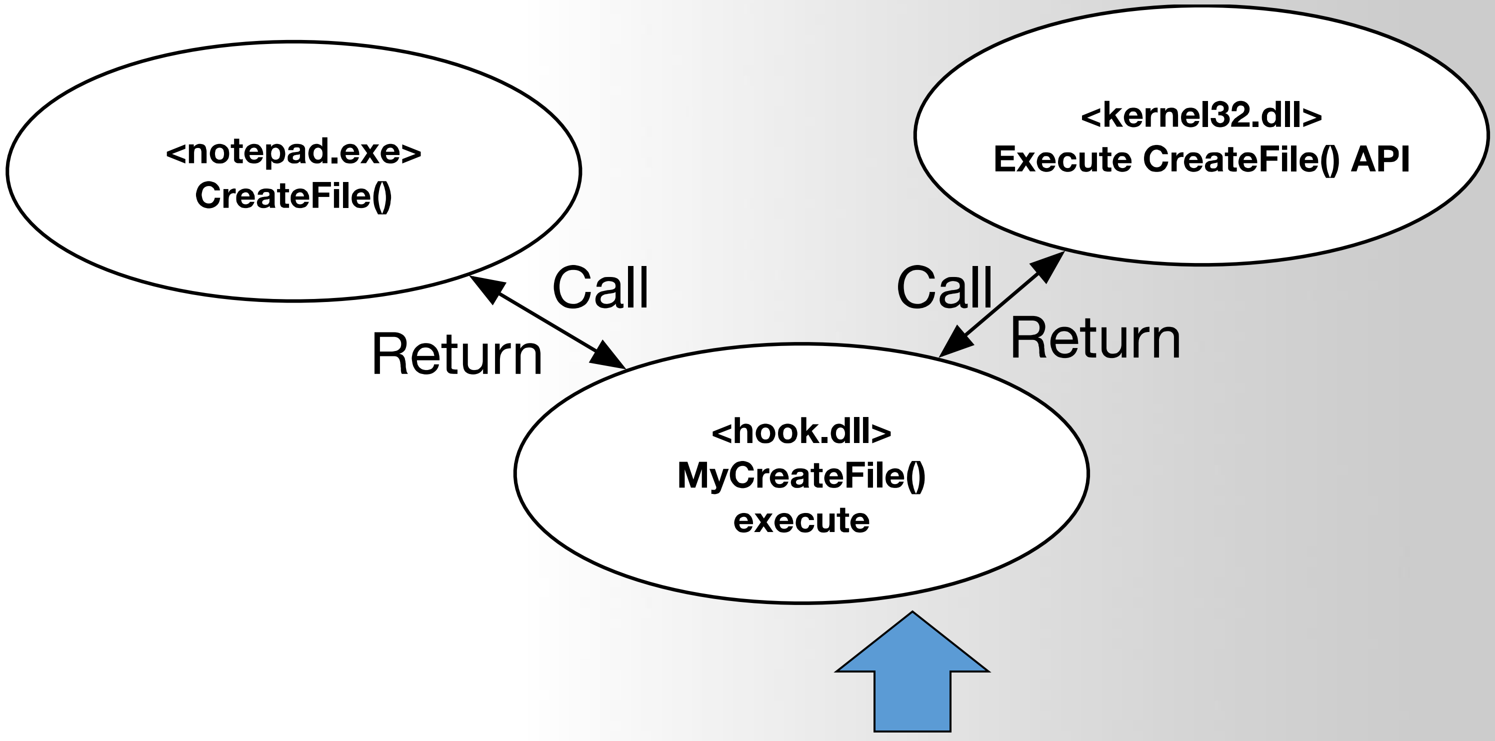
Open a file in Notepad



API Call (Normally)



API Hook



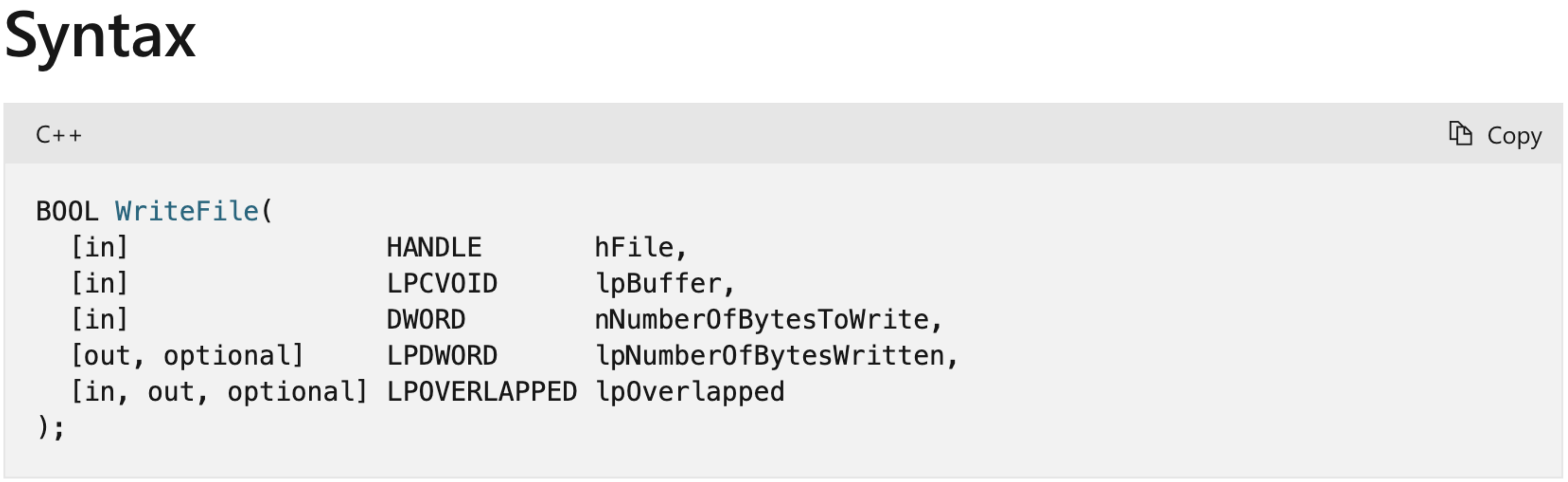
Detour Function

API Hook Tech Map

Method	Target	Location	Tech		API
Dynamic	**Process/Memory** **00000000** **- 7FFFFFFF**	**1) IAT** **2) Code** **3) EAT**	**Interactive Debug**		DebugActiveProcess GetThreadContext SetThreadContext
			Standalone Injection	**Independent Code**	CreateRemoteThread
				Dll File	Resistry (AppInit_DLLs) BHO (IE only)
					SetWindowsHookEx CreateRemoteThread

Hookdbg.exe

- kernel32!WriteFile() API


Syntax

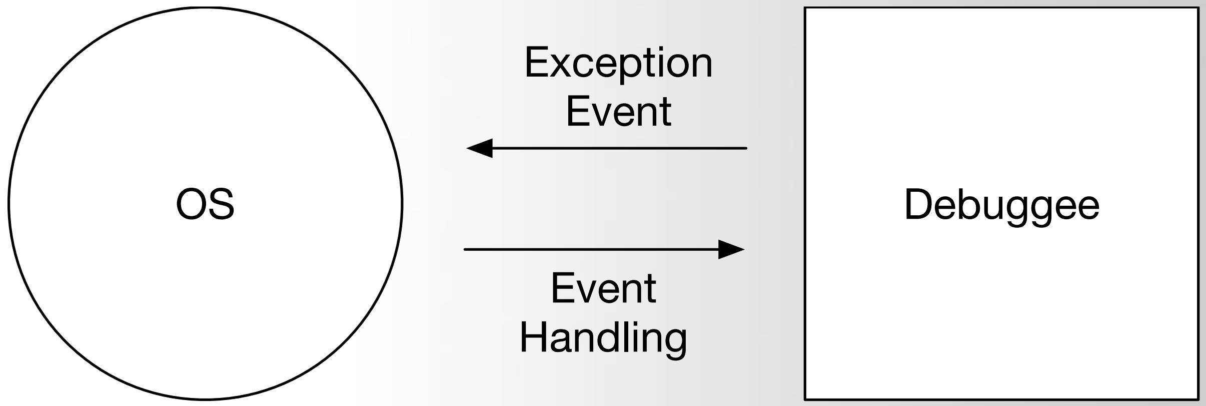
C++

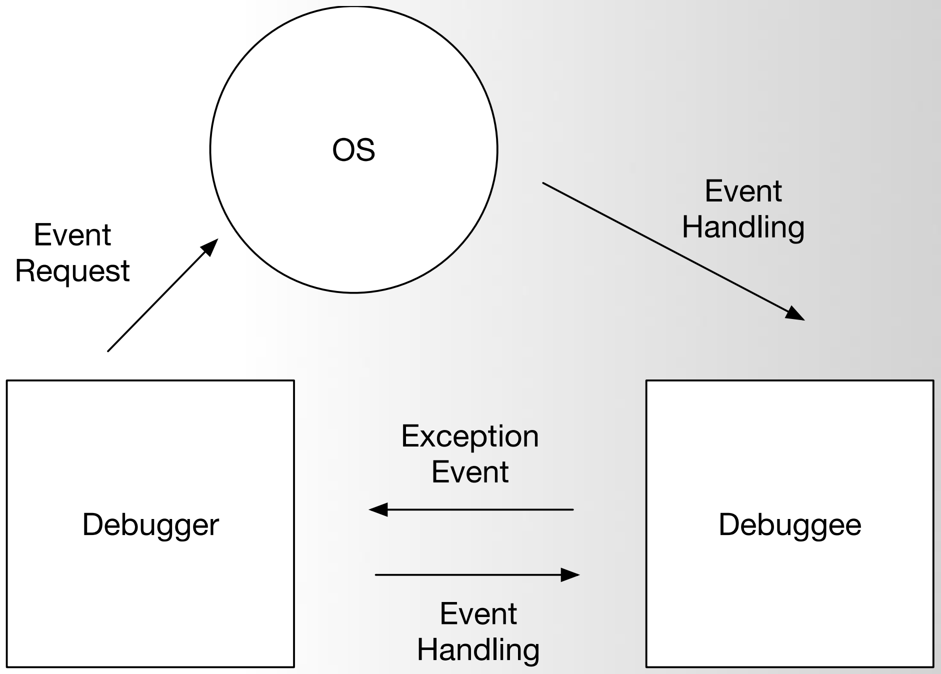
Copy

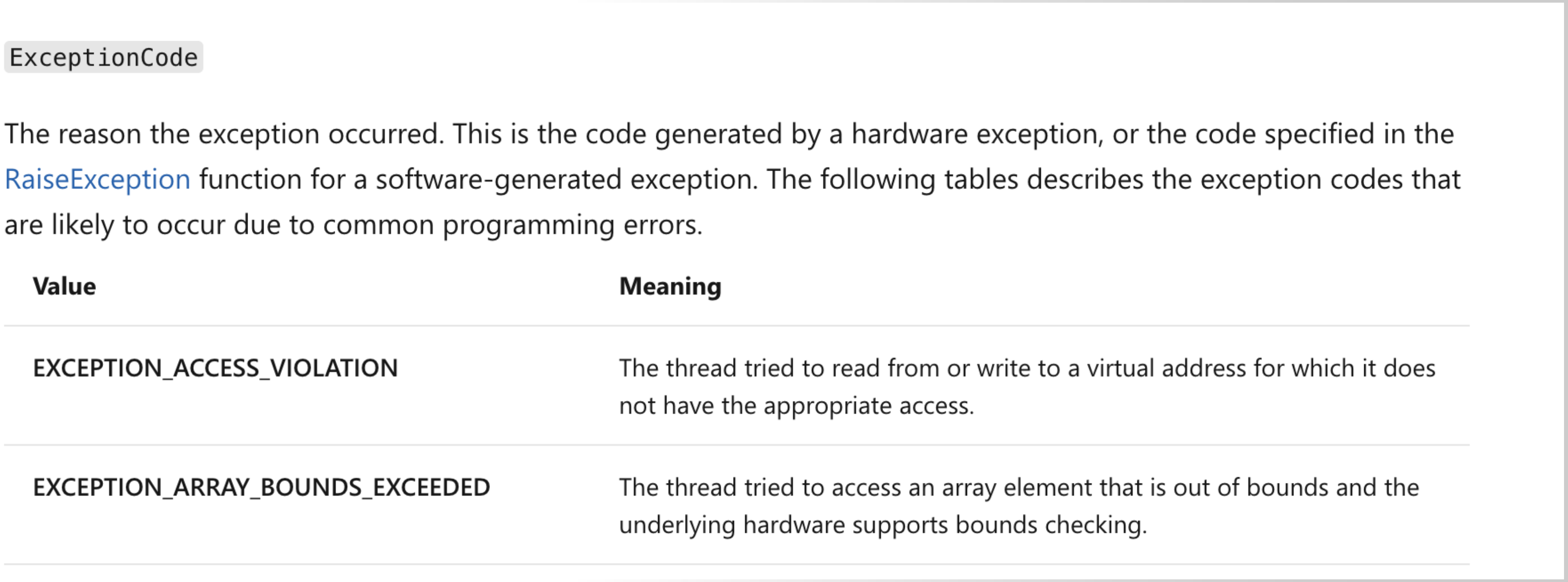
```
BOOL WriteFile(
  [in]                HANDLE       hFile,
  [in]                LPCVOID      lpBuffer,
  [in]                DWORD        nNumberOfBytesToWrite,
  [out, optional]     LPDWORD      lpNumberOfBytesWritten,
  [in, out, optional] LPOVERLAPPED lpOverlapped
);
```

https://docs.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-writefile

How Debugger Works







`ExceptionCode`

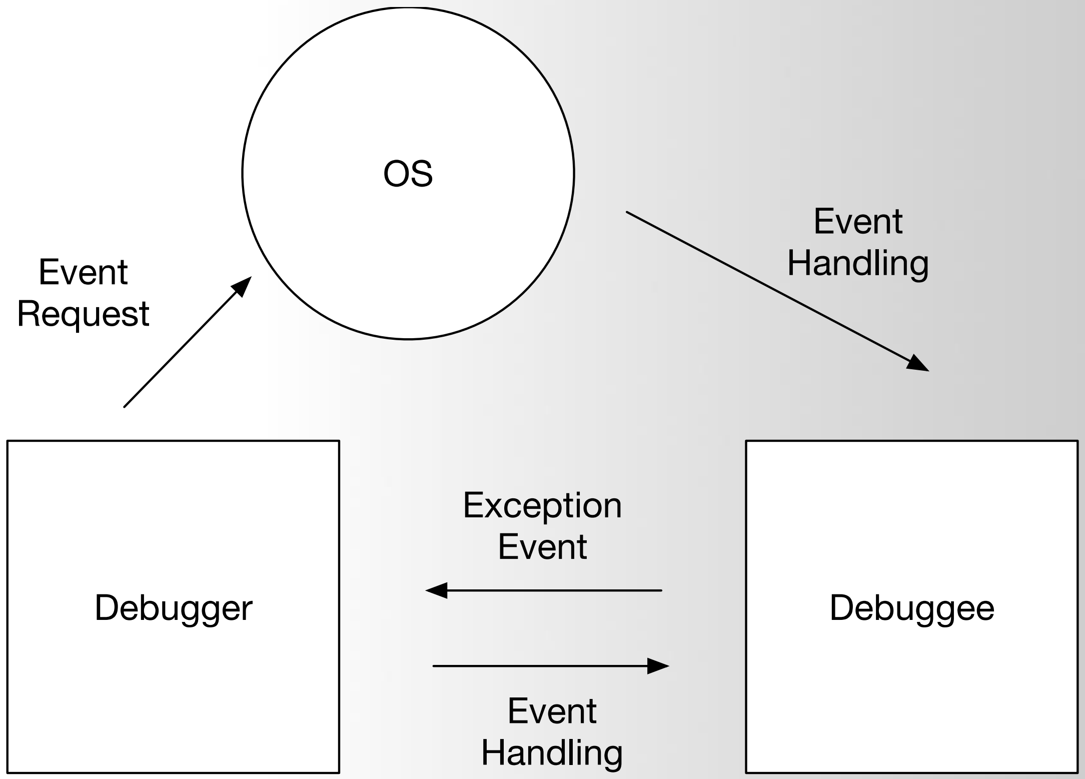
The reason the exception occurred. This is the code generated by a hardware exception, or the code specified in the RaiseException function for a software-generated exception. The following tables describes the exception codes that are likely to occur due to common programming errors.

Value	Meaning
EXCEPTION_ACCESS_VIOLATION	The thread tried to read from or write to a virtual address for which it does not have the appropriate access.
EXCEPTION_ARRAY_BOUNDS_EXCEEDED	The thread tried to access an array element that is out of bounds and the underlying hardware supports bounds checking.

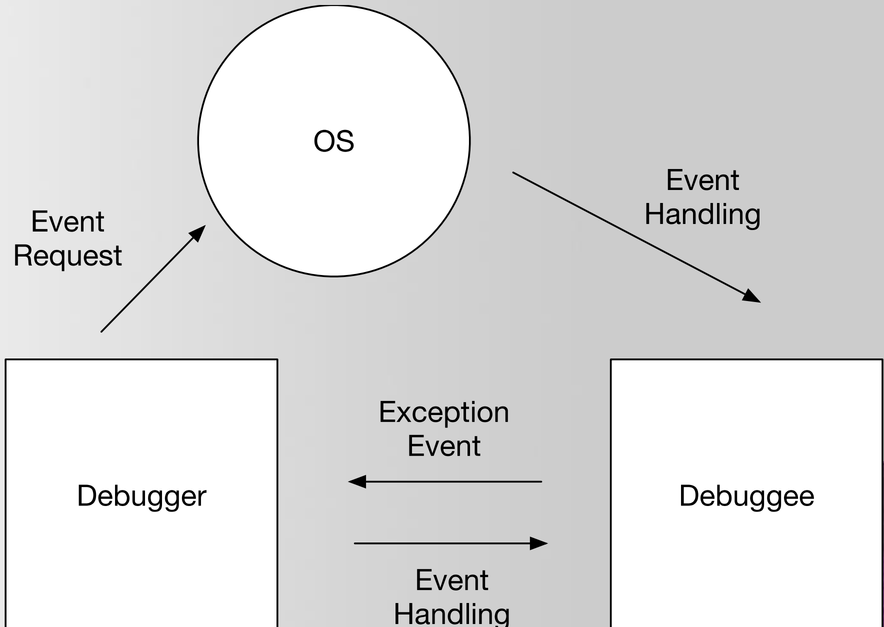
https://docs.microsoft.com/en-us/windows/win32/api/winnt/ns-winnt-exception_record

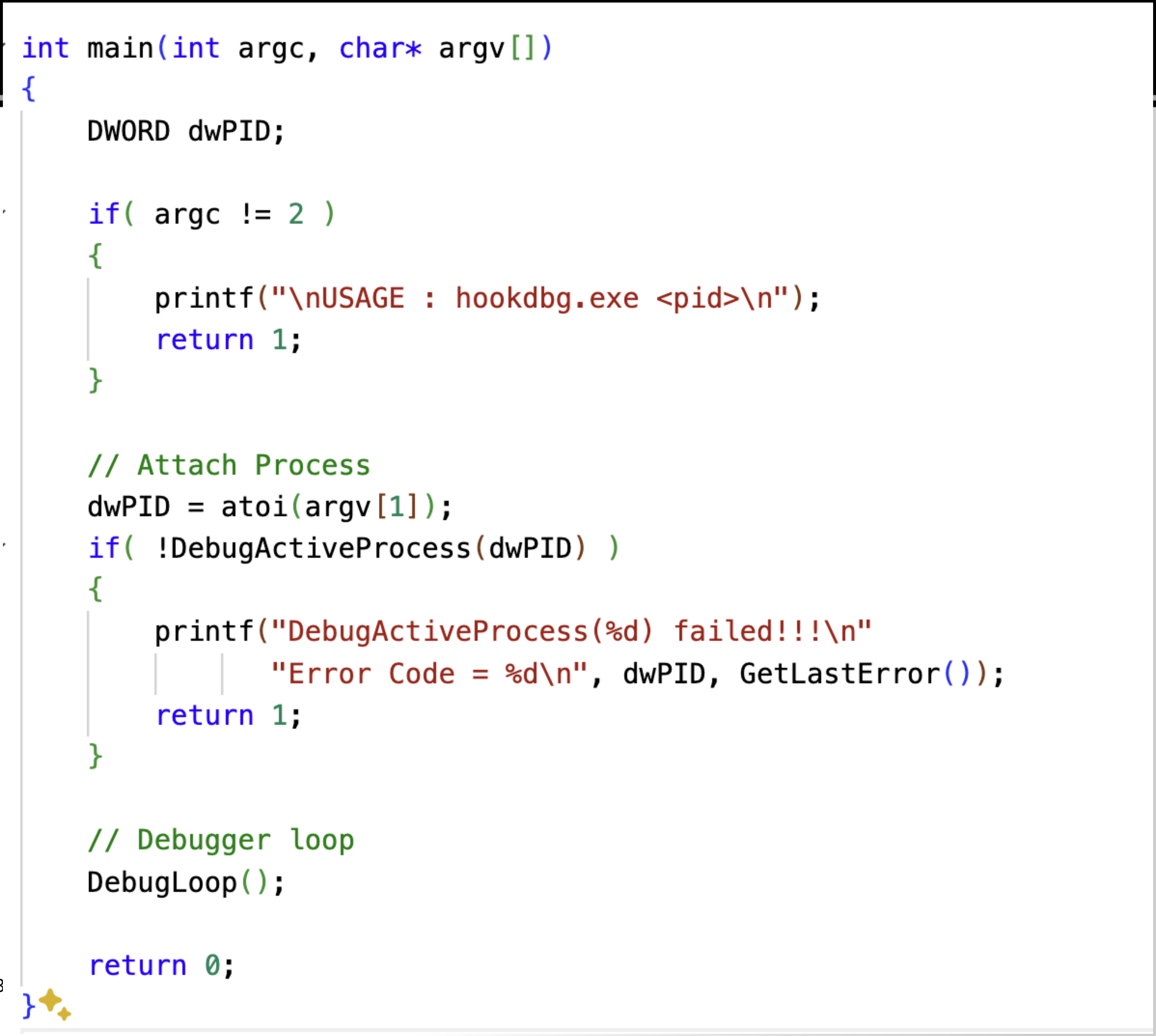
Debugging Techniques and Workflow

- Hooking APIs has been used in debugging techniques:
  - The basic idea is, in the "debugger-debuggee" state, to modify the starting part of the debuggee's API to **0xCC**, transferring control to the debugger to perform specified operations, and finally returning the debuggee to a running state.

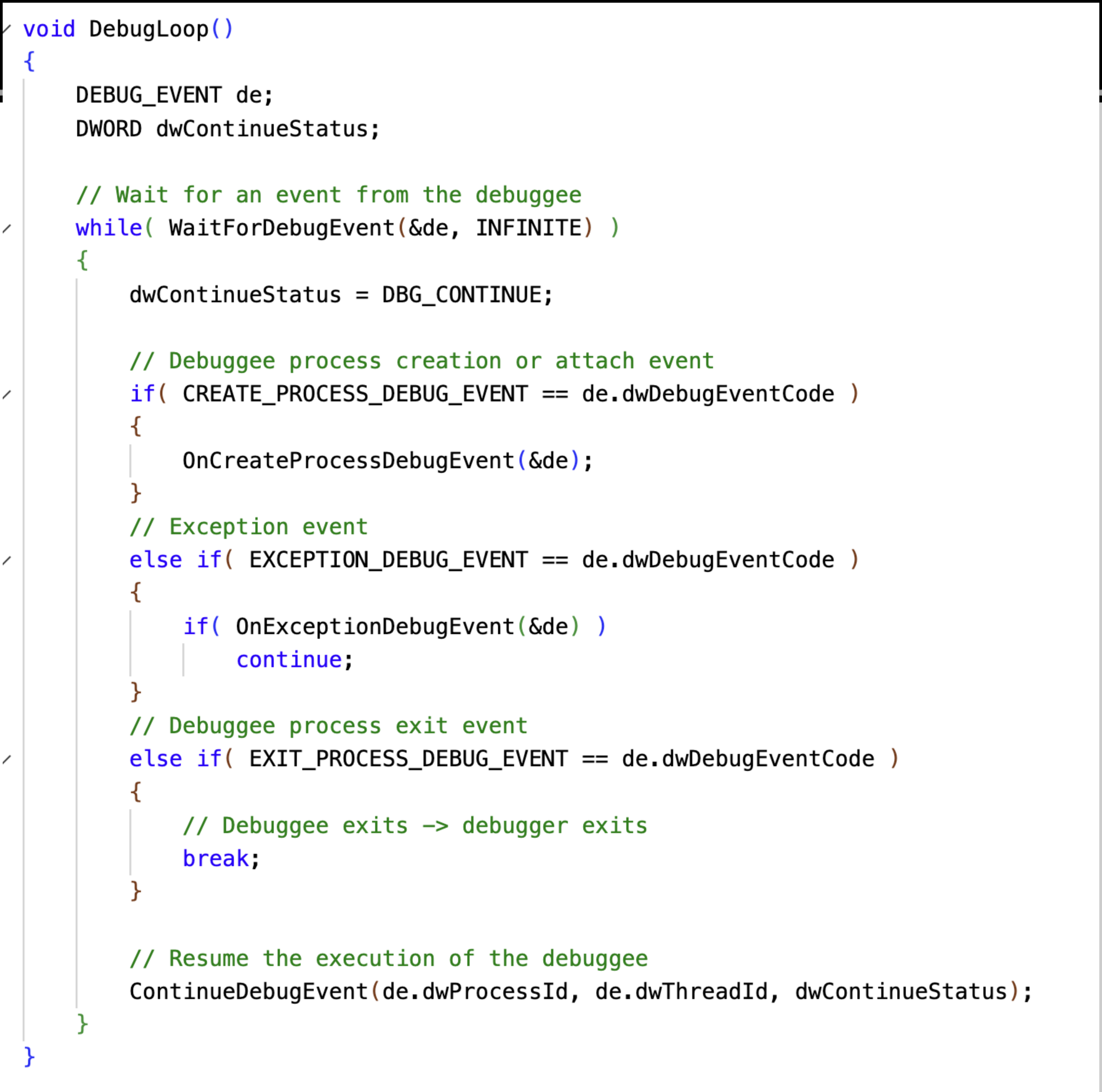


- The specific debugging process is as follows:
  1. Attach to the process you want to hook, making it the debuggee.
  2. Hook: Change the **first byte** of the API's starting address to **0xCC**.
  3. When the corresponding API is called, control is transferred to the debugger.
  4. Perform the necessary operations (operating parameters, return values, etc.).
  5. Unhook: Restore 0xCC to its original value (to ensure the API runs normally).
  6. Run the corresponding API (in a normal state without 0xCC).
  7. Hook: Modify it to 0xCC again (for continued hooking).
  8. Return control to the debuggee.

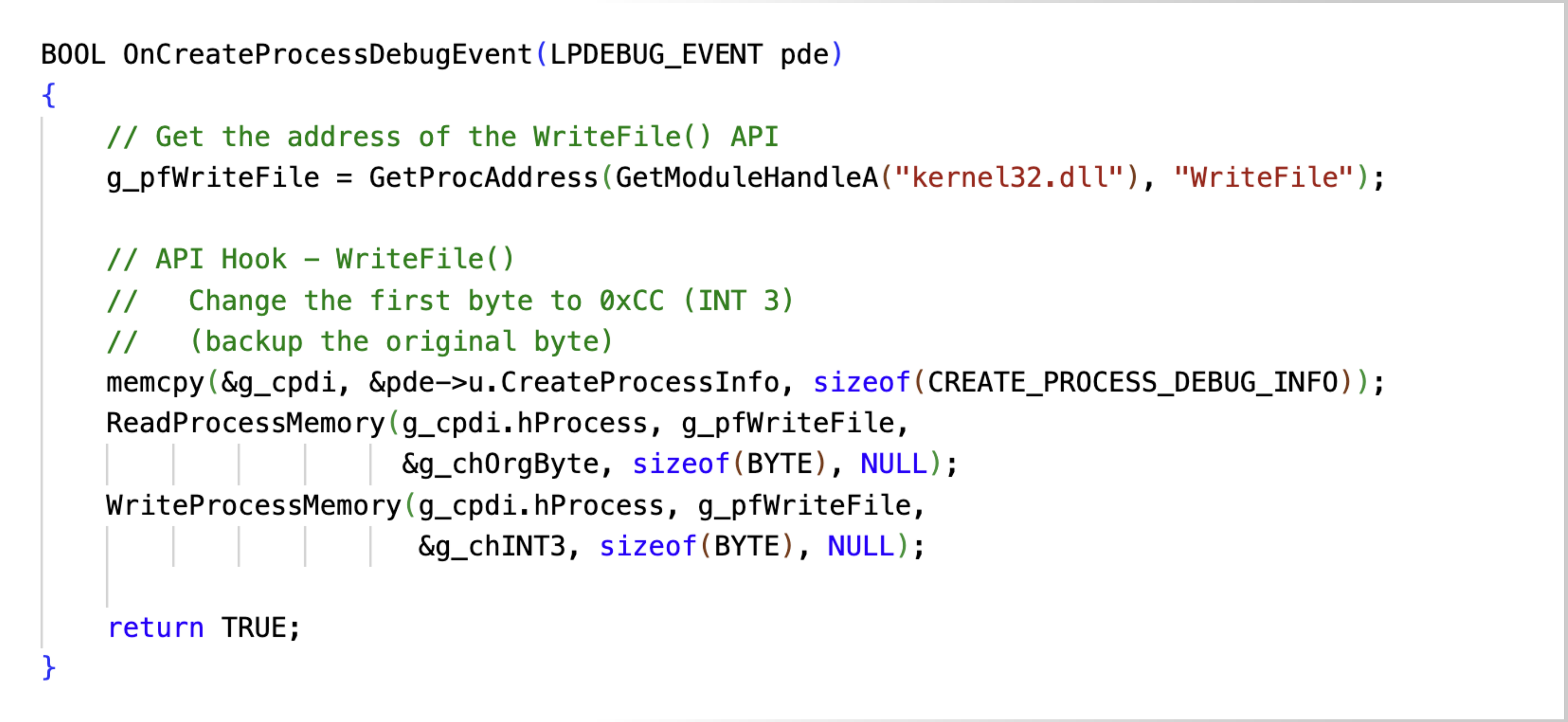




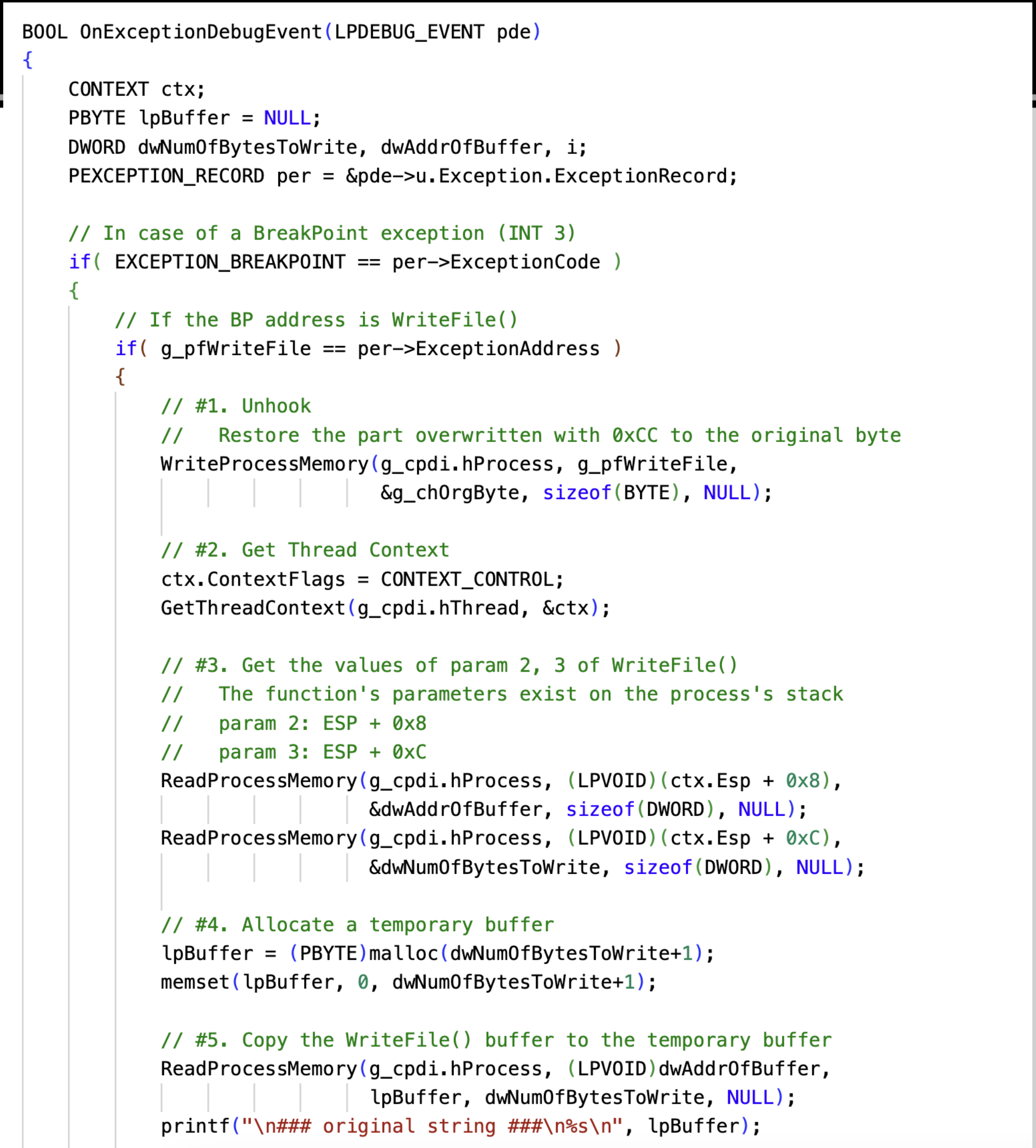
```c
int main(int argc, char* argv[])
{
    DWORD dwPID;

    if( argc != 2 )
    {
        printf("\nUSAGE : hookdbg.exe <pid>\n");
        return 1;
    }

    // Attach Process
    dwPID = atoi(argv[1]);
    if( !DebugActiveProcess(dwPID) )
    {
        printf("DebugActiveProcess(%d) failed!!!\n"
                   "Error Code = %d\n", dwPID, GetLastError());
        return 1;
    }

    // Debugger loop
    DebugLoop();

    return 0;
}
```



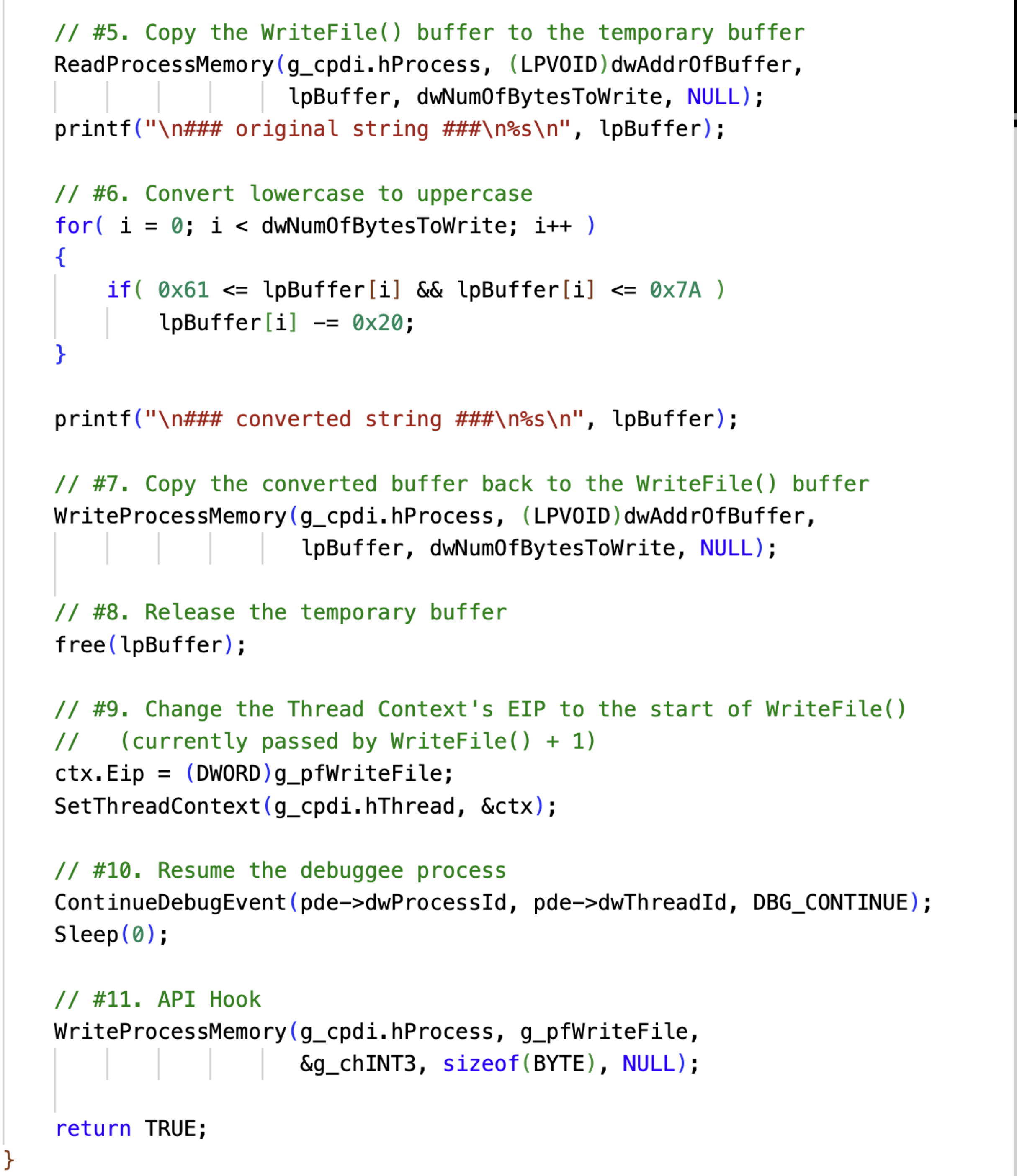
```
void DebugLoop()
{
    DEBUG_EVENT de;
    DWORD dwContinueStatus;

    // Wait for an event from the debuggee
    while( WaitForDebugEvent(&de, INFINITE) )
    {
        dwContinueStatus = DBG_CONTINUE;

        // Debuggee process creation or attach event
        if( CREATE_PROCESS_DEBUG_EVENT == de.dwDebugEventCode )
        {
            OnCreateProcessDebugEvent(&de);
        }
        // Exception event
        else if( EXCEPTION_DEBUG_EVENT == de.dwDebugEventCode )
        {
            if( OnExceptionDebugEvent(&de) )
                continue;
        }
        // Debuggee process exit event
        else if( EXIT_PROCESS_DEBUG_EVENT == de.dwDebugEventCode )
        {
            // Debuggee exits -> debugger exits
            break;
        }

        // Resume the execution of the debuggee
        ContinueDebugEvent(de.dwProcessId, de.dwThreadId, dwContinueStatus);
    }
}
```



```
BOOL OnCreateProcessDebugEvent(LPDEBUG_EVENT pde)
{
    // Get the address of the WriteFile() API
    g_pfWriteFile = GetProcAddress(GetModuleHandleA("kernel32.dll"), "WriteFile");

    // API Hook - WriteFile()
    //   Change the first byte to 0xCC (INT 3)
    //   (backup the original byte)
    memcpy(&g_cpdi, &pde->u.CreateProcessInfo, sizeof(CREATE_PROCESS_DEBUG_INFO));
    ReadProcessMemory(g_cpdi.hProcess, g_pfWriteFile,
                      &g_chOrgByte, sizeof(BYTE), NULL);
    WriteProcessMemory(g_cpdi.hProcess, g_pfWriteFile,
                       &g_chINT3, sizeof(BYTE), NULL);

    return TRUE;
}
```

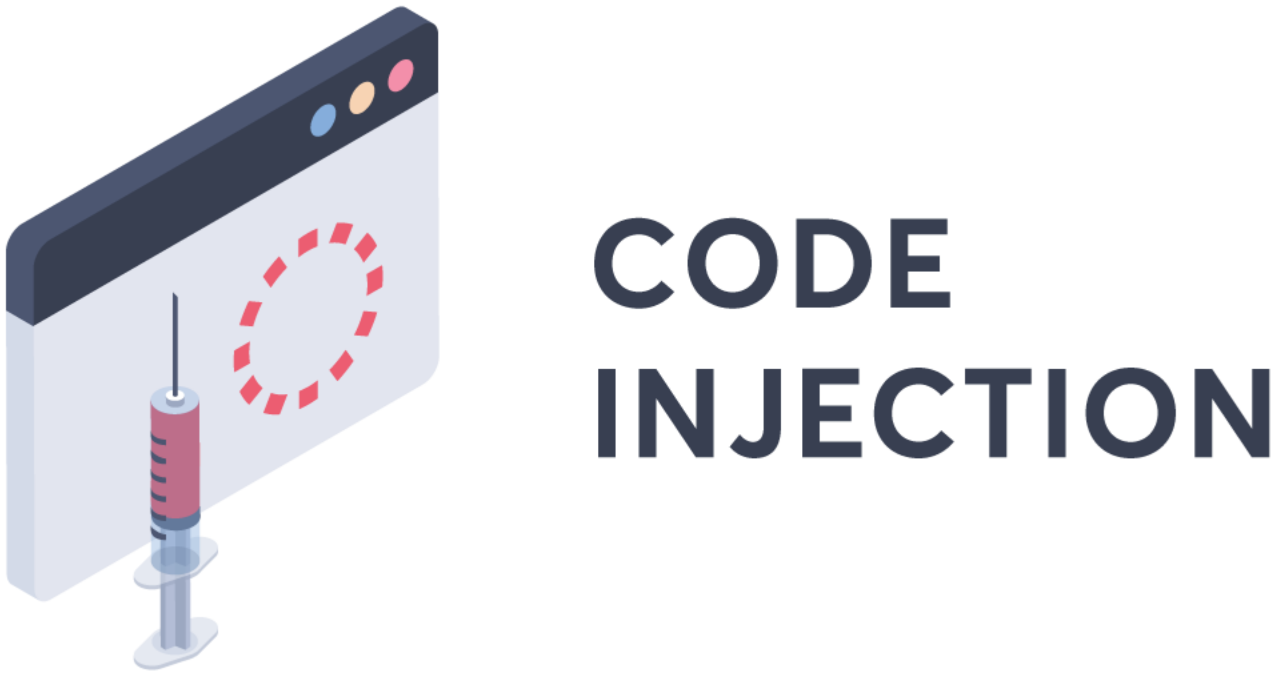


```
BOOL OnExceptionDebugEvent(LPDEBUG_EVENT pde)
{
    CONTEXT ctx;
    PBYTE lpBuffer = NULL;
    DWORD dwNumOfBytesToWrite, dwAddrOfBuffer, i;
    PEXCEPTION_RECORD per = &pde->u.Exception.ExceptionRecord;

    // In case of a BreakPoint exception (INT 3)
    if( EXCEPTION_BREAKPOINT == per->ExceptionCode )
    {
        // If the BP address is WriteFile()
        if( g_pfWriteFile == per->ExceptionAddress )
        {
            // #1. Unhook
            //   Restore the part overwritten with 0xCC to the original byte
            WriteProcessMemory(g_cpdi.hProcess, g_pfWriteFile,
                               &g_chOrgByte, sizeof(BYTE), NULL);

            // #2. Get Thread Context
            ctx.ContextFlags = CONTEXT_CONTROL;
            GetThreadContext(g_cpdi.hThread, &ctx);

            // #3. Get the values of param 2, 3 of WriteFile()
            //   The function's parameters exist on the process's stack
            //   param 2: ESP + 0x8
            //   param 3: ESP + 0xC
            ReadProcessMemory(g_cpdi.hProcess, (LPVOID)(ctx.Esp + 0x8),
                              &dwAddrOfBuffer, sizeof(DWORD), NULL);
            ReadProcessMemory(g_cpdi.hProcess, (LPVOID)(ctx.Esp + 0xC),
                              &dwNumOfBytesToWrite, sizeof(DWORD), NULL);

            // #4. Allocate a temporary buffer
            lpBuffer = (PBYTE)malloc(dwNumOfBytesToWrite+1);
            memset(lpBuffer, 0, dwNumOfBytesToWrite+1);

            // #5. Copy the WriteFile() buffer to the temporary buffer
            ReadProcessMemory(g_cpdi.hProcess, (LPVOID)dwAddrOfBuffer,
                              lpBuffer, dwNumOfBytesToWrite, NULL);
            printf("\n### original string ###\n%s\n", lpBuffer);
```



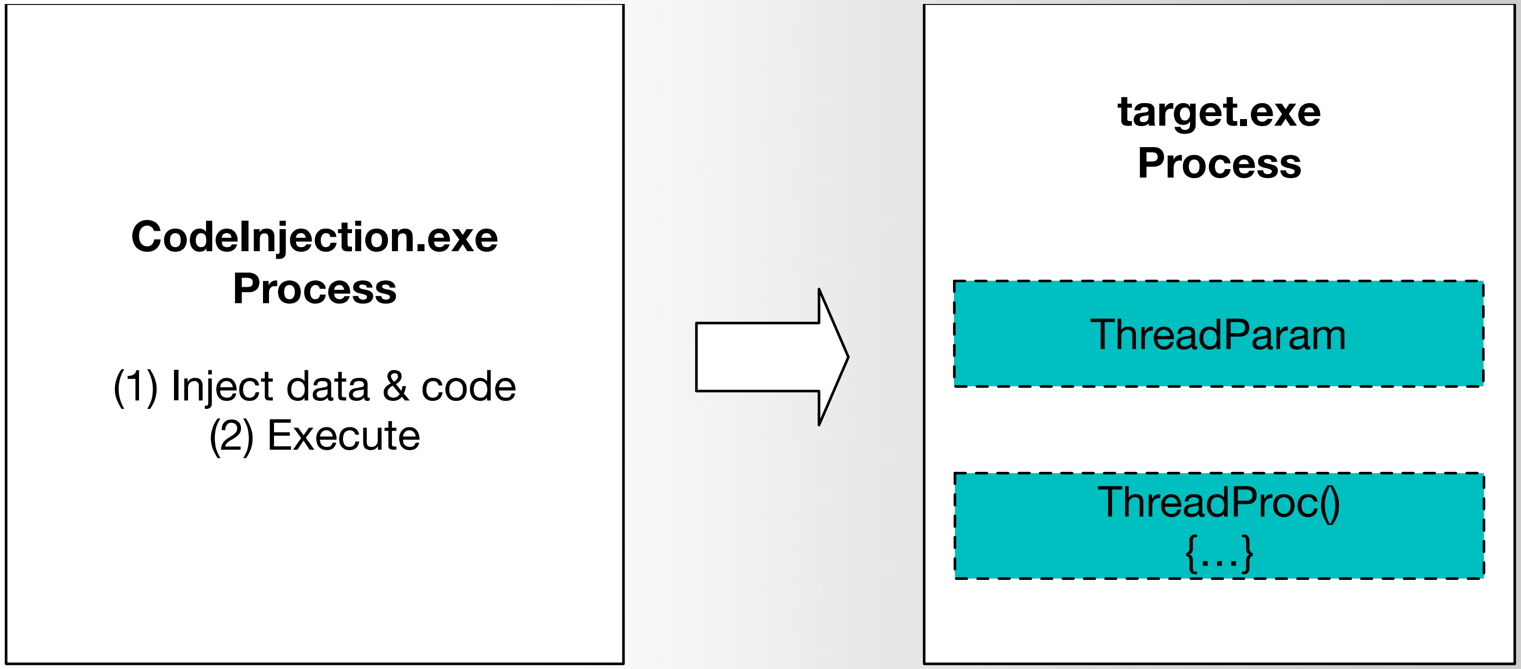
```
    // #5. Copy the WriteFile() buffer to the temporary buffer
    ReadProcessMemory(g_cpdi.hProcess, (LPVOID)dwAddrOfBuffer,
                      lpBuffer, dwNumOfBytesToWrite, NULL);
    printf("\n### original string ###\n%s\n", lpBuffer);

    // #6. Convert lowercase to uppercase
    for( i = 0; i < dwNumOfBytesToWrite; i++ )
    {
        if( 0x61 <= lpBuffer[i] && lpBuffer[i] <= 0x7A )
            lpBuffer[i] -= 0x20;
    }

    printf("\n### converted string ###\n%s\n", lpBuffer);

    // #7. Copy the converted buffer back to the WriteFile() buffer
    WriteProcessMemory(g_cpdi.hProcess, (LPVOID)dwAddrOfBuffer,
                       lpBuffer, dwNumOfBytesToWrite, NULL);

    // #8. Release the temporary buffer
    free(lpBuffer);

    // #9. Change the Thread Context's EIP to the start of WriteFile()
    //   (currently passed by WriteFile() + 1)
    ctx.Eip = (DWORD)g_pfWriteFile;
    SetThreadContext(g_cpdi.hThread, &ctx);

    // #10. Resume the debuggee process
    ContinueDebugEvent(pde->dwProcessId, pde->dwThreadId, DBG_CONTINUE);
    Sleep(0);

    // #11. API Hook
    WriteProcessMemory(g_cpdi.hProcess, g_pfWriteFile,
                       &g_chINT3, sizeof(BYTE), NULL);

    return TRUE;
}
```

Code Injection



Code injection is the term used to describe attacks that inject code into an application. That injected code is then interpreted by the application.

Code Injection (thread injection)



code → injected by ThreadProc()

data → injected as ThreadParam

Q & A