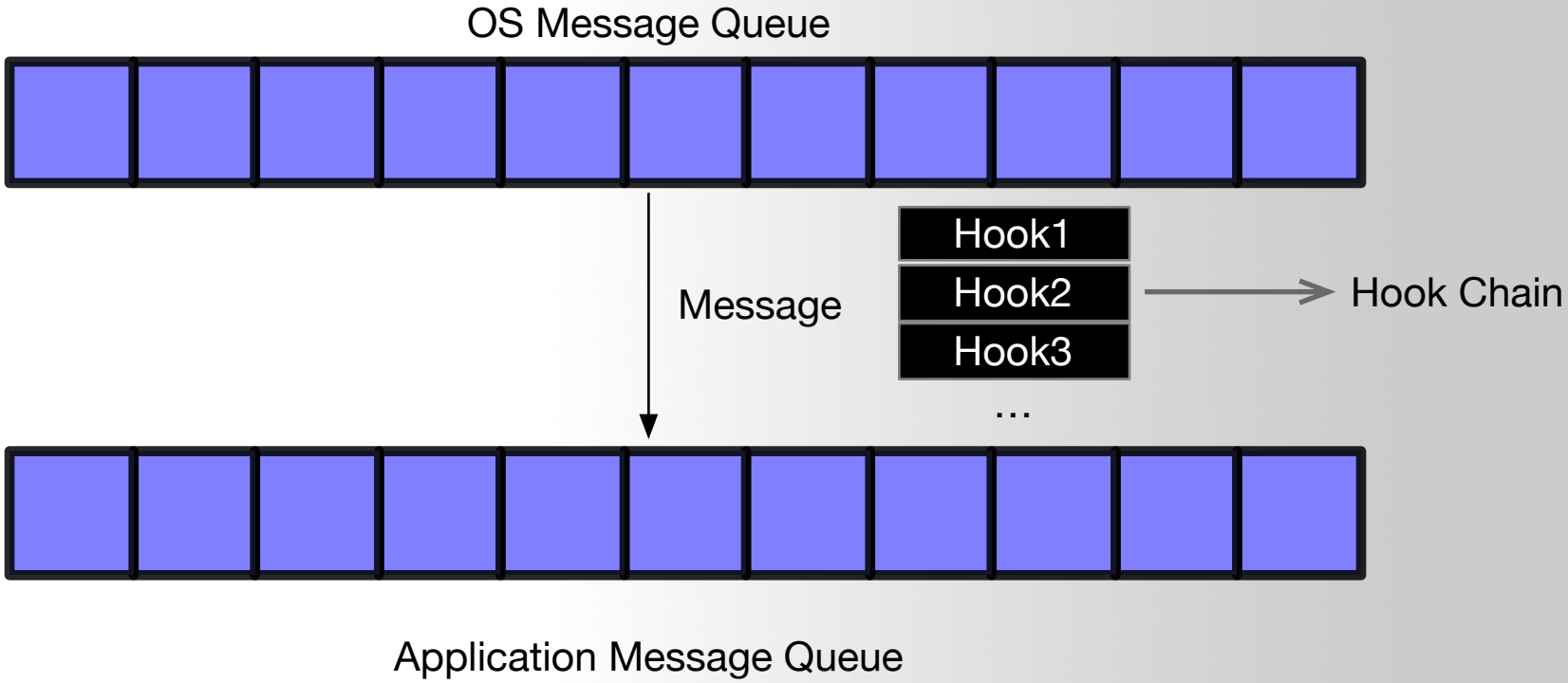
Class 10

CSC 471 Modern Malware Analysis
API Hooks (2) & Code Injection

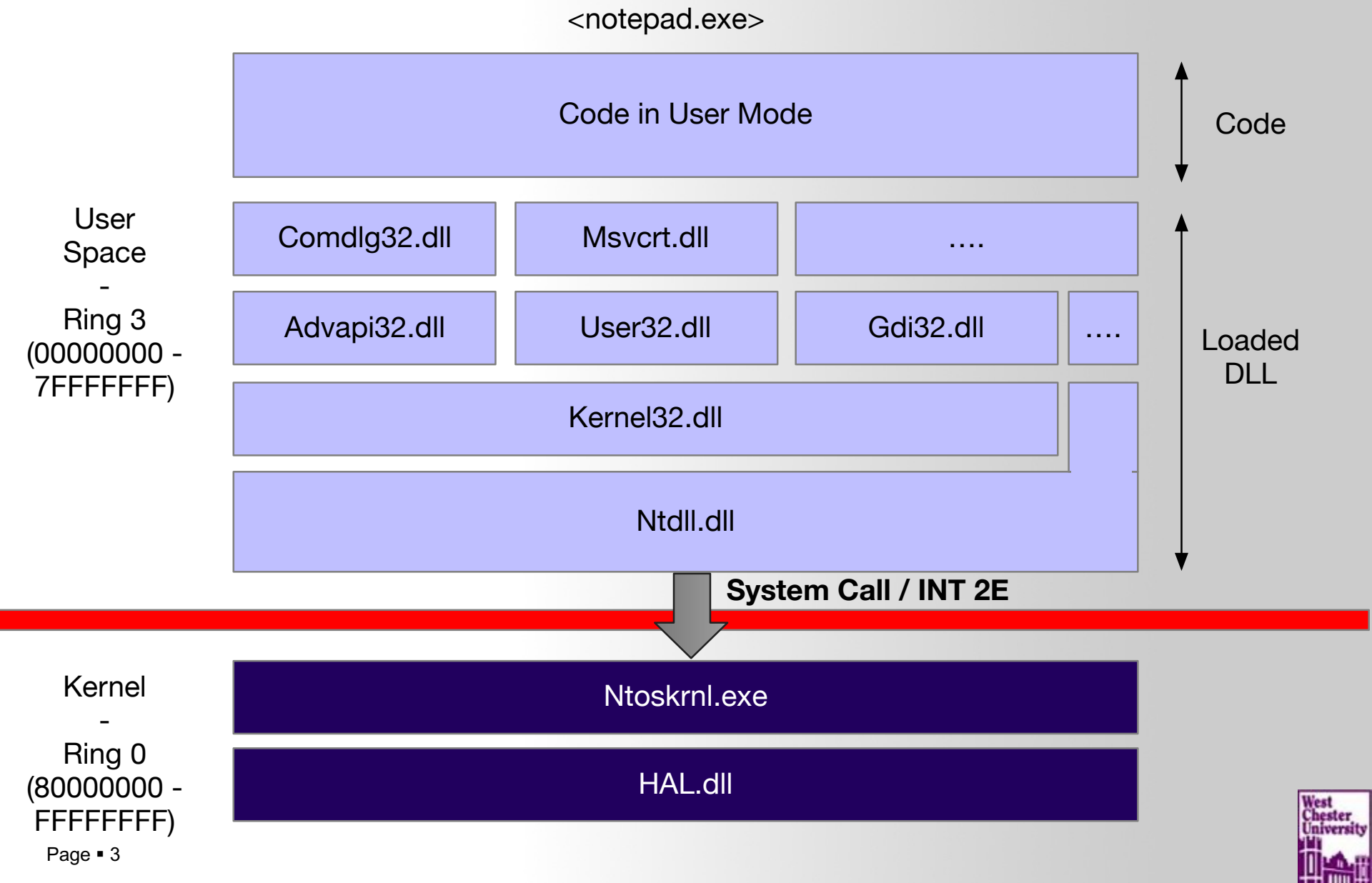
Si Chen (schen@wcupa.edu)

Review – Hook

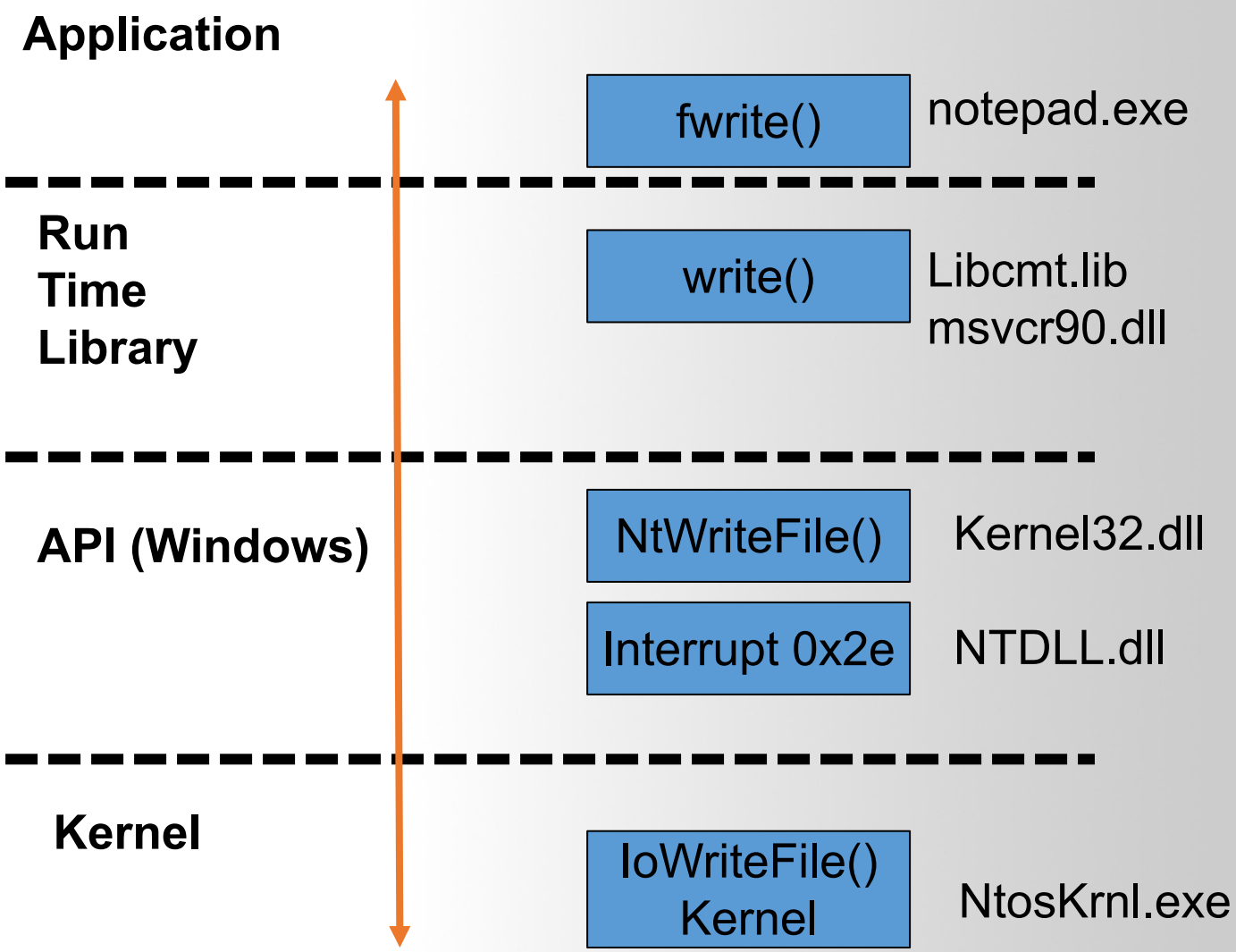
- A hook is a point in the system message-handling mechanism where an application can **install a subroutine** to monitor the message traffic in the system and process certain types of messages before they reach the target window procedure.



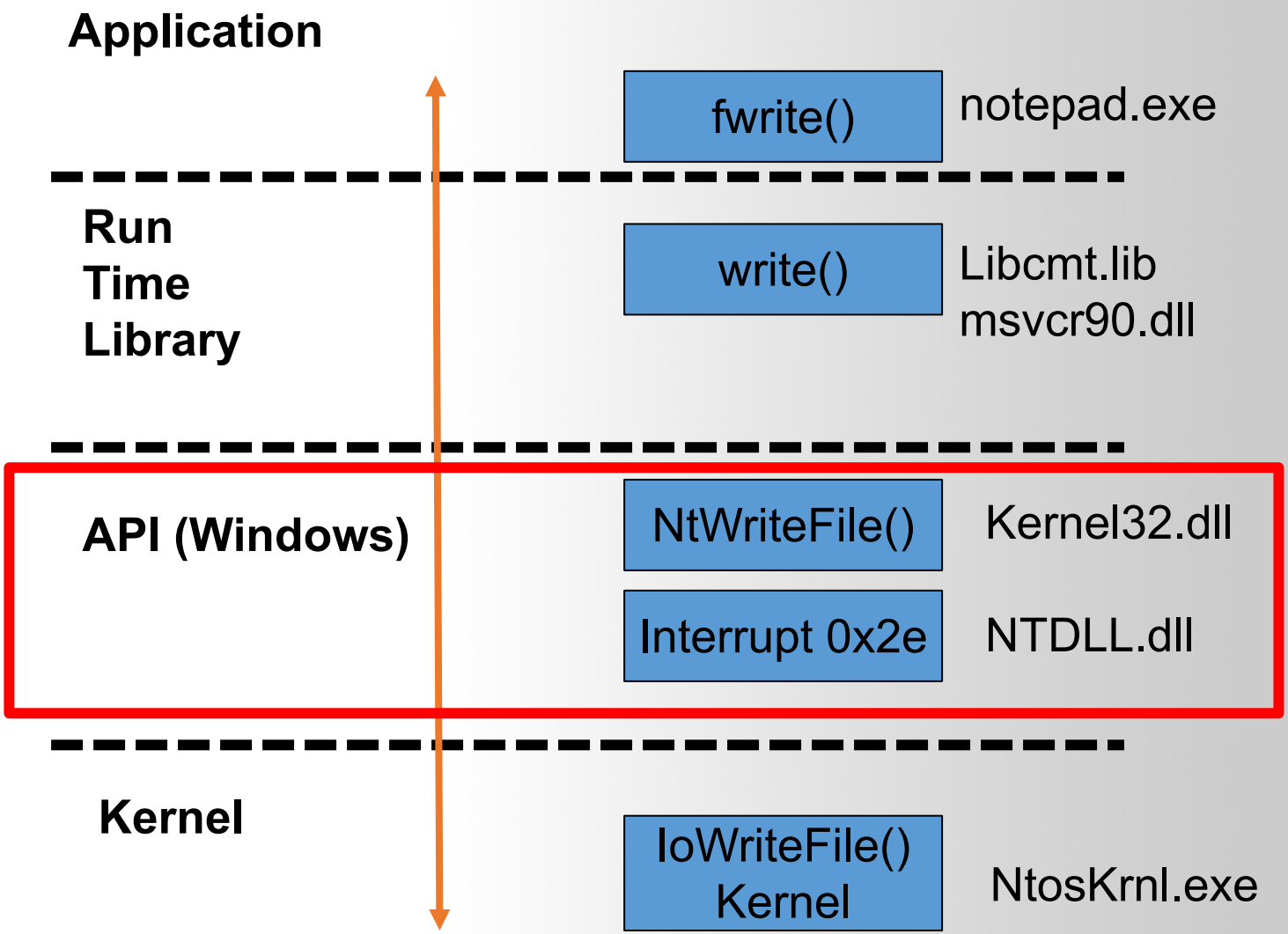
User Mode and Kernel



Write a file in Notepad



API Hook


Application
fwrite()
notepad.exe
Run
Time
Library
write()
Libcmt.lib
msvcr90.dll
API (Windows)
NtWriteFile()
Kernel32.dll
Interrupt 0x2e
NTDLL.dll
Kernel
IoWriteFile()
Kernel
NtosKrnl.exe

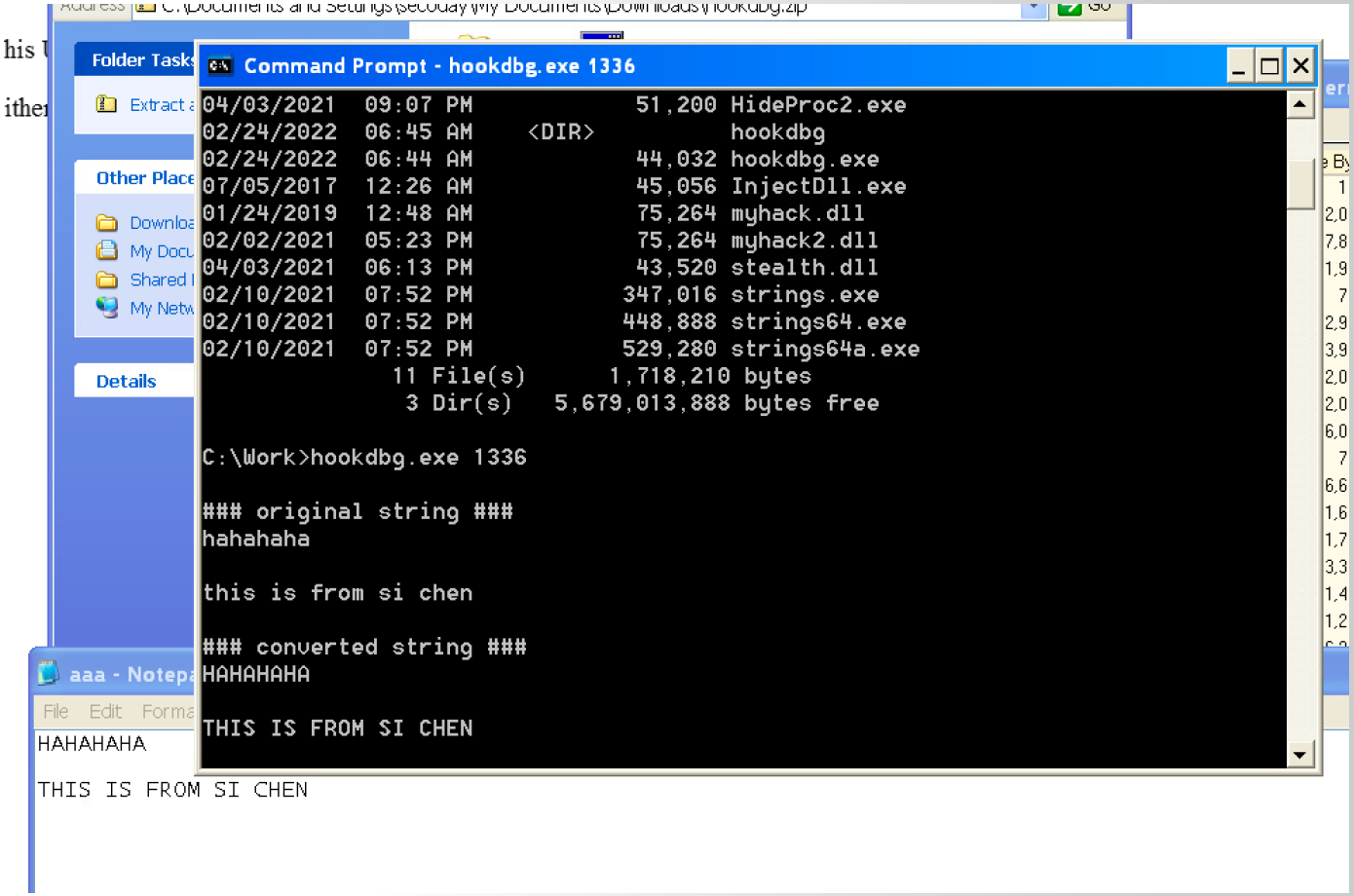
API Hook Tech Map

<table>
<tr><th>Method</th><th>Target</th><th>Location</th><th colspan="2">Tech</th><th>API</th></tr>
<tr><td rowspan="4">Dynamic</td><td rowspan="4">Process/Memory

00000000
- 7FFFFFFF</td><td rowspan="4">1) IAT
2) Code
3) EAT</td><td colspan="2">Interactive Debug</td><td>DebugActiveProcess
GetThreadContext
SetThreadContext</td></tr>
<tr><td rowspan="3">Standalone Injection</td><td>Independent Code</td><td>CreateRemoteThread</td></tr>
<tr><td rowspan="2">Dll File</td><td>Resistry (AppInit_DLLs)
BHO (IE only)</td></tr>
<tr><td>SetWindowsHookEx
CreateRemoteThread</td></tr>
</table>

Hookdbg.exe

- API hook for Notepad WriteFile() function



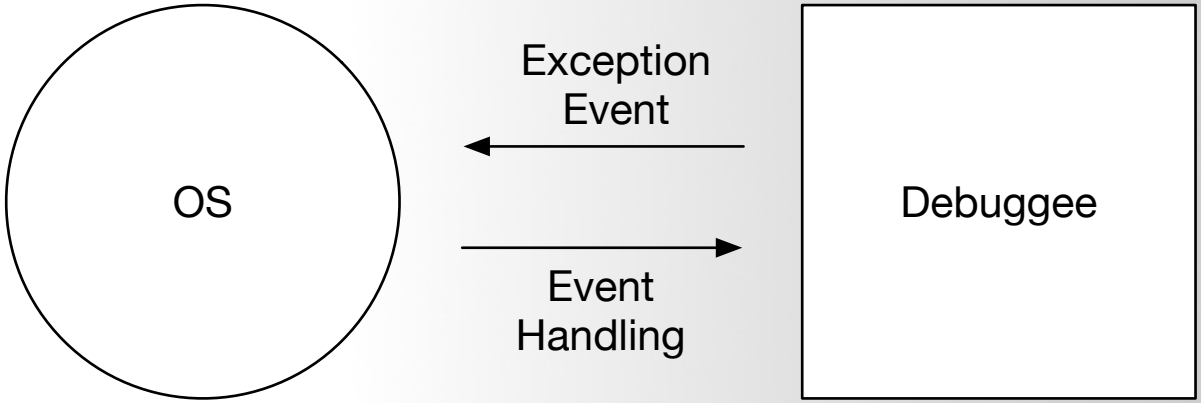
WriteFile() Definition from MSDN

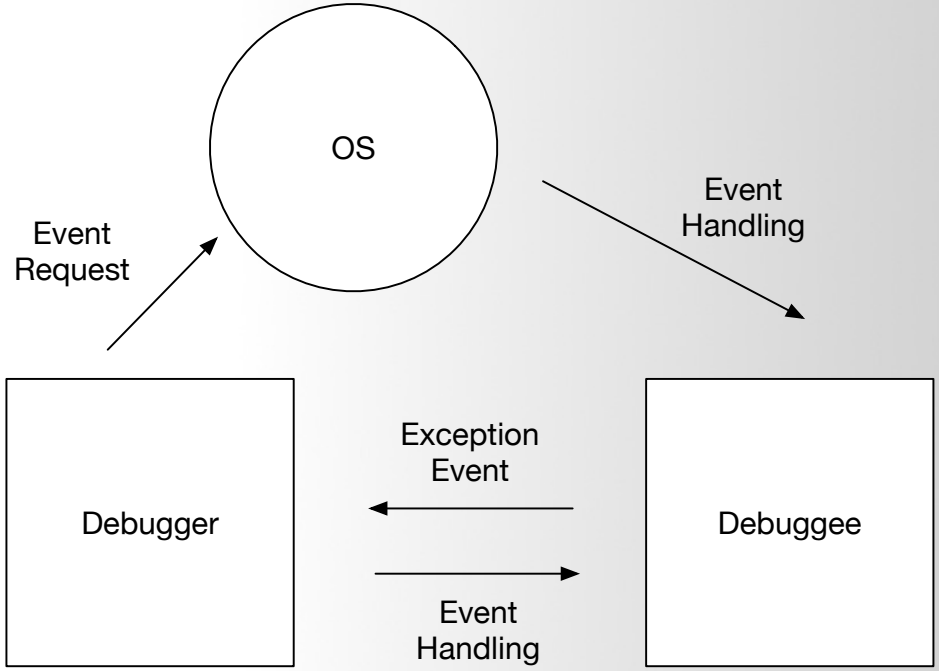
Syntax

C++ Copy

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

How Debugger Works





`ExceptionCode`

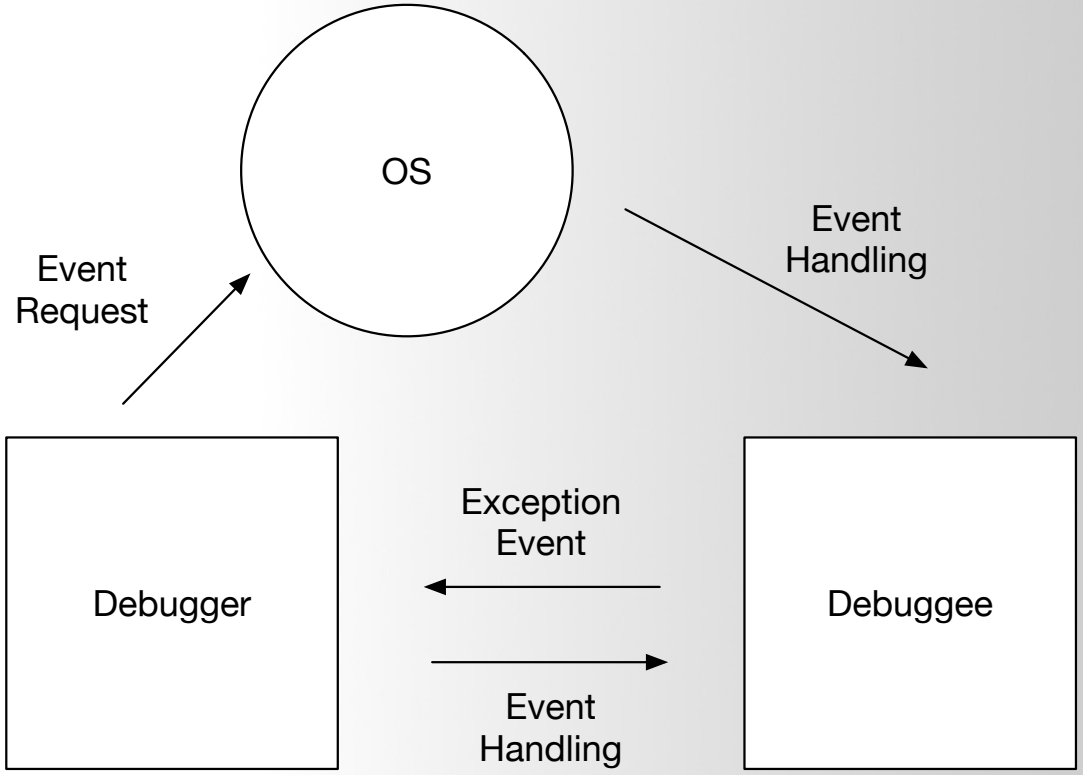
The reason the exception occurred. This is the code generated by a hardware exception, or the code specified in the RaiseException function for a software-generated exception. The following tables describes the exception codes that are likely to occur due to common programming errors.

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.

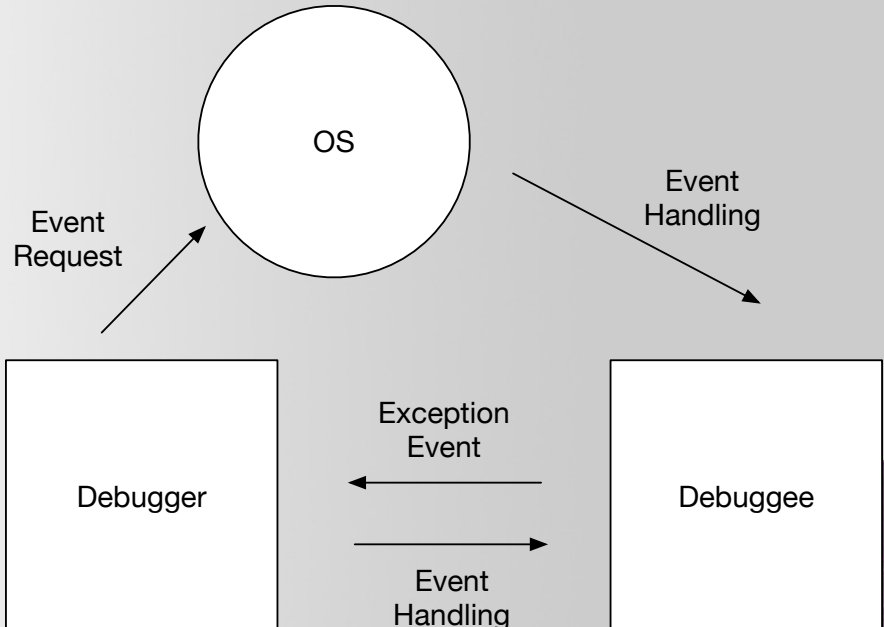
https://docs.microsoft.com/en-us/windows/win32/api/winnt/ns-winnt-exception_record

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.



```
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);
```

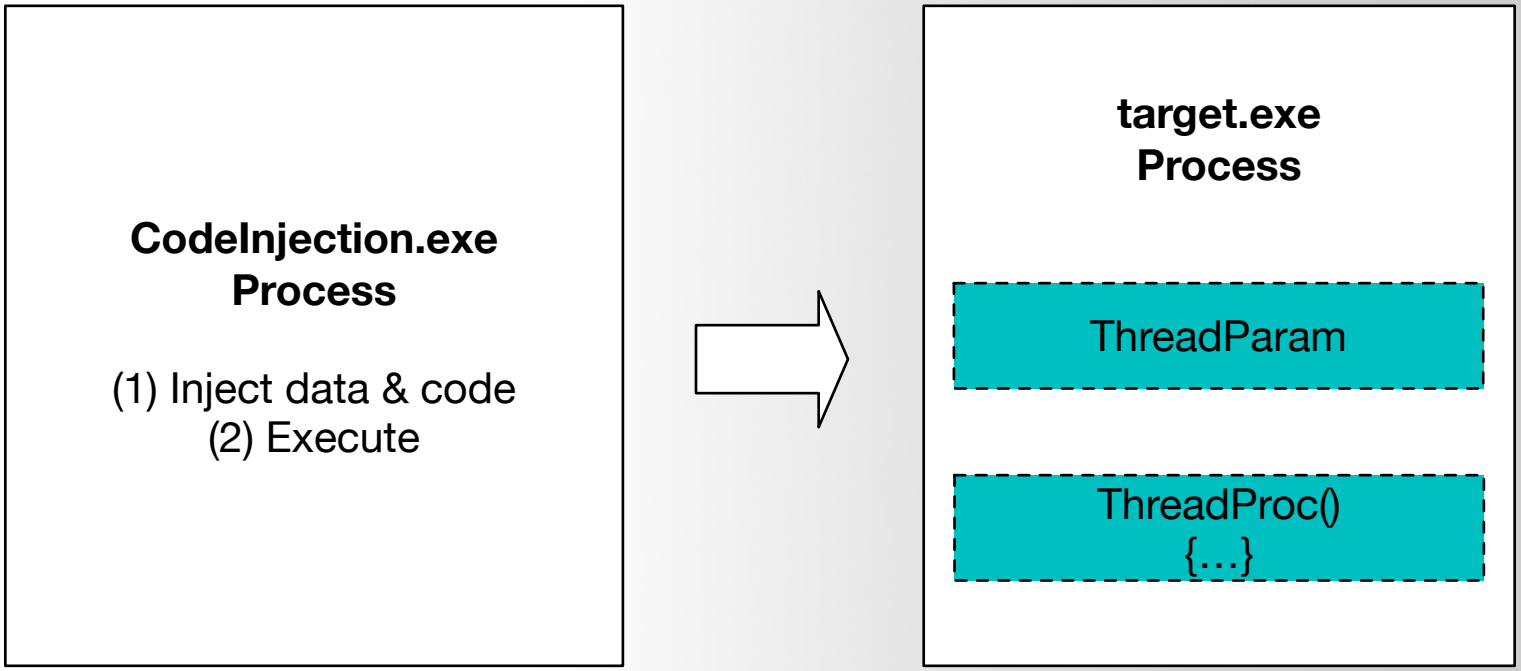
```
    // #5. Copy the WriteFile() buffer to the temporary buffer
    ReadProcessMemory(g_cpdi.hProcess, (LPVOID)dwAddrOfBuffer,
                      lpBuffer, dwNumOfBytesToWrite, NULL);
    printf("\n### original string ###\n%s\n", lpBuffer);

    // #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

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