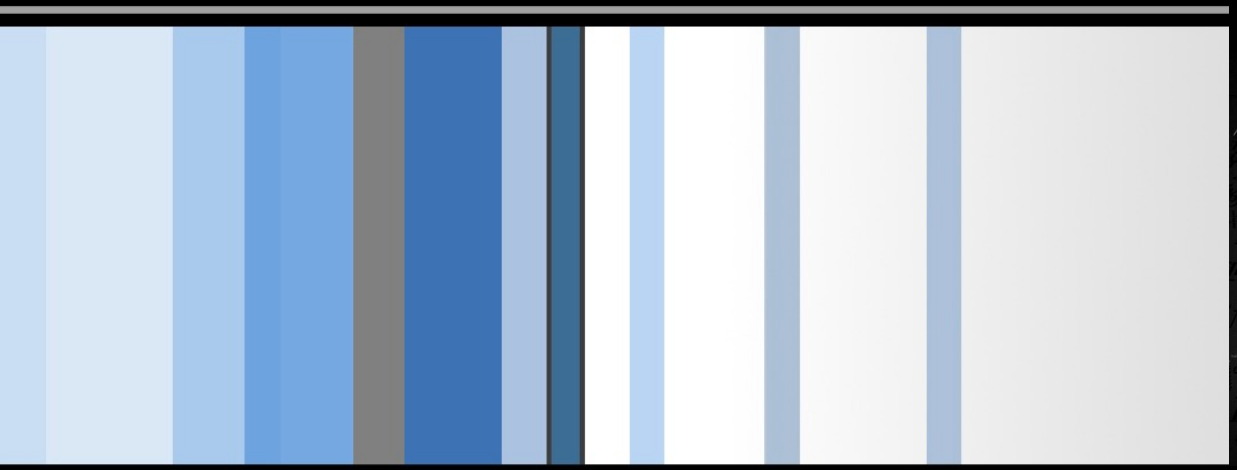
Class 11

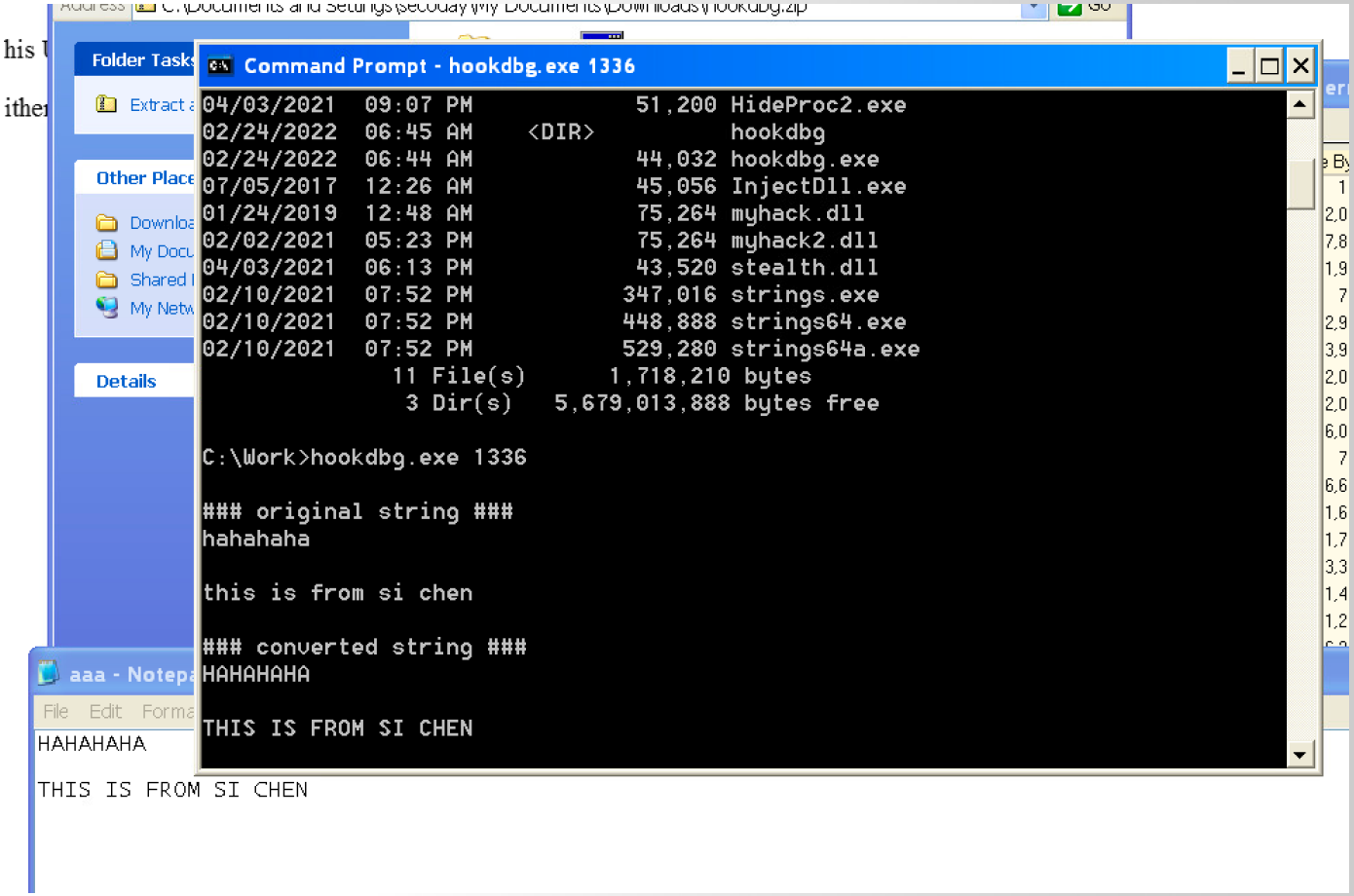
CSC 471 Modern Malware Analysis
Code Injection

Si Chen (schen@wcupa.edu)



Hookdbg.exe

- API hook for Notepad WriteFile() function



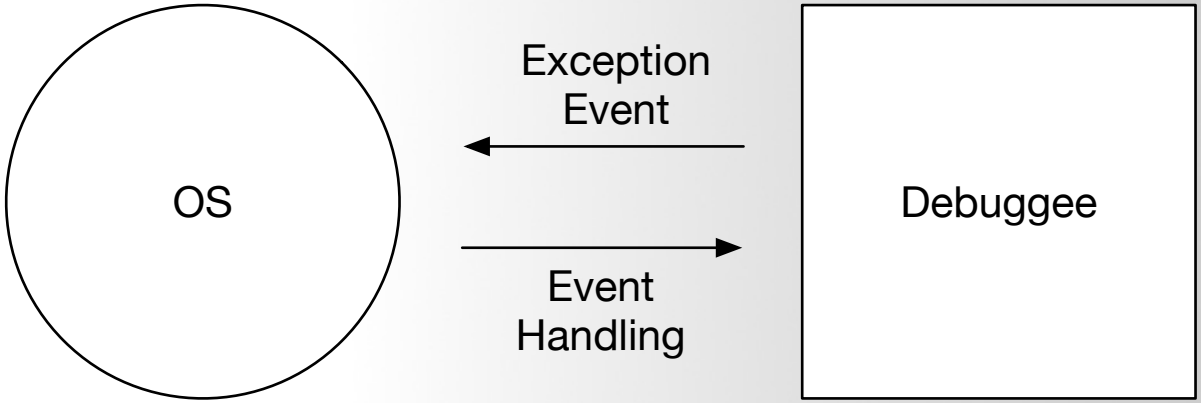
WriteFile() Definition from MSDN

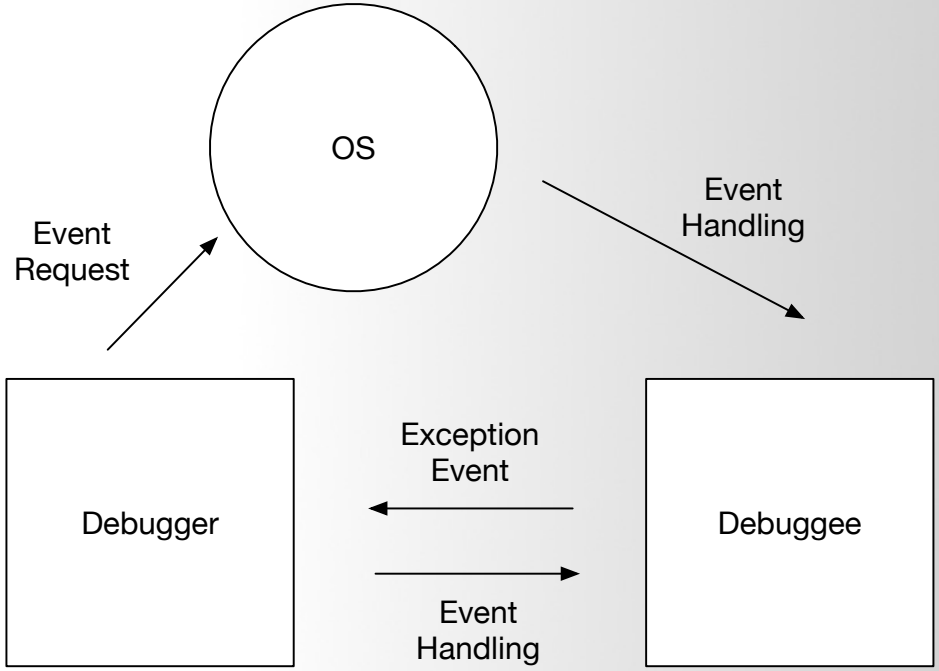
Syntax

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`

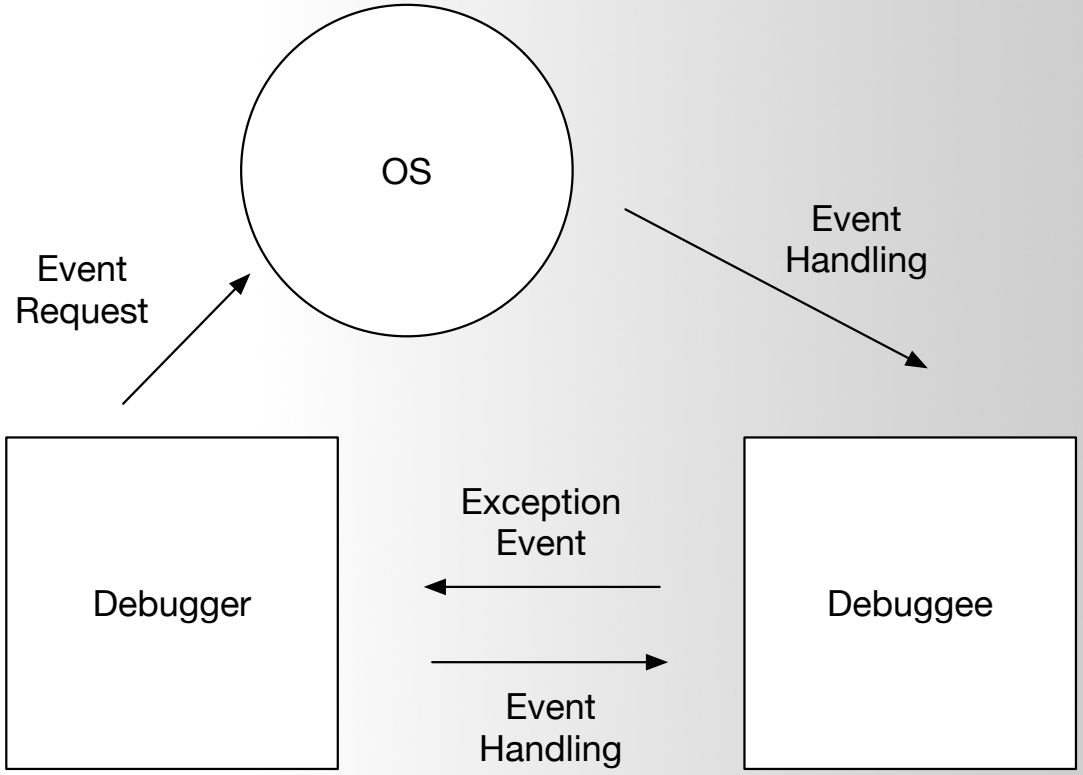
The reason the exception occurred. This is the code generated by a hardware exception, or the code specified in the RaiseException function for a software-generated exception. The following tables describes the exception codes that are likely to occur due to common programming errors.

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.

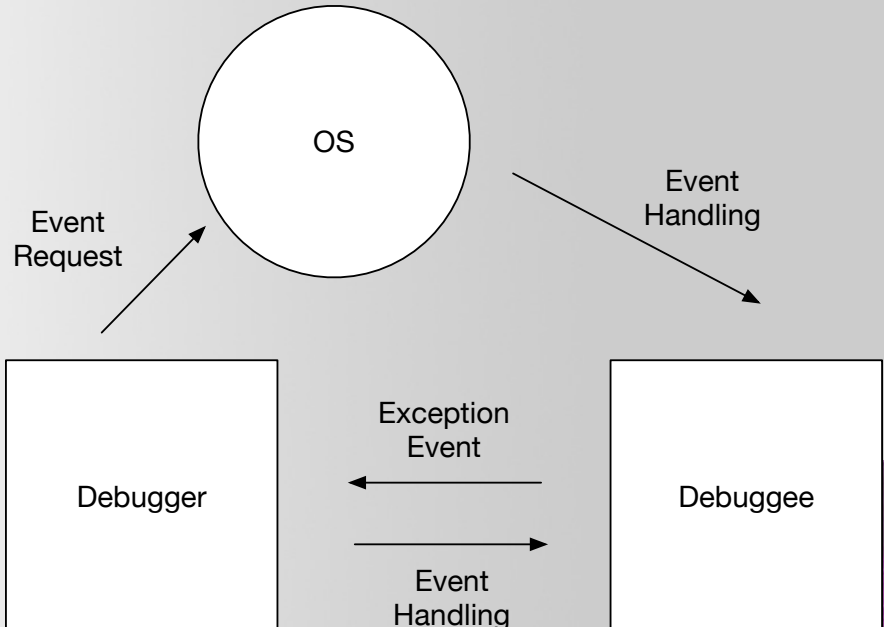
https://docs.microsoft.com/en-us/windows/win32/api/winnt/ns-winnt-exception_record

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);
```

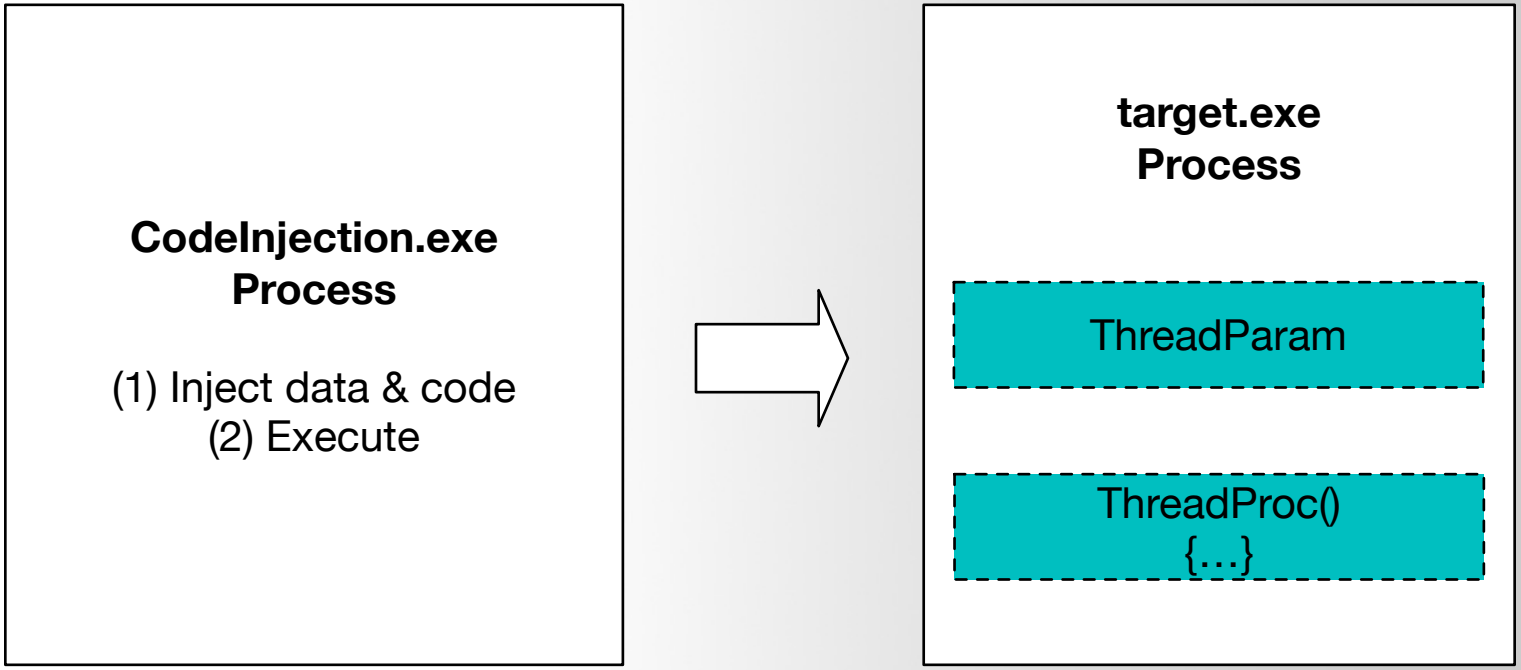
```
    // #5. Copy the WriteFile() buffer to the temporary buffer
    ReadProcessMemory(g_cpdi.hProcess, (LPVOID)dwAddrOfBuffer,
                      lpBuffer, dwNumOfBytesToWrite, NULL);
    printf("\n### original string ###\n%s\n", lpBuffer);

    // #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)



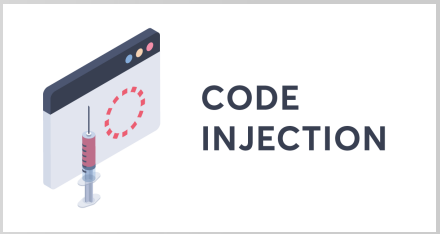
code → injected by ThreadProc()

data → injected as ThreadParam

Why Code Injection

- 1. **Use less memory** → you don't need to compile it as DLL
- 2. **Hard to detect** → DLL injection can easily be spotted, code injection is very sneaky.

- In short:
  - **DLL injection** is for huge code base and complex logic.
  - **Code injection** is for small code base with simple logic.



DLL Injection V.S. Code Injection

```
DWORD WINAPI ThreadProc(LPVOID lParam)
{
    MessageBoxA(NULL, "cs.wcupa.edu", "Dr. Chen", MB_OK);

    return 0;
}
```

Pop up a Windows message box

How to use DLL Injection to injection the code?

myhack.cpp

myhack.cpp 〉 No Selection

```
1  #include "windows.h"
2  #include "tchar.h"
3
4  #pragma comment(lib, "urlmon.lib")
5
6  #define DEF_URL         (L"http://www.naver.com/index.html")
7  #define DEF_FILE_NAME   (L"index.html")
8
9  HMODULE g_hMod = NULL;
10
11 DWORD WINAPI ThreadProc(LPVOID lParam)
12 {
13     TCHAR szPath[_MAX_PATH] = {0,};
14
15     if( !GetModuleFileName( g_hMod, szPath, MAX_PATH ) )
16         return FALSE;
17
18     TCHAR *p = _tcsrchr( szPath, '\\' );
19     if( !p )
20         return FALSE;
21
22     _tcscpy_s(p+1, _MAX_PATH, DEF_FILE_NAME);
23
24     URLDownloadToFile(NULL, DEF_URL, szPath, 0, NULL);
25
26     return 0;
27 }
28
29 BOOL WINAPI DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved)
30 {
31     HANDLE hThread = NULL;
32
33     g_hMod = (HMODULE)hinstDLL;
34
35     switch( fdwReason )
36     {
37     case DLL_PROCESS_ATTACH :
38         OutputDebugString(L"<myhack.dll> Injection!!! -- CSC 497/583 -- Dr. Chen");
39         hThread = CreateThread(NULL, 0, ThreadProc, NULL, 0, NULL);
40         CloseHandle(hThread);
41         break;
42     }
43
44     return TRUE;
45 }
```

DLL Injection V.S. Code Injection

https://learn.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-messagebox

How to use DLL Injection to injection the code?

```
#include "windows.h"

DWORD WINAPI ThreadProc(LPVOID lParam)
{
    MessageBoxA(NULL, "cs.wcupa.edu", "Dr. Chen", MB_OK);

    return 0;
}

BOOL WINAPI DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved)
{
    switch( fdwReason )
    {
        case DLL_PROCESS_ATTACH :
            CreateThread(NULL, 0, ThreadProc, NULL, 0, NULL);
            break;
    }

    return TRUE;
}
```

Compile it as MsgBox.dll and inject it to the target process same as DLL injection lab!

DLL Injection (MsgBox.dll)

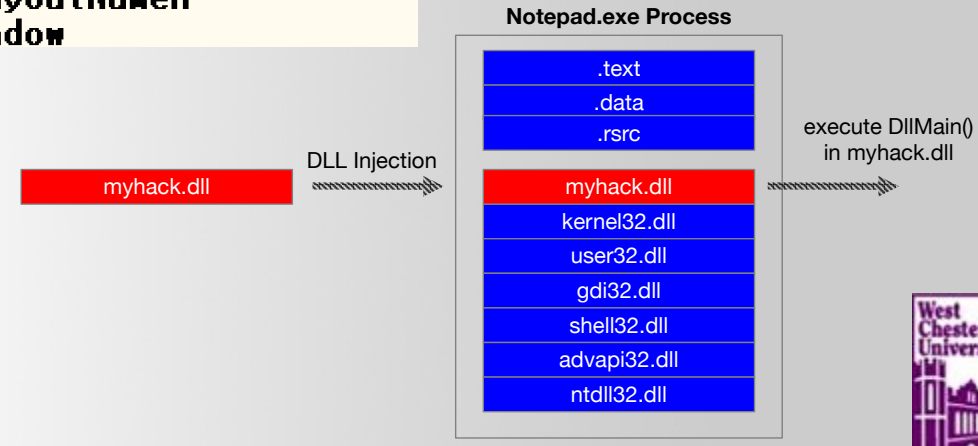
```
10001000  $  6A 00          PUSH 0                                  Type = MB_OK|MB_DEFBUTTON1|MB_APPLMODAL
10001002  .  68 D01C0110    PUSH OFFSET 10011CD0                    Caption = "Dr. Chen"
10001007  .  68 DC1C0110    PUSH OFFSET 10011CDC                    Text = "cs.wcupa.edu"
1000100C  .  6A 00          PUSH 0                                  hOwner = NULL
1000100E  .  FF15 0CD1001(  CALL DWORD PTR DS:[<&USER32.MessageBoxA] USER32.MessageBoxA
10001014  .  33C0           XOR EAX,EAX
10001016  L. C2 0400        RETN 4
```

Address	Hex dump	ASCII
10011CD0	44 72 2E 20 43 68 65 6E 00 00 00 00 63 73 2E 77	Dr. Chen□□□□cs.w
10011CE0	63 75 70 61 2E 65 64 75 00 00 00 00 00 00 00 00	cupa.edu□□□□□□□□
10011CF0	C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	À□□□□□□□□□□□□□□□

DLL Injection (MsgBox.dll)

```
10001000  $  6A 00          PUSH 0                                  Type = MB_OK|MB_DEFBUTTON1|MB_APPLMODAL
10001002  .  68 D01C0110    PUSH OFFSET 10011CD0                    Caption = "Dr. Chen"
10001007  .  68 DC1C0110    PUSH OFFSET 10011CDC                    Text = "cs.wcupa.edu"
1000100C  .  6A 00          PUSH 0                                  hOwner = NULL
1000100E  .  FF15 0CD1001(  CALL DWORD PTR DS:[<&USER32.MessageBoxA] USER32.MessageBoxA
10001014  .  33C0           XOR EAX,EAX
10001016  L. C2 0400        RETN 4
10001019     CC             INT3
1000101A     CC             INT3
1000101B     CC             INT3
[1000D10C]=7E4507EA (USER32.MessageBoxA)
```

```
7E45023C  .text   Export  OemKeyScan
7E45029E  .text   Export  MapVirtualKeyW
7E4502BB  .text   Export  OemToCharBuffW
7E4502F9  .text   Export  GetMenuCheckMarkDimensions
7E4507EA  .text   Export  MessageBoxA
7E450838  .text   Export  MessageBoxExW
7E45085C  .text   Export  MessageBoxExA
7E453497  .text   Export  CreateAcceleratorTableA
7E453631  .text   Export  GetKeyboardLayoutNameA
7E45370D  .text   Export  GetTaskmanWindow
```



Code Injection

You need to inject the code

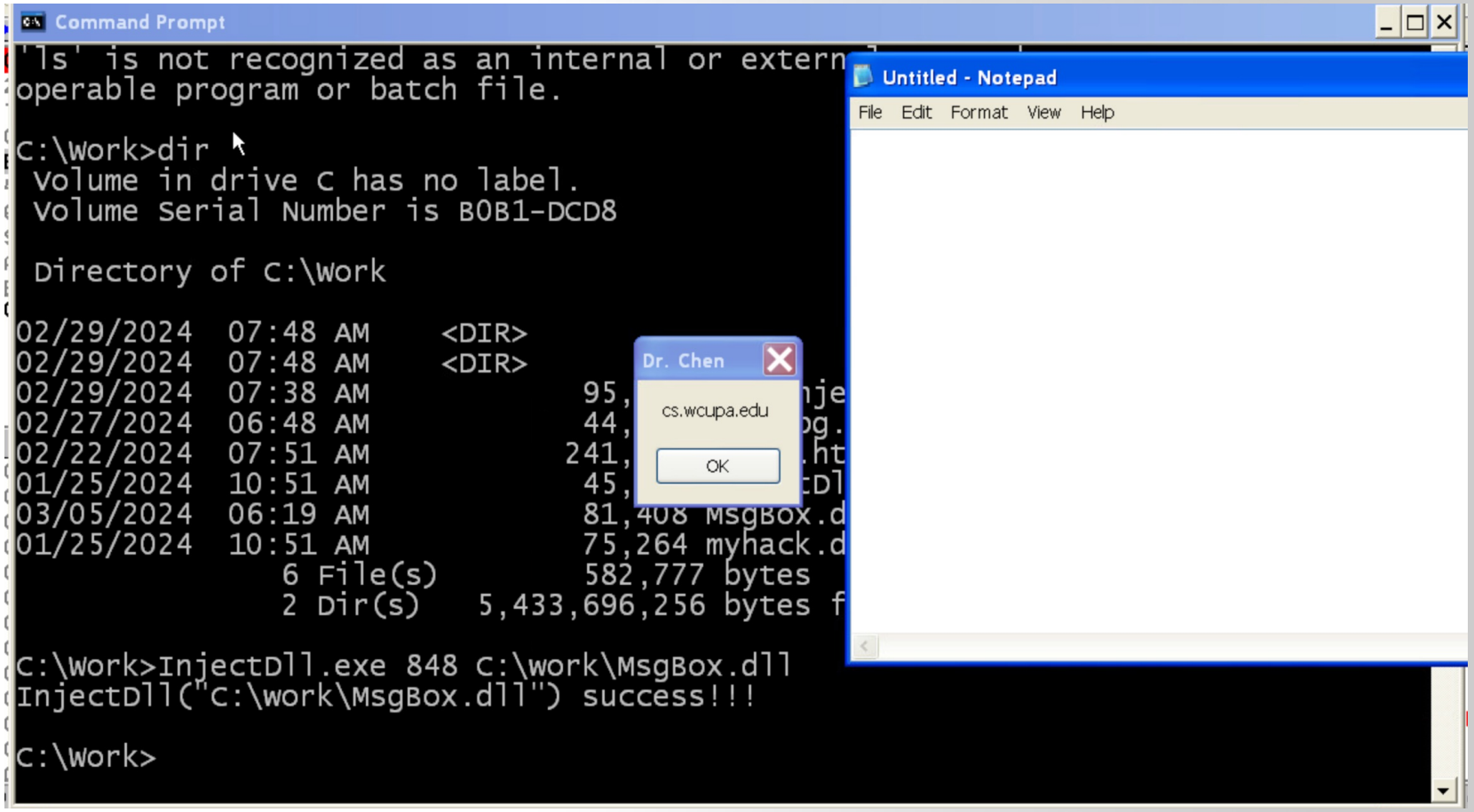
```
10001000  $  6A 00          PUSH 0                                  Type = MB_OK|MB_DEFBUTTON1|MB_APPLMODAL
10001002  .  68 D01C0110    PUSH OFFSET 10011CD0                    Caption = "Dr. Chen"
10001007  .  68 DC1C0110    PUSH OFFSET 10011CDC                    Text = "cs.wcupa.edu"
1000100C  .  6A 00          PUSH 0                                  hOwner = NULL
1000100E  .  FF15 0CD1001(  CALL DWORD PTR DS:[<&USER32.MessageBoxA  USER32.MessageBoxA
10001014  .  33C0           XOR EAX,EAX
10001016  L. C2 0400        RETN 4
```

And the data:

Address	Hex dump	ASCII
10011CD0	44 72 2E 20 43 68 65 6E 00 00 00 00 63 73 2E 77	Dr. Chen□□□□cs.w
10011CE0	63 75 70 61 2E 65 64 75 00 00 00 00 00 00 00 00	cupa.edu□□□□□□□□
10011CF0	C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	À□□□□□□□□□□□□□□□

```
7E45023C  .text  Export  OemKeyScan
7E45029E  .text  Export  MapVirtualKeyW
7E4502BB  .text  Export  OemToCharBuffW
7E4502F9  .text  Export  GetMenuCheckMarkDimensions
7E4507EA  .text  Export  MessageBoxA
7E450838  .text  Export  MessageBoxExW
7E45085C  .text  Export  MessageBoxExA
7E453497  .text  Export  CreateAcceleratorTableA
7E453631  .text  Export  GetKeyboardLayoutNameA
7E45370D  .text  Export  GetTaskmanWindow
```

Code Injection Example (CodeInjection.exe)



CodeInjection.cpp – main()

```
int main(int argc, char *argv[])
{
    DWORD dwPID     = 0;

    if( argc != 2 )
    {
        printf("\n USAGE  : %s <pid>\n", argv[0]);
        return 1;
    }

    // change privilege
    if( !SetPrivilege(SE_DEBUG_NAME, TRUE) )
        return 1;

    // code injection
    dwPID = (DWORD)atol(argv[1]);
    InjectCode(dwPID);

    return 0;
}
```

CodeInjection.cpp – ThreadProc()

```
// Define a structure to hold function pointers and strings for dynamic loading and execution.
// The structure is used to pass the function pointers and strings to the remote process.
typedef struct _THREAD_PARAM
{
    FARPROC pFunc[2];               // LoadLibraryA(), GetProcAddress()
    char    szBuf[4][128];          // "user32.dll", "MessageBoxA", "cs.wcupa.edu", "Dr. Chen"
} THREAD_PARAM, *PTHREAD_PARAM;

// Function pointer type definitions for dynamic loading.
// The function pointers are used to call the LoadLibraryA(), GetProcAddress(), and MessageBoxA() functions.
typedef HMODULE (WINAPI *PFLOADLIBRARYA)
(
    LPCSTR lpLibFileName
);

typedef FARPROC (WINAPI *PFGETPROCADDRESS)
(
    HMODULE hModule,
    LPCSTR lpProcName
);
```

CodeInjection.cpp – ThreadProc()

```
32
33  DWORD WINAPI ThreadProc(LPVOID lParam)
34  {
35      PTHREAD_PARAM   pParam      = (PTHREAD_PARAM)lParam;
36      HMODULE         hMod        = NULL;
37      FARPROC         pFunc       = NULL;
38
39      // LoadLibrary()
40      hMod = ((PFLOADLIBRARYA)pParam->pFunc[0])(pParam->szBuf[0]);    // "user32.dll"
41      if( !hMod )
42          return 1;
43
44      // GetProcAddress()
45      pFunc = (FARPROC)((PFGETPROCADDRESS)pParam->pFunc[1])(hMod, pParam->szBuf[1]);  // "MessageBoxA"
46      if( !pFunc )
47          return 1;
48
49      // MessageBoxA()
50      ((PFMESSAGEBOXA)pFunc)(NULL, pParam->szBuf[2], pParam->szBuf[3], MB_OK);
51
52      return 0;
53  }
54
```

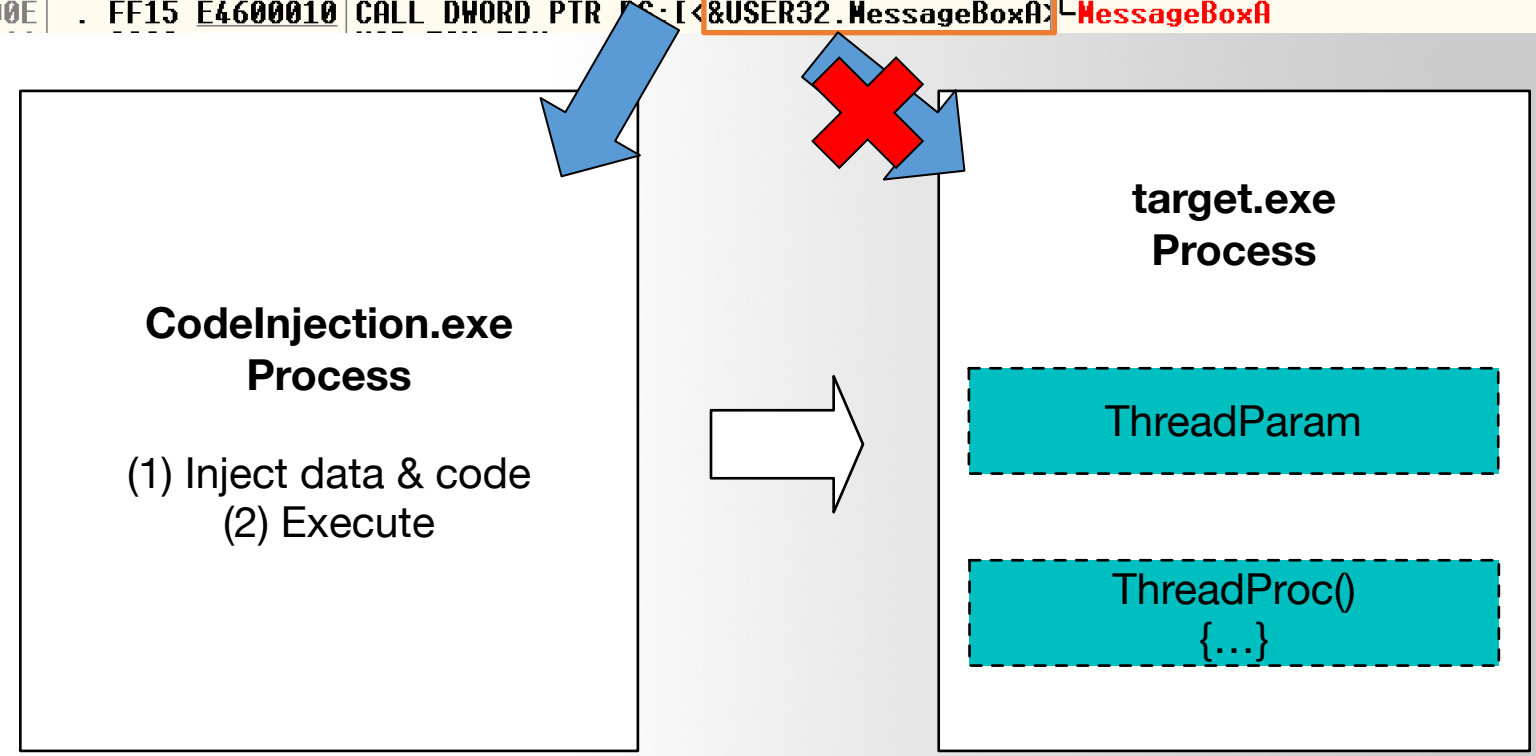
hMod = LoadLibraryA(“user32.dll”);

pFunc = GetProcAddress(hMod, “MessageBoxA”);

pFunc(NULL, “www.reversecore.com”, “ReverseCore”, MB_OK);

Cannot use the following address for Code Injection

```
10001000  . 6A 00          PUSH 0                                  Style = MB_OK|MB_APPLMODAL
10001002  . 68 1C780010    PUSH MsgBox.1000781C                    Title = "ReverseCore"
10001007  . 68 28780010    PUSH MsgBox.10007828                    Text = "www.reversecore.com"
1000100C  . 6A 00          PUSH 0                                  hOwner = NULL
1000100E  . FF15 E4600010  CALL DWORD PTR DS:[<&USER32.MessageBoxA> MessageBoxA
```



You cannot use the address provided for code injection.

Because MessageBoxA and Caption and "Text" are not loaded, these addresses cannot be used for code injection.

```
00401000  . 55              PUSH EBP
00401001  . 8BEC            MOV EBP,ESP
00401003  . 56              PUSH ESI
00401004  . 8B75 08         MOV ESI,DWORD PTR SS:[EBP+8]
00401007  . 8B0E            MOV ECX,DWORD PTR DS:[ESI]
00401009  . 8D46 08         LEA EAX,DWORD PTR DS:[ESI+8]
0040100C  . 50              PUSH EAX
0040100D  . FFD1            CALL ECX
0040100F  . 85C0            TEST EAX,EAX
00401011  . 75 0A           JNZ SHORT CodeInje.0040101D
00401013  > B8 01000000     MOV EAX,1
00401018  . 5E              POP ESI
00401019  . 5D              POP EBP
0040101A  . C2 0400         RETN 4
0040101D  > 8D96 88000000   LEA EDX,DWORD PTR DS:[ESI+88]
00401023  . 52              PUSH EDX
00401024  . 50              PUSH EAX
00401025  . 8B46 04         MOV EAX,DWORD PTR DS:[ESI+4]
00401028  . FFD0            CALL EAX
0040102A  . 85C0            TEST EAX,EAX
0040102C  . 74 E5           JE SHORT CodeInje.00401013
0040102E  . 6A 00           PUSH 0
00401030  . 8D8E 88010000   LEA ECX,DWORD PTR DS:[ESI+188]
00401036  . 51              PUSH ECX
00401037  . 81C6 08010000   ADD ESI,108
0040103D  . 56              PUSH ESI
0040103E  . 6A 00           PUSH 0
00401040  . FFD0            CALL EAX
00401042  . 33C0            XOR EAX,EAX
```

```
10001000  $  6A 00          PUSH 0                                  Type = MB_OK|MB_DEFBUTTON1|MB_APPLMODAL
10001002  .  68 D01C0110    PUSH OFFSET 10011CD0                    Caption = "Dr. Chen"
10001007  .  68 DC1C0110    PUSH OFFSET 10011CDC                    Text = "cs.wcupa.edu"
1000100C  .  6A 00          PUSH 0                                  hOwner = NULL
1000100E  .  FF15 0CD1001(  CALL DWORD PTR DS:[<&USER32.MessageBoxA> USER32.MessageBoxA
10001014  .  33C0           XOR EAX,EAX
10001016  L. C2 0400        RETN 4
10001019     CC             INT3
1000101A     CC             INT3
1000101B     CC             INT3
[1000D10C]=7E4507EA (USER32.MessageBoxA)
```

CodeInjection.cpp – InjectCode()

```
// Main injection function: performs process and thread injection into a target process.
BOOL InjectCode(DWORD dwPID)
{
    // Prepare the THREAD_PARAM structure with necessary function pointers and strings.
    // Open the target process with necessary privileges.
    // Allocate memory in the target process for THREAD_PARAM.
    // Write THREAD_PARAM to the allocated memory in the target process.
    // Allocate memory for the ThreadProc function in the target process and set it to executable.
    // Write the ThreadProc function to the allocated memory in the target process.
    // Create a remote thread in the target process that starts at the ThreadProc function.
    // Wait for the thread to complete execution.
    // Close handles and return TRUE on successful injection.

    HMODULE         hMod            = NULL;
    THREAD_PARAM    param           = {0,};
    HANDLE          hProcess        = NULL;
    HANDLE          hThread         = NULL;
    LPVOID          pRemoteBuf[2]   = {0,};
    DWORD           dwSize          = 0;

    hMod = GetModuleHandleA("kernel32.dll");

    // set THREAD_PARAM
    param.pFunc[0] = GetProcAddress(hMod, "LoadLibraryA");
    param.pFunc[1] = GetProcAddress(hMod, "GetProcAddress");
    strcpy_s(param.szBuf[0], "user32.dll");
    strcpy_s(param.szBuf[1], "MessageBoxA");
    strcpy_s(param.szBuf[2], "cs.wcupa.edu");
    strcpy_s(param.szBuf[3], "Dr. Chen");

    // Open Process
    if ( !(hProcess = OpenProcess(PROCESS_ALL_ACCESS,   // dwDesiredAccess
                                  FALSE,                // bInheritHandle
                                  dwPID)) )             // dwProcessId
    {
        printf("OpenProcess() fail : err_code = %d\n", GetLastError());
        return FALSE;
    }

    // Allocation for THREAD_PARAM
    dwSize = sizeof(THREAD_PARAM);
    if( !(pRemoteBuf[0] = VirtualAllocEx(hProcess,          // hProcess
                                         NULL,              // lpAddress
                                         dwSize,            // dwSize
                                         MEM_COMMIT,        // flAllocationType
                                         PAGE_READWRITE)) )    // flProtect
    {
        printf("VirtualAllocEx() fail : err_code = %d\n", GetLastError());
        return FALSE;
```

CodeInjection.cpp – InjectCode()

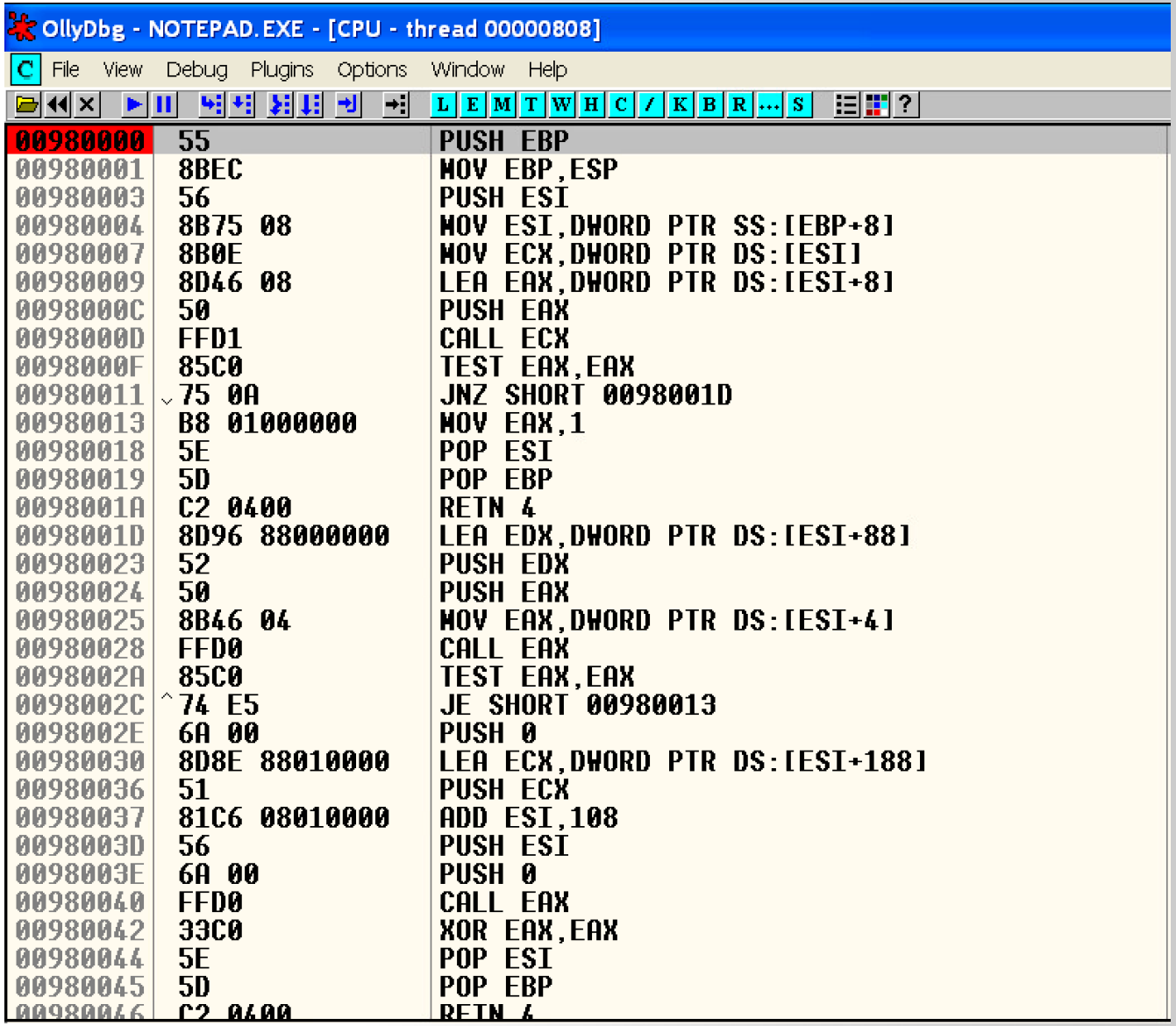
```
// Allocation for THREAD_PARAM
dwSize = sizeof(THREAD_PARAM);
if( !(pRemoteBuf[0] = VirtualAllocEx(hProcess,          // hProcess
                                     NULL,              // lpAddress
                                     dwSize,            // dwSize
                                     MEM_COMMIT,        // flAllocationType
                                     PAGE_READWRITE)) )    // flProtect
{
    printf("VirtualAllocEx() fail : err_code = %d\n", GetLastError());
    return FALSE;
}

if( !WriteProcessMemory(hProcess,                       // hProcess
                        pRemoteBuf[0],                  // lpBaseAddress
                        (LPVOID)&param,                 // lpBuffer
                        dwSize,                         // nSize
                        NULL) )                         // [out] lpNumberOfBytesWritten
{
    printf("WriteProcessMemory() fail : err_code = %d\n", GetLastError());
    return FALSE;
}

// Allocation for ThreadProc()
dwSize = (DWORD)InjectCode - (DWORD)ThreadProc;
if( !(pRemoteBuf[1] = VirtualAllocEx(hProcess,          // hProcess
                                     NULL,              // lpAddress
                                     dwSize,            // dwSize
                                     MEM_COMMIT,        // flAllocationType
                                     PAGE_EXECUTE_READWRITE)) )    // flProtect
{
    printf("VirtualAllocEx() fail : err_code = %d\n", GetLastError());
    return FALSE;
}

if( !WriteProcessMemory(hProcess,                       // hProcess
                        pRemoteBuf[1],                  // lpBaseAddress
                        (LPVOID)ThreadProc,             // lpBuffer
                        dwSize,                         // nSize
                        NULL) )                         // [out] lpNumberOfBytesWritten
{
    printf("WriteProcessMemory() fail : err_code = %d\n", GetLastError());
    return FALSE;
}

if( !(hThread = CreateRemoteThread(hProcess,            // hProcess
                                   NULL,                // lpThreadAttributes
                                   0,                   // dwStackSize
                                   (LPTHREAD_START_ROUTINE)pRemoteBuf[1],     // dwStackSize
                                   pRemoteBuf[0],       // lpParameter
```

CodeInjection.cpp – InjectCode()

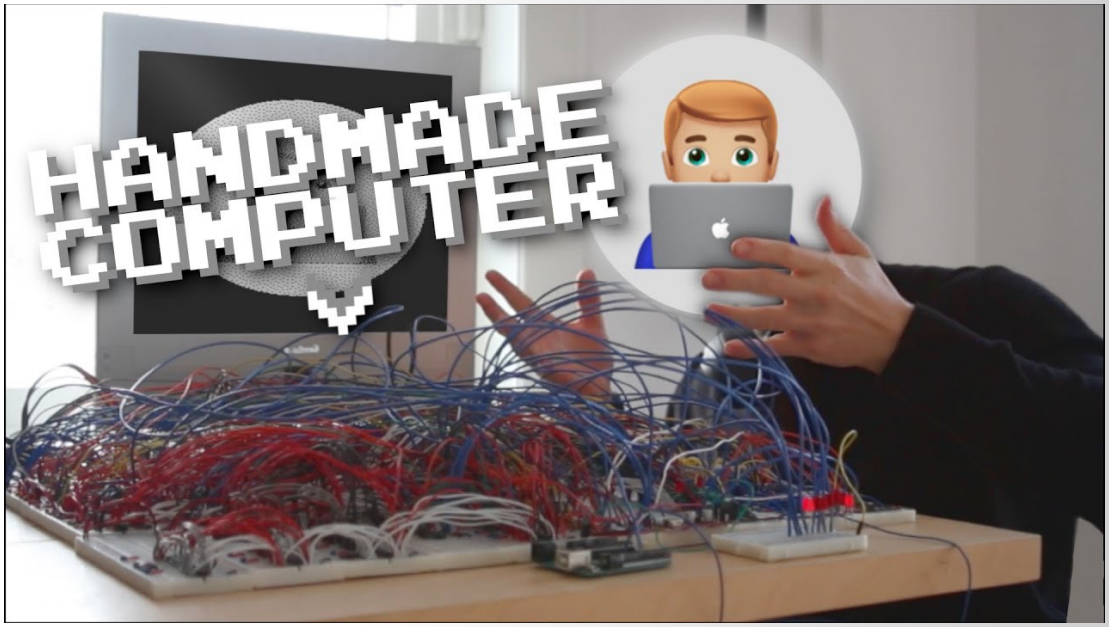
```
// Prepare the THREAD_PARAM structure with necessary function pointers and
strings.
// Open the target process with necessary privileges.
// Allocate memory in the target process for THREAD_PARAM.
// Write THREAD_PARAM to the allocated memory in the target process.
// Allocate memory for the ThreadProc function in the target process and set
it to executable.
// Write the ThreadProc function to the allocated memory in the target
process.
// Create a remote thread in the target process that starts at the ThreadProc
function.
// Wait for the thread to complete execution.
// Close handles and return TRUE on successful injection.
```

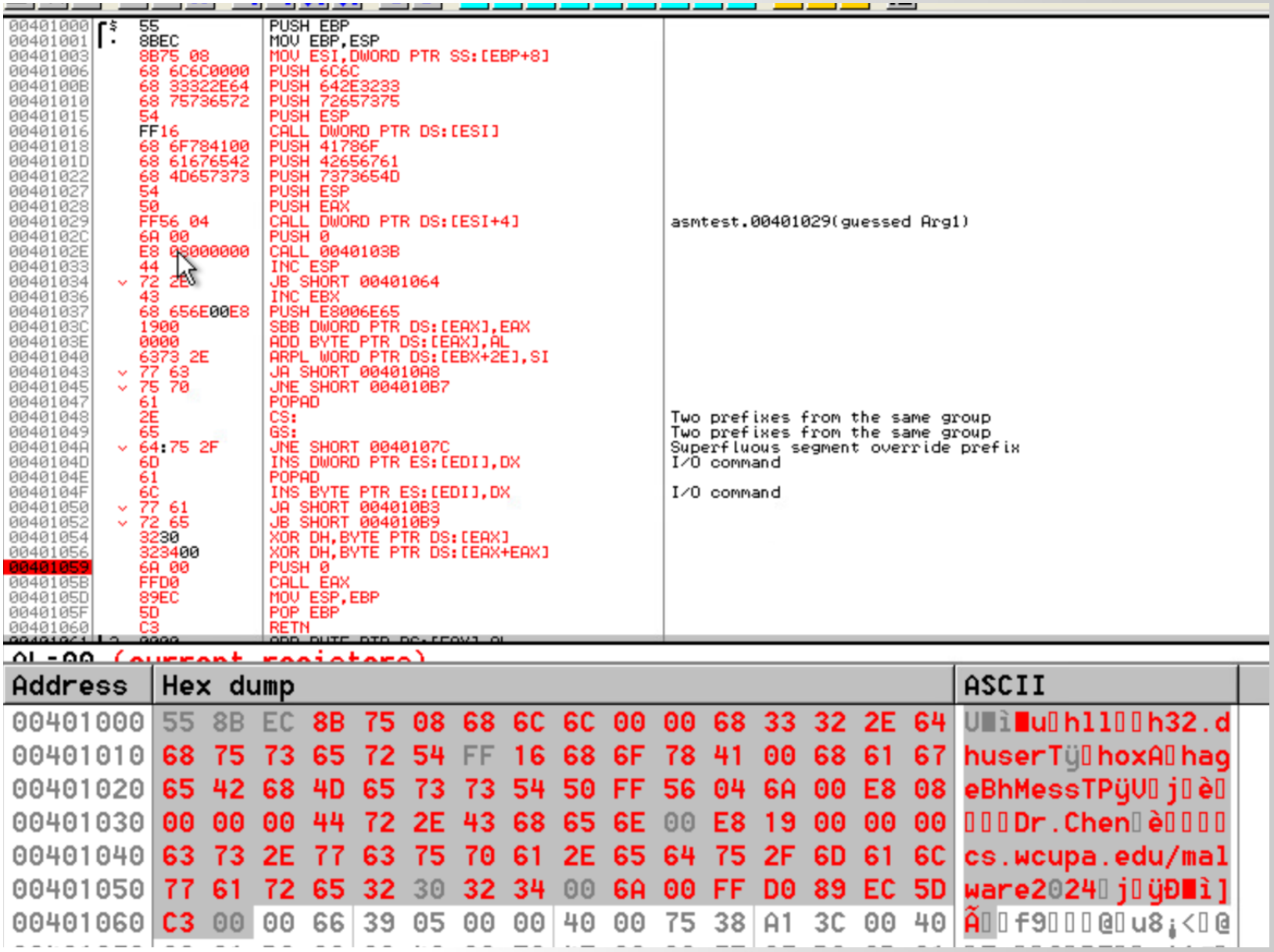
- OpenProcess()
- **//data: THREAD_PARAM**
- VirtualAllocEx()
- WriteProcessMemory()
- **//Code: ThreadProc()**
- VirtualAllocEx()
- WriteProcessMemory()
- CreateRemoteThread()

How to Debug Code Injection (OllyDBG)



Ancient forbidden technique: manual code injection.




00401000 $ 55 PUSH EBP
00401001 . 8BEC MOV EBP,ESP
00401003 8B75 08 MOV ESI,DWORD PTR SS:[EBP+8]
00401006 68 6C6C0000 PUSH 6C6C
0040100B 68 33322E64 PUSH 642E3233
00401010 68 75736572 PUSH 72657375
00401015 54 PUSH ESP
00401016 FF16 CALL DWORD PTR DS:[ESI]
00401018 68 6F784100 PUSH 41786F
0040101D 68 61676542 PUSH 42656761
00401022 68 4D657373 PUSH 7373654D
00401027 54 PUSH ESP
00401028 50 PUSH EAX
00401029 FF56 04 CALL DWORD PTR DS:[ESI+4]
0040102C 6A 00 PUSH 0
0040102E E8 08000000 CALL 0040103B
00401033 44 INC ESP
00401034 72 2E JB SHORT 00401064
00401036 43 INC EBX
00401037 68 656E00E8 PUSH E8006E65
0040103C 1900 SBB DWORD PTR DS:[EAX],EAX
0040103E 0000 ADD BYTE PTR DS:[EAX],AL
00401040 6373 2E ARPL WORD PTR DS:[EBX+2E],SI
00401043 77 63 JA SHORT 004010A8
00401045 75 70 JNE SHORT 004010B7
00401047 61 POPAD
00401048 2E CS:
00401049 65 GS:
0040104A 64:75 2F JNE SHORT 0040107C
0040104D 6D INS DWORD PTR ES:[EDI],DX
0040104E 61 POPAD
0040104F 6C INS BYTE PTR ES:[EDI],DX
00401050 77 61 JA SHORT 004010B3
00401052 72 65 JB SHORT 004010B9
00401054 3230 XOR DH,BYTE PTR DS:[EAX]
00401056 323400 XOR DH,BYTE PTR DS:[EAX+EAX]
00401059 6A 00 PUSH 0
0040105B FFD0 CALL EAX
0040105D 89EC MOV ESP,EBP
0040105F 5D POP EBP
00401060 C3 RETN
asmtest.00401029(guessed Arg1)
Two prefixes from the same group
Two prefixes from the same group
Superfluous segment override prefix
I/O command
I/O command
Address Hex dump ASCII
00401000 55 8B EC 8B 75 08 68 6C 6C 00 00 68 33 32 2E 64
00401010 68 75 73 65 72 54 FF 16 68 6F 78 41 00 68 61 67
00401020 65 42 68 4D 65 73 73 54 50 FF 56 04 6A 00 E8 08
00401030 00 00 00 44 72 2E 43 68 65 6E 00 E8 19 00 00 00
00401040 63 73 2E 77 63 75 70 61 2E 65 64 75 2F 6D 61 6C
00401050 77 61 72 65 32 30 32 34 00 6A 00 FF D0 89 EC 5D
00401060 C3 00 00 66 39 05 00 00 40 00 75 38 A1 3C 00 40

Q & A