Step by Step

다시 시작하는 윈도우 프로그래밍

처리되지 않은 예외필터와 크래시덤프 활용

모든 소프트웨어는 필연적으로 '실패' 할 수 있다. 단언컨대 실패하지 않는 소프트웨어란 없다. 마이크로소프트의 워드나 구글의 크롬과 같은 세계적인 소프트웨어조차도 때론 실패한다. 여기서의 실패는 프로그램의 중단을 의미하며, 이때 작업 결과물을 잃을 수 있는데 워드나 크롬과 같은 소프트웨어의 경우 예외처리 루틴 덕분에 실패 시에도 작업 내용을 잃는 경우가 드물다. 이처럼 예외처리는 견고한 소프트웨어 개발의 필수 요소인데, 지금부터 예외처리의 적용 방안을 함께 살펴보자.

테크니컬 레벨 ★★

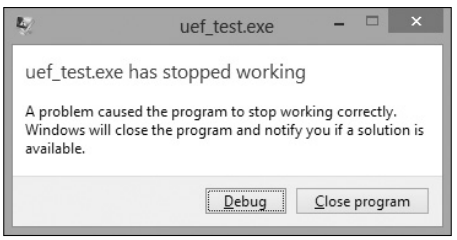
신영진 pop@jiniya.net, www.jiniya.net | 웰비아닷컴을 창업해 XIGNCODE3라는 게임 보안 솔루션을 개발하고 있다. 시스템 프로그래밍에 관심이 많고 다수의 PC 보안 프로그램 개발에 참여했다. 월간 마이크로소프트웨어의 오랜 필자이자 Microsoft MVP, 데브피아 비주얼 C++ 섹션 시삽으로도 활동하고 있다. Steve Barakatt과 Grey's Anatomy의 광팬이며, 한때는 WoW에 미쳤었다. 한마디로 말하면 괴짜다.

본격적인 예외처리 적용 방안을 살펴보기에 앞서 기본적인 예외처리의 역할을 이해하고 넘어가자. 프로그램에 크래시가 발생되는 간단한 예제인 〈리스트 1〉을 실행하면 윈도우 버전마다 조금씩 다를 수 있지만 〈그림 1〉과 같은 에러 메시지가 출력된다.

이 경우 사용자가 선택할 수 있는 것은 두 가지뿐인데, 이 소프트웨어를 디버깅하거나 종료시키는 것이 그것이다. 대부분의 사용자는 이러한 상황에서 대화상자를 닫는 것 외에 선택의 길이 없다. 결국 프로그램이 종료되고 지금까지 작업했던 결과물을 잃게 된다.

〈리스트 1〉 크래시가 발생되는 예

```
#include <stdio.h>

int main()
{
        int *a = NULL;
        *a = 3;

        return 0;
}
```



〈그림 1〉 〈리스트 1〉을 실행할 경우 출력되는 오류 보고 대화상자

이러한 '오류 보고 대화상자' 는 시스템 대화상자로 OS 사용자 경험상 동일한 경고 메시지가 출력되기 때문에 큰 문제가 없는 것으로 여길 수 있다. 그러나 사용자 측면에서 선택의 여지없이 해당 프로그램을 닫을 수밖에 없기에 시스템 대화상자가 출력되는 것에 어떤 의미가 있다고 볼 수도 없다.

만약 이러한 오류 보고 대화상자가 출력되지 않도록 하고 싶다면 어떻게 해야 할까? 바로 SetErrorMode 함수가 그러한 기능을 지원한다. 해당 함수에 SEM_NOGPFAULTERRORBOX 플래그를 지정하면 크래시가 발생해도 오류 보고 대화상자가 출력되지 않는다. 〈리스트 1〉에 이러한 기능을 추가한 예제는 〈리스트 2〉와 같으며, 이 예제를 실행하면 크래시가 발생하지만 어떠한 시스템 대화상자도 출력되지 않고 곧바로 프로그램이 종료된다.

〈리스트 2〉 크래시 대화 상자가 출력되지 않는 예제

```
#include <stdio.h>

int main()
{
        SetErrorMode(SEM_NOGPFAULTERRORBOX);

        int *a = NULL;
        *a = 3;

        return 0;
}
```

이처럼 SetErrorMode 함수를 이용하면 현재 프로세스의 에러 처리 모드를 손쉽게 설정할 수 있다. 선택 가능한 세부 옵션은 〈표 1〉에서 확인할 수 있는데, 일반적으로 프로그램 사용 환경과 관계없이 OS의 시스템 대화상자를 없애고 싶다면 SEM_FAILCRITICALERRORS, SEM_NOGPFAULTERRORBOX, SEM_NOOPENFILEERRORBOX를 조합해 지정하면 된다. 이렇게 시스템 대화상자가 출력되지 않게 설정해도 오류 발생 시 호출한 함수의 반환값이나 예외처리기를 통해 오류 코드가 전달된다. 그러므로 이러한 오류를 소프트웨어에서 직접 예외처리할 수 있다.

플래그	설명
0	시스템 디폴트로 설정
SEM_FAILCRITICALERRORS	치명적인 오류 발생 시 대화상자 출력 안 함
SEM_NOALIGNMENTFAULTEXCEPT	메모리 정렬 예외를 발생시키지 않음. 이 플래그가 설정된 경우 운영체제에서 메모리 정렬을 직접 해결할 수 있을 때에만 메모리 정렬 문제를 직접 수정함
SEM_NOGPFAULTERRORBOX	윈도우 오류 보고 대화상자 출력 안 함
SEM_NOOPENFILEERRORBOX	파일이 없는 경우 파일을 찾을 수 없다는 대화상자를 출력하지 않음. LoadLibrary 함수를 사용한 경우 DLL이 존재하지 않으면 OS에 따라 파일 없음 경고창이 출력되는 경우가 있음

〈표 1〉 SetErrorMode 함수의 세부 옵션

구조화된 예외처리기 활용

윈도우에서 프로그램이 실패하면 오류 보고 대화상자가 출력되거나 그 즉시 해당 프로그램이 종료된다. 이러한 두 가지 방식을 오류 처리라고 부를 수는 없을 것이다. 왜냐하면 해당 프로그램을 개발한 프로그래머가 예측하지 못한 경로를 통해 프로그램이 종료되기 때문이다. 그러므로 가장 좋은 예외처리 방법은 예외가 발생한 코드 블록을 예외처리기로 보호하는 방법이다.

지난 시간에 살펴본 /Eha 옵션도 그 중 하나인데, 이 옵션을 이용하면 C++의 catch(…)로 예외를 처리할 수 있다. 이 옵션 외에도 SEH 예외처리기로 코드를 직접 보호하는 것도 또 다른 방법이 될 수 있다.

〈리스트 3〉은 SEH 예외처리기를 활용한 예제다. 비주얼 C++에서는 '__try/__except/__finally'와 같은 키워드가 이러한 기능을 지원한다. 코드의 규모가 제법 큰 소프트웨어를 개발해봤다면 동일한 기능의 서로 다른 메커니즘을 함께 쓰는 것이 바람직하지 않음을 이미 알고 있을 것이다. 따라서 SEH 예외처리기와 C++ 예외처리기는 동시에 사용할 수 있더라도 이 중 하나만 사용하는 것이 바람직하다. 예컨대 C++로 개발한다면 해당 언어에서 제공하는 C++ 예외처리 메커니즘만을 선택하는 것이 좋다.

〈리스트 3〉 SEH 예외처리기를 이용한 예외처리

```
#include <stdio.h>

int main()
{
        __try
        {
                int *a = NULL;
                *a = 3;
        }
        __except(EXCEPTION_EXECUTE_HANDLER)
        {
                // 예외처리 코드
        }

        return 0;
}
```

처리되지 않은 예외필터를 사용한 예외처리

그러므로 예외처리 핸들러를 적절히 사용해 코드를 작성하는 습관이 중요하다. 그러나 안타깝게도 아무리 신경을 쓴들 어디선가 프로그램이 실패하게 마련이다. 코드 작성 중에도 놓친 부분이 있을 수 있고, 때론 외부 라이브러리에서 예외가 발생할 수도 있기 때문이다. 외부 라이브러리가 소스 코드도 제공한다면 직접 수정하면 되지만 대부분의 상용 라이브러리는 그렇지 않다. 결국 해당 라이브러리에 예외처리를 추가한다는 것은 애초에 시도조차 할 수 없는 경우가 허다하다. 프로그램 개발에 있어 모든 예외를 처리하는 것은 결국 불가능한 일일까? 반드시 구멍이 있을 수밖에 없다면 이러한 구멍을 메울 방법은 정녕 없는 것일까? 윈도우 시스템이 제공하는 '처리되지 않은 예외필터(Unhandled Exception Filter)'가 이러한 질문에 대한 답이 될 수 있다. 처리되지 않은 예외필터란 말 그대로 프로세스 내에서 처리되지 않은

예외가 발생할 경우 실행되는 필터 코드를 의미한다.

기본적으로 윈도우는 프로그램에서 예외가 발생하면 SEH 핸들러가 존재하는지를 먼저 조사한다. 그 과정은 스택을 거슬러 올라가면서 확장하는 식인데, 만약 적절한 예외처리 핸들러를 발견하면 그 지점부터 프로그램을 재개한다. 반면 스택의 마지막까지 적절한 예외처리 핸들러가 발견되지 않으면 처리되지 않은 예외필터가 등록돼 있는지를 검사하는데, 예외필터가 등록된 경우 해당 필터를 실행하고 없는 경우 그 지점에서 오류 보고 대화상자를 출력한다. 즉 처리되지 않은 예외필터를 등록하면 미처 예측하지 못한 예외가 발생해도 프로그램이 크래시되기 전에 예외처리를 할 수 있다.

처리되지 않은 예외필터는 SetUnhandledExceptionFilter 함수로 등록할 수 있으며, 그 원형은 다음과 같다.

```
LPTOP_LEVEL_EXCEPTION_FILTER WINAPI SetUnhandled
ExceptionFilter
(LPTOP_LEVEL_EXCEPTION_FILTER lpTopLevelExceptionFilter);
```

lpTolLevelExceptionFilter에 필터 함수 포인터를 전달하면 해당 함수가 필터로 등록된다. 반환값으로는 이전에 설정된 필터 함수 주소를 반환하는데, 만약 설정된 필터 함수가 없는 경우 NULL을 반환한다. 주의할 점은 처리되지 않은 예외필터는 프로세스별로 하나만 등록할 수 있다는 점이다. 새로운 필터가 등록되면 이전 예외필터는 OS에 의해 호출되지 않는다. 처리되지 않은 예외필터 함수의 원형은 다음과 같다.

```
LONG WINAPI UnhandledExceptionFilter
(struct _EXCEPTION_POINTERS *ExceptionInfo);
```

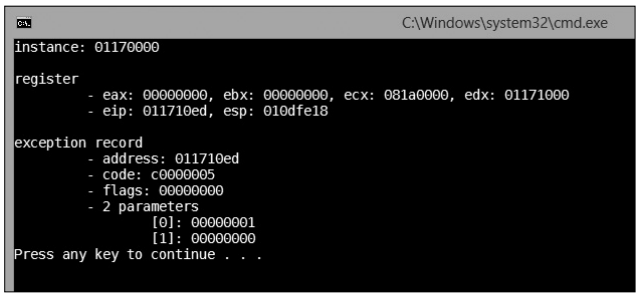
ExceptionInfo 구조체에는 예외 정보와 예외가 발생한 시점의 레지스터 정보가 넘어온다. 필터 함수가 EXCEPTION_EXECUTE_HANDLER를 반환하면 예외가 정상적으로 처리됐음을 OS에 알리는데, 이러한 통지를 받으면 윈도우는 해당 프로그램을 종료한다.

만약 EXCEPTION_CONTINUE_SEARCH가 반환되면 윈도우는 통상적인 크래시 처리 절차에 따라 프로그램을 종료한다. 이때 에러 모드가 오류 보고 대화상자를 출력하지 않도록 설정돼 있으면 그 즉시 프로그램이 종료될 것이다.

EXCEPTION_CONTINUE_EXECUTION가 반환되면 수정된 컨텍스트 레코드로부터 프로그램을 재개한다. 일반적으로는 넘어온 ExceptionInfo의 ContextRecord의 Eip 값을 변경해 프로그램의 재개 지점을 조정할 수 있다.

〈리스트 4〉 처리되지 않은 예외필터 예제

```
#include <stdio.h>
#include <windows.h>

LONG
WINAPI
ExFilter(PEXCEPTION_POINTERS info)
{
        printf("instance: %p\n\n",
GetModuleHandle(NULL));

        printf("register\n");
        printf("\t - eax: %08x, ebx: %08x, ecx: %08x,
edx: %08x\n"
                , info->ContextRecord->Eax
                , info->ContextRecord->Ebx
                , info->ContextRecord->Ecx
                , info->ContextRecord->Edx);

        printf("\t - eip: %08x, esp: %08x\n"
                , info->ContextRecord->Eip
                , info->ContextRecord->Esp);

        printf("\nexception record\n");
        printf("\t - address: %08x\n", info-
>ExceptionRecord->ExceptionAddress);
        printf("\t - code: %08x\n", info-
>ExceptionRecord->ExceptionCode);
        printf("\t - flags: %08x\n", info-
>ExceptionRecord->ExceptionFlags);
        printf("\t - %d parameters\n", info-
>ExceptionRecord->NumberParameters);

        for(int i=0; i<info->ExceptionRecord-
>NumberParameters; ++i)
        {
                printf("\t\t [%d]: %08x\n", i, info-
>ExceptionRecord->ExceptionInformation[i]);
        }

        return EXCEPTION_EXECUTE_HANDLER;
}

int main()
{
        SetUnhandledExceptionFilter(ExFilter);

        int *a = NULL;
        *a = 3;

        return 0;
}
```



〈그림 2〉 처리되지 않은 예외필터를 통해 확인 가능한 예외 정보

처리되지 않은 예외필터 예인 〈리스트 4〉는 코드에 등록된 필터를 통해 넘어온 예외처리 정보를 출력하고 프로그램을 종료한다. 그 결과는 〈그림 2〉와 같은데, 지금부터 출력된 각각의 예외 정보를 살펴보자. 여기서 instance는 실행된 프로그램이 로드된 주소를, register는 예외가 발생한 시점의 레지스터 정보를, exception record는 발생된 예외와 관련된 정보를 의미하는데 이를 통해 알 수 있는 보다 자세한 의미는 다음과 같다.

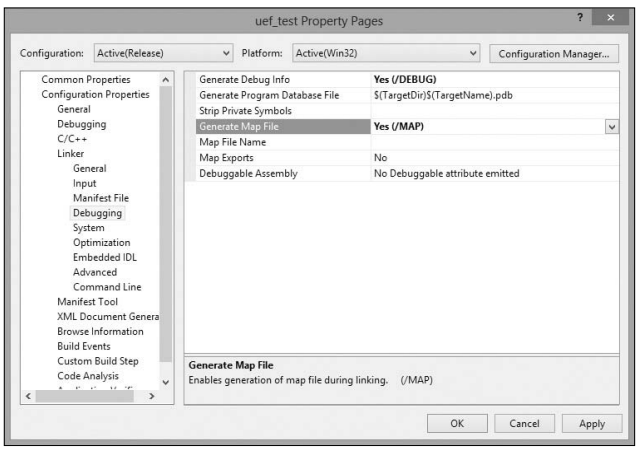
- 이 예외는 소프트웨어의 0x11710ed 번지에서 발생했다.
- 발생된 예외는 접근 위반을 의미하는 STATUS_ACCESS_VIOLATION(0xC0000005) 예외다.
- 이 예외는 메모리에 기록하는 과정 중에서 발생했는데, STATUS_ACCESS_VIOLATION의 첫 번째 매개변수가 0이면 읽기, 1이면 쓰기를 의미한다.
- 0 번지에 접근하다 예외가 발생했다. STATUS_ACCESS_VIOLATION의 두 번째 매개변수는 접근하려던 주소를 뜻한다.

맵 파일

처리되지 않은 예외필터를 이용하면 예외에 대한 보다 상세한 정보를 확인할 수 있다. 그러나 해당 정보만으로 실제 소프트웨어의 어느 지점에서 예외가 발생했는지는 알 수 없다. 이 경우 맵 파일을 활용하면 해당 예외 정보의 주소로부터 프로그램의 어느 위치에서 예외가 발생했는지도 추적할 수 있다.

맵 파일을 생성하기 위해서는 비주얼 스튜디오의 프로젝트 설정 대화상자에서 [Linker/Debugging] 메뉴의 맵 파일 생성 옵션을 변경하면 된다(〈그림 3〉 참조). 이곳에서 맵 파일이 생성되도록 설정했다면 컴파일 시 확장자가 map인 텍스트 파일 하나가 함께 만들어진다.

이렇게 생성된 맵 파일은 〈리스트 5〉와 같다. 맵 파일은 텍스트 파일인 만큼 어떠한 문서 편집기로도 읽을 수 있으며, 4개의 탭으로 구성된 각 항목은 각각 상대주소, 심벌명, 절대주소, 오브젝트 파일명을 의미한다. 〈그림 2〉를 다시 살펴보면 이 프로그램이 0x1170000 번지에 로드됐고 예외는 0x11710ed에서 발생했다. 이 정보를 토대로 맵 파일의 번지와 대조해보면 10e0의 main과 10f4의 security_check_cookie 사이에서 예외가 발생했음을 알 수 있다. 10f4보다는 예외가 발생한 번지가 작기 때문에 해당 예외는 main 함수의 루틴을 처리하는 과정에서 발생했음을 쉽게 유추할 수 있다. 즉, 프로그램의 전체 코드를 점검하지 않고 main 함수의 처리 부분만 꼼꼼히 살펴본다면 예외 발생의 원인을 쉽게 제거할 수 있는 것이다.



〈그림 3〉 비주얼 스튜디오 2008의 맵 파일 생성 옵션

〈리스트 5〉 생성된 맵 파일

```
// 중략

Address            Publics by Value    Rva+Base  Lib:Object

// 중략

0001:000000e0      _main
004010e0 f         uef_test.obj
0001:000000f4      @__security_check_cookie@4   004010f4 f
MSVCRT:secchk.obj
0001:000003a5      _mainCRTStartup              004013a5 f
MSVCRT:crtexe.obj
0001:000003af      ___report_gsfailure
004013af f         MSVCRT:gs_report.obj

// 중략
```

크래시덤프

앞서 소개한 예제처럼 맵 파일의 위치 정보를 통해 크래시 발생 범위를 좁힐 수 있었다. 그러나 실제로 우리가 접하게 되는 크래시 상황은 매우 복잡한 만큼 위치 정보만으로는 문제의 원인을 추적하기 어렵다. 이 경우 단순한 예외처리 정보보다는 크래시가 발생한 지점의 메모리 덤프를 분석하는 것이 더 효과적이다.

크래시덤프란 크래시가 발생한 시점의 메모리 상태를 기록한 파일이다. 이러한 덤프 파일을 이용하면 당시 메모리에 어떤 값

이 있었고, 어떤 경로를 통해 코드가 그 위치로 갔는지, 각 스레드는 어떤 지점을 실행하고 있는지, 어떤 모듈이 로드됐는지 등의 수많은 물음에 대한 답을 찾을 수 있다.

〈리스트 6〉 미니덤프 생성 코드

```
#include <stdio.h>
#include <windows.h>
#include <dbghelp.h>

typedef BOOL
(WINAPI *MiniDumpWriteDumpT)(HANDLE hProcess
                                    , DWORD ProcessId
                                    , HANDLE hFile
                                    , MINIDUMP_TYPE
DumpType
                                    ,
PMINIDUMP_EXCEPTION_INFORMATION ExceptionParam
                                    ,
PMINIDUMP_USER_STREAM_INFORMATION UserStreamParam
                                    ,
PMINIDUMP_CALLBACK_INFORMATION CallbackParam);

BOOL
CreateMiniDump(LPCWSTR dump_path, PEXCEPTION_POINTERS pex,
MINIDUMP_TYPE level)
{
        HANDLE file = CreateFile(dump_path
                        , GENERIC_READ | GENERIC_WRITE
                        , 0
                        , NULL
                        , CREATE_ALWAYS
                        , FILE_ATTRIBUTE_NORMAL
                        , NULL);

        if(file == INVALID_HANDLE_VALUE)
        {
                return FALSE;
        }

        HMODULE module = LoadLibraryW(L"dbghelp.dll");
        if(!module)
        {
                CloseHandle(file);
                return FALSE;
        }

        MiniDumpWriteDumpT pMiniDumpWriteDump;
        pMiniDumpWriteDump = (MiniDumpWriteDumpT)
GetProcAddress(module, "MiniDumpWriteDump");
        if(!pMiniDumpWriteDump)
        {
                CloseHandle(file);
                FreeLibrary(module);
                return FALSE;
        }

        MINIDUMP_EXCEPTION_INFORMATION mdei;

        mdei.ThreadId = GetCurrentThreadId();
        mdei.ExceptionPointers = pex;
        mdei.ClientPointers = TRUE;

        BOOL result =
pMiniDumpWriteDump(GetCurrentProcess()
                                , GetCurrentProcessId()
                                , file
                                , level
                                , pex ? &mdei : 0
                                , 0
                                , 0);

        FreeLibrary(module);
        CloseHandle(file);

        return result;
}

LONG
WINAPI
ExFilter(PEXCEPTION_POINTERS info)
{
        CreateMiniDump(L"c:\\dump.dmp", info,
MiniDumpFilterMemory);
        return EXCEPTION_EXECUTE_HANDLER;
}

int main()
{
        SetUnhandledExceptionFilter(ExFilter);
        SetErrorMode(SEM_NOGPFAULTERRORBOX);

        int *a = NULL;
        *a = 3;

        return 0;
}
```

메모리 덤프 파일을 생성하기 위해서는 dbghelp.dll에 있는 MiniDumpWriteDump 함수를 사용해야 한다. 〈리스트 6〉은 해당 함수를 사용해 메모리 덤프를 생성하는 예인데, CreateMiniDump 함수의 첫 번째 매개변수에는 생성될 덤프 파일의 경로를, 두 번째 매개변수에는 예외 정보를, 세 번째 매개변수에는 생성한 덤프 파일 타입을 대입하면 된다.

또한 덤프 파일 타입에 MiniDumpNormal을 대입하면 제한적인 메모리 정보만, MiniDumpWithFullMemory를 대입할 경우 모든 메모리 정보가 포함된 덤프 파일이 생성된다.

모든 메모리 정보가 저장된 풀덤프 파일의 용량은 해당 프로그램이 사용한 메모리양에 따라 결정되기 때문에 때론 풀덤프 파일의 용량이 기가바이트 단위를 넘을 수 있다. 그러므로 최종 사용자에게 배포하는 버전이라면 가급적 미니덤프 파일을 생성하는 편이 좋다. 미니덤프의 경우 제한된 메모리 정보만 저장하기 때문에 분석이 좀더 힘든 단점이 있다.

〈리스트 7〉 windbg 덤프 분석 내용

```
0:000> !analyze -v

// 중략

FAULTING_IP:
uef_test!main+15 [c:\users\codewiz\documents\visual studio
2008\projects\uef_test\uef_test.cpp @ 109]
00e511d5 c70003000000  mov  dword ptr [eax],3

EXCEPTION_RECORD: ffffffff -- (.exr 0xffffffffffffffff)
ExceptionAddress: 00e511d5 (uef_test!main+0x00000015)
          ExceptionCode: c0000005 (Access violation)
          ExceptionFlags: 00000000
NumberParameters: 2
          Parameter[0]: 00000001
          Parameter[1]: 00000000
Attempt to write to address 00000000

DEFAULT_BUCKET_ID:  NULL_POINTER_WRITE

// 중략

FAULTING_SOURCE_CODE:
          105:      SetUnhandledExceptionFilter(ExFilter);
          106:      SetErrorMode(SEM_NOGPFAULTERRORBOX);
          107:
          108:      int *a = NULL;
>         109:      *a = 3;
          110:      return 0;
          111: }
          112:

// 중략
```

〈리스트 6〉을 실행하고 생성된 덤프 파일을 windbg 프로그램으로 열어보자. '!analyze -v' 명령어로 덤프 정보를 분석하면 〈리스트 7〉과 같은 덤프 발생 위치에 대한 소스 코드 정보까지 확인할 수 있다. 만약 디버깅과 동일한 각종 명령어를 활용하면 좀더 자세한 정보를 얻을 수 있을 것이다.

이렇게 생성된 덤프 파일은 비주얼 스튜디오에서도 분석 가능하다. 비주얼 스튜디오의 [파일] → [열기] 메뉴에서 솔루션을 선택한 다음에 생성된 덤프 파일을 열고 디버깅을 시작하면 소스 코드의 어느 위치에서 크래시가 발생했는지를 확인할 수 있다. 이와 마찬가지로 windbg 프로그램도 비주얼 스튜디오의 디버깅을 통해 얻을 수 있는 동일한 정보를 조회할 수 있다.