Windows System Programming

교재 Sample

㈜ 아임구루 부설 아이오 교육센터 (www.ioacademy.co.kr)

SECTION **01**

어셈블리 언어와 C언어의 원리

요즘은 어셈블리언어를 거의 사용하지 않고 C/C++/Java 등의 고급 언어를 사용합니다. 하지만 간단한 어셈블리어를 읽고 이해 할 수 있다면 시스템에 대해 보다 깊이 있게 이해를 할 수 있게 됩니다.

TEB, PEB 접근 등 다양한 윈도우 시스템 고급 기법을 이해 하기 위해서 약간의 어셈블리에 대한 지식은 반드시 필요합니다.

또한 함수호출의 원리, Calling Convention, 지역 변수의 원리 등 C/C++의 근본적인 원리를 정확히 이해 할 수 있습니다.

❑ 주요 내용

- 어셈블리 언어의 기본 코드를 이해 하고 C 함수와 어셈블리 함수를 상호 호출하는 방법을 학습 합니다.
- 함수 호출의 정확한 원리를 이해하고 다양한 함수 호출규약(Calling Convention)을 정확히 이해하고 각각의 방식의 장단점을 학습 합니다 .
- 인라인 어셈블리를 사용해서 TEB, PEB 등의 윈도우 구조체에 접근하는 방법을 학습 합니다 .
- OllyDbg 를 사용해서 간단한 Crack, Reversing 방법을 학습 합니다 .

Item 1 - 01

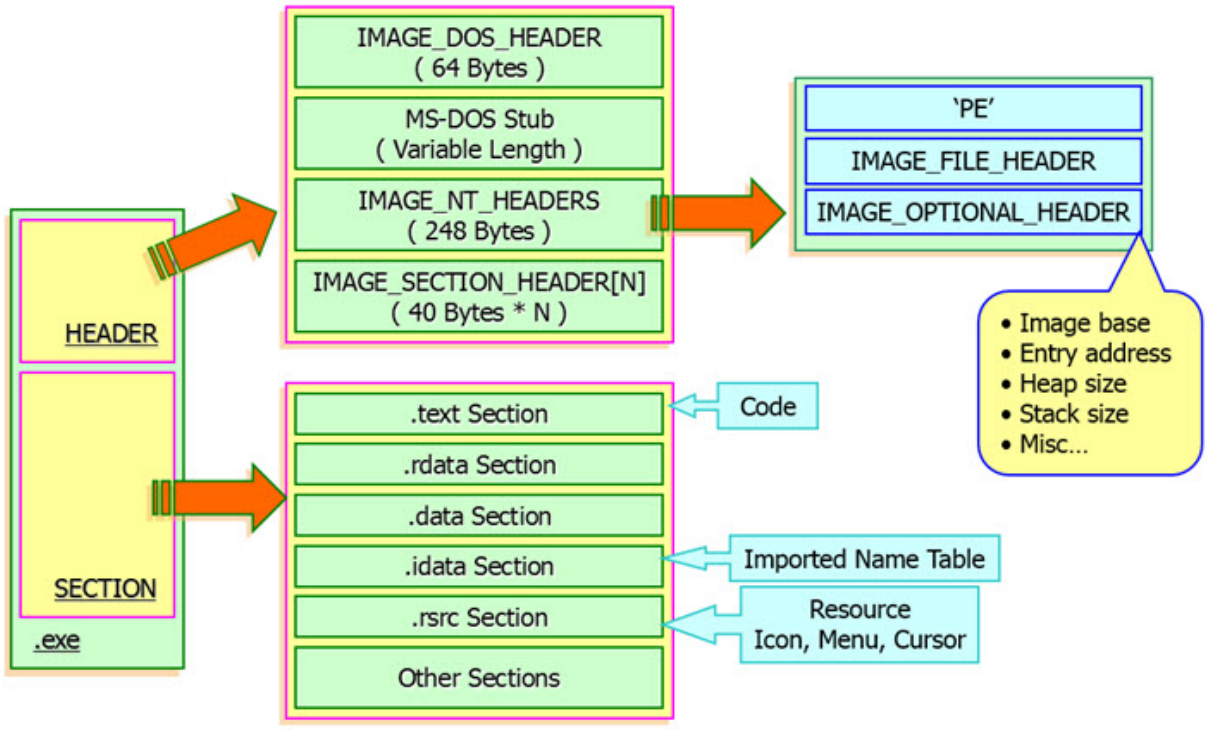
PE Format

❑ 핵심개념

- ✓ PE Format 의 구조
  - IMAGE_NT_HEADERS(IMAGE_FILE_HEADER, IMAGE_OPTIONAL_HEADER)
  - Sections
- ✓ PEview.exe
- ✓ 사용자 정의 섹션 만들기

1. 실행파일의 구조

- **P**ortable **E**xecutable File Format



2. 실행파일 헤더

PE 구조

<table>
<tr><td colspan="2">IMAGE_DOS_HEADER
MS DOS Stub</td><td>MS DOS 시절에 사용하던 헤더
현재는 사용되지 않는다.</td></tr>
<tr><td rowspan="3">IMAGE_NT_HEADERS</td><td>Signature</td><td>항상 "PE"로 되어 있다.</td></tr>
<tr><td>IMAGE_FILE_HEADER</td><td>Machine종류, 날짜등</td></tr>
<tr><td>IMAGE_OPTIONAL_HEADER</td><td>가장 중요한 헤더
AddressOfEntry, Image Base
Stack, Heap 크기등의 정보를 가진다</td></tr>
<tr><td colspan="2">IMAGE_SECTION_HEADER</td><td>각 Section의 정보를 담고 있다.
Section의 개수만큼 존재한다.</td></tr>
</table>

3. Sections

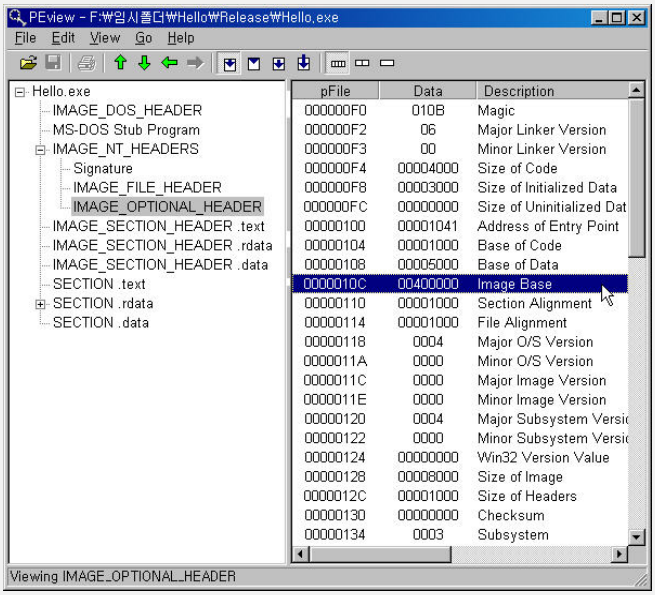
- 실행 파일은 PE 헤더 다음으로 Section 이라는 요소로 구분된다.
- 대부분의 표준 Section 은 "."으로 시작한다.

Section Name	Purpose
.bss	Uninitialized data
.CRT	Read-only C run-time data
.data	Initialized data
.debug	Debugging information
.didata	Delay imported names table
.edata	Exported names table
.idata	Imported names table
.rdata	Read-only run-time data
.reloc	Relocation table information
.rsrc	Resources
.text	Code
.tls	Thread-Local-Storage
.xdata	Exception Handling Table

4. PEView

http://wjradburn.com/software/

```
#include <stdio.h>

char s[] = "abcdefg";

void main()
{
    printf("hello, world\n");
}
```



5. 사용자 정의 섹션 추가 하기

```
#pragma data_seg()
#pragma code_seg()
```

```
char s1[] = "abcdefg";

#pragma data_seg("MYDATA")
char s2[] = "opqrstu";
#pragma data_seg()

#pragma code_seg("MYCODE")
void foo() { }
#pragma code_seg()

void main()
{
}
```

Item 1 - 02

함수 호출의 원리 1

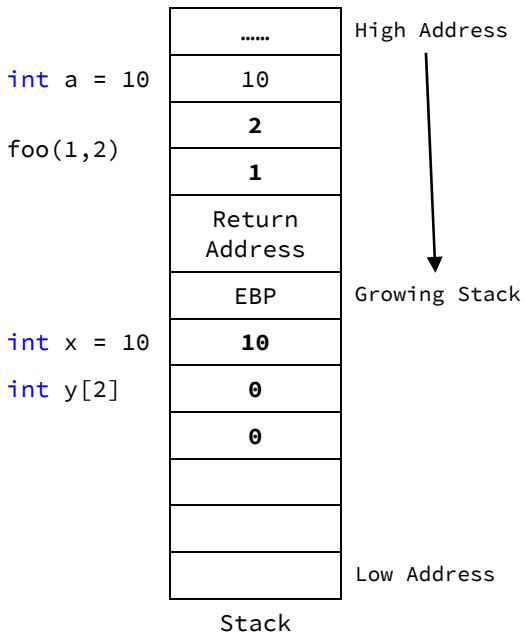
❑ 핵심개념

✓ 함수 호출의 원리
- 마지막 인자부터 stack 에 넣고 함수로 이동
- 함수의 반환 값은 EAX 레지스터를 사용

✓ __asm {} 키워드로 인라인 어셈블리 사용하기

1. 함수 호출과 stack 구조

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

void goo()
{
        printf("goo\n");
        exit(0);
}

void foo( int a, int b)
{
        int x = 10;
        int y[2]={0,0};

        y[2] = 20;        // A

        printf("%d\n", x);

        y[4] = (int)goo; // B
        return;
}
void main()
{
        int a = 10;
        foo( 1,2);
}
```



2. 인라인 어셈블리 사용하기

- __asm {} 키워드를 사용한 인라인 어셈블리

```
int Add(int a, int b)
{
	int c = a + b;

	return c;
}

int main()
{
	int n = 0;

	Add(1,2);

	__asm
	{
		mov  n, eax
	}
	printf("%d\n", n);
}
```

```
int Add(int a, int b)
{
	int c = a + b;

	__asm
	{
		mov	eax, c
	}
}

int main()
{
	int n = 0;

	n = Add(1,2);

	printf("%d\n", n);
}
```

Item 1 - 03

Hello, Assembly

❑ 핵심개념

✓ 어셈블리 소스의 이해
- http://www.nasm.us
- 어셈블리 기본 코드
- 전역변수 사용하기

✓ Compile & Linking
- Console 창에서 Command Line 컴파일 하기
- VC++과 nasm 의 연동

1. Hello, Assembly

- `cl main.c /c`
- `nasm -f win32 -o first. obj first.asm`
- `link main.obj first.obj`

main.c

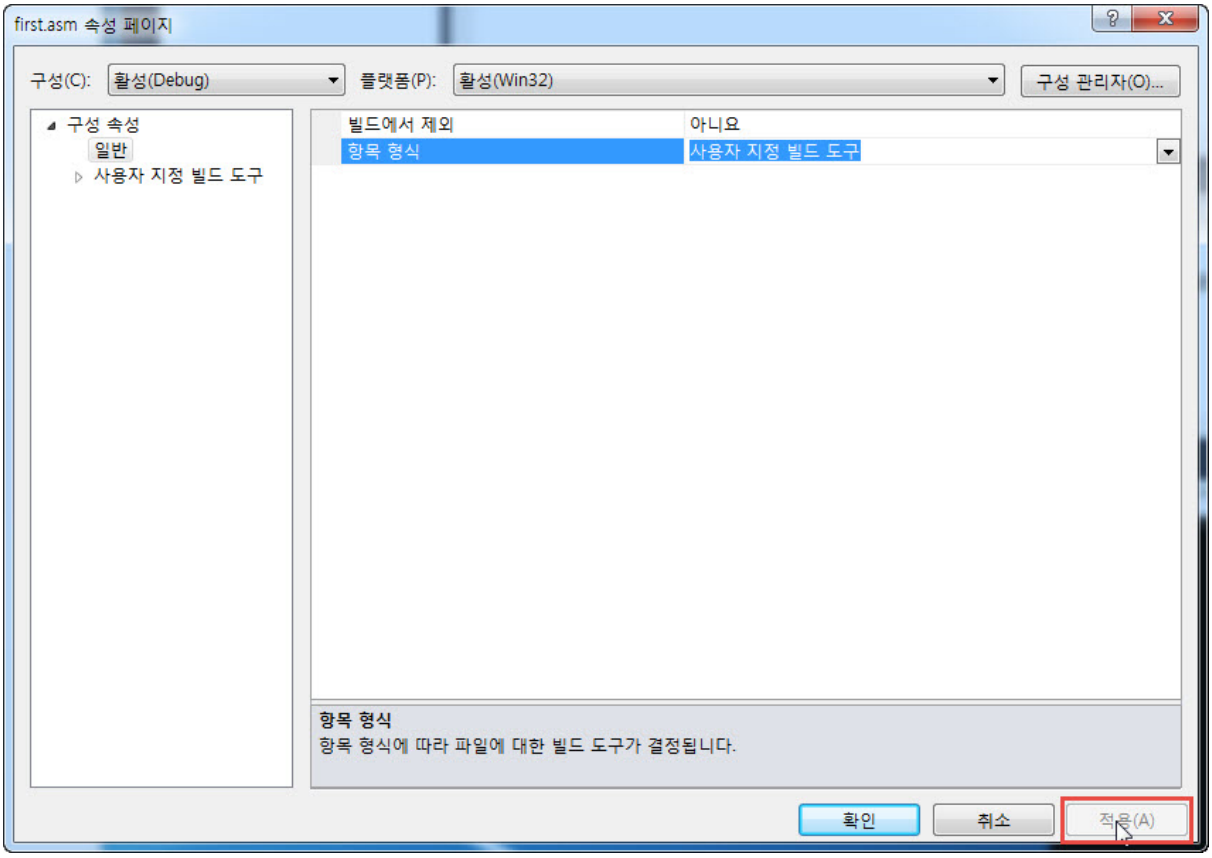
```c
#include <stdio.h>

int asm_main();

int main()
{
	int n = asm_main();

	printf("결과 : %d\n", n);
}
```

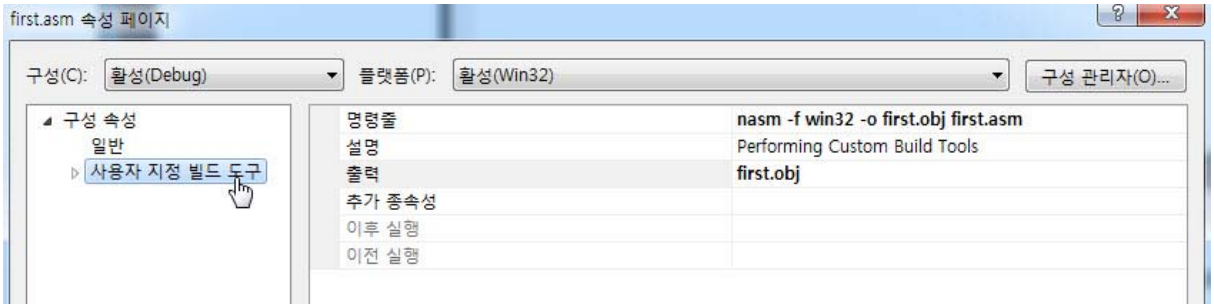
first.asm

```
segment	.data

segment	.text

	global _asm_main

_asm_main:

	mov		eax, 10
	ret
```

2. nasm 과 VC++ 연동하기

- first.asm 에서 오른쪽 버튼 => 속성 메뉴 선택
- 항목 형식에서 “사용자 지정 빌드” 선택후 적용 버튼
- “사용자 지정 빌드 도구” 선택후 “명령줄”/”출력” 항목에 추가





3. 어셈블리에서 전역변수 사용하기

- segment .data
- 전역변수 레이블(L1)은 주소의 의미
- 주소가 가르키는 값을 꺼내려면 dword [L1]
- MASM 의 경우 dword ptr[L1]

```
segment   .data

L1   DD  100           ; int L1 = 100,   DD : Define DWORD
L2   DW  10            ; short L2 = 10
L3   DB  "hello", 0    ; char L3[] = "hello"

segment   .text

        global  _asm_main

_asm_main:

        mov     dword [L1], 200   ; NASM : dword [L1] ,  MASM : dword ptr[L1]

        mov     eax, dword [L1]   ; L1은 주소,    [L1] 은 값.
        ret
```

Item 1 - 04

함수 호출의 원리 2

❑ 핵심개념

- ✓ Jmp 를 사용한 함수 호출 - 돌아올 주소를 알려 주어야 한다.
  - 레지스터를 통해서 돌아올 주소를 알려 주는 방식
  - 스택을 통해서 돌아올 주소를 알려 주는 방식
- ✓ call, ret 을 사용한 함수 호출

1. jmp 를 사용한 함수 호출

레지스터에 돌아올 주소를 담아서 전달

```
_asm_main:
        mov     ebx, A
        jmp     foo
A:
        ret

foo:
        mov     eax, 20
        jmp     ebx
```

stack 에 돌아올 주소를 담아서 전달

```
_asm_main:
        push    A
        jmp     foo
A:
        ret
foo:
        mov     eax, 20
        pop     ebx
        jmp     ebx
```

2. call / ret 를 사용한 함수 호출

- Jmp 의 stack 버전

```
_asm_main:
        call    foo
        ret
foo:
        ret
```

Item 1 - 05

함수 인자 전달 방식

❑ **핵심개념**

- ✓ 레지스터를 통한 인자 전달 – ECX, EDX 레지스터 사용
- ✓ 스택을 통한 인자 전달 – 인자 전달용 스택을 파괴해야 한다.
  - 호출 자(Caller)가 파괴 하는 방식
  - 피호출자(Callee)가 파괴 하는 방식
- ✓ 레지스터 vs 스택 의 장단점

1. 레지스터를 사용한 인자 전달 방식

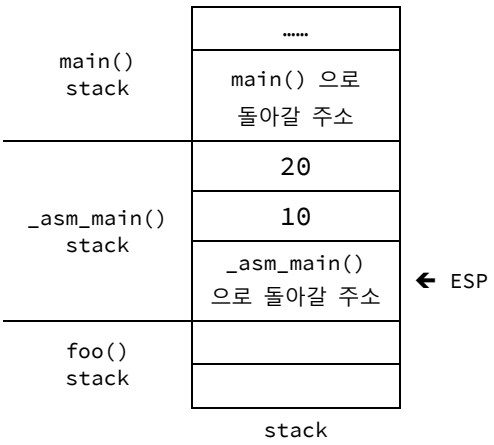
- ECX, EDX 레지스터를 사용해서 인자 전달
- 속도는 빠르지만 인자 개수에 제한이 있다.

```
_asm_main:

        mov     edx, 20
        mov     ecx, 10

        call    foo
        ret

foo:
        mov     eax, ecx   ; eax = ecx
        add     eax, edx   ; eax += edx
        ret
```

2. Stack 을 사용한 인자 전달 방식

- ESP 레지스터에 가장 최근에 사용한 스택의 주소에 있음.
- 왜 Runtime Error 가 발생하는가 ?
- Stack 구조가 중요

```
_asm_main:
        push    20
        push    10
        call    foo
        ret

foo:
        mov     eax, dword[esp+4]
        add     eax, dword[esp+8]

        ret
```



인자 전달에 사용한 stack 은 반드시 파괴 되어야 한다.

3. 인자 전달용 Stack 의 파괴

호출자(caller) 파괴 방식

```
_asm_main:
        push    20
        push    10
        call    foo

        sub     esp, 8
        ret
foo:
        mov     eax, dword[esp+4]
        add     eax, dword[esp+8]
        ret
```

장점 :

단점 :

피호출자(callee) 파괴 방식

```
_asm_main:
        push    20
        push    10
        call    foo
        ret

foo:
        mov     eax, dword[esp+4]
        add     eax, dword[esp+8]

        ret     8
```

장점 :

단점 :

Item 1 - 06

Stack Frame

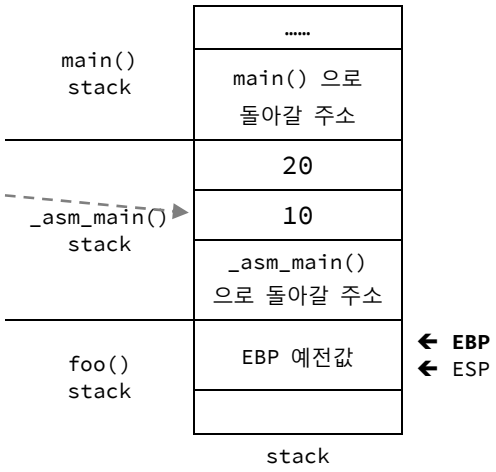
❑ 핵심개념

- ✓ EBP 레지스터를 사용한 Stack 관리
  - dword ptr [ebp+8] : 함수의 1 번째 인자
  - dword ptr [ebp-4] : 함수의 1 번째 지역변수
- ✓ 지역 변수의 원리
- ✓ 함수 Prolog 와 Epilog

1. Stack Frame

- EBP 레지스터를 사용해서 함수 인자 및 지역변수에 접근한다.
- 함수 시작시 EBP 의 현재값을 보관했다가 함수 종료 직전에 복구 해 준다.
- 함수 안에서 Stack 사용시 ESP 는 변하지만 EBP 는 변하지 않는다.
- 고정된 offset 으로 함수 인자에 접근할 수 있다.(EBP+8, EBP+12)

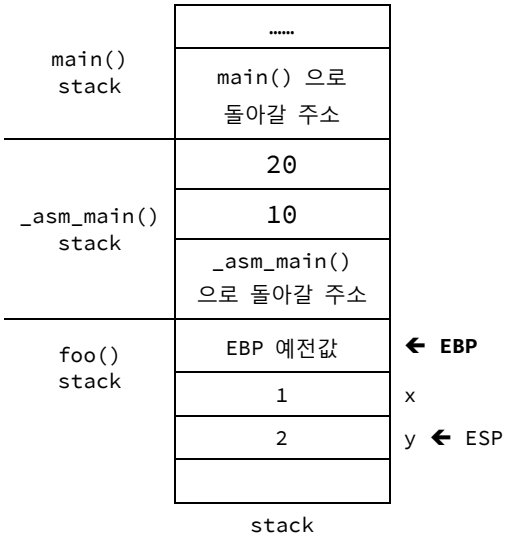
```
_asm_main:

        push    20
        push    10
        call    foo
        add     esp, 8
        ret
foo:
        push    ebp
        mov     ebp, esp

        mov     eax, dword[ebp+8]
        add     eax, dword[ebp+12]

        pop     ebp

        ret
```



2. 지역변수 사용하기

- 필요한 만큼의 stack 공간을 확보 : add esp, 필요한 지역변수 크기
- EBP 레지스터를 사용해서 접근 : EBP - 4
- 함수 호출 종료 시 지역변수가 사용하던 stack 은 파괴 되어야 한다

```
_asm_main:
        push    20
        push    10
        call    foo
        add     esp, 8
        ret
foo:
        push    ebp
        mov     ebp, esp
        sub     esp, 8 ; int x, y

        mov     dword[ebp-4], 1 ; x = 1
        mov     dword[ebp-8], 2 ; y = 2

        mov     eax, dword[ebp+8]
        add     eax, dword[ebp+12]

        mov     esp, ebp ; 지역변수 파괴
        pop     ebp
        ret
```



3. Prolog / Epilog

```
foo:
        ; Prolog
        push    ebp
        mov     ebp, esp
        sub     esp, 8

        ; Epilog
        mov     esp, ebp
        pop     ebp
        ret
```

Item 1 - 07

Coding High Level, Thinking Low Level

❑ 핵심개념

- ✓ C 언어가 만드는 어셈블리 코드 예측하기
- ✓ cl sample.c /FAs 로 어셈블리 소스 확인 하기

1. cl 컴파일러로 어셈블리 소스 확인 하기

- /FAs 옵션을 사용해서 어셈블리 소스 확인 하기
- cl sample.c /FAs
- notepad sample.asm

```
int Add(int a, int b)
{
	int c = 0;
	c = a + b;
	return c;
}
int main()
{
	int n = Add(1, 2);
}
```

2. cl 컴파일러 옵션

- /FAs : 어셈블리 소스를 만들기
- /Ob1 : 인라인 치환 적용
- /O2 : 최적화 적용
- /help : 도움말

Item 1 - 08

Calling Convention

❑ 핵심개념

- ✓ Calling Convention 의 정확한 개념
- ✓ 어셈블리 언어에서의 C 함수를 호출 하는 방법

1. Calling Convention

- 함수 인자가 어디를 통해서 전달되는가?
- 인자 전달용 stack 은 누가 파괴하는가?
- 함수 이름은 어떻게 변경 되는가?

Calling Convention	Argument Passing	Stack Maintenance	Name Decoration	Notes
__cdecl	Stack Right -> Left	호출자(Caller)	_함수이름() ex) _foo	C/C++함수의 기본 호출규약
__stdcall	Sack Right -> Left	피호출자(Callee)	_함수이름@인자크기 ex) _foo@12	Win32 API Visual Basic
__fastcall	2개까지는 ECX, EDX레지스터 사용 나머지는 Stack Right -> Left	피호출자(Callee)	@함수이름@인자크기 ex) @foo@12	Intel CPU
This	Stack Right -> Left this가 ecx레지스터로 전달	호출자(Caller)		C++클래스 COM
naked	Stack Right -> Left	호출자(Caller)		드라이버 개발 Custom Epilog/Prolog 제작

2. 어셈블리 소스로 Calling Convention 확인

main2.c

```c
#include <stdio.h>

void f1(int a, int b) { }

void __stdcall  f2(int a, int b) { }
void __fastcall f3(int a, int b) {}
void __fastcall f4(int a, int b, int c){ }

int main()
{
        f1(1, 2);
        f2(1, 2);
        f3(1, 2);
        f4(1, 2, 3);
}
```

main2.asm

```
_main   PROC
        push    ebp
        mov     ebp, esp

; f1(1, 2)
        push    2
        push    1
        call    _f1
        add     esp, 8
; f2(1, 2)
        push    2
        push    1
        call    _f2@8
; f3(1, 2)
        mov     edx, 2
        mov     ecx, 1
        call    @f3@8
; f4(1, 2, 3)
        push    3
        mov     edx, 2
        mov     ecx, 1
        call    @f4@12
```

3. 어셈블리 소스에서 C 함수 호출하기

- 사용자 정의 함수 호출
- C 표준 함수 호출
- Win32 API 호출

```c
#include <stdio.h>

int asm_main();

int main()
{
	asm_main();
}

void __cdecl    f1(int a, int b)
{
	printf("f1 : %d, %d\n", a, b);
}

void __stdcall  f2(int a, int b)
{
	printf("f2 : %d, %d\n", a, b);
}

void __fastcall f3(int a, int b)
{
	printf("f3 : %d, %d\n", a, b);
}

void __fastcall f4(int a, int b, int c)
{
	printf("f4 : %d, %d, %d\n",
				a, b, c);
}
```

```
segment  .data
S1   DB   "hello", 10, 0   ; "hello/n"

segment   .text
        global  _asm_main
        extern  _f1
        extern  _f2@8
        extern  @f3@8
        extern  @f4@12
        extern  _printf
        extern  _MessageBoxA@16
_asm_main:
; f1(1,2)
        push  2
        push  1
        call  _f1
        add   esp, 8
; f2(1,2)
        push    2
        push    1
        call    _f2@8
; f3(1,2)
        mov             edx, 2
        mov             ecx, 1
        call    @f3@8
; f4(1,2,3)
        push    3
        mov             edx, 2
        mov             ecx, 1
        call    @f4@12
; printf("hello\n")
        push    S1
        call    _printf
        add     esp, 4
; MessageBoxA(0, S1, S1, MB_OK)
        push    0
        push    S1
        push    S1
        push    0
        call    _MessageBoxA@16
        ret
```

Item 1 - 09

반복문과 제어문

❑ 핵심개념

- ✓ FLAG 레지스터
- ✓ Loop 명령과 ECX 레지스터

1. 반복문

LOOP 명령과 ECX 레지스터

```c
#include <stdio.h>

int asm_main();

int main()
{
        int n = asm_main();

        printf("결과 : %d\n", n);
}
```

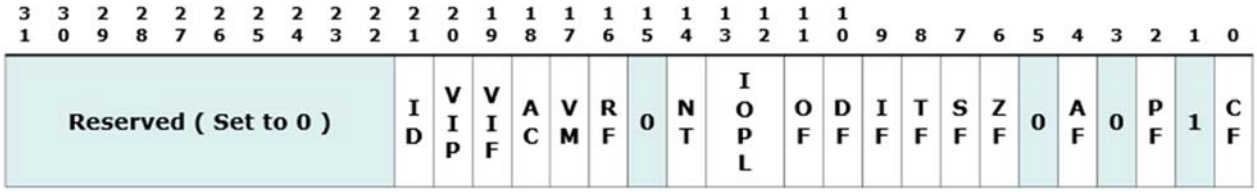
```
segment   .text
        global  _asm_main
_asm_main:
        mov     ecx, 10
        mov     eax, 0
AAA:
        add     eax, ecx ; eax += ecx
        loop    AAA     ; if ( --ecx != 0 )
                        ;       goto AAA
        ret
```

2. LOOP OPCODE

LOOP	Loop With ECX counter
LOOPZ/LOOPNZ	Loop With ECX and zero (not zero)
LOOPE/LOOPNE	Loop With ECX and equal (not equal)

3. FLAG 레지스터

연산의 결과에 따라 각 비트가 set/reset 되는 레지스터



4. FLAG 레지스터와 조건부 JMP

EFLAG 레지스터의 상태에 따라 조건부 JMP를 하는 OPCODE

```
segment   .text

        global  _asm_main

_asm_main:

        mov     ecx, 10
        mov     eax, 0

AAA:
        mov     ebx, ecx
        and     ebx, 1          ; 의미는 ?

        jz      BBB             ; if ( ZF == 1 ) jmp BBB

        add     eax, ecx        ; eax += ecx

BBB:
        loop    AAA             ; if ( --ecx != 0 ) goto AAA

        ret
```

5. Conditional JMP OPCODE

조건에 따라 JMP를 수행하는 OPCODE

JMP	Jump
JE/JNE	Jump if equal (not equal)
JZ/JNZ	Jump if zero (not zero)
JA/JNA	Jump if above (not above)
JAE/JNAE	Jump if above or equal (not above or equal)
JB/JNB	Jump if below (not below)
JBE/JNBE	Jump if below or equal (not below or equal)
JG/JNG	Jump if greater (not greater)
JGE/JNGE	Jump if greater or equal (not greater or equal)
JL/JNL	Jump if less (not less)
JLE/JNLE	Jump if less or equal (not less or equal)
JC/JNC	Jump if carry (not carry)
JO/JNO	Jump if overflow (not overflow)
JS/JNS	Jump if sign (not sign)
JPO/JPE	Jump if parity odd (even)
JP/JNP	Jump if parity (not parity)
JCXZ/JECXZ	Jump if register CX zero (ECX zero)

Item 1 - 10

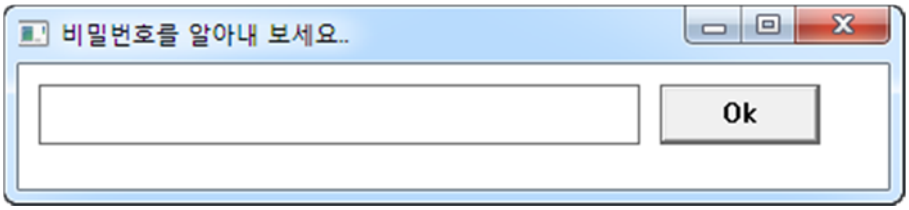
Reversing With OllyDbg

❑ 핵심개념

✓ OllyDbg 사용법
- http://www.ollydbg.de/
- Break Pointer when API call
- Code Memory, Data Memory, Stack, Register

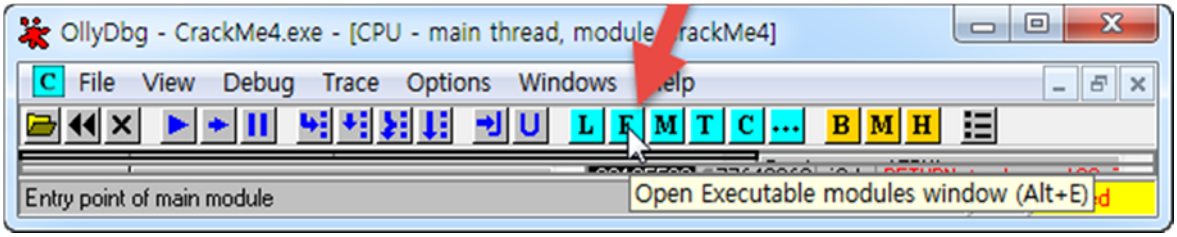
1. CrackMe

- PEView 를 사용해서 .data, .rdata 섹션 조사
- OllyDbg 를 사용해서 Reversing

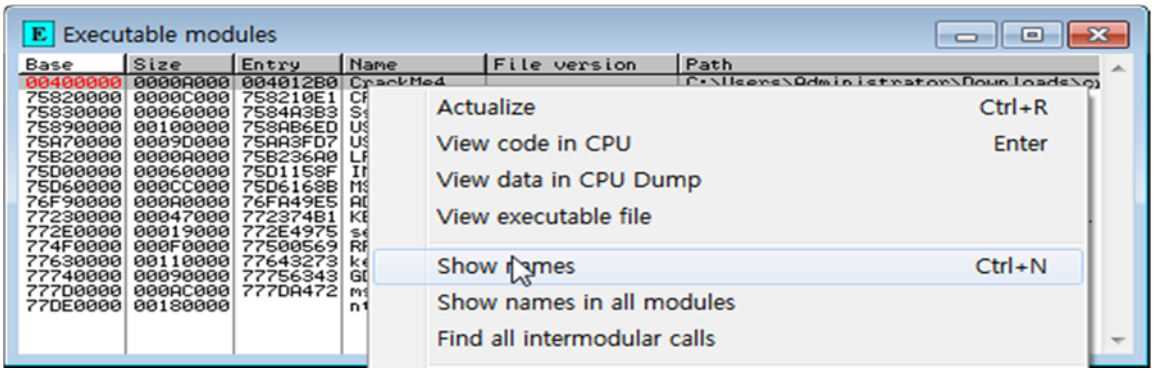


2. Using OllyDbg

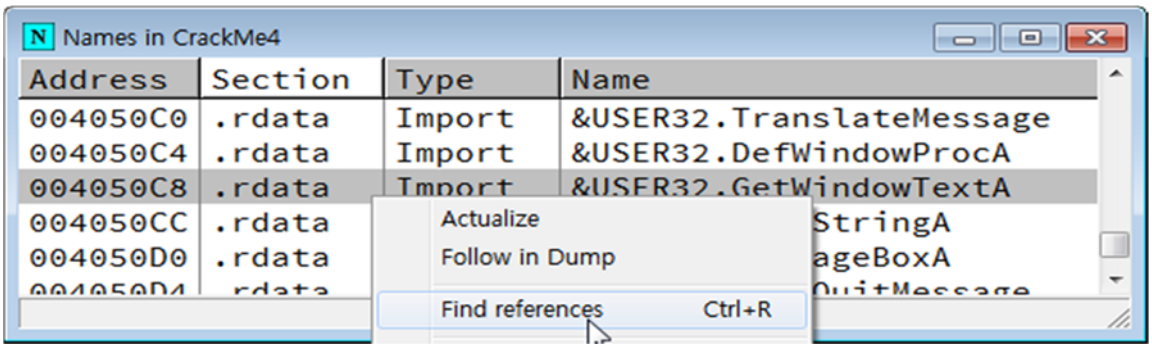
- Breakpoints when API call
- Step 1. 툴바에서 "E" 버튼 클릭



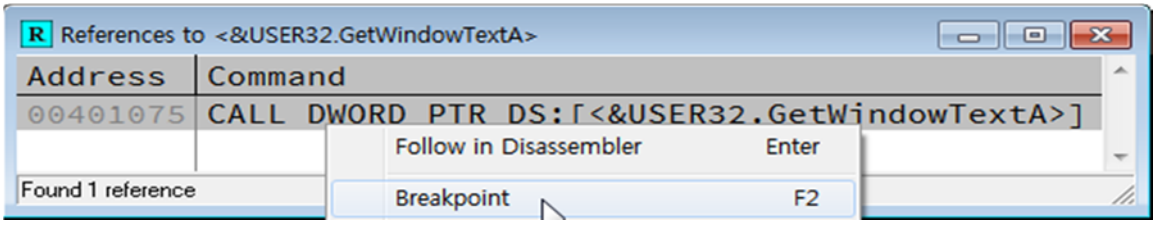
Step 2. 실행파일을 선택하고 오른쪽 버튼 클릭, Show names 메뉴 선택



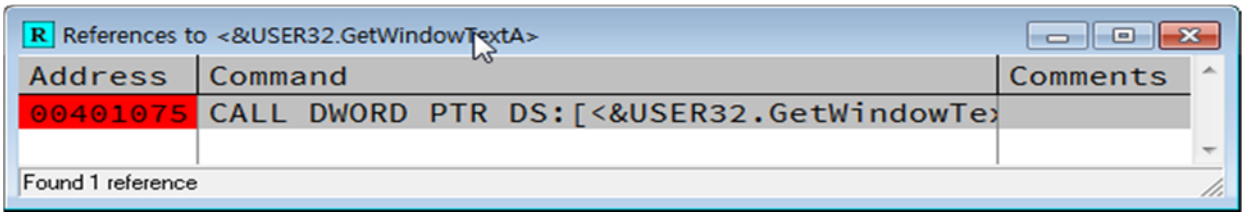
Step 3. GetWindowTextA 함수를 찾아서 오른쪽 버튼 클릭, Find Reference 메뉴 선택



Step 4. 함수 호출 구문에서 오른쪽 버튼 클릭, BreakPoint 메뉴 선택



Step 5. 빨간색으로 변경되었는지 확인



Item 1 - 11

Assembly 활용

❑ 핵심개념

✓ 인라인 어셈블리를 활용한 윈도우 구조체 접근
- TEB, PEB 의 주소 얻어내기
- GetLastError() 구현하기
- 현재 프로세스의 디버깅여부 알아내기

1. TEB(Thread Environment Block) 과 FS 레지스터

스레드당 1개의 구조체, FS 레지스터로 TEB의 주소관리

Position	Length	OS Versions	Description
FS:[0x00]	4	Win9x and NT	Current Structured Exception Handling (SEH) frame
FS:[0x04]	4	Win9x and NT	Stack Base / Bottom of stack (high address)
FS:[0x08]	4	Win9x and NT	Stack Limit / Ceiling of stack (low address)
FS:[0x0C]	4	NT	SubSystemTib
FS:[0x10]	4	NT	Fiber data
FS:[0x14]	4	Win9x and NT	Arbitrary data slot
FS:[0x18]	4	Win9x and NT	Linear address of TIB
---- End of NT subsystem independent part ----			
FS:[0x1C]	4	NT	Environment Pointer
FS:[0x20]	4	NT	Process ID (in some windows distributions this field is used as 'DebugContext')
FS:[0x24]	4	NT	Current thread ID
FS:[0x28]	4	NT	Active RPC Handle
FS:[0x2C]	4	Win9x and NT	Linear address of the thread-local storage array
FS:[0x30]	4	NT	Linear address of Process Environment Block (PEB)
FS:[0x34]	4	NT	Last error number
FS:[0x38]	4	NT	Count of owned critical sections
FS:[0x3C]	4	NT	Address of CSR Client Thread
FS:[0x40]	4	NT	Win32 Thread Information
FS:[0x44]	124	NT, Wine	Win32 client information (NT), user32 private data (Wine), 0x60 = LastError (Win95), 0x74 = LastError (WinME)
FS:[0xC0]	4	NT	Reserved for Wow64. Contains a pointer to FastSysCall in Wow64.
FS:[0xC4]	4	NT	Current Locale

FS:[0xC8]	4	NT	FP Software Status Register
FS:[0xCC]	216	NT, Wine	Reserved for OS (NT), kernel32 private data (Wine)
			herein: FS:[0x124] 4 NT Pointer to KTHREAD (ETHREAD) structure
FS:[0x1A4]	4	NT	Exception code
FS:[0x1A8]	18	NT	Activation context stack
FS:[0x1BC]	24	NT, Wine	Spare bytes (NT), ntdll private data (Wine)
FS:[0x1D4]	40	NT, Wine	Reserved for OS (NT), ntdll private data (Wine)
FS:[0x1FC]	1248	NT, Wine	GDI TEB Batch (OS), vm86 private data (Wine)
FS:[0x6DC]	4	NT	GDI Region
FS:[0x6E0]	4	NT	GDI Pen
FS:[0x6E4]	4	NT	GDI Brush
FS:[0x6E8]	4	NT	Real Process ID
FS:[0x6EC]	4	NT	Real Thread ID
FS:[0x6F0]	4	NT	GDI cached process handle
FS:[0x6F4]	4	NT	GDI client process ID (PID)
FS:[0x6F8]	4	NT	GDI client thread ID (TID)
FS:[0x6FC]	4	NT	GDI thread locale information
FS:[0x700]	20	NT	Reserved for user application
FS:[0x714]	1248	NT	Reserved for GL
FS:[0xBF4]	4	NT	Last Status Value
FS:[0xBF8]	532	NT	Static UNICODE_STRING buffer
FS:[0xE0C]	4	NT	Pointer to deallocation stack
FS:[0xE10]	256	NT	TLS slots, 4 byte per slot
FS:[0xF10]	8	NT	TLS links (LIST_ENTRY structure)
FS:[0xF18]	4	NT	VDM
FS:[0xF1C]	4	NT	Reserved for RPC
FS:[0xF28]	4	NT	Thread error mode (RtlSetThreadErrorMode)

2. TEB 와 PEB 의 주소 얻기

FS:[0x18], FS:[0x30]

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

void* GetTEBAddr()
{
	void* pTEB = 0;
	__asm
	{
		mov	eax, dword ptr FS:[0x18]
		mov	pTEB, eax
	}
	return pTEB;
}

void* GetPEBAddr()
{
```

```
        void* pPEB = 0;
        __asm
        {
                mov  eax, dword ptr FS : [0x30]
                mov  pPEB, eax
        }
        return pPEB;
}

int main()
{
        printf("Current Thread TEB Address : %p\n", GetTEBAddr());
        printf("Current Thread PEB Address : %p\n", GetPEBAddr());
}
```

3. GetLastError() 원리

- 스레드당 한 개의 에러 코드
- TEB 의 Offset 0x34.

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

int MyGetLastError()
{
        int value = 0;
        __asm
        {
                mov   eax, dword ptr FS : [0x34]
                mov   value, eax
        }
        return value;
}
int main()
{
        HWND h = CreateWindow(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0);

        if (h == 0)
        {
                printf("실패 : %d\n", GetLastError());
                printf("실패 : %d\n", MyGetLastError());
        }
}
```

SECTION **02**

Kernel Object

Windows OS는 대부분의 구성요소를 구조체를 사용해서 관리하는 객체 기반 OS입니다.

Windows OS가 만드는 객체는 크게 User Object, GDI Object, Kernel Object 로 구분 합니다. 이중 Kernel Object 는 시스템의 기능을 수행하는 객체로서 다양한 특징을 가지고 있습니다.

본 섹션에서는 Kernel Object의 다양한 특징을 이해하고 관련 기술을 학습합니다.

❑ 주요 내용

- 핸들의 개념을 이해하고 윈도우 핸들을 얻고 핸들을 사용해서 기존 윈도우의 다양한 속성을 변경하는 기법을 학습 합니다.
- User Object, GDI Object, Kernel Object 의 각각의 특징을 학습 합니다..
- Kernel Object 를 관리하는 Object Table 을 학습 합니다.
- 두 개 이상의 Process 간에 Kernel Object 를 공유하는 3 가지 기술을 학습 합니다.

Item 2 - 01

Handle 과 Window

❑ 핵심개념

- ✓ 객체를 구별하는 32/64 비트 정수
- ✓ 핸들과 타입 안정성
- ✓ SetWindowLong / GetWindowLong 을 사용한 Window Object 조작

1. Handle 의 개념

- 객체(윈도우, 펜, 브러시, 폰트등)를 구별하기 위한 고유한 번호 – 32/64 비트 정수
- 핸들을 알면 윈도우를 조작할 수 있다.
- Win32 API = 객체의 고유한 번호(핸들) + 핸들을 가지고 객체를 조작하는 함수

```
#include <Windows.h>
#include <stdio.h>
#include <conio.h>

int main()
{
        HWND hwnd = FindWindowA(0, "계산기");

        printf("계산기 윈도우 번호(핸들): %x\n", hwnd);

        _getch(); MoveWindow(hwnd, 0, 0, 300, 300, TRUE);
        _getch(); ShowWindow(hwnd, SW_HIDE);
        _getch(); ShowWindow(hwnd, SW_SHOW);
        _getch(); SetMenu(hwnd, 0);

        HRGN h = CreateEllipticRgn(0, 0, 300, 300);

        _getch(); SetWindowRgn(hwnd, 0, TRUE);
}
```

2. 핸들의 정체

- 구조체 포인터를 사용해서 서로 다른 종류의 핸들이 암시적 형 변환을 방지
- 타입 안정성을 고려한 설계
- typedef void* HANDLE;

```
struct HWND__ { int unused;};
struct HICON__{ int unused;};

typedef HWND__  *HWND;
typedef HICON__ *HICON;

typedef void* HANDLE;

void MoveWindow(HWND hwnd, int x, int y, int w, int h)
{
}

int main()
{
        HICON h = 0;
        MoveWindow(h, 0, 0, 0, 0);
}
```

Windows.h 에서는 DECLARE_HANDLE( ) 매크로 사용

```
#define DECLARE_HANDLE(name) \
                struct name##__{int unused;}; typedef struct name##__ *name
```

3. Window Object

하나의 윈도우를 관리하기 위해 생성되는 객체

"Windows 95 System Programing Secrets" , matt pietrick

```
typedef struct _WND32
{
        struct _WND32  *hWndNext;     // 00h (GW_HWNDNEXT) HWND of next sibling window
        struct _WND32  *hWndChild;    // 04h (GW_CHILD) First child window
        struct _WND32  *hWndParent;   // 08h Parent window handle
        struct _WND32  *hWndOwner;    // 0Ch Owning window handle
        RECTS          rectWindow;     // 10h Rectangle describing entire window
        RECTS   rectClient;        // 18h Rectangle for client area of window
        WORD    hQueue;           // 20h Application message queue handle
        WORD    hrgnUpdate;        // 22h window region needing an update
        WORD    wndClass;         // 24h handle to an INTWNDCLASS
        WORD    hInstance;         // 26h hInstance of creating application
        WNDPROC        lpfnWndProc; // 28h Window procedure address
```

```
        DWORD   dwFlags;            // 2Ch internal state flags
        DWORD   dwStyleFlags;      // 30h WS_XXX style flags
        DWORD   dwExStyleFlags;    // 34h WS_EX_XXX extended style flags
        DWORD   moreFlags;         // 38h flags
        HANDLE  ctrlID;            // 3Ch GetDlgCtrlId or hMenu
        WORD    windowTextOffset;  // 40h Offset of the window's text in atom heap
        WORD    scrollBar;         // 42h DWORD associated with the scroll bars
        WORD    properties;        // 44h Handle for first window property
        WORD    hWnd16;            // 46h Actual HWND value for this window
        struct _WND32* lastActive; // 48h Last active owned popup window
        HANDLE  hMenuSystem;       // 4Ch handle to the system menu
        DWORD   un1;               // 50h
        WORD    un2;               // 54h
        WORD    classAtom;         // 56h See also offs. 2 in the field 24 struct ptr
        DWORD   alternatePID;      // 58h
        DWORD   alternateTID;      // 5Ch
} WND32, *PWND32;
```

4. SetWindowLong() / GetWindowLong()

Window 구조체를 속성을 변경하는 함수

```
#include <Windows.h>
#include <stdio.h>
#include <conio.h>

void ModifyStyle(HWND hwnd, UINT remove, UINT add)
{
        int style = GetWindowLong(hwnd, GWL_STYLE);
        style = style | add;
        style = style & ~remove;
        SetWindowLong(hwnd, GWL_STYLE, style);
        SetWindowPos(hwnd, 0, 0, 0, 0, 0,
                SWP_NOMOVE | SWP_NOSIZE | SWP_NOZORDER | SWP_DRAWFRAME);
}
int main()
{
        HWND hwnd = FindWindowA(0, "계산기");
        _getch(); ModifyStyle(hwnd, WS_CAPTION,    WS_THICKFRAME);
        _getch(); ModifyStyle(hwnd, WS_THICKFRAME, WS_CAPTION);
}
```

Item 2 - 02

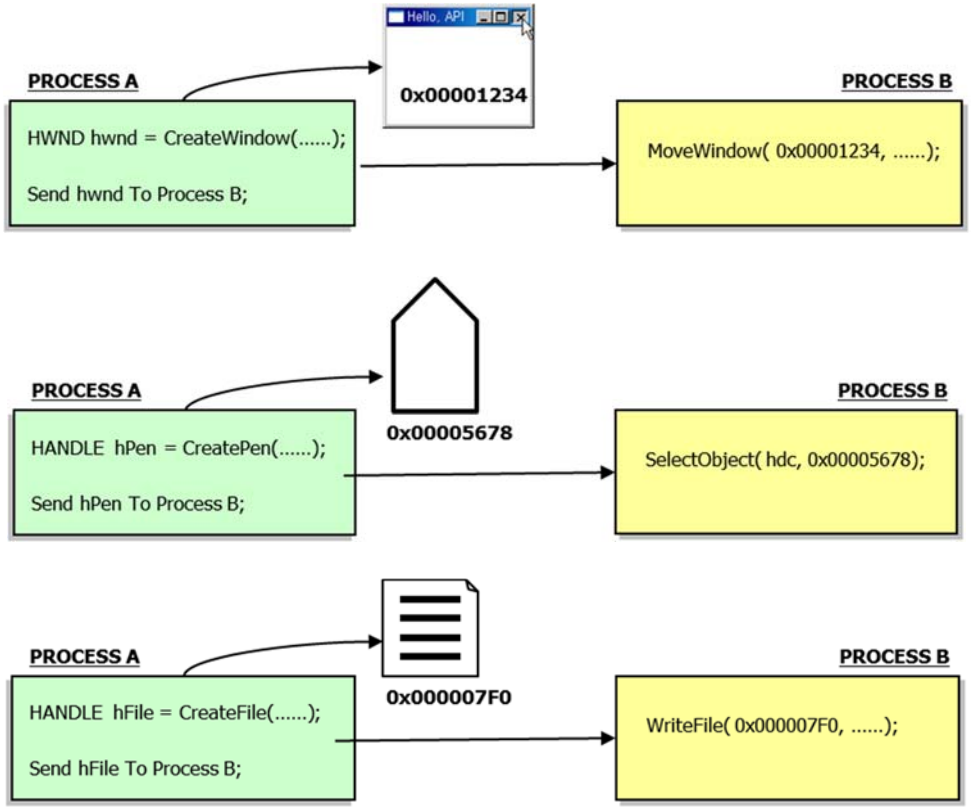
Object Category

❑ 핵심개념

- ✓ Object 의 3 가지 종류 와 특징
  - User Object, GDI Object, Kernel Object
- ✓ Kernel Object 의 특징

1. 프로세스간 핸들 공유

Window, Pen, File 핸들의 공유 문제



2. Object Category

Windows OS가 만드는 Object 의 3가지 종류

	User Object	GDI Object	Kernel Object
특징	윈도우와 관련된 Object.	그래픽 관련된 Object.	파일, 메모리, 프로세스, IPC 등과 같은 작업에 관련된 Object.
핸들의 특징	Public to All Process	Private	**상대적(Specific) 핸들**
파괴함수	DestroyXXX()	DeleteXXX()	CloseHandle() 참조계수 감소
관련 DLL	User32.dll	GDI32.dll	Kernel32.dll
종류	Accelerator table, Caret, Cursor, DDE conversation, Hook, Icon, Menu, Window, Window position	Bitmap, Brush, DC, Enhanced metafile, Enhanced-metafile DC, Font, Memory DC, Metafile, Metafile DC, Palette, Pen and extended pen, Region	Access token, Change notification, Communications, device, Console input, Console screen buffer, Desktop, Event,Event log, File, File mapping, Heap, Job, Mailslot, Module, Mutex, Pipe, Process, Semaphore, Socket, Thread, Timer, Timer queue, Timer-queue timer, Update resource, Window station

3. Kernel Object 특징

보안속성, 이름, 참조계수, Signal, WaitList

보안속성
이름
참조계수
Signal
Wait List

Item 2 - 03

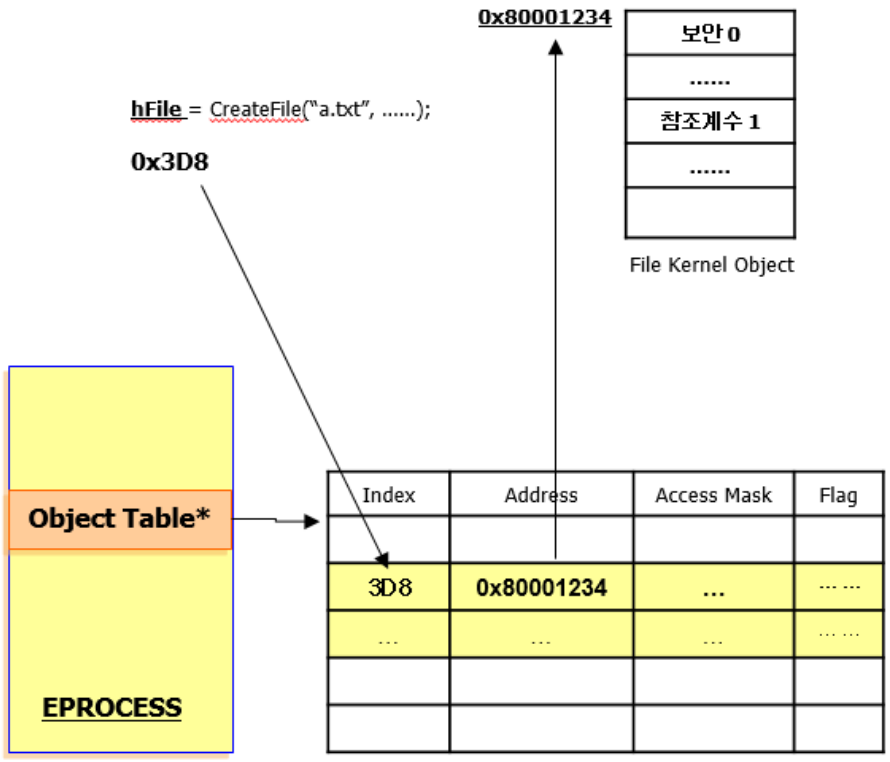
Kernel Object Handle Table

❑ 핵심개념

- ✓ EPROCESS 와 Object Table
- ✓ Process Explorer Object Table 조사 하기
- ✓ Native API 를 사용해서 Object Table 열거 하기

1. Process's Object Table

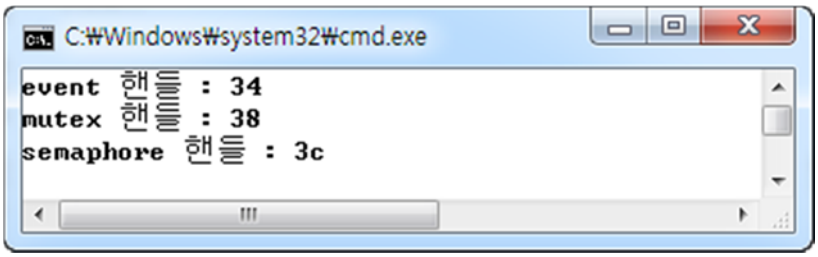
- 프로세스가 사용하는 Kernel Object 의 핸들을 관리 하는 Table
- User Object 와 GDI Object 는 포함되지 않는다.

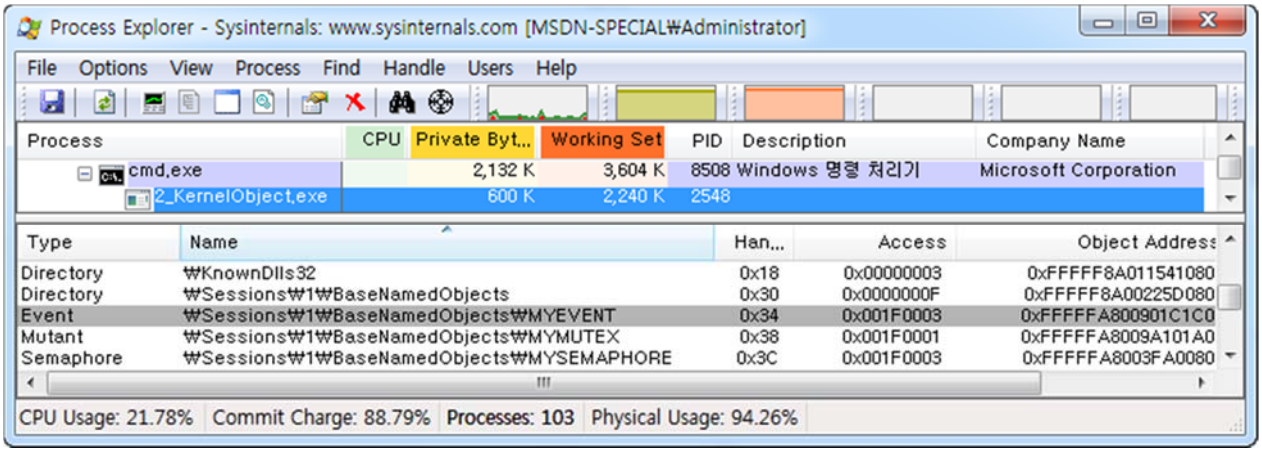


2. Process Explorer Utility

https://technet.microsoft.com/ko-kr/sysinternals/bb896653.aspx

```
#include <Windows.h>
#include <stdio.h>
#include <conio.h>

int main()
{
	_getch();
	HANDLE hEvent = CreateEventA( 0, 0, 0, "MYEVENT");
	printf("event 핸들 : %x\n", hEvent);
	_getch();
	HANDLE hMutex = CreateMutexA( 0, 0, "MYMUTEX");
	printf("mutex 핸들: %x\n", hMutex);
	_getch();
	HANDLE hSemaphore = CreateSemaphoreA( 0, 0, 3, "MYSEMAPHORE");
	printf("semaphore 핸들: %x\n", hSemaphore);
	_getch(); CloseHandle( hEvent);
	_getch(); CloseHandle( hMutex);
	_getch(); CloseHandle( hSemaphore);
}
```





3. Native API 를 사용한 Object Handle Table 열거 하기

ZwQuerySystemInformation를 사용한 Process Object Table 열거

```
#include <stdio.h>
#include <Windows.h>
#include <conio.h>

#define STATUS_INFO_LENGTH_MISMATCH     ((NTSTATUS)0xC0000004L)

typedef enum _SYSTEM_INFORMATION_CLASS {
    SystemBasicInformation,             // 0
    SystemProcessorInformation,         // 1
    SystemPerformanceInformation,       // 2
    SystemTimeOfDayInformation,         // 3
    SystemNotImplemented1,              // 4
    SystemProcessesAndThreadsInformation, // 5
    SystemCallCounts,                   // 6
    SystemConfigurationInformation,     // 7
    SystemProcessorTimes,               // 8
    SystemGlobalFlag,                   // 9
    SystemNotImplemented2,              // 10
    SystemModuleInformation,            // 11
    SystemLockInformation,              // 12
    SystemNotImplemented3,              // 13
    SystemNotImplemented4,              // 14
    SystemNotImplemented5,              // 15
    SystemHandleInformation,            // 16
    SystemObjectInformation,            // 17
    SystemPagefileInformation,          // 18
    SystemInstructionEmulationCounts,   // 19
    SystemInvalidInfoClass1,            // 20
    SystemCacheInformation,             // 21
    SystemPoolTagInformation,           // 22
    SystemProcessorStatistics,          // 23
    SystemDpcInformation,               // 24
    SystemNotImplemented6,              // 25
    SystemLoadImage,                    // 26
    SystemUnloadImage,                  // 27
    SystemTimeAdjustment,               // 28
    SystemNotImplemented7,              // 29
    SystemNotImplemented8,              // 30
    SystemNotImplemented9,              // 31
    SystemCrashDumpInformation,         // 32
    SystemExceptionInformation,         // 33
    SystemCrashDumpStateInformation,    // 34
    SystemKernelDebuggerInformation,    // 35
    SystemContextSwitchInformation,     // 36
    SystemRegistryQuotaInformation,     // 37
    SystemLoadAndCallImage,             // 38
    SystemPrioritySeparation,           // 39
    SystemNotImplemented10,             // 40
    SystemNotImplemented11,             // 41
    SystemInvalidInfoClass2,            // 42
    SystemInvalidInfoClass3,            // 43
    SystemTimeZoneInformation,          // 44
    SystemLookasideInformation,         // 45
    SystemSetTimeSlipEvent,             // 46
```

```
    SystemCreateSession,                // 47
    SystemDeleteSession,                // 48
    SystemInvalidInfoClass4,            // 49
    SystemRangeStartInformation,        // 50
    SystemVerifierInformation,          // 51
    SystemAddVerifier,                  // 52
    SystemSessionProcessesInformation   // 53
} SYSTEM_INFORMATION_CLASS;

typedef struct _SYSTEM_HANDLE_INFORMATION {
    ULONG ProcessId;
    UCHAR ObjectTypeNumber;
    UCHAR Flags;
    USHORT Handle;
    PVOID Object;
    ACCESS_MASK GrantedAccess;
} SYSTEM_HANDLE_INFORMATION, *PSYSTEM_HANDLE_INFORMATION;

typedef NTSTATUS (__stdcall *F)(SYSTEM_INFORMATION_CLASS,PVOID ,ULONG ,PULONG);

F ZwQuerySystemInformation = 0;

int main()
{
        HMODULE hDll = GetModuleHandleA("Ntdll.dll");
        ZwQuerySystemInformation = (F)GetProcAddress(hDll, "ZwQuerySystemInformation");

        int sz = sizeof(SYSTEM_HANDLE_INFORMATION);
        int count = 30000;
        void* buff = malloc(sz * count + 4);

        DWORD len;
        DWORD ret = ZwQuerySystemInformation(SystemHandleInformation, buff,
                                                sz * count + 4, &len);
        if (ret == STATUS_INFO_LENGTH_MISMATCH)
        {
                printf("버퍼가 작습니다..\n");
        }
        int n = *(int*)buff;
        printf("핸들 테이블 항목 개수 : %d\n", n);
        SYSTEM_HANDLE_INFORMATION* pHandle =
                                        (SYSTEM_HANDLE_INFORMATION*)((char*)buff + 4);
        for (int i = 0; i < n; i++)
        {
                printf("PID : %d, HANDLE : %x, TYPE : %d\n", pHandle[i].ProcessId,
                        pHandle[i].Handle, pHandle[i].ObjectTypeNumber);

        }
}
```

Item 2 - 04

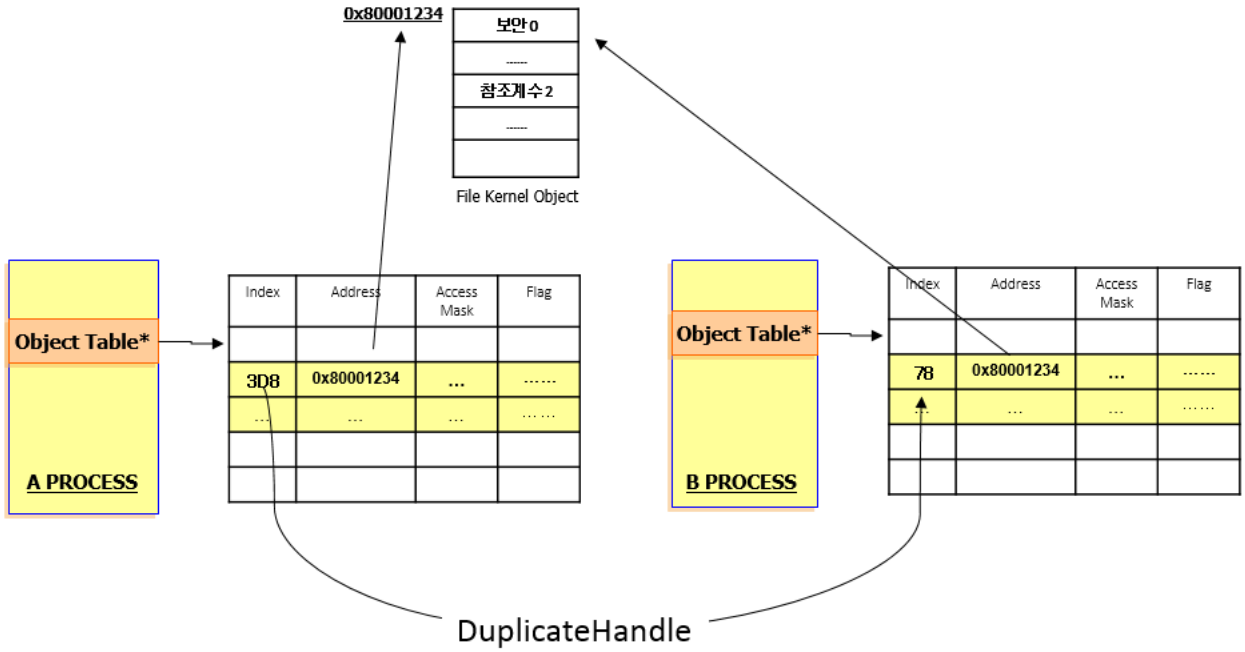
프로세스간 Kernel Object 공유 1

DuplicateHandle

❑ 핵심개념

- ✓ 프로세스간 Kernel Object Handle 을 복사하는 방법
- ✓ DuplicateHandle() 함수 사용법

1. 프로세스간 Kernel Object Handle 의 핸들 복사 개념



2. DuplicateHandle()을 사용한 Object Handle 복사 예제

A Process

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

int main()
{
	HANDLE hFile = CreateFileA("a.txt", GENERIC_READ | GENERIC_WRITE,
		FILE_SHARE_READ | FILE_SHARE_WRITE, 0, CREATE_ALWAYS,
		FILE_ATTRIBUTE_NORMAL, 0);

	printf("File Handle : %x\n", hFile);

	_getch();

	HWND hwnd = FindWindow(0, "B");
	DWORD pid;
	DWORD tid = GetWindowThreadProcessId(hwnd, &pid);
	HANDLE hProcess = OpenProcess(PROCESS_ALL_ACCESS, 0, pid);

	HANDLE h2;
	DuplicateHandle(GetCurrentProcess(), hFile, hProcess, &h2,
				0, 0, DUPLICATE_SAME_ACCESS);
	printf("B 프로세스에 복사해준 핸들(table index) : %x\n", h2);

	_getch();
	SendMessage(hwnd, WM_APP + 100, 0, (LPARAM)h2);
}
```

B Process – GUI Application

```
case WM_APP + 100:
	{
		HANDLE hFile = (HANDLE)lParam;
		char s[256] = "hello";
		DWORD len;
		BOOL b = WriteFile(hFile, s, 256, &len, 0);
		MessageBoxA( 0, b ? "성공":"실패",  "", 0);
	}
	return 0;
```

Item 2 - 05

프로세스간 Kernel Object 공유 2

Named Object

❑ 핵심개념

- ✓ CreateXXX() 함수와 OpenXXX() 함수의 차이점
- ✓ ERROR_ALREADY_EXISTS
- ✓ Kernel Object 에 상관없이 같은 이름 공간을 사용

1. 실행파일의 구조

동일한 이름의 Kernel Object 는 하나 이상 만들 수 없다.

해당 이름의 Kernel Object 가 이미 존재 할 경우 Open

2번 이상 실행

```
#include <Windows.h>
#include <stdio.h>
#include <conio.h>

int main()
{
	HANDLE hEvent = CreateEventA( 0, 0, 0, "MYEVENT");

	if ( GetLastError() == ERROR_ALREADY_EXISTS )
		printf("Open Event Handle\n");
	else
		printf("Create Event\n");
	_getch();
}
```

2. Create vs Open

Create : 해당 이름의 객체가 없으면 생성, 이미 존재하면 오픈

Open : 해당 이름의 객체가 없으면 실패, 이미 존재하면 오픈

Kernel Object 의 종류가 달라도 같은 이름을 사용하면 안된다.

```
HANDLE h1 = OpenEventA( EVENT_ALL_ACCESS, 0, "MYEVENT");    // Fail
HANDLE h2 = CreateEventA( 0, 0, 0, "MYEVENT");              // success ! Create
HANDLE h2 = CreateEventA( 0, 0, 0, "MYEVENT");              // success ! but Open
HANDLE h3 = OpenEventA( EVENT_ALL_ACCESS, 0, "MYEVENT");    // success ! Open
HANDLE h4 = CreateMutexA(0, 0, "MYEVENT");                  // fail
```

Item 2 - 06

프로세스간 Kernel Object 공유 3

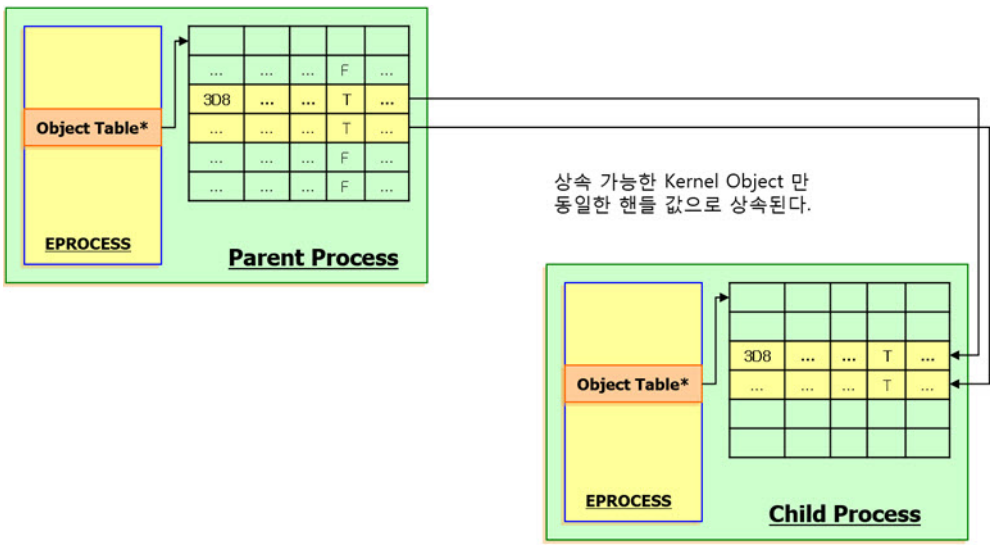
Inherit Kernel Object

❑ 핵심개념

- ✓ 커널 오브젝트 상속의 개념
- ✓ 상속 가능한 커널 오브젝트를 만드는 2 가지 방법
  - ■ 객체 생성시 SECURITY_ATTRIBUTES 구조체 사용
  - ■ 객체 생성후 SetHandleInformation() 함수 사용
- ✓ 파이프를 사용한 자식 프로세스의 출력을 Redirect 하는 방법
- ✓ Console Application 의 출력을 윈도우 창으로 보내는 방법
- ✓ ZwQueryObject() 함수를 사용해서 Kernel Object 의 상속여부 알아내기

1. Kernel Object 상속의 개념

- 부모 프로세스가 사용하던 커널 오브젝트를 자식 프로세스에게 상속 할 수 있다.
- CreateProcess()의 5 번째 파라미터를 TRUE 로 전달한다.
- 상속가능한 커널 오브젝트만 상속할 수 있다



2. 상속 가능한 Kernel Object 를 만드는 2 가지 방법

1. 보안 속성을 지정하는 구조체인 SECURITY_ATTRIBUTES 의 bInherit 항목을 TRUE 로 설정한 후 Kernel Object 를 생성한다

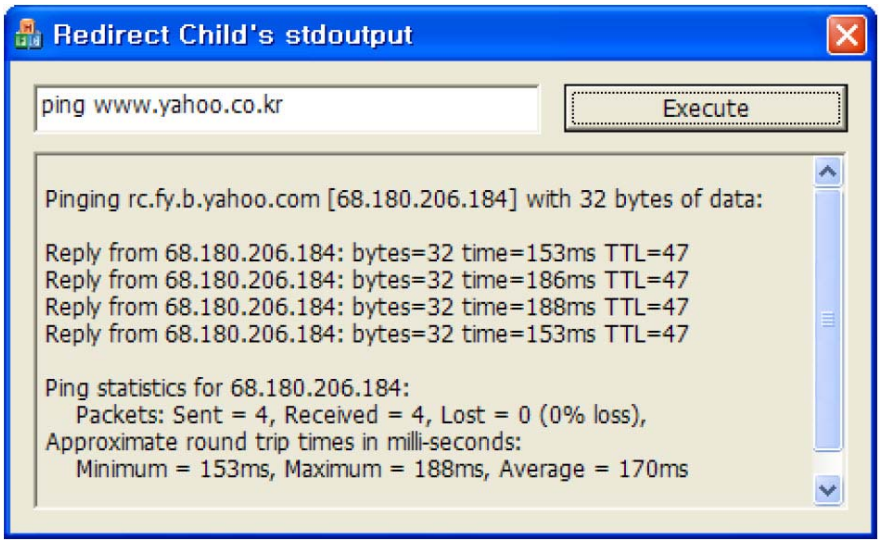
```
SECURITY_ATTRIBUTES sa;
sa.nLength   = sizeof(sa);
sa.bInheritHandle   = TRUE;
sa.lpSecurityDescriptor   = 0;

HANDLE hFile = CreateFile(_T("a.txt"), GENERIC_WRITE | GENERIC_READ, 0,
                          FILE_SHARE_READ | FILE_SHARE_WRITE,
                          &sa, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, 0);
```

2. 이미 생성된 Kernel Object 에 SetHandleInformation() 함수로 상속가능 여부를 지정한다.

```
HANDLE hFile = CreateFile(_T("a.txt"), GENERIC_WRITE | GENERIC_READ,
                          FILE_SHARE_READ | FILE_SHARE_WRITE,
                          0, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, 0);

SetHandleInformation(hFile, HANDLE_FLAG_INHERIT, HANDLE_FLAG_INHERIT);
```

3. Redirect Child Process's stdout

Console Application 의 출력을 Window에 출력하는 기술



- 익명의 파이프를 사용해서 자식 프로세스의 출력을 부모 프로세스에 연결한다.
- 파이프의 쓰기 핸들을 자식 프로세스에 상속 한다.
- 부모에서는 파이프의 쓰기 핸들을 반드시 닫아야 한다.

```
void ExecutePing( const TCHAR* url, HWND hEdit)
{
	TCHAR cmd[256] = _T("ping.exe ");
	_tcscat(cmd, url);

	HANDLE hReadPipe, hWritePipe;
	CreatePipe(&hReadPipe, &hWritePipe, 0, 1024);
	SetHandleInformation(hWritePipe, HANDLE_FLAG_INHERIT,
                                     HANDLE_FLAG_INHERIT);
	PROCESS_INFORMATION pi = {0};
	STARTUPINFO si = { sizeof(si) };
	si.hStdOutput = hWritePipe;
	si.dwFlags = STARTF_USESTDHANDLES;

	BOOL bRet = CreateProcess(0, cmd, 0, 0, TRUE, CREATE_NO_WINDOW, 0, 0,
                                                                  &si, &pi);
	if (bRet)
	{
		CloseHandle(hWritePipe);

		while (1)
		{
			char buffer[1024] = { 0 };
			DWORD len;
			ReadFile(hReadPipe, buffer, 1024, &len, 0);

			if (len <= 0) break;

			SendMessage(hEdit, EM_REPLACESEL, 0, (LPARAM)buffer);
		}
		CloseHandle(hReadPipe);
		CloseHandle(pi.hProcess);
		CloseHandle(pi.hThread);
	}
}
```

4. Kernel Object 의 상속 가능여부 조사하기

Native API 인 ZwQueryObject()를 사용하면 Kernel Object의 상속여부를 조사할 수 있다.

```
#include <Windows.h>
#include <tchar.h>
#include <stdio.h>
```

```
typedef enum _OBJECT_INFORMATION_CLASS {
        ObjectBasicInformation,
        ObjectNameInformation,
        ObjectTypeInformation,
        ObjectAllTypesInformation,
        ObjectHandleInformation
} OBJECT_INFORMATION_CLASS;

typedef struct _OBJECT_HANDLE_ATTRIBUTE_INFORMATION {
        BOOLEAN Inherit;
        BOOLEAN ProtectFromClose;
} OBJECT_HANDLE_ATTRIBUTE_INFORMATION, *POBJECT_HANDLE_ATTRIBUTE_INFORMATION;

typedef NTSTATUS (__stdcall *PFZwQueryObject)
                        (HANDLE,OBJECT_INFORMATION_CLASS,PVOID,ULONG, PULONG );

int main()
{
        PFZwQueryObject ZwQueryObject = (PFZwQueryObject)
                GetProcAddress( GetModuleHandle(_T("Ntdll.dll")),
                                "ZwQueryObject");

        ULONG len;
        OBJECT_HANDLE_ATTRIBUTE_INFORMATION ohai = {0};

        HANDLE hEvent = CreateEvent( 0, 0, 0, 0 );

        SetHandleInformation( hEvent, HANDLE_FLAG_INHERIT, HANDLE_FLAG_INHERIT);
        ZwQueryObject(hEvent, ObjectHandleInformation, &ohai, sizeof(ohai), &len);
        printf("inheritable : %d\n", ohai.Inherit);

        SetHandleInformation( hEvent, HANDLE_FLAG_INHERIT, 0);
        ZwQueryObject(hEvent, ObjectHandleInformation, &ohai, sizeof(ohai), &len);
        printf("inheritable : %d\n", ohai.Inherit);
}
```