Windows Debugging

(2) x86 stack inside

2015.03

레드스톤소프트

김태형 부장

http://kuaaan.tistory.com

IRON

RedStonesoft

Register 기초

X86 Register

특별한 목적으로 사용되는 레지스터

- ✓ EIP : 다음에 실행될 Instruction의 주소를 pointing
- ✓ EAX : 함수가 리턴할때 리턴값을 저장. 평소에는 범용으로 사용됨
- ✓ ESP : 현재 스택의 가장 위(마지막 push된 원소)를 pointing
  Stack에 새로운 원소가 PUSH될 때마다 ESP 감소. POP될 때마다 ESP 증가
- ✓ EBP : 현재 스택의 가장 바닥(Base)를 pointing

나머지 레지스터.

- ✓ EBX, ECX, EDX, ESI, EDI 등등... (이런게 있구나...)

Volatile Register .vs. Non-Volatile Register

Volatile Register

- ✓ 다른 함수를 호출하고 나면 값이 달라지는 레지스터
- ✓ EAX, ECX, EDX, ST0 - ST7, ES and GS

Non-Volatile Register

- ✓ 다른 함수를 호출한 후에도 값이 유지되는 레지스터
- ✓ EBX, EBP, ESP, EDI, ESI, CS and DS
- ✓ Non-Volatile Register를 다른 목적으로 사용하려면 먼저 Stack에 원래 값을 백업
  ➔ 함수가 리턴되기 직전에 백업받은 원래 값을 Non-Volatile Register에 복원

Non-Volatile Register를 사용 전에 백업받는 부분

```
KERNELBASE!CreateFileW+0x5e:
76d7c2f7 53              push    ebx
76d7c2f8 56              push    esi
76d7c2f9 8b7508            mov     esi,dword ptr [ebp+8]
76d7c2fc 56              push    esi
76d7c2fd 8d45d8            lea     eax,[ebp-28h]
76d7c300 50              push    eax
```

백업된 Non-Volatile Register를 복원하는 부분

```
KERNELBASE!CreateFileW+0x48f:
76d7c728 5f              pop     edi

KERNELBASE!CreateFileW+0x490:
76d7c729 5e              pop     esi
76d7c72a 5b              pop     ebx

KERNELBASE!CreateFileW+0x492:
76d7c72b c9              leave
76d7c72c c21c00            ret     1Ch
```

Assembly 기초

Assembly를 공부해야 하는 이유??

대부분은... '분석'만 할줄 알면 된다.

- ✓ api 안에서 죽었는데... 왜 죽었는지 모르겠을 때.
- ✓ OS 의 동작 원리가 궁금할 때. (Windows OS이 모듈이나 api 분석)
- ✓ 다른 모듈을 Reversing 해야 할 때 (악성 코드??)
- ✓ pdb가 없는 모듈을 디버깅할 때
- ✓ 프로그램이 코드와 부합하지 않는 동작을 할 때...

가끔은… '작성'해야 할 때도 있다.

- ✓ 후킹 코드 작성시

opcode, operand

- ✓ opcode : assembly 의 명령어 (= instruction)
- ✓ operand : opcode의 연산을 수행하는데 필요한 Data
- ✓ operand가 두개인 경우 오른쪽 operand가 src, 왼쪽 operand가 dst. (disassembler 의 종류에 따라 다를수도 있다.)

```
leave    ; opcode only
dec      eax        ; opcode + 1 operand
mov dword ptr [ebp+0Ch], 40000000h ; opcode + 2 operands
                    dst operand    src operand
```

dword ptr [ ]

- ✓ src에 사용될 때는 "[ ] 안에 있는 주소값이 가리키는 메모리에 저장된 값"을 의미. (pointer indirection)
- ✓ dst에 사용될 때는 "[ ] 안에 있는 주소값이 가리키는 메모리에 저장하라"는 의미.
- ✓ "dword" 는 포인터의 사이즈를 의미. (char* ➔ byte ptr[], short* ➔ word ptr[])
- ✓ 디스어셈블러의 종류에 따라 dword ptr [] 를 [] 로 표현하는 경우도 있으므로, [] 안의 값을 그대로 참조할지 [] 안의 주소가 가리키는 메모리에 저장된 값을 참조할지 여부는 "dword ptr" 키워드의 유무가 아니라 "operator의 종류"에 따라 판단해야 함.

```
mov BYTE PTR [ebx], 2;
mov WORD PTR [ebx], 2;
mov DWORD PTR [ebx], 2;
```

MOV, LEA

mov

- ✓ 우변의 '값'을 좌변에 대입
- ✓ 좌변에는 레지스터와 메모리 주소가 모두 가능, 우변에는 레지스터 / 메모리 주소 / 상수가 모두 가능 (우변 / 좌변에 모두 주소가 올 수는 없음.)
- ✓ 우변에 주소가 올 경우 주소값 대신 해당 주소에 저장된 '값'이 좌변에 대입됨.

```
mov     dword ptr [ebp-38h],2 ; 2라는 상수값을 ebp-38h 위치에 저장
mov     eax,dword ptr [ebp+14h] ; ebp+14h에 저장된 '값'을 eax에 저장
mov     dword ptr [ebp-54h],esi ; esi 에 들어있는 값을 ebp-54h 위치에 저장
```

lea

- ✓ 우변의 '주소값'을 좌변에 대입
- ✓ 좌변에는 레지스터만 가능, 우변에는 메모리 주소 사용 가능

```
lea     eax,[ebp-20h] ; ebp의 주소값에서 20h 뺀 '주소값'을 eax에 저장
```

PUSH, POP

push

- ✓ 주어진 operand 값을 스택의 꼭대기에 push함.
- ✓ operand 에 레지스터, 상수, 주소 값이 모두 올수 있음.
- ✓ operand에 주소가 올 경우 해당 주소에 저장된 '값'이 대입됨.
- ✓ push dword ptr [ebp-8]
  - ➔ sub esp, 4h / mov eax, dword ptr [ebp-8] / mov dword ptr [esp], eax

```
push    edi
push    0B7h
push    dword ptr [ebp-8]
```

pop

- ✓ 현재 스택의 꼭대기 값을 pop해서 주어진 레지스터에 대입함.
- ✓ pop edi ➔ mov edi, dword ptr [esp] / add esp 4h

```
pop esi
pop dword ptr [ebp-8]
```

Flag

비교문(test, cmp) 수행시 결과가 Flag에 임시 저장됨.
Conditional jump 시 조건 판단을 위해 Flag값을 참조함.
test나 cmp 결과가 어딘가에 저장되는구나... 정도로 이해!!

- ✓ CF - carry flagSet on high-order bit carry or borrow; cleared otherwise
- ✓ PF - parity flagSet if low-order eight bits of result contain an even number of "1" bits; cleared otherwise
- ✓ ZF - zero flagsSet if result is zero; cleared otherwise
- ✓ SF - sign flagSet equal to high-order bit of result (0 if positive 1 if negative)
- ✓ OF - overflow flagSet if result is too large a positive number or too small a negative number (excluding sign bit) to fit in destination operand; cleared otherwise

CMP, TEST

cmp

- ✓ 두 operand를 비교하기 위해 오른쪽 operand에서 왼쪽 operand를 뺀다.
- ✓ 결과는 ZF, CF 등에 저장됨.
- ✓ je, jle 등의 opcode와 함께 사용됨

```
cmp     word ptr [ebp-28h],ax
jbe     KERNELBASE!CreateFileW+0x8e (76d7c327)
```

test

- ✓ 두 operand를 비교하기 위해 두 operand의 AND 연산을 수행. (결과는 ZF 등에 저장)
- ✓ 분기문에서는 주로 0인지 아닌지를 비교하기 위해 사용.

```
test    eax,eax
je      KERNELBASE!CreateFileW+0x237 (76d7c4d0)
```

je와 jz은 동일한 opcode!! (jz로 해석!!)

조건 jump

JE와 JZ, JNE와 JNZ는 동일한 OpCode임
JA 는 unsigned, JG 는 signed 비교
JB 는 unsigned, JL 는 signed 비교
http://unixwiz.net/techtips/x86-jumps.html

instruction	Description	Flags
JE JZ	Jump if equal Jump if zero	ZF = 1
JNE JNZ	Jump if not equal Jump if not zero	ZF = 0
JB JNAE JC	Jump if below Jump if not above or equal Jump if carry	CF = 1
JNB JAE JNC	Jump if not below Jump if above or equal Jump if not carry	CF = 0
JBE JNA	Jump if below or equal Jump if not above	CF = 1 or ZF = 1
JA JNBE	Jump if above Jump if not below or equal	CF = 0 and ZF = 0
JL JNGE	Jump if less Jump if not greater or equal	SF <> OF
JGE JNL	Jump if greater or equal Jump if not less	SF = OF
JLE JNG	Jump if less or equal Jump if not greater	ZF = 1 or SF <> OF
JG JNLE	Jump if greater Jump if not less or equal	ZF = 0 and SF = OF

if문 패턴 분석

```
BOOL        MyTest(INT nParam)
{
    if (nParam > 0)
    {
        wprintf(L"input value is greator than zero!!\n");
    } else {
        wprintf(L"input valud is less or equal than zero!\n");
    }

    return TRUE;
}
```

```
BOOL        MyTest(INT nParam)
{
01351CD0  push        ebp
01351CD1  mov         ebp,esp
    if (nParam > 0)
01351CD3  cmp         dword ptr [nParam],0
01351CD7  jle         MyTest+18h (1351CE8h)
    {
        wprintf(L"input value is greator than zero!!\n");
01351CD9  push        135C130h
01351CDE  call        wprintf (135271Eh)
01351CE3  add         esp,4
    } else {
01351CE6  jmp         MyTest+25h (1351CF5h)
        wprintf(L"input valud is less or equal than zero!\n");
01351CE8  push        135C178h
01351CED  call        wprintf (135271Eh)
01351CF2  add         esp,4
    }

    return TRUE;
01351CF5  mov         eax,1
}
01351CFA  pop         ebp
```

if 문의 assembly 패턴!!

1. cmp / test
2. 조건부 jmp

※ 학습용 샘플 빌드시에는 Optimize 옵션을 Disable로 설정 (Default : Maximize Speed)

for문 패턴 분석

```
VOID            GuGuDan(INT nDan)
{
    for (DWORD nIndex = 1; nIndex <= 9; nIndex++)
    {
        wprintf(L"%d * %d = %d\n", nDan, nIndex, nDan*nIndex);
    }
}
```

for 문의 assembly 패턴!!

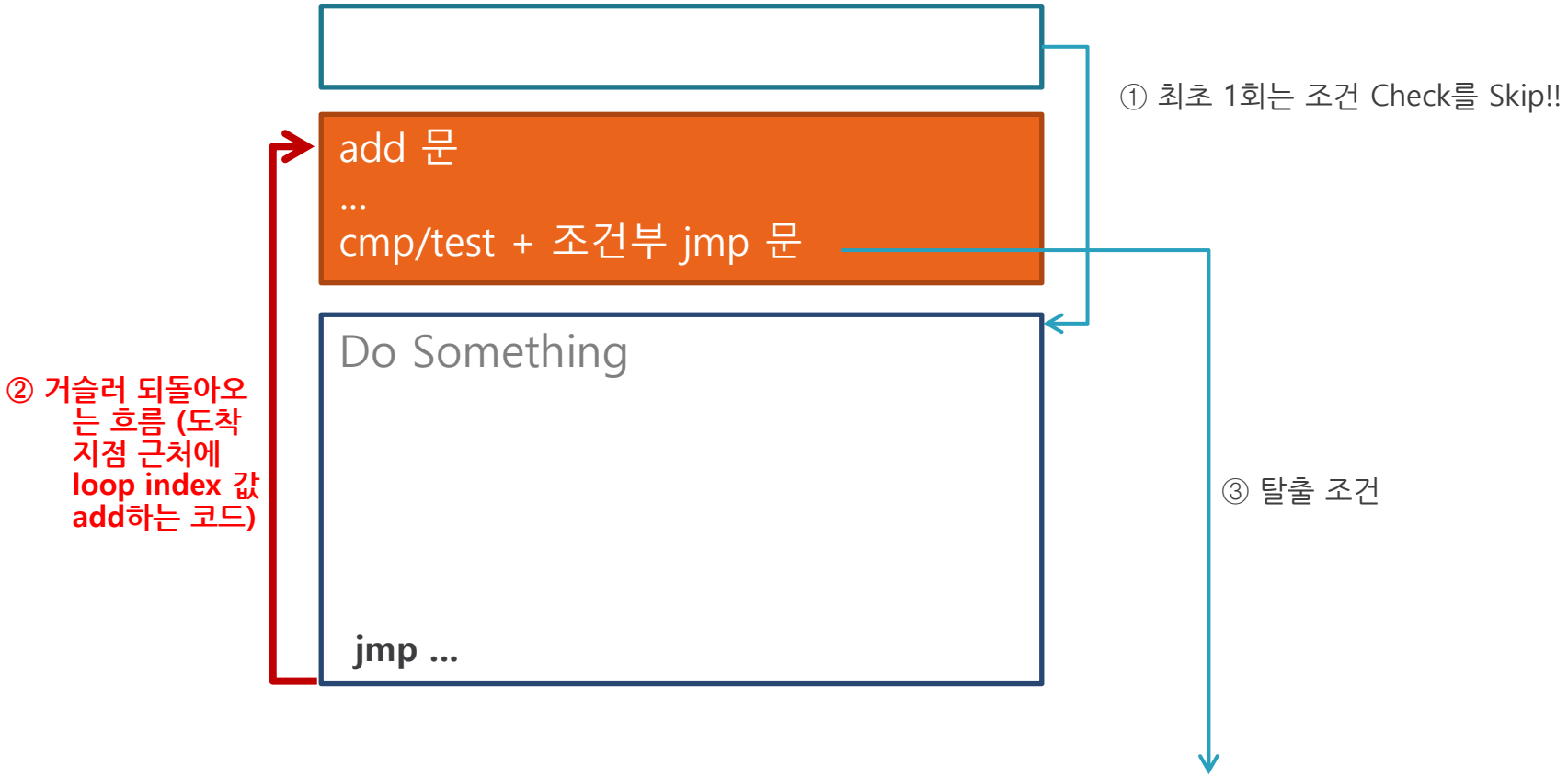
1. 최초 실행 시 한 블럭 건너뜀
2. 건너뛰었던 블록으로 되돌아옴 (jmp)
3. 되돌아온 위치에 add
4. add 다음에 cmp/test + jmp

※ 거슬러 올라오는 **jmp** 문은 반복문의 가장 큰 특징

```
push    ebp
mov     ebp,esp
push    ecx
mov     dword ptr [ebp-4],1                          } nIndex = 1
jmp     SimpleCall!GuGuDanFor+0x16 (012e1096)

SimpleCall!GuGuDanFor+0xd
mov     eax,dword ptr [ebp-4]                        } nIndex++
add     eax,1
mov     dword ptr [ebp-4],eax

SimpleCall!GuGuDanFor+0x16
cmp     dword ptr [ebp-4],9
jg      SimpleCall!GuGuDanFor+0x3c (012e10bc)        nIndex <= 9가 아니면

SimpleCall!GuGuDanFor+0x1c
mov     ecx,dword ptr [ebp+8]
imul    ecx,dword ptr [ebp-4]
push    ecx
mov     edx,dword ptr [ebp-4]
push    edx
mov     eax,dword ptr [ebp+8]
push    eax
push    offset SimpleCall!GS_ExceptionPointers+0x40 (012e212c)
call    dword ptr [SimpleCall!_imp__wprintf (012e20a0)]
add     esp,10h
jmp     SimpleCall!GuGuDanFor+0xd (012e108d)

SimpleCall!GuGuDanFor+0x3c
mov     esp,ebp
pop     ebp
ret
```

IRON

for문 패턴 분석


add 문
...
cmp/test + 조건부 jmp 문
Do Something
jmp ...
① 최초 1회는 조건 Check를 Skip!!
② 거슬러 되돌아오는 흐름 (도착 지점 근처에 loop index 값 add하는 코드)
③ 탈출 조건

while문 패턴 분석

```
VOID        GuGuDanWhile(INT nDan)
{
    INT nIndex = 1;

    while (nIndex <= 9)
    {
        wprintf(L"%d * %d = %d\n", nDan, nIndex, nDan * nIndex);
        nIndex++;
    }
}
```

while 문의 assembly 패턴!!

1. 반복문 시작 부분으로 되돌아옴 (jmp)
2. 되돌아온 위치 근처에 탈출문

```
SimpleCall!GuGuDanWhile
push    ebp
mov     ebp,esp
push    ecx
mov     dword ptr [ebp-4],1

SimpleCall!GuGuDanWhile+0xb
cmp     dword ptr [ebp-4],9
jg      SimpleCall!GuGuDanWhile+0x3a (012e103a)

SimpleCall!GuGuDanWhile+0x11
mov     eax,dword ptr [ebp+8]
imul    eax,dword ptr [ebp-4]
push    eax
mov     ecx,dword ptr [ebp-4]
push    ecx
mov     edx,dword ptr [ebp+8]
push    edx
push    offset SimpleCall!GS_ExceptionPointers+0x8 (012e20f4)
call    dword ptr [SimpleCall!_imp__wprintf (012e20a0)]
add     esp,10h
mov     eax,dword ptr [ebp-4]
add     eax,1
mov     dword ptr [ebp-4],eax
jmp     SimpleCall!GuGuDanWhile+0xb (012e100b)

SimpleCall!GuGuDanWhile+0x3a
mov     esp,ebp
pop     ebp
ret
```

do~while문 패턴 분석

```
VOID        GuGuDanDoWhile(INT nDan)
{
    INT nIndex = 1;

    do
    {
        wprintf(L"%d * %d = %d\n", nDan, nIndex, nDan * nIndex);
        nIndex++;
    } while (nIndex <= 9);
}
```

do~while 문의 assembly 패턴!!

1. 반복문 시작 부분으로 되돌아옴 (조건부 jmp)
2. 탈출문은 없을수도...

```
SimpleCall!GuGuDanDoWhile
push    ebp
mov     ebp,esp
push    ecx
mov     dword ptr [ebp-4],1

SimpleCall!GuGuDanDoWhile+0xb
mov     eax,dword ptr [ebp+8]
imul    eax,dword ptr [ebp-4]
push    eax
mov     ecx,dword ptr [ebp-4]
push    ecx
mov     edx,dword ptr [ebp+8]
push    edx
push    offset SimpleCall!GS_ExceptionPointers+0x24 (012e2110)
call    dword ptr [SimpleCall!_imp__wprintf (012e20a0)]
add     esp,10h
mov     eax,dword ptr [ebp-4]
add     eax,1
mov     dword ptr [ebp-4],eax
cmp     dword ptr [ebp-4],9
jle     SimpleCall!GuGuDanDoWhile+0xb (012e104b)

SimpleCall!GuGuDanDoWhile+0x38
mov     esp,ebp
pop     ebp
ret
```

※ 분석 목적이라면 굳이 반복문의 종류를 구분할 필요는 없음.

구조체 사용 코드 패턴 분석

```
typedef struct _MY_STRUCT
{
    INT     Product;
    INT     Sum;
} MY_STRUCT, *PMY_STRUCT;

INT     GetSum(PMY_STRUCT   pStruct)
{
    return pStruct->Sum;
}
```

※ 구조체의 멤버에 접근하는 예제

```
0:000> uf simplecall!GetSum
push    ebp
mov     ebp,esp
mov     eax,dword ptr [ebp+8]
mov     eax,dword ptr [eax+4]
pop     ebp
ret
```

```
typedef struct _MY_STRUCT
{
    INT     Product;
    INT     Sum;
} MY_STRUCT, *PMY_STRUCT;

INT     GetSum(PMY_STRUCT   pStruct)
{
    return pStruct->Product;
}
```

※ 구조체의 첫번째 멤버에 접근하는 예제

```
0:000> uf SimpleCall!GetSum
push    ebp
mov     ebp,esp
mov     eax,dword ptr [ebp+8]
mov     eax,dword ptr [eax]
pop     ebp
ret
```

구조체 사용 시의 assembly 패턴!!

1. 레지스터에 구조체의 주소를 저장
2. 저장된 구조체 주소에서 접근할 멤버의 Offset만큼 이동

calling convention

calling convention?

- ✓ 함수가 호출되는 규약
- ✓ 호출할 때 파라메터를 전달하는 방법, 함수 리턴 후 파라메터를 정리하는 방법이 calling convention에 의해 결정됨.

__cdecl

- ✓ 파라메터 전달 : 오른쪽 파라메터 > 왼쪽 파라메터 순서로 stack에 push
- ✓ 파라메터 정리 : 호출한 측(caller)에서!!
- ✓ CRT 함수들의 호출 규약

```
INT     __cdecl Sum(INT a, INT b, INT c)
{
    INT nSum = 0;
    nSum = a + b + c;
    return nSum;
}

int wmain( int argc, WCHAR** argv )
{
    INT nSum = Sum(1,2,3);

    wprintf(L"Sum : %d\n", nSum);

    return 0;
}
```

```
INT     __cdecl Sum(INT a, INT b, INT c)
{
00AB1CD0  push        ebp
00AB1CD1  mov         ebp,esp
00AB1CD3  push        ecx
    INT nSum = 0;
00AB1CD4  mov         dword ptr [nSum],0
    nSum = a + b + c;
00AB1CDB  mov         eax,dword ptr [a]
00AB1CDE  add         eax,dword ptr [b]
00AB1CE1  add         eax,dword ptr [c]
00AB1CE4  mov         dword ptr [nSum],eax
    return nSum;
00AB1CE7  mov         eax,dword ptr [nSum]
}
00AB1CEA  mov         esp,ebp
00AB1CEC  pop         ebp
00AB1CED  ret
```

```
int wmain( int argc, WCHAR** argv )
{
00AB1CF0  push        ebp
00AB1CF1  mov         ebp,esp
00AB1CF3  push        ecx
    INT nSum = Sum(1,2,3);
00AB1CF4  push        3
00AB1CF6  push        2
00AB1CF8  push        1
00AB1CFA  call        Sum (0AB1CD0h)
00AB1CFF  add         esp,0Ch
00AB1D02  mov         dword ptr [nSum],eax

    wprintf(L"Sum : %d\n", nSum);
00AB1D05  mov         eax,dword ptr [nSum]
00AB1D08  push        eax
```

__stdcall

- ✓ 파라메터 전달 : 오른쪽 파라메터 > 왼쪽 파라메터 순서로 stack에 push
- ✓ 파라메터 정리 : 호출된 측(callee)에서!!
- ✓ Windows api들의 호출 규약
- ✓ CALLBACK, APIENTRY 등으로 #define되어 사용되기도 함.

```
INT     __stdcall   Sum(INT a, INT b, INT c)
{
    INT nSum = 0;
    nSum = a + b + c;
    return nSum;
}

int wmain( int argc, WCHAR** argv )
{
    INT nSum = Sum(1,2,3);

    wprintf(L"Sum : %d\n", nSum);

    return 0;
}
```

```
INT     __stdcall   Sum(INT a, INT b, INT c)
{
00CA1CD0  push        ebp
00CA1CD1  mov         ebp,esp
00CA1CD3  push        ecx
    INT nSum = 0;
00CA1CD4  mov         dword ptr [nSum],0
    nSum = a + b + c;
00CA1CDB  mov         eax,dword ptr [a]
00CA1CDE  add         eax,dword ptr [b]
00CA1CE1  add         eax,dword ptr [c]
00CA1CE4  mov         dword ptr [nSum],eax
    return nSum;
00CA1CE7  mov         eax,dword ptr [nSum]
}
00CA1CEA  mov         esp,ebp
00CA1CEC  pop         ebp
00CA1CED  ret         0Ch
```

```
int wmain( int argc, WCHAR** argv )
{
00CA1CF0  push        ebp
00CA1CF1  mov         ebp,esp
00CA1CF3  push        ecx
    INT nSum = Sum(1,2,3);
00CA1CF4  push        3
00CA1CF6  push        2
00CA1CF8  push        1
00CA1CFA  call        Sum (0CA1CD0h)
00CA1CFF  mov         dword ptr [nSum],eax
```

thiscall

- ✓ 기본적으로 stdcall과 동일
- ✓ Class member 함수 호출시 사용되는 호출 규약
- ✓ this 포인터를 ecx 레지스터를 통해 전달

```
class CMyClass
{
public:
    INT     Sum(INT a, INT b, INT c);
    INT     m_nSum;
};

INT     CMyClass::Sum(INT a, INT b, INT c)
{
    m_nSum = a + b + c;
    return m_nSum;
}

int wmain( int argc, WCHAR** argv )
{
    CMyClass MyClass;
    INT nSum = MyClass.Sum(1,2,3);

    wprintf(L"Sum : %d\n", nSum);

    return 0;
}
```

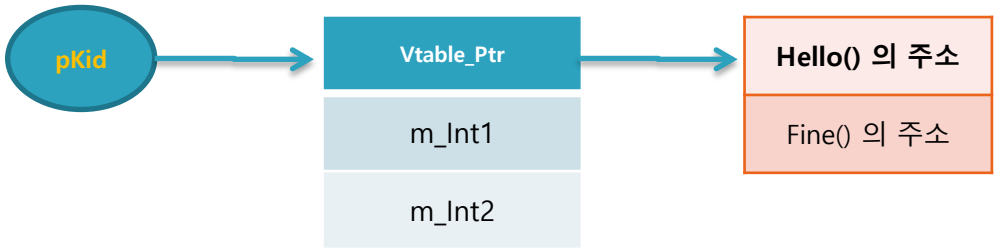
```
INT     CMyClass::Sum(INT a, INT b, INT c)
{
01111CD0  push        ebp
01111CD1  mov         ebp,esp
01111CD3  push        ecx
01111CD4  mov         dword ptr [ebp-4],ecx
    m_nSum = a + b + c;
01111CD7  mov         eax,dword ptr [a]
01111CDA  add         eax,dword ptr [b]
01111CDD  add         eax,dword ptr [c]
01111CE0  mov         ecx,dword ptr [this]
01111CE3  mov         dword ptr [ecx],eax
    return m_nSum;
01111CE5  mov         edx,dword ptr [this]
01111CE8  mov         eax,dword ptr [edx]
}
01111CEA  mov         esp,ebp
01111CEC  pop         ebp
01111CED  ret         0Ch
```

```
 int wmain( int argc, WCHAR** argv )
 {
00CF1D00  push        ebp
00CF1D01  mov         ebp,esp
00CF1D03  sub         esp,8
    CMyClass MyClass;
    INT nSum = MyClass.Sum(1,2,3);
00CF1D06  push        3
00CF1D08  push        2
00CF1D0A  push        1
00CF1D0C  lea         ecx,[MyClass]
00CF1D0F  call        CMyClass::Sum (0CF1CD0h)
00CF1D14  mov         dword ptr [nSum],eax
```

C++ 클래스 가상함수 호출

- ✓ 클래스 인스턴스의 시작부분에 가상함수 테이블의 포인터가 저장됨
- ✓ 함수 호출시 런타임에 가상함수 테이블에서 함수 주소를 구하여 call

```
class   CKid
{
public:
    virtual     VOID    Hello() {
        wprintf(L"Hello kiddy\n");
    }

    virtual     VOID    Fine()  {
        wprintf(L"I'm Fine\n");
    }

    CKid()  { m_Int1 = 0x10; m_Int2 = 0x20; }

    INT     m_Int1;
    INT     m_Int2;
};


int _tmain(int argc, _TCHAR* argv[])
{
    CKid        *pKid = new CKid;

    pKid->Hello();

    delete pKid;

    return 0;
}
```



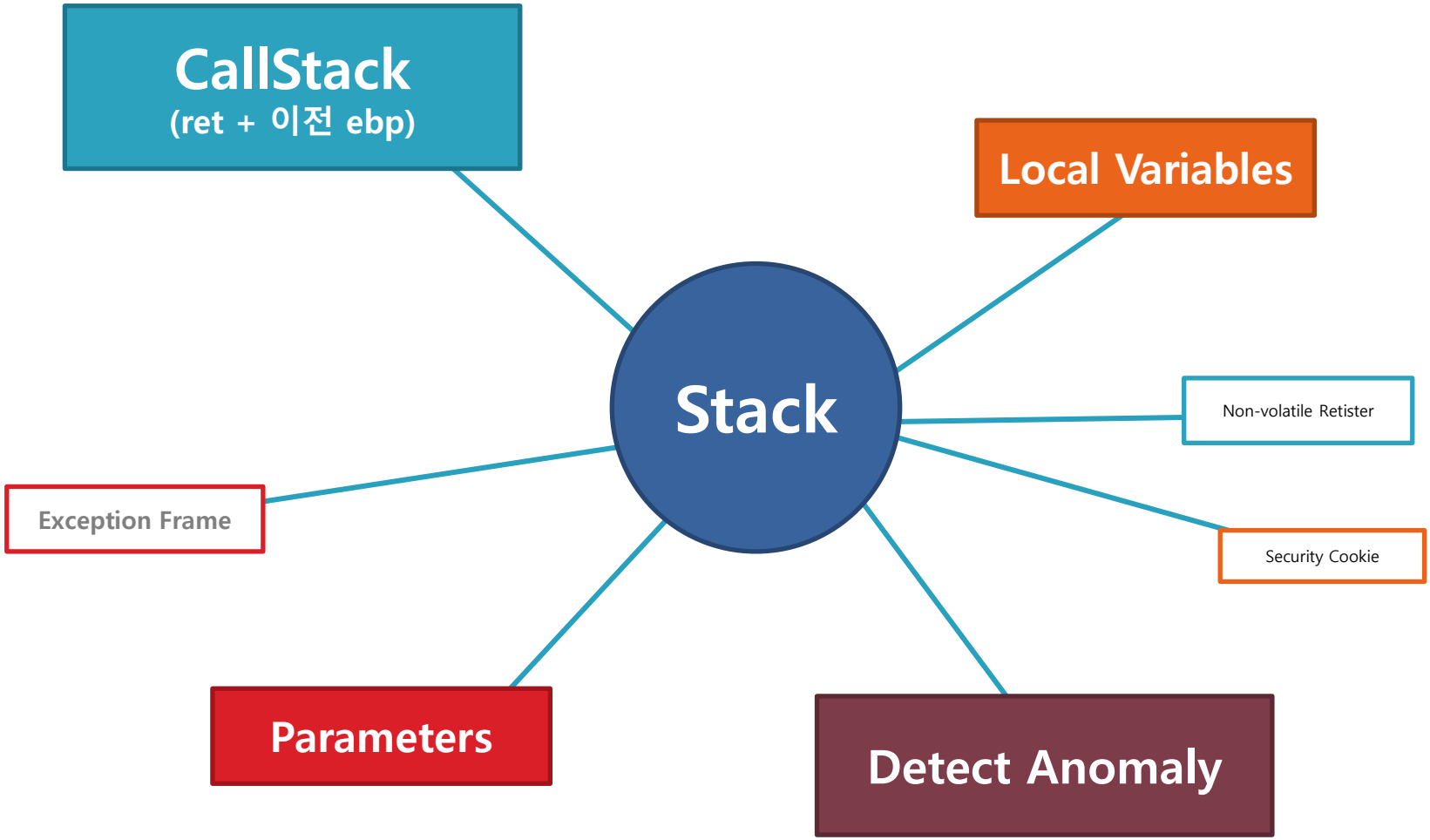

```
0:000> dv
            argc = 0n1
            argv = 0x00031a40
            pKid = 0x00035c88
0:000> dd 0x00035c88 L3
00035c88  010e2130 00000010 00000020
0:000> dps 0x010e2130  L3
010e2130  010e1000 VFuncTest!CKid::Hello [c:\u
010e2134  010e1020 VFuncTest!CKid::Fine [c:\us
010e2138  00000048
```

```
    pKid->Hello();
010E10AA  mov         edx,dword ptr [pKid]
010E10AD  mov         eax,dword ptr [edx]
010E10AF  mov         ecx,dword ptr [pKid]
010E10B2  mov         edx,dword ptr [eax]
010E10B4  call        edx
```

RedStonesoft

stack 기초

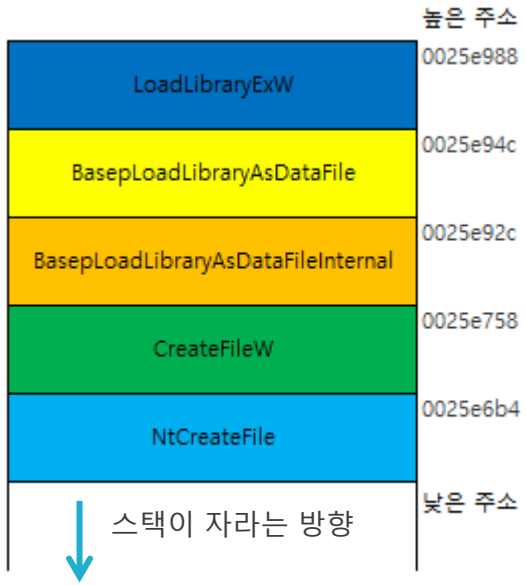
Stack에서 알 수 있는 정보들



Stack은 거꾸로 자란다!!

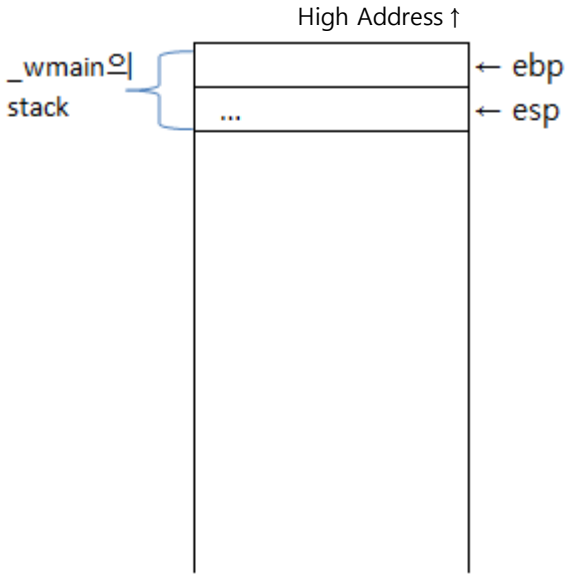
- ✓ stack은 항상 높은 주소 > 낮은 주소 방향으로 자란다. (Intel cpu 기준)
  - ✓ 새로운 변수가 할당될 때
  - ✓ 함수가 호출되어 새로운 스택 프레임이 생성될 때...
- ✓ 하지만 스택에서 사용되는 변수들은 낮은 주소 > 높은 주소 방향으로...

```
0:000> k
ChildEBP RetAddr
0025e6b4 76d7c5f7 ntdll!NtCreateFile
0025e758 76d7268f KERNELBASE!CreateFileW+0x35e
0025e92c 76d729ec KERNELBASE!BasepLoadLibraryAsDataFileInternal+0x288
0025e94c 76d72c3b KERNELBASE!BasepLoadLibraryAsDataFile+0x19
0025e988 756aeef1 KERNELBASE!LoadLibraryExW+0x18a
```



함수가 호출되는 과정에서 스택의 변화?

```
INT     _cdecl GuGuDan(IN INT nDanFrom, IN INT nDanTo, OPTIONAL OUT INT* pTotalProduct)
{
    INT     nDanIndex = 0;
    INT     nIndex = 0;
    INT     nProduct = 0;

    for (nDanIndex = nDanFrom; nDanIndex <= nDanTo; nDanIndex++)
    {
        for (nIndex = 1; nIndex <= 9; nIndex++)
        {
            wprintf(L"%d * %d = %d\n", nDanIndex, nIndex, nDanIndex * nIndex);
            nProduct *= nDanIndex * nIndex;
        }
        wprintf(L"\n");
    }

    if (pTotalProduct)
    {
        *pTotalProduct = nProduct;
    }

    return nProduct;
}

int _tmain(int argc, _TCHAR* argv[])
{
    INT nProduct = 0;
    INT nFrom = 3;
    INT nTo = 9;

    nProduct = GuGuDan(nFrom, nTo, NULL);

    wprintf(L"Result : %d\n", nProduct);

    return 0;
}
```



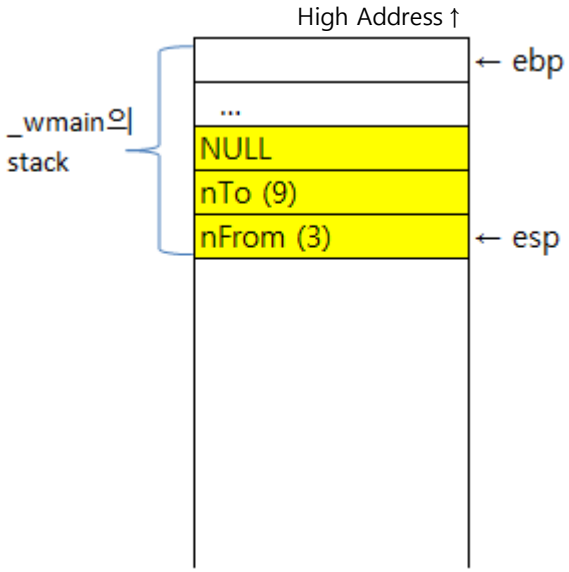
[샘플코드 : SimpleCall.exe]

함수가 호출되는 과정 (1) – parameter push

```
int _tmain(int argc, _TCHAR* argv[])
{
    INT nProduct = 0;
    INT nFrom = 3;
    INT nTo = 9;

    nProduct = GuGuDan(nFrom, nTo, NULL);
```

```
sub     esp,0Ch
mov     dword ptr [ebp-8],0  // nProduct
mov     dword ptr [ebp-0Ch],3  // nFrom
mov     dword ptr [ebp-4],9  // nTo
```

```
push    0       // NULL
mov     eax,dword ptr [ebp-4]   // nTo
push    eax
mov     ecx,dword ptr [ebp-0Ch]   // nFrom
push    ecx
call    SimpleCall!GuGuDan (013d1000)
```

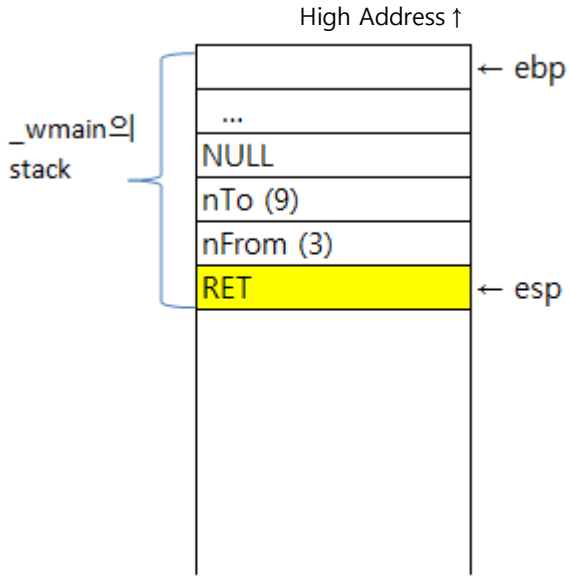


※ 함수의 prolog 이후에 push가 나오면 call을 위한 준비 단계!!

함수가 호출되는 과정 (2) – 함수 호출 (call)

✓ call instruction은 다음의 두 instruction으로 풀어 쓸 수 있다.

```
push eip (←return address)
jmp <function address>
```



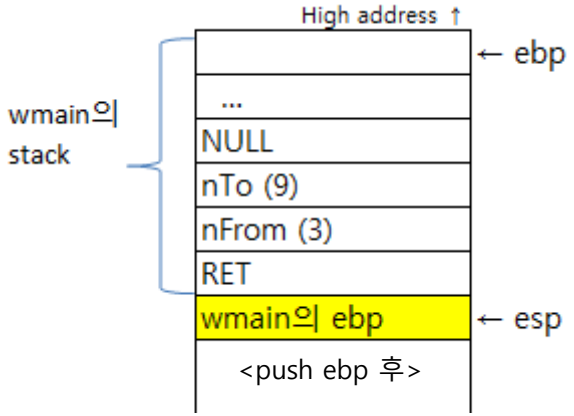

```
nProduct = GuGuDan(nFrom, nTo, NULL);
```

```
push    0       // NULL
mov     eax,dword ptr [ebp-4]   // nTo
push    eax
mov     ecx,dword ptr [ebp-0Ch]   // nFrom
push    ecx
call    SimpleCall!GuGuDan (013d1000)
```

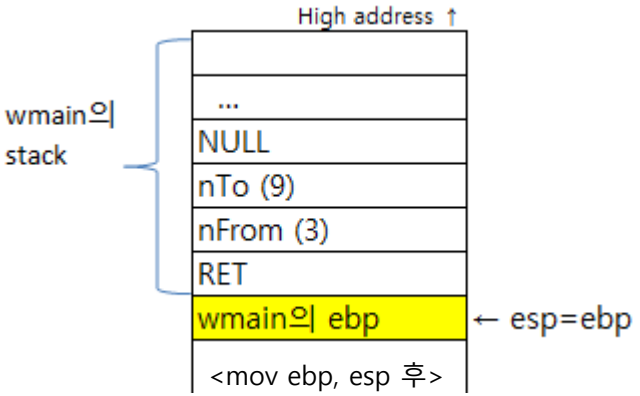
함수가 호출되는 과정 (3) – 새로운 stack frame 생성

✓ 새로운 함수는 (대부분) 다음과 같이 시작한다 ➔ 공식!!

```
mov edi, edi
push ebp
mov ebp, esp
```




```
#include "stdafx.h"
#include <windows.h>

INT     _cdecl GuGuDan(IN INT nDanFrom, IN INT nDanTo, OPTIONAL OUT INT* pTotalProduct)
{
00201000  push        ebp
00201001  mov         ebp,esp
00201003  sub         esp,0Ch
    INT     nDanIndex = 0;
00201006  mov         dword ptr [nDanIndex],0
    INT     nIndex = 0;
0020100D  mov         dword ptr [nIndex],0
    INT     nProduct = 0;
00201014  mov         dword ptr [nProduct],0
```



※ 빌드시 /hotpatch 옵션 추가하면 "mov edi, edi" 가 포함됨 (5bytes align을 위한 padding!!)

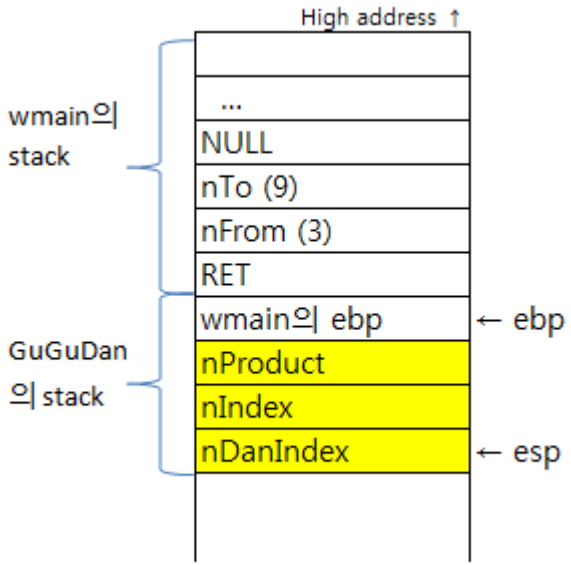
함수가 호출되는 과정 (4) – 로컬변수 생성, non-vol register 백업

- ✓ 로컬변수를 선언하기 위해 esp를 필요한 메모리 사이즈만큼 감소시킴
- ✓ Non-volatile register를 사용하기 위해 stack에 백업 (if required)

```
INT _cdecl GuGuDan(IN INT nDanFrom, IN INT nDanTo,
                   OPTIONAL OUT INT * pTotalProduct)
{
    INT nDanIndex = 0;
    INT nIndex = 0;
    INT nProduct = 0;
```

```
push    ebp
mov     ebp,esp
sub     esp,0Ch
mov     dword ptr [ebp-0Ch],0
mov     dword ptr [ebp-8],0
mov     dword ptr [ebp-4],0
```

※ 로컬 변수가 INT 3개이므로 esp를 4 * 3 = 12바이트만큼 감소시킴.



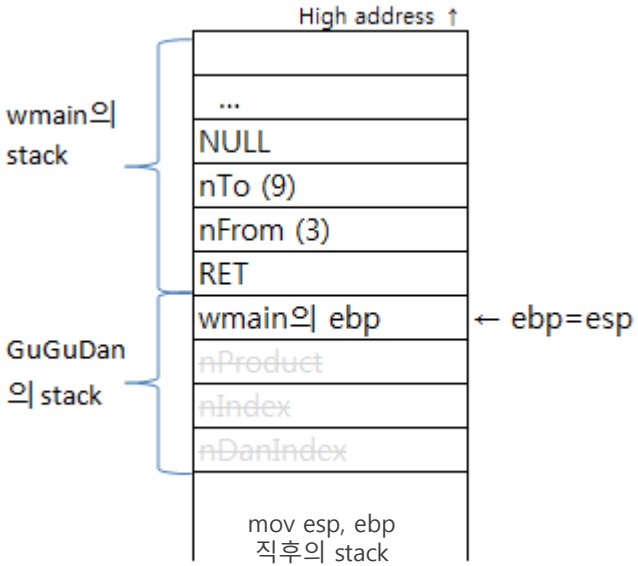
함수가 호출되는 과정 (5) – Do Something

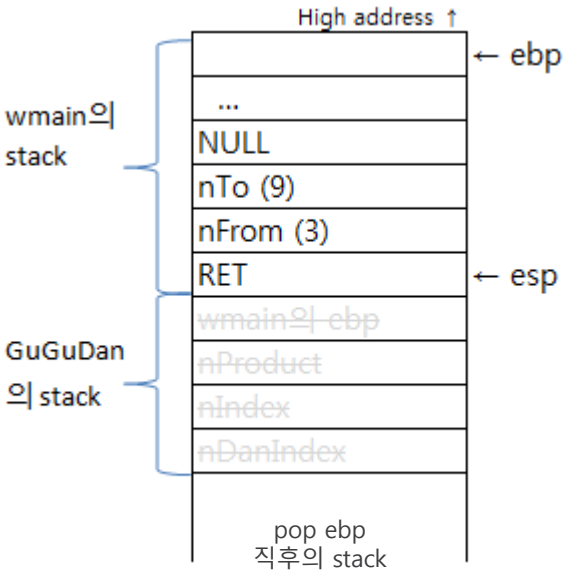
이미지 출처 : https://polyrouse.wordpress.com

함수가 호출되는 과정 (6) – Stack 정리 & non-vol register 복원

- ✓ 리턴값을 eax에 저장 (if any)
- ✓ 로컬변수용으로 할당한 메모리를 회수하기 위해 esp를 다시 증가시킴.
- ✓ 백업받은 non-volatile register 복원 (if any)
- ✓ 함수 prologue에서 백업한 ebp 레지스터 복원

```
mov     eax,dword ptr [ebp-4]
mov     esp,ebp
pop     ebp
ret
```

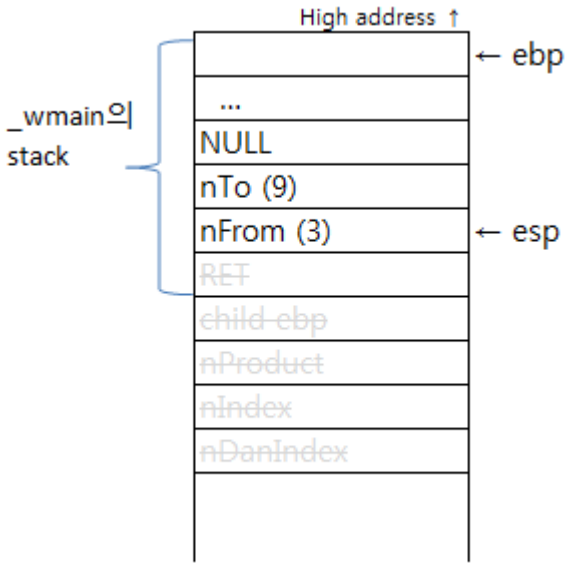




함수가 호출되는 과정 (6) – 함수 리턴 (ret)

✓ ret instruction은 다음과 같이 풀어쓸 수 있다.

```
pop ecx
jmp ecx
```



✓ ret instruction이 실행되면 현재 stack의 최상단에 들어있는 주소값으로 jmp한다.

✓ 따라서, ret 이 실행되는 시점에서 stack의 최상단에는 Return Address가 들어있어야 함.

※ 상기 샘플에서 ecx는 임의의 register를 표현함.

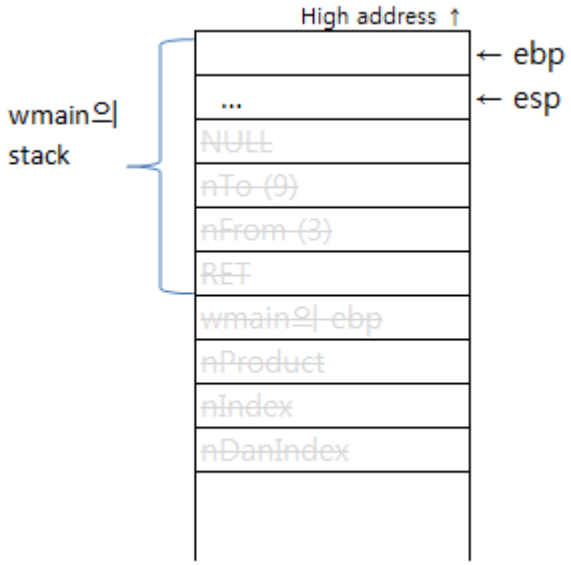
```
mov     eax,dword ptr [ebp-4]
mov     esp,ebp
pop     ebp
ret
```

함수가 호출되는 과정 (6) – parameter 정리

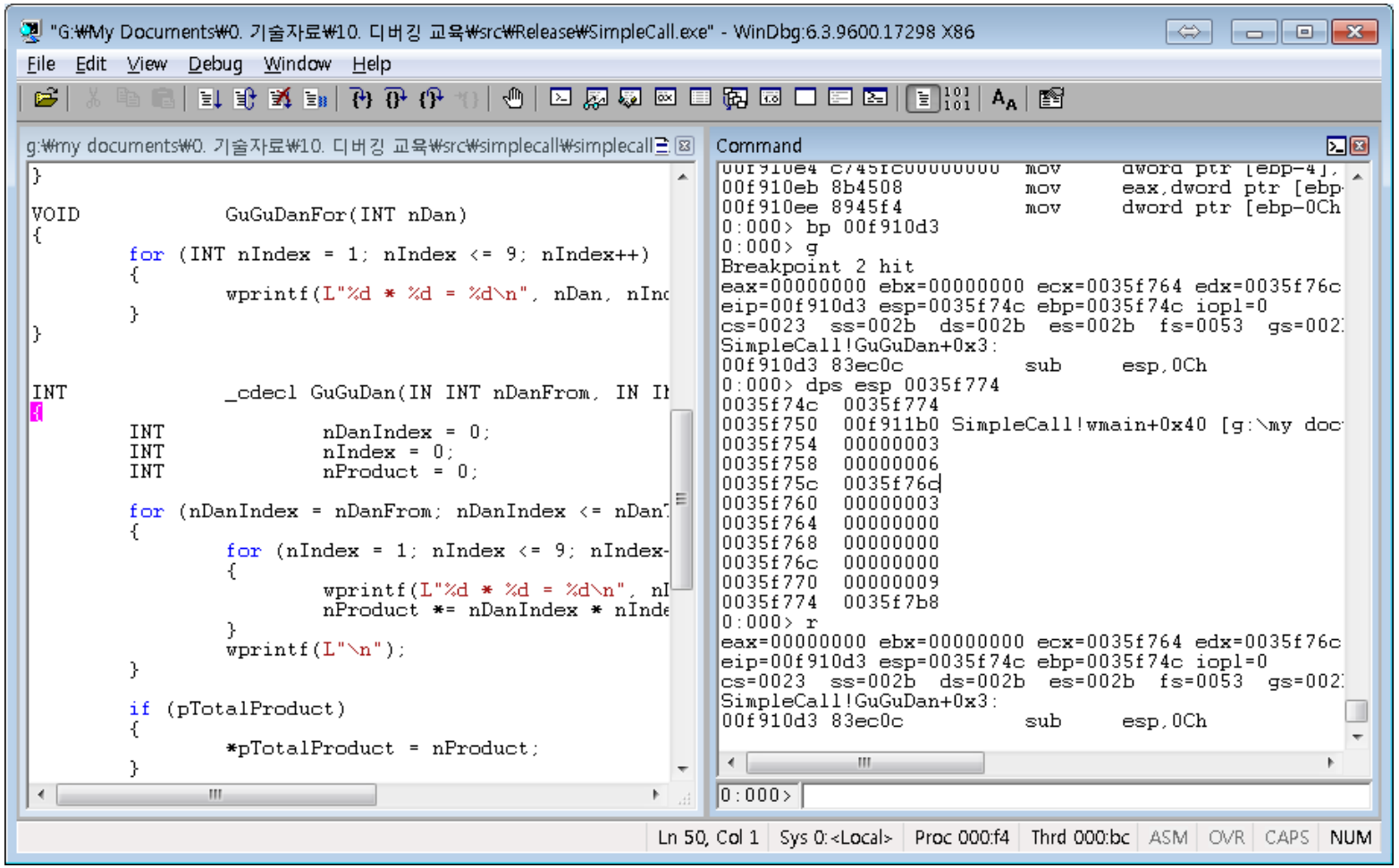
✓ _cdecl 호출이므로 call 한 측에서 parameter가 push된 stack을 정리. (_stdcall이면??)

```
int _tmain(int argc, _TCHAR* argv[])
{
    INT nProduct = 0;
    INT nFrom = 3;
    INT nTo = 9;

    nProduct = GuGuDan(nFrom, nTo, NULL);
```

```
call    SimpleCall!GuGuDan (013d1000)
add     esp,0Ch
```

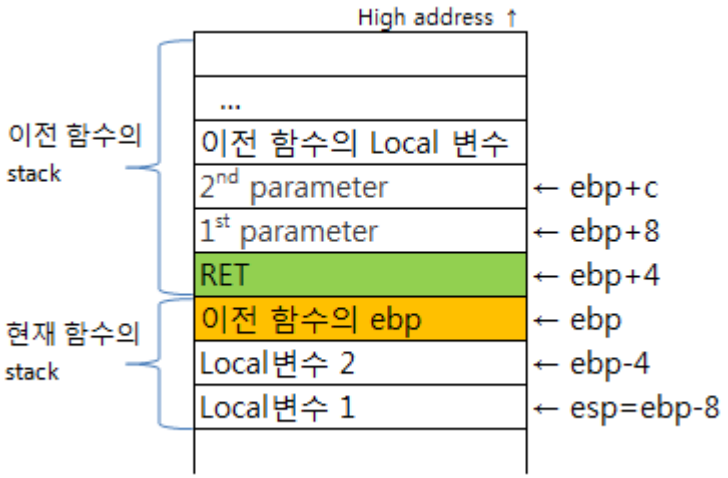


To see is to believe!!



[공식] 로컬변수 / parameter 를 참조하는 방법

- ✓ ebp+c : 2nd parameter
- ✓ ebp+8 : 1st parameter
- ✓ ebp+4 : Return Address
- ✓ ebp : 이전 함수의 ebp
- ✓ ….
- ✓ ebp-xx : local variable



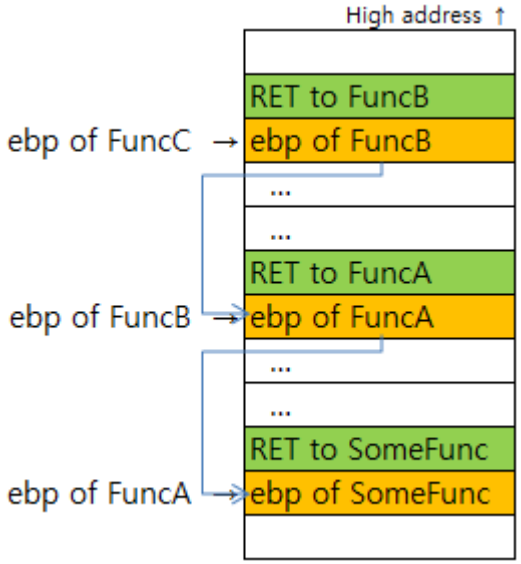
요약하자면..

ebp + xx : parameter
ebp + 4 : RET
ebp : 이전 함수의 ebp
ebp – xx : local variable

ebp는 callstack이라는 linked list의 포인터!!

- ✓ 각 ebp가 가리키는 위치에는 이전 함수의 ebp가 저장
- ✓ 각 ebp+4 위치에는 이전 함수로 돌아갈 Return Address가 저장됨
- ✓ ebp가 가리키는 곳의 값을 계속 따라가면 이전 함수의 return address chain 을 구성해낼 수 있음.

```
INT FuncC()
{
    return 0;
}

INT FuncB()
{
    return FuncC();
}

INT FuncA()
{
    return FuncB();
}
```



Windbg로 Stack 디버 보기 (1)

```
0:000> !teb
TEB at 7efdd000
    ExceptionList:        0030fc74
    StackBase:            00310000
    StackLimit:           0030d000
    SubSystemTib:         00000000
    FiberData:            00001e00
    ArbitraryUserPointer: 00000000
    Self:                 7efdd000
    EnvironmentPointer:   00000000
    ClientId:             00001ac0 . 00000ad0
    RpcHandle:            00000000
    Tls Storage:          7efdd02c
    PEB Address:          7efde000
    LastErrorValue:       0
    LastStatusValue:      0
    Count Owned Locks:    0
    HardErrorMode:        0

0:000> r
eax=0000000a ebx=00000000 ecx=00000000 edx=0015df98 esi=00000001 edi=00353370
eip=00351078 esp=0030fc14 ebp=0030fc20 iopl=0         nv up ei pl nz na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000206
SimpleCall!GuGuDan+0x78:
00351078 ebc3            jmp     SimpleCall!GuGuDan+0x3d (0035103d)

0:000> dps esp 00310000
0030fc14  00000003
0030fc18  00000001
0030fc1c  00000000
0030fc20  0030fc40
0030fc24  003510ca SimpleCall!wmain+0x2a
0030fc28  00000003
0030fc2c  00000009
0030fc30  00000000
  <이하 생략>
```

Windbg로 Stack 디벼 보기 (2)

✓ teb가 확인되지 않을 때는... 충분히 넓은 범위의 값을 다 찍어본다.

```
0:000> r ebp
ebp=0030fc20

0:000> dps esp esp+1000
0030fc14  00000003
0030fc18  00000001
0030fc1c  00000000
0030fc20  0030fc40
0030fc24  003510ca SimpleCall!wmain+0x2a
0030fc28  00000003
0030fc2c  00000009
0030fc30  00000000
0030fc34  00000003
0030fc38  00000000
0030fc3c  00000009
0030fc40  0030fc84
0030fc44  00351264 SimpleCall!__tmainCRTStartup+0x122
0030fc48  00000001
0030fc4c  00201258
0030fc50  00203d50
...
0030fc80  00000000
0030fc84  0030fc90
0030fc88  7723338a kernel32!BaseThreadInitThunk+0xe
0030fc8c  7efde000
0030fc90  0030fcd0
0030fc94  77c19f72 ntdll!__RtlUserThreadStart+0x70
0030fc98  7efde000
```

wmain의 스택
(wmain = __tmainCRTStartup의 child)

__tmainCRTStartup의 스택

```
0:000> kbn
 # ChildEBP RetAddr  Args to Child
00 0030fc20 003510ca 00000003 00000009 00000000 SimpleCall!GuGuDan+0x78
01 0030fc40 00351264 00000001 00201258 00203d50 SimpleCall!wmain+0x2a
02 0030fc84 7723338a 7efde000 0030fcd0 77c19f72 SimpleCall!__tmainCRTStartup+0x122
03 0030fc90 77c19f72 7efde000 71e4aa41 00000000 kernel32!BaseThreadInitThunk+0xe
04 0030fcd0 77c19f45 00351385 7efde000 00000000 ntdll!__RtlUserThreadStart+0x70
05 0030fce8 00000000 00351385 7efde000 00000000 ntdll!_RtlUserThreadStart+0x1b
```

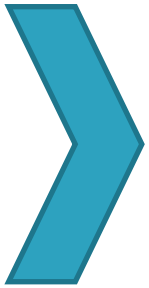
child의 ebp | child의 RET | child 에 전달된 아규먼트 | child 의 이름 (심볼)

※ Child 의 의미??

로컬변수 실종사건!!

- ✓ 로컬변수가 항상 스택에 선언되는 것은 아니며, 최적화 시 Register가 "잠시" 사용되는 경우가 많음
- ✓ 최적화 과정에서 함수 인라인화, 파편화 등 다양한 변화가 발생함. (Default : Maximize Speed)

```
0:000> uf simplecall!wmain
push    ebp
mov     ebp,esp
sub     esp,0Ch
mov     dword ptr [ebp-8],0
mov     dword ptr [ebp-0Ch],3
mov     dword ptr [ebp-4],9
push    0
mov     eax,dword ptr [ebp-4]
push    eax
mov     ecx,dword ptr [ebp-0Ch]
push    ecx
call    SimpleCall!GuGuDan (00351000)
add     esp,0Ch
mov     dword ptr [ebp-8],eax
mov     edx,dword ptr [ebp-8]
push    edx
push    offset SimpleCall!GS_ExceptionPointers+0x28 (00352114)
call    dword ptr [SimpleCall!_imp__wprintf (003520a0)]
add     esp,8
xor     eax,eax
mov     esp,ebp
pop     ebp
ret
0:000> dv    // wmain에서 로컬변수 프린트
          argc = 0n1
          argv = 0x00201258
         nFrom = 0n3
      nProduct = 0n0
           nTo = 0n9
```

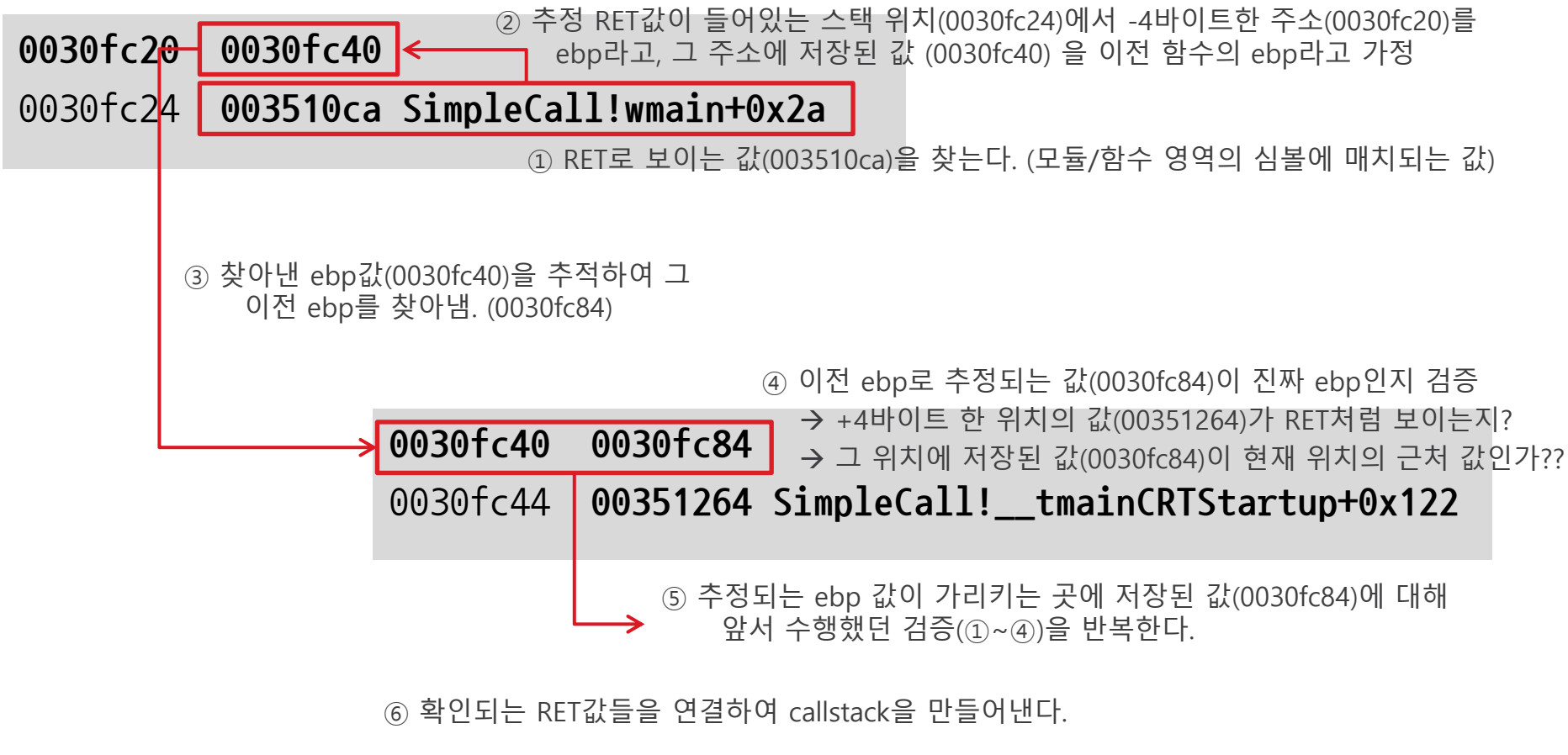


최적화 :
Maximize Speed!!

```
0:000> uf simplecall!wmain
call    SimpleCall!GuGuDan (000b1000)  // 파라메터가 없다???
push    eax
push    offset SimpleCall!`string' (000b2114)
call    dword ptr [SimpleCall!_imp__wprintf (000b20a0)]
add     esp,8
xor     eax,eax
ret

0:000> dv  // wmain에서 로컬변수 프린트... 로컬 변수가 어디갔지??
          argc = 0n594520
          argv = 0x00093d50

0:000> uf simplecall!GuGuDan
SimpleCall!GuGuDan
push    ebp
mov     ebp,esp
push    ecx
push    ebx
push    esi
push    edi
mov     dword ptr [ebp-4],0
mov     edi,3       // 앗 이것은... 하.. 하드코딩!!??

SimpleCall!GuGuDan+0x13
mov     esi,1
mov     ebx,edi
lea     ebx,[ebx]
 <이하 생략>
```

육안으로 callstack을 재구성해보자!! (1)

✓ dps 결과에서 ebp, ret 찾기



육안으로 callstack을 재구성해보자!! (2)

✓ 확인된 ret 값들을 연결해서 callstack을 만들어낸다.

```
// 수동으로 재구성한 callstack
0045f840  00d410ca SimpleCall!wmain+0x2a
0045f860  00d41264 SimpleCall!__tmainCRTStartup+0x122
0045f8a4  7723338a kernel32!BaseThreadInitThunk+0x12
0045f8b0  77c19f72 ntdll!RtlInitializeExceptionChain+0x63
0045f8f0  77c19f45 ntdll!RtlInitializeExceptionChain+0x36

// windbg가 자동으로 만들어주는 callstack과 비교해보자!!
0:000> k
ChildEBP RetAddr
0045f83c 00d410ca SimpleCall!GuGuDan+0x4c
0045f85c 00d41264 SimpleCall!wmain+0x2a
0045f8a0 7723338a SimpleCall!__tmainCRTStartup+0x122
WARNING: Stack unwind information not available. Following frames may be wrong.
0045f8ac 77c19f72 kernel32!BaseThreadInitThunk+0x12
0045f8ec 77c19f45 ntdll!RtlInitializeExceptionChain+0x63
0045f904 00000000 ntdll!RtlInitializeExceptionChain+0x36
```

IRON

육안으로 callstack을 재구성해보자!! (3)

- ✓ 주의 : 저렇게(097ffda8) 생긴 애는 ebp가 아님. (저건 예외 프레임)
- ✓ See also : http://www.microsoft.com/msj/0197/exception/exception.aspx

```
0:052> dps esp esp+100
...
097ffda8  097ffe24
097ffdac  61790ee0 tcpsvc!_except_handler4
097ffdb0  57055c30
097ffdb4  fffffffe
```

EXCEPTION_REGISTRATION 구조체

```
typedef struct _EXCEPTION_REGISTRATION
{
    struct _EXCEPTION_REGISTRATION*   prev;
    DWORD                             handler;
} EXCEPTION_REGISTRATION, *PEXCEPTION_REGISTRATION;
```

나 ebp !

난 exception frame !

[실습 1] Assembly 분석

[실습1] 도전!! GuGuDan()을 Decompile해보자!! (1)

```
0:000> uf simplecall!GuGuDan
SimpleCall!GuGuDan    8 01211000 55              push    ebp
    8 01211001 8bec            mov     ebp,esp
    8 01211003 83ec0c          sub     esp,0Ch
    9 01211006 c745f400000000  mov     dword ptr [ebp-0Ch],0
   10 0121100d c745f800000000  mov     dword ptr [ebp-8],0
   11 01211014 c745fc00000000  mov     dword ptr [ebp-4],0
   13 0121101b 8b4508          mov     eax,dword ptr [ebp+8]
   13 0121101e 8945f4          mov     dword ptr [ebp-0Ch],eax
   13 01211021 eb09            jmp     SimpleCall!GuGuDan+0x2c
      (0121102c)

SimpleCall!GuGuDan+0x23
   13 01211023 8b4df4          mov     ecx,dword ptr [ebp-0Ch]
   13 01211026 83c101          add     ecx,1
   13 01211029 894df4          mov     dword ptr [ebp-0Ch],ecx
```

Assembly
➔ C Source

```
INT     _cdecl GuGuDan(IN INT nDanFrom, IN INT nDanTo, OPTIONAL OUT INT* pTotalProduct)
{
    INT     nDanIndex = 0;
    INT     nIndex = 0;
    INT     nProduct = 0;

    for (nDanIndex = nDanFrom; nDanIndex <= nDanTo; nDanIndex++)
    {
        for (nIndex = 1; nIndex <= 9; nIndex++)
        {
            wprintf(L"%d * %d = %d\n", nDanIndex, nIndex, nDanIndex * nIndex);
            nProduct *= nDanIndex * nIndex;
        }
        wprintf(L"\n");
    }

    if (pTotalProduct)
    {
        *pTotalProduct = nProduct;
    }

    return nProduct;
}
```

[요령] Step by Stap!!

1. windbg에서 uf 커맨드로 assembly code 확인
2. ebp-xx , ebp+xx 를 변수명으로 Replace!!
3. Label과 goto 문을 활용해 '초벌 번역'
4. 화살표를 그려서 코드 블럭 간의 흐름을 파악
   - ➢ 거슬러 올라가는 화살표에 주목!!!
5. 초벌 번역된 C 소스를 보기 좋게 Decoration

Api 호출에서 힌트를 얻자!!

✓ 프로토타입이 알려진 WinApi 호출 부분을 분석하면 로컬 변수의 의미, 구조체 등에 대한 정보를 얻을 수 있음.

```
KERNELBASE!GetFileAttributesW:
mov     edi,edi
push    ebp
mov     ebp,esp
sub     esp,48h
push    esi
xor     esi,esi
push    esi
push    esi
lea     eax,[ebp-8]
push    eax
push    dword ptr [ebp+8]
call    dword ptr [KERNELBASE!_imp__RtlDosPathNameToNtPathName_U_WithStatus]
```

[ebp-8] = PUNICODE_STRING NtName

```
KERNELBASE!GetFileAttributesW+0x41:
…
mov     edi,dword ptr [ebp-4]
mov     dword ptr [ebp-18h],eax
lea       eax,[ebp-48h]
push      eax
lea       eax,[ebp-20h]
push      eax
mov     dword ptr [ebp-20h],18h
mov     dword ptr [ebp-1Ch],esi
mov     dword ptr [ebp-14h],40h
mov     dword ptr [ebp-10h],esi
mov     dword ptr [ebp-0Ch],esi
call    dword ptr [KERNELBASE!_imp__NtQueryAttributesFile]
```

[ebp-48h] = FileInformation
[ebp-20h] = ObjectAttributes
…
[ebp-20h] = ObjectAttributes->Length
[ebp-1Ch] = ObjectAttributes->RootDirectory
[ebp-18h] = ObjectAttributes->ObjectName
[ebp-14h] = ObjectAttributes->Attributes
…

RedStonesoft

[실습1] 도전!! GuGuDan()을 Decompile해보자!! (2)

```
SimpleCall!GuGuDan
push    ebp
mov     ebp,esp
sub     esp,0Ch
mov     dword ptr [ebp-0Ch],0
mov     dword ptr [ebp-8],0
mov     dword ptr [ebp-4],0
mov     eax,dword ptr [ebp+8]
mov     dword ptr [ebp-0Ch],eax
jmp     SimpleCall!GuGuDan+0x2c (0121102c)

SimpleCall!GuGuDan+0x23
mov     ecx,dword ptr [ebp-0Ch]
add     ecx,1
mov     dword ptr [ebp-0Ch],ecx
```

① windbg에서 uf 커맨드로 assembly code 확인

```
SimpleCall!GuGuDan
push    ebp
mov     ebp,esp
sub     esp,0Ch
mov     dword ptr [locvar_A],0
mov     dword ptr [locvar_B],0
mov     dword ptr [locvar_C],0
mov     eax,dword ptr [arg_A]
mov     dword ptr [locvar_A],eax
jmp     SimpleCall!GuGuDan+0x2c (0121102c)

SimpleCall!GuGuDan+0x23
mov     ecx,dword ptr [locvar_A]
add     ecx,1
mov     dword ptr [locvar_A],ecx
```

② ebp-xx , ebp+xx 를 변수명으로 Replace!!

```
GuGuDan(arg_A, arg_B, arg_C)
{
  locvar_A = arg_A;
  goto GuGuDan+0x2c;

GuGuDan+0x23:
   locvar_A++;

GuGuDan+0x2c:
  if (locvar_A > arg_B)  {
    goto GuGuDan+0x8a;
  }

GuGuDan+0x34:
    locvar_B = 1;
    goto GuGuDan+0x46;
```

③ Label과 goto 문을 활용해 '초벌 번역'

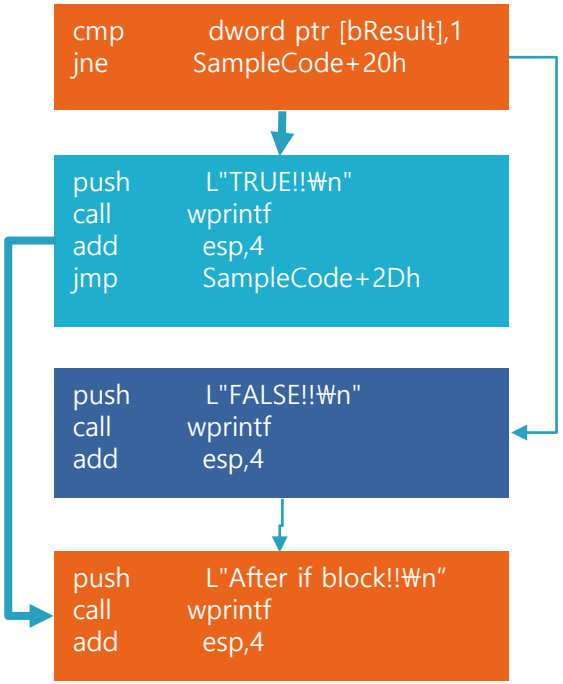
```
GuGuDan(arg_A, arg_B, arg_C)
{
  locvar_A = arg_A;

  for (locvar_A = arg_A; locvar_A <= arg_B; locvar_A++)
  {
       for (locvar_B = 1; locvar_B <= 9; locvar_B++)
       {
          wprintf(L"%d * %d = %d\n",
                  locvar_A, locvar_B, locvar_A * locvar_B);
          locvar_C = locvar_A * locvar_B * locvar_C;
       }

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

④ 초벌 번역된 C 소스를 보기 좋게 Decoration

하지만... 최적화가 적용된다면?

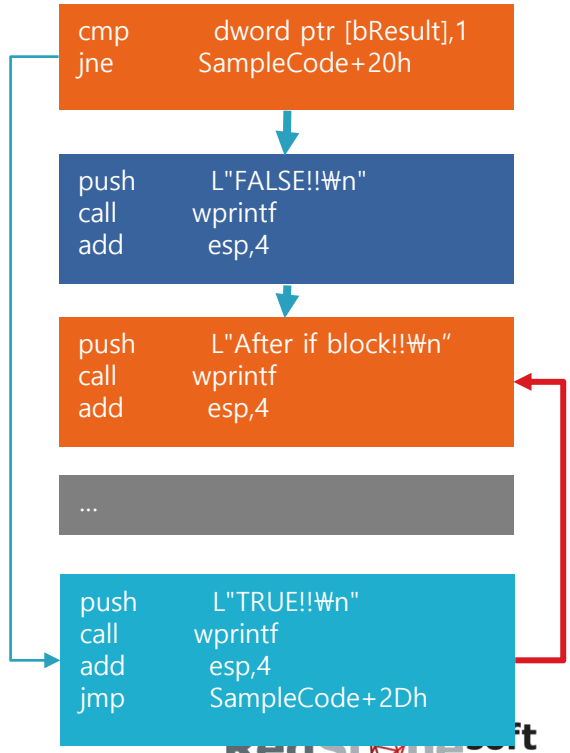
Profile-Guided Optimizations

- ✓ 실제 실행 결과를 분석하여 코드를 재배치 ➔ Cache Hit 율을 높이는 효과.
- ✓ 코드 블록의 배치가 뒤죽박죽…
- ✓ 거슬러 올라오는 jmp문은 순환문? 과연 그럴까??
- ✓ OS 모듈에 기본 적용



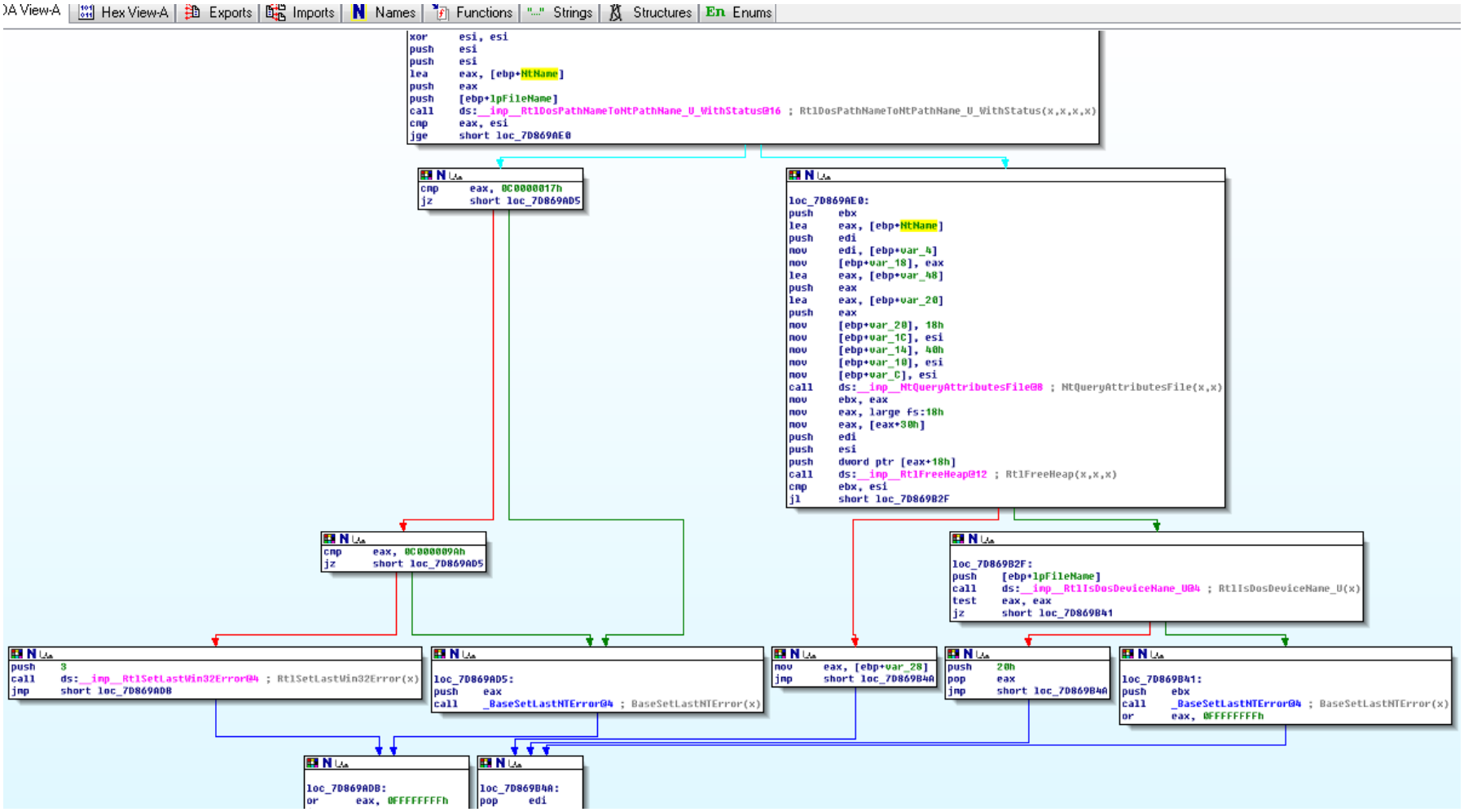

```
if (bResult == TRUE)
{
    wprintf(L"TRUE!!\n");   // 99.9% 빈도
} else {
    wprintf(L"FALSE!!\n");  // 0.1% 빈도
}

wprintf(L"After if block!!\n");
```

최적화 없음



프로파일 기반 최적화

궁극의 디컴파일러 IDA-Pro!!



[실습 2] Stack 분석 응용 (1)

Stack 분석 결과를 메모리 덤프 분석에 활용하자!!

Stack을 수동 분석해야 하는 경우??

- ✓ 최적화로 인해 stack이 자동 분석되지 않는 경우
- ✓ 모듈이 unload되어 crash가 발생하면서 stack이 분석되지 않는 경우
- ✓ Buffer Overflow가 발생하여 Return Address가 변조된 경우
- ✓ pdb가 없어서 callstack이 제대로 분석되지 않는 경우
- ✓ 자동분석된 결과가 의심스러운 경우
  1) 콜스택의 시작 함수가 일반적인 스레드 시작/초기화 함수가 아닌 경우
  2) 함수의 호출 순서가 상식적으로 이해되지 않는 경우
  3) 콜스택의 중간 부분이 비어있는 경우 (부정확할 수 있다는 경고메시지 출력)
  4) 콜스택이 끊겨있는 경우

삽 푸고 있는 Windbg에게 힌트를 주자!!

- ✓ WinDbg는 현재 Context를 기준으로 Callstack을 만들어냄.
- ✓ WinDbg가 자동 분석에 실패했을 때 수동으로 Context를 지정하면 더 나은 Callstack을 만들어낼 수(도) 있음!!
- ✓ 한번에 전체 callstack을 만들어낼 수 없는 경우도 많음. 경우에 따라서는 2~3 번 나누어 만들어 낸 stack을 이어 붙여서 해석하라.

[step 1] dps 결과를 분석하여 ebp 값을 추정
[step 2] 찾아낸 ebp 값을 힌트를 지정하여 callstack 재구성.
만족스러운 callstack이 나올 때까지 [step1] ~ [step2]를 반복
[step 3] 확인된 ebp 값으로 esp, eip 값을 추정
[step 4] .frame 명령에 ebp, esp, eip 값을 현재 Context로 지정.
(.frame에 힌트로 제공)
[step 5] stack 프레임을 이동, 로컬변수를 확인하는 등 디버깅 계속 진행...

[실습2] [dump2] - 데드락 원인분석 (심화)

- ✓ 전형적인 데드락 덤프 (part 1 교육자료 참조)
- ✓ !locks 결과에 따르면 36번 스레드는 Loarder Lock을 소유하고 있지만, callstack 상으로는 모듈 Load/Unload 혹은 Thread 시작/중지 등 Loader Lock을 소유한 정황이 없다.
- ✓ 자동분석된 callstck의 일부가 깨져 보이며, 분석된 callstack이 스레드 시작/초기화 관련 함수에서 시작하지 않는다. ➔ 자동 분석결과를 믿을 수 없음!!!
- ✓ **미션 : LoaderLock이 잠기게 된 이유를 확인하라!!**

※ Special thanks to 김지훈 at http://www.slideshare.net/devgrapher/windbg-28727619#

```
0:036> !locks

CritSec ntdll!LdrpLoaderLock+0 at 77c77340
WaiterWoken        No
LockCount          13
RecursionCount     1
OwningThread       13b4
EntryCount         0
ContentionCount    9d
*** Locked
```

```
0:036> k
ChildEBP RetAddr
07f5d76c 77be6a64 ntdll!KiFastSystemCallRet
07f5d770 77bd2278 ntdll!NtWaitForSingleObject+0xc
07f5d7d4 77bd215c ntdll!RtlpWaitOnCriticalSection+0x13e
07f5d7fc 777bce18 ntdll!RtlEnterCriticalSection+0x150
07f5d848 777bd130 user32!BitmapFromDIB+0x1f
07f5d8ac 777bd6db user32!ConvertDIBBitmap+0x12b
07f5db74 777bccbf user32!ConvertDIBIcon+0x112
07f5dbc8 777bcdef user32!ObjectFromDIBResource+0xc8
07f5de20 777bf15d user32!LoadIcoCur+0x197
07f5de40 0aec14f2 user32!LoadIconW+0x1b
WARNING: Stack unwind information not available. Following frames
       may be wrong.
07f5de6c 77bfa15b ShellStreams!DllUnregisterServer+0x22c72
07f5de80 ffffffff ntdll!RtlInitializeCriticalSection+0x12
07f5deb8 77655452 0xffffffff
07f5dec8 00000000 kernel32!DeactivateActCtx+0x31
```

[실습2][step 1~2] ebp 값을 확인 ←→ callstack 테스트

✓ ebp 추정 : 앞서 했던 대로 하면 됨...

```
0:036> ~~[13b4]s
<…>
ntdll!KiFastSystemCallRet:
77be70f4 c3              ret
0:036> dps esp esp+1000
07f5d770  77be6a64 ntdll!NtWaitForSingleObject+0xc
07f5d774  77bd2278 ntdll!RtlpWaitOnCriticalSection+0x13e
07f5d778  000020cc
07f5d77c  00000000
07f5d780  00000000
<중간 생략>
07f5de7c  07f5df08
07f5de80  0aee148b ShellStreams!DllUnregisterServer+0x42c0b
07f5de84  ffffffff
<중간 생략>
07f5def4  0ae90000 ShellStreams
07f5def8  6fff0e30 msvcr80!operator new+0x1d
07f5defc  00000068
07f5df00  00000004
07f5df04  cd8dafae
07f5df08  07f5df20
07f5df0c  0aee1534 ShellStreams!DllUnregisterServer+0x42cb4
07f5df10  0000000b
07f5df14  0ae92acb ShellStreams+0x2acb
07f5df18  cd8dae12
07f5df1c  14377dd8
07f5df20  07f5df84
07f5df24  0aedd11b ShellStreams!DllUnregisterServer+0x3e89b
07f5df28  00000000
```

Trial and Error

```
0:036> k = 07f5df08
ChildEBP RetAddr
07f5d76c 77be6a64 ntdll!KiFastSystemCallRet
07f5d770 77bd2278 ntdll!NtWaitForSingleObject+0xc
07f5df08 0aee1534 ntdll!RtlpWaitOnCriticalSection+0x13e
WARNING: Stack unwind information not available. Following frames may be wrong.
07f5df20 0aedd11b ShellStreams!DllUnregisterServer+0x42cb4
07f5df94 0aedc96e ShellStreams!DllUnregisterServer+0x3e89b
07f5e0b0 77c0052e ShellStreams!DllUnregisterServer+0x3e0ee
07f5e20c 77bbe114 ntdll!LdrpLoadDll+0x4d1
07f5f968 77bd0846 ntdll!_load_config_used+0x6c
07f5fab8 77bbe114 ntdll!TppWorkerThread+0x572
07f5fb14 77c037be ntdll!_load_config_used+0x6c
07f5fb20 00000000 ntdll!_RtlUserThreadStart+0x1b

0:036> ub 77c0052e // 임의의 RET에 대해 callstack 검증!!
ntdll!LdrpLoadDll+0x49a:
77c00506 0f8410f7ffff    je      ntdll!LdrpLoadDll+0x557 (77bffc1c)
77c0050c 803d7070c77700  cmp     byte ptr [ntdll!LdrpLdrDatabaseIsSetup
      (77c77070)],0
77c00513 0f8403f7ffff    je      ntdll!LdrpLoadDll+0x557 (77bffc1c)
77c00519 393d0c70c777    cmp     dword ptr [ntdll!g_ShimsEnabled
      (77c7700c)],edi
77c0051f 0f8524fafbff    jne     ntdll!LdrpLoadDll+0x4b5 (77bbff49)
77c00525 895dfc          mov     dword ptr [ebp-4],ebx
77c00528 57              push    edi
77c00529 e8f61e0000      call    ntdll!LdrpRunInitializeRoutines (77c02424)
         // RET 직전 인스트럭션이 다음 프레임에 대한 call인지 확인!!
```

[실습2][step 3] 선택된 ebp 를 기반으로 esp, eip 유추

✓ 현재의 스택 프레임이 선택된 ebp 값이었을 때 esp, eip에 어떤 값이 들어있을 지를 상상해 보라!!

```
0:036> dps esp esp+1000
07f5e784  76086494 ole32!gComProcessActivator
...
07f5e21c  07f5e250
07f5e220  77c02322 ntdll!LdrLoadDll+0x92
07f5e224  07f5e27c
...
07f5e248  07520750
07f5e24c  110c87ac
07f5e250  07f5e28c
07f5e254  75c98c19 KERNELBASE!LoadLibraryExW+0x1d3
07f5e258  110c87ac
```

해당 함수의 스택

2. 해당 함수의 Return Address를 eip로 선정

3. 해당 함수 스택 내 임의의 주소값을 esp로 정함.

1. 이게 선정된 ebp라면...

[실습2][step 4] 선택된 ebp,esp,eip를 현재 Context에 Set

✓ 선택한 ebp, esp, eip 값을 .frame에 힌트로 제공

.frame (Set Local Context)

The **.frame** command specifies which local context (scope) is used to interpret local variables or displays the current local context.

```
.frame [/c] [/r] [FrameNumber]
.frame [/c] [/r] = BasePtr [FrameIncrement]
.frame [/c] [/r] = BasePtr StackPtr InstructionPtr
```

```
0:036> .frame /c = 07f5e250 07f5e24c 77c02322
...
ntdll!LdrLoadDll+0x92:
77c02322 8bf0            mov     esi,eax

0:036> r
Last set context:
eax=0d2d735c ebx=07f5ea70 ecx=00000003 edx=00000000 esi=778192e0 edi=00000000
eip=77c02322 esp=07f5e24c ebp=07f5e250 iopl=0
ntdll!LdrLoadDll+0x92:
77c02322 8bf0            mov     esi,eax
```

[실습2][step 5] set된 Context에서 디버깅 진행

✓ stack frame을 이동하거나 로컬변수를 확인하는 등등...

```
0:036> kn
 # ChildEBP RetAddr
00 07f5e250 75c98c19 ntdll!LdrLoadDll+0x92
01 07f5e28c 75f69d43 KERNELBASE!LoadLibraryExW+0x1d3
02 07f5e2a8 75f69cc7 ole32!LoadLibraryWithLogging+0x16
03 07f5e2cc 75f69bb6 ole32!CClassCache::CDllPathEntry::LoadDll+0xa9
04 07f5e2fc 75f690be
      ole32!CClassCache::CDllPathEntry::Create_rl+0x37
05 07f5e548 75f68f93
      ole32!CClassCache::CClassEntry::CreateDllClassEntry_rl+0xd4
06 07f5e590 75f68e99
      ole32!CClassCache::GetClassObjectActivator+0x224
07 07f5e5c8 75f68c57 ole32!CClassCache::GetClassObject+0x30
08 07f5e644 75f83170
      ole32!CServerContextActivator::CreateInstance+0x110
09 07f5e684 75f68dca
      ole32!ActivationPropertiesIn::DelegateCreateInstance+0x108
0a 07f5e6d8 75f68d3f ole32!CApartmentActivator::CreateInstance+0x112
0b 07f5e6f8 75f68ac2 ole32!CProcessActivator::CCICallback+0x6d
0c 07f5e718 75f68a73 ole32!CProcessActivator::AttemptActivation+0x2c
0d 07f5e754 75f68e2d ole32!CProcessActivator::ActivateByContext+0x4f
0e 07f5e77c 75f83170 ole32!CProcessActivator::CreateInstance+0x49
0f 07f5e7bc 75f82ef4
      ole32!ActivationPropertiesIn::DelegateCreateInstance+0x108
...
27 07f5f968 77bd0846 ntdll!RtlpTpWorkCallback+0x11d
28 07f5fac8 7765ed6c ntdll!TppWorkerThread+0x572
29 07f5fad4 77c037eb kernel32!BaseThreadInitThunk+0xe
2a 07f5fb14 77c037be ntdll!__RtlUserThreadStart+0x70
2b 07f5fb2c 00000000 ntdll!_RtlUserThreadStart+0x1b
```

```
0:036> .frame /c f
..
ole32!ActivationPropertiesIn::DelegateCreateInstance+0x108:
75f83170 8b4dfc          mov     ecx,dword ptr [ebp-4] ss:0023:07f5e7b8=13dcfe0d
0:036> dv
            this = 0x07f5ea70
        pUnkOuter = 0x00000000
    ppActPropsOut = 0x07f5f17c
pReplaceClassInfo = 0xffffffff
            clsid = struct _GUID {07f5f0e0-0000-0000-70ea-f5071ceaf507}
       pClassInfo = 0x00000001
```

[실습2] 결론

- ✓ 추정해낸 ebp 값을 이용해 콜스택을 구성 → 현재 DLL Load가 진행중임을 확인!!
- ✓ ShellStreams.dll의 DllMain에서 동기화를 수행하는 api를 호출한 것이 DeadLock의 원인
  (참고 : http://www.jiniya.net/tt/788)

```
0:036> kbn = 07f5e21c

 # ChildEBP RetAddr  Args to Child
00 07f5d76c 77be6a64 77bd2278 000020cc 00000000 ntdll!KiFastSystemCallRet
01 07f5d770 77bd2278 000020cc 00000000 00000000 ntdll!NtWaitForSingleObject+0xc
02 07f5e21c 77c02322 07f5e27c 07f5e248 00000000 ntdll!RtlpWaitOnCriticalSection+0x13e
03 07f5e250 75c98c19 110c87ac 07f5e294 07f5e27c ntdll!LdrLoadDll+0x92
04 07f5e28c 75f69d43 00000000 00000000 110c87ac KERNELBASE!LoadLibraryExW+0x1d3
05 07f5e2a8 75f69cc7 00000000 07f5e324 00000008 ole32!LoadLibraryWithLogging+0x16
06 07f5e2cc 75f69bb6 07f5e324 07f5e2f0 07f5e2f4 ole32!CClassCache::CDllPathEntry::LoadDll+0xa9
07 07f5e2fc 75f690be 07f5e324 07f5e60c 07f5e31c ole32!CClassCache::CDllPathEntry::Create_rl+0x37
08 07f5e548 75f68f93 00000001 07f5e60c 07f5e578 ole32!CClassCache::CClassEntry::CreateDllClassEntry_rl+0xd4
09 07f5e590 75f68e99 00000001 002f607c 07f5e5bc ole32!CClassCache::GetClassObjectActivator+0x224
0a 07f5e5c8 75f68c57 07f5e60c 00000000 07f5ec14 ole32!CClassCache::GetClassObject+0x30
<이하 생략>

0:036> du 07f5e324
07f5e324  "C:\Program Files\Common Files\Ap"
07f5e364  "ple\Internet Services\ShellStrea"
07f5e3a4  "ms.dll "

.frame /c = 07f5e21c 07f5e24c 77c02322
```

[실습 3] Stack 분석 응용 (2)

[실습3] [dump9] - 비정상 종료 원인 분석

- ✓ Stack의 내용을 육안으로 분석해야 할 때도 있음!!
- ✓ 로컬변수, 아규먼트 등 스택의 내용을 분석하여 장애 발생 시의 상황을 추정해 보자.
- ✓ "CAccept::ManagerThread" 에 진입한 이후에 도대체 무슨 일이??

```
0:052> k
ChildEBP RetAddr
097ffdb8 616b1c1e tcpsvc!CLock::`vftable' // 자동 분석 결과가 이상하다???
097ffdd4 616b2ac7 tcpsvc!CAccept::ManagerThread+0x3e
097ffdfc 617910a9 tcpsvc!XThreads::Handler+0x97
097ffe34 61791133 tcpsvc!_callthreadstartex+0x1b
097ffe40 7613336a tcpsvc!_threadstartex+0x64
WARNING: Stack unwind information not available. Following frames may be wrong.
097ffe4c 773f9f72 kernel32!BaseThreadInitThunk+0x12
097ffe8c 773f9f45 ntdll!RtlInitializeExceptionChain+0x63
097ffea4 00000000 ntdll!RtlInitializeExceptionChain+0x36
```

[실습3] [dump9] - 비정상 종료 원인 분석

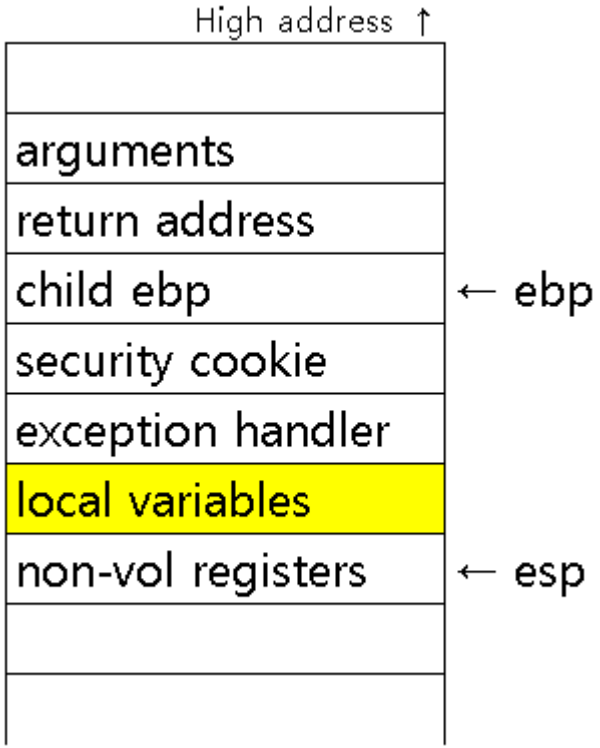
✓ stack 내용을 분석하여 잃어버린 함수의 흐름을 쫓아가보자.

```
0:052> dps esp esp+100
097ffd64  616c39f3 tcpsvc!CTCPService::AcceptCallback+0x1b3
097ffd68  00000001
097ffd6c  617c3050 tcpsvc!`string'+0xcfc
097ffd70  616c39dd tcpsvc!CTCPService::AcceptCallback+0x19d
097ffd74  3f054ad0
097ffd78  00000000
097ffd7c  00000000
097ffd80  03ee4420
097ffd84  00000000
097ffd88  01000000 xserver!__ImageBase
097ffd8c  00f52c10
097ffd90  61823d90 netserver!CNetServer::ConnectCallback
097ffd94  03f48578
097ffd98  00000000
097ffd9c  02992c98
097ffda0  00000011
097ffda4  097ff928
097ffda8  097ffe24
097ffdac  61790ee0 tcpsvc!_except_handler4
097ffdb0  57055c30
097ffdb4  fffffffe
097ffdb8  097ffdd4
097ffdbc  616b1c1e tcpsvc!CAccept::ManagerThread+0x3e
```

수수께끼의 답은 이 안에 있다!!

stack의 좀 더 자세한 지도

- ✓ 함수의 중간에서 할당된 로컬 변수도 실제로는 함수 시작 시 일괄 할당됨.
- ✓ 로컬변수는 선언된 순서대로 stack에 할당되지 않음.
- ✓ 실제로 스택엔 로컬변수, return address, arguments 외에도 많은 것이...



To Be Continued…

~~(1) Debugging 개론~~
~~(2) x86 stack inside~~
(3) heap inside
(4) x64 stack inside