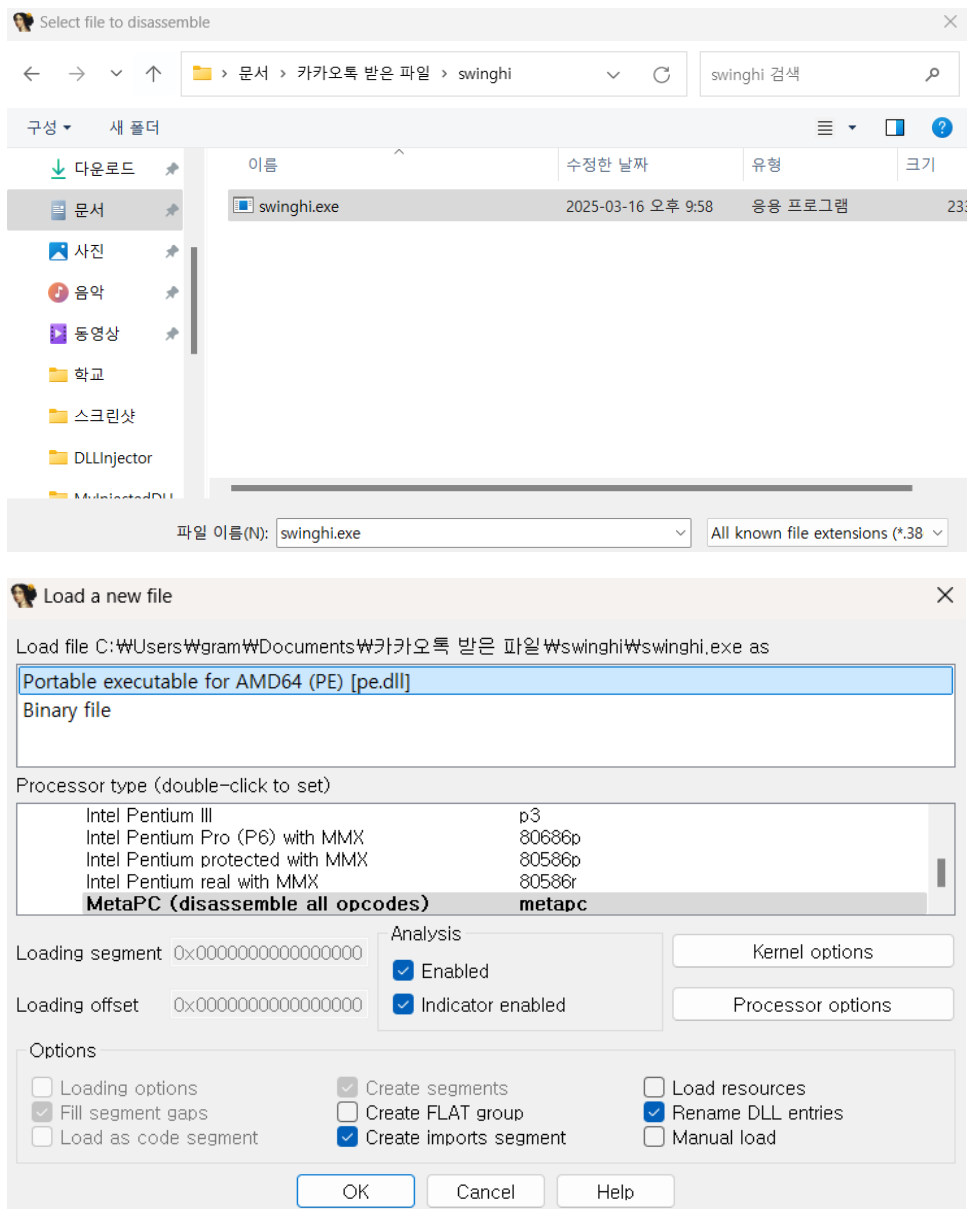


Swinghi.exe 정적/동적 분석 실습

32기 정유나

정적 분석



IDA를 켜서 다운받고 압축 해제해둔 swinghi.exe파일을 선택한다.

1. 문자열 검색

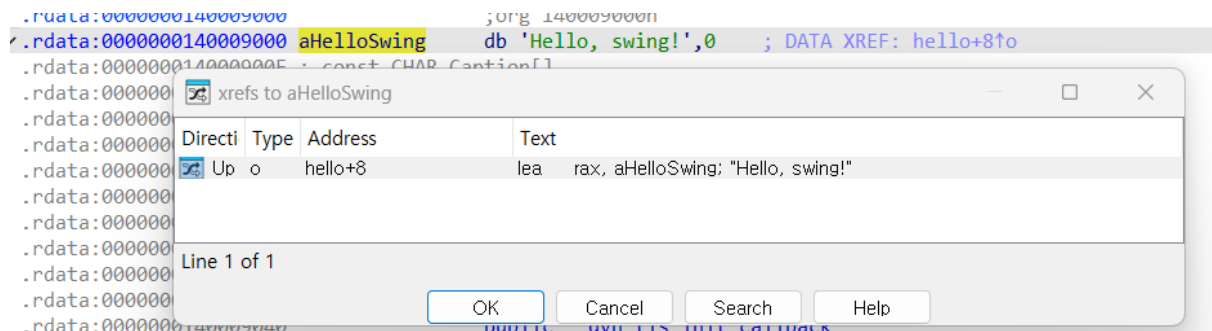
Address	Length	Type	String
.rdata:0000000E	0000000E	C	Hello, swing!
.rdata:00000009	00000009	C	Greeting
.rdata:00000012	00000012	C	b: %d\Wnc: %d\Wnd: %d
.rdata:00000013	00000013	C	Calculation Result
.rdata:0000001F	0000001F	C	Argument domain error (DOMAIN)
.rdata:0000001C	0000001C	C	Argument singularity (SIGN)
.rdata:00000020	00000020	C	Overflow range error (OVERFLOW)
.rdata:00000025	00000025	C	Partial loss of significance (PLOSS)
.rdata:00000023	00000023	C	Total loss of significance (TLOSS)
.rdata:00000036	00000036	C	The result is too small to be represented (UNDERFLOW)
.rdata:0000000E	0000000E	C	Unknown error
.rdata:0000002B	0000002B	C	_matherr(): %s in %s(%g, %g) (retval=%g)\Wn
.rdata:0000001C	0000001C	C	Mingw-w64 runtime failure:\Wn
.rdata:00000020	00000020	C	Address %p has no image-section
.rdata:00000031	00000031	C	VirtualQuery failed for %d bytes at address %p
.rdata:00000027	00000027	C	VirtualProtect failed with code 0x%x
.rdata:00000032	00000032	C	Unknown pseudo relocation protocol version %d,\Wn
.rdata:0000002A	0000002A	C	Unknown pseudo relocation bit size %d,\Wn
.rdata:00000053	00000053	C	%d bit pseudo relocation at %p out of range, targeting %p, yielding the value %p,\Wn
.rdata:00000007	00000007	C	(null)
.rdata:00000009	00000009	C	Infinity
.rdata:00000014	00000014	C	GCC: (GNU) 13-win32
.rdata:00000014	00000014	C	GCC: (GNU) 13-win32
.rdata:00000014	00000014	C	GCC: (GNU) 13-win32
.rdata:00000014	00000014	C	GCC: (GNU) 13-win32
.rdata:00000014	00000014	C	GCC: (GNU) 13-win32
.rdata:00000014	00000014	C	GCC: (GNU) 13-win32
.rdata:00000014	00000014	C	GCC: (GNU) 13-win32
.rdata:00000014	00000014	C	GCC: (GNU) 13-win32
.rdata:00000014	00000014	C	GCC: (GNU) 13-win32
.rdata:00000014	00000014	C	GCC: (GNU) 13-win32
.rdata:00000014	00000014	C	GCC: (GNU) 13-win32
.rdata:00000014	00000014	C	GCC: (GNU) 13-win32
.rdata:00000014	00000014	C	GCC: (GNU) 13-win32
.rdata:00000014	00000014	C	GCC: (GNU) 13-win32

IDA는 바이너리에 포함된 문자열을 쉽게 찾을 수 있도록 문자열 탐색 기능을 제공하고 있다. Shift + F12를 눌러 이 기능을 사용할 수 있다. 위에서 바이너리에 포함된 문자열이 열거된 Strings 창을 확인할 수 있다.

이 중 'Hello, swing!' 이라는 문자열을 더블 클릭하면 아래와 같은 결과를 확인할 수 있다.

```
.rdata:0000000140009000 ; Segment type: Pure data
.rdata:0000000140009000 ; Segment permissions: Read
.rdata:0000000140009000 _rdata segment para public 'DATA' use64
.rdata:0000000140009000 assume cs:_rdata
.rdata:0000000140009000 ;org 140009000h
v .rdata:0000000140009000 aHelloSwing db 'Hello, swing!',0 ; DATA XREF: hello+8fo
.rdata:000000014000900E ; const CHAR Caption[]
.rdata:000000014000900E Caption db 'Greeting',0 ; DATA XREF: hello+23fo
.rdata:0000000140009017 ; const char Format[]
.rdata:0000000140009017 Format db 'b: %d',0Ah ; DATA XREF: cal+4Ffo
.rdata:000000014000901D db 'c: %d',0Ah
.rdata:0000000140009023 db 'd: %d',0
.rdata:0000000140009029 ; const CHAR aCalculationRes[]
.rdata:0000000140009029 aCalculationRes db 'Calculation Result',0 ; DATA XREF: cal+68fo
.rdata:000000014000903C align 20h
.rdata:0000000140009040 public __dyn_tls_init_callback
.rdata:0000000140009040 ; const PIMAGE_TLS_CALLBACK __dyn_tls_init_callback
.rdata:0000000140009040 __dyn_tls_init_callback dq offset __dyn_tls_init ; DATA XREF: .rdata:_refptr__dyn_tls_ini
.rdata:0000000140009048 align 20h
.rdata:0000000140009060 public _tls_used
.rdata:0000000140009060 ; const IMAGE_TLS_DIRECTORY tls_used
> .rdata:0000000140009060 _tls_used IMAGE_TLS_DIRECTORY <14000F000h, 14000F008h, 14000C08Ch, 14000E03
```

2. 상호 참조 추적



위의 문자열 검색 결과에서 'Hello, swing!'이라는 문자열이 어디에 사용되는지 상호 참조(Cross Reference, XRef) 기능을 통해 확인해볼 것이다. aHelloSwing을 클릭하고 상호 참조의 단축키 X를 누르면 xrefs(cross reference) 창을 볼 수 있다.

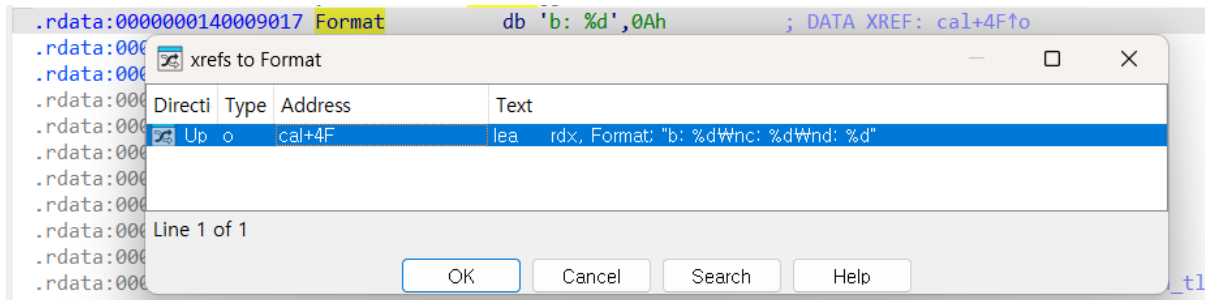
이 창에는 해당 변수를 참조하는 모든 주소가 출력되는데, aHelloSwing의 경우 hello함수에 사용되고 있음을 확인할 수 있다.

The screenshot shows a window displaying assembly code for a function named 'hello'. The code is as follows:

```
; Attributes: bp-based frame

public hello
hello proc near
push    rbp
mov     rbp, rsp
sub     rsp, 20h
lea     rax, aHelloSwing ; "Hello, swing!"
mov     cs:str, rax
mov     rax, cs:str
mov     r9d, 40h ; '@' ; uType
lea     r8, Caption ; "Greeting"
mov     rdx, rax ; lpText
mov     ecx, 0 ; hWnd
mov     rax, cs:__imp_MessageBoxA
call    rax ; __imp_MessageBoxA
nop
add     rsp, 20h
pop     rbp
retn
hello endp
```

Address의 hello 함수를 더블 클릭하여 이렇게 hello 함수를 볼 수 있다.



이번에는 다른 함수를 찾아보겠다. aHelloSwing에서 조금 내려간 Format의 상호 참조를 추적해보면 새로운 cal함수를 찾을 수 있다.

```
; Attributes: bp-based frame

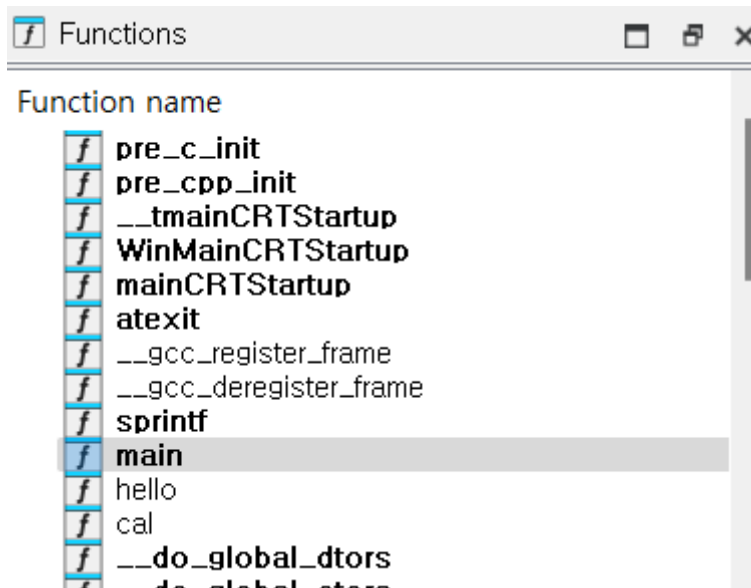
public cal
cal proc near

var_90= dword ptr -90h
Buffer= byte ptr -80h
var_14= dword ptr -14h
var_10= dword ptr -10h
var_C= dword ptr -0Ch
var_8= dword ptr -8
var_4= dword ptr -4

push    rbp
mov     rbp, rsp
sub     rsp, 0B0h
mov     [rbp+var_4], 3
mov     [rbp+var_8], 2
mov     edx, [rbp+var_4]
mov     eax, [rbp+var_8]
add     eax, edx
mov     [rbp+var_C], eax
mov     eax, [rbp+var_8]
sub     eax, [rbp+var_4]
mov     [rbp+var_10], eax
mov     eax, [rbp+var_4]
imul    eax, [rbp+var_8]
mov     [rbp+var_14], eax
mov     r8d, [rbp+var_10]
mov     ecx, [rbp+var_C]
lea     rax, [rbp+Buffer]
mov     edx, [rbp+var_14]
mov     [rsp+0B0h+var_90], edx
mov     r9d, r8d
mov     r8d, ecx
lea     rdx, Format ; "b: %d\\nc: %d\\nd: %d"
mov     rcx, rax ; Buffer
call    sprintf
lea     rax, [rbp+Buffer]
mov     r9d, 40h ; '@' ; uType
lea     r8, aCalculationRes ; "Calculation Result"
mov     rdx, rax ; lpText
mov     ecx, 0 ; hWnd
mov     rax, cs:__imp_MessageBoxA
call    rax ; __imp_MessageBoxA
nop
add     rsp, 0B0h
pop     rbp
retn
cal endp
```

cal함수 또한 xref창에서의 주소 더블클릭으로 이렇게 확인할 수 있다.

3. main 함수 찾은 뒤 디컴파일



main함수는 IDA의 functions창에서 쉽게 찾을 수 있다.

```
; Attributes: bp-based frame

; int __fastcall main(int argc, const char **argv, const char **envp)
public main
main proc near

dwMilliseconds= dword ptr -4

push    rbp
mov     rbp, rsp
sub     rsp, 30h
call    __main
mov     [rbp+dwMilliseconds], 1F4h
mov     eax, [rbp+dwMilliseconds]
mov     ecx, eax             ; dwMilliseconds
mov     rax, cs:__imp_Sleep
call    rax ; __imp_Sleep
call    hello
call    cal
mov     eax, 0
add     rsp, 30h
pop     rbp
retn
main endp
```

이게 main 함수의 어셈블리이다. 여기서 F5를 눌러 디컴파일된 결과를 볼 수 있다.

```
int __fastcall main(int argc, const char **argv, const char **envp)
{
    _main();
    Sleep(0x1F4u);
    hello();
    cal();
    return 0;
}
```

이렇게 디컴파일된 결과를 볼 수 있다.

그러면 이제 어셈블리와 디컴파일 결과를 비교하며 main 함수를 분석해보자.

1. 스택 프레임 설정

```
push    rbp
mov     rbp, rsp
sub     rsp, 30h
```

push rbp: 현재 rbp 값을 스택에 저장하여 이전 프레임을 보존함.

mov rbp, rsp: 현재 rsp 값을 rbp로 설정하여 새로운 스택 프레임을 만듦.

sub rsp, 30h: 지역 변수를 위한 공간 확보 (0x30 바이트).

2. _main() 함수 호출

```
call    __main
```

```
3 | main();
```

_main()을 호출한다.

3. Sleep(500) 호출

```
call    __imp_Sleep
mov     [rbp+dwMilliseconds], 1F4h
mov     eax, [rbp+dwMilliseconds]
mov     ecx, eax ; dwMilliseconds
mov     rax, cs:__imp_Sleep
call    rax ; __imp_Sleep
```

```
4 | Sleep(0x1F4u);
```

0x1F4 (500)를 dwMilliseconds 변수에 저장.

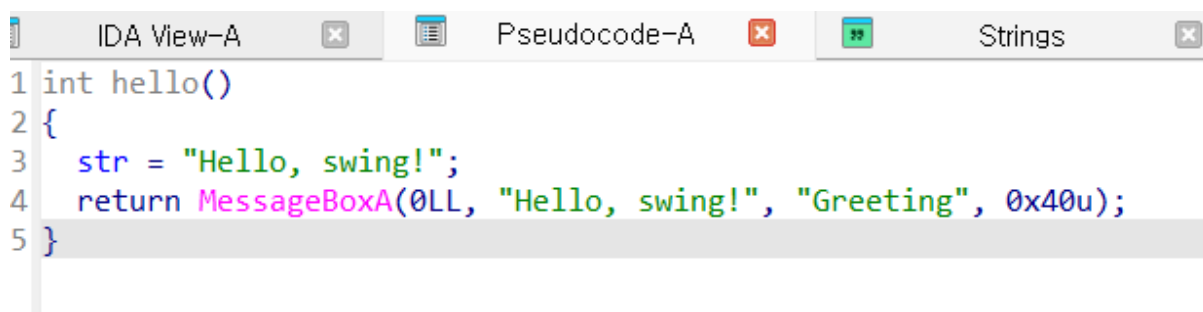
이를 ecx 레지스터로 이동하여 Sleep() 함수의 인자로 전달.

call rax를 통해 Sleep()을 호출.

4. hello() 함수 호출

```
call    hello
5  hello();
```

hello() 함수를 호출한다.



The screenshot shows the IDA Pro interface with three tabs: 'IDA View-A', 'Pseudocode-A', and 'Strings'. The 'Pseudocode-A' tab is active, displaying the following C code for the 'hello' function:

```
1 int hello()
2 {
3     str = "Hello, swing!";
4     return MessageBoxA(0LL, "Hello, swing!", "Greeting", 0x40u);
5 }
```

hello함수를 디컴파일한 결과는 위와 같은데, 여기서 MessageBoxA() 함수의 사용, "Hello, swing!"과 "Greeting"의 문자열의 사용 등을 보아 해당 문자열들이 표시되는 메시지창을 표시하는 함수임을 알 수 있다.

특히 hello함수의 어셈블리를 참고하여 MessageBoxA() 인자들을 분석해보면 lpText = "Hello, swing!" → 메시지 박스의 본문 텍스트, lpCaption = "Greeting" → 메시지 박스의 제목임을 알 수 있다.

즉 hello함수는 "Greeting"이라는 제목의 메시지 박스를 띄우고, 본문에 "Hello, swing!"이라는 문자열을 표시하는 함수라 결론지을 수 있다.

5. cal() 함수 호출

```
call    hello
call    cal
6  cal();
```

cal() 함수를 호출한다.

```

int cal()
{
    char Buffer[108]; // [rsp+30h] [rbp-80h] BYREF
    int v2; // [rsp+9Ch] [rbp-14h]
    int v3; // [rsp+A0h] [rbp-10h]
    int v4; // [rsp+A4h] [rbp-Ch]
    int v5; // [rsp+A8h] [rbp-8h]
    int v6; // [rsp+AC] [rbp-4h]

    v6 = 3;
    v5 = 2;
    v4 = 5;
    v3 = -1;
    v2 = 6;
    sprintf(Buffer, "b: %d\nc: %d\nd: %d", 5, -1, 6);
    return MessageBoxA(0LL, Buffer, "Calculation Result", 0x40u);
}

```

cal함수를 디컴파일한 결과는 위와 같다. 이 코드를 참고해보면 cal함수는 지역 변수들을 초기화하고 무언가 계산을 한 후, MessageBoxA 함수를 사용하여 그 결과를 메시지창으로 표시하는 함수임을 짐작할 수 있다.

어셈블리를 참고해보면 $v4 = v5 + v6 = 2 + 3 = 5$, $v3 = v5 - v6 = 2 - 3 = -1$, $v2 = v5 * v6 = 3 * 2 = 6$ 등을 확인할 수 있는데, 이를 보아 cal함수는 3과 2라는 두 숫자를 저장 ($v6 = 3$, $v5 = 2$) 한 뒤 이를 사용하여 덧셈 (5), 뺄셈 (-1), 곱셈 (6)을 수행하고 이를 메시지창으로 표시하는 역할을 한다는 것을 알 수 있다.

6. 반환값 설정 후 종료

```

mov     eax, 0
add     rsp, 30h
pop     rbp
retn

7 | return 0;
8 | }

```

eax에 0을 저장하여 반환값 설정 (return 0;).

rsp를 원래 값으로 복원 (add rsp, 30h).

rbp를 복원 (pop rbp).

retn을 통해 함수 종료.

결론적으로

- hello() 함수는 "Hello, swing!"이라는 메시지를 포함한 "Greeting"이라는 제목의 메시지 박스를 띄운다.
- cal() 함수는 3과 2의 덧셈, 뺄셈, 곱셈 결과를 계산한 뒤 "Calculation Result"라는 제목의 메시지 박스를 띄운다.

main() 함수는

1. _main()을 호출하여 초기화한 후,
2. 0.5초 대기 (Sleep(500))
3. hello() 메시지 박스 출력
4. cal() 메시지 박스 출력
5. 프로그램 종료

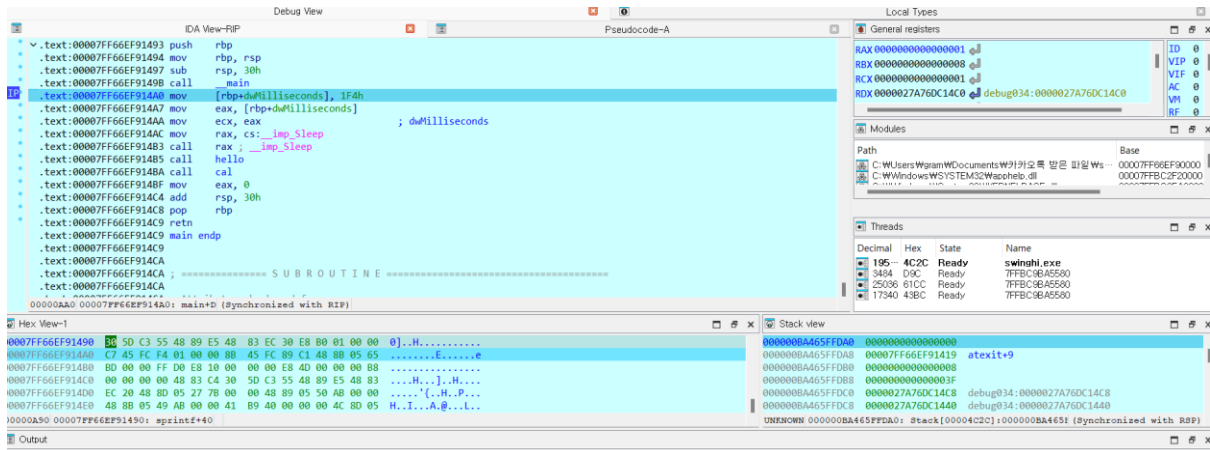
라는 순서로 실행된다.

동적 분석

1. 중단점 설정 및 F9 실행

```
int __fastcall main(int argc, const char **argv, const char **envp)
{
    _main();
    Sleep(0x1F4u);
    hello();
    cal();
    return 0;
}
```

Main 함수에 중단점을 설정한다. 이후 디버깅을 시작한다.



이렇게 동적 분석을 할 수 있는 환경을 만들 수 있다.

2. 한 단계 실행 F8로 main 동작 분석

1. sub rsp, 30h 명령어를 실행하여 main 함수가 사용할 스택 공간을 0x30(48) 바이트만큼 확보한다.
2. mov [rbp+dwMilliseconds], 1F4h 실행으로 dwMilliseconds 변수에 0x1F4(=500)를 저장한다.
3. mov eax, [rbp+dwMilliseconds] 실행으로 dwMilliseconds의 값을 eax 레지스터에 로드한다. eax = 500 (0x1F4)가 된다.
4. mov ecx, eax 실행으로 eax 값을 ecx로 이동시킨다. 이는 Sleep 함수의 첫 번째 인자로 사용될 값이다. 즉, ecx = 500 (0x1F4)이므로, Sleep(500)이 실행될 예정이다.
5. call Sleep 실행으로 Sleep(500)이 호출된다. 프로그램이 500ms 동안 대기 상태에 들어간다. 반환 주소가 스택에 저장된다.
6. call hello 실행으로 hello 함수가 호출된다. hello 함수 실행을 위해 현재 실행 주소가 스택에 저장되고, hello 함수로 분기한다. hello 함수의 실행이 끝나면 스택에서 반환 주소를 복원하고 원래 실행 흐름으로 복귀한다.
7. call cal 실행으로 cal 함수가 호출된다. cal 함수 실행을 위해 현재 실행 주소가 스택에 저장되고, cal 함수로 분기한다. cal 함수의 실행이 끝나면 스택에서 반환 주소를 복원하고 원래 실행 흐름으로 복귀한다.
8. mov eax, 0 실행으로 eax에 반환값 0을 설정한다. add rsp, 30h 실행으로 sub rsp, 30h로 확보했던 스택 공간을 해제한다. pop rbp 실행으로 rbp 값을 복원한다.
9. ret 명령어가 실행되면서 main 함수가 종료된다. 스택에 저장된 반환 주소로 복귀하여 OS로 실행 흐름이 돌아간다.

3. f7로 hello()내부로 들어가기

```

.text:00007FF66EF914B3 call rax ; __imp_Sleep
.text:00007FF66EF914B5 call hello
.text:00007FF66EF914BA call cal
.text:00007FF66EF914BF mov eax, 0
.text:00007FF66EF914C4 add rsp, 30h
.text:00007FF66EF914C8 pop rbp
.text:00007FF66EF914C9 retn
.text:00007FF66EF914C9 main endp

```

Hello 함수를 호출하는 부분에 중단점을 설정하고 실행한다.

```

.text:00007FF66EF9149B call __main
.text:00007FF66EF914A0 mov [rbp+dwMilliseconds], 1F4h
.text:00007FF66EF914A7 mov eax, [rbp+dwMilliseconds]
.text:00007FF66EF914AA mov ecx, eax ; dwMilliseconds
.text:00007FF66EF914AC mov rax, cs:__imp_Sleep
.text:00007FF66EF914B3 call rax ; __imp_Sleep
.text:00007FF66EF914B5 call hello
.text:00007FF66EF914BA call cal
.text:00007FF66EF914BF mov eax, 0

```

디버깅을 다시 시작해서 hello함수에 도달한다.

view-HIP Pseudocode

```

.text:00007FF66EF914CA ; int hello()
.text:00007FF66EF914CA public hello
.text:00007FF66EF914CA hello proc near
.text:00007FF66EF914CA push rbp
.text:00007FF66EF914CB mov rbp, rsp
.text:00007FF66EF914CE sub rsp, 20h
.text:00007FF66EF914D2 lea rax, aHelloSwing ; "Hello, swing!"
.text:00007FF66EF914D9 mov cs:str, rax
.text:00007FF66EF914E0 mov rax, cs:str
.text:00007FF66EF914E7 mov r9d, 40h ; '@' ; uType
.text:00007FF66EF914ED lea r8, Caption ; "Greeting"
.text:00007FF66EF914F4 mov rdx, rax ; lpText
.text:00007FF66EF914F7 mov ecx, 0 ; hWnd
.text:00007FF66EF914FC mov rax, cs:__imp_MessageBoxA
.text:00007FF66EF91503 call rax ; __imp_MessageBoxA
.text:00007FF66EF91505 nop
.text:00007FF66EF91506 add rsp, 20h
.text:00007FF66EF9150A pop rbp
.text:00007FF66EF9150B retn
.text:00007FF66EF9150B hello endp

```

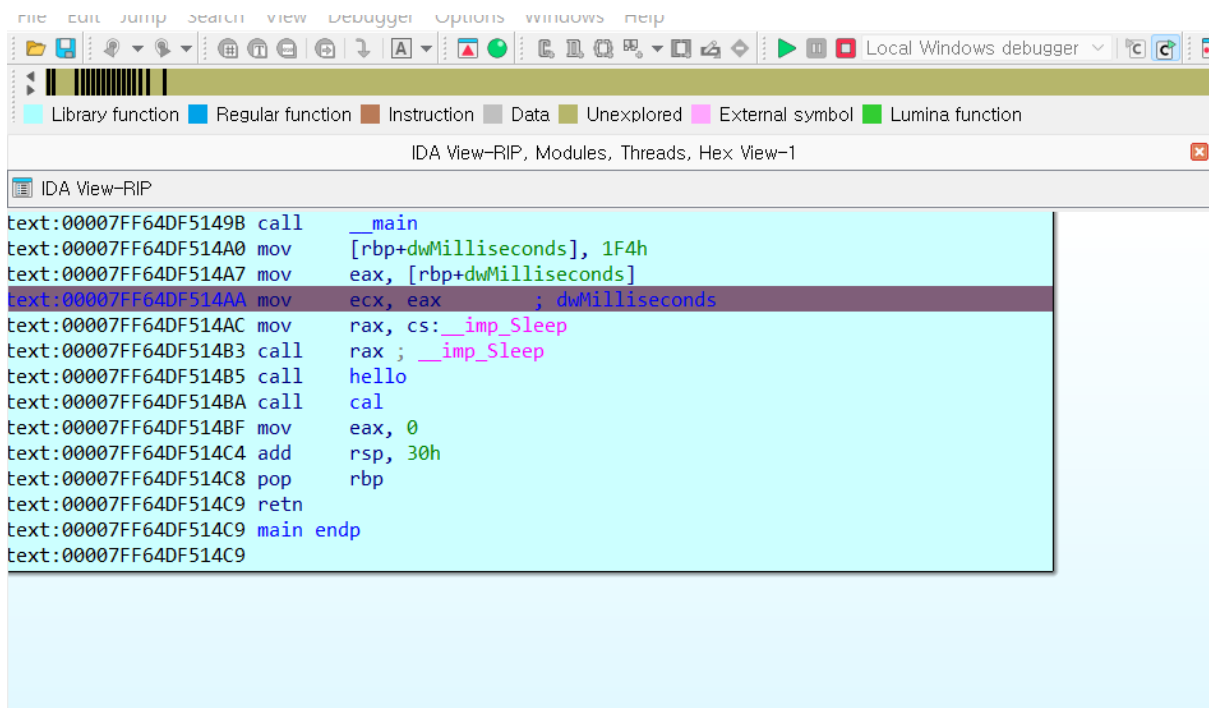
단축키를 통해 hello 함수 내부로 들어갈 수 있다.

4. 실행 중인 프로세스 조작하기

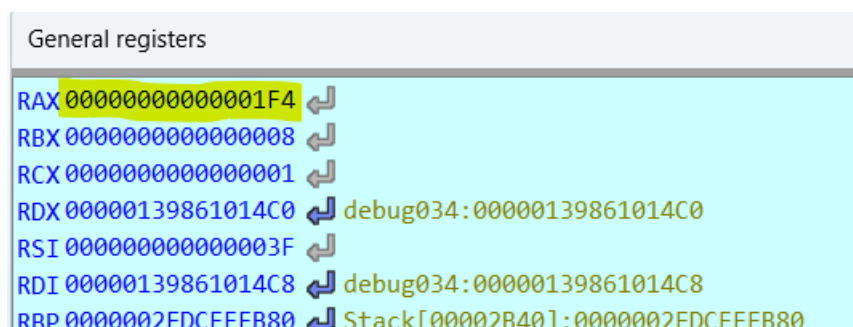
```
push    rbp
mov     rbp, rsp
sub     rsp, 30h
call    __main
mov     [rbp+dwMilliseconds], 1F4h
mov     eax, [rbp+dwMilliseconds]
mov     ecx, eax ; dwMilliseconds
mov     rax, cs:__imp_Sleep
call    rax ; __imp_Sleep
call    hello
call    cal
```

delay 값을 1000000(0xF4240)으로 변경하여 1000초 동안 프로세스를 정지시키는 실습이다.

mov ecx, eax (0x1F4 값을 ecx에 저장하는 부분)에 중단점을 설정한다.



단축키로 실행한다. mov ecx, eax에서 eax 값을 RCX에 전달했으므로 여기서는 스택이 아니라 레지스터로 전달된다. 따라서 레지스터 창을 켜다.



현재 0x1F4 (500)으로 설정되어 있으므로 0xF4240 (1000000)로 바꿔주어야 한다.

```
FDD: failed to get FDD file details  
Command "ModifyRegister" failed  
Command "ModifyRegister" failed  
Command "ModifyRegister" failed  
Command "ModifyRegister" failed
```

그런데 modify register가 실행이 안된다.