

Dreamhack Quiz: x86 Assembly

Q: end로 점프하면 프로그램이 종료된다고 가정하자. 프로그램이 종료됐을 때, 0x400000 부터 0x400019까지의 데이터를 대응되는 아스키 문자로 변환하면?

[Register]

rcx = 0

rdx = 0

rsi = 0x400000

[Memory]

0x400000 | 0x67 0x55 0x5c 0x53 0x5f 0x5d 0x55 0x10

0x400008 | 0x44 0x5f 0x10 0x51 0x43 0x43 0x55 0x5d

0x400010 | 0x52 0x5c 0x49 0x10 0x47 0x5f 0x42 0x5c

0x400018 | 0x54 0x11 0x00 0x00 0x00 0x00 0x00 0x00

[code]

1: mov dl, BYTE PTR[rsi+rcx]

2: xor dl, 0x30

3: mov BYTE PTR[rsi+rcx], dl

4: inc rcx

5: cmp rcx, 0x19

6: jg end

7: jmp 1

정답: Welcome to assembly world!

[코드 분석]

mov dl, BYTE PTR [rsi+rcx]

rcx와 rsi에 더한 주소에서 값의 1byte를 가져와 dl(register)에 저장한다. rcx = 0이고, rsi = 0x400000이므로, 0x400000 주소의 값을 dl에 가져온다.

xor dl, 0x30

dl에 저장된 값을 0x30과 XOR 연산한다.

mov BYTE PTR [rsi+rcx], dl

변환된 dl의 값을 다시 원래 메모리 위치에 저장한다.

```
inc rcx
```

rcx 1증가.

```
cmp rcx, 0x19
```

rcx의 값이 0x19(25)인지 비교한다. 비교 결과에 따라 ZF(zero flag 값이 같을 때), SF(sign flag 결과가 음수일 때), CF(carry flag 빌림이 발생할 때) 등이 설정된다. cmp는 두 번째 피연산자를 첫 번째 피연산자에서 빼는 연산을 수행한다.

```
jg end
```

jg는 조건부 점프 명령어로 rcx가 0x19보다 크면 end로 점프하여 프로그램이 종료된다. jg 명령어는 “Jump if Greater”을 의미하며 ZF가 설정되지 않고 SF와 OF(overflow flag)가 같을 때 점프한다. 즉 두 값 중 첫 번째 값이 두 번째 값보다 클 때 만족함을 의미한다.

rcx > 0x19: ZF=0, SF=0(rcx-0x19는 양수이므로), OF=0

rcx < 0x19: ZF=0, SF=1(rcx-0x19는 음수이므로), OF=0

rcx > 0x19: ZF=1(결과가 0이므로), SF=0, OF=0

```
jmp 1
```

rcx가 0x19보다 크지 않은 경우 다시 1번 명령어로 점프하여 위 과정을 반복한다.

[과정]

이 프로그램은 메모리의 25바이트를 1바이트씩 순차적으로 읽어 XOR 변환 (0x30과 XOR)한 후, 그 결과를 다시 같은 위치에 저장한다. 이후 최종적으로 변환된 데이터를 아스키 문자로 변환한다.

- (0) $0x67 \wedge 0x30 = 0x57 \rightarrow 'W'$
- (1) $0x55 \wedge 0x30 = 0x65 \rightarrow 'e'$
- (2) $0x5c \wedge 0x30 = 0x6c \rightarrow 'l'$
- (3) $0x53 \wedge 0x30 = 0x63 \rightarrow 'c'$
- (4) $0x5f \wedge 0x30 = 0x6f \rightarrow 'o'$
- (5) $0x5d \wedge 0x30 = 0x6d \rightarrow 'm'$
- (6) $0x55 \wedge 0x30 = 0x65 \rightarrow 'e'$
- (7) $0x10 \wedge 0x30 = 0x20 \rightarrow ' '$
- (8) $0x44 \wedge 0x30 = 0x74 \rightarrow 't'$
- (9) $0x5f \wedge 0x30 = 0x6f \rightarrow 'o'$

- (10) $0x10 \wedge 0x30 = 0x20 \rightarrow ' '$
- (11) $0x51 \wedge 0x30 = 0x61 \rightarrow 'a'$
- (12) $0x43 \wedge 0x30 = 0x73 \rightarrow 's'$
- (13) $0x43 \wedge 0x30 = 0x73 \rightarrow 's'$
- (14) $0x55 \wedge 0x30 = 0x65 \rightarrow 'e'$
- (15) $0x5d \wedge 0x30 = 0x6d \rightarrow 'm'$
- (16) $0x52 \wedge 0x30 = 0x62 \rightarrow 'b'$
- (17) $0x5c \wedge 0x30 = 0x6c \rightarrow 'l'$
- (18) $0x49 \wedge 0x30 = 0x79 \rightarrow 'y'$
- (19) $0x10 \wedge 0x30 = 0x20 \rightarrow ' '$
- (20) $0x47 \wedge 0x30 = 0x77 \rightarrow 'w'$
- (21) $0x5f \wedge 0x30 = 0x6f \rightarrow 'o'$
- (22) $0x42 \wedge 0x30 = 0x72 \rightarrow 'r'$
- (23) $0x5c \wedge 0x30 = 0x6c \rightarrow 'l'$
- (24) $0x54 \wedge 0x30 = 0x64 \rightarrow 'd'$
- (25) $0x11 \wedge 0x30 = 0x21 \rightarrow '!$

리버싱 도구

디버깅 도구 (Ollydbg, IDA pro, Immunity Debugger, Windbg)

파일을 어셈블리어로 디스어셈블해서 코드의 실행 흐름을 추적하고, 실시간 분석에 사용된다. 주로 코드의 로직을 이해하고, 오류를 찾는 데 중점을 둔다.

IDA Pro: 디스어셈블러와 디버깅 기능을 모두 제공하는 강력한 도구로, 복잡한 바이너리 분석에 적합.

→ x64dbg: 사용자 친화적인 인터페이스를 제공하며, 실시간으로 프로그램의 동작을 분석하는 데 유용.

정적 분석 도구 (Detours, Eceinfo, PEiD, PEView, Stud_PE, LoardPE, strings, Bintext)

바이너리를 실행하지 않고, 파일의 구조와 내부 구성 요소를 분석하는데 사용

PEView: PE 파일의 구조(예: 헤더, 섹션, 임포트/익스포트 테이블 등)를 분석하는 도구로, PE 파일의 내부를 상세히 파악하는 데 유용.

Strings: 바이너리 파일 내부에서 ASCII나 Unicode 문자열을 추출하여, 파일 내에 포함된 텍스트나 API 호출을 분석하는 데 사용.

동적 분석 도구-유저레벨 (Process Monitor, Process Explorer, Wireshark)

프로그램이 실행되는 동안 발생하는 다양한 시스템 활동을 모니터링하고 분석하는 데 사용된다. 프로그램이 실제로 어떻게 동작하는지, 어떤 시스템 리소스를 사용하는지 등을 확인하는 데 초점을 맞춘다.

Process Monitor: 파일 시스템, 레지스트리, 프로세스, 스레드 활동 등을 실시간으로 모니터링할 수 있는 도구로, 시스템에서 발생하는 다양한 이벤트를 추적할 수 있음.

Wireshark: 네트워크 트래픽을 캡처하고 분석하는 도구로, 네트워크 상에서 발생하는 모든 패킷을 확인하고 분석하는 데 사용.
--

동적 분석 툴-커널레벨 (GMER, PC Hunter)

시스템의 심층적인 동작을 분석하고, 특히 루트킷이나 기타 고급 악성코드를 탐지하는데 유용하다.

GMER: 루트킷 탐지에 특화된 도구로, 시스템 콜 훅킹이나 숨겨진 프로세스를 탐지하는 데 매우 효과적.
--

PC Hunter: 심층적인 커널 모드 분석이 필요한 경우 사용되며, 루트킷 탐지에 활용.
--

기타 리버싱 도구(패커/언패커 도구, 디컴파일러, 메모리 분석 도구, 레지스트리 분석 도구 e.t.c...)

UPX/Unpacker: 패킹된 파일을 원래의 상태로 복원하여 분석할 수 있는 도구.

Ghidra: 디컴파일러로, 바이너리 코드를 고수준 언어로 변환하는 도구.

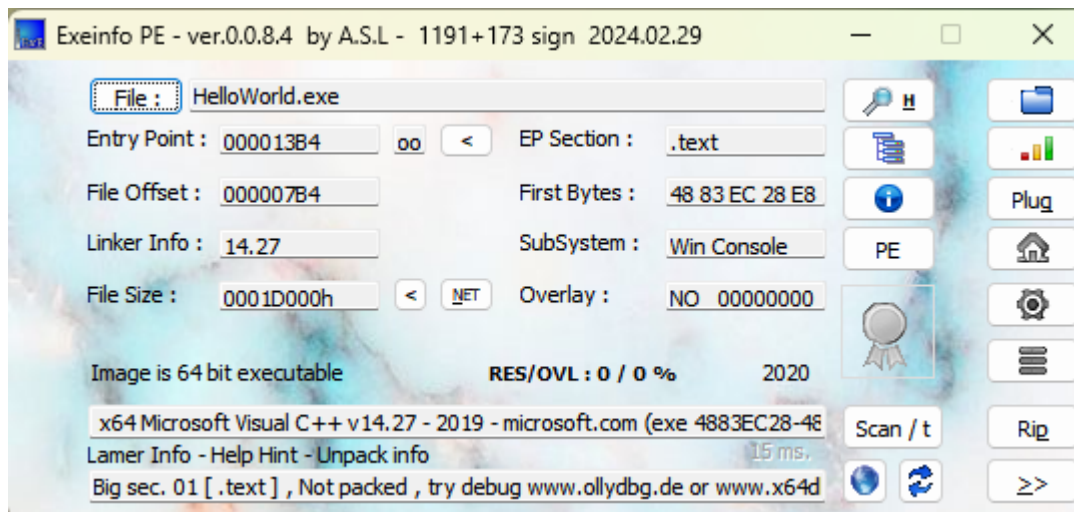
Volatility: 메모리 덤프를 분석하여 시스템 메모리에 존재하는 악성코드, 루트킷 등을 분석하는 데 사용되는 도구.

이 중에서 동적 분석(디스어셈블러 및 디버거), 그리고 정적 분석(PE 파일 분석)은 가장 핵심적인 역할을 한다. 리버싱을 수행할 때는 일반적으로 정적 분석 도구를 통해 파일의 기본적인 구조와 메타데이터를 먼저 파악하고, 이후 동적 분석 도구를 사용해 코드의 동작을 심층적으로 분석한다.

Helloworld.exe PE 파일 구조 분석 [Exeinfo PE]

드림핵 리버싱 강의의 helloworld 파일을 사용했다. → [Helloworld.exe](#)

주로 파일의 외형적인 구조와 속성을 확인하고, 파일이 어떻게 구성되어 있는지에 대한 기본적인 정보를 분석한다. 초기 분석 및 파일 식별에 사용된다.



Entry Point: 000013B4 → 프로그램이 실행될 때 처음으로 실행되는 코드의 주소로 이 프로그램의 entry point는 000013B4이다. EP Section: .text 를 통해 이 주소가 .text 섹션에 있음을 알 수 있다.

File Offset: 000007B4 → 진입 지점의 파일 오프셋 위치(해당 코드가 시작하는 위치)는 000007B4이다.

Linker Info: 14.27 → 이 정보는 파일이 visual studio 2019(버전 14.27) 링커를 사용해 링크되었음을 의미한다.

File Size: 0001D000h → 이 파일의 크기는 약 118,784 byte(약 116KB)로 변환된다.

First Bytes: 48 83 EC 28 E8 → 파일이 실행될 때 가장 먼저 실행되는 몇 바이트의 코드이다. 이 정보는 특정 인코딩을 가진 명령어로 디스어셈블링으로 기능을 파악할 수 있다.

SubSystem: Win Console → 이 정보는 해당 파일이 콘솔 응용 프로그램임을 나타낸다.

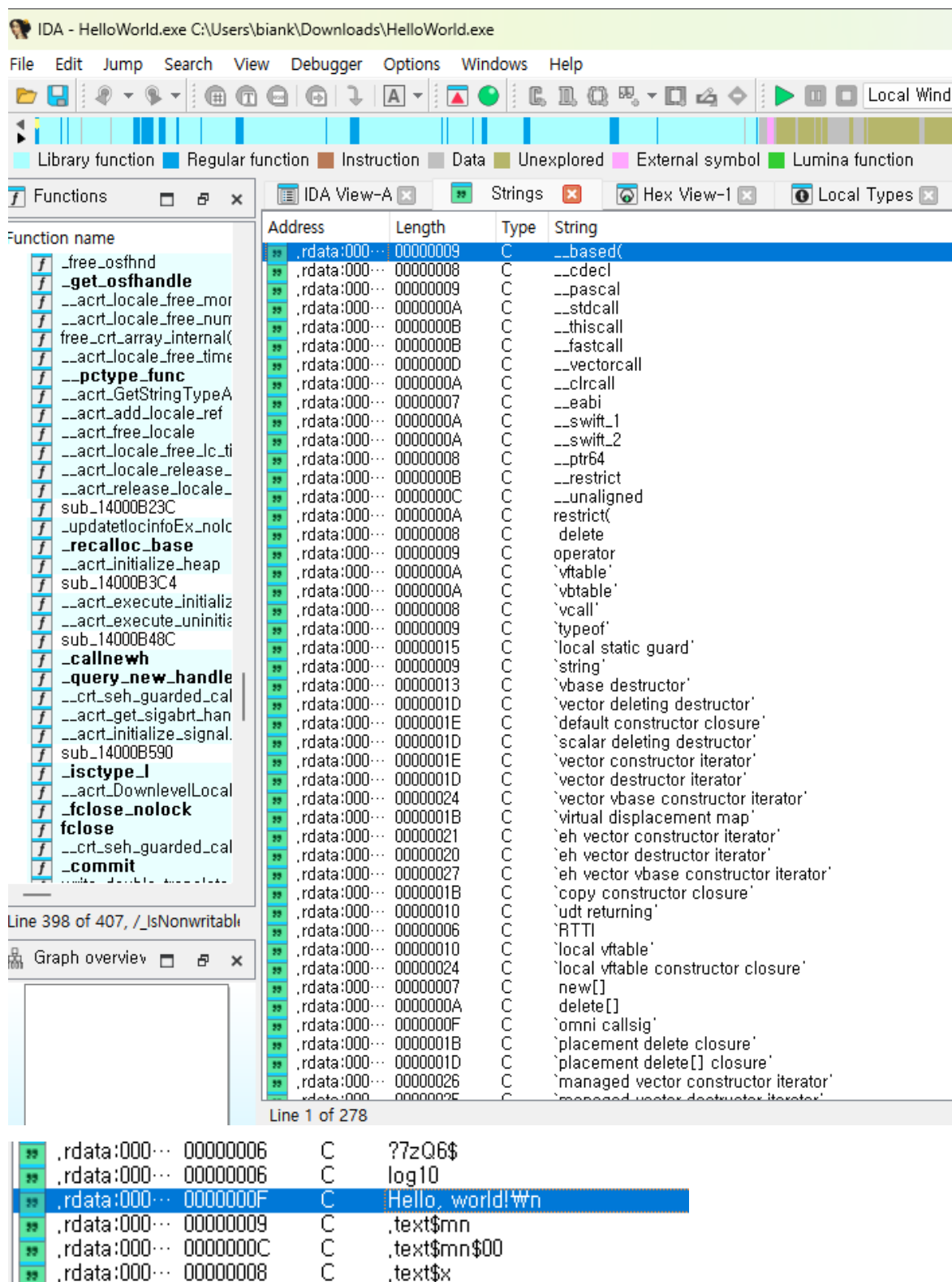
Overlay: No → 파일의 오버레이가 없다. 오버레이는 PE 파일의 일반적인 구조 외에 추가된 데이터로, 자주 사용되지 않는 리소스나 숨겨진 데이터를 포함할 수 있다.

Image is 64-bit executable → 이 파일이 64비트 실행 파일임을 나타낸다.

Lamer Info: Help Hint - Unpack info → 이 파일의 경우 Not packed로 파일이 압축되지 않음을 알 수 있다.

HelloWorld.exe 정적 분석 [IDA freeware]

문자열 검색 (Shift + F12) → 문자열이 열거된 Strings 창이 나온다.



IDA - HelloWorld.exe C:\Users\biangk\Downloads\HelloWorld.exe

File Edit Jump Search View Debugger Options Windows Help

Library function Regular function Instruction Data Unexplored External symbol Lumina function

Functions

Function name

- __free_osfhnd
- __get_osfhnd
- __acrt_locale_free_mor
- __acrt_locale_free_nun
- free_crt_array_internal
- __acrt_locale_free_time
- __pctype_func
- __acrt_GetStringTypeA
- __acrt_add_locale_ref
- __acrt_free_locale
- __acrt_locale_free_lc_ti
- __acrt_locale_release
- __acrt_release_locale
- sub_14000B23C
- __update_tlocinfoEx_nolc
- __realloc_base
- __acrt_initialize_heap
- sub_14000B3C4
- __acrt_execute_initialize
- __acrt_execute_uninitia
- sub_14000B48C
- __callnewh
- __query_new_handle
- __crt_seh_guarded_cal
- __acrt_get_sigabrt_han
- __acrt_initialize_signal
- sub_14000B590
- __isctype_l
- __acrt_DownlevelLocal
- __fclose_nolock
- fclose
- __crt_seh_guarded_cal
- __commit

Line 398 of 407, /_IsNonwritable

Graph overview

IDA View-A

Strings

Hex View-1

Local Types

Address	Length	Type	String
.rdata:000...	00000009	C	__based(
.rdata:000...	00000008	C	__cdecl
.rdata:000...	00000009	C	__pascal
.rdata:000...	0000000A	C	__stdcall
.rdata:000...	0000000B	C	__thiscall
.rdata:000...	0000000B	C	__fastcall
.rdata:000...	0000000D	C	__vectorcall
.rdata:000...	0000000A	C	__clrcall
.rdata:000...	00000007	C	__eabi
.rdata:000...	0000000A	C	__swift_1
.rdata:000...	0000000A	C	__swift_2
.rdata:000...	00000008	C	__ptr64
.rdata:000...	0000000B	C	__restrict
.rdata:000...	0000000C	C	__unaligned
.rdata:000...	0000000A	C	restrict(
.rdata:000...	00000008	C	delete
.rdata:000...	00000009	C	operator
.rdata:000...	0000000A	C	'vtable'
.rdata:000...	0000000A	C	'vtable'
.rdata:000...	00000008	C	'vcall'
.rdata:000...	00000009	C	'typeof'
.rdata:000...	00000015	C	'local static guard'
.rdata:000...	00000009	C	'string'
.rdata:000...	00000013	C	'vbase destructor'
.rdata:000...	0000001D	C	'vector deleting destructor'
.rdata:000...	0000001E	C	'default constructor closure'
.rdata:000...	0000001D	C	'scalar deleting destructor'
.rdata:000...	0000001E	C	'vector constructor iterator'
.rdata:000...	0000001D	C	'vector destructor iterator'
.rdata:000...	00000024	C	'vector vbase constructor iterator'
.rdata:000...	0000001B	C	'virtual displacement map'
.rdata:000...	00000021	C	'eh vector constructor iterator'
.rdata:000...	00000020	C	'eh vector destructor iterator'
.rdata:000...	00000027	C	'eh vector vbase constructor iterator'
.rdata:000...	0000001B	C	'copy constructor closure'
.rdata:000...	00000010	C	'udt returning'
.rdata:000...	00000006	C	'RTTI'
.rdata:000...	00000010	C	'local vtable'
.rdata:000...	00000024	C	'local vtable constructor closure'
.rdata:000...	00000007	C	new[]
.rdata:000...	0000000A	C	delete[]
.rdata:000...	0000000F	C	'omni callsig'
.rdata:000...	0000001B	C	'placement delete closure'
.rdata:000...	0000001D	C	'placement delete[] closure'
.rdata:000...	00000026	C	'managed vector constructor iterator'
.rdata:000...	00000025	C	'managed vector destructor iterator'

Line 1 of 278

.rdata:000...	00000006	C	?zQ6\$
.rdata:000...	00000006	C	log10
.rdata:000...	0000000F	C	Hello, world!\n
.rdata:000...	00000009	C	.text\$mn
.rdata:000...	0000000C	C	.text\$mn\$00
.rdata:000...	00000008	C	.text\$x

밑으로 내리다 보면 Hello, world!\n라는 문자열이 보인다. 이 문자를 더블클릭하면 다음과 같은 창이 나온다.

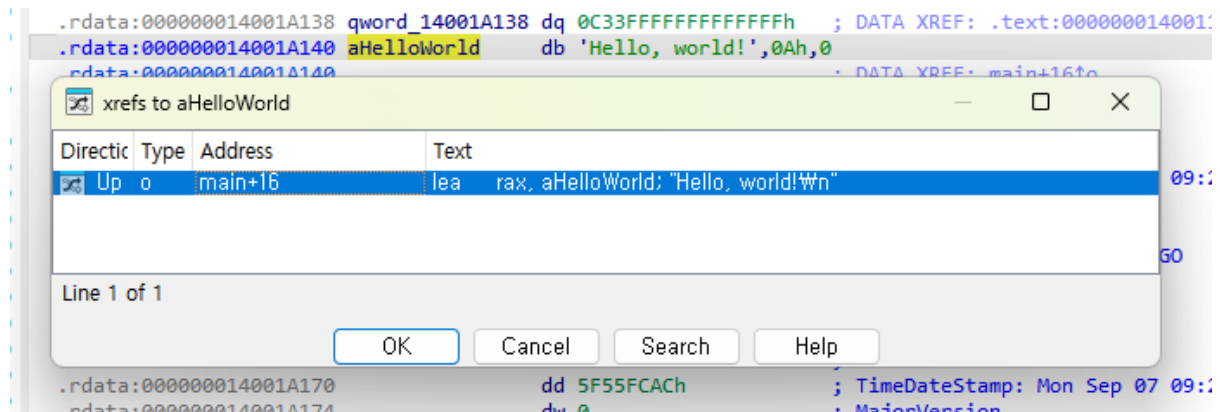
```

.rdata:000000014001A126 align 10h
.rdata:000000014001A130 qword_14001A130 dq 433FFFFFFFFFFFFh ; DATA XREF: .text:_frndfr
.rdata:000000014001A138 qword_14001A138 dq 0C33FFFFFFFFFFFFh ; DATA XREF: .text:0000000140011168fr
.rdata:000000014001A140 aHelloWorld db 'Hello, world!'\0Ah,0 ; DATA XREF: main+16fo
.rdata:000000014001A140
.rdata:000000014001A14F align 10h
.rdata:000000014001A150 ; Debug Directory entries
.rdata:000000014001A150 dd 0 ; Characteristics
.rdata:000000014001A154 dd 5F55FCACH ; TimeDateStamp: Mon Sep 07 09:26:04 202
.rdata:000000014001A158 dw 0 ; MajorVersion
.rdata:000000014001A15A dw 0 ; MinorVersion
.rdata:000000014001A15C dd 00h ; Type: IMAGE_DEBUG_TYPE_POGO
.rdata:000000014001A160 dd 2B8h ; SizeOfData
.rdata:000000014001A164 dd rva aGctl ; AddressOfRawData
.rdata:000000014001A168 dd 194C0h ; PointerToRawData
.rdata:000000014001A16C dd 0 ; Characteristics
.rdata:000000014001A170 dd 5F55FCACH ; TimeDateStamp: Mon Sep 07 09:26:04 202
.rdata:000000014001A174 dw 0 ; MajorVersion
.rdata:000000014001A176 dw 0 ; MinorVersion
.rdata:000000014001A178 dd 0Eh ; Type: IMAGE_DEBUG_TYPE_ILTCG
.rdata:000000014001A17C dd 0 ; SizeOfData

```

상호참조(Cross Reference, XRef) → x

여기서 상호참조 기능을 통해 함수를 자세하게 분석할 수 있다. 앞선 Hello, world!\n라는 문자열이 어디서 사용되는지 추적하기 위해 aHelloWorld를 클릭하고 x를 누른다.



상호참조 단축키를 누르면 xrefs창이 나타난다. 이 창에는 해당 변수를 참조하는 모든 주소가 출력된다. 항목을 더블클릭한다.

디컴파일 → F5

```

sub     rsp, 38h
mov     [rsp+38h+dwMilliseconds], 3E8h
mov     ecx, [rsp+38h+dwMilliseconds] ; dwMilliseconds
call    cs:Sleep
lea     rax, aHelloWorld ; "Hello, world!\n"
mov     cs:qword_14001DBE0, rax
mov     rcx, cs:qword_14001DBE0
call    sub_140001060
xor     eax, eax
add     rsp, 38h
retn
main endp

```

main 함수를 찾았으므로 F5키를 눌러 디컴파일 한다.

```
1 int __fastcall main(int argc, const char **argv, const char **envp)
2 {
3     Sleep(0x3E8u);
4     qword_14001DBE0 = (_int64)"Hello, world!\n";
5     sub_140001060("Hello, world!\n");
6     return 0;
7 }
```

[디컴파일 해석]

1. sleep함수를 호출하여 1초 대기한다,
2. qword_14001DBE0에 Hello, world!\n 문자열의 주소를 넣는다.
3. sub_140001060에 Hello,world!\n를 인자로 전달해서 호출한다.
4. 0을 반환한다.

→ qword_14001DBE0 변수는 값이 변경될 수 있는 전역 변수이므로 data 섹션에 존재한다. 이는 변수를 더블클릭해서도 확인할 수 있다.

```
.data:0000000014001DBE0 qword_14001DBE0 dq ? ; DATA XREF: main+1D1w
```

→ Hello, world!\n 문자열은 실행 도중 값이 변경될 일이 없는 상수이므로 rodata 섹션에 존재할 것이다.

→ sub_140001060는 printf()함수이다.

```
1 int64 sub_140001060(int64 a1, ...)
2 {
3     FILE *v1; // rax
4     va_list va; // [rsp+58h] [rbp+10h] BYREF
5
6     va_start(va, a1);
7     v1 = _acrt_iob_func(1u);
8     return (unsigned int)sub_140001010(v1, a1, 0LL, (_int64 *)va);
9 }
```

sub_140001060 함수를 디컴파일(F5) 해본 결과 va_start 함수를 통해 가변 인자를 처리하는 함수임을 알 수 있다. _acrt_iob_func 함수는 스트림을 가져올 때 사용되는 함수인데, 인자로 들어가는 1은 stdout을 의미한다. 따라서 문자열 인자를 받고 stdout 스트림을 내부적으로 사용하는 가변 함수임을 알 수 있다. 따라서 이런 정황들을 통해 printf 함수로 추정할 수 있다.

Helloworld.exe 동적 분석 [IDA freeware]

중단점(Break point, F2) & 실행(Run, F9) & 한 단계 실행(step over, F8)

1. main 함수에 중단점을 설정한다.

```
1 int __fastcall main(int argc, const char **argv, const char **envp)
2 {
3     Sleep(0x3E8u);
4     qword_14001DBE0 = (__int64)"Hello, world!\n";
5     sub_140001060("Hello, world!\n");
6     return 0;
7 }
```

2. 디버깅을 시작해서 main 함수까지 실행한다.

The screenshot shows the IDA View-RIP window with assembly code for the main function. The code is as follows:

```
.text:00000001400010F0 sub     rsp, 38h
.text:00000001400010E4 mov     [rsp+38h+dwMilliseconds], 3E8h
.text:00000001400010EC mov     ecx, [rsp+38h+dwMilliseconds] ; dwMilliseconds
.text:00000001400010F0 call    cs:Sleep
.text:00000001400010F6 lea     rax, aHelloWorld ; "Hello, world!\n"
.text:00000001400010FD mov     cs:qword_14001DBE0, rax
.text:0000000140001104 mov     rcx, cs:qword_14001DBE0
.text:000000014000110B call    sub_140001060
.text:0000000140001110 xor     eax, eax
.text:0000000140001112 add     rsp, 38h
.text:0000000140001116 retn
.text:0000000140001116 main endp
.text:0000000140001116
```

The General registers window shows the following values:

Register	Value	Comment
RAX	000000014001D2E8	data:dwo
RBX	000000000000496D30	debug019:
RCX	000000000000000001	debug019:
RDX	000000000000496D30	debug019:
RSI	000000000000000000	debug019:
RDI	00000000000049BCD0	debug019:

The Modules window shows the following modules:

Path
C:\Users\Wblank\Downloads\HelloWorld.exe
C:\WINDOWS\system32\Wapphelp.dll
C:\WINDOWS\system32\USER32.dll
C:\WINDOWS\system32\GDI32.dll

The Threads window shows the following threads:

Decimal	Hex	State	Name
13628	353C	Ready	HelloWorld.exe

The Hex View-1 window shows the following hex data:

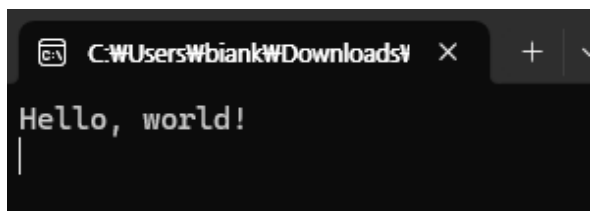
```
00000001400010B0 89 44 24 20 48 C7 44 24 28 00 00 00 88 44 24 .D5.H. $(....D$
00000001400010C0 20 48 30 4C 24 30 48 33 CC E8 62 00 00 48 83 .H.L50H3..b...H
00000001400010D0 C4 48 C3 CC CC CC CC CC CC CC CC CC CC CC CC
00000001400010E0 48 83 EC 38 C7 44 24 20 E8 03 00 00 88 4C 24 20 H....D5....L$
00000001400010F0 FF 15 0A 0F 01 00 48 8D 05 43 90 01 00 48 89 05 ....H....C...H.
0000000140001100 DC CA 01 00 48 88 00 D5 CA 01 00 E8 50 FF FF FF ....H....C...H.
0000000140001110 33 C0 48 83 C4 38 C3 CC CC CC CC CC CC CC CC 3.....
0000000140001120 CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC ff.....
0000000140001130 4A 7B 00 01 4F 01 00 F2 7C 17 4A C1 C1 1A 6A F7 H- 3. f
000004F0 00000001400010F0: main+10
```

The Stack view window shows the following stack data:

Address	Value	Comment
0000000000014FE8	000000014001344	scrt_common_0x14
0000000000014FE9	0000000000000000	
0000000000014FEA	0000000000000000	
0000000000014FEB	0000000000000000	
0000000000014FEC	0000000000000000	
0000000000014FED	0000000000000000	
0000000000014FEE	0000000000000000	
0000000000014FEF	0000000000000000	
0000000000014FF0	0000000000000000	
0000000000014FF1	0000000000000000	
0000000000014FF2	0000000000000000	
0000000000014FF3	0000000000000000	
0000000000014FF4	0000000000000000	
0000000000014FF5	0000000000000000	
0000000000014FF6	0000000000000000	
0000000000014FF7	0000000000000000	
0000000000014FF8	0000000000000000	
0000000000014FF9	0000000000000000	
0000000000014FFA	0000000000000000	
0000000000014FFB	0000000000000000	
0000000000014FFC	0000000000000000	
0000000000014FFD	0000000000000000	
0000000000014FFE	0000000000000000	
0000000000014FFF	0000000000000000	
0000000000015000	0000000000000000	
0000000000015001	0000000000000000	
0000000000015002	0000000000000000	
0000000000015003	0000000000000000	
0000000000015004	0000000000000000	
0000000000015005	0000000000000000	
0000000000015006	0000000000000000	
0000000000015007	0000000000000000	
0000000000015008	0000000000000000	
0000000000015009	0000000000000000	
000000000001500A	0000000000000000	
000000000001500B	0000000000000000	
000000000001500C	0000000000000000	
000000000001500D	0000000000000000	
000000000001500E	0000000000000000	
000000000001500F	0000000000000000	
0000000000015010	0000000000000000	
0000000000015011	0000000000000000	
0000000000015012	0000000000000000	
0000000000015013	0000000000000000	
0000000000015014	0000000000000000	
0000000000015015	0000000000000000	
0000000000015016	0000000000000000	
0000000000015017	0000000000000000	
0000000000015018	0000000000000000	
0000000000015019	0000000000000000	
000000000001501A	0000000000000000	
000000000001501B	0000000000000000	
000000000001501C	0000000000000000	
000000000001501D	0000000000000000	
000000000001501E	0000000000000000	
000000000001501F	0000000000000000	
0000000000015020	0000000000000000	
0000000000015021	0000000000000000	
0000000000015022	0000000000000000	
0000000000015023	0000000000000000	
0000000000015024	0000000000000000	
0000000000015025	0000000000000000	
0000000000015026	0000000000000000	
0000000000015027	0000000000000000	
0000000000015028	0000000000000000	
0000000000015029	0000000000000000	
000000000001502A	0000000000000000	
000000000001502B	0000000000000000	
000000000001502C	0000000000000000	
000000000001502D	0000000000000000	
000000000001502E	0000000000000000	
000000000001502F	0000000000000000	
0000000000015030	0000000000000000	
0000000000015031	0000000000000000	
0000000000015032	0000000000000000	
0000000000015033	0000000000000000	
0000000000015034	0000000000000000	
0000000000015035	0000000000000000	
0000000000015036	0000000000000000	
0000000000015037	0000000000000000	
0000000000015038	0000000000000000	
0000000000015039	0000000000000000	
000000000001503A	0000000000000000	
000000000001503B	0000000000000000	
000000000001503C	0000000000000000	
000000000001503D	0000000000000000	
000000000001503E	0000000000000000	
000000000001503F	0000000000000000	
0000000000015040	0000000000000000	
0000000000015041	0000000000000000	
0000000000015042	0000000000000000	
0000000000015043	0000000000000000	
0000000000015044	0000000000000000	
0000000000015045	0000000000000000	
0000000000015046	0000000000000000	
0000000000015047	0000000000000000	
0000000000015048	0000000000000000	
0000000000015049	0000000000000000	
000000000001504A	0000000000000000	
000000000001504B	0000000000000000	
000000000001504C	0000000000000000	
000000000001504D	0000000000000000	
000000000001504E	0000000000000000	
000000000001504F	0000000000000000	
0000000000015050	0000000000000000	
0000000000015051	0000000000000000	
0000000000015052	0000000000000000	
0000000000015053	0000000000000000	
0000000000015054	0000000000000000	
0000000000015055	0000000000000000	
0000000000015056	0000000000000000	
0000000000015057	0000000000000000	
0000000000015058	0000000000000000	
0000000000015059	0000000000000000	
000000000001505A	0000000000000000	
000000000001505B	0000000000000000	
000000000001505C	0000000000000000	
000000000001505D	0000000000000000	
000000000001505E	0000000000000000	
000000000001505F	0000000000000000	
0000000000015060	0000000000000000	
0000000000015061	0000000000000000	
0000000000015062	0000000000000000	
0000000000015063	0000000000000000	
0000000000015064	0000000000000000	
0000000000015065	0000000000000000	
0000000000015066	0000000000000000	
0000000000015067	0000000000000000	
0000000000015068	0000000000000000	
0000000000015069	0000000000000000	
000000000001506A	0000000000000000	
000000000001506B	0000000000000000	
000000000001506C	0000000000000000	
000000000001506D	0000000000000000	
000000000001506E	0000000000000000	
000000000001506F	0000000000000000	
0000000000015070	0000000000000000	
0000000000015071	0000000000000000	
0000000000015072	0000000000000000	
0000000000015073	0000000000000000	
0000000000015074	0000000000000000	
0000000000015075	0000000000000000	
0000000000015076	0000000000000000	
0000000000015077	0000000000000000	
0000000000015078	0000000000000000	
0000000000015079	0000000000000000	
000000000001507A	0000000000000000	
000000000001507B	0000000000000000	
000000000001507C	0000000000000000	
000000000001507D	0000000000000000	
000000000001507E	0000000000000000	
000000000001507F	0000000000000000	
0000000000015080	0000000000000000	
0000000000015081	0000000000000000	
0000000000015082	0000000000000000	
0000000000015083	0000000000000000	
0000000000015084	0000000000000000	
0000000000015085	0000000000000000	
0000000000015086	0000000000000000	
0000000000015087	0000000000000000	
0000000000015088	0000000000000000	
0000000000015089	0000000000000000	
000000000001508A	0000000000000000	
000000000001508B	0000000000000000	
000000000001508C	0000000000000000	
000000000001508D	0000000000000000	
000000000001508E	0000000000000000	
000000000001508F	0000000000000000	
0000000000015090	0000000000000000	
0000000000015091	0000000000000000	
0000000000015092	0000000000000000	
0000000000015093	0000000000000000	
0000000000015094	0000000000000000	
0000000000015095	0000000000000000	
0000000000015096	0000000000000000	
0000000000015097	0000000000000000	
0000000000015098	0000000000000000	
0000000000015099	0000000000000000	
000000000001509A	0000000000000000	
000000000001509B	0000000000000000	
000000000001509C	0000000000000000	
000000000001509D	0000000000000000	
000000000001509E	0000000000000000	
000000000001509F	0000000000000000	
00000000000150A0	0000000000000000	
00000000000150A1	0000000000000000	
00000000000150A2	0000000000000000	
00000000000150A3	0000000000000000	
00000000000150A4	0000000000000000	
00000000000150A5	0000000000000000	
00000000000150A6	0000000000000000	
00000000000150A7	0000000000000000	
00000000000150A8	0000000000000000	
00000000000150A9	0000000000000000	
00000000000150AA	0000000000000000	
00000000000150AB	0000000000000000	
00000000000150AC	0000000000000000	
00000000000150AD	0000000000000000	
00000000000150AE	0000000000000000	
00000000000150AF	0000000000000000	
00000000000150B0	0000000000000000	
00000000000150B1	0000000000000000	
00000000000150B2	0000000000000000	
00000000000150B3	0000000000000000	
00000000000150B4	0000000000000000	
00000000000150B5	0000000000000000	
00000000000150B6	0000000000000000	
00000000000150B7	0000000000000000	
00000000000150B8	0000000000000000	
00000000000150B9	0000000000000000	

sub rsp, 38	main 함수가 사용할 스택 영역 확보
mov [rsp+38h+dwMilliseconds], 3E8h	rsp+0x20에 4바이트 값인 0x000003e8 저장
mov ecx, [rsp+38h+dwMilliseconds] ; dwMilliseconds	rsp+0x20에 저장된 값을 ecx에 옮김. 함수의 첫번째 인자를 설정.
call cs:Sleep	Sleep 함수 호출. ecx가 0x3e8이므로, Sleep(100)이 실행되어 1초간 실행이 멈춤.
lea rax, aHelloWorld ; "Hello, world!\n"	"Hello, world!\n" 문자열의 주소를 rax에 옮김.
mov cs:qword_14001DBE0, rax	rax의 값을 data 세그먼트의 주소인 0x14001a140에 저장한다.
mov rcx, cs:qword_14001DBE0	0x14001DBE0에 저장된 값을 rcx에 옮김. 이는 다음 호출할 함수의 첫번째 인자로 사용됨.
call sub_140001060	0x140001060 함수를 호출. (정적 분석을 통해 printf() 함수라고 추측한 것)

프로그램을 확인하면, Hello, world!가 출력되어 있다.



```

C:\Users\wbiank\Downloads>
Hello, world!
|

```

xor eax, eax	eax 레지스터 초기화.
add rsp, 38h	시작할 때 확장한 스택 영역을 다시 축소시킨다.
retn	ret으로 원래 실행 흐름으로 돌아간다.
main endp	main 함수의 끝을 나타내는 지시어

함수 내부로 진입하기(step into, F7)

1. 디버깅을 중단하고(Ctrl+F2), printf 함수를 호출하는 0x14000110b에 중단점을 설정한다.

```

.text:00000001400010E0 ; DATA XREF:
.text:00000001400010E0
.text:00000001400010E0 dwMilliseconds = dword ptr -18h
.text:00000001400010E0
.text:00000001400010E0 sub     rsp, 38h
.text:00000001400010E4 mov     [rsp+38h+dwMilliseconds], 3Eh
.text:00000001400010EC mov     ecx, [rsp+38h+dwMilliseconds]
.text:00000001400010F0 call    cs:Sleep
.text:00000001400010F6 lea     rax, aHelloWorld ; "Hello, w
.text:00000001400010FD mov     cs:qword_14001DBE0, rax
.text:0000000140001104 mov     rcx, cs:qword_14001DBE0
.text:0000000140001108 call    sub_140001060
.text:0000000140001110 xor     eax, eax
.text:0000000140001112 add     rsp, 38h
.text:0000000140001116 retn
.text:0000000140001116 main      endp
.text:0000000140001116 ; -----
.text:0000000140001117 byte_140001117 db 0Fh dup(0Cch) ; DATA XREF:

```

2. 디버깅을 다시 시작하고 continue(F9)를 클릭해서 printf 함수에 도달한다.

```

.text:00000001400010E0
.text:00000001400010E0
.text:00000001400010E0 ; int __fastcall main(int argc, const char **argv, const char **envp)
.text:00000001400010E0 main proc near
.text:00000001400010E0
.text:00000001400010E0 dwMilliseconds= dword ptr -18h
.text:00000001400010E0
.text:00000001400010E0 sub     rsp, 38h
.text:00000001400010E4 mov     [rsp+38h+dwMilliseconds], 3Eh
.text:00000001400010EC mov     ecx, [rsp+38h+dwMilliseconds] ; dwMilliseconds
.text:00000001400010F0 call    cs:Sleep
.text:00000001400010F6 lea     rax, aHelloWorld ; "Hello, world!\n"
.text:00000001400010FD mov     cs:qword_14001DBE0, rax
.text:0000000140001104 mov     rcx, cs:qword_14001DBE0
.text:0000000140001108 call    sub_140001060
.text:0000000140001110 xor     eax, eax
.text:0000000140001112 add     rsp, 38h
.text:0000000140001116 retn
.text:0000000140001116 main      endp
.text:0000000140001116

```

printf() 함수 호출 직전 중단된 모습

3. F7 단축키를 통해 함수 내부로 들어간다. 함수 내부로 rip가 이동한 것을 확인할 수 있다.

```

.text:0000000140001060
.text:0000000140001060 ; __unwind { // __GSHandlerCheck
.text:0000000140001060 mov     [rsp+arg_0], rcx
.text:0000000140001065 mov     [rsp+arg_8], rdx
.text:000000014000106A mov     [rsp+arg_10], r8
.text:000000014000106F mov     [rsp+arg_18], r9
.text:0000000140001074 sub     rsp, 48h
.text:0000000140001078 mov     rax, cs:__security_cookie
.text:000000014000107F xor     rax, rsp
.text:0000000140001082 mov     [rsp+48h+var_18], rax
.text:0000000140001087 lea     rax, [rsp+48h+arg_8]
.text:000000014000108C mov     [rsp+48h+var_20], rax
.text:0000000140001091 mov     ecx, 1 ; 1x
.text:0000000140001096 call    __acrt_iob_func
.text:000000014000109B mov     r9, [rsp+48h+var_20]
.text:00000001400010A0 xor     r8d, r8d
.text:00000001400010A3 mov     rdx, [rsp+48h+arg_0]
.text:00000001400010A8 mov     rcx, rax
.text:00000001400010AB call    sub_140001010
.text:00000001400010B0 mov     [rsp+48h+var_28], eax
.text:00000001400010B4 mov     [rsp+48h+var_20], 0
.text:00000001400010BD mov     eax, [rsp+48h+var_28]
.text:00000001400010C1 mov     rcx, [rsp+48h+var_18]
.text:00000001400010C6 xor     rcx, rsp ; StackCookie
.text:00000001400010C9 call    __security_check_cookie

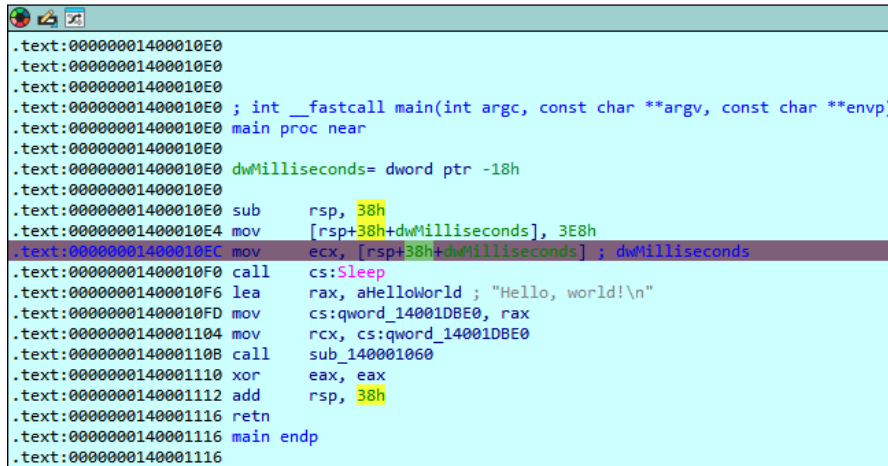
```

printf() 함수 내부에 진입한 상태

Appendix, 실행 중인 프로세스 조작하기

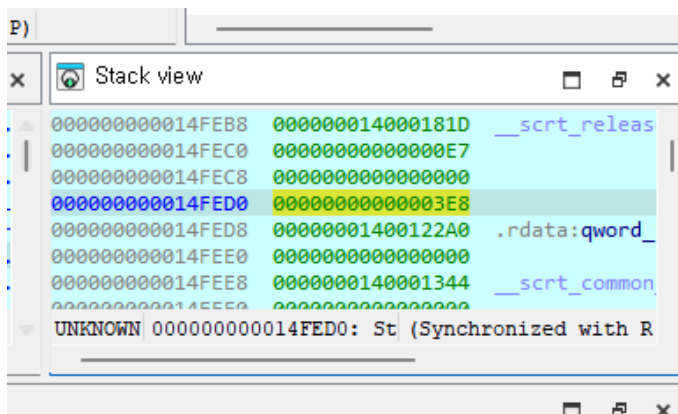
IDA를 이용하면 실행 중인 프로세스의 메모리를 조작할 수 있다. 기존 코드에서는 Sleep(delay=100)을 호출해서 1초 동안 프로세스를 정지시켰는데 delay 값을 1000000으로 조작해서 1000초 동안 프로세스를 정지시킨다.

1. delay를 Sleep 함수의 인자로 전달하는 부분에 중단점을 설정하고, 프로세스를 재시작한다.



```
.text:000000001400010E0
.text:000000001400010E0
.text:000000001400010E0
.text:000000001400010E0 ; int __fastcall main(int argc, const char **argv, const char **envp)
.text:000000001400010E0 main proc near
.text:000000001400010E0 dwMilliseconds= dword ptr -18h
.text:000000001400010E0
.text:000000001400010E0 sub     rsp, 38h
.text:000000001400010E4 mov     [rsp+38h+dwMilliseconds], 3E8h
.text:000000001400010EC mov     ecx, [rsp+38h+dwMilliseconds]; dwMilliseconds
.text:000000001400010F0 call    cs:Sleep
.text:000000001400010F6 lea     rax, aHelloWorld ; "Hello, world!\n"
.text:000000001400010FD mov     cs:qword_14001DBE0, rax
.text:00000000140001104 mov     rcx, cs:qword_14001DBE0
.text:00000000140001108 call    sub_140001060
.text:00000000140001110 xor     eax, eax
.text:00000000140001112 add     rsp, 38h
.text:00000000140001116 retn
.text:00000000140001116 main endp
.text:00000000140001116
```

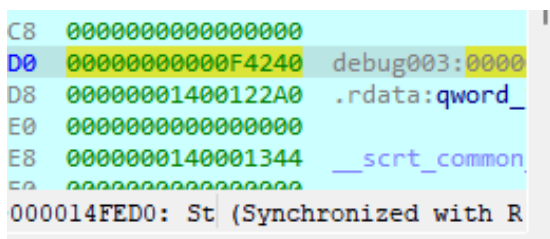
2. 스택을 보면 rsp+0x20에 delay 값인 0x3e8이 저장되어 있다.



```
Stack view
000000000014FEB8 0000000014000181D __scrt_releas
000000000014FEC0 00000000000000E7
000000000014FEC8 0000000000000000
000000000014FED0 00000000000003E8
000000000014FED8 000000001400122A0 .rdata:qword_
000000000014FEE0 0000000000000000
000000000014FEE8 00000000140001344 __scrt_common
000000000014FEE0 0000000000000000
UNKNOWN 00000000000014FED0: St (Synchronized with R
```

스택에 저장된 delay 변수 값

3. 해당 값을 클릭하고, F2를 누른 뒤 0xf4240(=100000)을 입력한다. 그리고 다시 F2를 눌러서 값을 저장한다.



```
C8 0000000000000000
D0 000000000000F4240 debug003:0000
D8 000000001400122A0 .rdata:qword_
E0 0000000000000000
E8 00000000140001344 __scrt_common
EA 0000000000000000
000014FED0: St (Synchronized with R
```

4. F9를 눌러서 Sleep 함수를 호출한다. 아까와 다르게 한참을 기다려도 프로세스가 재개되지 않는다.

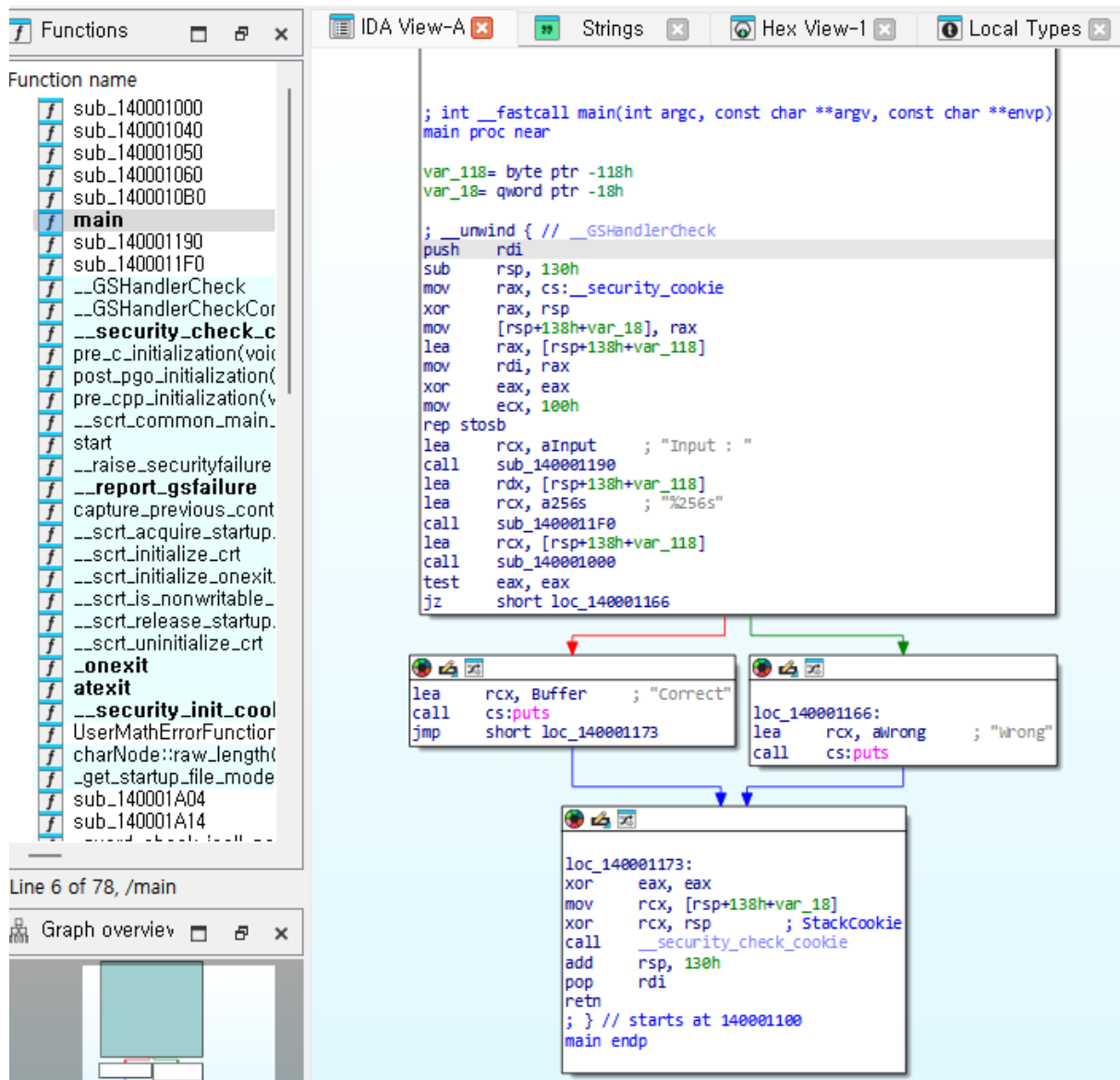
The screenshot shows a debugger window with the following components:

- Assembly View:** Displays assembly code with addresses and instructions. The instruction at address 000000001400010EC, `mov ecx, [rsp+38h+dwMilliseconds] ; dwMilliseconds`, is highlighted in red. Other instructions include `sub rsp, 38h`, `mov [rsp+38h+dwMilliseconds], 3E8h`, `call cs:Sleep`, `lea rax, aHelloWorld ; "Hello, world!\n"`, `mov cs:qword_14001DBE0, rax`, `mov rcx, cs:qword_14001DBE0`, `call sub_140001060`, `xor eax, eax`, `add rsp, 38h`, `retn`, and `main endp`.
- Threads View:** Shows a single thread with PID 9688, PPID 25D8, and state Running.
- Stack View:** Displays the stack frame for `main+10`. The stack contains various values, including `000000001400010F0: main+10` and `000000001400010F0: main+10`.

Reversing Basic Challenge #0

문제 파일 다운로드 → [dreamhack_basic_challenge_#0](#)

이 문제는 사용자에게 문자열 입력을 받아 정해진 방법으로 입력값을 검증하여 correct 또는 wrong을 출력하는 프로그램이 주어집니다. 해당 바이너리를 분석하여 correct를 출력하는 입력값을 찾으세요!



일단 Functions(함수)에서 main 함수를 찾았다. main 함수에서 사용자 입력과 미리 설정된 값을 비교하는 조건문을 통해 correct를 출력한다고 추론했다. main 함수를 자세히 보기 위해 디컴파일(F5)를 했다.

```

1 int __fastcall main(int argc, const char **argv, const char **envp)
2 {
3     char v4[256]; // [rsp+20h] [rbp-118h] BYREF
4
5     memset(v4, 0, sizeof(v4));
6     sub_140001190("Input : ", argv, envp);
7     sub_1400011F0("%256s", v4);
8     if ( (unsigned int)sub_140001000(v4) )
9         puts("Correct");
10    else
11        puts("Wrong");
12    return 0;
13 }

```

어떤 입력이 들어왔을 때 correct를 출력하는지 확인하기 위해 sub_140001000(v4)을 더블클릭했다.

```

1 BOOL __fastcall sub_7FF66E7C1000(const char *a1)
2 {
3     return strcmp(a1, "Compar3_the_str1ng") == 0;
4 }

```

이 프로그램은 사용자가 입력한 a1과 Compar3_the_string 문자열을 비교해서 두 문자열이 완전히 동일할 때 correct를 출력함을 알 수 있었다.

The screenshot shows the IDA View-RIP window with assembly code. The instruction at address 00007FF66E7C1164 is highlighted, which is a jump instruction: `jmp short loc_7FF66E7C1173`. To the right, the General registers window shows the values of RAX, RBX, RCX, RDX, RSI, and RDI. At the bottom, a debugger window shows the input string "Compar3_the_str1ng" and the output "Correct".

디버깅을 통해 Compar3_the_str1ng을 입력하고 Correct가 나오는지 확인했다.


```

uu$:-$.:$.:$.:$.:$.uu
uu$$$$$$$$$$$$$$$$$$$$uu
u$$$$$$$$$$$$$$$$$$$$u
u$$$$$$$$$$$$$$$$$$$$u
u$$$$$$$$$$$$$$$$$$$$u
u$$$$$$$$$$$$$$$$$$$$u
u$$$$$$$*   *$$$$*   *$$$$$u
*$$$$*       u$u       $$$*
$$$u        u$u        u$$$
$$$u        u$$$$u      u$$$
*$$$$uu$$$   $$$uu$$$*
*$$$$$$$*   *$$$$$$$*
u$$$$$$$u$$$$$$$u
u$=$*$=$*$=$*$=$u
uuu          $u$ $ $ $ $u$          uuu
u$$$          $$$u$u$u$$$          u$$$
$$$$$uu      *$$$$$$$$$*          uu$$$$$
u$$$$$$$$$$$$uu      *****      uuuu$$$$$$$$$
$$$$$***$$$$$$$$$$uuu      uu$$$$$$$$$***$$$*
***      **$$$$$$$$$$$$uu      **$***
uuuu      **$$$$$$$$$$$$uuu
u$$$$uuu$$$$$$$$$uu      **$$$$$$$$$$$$uuu$$$
$$$$$$$$$***      **$$$$$$$$$$$$$*
*$$$$$*      **$$$$$*
$$$*      PRESS ANY KEY!      $$$*

```

Petya 랜섬웨어는 2016년에 처음 등장한 악성 프로그램으로, 컴퓨터의 부팅 섹터를 암호화하여 시스템 접근을 차단하고 금전을 요구하는 악성 소프트웨어이다. 이 악성 소프트웨어는 Microsoft window 기반 시스템을 대상으로 하며 마스터 부트 레코드를 감염시켜, 하드 드라이브의 파일 시스템 테이블을 암호화하고 윈도우의 부팅을 차단하는 페이로드를 실행시킨다. 이 랜섬웨어는 초기에는 주로 이메일 첨부 파일을 통한 피싱 공격으로 유포되었으며, 이후 여러 가지 다른 공격 기법들이 결합되어 더욱 복잡한 형태로 발전했다. 새로운 변종은 이터널블루(NSA에 의해 개발된 것으로 간주되는 취약점 공격 도구) 익스플로잇을 통해 전파되며 NSA(미국 국가 안보국)에 의해 개발된 것으로 간주된다. 카스퍼스키 랩은 이 새로운 버전을 NotPetya로 부르며 변경사항을 실제로 되돌리지 못하도록 수정되었다.