



# WinDbg 튜토리얼

개요

바이너리 불러오기

아키텍처 지정 (Apple Silicon)

인터페이스

인터페이스 조정

디스어셈블리 뷰

레지스터 뷰

커맨드 뷰

일시정지와 종료하기

기초적인 커맨드

중단점 조작

메모리 조작

레지스터 조작

커맨드 정리

## 개요

WinDbg 튜토리얼 강의에서는 드림핵 워게임의 rev-basic-0 문제를 동적 분석하면서 인터페이스와 기본적인 커맨드의 사용을 연습해 보겠습니다.

<https://dreamhack.io/wargame/challenges/14/>

**문제 정보**

**Reversing 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!

획득한 입력값은 `DH{}` 포맷에 넣어서 인증해주세요.

예시) 입력 값이 `Apple_Banana` 일 경우 flag는 `DH{Apple_Banana}`

**Reference**

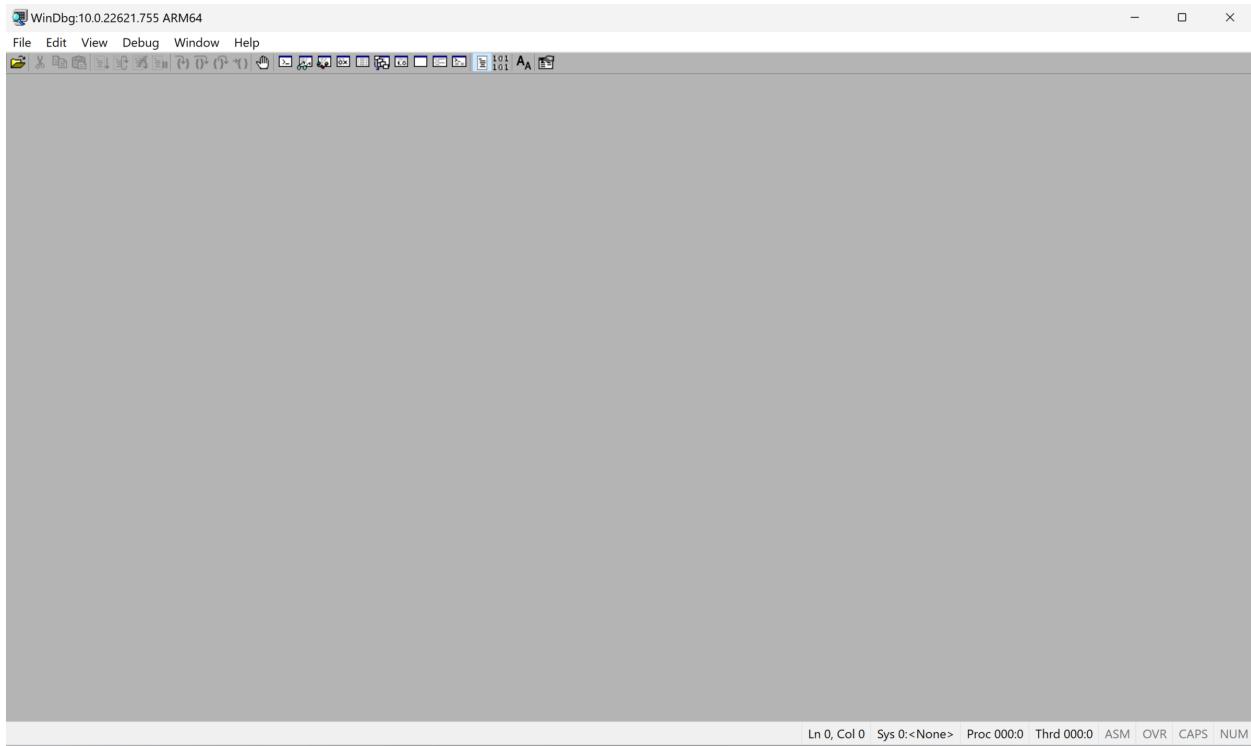
[Reverse Engineering Fundamental Roadmap](#)

**문제 파일**

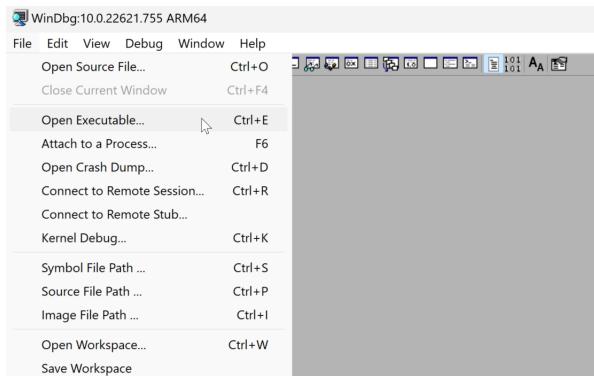
문제 파일 다운로드

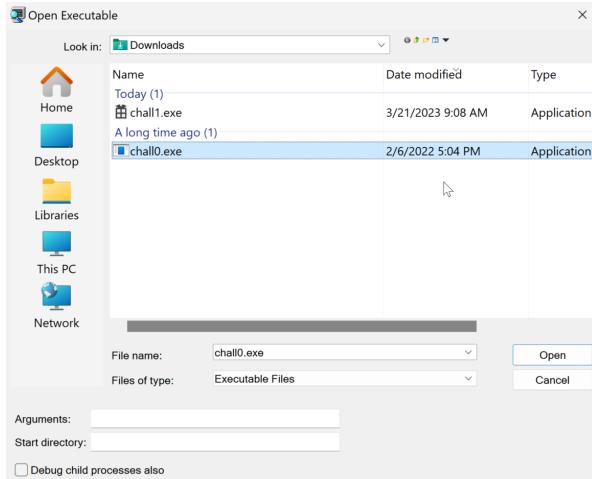
# 바이너리 불러오기

WinDbg (X64)를 실행하면 아래와 같은 초기 화면이 나타납니다. Apple Silicon 환경에서 가상 머신을 사용 중인 경우에는 WinDbg (ARM64)를 실행합니다.

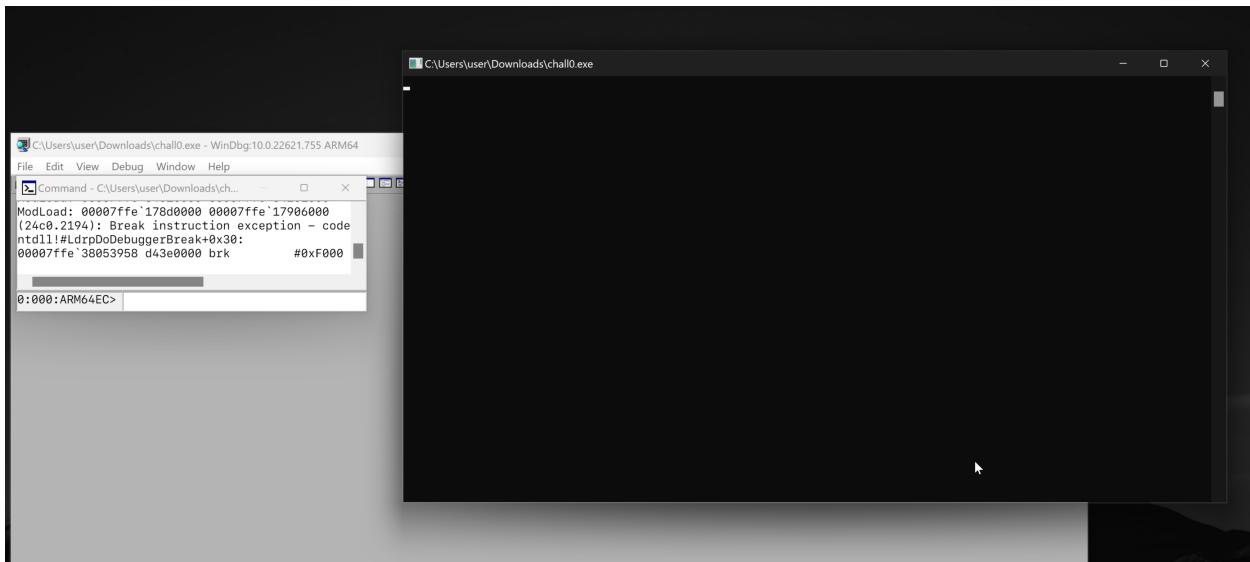


상단 메뉴의 File-Open Executable을 클릭한 후 바이너리 파일을 선택합니다. 분석할 파일인 `chal0.exe`를 선택하겠습니다.





바이너리를 선택하면 터미널 창이 새로 열리면서 일시정지된 상태로 실행되고, WinDbg 화면은 다음과 같은 상태가 됩니다. 이 상태에서 분석을 편리하게 하기 위해 인터페이스를 조금 조정하겠습니다.



## 아키텍처 지정 (Apple Silicon)

- Apple Silicon 환경에서 가상 머신을 사용 중인 경우에는 바이너리를 불러온 후 아래와 같이 이 커맨드 뷰의 입력 칸에 `.effmach amd64` 를 입력하고 Enter 키를 입력합니다.

C:\Users\user\Downloads\chall0.exe - WinDbg:10.0.22621.755 ARM64

File Edit View Debug Window Help

Command - C:\Users\user\Downloads\ch...

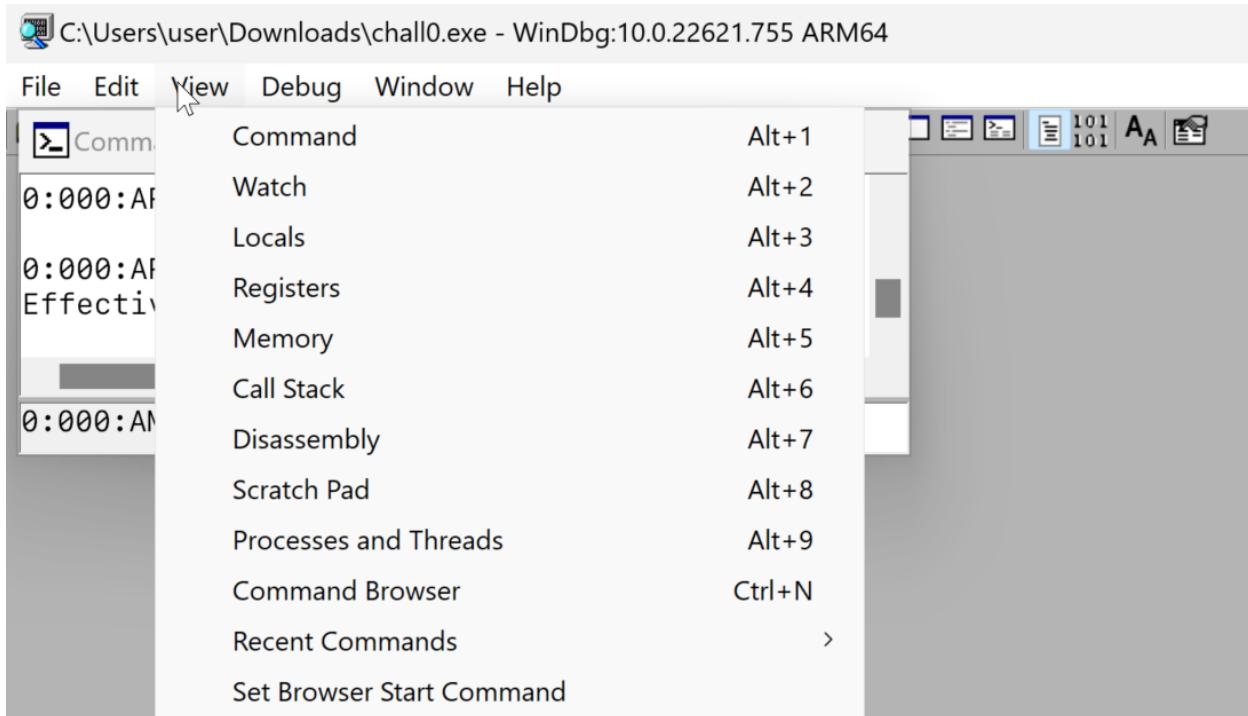
```
ntdll!#LdrpDoDebuggerBreak+0x30:  
00007ffe`38053958 d43e0000 brk      #0xF000  
0:000:ARM64EC> .effmac amd64  
^ Syntax error in '.'.
```

0:000:ARM64EC> .effmach amd64

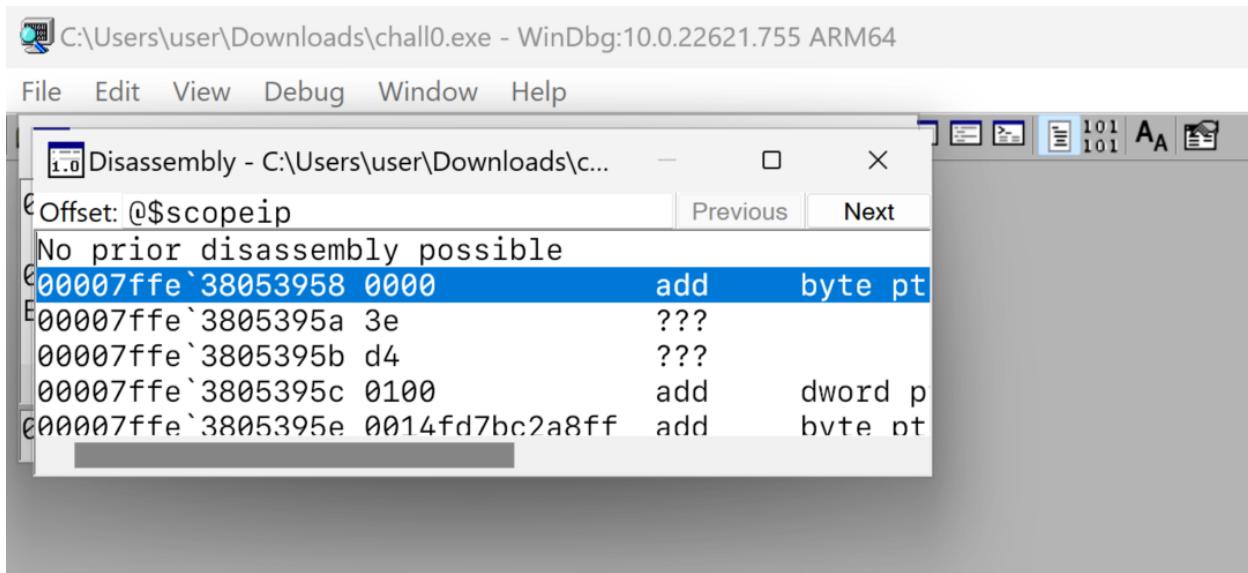
## 인터페이스

### 인터페이스 조정

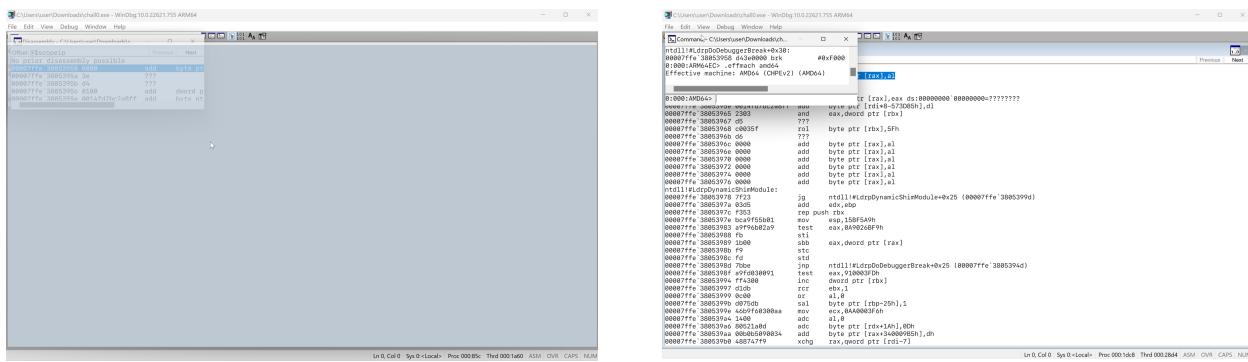
WinDbg에서 자주 사용하는 인터페이스는 디스어셈블리 뷰, 레지스터 뷰, 커맨드 뷰 등이 있습니다. 각각의 뷰는 상단 메뉴의 View 탭에서 접근할 수 있습니다.



상단 메뉴의 View-Disassembly를 클릭하면 아래와 같이 디스어셈블리 뷰가 창의 형태로 나타납니다.



이 상태에서 Disassembly라고 적힌 창의 상단 파란색 영역을 드래그하면 화면에서 뷰를 배치할 영역을 직접 선택할 수 있습니다. 디스어셈블리 뷰를 아래와 같이 화면 전체에 배치하도록 하겠습니다.



레지스터 뷰를 같은 방식으로 배치하여 보겠습니다. View 탭에서 Registers를 클릭하여 레지스터 뷰가 나타나도록 하고, 디스어셈블리 뷰를 배치한 방법과 동일하게 드래그하여 화면의 우측에 배치합니다.

C:\Users\user\Downloads\chall0.exe - WinDbg:10.0.22621.755 ARM64

File Edit View Debug Window Help

Command - C:\Users\user\Downloads\ch... Previous [1.0] Next

Registers

Customize...

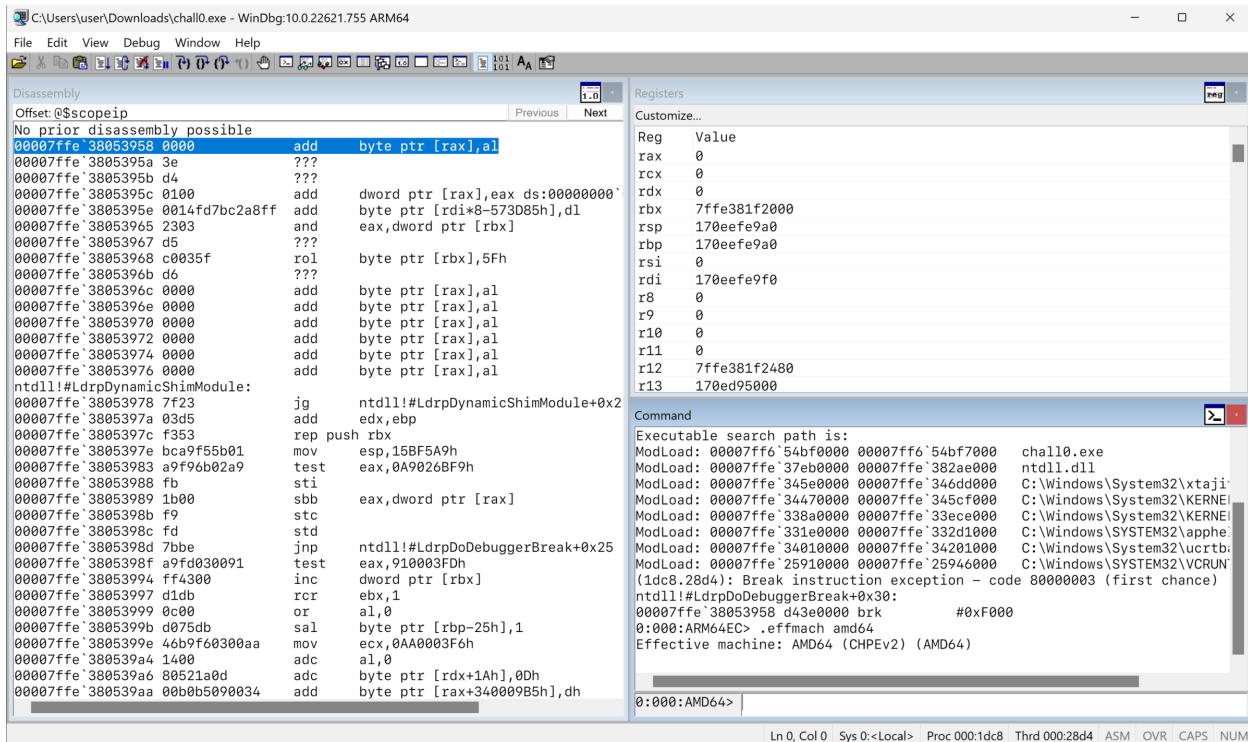
Reg	Value
rax	0
rcx	0
rdx	0
rbx	7ffe381f2000
rsp	170eefef9a0
rbp	170eefef9a0
rsi	0
rdi	170eefef9f0
r8	0
r9	0
r10	0
r11	0
r12	7ffe381f2480
r13	170ed95000
r14	7ffe344751a0
r15	7ffe0330
rip	7ffe3805395c
efl	3241
cs	33
ds	2b
es	2b
fs	53
gs	2b
ss	2b
dr0	0
dr1	0
dr2	0
dr3	0
dr6	0
dr7	0
fpcw	27f

ntdll!#LdrpDoDebuggerBreak+0x30:  
00007ffe 38053958 d43e0000 brk #0xF000  
0:000:ARM64EC> .effmach amd64  
Effective machine: AMD64 (CHPEv2) (AMD64)

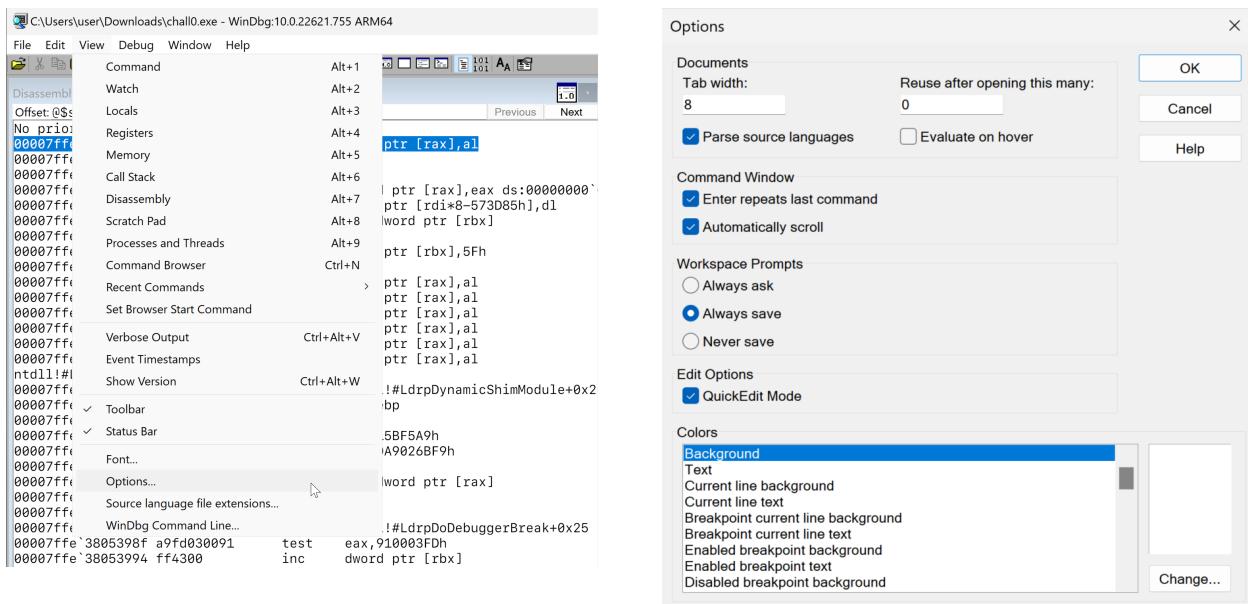
[rax],al

0:000:AMD64> tr [rax],eax ds:00000000`  
00007ffe 38053965 00141d7bd28011 add byte ptr [rdi\*8-573D85h],dl  
00007ffe 38053965 2303 and eax,dword ptr [rbx]  
00007ffe 38053967 d5 ???  
00007ffe 38053968 c0335f rol byte ptr [rbx],5Fh  
00007ffe 3805396b d6 ???  
00007ffe 3805396c 0000 add byte ptr [rax],al  
00007ffe 3805396e 0000 add byte ptr [rax],al  
00007ffe 38053970 0000 add byte ptr [rax],al  
00007ffe 38053972 0000 add byte ptr [rax],al  
00007ffe 38053974 0000 add byte ptr [rax],al  
00007ffe 38053976 0000 add byte ptr [rax],al  
ntdll!#LdrpDynamicShimModule:  
00007ffe 38053978 f723 jg ntdll!#LdrpDynamicShimModule+0x2  
00007ffe 3805397c 03d5 add edx,ebp  
00007ffe 3805397c f353 rep push rbx  
00007ffe 3805397e bca9f5b01 mov esp,15BF5A9h  
00007ffe 38053983 a9f96b02a9 test eax,0A9026BF9h  
00007ffe 38053988 fb sti  
00007ffe 38053989 1b00 sbb eax,dword ptr [rax]  
00007ffe 3805398b f9 stc  
00007ffe 3805398c fd std  
00007ffe 3805398d 7bbe jnp ntdll!#LdrpDoDebuggerBreak+0x25  
00007ffe 3805398f a9fd030091 test eax,910003FDh  
00007ffe 38053994 ff4300 inc dword ptr [rbx]  
00007ffe 38053997 d1db rcr ebx,1  
00007ffe 38053999 0c00 or al,0  
00007ffe 3805399b 0d75d sal byte ptr [rbp-25h],1  
00007ffe 3805399e 46b9f60300aa mov ecx,0AA0003F6h  
00007ffe 380539a4 1400 adc al,0  
00007ffe 380539a6 80521a0d adc byte ptr [rdx+1Ah],0Dh  
00007ffe 380539aa 00b0b5090034 add byte ptr [rax+340009B5h],dh

마지막으로 화면 좌측 상단에 남아있는 커맨드 뷰 창을 동일하게 드래그하여 우측 하단에 배치합니다.



추가적으로 인터페이스를 조정한 내용을 저장할 수 있습니다. 상단 메뉴의 View-Options를 클릭하면 아래와 같은 화면이 나타납니다. ‘Workspace Prompts’ 항목에서 ‘Always save’를 선택한 후 OK 버튼을 클릭합니다. 이제 같은 파일을 실행하는 경우에 한해 인터페이스가 저장됩니다.



## 디스어셈블리 뷰

디스어셈블리 뷰는 실행 중인 코드의 어셈블리를 출력하는 뷰입니다. 기본적으로 현재 실행 중인 인스트럭션의 어셈블리를 출력하는데, Offset 입력 칸에 함수의 이름 또는 주소를 입력하여 해당 위치의 어셈블리를 확인할 수도 있습니다. 다시 돌아가려면 Offset 입력 칸에 `@$scopeip`를 입력하고 Enter 키를 누릅니다.

The screenshot shows two side-by-side windows of the WinDbg Disassembly view. Both windows have a title bar 'Disassembly' and a status bar at the bottom right indicating '1.0'.  
The left window has an offset of '\$scopeip' and displays assembly code starting with 'No prior disassembly possible'. It includes several ADD instructions with immediate values like 0x00000000, 0x3e, 0xd4, etc., and some jumps to labels like 'ntdll!#LdrpDynamicShimModule'.  
The right window has an offset of 'chall0+0x1100' and displays assembly code for a sequence of pushes and moves. It starts with a push of rdi, followed by a series of pushes to stack locations, each preceded by a mov instruction from memory. The assembly code is highly repetitive, showing a loop or a large block of generated assembly.  
Both windows have a 'Previous' and 'Next' button at the bottom right.

## 레지스터 뷰

레지스터 뷰는 현재 CPU 레지스터들의 값을 보여주는 뷰입니다.

Registers	Value
<b>Customize...</b>	
Reg	Value
rax	0
rcx	0
rdx	0
rbx	7ffe381f2000
rsp	170eefef9a0
rbp	170eefef9a0
rsi	0
rdi	170eefef9f0
r8	0
r9	0
r10	0
r11	0
r12	7ffe381f2480
r13	170ed95000

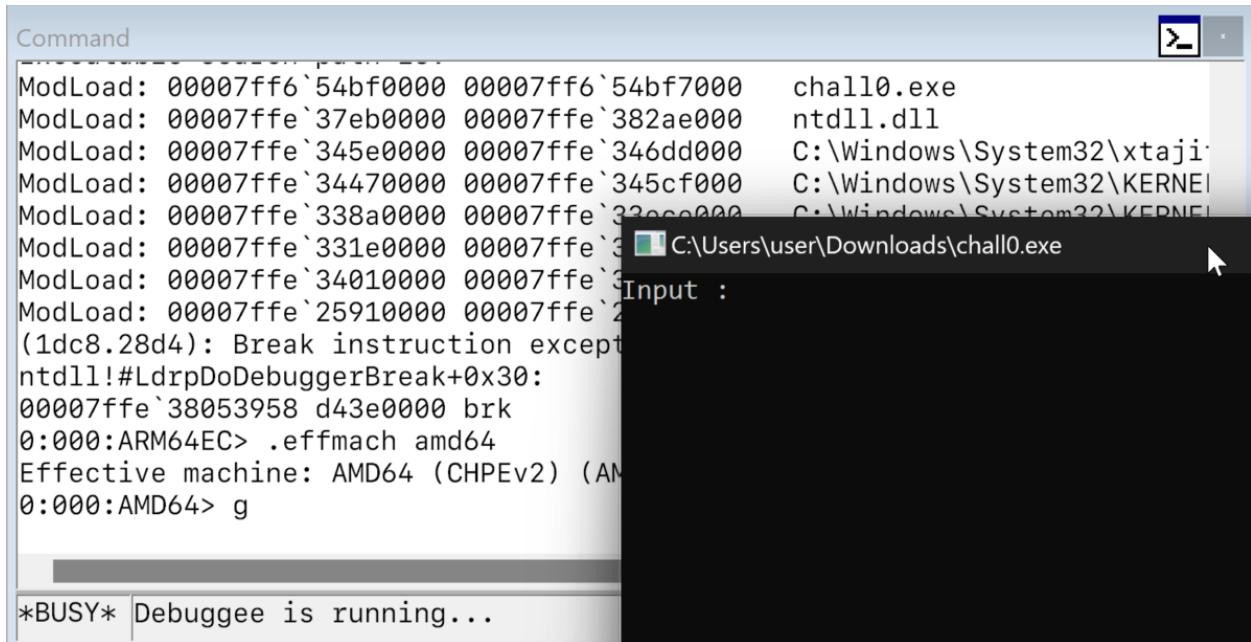
## 커맨드 뷰

커맨드 뷰는 커맨드를 실행하고 결과를 확인할 수 있는 WinDbg의 핵심적인 뷰입니다. 사용자가 입력한 커맨드 이외에 WinDbg가 출력하는 메시지도 기본적으로 커맨드 뷰에 표시됩니다.

```
Command
Executable search path is:
ModLoad: 00007ff6`54bf0000 00007ff6`54bf7000 chall0.exe
ModLoad: 00007ffe`37eb0000 00007ffe`382ae000 ntdll.dll
ModLoad: 00007ffe`345e0000 00007ffe`346dd000 C:\Windows\System32\xtaji...
ModLoad: 00007ffe`34470000 00007ffe`345cf000 C:\Windows\System32\KERNE...
ModLoad: 00007ffe`338a0000 00007ffe`33ece000 C:\Windows\System32\KERNE...
ModLoad: 00007ffe`331e0000 00007ffe`332d1000 C:\Windows\SYSTEM32\apphe...
ModLoad: 00007ffe`34010000 00007ffe`34201000 C:\Windows\System32\ucrtba...
ModLoad: 00007ffe`25910000 00007ffe`25946000 C:\Windows\SYSTEM32\VCRUN...
(1dc8.28d4): Break instruction exception - code 80000003 (first chance)
ntdll!#LdrpDoDebuggerBreak+0x30:
00007ffe`38053958 d43e0000 brk      #0xF000
0:000:ARM64EC> .effmach amd64
Effective machine: AMD64 (CHPEv2) (AMD64)

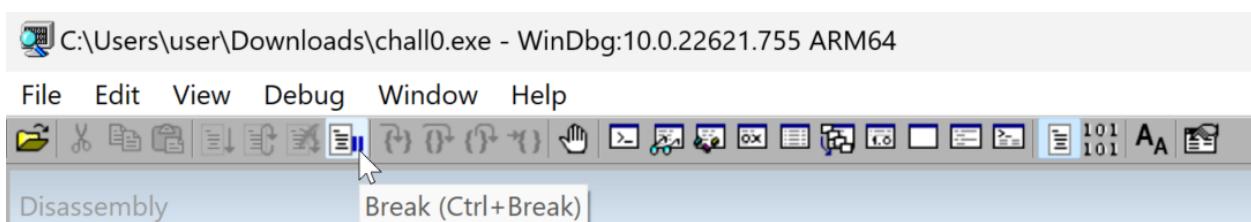
0:000:AMD64>
```

커맨드 뷰 하단의 입력 칸에 커맨드를 입력하고 Enter 키를 눌러 실행합니다. 예를 들어 바이너리를 계속 실행하는 커맨드인 `g` 를 실행하면 멈춰 있던 바이너리가 실행되면서 입력을 기다리는 것을 확인할 수 있습니다.

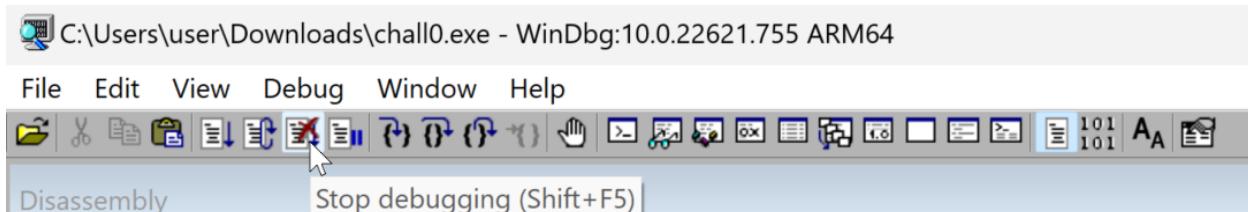


## 일시정지와 종료하기

상단 메뉴의 일시정지 모양 Break 버튼을 클릭하면 실행 중인 바이너리를 일시정지 시킬 수 있습니다.



Break 버튼 왼쪽의 X자 모양 ‘Stop Debugging’ 버튼, 도돌이표 모양 Restart 버튼을 클릭하면 각각 디버깅을 종료하거나, 바이너리를 다시 시작할 수 있습니다.



## 기초적인 커맨드

WinDbg를 효과적으로 사용하기 위해서는 필요에 따라 여러 가지 커マン드를 다를 수 있어야 합니다. `chall0.exe` 파일을 직접 동적 분석하면서 자주 사용되는 커マン드의 사용법을 알아보겠습니다.

## 중단점 조작

- `bp [address]` - `address` 주소에 중단점을 설치합니다.
- `bl` - 설정된 중단점을 나열합니다.
- `bd [breakpoint]` , `be [breakpoint]` - 중단점을 비활성화, 활성화합니다.

중단점(breakpoint)은 바이너리의 실행을 의도적으로 일시정지 하도록 할 지점을 의미합니다. 필요한 위치에 중단점을 설치하고 해당 지점에서 레지스터와 메모리의 값 등을 조사하는 기술은 기본적이지만 아주 중요한 동적 분석 기법입니다.

`bp [address]` 커マン드로 중단점을 설치할 수 있습니다. 입력 문자열을 검사하는 `sub_140001000` 함수를 호출하는 인스트럭션에 중단점을 설치해 보겠습니다.

```

    lea      rcx, [rsp+138h+var_118]
    call    sub_140001000
    test   eax, eax
    jz     short loc_140001166

```

```

    lea      rcx, Buffer ; "Correct"
    call   cs:puts
    jmp    short loc_140001173

```

```

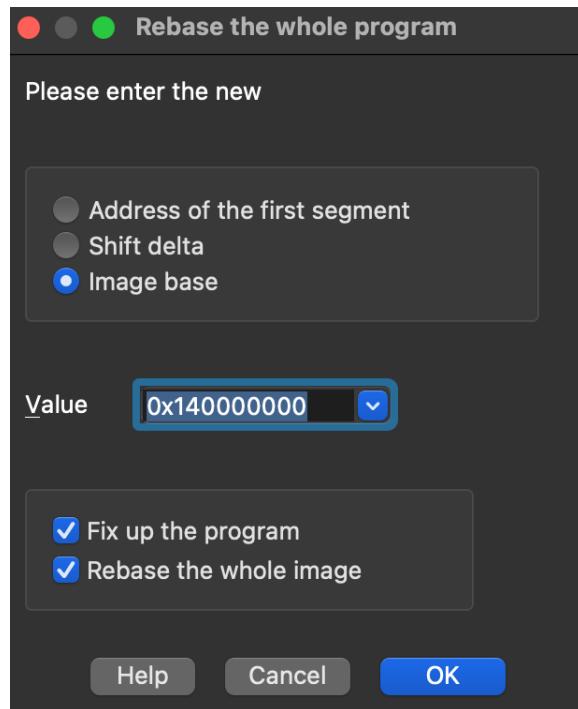
loc_140001166:
    lea      rcx, aWrong ; "Wrong"

```

,209) (352,662) 0000054E 000000014000114E: main+4E (Synchronized with Hex View-1)

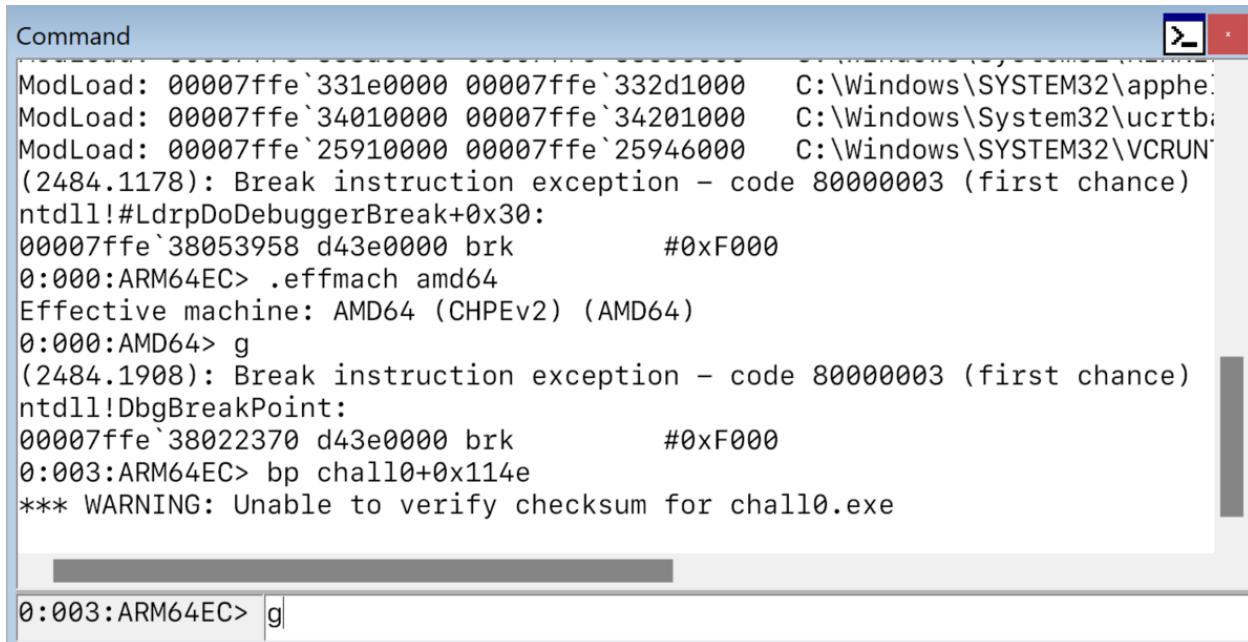
중단점을 설치할 위치는 `main+0x4e`입니다. 그런데 디버거가 바이너리의 함수 정보를 인식할 수 있도록 하는 심볼 파일이 없으므로 주소를 이용해 중단점을 설치해야 합니다.

주의할 점은 IDA에서 보이는 것과 같이 인스트럭션의 주소가 `0x14000114e`가 아니라는 것입니다. IDA 메뉴에서 Edit-Segments-Rebase program을 선택하면 바이너리의 Image base가 `0x140000000`으로 설정되어 있습니다. 이는 IDA가 임의로 설정한 주소입니다.



실제로 바이너리가 실행될 때는 메모리의 어떠한 주소에 로드될지 미리 알 수 없습니다. 따라서 인스트럭션의 주소는 `chall0.exe` 바이너리의 주소를 기준으로 오프셋 `0x114e`에 있다고 생각해야 합니다. WinDbg에서는 이러한 주소를 `chall0+0x114e`와 같이 나타낼 수 있습니다.

Windbg의 커マン드 뷰에서 `bp chall0+0x114e` 커맨드를 실행해 해당 인스트럭션에 중단점을 설치하고, `g` 커맨드로 중단점까지 실행하여 보겠습니다.

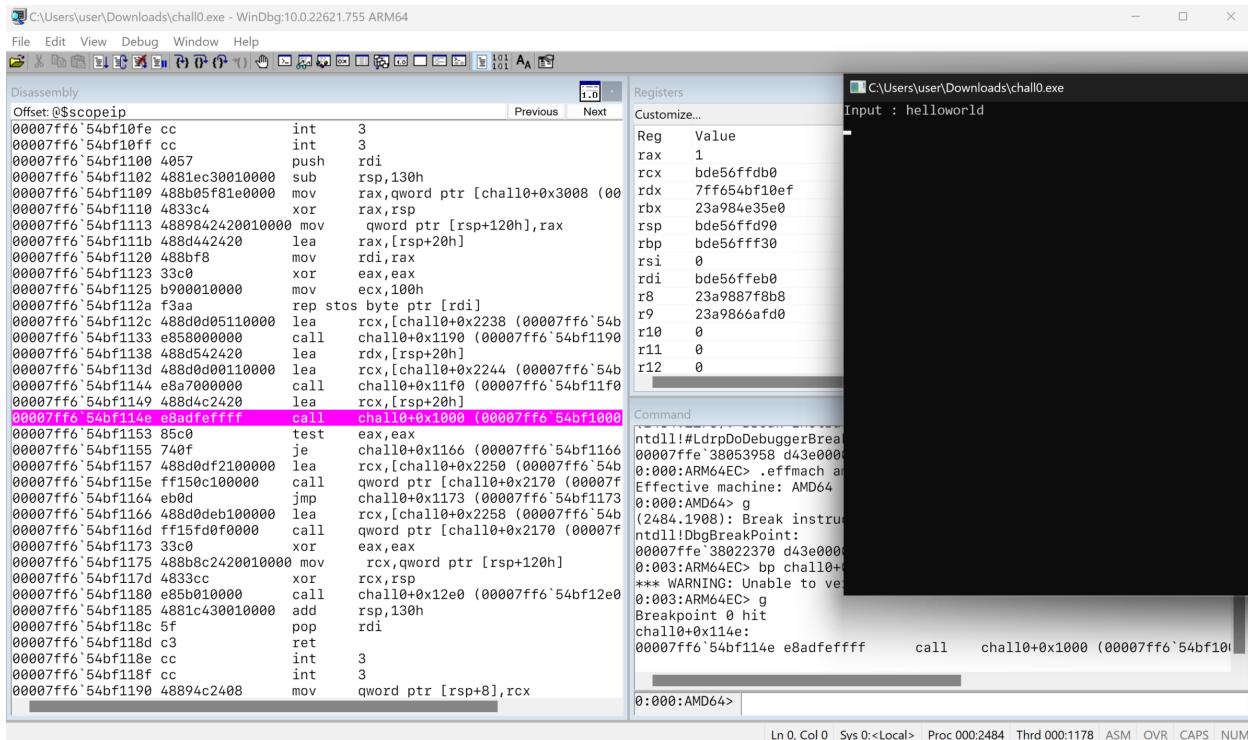


The screenshot shows the WinDbg command window with the following text:

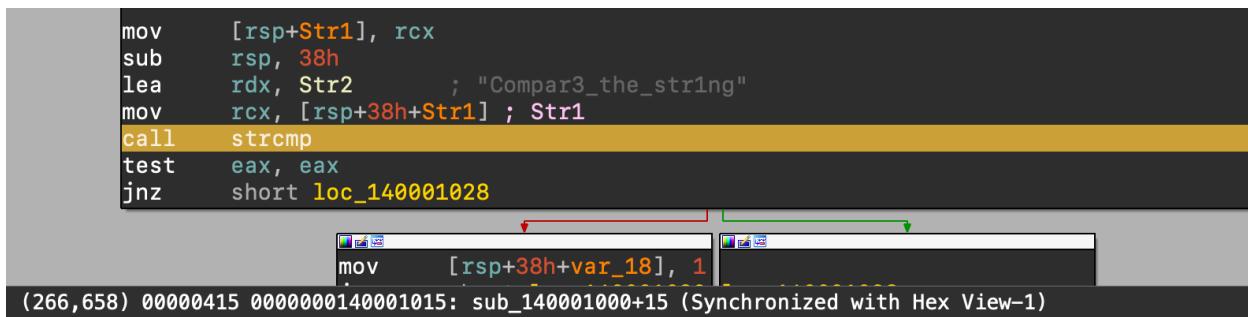
```
Command
ModLoad: 00007ffe`331e0000 00007ffe`332d1000 C:\Windows\SYSTEM32\apphe...
ModLoad: 00007ffe`34010000 00007ffe`34201000 C:\Windows\System32\ucrtba...
ModLoad: 00007ffe`25910000 00007ffe`25946000 C:\Windows\SYSTEM32\VCRUN...
(2484.1178): Break instruction exception - code 80000003 (first chance)
ntdll!#LdrpDoDebuggerBreak+0x30:
00007ffe`38053958 d43e0000 brk #0xF000
0:000:ARM64EC> .effmach amd64
Effective machine: AMD64 (CHPEv2) (AMD64)
0:000:AMD64> g
(2484.1908): Break instruction exception - code 80000003 (first chance)
ntdll!DbgBreakPoint:
00007ffe`38022370 d43e0000 brk #0xF000
0:003:ARM64EC> bp chall0+0x114e
*** WARNING: Unable to verify checksum for chall0.exe

0:003:ARM64EC> g
```

바이너리가 문자열 “helloworld”를 입력받은 후 중단점에 도달하여 일시정지 하였습니다.



이번에는 같은 방법으로 `sub_140001000` 함수 내에서 `strcmp` 함수를 호출하는 인스트럭션에 중단점을 설치하고 계속 실행하여 보겠습니다. 해당 인스트럭션의 주소는 `chall0+0x1015`입니다.



```

Command
00007ffe`38053958 d43e0000 brk          #0xF000
0:000:ARM64EC> .effmach amd64
Effective machine: AMD64 (CHPEv2) (AMD64)
0:000:AMD64> g
(2484.1908): Break instruction exception - code 80000003 (first chance)
ntdll!DbgBreakPoint:
00007ffe`38022370 d43e0000 brk          #0xF000
0:003:ARM64EC> bp chall0+0x114e
*** WARNING: Unable to verify checksum for chall0.exe
0:003:ARM64EC> g
Breakpoint 0 hit
chall0+0x114e:
00007ff6`54bf114e e8adfeffff      call     chall0+0x1000 (00007ff6`54bf100
0:000:AMD64> bp chall0+0x1015

0:000:AMD64> g

```

두 번째 중단점에 일시정지한 상황에서 `bl` 커맨드로 중단점을 나열하면, 첫 번째와 두 번째 중단점이 각각 0번, 1번 중단점으로 등록되어 있습니다. 왼쪽의 숫자 옆에 있는 'e'는 중단점이 활성화되어 있음을 나타냅니다. 중단점의 번호를 이용해 `bd 0`, `be 0` 과 같이 중단점을 비활성화, 활성화시킬 수 있습니다.

```

0:000:AMD64> bl
 0 e Disable Clear  AMD64 00007ff6`54bf114e      0001 (0001) 0:****
 1 e Disable Clear  AMD64 00007ff6`54bf1015      0001 (0001) 0:****

0:000:AMD64>

```

## 메모리 조작

- `d* [address]` - `address` 주소의 값을 특정 형식으로 출력합니다.
- `e* [address] [value]` - `address` 주소에 값 `value` 를 대입합니다.
  - `a` (아스키 문자열), `u` (유니코드 문자열), `b` (바이트), `w` (2바이트), `d` (4바이트), `q` (8바이트)

메모리 조작 커マン드를 이용하면 바이너리의 실행 도중 메모리의 값을 조사하거나 직접 대입할 수 있습니다. 실습을 위해 바이너리를 다시 시작하고, `chall0+0x114e`, `chall0+0x1015` 에 중단점을 다시 설치하겠습니다. Apple Silicon 환경의 경우 바이너리를 다시 시작한 이후 `.effmach amd64` 커マン드를 먼저 실행해야 합니다.

The screenshot shows the WinDbg command window with the following content:

```
Command
ModLoad: 00007ffe`34470000 00007ffe`345cf000 C:\Windows\System32\KERNEI
ModLoad: 00007ffe`338a0000 00007ffe`33ece000 C:\Windows\System32\KERNEI
ModLoad: 00007ffe`331e0000 00007ffe`332d1000 C:\Windows\SYSTEM32\apphe
ModLoad: 00007ffe`34010000 00007ffe`34201000 C:\Windows\System32\ucrtba
ModLoad: 00007ffe`25910000 00007ffe`25946000 C:\Windows\SYSTEM32\VCRUN
(24c0.fc): Break instruction exception - code 80000003 (first chance)
ntdll!#LdrpDoDebuggerBreak+0x30:
00007ffe`38053958 d43e0000 brk #0xF000
0:000:ARM64EC> .effmach amd64
Effective machine: AMD64 (CHPEv2) (AMD64)
0:000:AMD64> bp chall0+0x114e
*** WARNING: Unable to verify checksum for chall0.exe
0:000:AMD64> bp chall0+0x1015
0:000:AMD64> g

*BUSY* Debuggee is running...
```

`chall0+0x114e`에서 `sub_140001000` 함수의 인자는 `rcx` 레지스터에 보관되어 있습니다. `rcx` 레지스터가 나타내는 값은 인자 문자열의 주소입니다. 이 주소에 실제로 문자열이 존재하는지 확인하겠습니다.

Registers	
Customize...	
Reg	Value
rax	1
rcx	a9f477fd50
rdx	7ff654bf10ef
rbx	22ec83e35e0
rsp	a9f477fd30
rbp	a9f477fed0
rsi	0
rdi	a9f477fe50
r8	22ec85e2454
r9	22ec820afd0
r10	0
r11	0
r12	0

`d* [address]` 커맨드로 메모리의 값을 조사할 수 있습니다. 이 때 `*` 는 형식을 의미하며 `a` , `u` , `b` , `w` , `d` , `q` 중 하나입니다. 문자열의 주소임을 알고 있으므로 형식은 `a` 를 사용합니다. `da 2df990` 또는 `da rcx` 커맨드로 입력한 문자열이 실제로 메모리에 존재함을 확인할 수 있습니다.

```
0:000:AMD64> da rcx
000000a9`f477fd50  "helloworld"

0:000:AMD64>
```

`db rcx` , `dq rcx` 커맨드로 바이트 단위, 8바이트 단위로도 조사할 수 있습니다.

```
0:000:AMD64> db rcx
000000a9`f477fd50  68 65 6c 6c 6f 77 6f 72-6c 64 00 00 00 00 00 00 hello
000000a9`f477fd60  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ....
000000a9`f477fd70  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ....
000000a9`f477fd80  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ....
000000a9`f477fd90  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ....
000000a9`f477fdb0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ....
000000a9`f477fdc0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ....
```

```
0:000:AMD64>
```

`e* [address]` 커맨드로 메모리에 값을 대입할 수 있습니다. 이번에는 `rcx` 가 나타내는 주소에 다른 값을 대입하여 인자를 전혀 다른 문자열로 바꾸어 보겠습니다. `ea rcx "deadbeefcafebabe"` 커맨드로 문자열 “deadbeefcafebabe”를 대입합니다.

```
0:000:AMD64> ea rcx "deadbeefcafebabe"
```

```
0:000:AMD64>
```

`da rcx` , `db rcx` 커맨드 등으로 대입한 주소의 값을 조사하면 원래 있었던 “helloworld”와 전혀 다른 문자열인 “deadbeefcafebabe”가 존재하고 있습니다.

```
0:000:AMD64> da rcx
000000a9`f477fd50  "deadbeefcafebabe"
0:000:AMD64> db rcx
000000a9`f477fd50  64 65 61 64 62 65 65 66-63 61 66 65 62 61 62 65  deadl
000000a9`f477fd60  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ....
000000a9`f477fd70  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ....
000000a9`f477fd80  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ....
000000a9`f477fd90  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ....
000000a9`f477fdb0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ....
000000a9`f477fdc0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ....
```

```
0:000:AMD64>
```

## 레지스터 조작

- `r [register]` - 레지스터 `register`의 값을 조사합니다.
- `r [register]=[value]` - 레지스터 `register`에 값 `value`를 대입합니다.

`g` 커맨드로 계속 실행하면 라이브러리 함수 `strcmp`를 호출하는 `chall0+0x1015`에 일시정지합니다. `da rcx`, `da rdx`로 인자를 조사하면 문자열 “deadbeefcafebabe”, “Compar3\_the\_str1ng”입니다.

```
0:000:AMD64> g
Breakpoint 1 hit
chall0+0x1015:
00007ff6`54bf1015 e89e0e0000      call    chall0+0x1eb8 (00007ff6`54bf1el
0:000:AMD64> da rcx
000000a9`f477fd50  "deadbeefcafebabe"
0:000:AMD64> da rdx
00007ff6`54bf2220  "Compar3_the_str1ng"

0:000:AMD64>
```

`p`, `t` 커맨드로 인스트럭션을 한 개 실행할 수 있습니다. `p` 커맨드는 함수를 호출하는 인스트럭션이면 함수를 전부 실행하고, `t` 커맨드는 함수 내부로 진입합니다. `p` 커맨드로 `call` 인스트럭션을 실행합니다.

`r [register]` 커맨드로 레지스터의 값을 조사할 수 있습니다. 함수의 리턴 값이 보관되는 `rax` 레지스터를 `r rax` 커맨드로 조사하면 값은 1입니다.

```
0:000:AMD64> p
chall0+0x101a:
00007ff6`54bf101a 85c0          test    eax,eax
0:000:AMD64> r rax
rax=0000000000000001

0:000:AMD64>
```

값이 1인 이유는 `strcmp` 함수의 동작 상 두 문자열의 첫 바이트인 “d”와 “C”부터 서로 다르기 때문입니다. `rax` 레지스터의 값으로 0을 대입하면 바이너리는 `strcmp` 함수가 0을 리턴하였기 때

문에 두 문자열의 비교 결과가 같다고 판단할 것입니다.

`r [register]=[value]` 커맨드로 레지스터에 값을 대입할 수 있습니다. `r rax=0` 으로 `rax` 레지스터에 값 0을 대입합니다.

```
0:000:AMD64> r rax=0  
0:000:AMD64>
```

이후 `g` 커맨드로 계속 실행하면 바이너리가 “Correct”를 출력합니다. 대입한 값이 그대로 `sub_140001000` 함수의 리턴 값이 되었기 때문에 사용자가 문자열을 정확하게 입력했다고 판단한 것입니다.

```
Command >  
000000a9`f477fd50  "deadbeefcafebabe"  
0:000:AMD64> da rdx  
00007ff6`54bf2220  "Compar3_the_str1ng"  
0:000:AMD64> p  
chall0+0x101a:  
00007ff6`54bf101a  85c0  
0:000:AMD64> r rax  
rax=0000000000000001  
0:000:AMD64> r rax=0  
0:000:AMD64> g  
ModLoad: 00007ffe`31910000  
ModLoad: 00007ffe`34ca0000  
ntdll!#NtWaitForWorkViaWor  
00007ffe`3801ff24  f94003e9  
  
Input : helloworld  
Correct  
-  
0:002:ARM64EC>
```

## 커맨드 정리

커맨드	기능
<code>bp [address]</code>	중단점 설치
<code>bl</code>	중단점 나열

커맨드	기능
<code>bd [breakpoint]</code> , <code>be [breakpoint]</code>	중단점 비활성화, 활성화
<code>d* [address]</code>	메모리 값 조사
<code>e* [address] [value]</code>	메모리 값 대입
<code>r [register]</code>	레지스터 값 조사
<code>r [register]=[value]</code>	레지스터 값 대입
<code>g</code>	계속 실행 (continue)
<code>p</code>	인스트럭션 한 개 실행 (step over)
<code>t</code>	인스트럭션 한 개 실행 (step in)
<code>.restart</code>	다시 시작