

Evan's Debugger(EDB) Manual

By bl4ck3y3

(<http://hisjournal.net/blog>)

Abstract

이 문서는 리눅스의 디버깅툴인 Evan's Debugger을 소개하고 사용방법을 설명하기 위해 작성되었습니다. 이 문서에서 설명하는 Evan's Debugger는 0.9.6 버전임을 알립니다.



Content

0x01	Introduce	3
1.1	EDB와 GDB	3
0x02	Install	6
0x03	Plugins	8
3.1	Analyzer	8
3.2	BinarySearcher	8
3.3	Bookmarks	9
3.4	BreakpointManager	9
3.5	CheckVersion	9
3.6	DumpState	10
3.7	Environment	10
3.8	FunctionFinder	10
3.9	Hardware BreakpointManager	11
3.10	HeapAnalyzer	11
3.11	OpcodeSearcher	11
3.12	OpenFiles	12
3.13	ReferenceSearcher	13
3.14	StringSearcher	13
3.15	SymbolViewer	14
0x04	How to use	15
4.1	화면 구성	15
4.2	Disasm	16
4.3	Data Dump	17
4.4	Register	18
4.5	Bookmarks	19
4.6	Stack	20
4.7	Debug	21
0x05	Tip	22
5.1	문자열을 검색하여 루틴 찾기	22
5.2	Argument 지정	24
5.3	함수 호출 빈도로 루틴 찾기	25
	Reference	27

0x01. Introduce

리눅스에는 [GDB](#)라는 강력한 디버깅툴이 있습니다. ELF 파일에 대한 디버깅, 메모리를 덤프한 CORE 파일, 소스 파일, 커널에 대한 디버깅도 가능하죠. 하지만 GDB는 Command Line Interface 기반입니다. 사용상의 불편은 둘 째 치더라도 현재 디버깅하는 부분의 상태를 한 눈에 확인하기가 힘듭니다. 그래서 리눅스에서 디버깅하기 편한 툴이 뭐 없을까하고 찾아헤매었지요. 그러던 중에 찾은 것이 [Evan's Debugger](#)(이하 EDB로 통칭)입니다.

EDB는 Evan Teran이 만든 리눅스용 디버깅툴입니다. 현재 0.9.10 버전까지 릴리즈되었으며, GPL-2 라이선스에 의한 오픈소스입니다. QT4 기반의 Graphic User Interface인데, 사용해보면 [OllyDbg](#)와 많이 유사합니다. 실제로 OllyDbg가 EDB의 롤모델이라고 합니다.

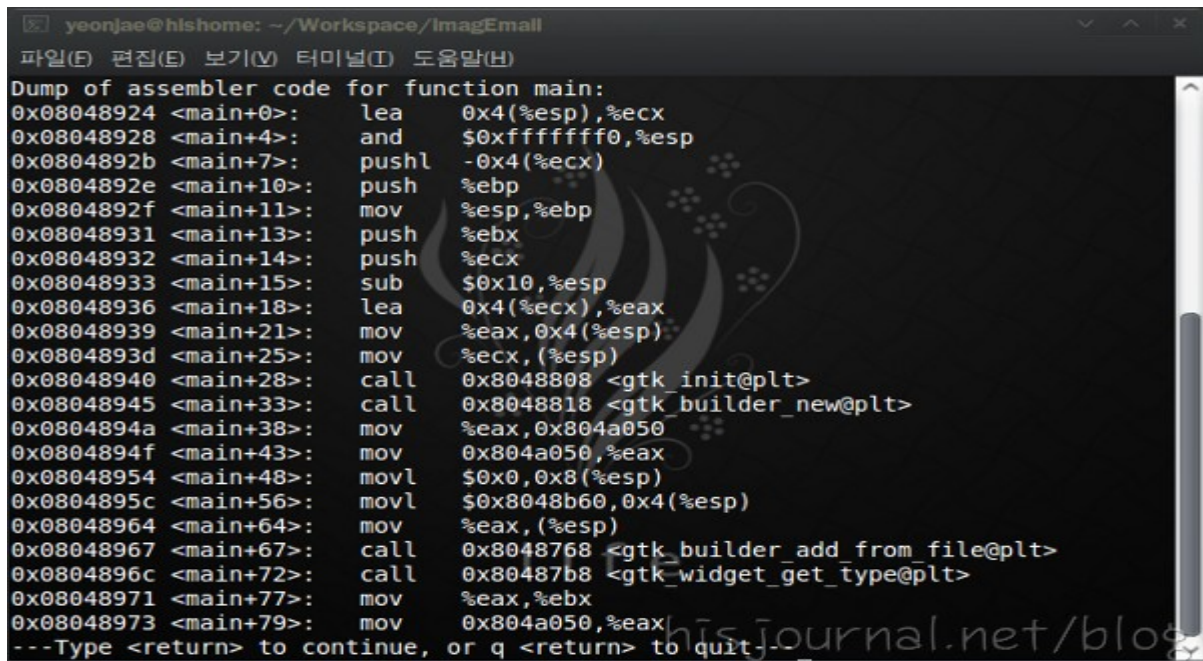
1.1. EDB와 GDB

앞서 소개에서도 GDB를 언급했습니다. 그만큼 GDB는 리눅스에서 땀해야 땀 수 없는 강력하고 대중적인 디버깅툴입니다. 리눅스에서 디버깅툴을 소개한다면 GDB가 빠질 수 없죠. 그래서 EDB와 GDB를 비교하는 자리를 마련했습니다.

EDB		GDB
GPL-2	라이선스	GPL
Linux	OS	Linux, HP-UX, Solaris, BSD, ...
x86, x86-64	플랫폼	x86, x86-64, s390, SPARC, RS/6000, ...
Graphic User Interface	인터페이스	Command Line Interface
ELF 파일 디버깅 메모리 CORE 파일 디버깅 프로세스 디버깅	기능	ELF 파일 디버깅 메모리 CORE 파일 디버깅 프로세스 디버깅 커널 디버깅 소스 레벨 디버깅 원격 디버깅
Intel	문법	AT&T
문서화 안 됨.	문서화	문서화 됨

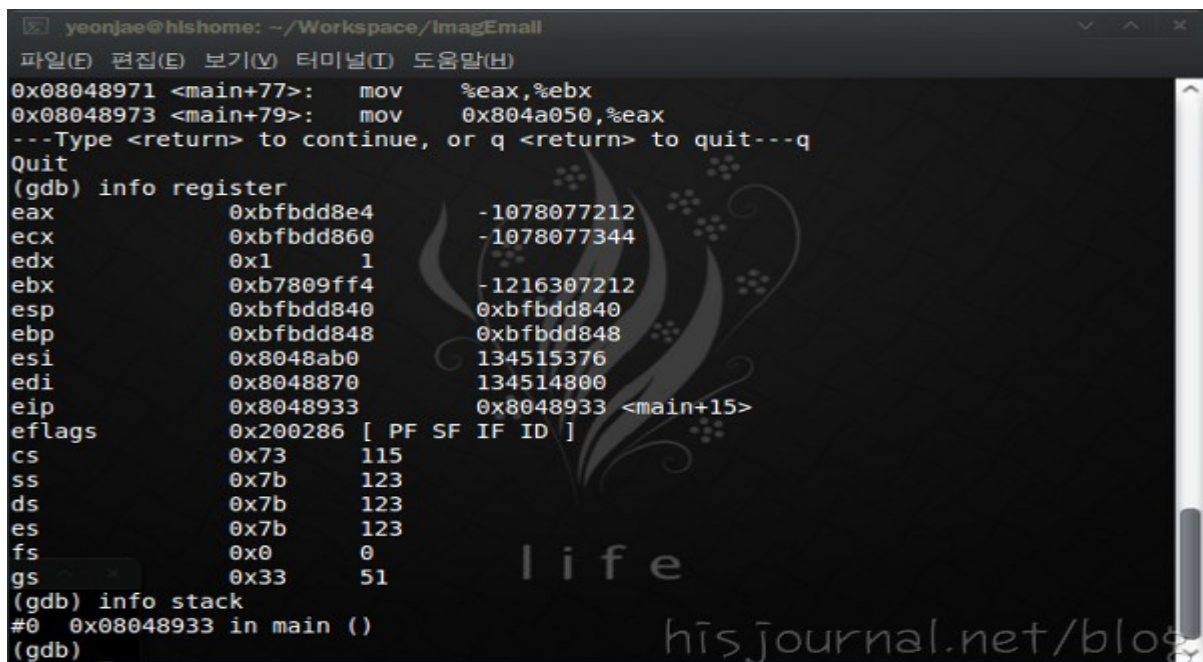
[표 1] EDB와 GDB 비교

[표 1]은 객관적으로 EDB와 GDB를 비교한 내용입니다. 대체로 GDB가 EDB보다 우위에 있음을 알 수 있습니다. 다만, EDB는 GDB와 달리 GUI 기반입니다. GUI와 CLI의 우위에 대해서는 논란의 여지가 있겠지만, 초보자도 쉽게 접근할 수 있냐는 물음에 대해서는 GUI가 분명 강점입니다.



```
yeonjae@hishome: ~/Workspace/ImgEmail
파일(E) 편집(E) 보기(V) 터미널(T) 도움말(H)
Dump of assembler code for function main:
0x08048924 <main+0>:    lea    0x4(%esp),%ecx
0x08048928 <main+4>:    and    $0xffffffff0,%esp
0x0804892b <main+7>:    pushl  -0x4(%ecx)
0x0804892e <main+10>:   push   %ebp
0x0804892f <main+11>:   mov    %esp,%ebp
0x08048931 <main+13>:   push   %ebx
0x08048932 <main+14>:   push   %ecx
0x08048933 <main+15>:   sub    $0x10,%esp
0x08048936 <main+18>:   lea    0x4(%ecx),%eax
0x08048939 <main+21>:   mov    %eax,0x4(%esp)
0x0804893d <main+25>:   mov    %ecx, (%esp)
0x08048940 <main+28>:   call   0x8048808 <gtk_init@plt>
0x08048945 <main+33>:   call   0x8048818 <gtk_builder_new@plt>
0x0804894a <main+38>:   mov    %eax,0x804a050
0x0804894f <main+43>:   mov    0x804a050,%eax
0x08048954 <main+48>:   movl   $0x0,0x8(%esp)
0x0804895c <main+56>:   movl   $0x8048b60,0x4(%esp)
0x08048964 <main+64>:   mov    %eax, (%esp)
0x08048967 <main+67>:   call   0x8048768 <gtk_builder_add_from_file@plt>
0x0804896c <main+72>:   call   0x80487b8 <gtk_widget_get_type@plt>
0x08048971 <main+77>:   mov    %eax,%ebx
0x08048973 <main+79>:   mov    0x804a050,%eax
---Type <return> to continue, or q <return> to quit---
```

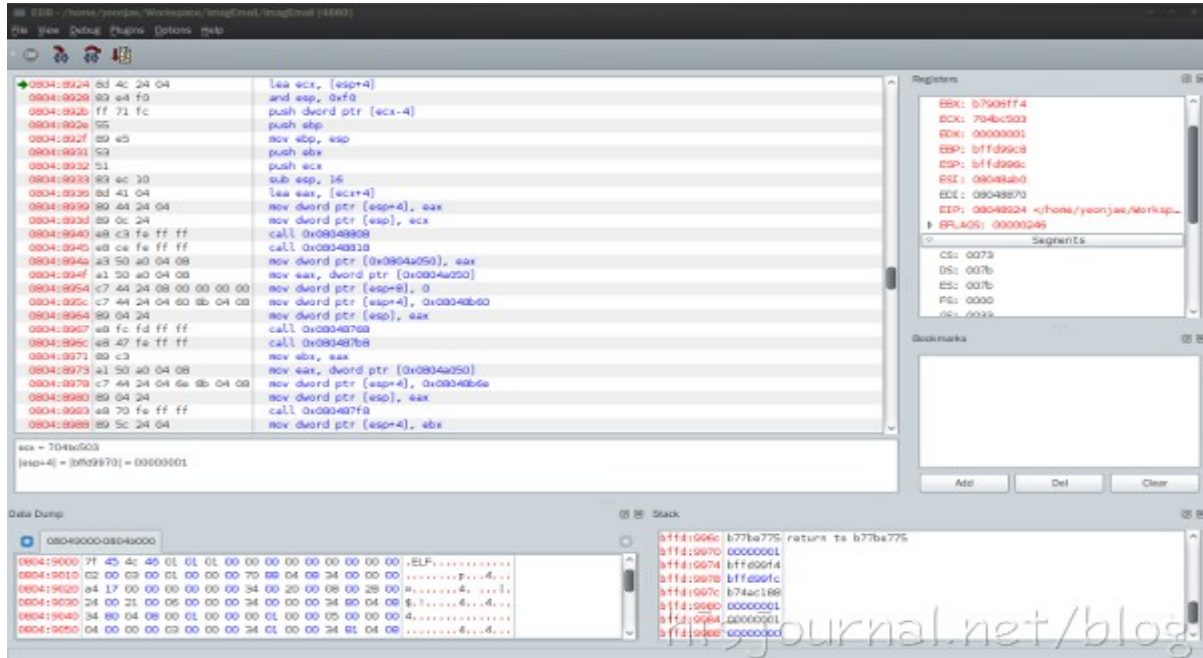
[그림 1] GDB에서의 Disasm 내용 확인



```
yeonjae@hishome: ~/Workspace/ImgEmail
파일(E) 편집(E) 보기(V) 터미널(T) 도움말(H)
0x08048971 <main+77>:   mov    %eax,%ebx
0x08048973 <main+79>:   mov    0x804a050,%eax
---Type <return> to continue, or q <return> to quit---q
Quit
(gdb) info register
eax             0xbfbdd8e4             -1078077212
ecx             0xbfbdd860             -1078077344
edx             0x1                    1
ebx             0xb7809ff4             -1216307212
esp             0xbfbdd840             0xbfbdd840
ebp             0xbfbdd848             0xbfbdd848
esi             0x8048ab0             134515376
edi             0x8048870             134514800
eip             0x8048933             0x8048933 <main+15>
eflags         0x200286 [ PF SF IF ID ]
cs              0x73                    115
ss              0x7b                    123
ds              0x7b                    123
es              0x7b                    123
fs              0x0                     0
gs              0x33                    51
(gdb) info stack
#0  0x08048933 in main ()
(gdb)
```

[그림 2] GDB에서의 레지스터, 스택 확인

GDB는 [그림 1]과 [그림 2]와 같이 어떤 내용을 확인하려면 그때마다 명령어를 입력해야하고 그 내용만 한 화면에서 확인할 수 있습니다.



[그림 3] EDB의 메인 화면

그러나 EDB는 [그림 3]과 같이 메인화면에서 모두 한 눈에 확인할 수 있습니다.

그리고 EDB는 문법에서 GDB와 차이가 있습니다. [그림 1]에서처럼 GDB는 Disasm 내용을 AT&T 문법으로 보여줍니다. 그러나 EDB는 [그림 3]과 같이 Intel 문법으로 보여주지요. 추후 EDB는 AT&T 문법으로도 볼 수 있을 예정입니다. 이것은 OllyDbg가 문법을 선택할 수 있는 것과 같습니다.

0x02. Install

원래는 <http://www.codef00.com/projects.php#Debugger> 에서 소스파일을 다운로드하여 컴파일해야 하지만, 사용자 환경에 의해 설치가 잘 되지 않는 경우가 있습니다. 그리고 소스파일로 설치하는 방법은 “정보보호119”의 binoopang 님이 문서로 잘 설명하셨기 때문에 여기서는 생략하겠습니다. 해당 문서의 주소는 아래 Reference에 남겨놓았습니다.

Fedora에서는 EDB를 패키지로 관리하고 있습니다. 만약 Fedora 사용자라면, 아래 명령어를 터미널에서 입력하면 쉽게 설치가 됩니다.

```
yum install edb
```

그런데 Ubuntu에서는 edb라는 이름으로 전혀 다른 패키지가 관리되고 있습니다. 그래서 [BackTrack 4](#)에서 관리되는 deb 패키지를 다운로드하여 설치해야 합니다. Ubuntu 사용자는 아래 링크를 따라 다운로드하여 deb 패키지를 설치하면 됩니다.

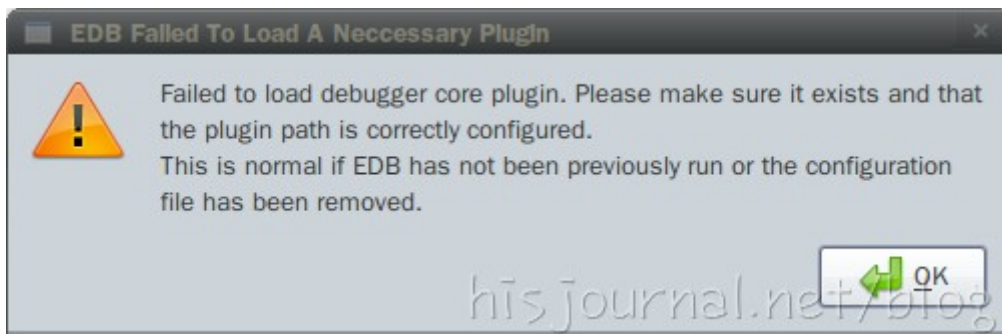
```
http://repo.offensive-security.com/dist/bt4/binary/evans-debugger.deb
```

만약 Gentoo Linux 사용자라면 개발자가 만들어놓은 스크립트 파일로 쉽게 설치할 수 있습니다. 아래 문서를 다운로드하여 실행하면 됩니다.

```
http://www.codef00.com/projects/edb-0.9.10.ebuild
```

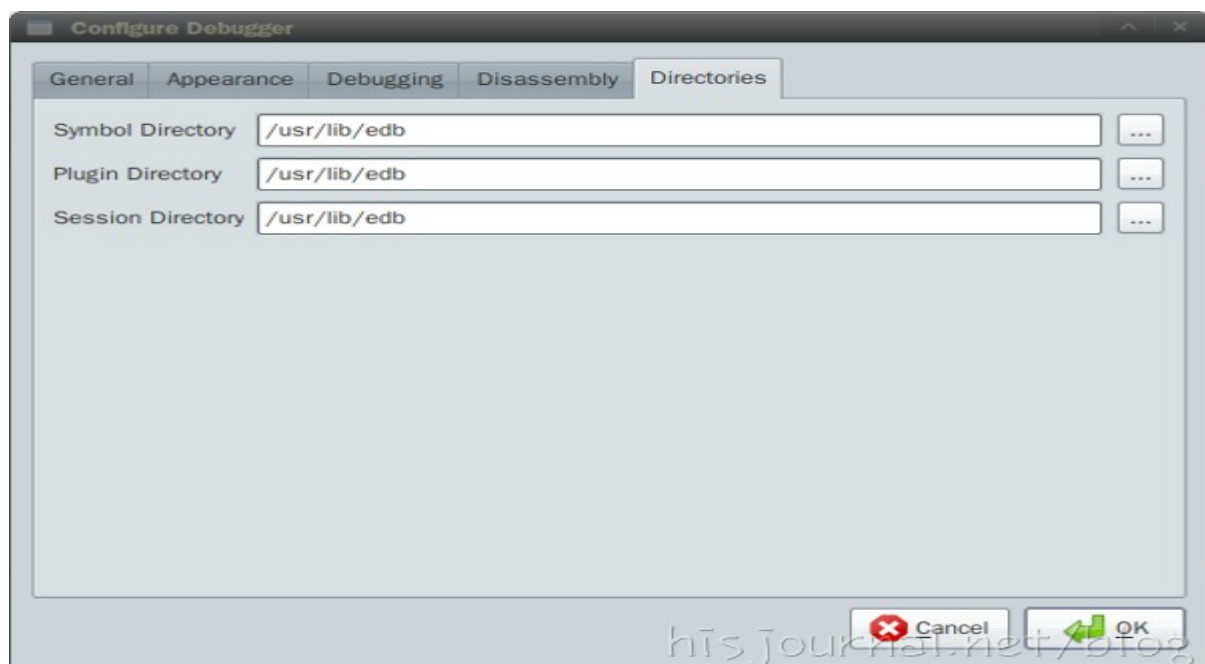
BackTrack 4는 Ubuntu 기반의 리눅스 배포버전으로, 해킹 분석을 위한 OS입니다. 여기에는 각종 다양한 분석툴들이 관리되고 있으며, EDB도 기본적으로 설치되어 있습니다.

설치 후 실행해서 [그림 4]와 같은 에러창이 나타난다면 플러그인의 경로 설정을 해줍니다.



[그림 4] Plugin을 로딩하지 못하면 나오는 에러창

[그림 4]의 OK 버튼을 누르면 Configure 창이 나옵니다. 거기서 Directories 탭을 선택하면 [그림 5]와 같습니다. 이 곳에서 플러그인의 경로를 설정하면 정상적으로 EDB가 동작합니다. 보통은 “/usr/lib/edb/” 아래에 플러그인들이 있습니다.



[그림 5] Plugin의 경로 설정

0x03. Plugins

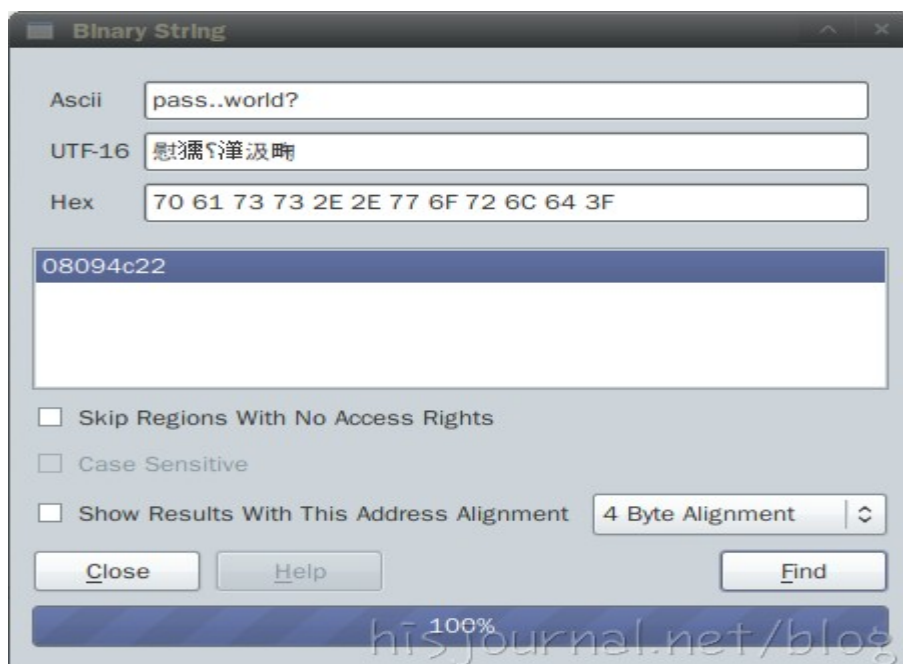
EDB는 분석하는데 유용한 기능들을 플러그인 형식으로 지원하고 있습니다. 지금은 모든 플러그인들을 EDB 개발자인 Evan Teran이 만들고 있습니다. 중간마다 내용이 없는 항목이 있습니다. 그것은 정상적으로 아직 동작되지 않는 플러그인들입니다.

3.1. Analyzer

Opcode를 더 정확하게 분석해줍니다. 그리고 함수 단위로 묶어서 경계를 나타내줍니다.

3.2. BinarySearcher

아스키, 유니코드, 헥스값을 검색하는 플러그인입니다.



[그림 6] BinarySearcher

[그림 6]과 같이 문자열이나 Opcode를 입력한 후 “Find” 버튼을 클릭하여 찾습니다. 검색 결과는 가운데 박스에 주소 형식으로 나옵니다. 즉, 검색 결과로 나온 주소로 찾아가면 해당 문자열이나 Opcode가 있습니다. 검색 결과를 더블클릭하면 Data Dump 창에 표시해줍니다. Disasm 창에서 보기 위해서는 직접 주소로 찾아가야 합니다. 찾아가는 방법은 이후 “How to use”에서 설명하도록 하겠습니다.

3.3. Bookmarks

EDB의 메인화면에 Bookmarks 창을 보여줍니다.

3.4. BreakpointManager



[그림 7] BreakpointManager

[그림 7]과 같은 브레이크포인트 관리창을 보여줍니다. 여기에서 브레이크포인트를 추가, 삭제할 수 있고 특정 조건을 설정할 수도 있습니다.

3.5. CheckVersion

새로운 버전이 있는지 확인합니다.



[그림 8] CheckVersion

만약 지금 사용중인 버전보다 최신 버전이 있다면 [그림 8]과 같은 창을 보여줍니다. EDB가 시작할 때

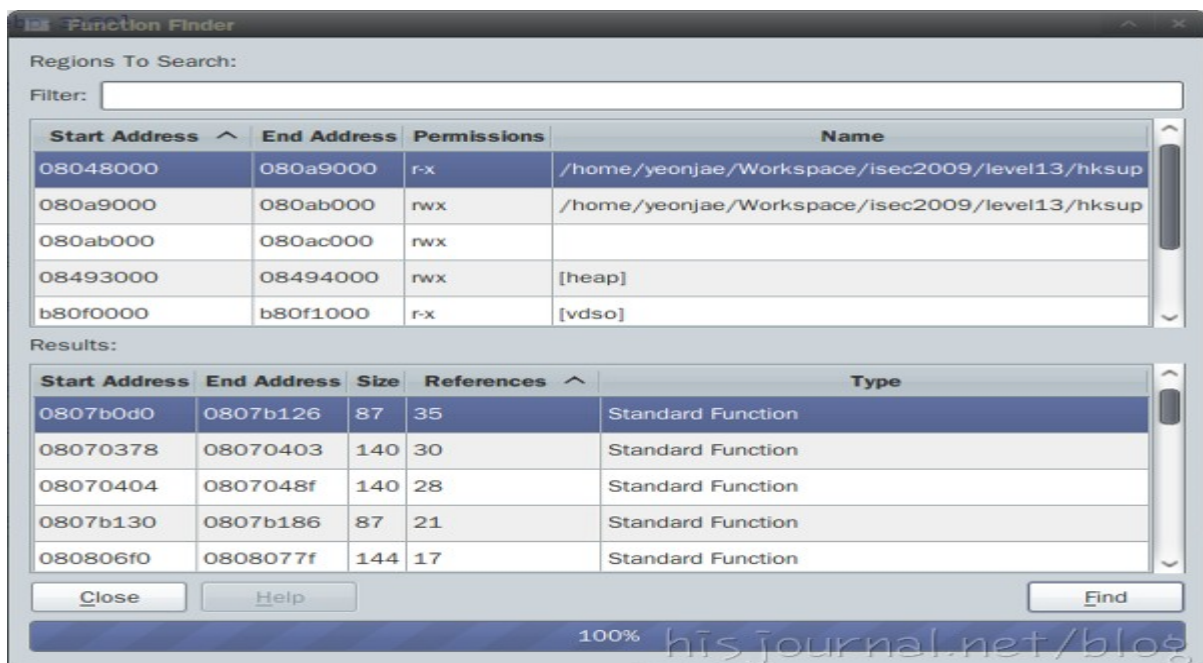
마다 새로운 버전이 있는지 확인하도록 설정할 수 있습니다.

3.6. DumpState

3.7. Environment

3.8. FunctionFinder

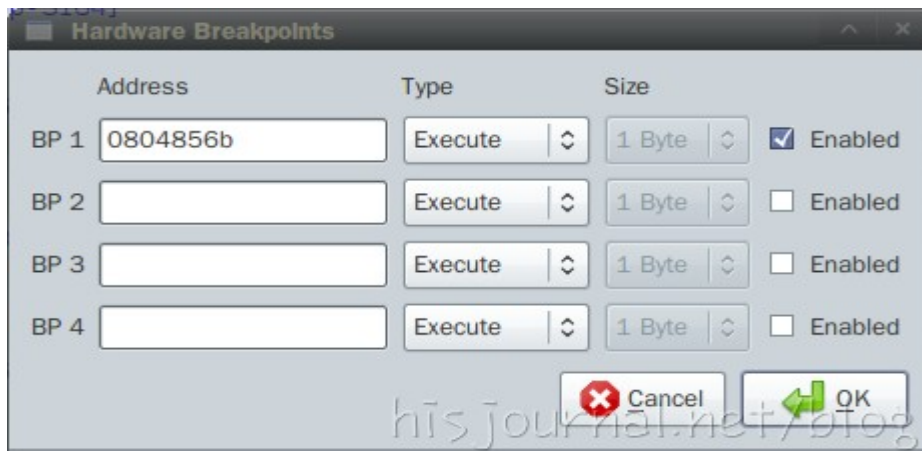
각 섹션별로 함수를 찾아주는 플러그인입니다.



[그림 9] FunctionFinder

[그림 9]와 같이 섹션을 선택한 후 “Find” 버튼을 누르면 그 섹션 안에 있는 함수들을 찾아냅니다. 함수의 시작과 끝의 주소가 나오고, 함수의 크기와 함수가 호출되는 횟수도 나옵니다. 함수를 선택하고 더블 클릭하면 Disasm 창에서 그 함수로 이동합니다.

3.9. Hardware BreakpointManager



[그림 10] Hardware BreakpointManager

[그림 10]과 같이 하드웨어 브레이크포인트를 관리합니다. Address에 브레이크포인트를 걸 주소를 입력하고 Type에서 실행, 쓰기, 읽기/쓰기 중에서 선택한 후 “Enabled”를 체크하면 브레이크포인트가 걸립니다.

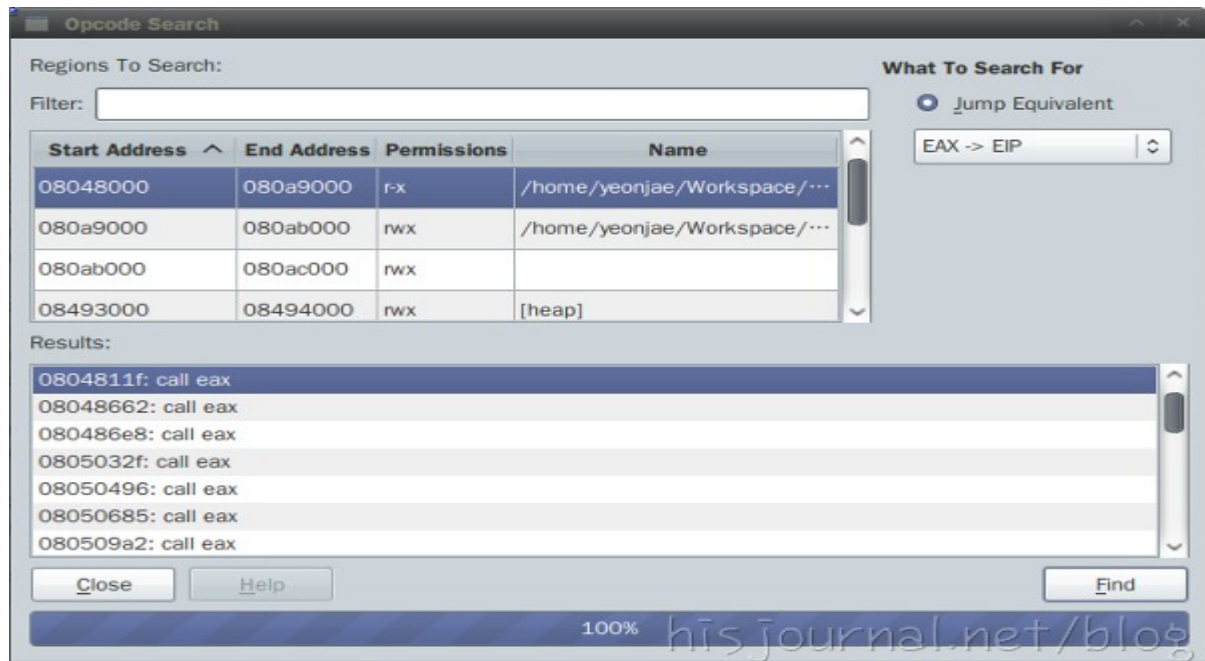
OllyDbg와 마찬가지로 하드웨어 브레이크포인트는 4개까지만 설정할 수 있습니다. 이것은 인텔 아키텍처에서 하드웨어 브레이크포인트를 저장하는 디버그 레지스터가 4개만 존재하기 때문입니다. 일반적인 브레이크포인트와 하드웨어 브레이크포인트의 차이는 아래 Reference를 참조하시기 바랍니다.

3.10. HeapAnalyzer

3.11. OpcodeSearcher

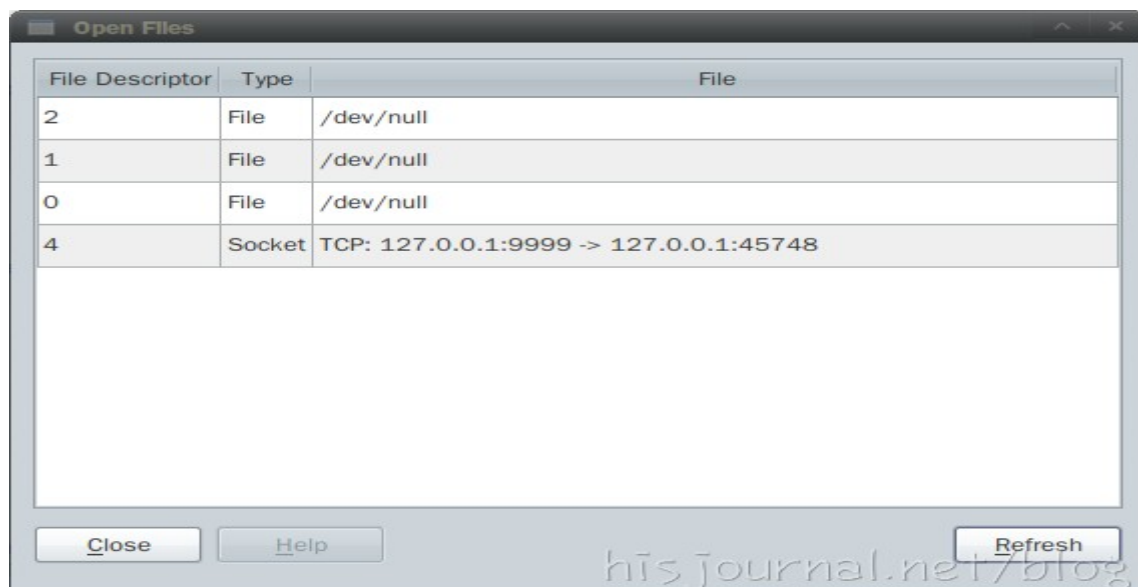
Opcode를 찾아주는 플러그인입니다. 앞의 BinarySeacher와 다른 점은 OpcodeSearcher는 각 섹션에서 특정 Opcode를 찾는다는 점입니다.

[그림 11]처럼 jump 명령어를 모두 찾아줍니다. 여기서 참조하는 레지스터값을 조건으로 둘 수 있습니다. [그림 11]의 결과는 모두 call eax나 jmp eax입니다. 이 중에서 하나를 선택하여 더블클릭하면 Disasm 창에서 그 Opcode가 있는 주소로 이동합니다.



[그림 11] OpcodeSearcher

3.12. OpenFiles



[그림 12] OpenFiles

디버깅 중인 프로세스가 열고 있는 파일들을 보여줍니다. 리눅스는 프로세스의 모든 것을 파일처럼 관리하기 때문에 특정 파일 뿐만 아니라 [그림 12]처럼 소켓에 대한, 메모리에 대한 것도 보여줍니다.

3.13. ReferenceSearcher

특정 주소를 참조하는 Opcode를 찾는 플러그인입니다.



[그림 13] ReferenceSearcher

[그림 13]과 같이 상단의 입력란에 주소를 입력하고 “Find” 버튼을 누르면 입력한 주소를 참조하는 Opcode가 있는 주소를 보여줍니다. 나온 결과를 선택하여 더블클릭하면 Data Dump 창에서 Opcode가 있는 곳으로 이동합니다.

3.14. StringSearcher

문자열만 찾아주는 string 프로그램과 같습니다. 섹션을 선택한 후 “Find” 버튼을 클릭하면, Results란에 문자열들이 모두 출력됩니다. BinarySearcher와 중복되는 듯 하지만, BinarySearcher는 Code를 중심으로 검색하고 StringSearcher는 문자열을 중심으로 검색한다는 점에서 차이가 있습니다.



[그림 14] StringSearcher

[그림 14]에서 출력된 문자열을 더블클릭하면 Data Dump창에서 문자열이 있는 지점으로 이동합니다.

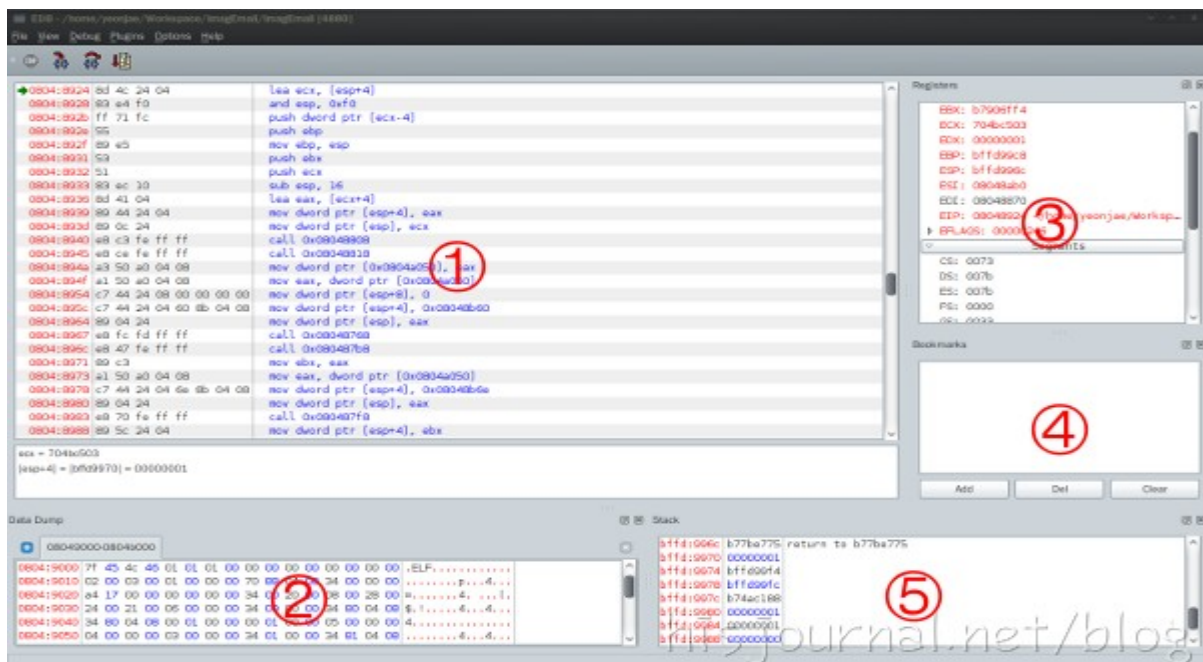
3.15 SymbolViewer

0x04. How to use

EDB의 주요기능들인 플러그인들을 알아보았습니다. 이제는 EDB를 어떻게 사용하는지 알아보겠습니다. 원래 EDB는 OllyDbg를 롤모델로 하여 제작되었기 때문에 인터페이스나 사용방법에 대해서도 OllyDbg와 많이 유사합니다. 그러므로 부담 없이 읽어 내려가시면 금방 익숙해질 것입니다.

4.1 화면 구성

화면 구성 역시 OllyDbg와 유사합니다. 그림을 보면서 하나하나 살펴보겠습니다.



[그림 15] EDB의 화면 구성

[그림 15]에서 EDB의 메인 화면을 5개의 영역으로 나누었습니다. 그 중 1번 영역이 Disasm 창입니다. EIP 주소와 Opcode, 그 Opcode를 해석하여 Intel 문법으로 출력한 부분이 있습니다. 아쉬운 점은 OllyDbg와 달리 코멘트가 없다는 점입니다. 분석할 때 코멘트의 유용성은 두 말 할 필요 없을 정도로 높기 때문에 매우 아쉬운 부분입니다. 아래 쪽의 작은 창은 OllyDbg와 마찬가지로 현재 EIP에 대한 정보를 보여줍니다.

2번 영역은 Data Dump 창입니다. 이 곳에서 메모리나 바이너리 파일의 Raw 값을 볼 수 있습니다.

그리고 아스키 값도 같이 볼 수 있습니다. 좌측의 “+” 버튼을 누르면 Data Dump 창이 하나 더 생기면서 탭이 활성화 됩니다. 그리고 우측의 “X” 버튼을 누르면 현재 보여지는 탭의 Data Dump 창을 닫습니다.

3번 영역은 Register 창입니다. 범용 레지스터, 플래그, 세그먼트, FPU의 정보를 볼 수 있습니다.

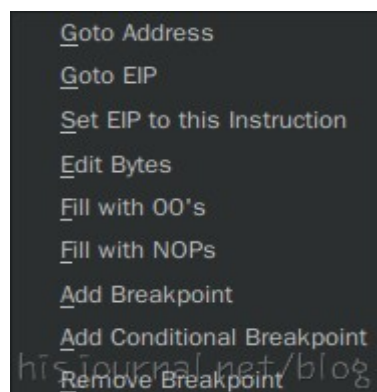
4번 영역은 Bookmarks 창입니다. Bookmarks 플러그인으로 보이거나 숨기게 할 수 있습니다.

5번 영역은 Stack 창입니다. 스택의 내용을 확인할 수 있습니다.

이 5개의 창은 회색 부분을 드래그&드롭을 이용하여 배치를 바꿀 수 있습니다. 혹은 별도의 창으로 떼어낼 수도 있습니다.

4.2. Disasm

Disasm 창에서 마우스 오른쪽 버튼을 클릭하면, 팝업 메뉴가 활성화됩니다.



[그림 16] Disasm 창의 메뉴

[그림 16]은 활성화된 메뉴를 찍어낸 것입니다. 9개의 메뉴가 있으며, 메뉴 이름이 매우 직관적이기 때문에 따로 설명하지는 않겠습니다. 다만, 주의할 점은 “Goto Address” 메뉴를 이용할 경우입니다. 일반적으로 주소를 08XXXXXX와 같이 입력하지만, 여기에서는 0x08XXXXXX와 같이 입력해야 이동함

니다.

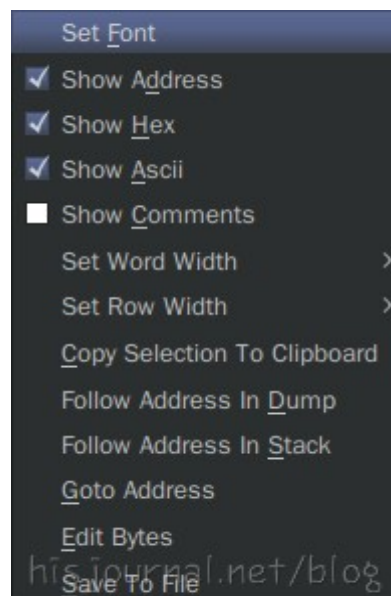
브레이크포인트를 걸 때, [그림 16]의 메뉴에서 선택해도 되지만, EIP를 더블클릭하여도 브레이크포인트가 설정됩니다.

Analyzer 플러그인을 사용하면, 함수 단위로 구분을 지어주고 jump 명령어가 있으면 화살표로 방향을 알려줍니다.

Disasm 창에서 참고할 점이 하나 있는데, 마우스휠로 내리거나 방향키로 내리면 Opcode와 해석한 내용이 바뀔 때가 있습니다. 이것은 1byte 단위씩 Opcode가 해석되다보니 1byte씩 밀릴 때마다 내용이 다르게 해석되기 때문입니다. 그럴 때는 다시 한 번 더 내려주면 정상적으로 보입니다.

4.3. Data Dump

Data Dump 창은 화면 구성을 설명할 때 언급했듯이 메모리나 바이너리 파일의 Raw 값을 보여줍니다. 이 곳에서 마우스 오른쪽 버튼을 클릭하면 [그림 17]과 같은 메뉴가 나옵니다.

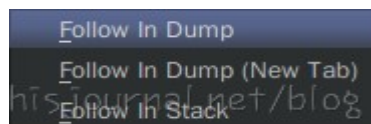


[그림 17] Data Dump 창의 메뉴

여기의 메뉴들 역시 직관적으로 설명되어 있기 때문에 별도로 설명하지는 않겠습니다. “Goto Address” 메뉴 역시 Disasm 창의 메뉴와 같습니다. Raw 값이 보이는 부분이나 아스키 값이 보이는 부분을 드래그하여 특정 부분만 선택할 수도 있으며, “Save To File”을 이용하면 수정한 Opcode를 파일로 저장할 수 있습니다.

4.4. Register

각각의 레지스터를 확인할 수 있는 Register 창은 마우스 오른쪽 버튼을 클릭하면 단 3개의 메뉴만 보입니다.



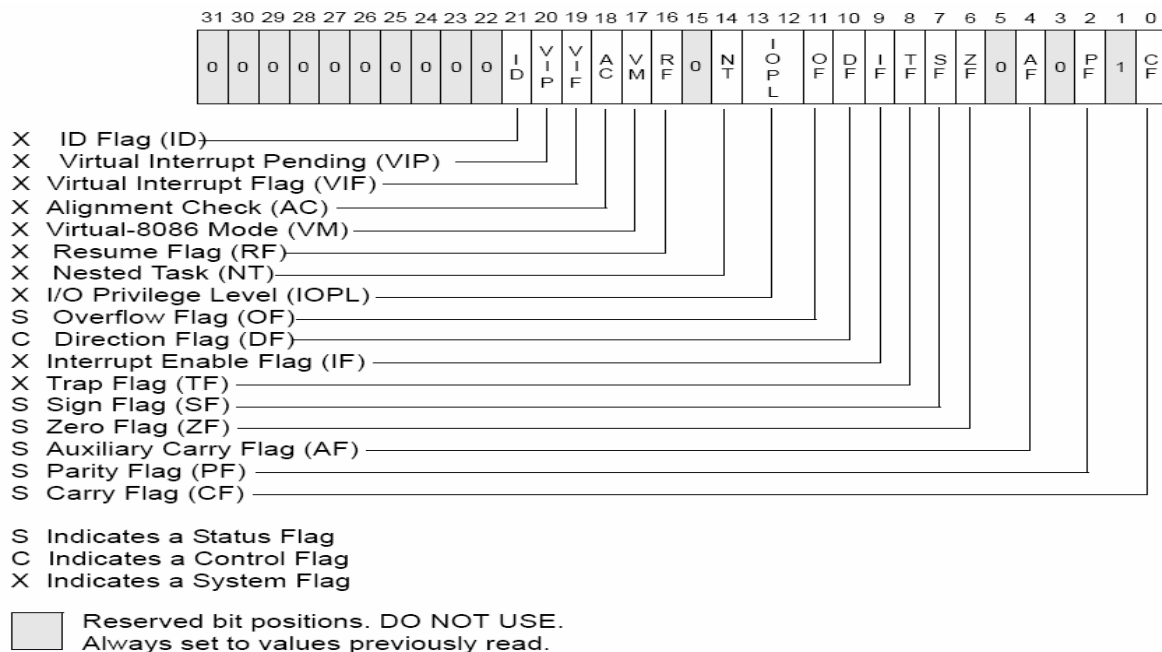
[그림 18] Register 창의 메뉴

[그림 18]과 같이 레지스터에 저장된 주소를 따라가는 메뉴만 있습니다. 첫 번째와 두 번째는 Data Dump 창에, 세 번째는 Stack 창에 내용을 보여줍니다. 두 번째 메뉴에서 New Tab 이라고 적혀 있는 것은 Data Dump 창에 새로운 탭을 활성화하여 보여주는 것입니다.

Register 창에서 중요한 것은 EFLAGS 값을 조작하는 방법입니다. OllyDbg에서는 간단히 클릭만 함으로써 플래그가 조작되었지만, EDB는 약간의 계산을 함으로써 조작을 해야합니다. 이 때 각 플래그의 위치와 값을 알아야 조작할 수 있습니다.

EFLAGS에 보이는 값은 16진수입니다. 즉, 각 플래그의 값들을 합친 값입니다. 그리고 EFLAGS 아래에 각 플래그의 상태가 보입니다. OF, DF, SF, ZF, AF, PF, CF의 7개 플래그가 있습니다. 대문자로 보이는 것은 비트 값이 1로 세팅되었다는 뜻이고, 소문자로 보이는 것은 비트 값이 0이라는 것입니다.

우선, 플래그를 조작하기 위해서 EFLAGS 값을 2진수 바이너리 값으로 바꾸어야 합니다. 가령, EFLAGS가 00000246이면, 2진수 바이너리 값은 01001000110입니다. 그리고 각 플래그의 위치를 고려하여 새로 세팅합니다.



[그림 19] 「해커 지망생들이 알아야 할 Buffer Overflow Attack의 기초」에서 발췌

각 플래그의 위치는 [그림 19]를 참조합니다. EFLAGS에는 IF, TF는 생략되어 있습니다. 이 플래그들은 특정 Opcode가 적용되지 않으면 보통 1로 세팅됩니다. 방금 전의 EFLAGS 값에서 ZF가 1로 세팅되어 있음을 알 수 있습니다. 이것을 0으로 바꾸면 010000001110이 되고 16진수로 변경하면, 206이 됩니다. EFLAGS를 더블클릭하여 편집 창을 띄운 후 변경한 값을 입력하고 “OK” 버튼을 누르면 플래그가 변경됩니다. 실제로 대문자였던 Z가 소문자 z로 바뀌는 것을 확인할 수 있습니다.

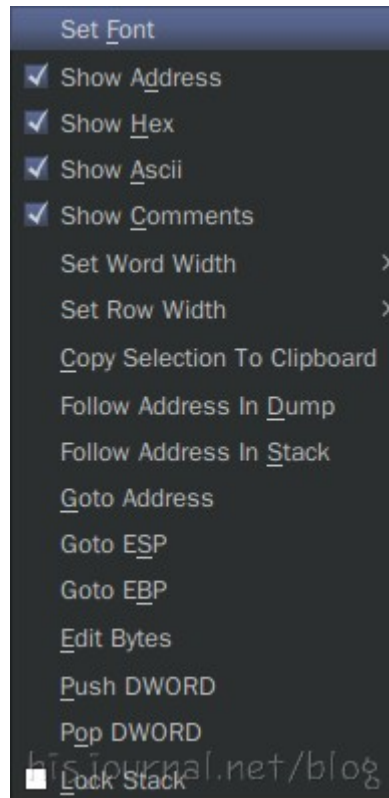
ZF는 jump 명령어에 많이 쓰이므로 자주 조작하게 되는 플래그입니다. 이 외의 플래그들에 대해서 자세한 내용은 Reference에 나와 있는 “WOWHACKER”의 달고나 님의 문서를 참조하시기 바랍니다.

4.5. Bookmarks

Bookmarks는 자주 들여다 볼 Opcode의 주소를 설정하여 간편하게 찾아갈 수 있는 기능입니다. “Add” 버튼을 눌러서 주소를 입력하고 “OK” 버튼을 클릭하면 북마크가 설정됩니다. 이 때 주소는 “Goto Address”처럼 0x08XXXXXX와 같은 형식으로 입력해야 합니다. “Del”과 “Clear” 버튼은 설정한 북마크를 지우는 메뉴입니다.

4.6. Stack

메모리의 스택을 확인할 수 있고 스택의 내용을 조작할 수 있습니다.



[그림 20] Stack 창의 메뉴

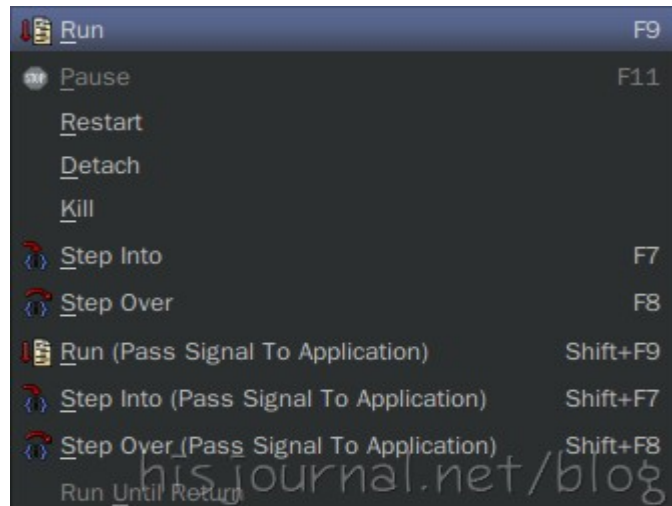
역시 이번에도 마우스 오른쪽 버튼을 클릭하면 [그림 20]과 같은 메뉴가 보입니다. 메뉴들이 직관적이므로 이번에도 따로 설명하지는 않겠습니다.

방금 전 스택을 조작할 수 있다고 하였는데, “Push DWORD”, “Pop DWORD”가 그 역할을 합니다. “Push DWORD”는 ESP를 4 감소시키면서 ESP의 내용을 변경합니다. 이 때 16진수 값이나 10진수 값을 넣을 수 있습니다. 반대로 “Pop DWORD”는 ESP를 4 증가시킵니다. 이 때는 단지 ESP만 변경할 뿐이지 스택의 내용은 변하지 않습니다.

“Lock Stack”는 ESP만 변경할 뿐, 스택의 내용은 변경시키지 않는 기능입니다.

4.7. Debug

디버깅을 위한 기초적인 기능들을 살펴보겠습니다.



[그림 21] Debug 메뉴

EDB의 상단 메뉴에서 “Debug”를 클릭하면, [그림 21]과 같은 메뉴가 보입니다. OllyDbg를 써 봤다면 꽤 친숙할 것입니다. 아래 [표 2]에서 각 메뉴를 정리했습니다.

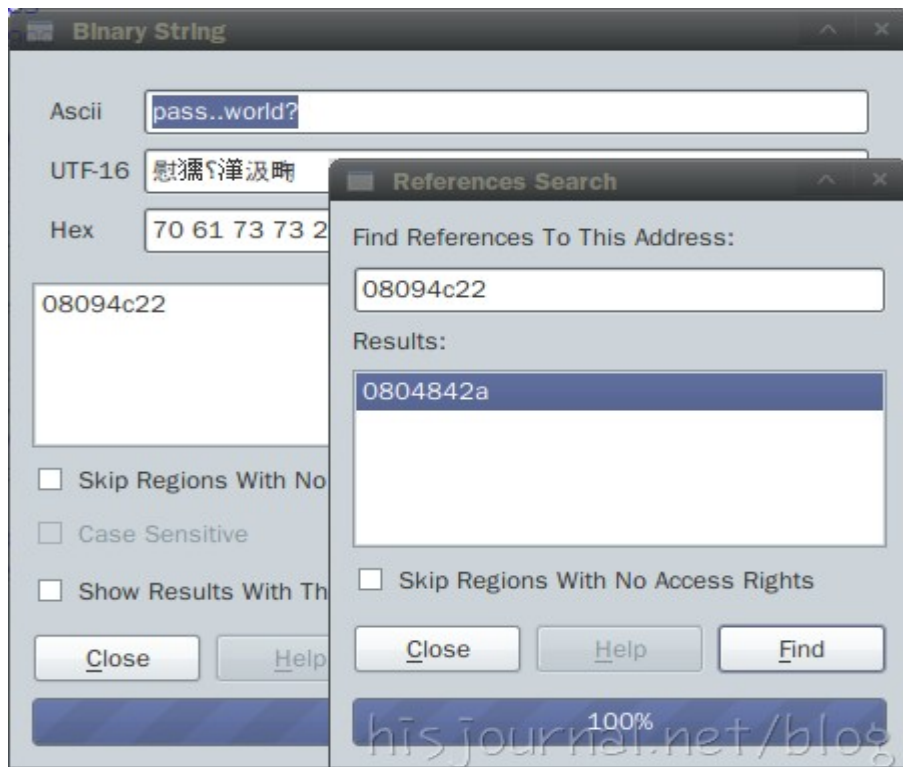
Menu	Hotkey	Description
Run	F9	실행. 입력대기 혹은 브레이크포인트까지 실행.
Pause	F11	정지.
Restart		현재 디버깅중인 프로세스를 다시 처음부터 디버깅.
Detach		현재 디버깅중인 프로세스를 디버깅 중단.
Kill		-
Step Into	F7	현재 EIP 실행. CALL을 만나면 CALL 내부의 EIP 가리킴.
Step Over	F8	현재 EIP 실행. CALL을 만나면 다음 EIP 가리킴.
Run (PSTA)	Shift+F9	-
Step Into (PSTA)	Shift+F7	-
Step Over (PSTA)	Shift+F8	-
Run Until Return		hisjournal.net/blog

[표 2] Debugging Menu

0x05. Tip

5.1. 문자열을 검색하여 루틴 찾기

프로그램을 분석할 때 문자열을 검색하여 루틴을 찾는 것은 반드시 알아야 할 기법이므로 EDB로 어떻게 할 수 있는지 알아보겠습니다.

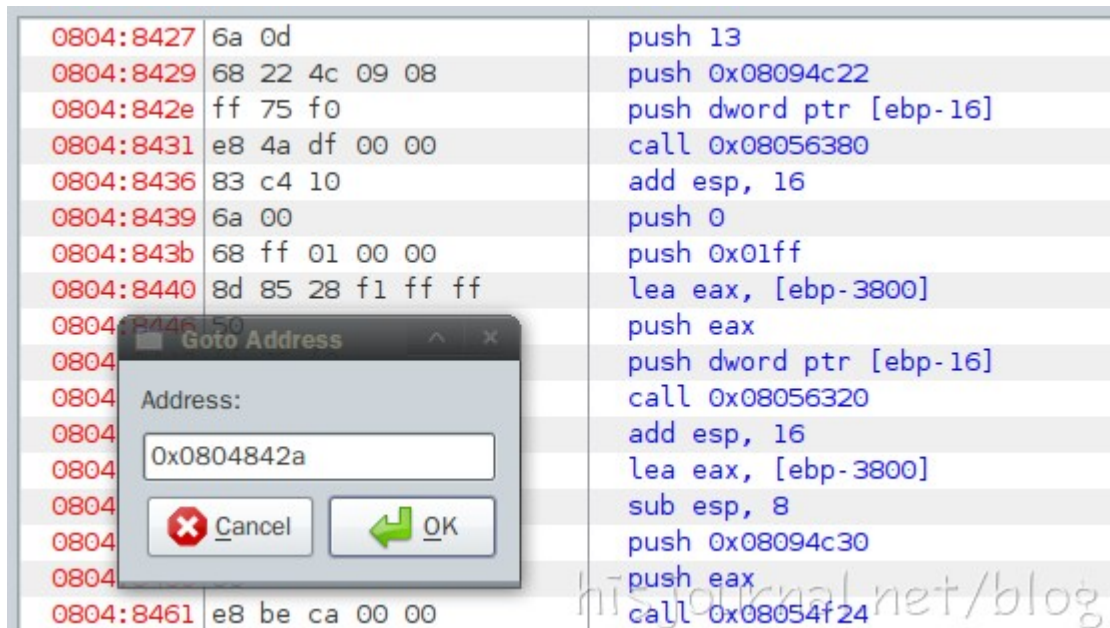


[그림 23] 문자열 검색

[그림 23]과 같이 BinarySearcher를 이용하거나 StringSearcher를 이용하여 문자열을 검색합니다. 그러면 문자열이 저장되어 있는 주소를 알 수 있습니다. 이 때 찾을 문자열은 프로그램을 실행했을 때 출력되는 문자열입니다. [그림 23]에서 “08094c22”를 더블클릭하면 Data Dump 창에서 문자열을 확인할 수 있습니다.

ReferenceSearcher에서 방금 전에 찾은 주소를 입력하여 문자열을 참조하는 Opcode를 찾습니다. [그림 23]에서는 0x0804842a에 해당 루틴이 있음을 찾았습니다.

루틴이 있는 주소를 알아냈으면 Disasm 창에서 루틴이 있는 곳으로 이동합니다.



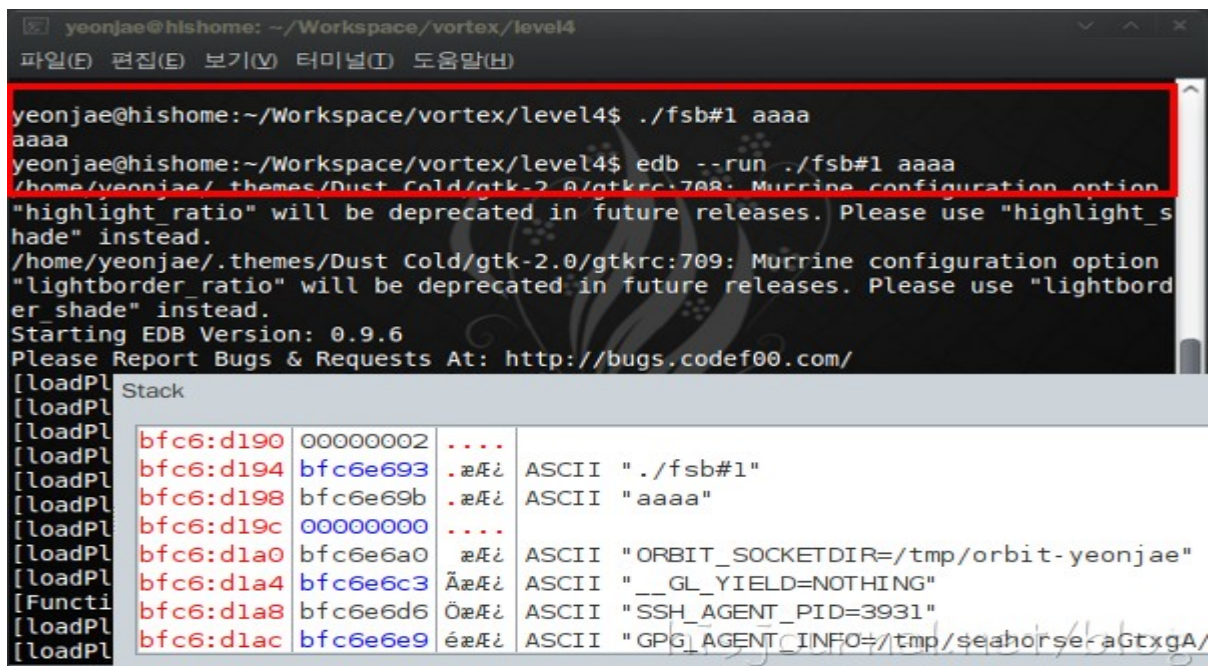
[그림 24] 문자열을 출력하는 루틴

[그림 24]처럼 “Goto Address”를 이용합니다. 이 때 0x08XXXXXX와 같이 입력해야 함을 주의해야 합니다. 0x0804842a로 이동하면 [그림 24]와 같은 루틴이 있습니다. 그리고 0x08048429에서 “pass..world?”가 저장된 주소를 스택에 넣는 것을 확인할 수 있습니다. 그리고 0x08048431에서 함수를 호출하여 문자열을 출력합니다.

5.2. Argument 지정

프로그램들 중에서는 실행할 때 인자를 입력해야 하는 경우가 있습니다. 그런데 EDB로 바이너리 파일을 열어서 분석하다 보면 이런 인자가 입력되지 않아 정상적인 분석이 힘들 때가 있습니다. 그런 경우를 대비하여 EDB에는 인자를 지정하는 메뉴가 있습니다. Options에서 Application Arguments를 열면 인자를 지정할 수 있는 창이 나타납니다. 이 곳에 인자를 입력한 후, 바이너리 파일을 열면 인자가 적용되었음을 알 수 있습니다.

또는 터미널에서 EDB를 실행하여 인자를 같이 넣어줄 수도 있습니다. 터미널에서 실행하기 위해서는 `-run` 옵션을 사용합니다.



```

yeonjae@hishome: ~/Workspace/vortex/level4
파일(F) 편집(E) 보기(V) 터미널(T) 도움말(H)

yeonjae@hishome:~/Workspace/vortex/level4$ ./fsb#1 aaaa
aaaa
yeonjae@hishome:~/Workspace/vortex/level4$ edb --run ./fsb#1 aaaa
/home/yeonjae/.themes/Dust Cold/gtk-2.0/gtkrc:708: Murrine configuration option
"highlight_ratio" will be deprecated in future releases. Please use "highlight_s
hade" instead.
/home/yeonjae/.themes/Dust Cold/gtk-2.0/gtkrc:709: Murrine configuration option
"lightborder_ratio" will be deprecated in future releases. Please use "lightbord
er_shade" instead.
Starting EDB Version: 0.9.6
Please Report Bugs & Requests At: http://bugs.codef00.com/

[loadPL] Stack
[loadPL]
[loadPL] bfc6:d190 00000002 .....
[loadPL] bfc6:d194 bfc6e693 .æÆ¿
[loadPL] bfc6:d198 bfc6e69b .æÆ¿ ASCII "./fsb#1"
[loadPL] bfc6:d19c 00000000 ..... ASCII "aaaa"
[loadPL] bfc6:d1a0 bfc6e6a0 æÆ¿ ASCII "ORBIT_SOCKETDIR=/tmp/orbit-yeonjae"
[loadPL] bfc6:d1a4 bfc6e6c3 ÅæÆ¿ ASCII "__GL_YIELD=NOTHING"
[loadPL] bfc6:d1a8 bfc6e6d6 ÔæÆ¿ ASCII "SSH_AGENT_PID=3931"
[loadPL] bfc6:d1ac bfc6e6e9 éæÆ¿ ASCII "GPG_AGENT_INFO=/tmp/seatdse-æGtxgA/

```

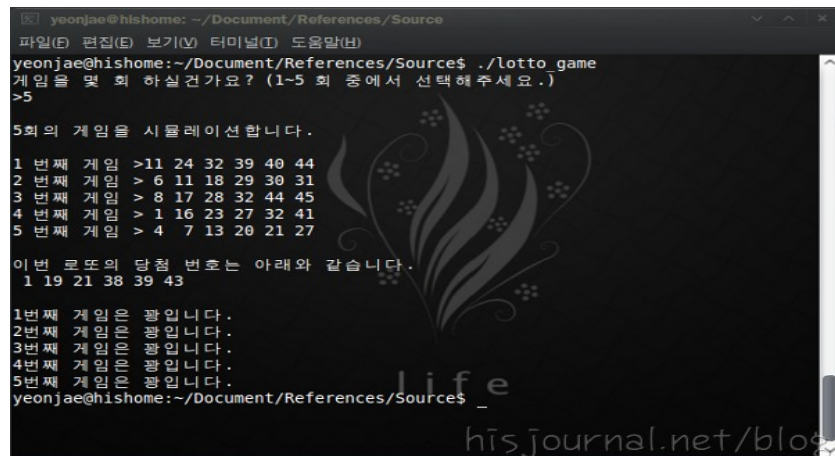
[그림 25] 터미널에서 EDB를 실행

[그림 25]는 터미널에서 `-run` 옵션을 이용하여 실행한 화면입니다. `fsb#1`이라는 프로그램은 인자로 문자열을 지정하여 출력하는 간단한 프로그램입니다. 빨간 박스 안을 보면 `-run` 옵션을 입력한 후 프로그램의 경로와 인자를 지정하였습니다. EDB의 Stack 창에서 인자 "aaaa"가 정확히 들어갔음을 알 수 있습니다.

`edb -run [경로] [인자]`

5.3. 함수 호출 빈도로 루틴 찾기

FunctionFinder를 이용하여 특정 함수가 호출되는 빈도로 루틴을 찾을 수도 있습니다. 이런 경우는 보통 암호/복호화가 이루어지는 프로그램을 분석할 때 쉽게 적용할 수 있는 방법입니다. 다음 예시로 들 프로그램은 로또를 시뮬레이션하는 프로그램입니다. 이 경우는 아직 EDB가 한글을 지원하지 않기 때문에 문자열을 검색할 수 없는 경우입니다.



```

yeonjae@hishome: ~/Document/References/Source
파일(F) 편집(E) 보기(V) 터미널(T) 도움말(H)
yeonjae@hishome:~/Document/References/Source$ ./lotto_game
게임을 몇 회 하실건가요? (1-5 회 중에서 선택해주세요.)
>5

5회의 게임을 시뮬레이션합니다.

1 번째 게임 >11 24 32 39 40 44
2 번째 게임 > 6 11 18 29 30 31
3 번째 게임 > 8 17 28 32 44 45
4 번째 게임 > 1 16 23 27 32 41
5 번째 게임 > 4 7 13 20 21 27

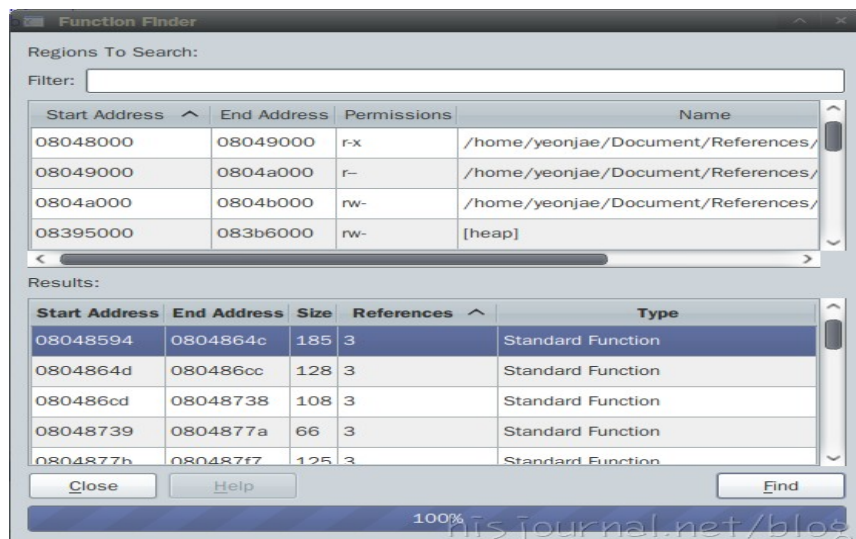
이번 로또의 당첨 번호는 아래와 같습니다.
1 19 21 38 39 43

1번째 게임은 틀립니다.
2번째 게임은 틀립니다.
3번째 게임은 틀립니다.
4번째 게임은 틀립니다.
5번째 게임은 틀립니다.
yeonjae@hishome:~/Document/References/Source$

```

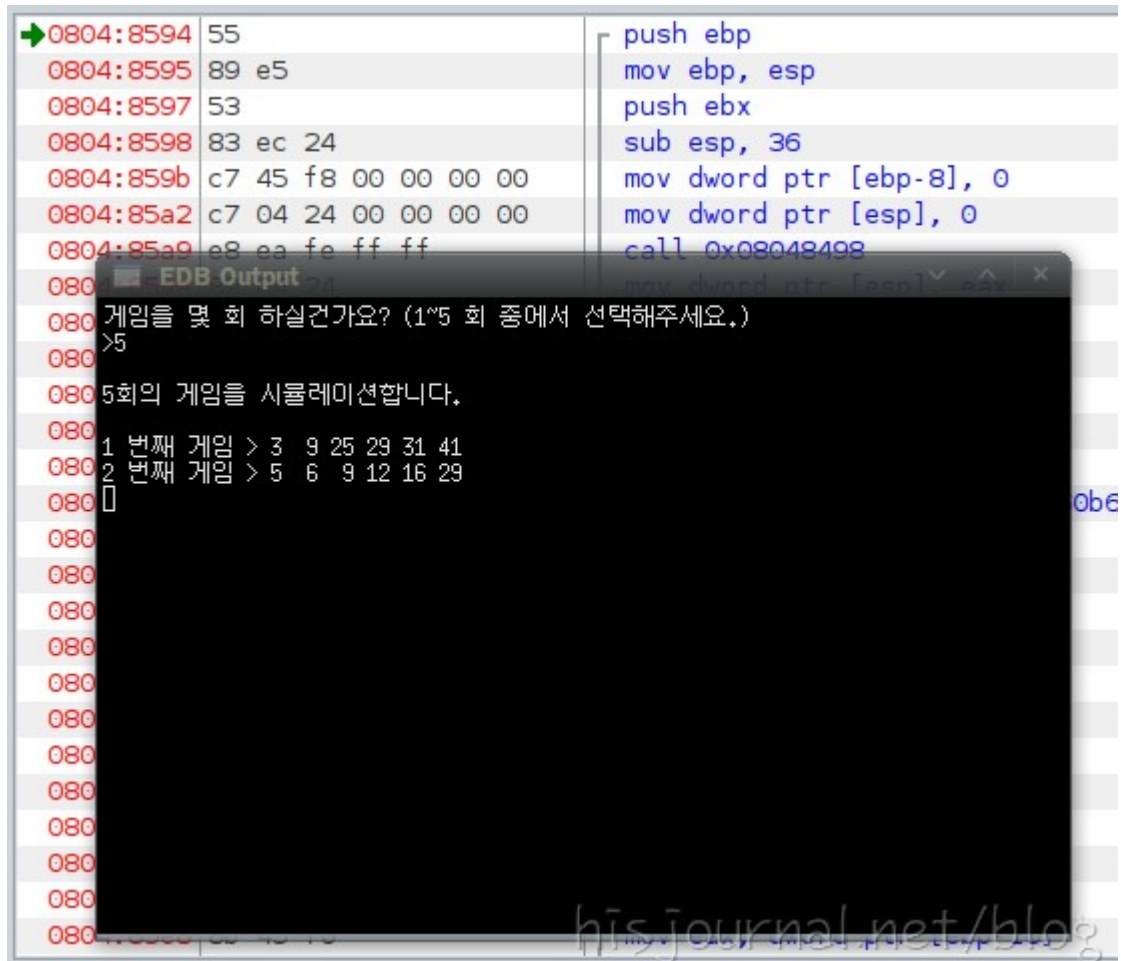
[그림 26] 로또 시뮬레이션

하지만 프로그램을 실행해보니 특정 함수가 수 차례 반복적으로 호출되는 것을 추측할 수 있습니다. 이럴 때 역시 함수가 호출되는 빈도를 이용하여 특정 루틴을 찾을 수가 있겠습니다.



[그림 27] 함수의 호출 빈도로 검색

FunctionFinder로 함수가 있을 법한 섹션을 검색합니다. 보통은 제일 상단의 섹션에 대부분의 함수가 있지만, 꼭 그렇다는 보장이 없으므로 몇 차례의 시행 착오가 필요합니다. [그림 27]처럼 “References”를 클릭하면 호출 빈도순으로 정렬됩니다. 여기서 제일 많이 호출되는 함수나 특정 횟수 만큼 호출되는 함수를 선택하여 더블클릭합니다.



[그림 28] 시뮬레이션한 게임을 출력하는 루틴

Disasm 창에 [그림 28]과 같이 0x08048594의 루틴이 보입니다. 함수의 시작 지점에 브레이크포인트를 걸어서 F9를 눌러보니 F9를 누를 때마다 시뮬레이션된 게임이 하나씩 출력됨을 확인하였습니다.

이 같이 함수의 호출 빈도로 루틴을 찾는 방법은 매우 유용하나 여러 차례의 시행착오를 겪어야 하며 그런 경험을 바탕으로 활용할 수 있는 분석 방법입니다.

Reference

edb :: <http://www.codef00.com/projects.php#Debugger>

binoopang 님의 문서 :: <http://linux-virus.springnote.com/pages/3512025>

디버거의 구현과 원리 :: <http://nchovy.kr/forum/2/article/440>

[해커 지망생들이 알아야 할 Buffer Overflow Attack의 기초](#) by 달고나 (WOWHACKER)