

# ThinctDbg 用法说明书

V1.0.0

## 一、RuntimeTrace 运行时反汇编代码跟踪

1. 脚本可以将运行过程中**执行过的反汇编代码**作为路径, 按照要求记录下来。

- a. 直接阅读分析这样的单一路径的代码简单于庞杂的整体分析;
- b. 基于不同的输入, 可以得到不同的执行路径;
- c. 可以综合多个路径来分析或猜测功能函数;
- d. IDA 静态分析功能强大, 借助路径分析可以更加方便分析反汇编代码。

记录会写入到 AddrFlowEasy.asm 文件中。在执行过程中, 同样会**记录遇到的内存访问**。

2. 脚本参数用法

```
.\RuntimeTrace.py -S 0x00402029 -E 0x0040206A  
-StepIn 0x00402064 -StepIn 0x68B09B26 -MustAddr  
0x68B09B0A -PauseOnce 0x68B09B0A
```

-S

指定分析的起点,-E 指定分析的终点. 也就是单步过程中必须要经过这两处地址.

-StepIn

指定的地址如果是遇到 call 的目标地址, 就执行 step in.

-MustAddr

指定的地址则必须执行到的地址, 也就是说即使程序执行到-E 指定的点, 如果仍有 MustAddr 指定的地址没有达到, 那么就继续执行.

**.\RuntimeTrace.py -S 0x01071AD0 -E 0x01071BC1  
-StartInModules 0x01060000 -EndInModules  
0x01076FF2**

**-S**

指定分析的起点,-E 指定分析的终点. 也就是单步过程中必须要经过这两处地址.

**-StepIn**

指定的地址如果是遇到 call 的目标地址, 那么就会执行 step in.

**-StartInModules**

指定允许记录的模块起始点, **-EndInModules** 指定允许记录的模块终点。如果 step in 和 step out 的地址在指定的模块范围内, 就继续执行。否则会执行 step out 直到单步到允许的地址范围内。

**.\RuntimeTrace.py -S 0x004011A0 -E 0x004012ED  
-StartInModules 0x00400000 -EndInModules  
0x00402FFF -noEnablePrtESP**

**-S**

指定分析的起点,-E 指定分析的终点. 也就是单步过程中必须要经过这两处地址.

**-StepIn**

指定的地址如果是遇到 call 的目标地址, 那么就会执行 step in.

**-StartInModules**

指定允许记录的模块起始点, **-EndInModules** 指定允许记录的模块终点。如果 step in 和 step out 的地址在指定的模块范围内, 就继续执行。否则会执行 step out 直到单步到允许的地址范围内。

**-noEnablePrtESP**

执行反汇编的过程中, 不记录 ESP 的值。

**.\RuntimeTrace.py -S 0x004011A0 -E 0x004012ED  
-StartInModules 0x00400000 -EndInModules  
0x00402FFF -noEnablePrtESP -ModifyCallAddr**

**-S**

指定分析的起点, -E 指定分析的终点. 也就是单步过程中必须要经过这两处地址.

**-StepIn**

指定的地址如果是遇到 call 的目标地址, 那么就会执行 step in.

**-StartInModules**

指定允许记录的模块起始点, -EndInModules 指定允许记录的模块终点. 如果 step in 和 step out 的地址在指定的模块范围内, 就继续执行. 否则会执行 step out 直到单步到允许的地址范围内。

**-noEnablePrtESP**

执行反汇编的过程中, 不记录 ESP 的值。

**-ModifyCallAddr**

将 call 的目标地址修改。比如 call 0x12345678 改成 mov eax,0x12345678 和 call eax.

3. 下面展示的是记录文件 AddrFlowEasy.asm 的部分内容:

```
;esp : 0x0019FD1C
;ebp : 0x0019FE0C
/*0x00411959*/      rep stosd
/*0x0041195B*/      mov ecx, 0x41C00D
/*0x00411960*/      call 0x0041132F
;esp : 0x0019FD18
;/*0x0041132F*/      jmp 0x004119D0
/*0x004119D0*/      push ebp
;esp : 0x0019FD14
/*0x004119D1*/      mov ebp, esp
;ebp : 0x0019FD14
/*0x004119D3*/      sub esp, 0x8
;esp : 0x0019FD0C
/*0x004119D6*/      mov dword ptr ss:[ebp-0x4],
                    ecx
;[ebp-0x4]=[0x0019FD10]=0x0019FD18
;[ebp-0x4]=[0x0019FD10]=0x0041C00D      <-- Modify
/*0x004119D9*/      mov eax, dword ptr
                    ss:[ebp-0x4]
;[ebp-0x4]=[0x0019FD10]=0x0041C00D
/*0x004119DC*/      mov dword ptr ss:[ebp-0x8],
                    eax
;[ebp-0x8]=[0x0019FD0C]=0x00288000
;[ebp-0x8]=[0x0019FD0C]=0x0041C00D      <-- Modify
/*0x004119DF*/      mov ecx, dword ptr
                    ss:[ebp-0x4]
;[ebp-0x4]=[0x0019FD10]=0x0041C00D
/*0x004119E2*/      movzx edx, byte ptr ds:[ecx]
;[ecx]=[0x0041C00D]=0x00000101
```

## 二、IDAAnalyze 获取 IDA 中的反汇编代码

遍历 IDA 所加载模块的所有函数的反汇编代码。按照如下格式保存到了

C:\\DisasmSet 文件中:

0x004125FD	jz	short loc_41260D
0x004125FF	call	ds:IsDebuggerPresent
0x00412605	test	eax, eax
0x00412607	jnz	loc_41270C
0x0041260D	push	104h; unsigned int
0x00412612	lea	eax, [ebp+var_414]
0x00412618	push	eax; wchar_t *
0x00412619	lea	eax, [ebp+var_E38]
0x0041261F	push	eax; int *
0x00412620	push	104h; char
0x00412625	lea	eax, [ebp+var_20C]
0x0041262B	push	eax; wchar_t *
0x0041262C	lea	eax, [esi-5]
0x0041262F	push	eax; unsigned __int8 *

目前这样的输出文件，被后面章节的 BreakpointTool 使用。因为使用 X64 Dbg 的 LyScript 来分析机器码对应的反汇编时，经常会出现错误，分析出的反汇编代码，在 x64dbg 实际的代码段中并没有出现。那么可以借助这里得到的反汇编代码进行比对来进行纠错。

### 三、BreakpointTool 设置断点探查消息处理函数

```
.\python BreakpointTool.py -S 0x400000 -E 0x410000  
-Step 100
```

-S

指定分析的起点,-E 指定分析的终点. 也就是单步过程中必须要经过这两处地址.

-E

指定的地址如果是遇到 call 的目标地址, 就执行 step in.

-Step

指定的地址则必须执行到的地址, 也就是说即使程序执行到-E 指定的点, 如果仍有 MustAddr 指定的地址没有达到, 那么就继续执行.

按照指定的范围, 提取地址和对应的反汇编代码。不过在这个提取过程是需要将 DisasmSet 文件中的代码与 X64 Dbg 提取的反汇编代码进行比对, 只有两者在地址一致且对应的反汇编代码的操作码一致的情况下才作为设置断点的有效代码。