**WinDbg Preview TTD小白入门**

**☆ TTD技术简介**

TTD是"Time Travel Debugging"的缩写，可以理解成轻量级、进程级的VMware 7之前的Record/Replay功能。VMware 7那个功能是系统级、OS级的录制/重放，TTD只针对单个进程。TTD只能对付用户态进程，无法用于内核态调试。

TTD的背后是Nirvana/iDNA技术，Linux上也有类似的技术。

"录制"阶段会生成.run文件，"重放"阶段所有操作都围绕.run文件进行，我称之为"鞭尸"。若录制时长较长，执行到的代码块较复杂，生成的.run文件可能非常之大。

"鞭尸"时，可以对已被覆盖的缓冲区设置数据断点，反向(逆序)执行，定位向缓冲区写入数据的代码逻辑。VMware 7的Record/Replay好像不能反向执行？记不清了。

"数据断点+反向执行"是TTD技术最经典的应用，但是，这是一种弱智的应用方式。在复杂场景中，TTD执行无论顺序、逆序都很耗时，高效作法是用TTD.Memory实现数据断点的目的，初学者切记之。

**☆ TTDTest\_0**

本来我只想演示notepad、Calculator的，后来考虑那样入门对小白太难了，就先演示些更简单的例子吧。

**1) TTDTest\_0.c**

/\*

\* Visual Studio 2019

\*

\* cl TTDTest\_0.c /FeTTDTest\_0.exe /nologo /Os /GS- /guard:cf- /W3 /WX /D "WIN32" /D "NDEBUG" /D "\_CONSOLE" /MD /link /RELEASE

\*/

#include <stdio.h>

static void foo ( int a, int b, int c )

{

int x = a + b;

int y = x / c;

}

int main ( int argc, char \* argv[] )

{

int a = 2;

int b = 1;

int c = 0;

foo( a, b, c );

printf( "ok\n" );

return 0;

}

本例执行时触发除零异常。后面假设没有源码，通过TTD调试了解发生了什么。

**2) TTD调试TTDTest\_0.exe**

开管理员级WinDbg Preview，否则无法使用TTD技术。在管理员级cmd中执行

"X:\Green\windbgx\1.2111.9001.0\DbgX.Shell.exe"

这是假设用绿色版，如果有安装版，在管理员级cmd中执行

windbgx

File

Launch executable (advanced)

Executable

X:\work\MSDN\_46\TTDTest\_0.exe

Start directory

X:\work\MSDN\_46\

Target architecture

Autodetect

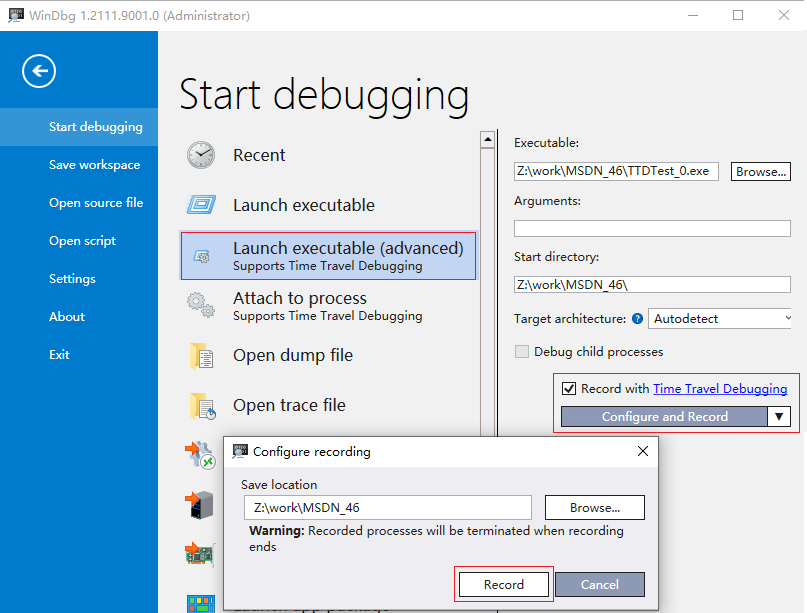
Record with Time Travel Debugging (选中)

Configure and Record

Save location

X:\work\MSDN\_46

Record



这些都是自解释的，不需要写这么细。点击Record之后就开始执行并记录，缺省没有停在ibp(初始化断点)，而是停在ntdll!LdrInitializeThunk，这个点比ibp还要早。g起来，直至触发除零异常，依次生成TTDTest\_001.run、TTDTest\_001.idx。

> r

rax=0000000000000003 rbx=00000236b75a4ac0 rcx=0000000000000003

...

TTDTest\_0+0x1024:

00007ff7`62341024 f77c2430 idiv eax,dword ptr [rsp+30h] ss:00000097`e64ff730=00000000

位于0x97e64ff730的除数为零，idiv指令触发除零异常。

**2.1) 数据断点+反向执行**

这是个简单示例，用IDA静态看两眼就知道root cause。如果是复杂场景，可以对0x97e64ff730设置数据断点，从触发异常的现场反向执行。

有调试符号及源码的情况下，反向执行非常直观。现实中可能没有调试符号及源码，应该以此为前提练习TTD技术。

ba w1 0x97e64ff730;g-

"g-"是"g"的逆操作，反向执行。当数据断点命中时查看附近代码

> r

rax=00007ffe79e607a8 rbx=00000236b75a4ac0 rcx=0000000000000002

rdx=0000000000000001 rsi=0000000000000000 rdi=00000236b75a8be0

rip=00007ff762341005 rsp=00000097e64ff718 rbp=0000000000000000

r8=0000000000000000 r9=00000097e64ff6d8 r10=0000000000000012

...

> u @rip-5 l 2

TTDTest\_0+0x1000:

00007ff7`62341000 4489442418 mov dword ptr [rsp+18h],r8d

00007ff7`62341005 89542410 mov dword ptr [rsp+10h],edx

0x7ff762341000处代码对除数赋零。对于本例，这已经是root cause。

GUI界面上有四种反向执行，可以鼠标操作

Go Back // g-

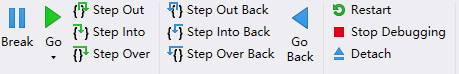
Step Out Back // g-u

Step Into Back // t-

Step Over Back // p-

我是习惯了命令行，所以用"g-"。

要点在于，所有信息都存储在.run文件中，可以不断正向、反向执行，整个过程不怕丢失调试的中间状态，可以让反向执行去触发数据断点，这是梦寐以求的功能。显然，只能以只读模式使用.run文件，不能在调试过程中手工更改寄存器、内存。



**2.2) 用TTD.Memory实现数据断点的目的**

.idx文件是基于.run事后生成的，可以删除并重新生成。有了.idx文件，可以用dx命令查询对地址0x97e64ff730进行写操作的所有代码，效率比"数据断点+反向执行"高。

r $t0=0x97e64ff730;r $t1=8;dx -r2 @$cursession.TTD.Memory(@$t0,@$t0+@$t1,"w").Where(m=>m.Value==0)

这是查询对指定内存写入0的代码

EventType : 0x1

ThreadId : 0x32b4

UniqueThreadId : 0x2

TimeStart : 56:30 [Time Travel]

AccessType : Write

IP : 0x7ff762341000

Address : 0x97e64ff730

Size : 0x4

Value : 0x0

OverwrittenValue : 0x1f

OverwrittenValue是原来的值，Value是现在的值，直接定位到0x7ff762341000。

"u 0x7ff762341000 l 1"只能看代码，看不到上下文，可以用!tt切换到那个时间点

> !ttdext.tt 56:30

Setting position: 56:30

TTDTest\_0+0x1000:

00007ff7`62341000 4489442418 mov dword ptr [rsp+18h],r8d ss:00000097`e64ff730=0000001f

> r

rax=00007ffe79e607a8 rbx=00000236b75a4ac0 rcx=0000000000000002

rdx=0000000000000001 rsi=0000000000000000 rdi=00000236b75a8be0

rip=00007ff762341000 rsp=00000097e64ff718 rbp=0000000000000000

r8=0000000000000000 r9=00000097e64ff6d8 r10=0000000000000012

...

TTDTest\_0+0x1000:

00007ff7`62341000 4489442418 mov dword ptr [rsp+18h],r8d ss:00000097`e64ff730=0000001f

!tt切换过去后就能看到上下文，比如0x97e64ff730处原来的dword是0x1f，在此被赋值零。!tt相当于上帝模式的g，对每个时间点的情况进行验尸，无论正反向。

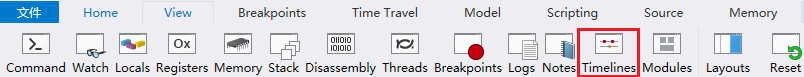
下面这两条命令的效果一样，但后者更高效

ba w1 0x97e64ff730;g-

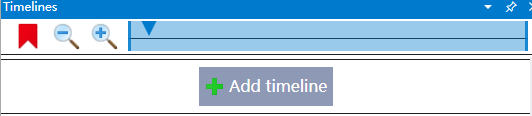
r $t0=0x97e64ff730;r $t1=8;dx -r1 @$cursession.TTD.Memory(@$t0,@$t0+@$t1,"w").Last().TimeStart.SeekTo()

**2.3) GUI中的Timelines**

View->Timelines



让GUI左下角出现Timelines区域，从而直观观察对指定内存的各种访问



Add timeline

Timeline Type

Memory Accesses // 有多种类型，比如异常、断点、函数调用、内存访问

Start Address

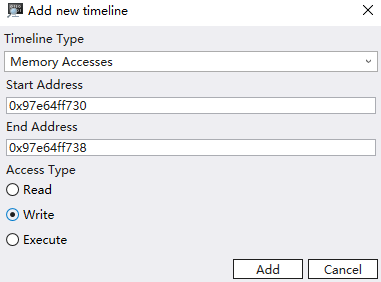
0x97e64ff730 // 欲监控的内存地址

End Address

0x97e64ff738 // 左闭右开区间

Access Type

Write // 只监控写操作



之后Timelines区域新增一条timeline，其上每个褐色棱块对应一次写操作。本例中双击最后一个褐色棱块，相当于

dx @$cursession.TTD.Memory(0x97e64ff730,0x97e64ff738,"w")[0x1]

本例总共只有两次写操作，所以[1]表示最后一次，其他示例未必如此。

