Windows调试原理-part0

40kO / 2019-06-08 08:59:00 / 浏览数 4429 安全技术 二进制安全 顶(0) 踩(0)

前言
在动态分析的过程中，调试器是必不可少的工具。理解调试器的工作原理是有一定好处的，特别是在与反调试的对抗中，如双进程保护就是利用一个进程只能同时被一个调试
接下来会给出一个简易调试器的例子，用于理解调试器的工作机制，麻雀虽小，五脏俱全。

simple example
先写一个拥有最基本的处理调试事件能力的程序，当它发现程序有一个软件断点即0xcc指令时，使EIP+1，并恢复之前的线程

```
#include <Windows.h>
#include <iostream>

BOOL Debug(DWORD pid)
{
   if (pid == 0)
   {
       MessageBox(NULL, "please enter pid", "!!!!", MB_OK);
       return FALSE;
   }
   if (!DebugActiveProcess(pid))
   {
       MessageBox(NULL, "debug process wrong", "!!!!", MB_OK);
       return FALSE;
   }
   while (TRUE)
   {
       DEBUG_EVENT debug_event;
       WaitForDebugEvent(&debug_event, INFINITE);
       switch (debug_event.dwDebugEventCode)
       {
       case EXCEPTION_DEBUG_EVENT:
           if (debug_event.u.Exception.ExceptionRecord.ExceptionCode == EXCEPTION_BREAKPOINT)
           {
               MessageBox(NULL, "find a break point", "!!!!", MB_OK);
               HANDLE hThread = debug_event.u.CreateProcessInfo.hThread;
               CONTEXT context;
               GetThreadContext(hThread, &context);
               context.Eip++;
               SetThreadContext(hThread, &context);
           }
       default:
           break;
       }
       ContinueDebugEvent(pid, debug_event.dwThreadId, DBG_CONTINUE);
   }
   return TRUE;
}

int main()
{
   DWORD pid, tid;
   std::cout << "please enter the process id" << std::endl;
   std::cin >> pid;
   Debug(pid);
   return 0;
}
```

接着我们写一个目标程序，我们用内联汇编写一行int 3指令，即0xcc

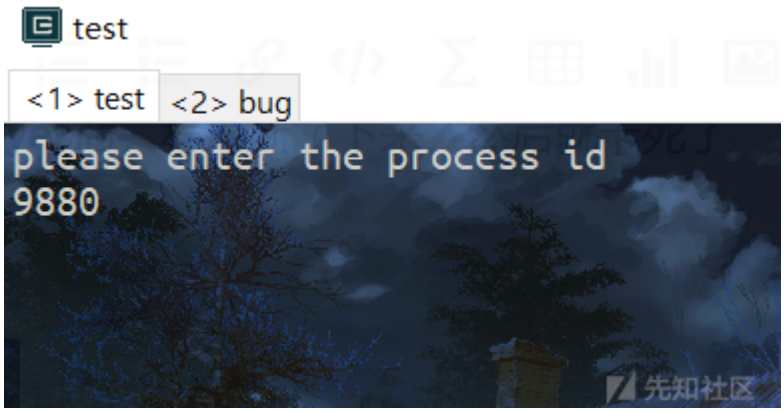
```
#include <iostream>
#include <string>

void bug()
{
   _asm int 3;
```

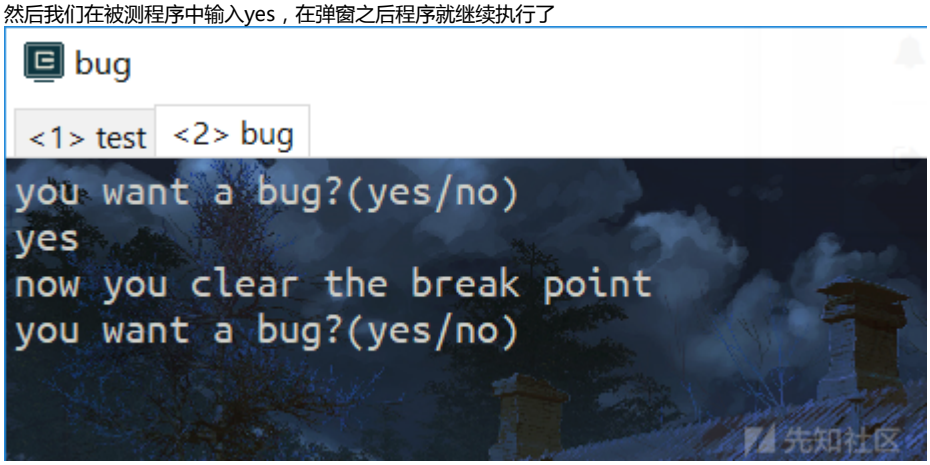
```
    std::cout << "now you clear the break point" << std::endl;
}

int main()
{
    while (true)
    {
        std::cout << "you want a bug?(yes/no)" << std::endl;
        std::string answer;
        std::cin >> answer;
        if (answer == "yes")
            bug();
        else if (answer == "no")
            continue;
    }
    return 0;
}
```

接下来我们进行测试，首先找到被测程序的process id，然后输入到调试器中



然后我们在被测程序中输入yes，在弹窗之后程序就继续执行了



这说明我们的调试器起了作用，因为int 3指令会让程序中断下来，等待一个异常处理，而这里我们的调试器使其步过了这一指令，并恢复其执行。

下面来解释一下调试器部分的代码
DebugActiveProcess这个函数表示附加到目标进程上，并对其进行调试。这之后，调试器与进程间就算是建立起了通信。它的参数就是一个process id。
通信方式是使用WaitForDebugEvent和ContinueDebugEvent两个API，前者用于接收被调试程序触发的调试事件，目标进程触发一个事件后就进行中断，等待调试器处理event。
DEBUG_EVENT结构定义如下

```
typedef struct _DEBUG_EVENT {
    DWORD dwDebugEventCode;
    DWORD dwProcessId;
    DWORD dwThreadId;
    union {
        EXCEPTION_DEBUG_INFO Exception;
        CREATE_THREAD_DEBUG_INFO CreateThread;
        CREATE_PROCESS_DEBUG_INFO CreateProcessInfo;
        EXIT_THREAD_DEBUG_INFO ExitThread;
        EXIT_PROCESS_DEBUG_INFO ExitProcess;
        LOAD_DLL_DEBUG_INFO LoadDll;
        UNLOAD_DLL_DEBUG_INFO UnloadDll;
        OUTPUT_DEBUG_STRING_INFO DebugString;
```

```
        RIP_INFO RipInfo;
    } u;
} DEBUG_EVENT, *LPDEBUG_EVENT;
```

可以看到debug_event结构中有一些基本信息，如进程ID和线程ID。且debug
event种类很多，用一个union来表示各种事件的信息，而具体是哪一种事件，则由dwDebugEventCode字段来决定。这里我们要对0xcc进行处理，那这个debug_event.d

通过这个例子，我们简单的了解了调试器的大体运作方式，它实质功能上是对从进程接收到的各种调试事件进行处理，至于这几个API的native层是如何实现的，以后有机会

上一篇：linux病毒技术之data段感染 下一篇：How Red Teams Byp...

1. 0 条回复
   - 动动手指，沙发就是你的了！