**东方深秘录 反调试分析**

**一、反调试分析**

反调试工作主要由这两个函数完成：（地址为关闭 ASLR 后的地址）

sub\_006585A0: antidebugger1 在程序刚启动时被执行（蓝色、紫色、绿色）

sub\_00659620: antidebugger2 在游戏过程中执行（红色）

这两个函数完成的工作可以由下表表示：

|  |  |
| --- | --- |
| **游戏进程** | **调试器进程** |
| **游戏开始前** | |
| 检查自身进程加载的模块 |  |
| 检查调试器是否存在（不存在） |
| 检查参数（未发现特殊的参数） |
| 创建一块共享内存 |
| 向共享内存写入自己的PID等信息 |
| 用特殊的参数创建一个自己的新进程 |
| 循环检查，等待调试器附加 | 检查自身进程加载的模块 |
| 检查调试器是否存在（不存在） |
| 检查参数（发现特殊的参数） |
| 设置进程权限，禁止调试器附加 |
| 创建一个特殊的“窗口” |
| 打开共享内存 |
| 读取共享内存的中的PID |
| 调试器附加到游戏进程 |
| 发现调试器附加 |  |
| 设置进程权限 |
| 开始游戏 |
| **游戏开始后** | |
| 每隔一段时间做如下事情：  检查调试器创建的“窗口”  检查共享内存的时间戳  检查NtGlobalFlags标记（不存在）  检查调试器是否存在（存在）  枚举窗口检查黑名单 | 每隔一段时间做如下事情：  处理“窗口”消息循环  处理调试消息循环  检查调试器是否存在（不存在）  检查NtGlobalFlags标记（不存在）  向共享内存写入时间戳 |

需要指出的是：

1. 整个反调试是由两个进程协同完成的，若调试器进程僵死则游戏进程很快就会发现。
2. 简单地屏蔽掉 IsDebuggerPresent反而会导致游戏发现自己被调试。
3. PEB NtGlobalFlags标记仅会在被调试程序是由调试器启动时被设置。
4. 在程序反调试开始处，程序会检查自身进程加载的模块，程序会比对模块的路径与系统路径（这一般是C:\Windows\system32），若模块的路径在字典序上小于系统路径则程序会退出（“小于”是因为程序会加载C:\Windows\syswow64下的模块）。

另外还有若干个函数用来反调试，这里就不仔细分析了：

sub\_00657FA0: 逃脱调试器并结束进程

sub\_00658020: 设置进程权限，防止调试器附加

sub\_00659480: 枚举窗口的回调函数，检查敏感窗口

sub\_00658F80: 检查加密存储的窗口黑名单

**二、解决方案**

虽然游戏的反调试看似很复杂，但是却很容易被绕过。由于所有的反调试代码都被写在了那两个函数之中，所以只要屏蔽那两个函数即可。

实际上，那两个函数是有返回值的，然而它们的调用者却根本没有检查返回值。所以我们直接把那两个函数的第一条指令设置为RETN就可以了。

具体来说，就是把th145.exe的偏移量为002579A0和00258A20的两个字节置为C3即可。

**三、一个奇怪的现象**

在研究过程中，我发现了一个奇怪的现象，就是当使用调试器启动程序时，游戏会运行得非常缓慢、卡顿（正常启动后再附加无此现象）。我一开始以为是作者故意为之，后来我发现这是操作系统的一个特性。当调试器存在时，动态内存分配函数HeapAlloc与HeapFree会比平常慢。据我测试大概慢三个数量级左右。可以用以下代码验证：

|  |
| --- |
| #include <stdio.h>  #include <time.h>  #include <windows.h>  #define SZ 10  #define N 100000  **int** main()  {  **static** **void** \*ptr[N];  **int** i;  **clock\_t** t1, t2;  **HANDLE** hHeap;  hHeap = GetProcessHeap();  printf("IsDebuggerPresent=%d\n", IsDebuggerPresent());  t1 = clock();  **for** (i = 0; i < N; i++)  ptr[i] = HeapAlloc(hHeap, 0, SZ);  **for** (i = 0; i < N; i++)  HeapFree(hHeap, 0, ptr[i]);  t2 = clock();  printf("time=%f\n", (**double**)(t2 - t1) / CLOCKS\_PER\_SEC);  system("pause");  **return** 0;  } |

上述代码的输出（示例）：

|  |  |
| --- | --- |
| **正常运行** | **使用调试器加载并运行** |
| IsDebuggerPresent=0  time=0.014000 | IsDebuggerPresent=1  time=13.239000 |

可以得出结论，由于程序大量使用动态内存分配，而使用调试器加载使动态内存分配变慢许多，因此游戏运行变得缓慢。

我目前想到的解决办法是：用高效的内存分配器（比如jemalloc）把原来的内存分配其替换掉；另一种解决方案是清除调试器产生的标记（但是我不知道该怎么做）。

**四、总结**

1. 反调试代码设计得十分巧妙，然而由于把所有反调试代码放在了一起，很容易就被屏蔽掉。
2. 关闭ASLR对动态调试、静态分析很有好处。
3. 使用调试器启动程序会严重影响HeapAlloc与HeapFree的速度。

**五、参考资料**

《反调试技巧总结-原理和实现》

http://www.cnblogs.com/huhu0013/archive/2011/07/05/2098358.html

**六、附录**

本文使用的游戏版本为 ver1.01，原游戏可执行文件的 MD5 校验和如下：

f54bdd766533707cfbc82a1e9c86d24f th145.exe

两个反调试函数的流程图如下：

|  |  |
| --- | --- |
| **sub\_006585A0:**  **antidebugger1** | **sub\_00659620:**  **antidebugger2** |
| antidebugger1.png | antidebugger2.png |