[网络安全自学篇] 九十二.《Windows黑客编程技术详解》之病毒启动技术创建进程API、突破SESSION0隔离、内存加载详解（3）

原创 Eastmount 2020-07-27 17:41:49 5450 收藏 86 版权

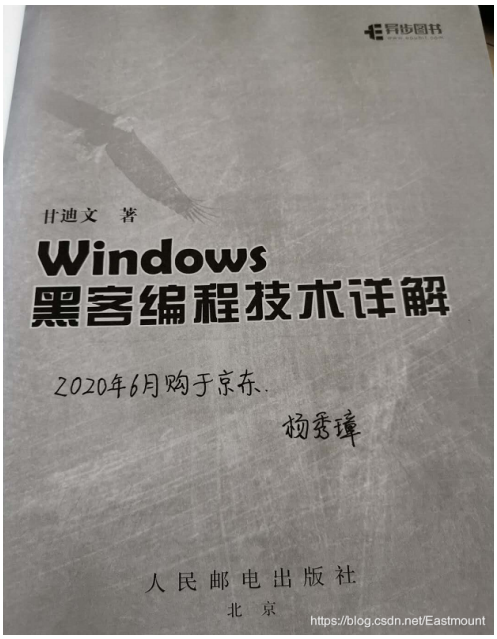
分类专栏： 网络安全自学篇 系统安全与恶意代码识别 Windows黑客编程 文章标签： Windows黑客编程 启动技术 病毒分析 内存加载 安全攻防

这是作者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，希望您喜欢，一起进步。这篇文章将带着大家来学习《Windows黑客编程技术详解》，其作者是甘迪文老师，推荐大家购买来学习。作者将采用实际编程和图文结合的方式进行分享，并且会进一步补充知识点。第三篇文章主要介绍木马病毒启动技术，包括创建进程API、突破SESSION0隔离、内存加载详解，希望对您有所帮助。

病毒木马植入模块成功植入用户计算机后，便会开启攻击模块来对用户计算机数据实施窃取和回传等操作。通常植入和攻击是分开在不同模块之中的，这里的模块指的是DLL、exe或其他加密的PE文件等。只有当前植入模块成功执行后，方可继续执行攻击模块，同时会删除植入模块的数据和文件。

模块化开发的好处不单单是便于开发管理，同时也可以减小因某一模块的失败而导致整个程序暴露的可能性。本文重点介绍病毒木马启动技术，包括：

- 创建进程API：介绍使用WinExec、ShellExecute以及CreateProcess创建进程
- 突破SESSION 0隔离创建进程：主要通过CreateProcessAsUser函数实现用户进程创建
- 内存直接加载运行：模拟PE加载器，直接将DLL和exe等PE文件加载到内存并启动运行



文章目录

作者的github资源：
软件安全：https://github.com/eastmountyxz/Software-Security-Course
其他工具：https://github.com/eastmountyxz/NetworkSecuritySelf-study
Windows-Hacker：https://github.com/eastmountyxz/Windows-Hacker-Exp

> 声明：本人坚决反对利用教学方法进行犯罪的行为，一切犯罪行为必将受到严惩，绿色网络需要我们共同维护，更推荐大家了解它们背后的原理，更好地进行防护。

前文学习：
[网络安全自学篇] 一.入门笔记之看雪Web安全学习及异或解密示例
[网络安全自学篇] 二.Chrome浏览器保留密码功能渗透解析及登录加密入门笔记
[网络安全自学篇] 三.Burp Suite工具安装配置、Proxy基础用法及暴库示例
[网络安全自学篇] 四.实验吧CTF实战之WEB渗透和隐写术解密
[网络安全自学篇] 五.IDA Pro反汇编工具初识及逆向工程解密实战
[网络安全自学篇] 六.OllyDbg动态分析工具基础用法及Crakeme逆向
[网络安全自学篇] 七.快手视频下载之Chrome浏览器Network分析及Python爬虫探讨
[网络安全自学篇] 八.Web漏洞及端口扫描之Nmap、ThreatScan和DirBuster工具
[网络安全自学篇] 九.社会工程学之基础概念、IP获取、IP物理定位、文件属性
[网络安全自学篇] 十.论文之基于机器学习算法的主机恶意代码
[网络安全自学篇] 十一.虚拟机VMware+Kali安装入门及Sqlmap基本用法
[网络安全自学篇] 十二.Wireshark安装入门及抓取网站用户名密码（一）
[网络安全自学篇] 十三.Wireshark抓包原理（ARP劫持、MAC泛洪）及数据流追踪和图像抓取（二）
[网络安全自学篇] 十四.Python攻防之基础常识、正则表达式、Web编程和套接字通信（一）
[网络安全自学篇] 十五.Python攻防之多线程、C段扫描和数据库编程（二）
[网络安全自学篇] 十六.Python攻防之弱口令、自定义字典生成及网站暴库防护
[网络安全自学篇] 十七.Python攻防之构建Web目录扫描器及ip代理池（四）
[网络安全自学篇] 十八.XSS跨站脚本攻击原理及代码攻防演示（一）
[网络安全自学篇] 十九.Powershell基础入门及常见用法（一）
[网络安全自学篇] 二十.Powershell基础入门及常见用法（二）
[网络安全自学篇] 二十一.GeekPwn极客大赛之安全攻防技术总结及ShowTime
[网络安全自学篇] 二十二.Web渗透之网站信息、域名信息、端口信息、敏感信息及指纹信息收集
[网络安全自学篇] 二十三.基于机器学习的恶意请求识别及安全领域中的机器学习
[网络安全自学篇] 二十四.基于机器学习的恶意代码识别及人工智能中的恶意代码检测
[网络安全自学篇] 二十五.Web安全学习路线及木马、病毒和防御初探
[网络安全自学篇] 二十六.Shodan搜索引擎详解及Python命令行调用
[网络安全自学篇] 二十七.Sqlmap基础用法、CTF实战及请求参数设置（一）
[网络安全自学篇] 二十八.文件上传漏洞和Caidao入门及防御原理（一）
[网络安全自学篇] 二十九.文件上传漏洞和IIS6.0解析漏洞及防御原理（二）

[网络安全自学篇] 三十.文件上传漏洞、编辑器漏洞和IIS高版本漏洞及防御（三）
[网络安全自学篇] 三十一.文件上传漏洞之Upload-labs靶场及CTF题目01-10（四）
[网络安全自学篇] 三十二.文件上传漏洞之Upload-labs靶场及CTF题目11-20（五）
[网络安全自学篇] 三十三.文件上传漏洞之绕狗一句话原理和绕过安全狗（六）
[网络安全自学篇] 三十四.Windows系统漏洞之5次Shift漏洞启动计算机
[网络安全自学篇] 三十五.恶意代码攻击溯源及恶意样本分析
[网络安全自学篇] 三十六.WinRAR漏洞复现（CVE-2018-20250）及恶意软件自启动劫持
[网络安全自学篇] 三十七.Web渗透提高班之hack the box在线靶场注册及入门知识（一）
[网络安全自学篇] 三十八.hack the box渗透之BurpSuite和Hydra密码爆破及Python加密Post请求（二）
[网络安全自学篇] 三十九.hack the box渗透之DirBuster扫描路径及Sqlmap高级注入用法（三）
[网络安全自学篇] 四十.phpMyAdmin 4.8.1后台文件包含漏洞复现及详解（CVE-2018-12613）
[网络安全自学篇] 四十一.中间人攻击和ARP欺骗原理详解及漏洞还原
[网络安全自学篇] 四十二.DNS欺骗和钓鱼网站原理详解及漏洞还原
[网络安全自学篇] 四十三.木马原理详解、远程服务器IPC$漏洞及木马植入实验
[网络安全自学篇] 四十四.Windows远程桌面服务漏洞（CVE-2019-0708）复现及详解
[网络安全自学篇] 四十五.病毒详解及批处理病毒制作（自启动、修改密码、定时关机、蓝屏、进程关闭）
[网络安全自学篇] 四十六.微软证书漏洞CVE-2020-0601 (上)Windows验证机制及可执行文件签名复现
[网络安全自学篇] 四十七.微软证书漏洞CVE-2020-0601 (下)Windows证书签名及HTTPS网站劫持
[网络安全自学篇] 四十八.Cracer第八期——(1)安全术语、Web渗透流程、Windows基础、注册表及黑客常用DOS命令
[网络安全自学篇] 四十九.Procmon软件基本用法及文件进程、注册表查看
[网络安全自学篇] 五十.虚拟机基础之安装XP系统、文件共享、网络快照设置及Wireshark抓取BBS密码
[网络安全自学篇] 五十一.恶意样本分析及HGZ木马控制目标服务器
[网络安全自学篇] 五十二.Windows漏洞利用之栈溢出原理和栈保护GS机制
[网络安全自学篇] 五十三.Windows漏洞利用之Metasploit实现栈溢出攻击及反弹shell
[网络安全自学篇] 五十四.Windows漏洞利用之基于SEH异常处理机制的栈溢出攻击及shell提取
[网络安全自学篇] 五十五.Windows漏洞利用之构建ROP链绕过DEP并获取Shell
[网络安全自学篇] 五十六.i春秋老师分享小白渗透之路及Web渗透技术总结
[网络安全自学篇] 五十七.PE文件逆向之什么是数字签名及Signtool签名工具详解（一）
[网络安全自学篇] 五十八.Windows漏洞利用之再看CVE-2019-0708及Metasploit反弹shell
[网络安全自学篇] 五十九.Windows漏洞利用之MS08-067远程代码执行漏洞复现及shell深度提权
[网络安全自学篇] 六十.Cracer第八期——(2)五万字总结Linux基础知识和常用渗透命令
[网络安全自学篇] 六十一.PE文件逆向之数字签名详细解析及Signcode、PEView、010Editor、Asn1View等工具用法（二）
[网络安全自学篇] 六十二.PE文件逆向之PE文件解析、PE编辑工具使用和PE结构修改（三）
[网络安全自学篇] 六十三.hack the box渗透之OpenAdmin题目及蚁剑管理员提权（四）
[网络安全自学篇] 六十四.Windows漏洞利用之SMBv3服务远程代码执行漏洞（CVE-2020-0796）复现及详解
[网络安全自学篇] 六十五.Vulnhub靶机渗透之环境搭建及JIS-CTF入门和蚁剑提权示例（一）
[网络安全自学篇] 六十六.Vulnhub靶机渗透之DC-1提权和Drupal漏洞利用（二）
[网络安全自学篇] 六十七.WannaCry勒索病毒复现及分析（一）Python利用永恒之蓝及Win7勒索加密
[网络安全自学篇] 六十八.WannaCry勒索病毒复现及分析（二）MS17-010利用及病毒解析
[网络安全自学篇] 六十九.宏病毒之入门基础、防御措施、自发邮件及APT28样本分析
[网络安全自学篇] 七十.WannaCry勒索病毒复现及分析（三）蠕虫传播机制分析及IDA和OD逆向
[网络安全自学篇] 七十一.深信服分享之外部威胁防护和勒索病毒对抗
[网络安全自学篇] 七十二.逆向分析之OllyDbg动态调试工具（一）基础入门及TraceMe案例分析
[网络安全自学篇] 七十三.WannaCry勒索病毒复现及分析（四）蠕虫传播机制全网源码详细解读
[网络安全自学篇] 七十四.APT攻击检测溯源与常见APT组织的攻击案例
[网络安全自学篇] 七十五.Vulnhub靶机渗透之bulldog信息收集和nc反弹shell（三）
[网络安全自学篇] 七十六.逆向分析之OllyDbg动态调试工具（二）INT3断点、反调试、硬件断点与内存断点
[网络安全自学篇] 七十七.恶意代码与APT攻击中的武器（强推Seak老师）
[网络安全自学篇] 七十八.XSS跨站脚本攻击案例分享及总结（二）
[网络安全自学篇] 七十九.Windows PE病毒原理、分类及感染方式详解

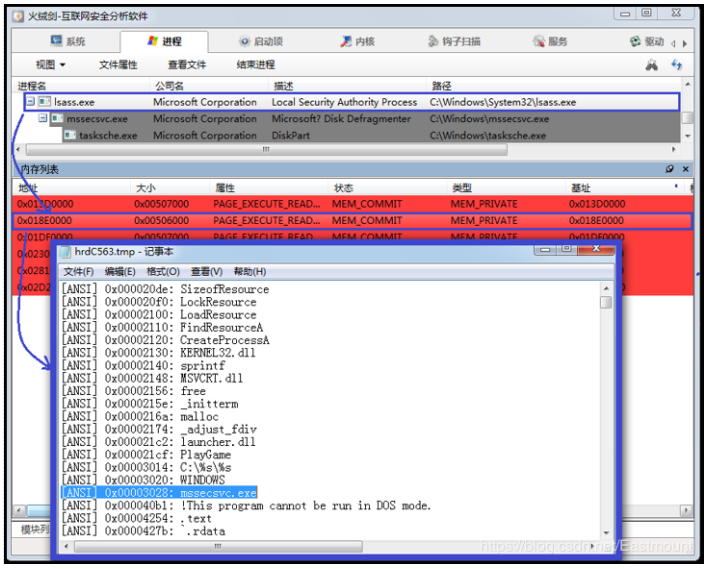
[网络安全自学篇] 八十.WHUCTF之WEB类解题思路WP（代码审计、文件包含、过滤绕过、SQL注入）
[网络安全自学篇] 八十一.WHUCTF之WEB类解题思路WP（文件上传漏洞、冰蝎蚁剑、反序列化phar）
[网络安全自学篇] 八十二.WHUCTF之隐写和逆向类解题思路WP（文字解密、图片解密、佛语解码、冰蝎流量分析、逆向分析）
[网络安全自学篇] 八十三.WHUCTF之CSS注入、越权、csrf-token窃取及XSS总结
[网络安全自学篇] 八十四.《Windows黑客编程技术详解》之VS环境配置、基础知识及DLL延迟加载详解
[网络安全自学篇] 八十五.《Windows黑客编程技术详解》之注入技术详解（全局钩子、远线程钩子、突破Session 0注入、APC注入）
[网络安全自学篇] 八十六.威胁情报分析之Python抓取FreeBuf网站APT文章（上）
[网络安全自学篇] 八十七.恶意代码检测技术详解及总结
[网络安全自学篇] 八十八.基于机器学习的恶意代码检测技术详解
[网络安全自学篇] 八十九.PE文件解析之通过Python获取时间戳判断软件来源地区
[网络安全自学篇] 九十.远控木马详解及APT攻击中的远控
[网络安全自学篇] 九十一.阿里云搭建LNMP环境及实现PHP自定义网站IP访问 (1)

前文欣赏：
[渗透&攻防] 一.从数据库原理学习网络攻防及防止SQL注入
[渗透&攻防] 二.SQL MAP工具从零解读数据库及基础用法
[渗透&攻防] 三.数据库之差异备份及Caidao利器
[渗透&攻防] 四.详解MySQL数据库攻防及Fiddler神器分析数据包

一.创建进程API

在一个进程中创建并启动一个新进程，无论是对于病毒木马程序还是普通的应用程序而言，这都是一个常见的技术。比如我前面文章介绍的WannaCry，蠕虫初始化操作后，它会创建局域网或公网传播的进程，在一系列操作后通过APC注入将生成的dll注入到系统进程lsass.exe，接着释放资源mssecsvc.exe，最后释放勒索程序tasksche.exe。

- 七十三.WannaCry勒索病毒复现及分析（四）蠕虫传播机制全网源码详细解读



启动进程最简单的方法是直接通过调用WIN32 API函数创建新进程，用户层上提供，微软提供了函数来实现创建进程，包括：

- WinExec
- ShellExecute

- CreateProcess
- ...

这些函数除了创建进程外，还能执行CMD命令等功能，接着我们将详细介绍使用WinExec、ShellExecute、CreateProcess函数来创建进程。参考以下文档和这本书籍：

- https://docs.microsoft.com/zh-cn/windows/win32/api/winbase/nf-winbase-winexec
- https://docs.microsoft.com/en-us/previous-versions/aa908775(v=msdn.10)
- WINEXEC, SHELLEXECUTE, CREATEPROCESS - 篱笆博客

1.函数介绍

(1) WinExec函数
运行指定的应用程序。

```
UINT WINAPI WinExec(
  _In_  LPCSTR lpCmdLine,
  _In_  UINT uCmdShow
);
```

参数：

- lpCmdLine：指向一个空结束的字符串，串中包含将要执行的应用程序的命令行（文件名加上可选参数）。
- uCmdShow：定义Windows应用程序的窗口如何显示，SW_HIDE表示隐藏窗口并激活其他窗口；SW_SHOWNORMAL表示激活并显示一个窗口。

返回值：
如果函数调用成功，则返回值大于31；若函数调用失败，则返回值为下列之一：

值	含义
0	系统内存或资源已耗尽
ERROR_BAD_FORMAT	EXE文件无效（非Win32.EXE或.EXE影像错误）
ERROR_FILE_NOT_FOUND	指定的文件未找到
ERROR_PATH_NOT_FOUND	指定的路径未找到

用途：
虽然Microsoft认为WinExec已过时，但是在许多时候，简单的WinExec函数仍是运行新程序的最好方式。简单地传送作为第一个参数的命令行，接着决定如何显示程序（第二个参数）就能实现。通常将其设置为SW_SHOW，也可尝试SW_MINIMIZED或 SW_MAXIMIZED。WinExec不允许用CreateProcess获得的所有选项，而它的确简单。

(2) ShellExecute函数
运行一个外部程序或者打开一个已注册的文件、目录或者打印一个文件等，并对外部程序进行一定程序的控制。

```
HINSTANCE ShellExecute(
    HWND hwnd,
    LPCTSTR lpOperation,
    LPCTSTR lpFile,
    LPCTSTR lpParameters,
    LPCTSTR lpDirectory,
```

```
    INT nShowCmd
);
```

参数：

- hwnd：指向父窗口的窗口句柄，此窗口接收应用程序产生的任何信息框。
- lpOperation：一个空结束的字符串地址，此字符串指定要执行的操作。常用的操作字符串包括：
  – open打开指定项目
  – print打印指定文件
  – edit启动编辑器并编辑文档内容
  – explore搜索指定文件夹，find在指定目录启动搜索
- lpFile：一个空结束的字符串地址，此字符串指定要打开或打印的文件或者是要打开或搜索的文件夹。
- lpParameters：假如参数lpFile指定一个可执行文件，lpParameters则是一个空结束的字符串地址，此字符串指定要传递给应用程序的参数。假如lpFile指定一个文档文件，lpParameters应为NULL。
- lpDirectory：一个空结束的字符串地址，此字符串指定默认目录。
- nShowCmd：假如lpFile指定一个可执行文件，nShowCmd表明应用程序打开时如何显示。假如lpFile指定一个文档文件，nShowCmd应为空。

返回值：

- 若函数调用成功，则返回值大于32，否则为一个小于等于32的错误值。

注意，可以用此函数打开或搜索一个外壳文件夹。打开文件夹或搜索文件夹可用下面的形式：

```
ShellExecute(handle, NULL, path_to_folder, NULL, NULL, SW_SHOWNORMAL);
ShellExecute(handle, "open", path_to_folder, NULL, NULL, SW_SHOWNORMAL);
ShellExecute(handle, "explore", path_to_folder, NULL, NULL, SW_SHOWNORMAL);
```

用途：
ShellExecute命令虽已过时但易于得到。该命令向命令解释程序提出打开、浏览或打印文档或文件夹的请求，虽然可以用ShellExecute运行程序，但通常只发送文档名，而命令解释程序则决定要运行那个程序。另外在打开目录文件夹时，ShellExecute命令非常有用。

(3) CreateProcess函数
创建一个新进程及主线程，新进程在调用进程的安全的上下文中运行。

```
BOOL CreateProcess(
  LPCTSTR lpApplicationName,
  LPTSTR lpCommandLine,
  LPSECURITY_ATTRIBUTES lpProcessAttributes,
  LPSECURITY_ATTRIBUTES lpThreadAttributes,
  BOOL bInheritHandles,
  DWORD dwCreationFlags,
  LPVOID lpEnvironment,
  LPCTSTR lpCurrentDirectory,
  LPSTARTUPINFO lpStartupInfo,
  LPPROCESS_INFORMATION lpProcessInformation
);
```

参数：

- lpApplicationName：指向一个以空结尾的串，指定了要执行的模块.

- lpCommandLine：指向一个以空结尾的串，该串定义了要执行的命令行。
- lpProcessAttributes：指向一个SECURITY_ATTRIBUTES结构，该结构决定了返回的句柄是否可被子进程继承。
- lpThreadAttributes：指向一个SECURITY_ATTRIBUTES结构，该结构决定了返回的句柄是否可被子进程继承。
- bInheritHandles,：表明新进程是否从调用进程继承句柄。
- dwCreationFlags：定义控制优先类和进程创建的附加标志。
- lpEnvironment：指向一个新进程的环境块。
- lpCurrentDirectory：指向一个以空结尾的串，该串定义了子进程的当前驱动器和当前目录。
- lpStartupInfo：指向一个STARTUPINFO结构，该结构定义了新进程的主窗口将如何显示。
- lpProcessInformation：指向PROCESS_INFORMATION结构，该结构接受关于新进程的表示信息。

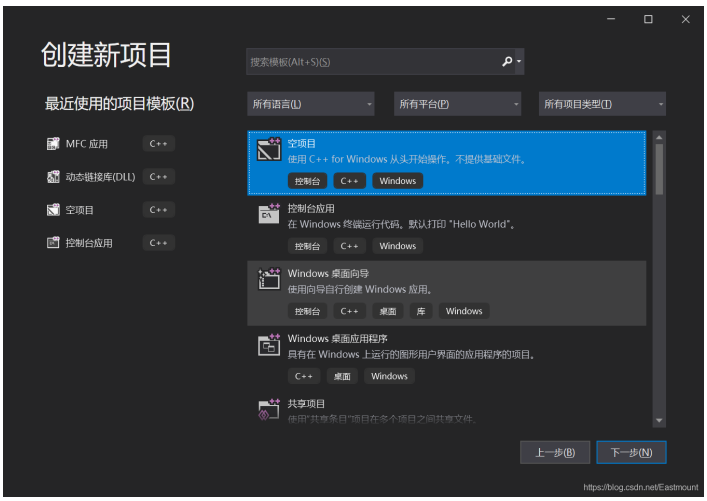
返回值：

- 若函数调用成功，则返回值不为0；若函数调用失败，返回值为0。

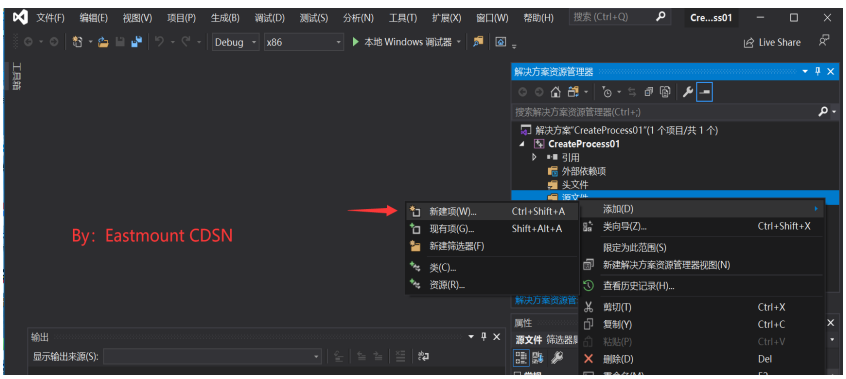
用途：
ShellExecute和WinExec命令用于简单的作业。如果要完全控制一个新进程，就必须调用CreateProcess。

2.编程实现

(1) WinExec
第一步，新建C++空项目，项目名称为“CreateProcess01”。



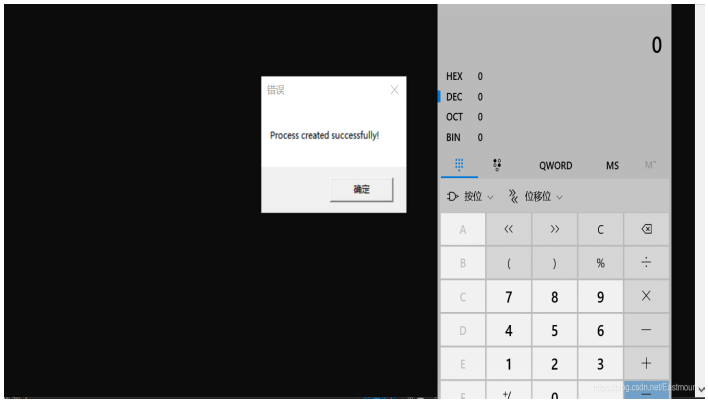
第二步，新建源文件“main.cpp”。



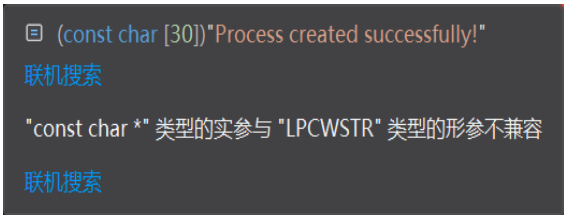
第三步，编写如下代码，直接调用WinExec函数创建进程。

```
#include<windows.h>
#include<string.h>
#include<iostream>
using namespace std;

//创建进程
bool CreateProcess_WinExec(char* name)
{
    UINT uiRet = 0;
    uiRet = WinExec(name, SW_SHOW);
    switch (uiRet)
    {
        case 0:{
            cout << "The system is out of memory or resources." << endl;
            return false;
        }
        case ERROR_BAD_FORMAT:{
            cout << "The.exe file is invalid." << endl;
            return false;
        }
        case ERROR_FILE_NOT_FOUND:{
            cout << "The specified file was not found." << endl;
            return false;
        }
        case ERROR_PATH_NOT_FOUND:{
            cout << "The specified path was not found." << endl;
            return false;
        }
        default:
            break;
        }

    return true;
}

//主函数
int main(int argc, char* argv[])
{
    char name[20];
    strcpy(name, "calc.exe");

    cout << "Opening with WinExec\n" << endl;
    if (CreateProcess_WinExec(name)) {
        MessageBoxA(NULL, "Process created successfully!", NULL, MB_OK);
    }

    return 0;
}
```

在上述代码中，WinExec函数只有两个参数，第一个参数指定程序路径或者CMD命令行，第二个参数指定显示方式。若返回结果大于31，则表示WinExec执行成功。显示结果如下图所示，成功打开计算器程序（calc.exe）。注意，该EXE文件需要提前放置该工程文件夹中。



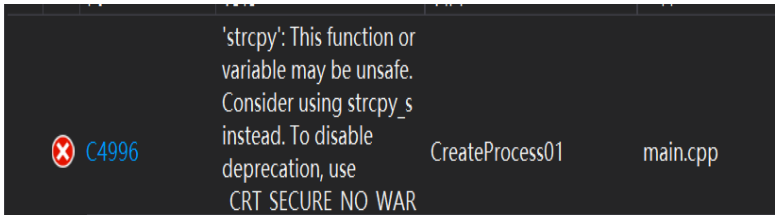
编程过程中会遇到各种错误，请大家一定实际去编写代码，学会谷歌百度独立解决。 比如"const char *"类型的实参与“LPCWSTR”类型的形参不兼容，这是由于字符编码问题引起的，VC6 默认使用的 MBCS(多字节字符集) 编码，而VS2010及高版本VS默认使用Unicode编码。



解决方法有四种：

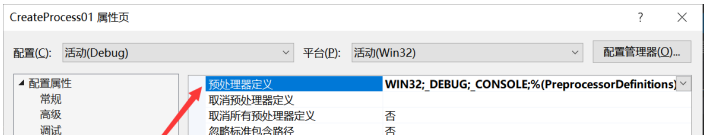
- 配置属性-常规-字符集-Unicode
- MessageBox修改为MessageBoxA函数：MessageBoxA(NULL,“nihao”,“ahfdkj”,MB_OK);
- 强制增加“L”进行转换：MessageBox(NULL,L"nihao",L"ahfdkj",MB_OK);
- 强制增加TEXT函数转换：MessageBox(NULL, TEXT(“nihao”), TEXT(“ahfdkj”), MB_OK);

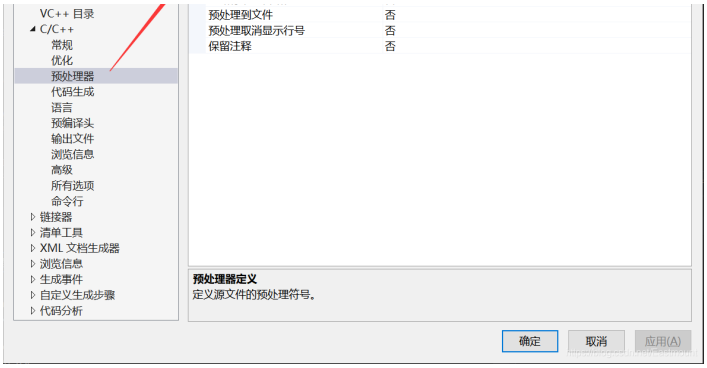
第二个常见错误是“C4996 ‘strcpy’ : This function or variable may be unsafe. Consider using strcpy_s instead. To disable deprecation, use _CRT_SECURE_NO_WARNINGS. See online help for details.”。

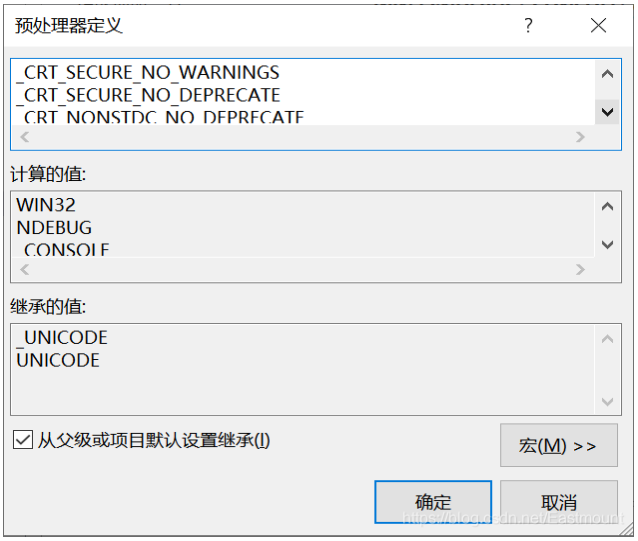


解决方法如下：

- 项目->项目属性->C/C++->预处理器->预处理器定义
- 编辑中添加下面内容
  _CRT_SECURE_NO_WARNINGS
  _CRT_SECURE_NO_DEPRECATE
  _CRT_NONSTDC_NO_DEPRECATE
- Debug和Release都需要添加







---

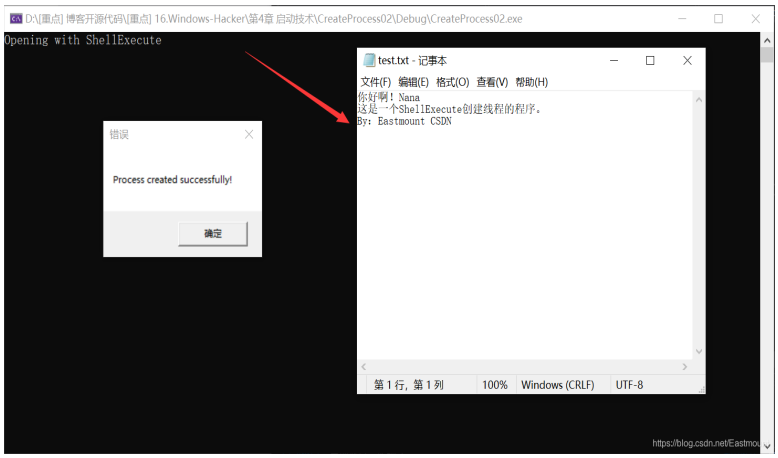
(2) ShellExecute

同样的不步骤我们实现ShellExecute函数，它会打开文本文件，注意TXT文件需要放到同一个目录下。

```
#include<windows.h>
#include<string.h>
#include<iostream>
using namespace std;

//主函数
int main(int argc, char* argv[])
{
    cout << "Opening with ShellExecute\n" << endl;

    HINSTANCE hInstance = 0;
    hInstance = ShellExecute(NULL, TEXT("open"), TEXT("test.txt"), NULL, NULL, SW_SHOW);
    if ((DWORD)hInstance > 32) {
        MessageBoxA(NULL, "Process created successfully!", NULL, MB_OK);
    }
    else {
        MessageBoxA(NULL, "Fail!", NULL, MB_OK);
    }

    return 0;
}
```

输出结果如下图所示，成功打开了TXT文件。ShellExecute函数不仅可以运行EXE文件，也可以运行已经关联的文件，包括打开网页、发送邮件、以默认方式打开文件、打开目录、打印文件等。若返回值大于32，则表示执行成功，否则

执行失败。



---

(3) ShellExcuteEx

同时，作者补充ShellExcuteEx()函数，它相对于ShellExcute()函数而言，采用了结构体传递参数，使得函数调用简洁了很多。函数原型：

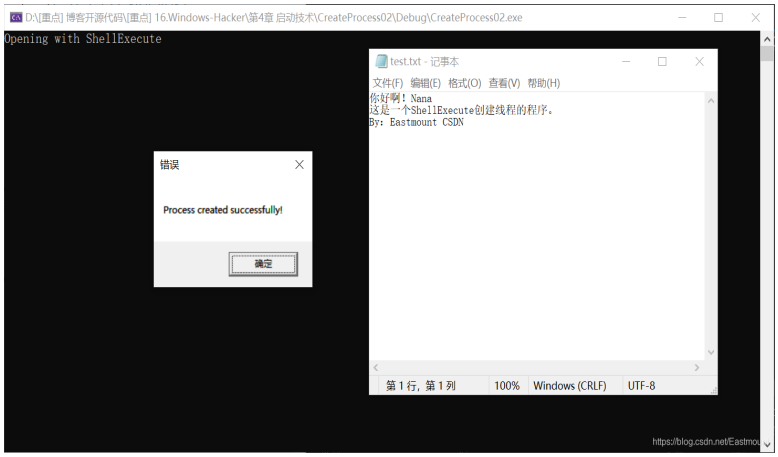
```
BOOL ShellExecuteEx(
  _Inout_ SHELLEXECUTEINFO *pExecInfo
);
```

完整代码如下：

```
#include<windows.h>
#include<string.h>
#include<iostream>
#include <tchar.h>
using namespace std;

//打开TXT文件
bool CreateProcess_ShellExecuteEX()
{
    SHELLEXECUTEINFO ShellInfo;
    memset(&ShellInfo, 0, sizeof(ShellInfo));
    ShellInfo.cbSize = sizeof(ShellInfo);
    ShellInfo.hwnd = NULL;
    ShellInfo.lpVerb = _T("open");
    ShellInfo.lpFile = _T("test.txt");
    ShellInfo.nShow = SW_SHOWNORMAL;
    ShellInfo.fMask = SEE_MASK_NOCLOSEPROCESS;
    BOOL bResult = ShellExecuteEx(&ShellInfo);
    return true;
}

//打开浏览器
bool CreateProcess_ShellExecuteEx_Web()
{
    SHELLEXECUTEINFO ShellInfo;
    memset(&ShellInfo, 0, sizeof(ShellInfo));
    ShellInfo.cbSize = sizeof(ShellInfo);
    ShellInfo.hwnd = NULL;
    ShellInfo.lpVerb = _T("open");
    ShellInfo.lpFile = _T("https://www.baidu.com/");
```

```
    ShellInfo.nShow = SW_SHOWNORMAL;
    ShellInfo.fMask = SEE_MASK_NOCLOSEPROCESS;
    BOOL bResult = ShellExecuteEx(&ShellInfo);
    return true;
}

//主函数
int main(int argc, char* argv[])
{
    cout << "Opening with ShellExecute\n" << endl;

    if (CreateProcess_ShellExecuteEX()) {
        MessageBoxA(NULL, "Process created successfully!", NULL, MB_OK);
    }
    else {
        MessageBoxA(NULL, "Fail!", NULL, MB_OK);
    }

    if (CreateProcess_ShellExecuteEx_Web()) {
        MessageBoxA(NULL, "Process created successfully!", NULL, MB_OK);
    }
    else {
        MessageBoxA(NULL, "Fail!", NULL, MB_OK);
    }
    return 0;
}
```

运行结果如下图所示，成功打开了TXT文件和默认360浏览器。



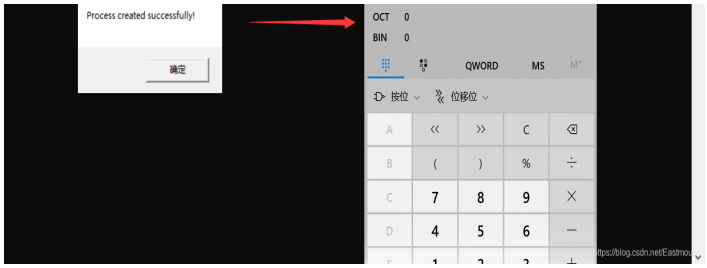
(4) CreateProcess

CreateProcess()函数是当前主流的创建进程的函数，具体代码如下：

```
#include<windows.h>
#include<string.h>
#include<iostream>
#include <tchar.h>
using namespace std;

bool CreateProcess_CreateProcess()
{
    PROCESS_INFORMATION pi;                     //进程信息
    STARTUPINFO si;                             //进程启动信息
    memset(&si, 0, sizeof(STARTUPINFO));
    si.cb = sizeof(si);
    si.wShowWindow = SW_SHOW;
    si.dwFlags = STARTF_USESHOWWINDOW;        //指定wShowWindow成员有效
    BOOL bRet = FALSE;
    bRet = ::CreateProcess(_T("calc.exe"), NULL, NULL, FALSE, NULL, NULL, NULL, NULL, &si, &pi);
    if (bRet) {
        //不适用的句柄最好关掉
        ::CloseHandle(pi.hThread);
        ::CloseHandle(pi.hProcess);
        return TRUE;
    }
    return FALSE;
}

//主函数
int main(int argc, char* argv[])
{
    cout << "Opening with ShellExecute\n" << endl;

    if (CreateProcess_CreateProcess()) {
        MessageBoxA(NULL, "Process created successfully!", NULL, MB_OK);
    }
    else {
        MessageBoxA(NULL, "Fail!", NULL, MB_OK);
    }
    return 0;
}
```

最终运行程序显示如下图所示的效果，与WinExec和ShellExecute函数相比而言，CreateProcess函数的参数更多，使用起来更复杂。我们重点关注5个参数：

- 执行模块名称的参数lpApplicationName
- 执行命令行的参数lpCommandLine
- 控制进程优先级和创建进程标志的参数dwCreationFlags
- 指向STARTUPINFO信息结构的参数lpStartupInfo
- 指向PROCESS_INFORMATION信息结构的参数lpProcessInformation



3.简单小结

该部分主要通过调用WinExec函数、ShellExecute函数，以及CreateProcess函数来创建进程，实现程序的关键是对函数参数的理解。其中，除了进程路径参数较为重要之外，窗口显示方式也值得注意。

对WinExec和ShellExecute函数设置为SW_HIDE方式可隐藏运行程序窗口，并且成功隐藏执行CMD命令的窗口，对于其他程序窗口不能成功隐藏。而CreateProcess函数在指定窗口显示方式的时候，需要在STARTUPINFO结构体中将启用标志设置为STARTF_USESHOWWINDOW，表示wShowWindows成员显示方式有效。然后将wShowWindow置为SW_HIDE隐藏窗口，创建方式为CREATE_NEW_CONSOLE创建一个新控制台，这样可以成功隐藏执行CMD命令行的窗口，而其他程序窗口则不能成功隐藏。

如果在一个进程中想要创建以隐藏方式运行的进程，即隐藏进程窗口，则可以通过SendMessage向窗口发送SW_HIDE隐藏消息，也可以通过ShowWindow函数设置SW_HIDE来使窗口影藏。这两种实现方式的前提是已获取了窗口的句柄。

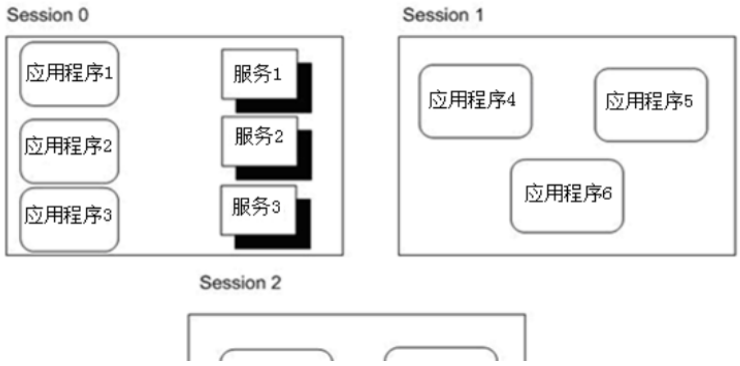
总之，WinExec只用于可执行文件，虽然使用方便，但是函数较老。ShellExcute函数可以通过Windows外壳打开任意文件，非可执行文件自动通过关联程序打开对应的可执行文件，区别不大，不过ShellExcute可以指定运行时的工作路径。WinExec必须得到GetMessage或超时之后才返回，而ShellExcute和CreateProcess都是无需等待直接返回的。

二.突破SESSION 0隔离创建进程

1.SESSION 0隔离

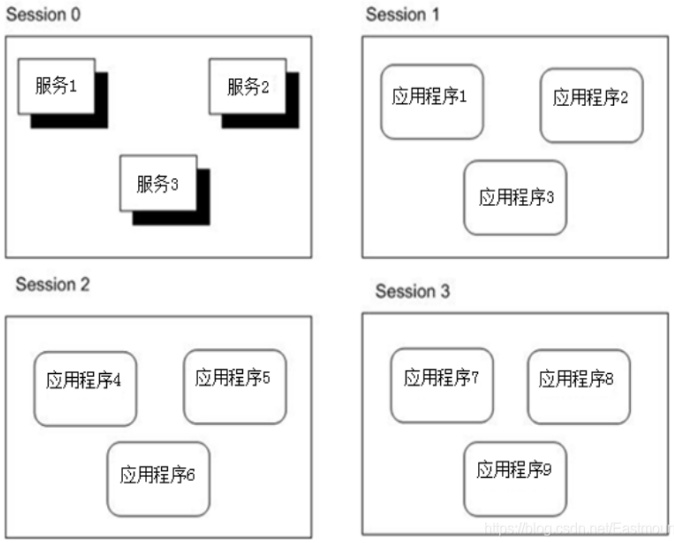
病毒木马通常会把自己注入系统服务进程或是伪装成系统服务进程，并运行在SESSION 0中。 处于SESSION 0中的程序能正常执行普通程序的绝大部分操作，但是个别操作除外。例如处于SESSION 0中的系统服务进程，无法与普通用户进程通信，不能通过Windows消息机制进行通信，更不能创建普通的用户进程。

那么，什么是Session 0隔离呢?
前一篇文章 “注入技术详解”我普及过，在Windows XP、Windows Server 2003，以及更老版本的Windows操作系统中，服务和应用程序使用相同的会话（Session）运行，而这个会话是由第一个登录到控制台的用户启动的。该会话就叫做Session 0，如下图所示，在Windows Vista之前，Session 0不仅包含服务，也包含标准用户应用程序。



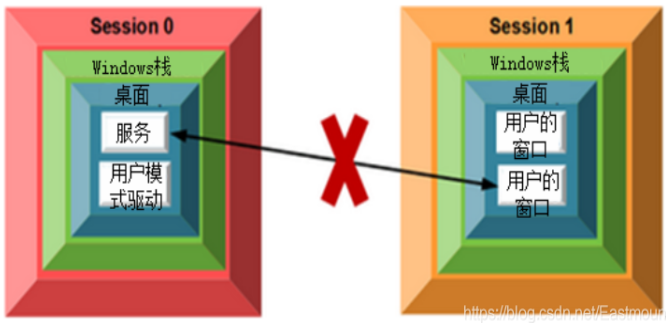
将服务和用户应用程序一起在Session 0中运行会导致安全风险，因为服务会使用提升后的权限运行，而用户应用程序使用用户特权（大部分都是非管理员用户）运行，这会使得恶意软件以某个服务为攻击目标，通过“劫持”该服务，达到提升自己权限级别的目的。

从Windows Vista开始，只有服务可以托管到Session 0中，用户应用程序和服务之间会被隔离，并需要运行在用户登录到系统时创建的后续会话中。例如第一个登录的用户创建 Session 1，第二个登录的用户创建Session 2，以此类推，如下图所示。



使用不同会话运行的实体（应用程序或服务）如果不将自己明确标注为全局命名空间，并提供相应的访问控制设置，将无法互相发送消息，共享UI元素，或共享内核对象。这一过程如下图所示，这就是所谓的Session 0隔离。

- 参考文章：穿透Session 0 隔离（一）- 李老师



虽然Windows 7及以上版本的SESSION 0给服务层和应用层间的通信造成了很大的难度，这并不代表没有办法实现服务层与应用层的通信与交互。微软提供了一系列以WTS（Windows Terminal Service，Windows终端服务）开头的函数，从而可以完成服务层与应用层的交互。接下来，我们将分享突破SESSION 0隔离，在服务程序中创建用户桌面进程。

2.函数介绍

(1) WTSGetActiveConsoleSessionId函数
检索控制台会话的标志服Session Id，控制台会话是当前连接到物理控制台的会话。

```
DWORD WTSGetActiveConsoleSessionId (VOID)
```

返回值：

- 如果函数执行成功，则返回连接到物理控制台的会话标识符
- 如果没有连接到物理控制台的话，如物理控制台会话正在附加或分离，则此函数返回0xFFFFFFFF

(2) WTSQueryUserToken函数
获取由Session Id指定的登录用户的主访问令牌，要想成功调用次功能，则调用应用程序必须在本地系统账户的上下文中运行，并具有SE_TCB_NAME特权。

```
BOOL WTSQueryUserToken(
  ULONG   SessionId,
  PHANDLE phToken
);
```

参数：

- SessionId：远程桌面服务会话标识符，在服务上下文中运行的任何程序都具有一个值为0的会话标识符。
- phToken：如果该功能成功，则会收到一个指向登录用户令牌句柄的指针。注意，必须用CloseHandle函数才能关闭该句柄。

返回值：
如果函数成功，则返回值非零，phToken参数指向用户的主令牌；如果函数失败，则返回值为零。

(3) DuplicateTokenEx函数
创建一个新的访问令牌，它与现有令牌重复，此功能可以创建主令牌或模拟令牌。

```
BOOL WINAPI DuplicateTokenEx(
    _in_     HANDLE   hExistingToken,
    _in_     DWORD    dwDesiredAccess,
    _in_opt_ LPSECURITY_ATTRIBUTES lpTokenAttributes,
    _in_     SECURITY_IMPERSONATION_LEVEL ImpersonationLevel,
    _in_     TOKEN_TYPE   TokenType,
    _out_    PHANDLE  phNewToken
);
```

参数：

- hExistingToken：使用TOKEN_DUPLICATE访问权限打开访问令牌的句柄。
- dwDesiredAccess：指定新令牌的请求访问权限，要想请求对调用者有效的所有访问权限，请指定MAXIMUM_ALLOWED。
- lpTokenAttributes：指向SECURITY_ATTRIBUTES结构的指针，该结构指定新令牌的安全描述符，并确定子进程是否可以继承令牌。
- ImpersonationLevel：指定SECURITY_IMPERSONATION_LEVEL枚举中指示新令牌模拟级别的值。
- TokenType：包括两个值，TokenPrimary表示新令牌可以在CreateProcessAsUser函数中使用的主令牌；TokenImpersonation表示新令牌是一个模拟令牌。

- phNewToken：指向接收新令牌的HANDLE变量的指针。新令牌使用完成后，调用CloseHandle函数来关闭令牌句柄。

返回值：
如果函数成功，则返回值一个非零值；如果函数失败，则返回值为零。

(4) CreateEnvironmentBlock函数
创建指定用户的环境变量，然后可以将此块传递给CreateProcessAsUser函数。

```
BOOL CreateEnvironmentBlock(
  LPVOID *lpEnvironment,
  HANDLE hToken,
  BOOL   bInherit
);
```

参数：

- lpEnvironment：该函数返回时，已接收到指向新环境模块的指针。
- hToken：Logon为用户，从LogonUser函数返回。如果是主令牌，则令牌必须具有TOKEN_QUERY和TOKEN_DUPLICATE访问权限。如果是模拟令牌，则必须具有TOKEN_QUERY权限。如果此参数为NULL，则返回的环境块包含系统变量。
- bInherit：指定是否可以继承当前进程的环境。如果该值为TRUE，则该进程将继承当前进程的环境；如果此值为FALSE，则该进程不会继承当前进程的环境。

返回值：
如果函数成功，则返回TRUE；如果函数失败，则返回FALSE。

(5) CreateProcessAsUser函数
创建一个新进程及主线程，新进程在由指定令牌的用户安全上下文中运行。

```
BOOL WINAPI CreateProcessAsUser(
    _In_opt_    HANDLE                hToken,
    _In_opt_    LPCTSTR               lpApplicationName,
    _Inout_opt_ LPTSTR                lpCommandLine,
    _In_opt_    LPSECURITY_ATTRIBUTES lpProcessAttributes,
    _In_opt_    LPSECURITY_ATTRIBUTES lpThreadAttributes,
    _In_        BOOL                  bInheritHandles,
    _In_        DWORD                 dwCreationFlags,
    _In_opt_    LPVOID                lpEnvironment,
    _In_opt_    LPCTSTR               lpCurrentDirectory,
    _In_        LPSTARTUPINFO         lpStartupInfo,
    _Out_       LPPROCESS_INFORMATION lpProcessInformation
);
```

参数：

- hToken：表示用户主令牌的句柄。
- lpApplicationName：要执行模块的名称。
- lpCommandLine：要执行的命令行。
- lpProcessAttributes：指向SECURITY_ATTRIBUTES结构的指针，该结构指定新进程对象的安全描述符。
- lpThreadAttributes：指向SECURITY_ATTRIBUTES结构的指针，该结构指定新线程对象的安全描述符。

- dwCreationFlags：控制优先级和进程创建的标志。
- lpEnvironment：指向新进程环境块的指针。
- lpCurrentDirectory：指向进程当前目录的完整路径。

返回值：
如果函数成功，则返回TRUE；如果函数失败，则返回FALSE。

3.编程实现

由于SESSION 0的隔离，使得在系统服务进程内不能直接调用CreateProcess等函数创建进程，而只能通过CreateProcessAsUser函数来创建。这样，创建的进程才会显示UI界面，与用户进行交互。

在SESSION 0中创建用户桌面进程具体的实现流程如下。

- 首先，调用WTSGetActiveConsoleSessionId函数来获取当前程序的会话ID，即SESSION Id。调用该函数不需要任何参数，直接返回Session Id，根据Session Id继续调用WTSQueryUserToken函数来检索用户令牌，并获取丢赢得用户令牌句柄。在不需要使用用户令牌句柄时，可以调用CloseHandle函数来释放句柄。
- 其次，使用DuplicateTokenEx函数创建一个新令牌，并复制上面获取的用户令牌。设置新令牌的访问权限为MAMIMUM_ALLOWED，这表示获取所有令牌权限。新访问令牌的模拟级别为SecurityIndentifiction，而且令牌类型为TokenPromary，这表示新令牌是可以在CreateProcessAsUser函数中使用的主令牌。
- 最后，根据新令牌调用CreateEnvironmentBlock函数创建一个环境块，用来传递给CreateProcessAsUser使用。在不需要使用进程环境块时，可以通过调用DestroyEnvironmentBlock函数进行释放。获取环境块后，就可以调用CreateProcessAsUser来创建用户桌面进程。CreateProcessAsUser函数的用法以及参数的含义与CreateProcess函数的用法和参数的含义类似。新令牌句柄作为用户主令牌的句柄，指定创建进程的路径，设置优先级和创建标志，设置STARTUPINFO结构信息，获取PROCESS_INFORMATION结构信息。

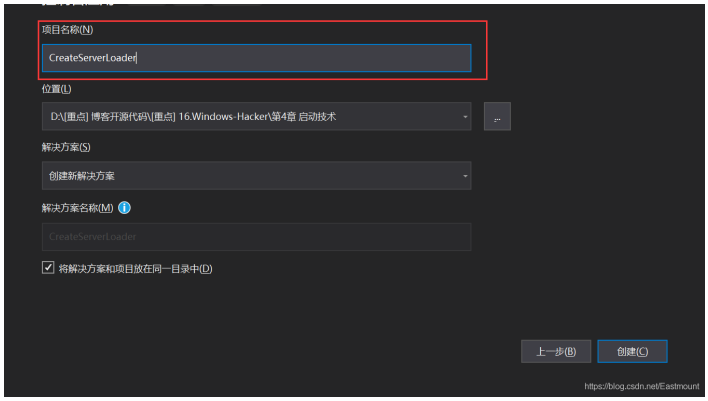
经过上述操作后，就完成了用户桌面进程的创建。但是，上述方法创建的用户桌面进程并没有继承服务的系统权限，只有普通权限。要想创建一个有系统权限的子进程，这可以通过设置进程访问令牌的安全描述符来实现，具体步骤此处不进行详细介绍。

代码实现：
因为程序要实现的是突破SESSION 0会话隔离，在系统服务程序中创建用户桌面进程。程序必须要注册成为一个系统服务进程，这样才处于SESSION 0会话中。服务程序的入口点与普通程序的入口点不同，需要通过调用函数StartServiceCtrlDispatcher来设置服务入口点函数。

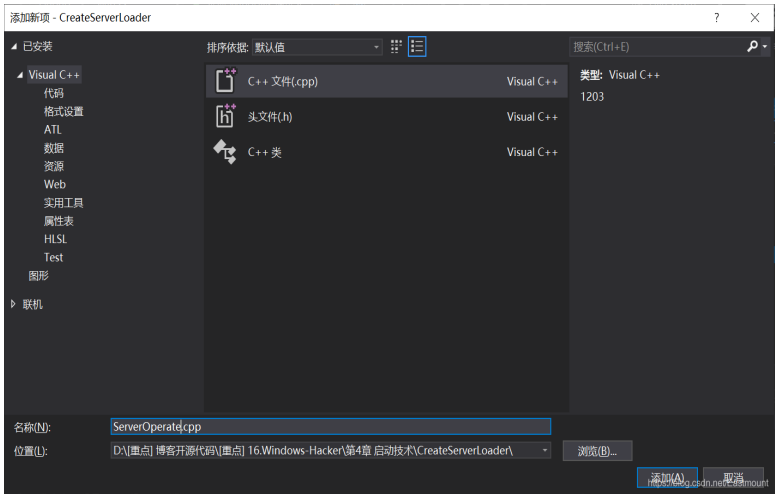
同时，为了将测试程序加载到服务进程中，还需要开发一个服务加载器CreateServerLoader.exe。在 main 函数中，设置服务入口点函数，成为服务程序，并在服务程序中调用上述封装好的函数进行测试。

- 首先，以管理员身份运行服务加载器CreateServerLoader.exe，这样服务加载器会将CreateProcessAsUser_Test.exe程序加载为服务进程，便会执行上述创建用户进程的代码。
- 接着，服务加载器提示创建和启动服务成功后，立即成功显示对话框和启动程序，而且窗口界面也成功显示。

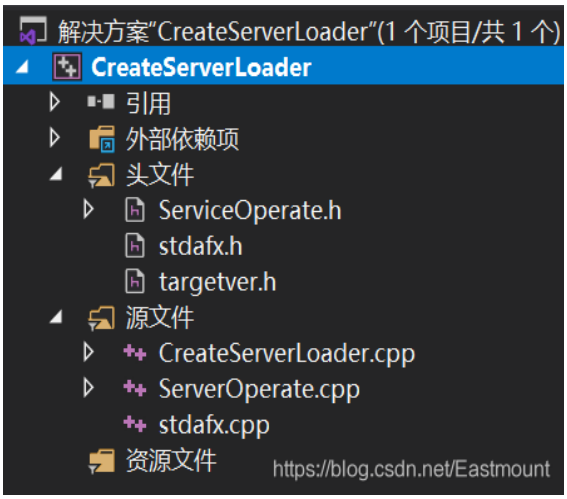
CreateServerLoader程序
第一步，创建控制台应用程序“CreateServerLoader”。



第二步，为程序添加对应的CPP文件和头文件。



添加的文件如下图所示，如果没有自带的stdafx.h文件，均可以自行创建。



第三步，撰写相关代码如下。
ServerOperate.cpp

```
#include "ServiceOperate.h"
#include "stdafx.h"

#undef UNICODE
```

```
void ShowError(char* lpszText)
{
    char szErr[MAX_PATH] = { 0 };
    ::wsprintf(szErr, "%s Error!\nError Code Is:%d\n", lpszText, ::GetLastError());
#ifdef _DEBUG
    ::MessageBox(NULL, szErr, "ERROR", MB_OK | MB_ICONERROR);
#endif
}

// 0 加载服务    1 启动服务    2 停止服务    3 删除服务
BOOL SystemServiceOperate(char* lpszExePath, int iOperateType)
{
    BOOL bRet = TRUE;
    char szName[MAX_PATH] = { 0 };

    ::lstrcpy(szName, lpszExePath);
    // 过滤掉文件目录，获取文件名
    ::PathStripPath(szName);

    SC_HANDLE shOSCM = NULL, shCS = NULL;
    SERVICE_STATUS ss;
    DWORD dwErrorCode = 0;
    BOOL bSuccess = FALSE;
    // 打开服务控制管理器数据库
    shOSCM = ::OpenSCManager(NULL, NULL, SC_MANAGER_ALL_ACCESS);
    if (!shOSCM)
    {
        ShowError("OpenSCManager");
        return FALSE;
    }

    if (0 != iOperateType)
    {
        // 打开一个已经存在的服务
        shCS = OpenService(shOSCM, szName, SERVICE_ALL_ACCESS);
        if (!shCS)
        {
            ShowError("OpenService");
            ::CloseServiceHandle(shOSCM);
            shOSCM = NULL;
            return FALSE;
        }
    }

    switch (iOperateType)
    {
        case 0:
        {
            // 创建服务
            // SERVICE_AUTO_START   随系统自动启动
            // SERVICE_DEMAND_START 手动启动
            shCS = ::CreateService(shOSCM, szName, szName,
                SERVICE_ALL_ACCESS,
                SERVICE_WIN32_OWN_PROCESS | SERVICE_INTERACTIVE_PROCESS,
                SERVICE_AUTO_START,
                SERVICE_ERROR_NORMAL,
                lpszExePath, NULL, NULL, NULL, NULL, NULL);
            if (!shCS)
            {
                ShowError("CreateService");
                bRet = FALSE;
            }
```

```
            break;
        }
        case 1:
        {
            // 启动服务
            if (!::StartService(shCS, 0, NULL))
            {
                ShowError("StartService");
                bRet = FALSE;
            }
            break;
        }
        case 2:
        {
            // 停止服务
            if (!::ControlService(shCS, SERVICE_CONTROL_STOP, &ss))
            {
                ShowError("ControlService");
                bRet = FALSE;
            }
            break;
        }
        case 3:
        {
            // 删除服务
            if (!::DeleteService(shCS))
            {
                ShowError(TEXT("DeleteService"));
                bRet = FALSE;
            }
            break;
        }
        default:
            break;
        }
        // 关闭句柄
        if (shCS)
        {
            ::CloseServiceHandle(shCS);
            shCS = NULL;
        }
        if (shOSCM)
        {
            ::CloseServiceHandle(shOSCM);
            shOSCM = NULL;
        }

    return bRet;
}
```

CreateServerLoader.cpp

```
// ServiceLoader.cpp : 定义控制台应用程序的入口点。
//

#include "stdafx.h"
#include "ServiceOperate.h"

int _tmain(int argc, _TCHAR* argv[])
{
    BOOL bRet = FALSE;
```

```
    char szExePath[] = "CreateProcessAsUser_Test01.exe";

    // 加载服务
    bRet = SystemServiceOperate(szExePath, 0);
    if (bRet)
    {
        printf("INSTALL OK.\n");
    }
    else
    {
        printf("INSTALL ERROR.\n");
    }
    // 启动服务
    bRet = SystemServiceOperate(szExePath, 1);
    if (bRet)
    {
        printf("START OK.\n");
    }
    else
    {
        printf("START ERROR.\n");
    }
    system("pause");

    // 停止服务
    bRet = SystemServiceOperate(szExePath, 2);
    if (bRet)
    {
        printf("STOP OK.\n");
    }
    else
    {
        printf("STOP ERROR.\n");
    }

    // 卸载服务
    bRet = SystemServiceOperate(szExePath, 3);
    if (bRet)
    {
        printf("UNINSTALL OK.\n");
    }
    else
    {
        printf("UNINSTALL ERROR.\n");
    }

    return 0;
}
```

ServiceOperate.h

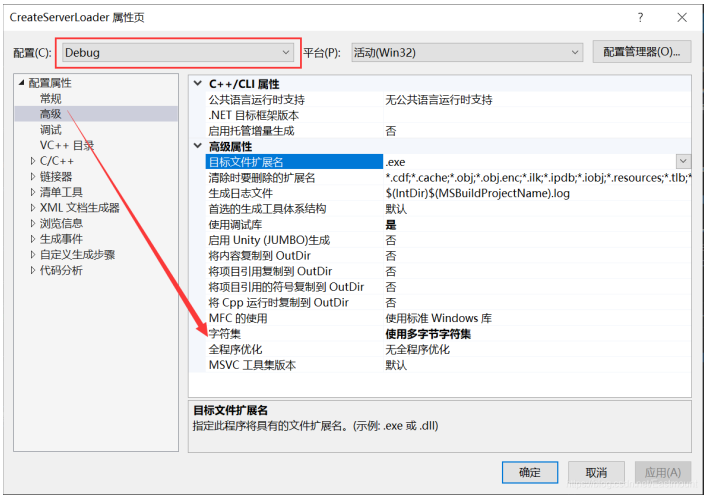
```
#ifndef _SERVICE_OPERATE_H_
#define _SERVICE_OPERATE_H_

#include <Windows.h>
#include <Shlwapi.h>
#pragma comment(lib, "Shlwapi.lib")

// 0 加载服务    1 启动服务    2 停止服务    3 删除服务
BOOL SystemServiceOperate(char* lpszDriverPath, int iOperateType);
```

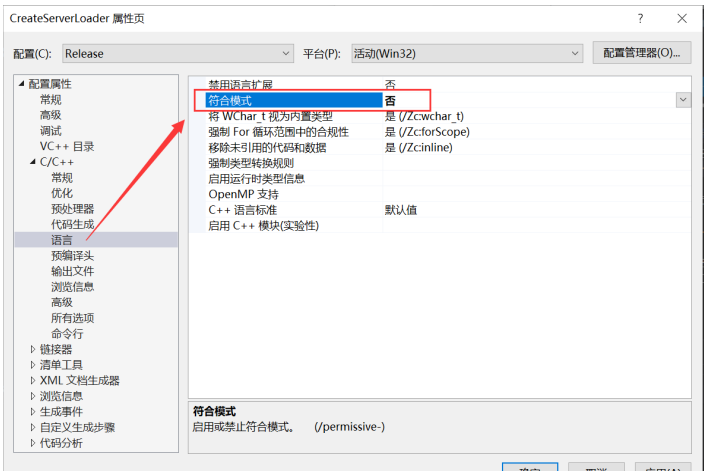
```
#endif
```

> 推荐大家从作者Github下载完整代码或私信我发给你，更推荐大家手动编写，理解过程。
>
> - https://github.com/eastmountyxz/Windows-Hacker-Exp

第四步，如果在撰写代码过程中出现错误，请学会百度解决。如"const char *"类型的实参与“LPCWSTR”类型的形参不兼容。

- 在VS中依次点击“配置属性->高级->字符集->使用多字节字符集”。
- 在VS中依次点击“项目->属性->C/C++->语言->符合模式”，将原来的“是”改为“否”即可。
- 单击确定即可，再次编译就不会再出现此类错误提示了，注意Debug和Release模式要选择好，建议两个都进行设置。

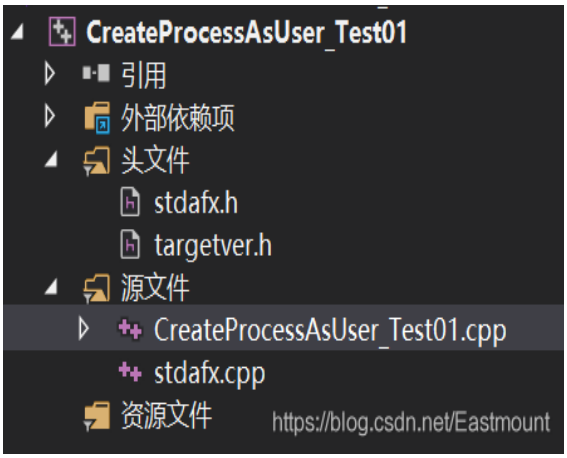




CreateProcessAsUser_Test01程序

第一步，创建控制台应用程序“CreateProcessAsUser_Test01”。

第二步，为程序添加对应的CPP文件和头文件。添加的文件如下图所示，如果没有自带的stdafx.h文件，均可以自行创建。



第三步，撰写相关代码如下。

CreateProcessAsUser_Test01.cpp

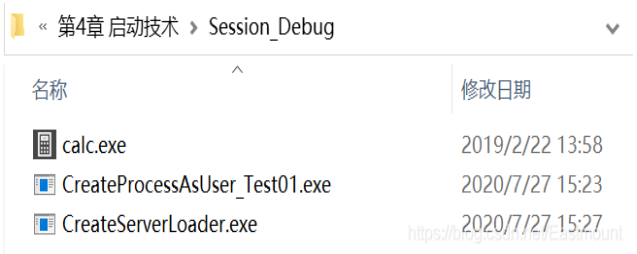
```
// CreateProcessAsUser_Test.cpp : 定义控制台应用程序的入口点。
//

#include "stdafx.h"
#include <Windows.h>

#include <UserEnv.h>
#include <WtsApi32.h>
#pragma comment(lib, "UserEnv.lib")
#pragma comment(lib, "WtsApi32.lib")


// 服务入口函数以及处理回调函数
void __stdcall ServiceMain(DWORD dwArgc, char* lpszArgv);
void __stdcall ServiceCtrlHandle(DWORD dwOperateCode);
void DoTask();
```

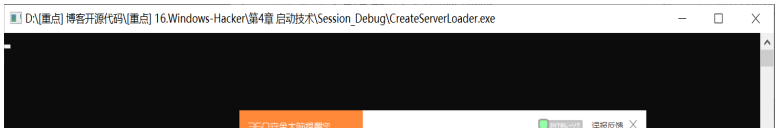
```
// 显示消息对话框
void ShowMessage(TCHAR* lpszMessage, TCHAR* lpszTitle);
// 创建用户进程
BOOL CreateUserProcess(char* lpszFileName);

// 全局变量
char g_szServiceName[MAX_PATH] = "CreateProcessAsUser_Test01.exe";    // 服务名称
SERVICE_STATUS g_ServiceStatus = { 0 };
SERVICE_STATUS_HANDLE g_ServiceStatusHandle = { 0 };


int _tmain(int argc, _TCHAR* argv[])
{
    // 注册服务入口函数
    SERVICE_TABLE_ENTRY stDispatchTable[] = { { g_szServiceName, (LPSERVICE_MAIN_FUNCTION)ServiceMain }, {
NULL, NULL } };
    ::StartServiceCtrlDispatcher(stDispatchTable);

    return 0;
}

void __stdcall ServiceMain(DWORD dwArgc, char* lpszArgv)
{
    g_ServiceStatus.dwServiceType = SERVICE_WIN32;
    g_ServiceStatus.dwCurrentState = SERVICE_START_PENDING;
    g_ServiceStatus.dwControlsAccepted = SERVICE_ACCEPT_STOP;
    g_ServiceStatus.dwWin32ExitCode = 0;
    g_ServiceStatus.dwServiceSpecificExitCode = 0;
    g_ServiceStatus.dwCheckPoint = 0;
    g_ServiceStatus.dwWaitHint = 0;

    g_ServiceStatusHandle = ::RegisterServiceCtrlHandler(g_szServiceName, ServiceCtrlHandle);

    g_ServiceStatus.dwCurrentState = SERVICE_RUNNING;
    g_ServiceStatus.dwCheckPoint = 0;
    g_ServiceStatus.dwWaitHint = 0;
    ::SetServiceStatus(g_ServiceStatusHandle, &g_ServiceStatus);

    // 自己程序实现部分代码放在这里
    DoTask();
}

void __stdcall ServiceCtrlHandle(DWORD dwOperateCode)
{
    switch (dwOperateCode)
    {
        case SERVICE_CONTROL_PAUSE:
        {
            // 暂停
            g_ServiceStatus.dwCurrentState = SERVICE_PAUSED;
            break;
        }
        case SERVICE_CONTROL_CONTINUE:
        {
            // 继续
            g_ServiceStatus.dwCurrentState = SERVICE_RUNNING;
            break;
        }
        case SERVICE_CONTROL_STOP:
        {
            // 停止
            g_ServiceStatus.dwWin32ExitCode = 0;
```

```
            g_ServiceStatus.dwCurrentState = SERVICE_STOPPED;
            g_ServiceStatus.dwCheckPoint = 0;
            g_ServiceStatus.dwWaitHint = 0;
            ::SetServiceStatus(g_ServiceStatusHandle, &g_ServiceStatus);
            break;
        }
        case SERVICE_CONTROL_INTERROGATE:
        {
            // 询问
            break;
        }
        default:
            break;
        }
}

void DoTask()
{
    // 自己程序实现部分代码放在这里
    // 显示对话框
    ShowMessage("Hi Eastmount\nThis is From Session 0 Service!\n", "HELLO");
    // 创建用户桌面进程
    CreateUserProcess("calc.exe");
}

void ShowMessage(TCHAR* lpszMessage, TCHAR* lpszTitle)
{
    // 获取当前的Session ID
    DWORD dwSessionId = ::WTSGetActiveConsoleSessionId();
    // 显示消息对话框
    DWORD dwResponse = 0;
    ::WTSSendMessage(WTS_CURRENT_SERVER_HANDLE, dwSessionId,
        lpszTitle, (1 + ::lstrlen(lpszTitle)),
        lpszMessage, (1 + ::lstrlen(lpszMessage)),
        0, 0, &dwResponse, FALSE);
}

// 突破SESSION 0隔离创建用户进程
BOOL CreateUserProcess(char* lpszFileName)
{
    BOOL bRet = TRUE;
    DWORD dwSessionID = 0;
    HANDLE hToken = NULL;
    HANDLE hDuplicatedToken = NULL;
    LPVOID lpEnvironment = NULL;
    STARTUPINFO si = { 0 };
    PROCESS_INFORMATION pi = { 0 };
    si.cb = sizeof(si);

    do
    {
        // 获得当前Session ID
        dwSessionID = ::WTSGetActiveConsoleSessionId();

        // 获得当前Session的用户令牌
        if (FALSE == ::WTSQueryUserToken(dwSessionID, &hToken))
        {
            ShowMessage("WTSQueryUserToken", "ERROR");
            bRet = FALSE;
            break;
        }
```

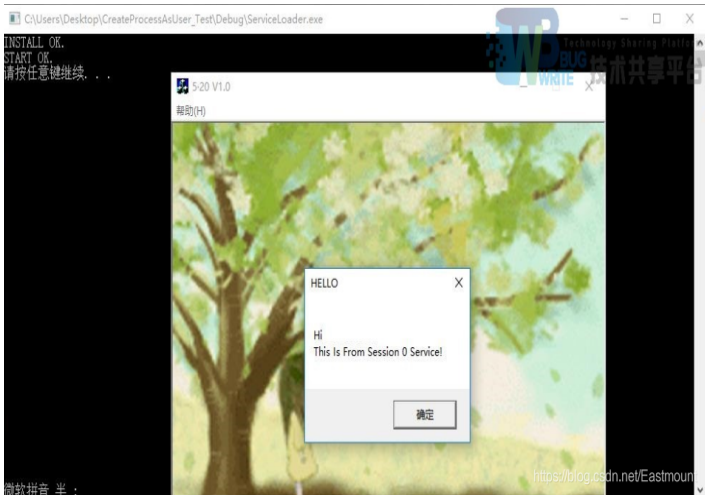
```
        // 复制令牌
        if (FALSE == ::DuplicateTokenEx(hToken, MAXIMUM_ALLOWED, NULL,
            SecurityIdentification, TokenPrimary, &hDuplicatedToken))
        {
            ShowMessage("DuplicateTokenEx", "ERROR");
            bRet = FALSE;
            break;
        }

        // 创建用户Session环境
        if (FALSE == ::CreateEnvironmentBlock(&lpEnvironment,
            hDuplicatedToken, FALSE))
        {
            ShowMessage("CreateEnvironmentBlock", "ERROR");
            bRet = FALSE;
            break;
        }

        // 在复制的用户Session下执行应用程序，创建进程
        if (FALSE == ::CreateProcessAsUser(hDuplicatedToken,
            lpszFileName, NULL, NULL, NULL, FALSE,
            NORMAL_PRIORITY_CLASS | CREATE_NEW_CONSOLE | CREATE_UNICODE_ENVIRONMENT,
            lpEnvironment, NULL, &si, &pi))
        {
            ShowMessage("CreateProcessAsUser", "ERROR");
            bRet = FALSE;
            break;
        }
    } while (FALSE);

    // 关闭句柄，释放资源
    if (lpEnvironment)
    {
        ::DestroyEnvironmentBlock(lpEnvironment);
    }
    if (hDuplicatedToken)
    {
        ::CloseHandle(hDuplicatedToken);
    }
    if (hToken)
    {
        ::CloseHandle(hToken);
    }
    return bRet;
}
```

核心代码CreateUserProcess函数则为突破SESSION 0隔离创建用户进程。然后将运行的EXE文件放置在同一个文件夹下，接着以管理员身份运行“CreateServerLoader.exe”即可。

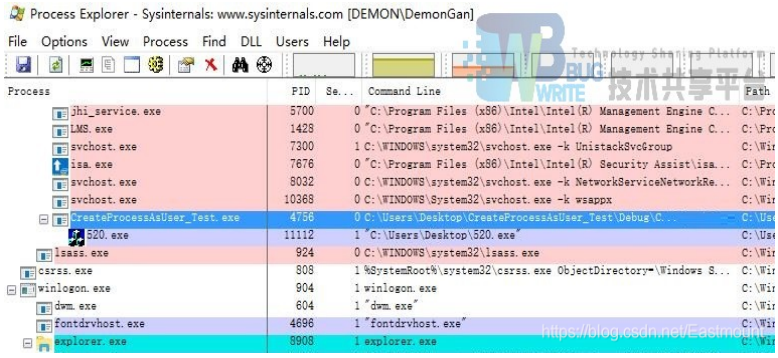




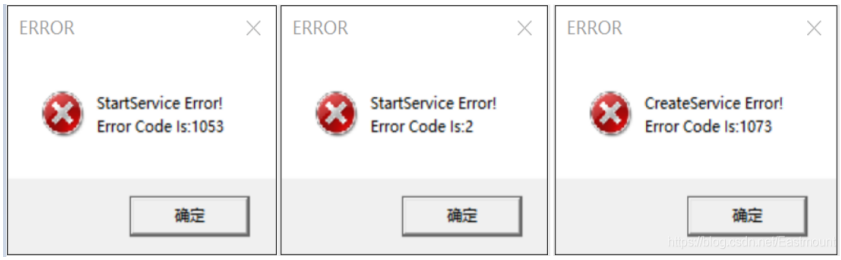
服务加载器CreateServerLoader.exe会将CreateProcessAsUser_Test01.exe程序加载为服务进程，从而执行创建用户进程的代码。服务加载器提示创建和启动服务成功后，立即成功显示对话框和启动程序，而且窗口界面也成功显示。



然后，使用进程查看器ProcessExplorer.exe查看CreateProcessAsUser_Test.exe进程以及520.exe进程的SESSION值，如下图所示，CreateProcessAsUser_Test.exe进程处于SESSION 0，而520.exe处于SESSION 1。



PS：然而，作者运行时遇到各种错误，如“StartService fails with Error Code is 1053”等，我是真的菜！



最后简单总结下，突破SESSION 0会话隔离创建用户进程，要求程序处于SESSION 0会发才会有效。创建服务程序，需要在main中设置服务程序入口点函数，这样才能成功为程序创建系统服务。该程序的实现关键是调用

CreateProcessAsUser函数。需要程序创建并复制一个新的访问令牌，并获取访问令牌的进程环境块信息。

同时，本文介绍的方法并没有对进程访问令牌进行设置，所以导致创建出来的用户桌面进程是用户默认权限而已，并没有继承SYSTEM权限。大家也可以通过挂钩CreateProcessAsUser函数监控进程创建。

三.内存直接加载运行

很多病毒木马都具有模拟PE加载器的功能，它们把DLL或者exe等PE文件从内存中直接加载到病毒木马的内存中去执行，不需要通过LoadLibrary等现成的API函数去操作，以此躲过杀毒软件的拦截检测。

这种技术当然有积极的一面。假如程序需要动态调用DLL文件，内存加载运行技术可以把这些DLL作为资源插入到自己的程序中。此时直接在内存中加载运行即可，不需要再将DLL释放到本地。

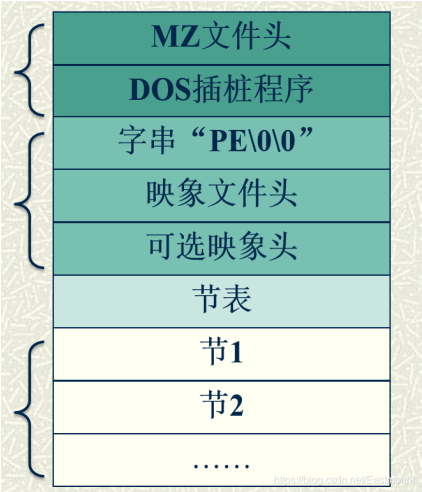
本部分针对DLL和exe这两种PE文件进行介绍，分别剖析如何直接从内存中加载运行。这两种文件具体的实现原理相同，只需掌握其中一种，另一种也就容易掌握了。

1.实现原理

要想完全理解透彻内存直接加载运行技术，需要对PE文件结构有比较详细的了解，至少要了解PE格式的导入表、导出表以及重定位表的具体操作过程。因为内存直接加载运行技术的核心就是模拟PE加载器加载PE文件的过程，也就是对导入表、导出表以及重定位表的操作过程。

作者之前的博客详细讲解过PE文件的结构：

- [网络安全自学篇] 六十二.PE文件逆向之PE文件解析、PE编辑工具使用和PE结构修改（三）



那么程序需要进行哪些操作便可以直接从内存中加载运行DLL或是exe文件呢？以加载DLL为例介绍。

- 首先要把DLL文件按照映像对齐大小映射到内存中，切不可直接将DLL文件数据存储到内存中。因为根据PE结构的基础知识可知，PE文件有两个对齐字段，一个是映像对齐大小SectionAlignment，另一个是文件对齐大小FileAlignment。其中，映像对齐大小是PE文件加载到内存中所用的对齐大小，而文件对齐大小是PE文件存储在本地磁盘所用的对齐大小。一般文件对齐大小会比映像对齐大小要小，这样文件会变小，以此节省磁盘空间。
- 然而，成功映射内存数据之后，在DLL程序中会存在**硬编码数据**，硬编码都是以默认的加载基址作为基址来计算的。由于DLL可以任意加载到其他进程空间中，所以DLL的加载基址并非固定不变。当改变加载基址的时候，硬编码也要随之改变，这样DLL程序才会计算正确。但是，如何才能知道需要修改哪些硬编码呢？换句话说，如何知道硬编码的位置？答案就藏在PE结构的**重定位表**中，重定位表记录的就是程序中所有需要修改的硬编码的相对偏移位置。

- 根据重定位表修改硬编码数据后，这只是完成了一半的工作。DLL作为一个程序，自然也会调用其他库函数，例如MessageBox。那么DLL如何知道MessageBox函数的地址呢？它只有获取正确的调用函数地址后，方可正确调用函数。PE结构使用导入表来记录PE程序中所有引用的函数及其函数地址。
- 在DLL映射到内存之后，需要根据导入表中的导入模块和函数名称来获取调用函数的地址。若想从导入模块中获取导出函数的地址，最简单的方式是通过GetProcAddress函数来获取。但是为了避免调用敏感的WIN32 API函数而被杀软拦截检测，本书采用直接遍历PE结构导出表的方式来获取导出函数地址，这要求读者熟悉导出表的具体操作原理。

完成上述操作之后，DLL加载工作才算完成，接下来便是获取入口地址并跳转执行以便完成启动。具体的实现流程总结如下：

- 首先，在DLL文件中，根据PE结构获取其加载映像的大小SizeOfImage，并根据SizeOfImage在自己的程序中申请可读、可写、可执行的内存，那么这块内存的首地址就是DLL的加载基址。
- 其次，根据DLL中的PE结构获取其映像对齐大小SectionAlignment，然后把DLL文件数据按照SectionAlignment复制到上述申请的可读、可写、可执行的内存中。
- 接下来，根据PE结构的重定位表，重新对重定位表进行修正。然后，根据PE结构的导入表，加载所需的DLL，并获取导入函数的地址并写入导入表中。
- 接着，修改DLL的加载基址ImageBase。最后，根据PE结构获取DLL的入口地址，然后构造并调用DllMain函数，实现DLL加载。

而exe文件相对于DLL文件实现原理唯一的区别就在于构造入口函数的差别，exe不需要构造DllMain函数，而是根据PE结构获取exe的入口地址偏移AddressOfEntryPoint并计算出入口地址，然后直接跳转到入口地址处执行即可。

要特别注意的是，对于exe文件来说，重定位表不是必需的，即使没有重定位表，exe也可正常运行。因为对于exe进程来说，进程最早加载的模块是exe模块，所以它可以按照默认的加载基址加载到内存。对于那些没有重定位表的程序，只能把它加载到默认的加载基址上。如果默认加载基址已被占用，则直接内存加载运行会失败。

2.编程实现

核心代码如下，推荐大家去github下载代码阅读。推荐两篇文章：

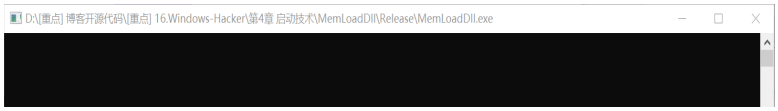
- DLL内存加载 - 墨鱼菜鸡
- 内存直接加载运行DLL文件 - 自己的小白

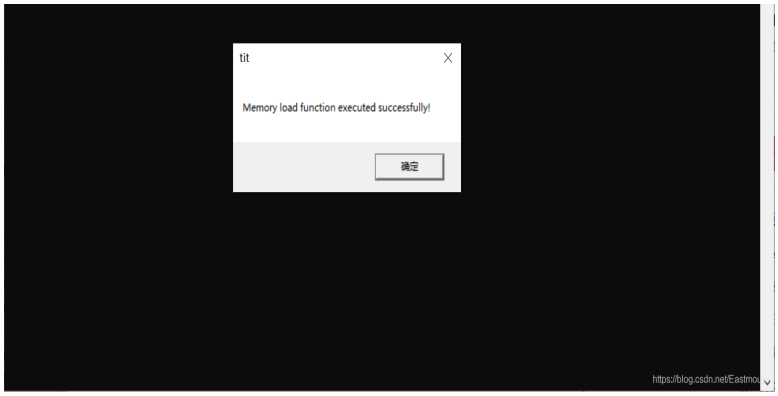
```
// 模拟LoadLibrary加载内存DLL文件到进程中
// lpData: 内存DLL文件数据的基址
// dwSize: 内存DLL文件的内存大小
// 返回值: 内存DLL加载到进程的加载基址
LPVOID MmLoadLibrary(LPVOID lpData, DWORD dwSize)
{
    LPVOID lpBaseAddress = NULL;

    // 获取镜像大小
    DWORD dwSizeOfImage = GetSizeOfImage(lpData);

    // 在进程中开辟一个可读、可写、可执行的内存块
    lpBaseAddress = ::VirtualAlloc(NULL, dwSizeOfImage, MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READWRITE);
    if (NULL == lpBaseAddress)
    {
```

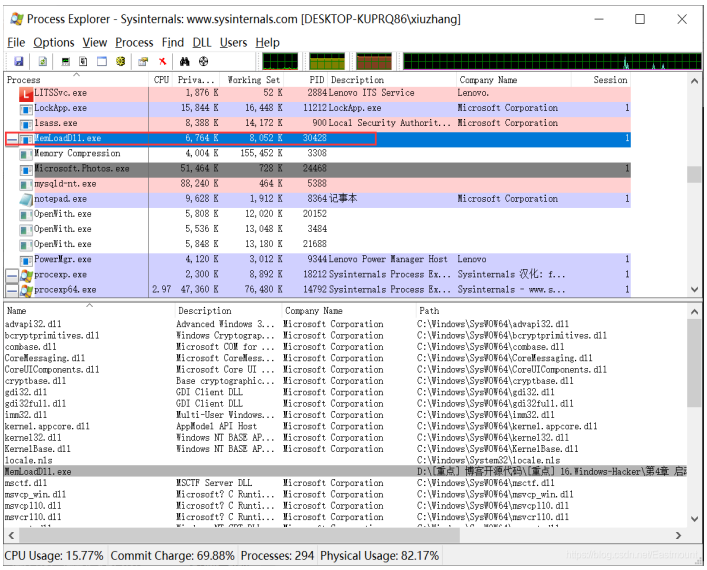
```
            ShowError("VirtualAlloc");
            return NULL;
        }
        ::RtlZeroMemory(lpBaseAddress, dwSizeOfImage);

        // 将内存DLL数据按SectionAlignment大小对齐映射到进程内存中
        if (FALSE == MmMapFile(lpData, lpBaseAddress))
        {
            ShowError("MmMapFile");
            return NULL;
        }

        // 修改PE文件重定位表信息
        if(FALSE == DoRelocationTable(lpBaseAddress))
        {
            ShowError("DoRelocationTable");
            return NULL;
        }

        // 填写PE文件导入表信息
        if (FALSE == DoImportTable(lpBaseAddress))
        {
            ShowError("DoImportTable");
            return NULL;
        }

        //修改页属性。应该根据每个页的属性单独设置其对应内存页的属性。
        //统一设置成一个属性PAGE_EXECUTE_READWRITE
        DWORD dwOldProtect = 0;
        if (FALSE == ::VirtualProtect(lpBaseAddress, dwSizeOfImage, PAGE_EXECUTE_READWRITE, &dwOldProtect))
        {
            ShowError("VirtualProtect");
            return NULL;
        }

        // 修改PE文件加载基址IMAGE_NT_HEADERS.OptionalHeader.ImageBase
        if (FALSE == SetImageBase(lpBaseAddress))
        {
            ShowError("SetImageBase");
            return NULL;
        }

        // 调用DLL的入口函数DllMain,函数地址即为PE文件的入口点IMAGE_NT_HEADERS.OptionalHeader.AddressOfEntryPoint
        if (FALSE == CallDllMain(lpBaseAddress))
        {
            ShowError("CallDllMain");
            return NULL;
        }

        return lpBaseAddress;
    }
```

于如何修改重定位表、导入表以及遍历导出表等操作在此就不详细说明，读者直接阅读代码即可，代码中均有详细的注释说明。直接内存加载运行TestDll.dll文件，若成功执行TestDll.dll入口处的弹窗代码，弹窗提示则说明加载运行成功，如下图所示。

- 代码地址：启动技术-MemLoadDll（内存加载DLL）







该部分知识对于初学者来说，理解起来比较复杂。但是，只要熟悉PE结构，这个程序理解起来就会容易得多。对于重定位表、导入表，以及导出表部分的具体操作并没有详细讲解。如果没有了解PE结构，那么理解起来会有些困难；如果了解了PE结构，那么就很容易理解该部分知识。

可以通过暴力枚举PE结构特征头的方法，来枚举进程中加载的所有模块，它与通过正常方法获取到的模块信息进行比对，从而判断是否存在可疑的PE文件。

四.总结

写到这里，这篇文章就介绍完毕，希望对您有所帮助，最后进行简单的总结下。

- 创建进程API：介绍使用WinExec、ShellExecute以及CreateProcess创建进程
- 突破SESSION 0隔离创建进程：主要通过CreateProcessAsUser函数实现用户进程创建
- 内存直接加载运行：模拟PE加载器，直接将DLL和exe等PE文件加载到内存并启动运行

学安全一年，认识了很多安全大佬和朋友，希望大家一起进步。这篇文章中如果存在一些不足，还请海涵。作者作为网络安全初学者的慢慢成长路吧！希望未来能更透彻撰写相关文章。同时非常感谢参考文献中的安全大佬们的文章分享，深知自己很菜，得努力前行。

(By:Eastmount 2020-07-26 星期一 晚上9点写于武汉 http://blog.csdn.net/eastmount/)

参考文献：

[1] https://github.com/eastmountyxz/Windows-Hacker-Exp/blob/master/4.启动技术
[2] WINEXEC, SHELLEXECUTE, CREATEPROCESS - 篱笆博客
[3] https://docs.microsoft.com/en-us/previous-versions/aa908775(v=msdn.10)
[4] https://docs.microsoft.com/zh-cn/windows/win32/api/winbase/nf-winbase-winexec
[5] https://docs.microsoft.com/en-us/windows/win32/api/wtsapi32/nf-wtsapi32-wtsqueryusertoken
[6] https://docs.microsoft.com/zh-cn/windows/win32/api/winsvc/nf-winsvc-startservicew
[7] Windows创建进程 - m_buddy
[8] 关于“Error: “const char *” 类型的实参与 "LPCWSTR"类型的形参不兼容”错误的解决方案
[9] 穿透Session 0 隔离（一）- 李敬然老师
[10] C++创建Windows后台服务程序 - blade1080
[11] 系统权限服务创建桌面进程(进程也是系统权限)
[12] 突破Session0隔离在系统服务程序中创建用户桌面进程
[13] DLL内存加载 - 墨鱼菜鸡
[14] 内存直接加载运行DLL文件 - 自己的小白