目录

探究底层机制：二进制安全

- 最新版本：`v0.9.2`
- 更新时间：`20230901`

简介

整理二进制安全，即PWN；先对二进制安全概述；再介绍相关基础知识，比如编程语言，包括汇编语言X86、ARM等；体系架构的寄存器X86、ARM和堆栈，以及可执行文件；再总结Windows相关的二进制安全，包括安全机制的CFG、DEP、GS、SafeSEH、SEHOP，和攻击技术，包括缓冲区溢出、ROP、Heap spray、Shellcode等；再整理二进制相关工具，包括反编译器的dnSpy、ILSpy，静态安全检工具的winchecksec，可执行文件工具的Exeinfo PE，查壳工具的Detect It Easy、PEiD，逆向工具的IDA、Ollydbg、WinDbg，内存修改工具的CE、MHS、ModifyMemory；以及通用的知识，比如安全机制中的ASLR最后附上参考资料。

源码+浏览+下载

本书的各种源码、在线浏览地址、多种格式文件下载如下：

HonKit源码

- crifan/explore_underlying_mechanism_binary_security: 探究底层机制：二进制安全

如何使用此HonKit源码去生成发布为电子书

详见：crifan/honkit_template: demo how to use crifan honkit template and demo

在线浏览

- 探究底层机制：二进制安全 book.crifan.org
- 探究底层机制：二进制安全 crifan.github.io

离线下载阅读

- 探究底层机制：二进制安全 PDF
- 探究底层机制：二进制安全 ePub
- 探究底层机制：二进制安全 Mobi

版权和用途说明

鸣谢

感谢我的老婆**陈雪**的包容理解和悉心照料，才使得我 `crifan` 有更多精力去专注技术专研和整理归纳出这些电子书和技术教程，特此鸣谢。

其他

作者的其他电子书

本人 `crifan` 还写了其他 `150+` 本电子书教程，感兴趣可移步至：

crifan/crifan_ebook_readme: Crifan的电子书的使用说明

关于作者

关于作者更多介绍，详见：

关于CrifanLi李茂 – 在路上

二进制安全概述

如之前信息安全概览所总结的，安全方向，大致可以分为：

- 根据不同 `端` = `目标` = `设备` 或 `侧重点` = `方向` 分
  - 远程 -> `Web端` ： `网络安全` = `Web安全` = `互联网安全`
    - 不同侧重点 = 攻防
      - 攻 = 攻击： `渗透测试`
      - 防 = 防护：
        - `安全开发` = `安全功能开发`
        - `安全分析`
  - 本地=本机 -> `设备端`
    - `PC端` = `桌面端` ： `计算机安全`
      - 包含
        - `Windows`
          - 涉及领域： `二进制安全` ~= `PWN`
            - -> `Window漏洞挖掘` ~= `Windows漏洞分析`
        - `Mac`
        - `Linux`
    - `移动端` ： `移动安全`
      - 包含
        - `Android`
        - `iOS`
    - `IoT端` ： `物联网安全` ~= `工控安全`
    - 其他特定设备
      - `WiFi安全`

本教程主要介绍其中的，侧重 `本机` 的、 `PC端` 的：**二进制安全=PWN**

涉及内容

- `二进制安全` 往往涉及到
  - 漏洞 ~= 漏洞安全 ~= 漏洞挖掘
    - 不同操作系统
      - Windows
        - Window漏洞安全 ~= Windows漏洞挖掘
  - 逆向 ~= 逆向工程
    - 不同系统或端
      - PC端
        - Windows
          - Windows逆向
      - 移动端
        - Android
          - Android逆向
        - iOS
          - iOS逆向
  - 底层技术 -> 底层安全
    - 编译和链接
    - 汇编语言
      - X86
      - ARM

        - 二进制文件格式
            - Windows
                - exe
            - Linux
                - elf
            - Mac
                - app
                - dmg
                - pkg
        - 堆和栈
            - 缓冲区溢出

相关工作方向

- 概述型
    - 二进制安全研发工程师
- 漏洞
    - 二进制漏洞分析
        - Windows漏洞分析
    - 二进制漏洞挖掘
        - Windows漏洞挖掘
- 病毒
    - 恶意代码分析
- 游戏领域
    - 游戏外挂
        - 游戏安全工程师

基础知识

折腾二进制安全，往往需要具备一定的 `基础知识` ，也称为 `背景知识` 。

二进制安全常涉及的知识有：

- 操作系统
  - 内存管理
  - 进程线程
  - 内核
- 文件格式
  - PE
- Shellcode编写技术
- 等

相关的一些书籍：

- 《漏洞战争》
- 《C++反汇编与逆向技术揭秘》
- 《Exploit编写系列教程》
- 《软件调试》
- 《0day：软件调试分析技术》

转个某人整理的，二进制安全相关基础知识的概览：

编程语言

- 编程语言种类
  - 机器语言：用二进制编码表示每条指令，是计算机能直接识别和执行的语言。
    - 原理：底层都是0和1的二进制数据
    - 注：理论上来说，程序员也可以写机器语言
      - 只不过太难写，以及没必要直接写，所以实际上没人直接写机器语言
  - 汇编语言
    - 实质和机器语言是相同的，都是直接对硬件操作，只不过指令采用英文缩写的标识符，更容易识别和记忆
    - 组成
      - 指令，伪指令，宏指令
    - 逻辑
      - 你（程序员）写汇编的字母（和要操作的数字等），汇编程序帮你把 汇编的字母（和数字）翻译成0和1的二进制
        - 这样机器才能看懂，才能运行对应程序
  - 高级语言
    - 编写的程序不能直接被计算机识别，必须经过转换（目标代码即机器码）才能被执行
    - 按照转换方式分类
      - 解释类：**边**翻译**边**执行
      - 编译类：**先**翻译**再**执行
    - 常见语言
      - `C`
        - C语言的特点
          - c语言允许直接访问物理地址，可以直接对硬件进行操作
          - c语言程序代码质量高，程序执行效率高
          - c语言使用范围大，可移植性好
      - `C++`
        - 由于：C++编译器优化程度太大
        - 结果：世界上没有C++反编译器
          - 也没有完美的C语言反编译器
            - 很难做到完美

汇编语言

- 汇编
  - `汇编 = 汇编语言 = 汇编指令`
  - 是什么
    - 汇编大多是指汇编语言，汇编程序
    - 把汇编语言翻译成机器语言的过程称为汇编
  - 汇编语言
    - 在汇编语言中，用符号代替机器语言的二进制码，就把机器语言变成了汇编语言
      - 是用助记符，符号和数字等来表示指令的程序设计语言，它与机器语言指令是一一对应的
  - 汇编程序
    - 用汇编语言编写的程序，机器不能直接识别。要由一种程序将汇编语言翻译成机器语言，这种起翻译作用的程序叫汇编程序。
      - 汇编程序是系统软件中语言处理的系统软件
  - 特点
    - 由于汇编更接近机器语言，能够直接对硬件进行操作，生成的程序与其他的语言相比具有更高的运行速度，占用更小的内存
  - 应用
    - 因此在一些对于时效性要求很高的程序，许多大型程序的核心模块以及工业控制方面大量应用
  - 种类
    - 有多少中不同内核的CPU，就有多少种汇编语言
  - 总结
    - 不同内核的CPU，必须有对应的汇编语言编译器将汇编语言编写的程序编译成对应CPU的机器语言代码，CPU才能正确识别和执行这些代码
    - 不同架构的CPU的汇编指令集并不相同
    - 不同的汇编程序有不同的汇编语言规定

通用知识

和逆向和破解相关的汇编语言的通用知识：

- 逆向中关键的指令：
  - `ldr`，`mov`，读取指令，从地址读取数据到寄存器。
  - `str`，保存指令，保存数据到寄存器。
  - `b`，跳转指令，跳转到某个地址。
  - `cmp`，比较指令，说明这里有分支。

X86汇编语言

- 常见X86汇编语言类型
  - ASM
  - MASM
  - TASM
  - OPTASM
  - 等

Intel 8086的汇编语言指令

- Intel 8086的指令
  - 概述
    - 117条基本指令
      - 6个功能组
        - 数据传送类指令
          - MOV/XCHG，PUSH/POP，LEA
        - 算数运算类指令
          - ADD/ADC/INC，SUB/SBB/DEC/CMP/NEG，MUL/IMUL，DIV/IDIV
        - 位操作类指令
          - AND/OR/XOR/NOT/TEST
        - 串操作类指令
        - 控制转移类指令
          - JMP/JCC/LOOP，CALL/RET，INT n
        - 处理机控制类指令
          - NOP
    - 其他
      - 伪指令

数据传送类指令

- 数据传送类指令
  - 概述：计算机中最基本，最重要的一种操作，传送指令也是最常用的一类指令
  - 作用：传送指令把数据从一个位置传送到另一个位置。
  - 特点：除标志寄存器传送指令外，均不影响标志位
  - 包含：MOV XCHG PUSH POP LEA
    - 传送指令MOV
      - 把一个字节或操作数从源地址传送至目的地址
    - 交换指令XCHG
      - 把两个地方的数据进行互换
        - 寄存器与寄存器之间对换数据
        - 寄存器与存储器之间对换数据
        - 不能在存储器与存储器之间对换数据
    - 进栈指令PUSH
      - PUSH
        - Push r16/m16/seg 操作过程：1. SP<-SP-2 2.SS:[SP]<-r16/m16
      - POP
        - 出栈指令，操作与PUSH相反
      - 算数运算类指令
        - 概述：四则运算计算机经常进行的一种操作

- 作用：实现二进制（十进制）数据的四则运算
- 特点：算术运算类指令往往对标志有影响
- 建议
  - 掌握 ADD/ADC/INC , SUB/SBB/DEC/NEG/CMP
  - 熟悉 MUL/IMUL , DIV/IDIV
  - 理解 CBW/CWD , DAA/DAS , AAA/AAS/AAM/AAD
- 包含
  - 加法指令ADD
    - 功能：ADD指令将源与目的操作数相加，结果送到目的操作数
      - ADD指令按状态标志的定义相应设置状态标志
    - 语法：
      - ADD reg, imm/reg/mem reg<-reg+imm/reg/mem
  - 带进位加法指令ADC
    - 功能：ADC指令将源与目的的操作数相加，在加上进位CF标志，结果送到目的操作数
      - ADC指令按状态标志的定义相应设置状态标志
    - 用途：ADC指令主要与ADD配合，实现多精度加法运算
    - 语法
      - ADD reg , imm/reg/mem
      - ADC mem , imm/reg
  - 增量指令INC
    - 功能：对操作数加1（增量）
      - 不影响进位CF标志，按定义设置其他状态标志
    - 语法
      - INC reg/mem reg/mem<-reg/mem+1
  - 减法指令SUB
    - 功能：将目的操作数减去源操作数，结果送到目的操作数
      - 按照定义相应设置状态标志
    - 语法
      - SUB reg , imm/reg/mem reg<-reg-imm/reg/mem
  - 带借位减法指令SBB
    - 功能：SBB指令将目的操作数减去源操作数，在减去借位CF（进位），结果送到目的操作数
    - 用途：SBB指令主要与SUB配合，实现多精度减法运算
    - 语法：SBB reg , imm/reg/mem reg<-reg-imm/reg/mem-CF
  - 减量指令DEC
    - 功能：DEC指令对操作数减1（减量）
    - 语法：DEC reg/mem reg/mem<-reg/mem-1
    - 说明：INC指令和DEC指令都是单操作数指令，主要用于对计数器和地址指针的调整
  - 求补指令NEG
    - 功能：NEG指令对操作数执行求补运算：用0减去操作数，然后结果返回操作数，求补运算也可以表达成，将操作数按位取反后加1
      - NEG指令对标志的影响与用0作减法的SUB指令一样
    - 语法：NEG reg/mem reg/mem<-0-reg/mem
  - 比较指令CMP
    - 功能：将目的操作数减去源操作数
      - 按照定义相应设置状态标志
    - 说明：执行的功能与SUB指令类似，但结果不回送目的操作数
    - 语法：CMP reg , imm/reg/mem reg—imm/reg/mem

位操作类指令

- 位操作类指令
  - 概述：以二进制为基本单位进行数据的操作。一类常用的指令
  - 包含

- 逻辑运算指令
  - ADD(与)
    - 功能：对两个操作数执行逻辑与运算，结果送到目的操作数。
    - 语法：ADD des , src des<-des^src
  - OR(或)
    - 功能：对两个操作数执行逻辑或运算，结果送到目的操作数
    - 语法：OR dest , src dest<-destv src
  - XOR(异或)
    - 功能：对两个操作数执行逻辑异或运算，结果送到目的操作数
    - 语法：XOR dest , src
  - NOT(非)
    - 功能：对一个操作数执行逻辑非运算
      - 按位取反，原来的0的位变1，原来的1的位变0
    - 语法：NOT reg/mem
  - TEST(测试)
- 移位指令
  - SHL(逻辑左移)
  - SHR(逻辑右移)
  - SAL(算术左移)
  - SAR(算术右移)
- 循环移位指令
  - ROL(左循环移位)
  - ROR(右循环移位)
  - RCL(带进位左循环移位)
  - RCR(带进位右循环移位)

串操作类指令

- 串操作类指令

控制转移类指令

- 控制转移类指令
  - 概述：用于实现分支，循环，过程等程序结构，是仅次于传送指令的常用指令
  - 建议：
    - 重点掌握：JMP/JCC/LOOP、CALL/RET、INT n/IRET 常用系统功能调用
    - 一般了解：LOOPZ/LOOPNZ INTO
  - 包含
    - 无条件转移指令JMP
      - 语法：JMP label
      - 作用：
        - 程序转向label标号指定的地址（标号要在程序其他位置标出）
      - 说明：
        - 只要执行无条件转移指令JMP，不需要任何条件，就使程序转到指定的目的地址处，从目标地址
        - 操作数是要转移到的目标地址开始执行指令
      - 原理
        - 程序的执行地址，是由段寄存器CS和指令指针IP共同确定的，即当前指令的地址为CS：IP
        - 程序的跳转是通过修改CS和IP的值来实现的
    - 条件转移指令JCC
      - 语法：Jcc label
        - 条件满足，发生转移：IP<-IP+8位位移量
        - 条件不满足，顺序执行
      - 说明：

        - 指定的条件cc如果成立，程序转移到由标号label指定的目标地址去执行指令，条件不成立，则程序将顺序执行下一条指令
        - 操作数label是短转移指令，要跳转的地址必须距当前IP地址-128~+127个单元的范围之内
        - Jcc指令不影响标志
          - 但要利用标志
            - 根据利用的标志位不同，16条指令分为3种情况
              - 判断单个标志位状态
              - 比较无符号数高低
              - 比较有符号数大小
    - 循环指令LOOP
      - 语法：LOOP label
      - 功能：循环指令是一种特殊的转移指令，当满足某条件时，反复执行一系列操作，知道不满足为止
      - 说明：循环指令利用CX寄存器作为计数器
    - 子程序指令
      - 语法：
        - 4种类型
          - CALL label； 段内调用，相对寻址。
          - CALL r16/m16；段内调用，间接寻址
          - CALL far ptr label；段间调用，直接寻址
          - CALL far ptr mem；段间调用，间接寻址
      - 原理：
        - 子程序是完成特定功能的一段程序
        - 当主程序（调用程序）需要执行这个功能时，采用CALL调用指令转移到该子程序的起始处执行
        - 当运行完子程序功能后，采用RET返回指令回到主程序继续执行
      - 说明
        - 子程序通常是与主程序分开完成特定功能的一段程序，程序中有时要反复的实现相同的功能只不过参数不同而已，把仅参数不同功能重复的程序编写成为子程序，执行这个功能时，就可以调用该子程序，执行完成后在返回主程序。
    - 中断指令
      - 语法=中断指令：INT
        - 举例：
          - INT i8
        - 特殊：
          - IRET：中断返回指令，实现中断返回
          - INTO：溢出中断指令
      - 作用：中断是一种改变程序执行顺序的方法，在程序运行时，遇到某些需要紧急处理的情况，如停电，数据的实时接收，溢出等，处理器暂停主程序的执行，转去执行中断处理程序。
      - 分类
        - 内部中断
        - 外部中断

处理机控制类指令

- 处理机控制类指令
  - 概述：对CPU状态进行控制的指令
  - 包含
    - 空操作指令NOP
      - 语法：
        - NOP CS：SS：DS：ES
      - 作用：不执行任何操作，但占用一个字节存储单元，空耗一个指令执行周期。
      - 用途
        - NOP常用于程序调试
          - 在需要预留指令空间时用NOP填充

        - 代码空间多余时也可以用NOP填充
        - 还可以用NOP实现软件延时
    - 其他
        - LOCK HLT ESC WAIT
- 说明
    - 事实上，NOP和XCHG，AX，的指令代码一样都是90H
    - 段超越前缀指令：在允许段超越的存储器操作数之前，使用段超越前缀指令，将采用指定的段寄存器寻址操作数
        - CS：使用代码段的数据
        - SS：使用堆栈段的数据
        - DS：使用数据段的数据
        - ES：使用附加段的数据

伪指令

- 伪指令
    - 概述：
        - 没有对应的机器码的指令，最终不被CPU所执行
        - 伪指令是由编译器来执行的指令
            - 编译器根据伪指令来进行相关的编译工作
    - 语法
        - segment和ends是一对成对使用的伪指令
            - 这是在写可被编译器编译的汇编程序时，必须要用到的一对伪指令
        - segment和ends的功能是定义一个段
            - segment说明一个段开始
                - 语法：段名 segment
            - ends说明一个段结束
                - 语法：段名 ends
    - 说明
        - 一个汇编程序是由多个段组成的，这些段被用来存放代码，数据或当作栈空间来使用
        - 一个有意义的汇编程序中至少要有一个段，这个段用来存放代码
    - 注意
        - 不要搞混end和ends
            - end是汇编语言的结束
                - 一个汇编程序的结束标记，编译器在编译汇编程序的过程中，如果碰到了伪指令end，就结束对源程序的编译
            - ends是伪指令的结束

ARM汇编语言

体系架构

- PC端
  - X86
- 移动端
  - ARM

寄存器

- 寄存器
  - 是什么：存储信息的单元或者说是器件
    - 注：这里讨论的寄存器都是CPU中的寄存器，位于CPU内部，而内存位于CPU外部
  - 使用
    - 对于一个汇编程序员来说，CPU中主要可以使用的也就是寄存器而已
      - 对比
        - 电脑 的内存
        - CPU的寄存器
    - 编程序员可以使用指令来读写CPU中的寄存器，从而实现对于CPU的控制
  - 特点
    - 不同的CPU，寄存器的个数和结构都是不一样的

X86寄存器

- 8086 CPU中寄存器总共为14个：且均为16位（32位和64位均以16位为基础）
  - 14个寄存器：AX BX CX DX SP BP SI DI IP FLAG CS DS SS ES
  - 根据类型分
    - 通用寄存器
      - 数据寄存器：AX，BX，CX，DX
        - AX：累加寄存器，也称为累加器
        - BX：基地址寄存器
        - CX：计数器寄存器
        - DX：数据寄存器
      - 指针寄存器：SP和BP
        - SP：堆栈指针寄存器
        - BP：基指针寄存器
      - 变址寄存器：SI和DI
        - SI：源变址寄存器
        - DI：目的变址寄存器
    - 控制寄存器
      - IP：指令指针寄存器
      - FLAG：标志寄存器
    - 段寄存器
      - CS：代码段寄存器
      - DS：数据段寄存器
      - SS：堆栈段寄存器
      - ES：附加段寄存器

x86-64 的调用约定

x86-64 有16个64位寄存器，分别是：

rax，rbx，rcx，rdx，esi，edi，rbp，rsp，r8，r9，r10，r11，r12，r13，r14，r15

寄存器	描述
rax	作为函数返回值使用
rsp	栈指针寄存器，指向栈顶
rdi，rsi，rdx，rcx，r8，r9	依次用作函数参数；如果断点在 OC 方法的第一行，那 rdi 就是 self，rsi 就是 cmd
rbx，rbp，r10，r11，r12，r13，r14，r15	通用寄存器

ARM 寄存器

- ARM寄存器和架构
  - 32位 arm
  - 64位 arm

32位arm的调用约定

寄存器	描述
r0-r3	传递参数与返回值。如果断点在 OC 方法的第一行，那 r0 就是 self，r1 就是 cmd。如果超过四个参数，或者一些例如结构体的参数超过了32位 bit，那么参数将会通过栈来传递；返回值一般都在 r0 上
r4-r6, r8, r10-r11	没有特殊规定，通用寄存器
r7	栈帧指针寄存器(Frame Pointer)，指向前一个保存的栈帧(stack frame)和链接寄存器(link register， lr)在栈上的地址
r9	操作系统保留
r12	IP 寄存器(intra-procedure scratch)
r13	SP 寄存器(stack pointer)，是栈顶指针
r14	LR 寄存器(link register)，存放函数返回后需要继续执行的指令地址
r15	PC 寄存器(program counter)，指向当前指令地址
CPSR	当前程序状态寄存器(Current Program State Register)，在用户状态下存放像 condition 标志中断禁用等标志

arm64的调用约定

arm64有 r0 - r30 是31个通用整形寄存器，PC 不能再作为寄存器直接访问。每个寄存器可以存取一个64位大小的数。 当使用 x0 - x30 访问时，它就是一个64位的数。当使用 w0 - w30 访问时，访问的是这些寄存器的低32位。

寄存器	描述
x0–x7	传递参数与返回值。如果参数个数超过了8个，多余的参数会存在栈上；返回值一般都在 x0 上
x29	栈帧指针寄存器(Frame Pointer)，指向前一个保存的栈帧(stack frame)和链接寄存器(link register， lr)在栈上的地址
x31	SP 寄存器(stack pointer)，是栈顶指针；根据不同指令，也有可能是 zero register
x30	LR 寄存器(link register)，存放函数的返回地址
CPSR	当前程序状态寄存器(Current Program State Register)，在用户状态下存放像 condition 标志中断禁用等标志

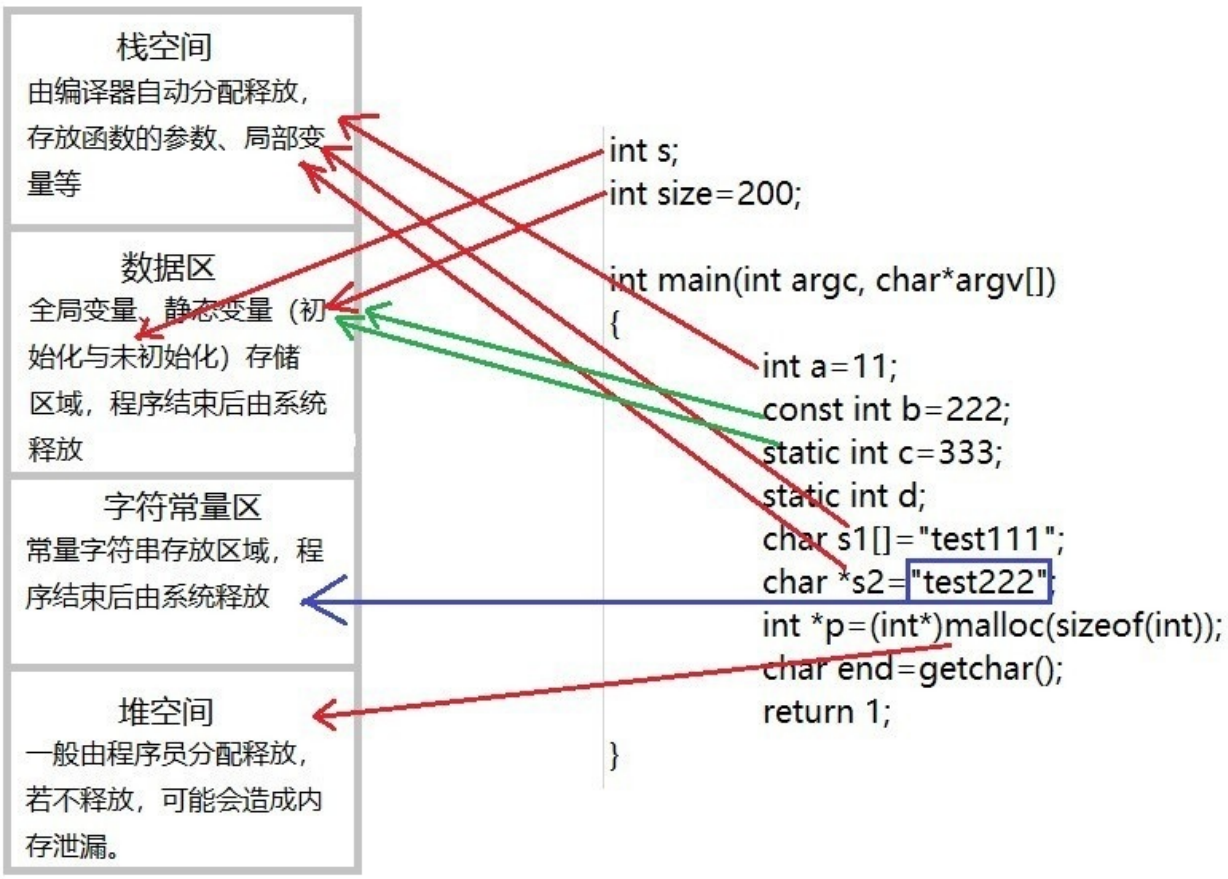
堆栈

- 堆栈
  - 是什么：堆栈都是一种数据项按序排序的数据结构
  - 特点：只能在一端（称为栈顶）对数据项进行插入和删除
    - 堆：队列优先，先进先出
    - 栈：先进后出
  - 功能：暂时存放数据和地址
  - 用途：通常用来保护断电和现场
  - 操作：堆栈中定义了一些操作
    - 两个最重要的是PUSH和POP
      - PUSH操作：在堆栈的顶部加入一个元素
      - POP操作：相反，在堆栈顶部移去一个元素，并将堆栈的大小减一

相关安全问题

- 数组越界访问
- 堆溢出
- 栈溢出
  -

堆与栈的分配



可执行文件

常见可执行文件格式

- 不同系统
  - 最早
    - `COFF`
      - = `Common Object File Format`
  - `Windows`
    - `PE`
      - = `Portable Executable`
  - `Linux` 类系统：`Linux`、`Unix`、`Mac` 等
    - ELF
      - = `Executable and Linkable Format`
  - `Mac`
    - Mach-O
- 其他相关
  - 二进制往往还根据系统不同而略有不同
    - 举例
      - Windows系统
        - `32位` = `x86`
        - `64位` = `x64`

Windows

此处介绍二进制安全在Windows方面的应用。

Windows常用漏洞分析方法

- 静态分析
- 动态调试
- 补丁比较
- 污点追踪

概述：

软件逆向

- 软件逆向
  - =软件逆向工程
  - 定义
    - 通过反汇编和调试等手段，分析计算机程序的二进制可执行代码从而获得程序的算法细节和实现原理的技术。
  - 研究对象
    - 没有公开源代码的计算机程序
      - 主要是已经经过编译的二进制可执行代码
        - 举例
          - win32平台上
            - PE文件
              - 常见文件格式
                - exe
                - dll
  - 分类

        - 系统级逆向
            - 大范围分析观察，整体把握
        - 代码级逆向
            - 程序二进制码中提取设计理念和算法
    - 步骤
        - 研究保护方法，去除保护功能
            - 解码/反汇编（目标二进制代码）
        - 反汇编目标软件，定位功能函数
            - 中间语言翻译（汇编或类汇编代码）
        - 分析汇编代码
            - 数据流分析（各级中间语言）
        - 修改汇编代码或还原高级源代码
            - 其他分析和优化（高级抽象代码）
    - 工具
        - `Ollydbg`：动态追踪工具，插件较好较多
        - `Windbg`：用户态和内核态调试工具
        - IDA：交互式反汇编器
        - `PEID`：著名的查壳工具
        - `C32Asm`：反汇编程序，可直接修改软件内部代码，有十六进制编辑模式
    - 主要应用
        - 软件破解：破解软件的版权让用户不支付授权费用就可以使用软件的全部功能。
        - 病毒和恶意程序的分析：恶意程序的传播机制和危害并设计出解，分析病毒解决办法。
        - 系统漏洞分析：分析漏洞原理，设计补丁程序或者编写利用程序（Exploit）
        - 分析不公开的文件格式，协议等
        - 分析windows或mac平台上的硬件驱动程序编写linux下的相应驱动
        - 挖掘消费电子产品的潜能
        - 挖掘操作系统未文档化的API，发现更多内幕
        - 计算机犯罪取证

Windows的安全机制

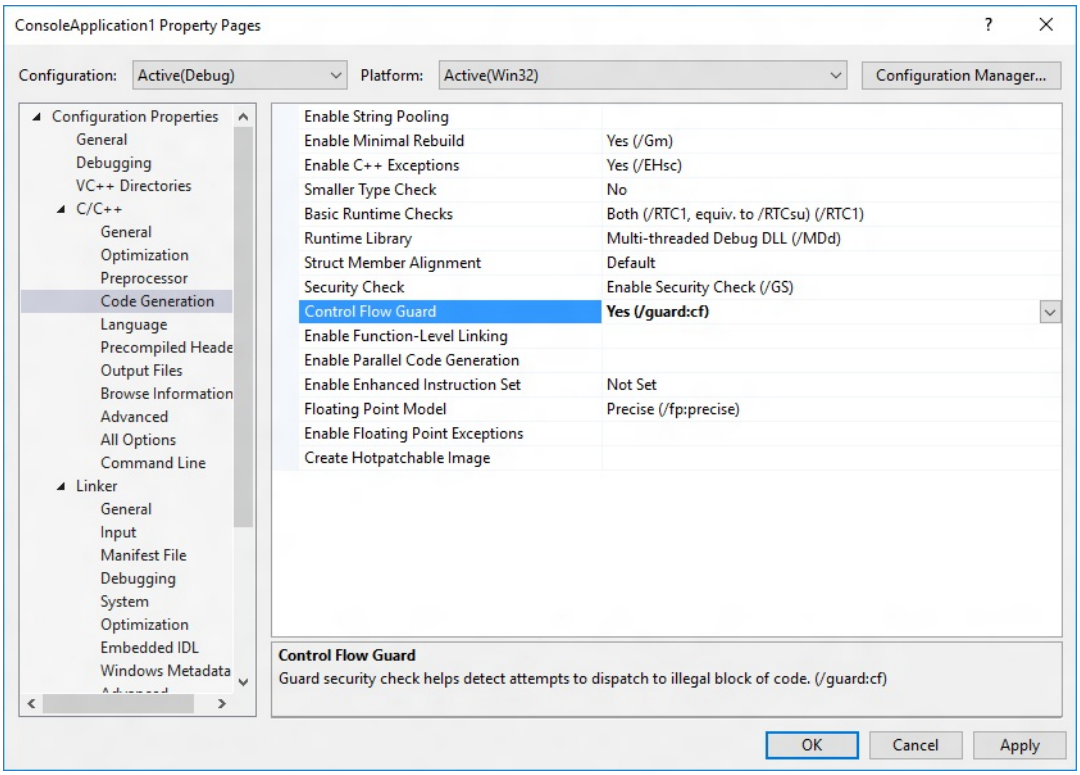
此处介绍关于Windows的相关安全机制，也对应着Windows相关的漏洞攻防技术。

- Windows的漏洞安全
  - 概述
    - 在过去的二十多年当中，微软在提高操作系统的安全性方面一直做着不懈的努力。
    - 从Win98到WinXP、Vista、Win7再到最新的Win10，每个版本的发布都会带来安全性质的飞跃。
    - 除了在安全功能的保护下大大提高了系统安全性，微软还在内存保护方面做了很多的工作，来提高内存保护的安全性
      - 堆栈保护机制：GS
      - 异常处理保护：SafeSEH
      - 数据执行保护：DEP
      - 地址空间分布随机化：ASLR
      - 结构化异常覆盖保护：SEHOP

心得

- 在用 `vs` 较高版本编译程序测试溢出等的时候，注意要关闭保护功能(如常见的 `DEP` 以及 `ASLR`)
  - 在项目菜单的属性里面设置

CFG

- `CFG`
  - = `Control Flow Guard` = 控制流保护
  - 是什么：一个高度优化的平台级的安全特性
  - 开启了CFG
    - 会告诉编译器，在编译期间，去分析代码的控制流，找到有哪些间接调用，并记录下来，用于分析编译出的二进制文件
    - Windows对于程序的每个间接调用都会做个检查
      - 如果检测发现不合法的，会抛出异常 `` `RaiseFastFailException ``
  - 细节
    - 通过对于一个程序可以从哪里开始运行加上严格的限制
      - 使得想要通过缓冲区溢出等漏洞去破解以执行任意代码的难度的大大增加
  - 支持情况
    - `Microsoft Visual Studio 2015` 之后才支持
  - 如何开启
    - 编译器参数
      - 语法：`/guard:cf`
    - 界面设置
      - 

  - 如何确认一个程序支持了CFG?
    - 用 `dumpbin` 查看导出信息
      - `dumpbin /headers /loadconfig test.exe`
        - 输出中包含 `Guard`

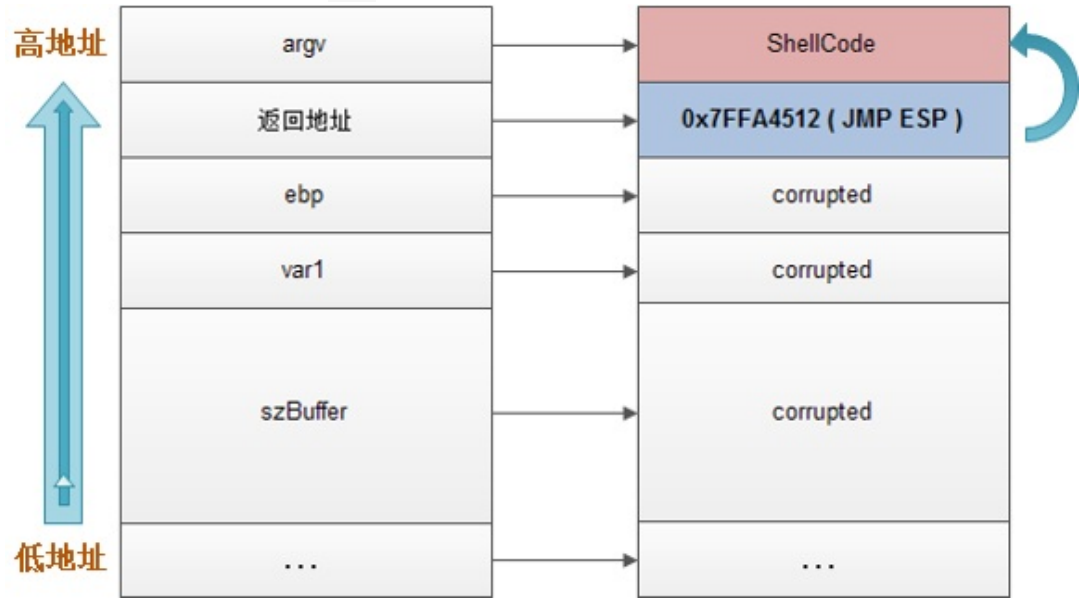
```
OPTIONAL HEADER VALUES
            20B magic # (PE32+)
          12.10 linker version
         1BFE00 size of code
         282600 size of initialized data
            200 size of uninitialized data
          9E090 entry point (000000014009E090)
           1000 base of code
      140000000 image base (0000000140000000 to 0000000140447FFF)
           1000 section alignment
            200 file alignment
          10.00 operating system version
          10.00 image version
          10.00 subsystem version
              0 Win32 version
         448000 size of image
            400 size of headers
         4589A6 checksum
              2 subsystem (Windows GUI)
            C1C0 DLL characteristics
                   Dynamic base
                   Check integrity
                   NX compatible
                   Guard
                   Terminal Server Aware
```

- 加载配置中包含 `CF Instrumented` 和 `FID table present`

```
Section contains the following load config:

           000000A0 size
                  0 time date stamp
               0.00 Version
                  0 GlobalFlags Clear
                  0 GlobalFlags Set
                  0 Critical Section Default Timeout
                  0 Decommit Free Block Threshold
                  0 Decommit Total Free Threshold
    0000000000000000 Lock Prefix Table
                  0 Maximum Allocation Size
                  0 Virtual Memory Threshold
                  0 Process Heap Flags
                  0 Process Affinity Mask
                  0 CSD Version
               0000 Reserved
    0000000000000000 Edit list
    000000014023C008 Security Cookie
    00000001401C41A0 Guard CF address of check-function pointer
    00000001401C41A8 Guard CF address of dispatch-function pointer
    00000001401C42A8 Guard CF function table
                E95 Guard CF function count
           00003500 Guard Flags
                  CF Instrumented
                  FID table present
                  Protect delayload IAT
                  Delayload IAT in its own section
```

DEP

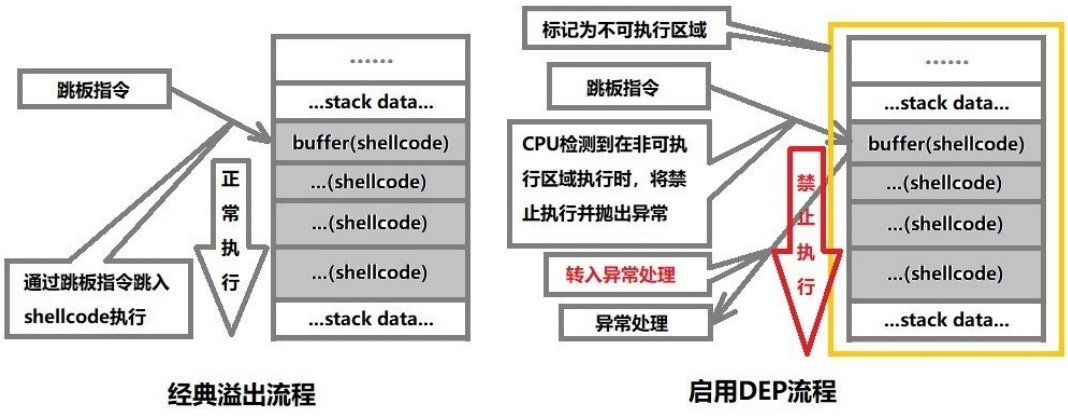
- `DEP`
  - = `Data Execution Prevention` = `数据执行保护`
  - 背景
    - 早期操作系统没有区分数据和代码，`EIP` 指向哪里就去哪里执行



    - -> 溢出攻击的根源在于现代计算机对数据和代码没有明确区分这一先天缺陷，就目前来看重新去设计计算机体系结构基本上是不可能的，我们只能靠向前兼容的修补来减少溢出带来的损害，DEP（数据执行保护，Data Execution Prevention）就是来弥补计算机对数据和代码混淆这一缺陷的
      - 微软从WinXP SP2开始提供这种技术支持
  - 含义：`堆`，`栈` 上的内存页属性为不可执行，执行会出错
  - 目的：防止某些内存区块，比如栈，被执行
  - 如何开启
    - 语法：
      - 开启：`/NXCOMPAT`
      - 关闭：`/NXCOMPAT:NO`
    - 选项设置

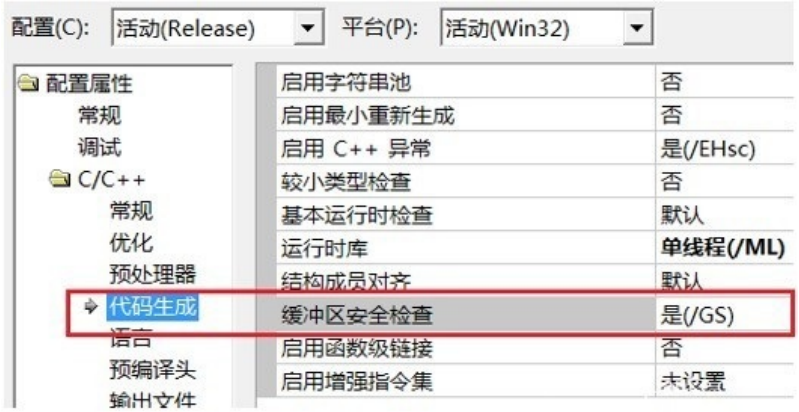
- 效果
  - 开启了DEP后，能帮助检测到异常情况：

  - 开启DEP前后的流程对比



- 详解
  - `DEP` 能够在内存上执行额外检查以帮助防止在系统上运行恶意代码
  - `DEP` 是一套软硬件技术，能够在内存上执行额外检查以帮助防止在系统上运行恶意代码。在 `Microsoft Windows XP Service Pack 2` 及以上版本的 `Windows` 中，由硬件和软件一起强制实施 `DEP`
  - 支持 `DEP` 的 `CPU` 利用一种叫做 `No eXecute = 不执行` 的技术识别标记出来的区域。如果发现当前执行的代码没有明确标记为可执行（例如程序执行后由病毒溢出到代码执行区的那部分代码），则禁止其执行，那么利用溢出攻击的病毒或网络攻击就无法利用溢出进行破坏了。如果 `CPU` 不支持 `DEP`，Windows 会以软件方式模拟出 `DEP` 的部分功能
- 相关
  - 如果同时开启了 `DEP` 和 `ASLR`
    - 会让破解非常困难
      - 背景：一般用 `shellcode` 和 `ROP` 技术去破解
  - `ROP`
    - = `Return Oriented Programming`
      - 早期叫：`Ret2Libc`
      - 实现原理
        - `ROP` 由一系列的 `Gadget` 组成
        - 所谓 `ROP Gadget`，就是一系列以retn结尾的指令，所有的这些 `Gadget` 组合起来就能完成特定的任务
          - 比如调用 `VirtualProtect` 给指定的内存块添加可执行属性
  - 寄存器
    - `eip`
    - `esp + offset`

GS

- `GS`
  - = `Buffer Security Check` = `缓冲区安全检查`
    - 别称：
      - `Stack cookie`
      - `Security Cookie`
        - `OD` 、 `IDA` 中称为 `Security Cookie`
      - `GS cookie protection`
      - `GS security protection`
  - 是什么：一个**编译器参数**
  - 功能和作用：用于缓存安全检查，检查缓存是否溢出
    - 决定编译器**是否生成**用于检测是否发生了缓冲区溢出 `buffer overruns` 的**代码**
  - 如何设置
    - Windows
      - IDE：`Visual Studio`
        - 编译器：`MSVC Compiler`
          - 编译器参数：`GS`
            - 如何配置：
              - 开启：`/GS`
                - VS安全编译选项中的GS
                - 

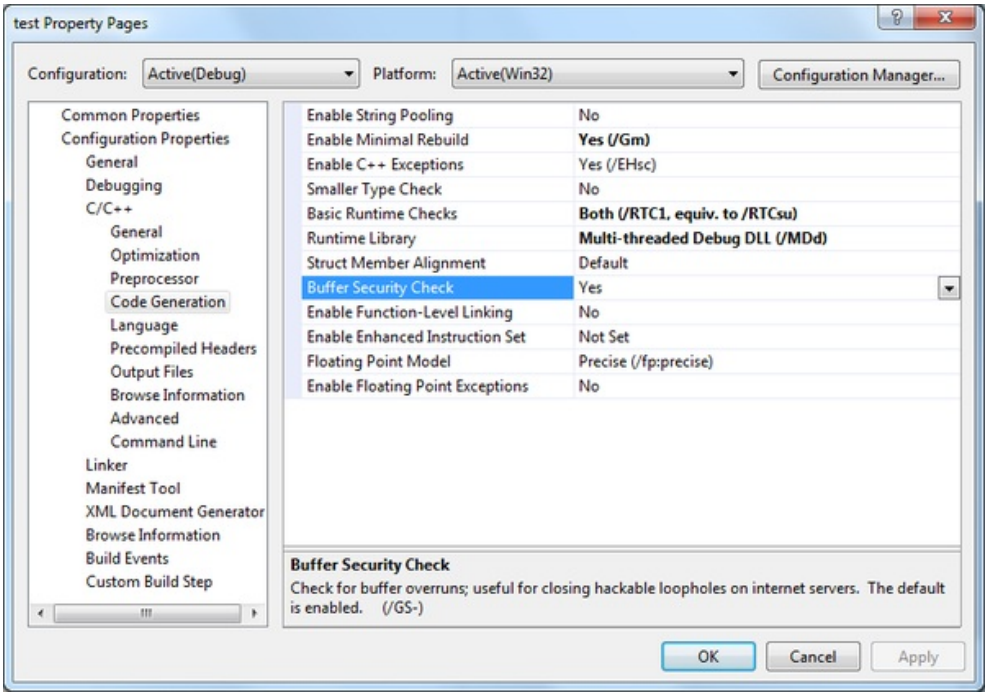
              - 关闭：`/GS-`
        - 代码中配置
          - 文件级别
            - `strict_gs_check` pragma
              - 语法
                - `#pragma strict_gs_check( [ push, ] { on | off } )`
                - `#pragma strict_gs_check( pop )`
              - 举例：

```
// pragma_strict_gs_check.cpp
// compile with: /c

#pragma strict_gs_check(on)

void ** ReverseArray(void **pData,
                     size_t cData)
{
    // *** This buffer is subject to being overrun!! ***
```

```
        void  pReversed[20];

        // Reverse the array into a temporary buffer
        for (size_t j = 0, i = cData; i ; --i, ++j)
            // *** Possible buffer overrun!! ***
                pReversed[j] = pData[i];

        // Copy temporary buffer back into input/output buffer
        for (size_t i = 0; i < cData ; ++i)
            pData[i] = pReversed[i];

        return pData;
    }
```

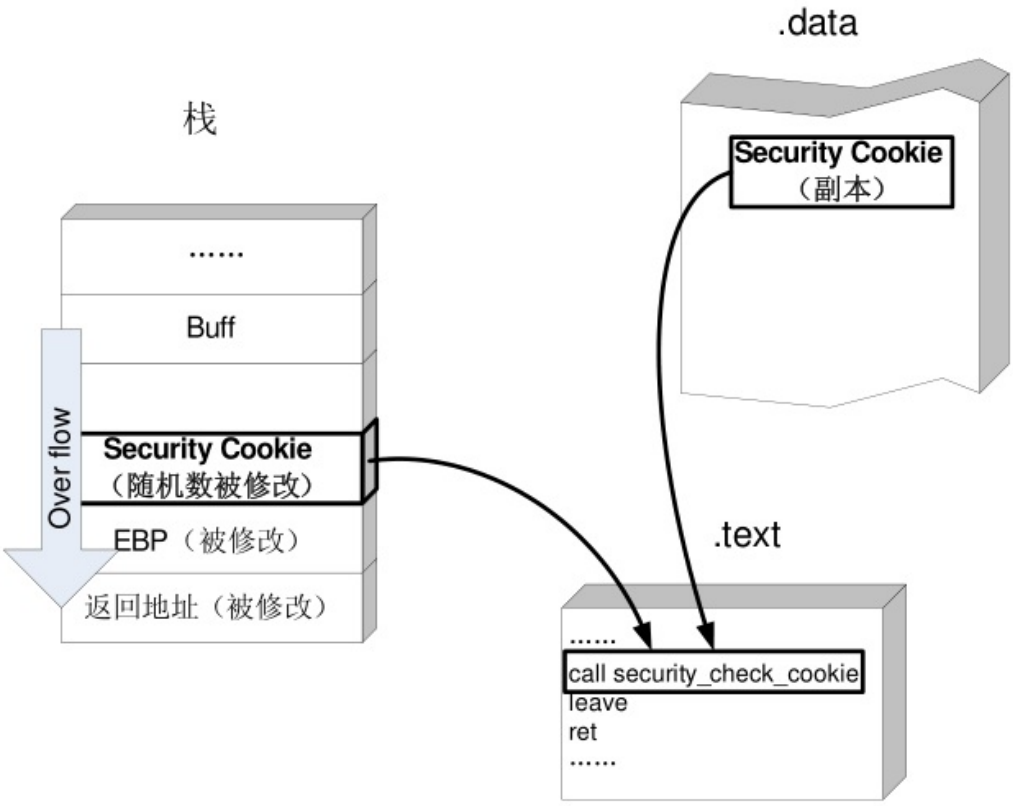
- 函数级别
  - 作用：指定某个函数不需要安全缓存检查
  - 使用场景：你自己是个专家，会用手动做代码检查或者用其他手段确保代码很安全，无需检查
  - 语法：`__declspec(safebuffers)`
  - 举例：

```
// compile with: /c /GS
typedef struct {
    int x[20];
} BUFFER;
static int checkBuffers() {
    BUFFER cb;
    // Use the buffer...
    return 0;
};
static __declspec(safebuffers)
    int noCheckBuffers() {
    BUFFER ncb;
    // Use the buffer...
    return 0;
}
int wmain() {
    checkBuffers();
    noCheckBuffers();
    return 0;
}
```

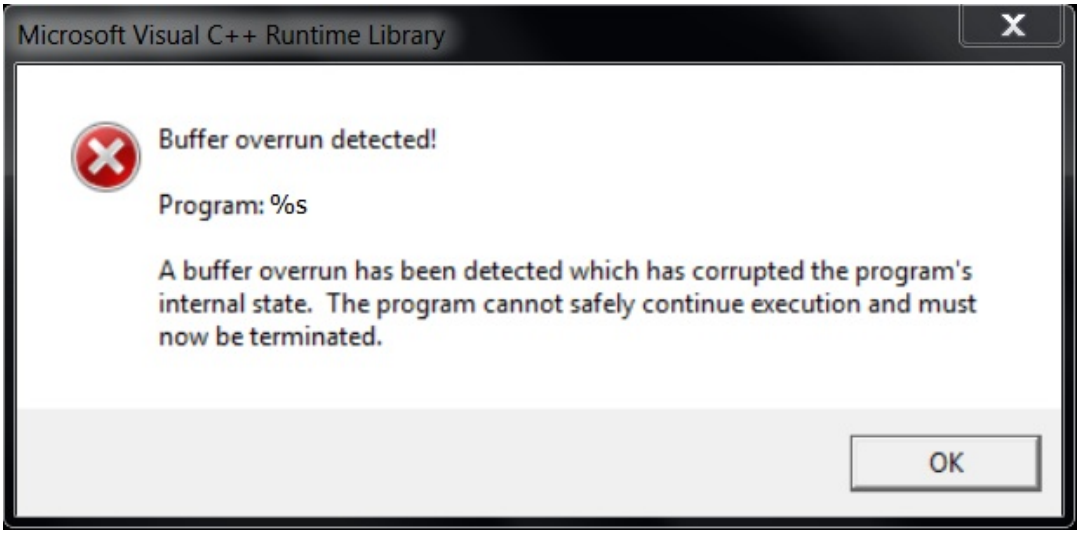
- UI界面中配置
- 

- Intel

        - 编译器：`Intel C++ Compiler`
            - 参数
                - `Windows`：
                    - 语法：`/GS[:keyword]`
                        - `keyword`：GS的`level`
                            - `off` -> `/GS[:off]`：忽略，关闭GS
                                - = `/GS-`
                            - `partial` -> `/GS[:partial]`：用`Microsoft Visual Studio 2008`的标准=level
                            - `strong` -> `/GS[:strong]`：提供完整的安全检查，兼容最新版`Microsoft Visual Studio`的标准
                                - = `/GS`
                - `Linnux` / `macOS`
                    - 开启：`-fstack-security-check`
                    - 关闭：`-fno-stack-security-check`
    - 实现原理
        - 图解

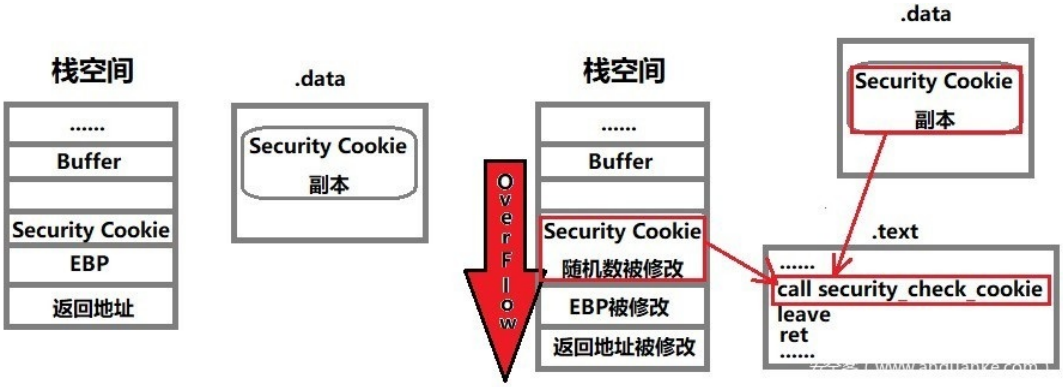


        -
        - 文字解释
            - 内部是利用`GS缓存区` = `GS Buffers`实现缓冲区溢出的检测
            - 其在编译器检测到的易受缓冲区溢出攻击的函数中创建`security cookie`
                - 如果攻击者写的代码，超过缓冲区长度，覆盖了`返回地址`、`异常处理程序的地址`、`易受攻击的函数参数`，则运行时(runtime)会覆盖`security cookie`。
                    - 运行时会在允许执行代码跳转到此地址或返回这些函数参数之前，检测`cookie`的完整性，以避免攻击
    - 使用效果
        - 运行时如果发生缓存区溢出会报错



如何绕开GS

- 利用未被保护的内存
  - 系统为了将GS对性能的影响降到最小, 并不是所有的函数都会保护. 例如, 一个函数中不包含4字节以上的缓冲区时, 即使GS处于开启状态, 这个函数也是不受保护的. 因此, 可以针对这类函数, 构造巧妙的shellcode进行溢出
- 通过猜测cookies值
  - GS保护机制采用了几个较弱的熵源，攻击者可以对其进行计算并使用计算结果来预测cookie值，但是这种犯法只适用于针对本地系统的攻击（攻击者拥有该机器的访问权限）
    - 论文链接
      - http://uninformed.org/?v=7&a=2&t=pdf
- 覆盖虚函数(指针)
  - 程序只有在函数返回时, 才会去检查 `Security Cookie` ， 而我们在程序检查 `Security Cookie` 之前劫持程序流程的话，就可以实现对程序的溢出。例如使用 `C++` 的虚函数溢出即可实现上述功能
    - 校验Security Cookie



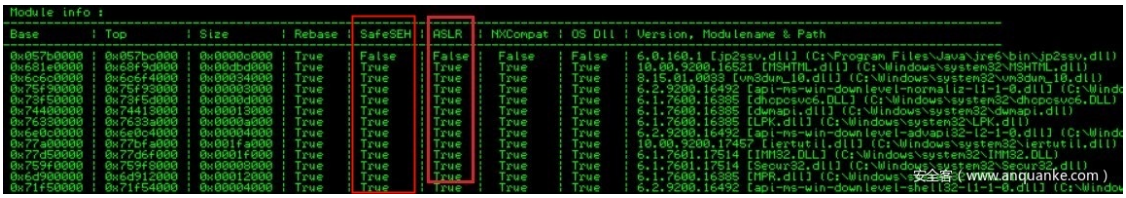
- 攻击异常处理
  - GS机制没有对 `SEH` 提供保护, 因此可以通过攻击程序的异常处理机制达到绕过GS的目的. 通过构造超长的字符串覆盖掉异常处理指针, 然后触发一个异常, 程序就会转入异常处理; 由于异常处理指针已被覆盖, 因此可以通过劫持SEH来控制程序的后续流程.
- 同时替换栈和 `.data` 中的 `Cookie`
  - 若要在 `Security Cookie` 正常工作的情况下实现绕过, 由于 `Cookie` 具有很强的随机性, 难以猜测, 所以只能同时替换栈和 `.data` 中的 `Cookie` 实现绕过. 构造特殊的 `shellcode` 使用相同的值覆盖栈和 `.data` 中的 `Cookie` , 即可实现GS绕过.

相关知识

gcc编译器参数：`fstack-protector`

- gcc通用编译参数
  - `fstack-protector`
    - 功能：开启或关闭某些或所有函数的栈溢出的安全检查
    - 语法
      - `Linux` / `macOS`
        - 语法
          - 开启：`-fstack-protector[-keyword]`
          - 关闭：`-fno-stack-protector[-keyword]`
        - 解释
          - `keyword`
            - `strong` -> `-fstack-protector-strong`：任何类型的缓存(any type of buffer)都进行栈的溢出的安全检查
            - `all` -> `-fstack-protector-all`：每个函数(every routine)都进行安全检查
            - 无参数-> `-fstack-protector`：对于每个字符串缓存(string buffer)的栈溢出进行安全检查
    - 内部实现
      - 优先用 `gcc/glibc` 的实现
        - 如果没有，其次用 `Intel` 的实现
          - 等价于 `-fstack-security-check`

SafeSEH

- `SafeSEH`
  - = `Safe Exception Handlers`
  - 作用：异常处理保护
    - 如果开启了 `SafeSEH` ，那么编译器只会生成一个镜像，其 `安全异常处理程序` 的 `静态表`
      - 用于告诉操作系统，此镜像文件的异常处理程序在哪
  - 目的：避免了攻击者重写异常处理程序的控制流程
  - 语法
    - 开启： `/SAFESEH`
    - 关闭： `/SAFESEH:NO`
  - 详解
    - 在Win XP SP2及后续版本的系统中，微软引入了著名的S.E.H校验机制SafeSEH。原理很简单，在程序员调用异常处理函数前，对要调用的异常处理函数进行一系列的有效性校验，当发现异常处理函数不可靠时将终止异常处理函数的调用。 SafeSEH实现需要操作系统与编译器的双重支持，二者缺一都会降低SafeSEH的保护能力。在VS2003及后续版本中默认启用。
  - 注：
    - 只支持： `x86` 平台
    - 不支持：那些已经标注了异常处理程序的平台
      - 举例
        - `x64`
          - 相关工具
            - `ml64.exe` = `x64` 的 `Microsoft Assembler`
              - 支持给程序生成 `SEH` 信息(`XDATA` 和 `PDATA`)信息
        - `ARM`
      - 这些平台：异常处理程序都已被放到了固定的 `PDATA`
  - 举例
    - 

SEHOP

- `SEHOP` = `Structured Exception Handler Overwrite Protection` = `结构化异常处理覆盖保护`
  - 背景
    - `SEH攻击`：指通过栈溢出或者其他漏洞，使用精心构造的数据覆盖结构化异常处理链表上面的某个节点或者多个节点，从而控制EIP（控制程序执行流程）
    - `SEHOP`：是是微软针对这种攻击提出的一种安全防护方案
  - 详解
    - 微软最开始提供这个功能是在2009年，支持的系统包括Windows Vista Service Pack 1、 Windows 7、Windows Server 2008 和 Windows Server 2008 R2，以及它们的后续版本。它是以一种SEH扩展的方式提供的，通过对程序中使用的SEH结构进行一些安全检测，来判断应用程序是否受到了SEH攻击。SEHOP的核心是检测程序栈中的所有SEH结构链表，特别是最后一个SEH结构，它拥有一个特殊的异常处理函数指针，指向的是一个位于NTDLL中的函数。异常处理时，由系统接管分发异常处理，因此上面描述的检测方案完全可以由系统独立来完成，正因为SEH的这种与应用程序的无关性，因此应用程序不用做任何改变，你只需要确认你的系统开启了SEHOP即可。在Windows Server 2008 和 Windows Server 2008 R2下SEHOP默认是开启的，而在Windows Vista Service Pack 1、 Windows 7下默认则是关闭的
    - SEHOP的任务就是检查这条S.E.H链的完整性，在程序转入异常处理前SEHOP会检查S.E.H链上最后一个异常处理函数是否为系统固定的终极异常处理函数。如果是，则说明这条S.E.H链没有被破坏，程序可以去执行当前的异常的处理函数；如果检测到最后一个异常处理函数不是终极BOSS，则说明S.E.H链被破坏，可能发生了S.E.H覆盖攻击，程序将不会去执行当前的异常处理函数。
    - 攻击时将S.E.H结构中的异常处理函数地址覆盖为跳板指令地址，跳板指令根据实际情况进行选择。当程序出现异常的时候，系统会从S.E.H链中取出异常处理函数来处理异常，异常处理函数的指针已经被覆盖，程序的流程就会被劫持，在经过一系列跳转后转入Shellcode执行。
    - 由于覆盖异常处理函数指针时同时覆盖了指向下一异常处理结构的指针，这样的话，S.E.H链就会被破坏，从而被SEHOP机制检测到。
    - 作为对SafeSEH强有力的补充，SEHOP检查是在SafeSEH的RtlIsValidHandler函数校验前进行的，也就是说利用攻击加载模块之外的地址、堆地址和未启用SafeSEH模块的方法都行不通了，必须要考虑其它的方法
  - 突破SEHOP方法
    - 不去攻击S.E.H，而是攻击函数返回地址或虚函数等
    - 利用未启用SEHOP的模块
    - 伪造S.E.H链

攻击技术

缓冲区溢出

- 缓冲区溢出
  - 会导致
    - 覆盖了
      - 函数返回值
        - overwrite a function's return address
      - 异常处理地址
        - exception handler address
      - 某些类型的参数
        - certain types of parameters
  - 黑客
    - 会利用 缓冲区溢出 去对于
      - 没有强制实现缓存大小的限制的那些代码
    - 实现漏洞检测和攻击

缓冲区溢出代码举例：

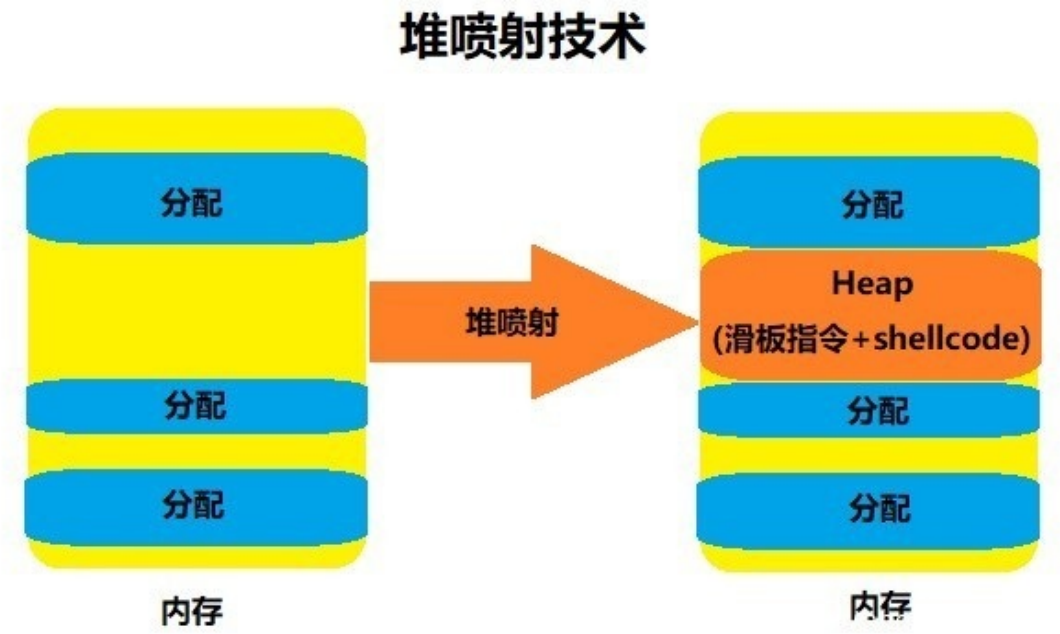
```
// compile with: /c /W1
#include <cstring>
#include <stdlib.h>
#pragma warning(disable : 4996)   // for strcpy use

// Vulnerable function
void vulnerable(const char *str) {
   char buffer[10];
   strcpy(buffer, str); // overrun buffer !!!

   // use a secure CRT function to help prevent buffer overruns
   // truncate string to fit a 10 byte buffer
   // strncpy_s(buffer, _countof(buffer), str, _TRUNCATE);
}

int main() {
   // declare buffer that is bigger than expected
   char large_buffer[] = "This string is longer than 10 characters!!";
   vulnerable(large_buffer);
}
```

ROP

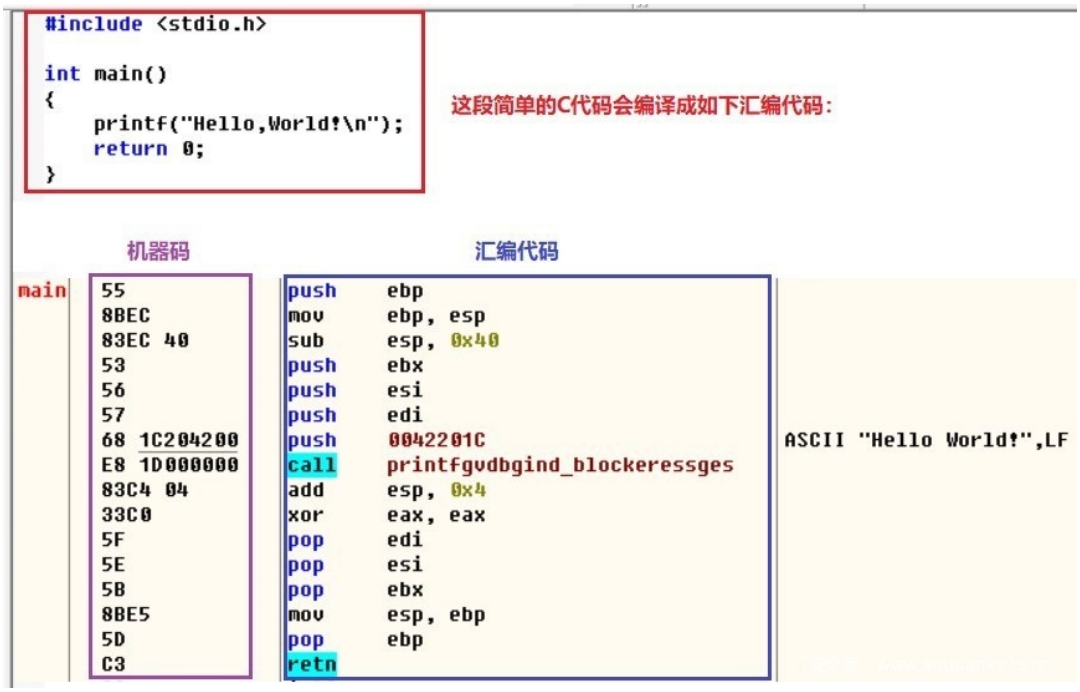
- `ROP` = `Return-Oriented Programming` = `面向返回编程`
  - 是什么：一个攻击手段=技术
    - 借用libc代码段里面的多个retq前的一段指令拼凑成一段有效的逻辑，从而达到攻击的目标
  - 解释
    - 为什么是retq？
      - 因为retq指令返到哪里执行，由栈上的内容决定，而这是攻击者很容易控制的地址
    - 那参数如何控制？
      - 就是利用retq执行前的pop reg指令，将栈上的内容弹到指令的寄存器上，来达到预期
    - 一段retq指令未必能完全到想攻击目标的前提条件，那可在栈上控制retq指令跳到另一段retq指令表，如果它还达不到目标，再跳到另一段retq，直到攻击目标实现
    - 在ret2plt攻击方法，我们使用PPR(pop， pop， ret)指令序列，实现顺序执行多个strcpy函数调用，其实这就是一种最简单的ROP用法，ROP更是ret2plt的升级版
    - ROP方法技巧性很强，那它能完全胜任所有攻击吗？返回语句前的指令是否会因为功能单一，而无法实施预期的攻击目标呢？业界大牛已经过充分研究并证明ROP方法是图灵完备的，换句话说， ROP可以借用libc的指令实现任何逻辑功能

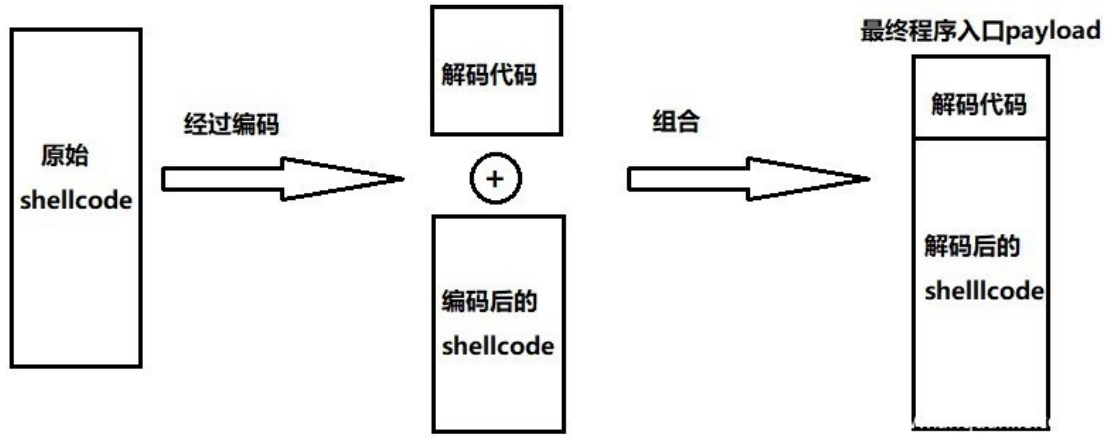
Heap Spray

- `Heap spray` = `堆喷射技术`
  - 是什么：一种payload传递技术
    - 借助堆来将Shellcode放置在可预测的堆地址上，然后稳定地跳入Shellcode
  - 背景
    - 不论是基于栈溢出还是堆溢出的缓冲区漏洞攻击 ，在攻击者成功造成系统溢出后都必须考虑跳转地址（如函数返回点）的覆盖。在以往的攻击中，如何确定Shellcode在内存中位置，使得跳转地址被覆盖为Shellcode的起始地址是攻击者需要精确计算的。并且这往往也是攻击中最难以实现的部分。另外，考虑到一些外部环境因素比如操作系统版本的不同，使得溢出攻击变得更加难以实现
  - Heap spray技术大大缓解了这一问题
    - 起初，技术人员发现可以通过Javascript申请大量的堆内存来消耗资源，造成目标主机的瘫痪。但是并没有进一步利用。直到后来，SkyLined在2004年为IE的IFRAME漏洞所写的exploit中才第一次正式提出了Heap spray。
    - 之后经过不断发展，Heap spray逐渐成为网页挂马的常用技术，并且被利用到文档攻击中（如PDF阅读器）
  - 原理



Shellcode

- `Shellcode` = `脚本代码`
  - 是什么：Shellcode是一小段代码 = 载荷
  - 作用：实现利用软件漏洞
  - 名称由来：被称为 `Shellcode` 是因为它通常启动一个命令终端=shell
    - 攻击者可以通过这个终端控制受害的计算机
  - 常见形式：（用）机器码（编写shellcode）
  - 重点：要懂汇编
    - 举例
    - 

  - Shellcode编码与解码
    - 

Windows安全工具

此处介绍Windows在二进制安全方面的工具、软件、框架等。

- Windows中最常用的几个调试工具
  - 应用层动态调试工具：`OD` = `OllyDbg` = `Olly DBG`
  - 强大的Windows调试工具：`WinDBG` = `WinDbg`
  - 漏洞分析专用调试器：`Immunity Debugger`
  - 开源的安全漏洞检测工具：`Metasploit`
  - 跨平台
    - 神级反汇编工具：`IDA Pro`
    - `LLDB`

其他细节：

- CAIN
  - 是什么：Oxid.it开发的一个针对Microsoft操作系统的免费口令恢复和网络嗅探测试工具
  - 特点：在口令破解上很有一套技术
  - 功能
    - 网络嗅探，网络欺骗，破解加密口令、解码被打乱的口令、显示口令框、显示缓存口令和分析路由协议，甚至还可以监听内网中他人使用VOIP拨打电话等。

下面详细介绍。

反汇编器

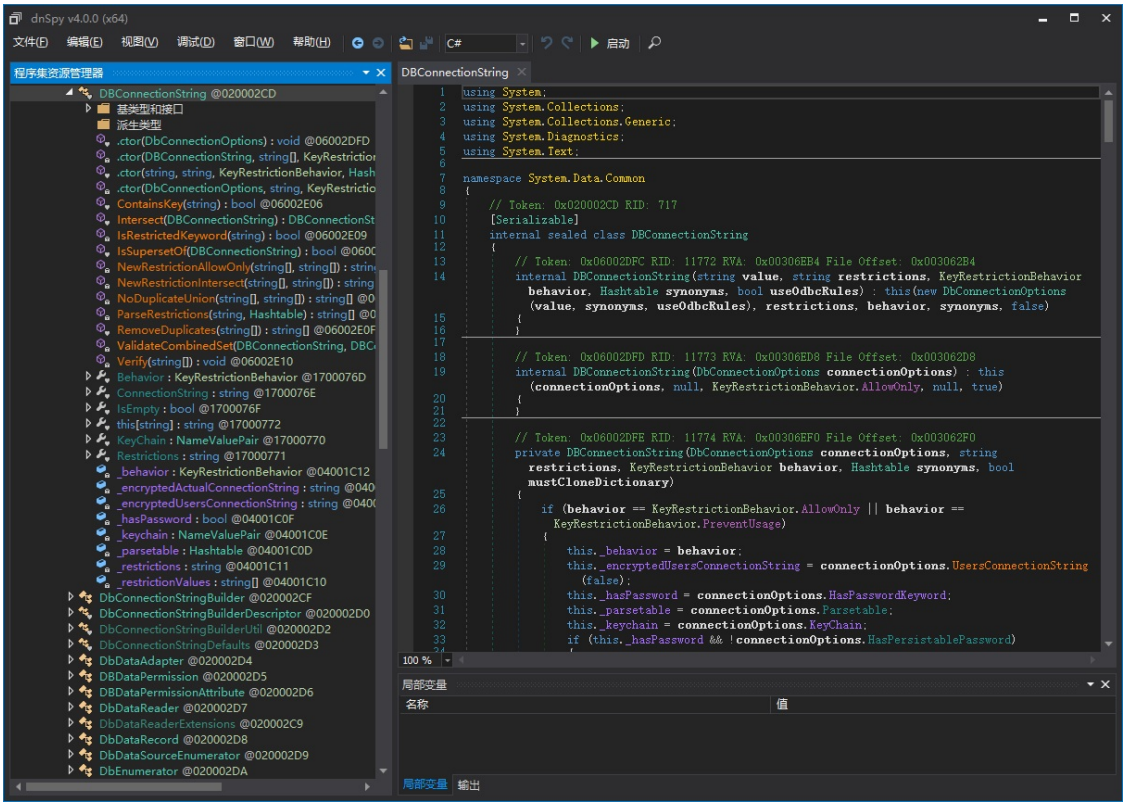
- `disassembler` = `反汇编`
  - `反汇编引擎/框架`
    - 名称
      - `反汇编引擎` = `disassembler engine`
      - `反汇编框架` = `disassembler framework`
    - 常见
      - `llvm`
      - `capstone`
        - 详见独立子教程：Capstone
      - `Ollydbg` 的 `ODDisasm`
        - `ODDisasm` = `Ollydbg Disassembler` = Disasm()
        - 官网
          - Assembling and disassembling
      - `BeaEngine`
      - `udis86`
      - `XDE`
    - 对比
      - udis86 vs BeaEngine vs capstone
        - 细节
          - 性能：`udis86 > BeaEngine > capstone`
          - 解码能力：`capstone > BeaEngine > udis86` (udis86不支持寄存器分析,其余解码能力是相近的)
          - 平台支持：`capstone > (udis86 = BeaEngine)`
          - X86扩展指令集：`capstone` > (`udis86` ≈ `BeaEngine`)
        - 结论
          - 如果你需要的是一个X86/64下的性能又好同时解码能力又强的反汇编引擎，并且不需要寄存器分析这种特技的话，那么udis86合适你
          - 如果你还需要带寄存器分析功能的话，那么BeaEngine与capstone合适你；如果你还需要ARM构架支持的话，capstone应该会更适合你

udis86

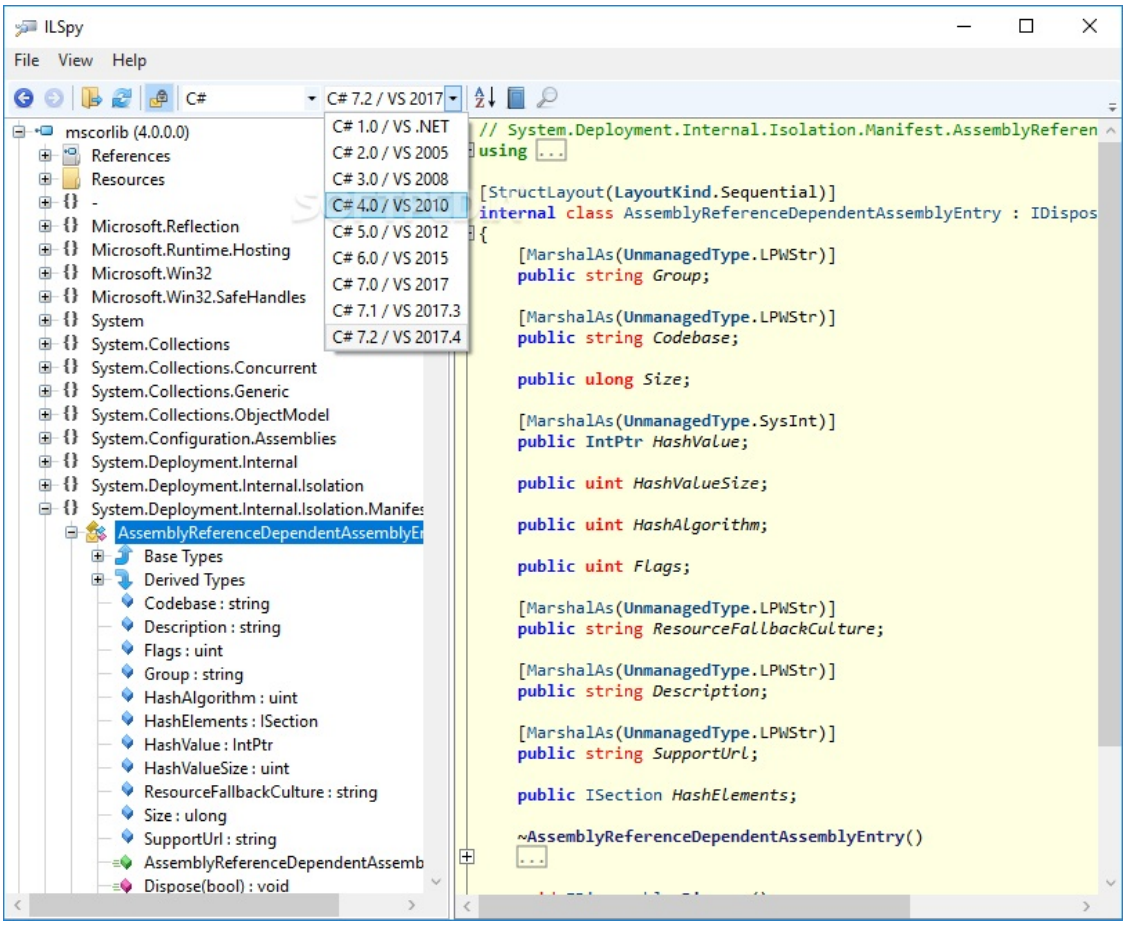
- `udis86`
  - 是什么：X86架构的反汇编工具
    - `反汇编工具` = `反汇编器` = `反汇编框架`
  - 一句话描述：Udis86 is an easy-to-use, minimalistic disassembler library (libudis86) for the x86 class of instruction set architectures. It has a convenient interface for use in the analysis and instrumentation of binary code.
  - 概述
    - Udis86 is a disassembler for the x86 and x86-64 class of instruction set architectures. It consists of a C library called libudis86 which provides a clean and simple interface to decode a stream of raw binary data, and to inspect the disassembled instructions in a structured manner.
    - udis86支持的X86扩展指令集有：MMX, FPU (x87), AMD 3DNow, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, AES, AMD-V, INTEL-VMX, SMX
    - Udis86提供了一套反汇编的第三方库，用户可自行编写代码进行扩展。udcli是基于Udis86的反汇编工具，集成在Udis86项目里，通过命令行可快速实现反汇编
  - 主页
    - GitHub
      - https://github.com/vmt/udis86

反编译器

dnSpy

- `dnSpy`
  - 一句话描述：a debugger and .NET assembly editor
  - 特点：无需源码也能修改`.NET`程序
  - 功能
    - 功能组件
      - 编译器
      - 调试器
      - 汇编编辑器
    - 支持编写插件扩展功能
  - 截图
    - 
  - 资料
    - Github
      - 0xd4d/dnSpy: .NET debugger and assembly editor
        - https://github.com/0xd4d/dnSpy

ILSpy

- `ILSpy`
  - 截图
    - 

  - 资料
    - Github
      - icsharpcode/ILSpy: .NET Decompiler with support for PDB generation, ReadyToRun, Metadata (&more) - cross-platform!
        - https://github.com/icsharpcode/ILSpy

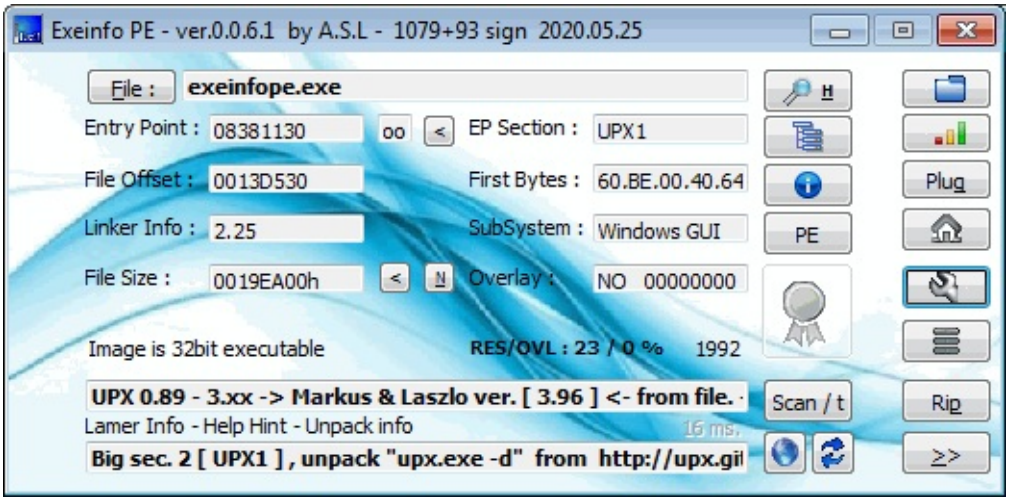
静态安全检工具

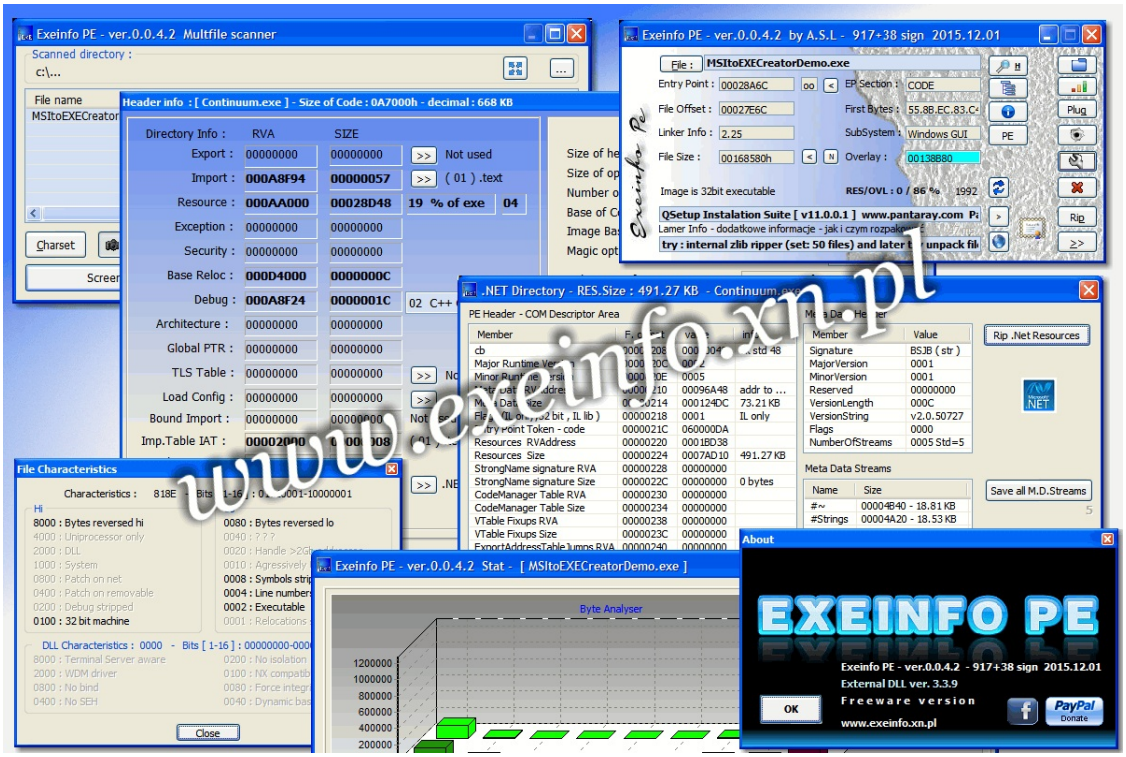
wincheckseс

- `wincheckseс`
  - 作用：Windows静态安全检测工具
  - 支持检测如下 `安全特性`
    - `ASLR`
      - `/DYNAMICBASE` with stripped relocation entries edge-case
      - `/HIGHENTROPYVA` for 64-bit systems
    - `Code integrity/signing:`
      - `/INTEGRITYCHECK`
      - Authenticode-signed with a valid (trusted, active) certificate (currently unsupported on Linux)
    - `DEP`
      - 别称：`W^X`，`NX`
    - `Manifest isolation`
      - `/ALLOWISOLATION`
    - `SEH` 和 `SafeEH`
      - `SEH` = `Structured Exception Handling`
    - `Control Flow Guard` 和 `Return Flow Guard instrumentation`
    - `Stack cookie`
      - `/GS`
  - 资料
    - Github
      - trailofbits/wincheckseс: Checksec, but for Windows: static detection of security mitigations in executables
        - https://github.com/trailofbits/wincheckseс

可执行文件工具

Exeinfo PE

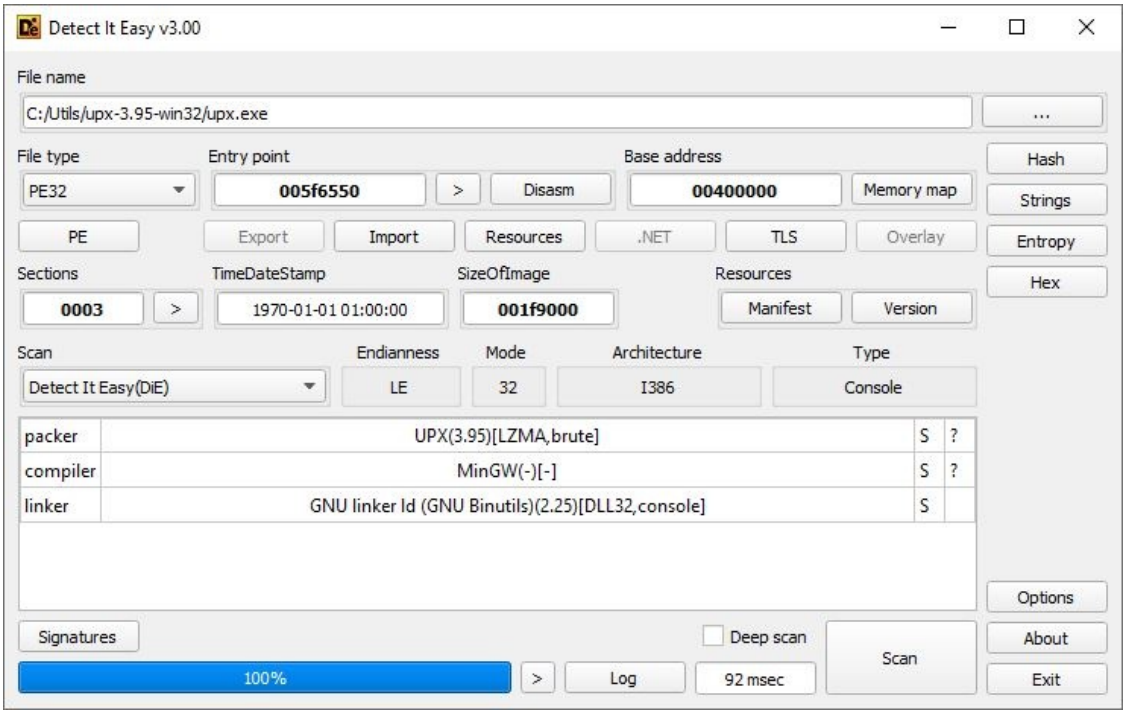
- `Exeinfo PE`
  - 一句话描述：Packer compressor detector / unpack info / internal exe tools
  - 支持平台
    - Android
    - Linux
    - Mac
  - 截图
    - 

    - 

  - 资料
    - 官网
      - Exeinfo PE by A.S.L - compression detector and data detector
        - http://www.exeinfo.xn.pl
      - mirror
        - http://exeinfo.byethost18.com

查壳工具

Detect It Easy

- `Detect It Easy`
  - 简称：`DIE`
  - 截图
    - 

  - 资料
    - GitHub
      - horsicq/Detect-It-Easy: Program for determining types of files for Windows, Linux and MacOS.
        - https://github.com/horsicq/Detect-It-Easy
    - 官网
      - .:NTInfo:.
        - http://ntinfo.biz/index.html

PEiD

- `PEiD`
  - 概述
    - PEiD(PE Identifier)是一款著名的查壳工具，其功能强大，几乎可以侦测出所有的壳，其数量已超过470种PE文档的加壳类型和签名
  - 扫描模式
    - 正常扫描模式：可在PE文档的入口点扫描所有记录的签名
    - 深度扫描模式：可深入扫描所有记录的签名，这种模式要比上一种的扫描范围更广、更深入
    - 核心扫描模式：可完整地扫描整个PE文档，建议将此模式作为最后的选择。PEiD内置有差错控制的技术，所以一般能确保扫描结果的准确性。前两种扫描模式几乎在瞬间就可得到结果，最后一种有点慢，原因显而易见
  - 命令选项
    - peid -time
      - 显示信息
    - peid -r
      - 扫描子目录
    - peid -nr
      - 不扫描子目录
    - peid -hard
      - 采用核心扫描模式
    - peid -deep
      - 采用深度扫描模式
    - peid -norm
      - 采用正常扫描模式

逆向工具

IDA

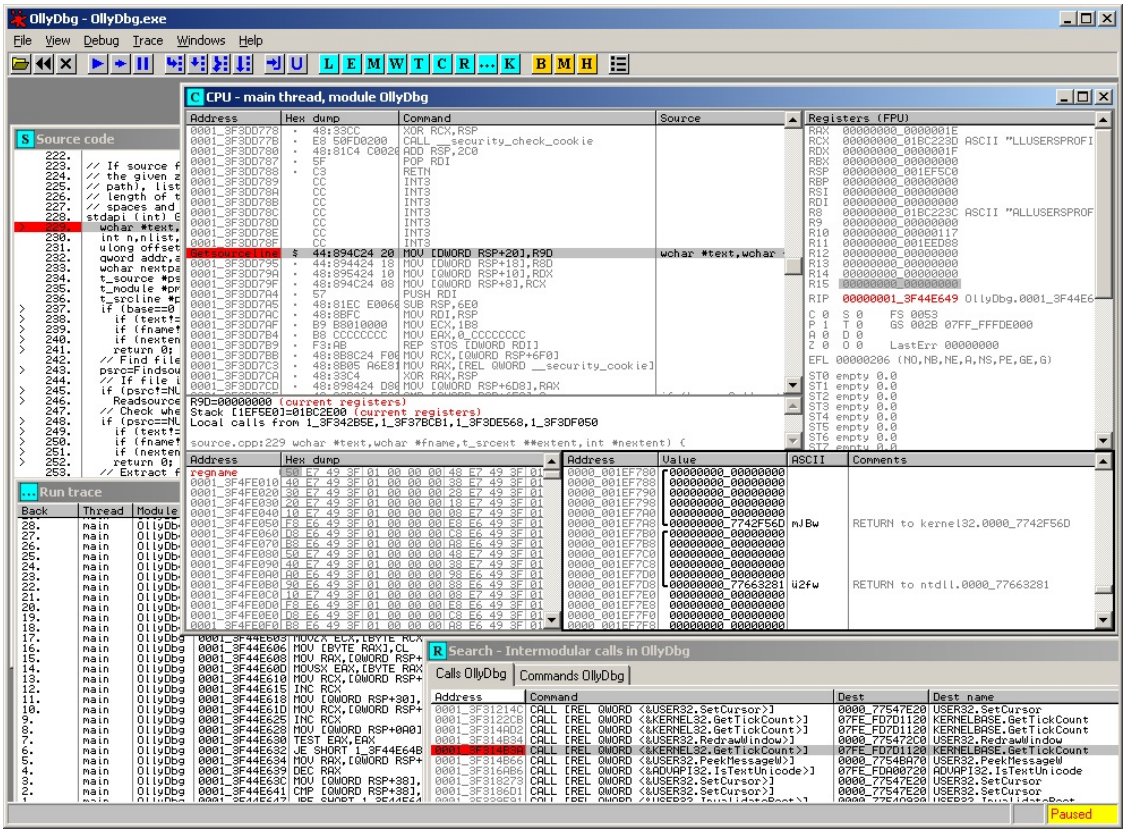
- `IDA`
  - = `IDA Pro`
  - 支持平台
    - `Windows`
    - `Mac`
    - `Linux`
  - 多个维度看
    - IDA Pro is a disassembler
      - A disassembler is a piece of software used to translate machine code into a human readable format called assembly language
    - IDA Pro is a debugger
      - A debugger is a computer program that assists in the detection and correction of errors in other computer programs
    - IDA Pro is interactive
    - IDA Pro is programmable
      - A plugin architecture allows a program to call external code at certain points without knowing all the details of that code in advance, therefore adding functionalities to the calling program
  - 用途
    - Hostile Code analysis
    - Vulnerability research
    - Commercial-off-the-shelf (COTS) validation
    - Privacy protection
    - Other uses
  - 评价
    - IDA是一个世界顶级的Ring3交互式反汇编工具
    - 是最强大的静态分析工具(也可动态调试)
    - 在分析整体程序流程、算法方面非常有优势
  - 应用举例
    - 游戏破解
      - 即使不能过掉反外挂系统，也可以用dump下游戏内存静态分析出游戏明文收发包等重要函数

- 详见独立子教程
  - 逆向利器：IDA

Ollydbg

TODO:

- 在看雪上有人翻译了国外人写的<使用ollydbg从零开始cracking>，还是挺详细的介绍了od的使用，推荐阅读和动手跟一遍。
  - 使用OllyDbg从零开始Cracking(已完结)-『外文翻译』-看雪安全论坛
    - https://bbs.pediy.com/thread-184679.htm

- `Ollydbg`
  - 一句话描述：OllyDbg is a `32-bit` = `x86` assembler level analysing debugger for Microsoft® Windows®. Emphasis on binary code analysis makes it particularly useful in cases where source is unavailable
  - 功能和特点
    - It traces registers, recognizes procedures, API calls, switches, tables, constants and strings, as well as locates routines from object files and libraries.
    - It has a user friendly interface, and its functionality can be extended by third-party plugins
  - 名字
    - `Olly` 来源于作者： `Oleh Yuschuk`
  - 评价
    - OllyDbg是windows平台下Ring 3级调试器
    - 界面友好
    - 非常容易上手
    - 支持插件扩展功能
    - 是当今最流行最强大的动态调试工具
  - 截图
    - 

  - 应用举例
    - 游戏破解

        - 目前做辅助的基本均用此调试器分析需要的资源结构、内存数据、功能函数用于开发辅助功能
    - 资料
        - 官网
            - v1
                - OllyDbg v1.10
                    - http://www.ollydbg.de
            - v2
                - OllyDbg 2.0
                    - http://www.ollydbg.de/version2.html

WinDbg

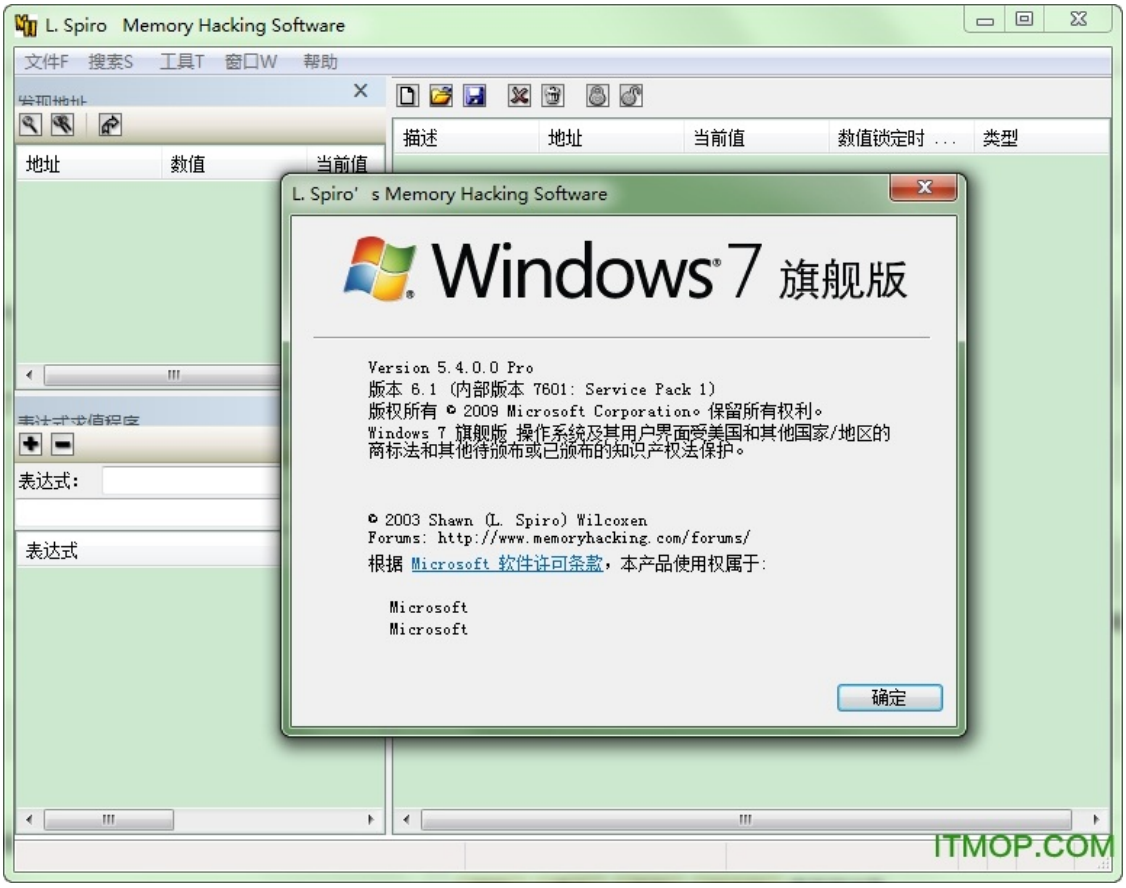
- `WinDbg`
  - 评价
    - 是windows平台下强大的Ring3和Ring0调试工具
  - 应用举例
    - 在游戏漏洞挖掘中，主要用于调试分析反外挂驱动的保护点，以便写出反反外挂驱动

内存修改工具

CE

- `CE`
  - = `Cheat Engine`
  - 一句话简介：内存修改编辑工具
  - 说明
    - a development environment focused on modding games and applications for personal use
  - 资料
    - GitHub
      - cheat-engine/cheat-engine: Cheat Engine. A development environment focused on modding
        - https://github.com/cheat-engine/cheat-engine
    - 官网
      - Cheat Engine
        - https://www.cheatengine.org

MHS

- `MHS`
  - = `Memory Hacking Software`
  - 截图
    - 

  - 主要功能
    - Searching
    - View RAM In Real-Time
    - Debugging
    - Disassembling
    - Inject Code
    - Inject DLL’s
    - Scripting
    - Hotkeys
    - Real-Time Expression Evaluator
  - 资料
    - 官网
      - L. Spiro's Memory Hacking Software
        - http://memoryhacking.com
      - 下载
        - http://memoryhacking.com/download.php

ModifyMemory

二进制编辑工具

010Editor

- `010Editor`
  - 概述
    - 010Editor是一个专业的文本编辑器和十六进制编辑器，旨在快速轻松地编辑计算机上任何文件的内容

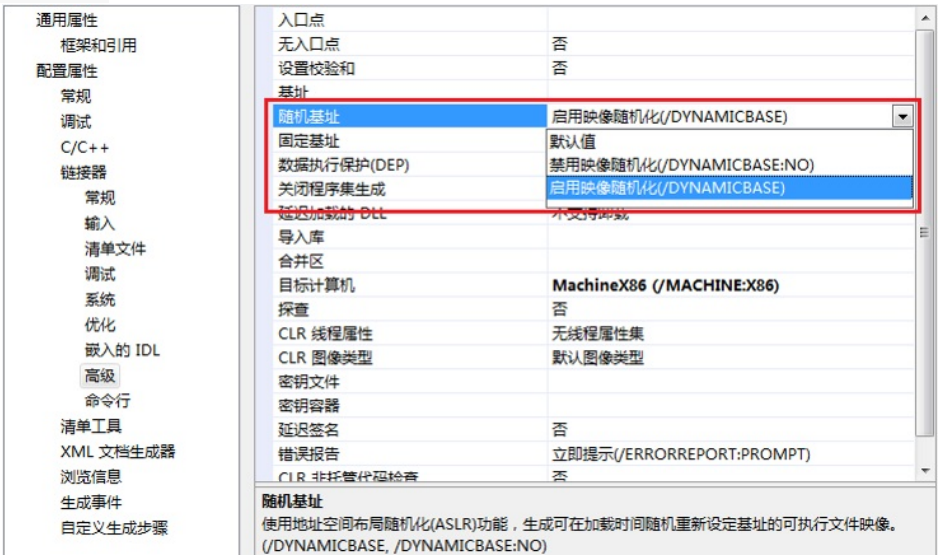
跨平台

此处介绍支持多个平台 = 跨平台 的一些，和二进制相关的内容。

安全机制

此处介绍一些跨平台的通用的安全机制。

ASLR

- `ASLR`
  - = `Address Space Layout Randomization` = 地址空间布局随机化
  - 是什么：是一种针对缓冲区溢出的安全保护技术
  - 背景
    - 没有ASLR时
      - 每次进程执行，加载到内容中，代码所处堆栈stack的位置都是相同的
        - 容易被识别出所在位置
          - 容易被破解
  - 如果开启了 `ASLR` =机制：
    - 操作系统加载器会针对基地址再去加上一个随机生成的偏移地址，然后再去加载程序模块
      - = 借助ASLR，PE文件每次加载到内存的起始地址都会随机变化
      - 通过对堆、栈、共享库映射等线性区布局的随机化
        - 增加攻击者预测目的地址的难度
          - 防止攻击者直接定位攻击代码位置，达到阻止溢出攻击的目的
  - 提示
    - （虚拟地址）此技术需要操作系统和软件相配合
  - 目的
    - 对付破解的一种有效的手段
    - 让程序被破解更加难
      - 破解程序一般指的是，运行 `shellcode`
  - 系统支持ASLR的情况
    - `Linux`
      - `FreeBSD`
    - `Mac`
    - `Windows`
      - `PE` 头文件中会设置 `IMAGE_DLL_CHARACTERISTICS_DYNAMIC_BASE` 标示来说明其支持 `ASLR`
  - 如何开启
    - Windows
      - 语法：
        - 开启：`/DYNAMICBASE`
          - `Visual Studio` 项目属性的配置
            - 

        - 关闭： `/DYNAMICBASE:NO`
    - 使用此技术后，杀死某程序后重新开启
        - `Linux` ： 地址会改变
        - `Windows` ： 地址不会改变，重启系统才会改变
    - ASLR主要影响几种部分
        - 模块随机化
        - 堆栈随机化
        - PEB/TEB随机化
- 相关
    - 目前大部分主流操作系统都已经实现了ASLR
        - `PIE` VS `ASLR`
            - `Linux`
                - `PIE` = `Position-Independent Execute` = `地址无关可执行文件`
                    - 编译时
                        - 将程序编译为位置无关
                    - 地址随机化针对：代码段和数据段(`.data` 段 `.bss` 段)
                - ASLR：
                    - 地址随机化针对：其他内存地址
            - `Linux` 的 `ASLR` + `PIE` 作用 == `Window` 下 `ASLR` 的作用

如何绕过ASLR

- 攻击未启用ASLR的模块
    - 虽然有映像随机化，但有可能进程中存在未启用ASLR的模块。 前面提到的 `ROP` 技术要求从一个固定的地址获取 `Gadget` ，如果进程中存在未启用 `ASLR` 的模块，那么就可以从那个模块获取 `Gadget` 了。 使用 `OD` 的 `OllyFindAddr` 插件可以快速找到进程空间中未启用ASLR的模块
- 堆喷射（ `HeapSpray` ）技术
    - 虽然有堆栈随机化，不过 `HeapSpray` 技术将 `ShellCode` 布局到 `0x0C0C0C0C` （或者其他指定的地址上，通常这个地址要比较大），并不会受堆栈随机化的影响。 其实， `HeapSpray` 中使用 `ROP` 绕过 `DEP` 的时候，就使用了前面提到的 `攻击未启用ASLR的模块` 。 只是， `HeapSpray` 把 `ShellCode` 布局在堆上
- 覆盖部分返回地址
    - 映像随机化中，虽然模块的加载基地址发生变化，但是各模块的入口点地址的低位字不变，只有高位字进行了随机化处理。
    - 对于地址 `0x12345678` ，其中 `5678` 部分是固定的，如果存在缓冲区溢出，可以通过 `memcpy` 对后两个字节进行覆盖，可以将其设置为 `0x12340000 ~ 0x1234FFFF` 中的任意一个值。
    - 如果通过 `strcpy` 进行覆盖，因为 `strcpy` 会复制末尾的结束符 `0x00` ，那么可以将 `0x12345678` 覆盖为 `0x12345600` ，或者 `0x12340001 ~ 0x123400FF`
    - 部分返回地址覆盖，可以使得覆盖后的地址相对于基地址的距离是固定的，可以从基地址附近找可以利用的跳转指令
    - 这种方法的通用性不是很强，因为覆盖返回地址时栈上的 `Cookie` 会被破坏。不过具体问题具体分析，为了绕过操作系统的安全保护机制需要考虑各种各样的情况
- Java Applet Spray
    - Java Applet中动态申请的内存空间具有可执行属性（PAGE_EXECUTE_READWRITE），类似HeapSpray技术，

可以在固定的地址上分配滑板指令(如NOP)和ShellCode，然后跳转到那个地址上面去执行。 和常规的HeapSpray不同，Applet申请空间的上限为100MB，而常规的HeapSpray可以达到1GB

- JIT Spray
  - JIT (Just In Time Compilation) 即时编译，也就是解释器（比如Python解释器）。主要思想是将 ActionScript代码中进行大量的XOR操作。然后编译成字节码，并且多次更新到Flash VM中，这样它会建立很多带有恶意Xor操作的内存块
- Tombkeeper在CanSecWest 2013上提出的基于SharedUserData的方法

附录

下面列出相关参考资料。

参考资料

- 认识二进制安全与漏洞攻防技术 （Windows平台） - 安全客，安全资讯平台
- 我的安全之路——二进制与逆向篇_giantbranch的专栏-CSDN博客_pwn和逆向的区别
- 二进制安全如何入门以及未来的就业前景如何？ - 知乎
- 浅谈二进制漏洞研究与病毒研究 - 安全客，安全资讯平台
- /NXCOMPAT (Compatible with Data Execution Prevention) | Microsoft Docs
- Control Flow Guard - Win32 apps | Microsoft Docs
- safebuffers | Microsoft Docs
- 内存保护机制及绕过方案——通过覆盖虚函数表绕过/GS机制 - zhang293 - 博客园
- Windows安全机制---栈保护：GS机制_每昔的博客-CSDN博客_charshellcode
- 绕过GS的栈溢出攻击原理 | Kumqu’s Blog
- 绕过GS安全编译的方法 | Introspelliam
- 通关栈溢出（四）：缓冲区溢出的防御技术及绕过 - Hello! CytQ
- visual studio - safeseh gs on g++ - Stack Overflow
- SafeSEH利用（DEP/ASLR disabled） - 简书
- gdbinit/MachOView: MachOView fork
- KEYSTONE: Next Generation Assembler Framework
- 神器如 dnSpy，无需源码也能修改 .NET 程序_walterlv - 吕毅-CSDN博客_dnspy
- dnSpy - 一款 .NET 程序逆向工具]
- 使用dnSpy对目标程序(EXE或DLL)进行反编译修改并编译运行 - jack_Meng - 博客园
- 工具推荐：逆向破解利器OllyDbg - 知乎
- OllyDbg - Wikipedia
- Download OllyDbg 2.01
- 七周年礼物第五弹之一：吾爱破解专用版Ollydbg 2016年1月21日更新
- OllyDbg Download (2020 Latest) for Windows 10, 8, 7
- OllyDbg使用入门 | M0rk's Blog
- Microsoft Visual C++ - 维基百科，自由的百科全书
- windows - How to disable buffer overflow checking in the Visual C++ Runtime? - Stack Overflow
- Intel® C++ Compiler 19.1 Developer Guide and Reference
- /GS (Buffer Security Check) | Microsoft Docs
- MSVC Compiler Options | Microsoft Docs
- GS buffer
- fstack-protector
- OllyDbg使用入门 | M0rk's Blog
- 漏洞挖掘工程师
- strict_gs_check pragma | Microsoft Docs
- trailofbits/winchecksec: Checksec, but for Windows: static detection of security mitigations in executables
- c++ - How to check whether an EXE has /GS security protection on Windows? - Stack Overflow
- IMAGE_LOAD_CONFIG_DIRECTORY32 (winnt.h) - Win32 apps | Microsoft Docs
- GS cookie protection – effectiveness and limitations - Microsoft Security Response Center
- Security Features in MSVC | C++ Team Blog
- /SAFESEH (Image has Safe Exception Handlers) | Microsoft Docs
- MASM for x64 (ml64.exe) | Microsoft Docs
- On the effectiveness of DEP and ASLR - Microsoft Security Response Center
- exploit - How do ASLR and DEP work? - Information Security Stack Exchange
- DEP/ASLR 原理及攻击_运维_forevertingting的博客-CSDN博客
- windows - How to bypass DEP and ASLR at the same time? - Information Security Stack Exchange
- buffer overflow - How does SEH based exploit bypass DEP and ASLR? - Information Security Stack Exchange
- exploit - Stack Overflows - Defeating Canaries, ASLR, DEP, NX - Information Security Stack Exchange
- buffer overflow - Bypass Full ASLR+DEP exploit mitigation - Information Security Stack Exchange

- /DYNAMICBASE (Use address space layout randomization) | Microsoft Docs
- Whitepaper on Bypassing ASLR/DEP
- ASLR/DEP绕过技术概览 – ArkTeam
- Why it's important to turn on DEP and ASLR Windows security features
- 安全保护技术ASLR绕过启示录 - FreeBuf互联网安全新媒体平台
- 栈溢出基本ROP绕过ASLR和NX保护_网络_ditto的博客-CSDN博客
- “优雅”的Linux漏洞：用罕见方式绕过ASLR和DEP保护机制 - 云+社区 - 腾讯云
- Windows溢出保护原理与绕过方法概览 | riusksk's blog
- Abysssec Information Security and Vulnerability Research Group
- IE漏洞学习笔记（一）Heap Spray - 安全客，安全资讯平台
- PWN入门系列（四）：栈终结篇 - 安全客，安全资讯平台
- pwn入门之栈溢出练习 - FreeBuf专栏·i春秋学院
- Linux pwn入门教程(1)——栈溢出基础 - 知乎
- ctf中pwn入门指南 | ditto's blog
- 零基础，如何进行漏洞挖掘 - 知乎
- 从ctf入门漏洞挖掘_网络_tangsilian的博客-CSDN博客
- 网站漏洞挖掘 - 云+社区 - 腾讯云
- [思路/技术] 如何入门漏洞挖掘，以及提高自己的挖掘能力。（干货）-CanMeng'Blog - 一个WEB安全渗透的技术爱好者
- 谈高效漏洞挖掘之Fuzzing的艺术 - FreeBuf互联网安全新媒体平台
- [求助]怎样学习漏洞挖掘?-『经典问答』-看雪安全论坛
- [原创]各类漏洞挖掘方法辨析-『二进制漏洞』-看雪安全论坛
- [原创]游戏漏洞挖掘概述-『二进制漏洞』-看雪安全论坛
- mhs6.2汉化版下载|MHS(内存修改工具)下载v6.2 汉化 版_ IT猫扑网
- [转载] 漏洞挖掘小白入坑指南 - 个人文章 - SegmentFault 思否
- 漏洞挖掘 | 安全脉搏
- 【技术分享】漏洞挖掘高级方法 - 安全客，安全资讯平台
- 内存搜索、修改器（附VC6源码）*BeanJoy*的专栏-*CSDN*博客内存搜索修改器
- radare2 Alternatives and Similar Software - AlternativeTo.net
- Radare2 学习笔记：从入门到精通 1. Radare2 简介，及安装_Tangent's blog-CSDN博客_radare2 安装
- 老司机带你玩转Radare2 - 简书
- The Ultimate Disassembly Framework – Capstone – The Ultimate Disassembler
- Capstone & LLVM – Capstone – The Ultimate Disassembler
- 各种开源汇编、反汇编引擎的非专业比较 - simpower的个人空间 - OSCHINA
- Capstone引擎正式支持RISC-V架构 - 51CTO.COM
- Capstone反汇编引擎数据类型及API分析及示例(一) | K's House/#Capstone%E5%8F%8D%E6%B1%87%E7%BC%96%E5%BC%95%E6%93%8E%E6%95%B0%E6%8D%AE%E7%B1%BB%E5%9E%8B%E5%8F%8AAPI%E5%88%86%E6%9E%90%E5%8F%8A%E7%A4%BA%E4%BE%8B-%E4%B8%80)
- 反汇编工具capstone的使用_归来仍少年的博客-CSDN博客_capstone反汇编
- 全能型反汇编引擎 – Capstone-Engine - 云+社区 - 腾讯云
- Android平台下使用Capstone反汇编引擎(HEX转ARM64) - 『移动安全区』 - 吾爱破解 - LCG - LSG |安卓破解|病毒分析|www.52pojie.cn
- Linux Udis86 反汇编引擎使用 - 小黑电脑
- Udis86 Disassembler Library for x86 / x86-64
- Linux Udis86 反汇编引擎使用-云社区-华为云
- Assembling and disassembling
- ELF文件格式 · Android逆向：静态分析
- Mach-O
- 逆向利器：IDA
-