正向书籍：https://zhuanlan.zhihu.com/p/109413416?from\_voters\_page=true

反汇编使用工具：OD（动态 用于游戏辅助）IDA（静态 用于取消软件授权认证检测）。

逆向过程：机器语言-汇编语言（-高级语言） 汇编语言自动转化成高级语言的准确率十分低，因此逆向工作基本都是在汇编语言的层面进行。

汇编语言的组成：汇编指令 伪指令 其他符号（后两者不会产生机器码）。

CPU与外部器件的信息交互：地址信息 控制信息(读or写) 数据信息，三种信息的交互对应于逻辑上划分的三种总线。

32位CPU地址总线的宽度为32，最多管理2^32个内存单元，一个内存单元的大小为1Btye（8bit），即最大有效内存为4GB。

实现64位的效率必须要CPU 系统 软件这三者都为64位。

在C语言中32位程序指针大小为4字节，64位程序指针大小为8字节。

RAM随机储存器（主储存器）；ROM 只读储存器 装有BIOS。

8086CPU地址总线宽度为20，所有的寄存器都是16位的，可以存放两个字节（即一个字；1word = 2Byte，可以分为高八位和低八位）。其中AX，BX，CX，DX用来存放一般性数据，被称为通用寄存器。（AX又可分为AH和AL，目的是向下兼容8位寄存器）。

汇编指令不区分大小写。

AX = 00C5H，则在执行 ADD AL,93H 后，AX = 0058H。低八位的溢出不会进位到高八位的寄存器中。

8086CPU有20位地址总线，但它内部为16位结构，只能传送16位的地址，表现出的寻址能力只有64KB，因此采用一种在内部用两个16位地址合成的方法来表示一个20位的物理地址。

地址加法器合成物理地址的方法：物理地址 = 段地址 \* 16（即基础地址）+ 偏移地址。（段地址 \* 16即左移4位，变成一个20位的地址，再加上一个16位的偏移地址，最后得到一个20位的物理地址）。

8086CPU一个段的起始物理地址（偏移量为0的时候）一定是16的倍数，一个段的最大长度为偏移地址的最大容量，即64KB。（储存空间为2^20Byte，即为1MB）。

内存单元21F60H可以表示为2000:1F60。即段地址:偏移地址。

8086CPU有4个段寄存器：CS，DS，SS，ES。（code segment, data segment, stack segment, extract segment）。

CS和IP是8086CPU中最重要的两个寄存器，它们（即CS:IP）指示了CPU当前要读取指令的地址。（IP为指令指针寄存器，instruction pointer）。

在8086CPU启动或复位时，CS = FFFFH，IP = 0000H，FFFF0H单元中的指令是8086PC机开机后执行的第一条指令，所以系统一般储存在从FFFF0H起始的这个内存段中。使木马程序不被检测到的一种手段是改变CS和IP的初始值，使CPU刚运行的时候就直接定位到木马程序所在的内存段。破解软件同理，将CS:IP的初始位置直接指向软件的实际有效代码，就可以跳过原本进入软件一开始的权限认证部分。

改变CS和IP的值：JMP CS:IP。

只改变IP的值：JMP AX（可以理解为MOV IP,AX）。

执行 JMP AX这条语句后总共修改了两次IP的值：读取完这条指令会修改一次，把IP值定位向这条指令末尾，然后执行这条指令的时候再修改一次，把IP值改为寄存器AX中的值。

内存中字的储存分为小端和大端。Little-Endian：低位字节放在内存的低地址端；Big-Endian：低位字节放在内存的高地址端。

int a = 200000（30D40H）; int\* p = &a; short\* sp = (short\*) p; 则\*p = 3392（0D40H）(在小端系统的前提下)。

DS寄存器(数据地址段寄存器)：储存要访问的数据的段地址。比如若访问10000H（1000：0）单元里的内容，可以通过如下指令：MOV BX, 1000H MOV DS, BX MOV AL, [0]。（由于硬件设置的原因，不能直接将值赋给DS寄存器，必须要通过一个通用寄存器来中转）。

8086CPU的入栈和出栈操作都是以2字节为单位进行的。

任意时刻SS:SP（stack segment:stack pointer）指向栈顶元素。

PUSH AX的执行过程：（1）SP = SP – 2；（2）将AX寄存器中的内容送入SS:SP指向的内存单元处，SS:SP此时指向新栈顶。

POP AX的执行过程：（1）将SS:SP指向的内存单元处的内容送入AX寄存器中，（2）SP = SP + 2。SS:SP此时指向新栈顶。

栈空时SS:SP指向栈空间最高地址单元的下一个单元（即在栈空间外）。也可以表示为栈底字单元的偏移地址+2。

POP指令并不会清除掉原来的栈顶数据，只是改变了SS:SP指向的位置（浅层次删除之后的文件还可以再恢复回来即是这个原理）；再次使用PUSH指令后会覆盖掉原来的栈顶数据。

8086CPU中不能自动限定栈顶和栈底的范围。

PUSH POP还可用于操作段寄存器（便于保护函数现场）。

代码段，数据段，栈段关键在于CS，IP，DS，SS，SP这5个寄存器的设置。

汇编程序中返回的程序段是固定的两条语句：MOV AX 4C00H int 21H。

CX寄存器通常用来存放循环次数。

CPU执行loop s时进行两步操作：（1）（CX）=（CX）- 1；（2）判断（CX）是否为0，不为0则跳转至s处继续执行，如果为0则执行下一条语句。

段前缀：显式指定地址段的位置而不是采用默认位置段。比如MOV [0],0显式指定出来后就为MOV DS:[0],0，也可以把DS改为CS，SS等。

and\or 按位与或运算。

SI和DI是与BX功能相近的寄存器，但是不能分为两个8位寄存器来使用。

MOV AX,[BX+idata]即为(AX) = ((DS) \* 16 + (BX) + idata)。也可以把idata替换为SI或DI，即完成两个寄存器内容的相加。（idata是立即数，可以理解为常数，汇编指令中的立即数都会在对应的机器指令中出现）。

无条件转移的段内短转移：jmp short 标识符号。此种转移最多向前128字节，或向后127字节，即在-128~127的范围内。（段内短转移效率比较高，在对应的机器码中不会直接给出转移的目标地址，而是会给出读完这条段内短转移指令之后的CS:IP需要改变的偏移量，该偏移量只有2位十六进制数即1Byte的容量，因此偏移量覆盖的范围只有256字节。该偏移量的计算需要编译器来完成，这是牺牲编译环节而提高最后程序运行效率的一种手段）。

无条件转移的段内近转移：jmp near ptr 标识符号。偏移量有4位十六进制数，覆盖范围为64KB，即在-32768~32767的范围内。

无条件转移的段间转移（远转移）：jmp far ptr 标识符号。

MOV AX,0000H JMP AX执行完后即为IP = 0000H。

段间转移的一种方式：mov ax,0123H mov [bx],ax mov word ptr [bx+2],0 jmp dword ptr [bx] 执行后(CS) = 0，(IP) = 0123H。其中， word ptr和dword ptr表示操纵内存单元的范围，dword ptr [bx]表示从(BX)所在的内存单元起从低位到高位选取double word即4字节的数据，这个四字节数据中的低16位表示IP，高16位表示CS。

有条件的段内短转移：jcxz 标识符号 若(CX) = 0则转移到标识符号处执行。范围为-128~127Byte。

jcxz 标号 语句相当于if((cx) == 0) jmp short 标号。（jump if cx zero）。

loop 标号 语句也是短转移，相当于(cx)-- if((cx) != 0) jmp short 标号。对IP的修改范围为-128~127。

inc 自加；dec 自减。

执行ret（近转移）：(1) (IP) = ((SS) \* 16 + (SP))；(2) (SP) = (SP) + 2。相当于执行POP IP。

执行retf（远转移）：（1）(IP) = ((SS) \* 16 + (SP))；(2) (SP) = (SP) + 2；(3) (CS) = ((SS) \* 16 + (SP))；(4) (SP) = (SP) + 2。相当于执行POP IP POP CS。

call 标号 将当前的IP压栈后转移到标号处执行。相当于执行push IP jmp near ptr 标号。（注意此时的IP指的是读完call这句语句之后的IP）。

call far ptr 标号 执行段间转移。相当于执行 push CS push IP jmp far ptr 标号。

call 16位寄存器。call word ptr 内存单元地址。call dword ptr 内存单元地址。

mul byte ptr 内存单元地址：自动与AL寄存器内的数据相乘，结果储存在AX寄存器中；mul word ptr 内存单元地址：自动与AX寄存器内的数据相乘，结果的低位储存在AX中，高位储存在DX中。

对标志寄存器产生影响的指令：add sub mul div inc dec and or cmp等涉及到运算的指令；不产生影响的指令：mov push pop等传送指令。

重要的标志位：ZF：运算结果为0则为1；PF：运算结果有偶数个1则为1；SF：运算结果为负则为1；CF：进位和借位的标志位（只是针对无符号数运算而言）；OF：溢出标志位（只是针对有符号数运算而言），如果发生溢出，OF为1。

cmp 操作数1 操作数2。执行操作数1-操作数2，但是不对结果进行保存，只影响标志寄存器的各个标志位。

条件跳转指令：je(jump equal) jne ja(jump above) jna jb(jump below) jnb。

MOVSX：带符号扩展传送；MOVZX：0扩展传送。

lea 操作数A，操作数B 其中A必须为一个寄存器。

je指令和jz指令等价，都是在ZF为1时进行跳转。

32位反汇编中的有符号跳转指令，小于用l（less）表示，大于用g（greater）表示。如JLE和JNG都表示在小于等于时进行跳转；JL与JNGE等效。

无符号跳转指令，小于用b表示，大于用a表示。

EBP可以理解为栈底指针，加上一个偏移量之后可以在一个CALL里面访问临时变量。一般与ESP配合使用。

一个空函数转化为汇编语句之后有四句话：push ebp mov ebp,esp pop ebp ret。（在执行CALL语句之前用push语句给该函数传递参数，此时esp指向栈顶，将esp的值传递给ebp即是把传递给函数的参数的栈顶与函数内部生成的临时变量的栈底联系起来，形成一段连续的内存空间。如果该函数不为空函数，则在mov ebp,esp之后还有一句sub esp,某立即值。【该立即值通常为4的倍数，便于计算机的计算】从现在的esp到ebp这一段内存空间即为给该函数内需要使用的变量开辟的内存空间。在pop ebp之前还有一句话：mov esp,ebp将esp返回到该函数的栈底。在执行ret语句返回上一级之后，再执行add esp,某立即值。【该立即值为给函数传参时压栈的空间大小，即舍弃原来传递进去的参数值，等待下一次压栈操作时就将上一次的传参值覆盖掉】）。

CALL栈平衡：进CALL前与出CALL后，ESP和EBP的值不能变。

[ebp-某立即数]可以理解为函数中的临时变量；[ebp+某立即数]可以理解为传入该函数的参数。

函数调用约定：①cdecl：即C Declaration，vs里默认的调用方式，传参时参数从右向左依次入栈。函数执行完ret语句之后需要用 sub esp,某立即值 来手动清栈。②stdcall：C++的标准调用方式。函数返回时执行 ret 某立即值，这样在返回上一级函数的时候就自动清栈。③fastcall：优化传参速度，将前若干个参数用寄存器来传递，这样比用堆栈传递参数更快。（注意如果是以fastcall调用约定传参，则有可能在CALL语句前看不到标志性的PUSH指令）。

不定参数的传参方式只能用cdecl调用约定传参，比如调用printf函数的CALL后面一定会有一句手动清栈的语句。

FPU：float point unit，用于处理浮点数。MMX：multi media extensions，多媒体扩展，用于处理音视频信号及图像信息等。在物理上这两种都是同一组寄存器。

在CE中使用aobscan(标识符, \*\* \*\* \*\*)对某行代码进行注入的时候可以直接定位到需要修改的那一行代码，但是在[ENABLE]的最后要加上registersymbol 标识符，在[DISABLE]的最后加上unregistersymbol 标识符。如果没有注册该标识符，那么在激活脚本的时候可以正常定位到需要修改的那一行代码，但是在取消激活的时候就会出错，因为该行代码变成了jmp 某内存空间，特征码已经发生了改变，所以无法再搜索到该行代码，也就无法再还原。

define某基址为一个标识符号，不再使用该标识符之后需要unregistersymbol，不然不会注销标识符。

globalalloc(标识符, 4)相当于registersymbol(标识符) alloc(标识符, 4)。最后同样需要使用unregistersymbol。

使用CE的aobscan注入模板的时候需要严格注意INJECT标识符的编号。如果套用了多处aobscan模板而没有对INJECT标识符进行标号的话，那么在取消激活某条脚本时如果触发了unregistersymbol(INJECT)语句就会导致其他含有INJECT的语句无法取消激活。

使用CE找特征码时，如遇见搜不到数据的情况可以把Writable和CopyOnWrite这两个选项前打上灰块。（黑魂1受死版需要Writable，怪物猎人世界需要CopyOnWrite）

创建线程：alloc(newmem，256) createthread(newmem) newmem:\*\*\*\*\*\*\*\* ret.

函数返回值：8位放在al，16位放在ax，32位放在eax，64位放在rax。CALL结束之后，mov [某内存单元], rax xor rax, rax。

switch…case语句：如果分支情况较少，生成的汇编代码类似if…else if…else的结构；如果分支情况较多且连续，生成大表跳转；如果分支情况较多，且中间不连续，则在生成大表的基础上先查小表，便于精简掉大表中本来用default填充的情况。

函数指针的用法：将代码的硬编码存放在一个数组里面，即把代码从代码段转移到数据段，然后定义一个函数指针来访问这个数组即可执行相应函数。比如定义了一个int func(int a, int b)的函数，现在把func的硬编码存放在unsigned char arr[ ] { }数组里面，然后定义函数指针int (\*fp) (int, int)，把函数指针指向数组fp = (int (\*) (int, int)) & arr，就可以执行函数fp(a, b)。这样做的作用是可以把代码段的部分内容清零，即实现软件的加壳。(注意在类型强制转换的时候这里的arr类型是unsigned char [n]，而不是unsigned char\*，因此不用取址符号无法进行强制转换，只有用&符号，&arr的类型为unsigned char (\*) [n]，变成一个指针类型才能强制转化为另一个指针类型。这里也体现了数组的标识符不能完全等效于指针，指针的操作更加自由)。

DOS头标志：4D5A（MZ） PE头标志：5045（PE）。

给数组取址（&）相当于升高维数；取内容（\*）相当于降低维数，降低维数后如果还是一个数组，那么取到的内容就是降维数组的地址，如果不是数组，那么取到的内容就是数组成员的值。如三维数组int arr[3][3][3]，\*arr相当于二维数组，也即是一维数组取址，因此\*arr的类型为int (\*) [3]。

CE中查找是哪句话跳转到某个地址：search-find assembly code: j\* 地址。其中j\* 同时涵盖了je, jne等多种跳转情况。

CE使用自动汇编注入之后游戏崩溃很大可能是有其他语句跳转到被注入修改的代码处，这种情况下可以用查找汇编代码的功能验证一下。也可能是诸如jmp dword ptr [eax\*4+0041A810]等用switch-case跳转表来表示跳转的情况，这时直接在内存搜索疑似跳转点的地址即可找到跳转表的位置（注意把搜索条件更改为方块）。

调用约定：函数调用者（caller）负责清理储存在栈中的参数称为cdecl方式；被调用者（callee）负责清理栈中参数称为stdcall方式。fastcall：与stdcall调用方式类似，不同的是使用ECX、EDX两个寄存器传递前两个参数，传参前和传参后可能需要备份这两个寄存器的值。