- 栈溢出
  - 基础知识
    - 栈结构
      - 了解rsp、canary、rbp、返回地址在栈中的位置
    - 函数传参方式
      - x86函数参数在函数返回地址上方
      - x64前六个整型或指针参数依次存放在rdi, rsi, rdx, rcx, r8, r9寄存器，其余参数同x86
    - 栈溢出原理
      - 程序向栈中变量写入的字节超过了变量本身申请的空间大小，导致其他变量的值被更改
    - 栈溢出后果
      - 程序向栈上写入数据
      - 写入数据大小没有被很好控制
  - 常规利用
    - 覆盖
      - 覆盖返回地址
        - 程序存在目标函数
          - 覆盖返回地址为目标函数
          - 程序不存在目标函数
            - 覆盖GOT表
            - 中断调用
        - 确定目标函数
          - 根据IDA、RSP、RBP等溢出变量的相对距离计算
        - 确定填充长度
          - 覆盖返回地址
            - 使用peda工具
              - 'pattern create xxx' 生成xxx个字节无序字符串，手动写入溢出位置
              - 查看程序崩溃时RBP中存储的值ppp，利用'pattern offset ppp' 确定溢出变量到rbp(ebp)相对距离
          - 其余
            - 特点
              - 需要控制程序本身已有代码
            - 利用方式
              - 覆盖返回地址为目标代码地址
        - ret2text
        - 特点
          - 存在可写入、可执行的存空间(gdb下，使用vmmap查看)
        - 利用方式
          - 向可利用内存空间写入shellcode
          - 覆盖返回地址为shellcode地址
      - ret2shellcod
      - 特点
        - 无目标函数
        - 系统调用
          - 将调用目标函数需要的参数入对应寄存器
          - 执行int 0x80
        - 预备知识
          - 例:目标函数execve("/bin/sh", NULL, NULL)
            - EAX = 0xb(系统调用号)
            - EBX = addr("/bin/sh")(参数一)
            - ECX = 0(参数二)
            - EDX = 0(参数三)
        - ret2syscall
        - 利用方式
          - 将gadget排列在payload中，注意栈平衡(重点，难点)
          - 寻找控制对应寄存器的gadget
            - ROP工具
              - 使用ROPgadget查找程序中所需的gadget
            - 突破方式:使用gdb反汇编查看正常程序函数调用过程中的汇编代码，掌握函数返回，参数调用顺序
            - 参考wp中的payload，在脑海中模拟，一步步执行，强化理解
      - 特点
        - 控制程序执行libc中的函数
        - 是否存在"/bin/sh"字符串
          - 是
            - 将地址写入函数参数处
          - 否
            - 将字符串写入已知内存处，将地址写入函数参数处
          - 是
            - system_plt地址(使用pwntools, plt函数)覆盖返回地址
            - 预备知识
              - 1. libc.so动态连接程序函数之间相对偏移固定
              - 2. ASLR只打乱基址，即地址最低12位不会变化，那么如果知道某函数实际地址，就可通过对最低12位地址的匹配得到程序动态解数本
              - 工具:使用LibcSearcher工具可做化动态适应过程，同时可通过手动计算出的基址及当前程序目标函数地址
                - 通过查出函数在got表中某函数的真实地址
                - 向LibcSearch函数，输入某函数名，某函数真实地址，返回libc动态适配版本
                - 基地址 = 某函数真实地址-libc.dump("某函数名")
                - 目标函数地址 = 基地址 + libc.dump("目标函数名")
            - 利用方式
              - 提示:利用工具前，应理解工具的工作原理，不能盲目使用工具
    - ret2libc
    - 利用方式(例:调用system函数)
      - pltl表是否存在system函数
        - 否(分析开启aslr情况下)
          - 特点
            - x64下的__libc_csu_init函数中有可利用的gadgets
            - gdb反汇编__libc_csu_init函数，掌握expFrcsu()函数
          - 预备知识
            - 0x40061E~0x400622 = 0x400600~0x400606(参数最低)
            - 0x400609: call qword ptr [r12 + rbx*8]　r12寄存器存储目标函数地址，rbx = 0时，可以调用目标函数
            - 最重要步骤:
              - 1.手动在栈上布置目标函数参数
              - 2.通过pop指令，弹出参数至寄存器r13, r14, r15
              - 3.通过mov指令，将r13, r14, r15的值赋给rsi, edi, rdx
            - 0x400611~0x400614　为了不执行0x400614处的跳转指令，rbx, rbp应相等，即需满足rbp = 1
            - 第二次调用0x400616~0x400622　应栈区填充0x38字节垃圾数据，平衡栈
            - 0x40060d: call qword ptr [rbx + 0x38]
          - 利用方式
            - 以ret_will(出现bug)函数为模板，更改对应部参数，目标函数地址，返回地址
          - 崩溃后会重启，重启后内存信息不变
        - ret2csu
        - 是
          - 将地址写入函数参数处
          - 将字符串写入已知内存处，将地址写入函数参数处
          - system_plt地址(使用pwntools, plt函数)覆盖返回地址
          - 预备知识
            - 1. libc.so动态连接程序函数之间相对偏移固定
            - 2. ASLR只打乱基址，即地址最低12位不会变化，那么如果知道某函数实际地址，就可通过对最低12位地址的匹配得到程序动态解数本
            - 工具:使用LibcSearcher工具可做化动态适应过程，同时可通过手动计算出的基址及当前程序目标函数地址
              - 通过查出函数在got表中某函数的真实地址
              - 向LibcSearch函数，输入某函数名，某函数真实地址，返回libc动态适配版本
              - 基地址 = 某函数真实地址-libc.dump("某函数名")
              - 目标函数地址 = 基地址 + libc.dump("目标函数名")
            - 提示:利用工具前，应理解工具的工作原理，不能盲目使用工具
    - BROP
    - 利用方式
      - 控制输出函数
        - 例:寻找puts函数
          - 1. 参数
            - 使用__libc_csu_init中的gadget控制rdi为0x400000
          - 2. 探测
            - 预备知识
              - elf文件头(0x400000)处存储"\x7fELF"
            - 方式
              - 1. 依次以0x400000后的地址作为探测地址，覆盖返回地址
              - 2. 若程序输出"\x7fELF"则证明此探测地址为puts函数的plt表
        - 寻找__libc_csu_init中的gadgets
          - 1. 寻找stop gadget
            - 预备知识
              - probe
                - 待探测地址
              - stop gadget
                - 不会使程序崩溃的地址
              - trap gadget
                - 导致程序崩溃的地址(0x44)(0)就可以作为trap gadget)
            - 从0x400000开始，不断递增地址，并用移动后的地址覆盖返回地址，若程序不崩溃，此地址即为stop gadget
          - 2. 布置栈
            - 1. 使用probe,stop,stop,stop

- frame faking
  - 预备知识
    - 程序出口
      - leave
        - 1. mov esp, ebp
        - 2. pop ebp
      - ret
        - #pop eip
  - 模拟执行
    - 模拟栈
      - 上一函数运行结束，准备执行leave，ret
        - 0xffffffff: fake ebp(A_addr) | return addr(leave_ret_addr)
        - A_addr: B_addr | taget_fun_addr
    - 运行
      - 上一函数执行leave，ret后
        - esp指向0xffffffff
        - ebp指向A_addr
        - eip将要执行leave_ret_addr
      - 执行完leave_ret_addr后
        - esp指向A_addr
        - ebp格式化字符串通过索引递推的方式得到参数指向B_addr
        - eip将要执行target_fun_addr
  - 利用方式
    - 同stack pivoting
    - 例:2018 安恒 over
      - 1. write不给输入值补"\0"，可用puts泄漏rbp
      - 2. 分析函数可知泄漏值为main函数栈帧，需计算溢出变量地址
        - main函数调用sub_400676函数时，需call xxx，push rbp，故栈抬高0x20
        - sub_400676函数内部，需要sub rsp 0x50为变量开辟空间，故栈抬高0x50
        - 综上栈溢出变量地址为:stack(泄漏值) - 0x70
      - 3. 第一次payload运行可参考上面的模拟运行，最终泄漏puts真实地址
      - 4. 计算.so动态库加载基址
      - 5. 通过ROPgadget 分析.so得到的地址 + 基址 = 运行时的真实地址
      - 6. 运行第二次payload，执行execve，重点分析为什么stack - 0x30　　通过gdb调试发现，第一次payload运行后rsp改变，故对于栈上的数据构造，应该与调试紧密结合
- stack smash
  - 预备知识
    - 栈溢出修改canary后，程序执行__stack_chk_fail函数
    - __stack_chk_fail函数调用__fortify_fail，输出argv[0]
  - 利用方式
    - 使用目的地址覆盖argv[0]
    - 程序报错后，输出目标地址内容
- partial overwrite
  - 预备知识　　ASLR + PIE开启后低12位页内偏移固定(涉及内存分页机制)
  - 利用方式　　修改低12位地址，绕过PIE
- SROP
  - 预备知识
    - 软中断流程
      - 1. 内核向进程发送signal，进程挂起，进入内核态
      - 2. 内核为进程保存上下文(将寄存器，signal信息，sigreturn地址入栈)—sigreturn地址位于栈顶
      - 3. 程序执行signal handler，return后，执行sigreturn代码(sigreturn用于恢复之前保存的上下文)
    - 利用点
      - 1. 在栈中伪造signal Frame
      - 2. 程序主动调用sigreturn，此时寄存器的值可被控制
    - 利用方法
      - 1. 在栈中伪造signal Frame，重点构造rip(控制执行流程)，rsp(劫持栈)，参数寄存器
      - 2. 触发sigreturn
        - 1. 覆盖返回地址为sigreturn
        - 2. 使用系统调用(syscall)，如果希望构成攻击链需要syscall，ret
        - 难点，寻找gadget
  - 例: smallest-pwn
    - 1. 分析程序
      - 1. 通过系统调用实现read(0，$rsp，400)，向栈写入400 字节
      - 2. 栈顶处为程序返回地址，通过控制栈顶数据，控制程序流程
      - 3. 源程序无返回地址，执行read后，自动崩溃
    - 2. 分析exp
      - 1. 向栈写入三遍程序开始地址(目的:使下一步操作时，程序跳转至开头，不会崩溃)
      - 2. \xb3覆盖栈上第一个字节，修改第二遍开始程序地址为0x4000b3
      - 3. 因为read读入一个字符\xb3，故rax = 1(涉及汇编返回值)
      - 4. 程序再次执行时:跳转至0x4000b3(避过xor rax，rax)，保持rax = 1
      - 5. 此时程序执行write(1，$rsp，400)，读取栈上内容并输出(重点输出第三遍开始地址后的内容，此处为stack_addr)—难点
      - 6. 构造signal Frame(控制寄存器执行read系统调用读取外界数据，特别劫持rsp到stack_addr，将数据写入与stack_addr偏移固定处)
      - 7. 通过read读取15个字符，控制rax = 15，执行sys_call(调用sigreturn)，实现5
      - 8. 向被劫持后的rsp处写入signal Frame(构造system系统调用，"bin/sh"地址通过rsp + 相对距离获得)，"bin/sh"字符串
      - 9. 再次触发sigreturn，获得shell

- 格式化字符串漏洞
  - 预备知识(x86)
    - 1. 参数在栈中的位置
      - 调用函数时参数自右向左依次入栈
      - 格式化字符串调用参数时
        - ebp - 0x8处为格式化字符串地址(参数一)
        - ebp - 0xc，ebp -0x10，ebp -0x14.....分别为参数二，参数三，参数四.....
          - 格式化字符串函数通过在栈上递推的方式调用多个参数，也就意味着，格式化字符串函数调用完传入参数后，可以继续输出栈上的未知数据(重点)
    - 2. 格式化字符串函数运行规则
      - 1. 函数获得第一个参数格式化(格式化字符串)
      - 2. 分析第一个参数
        - 当前字符非%，直接输出
        - 当前字符是%，继续读取下一字符
          - 没有字符，报错
          - 当前的字符是%，输出%
          - 否则，根据对应字符意义，获取对应参数，输出参数
    - 建议:
      - 自行编写有关格式化字符串的c语言程序，以各种姿势输出多个参数，熟悉参数输出
      - 使用gdb反汇编程序，重点观察栈区参数与格式化字符串函数输出关系
  - 利用原理
    - 程序崩溃
      - 格式化字符串:"%s%s%s....."
      - %s需要拥有合法地址的参数进行解析，栈上存在不合法地址，遭遇后，程序崩溃
    - 泄漏内存
      - 栈上
        - 获取栈内存信息　　格式化字符串:"%x"(以16进制无符号整型输出栈上信息)或"%p"(以地址的形式输出栈上信息)
        - 获取栈上地址指向处内容　　格式化字符串:"%s"(将栈上信息作为地址变量，输出地址指向处信息)
        - 获取指定栈上指定参数
          - 预备知识　　%n$可定位至第n+1个参数
          - 格式化字符串:"%n$x"(以十六进制无符号整型输出栈上第n+1个参数信息)
      - 任意地址
        - 预备知识
          - 1. 格式化字符串函数调用参数一(格式化字符串)时，实际上是引用了格式化字符串的地址
          - 2. 在存在格式化字符串漏洞printf(s)[s为变量]的程序中，s作为局部变量，会存储在栈上
          - 3. 变量s在调用格式化字符串函数之前完成声明过程，所以s在栈上存储的位置应高于格式化字符串函数参数
          - 4. 理论上讲，变量s可以作为格式化字符串的参数
          - 栈排布　　s_addr，参数二，参数三，参数四.....s(参数k + 1)，参数k + 2，.....
        - 利用方式
          - 1. 变量 s = "target_addr%k$s"
          - 2. 此时栈排布　　s_addr，参数二，参数三，参数四.....s(参数 k + 1)，参数k + 2
          - 3. 探测变量s位于格式化字符串函数第几个参数
            - 1. printf("AAAA %p%p%p%p%p.....")
            - 2. 输出内容"AAAA xxxxxxxxxxxxx 0x41 0x41 0x41 0x41....."
            - 3. 根据AAAA与0x41的相对距离确定变量s位置(假设变量s为函数第k + 1参数)
          - 4. 结果　　%k$s解析target_addr，输出目标地址处内容
    - 写
      - 预备知识　　格式化字符串"%n"，将前面输出字符的个数写入对应整型指针参数处
      - 栈上
        - 1. 原理同泄漏栈上变量
        - 2. 区别: 使用"%n"，将已输出字符个数写入目标地址
      - 任意地址
        - 小数字
          - 预备知识　　若向目地址覆盖小数值(小于字长)，那么将addr放在前面是不合适的(x86下addr为4字符，若将addr放在字符串前面，则会输出至少四个字符，即无法将目的地址覆盖为小于4的数值)
          - 利用方法
            - 1. 将变量s中的target_addr放在s字符串末尾，即保证覆盖target_addr处内容时，并未输出target_addr
            - 2. 将变量s中的target_addr移动至末尾后要注意
              - 1. 偏移位置应作适当改变
        - 大数字
          - 预备知识　　%hn可一次写入两字节
          - 利用方法
            - 1. 将变量s中的target_addr放在s字符串末尾，即保证覆盖target_addr处内容时，并未输出target_addr
            - 2. 将变量s中的target_addr移动至末尾后要注意
              - 1. 偏移位置应作适当改变
  - 利用方式
    - 1. 泄漏内存
      - 1. 泄漏栈内存
        - 1. 泄漏栈上变量
        - 2. 泄漏栈上指向处内容
        - 3. 泄漏栈上指定参数
      - 2. 泄漏任意地址
        - 1. 泄漏任意地址内容
        - 2. 泄漏任意地址指向处内容
    - 2. 写
      - 1. 写栈上变量
      - 2. 写栈上指向处内容
      - 3. 写任意地址
        - 1. 写任意地址内容
        - 2. 写任意地址指向处内容

- 二进制漏洞利用
  - 利用方式
    - 覆盖内存
      - 任意地址
        - 原理同泄漏任意地址，对于覆盖的数值大小，存在相应技巧
          - 大数字
            - 输出大量字符　速度慢
            - 按字节覆盖
              - 预备知识　数值在内存中按小端存储(低地址存储低位数据)
              - 利用方式
                - 对目标地址各个字节单独覆盖
                - 如何覆盖单个字节?
                - hh:写入一个字节
                - h:写入两个字节
          - 3. 解析目标地址前输出目标数值个字符
          - 2. 注意字节对齐
        - 总结:确定目标地址与第一个参数的偏移，根据具体情况使用%n向目标地址写入新数据
    - x64格式化字符串漏洞
      - 与x86区别: x64前六个参数存入寄存器，后面的参数以此入栈
      - 例: pwn200 GoodLuck
        - 1. b printf使程序在调用printf时中断
        - 2. 查看栈区，栈顶为返回地址，栈上偏移5处是flag位置，即栈上有四个printf函数参数
        - 3. 考虑六个存入寄存器的参数，故flag位于格式化字符串的第十个参数，使用%9$定位
    - hijack GOT
      - 预备知识
        - PLT表(text)　程序调用动态库中函数时使用慢启动的方式，第一次执行某函数plt时，会先将函数真实地址放入对应got表，然后程序跳转至got表，以后再执行该函数时，直接跳转至got表
        - GOT表(data)　got表中存储函数地址
      - 利用方法
        - 1. 利用格式化字符串漏洞泄漏出函数真实地址 (利用格式化字符串漏洞输出出某函数_got表内容)　利用技巧
          - 1. 泄漏任意地址数据
          - 2. 函数got表地址可用pwntools中got函数获得
        - 2. 根据某函数真实地址，确定libc版本(使用LibcSearcher工具)
        - 3. 根据某函数真实地址，计算动态库加载基址:base = 真实地址 – libc.dump(xxx)
        - 4. 根据动态库加载基址计算目标函数真实地址:target_fun_addr = base + libc.dump("target_fun")
        - 5. 利用格式化字符串漏洞修改某got表内容为目标函数地址　利用技巧
          - 1. 修改任意地址数据为较大数据
          - 2. 可以利用pwntools，fmtstr_payload(自动化生成将A地址处数据覆盖为B地址的格式化字符串漏洞payload)简化payload设计(建议新手手写payload加深理解)
    - hijack retaddr　利用方式
      - 1. 寻找返回地址与格式化字符串的偏移
      - 2. 覆盖返回地址为目标函数地址(修改栈上数据)
    - blind　利用方法
      - 泄漏binary
      - 劫持got表
- ptmalloc
  - 堆管理器作用
    - 第一次接收用户申请时，向操作系统申请内存，系统返回一块很大的内存，只有此内存不足够用来满足用户以后的申请后，才会继续向操作系统申请
    - 管理用户释放的内存片段
  - 堆操作
    - malloc(n)
      - 基础
        - n = 0，返回系统默认的最小堆块
        - n为负数(在内存中以补码存在)，但malloc参数是无符号数，故产生混淆，分配很大的内存空间(如果分配成功)
      - malloc背后
        - 1. sbrk(glibc提供)，brk(操作系统提供)
          - 用于向操作系统申请内存
          - 堆的首尾
            - start _brk
              - 指向堆起始位置
              - 不开启ALSR，start_brk指向data/bss段结尾
              - 开启ALSR，start_brk指向data/bss段后面随机偏移处
            - brk
              - 指向堆结束位置
              - 未分配堆区时brk与start_brk指向同一地址
        - 2 . mmap　创建映射段
    - free(p_addr)
      - 基础
        - p_addr = NULL　函数不执行操作
        - p_addr被释放后，再次释放　构成double free
        - 释放很大的内存　程序将内存空间归还系统
      - free背后
        - 1. bin管理机制
        - 2. munmap　释放由mmap申请的映射段空间
    - arena
      - arena(为提高程序效率，操作系统一次性分配的较大连续内存区域)
        - main_arena(主线程的arena，在堆区)
        - thread_arena(子线程的arena，在映射段)
      - arena的数量与内核数量有关
    - 小总结
      - 1. 程序第一次malloc时，heap分为两块，一块给用户，剩下的那块是top chunk(即堆中物理地址最高的chunk)
      - 2. 线程分配内存时先从bins中寻找，如果没有则会分割对应的top_chunk
      - 3. top_chunk不够用时，main_arena通过sbrk拓展，thread_arena通过mmap拓展
  - 结构
    - malloc chunk
      - chunk结构体：pre_size | size | fd | bk | fd_nextsize | bk_nextsize
        - x86下，都是四字节类型
        - x64下，都是八字节类型
      - 已分配chunk
        - 1. pre_size + size为chunk header
        - 2. size之后字段全部为data区域，malloc()返回的地址即为size之后的地址
        - 前一个块状态
          - 已分配(pre_inuse = 1)　当前chunk pre_size字段可被上一个chunk data区域利用
          - 已释放(pre_inuse = 0)　当前chunk pre_size表示上一chunk大小
      - 已释放chunk
        - 1. pre_size + size为chunk header
        - 2. fd，bk为chunk在bin链表中的上一个下一个chunk(fd_nextsize和bk_nextsize同理，且用于large bin)
    - bin
      - 管理空闲chunk
      - fast bin
        - 1. 管理程序释放的较小内存
        - 2. 使用单链表管理，每个bin采取LIFO策略，即最近释放的chunk会被更早的分配
        - 3. 当用户需要的chunk小于fast bin最大大小时，会先进入fast bin寻找空闲chunk
        - 4. fast bin不会改变chunk的inuse，故不会发生与相邻chunk的合并
        - 5. malloc_consolidate用于fast bin中的chunk合并
      - bins数组
        - small bins
          - small bins由62个循环双向链表组成，每个链表中存储的chunk大小相同
          - 采用FIFO策略
          - 每个链表中存储的chunk大小相差2×SIZE_SZ(x86:8字节，x64:0x10字节)
        - large bins
          - large bins由63个bin组成，每个bin中chunk大小处于同一区间范围
          - 63个bin分成6组，组内bin chunk之间大小公差一致
          - 每个bin中的chunk都是sorted过的
        - unsorted bin
          - 空闲chunk回归所属bin之前的缓冲区，chunk大小无规定
          - 来源
            - 较大chunk被分割后，剩余部分大于MINSIZE，该chunk会被放入unsorted bin
            - 释放不属于fast bin的chunk，该chunk不与top chunk相邻，该chunk会被置入unsorted bin
- off_by_one
  - 字节溢出，覆盖其他区域
  - 例: Asis CTF 2016 b00ks
    - 难点:
      - 通过IDA分析程序(重点在于分析结构体)
      - 开启RELRO不能劫持got表
    - 利用方法
      - 1. create32字节author_name，create第一本书
      - 2. 第一本书结构体指针覆盖author_name溢出的中断字符\x00
      - 3. 输出author_name，会泄露第一本书结构体指针
      - 4. 第一本书结构体指针低字节被覆盖后会变成指向第一本书describe的指针
      - 5. 修改第一本书describe，伪造成book结构体(使结构体中book_describe指针指向第二本书结构体存储book_describe指针处)——使用这种方法可以任意读取改写内存
      - 6. create第二本书，describe 处malloc大量空间，使得程序通过mmap分配堆空间给book2_describe
      - 7. 输出第一本书内容，重点通过输出book1_name获得book2_describe指针，计算映射区域加载基址
      - 8. 通过基址计算system在.so中的真实位置

- 堆溢出
  - 漏洞利用
    - chunk_extend
      - 后向overlapping
        - 修改当前chunk，size值(使当前chunk能够与目标chunk发生重叠)，free–>malloc(size-0x10)，此时新的chunk能够控制目标chunk
      - 前向overlapping
        - 修改当前chunk，pre_inuse(标记前一个chunk为free状态)，pre_size(使当前chunk free时可通过unlink跨越多个块与目标块发生重叠)，free–>malloc(适当值)
      - 例:hitcon trainging lab13
        - 1. 执行create函数时，程序malloc(0x10)指向结构体指针，再malloc("xxx")——(这里注意malloc(0x10)，后面会用到)，指向describe
        - 2. 执行edit函数时，可修改describe内容并溢出一个字节(溢出字节覆盖结构体chunk的size)　控制结构体chunk size值为0x41，使得结构体chunk与对应describe chunk重叠
        - 3. 执行delete函数，free(结构体指针) + free(describe)
        - 4. 再次执行create时，程序malloc(0x10)指向结构体指针(使用原describe chunk)，程序malloc(0x30)指向describe(使用字节溢出后的结构体chunk)　在这里，describe chunk包含结构体chunk
        - 5. 通过修改describe数据，可以控制结构体中*describe的值　即通过修改describe，可以达到读取与修改任意地址的目的
        - 6. 利用5，泄漏got表，计算基地址，得到system地址，劫持got表中free为system
        - 7. 执行delete函数时，先执行free(content)，故提前修改某content地址为"bin/sh\x00"地址，获得shell
    - unlink
      - 预备知识
        - 触发条件
          - 当前chunk(small chunk) free时，检查其是否存在相邻已释放chunk，若存在则需先将已释放chunk从smallbin中取出，取出的机制就是unlink
        - unlink原理(x86架构下)——x64同理
          - FD = P->fd　= target_addr – 12
          - BK = P->bk　= except_value
          - FD->bk = BK　= *(target_addr – 12 + 12) = BK = except_value
          - BK->fd = FD　= *(except_value + 8) = FD = target_addr – 12
          - 结果: target_addr处被修改为except_value，except_value + 8处被修改为target_addr – 12
        - unlink安全检查　检查FD->bk是否与BK->fd指向同一区域
        - 现代unlink利用
          - 模拟待unlink chunk　pre_size | size | FD: target_addr – 12 | BK: except_value
          - 结果　target_addr处存储target_addr – 12
      - 例: HITCON stkof
        - 1. 程序没有setbuf，所以输入输出时，程序会自动申请缓存区(chunk)　先执行一些需要申请缓存区的操作atoll( )，方便后面利用
        - 2. alloc()两个chunk，第二个chunk需为small chunk(>0x70)　程序在bss处存在数组用于存储alloc()产生的heap的指针(已创建了三个heap: global[0]~global[1])
        - 3. fill()用于编辑已创建chunk，不检查输入size，故存在溢出漏洞
          - 编辑第二个chunk(global[1])，构造fake_chunk
          - 讲解此处payload
            - p64(0) + p64(0x20)　构造fake_chunk head，此fake_chunk size为0x20，且为释放状态
            - p64(head + 16 – 0x18) + p64(head + 16 – 0x10)
              - 构造fake_chunk fd与bk，此处特殊构造是为了执行unlink时，修改head + 16处指向head + 16 – 0x18
              - 即使得global[2] = &global[– 1]
            - p64(0x20) + payload.ljust(0x30, "a")
              - 1. p64(0x20)我认为是没有必要的(可以去掉这一步)，不太懂作者为什么这样写
              - 2. ljust()目的是凑够0x30字节数据，明确覆盖边界
            - p64(0x30) + p64(0x90)
              - 根据已创建chunk可知，向第二个chunk最大写入0x30个字节，多余字节会溢出，并依次覆盖chunk_3，pre_size，size
              - 这一步用意在于修改chunk_3的上一个chunk为fake_chunk，并且声明，fake_chunk已free
          - 结果　global[2] = &global[– 1]
        - 4. 编辑global[2]，使得global[0]~global[2]都指向某函数got表　结合fill()编辑功能，达成劫持got表目的
        - 5. 走流程，获得shell
    - use after free
      - 特点　free时，对应指针没有设置为NULL
      - 例: lab 10 hacknote
        - 利用点
          - 1. 执行delete后，point没有被设置为NULL
          - 2. print_note执行指针指向处函数
          - 3. delete后仍可使用函数指针
        - 利用技巧
          - fastbin采用FILO机制进行管理
          - 通过控制fastbin中bin的存取顺序，使得add_note时，content可指向前面的结构体
          - 修改前面某结构体函数指针为目标函数地址，get_shell
    - Attack
      - fastbin
        - 前提
          - 1. chunk内容可控
          - 2. 漏洞存在于fastbin
        - 利用方法
          - fastbin double free
            - 原理
              - 1. fastbin 使用单链表管理释放chunk(fd指针)
              - 2. fastbin free时，不清空pre_inuse
              - 3. fastbin free时，仅检测头部的chunk是否存在double free
            - 利用技巧
              - 先free(chunk1)然后free(chunk2)此时，chunk2位于链表头部
              - free(chunk1)，chunk1并非位于链表头部，所以不检测chunk1，double free成功
            - double free后fastbin特点
              - 1. malloc时，chunk1，chunk2会构成分配循环，被多次分配
              - 2. 分配chunk1后可以对chunk1 fd字段修改，即fastbin中的chunk1指向修改后的地址，将fastbin迁移至目标区域　注意:malloc()时，会检测目标地址size是否与fastbin链表对应size相同，应提前修改
          - house of spirit
            - 原理　在目标地址处伪造fastbin chunk，释放目标地址，分配时可获得目标地址chunk
            - 利用方法
              - 1. fake chunk ismmap = 0，free mmap chunk，机制不同
              - 2. fake chunk地址对齐，x86 = 8，x64 = 0x10
              - 3. size 符合fastbin要求
              - 4. 注意避免double free
              - 5. 下一个chunk的size > 2*SIZE_SZ&size < system_mem
          - alloc to stack　原理　劫持chunk fd指向栈，控制返回地址等关键数据，注意size的构造
          - arbitrary alloc　原理　alloc to stack 的拓展，只要存在合适的size，将chunk分配至任意可写内存中
        - 例: hack.lu oreo
          - 分析程序　add存在堆溢出
          - 分析exp
            - 1. name溢出覆盖next，为got["puts"]地址
            - 2. 输出功能，泄漏puts真实地址　获得libc基址，system真实地址
            - 3. 希望使用arbitrary alloc劫持栈到bss段
              - 1. 目标chunk应包含notice
              - 2. 结构体size = 0x40，由notice向下寻找可伪造size处(发现notice – 4即rifle_cnt数值可控)
              - 3. add 0x40个结构体后，rifle_cnt变为0x40(order时，沿单链表依次free，此处将next都设置为NULL，防止程序free这些chunk，使得构造的fake chunk能够在fastbin 链表头部)
              - 4. 需要使fake chunk的下一个chunk size合理(发现notice处可写入128字节，足够覆盖fake chunk size字节)
            - 4. order(fake chunk位于fastbin 头部)
            - 5. add 分配至fake chunk区域(更改notice地址为strlen_got)
            - 6. message，向strlen_got处填入system真实地址
            - 7. 解释为什么写入system_addr后还加入"; /bin/sh\x00"
              - strlen未覆盖前，执行strlen(notice)
              - strlen被覆盖后，执行system(p32(strlen_got))　显然此时参数不合适，不能获得shell
              - 解释system("ls"; "/bin/sh") = system("ls") + system(/bin/sh)
              - 7处参数区域为0xaaaa; /bin/sh\x00　执行system(p32(strlen_got)) + system("bin/sh")
            - 8. 获得shell
      - Unsorted bin
        - 利用方法
          - 1. 修改unsorted bin bk字段为target_addr – 2*SIZE_ZE
          - 2. 注意释放目标chunk时不被top_chunk合并
        - 结果　*target_addr = unsorted_bin_addr
        - 作用
          - 修改循环次数，使程序多次循环
          - 修改global_max_fast(fastbin最大值)，使得更多的chunk被视为fast bin
- 主要参考ctf_wiki，对于难点，要点做较为详细的解释，仅供新手入门及知识点梳理--------povcfe