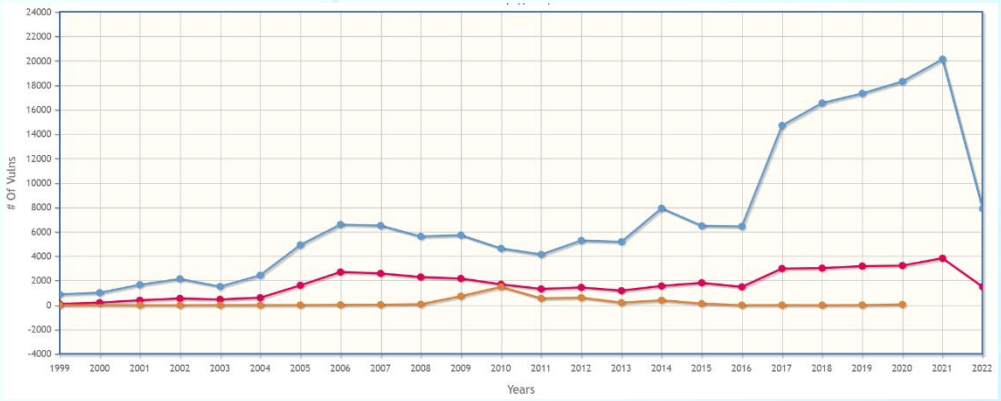
《软件安全漏洞分析与发现》

软件漏洞基础知识和原理

闫佳
中国科学院软件研究所
2025年5月14日

1.1 背景

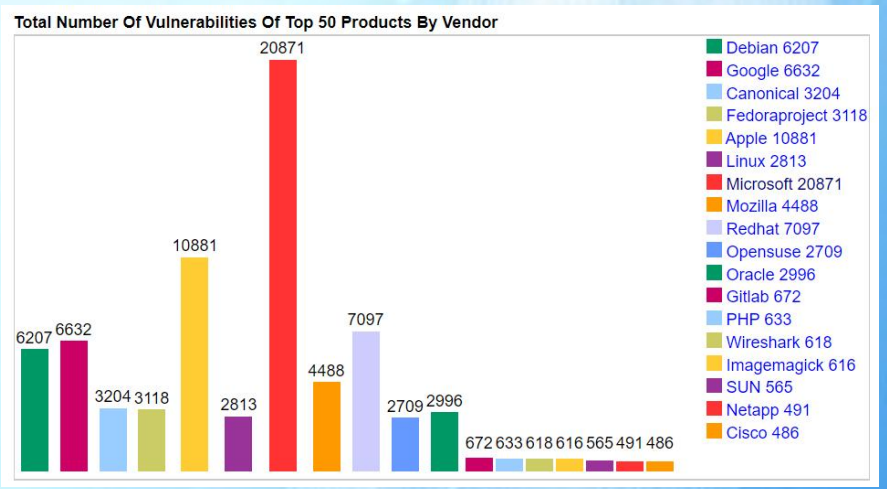
- 软件漏洞是信息安全的首要威胁
  - 高级持续性威胁(Advanced Persistent Threat, APT) 通常会采用高威胁软件漏洞突破防御体系
    - Stuxnet将伊朗核计划推迟了两年
  - 漏洞很难避免
    - Google, Microsoft在内的知名公司纷纷采用各种手段提高代码质量和安全检查能力，漏洞仍然不断出现



漏洞数量

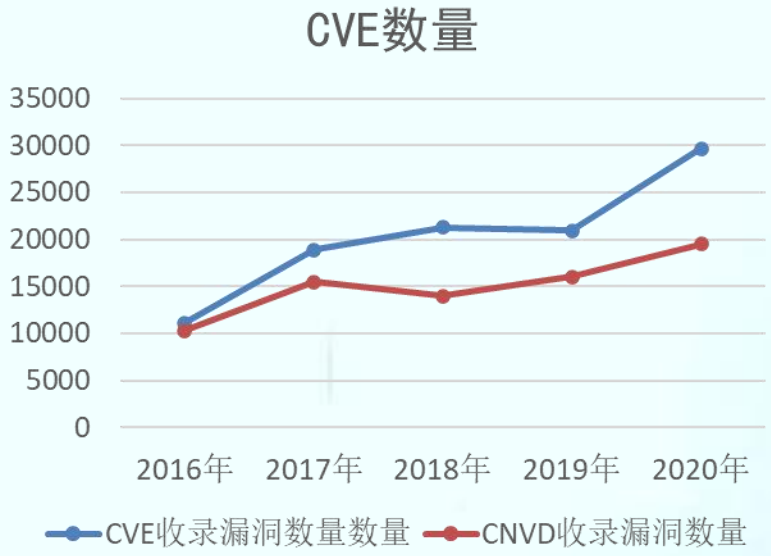
代码执行漏洞数量

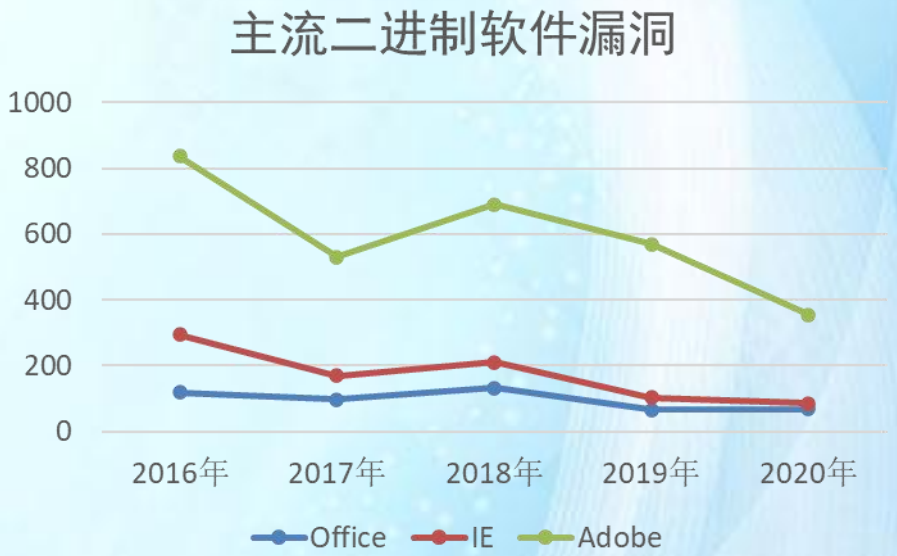
可用exploit数量



1.1 背景

- 国际权威漏洞数据库CVE收录漏洞数量逐年攀升，但高价值可利用漏洞仍然稀缺





国内外漏洞征集奖励计划

漏洞征集赏金计划	征集机构	高价值漏洞赏金金额
女娲计划	奇安信	20万-120万人民币
Zerodium计划	趋势科技	5万-250万美元
BugHunter	谷歌	1万美元+
Microsoft Bounty Programs	微软	10万-20万美元

1.1 背景

- 软件漏洞定义
  - 指能够引起严重后果的计算机安全缺陷，该缺陷使得系统或其应用数据的保密性、完整性、可用性、访问控制、监测机制等面临威胁
- 软件漏洞的特点
  - 机理复杂
    - 内存破坏漏洞：堆溢出、释放后重用、类型混淆
    - 程序执行时序逻辑引起漏洞
  - 产生原因多样
    - 栈溢出、堆溢出、整数溢出、格式化字符串、双重释放、内存泄漏
    - XSS、SQL注入等
  - 利用多样性
    - 程序崩溃（程序直接退出，服务宕机）
    - 拒绝服务（程序进入大规模数据运算或死循环，无法响应请求）
    - 攻击代码执行（注入ShellCode执行）

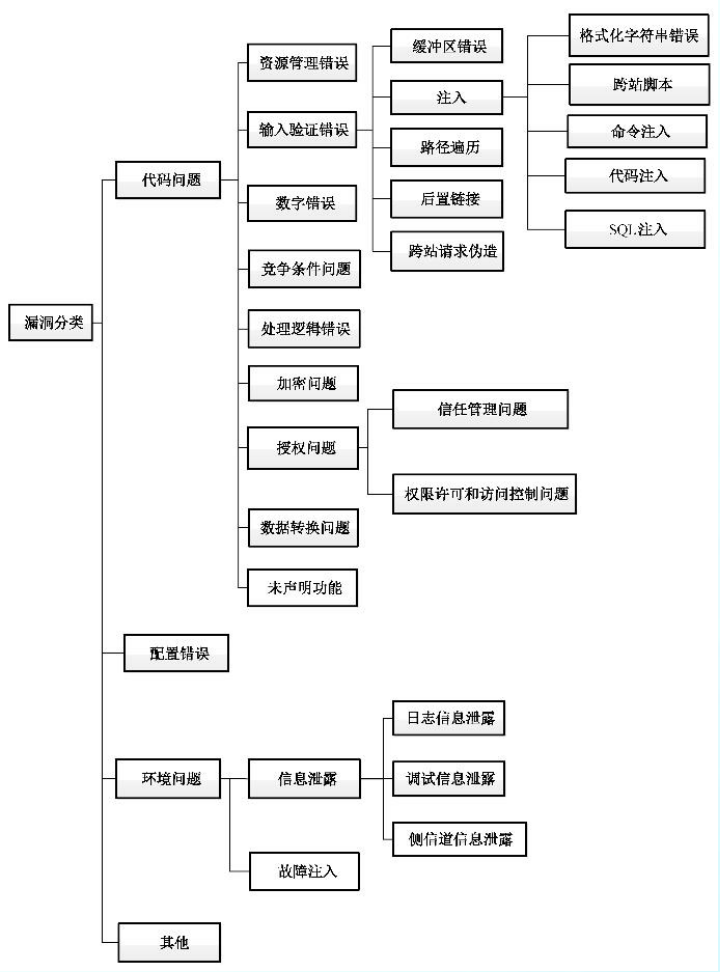
1.2 软件漏洞分类方法

- 软件漏洞分类方法
  - 国外安全机构和厂商的指标体系和量化评估技术
    - NVD（美国国家漏洞数据库）
    - US-CERT(美国应急响应组)
    - Google、Github、RedHat、Microsoft 等生产厂商和机构
  - 我国的漏洞数据库
    - CNNVD（中国国家信息安全漏洞库：中国信息安全测评中心）
    - CNVD（国家信息安全漏洞共享平台：CNCERT）
    - NVDB（网络安全威胁和漏洞信息共享平台：工信部）

1.2 软件漏洞分类方法

– 软件漏洞分类方法

- – 《信息安全技术 网络安全漏洞分类分级指南》2020年11月发布
  - – 分类准则：基于漏洞产生或触发的技术原因对漏洞进行划分



- **代码问题**：指网络产品和服务的代码开发过程中因设计或实现不当而导致的漏洞
- **配置错误**：指网络产品和服务或组件在使用过程中因配置文件、配置参数或因默认不安全的配置状态而产生的漏洞
  - 例如弱口令、错误配置证书等
- **环境问题**：因受影响组件部署运行环境的原因导致的安全问题
  - 例如侧信道、故障注入等

1.2 软件漏洞分类方法

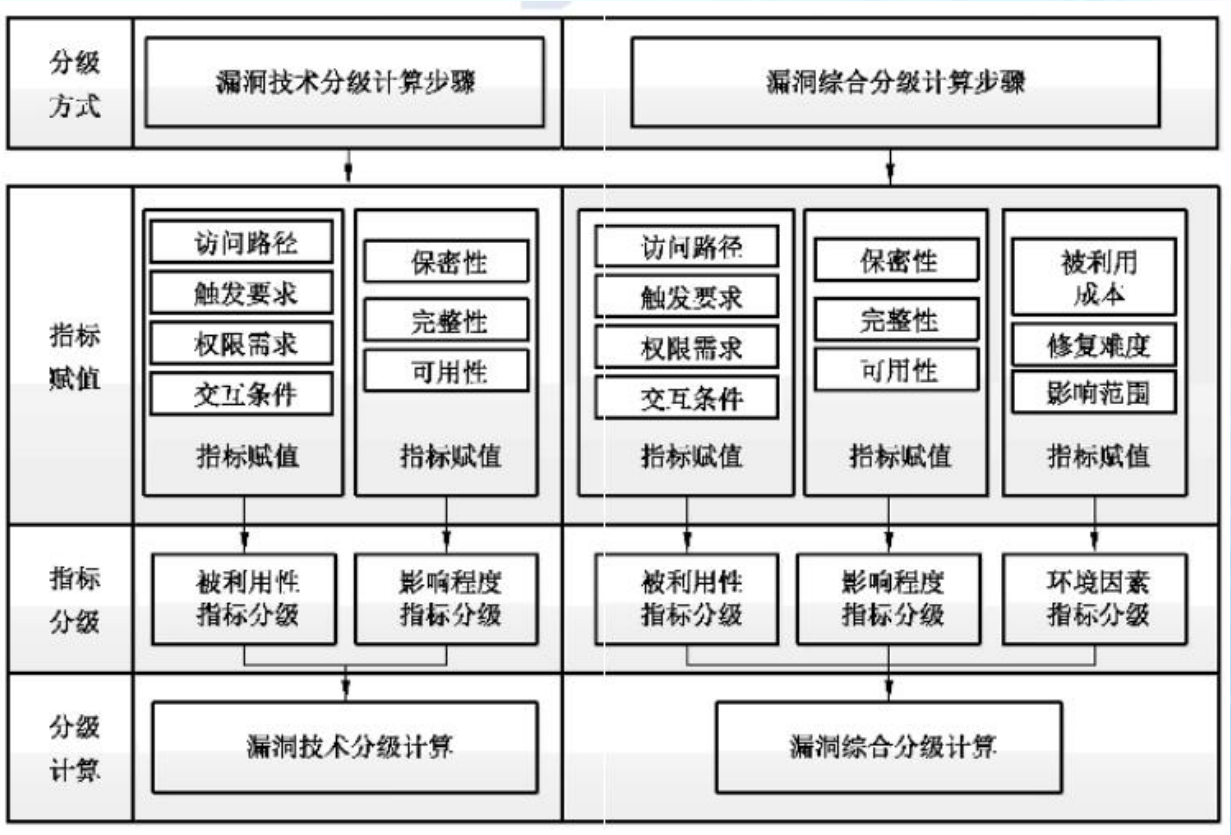
TCA

– 其他软件漏洞分类方法

- CVE 漏洞分类方法 ： CWE
  - 分类准则：根据程序属性特征、存在范围划分为不同的类别
  - 主要分类：包括跨站脚本、缓冲区溢出、SQL注入、PHP远程包含、目录遍历、信息泄漏、格式化字符串等漏洞类型等37种类型的漏洞
- Microsoft分类方法
  - 分类准则：在产品类别、漏洞产生的影响和利用方法两个维度上对软件漏洞分类
- Fortify分类方法
  - 分类准则：针对源代码，在程序编码的层次上归纳出在软件实现过程中的各种代码缺陷，把具有相同特征的代码缺陷划归为一类，间接实现了漏洞分类

1.2 软件漏洞分类方法

- 软件漏洞分级方法
  - 采用分级的方式网络安全漏洞潜在危害的程度进行描述《信息安全技术 网络安全漏洞分类分级指南》
    - 技术分级和综合分级（超危、高危、中危、低危）



1.2 软件漏洞分类方法

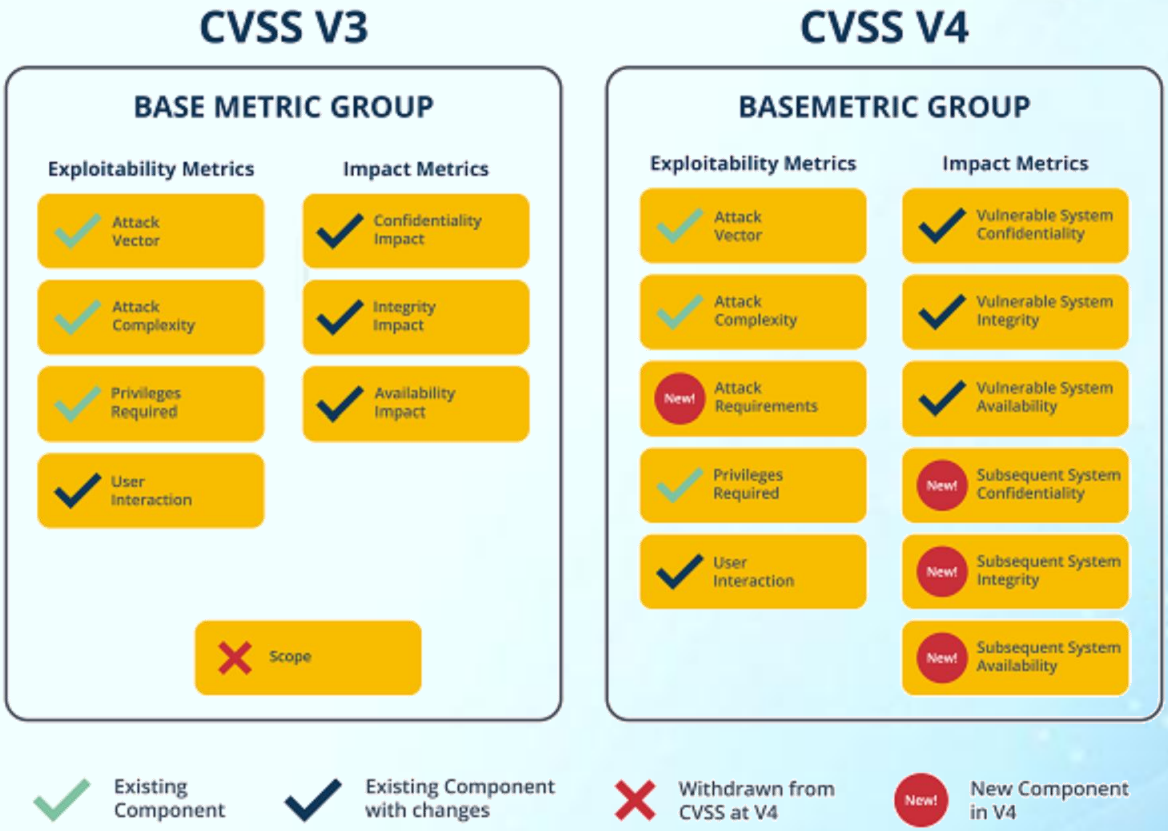
- 软件漏洞分级方法
  - 采用分级的方式网络安全漏洞潜在危害的程度进行描述《信息安全技术 网络安全漏洞分类分级指南》
    - 技术分级和综合分级（超危、高危、中危、低危）

示例：心脏滴血漏洞的分级

指标类	指标子类	描述	赋值说明	分级说明
被利用性	访问路径	通过网络远程访问	网络	9
	触发要求	无需特定环境，普通环境即可触发	低	
	权限需求	无需任何特权信息或身份验证	无	
	交互条件	漏洞触发无需用户或系统的参与或配合	不需要	
影响程度	保密性	攻击者从内存中可读取多达 64 KB 的数据，通过该漏洞读取每次攻击泄露出来的信息，可轻松获取到服务器的私钥、用户 cookie 和密码等	严重	4
	完整性	漏洞对完整性不产生影响	无	
	可用性	漏洞对可用性不产生影响	无	
环境因素	被利用成本	协议本身漏洞，直接暴露于公网之下，容易被利用	低	7
	修复难度	已有较为完善的修复方案，修复难度不大	低	
	影响范围	影响范围广泛	高	

1.2 软件漏洞分类方法

- 其他软件漏洞分级评价方法
  - CVSS（通用漏洞评价体系）
    - 使用0~10 之间的数字进行风险评价



CVSS v4修改内容

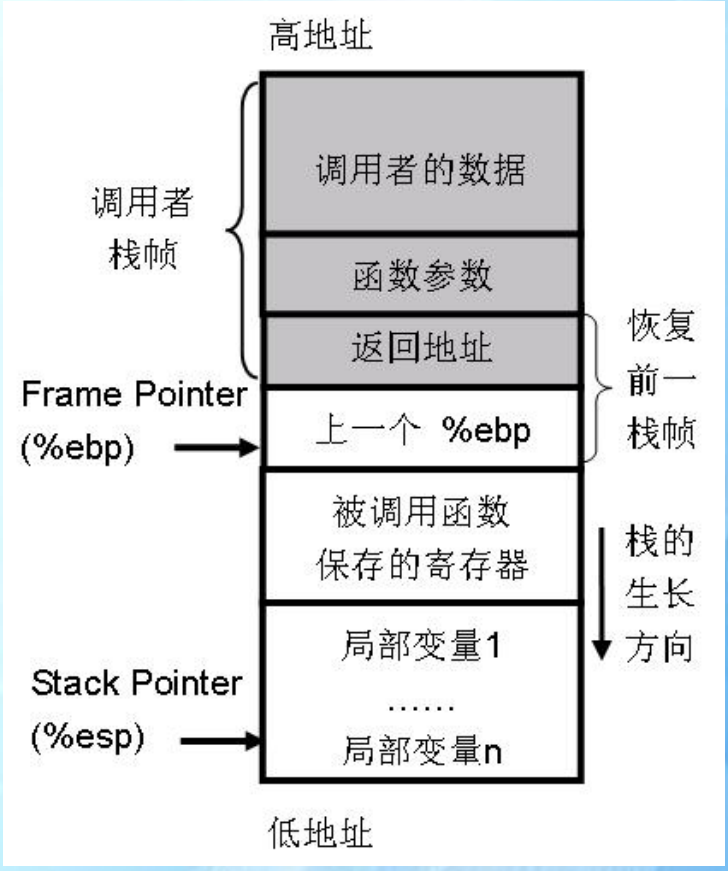
- 攻击发生的条件（Attack Requirements）
- 危害范围精细化描述评估
  - Vulnerable System
  - Subsequent System

1.3 典型软件漏洞类型

- 典型软件漏洞类型
  - 危害最大和最常见的控制流劫持类漏洞
    - 栈溢出漏洞
      - 最早被发现，影响最大、持续时间最长的漏洞
    - 格式化字符串漏洞
      - 经典漏洞类型，目前已大幅减少
    - 堆溢出漏洞
      - 栈溢出漏洞快速减少后的主要漏洞类型，是当前漏洞分析和研究的重点
    - 释放后重用漏洞（UAF）
      - 浏览器等脚本解析程序存在的主要漏洞
    - 整数溢出等漏洞
      - 主要出现在图片解析程序、富文本格式文件解析程序中，引发栈溢出或堆溢出
  - 其他漏洞
    - SQL注入漏洞
      - 程序对输入校验不严格，导致原有的程序逻辑失效
    - 跨站脚本漏洞（XSS）
      - 主要出现在web应用中，允许恶意web用户将代码植入到提供给其它用户使用的页面中

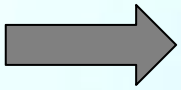
1.3 典型软件漏洞类型

- 栈溢出漏洞
  - 定义：
    - 攻击者通过恶意操控内存复制操作，改写不应当被篡改的栈上其他数据
  - 栈溢出漏洞产生根源
    - 栈中数据可写
    - 栈中的数据可当作代码执行
  - 软件如何使用栈
    - 函数调用
      - 栈中将被依次压入：参数，返回地址，EBP，局部变量
    - 栈的使用从高地址向低地址方向延伸



函数调用栈的工作方式（cdecl）

```
int callee(int a, int b, int c) {
    return a + b + c;
}

int caller(void) {
    int ret;
    ret = callee(1, 2, 3);
    ret += 4;
    return ret;
}
```

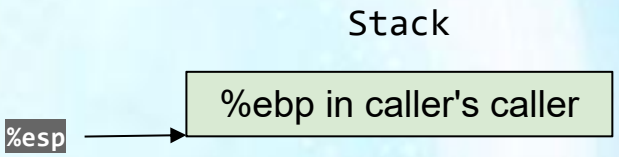


```
00000012 <caller>:
  12:   55                push   %ebp
  13:   89 e5             mov    %esp,%ebp
  15:   83 ec 10          sub    $0x10,%esp
  18:   6a 03             push   $0x3
  1a:   6a 02             push   $0x2
  1c:   6a 01             push   $0x1
  1e:   e8 fc ff ff ff    call   1f <caller+0xd>
  23:   83 c4 0c          add    $0xc,%esp
  26:   89 45 fc          mov    %eax,-0x4(%ebp)
  29:   83 45 fc 04       addl   $0x4,-0x4(%ebp)
  2d:   8b 45 fc          mov    -0x4(%ebp),%eax
  30:   c9                leave
  31:   c3                ret

00000000 <callee>:
   0:   55           push   %ebp
   1:   89 e5        mov    %esp,%ebp
   3:   8b 55 08     mov    0x8(%ebp),%edx
   6:   8b 45 0c     mov    0xc(%ebp),%eax
   9:   01 c2        add    %eax,%edx
   b:   8b 45 10     mov    0x10(%ebp),%eax
   e:   01 d0        add    %edx,%eax
  10:   5d           pop    %ebp
  11:   c3           ret
```

- x86
  - 使用栈来传递参数
  - 使用 eax 存放返回值
- amd64
  - 前6个参数依次存放于 rdi、rsi、rdx、rcx、r8、r9 寄存器中
  - 第7个以后的参数存放于栈中

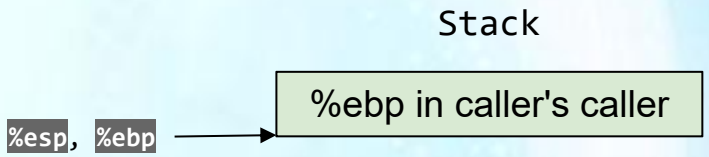
函数调用栈的工作方式（cdecl）

```
00000012 <caller>:
  12:   55              push   %ebp
%eip ->13:   89 e5           mov    %esp,%ebp
  15:   83 ec 10        sub    $0x10,%esp
  18:   6a 03           push   $0x3
  1a:   6a 02           push   $0x2
  1c:   6a 01           push   $0x1
  1e:   e8 fc ff ff ff  call   1f <caller+0xd>
  23:   83 c4 0c        add    $0xc,%esp
  26:   89 45 fc        mov    %eax,-0x4(%ebp)
  29:   83 45 fc 04     addl   $0x4,-0x4(%ebp)
  2d:   8b 45 fc        mov    -0x4(%ebp),%eax
  30:   c9              leave
  31:   c3              ret

00000000 <callee>:
   0:   55          push   %ebp
   1:   89 e5       mov    %esp,%ebp
   3:   8b 55 08    mov    0x8(%ebp),%edx
   6:   8b 45 0c    mov    0xc(%ebp),%eax
   9:   01 c2       add    %eax,%edx
   b:   8b 45 10    mov    0x10(%ebp),%eax
   e:   01 d0       add    %edx,%eax
  10:   5d          pop    %ebp
  11:   c3          ret
```



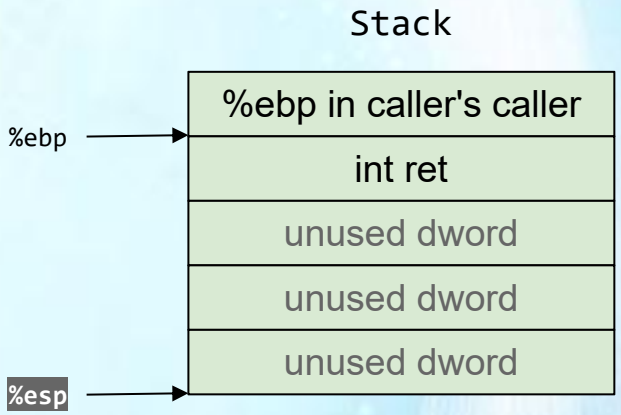
函数调用栈的工作方式（cdecl）

```
00000012 <caller>:
  12:   55              push   %ebp
  13:   89 e5           mov    %esp,%ebp
%eip ->15:   83 ec 10       sub    $0x10,%esp
  18:   6a 03           push   $0x3
  1a:   6a 02           push   $0x2
  1c:   6a 01           push   $0x1
  1e:   e8 fc ff ff ff  call   1f <caller+0xd>
  23:   83 c4 0c        add    $0xc,%esp
  26:   89 45 fc        mov    %eax,-0x4(%ebp)
  29:   83 45 fc 04     addl   $0x4,-0x4(%ebp)
  2d:   8b 45 fc        mov    -0x4(%ebp),%eax
  30:   c9              leave
  31:   c3              ret

00000000 <callee>:
   0:   55           push   %ebp
   1:   89 e5        mov    %esp,%ebp
   3:   8b 55 08     mov    0x8(%ebp),%edx
   6:   8b 45 0c     mov    0xc(%ebp),%eax
   9:   01 c2        add    %eax,%edx
   b:   8b 45 10     mov    0x10(%ebp),%eax
   e:   01 d0        add    %edx,%eax
  10:   5d           pop    %ebp
  11:   c3           ret
```



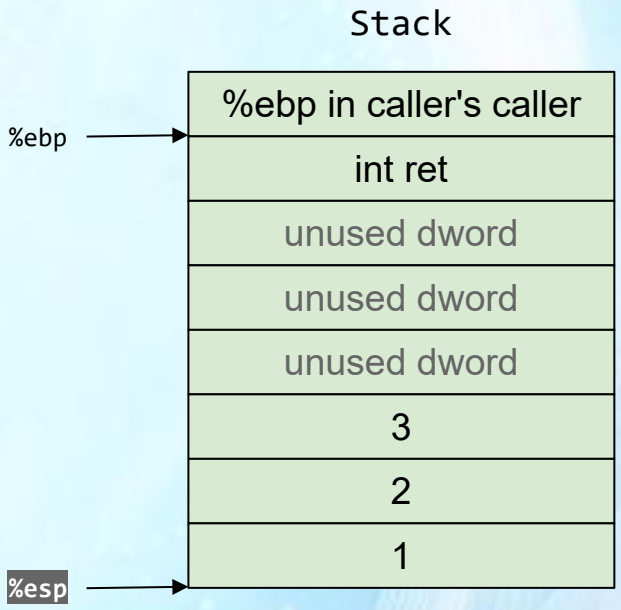
函数调用栈的工作方式（cdecl）

```
00000012 <caller>:
  12:   55              push   %ebp
  13:   89 e5           mov    %esp,%ebp
  15:   83 ec 10        sub    $0x10,%esp
%eip ->
  18:   6a 03           push   $0x3
  1a:   6a 02           push   $0x2
  1c:   6a 01           push   $0x1
  1e:   e8 fc ff ff ff  call   1f <caller+0xd>
  23:   83 c4 0c        add    $0xc,%esp
  26:   89 45 fc        mov    %eax,-0x4(%ebp)
  29:   83 45 fc 04     addl   $0x4,-0x4(%ebp)
  2d:   8b 45 fc        mov    -0x4(%ebp),%eax
  30:   c9              leave
  31:   c3              ret

00000000 <callee>:
   0:   55          push   %ebp
   1:   89 e5       mov    %esp,%ebp
   3:   8b 55 08    mov    0x8(%ebp),%edx
   6:   8b 45 0c    mov    0xc(%ebp),%eax
   9:   01 c2       add    %eax,%edx
   b:   8b 45 10    mov    0x10(%ebp),%eax
   e:   01 d0       add    %edx,%eax
  10:   5d          pop    %ebp
  11:   c3          ret
```



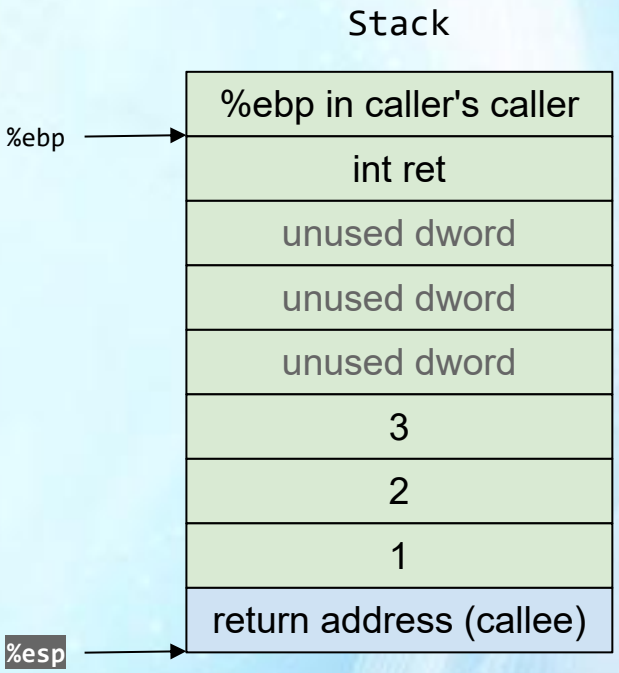
函数调用栈的工作方式（cdecl）

```
00000012 <caller>:
  12:   55              push   %ebp
  13:   89 e5           mov    %esp,%ebp
  15:   83 ec 10        sub    $0x10,%esp
  18:   6a 03           push   $0x3
  1a:   6a 02           push   $0x2
  1c:   6a 01           push   $0x1
%eip ->1e:   e8 fc ff ff ff  call   1f <caller+0xd>
  23:   83 c4 0c        add    $0xc,%esp
  26:   89 45 fc        mov    %eax,-0x4(%ebp)
  29:   83 45 fc 04     addl   $0x4,-0x4(%ebp)
  2d:   8b 45 fc        mov    -0x4(%ebp),%eax
  30:   c9              leave
  31:   c3              ret

00000000 <callee>:
   0:   55           push   %ebp
   1:   89 e5        mov    %esp,%ebp
   3:   8b 55 08     mov    0x8(%ebp),%edx
   6:   8b 45 0c     mov    0xc(%ebp),%eax
   9:   01 c2        add    %eax,%edx
   b:   8b 45 10     mov    0x10(%ebp),%eax
   e:   01 d0        add    %edx,%eax
  10:   5d           pop    %ebp
  11:   c3           ret
```



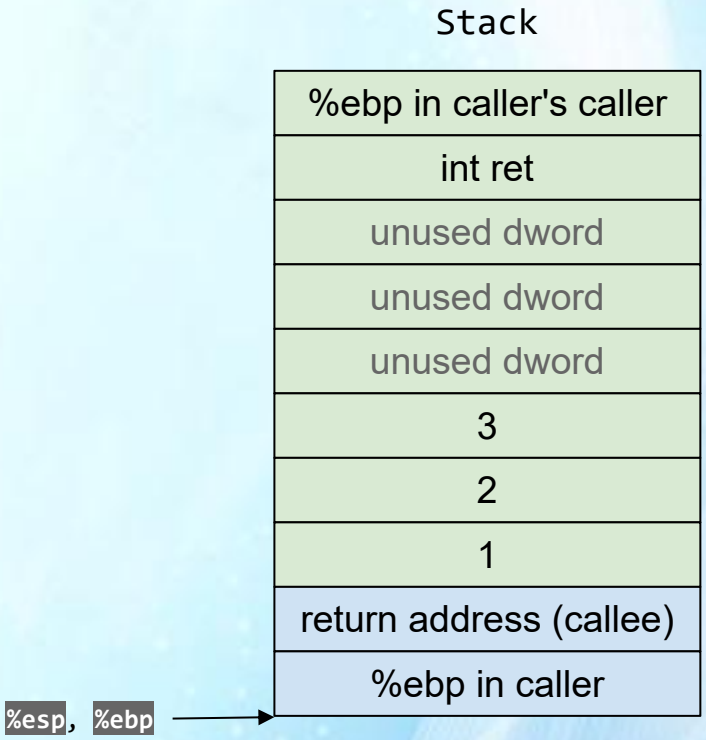
函数调用栈的工作方式（cdecl）

```
00000012 <caller>:
  12:   55              push   %ebp
  13:   89 e5           mov    %esp,%ebp
  15:   83 ec 10        sub    $0x10,%esp
  18:   6a 03           push   $0x3
  1a:   6a 02           push   $0x2
  1c:   6a 01           push   $0x1
  1e:   e8 fc ff ff ff  call   1f <callee>
  23:   83 c4 0c        add    $0xc,%esp
  26:   89 45 fc        mov    %eax,-0x4(%ebp)
  29:   83 45 fc 04     addl   $0x4,-0x4(%ebp)
  2d:   8b 45 fc        mov    -0x4(%ebp),%eax
  30:   c9              leave
  31:   c3              ret

00000000 <callee>:
%eip ->  0:   55           push   %ebp
   1:   89 e5        mov    %esp,%ebp
   3:   8b 55 08     mov    0x8(%ebp),%edx
   6:   8b 45 0c     mov    0xc(%ebp),%eax
   9:   01 c2        add    %eax,%edx
   b:   8b 45 10     mov    0x10(%ebp),%eax
   e:   01 d0        add    %edx,%eax
  10:   5d           pop    %ebp
  11:   c3           ret
```



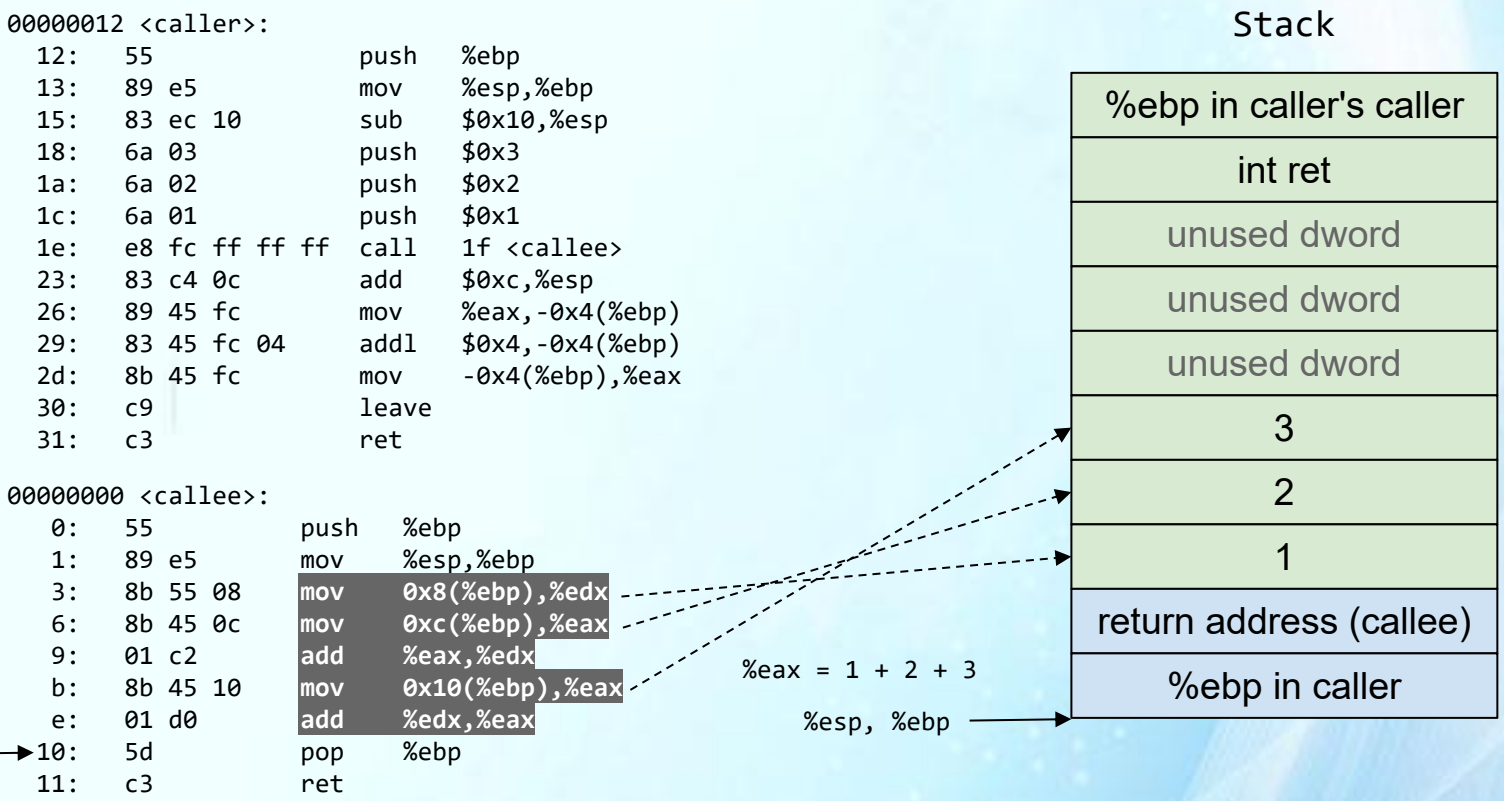
函数调用栈的工作方式（cdecl）

```
00000012 <caller>:
  12:   55              push   %ebp
  13:   89 e5           mov    %esp,%ebp
  15:   83 ec 10        sub    $0x10,%esp
  18:   6a 03           push   $0x3
  1a:   6a 02           push   $0x2
  1c:   6a 01           push   $0x1
  1e:   e8 fc ff ff ff  call   1f <callee>
  23:   83 c4 0c        add    $0xc,%esp
  26:   89 45 fc        mov    %eax,-0x4(%ebp)
  29:   83 45 fc 04     addl   $0x4,-0x4(%ebp)
  2d:   8b 45 fc        mov    -0x4(%ebp),%eax
  30:   c9              leave
  31:   c3              ret

00000000 <callee>:
   0:   55              push   %ebp
   1:   89 e5           mov    %esp,%ebp
%eip →  3:   8b 55 08        mov    0x8(%ebp),%edx
   6:   8b 45 0c        mov    0xc(%ebp),%eax
   9:   01 c2           add    %eax,%edx
   b:   8b 45 10        mov    0x10(%ebp),%eax
   e:   01 d0           add    %edx,%eax
  10:   5d              pop    %ebp
  11:   c3              ret
```



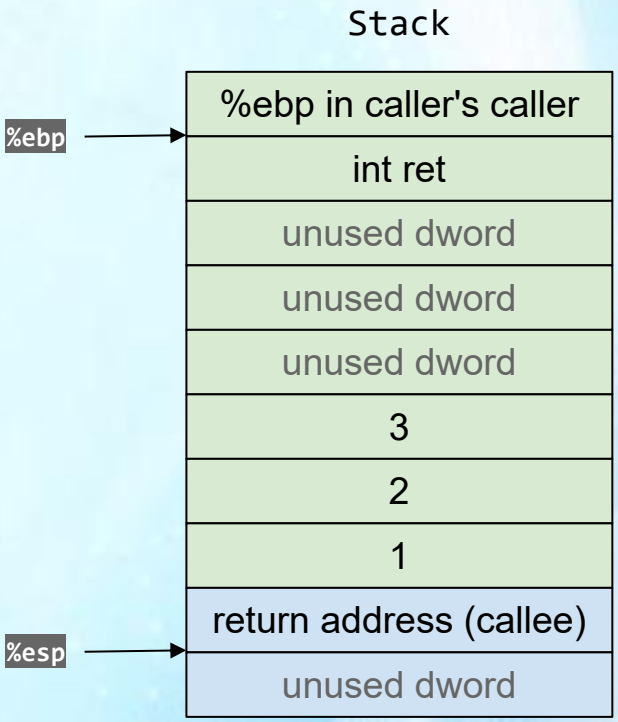
函数调用栈的工作方式（cdecl）

```
00000012 <caller>:
  12:   55              push   %ebp
  13:   89 e5           mov    %esp,%ebp
  15:   83 ec 10        sub    $0x10,%esp
  18:   6a 03           push   $0x3
  1a:   6a 02           push   $0x2
  1c:   6a 01           push   $0x1
  1e:   e8 fc ff ff ff  call   1f <callee>
  23:   83 c4 0c        add    $0xc,%esp
  26:   89 45 fc        mov    %eax,-0x4(%ebp)
  29:   83 45 fc 04     addl   $0x4,-0x4(%ebp)
  2d:   8b 45 fc        mov    -0x4(%ebp),%eax
  30:   c9              leave
  31:   c3              ret

00000000 <callee>:
   0:   55              push   %ebp
   1:   89 e5           mov    %esp,%ebp
   3:   8b 55 08        mov    0x8(%ebp),%edx
   6:   8b 45 0c        mov    0xc(%ebp),%eax
   9:   01 c2           add    %eax,%edx
   b:   8b 45 10        mov    0x10(%ebp),%eax
   e:   01 d0           add    %edx,%eax
%eip ---> 10:   5d      pop    %ebp
  11:   c3              ret
```



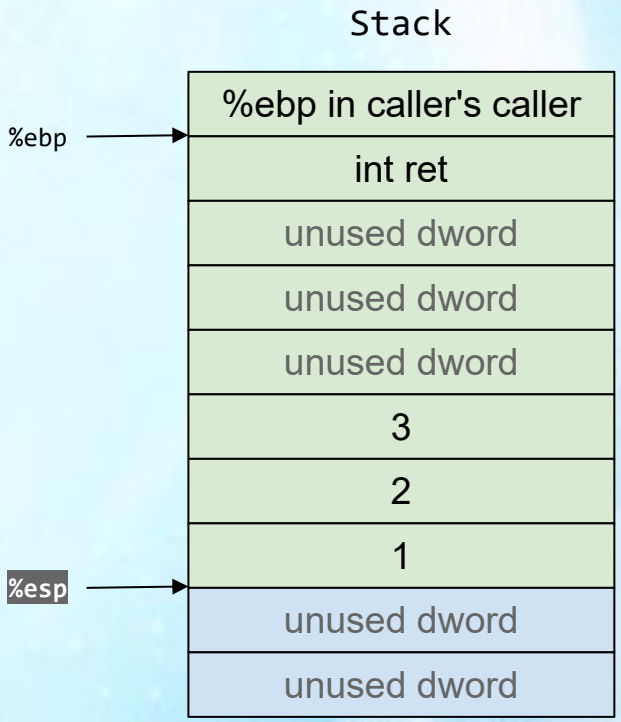
函数调用栈的工作方式（cdecl）

```
00000012 <caller>:
  12:   55              push   %ebp
  13:   89 e5           mov    %esp,%ebp
  15:   83 ec 10        sub    $0x10,%esp
  18:   6a 03           push   $0x3
  1a:   6a 02           push   $0x2
  1c:   6a 01           push   $0x1
  1e:   e8 fc ff ff ff  call   1f <callee>
  23:   83 c4 0c        add    $0xc,%esp
  26:   89 45 fc        mov    %eax,-0x4(%ebp)
  29:   83 45 fc 04     addl   $0x4,-0x4(%ebp)
  2d:   8b 45 fc        mov    -0x4(%ebp),%eax
  30:   c9              leave
  31:   c3              ret

00000000 <callee>:
   0:   55          push   %ebp
   1:   89 e5       mov    %esp,%ebp
   3:   8b 55 08    mov    0x8(%ebp),%edx
   6:   8b 45 0c    mov    0xc(%ebp),%eax
   9:   01 c2       add    %eax,%edx
   b:   8b 45 10    mov    0x10(%ebp),%eax
   e:   01 d0       add    %edx,%eax
  10:   5d          pop    %ebp
%eip → 11:   c3          ret
```



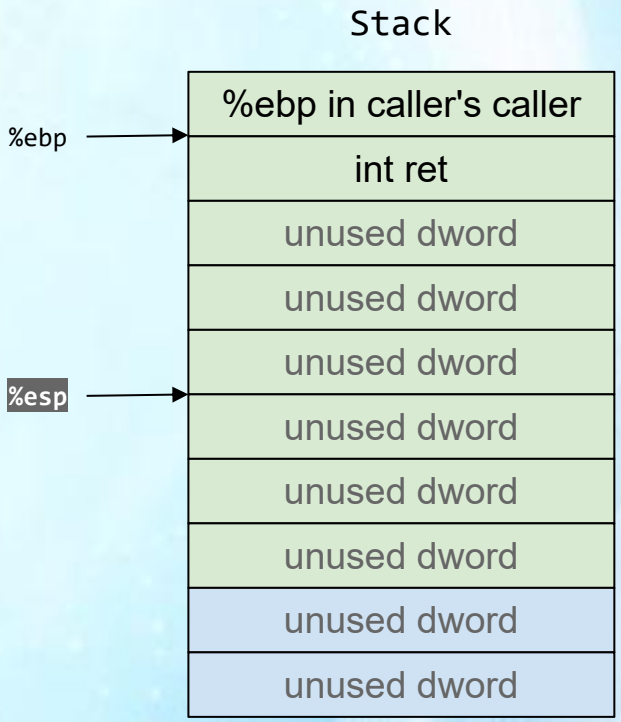
函数调用栈的工作方式（cdecl）

```
00000012 <caller>:
  12:   55              push   %ebp
  13:   89 e5           mov    %esp,%ebp
  15:   83 ec 10        sub    $0x10,%esp
  18:   6a 03           push   $0x3
  1a:   6a 02           push   $0x2
  1c:   6a 01           push   $0x1
  1e:   e8 fc ff ff ff  call   1f <callee>
%eip→ 23:   83 c4 0c        add    $0xc,%esp
  26:   89 45 fc        mov    %eax,-0x4(%ebp)
  29:   83 45 fc 04     addl   $0x4,-0x4(%ebp)
  2d:   8b 45 fc        mov    -0x4(%ebp),%eax
  30:   c9              leave
  31:   c3              ret

00000000 <callee>:
   0:   55           push   %ebp
   1:   89 e5        mov    %esp,%ebp
   3:   8b 55 08     mov    0x8(%ebp),%edx
   6:   8b 45 0c     mov    0xc(%ebp),%eax
   9:   01 c2        add    %eax,%edx
   b:   8b 45 10     mov    0x10(%ebp),%eax
   e:   01 d0        add    %edx,%eax
  10:   5d           pop    %ebp
  11:   c3           ret
```



函数调用栈的工作方式（cdecl）

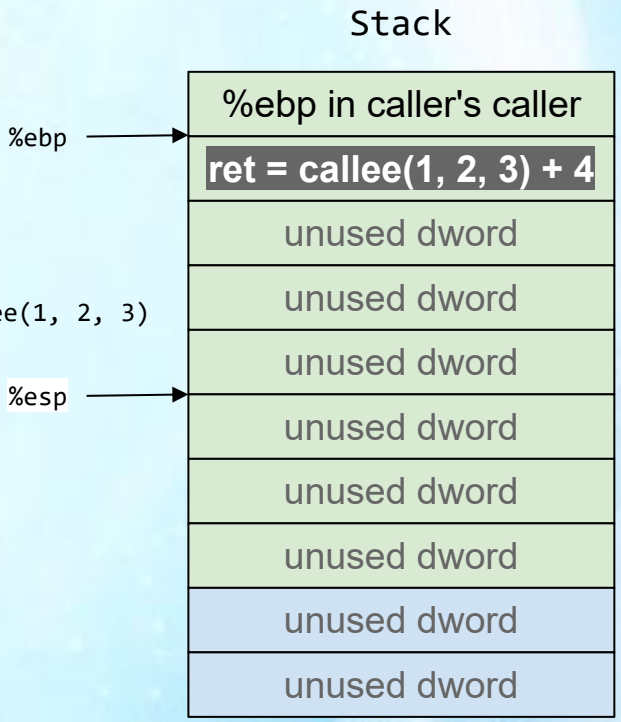
```
00000012 <caller>:
  12:   55              push   %ebp
  13:   89 e5           mov    %esp,%ebp
  15:   83 ec 10        sub    $0x10,%esp
  18:   6a 03           push   $0x3
  1a:   6a 02           push   $0x2
  1c:   6a 01           push   $0x1
  1e:   e8 fc ff ff ff  call   1f <callee>
  23:   83 c4 0c        add    $0xc,%esp
%eip → 26:   89 45 fc        mov    %eax,-0x4(%ebp)
  29:   83 45 fc 04     addl   $0x4,-0x4(%ebp)
  2d:   8b 45 fc        mov    -0x4(%ebp),%eax
  30:   c9              leave
  31:   c3              ret

00000000 <callee>:
   0:   55           push   %ebp
   1:   89 e5        mov    %esp,%ebp
   3:   8b 55 08     mov    0x8(%ebp),%edx
   6:   8b 45 0c     mov    0xc(%ebp),%eax
   9:   01 c2        add    %eax,%edx
   b:   8b 45 10     mov    0x10(%ebp),%eax
   e:   01 d0        add    %edx,%eax
  10:   5d           pop    %ebp
  11:   c3           ret
```



函数调用栈的工作方式（cdecl）

```
00000012 <caller>:
  12:   55              push   %ebp
  13:   89 e5           mov    %esp,%ebp
  15:   83 ec 10        sub    $0x10,%esp
  18:   6a 03           push   $0x3
  1a:   6a 02           push   $0x2
  1c:   6a 01           push   $0x1
  1e:   e8 fc ff ff ff  call   1f <callee>
  23:   83 c4 0c        add    $0xc,%esp
  26:   89 45 fc        mov    %eax,-0x4(%ebp)
  29:   83 45 fc 04     addl   $0x4,-0x4(%ebp)
  2d:   8b 45 fc        mov    -0x4(%ebp),%eax
%eip→ 30:   c9              leave
  31:   c3              ret

00000000 <callee>:
   0:   55              push   %ebp
   1:   89 e5           mov    %esp,%ebp
   3:   8b 55 08        mov    0x8(%ebp),%edx
   6:   8b 45 0c        mov    0xc(%ebp),%eax
   9:   01 c2           add    %eax,%edx
   b:   8b 45 10        mov    0x10(%ebp),%eax
   e:   01 d0           add    %edx,%eax
  10:   5d              pop    %ebp
  11:   c3              ret
```

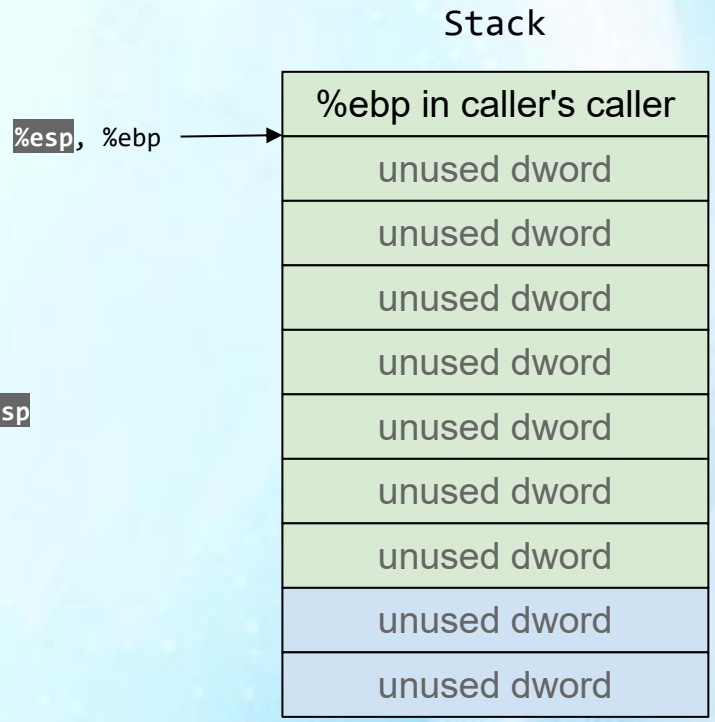
%eax = callee(1, 2, 3)



函数调用栈的工作方式（cdecl）

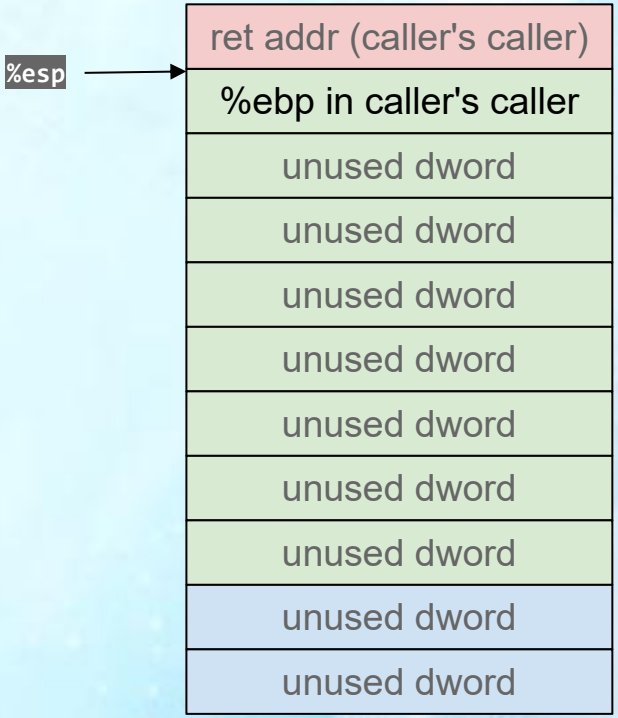
```
00000012 <caller>:
  12:   55              push   %ebp
  13:   89 e5           mov    %esp,%ebp
  15:   83 ec 10        sub    $0x10,%esp
  18:   6a 03           push   $0x3
  1a:   6a 02           push   $0x2
  1c:   6a 01           push   $0x1
  1e:   e8 fc ff ff ff  call   1f <callee>
  23:   83 c4 0c        add    $0xc,%esp
  26:   89 45 fc        mov    %eax,-0x4(%ebp)
  29:   83 45 fc 04     addl   $0x4,-0x4(%ebp)
  2d:   8b 45 fc        mov    -0x4(%ebp),%eax
  30:   c9              leave
%eip
  31:   c3              ret

00000000 <callee>:
   0:   55              push   %ebp
   1:   89 e5           mov    %esp,%ebp
   3:   8b 55 08        mov    0x8(%ebp),%edx
   6:   8b 45 0c        mov    0xc(%ebp),%eax
   9:   01 c2           add    %eax,%edx
   b:   8b 45 10        mov    0x10(%ebp),%eax
   e:   01 d0           add    %edx,%eax
  10:   5d              pop    %ebp
  11:   c3              ret
```

leave → mov %ebp, $esp; pop %ebp



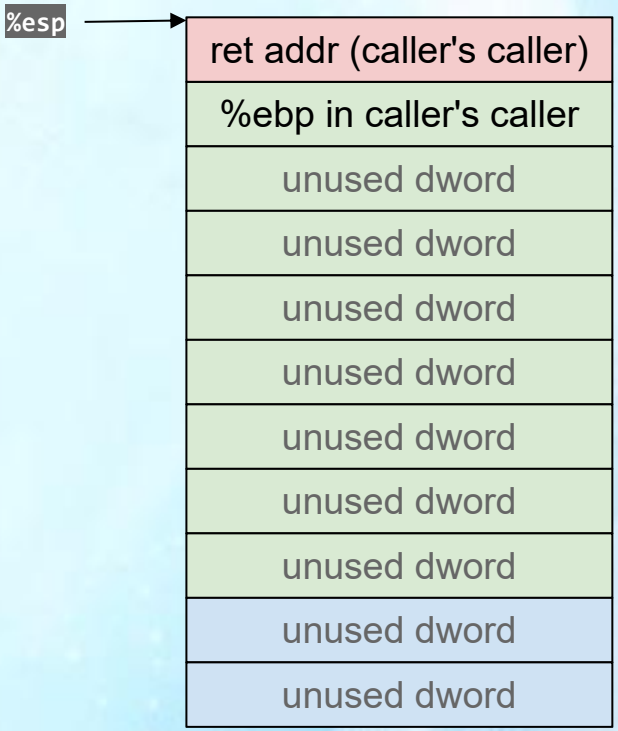
函数调用栈的工作方式（cdecl）

```
00000012 <caller>:
  12:   55              push   %ebp
  13:   89 e5           mov    %esp,%ebp
  15:   83 ec 10        sub    $0x10,%esp
  18:   6a 03           push   $0x3
  1a:   6a 02           push   $0x2
  1c:   6a 01           push   $0x1
  1e:   e8 fc ff ff ff  call   1f <callee>
  23:   83 c4 0c        add    $0xc,%esp
  26:   89 45 fc        mov    %eax,-0x4(%ebp)
  29:   83 45 fc 04     addl   $0x4,-0x4(%ebp)
  2d:   8b 45 fc        mov    -0x4(%ebp),%eax
  30:   c9              leave  ->  mov   %ebp, $esp
%eip                                 pop   %ebp
  31:   c3              ret

00000000 <callee>:
   0:   55              push   %ebp
   1:   89 e5           mov    %esp,%ebp
   3:   8b 55 08        mov    0x8(%ebp),%edx
   6:   8b 45 0c        mov    0xc(%ebp),%eax
   9:   01 c2           add    %eax,%edx
   b:   8b 45 10        mov    0x10(%ebp),%eax
   e:   01 d0           add    %edx,%eax
  10:   5d              pop    %ebp
  11:   c3              ret
```



函数调用栈的工作方式（cdecl）

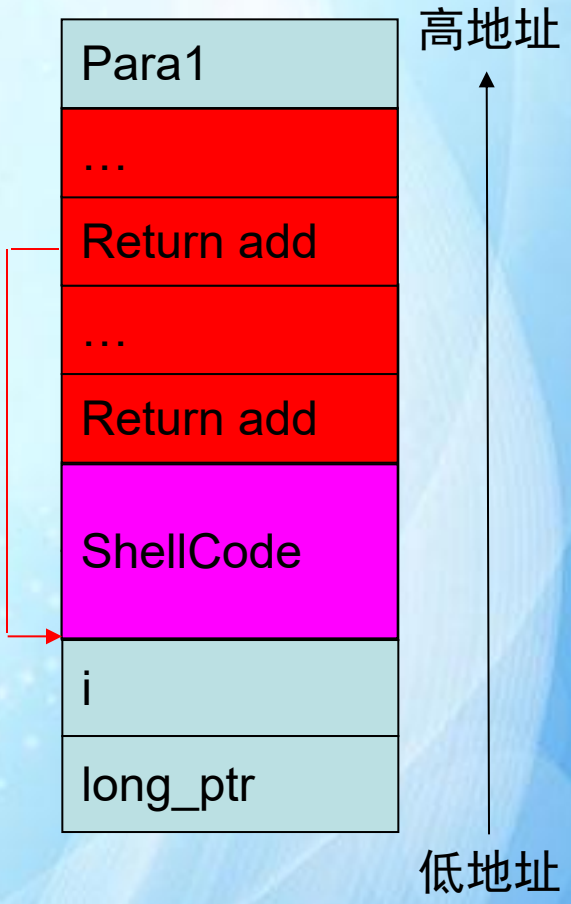
```
00000012 <caller>:
  12:   55              push   %ebp
  13:   89 e5           mov    %esp,%ebp
  15:   83 ec 10        sub    $0x10,%esp
  18:   6a 03           push   $0x3
  1a:   6a 02           push   $0x2
  1c:   6a 01           push   $0x1
  1e:   e8 fc ff ff ff  call   1f <callee>
  23:   83 c4 0c        add    $0xc,%esp
  26:   89 45 fc        mov    %eax,-0x4(%ebp)
  29:   83 45 fc 04     addl   $0x4,-0x4(%ebp)
  2d:   8b 45 fc        mov    -0x4(%ebp),%eax
  30:   c9              leave
  31:   c3              ret

00000000 <callee>:
   0:   55           push   %ebp
   1:   89 e5        mov    %esp,%ebp
   3:   8b 55 08     mov    0x8(%ebp),%edx
   6:   8b 45 0c     mov    0xc(%ebp),%eax
   9:   01 c2        add    %eax,%edx
   b:   8b 45 10     mov    0x10(%ebp),%eax
   e:   01 d0        add    %edx,%eax
  10:   5d           pop    %ebp
  11:   c3           ret
```



1.3 典型软件漏洞类型

- 栈溢出漏洞

  - 产生机制：
    - 先放入栈的函数返回地址在前，后放入栈的局部变量数据一旦过长，就会覆盖到前面的地址
  - 程序错误：
    - 程序未能检查输入数据和栈空间是否匹配，导致向栈中写入了超出其数据区域容纳能力的数据，返回地址就会被覆盖
  - 利用：
    - 当被覆盖的返回地址被返回指令引用时，程序控制流转移方向即可能被攻击者劫持



1.3 典型软件漏洞类型

- 栈溢出漏洞

常见的存在栈溢出的代码

```
int len;
char buf[0x100];
scanf("%d", &len);
read(0, buf, (unsigned int)len);
```

```
for(int i=0; i <= len; i++)
```

```
int i=0;
char buf[0x80];
fgets(buf, sizeof(buf), stdin);
for(i=0; i < sizeof(buf); i++) {
    if(buf[i] = 'n') {
        break;
    }
}
buf[i] = 0; // off by null
```

- 栈溢出漏洞

栈溢出实例

```
int main() {
    setbuf(stdin, NULL);
    setbuf(stdout, NULL);
    uint64_t i;
    char buf[0x80];
    char *ptr;

    puts("enter username:");
    for (i = 0, ptr = (char *) &buf; i <= sizeof(buf); i++) {
        *(ptr++) = (char) getchar();
    }
    return 0;
}
```

- 格式化字符串漏洞
  - 标准化输出函数的格式化字符串参数的漏洞

常见格式化字符串函数

函数	基本介绍
printf	输出到stdout
fprintf	输出到指定FILE流
vprintf	根据参数列表格式化输出到stdout
vfprintf	根据参数列表格式化输出到FILE流
sprintf	输出到字符串
snprintf	输出指定字节数到字符串
vsprintf	根据参数列表格式化输出到字符串
vsnprintf	根据参数列表格式化输出指定字节到字符串

1.3 典型软件漏洞类型

- 格式化字符串漏洞

%[parameter][flags][field width][.precision][length]type
例如 %2$08x %.8lf

parameter：n$ ，获取格式化字符串中的指定第 n 个参数

flags：在 width 设置后指定可以用来作为填充的内容之类的内容

field width：输出的最小宽度

precision：输出的最大长度

length，输出的长度

hh，输出一个字节

h，输出一个双字节

1.3 典型软件漏洞类型

TCA

- 格式化字符串漏洞

%[parameter][flags][field width][.precision][length]type
例如 %2$08x %.8lf

type

- d/i：有符号整数； u：无符号整数
- x/X：16进制，o：8进制
- s：所有字节
- c：char类型单个字符
- p，void * 型，输出对应变量的值。
  - printf("%p",a) 用地址的格式打印变量 a 的值
  - printf("%p", &a) 打印变量 a 所在的地址
- n，不输出字符，但是把已经成功输出的字符个数写入对应的整型指针参数所指的变量。
  - hhn 写一字节
  - hn 写两字节
  - n 写四字节
  - ln 32 位写四字节，64 位写八字节
  - lln 写八字节

1.3 典型软件漏洞类型

- 格式化字符串漏洞

```
#include<stdio.h>
 int main() {
    char t[256] = “AAAA-%p.%p.%p.%p.%p.%p.%p";
    printf(t);
    return 0;
}
```

X86 32位

```
→  basic ./test_32
AAAA-0x340.0x340.0x565a620c.0x340.0x340.0x340.0x41414141%
```

```
pwndbg> stack 40
00:0000|  esp 0xffffd2f0 —▸ 0xffffd30c ◂— 'AAAA-%p.%p.%p.%p.%p.%p.%p'
01:0004|-134 0xffffd2f4 ◂— 0x340
02:0008|-130 0xffffd2f8 ◂— 0x340
03:000c|-12c 0xffffd2fc —▸ 0x5655620c (main+31) ◂— add ebx, 0x2dc8
04:0010|-128 0xffffd300 ◂— 0x340
... ↓        2 skipped
```

1.3 典型软件漏洞类型

- 格式化字符串漏洞

```
#include<stdio.h>
 int main() {
    char t[256] = “AAAA-%p.%p.%p.%p.%p.%p.%p";
    printf(t);
    return 0;
}
```

X64 64位

```
AAAA-0x7ffeb8006598.0x70.0x5595d2b3b300.0x7fd593486f10.0x7fd5934a5040.0x2e70252d
41414141.0x70252e70252e7025$
```

64 位程序先使用 rdi、rsi、rdx、rcx、r8、r9 寄存器作为函数参数的前六个参数，多余的参数会依次压在栈上

```
pwndbg> i r
rax            0x0                 0
rbx            0x0                 0
rcx            0x5595d2b3b300      94101973480192
rdx            0x70                112
rsi            0x7ffeb8006598      140731985454488
rdi            0x7ffeb8006370      140731985453936
rbp            0x7ffeb8006480      0x7ffeb8006480
rsp            0x7ffeb8006370      0x7ffeb8006370
r8             0x7fd593486f10      140555275759376
r9             0x7fd5934a5040      140555275882560
r10            0x7fd59349f908      140555275860232
r11            0x7fd5934ba660      140555275970144
r12            0x7ffeb8006598      140731985454488
r13            0x5595d2b3b169      94101973479785
r14            0x0                 0
r15            0x7fd5934d9040      140555276095552
rip            0x5595d2b3b2df      0x5595d2b3b2df <m
eflags         0x246               [ PF ZF IF ]
```

1.3 典型软件漏洞类型

- 格式化字符串漏洞

存在格式化字符串漏洞的程序

```
#include<stdio.h>
int main() {
    setbuf(stdin, NULL);
    setbuf(stdout, NULL);
    char buf[0x100] = {};
    while (1) {
        puts("username:");
        read(0, buf, 0x300);
        printf(buf);
    }
    return 0;
}
```

- 利用方式一：覆盖内存

%k$n 可以覆盖第 k 个参数指向的地址为已经输出的字符数量

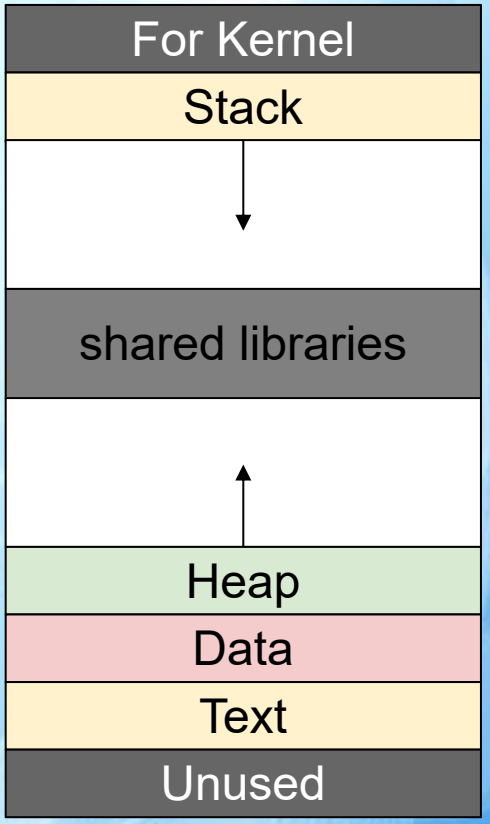
- 利用方式二：信息泄漏

%k $ p 可以打印第k个参数作为地址指向的内存内容

- 堆溢出漏洞

什么是堆？

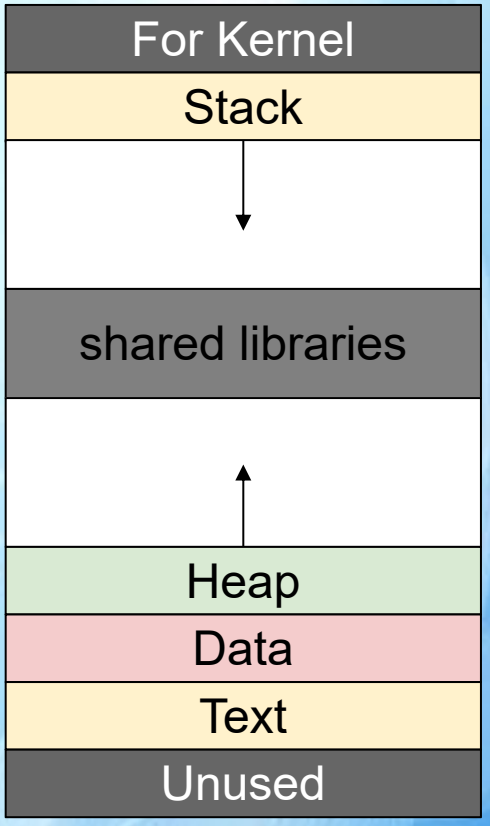
- 用户可动态申请的虚拟地址空间的一块连续的线性区域，申请大小由用户指定
  - 响应用户的申请内存请求，向操作系统申请内存，然后将其返回给用户程序
  - 用户释放的内存将适时归还给操作系统



- 堆溢出漏洞

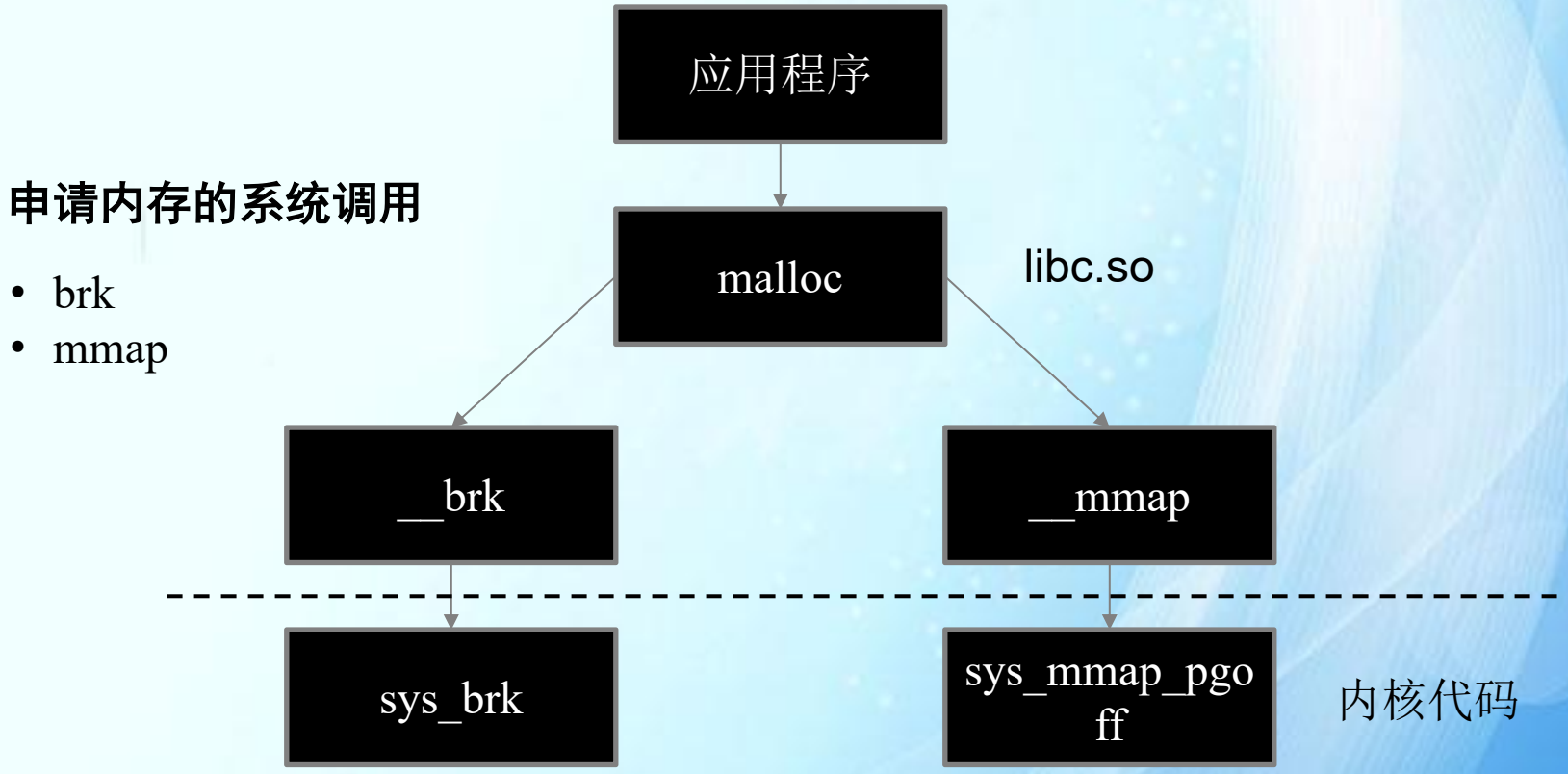
怎么管理堆？

- HeapAlloc – Windows用户态堆管理器
- dlmalloc – 最早的堆管理器（since 1980）
- ptmalloc2 – glibc
- jemalloc – FreeBSD and Firefox
- tcmalloc – Google
- libumem – Solaris



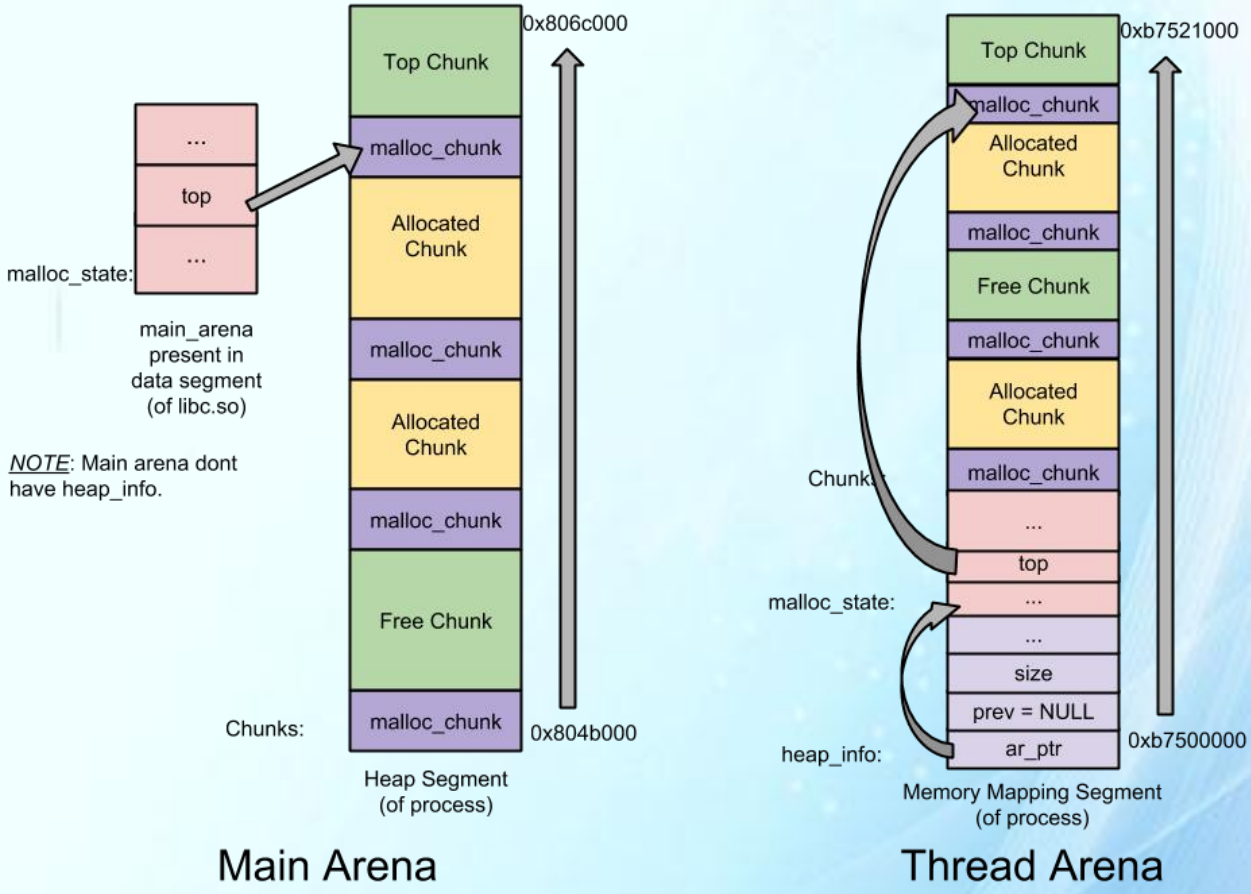
■Linux堆管理机制

- 默认采用ptmalloc堆（由libc.so.6库实现），封装了内存分配系统调用，提供动态内存分配接口，同时高效地管理由系统调用申请来的内存



■Linux堆管理机制

- Ptmalloc堆从操作系统申请来的内存统一存放在内存池arena中



■Linux堆管理机制

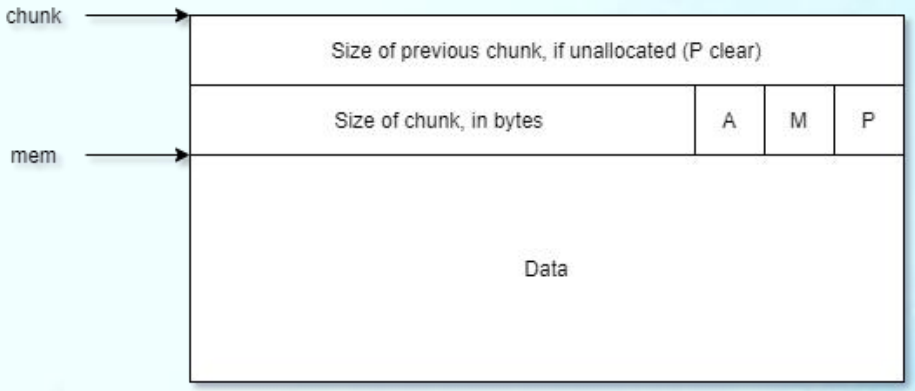
- Ptmalloc堆从操作系统申请来的内存统一存放在内存池arena中

```
struct malloc_state {
    /* Serialize access.  */
    __libc_lock_define(, mutex);
    /* Flags (formerly in max_fast).  */
    int flags;
    /* Fastbins */
    mfastbinptr fastbinsY[ NFASTBINS ];
    /* Base of the topmost chunk -- not otherwise kept in a bin */
    mchunkptr top;
    /* The remainder from the most recent split of a small request */
    mchunkptr last_remainder;
    /* Normal bins packed as described above */
    mchunkptr bins[ NBINS * 2 - 2 ];
    /* Bitmap of bins, help to speed up the process of determinating if a given bin is definitely empty.*/
    unsigned int binmap[ BINMAPSIZE ];
    /* Linked list, points to the next arena */
    struct malloc_state *next;
    /* Linked list for free arenas.  Access to this field is serialized
       by free_list_lock in arena.c.  */
    struct malloc_state *next_free;
    /* Number of threads attached to this arena.  0 if the arena is on
       the free list.  Access to this field is serialized by
       free_list_lock in arena.c.  */
    INTERNAL_SIZE_T attached_threads;
    /* Memory allocated from the system in this arena.  */
    INTERNAL_SIZE_T system_mem;
    INTERNAL_SIZE_T max_system_mem;
};
```

■Linux堆管理机制

- Ptmalloc 堆块：用户申请内存的单位，也是堆管理器管理内存的基本单位

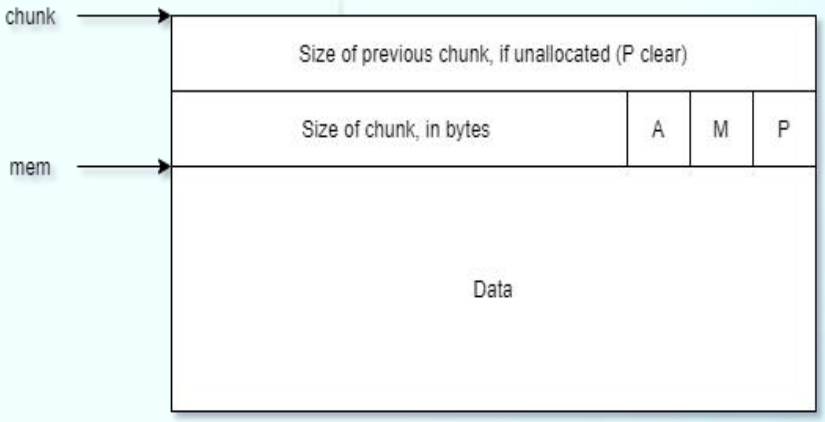
  malloc()返回的指针指向一个chunk的数据区域



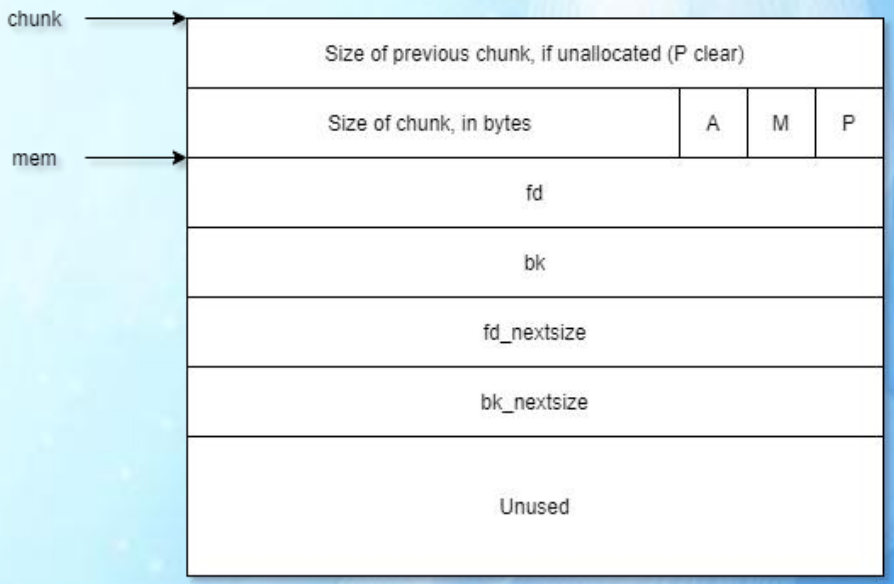

```
struct malloc_chunk {
  INTERNAL_SIZE_T      prev_size;  /* Size of previous chunk (if free).  */
  INTERNAL_SIZE_T      size;       /* Size in bytes, including overhead. */

  struct malloc_chunk* fd;         /* double links -- used only if free. */
  struct malloc_chunk* bk;

  /* Only used for large blocks: pointer to next larger size.  */
  struct malloc_chunk* fd_nextsize; /* double links -- used only if free. */
  struct malloc_chunk* bk_nextsize;
};
```

■Linux堆管理机制

- 堆块Chunk分类
  - 按状态：malloc、free
  - 按大小：fast、small、large、tcache
  - 按特定功能：top chunk、last remainder chunk



已分配堆块



释放堆块

■Linux堆管理机制

- 堆布局实例

```
char password[] = “UCAS_20250517";
int main(int argc, char *argv[])
{
  char *buf = (char *)malloc(100);
  char *secret = (char *)malloc(100);
  strcpy(secret, password);
  printf("Password:");
  scanf("%s", buf);
  if (!strcmp(buf, secret)) {
    printf("Password OK :)\n");
  } else {
    printf("Invalid Password! %s\n", buf);
  }
  return 0;
}
```

```
pwndbg> heap
Allocated chunk | PREV_INUSE
Addr: 0x55d961d0f000
Size: 0x290 (with flag bits: 0x291)

Allocated chunk | PREV_INUSE
Addr: 0x55d961d0f290
Size: 0x70 (with flag bits: 0x71)

Allocated chunk | PREV_INUSE
Addr: 0x55d961d0f300
Size: 0x70 (with flag bits: 0x71)

Top chunk | PREV_INUSE
Addr: 0x55d961d0f370
Size: 0x20c90 (with flag bits: 0x20c91)
```

■Linux堆管理机制

- 已释放（空闲）堆块的管理
  - 管理 arena 中空闲 chunk 的结构，以数组的形式存在，数组元素为相应大小的 chunk 链表的链表头，存在于 arena 的 malloc_state 中

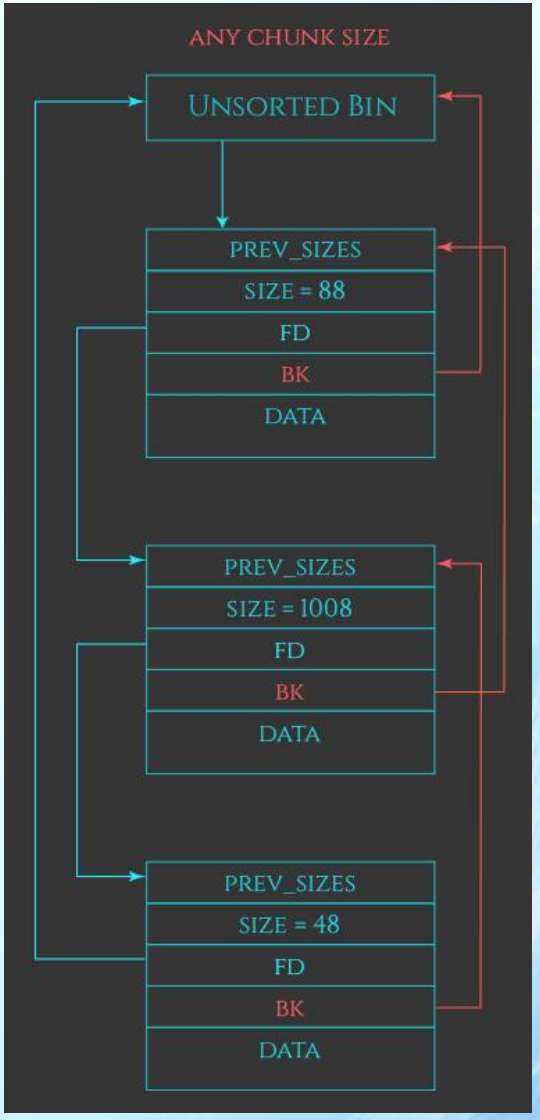
- unsorted bin
  - 管理刚刚释放还为分类的 chunk
- fast bins ： 单向列表
  - 管理 16、24、32、40、48、56、64 Bytes 的 free chunks
- small bins ： 62 个循环双向链表
  - 管理 16、24、32、40、 …… 、504 Bytes 的 free
- large bins ： 63 个循环双向链表
  - 管理大于 504 Bytes 的 free chunks
- tcache：64个单链表（每个链表通常不超过7个堆块）

■Linux堆管理机制

- Unsorted bin

Pwndbg命令

- unsorted
- p *(mchunkptr)0x55555

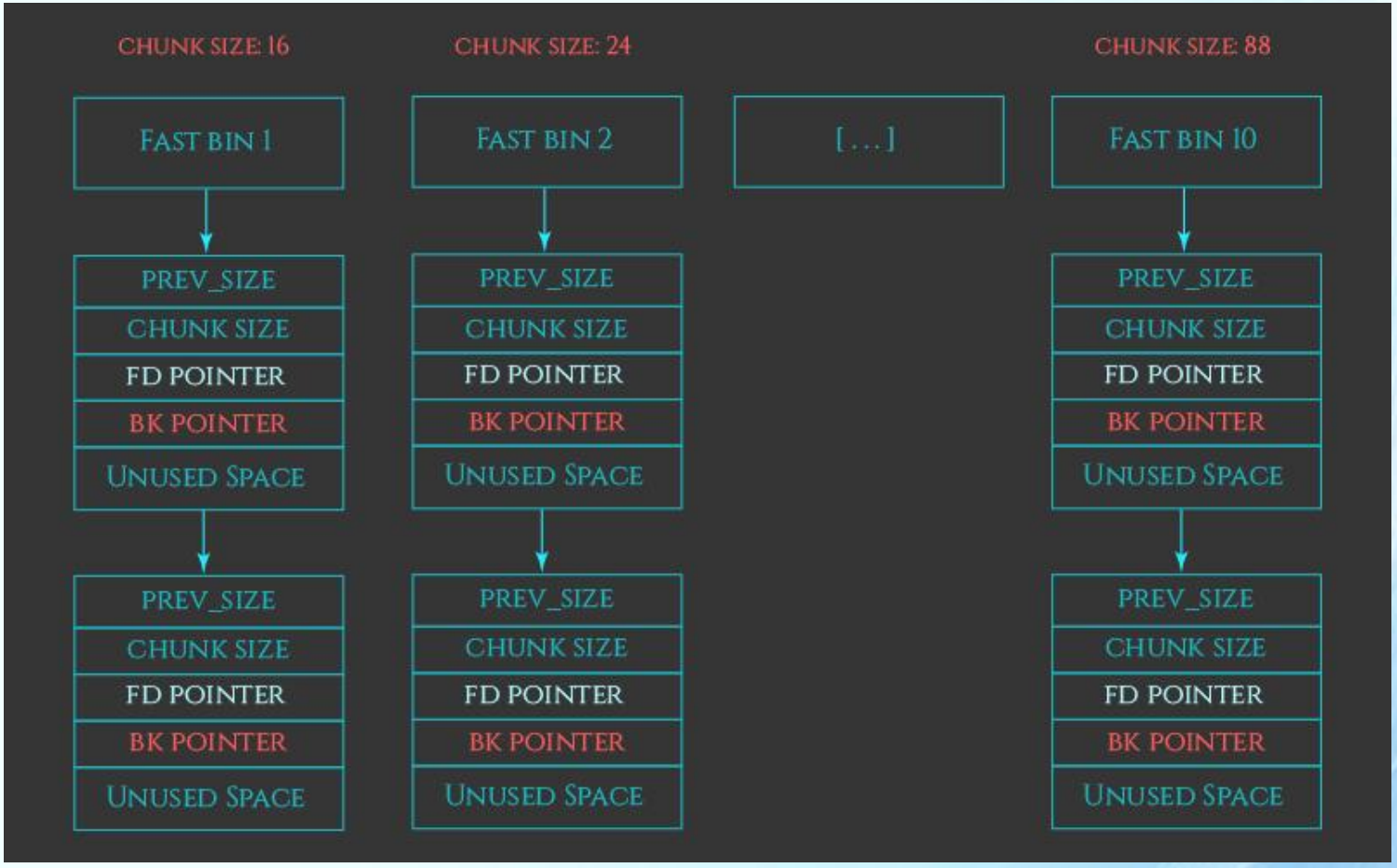


■Linux堆管理机制

- fast bin

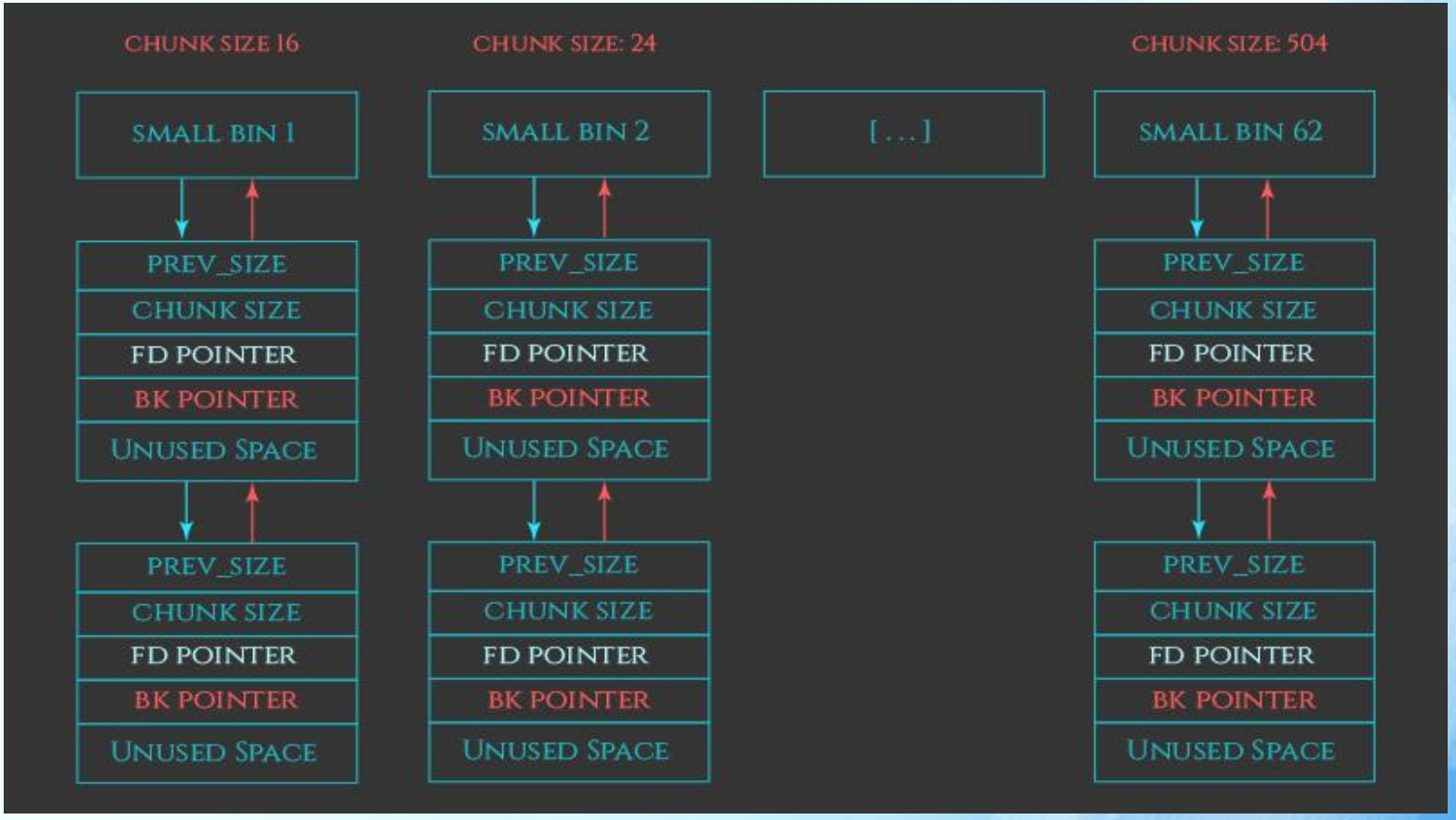
Pwndbg命令

- fastbins
- p *(mchunkptr)0x55555



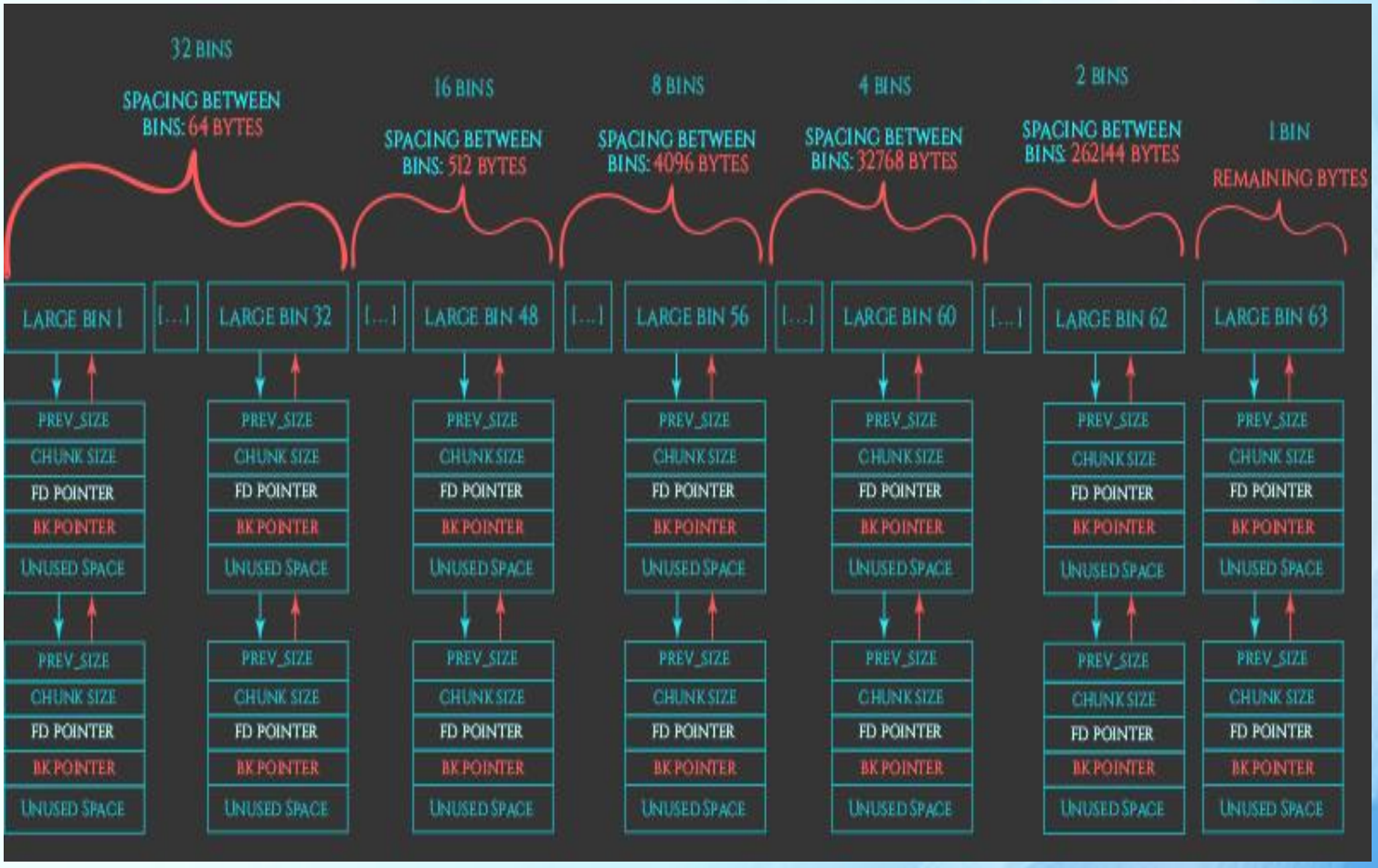
■Linux堆管理机制

- small bin



■Linux堆管理机制

- large bin



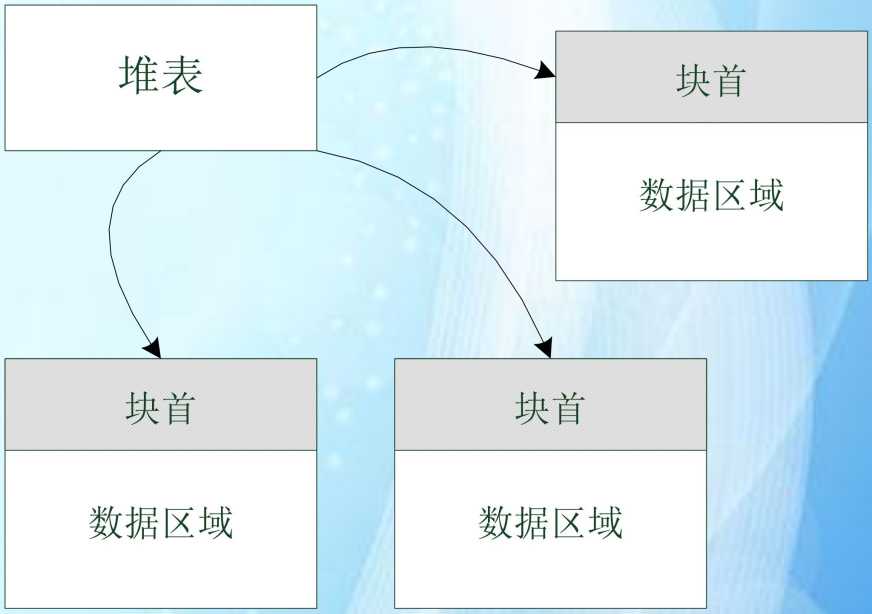
■Linux堆管理机制

- 堆分配和释放实例

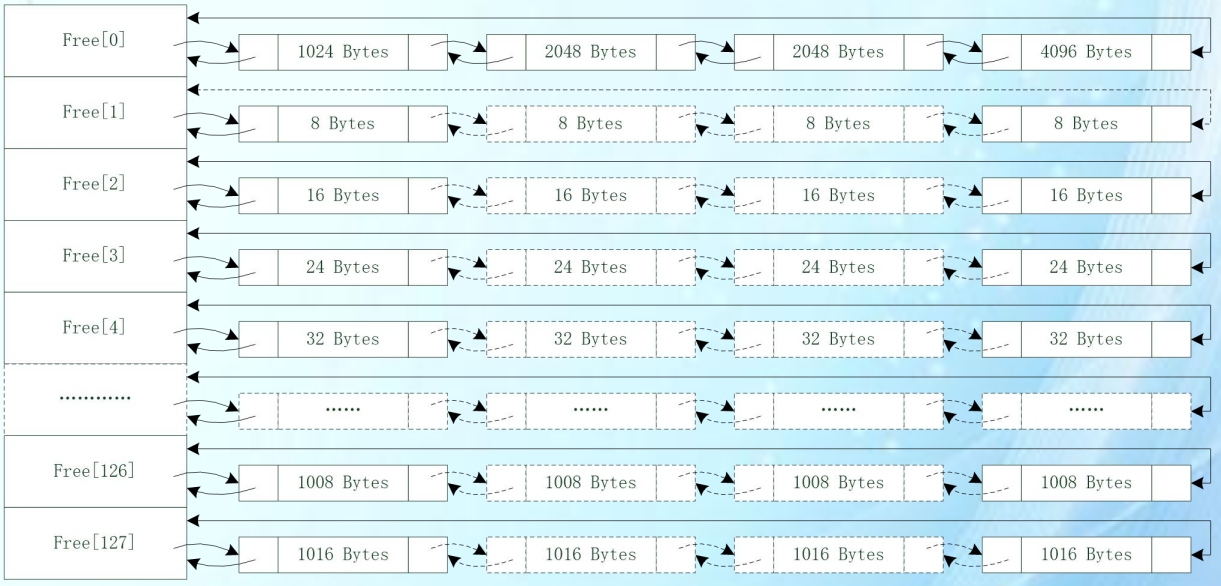
```
int main() {
  void *fb_0 = malloc(16);
  void *fb_1 = malloc(32);
  void *fb_2 = malloc(16);
  void *fb_3 = malloc(32);
  prompt("Stage 1\n");

  free(fb_1);
  free(fb_3);
  prompt("Stage 2\n");

  free(fb_0);
  free(fb_2);
  void *zb_0 = malloc(32);
  prompt("Stage 3\n");
  void *nb_0 = malloc(100);
  void *nb_1 = malloc(120);
  void *nb_2 = malloc(140);
  void *nb_3 = malloc(160);
  void *nb_4 = malloc(180);
  prompt("Stage 4\n");

  free(nb_1);
  free(nb_3);
  prompt("Stage 5\n");

  void *nb_5 = malloc(240);
  prompt("Stage 6\n");

  free(nb_2);
  prompt("Stage 7\n");

  return 0;
}
```

1.3 典型软件漏洞类型

- Windows 堆管理机制
  - HeapAlloc、HeapFree
  - 堆块结构
    - 堆块结构：块首和块身
    - 块首：堆块头部的几个字节
      - 标识这个堆块自身的信息
        - 大小、状态（空闲/占用）
    - 块身：数据区
      - 紧跟在块首后面，最终分配给用户使用的数据区
  - 堆管理器返回的指针指向块身的起始位置
    - 程序在操作堆空间时感觉不到块首存在



- Windows 堆管理机制
  - 堆表：堆表一般位于堆区的起始位置
    - 用于索引堆区中所有堆块的重要信息，包括位置、大小、状态等
  - 堆表的数据结构决定了整个堆区的组织方式
    - 设计目标是：快速检索空闲块，提升堆分配释放效率
  - 堆表采用平衡二叉树等高级数据结构优化查找效率

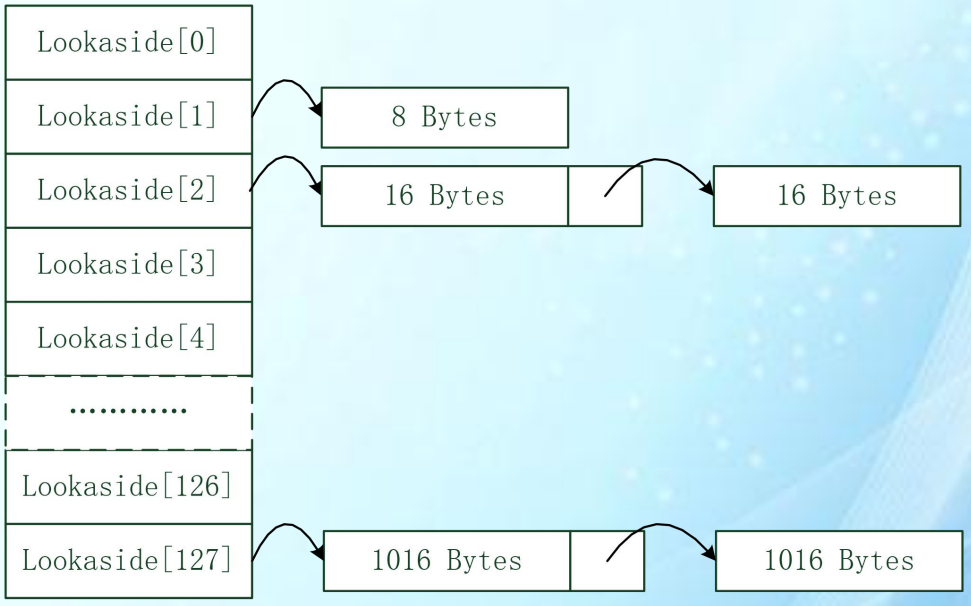


■Windows 堆管理机制

- Windows系统默认堆机制两种主要堆表
  - 空闲双向链表Freelist（简称空表）
  - 快速单向链表Lookaside（简称快表）
- 空表
  - 包含128项的数组，每个成员是一个空闲堆块链表
  - 每个空闲堆块链表中堆块大小固定且相同
    - 索引项 * 8字节
  - 空表索引free[0]所标识的空表
    - 这条双向链表链入了所有大于等于1024字节的堆块（小于512KB），堆块按照各自大小在零号空表中升序地依次排列下去
  - 每个空闲堆块的块首中包含一对指针：将空闲堆块链入双向链表

Windows 堆管理机制

- 快表是Windows用来加速堆块分配而采用的一种堆表
- 快表也有128条，组织结构与空表类似，但堆块按照单链表组织
- 快表总是被初始化为空，而且每条快表最多只有4个结点，故很快就会被填满



1.3 典型软件漏洞类型

- 堆溢出漏洞
  - 定义
    - 没有严格检验数据长度，导致写入范围大于堆块的大小，覆盖了堆块数据，引发程序崩溃或控制流劫持
  - 产生原因
    - 堆管理机制
    - 未正确校验的堆块访问
  - 主要类型
    - 堆块的管理结构被覆盖
      - 由于操作系统的堆释放函数没有对指针的有效性进行校验，导致能够对内存中的任意地址写入任意内容
    - 堆块的数据内容被覆盖
      - 后续堆中存放的函数指针等被前一个堆的内容覆盖，导致后续调用产生时转入攻击者预设的代码

1.3 典型软件漏洞类型

- 堆溢出漏洞：覆盖堆管理结构
  - 堆溢出造成堆管理结构被覆盖，会影响到堆管理机制，可能被攻击者利用

典型漏洞利用方法 DWORD Shoot

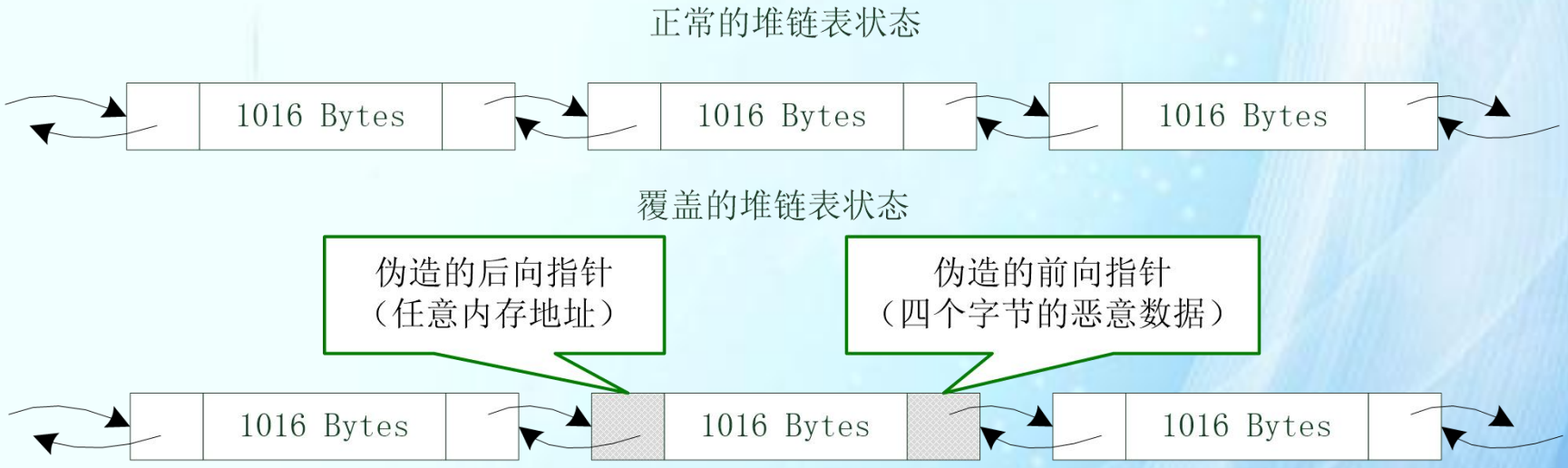
① 假设Heap1和Heap2是两个相邻的堆块，程序写入数据到堆块Heap1，由于堆溢出发生，导致覆盖了Heap2的堆块结构

② 当堆管理器在释放Heap2时，在将堆块回收到链表时产生异常，回收链表的伪代码为

```
/* 释放堆块-回收到链表 Take a chunk off a bin list */
Int remove(ListNode *node)
{
        node -> Blink -> Flink =  node -> Flink;
        Node -> Flink -> Blink = node -> Blink;
}
```

DWORD Shoot

- 堆溢出漏洞
  - 覆盖堆管理结构
    - 攻击者：改写块首中的前向指针(Flink)和后向指针(Blink)
    - 漏洞触发：在堆块释放时，可向任意内存地址（后向指针）写入一个四字节DWORD数据（前向指针）



1.3 典型软件漏洞类型

- 堆溢出漏洞
  - 覆盖堆管理结构
    - DWORD Shoot 任意内存写能够实现
      - 改写内存参数、改写重要变量，改写字符串的长度、格式串等，让改写后再产生别的溢出等
      - 改写函数指针性质的调用入口，如果能够定位到ShellCode，基本上就可以有一种堆溢出的利用办法
  - 覆盖后续堆内容
    - 产生原因
      - 没有检查输入数据的长度，导致在申请的堆空间被输入数据填满后，后续的内存空间也被覆盖
    - 利用
      - 被错误覆盖的后续数据区域，可能原本用来存放敏感数据，如函数指针、密钥等

1.3 典型软件漏洞类型

- 释放后重用漏洞（Use After Free，UAF）
  - 形成机制
    - 释放后重用漏洞的机理是已经被释放的对象或内存，被另一次申请所获取，并且内容在被后续指令修改后，被原有的对象引用
  - 形成条件
    - 操作系统基于内存缓冲池（快表/空表）的堆管理机制
    - 程序员释放堆块后，没有将所有指向该堆块的指针清除

- 释放：内存块(小内存块)被程序释放后，堆管理器将其放入缓冲池，供下次申请内存使用，此时程序员并未清除指向该堆块的指针
- 再分配：程序后续操作申请相同大小的内存时，很有可能获取到的是程序前面释放的内存，申请内存中可以填写任意内容（比如覆盖某个函数指针）
- 重用：基于未清除的指针实现恶意操作（调用被覆盖的函数指针）

1.3 典型软件漏洞类型

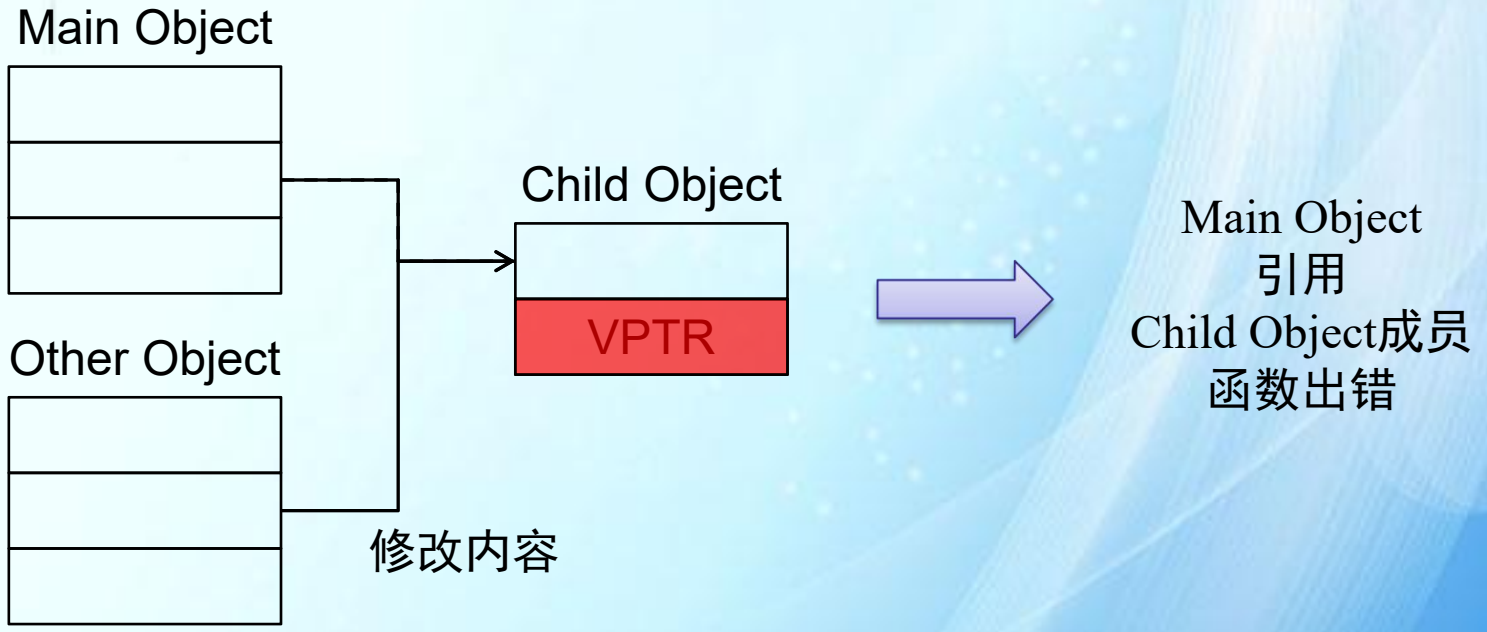
- 释放后重用漏洞
  - 示例：

① 假设存在一个对象MainObject，该对象向操作系统申请了另一个对象ChildObject，记录了ChildObject的指针

② 如果对象ChildObject被释放，同时MainObject中指向ChildObject的指针没有清除

③ 当后续函数申请内存时，可能分配到对象ChildObject原来占用的堆块

④ 此时对象MainObject引用ChildObject指针，触发异常，进而形成UAF漏洞



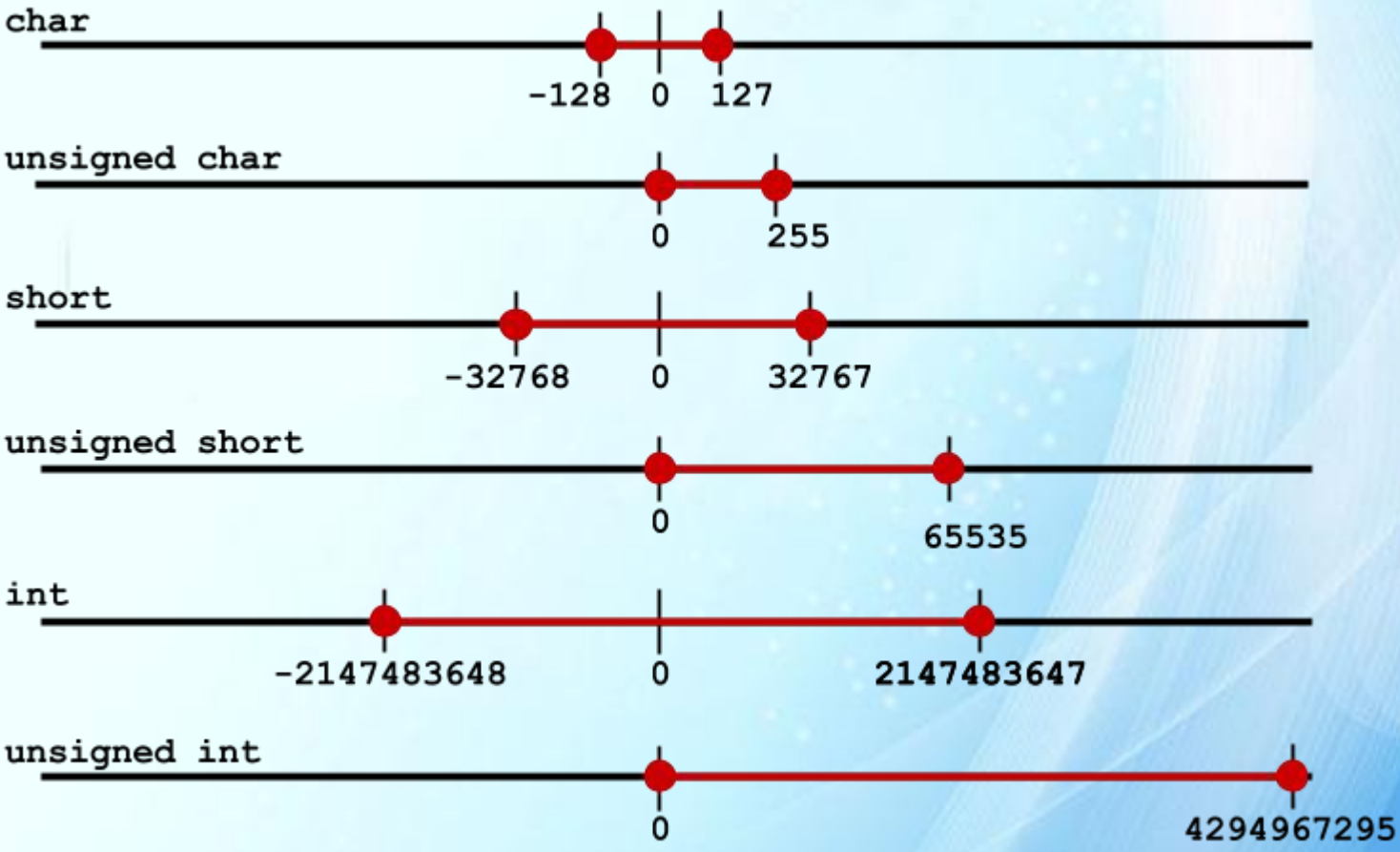
1.3 典型软件漏洞类型

- 整数溢出漏洞
  - 形成机理
    - 形成原因：程序对于整型数据类型的处理错误
      - 整数在计算机的表示有其取值范围，超过范围的整数会被截断，如果这个整数是用于记录某块内存块的长度，则可能由于截断长度和预期长度的不匹配，导致整数溢出
    - 漏洞危害：可转换为缓冲区溢出漏洞，覆盖堆栈数据
  - 整数溢出分类
    - 算术溢出（arithmetic overflow）
      - 由于数据上溢或下溢引发的漏洞
    - 宽度溢出（widthness overflow）
      - 由于数据截断引发的漏洞

1.3 典型软件漏洞类型

- 整数溢出漏洞

C语言变量类型的数值范围



1.3 典型软件漏洞类型

- 整数溢出漏洞
  - 算术溢出

成因：C语言的有符号整数运算与无符号整数运算处理

- 如果都为无符号整型或有符号整型，则按照其现有类型进行计算
- 如果一个为有符号整型，一个为无符号整型，首先将有符号整型转换为无符号整型，然后进行计算

溢出发生时 →

■ 编译器差异导致结果不同

C语言标准的定义中，对有符号整型计算溢出的处理没有详细的定义，不同编译器的实现有差异，往往会给出不同的结果

■ 溢出的数值与预先设想有差异导致安全问题

整数溢出导致栈、堆溢出

1.3 典型软件漏洞类型

- 整数溢出漏洞
  - 算术溢出（x86 32位程序为例）

```
int myfunction(int *array, int len) {
    int *myarray, i;
    // 这里如果len是一个很大的正数，那么乘法的结果发生溢出，
    // 所以分配的内存长度会比较小(len * sizeof(int) % 0x10000 0000)
    myarray = malloc(len * sizeof(int));
    if(myarray == NULL){
        return -1;
    }
    // 溢出发生了！
    for(i = 0; i < len; i++){
        myarray[i] = array[i];
    }
    return myarray;
}
```

1.3 典型软件漏洞类型

- 整数溢出漏洞
  - 宽度溢出（x86 32位程序为例）
    - 宽度溢出通常的产生原因是由于长度检查时使用了截断的变量，而在操作缓冲区时使用了未截断的变量

```
int main(int argc, char *argv[])
{
    unsigned short s;
    int i;
    char buf[80];
    i = atoi(argv[1]); // 如果argv[1]超过short表示范围（比如65536+10），注意i的使用
    s = i; // 那么在这里s被截断，得到了一个看上去合理的正数 （比如10）
    if(s >= 80){ // 这里的检查，显然没有阻止这种情况
        printf("Oh no you don't!");
        return -1;
    }
    memcpy(buf, argv[2], i); // 溢出发生，被拷贝的长度可以非常大（65536+10）
    return 0;
}
```

1.3 典型软件漏洞类型

- 逻辑漏洞
  - XSS漏洞（Cross Site Scripting，跨站脚本攻击)
    - 定义：
      - 该漏洞会导致恶意web用户将代码植入到提供给其它用户使用的页面中，是一种经常出现在web应用中的计算机安全漏洞
    - 非持久性XSS攻击
      - 非持久型XSS攻击要求用户访问一个被攻击者篡改后的链接，用户访问该链接时，被植入的攻击脚本被用户游览器执行，从而达到攻击目的
      - 非持久型XSS攻击是一次性的，也被成为反射型XSS攻击，仅对当次的页面访问产生影响
    - 持久性XSS攻击
      - 当攻击者提交到web应用程序里的数据会永久性的存储到服务器的时候会产生这类漏洞，(比如数据库，文件系统，其他位置)，如果没有经过HTML编码过滤，那么每一个访问该页面的用户都会被攻击，典型的例子就是在线留言板，它允许用户提交数据

1.3 典型软件漏洞类型

TCA

- XSS漏洞典型案例

该页面从URI获取 'name' 参数，并在页面显示

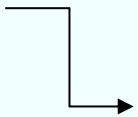
```
<?php
$name = $_GET['name'];
echo "Welcome $name<br>";
?>
```

攻击者可将自定义脚本代码嵌入“name”参数，例如，构造如下Web URL链接：

```
index.php?name=<script>alert('attacked')</script>
```

当某个用户点击该链接时，其中脚本内容会被执行，带'attacked'的告警提示框弹出

更进一步，如果攻击者构造如下URL链接，点击该链接用户将可能会跳转至攻击者提供的其他网址



```
index.php?name=<script>window.onload = function() {
var link=document.getElementsByTagName("a");
link[0].href="http://attacker-site.com/";}</script>
```

1.3 典型软件漏洞类型

- 逻辑漏洞
  - SQL注入漏洞
    - 形成的主要原因：程序员没有对用户输入数据的合法性进行判断，使用户可非法注入并执行指定的数据库查询代码

例如，某个网站的登录验证的SQL查询代码为：

```
strSQL = "SELECT * FROM users WHERE (name = '" + userName + "') and (pw = '"+ passWord +"');"
```

攻击者通过Web页面填入

```
userName ：  "1' OR '1'='1"
passWord ：  "1' OR '1'='1"
```

则登录验证SQL字符串为：

```
strSQL = "SELECT * FROM users WHERE (name = '1' OR '1'='1') and (pw = '1' OR '1'='1');"
```

上述SQL脚本执行后，验证通过，实现无账号密码，也能够登录网站

1.4 软件漏洞防护机制

- 软件漏洞利用缓解机制
  - 目标
    - 针对软件漏洞触发和利用特点制定的防御机制
      - 例如，针对控制流劫持类漏洞攻击的溢出过程、控制流转移过程、攻击代码执行过程等
  - 典型保护机制
    - 针对于栈溢出的Stack Cookie
    - 针对于堆溢出的SAFE Unlink 堆释放操作
    - 针对于控制流劫持攻击的数据区执行保护（DEP）
    - 地址随机化（ASLR）
    - 控制流完整性：CFG、XFG

1.4 软件漏洞防护机制

- 栈保护（Stack Cookie）
  - 进入函数后立即往栈中压入一个随机数Cookie
  - 当栈溢出发生时，Cookie会被溢出数据覆盖
  - 函数返回前会将预先保存的值和从栈中取出的Cookie值进行比较
    - 如果Cookie值与预先保存的相同，则正常执行
    - 如果Cookie和事先保存的数据不相同，则系统认为发生栈溢出

1.4 软件漏洞防护机制

- 栈保护（Stack Cookie）
  - Cookie获得方法
    - 编译阶段：创建Security Cookie全局变量
      - 编译器将其写入到.data段里，即在PE文件中保存了Cookie的初始值
    - 运行阶段：程序加载后生成随机Cookie
      - Windows装载器加载程序：创建进程，为栈分配内存，把 EIP设置为PE里的OEP字段等
    - Cookie计算的代码是公开的
      - 通过算法保证Cookie值随机，攻击者很难在编写ShellCode时准确猜出Cookie值

```
helloworld!wWinMainCRTStartup [f:\dd\vctools\crt_bld\self_x86\crt\src\crtexe.c @ 361]:
  361 00b22090 8bff            mov     edi,edi
  361 00b22092 55              push    ebp
  361 00b22093 8bec            mov     ebp,esp
  368 00b22095 e8c5efffff      call    helloworld!ILT+90(___security_init_cookie) (00b2105f)

  370 00b2209a e811000000 call    helloworld!__tmainCRTStartup (00b220b0)
  371 00b2209f 5d              pop     ebp
  371 00b220a0 c3              ret
helloworld!ILT+540(_wWinMainCRTStartup):
00b21221 e96a0e0000          jmp     helloworld!wWinMainCRTStartup (00b22090)
```

Cookie初始化

程序入口

1.4 软件漏洞防护机制

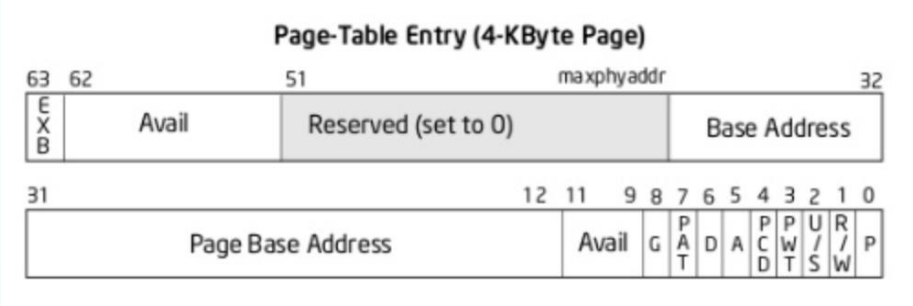
- 安全堆释放操作（SAFE-Unlink）
  - 删除堆块（ free ）时进行运行时检查机制
    - 前块的前驱指针必须指向当前堆块
    - 后块的后继指针必须指向当前堆块

```
/* Take a chunk off a bin list */
void unlink(malloc_chunk *P, malloc_chunk *BK, malloc_chunk *FD)
{
    FD = P->fd;
    BK = P->bk;
    if (__builtin_expect (FD->bk != P || BK->fd != P, 0)) {
        malloc_printerr(check_action
                        ,"corrupted double-linked list",P);
    } else {
        FD->bk = BK;
        BK->fd = FD;
    }
}
```

判断指针是否合法

1.4 软件漏洞防护机制

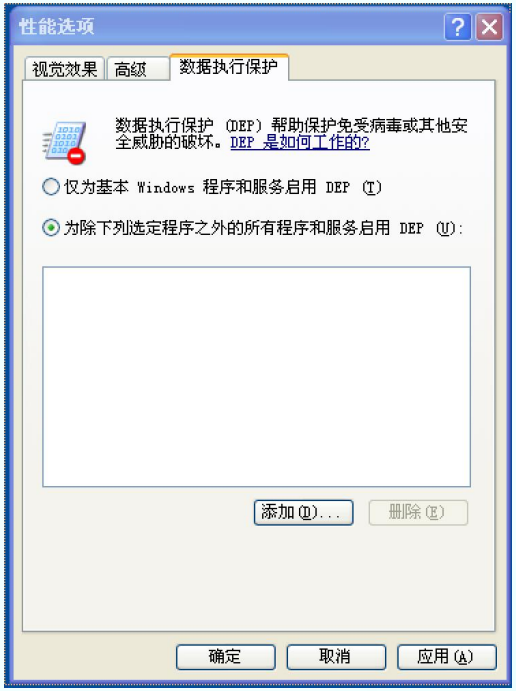
- 数据区执行保护(DEP)
  - 针对的主要目标是缓冲区溢出
    - 缓冲区溢出攻击需要在程序存放数据的区域写入可执行的攻击代码，然后劫持控制流转移到攻击代码
  - 实现原理
    - 页表项中添加NX位（1比特）
      - 标记只包含数据的内存位置为非可执行状态(Not Executable, NX)
    - 当应用程序试图从标记为NX的内存位置执行代码时，系统将报出一个数据访问异常
    - Windows的异常处理将阻止应用程序执行数据区域，从而达到截断非法控制流转移，保护系统防止溢出的目的



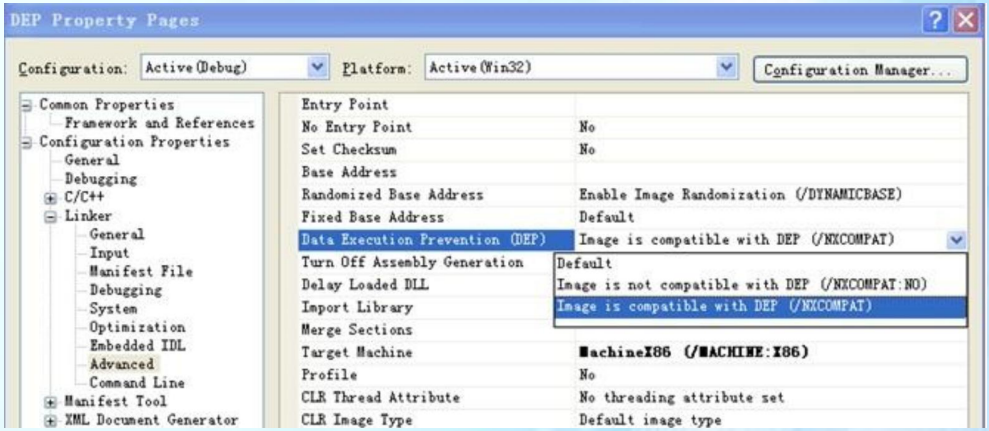
1.4 软件漏洞防护机制

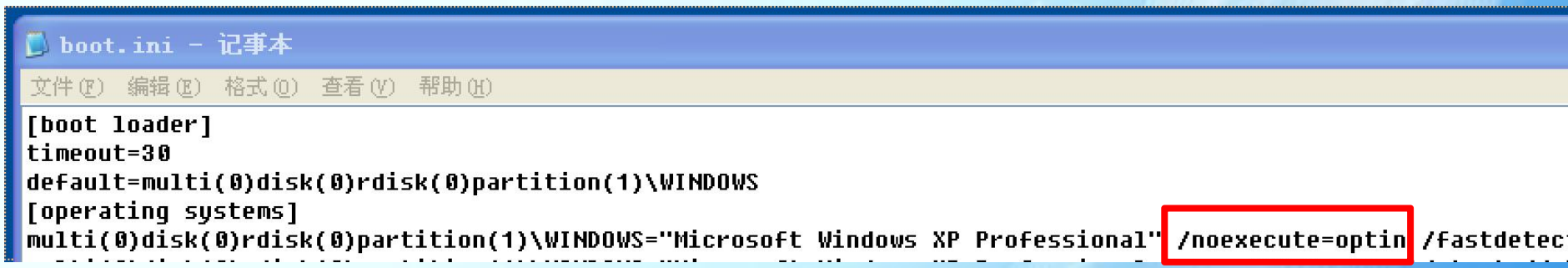
- 数据区执行保护(DEP)

DEP设置界面



软件编译时兼容DEP





1.4 软件漏洞防护机制

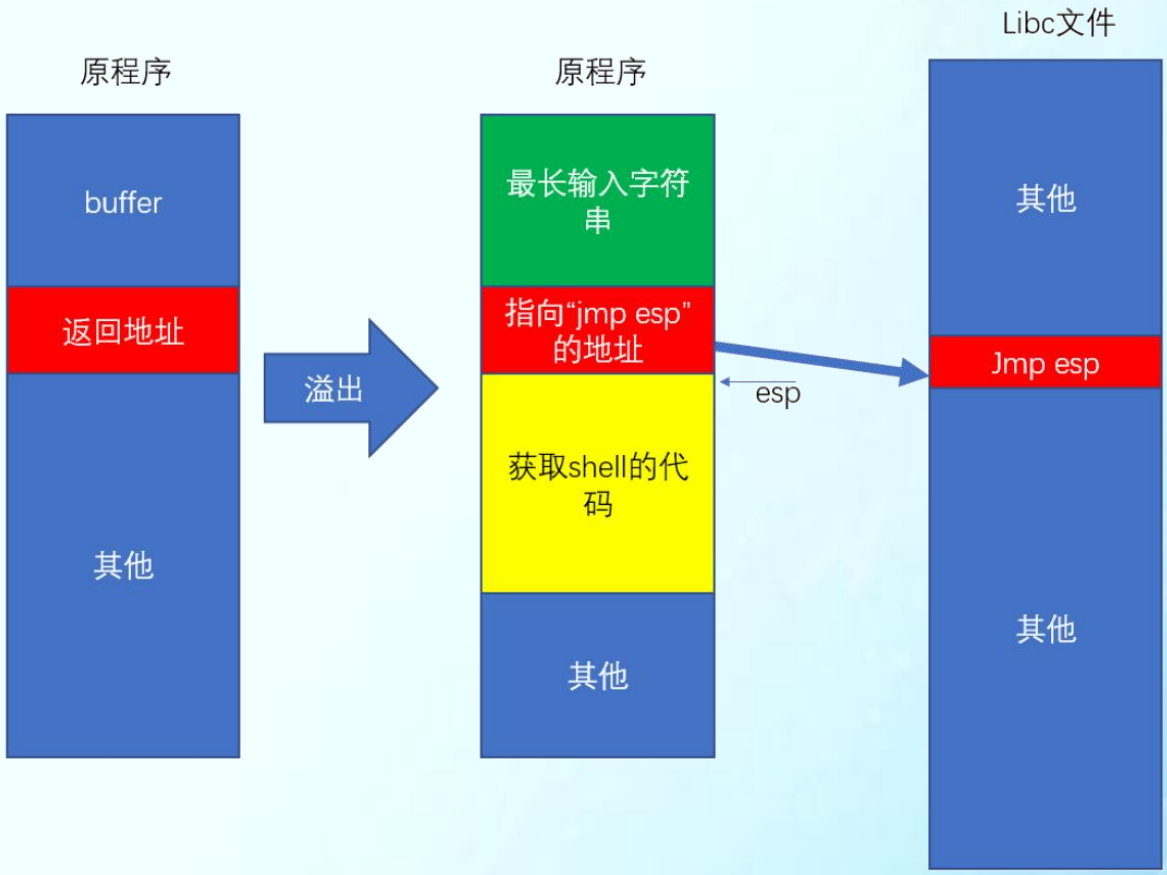
- 数据区执行保护(DEP)
  - DEP通常需要CPU 硬件支持
    - Intel CPU: Execute Disable Bit (XD)
    - AMD: no-execute page-protection (NX)
  - 其他类DEP机制（SafeSEH）
  - 局限性：
    - 第三方软件的兼容性问题
    - 操作系统提供了在用户态关闭该机制的接口

1.4 软件漏洞防护机制

- 地址随机化ASLR(Address space layout randomization)
  - 目标
    - 防止因内存破坏导致的漏洞利用攻击
  - 实现原理
    - 通过对堆、栈、共享库映射等线性区布局的随机化，通过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
  - ASLR可以有效的降低缓冲区溢出攻击的成功率
    - Linux、FreeBSD、Windows等主流操作系统都已采用了该技术
  - 主要方式
    - PE文件加载地址随机化
    - 堆栈随机化
    - PEB、TEB 随机化

1.4 软件漏洞防护机制

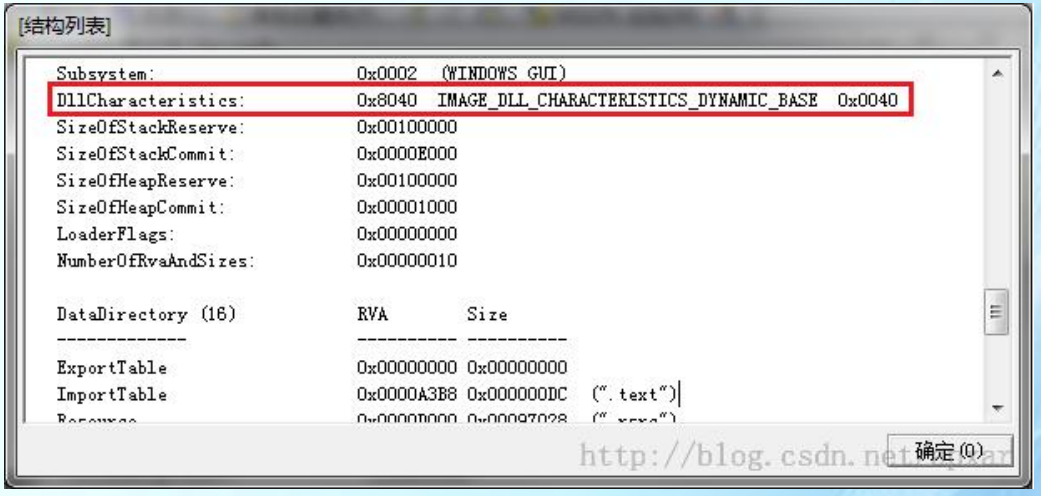
- 地址随机化ASLR(Address space layout randomization)
  - ASLR能够防御的漏洞利用方式 – JMP ESP




```
JMP ESP Address: 7408c38e
JMP ESP Address: 740f9ce3
JMP ESP Address: 740f9da3
JMP ESP Address: 740f9f63
JMP ESP Address: 7411602b
JMP ESP Address: 7411602f
JMP ESP Address: 74116033
JMP ESP Address: 74116037
JMP ESP Address: 7411603b
JMP ESP Address: 7411603f
```

1.4 软件漏洞防护机制

- 地址随机化ASLR
  - PE文件加载地址随机化
    - 对程序映像虚拟地址进行随机处理，该地址在系统启动时初始化
      - 加载地址随机化可以通过注册表来设置，加载地址随机化使得通过的跳板指令无效
      - 映像随机化只对加载基址的前两个字节做了随机处理，各模块入口点的低位2字节不变
    - 需要操作系统和程序的双重支持，在PE头设置属性
      - IMAGE_DLL_CHARACTERISTICS_DYNAMIC_BASE标识表明其支持ASLR



1.4 软件漏洞防护机制

- 地址随机化ASLR
  - 堆栈随机化
    - 堆栈被随机化后，基址在每次加载程序时确定
    - 在缓冲区溢出攻击中，通常使用jmp esp等跳转指令进行控制流转移，使用堆栈的基地址寻址，因而堆栈随机化对溢出利用的影响有限
  - PEB、TEB 随机化（从Windows XP SP2开始引入）
    - PEB、TEB存储了进程和线程的大量关键数据
    - 固定基址容易被攻击者利用
      - PEB:0x7FFDF000，TEB:0x7FFDE000

1.4 软件漏洞防护机制

- 地址随机化ASLR
  - 地址随机化的局限性
    - ASLR范围有限，可以通过信息泄漏绕过
      - 攻击者可通过程序进程表结构来获得特定DLL的基址
    - 其他不支持 ASLR 的程序或模块会影响防护能力
      - 如果受保护进程依赖的模块没有开启ASLR机制，则该模块可作为攻击跳板

1.4 软件漏洞防护机制

- Windows Control Flow Guard (CFG)

部署在最新的Windows 8.1，Windows 10上

- 粗粒度的CFI：所有的有效跳转地址为一个全局的集合（Bitmap），即不精确的为每一个间接跳转指定一个有效跳转地址
- 前向CFI：只考略call，jump的直接跳转和间接跳转，没有考虑ret的情况

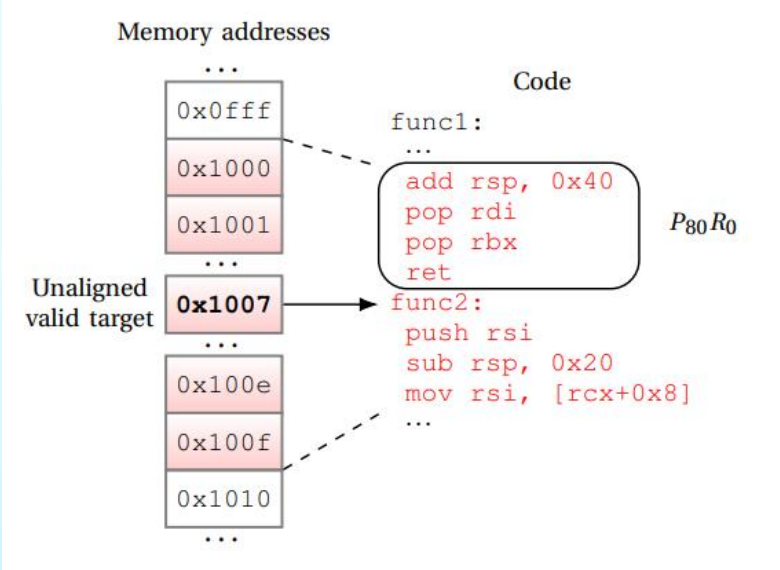
有效跳转目标集合定义(Bitmap)

00：该地址范围没有有效的跳转地址

01：地址范围包含导出抑制表目标

10：该范围的第一个地址有效

11：地址范围的所有地址均有效



1.4 软件漏洞防护机制

- Xtended Flow Guard (XFG)

更细粒度的CFG：通过类型签名检查限制间接调用或跳转

① 编译时：可以作为间接调用/跳转目标的函数生成一个基于类型签名的哈希

② 动态运行：间接调用点上，进行哈希检查，仅允许具有预期签名哈希的函数

CFG instrumentation: Call Site

```
mov  rax, [rsi+0x98]                ; load target address
call [__guard_dispatch_icall_fptr]
```

Target

```
.align 0x10
function:
     push rbp
     push rbx
     push rsi
     ...
```

xFG instrumentation : Call Site

```
mov  rax, [rsi+0x98]                  ; load target address
mov  r10, 0xdeadbeefdeadbeef         ; load function tag
call [__guard_dispatch_icall_fptr_xfg] ; will check tag
```

Target

```
.align 0x10
dq 0xcccccccccccccccc ; just alignment
dq 0xdeadbeefdeadbeef ; function tag
function:
     push rbp
     push rbx
     push rsi
```

谢谢