Week 10

Nite

2025-03-19T12:18:00+11:00

阅读 PDF

软件与网络安全 (Software & Network Security)

安全系统设计 (Design of a Secure System)

设计安全系统通常采用自顶向下的方法：

1. **威胁建模 (Threat Model)**：分析可能存在的风险。
2. **安全策略 (Security Policy)**：明确安全机制旨在实现的目标。
3. **安全机制 (Security Mechanisms)**：如何实施这些安全策略。

安全系统的设计遵循**环形设计 (Ring Design)** 原则：

- 每个对象都有关联的安全属性。
- 每个主体都有安全许可。
- 目标是限制环之间的交互。**内环具有更高的安全性，外环的安全性较低。**

可信操作系统示例 (Example: Trusted Operating System)

一个可信操作系统 (Trusted Operating System) 的典型分层结构：

- **硬件 (Hardware)** (最内层)
- **安全内核 (Security Kernel)**
- **操作系统 (Operating System)**
- **用户应用程序 (User Applications)** (最外层)

其中，硬件、安全内核和操作系统部分构成了**可信计算基 (Trusted Computing Base, TCB)**。**TCB 的失败会导致整个安全系统的失败。**

Bell-LaPadula 模型

Bell-LaPadula 模型是在军事和情报数据安全背景下发展起来的，也被称为“多级安全 (Multilevel Security, MLS)”系统。该模型定义了两个主要的安全属性：

1. **简单安全属性 (Simple Security Property) (NRU - No Read Up)**：
    - 任何进程都不能读取更高级别的数据。
    - 例如：拥有“秘密 (Secret)”权限的代理爱丽丝不能读取“绝密 (Top-Secret)”文件。

2. **星号属性 (*-Property) (NWD - No Write Down)**：
    - 任何进程都不能向更低级别写入数据。
    - 例如：拥有“秘密 (Secret)”权限的代理爱丽丝不能写入“公开 (Public)”文件。

隐蔽信道 (Covert Channels)

Bell-LaPadula 模型旨在阻止具有不同访问权限的主体进行通信。然而，存在一个问题：**隐蔽信道 (Covert Channels)**，即安全策略执行者未检测到的通信渠道。

示例：文件锁定 (File Locking)

- 高权限主体频繁锁定和解锁一个文件。
- 低权限主体检查该文件的锁定状态。
- **通过同步计时器，这种方式可能达到 1000 比特/秒的传输速率。**

安全评估：橙皮书 (The Orange Book)

可信计算机系统评估准则 (Trusted Computer Systems Evaluation Criteria, TCSEC)，由美国国防部于 1979 年发布，俗称“橙皮书”。它建立了一套计算机系统安全性的评级体系：

- **D 级：最小保护 (Minimal protection)**。任何系统都可以获得此评级。
- **C1 级：自主安全保护 (Discretionary security protection)**。用户可以禁用安全机制。
- **C2 级：受控访问保护 (Controlled access protection)**。提供基于用户的保护。
- **B1 级：标记安全保护 (Labeled protection)**。每个对象都有标签。
- **B2 级：结构化保护 (Structured protection)**。需要进行更多的操作系统模块验证。
- **B3 级：安全域 (Security domains)**。模块化的操作系统设计，清晰的安全策略。
- **A1 级：已验证设计 (Verified design)**。经过形式化验证的系统设计。

其中，C 级为自主保护，B 级为强制保护，A 级为已验证保护。

常见的编程错误 (Common Programming Mistakes)

许多安全漏洞源于常见的编程错误：

- 糟糕的设计选择/假设 (Poor design choices / assumptions)
- **未能检查用户提供的输入 (Failing to check user supplied input)**
- **缓冲区溢出 (Buffer overflows)**
- 不正确的权限级别和隔离 (Incorrect levels and separation of privileges)
- 可预测的随机数来源 (Predictable sources of randomness)
- 糟糕的默认设置 (Poor default settings)
- 糟糕的故障模式 (Poor failure modes)
- 不充分的测试 (Insufficient testing)
- 未能妥善保护机密信息 (Failure to protect secrets properly)
- 糟糕的文档和注释 (Poor documentation and commenting)
- 对遗留代码维护不善 (Poor maintenance of legacy code)

缓冲区溢出 (Buffer Overflows)

缓冲区溢出主要是由于**糟糕的编程实践、编程语言缺乏安全特性以及缺乏适当的代码审查**造成的。

利用缓冲区溢出相对容易：

1. 在源代码中找到问题。
2. 精心构造一个漏洞利用程序 (Exploit)。
3. 存在自动漏洞利用程序生成器，可以为特定处理器和操作系统植入正确的 Shellcode。

缓冲区溢出原理解释

假设服务器包含函数 `foo`：

```
void foo(char *str) {
    char buf[128];
    strcpy(buf, str); // 将用户提供的字符串 str 复制到 buf 中
}
```

当函数被调用时，计算机内存中的栈 (Stack) 结构如下（栈顶在高地址）：

```
buf[0]
......
buf[124]
sfp         (栈帧指针 - Saved Frame Pointer)
ret addr    (返回地址 - Return Address)
str         (参数 str 的地址)
```

如果用户提供的输入指针 `*str` 指向的字符串长度超过缓冲区大小（例如，136 字节），`strcpy()` 函数会继续写入，覆盖栈上更高地址的内容。

`strcpy()` 后栈的情况：

```
buf[0]
......
buf[124]
新的sfp值 (被覆盖的 sfp)
新的返回地址 (被覆盖的 ret addr)
str
```

基本栈溢出利用 (Basic Stack Exploit)

主要问题在于 `strcpy()` 函数没有进行范围检查。假设 `*str` 的内容经过精心构造，使得 `strcpy()` 执行后栈变为：

```
攻击代码的起始部分...
...
buf[128] (可能包含部分攻击代码或填充)
buf[132] (新的返回地址，指向攻击代码)
攻击代码的其余部分...
```

新的返回地址 (New ret addr) 指向攻击代码的起始位置。攻击代码示例：`execv("/bin/sh", 0)` (执行一个 shell)。当函数 `foo` 退出时，它会使用被覆盖的新的返回地址，从而开始执行攻击代码，最终可能给予攻击者一个 shell。**在此示例中，攻击代码在栈上运行。**

利用缓冲区溢出的实例

假设上述代码来自一个 Web 服务器，`foo()` 函数被浏览器发送的 URL 调用。攻击者可以创建一个 200 字节的 URL 来在 Web 服务器上获取 shell。

一些复杂因素：

- 攻击代码不应包含 `\0` (空字符)。
- 溢出不应在 `foo()` 函数退出前导致程序崩溃。

此类缓冲区溢出示例 (www.us-cert.gov)：

- TA07-089A: Microsoft Windows 动画光标缓冲区溢出 (2007 年 3 月)。
- TA05-362A: Microsoft Windows 图元文件处理缓冲区溢出 (2005 年 12 月)。

更通用的漏洞利用

可以通过将栈段标记为不可执行来阻止基本的栈溢出利用，但存在许多绕过方法，并且这不能阻止更通用的溢出利用。

通用缓冲区溢出利用通常包括两步：

1. 设法将攻击代码放入程序空间。
2. 使程序执行攻击代码。

插入攻击代码的方法

- 将代码放入栈变量（局部变量）。
- 将代码放入堆变量（`malloc` 分配的变量）。
- 将代码放入静态数据段（静态变量）。
- **利用现有代码：返回到 libc (Return-to-libc)**。
  - 使函数指针或返回地址指向 libc 中的 `exec` 函数。
  - 同时覆盖其第一个参数为 `/bin/sh`。

查找缓冲区溢出

黑客通常通过以下方式查找缓冲区溢出：

1. 在本地机器上运行软件。
2. 向用户输入字段提供长字符串，例如 `"XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"`。
3. 如果软件崩溃，搜索核心转储 (Core Dump) 中的长字符串以找到溢出位置。
4. 也存在一些自动化工具，如 eEye Retina, ISIC。

防止缓冲区溢出

主要问题在于 `strcpy()`、`strcat()`、`sprintf()` 等函数没有范围检查。所谓的“安全”版本如 `strncpy()`、`strncat()` 也常常被误用：

- `strncpy()` 可能导致缓冲区未以空字符结尾。
- **在 `strncpy()`、`strncat()` 中，“差一错误 (Off-by-one bugs)”很常见** –忘记为空终止符留出空间。例如：`strncpy(dest, src, strlen(src)+1)`

防御措施：

1. **静态源代码分析 (Static source code analysis)。**
2. **运行时检查 (Run time checking)。**
3. **黑盒测试 (Black box testing)** (例如 eEye Retina, ISIC)。

静态源代码分析

静态地检查源代码以检测缓冲区溢出。

- 内部代码审查 (Internal code reviews)。
- 咨询公司 (需要高昂费用)。
- 自动化工具 (仍需要专业知识来运行和分析结果)。

运行时检查

- 运行时范围检查 (会显著降低性能，对于 C 和 C++ 等语言较难实现)。
- **栈完整性运行时测试 (Run time tests for stack integrity)。**
  - 在栈帧中嵌入“金丝雀 (Canaries)”，并在函数返回前验证其完整性。

金丝雀类型 (Canary Types)

- **随机金丝雀 (Random canary)：**
  - 程序启动时选择一个随机字符串。
  - 将金丝雀字符串插入每个栈帧。
  - 函数返回前验证金丝雀。
  - 要破坏随机金丝雀，攻击者必须获知当前的随机字符串。
- **终止符金丝雀 (Terminator canary)：**
  - 金丝雀的值为 0、换行符、回车符、EOF (文件结束符) 等。
  - 字符串处理函数不会复制超出终止符的内容。
  - 因此，攻击者不能使用字符串函数来破坏栈。但其他函数仍可能存在问题。

其他安全漏洞

竞争条件 (Race Conditions)

竞争条件攻击 (Race condition attack) 利用了多个指令事务并非原子性 (Atomic) 执行的事实。

示例 1：旧版 UNIX “mkdir”命令中的竞争条件

`mkdir` 执行分为两个阶段：

1. 分配存储空间。
2. 将所有权转移给用户。

攻击：

1. 用户发起 `mkdir`。
2. 用户迅速将新目录（参数）替换为例如 `/etc/passwd` 的符号链接。
3. `mkdir` 进程将 `/etc/passwd` 的所有权转移给用户。许多此类问题也存在于 `/tmp` 目录下的临时文件中。

示例 2：打印服务器配额的竞争条件

用户打印文件的流程：

1. 用户打印文件。
2. 打印服务器确定文件成本。
3. 打印服务器检查账户余额并扣款。
4. 打印服务器将文件加入打印队列。

攻击：

```
lpr smallfile  # 提交一个小文件打印
sleep (1)      # 等待服务器处理到检查余额和扣款步骤
cat bigfile > smallfile # 迅速用大文件替换小文件内容
```

定时攻击 (Timing Attacks)

定时攻击 (Timing attacks) 基于设备响应时间来提取秘密信息，是一种“旁道攻击 (Side Channel Attack)”。

适用于：智能卡、手机、PCI 卡、网络软件等。

示例：

- RSA 幂运算。
- **密码检查和长度判断。**
- 按键间隔时间分析 (例如，对 SSH 的攻击)。

密码检查代码示例（易受定时攻击）：

```
int password_check(char *inp, char *pwd) {
    if (strlen(inp) != strlen(pwd)) return 0; // 长度不同则直接返回
    for(int i=0; i < strlen(pwd); ++i) {
        if (inp[i] != pwd[i]) return 0; // 逐字节比较，一旦不同则返回
    }
    return 1;
}
```

一个简单的定时攻击可以逐个字符地暴露密码长度和密码本身，因为比较不同字符或不同长度的密码所需时间会有差异。

防止定时攻击的修正代码示例：

```
int password_check(char *inp, char *pwd) {
    int oklen = 1;
    if (strlen(inp) != strlen(pwd)) oklen = 0; // 记录长度是否匹配

    int ok = 1;
    // 即使长度不匹配，也完整执行循环，确保比较时间一致
    // 或者确保比较的字符数固定，不足则比较填充值
    int len_to_compare = strlen(pwd); // 或者是一个固定的最大长度
    for(int i=0; i < len_to_compare; ++i) {
```

```
        if (oklen == 0 || i >= strlen(inp) || inp[i] != pwd[i]) { // 处理长度不足或不匹配的情况
            ok = ok & 0;
        } else {
            ok = ok & 1;
        }
    }
    return ok & oklen;
}
```

(注：上述修正代码旨在说明原理，实际安全的密码比较应使用恒定时间比较函数，避免依赖输入数据来改变执行路径或操作次数。一个更简单且常用的方法是比较两个字符串的哈希值，但哈希比较本身也需要注意定时攻击的风险，除非哈希函数和比较过程是恒定时间的。)

网络协议回顾 (Network Protocols Revision)

OSI 模型 (OSI Model)

OSI (Open Systems Interconnection) 模型是一种概念化网络系统的方式，其中每一层都是对其下一层的抽象。

> 记忆方法（自顶向下）：All People Seem To Need Data Processing

- **主机层 (Host Layers)**：
  - 应用层 (Application Layer)：网络进程到应用。
  - 表示层 (Presentation Layer)：数据表示和加密。
  - 会话层 (Session Layer)：主机间通信。
- **介质层 (Media Layers)**：
  - 传输层 (Transport Layer)：端到端连接和可靠性。
  - 网络层 (Network Layer)：路径确定和 IP (逻辑寻址)。
  - 数据链路层 (Data Link Layer)：MAC 和 LLC (物理寻址)。
  - 物理层 (Physical Layer)：介质、信号和二进制传输。

各层详解

- **物理层 (Layer 1: Physical Layer)**：处理比特流，以及它们如何在介质上进行调制、复用和编码。
  - 示例：无线 (Wireless)、铜质双绞线 (Copper Twisted Pair)、光纤 (Optic Fibre)。
- **数据链路层 (Layer 2: Data Link Layer)**：处理帧 (Frames)，关注数据如何在同一局域网 (LAN) 上的设备间传输。
  - 提供服务：数据传输、错误校验、冲突检测。
  - **MAC 地址 (MAC Addresses)**：设备在第二层使用 MAC 地址通信。每个设备通常拥有唯一的 MAC 地址（工厂编程，但现在也可重新分配）。不同制造商有不同的 MAC 地址范围。
  - **地址解析协议 (Address Resolution Protocol, ARP)**：发送方在局域网上发送数据前，若只有目标 IP 地址，可使用 ARP 来确定对应的 MAC 地址。
- **网络层 (Layer 3: Network Layer)**：处理包 (Packets)，关注数据如何在不同网络间路由。
  - 涉及 **IP 地址 (IP Addresses)**，确定包在两点间的最有效路由。

- **IP 地址类型**：
  * IPv4: 32 位地址 (例如 172.16.254.1)，点分十进制表示。
  * IPv6: 128 位地址 (例如 2001:db8:0:1234:0:567:8:1)，十六进制字符串表示，用于应对 IPv4 地址枯竭。
- **子网地址 (Subnet Addresses)**：将大网络划分为小子网。IP 地址被分为网络地址 (Network Address) 和主机地址 (Host Address)，由**子网掩码 (Subnet Mask)** 定义。例如：`192.168.1.150/24` (子网掩码为 `255.255.255.0`)，表示子网 `192.168.1.0/24` 中的主机号 `150`。主机号 `0` 和 `255` 通常保留。
- **路由 (Routing)**：不同网络（子网）上的主机通信，包必须通过**路由器 (Router)** 在网络间传递。路由器根据路由表 (Routing tables) 转发流量。
- **网络地址转换 (Network Address Translation, NAT)**：允许私有网络中的多个设备共享单个公共 IP 地址访问互联网。
  * 当内部网络设备（如 `192.168.10.105`）访问外部服务器（如 Google DNS `8.8.8.8`）时，路由器（公共 IP `221.21.81.133`）会将源 IP 地址重写为其公共 IP，并在收到响应后，将目标 IP 地址重写回原始主机。
- **NAT：端口转发 (Port Forwarding)**：允许来自互联网的特定端口的入站请求被转发到内部网络的特定主机和服务。例如，将路由器公共 IP 的 21 端口 (FTP) 的请求转发到内部服务器 `192.168.50.20`。这常被称为一种“防火墙 (Firewall)”（隐式防火墙）。

- **虚拟局域网 (Virtual LANs, VLANs)**：在第二层交换机上，通过为不同端口分配不同的 VLAN ID，可以将物理交换机划分为多个虚拟交换机，从而隔离不同子网的流量，防止主机通过简单更改 IP 地址来切换子网。
- **传输层 (Layer 4: Transport Layer)**：定义数据包如何在两个端点之间发送（不必在同一网段）。
  - 最常见的协议：
    * **UDP (User Datagram Protocol)**：尽力而为的无连接协议。不保证送达、顺序或进行重传。操作单位是 UDP 数据报 (Datagrams)。
    * **TCP (Transmission Control Protocol)**：面向连接的协议，负责数据包的可靠交付。
      - **TCP 属性**：
        + 使用**三次握手 (TCP 3-way handshake)** 建立连接。
        + 确保应用层数据的有序交付。
        + 重传丢失的数据包。
        + 具有可扩展的“滑动窗口 (Sliding Window)”机制，以实现尽可能快的传输速度同时避免拥塞。
      - 操作单位是 TCP 段 (Segments)。
      - **TCP 端口 (TCP Ports)**：TCP 段分配给特定端口 (1-65535)，用于定义它们所针对的应用程序。通常有目标端口（标准低位端口）和源端口（高位端口）。常见端口号：HTTP: 80, HTTPS: 443, FTP: 21, SSH: 22, DNS: 53。

网络安全 (Network Security)

IP、TCP/IP 和相关协议

- **互联网协议 (Internet Protocol, IP)**：一个无状态协议，用于使用 32 位地址（例如 `129.78.13.49`）在机器间发送数据包。
- 许多服务在 IP 之上使用 **TCP (Transmission Control Protocol)**，即 **TCP/IP**，以提供面向连接的电路。
- IP 地址通过 **域名系统 (Domain Name System, DNS)** 与名称地址（例如 `canvas.sydney.edu.au`）相互转换。
- 大多数本地网络使用以太网，其中机器具有唯一的以太网（或 MAC）地址，通过 **ARP (Address Resolution Protocol)** 映射到 IP 地址。

TCP/IP 三次握手 (TCP/IP Three Way Handshake)

TCP 使用 32 位序列号 (Sequence Numbers) 来识别丢失的数据包并重排乱序到达的数据包。序列号以特定速率递增，并且每个新连接都会大幅增加（例如 BSD Unix 栈中每秒递增 128,000 次，每个新连接增加 64,000）。信道的每个方向都有一个序列号。

由于数据包可能乱序到达，存在一个有效序列号的窗口 $SN, ..., SN + window$。不在此范围内的数据包被视为无效并丢弃。如果接收到的数据包在此范围内但大于当前序列号 +k，则认为其乱序到达并存储起来，等待中间的数据包。

网络攻击类型

- **包嗅探 (Packet Sniffing)**：监听原始网络流量（即窃听）。
  - 由于互联网上大部分信息未加密，嗅探特定链路可能泄露机密信息，如登录名/密码、电子邮件流量（POP3/IMAP 默认未加密，包括密码!）、以及对其他攻击有用的信息（如序列号）。
  - 通常局限于局域网协议（如以太网、802.11），因为嗅探其他协议的设备昂贵，且在高速率下处理数据包需要专门硬件。
- **欺骗 (Spoofing)**：伪造数据包的过程。
  - 通常用于冒充他人或利用协议/实现错误。
  - 两类欺骗攻击：
    * **非盲欺骗 (Non-blind spoofing)**：攻击者既可以向网络注入数据包，也可以嗅探到响应。
    * **盲欺骗 (Blind spoofing)**：攻击者无法看到对其伪造数据包的响应。

  简单欺骗示例：假设 Bob 信任 Alice（例如通过 `/etc/hosts.equiv` 文件）。如果 Alice 的机器关闭，而 Mallory 在同一局域网上，Mallory 只需将其 IP 地址设置为 Alice 的地址，Bob 就会相信 Mallory 是 Alice。

  另一个欺骗示例（Alice 在线）： Mallory 尝试打开一个到 Bob 的连接，冒充 Alice：

  1. Mallory $\rightarrow$ Bob: $SYN(SN_A)$ (Mallory 发送一个 SYN 包给 Bob，声称来自 Alice 的 IP，并使用一个序列号 SN_A)
  2. Bob $\rightarrow$ Alice: $ACK(SN_A + 1), SYN(SN_B)$ (Bob 回复给真正的 Alice，确认 SN_A 并发送自己的 SYN 及序列号 SN_B)

3. Alice $\rightarrow$ Bob: RST (Alice 收到一个不是她发起的连接请求的 ACK，于是发送 RST 包重置连接) 真正的 Alice 可以拆除连接。但是，Mallory 可以在 Alice 发送 TCP RST 包之前对 Alice 执行**拒绝服务攻击 (Denial-of-Service attack)**。

- **拒绝服务攻击原则 (Denial of Service Principles)**：
  - 找到任何一种资源并耗尽它：
    * 带宽 (Bandwidth)
    * CPU 或路由器处理能力
    * 内存、磁盘空间
    * 文件描述符、套接字（或其他操作系统资源）
    * 人类的认知极限
  - 尽可能多地控制攻击者（例如，僵尸网络 - Botnets）。
  - 寻找放大器（例如，利用协议特性放大攻击流量）。
  - 选择带宽充足的放大器。

可能的考点与示例考题

可能考点

- 安全系统设计的**环形原则 (Ring Design)** 和**可信计算基 (Trusted Computing Base, TCB)** 的概念。
- **Bell-LaPadula 模型**的简单安全属性 (No Read Up) 和星号属性 (No Write Down)。
- **隐蔽信道 (Covert Channels)** 的概念及其利用方式。
- **橙皮书 (Orange Book)** 的安全评级等级及其含义。
- 常见的导致安全漏洞的编程错误，特别是**未能检查用户输入**和**缓冲区溢出**。
- **缓冲区溢出 (Buffer Overflows)** 的原理、利用方式（包括栈溢出和返回到 libc）及防御措施（静态分析、运行时检查、金丝雀）。
- **竞争条件 (Race Conditions)** 的原理和示例。
- **定时攻击 (Timing Attacks)** 的原理、适用场景以及如何通过代码设计来防范。
- OSI 七层模型各层的主要功能，特别是物理层、数据链路层（MAC 地址、ARP）、网络层（IP 地址、子网、路由、NAT、端口转发、VLAN）和传输层（TCP、UDP、TCP 端口、三次握手）。
- 网络攻击类型：**包嗅探 (Packet Sniffing)**、**IP 欺骗 (IP Spoofing)**（包括非盲和盲欺骗）、**拒绝服务攻击 (Denial of Service, DoS)** 的基本原理。

示例考题

问题 1：解释 Bell-LaPadula 安全模型中的“简单安全属性 (No Read Up)” 和 “星号属性 (No Write Down)”。为什么即使严格执行这些属性，系统仍可能存在隐蔽信道 (Covert Channels)？请举一个隐蔽信道的例子。

参考答案：

中文：

- **简单安全属性 (No Read Up - NRU)：**指一个主体（如进程或用户）不能读取安全级别高于其自身权限级别的数据对象。例如，一个持有“秘密”级别权限的用户不能读取“绝密”级别的文件。
- **星号属性 (*-Property / No Write Down - NWD)：**指一个主体不能向安全级别低于其自身权限级别的数据对象写入信息。例如，一个持有“秘密”级别权限的用户不能向“公开”级别的文件写入数据，以防止信息从高级别泄露到低级别。

- **隐蔽信道存在原因：** Bell-LaPadula 模型主要关注显式的信息流控制（基于读写操作和安全级别）。然而，系统中的其他共享资源或机制可能被滥用来间接传递信息，这些通信路径并未被模型的策略所覆盖，从而形成隐蔽信道。
- **隐蔽信道示例：文件锁定 (File Locking)**。一个高安全级别的主体可以通过以特定模式（例如，代表二进制的 1 和 0）频繁地锁定和解锁一个低安全级别主体也能观察到的共享文件。低安全级别的主体通过持续检查该文件的锁定状态，就能解码出高安全级别主体传递的信息，即使它们之间没有直接的、符合 Bell-LaPadula 策略的读写操作。

English:

- **Simple Security Property (No Read Up - NRU):** This property states that a subject (e.g., a process or user) cannot read data objects that have a security level higher than the subject's own clearance level. For example, a user cleared for "Secret" level cannot read a "Top Secret" file.
- ***-Property (Star Property / No Write Down - NWD):** This property states that a subject cannot write information to data objects that have a security level lower than the subject's own clearance level. For example, a user cleared for "Secret" level cannot write to a "Public" file, to prevent information leakage from a higher to a lower level.
- **Reason for Covert Channels:** The Bell-LaPadula model primarily focuses on explicit information flow control (based on read/write operations and security levels). However, other shared resources or mechanisms within the system can be exploited to indirectly transfer information. These communication paths are not covered by the model's policy, thus forming covert channels.
- **Example of a Covert Channel: File Locking.** A subject with a high security clearance can frequently lock and unlock a shared file (e.g., in a pattern representing binary 1s and 0s) that can also be observed by a subject with a low security clearance. The low-security subject, by continuously checking the lock status of this file, can decode the information being transmitted by the high-security subject, even though there are no direct read/write operations between them that would violate the Bell-LaPadula policy.

问题 2：什么是缓冲区溢出 (Buffer Overflow)？请描述一种利用栈缓冲区溢出的基本方法，并列举至少两种防御缓冲区溢出的技术。

参考答案：

中文：

- **定义：**缓冲区溢出是一种常见的软件安全漏洞，当程序试图向内存中的缓冲区写入超出其预留容量的数据时发生。这些多余的数据会覆盖相邻内存区域，可能破坏数据、程序状态，甚至改变程序的执行流程。
- **基本栈溢出利用方法：**
  1. **找到漏洞：**识别程序中存在向固定大小的栈缓冲区复制用户可控数据的函数，且未使用边界检查（例如使用 `strcpy()`）。
  2. **构造恶意输入：**攻击者精心构造一个超长输入字符串。这个字符串通常包含：
     - 一段**填充数据 (Padding)**，用于填满缓冲区。
     - 一个**新的返回地址 (New Return Address)**，该地址指向攻击者植入的恶意代码。
     - **恶意代码 (Shellcode)**，通常是执行特定操作（如打开一个 shell）的一小段机器码。
  3. **覆盖返回地址：**当易受攻击的函数执行时，超长输入会溢出缓冲区，覆盖保存在栈上的原始函

数返回地址，将其替换为指向恶意代码的新返回地址。
4. **执行恶意代码：**当函数执行完毕，试图返回时，它会跳转到被篡改的返回地址，从而执行攻击者的恶意代码。

- **防御技术：**
1. **使用边界检查的函数/安全的编程实践：**避免使用像 `strcpy()`、`gets()` 这样不安全的函数，改用有边界检查的替代品（如 `strncpy()`、`fgets()`，并正确处理长度和终止符），或者使用更安全的语言特性（如 C++ 的 `std::string`）。
2. **栈保护机制/金丝雀 (Stack Canaries)：**在编译时，在栈帧中的返回地址前插入一个称为“金丝雀”的随机值。在函数返回前，检查该金丝雀值是否被改变。如果被改变，说明可能发生了缓冲区溢出，程序可以中止执行以防止恶意代码执行。
3. **地址空间布局随机化 (Address Space Layout Randomization, ASLR)：**使得栈、堆、共享库等内存区域的加载地址随机化，增加攻击者预测恶意代码或关键函数（如 libc 中的函数）地址的难度。
4. **数据执行保护 (Data Execution Prevention, DEP) / NX 位 (No-eXecute bit)：**将内存区域标记为不可执行（如栈和堆），即使攻击者成功将恶意代码写入这些区域，CPU 也不会执行它们。

English:

- **Definition:** A buffer overflow is a common software security vulnerability that occurs when a program attempts to write more data to a buffer in memory than it is allocated to hold. This excess data overwrites adjacent memory locations, potentially corrupting data, program state, or even altering the program’s execution flow.
- **Basic Stack Overflow Exploitation Method:**
1. **Find Vulnerability:** Identify a function in the program that copies user-controllable data into a fixed-size stack buffer without bounds checking (e.g., using `strcpy()`).
2. **Craft Malicious Input:** The attacker crafts an overly long input string. This string typically contains:
   - **Padding** to fill the buffer.
   - A **New Return Address** that points to the attacker’s injected malicious code.
   - **Shellcode**, which is a small piece of machine code that performs a specific action (e.g., opens a shell).
3. **Overwrite Return Address:** When the vulnerable function executes, the oversized input overflows the buffer, overwriting the original function return address stored on the stack with the new return address pointing to the shellcode.
4. **Execute Malicious Code:** When the function finishes and attempts to return, it jumps to the tampered return address, thereby executing the attacker’s malicious code.
- **Defense Techniques:**
1. **Use Bounds-Checking Functions / Secure Programming Practices:** Avoid using unsafe functions like `strcpy()` and `gets()`. Instead, use bounded alternatives (like `strncpy()`, `fgets()`, handling lengths and terminators correctly) or safer language features (like C++’s `std::string`).
2. **Stack Protection Mechanisms / Canaries:** At compile time, insert a random value called a “canary” on the stack frame before the return address. Before the function returns, this canary value is checked. If it has been altered, a buffer overflow has likely occurred, and the program can terminate to prevent the execution of malicious code.
3. **Address Space Layout Randomization (ASLR):** Randomizes the memory locations of

the stack, heap, shared libraries, etc., making it harder for attackers to predict the addresses of their shellcode or critical functions (like those in libc).

4. **Data Execution Prevention (DEP) / NX bit (No-eXecute bit):** Marks memory regions (like the stack and heap) as non-executable. Even if an attacker successfully writes malicious code to these areas, the CPU will not execute it.

问题 3：什么是定时攻击 (Timing Attack)？请举例说明一个容易受到定时攻击的密码比较函数的缺陷，并简述如何修正该缺陷以提高安全性。

参考答案：

中文：

- **定义：**定时攻击是一种旁道攻击，攻击者通过精确测量密码算法或安全操作（如密码比较）执行所需的时间来推断敏感信息（如密钥、密码长度或内容）。其基本原理是，不同的输入或操作路径可能导致执行时间的微小但可测量的差异。
- **易受攻击的密码比较函数缺陷示例：**

```
int vulnerable_password_check(char *userInput, char *storedPassword) {
    // 1. 长度比较可能提前退出
    if (strlen(userInput) != strlen(storedPassword)) {
        return 0; // 错误：长度不同，执行时间短
    }
    // 2. 逐字节比较，一旦不匹配即提前退出
    for (int i = 0; i < strlen(storedPassword); ++i) {
        if (userInput[i] != storedPassword[i]) {
            return 0; // 错误：字符不匹配，执行时间取决于第一个不匹配字符的位置
        }
    }
    return 1; // 匹配成功
}
```

缺陷在于：

1. 如果输入的密码长度与存储的密码长度不同，函数会立即返回，执行时间非常短。攻击者可以通过尝试不同长度的输入来推断出正确密码的长度。
2. 即使长度相同，函数也是逐个字符比较，一旦遇到不匹配的字符就会立即返回。这意味着，如果输入的第一个字符就错了，执行时间会比第一个字符正确但第二个字符错误的情况要短。攻击者可以逐个字符猜测，通过观察响应时间的变化来确认每个字符是否正确。

- **修正缺陷以提高安全性：**核心思想是确保无论输入如何（在一定范围内），比较操作的执行时间都是恒定的，或者至少与秘密信息（如密码内容）无关。一种改进方法是确保比较操作总是执行相同数量的步骤，并且不提前退出。

```
int improved_password_check(char *userInput, char *storedPassword) {
    int storedLen = strlen(storedPassword);
    int inputLen = strlen(userInput);
    int result = 1; // 假设匹配

    // 步骤 1: 比较长度，但不基于此提前返回，而是记录结果
```

```
    if (inputLen != storedLen) {
        result = 0;
    }

    // 步骤 2: 总是比较固定数量的字符 (例如, 存储密码的长度)
    // 即使长度不匹配, 也继续“假比较”以耗时
    // 更安全的方法是使用专门的恒定时间比较函数
    for (int i = 0; i < storedLen; ++i) {
        // 如果输入长度不足, 则比较输入中的字符与一个无关值,
        // 或者确保这里的比较不泄露信息。
        // 关键是循环次数固定。
        if (i >= inputLen || userInput[i] != storedPassword[i]) {
            // 即使发现不匹配, 也不立即返回
            // result &= 0; // 这是一个常见但不完全安全的做法, 因为后续比较可能仍有时间
        }
    }
    // 一种更安全的比较方法是, 对所有字节进行异或, 然后检查结果是否为 0,
    // 并确保所有操作 (包括条件分支) 的时间是恒定的。
    // 最终, 将长度比较的结果和内容比较的结果结合起来。
    // 实践中, 通常使用加密库提供的恒定时间字符串比较函数。
    // 简化的示意:
    int comparison_match = 1;
    for (int i = 0; i < storedLen; i++) {
        char a = (i < inputLen) ? userInput[i] : 0; // 安全处理长度不足
        char b = storedPassword[i];
        if (a != b) {
            comparison_match = 0;
        }
    }
    return (result & comparison_match);
}
```

更推荐的做法是使用专门设计的、经过验证的恒定时间比较函数 (Constant-Time Comparison Functions)，这类函数通常在安全的加密库中提供，它们能确保比较操作的时间不依赖于被比较数据的内容。

English:

- **Definition:** A timing attack is a side-channel attack where an attacker infers sensitive information (such as cryptographic keys, password length, or content) by precisely measuring the time it takes for a cryptographic algorithm or a security operation (like password comparison) to execute. The underlying principle is that different inputs or operational paths can lead to small but measurable differences in execution time.
- **Example of a Vulnerable Password Comparison Function:**

```
int vulnerable_password_check(char *userInput, char *storedPassword) {
    // 1. Length comparison may exit early
```

```
    if (strlen(userInput) != strlen(storedPassword)) {
        return 0; // Flaw: Different lengths, short execution time
    }
    // 2. Byte-by-byte comparison, exits early on mismatch
    for (int i = 0; i < strlen(storedPassword); ++i) {
        if (userInput[i] != storedPassword[i]) {
            return 0; // Flaw: Character mismatch, execution time depends on the position of
        }
    }
    return 1; // Match successful
}
```

The flaws are:

1. If the length of the input password differs from the stored password, the function returns immediately, resulting in a very short execution time. An attacker can infer the length of the correct password by trying inputs of different lengths.
2. Even if the lengths are the same, the function compares character by character and returns immediately upon encountering a mismatch. This means if the first character of the input is wrong, the execution time will be shorter than if the first character is correct but the second is wrong. An attacker can guess character by character, observing changes in response time to confirm if each character is correct.

- **Correcting the Flaw for Improved Security:** The core idea is to ensure that the execution time of the comparison operation is constant, regardless of the input (within certain bounds), or at least independent of the secret information (like the password content). One way to improve this is to ensure the comparison operation always performs the same number of steps and does not exit early.

```
int improved_password_check(char *userInput, char *storedPassword) {
    int storedLen = strlen(storedPassword);
    int inputLen = strlen(userInput);
    int result = 1; // Assume match

    // Step 1: Compare lengths, but don't return early based on this; record the result
    if (inputLen != storedLen) {
        result = 0;
    }

    // Step 2: Always compare a fixed number of characters (e.g., length of stored password)
    // Even if lengths don't match, continue "dummy comparisons" to consume time.
    // A safer method uses dedicated constant-time comparison functions.
    int comparison_match = 1;
    for (int i = 0; i < storedLen; i++) {
        // Safely handle insufficient input length
        char a = (i < inputLen) ? userInput[i] : 0;
        char b = storedPassword[i];
```

```
        // This simple check is still not perfectly constant time due to branching
        // A better constant-time approach involves bitwise operations that don't branch base
        // For example, accumulate differences and check if the accumulator is zero at the en
        if (a != b) {
            comparison_match = 0;
        }
    }
    return (result & comparison_match);
}
```

It is highly recommended to use specifically designed and validated Constant-Time Comparison Functions, which are typically provided in secure cryptographic libraries. These functions ensure that the time taken for the comparison operation does not depend on the content of the data being compared.