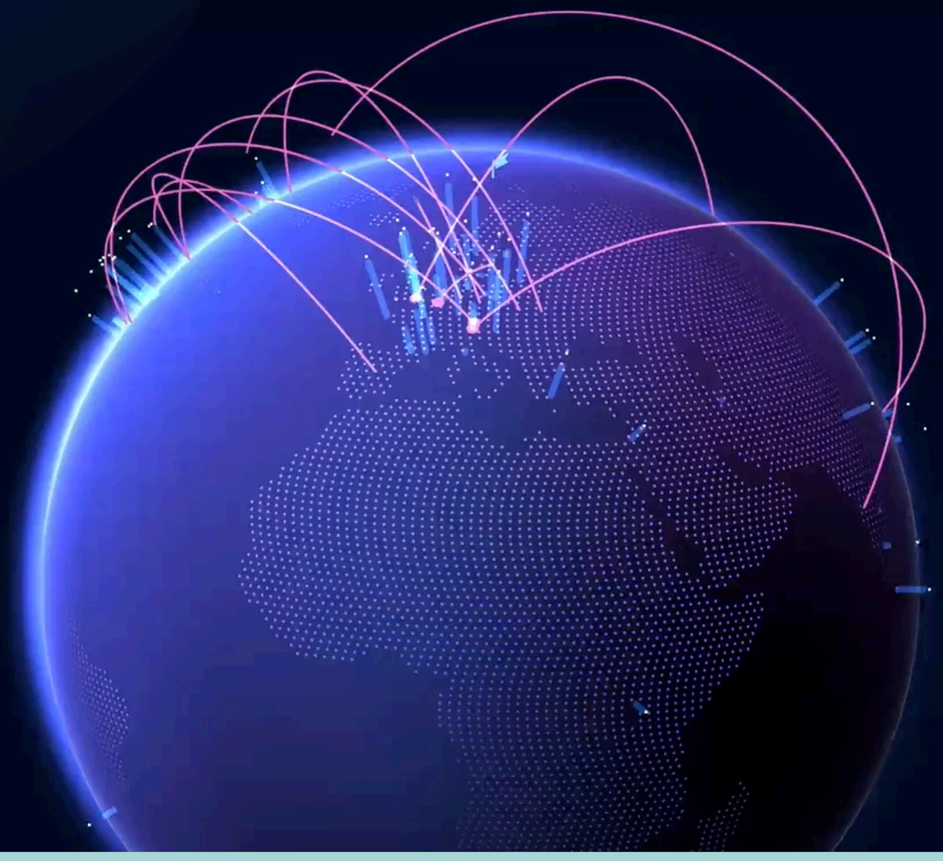



# 计算机网络进阶

红岩网校工作站运维安全部 黄凯升

# Table of Contents

# Part I. Network Debugging

网络调试

# 代理

- 代理 (Proxy) 通常工作在应用层，应用程序进行网络请求时，先连接代理服务器，然后由代理服务器进行实际的网络请求。
- 利用代理服务器，我们可以截获应用程序的流量。
- 常用的代理协议有 SOCKS 代理和 HTTP 代理。
  - SOCKS 协议的最新版本是 SOCKS5，能够支持远程解析、TCP 和 UDP。
  - HTTP 代理协议行为分为两种，对于 HTTP 连接直接使用 GET、POST 等方法。对于非 HTTP（包括 HTTPS）协议，采用 CONNECT 方法。

# 代理抓包

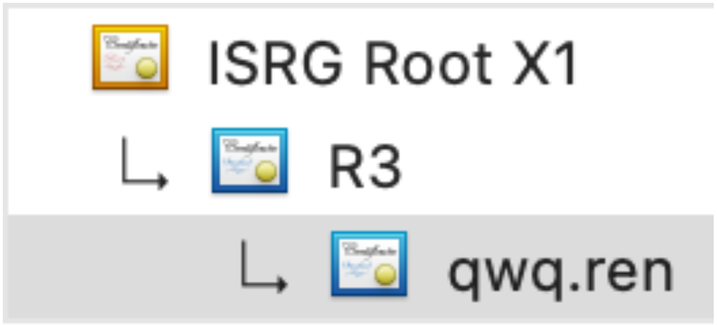
- 抓包是指截获网络请求和响应，以便分析应用中网络异常的情况。
- 桌面端，代理抓包常用的工具有 Fiddler (付费)、Charles (付费)、mitmproxy (开源免费) 等。
- 移动端，代理抓包常用的工具有 Stream (iOS 端，免费)，HTTP Toolkit (Android 端，免费)
- 由于 Fiddler 和 Charles 是付费的，我们讲解一下 mitmproxy。

# HTTPS 抓包

- 我们并不能直接抓取到 https 访问的内容，这是因为 https 进行了加密。
- mitmproxy 可以抓取 https 的流量，它的原理是劫持了流量并自行签署了一个证书，从而假扮目标网站。
- 但我们使用 mitmproxy 抓包下访问 https 网站时，会发现报了证书错误，这是因为它自签名的证书不被系统信任。
- mitmproxy 扮演的是“中间人 (Man-In-The-Middle, MITM)”的角色。

# 证书

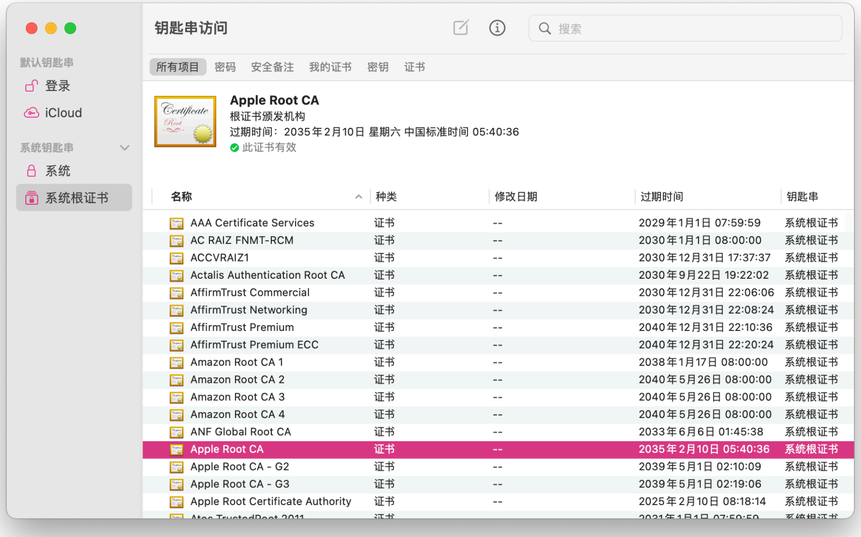
- https 通过 SSL/TLS 实现了 http 流量的加密，但是它如何保证对方服务器可信呢?
- 现代互联网中，我们使用“数字证书”来进行通信的可信性验证和加密传输。然而，如何验证证书本身是否可信又是一个问题。
- 证书信任链体系是目前应用最为广泛的方式。




- 如右图， qwq.ren 是我的博客的域名，其证书由 R3 签发，可以认为 R3 为我对于该域名的合法持有做了担保。
- 同时可以看到，R3 的证书由 ISRG Root X1 签发，也就是说，ISRG Root X1 又对 R3 做了担保。

# CA 与根证书

- 像这样，负责签发证书的机构就被称之为 CA (Certification Authority)。CA 分为两种，中级 CA 和根 CA。
  - 根 CA：数量较少，其证书由自己签署，受到操作系统和浏览器信任。
  - 中级 CA：数量较多，其证书由根 CA 或其它中级 CA 签署，其是否受信任取决于它的上级 CA。
- 根 CA 自己的证书称为根证书，根证书是证书信任链的终点，是整个证书信任链的安全基石。通常以内置在操作系统中的形式分发。
- 思考：信任第三方根证书有什么风险?



# 加密

- 加密分为对称加密和非对称加密两种。
- 对称加密：使用同一份密钥进行加密和解密操作。
  - 常见对称加密算法：AES、DES 等。
- 非对称加密：密钥为一对，分别是公钥和私钥，公钥可以公开，私钥保密。使用公钥加密的信息只有使用私钥才能解密，而使用私钥签署的信息可以由公钥验证其不是伪造的。
  - 常见非对称加密算法：RSA、ECC 等。
- 由于两类加密算法的密码学特性，通常非对称加密需要更长的密钥长度。例如依赖矩阵变换的 AES 算法只需要 256 位密钥长度就可以达到足够的安全性，而依赖于大数的质因数分解困难性的 RSA 算法则通常需要 2048 位密钥长度才能足够安全。
- 非对称加密通常性能也远低于对称加密算法。因此 https 仅使用非对称加密来进行密钥交换的握手，握手完成后便建立了对称加密传输。

# HTTP 工具

- 这一部分我们将以演示为主。
- 对于 HTTP 活动的调试，除了进行代理抓包外，浏览器的“开发者工具”也是我们的好帮手。
- 在命令行下，我们通常通过 curl 来调试和请求 HTTP。
- 在测试接口等场景，Postman 这类应用也十分常用。

# TCP 调试

- TCP 连接的调试可以使用 telnet 或者 netcat (nc)。
- telnet 和 nc 都将 TCP 连接和标准输入输出连接起来。
- telnet 和 nc 使用方式类似。以 nc 为例，当我们需要连接到 redrock.team 的 23 端口时，只需：
  - `$ nc redrock.team 23`
- 区别在于，nc 还可以监听端口，只需：
  - `$ nc -l 23`
  - 这样 nc 就监听了 23 端口

# 网卡抓包

- 代理抓包虽然能够抓到应用层的包，但当我们遇到复杂的网络问题时，代理抓包并不能帮助我们。
- Wireshark / tcpdump 可以针对网卡进行抓包，在这种抓包方式下，我们可以获得所有通过该网卡的数据包和以太帧信息。

# IPv6 链路本地地址

- IPv6 从设计之初就考虑到了本地网络设备的点对点互联问题，因此 IPv6 有一个称为链路本地地址 (Link-local address) 的特殊地址。
- IPv6 标准规定链路本地地址必须存在，fe80::/10 下的地址为链路本地地址。
- 链路本地地址由于和数据链路层强相关，因此要通过链路本地地址访问设备必须附带网卡名称。
- 链路本地地址通常使用 EUI-64 方式生成。
  - EUI-64 是 IPv6 无状态地址自动配置 (SLAAC) 中常用的地址生成方式，它根据网卡 MAC 生成地址的后 64 位。方法是，取 MAC 地址将其第一个字节的第七位取反，并在 MAC 地址的中间插入 0xFFFE。
  - 例：00:e2:c5:87:a2:35，所生成的链路本地地址是 fe80::02e2:c5ff:fe87:a235。

# 特殊的组播地址

- 组播地址往往在网络调试中有着妙用。
- 对于 IPv4，有一个特殊的组播地址 224.0.0.1，所有主机默认均在这个组播组中。可以用于探测本地网络中的主机。
- 对于 IPv6，也有一个特殊的组播地址 ff02::1，注意 IPv6 的这个组播地址也是一种链路上的地址，需要附带网卡名。
- 考虑链路本地地址的特性，当我们想要寻找“失落的主机”时，可以利用链路本地地址和组播地址来寻找它。

# Part II. Intro to Socket Programming

初探 socket 编程

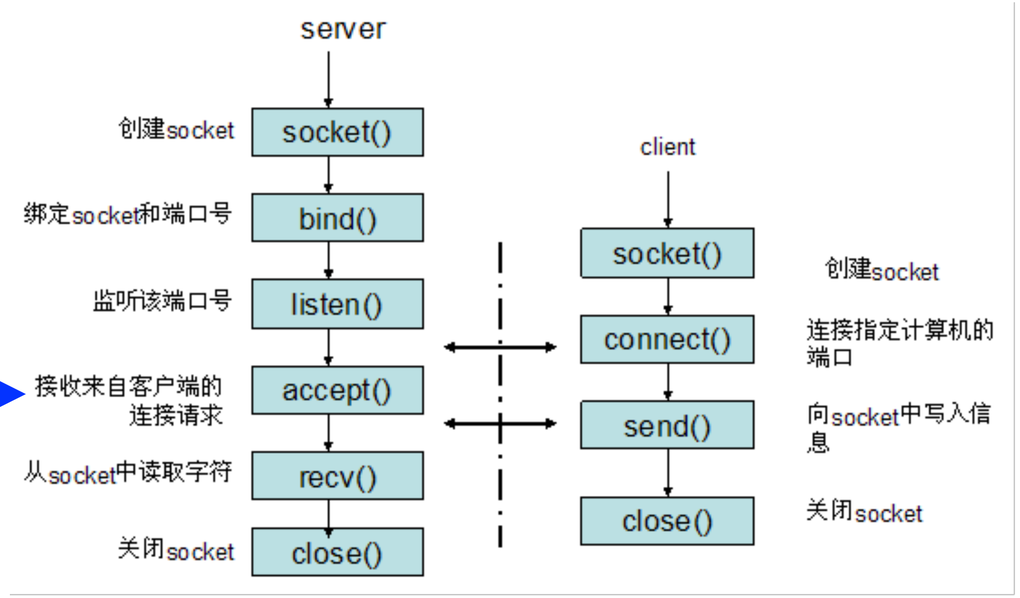
# 进程间通信的方式

- 思考一下，对于两个进程而言，有哪些方式可以让它们互相通信?
- 我们前面已经学过的方式主要是使用管道连接进程的输入输出流。
  - 除此之外，还有共享内存、信号量等通信方式。
- 然而，在网络上，我们不能直接将两个进程用管道连接起来。因为进程彼此无法找到对方。幸好在 TCP/IP 协议族中，我们有网络层的 IP 地址来唯一标识一台主机，还有传输层协议和端口来唯一标识一台主机上的进程。
- 因此，我们只需要使用 (协议, IP, 端口) 这个三元组即可唯一确定一台主机上的进程。在这个思想指导下就形成了套接字 (socket)。

# 什么是 socket

- 让我们回顾一下 UNIX 哲学 —— “一切皆文件”。
- 事实上，我们可以把一个连接看成是一个“文件”，发送数据认为是写文件，接收数据认为是读文件 。连接开始时打开“文件”，连接结束时关闭“文件”。
- socket 就是典型的“打开-读写-关闭”模型。对于 TCP socket，其流程如下图：

注意，这个过程会派生一个新的 socket



# TCP socket (以 Python 为例)

- socket 相关位于 socket 包中。
- 对于 TCP 服务器而言，应当遵循标准的创建 socket、绑定 (bind) socket、监听 (listen) 端口和接受连接的步骤。
- 创建 socket 使用 socket.socket，其函数声明为：
  - `socket(family=-1, type=-1, proto=-1, fileno=None)`
  - family 指定了协议族，通常对于 IPv4 选择 socket.AF_INET (默认)，对于 IPv6 选择 socket.AF_INET6。
  - type 指定了协议类型，TCP 选择 socket.SOCK_STREAM。
  - socket.socket 会返回一个 socket 对象，下面假定该对象为 sock。

# TCP socket (以 Python 为例)

- 对于之前建立的 socket 对象，我们还需要对 socket 进行 bind 操作。bind 操作向操作系统声明了一个 IP 和端口的二元组，表示对于发往特定 IP 和特定端口的数据是交给本 socket 的。
  - 只有发往被 bind 的 IP 的数据包才会被送到本 socket。对 IPv4 而言，0.0.0.0 表示 bind 任意地址，即凡是发往本机该端口的包都能够被收到。在 IPv6 中任意地址为 ::。
- 例如，我们要 bind 任意 IP 的 4444 端口：
  - `sock.bind(("0.0.0.0", 4444))`
- 注意，即便我们不进行下一步的监听，被 bind 的 socket 其它程序也无法使用。

# TCP socket (以 Python 为例)

- 然后需要的就是监听端口，只需执行 `sock.listen()` 即可。
  - listen 有一个参数 backlog，表示半连接状态队列的长度
- 然后我们的 socket 就可以开始接受连接了，循环调用 accept 方法即可。
  - accept 返回两个值，表示当前连接的一个子 socket 和一个对端 IP 和端口的二元组。
- 对这个表示当前连接的子 socket，可以通过 recv 和 send 方法收发数据。
  - 因为 TCP 是流式传输，recv 需要指定一个缓冲区大小，表示一次最多取多少数据。
  - send 方法只需要传入一个字节数组即可发送数据。

# TCP socket (以 Python 为例)

- 对于客户端而言，连接 socket 也十分简单。
- 首先是创建 socket，然后通过 `sock.connect()` 传入一个主机+端口的二元组即可，例如：
  - 连接 www.baidu.com 的 80 端口：`sock.connect(("www.baidu.com", 80))`
- 对于客户端连接服务器，并没有派生新的 socket 的必要，我们只需要在这个 socket 上进行数据收发即可。
- 对 TCP 连接关闭之前应当先进行 shutdown 操作。shutdown 操作有三种参数：
  - SHUT_RD：关闭输入流，socket 无法接收数据，但仍然可以发送。
  - SHUT_WR：关闭输出流，socket 无法发送数据，但仍然可以接收。
  - SHUT_RDWR：同时关闭输入输出流，socket 无法收发数据。

# 阻塞与多线程

- 线程的概念：线程 (thread) 是操作系统能够进行运算调度的最小单位。一条线程指的是进程中的一个单一顺序的控制流。多个线程之间可以并发执行。
- send 和 recv 都是阻塞的，这就意味着，如果数据发送十分缓慢或者没有数据可供读取，send/recv 会一直占用线程。
- 因此，我们可以选择对于每个 socket，我们分配一对读写线程来操作，避免单个线程被阻塞而无法提供服务。
- Python 中使用 threading 模块来操作线程：
  - 创建线程 `threading.Thread(target=thread_func, args=(), kwargs={})`，返回一个线程实例。
  - 使用 start 方法启动线程
  - 使用 join 方法在当前线程中等待某线程完成

# 多线程的资源竞态问题

- 在多线程程序中，由于内存是共享的，可能会发生两个线程同时读写一个变量的情况。
- 为了避免这种情况，人们提出了“锁”的机制，锁主要分为两种：
  - 自旋锁：线程不断争抢锁，得到锁的线程得以访问资源，其它线程继续尝试争抢
  - 互斥锁：线程争抢锁，得到锁的线程得以访问资源，其它线程陷入休眠，锁释放时被唤醒
- 这两种锁各有各的优缺点：
  - 自旋锁：线程不断尝试争抢锁的忙等待状态消耗大量 CPU，但优点是没有唤醒开销速度快。通常用于被锁保护的代码执行时间较短的场景。
  - 互斥锁：线程争抢锁失败时休眠，几乎不消耗 CPU，但引入了唤醒时的额外开销速度较慢。通常用于被锁保护的代码执行时间较长的场景。
- 锁通常由操作系统实现的**原子**操作 API 实现，底层往往依赖处理器的原子指令。

# I/O 复用

- 思考一下，我们如何知道一个 socket 是准备好读写的?
- 考虑以下场景：
  - 客户端连接上后不发送数据，或者以非常缓慢的方式发送，这时候 recv 要么大部分时间在等待直到单次 recv 超时，要么不断地去触发
  - 对方接收数据过于缓慢，我方发送的数据达到了内核 TCP 缓冲区上限，此时使用 send 无法发出数据而陷入阻塞
- 以上两个场景对于网络应用程序而言都是致命的，那么如何解决这个问题呢?
- 操作系统往往提供了 API 来等待 I/O 完成。例如 Linux 提供了 epoll，由操作系统来监听多个文件描述符，可用时返回给程序，避免程序进行大量轮询。
- Python 的 selectors 模块封装了 I/O 复用的 API。

# 字节序

- 字节序是指对于一个多字节变量，例如 32 位整数高低位的排列方式。
  - 大端序 (Big-endian)：多位数的高位位于内存中较低的地址处，高位放在较低的地址处
  - 小端序 (Little-endian)：多位数的低位位于内存中较低的地址处，高位放在较高的地址处
- 例如 19260817 其十六进制表示为 0x01317D1E，则：
  - 大端序下，内存中存储的值为 0x01, 0x31, 0x7D, 0x1E
  - 小端序下，内存中存储的值为 0x1E, 0x7D, 0x31, 0x01
- 现代大部分处理器架构 (x86, arm) 等均采用小端序。
- 在网络中，标准规定网络字节序应当为大端序。

# 换行符

- 由于历史原因，换行符也有着很大的差异：
  - Windows 使用 CR LF 即 “\r\n” 作为换行符
  - 经典 Mac OS（现已几乎绝迹）使用 CR 即 “\r” 作为换行符
  - Linux、现代 macOS、BSD 等类 Unix 操作系统使用 LF 即 “\n” 作为换行符。
- 网络中，标准规定使用 CRLF 即 “\r\n” 作为换行符。