可靠通讯

- **零信任网络**
  - **边界安全**
    - **基于边界的安全模型：**根据某类与宿主机相关的特征，如机器所处的位置、IP 地址、子网等，把网络划分为不同的区域，对应于不同风险级别和允许访问的网络资源权限，将安全防护措施集中部署在各个区域的边界之上，重点关注跨区域的网络流量。相关VPN、DMZ、防火墙、内网、外网等概念。
    - **思路：**在可用性和安全性之间权衡取舍，着重对经过网络区域边界的流量进行检查，对可信任区域内部机器之间的流量则给予直接信任或较为宽松的处理策略，减小了安全设施对整个应用系统复杂度的影响，以及网络传输性能的额外损耗。
    - **缺陷：**边界上的防御措施即使自身能做到永远滴水不漏牢不可破，一旦可信的网络区域中的某台服务器被攻陷，那边界安全措施就成了马其诺防线，攻击者很快就能以一台机器为跳板，侵入到整个内网。
  - **零信任安全模型**
    - **中心思想：**不应当以某种固有特征来自动信任任何流量，除非明确得到了能代表请求来源的身份凭证，否则一律不会有默认的信任关系。
    - **特征**
      - **零信任网络不等同于放弃在边界上的保护设施**
      - **身份只来源于服务：**服务所部署的 IP 地址、服务实例的数量随时都可能发生变化，因此，身份只能来源于服务本身所能够出示的身份凭证（通常是数字证书），而不再是服务所在的 IP 地址、主机名或者其它特征。
      - **服务之间也没有固有的信任关系：**只有已知明确授权的调用者才能访问服务，阻止攻击者通过某个服务节点中的代码漏洞来越权调用到其他服务。如果某个服务节点被成功入侵，这一原则可阻止攻击者执行扩大其入侵范围。
      - **集中、共享的安全策略实施点：**微服务分散治理，但涉及安全的非功能性需求最好除外。要写出高度安全的代码极为不易，为此付出的精力甚至可能远高于业务逻辑本身；让服务各自处理安全问题很容易会出现实现不一致或者出现漏洞时要反复修改多处地方，还有一些安全问题如果不立足于全局是很难彻底解决的。
      - **受信的机器运行来源已知的代码：**限制了服务只能使用认证过的代码和配置，并且只能运行在认证过的环境中。零信任安全针对软件供应链（编写代码、自动化测试、自动集成、漏洞扫描、发布上线）的每一步都加入了安全控制策略。
      - **自动化、标准化的变更管理：**独立于应用的安全基础设施，可以让运维人员轻松地了解基础设施变更对安全性的影响，在不影响生产环境的情况下发布安全补丁。
      - **强隔离性的工作负载**
    - **最终目的：**实现整个基础设施之上的自动化安全控制，服务所需的安全能力可以与服务自身一起，以相同方式自动进行伸缩扩展。对于程序来说，做到安全是日常，风险是例外；对于人类来说，做到袖手旁观是日常，主动干预是例外。
  - **Google探索**
    - 为了在网络边界上保护内部服务免受 DDoS 攻击，设计了名为 Google Front End（名字意为“最终用户访问请求的终点”）的边缘代理，负责保证此后所有流量都在 TLS 之上传输，并自动将流量路由到适合的可用区域之中。
    - 为了强制身份只来源于服务，设计了名为 Application Layer Transport Security（应用层传输安全）的服务认证机制，这是一个用于双向认证和传输加密的系统，自动将服务与它的身份标识符绑定。
    - 为了确保服务间不再有默认的信任关系，设计了 Service Access Policy（服务访问策略）来管理一个服务向另一个服务发起请求时所需提供的认证、鉴权和审计策略，并支持全局视角的访问控制与分析。
    - 为了实现仅以受信的机器运行来源已知的代码，设计了名为 Binary Authorization的部署时检查机制，确保在软件供应链的每一个阶段，都符合内部安全检查策略，并对此进行授权与鉴权。同时设计了名为 Host Integrity的机器安全启动程序，在创建宿主机时自动验证包括 BIOS、BMC、Bootloader 和系统内核数字签名。
    - 为了工作负载能够具有强隔离性，设计了名为gVisor的轻量级虚拟化方案，解决容器共享操作系统内核而导致隔离性不足的安全缺陷，做法都是为每个容器提供了一个独立的虚拟 Linux 内核。
- **服务安全**
  - **背景：**介绍在前微服务时代和云原生时代分别是如何实现安全传输、认证和授权的，通过这两者的对比，探讨在微服务架构下，应如何将业界的安全技术标准引入并实际落地，实现零信任网络下安全的服务访问。
  - **建立信任**
    - **前提：**网络世界达成信任只能采用基于权威公证人的信任，即公开密钥基础设施PKI，是构建传输安全层TLS的必要基础。**启用 TLS 对传输通道本身进行加密是让发送者发出的内容只有接受者可以解密是唯一具备可行性的方案**。
    - 建立 TLS 传输只要在部署服务器时预置好CA 根证书，以后用该 CA 为部署的服务签发 TLS 证书。这属于必须集中在基础设施中自动进行的安全策略实施点，面对数量庞大且自动扩缩的服务节点，依赖运维人员手工去部署轮换根证书是不可能的。
    - 微服务中...
      - **单向 TLS 认证：**只需要服务端提供证书，客户端通过服务端证书验证服务器的身份，但服务器并不验证客户端的身份。单向 TLS 用于公开的服务，即任何客户端都被允许连接到服务进行访问，它保护的重点是客户端免遭冒牌服务器的欺骗。
      - **双向 TLS 认证：**客户端、服务端双方都要提供证书，双方各自通过对方提供的证书来验证对方的身份。双向 TLS 用于私密的服务，它除了保护客户端不连接到冒牌服务器外，也保护服务端不遭到非法用户的越权访问。
  - **认证**
    - **服务认证**
      - **云原生：...**
        - 如果每一个服务提供者、调用者均受 Istio 管理，那 mTLS 就是最理想的认证方案，只需要简单的配置，即可对某个K8S名称空间范围内流量均启用 mTLS。
        - 仍存在部分不受 Istio 管理的服务端或者客户端，也可以将 mTLS 传输声明为宽容模式，即受 Istio 管理的服务会允许同时接受明文

和 mTLS 两种流量，明文流量仅用于与那些不受 Istio 管理的节点进行交互，自行想办法解决明文流量的认证问题；而对于服务网格内部的流量，就可以使用 mTLS 认证。一旦所有服务都完成Istio迁移，便可将整个系统设置为严格 TLS 模式。

- **前微服务时代：...**
  - 客户端与认证服务器约定好一组只有自己知道的密钥（Client Secret），由运维人员在线下自行完成，通过参数传给服务，而不是由开发人员在源码或配置文件中直接设定。**密钥就是客户端的身份证明，客户端调用服务时，会先使用该密钥向认证服务器申请到 JWT 令牌，然后通过令牌证明自己的身份，最后访问服务。**
  - **每一个对外提供服务的服务端，都扮演着 OAuth 2 中的资源服务器的角色，它们均声明为要求提供客户端模式的凭证。**
  - Spring Security 提供的过滤器会自动拦截请求、驱动认证、授权检查的执行，申请和验证 JWT 令牌等操作在开发期对程序员，运行期对用户都能做到相对透明。
  - **缺陷：**仍然是一种应用层面的不加密传输的解决方案，对传输过程中内容被监听、篡改、以及被攻击者在传输途中拿到 JWT 令牌后再去冒认调用者身份调用其他服务都是无法防御的。**不适用于零信任安全模型，只能在默认内网节点间具备信任关系的边界安全模型上才能良好工作。**

- **请求认证**
  - **云原生：**Istio仍然能做到单纯依靠基础设施解决问题，整个认证过程无需应用程序参与。当来自最终用户的请求进入服务网格时，Istio会自动根据配置中的JSON Web Key Set来验证令牌的合法性，如果令牌没有被篡改过且在有效期内，就信任 Payload 中的用户身份，并从令牌的 Iss 字段中获得 Principal。
    - **JWKS ：** 是一组 JWK（存储密钥的纯文本格式） 的集合，能通过 JWT 令牌 Header 中的 KID（Key ID）来自动匹配出应该使用哪个 JWK 来验证签名
  - **前微服务时代：**依然是采用 JWT 令牌作为用户身份凭证的载体，认证过程依然在 Spring Security 的过滤器里中自动完成。Spring Security 已经做好了认证所需的绝大部分的工作，真正要开发者去编写的代码是令牌的具体实现。

- **授权**
  - **云原生：**Istio可以通过配置，限制名空间的内部流量只能流经哪些资源或操作。虽然不可能跟由代码实现的授权相比，但多数场景已经够用了。在便捷性、安全性、无侵入、统一管理等，Istio 这种在基础设施上实现授权方案显然就要更具优势。
  - **前微服务时代：**Spring Cloud中授权控制还是使用 Spring Security、通过应用程序代码来实现的。Spring Security 授权方法有两种，一种是使用Configurer来进行集中配置，另一种写法是通过 Spring 的全局方法级安全以及@RolesAllowed注解来做授权控制。后者对代码的侵入性更强，要以注解的形式分散写到每个服务甚至是每个方法中，但好处是能以更方便的形式做出更加精细的控制效果。