最近发现密码学很有意思，刚好还和工作有点关系，就研究了一下，本文是其中一部分笔记和一些思考。

密码学理论艰深，概念繁多，本人知识水平有限，错误难免，如果您发现错误，请务必指出，非常感谢!

本文禁止转载

本文目标：

1. 学习鉴赏TLS协议的设计，透彻理解原理和重点细节
2. 跟进一下密码学应用领域的历史和进展
3. 整理现代加密通信协议设计的一般思路

本文有门槛，读者需要对现代密码学有清晰而系统的理解，建议花精力补足背景知识再读。本文最后的参考文献里有一些很不错的学习资料。

一 . TLS协议的设计目标：

1. 密码学的方法论

密码学和软件开发不同，软件开发是工程，是手艺，造轮子是写代码的一大乐趣。软件开发中常常有各种权衡，一般难有明确的对错，一般还用建筑来比拟软件的结构，设计的优雅被高度重视。

密码学就不一样了。密码学是科学，不是工程，有严格的技术规范，严禁没有经过学术训练者随意创造。要求严谨的理论建模，严密的数学证明。很少有需要权衡的地方，正确就是正确，错误就是错误。又由于密码学过去在军事上的重要价值，各国政府一直投入大量人力物力财力，不断深入强化己方的算法，破解对手的算法，所以密码学就是一种残酷的军备竞赛。

- 密码学有很多的陷阱（下文会介绍几个），设计使用密码学的协议或者软件，是极其容易出错，高风险的专业活动，单纯的码农背景是做不了的。本着**不作死就不会死**的伟大理念，首先推荐读者尽可能使用 TLS 这种标准化，开源，广泛使用，久经考验，高性能的协议。本文也只是整理一点粗浅的科普常识，读完这篇文章，并不能使读者具有设计足够安全的密码学协议的能力。
- 密码学经过几十年的军备竞赛式发展，已经发展出大量巧妙而狡猾的攻击方法，我们使用的算法，都是在所有已知的攻击方法下都无法攻破的，由于我们大多数码农并没有精力去了解最前沿的攻击方法，所以我们其实并没有能力去评价一个加密算法，更没有能力自己发明算法。所以最好跟着业界的主流技术走，肯定不会有大错。
- 现代密码学近20年进展迅猛，现在搞现代密码学研究的主要都是数学家，在这个领域里面以一个码农的知识背景，已经很难理解最前沿的东西，连正确使用加密算法都是要谨慎谨慎再谨慎的。一个码农，能了解密码学基本概念，跟进密码学的最新应用趋势，并正确配置部署TLS这种协议，就很不错了。
- 密码学算法很难被正确地使用，各种细节非常容易出错。 例如：
    - 1.大多数码农都听说过aes，可是大多数都不了解细节，比如：aes应该用哪种模式？应该用哪种padding？IV/nonce应该取多少bit？IV/nonce应该怎么生成？ key size应该选多大？key应该怎么生成？应不应该加MAC？MAC算法的选择？MAC和加密应该怎么组合？
    - 2.大多数知道RSA的码农分不清 RSASSA-PKCS1-v1_5 ， RSAES-OAEP 和 RSASSA-PSS
    - 3.更多错误参见 这个stackoverflow问答，强烈推荐仔细阅读

- 密码学算法很难被正确地实现(代码实现过程中会引入很多漏洞，比如HeartBleed，比如各种随机数生成器的bug，时间侧通道攻击漏洞)
- 不能一知半解，绝对不能在一知半解的情况下就动手设计密码学协议。犹如“盲人骑瞎马，夜班临深池”。
- 不能闭门造车，密码学相关协议和代码一定要开源，采用大集市式的开发，接受peer review，被越多的人review，出漏洞的可能越小（所以应该尽可能使用开源组件）

2. TLS的设计目标

TLS的设计目标是构建一个安全传输层（Transport Layer Security ），在基于连接的传输层（如tcp）之上提供：

1. 密码学安全 (1). 保密， message privacy (保密通过加密encryption实现，所有信息都加密传输，第三方无法窃听) (2). 完整性， message integrity（ 通过MAC校验机制，一旦被篡改，通信双方会立刻发现 ） (3). 认证， mutual authentication （双方认证,双方都可以配备证书，防止身份被冒充 ）
2. 互操作，通用性 （ 根据公开的rfc，任何符合rfc的软件实现都可以互操作，不受限于任何专利技术）
3. 可扩展性 (通过扩展机制 tls_ext可以添加功能，有大量的新功能，都是通过扩展添加的)
4. 高效率 （通过session cache，恰当部署cache之后，tls的效率很高）

请认准这几个目标，在后文中，会逐一实现。

3. TLS的历史

- 1995: SSL 2.0, 由Netscape提出，这个版本由于设计缺陷，并不安全，很快被发现有严重漏洞，已经废弃。
- 1996: SSL 3.0. 写成RFC，开始流行。目前(2015年)已经不安全，必须禁用。
- 1999: TLS 1.0. 互联网标准化组织ISOC接替NetScape公司，发布了SSL的升级版TLS 1.0版.
- 2006: TLS 1.1. 作为 RFC 4346 发布。主要fix了CBC模式相关的如BEAST攻击等漏洞
- 2008: TLS 1.2. 作为RFC 5246 发布 。增进安全性。目前(2015年)应该主要部署的版本，请确保你使用的是这个版本
- 2015之后: TLS 1.3，还在制订中，支持0-rtt，大幅增进安全性，砍掉了aead之外的加密方式

由于SSL的2个版本都已经退出历史舞台了，所以本文后面只用TLS这个名字。 读者应该明白，一般所说的SSL就是TLS。

二. TLS协议的原理

1. 自顶向下，分层抽象

构建软件的常用方式是分层，把问题域抽象为多层，每一层的概念定义为一组原语，上一层利用下一层的组件构造实现，并被上一层使用，层层叠叠即成软件。

- 例如在编程语言领域中，汇编语言为一层，在汇编上面是C/C++等静态编译语言，C/C++之上是python/php/lua等动态类型脚本语言层，之上常常还会构造领域特定的DSL
- 在网络架构中，以太网是一层，其上是ip协议的网络层，ip之上是tcp等传输层，tcp之上是http等

应用层

密码学通信协议也是分层构造得到。大致可以这么分层：

1. 最底层是基础算法原语的实现，例如: aes , rsa， md5, sha256，ecdsa, ecdh 等（举的例子都是目前的主流选择，下同）
2. 其上是选定参数后，符合密码学里标准分类的算法，包括块加密算法，签名算法，非对称加密算法，MAC算法等，例如： aes-128-cbc-pkcs7，rsaes-oaep ，rsassa-pkcs1-v1_5, hmac-sha256，ecdsa-p256，curve25519 等
3. 再其上，是把多种标准算法组合而成的半成品组件，例如：对称传输组件例如 aes-128-cbc + hmac-sha256，aes-128-gcm，认证密钥协商算法: rsassa-OAEP + ecdh-secp256r1，数字信封：rsaes-oaep + aes-cbc-128 + hmac-sha256 ，文件密码加密存储组件：pbkdf2+aes-128-cbc-hmac-sha256，密钥扩展算法 PRF-sha256 等
4. 再其上，是用各种组件拼装而成的各种成品密码学协议/软件，例如：tls协议，ssh协议，srp协议，gnupg文件格式，iMessage协议，bitcoin协议等等

第1层，一般程序员都有所了解，例如rsa，简直路人皆知; md5 被广泛使用(当然，也有广泛的误用) 第2层，各种莫名其妙的参数，一般很让程序员摸不着头脑，需要深入学习才能理清。 第3层，很多程序员自己造的轮子，往往说白了就是想重复实现第3层的某个组件而已。 第4层，正确地理解，使用，部署这类成熟的开放协议，并不是那么容易。很多的误用来源于不理解，需要密码学背景知识，才能搞懂是什么，为什么，怎么用。

最难的是**理论联系实际**。面对一个一团乱麻的实际业务问题，最难的是从中抽象分析出其本质密码学问题，然后用密码学概念体系给业务建模。在分析建模过程中，要求必须有严密的，体系化的思考方式。不体系化的思考方式会导致疏漏，或者误用。

第2层中，密码学算法，常见的有下面几类：

1. 块加密算法 block cipher: AES, Serpent, 等
2. 流加密算法 stream cipher: RC4，ChaCha20 等
3. Hash函数 hash funtion:MD5，sha1，sha256，sha512 , ripemd 160，poly1305 等
4. 消息验证码函数 message authentication code: HMAC-sha256，AEAD 等
5. 密钥交换 key exchange: DH，ECDH，RSA，PFS方式的（DHE，ECDHE）等
6. 公钥加密 public-key encryption: RSA，rabin-williams 等
7. 数字签名算法 signature algorithm:RSA，DSA，ECDSA (secp256r1 , ed25519) 等
8. 密码衍生函数 key derivation function: TLS-12-PRF(SHA-256) , bcrypto，scrypto，pbkdf2 等
9. 随机数生成器 random number generators: /dev/urandom 等

每个类别里面的都有几个算法不断竞争，优胜劣汰，近几十年不断有老的算法被攻破被淘汰，新的算法被提出被推广。这一块话题广，水很深，内容多，陷阱也多，后续byron会翻译整理一系列文章，分享一下每一类里面个人收集的资料。 在此推荐一下 开源电子书crypto101，讲的很透彻，而且很易读)

设计一个加密通信协议的过程，就是自顶向下，逐步细化，挑选各类组件，拼装成完整协议的过程

3. TLS CipherSuite

从上述分层的角度看，TLS大致是由3个组件拼成的：

- **1.对称加密传输组件**，例如aes-128-gcm(这几个例子都是当前2015年最主流的选择);
- **2.认证密钥协商组件**，例如rsa-ecdhe;
- **3.密钥扩展组件**，例如TLS-PRF-sha256

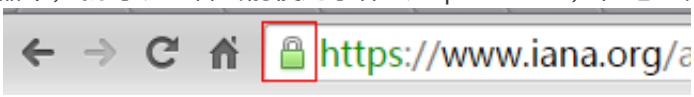
这些组件可以再拆分为5类算法，在TLS中，这5类算法组合在一起，称为一个CipherSuite：authentication (认证算法) encryption (加密算法) message authentication code (消息认证码算法 简称MAC) key exchange (密钥交换算法) key derivation function （密钥衍生算法)

TLS协议设计之初就考虑到了这每一类算法的演变，所以没有定死算法，而是设计了一个算法协商过程，来允许加入新的算法(简直是软件可扩展性设计的典范！)，协商出的一个算法组合即一个CipherSuite TLS CipherSuite 在 iana 集中注册，每一个CipherSuite分配有 一个2字节的数字用来标识，可以在 iana的注册页面 查看

iana注册页面截图：

0xC0, 0x2F	TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256	Y	[RFC5289]
0xC0, 0x30	TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384	Y	[RFC5289]
0xC0, 0x31	TLS_ECDH_RSA_WITH_AES_128_GCM_SHA256	Y	[RFC5289]
0xC0, 0x32	TLS_ECDH_RSA_WITH_AES_256_GCM_SHA384	Y	[RFC5289]

在浏览器中，就可以查看当前使用了什么 CipherSuite，在地址栏上，点击一个小锁的标志，就可以看到了。



服务器端支持的CipherSuite列表，如果是用的openssl，可以用 openssl ciphers -V | column -t 命令查看，输出如：

```
0xC0,0x2F  -  ECDHE-RSA-AES128-GCM-SHA256    TLSv1.2  Kx=ECDH  Au=RSA    Enc=AESGCM(128)  Mac=AEAD
0xC0,0x2B  -  ECDHE-ECDSA-AES128-GCM-SHA256  TLSv1.2  Kx=ECDH  Au=ECDSA  Enc=AESGCM(128)  Mac=AEAD
0xC0,0x27  -  ECDHE-RSA-AES128-SHA256        TLSv1.2  Kx=ECDH  Au=RSA    Enc=AES(128)     Mac=SHA256
0xC0,0x23  -  ECDHE-ECDSA-AES128-SHA256      TLSv1.2  Kx=ECDH  Au=ECDSA  Enc=AES(128)     Mac=SHA256
```

例如其中这一行(这个是目前的主流配置):

```
0xC0,0x2F - ECDHE-RSA-AES128-GCM-SHA256 TLSv1.2 Kx=ECDH Au=RSA Enc=AESGCM(128) Mac=AEAD
```

表示： 名字为`ECDHE-RSA-AES128-GCM-SHA256`的CipherSuite ，用于 TLSv1.2版本，使用 ECDHE 做密钥交换，使用RSA做认证，使用AES-128-gcm做加密算法，MAC由于gcm作为一种aead模式并不需要，所以显示为aead，使用SHA256做PRF算法。

可以参考 man 1 ciphers

要注意的是，由于历史兼容原因，tls标准，和openssl的tls实现中，有一些极度不安全的CipherSuite，一定要禁用，比如：

- EXP , EXPORT

  一定要禁用。EXPORT表示上世纪美国出口限制弱化过的算法，早已经被攻破，TLS的FREAK 攻击就是利用了这类坑爹的算法。 eNULL, NULL

  一定要禁用。NULL表示不加密！默认是禁用的。 aNULL： 一定要禁用。表示不做认证(authentication) ，也就是说可以随意做中间人攻击。

- ADH

  一定要禁用。表示不做认证的 DH 密钥协商。

上面是举个例子，读者不要自己去研究怎么配置，这太容易搞错。 请按照mozilla官方给出的这个权威文档，复制粘贴就好了。

CipherSuite的更多解释，配置方法等，可以参考byron之前写的一篇文章 SSL/TLS CipherSuite 介绍

4. 协议分层

TLS是用来做加密数据传输的，因此它的主体当然是一个对称加密传输组件。为了给这个组件生成双方共享的密钥，因此就需要先搞一个认证密钥协商组件，故，TLS协议自然分为：

1. 做对称加密传输的record协议 ，the record protocol
2. 做认证密钥协商的handshake协议，the handshake protocol

还有3个很简单的辅助协议：

1. changecipher spec 协议，the changecipher spec protocol, 用来通知对端从handshake切换到record协议(有点冗余，在TLS1.3里面已经被删掉了)
2. alert协议，the alert protocol, 用来通知各种返回码，
3. application data协议， The application data protocol，就是把http，smtp等的数据流传入record层做处理并传输。

这种 **认证密钥协商 + 对称加密传输** 的结构，是**绝大多数加密通信协议的通用结构**，在后文的更多协议案例中，我们可以看到该结构一再出现。

这5个协议中： record协议在tcp流上提供分包， 图片来自网络：

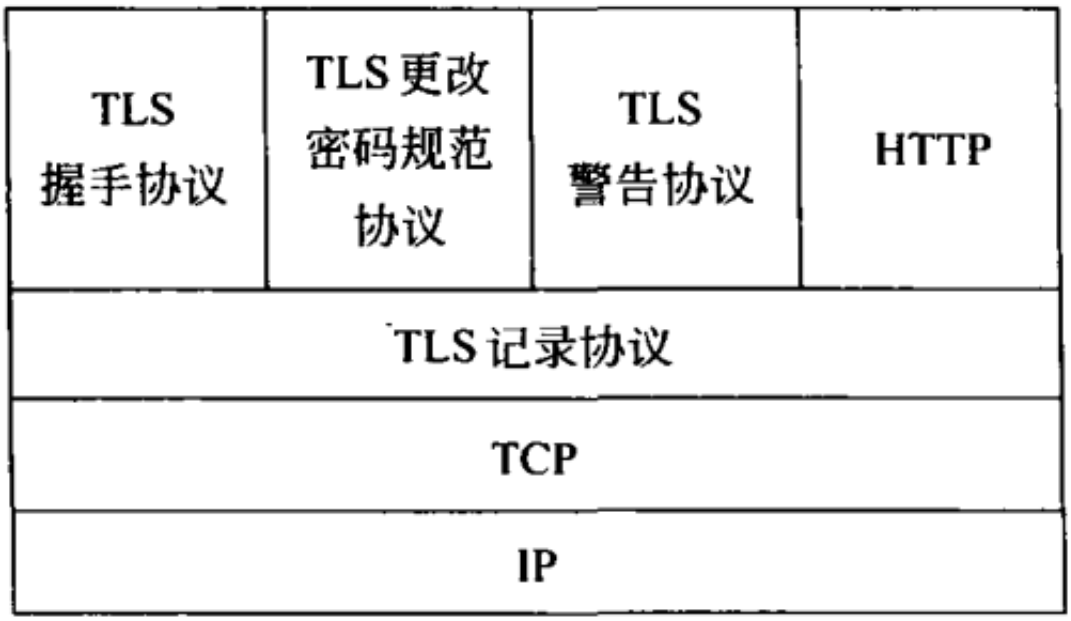


图 13.19　TLS 协议栈

其它的: handshake protocol, alert protocol, changeCipherSpec protocol, application data protocol都封装在record protocol的包里，然后在tcp上传输（此处以tcp举例，也有可能是udp，或者随便什么ipc机制等）

下文分别介绍，内容主要是翻译自 RFC5246，RFC5077，RFC4492

5. record 协议

record协议做应用数据的对称加密传输，占据一个TLS连接的绝大多数流量，因此，先看看record协议
图片来自网络:

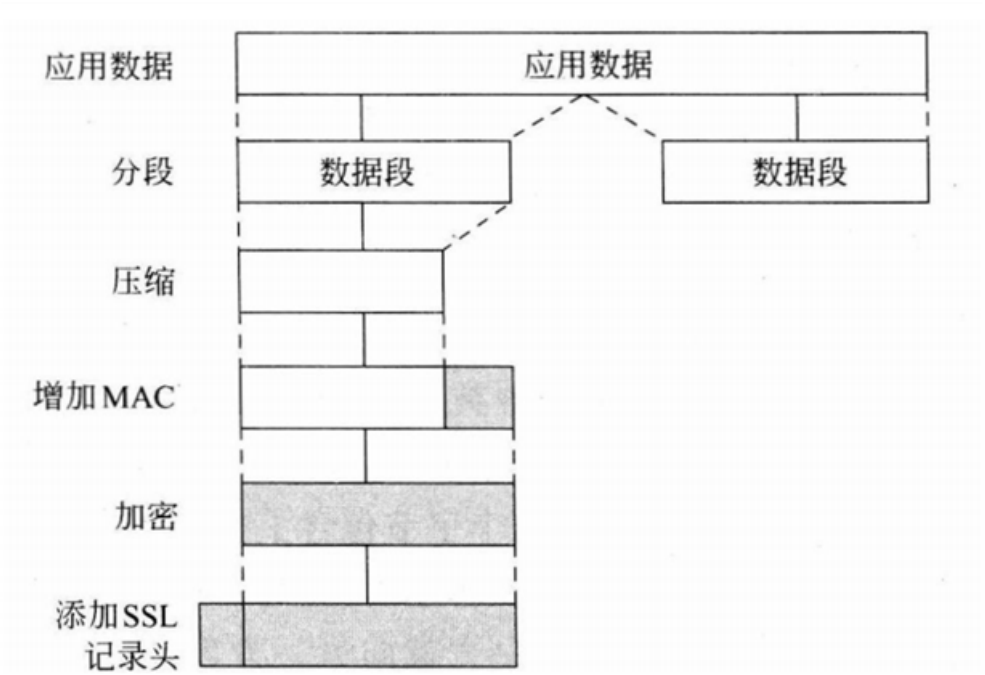


图 13.20　TLS 记录协议的操作过程

Record 协议 – 从应用层接受数据，并且做:

1. 分片，逆向是重组
2. 生成序列号，为每个数据块生成唯一编号，防止被重放或被重排序
3. 压缩，可选步骤，使用握手协议协商出的压缩算法做压缩
4. 加密，使用握手协议协商出来的key做加密/解密
5. 算HMAC，对数据计算HMAC，并且验证收到的数据包的HMAC正确性
6. 发给tcp/ip，把数据发送给 TCP/IP 做传输(或其它ipc机制)。

1. SecurityParameters

record层的上述处理，完全依据下面这个SecurityParameters里面的参数进行：

```
struct { ConnectionEnd entity; PRFAlgorithm prf_algorithm;
BulkCipherAlgorithm bulk_cipher_algorithm; CipherType cipher_type;
uint8 enc_key_length; uint8 block_length; uint8 fixed_iv_length; uint8
record_iv_length; MACAlgorithm mac_algorithm; uint8 mac_length; uint8
mac_key_length; CompressionMethod compression_algorithm; opaque
master_secret[48]; opaque client_random[32]; opaque server_random[32];
} SecurityParameters;
```

record 层使用上面的SecurityParameters生成下面的6个参数（不是所有的CipherSuite都需要全部6个，如果不需要，那就是空）：

```
client write MAC key server write MAC key client write encryption key
server write encryption key client write IV server write IV
```

当handshake完成，上述6个参数生成完成之后，就可以建立连接状态，连接状态除了上面的SecurityParameters，还有下面几个参数，并且随着数据的发送/接收，更新下面的参数：

- compression state : 当前压缩算法的状态。
- cipher state : 加密算法的当前状态，对块加密算法比如aes，包含密码预处理生成的轮密钥(感谢温博士指出) “round key”，还有IV等；对于流加密，包含能让流加密持续进行加解密的状态信息
- sequence number : 每个连接状态都包含一个sequence number，并且读和写状态有不同的sequence number。当连接开始传输数据时，sequence number必须置为0. sequence number是uint64类型的，并且不得超过 264−1264−1 。s. Sequence number不得回绕。如果一个TLS实现无法避开回绕一个sequence number，必须进行重协商。sequence number在每个record被发送时都增加1。并且传输的第1个Record必须使用0作为sequence number。

此处有几个问题值得思考：

(1). 为什么MAC key , encryption key, IV 要分别不同?

在密码学中，对称加密算法一般需要encryption key，IV两个参数，MAC算法需要MAC key参数，因此这3个key用于不同的用途。 当然，不是所有的算法都一定会用到这3个参数，例如新的aead型算法，就不需要MAC key。

(2). 为什么client和server要使用不同的key 如果TLS的双方使用相同的key，那么当使用stream cipher加密应用数据的时候，stream cipher的字节流在两个方向是一样的，如果攻击者知道TLS数据流一个方向的部分明文（比如协议里面的固定值），那么对2个方向的密文做一下xor，就能得到另一个方向对应部分的明文了。

还有，当使用 aead 比如 aes-gcm 做加密的时候，aead标准严格要求，**绝对不能用相同的 key+nonce 加密不同的明文**，故如果TLS双方使用相同的key，又从相同的数字开始给nonce递增，那就不符合规定，会直接导致 aes-gcm 被攻破。

参考: http://crypto.stackexchange.com/questions/2878/separate-read-and-write-keys-in-tls-key-material

2. record层分段

如上图所示，对要发送的数据流，首先分段，分段成如下格式：

```
struct { uint8 major; uint8 minor; } ProtocolVersion; enum {
change_cipher_spec(20), alert(21), handshake(22),
application_data(23), (255) } ContentType; struct { ContentType
type; ProtocolVersion version; uint16 length; opaque
fragment[TLSPlaintext.length]; } TLSPlaintext;
```

- version字段 : ，定义当前协商出来的TLS协议版本，例如 TLS 1.2 version 是 { 3, 3 }
- length字段 : 即长度，tls协议规定length必须小于 214214，一般我们不希望length过长，因为解密方需要收完整个record，才能解密，length过长会导致解密方需要等待更多的rtt，增大latency，破坏用户体验，参考 Web性能权威指南 TLS那一章。
- type字段 : ，用来标识当前record是4种协议中的哪一种，
- record压缩

  TLS协议定义了可选的压缩，但是，由于压缩导致了 2012 年被爆出CRIME攻击，BREACH攻击，所以在实际部署中，**一定要禁用压缩**。 http://www.unclekevin.org/?p=640 http://www.freebuf.com/articles/web/5636.html

3. record层的密码学保护

record层的密码学保护:

经过处理后的包格式定义如下：

```
struct { ContentType type; ProtocolVersion version; uint16 length;
select (SecurityParameters.cipher_type) { case stream:
GenericStreamCipher; case block: GenericBlockCipher; case aead:
GenericAEADCipher; } fragment; } TLSCiphertext;
```

TLS协议设计目标中的 1.保密(encryption) 2.完整性(authentication) ， 和防重放就在这里实现。 实现方式有3类：

1. Block Cipher (CBC mode of operation) + HMAC：例如 aes-128-cbc+hmac-sha256
2. Stream Cipher (RC4) + HMAC
3. Authenticated-Encryption using block cipher (GCM/CCM 模式)：例如 aes-128-gcm

1.Block Cipher+HMAC 和 2.Stream Cipher + HMAC 的各类算法目前（2015年）都已经爆出各种漏洞(后文解释)，目前最可靠的是 3.Authenticated-Encryption 类的算法，主要就是aes-gcm，下一代的TLS v1.3干脆只保留了3.Authenticated-Encryption，把1和2直接禁止了(所以。。。你真的还要继续用aes-cbc吗？)。

GCM模式是AEAD的，所以不需要MAC算法。 GCM模式是AEAD的一种，AEAD 的 作用类似于 Encrypt-then-HMAC ，例如 Sha256 + Salt + AES + IV

此处需要介绍一个**陷阱**。 在密码学历史上，出现过3种加密和认证的组合方式：

1. Encrypt-and-MAC
2. MAC-then-Encrypt
3. Encrypt-then-MAC

在TLS协议初定的那个年代，人们还没意识到这3种组合方式的安全性有什么差别，所以TLS协议规定使用 2.MAC-then-Encrypt，即先计算MAC，然后把 “明文+MAC” 再加密(块加密或者流加密)的方式，做流加密+MAC，和块加密+MAC。 但是，悲剧的是，近些年，人们发现 MAC-then-Encrypt 这种结构导致了 很容易构造padding oracle 相关的攻击，例如这在TLS中，间接形成被攻击者利用，这间接导致了 BEAST 攻击 , Lucky 13攻击 (CVE-2013-0169), 和 POODLE 攻击 (CVE-2014-3566).

目前因此，学术界已经一致同意： **Encrypt-then-MAC 才是最安全的!** tls使用的是 MAC-then-Encrypt 的模式，导致了一些问题。 具体比较，参见： http://cseweb.ucsd.edu/~mihir/papers/oem.pdf https://www.iacr.org/archive/crypto2001/21390309.pdf http://crypto.stackexchange.com/questions/202/should-we-mac-then-encrypt-or-encrypt-then-mac https://news.ycombinator.com/item?id=4779015 http://tozny.com/blog/encrypting-strings-in-android-lets-make-better-mistakes/

鉴于这个陷阱如此险恶，学术界有人就提出了，干脆把Encrypt和MAC直接集成为一个算法，在算法内部解决好安全问题，不再让码农选择，避免众码农再被这个陷阱坑害，这就是AEAD（Authenticated-Encryption With Additional data）类的算法，GCM模式就是AEAD最重要的一种。

4. record层的密码学保护–MAC

TLS record 层 MAC的计算方法：

```
MAC(MAC_write_key, seq_num + TLSCompressed.type +
TLSCompressed.version + TLSCompressed.length +
TLSCompressed.fragment);
```

其中的seq_num是当前record的 sequence number，每条record都会++， 可以看到把 seq_num，以及record header里面的几个字段也算进来了，这样**解决了防重放问题**，并且保证record的任何字段都不能被篡改。

算完MAC，格式如下：

```
stream-ciphered struct { opaque content[TLSCompressed.length]; opaque
MAC[SecurityParameters.mac_length]; } GenericStreamCipher;
```

然后根据SecurityParameters.cipher_type，选择对应的对称加密算法进行加密，分类解说如下：

5. record层的密码学保护–stream cipher

stream cipher: 算stream cipher，stream cipher的状态在连续的record之间会复用。 stream cipher的主力是RC4，但是目前RC4已经爆出多个漏洞，所以实际中基本不使用流加密没法，详情请见：

https://tools.ietf.org/html/rfc7457#section-2.5

[FreeBuf] RC4加密已不再安全，破解效率极高

http://www.imperva.com/docs/HII_Attacking_SSL_when_using_RC4.pdf

6. record层的密码学保护– CBC block cipher

CBC模式块加密 TLS目前靠得住的的块加密cipher也不多，基本就是AES（最靠谱，最主流），Camellia，SEED，（3DES，IDEA之类已经显得老旧，DES请禁用），加密完的格式如下：

```
struct { opaque IV[SecurityParameters.record_iv_length]; block-ciphered
struct { opaque content[TLSCompressed.length]; opaque
MAC[SecurityParameters.mac_length]; uint8
padding[GenericBlockCipher.padding_length]; uint8 padding_length; }; }
GenericBlockCipher;
```

这个值得说道说道，因为我们码农平常在业界还能看到很多用AES-CBC的地方，其中的几个参数：

- IV

  ： 要求**必须用密码学安全的伪随机数生成器(CSPRNG)**生成，并且必须是不可预测的，在Linux下，就是用用/dev/urandom，或者用 openssl 库的 RAND_bytes()。注意：TLS 在 1.1版本之前，没有这个IV字段，前一个record的最后一个block被当成下一个record的IV来用，然后粗大事了，这导致了 BEAST攻击。 所以，TLS1.2改成了这样。 (**还在使用CBC的各位，建议关注一下自己的IV字段是怎么生成出来的。如果要用，做好和TLS1.2的做法保持一致**)。其中 SecurityParameters.record_iv_length 一定等于 SecurityParameters.block_size. 例如 AES-256-CBC的 IV 一定是16字节长的，因为AES 128/192/256 的block size都是16字节。padding

使用CBC常用的PKCS 7 padding（在block size=16字节这种情况下，和pkcs 5的算法是一回事，java代码里面就可以这么用这个case里，和pkcs 5的结果是一样的）

- padding_length

  就是PKCS 7 padding的最后一个字节

注意2个险恶的陷阱：

1. 实现的代码必须在收到全部明文之后才能传输密文，否则可能会有BEAST攻击
2. 实现上，根据MAC计算的时间，可能进行时间侧通道攻击，因此必须确保–**运行时间和padding是否正确无关**。

7. record层的密码学保护– AEAD cipher

AEAD 到了我们重点关注的AEAD，AEAD是新兴的主流加密模式，是目前最重要的模式，其中主流的AEAD模式是 aes-gcm-128/aes-gcm-256/chacha20-poly1305

AEAD加密完的格式是：

```
struct { opaque nonce_explicit[SecurityParameters.record_iv_length]; aead-ciphered struct { opaque content[TLSCompressed.length]; }; } GenericAEADCipher;
```

AEAD ciphers的输入是: key，nonce, 明文,和 “additional data”. key是 client_write_key 或者 the server_write_key. 不需要使用 MAC key.

每一个AEAD算法都要指定不同的nonce构造算法，并指定 GenericAEADCipher.nonce_explicit 的长度. 在TLS 1.2中，规定很多情况下，可以按照rfc5116 section 3.2.1的技术来做。其中record_iv_length是nonce的显式部分的长度，nonce的隐式部分从key_block作为 client_write_iv和 and server_write_iv得出，并且把显式部分放在 GenericAEAEDCipher.nonce_explicit 里.

在TLS 1.3 draft中，做了更改:

1. 规定 AEAD算法的 nonce的长度规定为 max(8 bytes, N_MIN)，即如果N_MIN比8大，就用N_MIN; 如果比8小，就用8。
2. 并且规定 N_MAX小于8字节的AEAD不得用于TLS。
3. 规定TLS AEAD中每条record的nonce通过下面的方法构造出来： 64bit的sequence number的右侧填充0，直到长度达到iv_length。然后把填充过的sequence number和静态的 client_write_iv或server_write_iv （根据发送端选择）做异或(XOR)。异或完成后，得到的 iv_length 的nonce就可以做每条record的nonce用了。

   AEAD输入的明文就是 TLSCompressed.fragment (记得上面的介绍吗？AEAD是MAC和encrypt的集成，所以输入数据不需要在算MAC了).

   AEAD输入的additional_data 是:

```
1
2
```
```
additional_data = seq_num + TLSCompressed.type + TLSCompressed.version
+ TLSCompressed.length;
```

“+” 表示字符串拼接。 可以看到，此处类似上面的MAC计算，算入了seq_num来防重放，type,version,length等字段防止这些元数据被篡改。

```
1
2
```
```
AEADEncrypted = AEAD-Encrypt(write_key, nonce, plaintext,
additional_data)
```

解密+验证完整性：

```
1 2
3
```
```
TLSCompressed.fragment = AEAD-Decrypt(write_key, nonce,
AEADEncrypted, additional_data)
```

如果解密/验证完整性失败,就回复一条 fatal bad_record_mac alert 消息.

aes-gcm的iv长度，nonce长度，nonce构成等，后续再深入探讨。

8. record层的密码学保护– Key扩展

Key 扩展

TLS握手生成的master_secret只有48字节，2组encryption key, MAC key, IV加起来，长度一般都超过48，(例如 AES_256_CBC_SHA256 需要 128字节),所以，TLS里面用1个函数，来把48字节延长到需要的长度，称为PRF：

```
1
2
3
4
```
```
key_block = PRF(SecurityParameters.master_secret, "key expansion",
SecurityParameters.server_random + SecurityParameters.client_random);
```

然后，key_block像下面这样被分割：

```
client_write_MAC_key[SecurityParameters.mac_key_length]
server_write_MAC_key[SecurityParameters.mac_key_length]
client_write_key[SecurityParameters.enc_key_length]
server_write_key[SecurityParameters.enc_key_length]
client_write_IV[SecurityParameters.fixed_iv_length]
server_write_IV[SecurityParameters.fixed_iv_length]
```

TLS使用HMAC结构，和在CipherSuite中指定的hash函数（安全等级起码是SHA256的水平） 来构造PRF,

首先定义P_hash，把(secret,seed)扩展成无限长的字节流：

```
P_hash(secret, seed) = HMAC_hash(secret, A(1) + seed) +
HMAC_hash(secret, A(2) + seed) + HMAC_hash(secret, A(3) + seed) + ...
```

其中"+"表示字符串拼接。 A() 定义为:

```
A(0) = seed A(i) = HMAC_hash(secret, A(i-1))
```

TLS的 PRF 就是把 P_hash 应用在secret上:

```
PRF(secret, label, seed) = P_<hash>(secret, label + seed)
```

其中 label 是一个协议规定的，固定的 ASCII string.

要注意的是，TLS 1.3里面已经废弃了这种方式，改为使用**更靠谱的 HKDF**，HKDF 也是 html5的WebCryptoAPI的标准算法之一。

5. handshake 协议

handshake protocol重要而繁琐。

TLS 1.3对握手做了大修改，下面先讲TLS 1.2，讲完再介绍一下分析TLS 1.3.

1.handshake的总体流程

handshake protocol用于产生给record protocol使用的SecurityParameters。 在handshake中:

- 客户端和服务器端协商TLS协议版本号和一个CipherSuite,
- 认证对端的身份（可选，一般如https是客户端认证服务器端的身份），

- 并且使用密钥协商算法生成共享的master secret。

步骤如下：

- 交换Hello消息，协商出算法，交换random值，检查session resumption.
- 交换必要的密码学参数，来允许client和server协商出premaster secret。
- 交换证书和密码学参数，让client和server做认证，证明自己的身份。
- 从premaster secret和交换的random值 ，生成出master secret。
- 把SecerityParameters提供被record层。
- 允许client和server确认对端得出了相同的SecurityParameters，并且握手过程的数据没有被攻击者篡改。

Handshake的结果是在双方建立相同的Session，Session 包含下列字段：

1. session identifier session id，用来唯一标识一个session，在session 恢复的时候，也要用到
2. peer certificate 对端的 X509v3 格式证书. 如果不需要认证对端的身份，就为空。
3. compression method 压缩算法，一般被禁用
4. cipher spec CipherSuite，如上文介绍，包含： 用于生成key的pseudorandom function (PRF) , 块加密算法例如AES, MAC算法 (例如 HMAC-SHA256). 还包括一个 mac_length字段，在后文的we握手协议介绍
5. master secret 48字节的，client和server共享密钥。
6. is resumable 一个标志位，用来标识当前session是否能被恢复。

以上字段，随后被用于生成 record层的SecurityParameters，多个连接可以通过握手协议的session恢复功能来复用同一个session。

握手协议使用 非对称加密/密钥协商/数字签名 3类算法， 因此要求读者对这3类算法概念清晰，能准确区分。 在此澄清一下,： 非对称的算法分为3类: ,

- 非对称加密，有：RSAES-PKCS1-v1_5，RSAES-OAEP ,Rabin-Williams-OAEP, Rabin-Williams-PKCS1-v1_5等
- 非对称密钥协商，有：DH，DHE，ECDH，ECDHE 等
- 非对称数字签名：RSASSA-PKCS1-v1_5，RSASSA-PSS，ECDSA，DSA，ED25519 等

另外，**非对称加密算法，可以当作密钥协商算法来用，所以 RSAES-PKCS1-v1_5，RSAES-OAEP 也可以当作密钥协商算法来用**。

插播一段 RSA：

RSA的实际工程应用，要遵循PKCS#1 标准，见 https://www.ietf.org/rfc/rfc3447

其中的 RSAES-PKCS1-v1_5 和 RSASSA-PKCS1-v1_5 是使用RSA算法的两种不同scheme（体制）。 RSAES表示 RSA Encryption schemes，即非对称加密， RSAES有：RSAES-OAEP，RSAES-PKCS1-v1_5 两种，其中RSAES-OAEP更新更安全

RSASSA表示 Signature schemes with appendix，即appendix模式(appendix和recovery的区别请参看密码学教材)的非对称数字签名算法。 RSASSA有： RSASSA-PSS, RSASSA-PKCS1-v1_5 两种， 其中RSASSA-PSS更新更安全

RSA还有一个缺陷，就是很容易被时间侧通道攻击，所以现在的RSA实现都要加 blinding ，后文有介绍。

可以看到，RSA是一种很特殊的算法，既可以当非对称加密算法使用，又可以当非对称数字签名使用。这一点很有迷惑性，其实很多用RSA的人都分不清自己用的是RSA的哪种模式。

相比之下，ECC(椭圆曲线)这一块的算法就很清晰，ECDSA只能用作数字签名，ECDH只能用作密钥交换。

分清楚 RSAES-PKCS1-v1_5 和 RSASSA-PKCS1-v1_5 有什么用涅?

PKCS#1规范解释：

> A generally good cryptographic practice is to employ a given RSA key pair in only one scheme. This avoids the risk that vulnerability in one scheme may compromise the security of the other, and may be essential to maintain provable security.

FIPS PUB 186-3 美国标准规定：

> An RSA key pair used for digital signatures shall only be used for one digital signature scheme (e.g., ANS X9.31, RSASSA-PKCS1 v1.5 or RSASSA-PSS; see Sections 5.4 and 5.5). In addition, an RSA digital signature key pair shall not be used for other purposes (e.g., key establishment).

一对密钥只做一个用途，要么用作非对称加解密，要么用作签名验证，别混着用！ 一对密钥只做一个用途，要么用作非对称加解密，要么用作签名验证，别混着用！ 一对密钥只做一个用途，要么用作非对称加解密，要么用作签名验证，别混着用！

这个要求，决定了一个协议的 PFS（前向安全性），在斯诺登曝光NSA的“今日捕获，明日破解”政策后，**越发重要**。

https://news.ycombinator.com/item?id=5942534

http://news.netcraft.com/archives/2013/06/25/ssl-intercepted-today-decrypted-tomorrow.html

https://lwn.net/Articles/572926/

https://www.eff.org/deeplinks/2014/04/why-web-needs-perfect-forward-secrecy

http://www.wired.com/2013/10/lavabit_unsealed

PFS反映到密钥协商过程中，就是：

- **不要使用RSA做密钥协商，一定只用RSA做数字签名。**
- **不要把ECDH的公钥固定内置在客户端做密钥协商**

后文可以看到这一原则在 TLS 1.3, QUIC，Apple的iMessage等协议中一再贯彻。

非对称RSA/ECC这个话题比较大了，后面有空再写文章吧，读者可以先看一下参考资料，里面有清晰的介绍。

插播结束，继续TLS。

由于设计的时候，就要考虑兼容性，而且实际历史悠久，所以TLS协议90年代曾经使用的一些算法，现在已经被破解了，例如有的被发现漏洞(rc4)，有的密钥长度过短(例如曾经美帝有出口限制，限制RSA 在512比特以下，对称加密密钥限制40比特以下，后来2005年限制被取消)，但是考虑到兼容，现在的TLS实现中，还是包含了这种已经被破解的老算法的代码。这样，如果攻击者可以干扰握手过程，诱使client和server使用这种已经被破解的算法，就会威胁TLS协议的安全，这被称为“降级攻击”。

为了在握手协议解决降级攻击的问题，TLS协议规定：client发送ClientHello消息，server必须回复ServerHello消息，否则就是fatal error，当成连接失败处理。ClientHello和ServerHello消息用于建立client和server之间的安全增强能力，ClientHello和ServerHello消息建立如下属性：

- Protocol Version
- Session ID
- Cipher Suite
- Compression Method.

另外，产生并交换两个random值 ClientHello.random 和 ServerHello.random

密钥协商使用四条： server的Certificate，ServerKeyExchange，client的Certificate，ClientKeyExchange 。TLS规定以后如果要新增密钥协商方法，可以订制这4条消息的数据格式，并且指定这4条消息的使用方法。密钥协商得出的共享密钥**必须足够长，当前定义的密钥协商算法生成的密钥长度必须大于46字节**。

在hello消息之后，server会把自己的证书在一条Certificate消息里面发给客户端(如果需要做服务器端认证的话，例如https)。 并且，如果需要的话，server会发送一条ServerKeyExchange消息，（例如如果服务器的证书只用做签名，不用做密钥交换，或者服务器没有证书）。client对server的认证完成后，server可以要求client发送client的证书，如果这是协商出来的CipherSuite允许的。下一步，server会发送ServerHelloDone消息，表示握手的hello消息部分已经结束。然后server会等待一个client的响应。如果server已经发过了CertificateRequest消息，client必须发送Certificate消息。然后发送ClientKeyExchange消息，并且这条消息的内容取决于ClientHello和ServerHello消息协商的算法。如果client发送了有签名能力的证书，就显式发送一个经过数字签名的CertificateVerify消息，来证明自己拥有证书私钥。

然后，client发送一个ChangeCipherSpec消息，并且client拷贝待定的Cipher Spec到当前的Cipher Spec。然后client立即用新算法+新key+新密钥 发送Finished消息。收到后，server发送自己的ChangeCipherSpec消息，作为响应，并且拷贝待定的Cipher Spec到当前的Cipher Spec。此时，握手就完成了，client和server可以开始交换应用层数据（如下图所示）。应用层数据不得在握手完成前发送。

引用一个来自网络的图片：

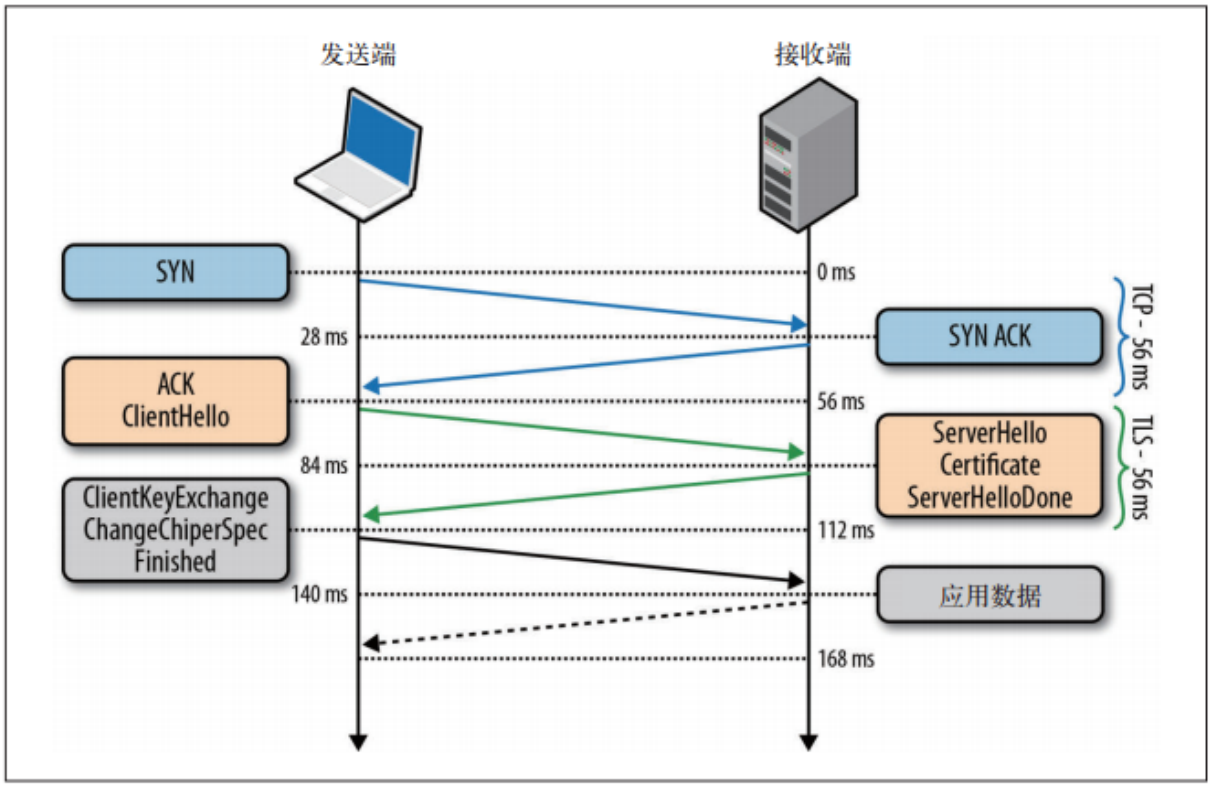


图 4-3：简短 TLS 握手协议

```
Client Server ClientHello --------> ServerHello Certificate* ServerKeyExchange* CertificateRequest* <-------- ServerHelloDone Certificate* ClientKeyExchange CertificateVerify* [ChangeCipherSpec] Finished --------> [ChangeCipherSpec] <-------- Finished Application Data <-------> Application Data Figure 1. Message flow for a full handshake * 表示可选的消息，或者根据上下文在某些情况下会发送的消息。Indicates optional or situation-dependent messages that are not always sent.
```

注：为了帮助解决管道阻塞的问题，ChangeCipherSpec是一个独立的TLS protocol content type，并不是一个握手消息。

TLS的完整握手过程，要进行RSA/ECDH/ECDSA等非对称计算，非对称计算是很慢的。关于非对称的性能： 例如在2015年的服务器cpu： Intel(R) Xeon(R) CPU E3-1230 V2 @ 3.30GHz 上， 使用如下命令测试：

```
openssl speed rsa2048 openssl speed ecdsap256 openssl speed ecdhp256
openssl speed aes-128-cbc openssl speed -evp aes-128-cbc
```

结果如下表：

算法	性能	性能
RSA-2048	私钥运算 723.7 次/秒	公钥运算 23505.8 次/秒
256 bit ecdsa (nistp256)	签名 8628.4 次/秒	验证 2217.0 次/秒
256 bit ecdh (nistp256)	ECDH协商 2807.8 次/秒	
aes-128-cbc	加密 121531.39 K/秒	
aes-128-cbc 使用aesni硬件加速	加密 683682.13 K/秒	

注：非对称的单位是 次/秒，这是由于非对称一般只用于处理一个block， 对称的单位是 K/秒，因为对称一般用于处理大量数据流，所以单位和流量一样。 可以给非对称的 次/秒 乘以 block size ，就可以和对称做比较了。例如rsa-2048，723.7*2048/8/1024=185.2672 K/秒， 故 **RSA-2048 私钥运算性能 是 aes-128-cbc 的 1.5/10001.5/1000。是aesni的 2.6/100002.6/10000。**

如上，性能数据惨不忍睹， 简直不能忍！！！

有鉴于此，TLS从设计之初，就采用了万能手段–加cache，有2种cache手段：session id，和session ticket。把握手的结果直接cache起来，绕过握手运算。

当client和server决定恢复一个之前的session，或复用一个已有的session时(可以不用协商一个新的SecurityParameters)，消息流程如下：

客户端使用要被恢复的session，发送一个ClientHello，把Session ID包含在其中。server在自己的session cache中，查找客户端发来的Session ID，如果找到，sever把找到的session 状态恢复到当前连接，然后发送一个ServerHello，在ServerHello中把Session ID带回去。然后，client和server都必须ChangeCipherSpec消息，并紧跟着发送Finished消息。这几步完成后，client和server 开始交换应用层数据（如下图所示）。如果server在session cache中没有找到Session ID，那server就生成一个新的session ID在ServerHello里给客户端，并且client和server进行完整的握手。

流程图如下：

```
Client Server ClientHello --------> ServerHello [ChangeCipherSpec]
<-------- Finished [ChangeCipherSpec] Finished --------> Application
Data <-------> Application Data Figure 2. Message flow for an
abbreviated handshake
```

3. handshake 协议外层结构

从消息格式来看，TLS Handshake Protocol 在 TLS Record Protocol 的上层. 这个协议用于协商一个session的安全参数。 Handshake 消息（例如ClientHello，ServerHello等） 被包装进TLSPlaintext结构里面，传入TLS record层，根据当前session 状态做处理，然后传输。

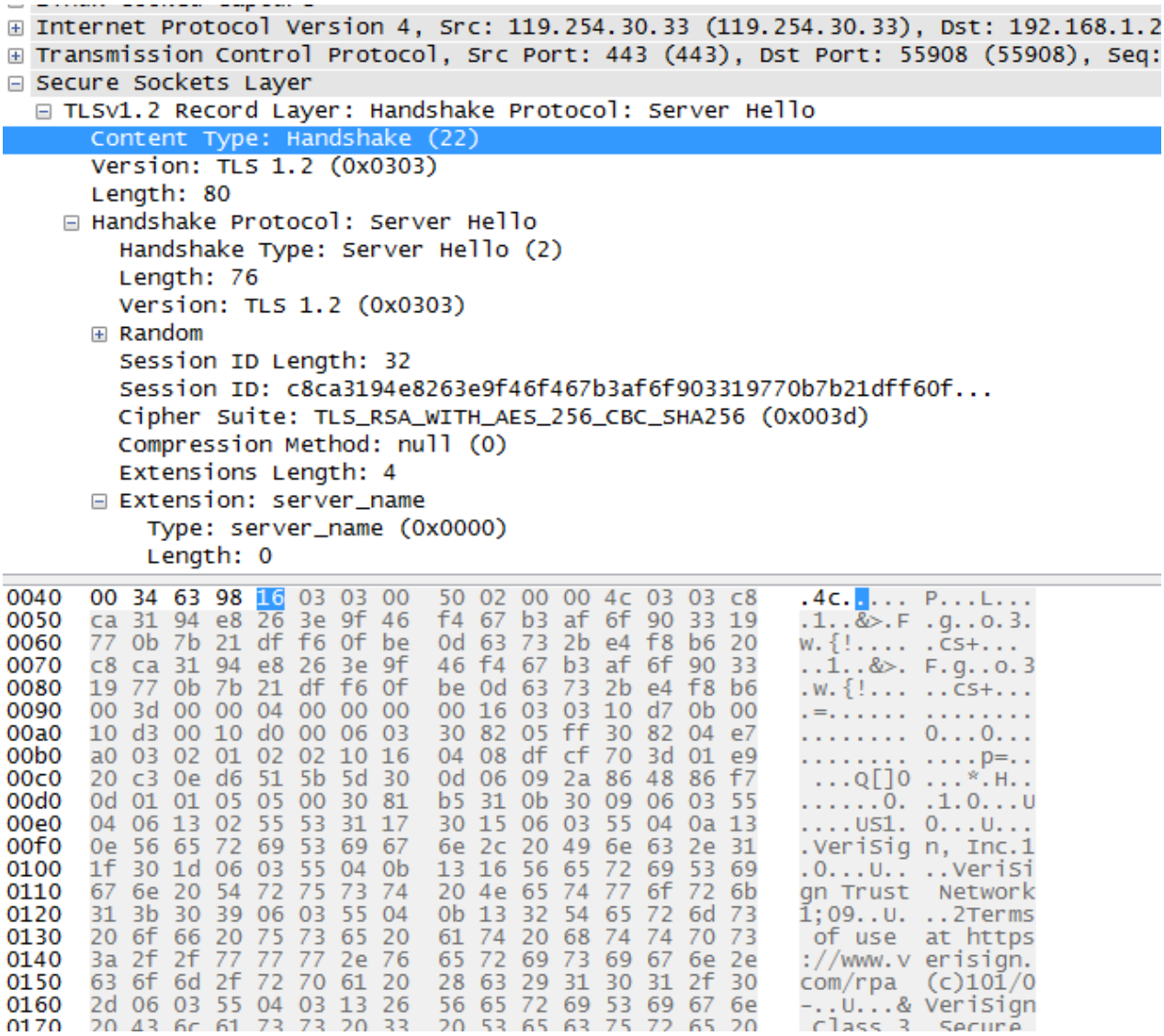
如下：

```
enum { hello_request(0), client_hello(1), server_hello(2),
certificate(11), server_key_exchange (12), certificate_request(13),
server_hello_done(14), certificate_verify(15), client_key_exchange(16),
finished(20), (255) } HandshakeType; struct { HandshakeType msg_type;
/* handshake type */ uint24 length; /* bytes in message */ select
(HandshakeType) { case hello_request: HelloRequest; case client_hello:
ClientHello; case server_hello: ServerHello; case certificate:
Certificate; case server_key_exchange: ServerKeyExchange; case
certificate_request: CertificateRequest; case server_hello_done:
ServerHelloDone; case certificate_verify: CertificateVerify; case
client_key_exchange: ClientKeyExchange; case finished: Finished; case
session_ticket: NewSessionTicket; /* NEW */ } body; } Handshake;
```

TLS协议规定，handshake 协议的消息必须按照规定的顺序发，收到不按顺序来的消息，当成fatal error处理。也就是说，TLS协议可以当成状态机来建模编码。

下面按照消息发送必须遵循的顺序，逐个解释每一条握手消息。

handshake协议的外层字段，见这个抓包：



4. handshake – ClientHello，ServerHello，HelloRequest

Hello消息有3个：ClientHello, ServerHello，HellloRequest 逐个说明：

4.1 Client Hello

当客户端第一次连接到服务器时，第一条message必须发送ClientHello。 另外，rfc里规定，如果客户端和服务器支持重协商，在客户端收到服务器发来的HelloRequest后，也可以回一条ClientHello，在一条已经建立的连接上开始重协商。(重协商是个很少用到的特性。)

消息结构：

```
struct { uint32 gmt_unix_time; opaque random_bytes[28]; } Random;
opaque SessionID<0..32>; uint8 CipherSuite[2]; enum { null(0),
(255) } CompressionMethod; struct { ProtocolVersion
client_version; Random random; SessionID session_id; CipherSuite
cipher_suites<2..2^16-2>; CompressionMethod
compression_methods<1..2^8-1>; select (extensions_present) { case
false: struct {}; case true: Extension extensions<0..2^16-1>; }; }
ClientHello;
```

Random 其中： gmt_unix_time 是 unix epoch时间戳。 random_bytes 是 28字节的，用密码学安全随机数生成器 生成出来的随机数。

密码学安全的随机数生成，这是个很大的话题，也是一个大陷阱，目前最好的做法就是用 /dev/urandom，或者openssl库的 RAND_bytes()

历史上，恰好就在SSL的random_bytes这个字段，NetScape浏览器早期版本被爆出过随机数生成器漏洞。 被爆菊的随机数生成器使用 pid + 时间戳 来初始化一个seed，并用MD5(seed)得出结果。 见 http://www.cs.berkeley.edu/~daw/papers/ddj-netscape.html， 建议读者检查一下自己的随机数生成器。

- **client_version**

  客户端支持的最高版本号。

- **random**

  客户端生成的random。

ClientHello.session_id 唯一标识一个session，用来做session cache。如果为空，表示不做复用，要求服务器生成新的session。 session_id的来源有：

1. **之前的协商好的连接的session_id**
2. **当前连接的session_id**
3. **当前也在使用中的另一条连接的session_id**

其中第三种允许不做重新握手，就同时建立多条独立的安全连接。这些独立的连接可能顺序创建，也可以同时创建。一个SessionID当握手协商的Finished消息完成后，就合法可用了。存活直到太旧被移除，或者session 关联的某个连接发生fatal error。SessionID的内容由服务器端生成。

注：由于SessionID的传输是不加密，不做MAC保护的，服务器不允许把私密信息发在里面，不能允许伪造的SessionID在服务器造成安全问题。（握手过程中的数据，整体是受Finished消息的保护的）

ClientHello.cipher_suites字段，包含了客户端支持的CipherSuite的列表，按照客户端希望的优先级排序，每个CipherSuite有2个字节，每个CipherSuite由：一个密钥交换算法，一个大量数据加密算法(需要制定key length参数)，一个MAC算法，一个PRF 构成。服务器会从客户端发过来的列表中选择一个；如果没有可以接受的选择，就返回一个 handshake failure 的 alert，并关闭连接。如果列表包含服务器不认识，不支持，或者禁用的CipherSuite，服务器必须忽略。 如果SessionID不为空，则cipher_suites里面起码要包含客户端cache的session里面的那个CipherSuite

compression_methods，类似地，ClientHello里面包含压缩算法的列表，按照客户端优先级排序。当然，如前介绍，服务器一般禁用TLS的压缩。

compression_methods 后面可以跟一组扩展(extensions)， extensions都是可选的，比较有用的扩展如： SNI, session ticket，ALPN，OCSP 等，后文介绍。

客户端发送了ClientHello后，服务器端必须回复ServerHello消息，回复其他消息都会导致 fatal error 关闭连接。

4.2 Server Hello

当收到客户端发来的ClientHello后，正常处理完后，服务器必须回复ServerHello。

消息结构：

```
struct { ProtocolVersion server_version; Random random; SessionID session_id; CipherSuite cipher_suite; CompressionMethod compression_method; select (extensions_present) { case false: struct {}; case true: Extension extensions<0..2^16-1>; }; } ServerHello;
```

- server_version

  服务器选择 ClientHello.client_version 和 服务器支持的版本号 中的最小的。

- random

  服务器生成的random，必须确保和客户端生成的random没有关联。

- session_id

  服务器为本连接分配的SessionID。如果ClientHello.session_id不为空，服务器会在自己的本地做查找。

- 如果找到了匹配，并且服务器决定复用找到的session建立连接，服务器应该把ClientHello.session_id同样的 session id填入ServerHello.session_id，这表示恢复了一个session，并且双方会立即发送Finished消息。
- 否则，回复一个和ClientHello.session_id不同的Serverhello.session_id，来标识新session。服务器可以回复一个空的session_id，来告诉客户端这个session不要cache，不能恢复。 如果一个session 被恢复了，那必须恢复成之前协商的session里面的 CipherSuite。要注意的是，并不要求服务器一定要恢复session， 服务器可以不做恢复。

在实践中，session cache在服务器端要求key-value形式的存储，如果tls服务器不止一台的话，就有一个存储怎么共享的问题，要么存储同步到所有TLS服务器的内存里，要么专门搞服务来支持存储，并使用rpc访问， 无论如何，都是很麻烦的事情，相比之下，后文要介绍的session ticket就简单多了，所以一般优先使用session ticket。

- cipher_suite

  服务器选定的一个CipherSuite。如果是恢复的session，那就是session里的CipherSuite。

- compression_method

  跟上面类似。

- extensions

  扩展列表。要注意的是，ServerHello.extensions 必须是 ClientHello.extensions的子集。

4.3 Hello Extensions

The extension 的格式是:

```
1 2 3
4 5 6
7 8 9
```

```
struct { ExtensionType extension_type; opaque
extension_data<0..2^16-1>; } Extension; enum {
signature_algorithms(13), (65535) } ExtensionType;
```

其中:

- “extension_type” 标识是哪一个扩展类型。
- “extension_data” 一坨二进制的buffer，扩展的数据体，各个扩展自己做解析。

extension_type 只能出现一次，ExtensionType之间不指定顺序。

extensions 可能在新连接创建时被发送，也可能在要求session恢复的时候被发送。所以各个extension都需要规定自己再完整握手和session恢复情况下的行为。 这些情况比较琐碎而微妙，具体案例要具体分析。

4.4 Hello Request

服务器任何时候都可以发送 HelloRequest 消息。

HelloRequest的意思是，客户端应该开始协商过程。客户端应该在方便的时候发送ClientHello。服务器不应该在客户端刚创建好连接后，就发送HelloRequest，此时应该让客户端发送ClientHello。

客户端收到这个消息后，可以直接忽略这条消息。 服务器发现客户端没有响应HelloRequest后，可以发送fatal error alert。

消息结构:

```
1
```

```
struct { } HelloRequest;
```

HelloRequest不包含在握手消息的hash计算范围内。

5. handshake – Server Certificate

当服务器确定了CipherSuite后，根据CipherSuite里面的认证算法，如果需要发送证书给客户端，那么就发送 Server Certificate消息给客户端。Server Certificate总是在ServerHello之后立即发送，所以在同一个RTT里。

Server Certificate里面包含了服务器的证书链。

消息结构：

```
1 2 3 4 5
```

```
opaque ASN.1Cert<1..2^24-1>; struct { ASN.1Cert certificate_list<0..2^24-1>; } Certificate;
```

- certificate_list

  证书列表，发送者的证书必须是第一个，后续的每一个证书都必须是前一个的签署证书。根证书可以省略

证书申请的时候，一般会收到好几个证书，有的需要自己按照这个格式来拼接成证书链。

如果服务器要认证客户端的身份，那么服务器会发送Certificate Request消息，客户端应该也以 这条Server Certificate消息的格式回复。

服务器发送的证书必须：

- 证书类型必须是 X.509v3。除非明确地协商成别的了(比较少见，rfc里提到了例如 OpenPGP格式)。
- 服务器证书的公钥，必须和选择的密钥交换算法配套。

密钥交换+认证算法	配套的证书中公钥类型
RSA / RSA_PSK	RSA 公钥；证书中必须允许私钥用于加密 （即如果使用了X509V3规定的key usage扩展， keyEncipherment比特位必须置位） **这种用法没有前向安全性，因此在 TLS 1.3中被废弃了**
DHE_RSA / ECDHE_RSA	RSA 公钥；证书中必须允许私钥用于签名(即如果使用了X509V3规定的key usage扩展， digitalSignature比特位必须置位)，并且允许server key exchange消息将要使用的签名模式(例如 PKCS1_V1.5 ， OAEP等)和hash算法(例如sha1, sha256等)
DHE_DSS	DSA 公钥; 历史遗留产物，从来没有被大规模用过，安全性差，废弃状态。证书必须允许私钥用于签名，必须允许server key exchange消息中使用的hash算法。
DH_DSS / DH_RSA	Diffie-Hellman 公钥; 要求key usage里面的keyAgreement比特位必须置位。**这种用法没有前向安全性，因此在 TLS 1.3中被废弃了**
ECDH_ECDSA / ECDH_RSA	能做 ECDH 用途的公钥；公钥必须使用 客户端支持的ec曲线和点格式。**这种用法没有前向安全性，因此在 TLS 1.3中被废弃了**
ECDHE_ECDSA	ECDSA用途的公钥；证书必须运输私钥用作签名，必须允许server key exchange消息里面要用到的hash算法。公钥必须使用客户端支持的ec曲线和点格式。

- **“server_name” 和 “trusted_ca_keys” 扩展用于引导证书选择。**

其中有5种是ECC密钥交换算法：ECDH_ECDSA, ECDHE_ECDSA, ECDH_RSA, ECDHE_RSA, ECDH_anon。 ECC（椭圆曲线）体制相比RSA，由于公钥更小，性能更高，所以在移动互联网环境下越发重要。 以上ECC的5种算法都用ECDH来计算premaster secret， 仅仅是ECDH密钥的生命周期和认证算法不同。 其中只有 ECDHE_ECDSA 和 ECDHE_RSA 是前向安全的。

如果客户端在ClientHello里提供了 “signature_algorithms” 扩展，那么服务器提供的所有证书必须用 “signature_algoritms"中提供的 hash/signature算法对 之一签署。要注意的是，这意味着，一个包含某种签名算法密钥的证书，可能被另一种签名算法签署（例如，一个RSA公钥可能被一个ECDSA公钥签署）。(这在TLS1.2和TLS1.1中是不一样的，TLS1.1要求所有的算法都相同。)注意这也意味着 DH_DSS,DH_RSA,ECDH_ECDSA,和ECDH_RSA 密钥交换不限制签署证书的算法。固定DH证书可能使用"signature_algorithms"扩展列表中的 hash/签名算法对 中的某一个签署。名字 DH_DSS, DH_RSA, ECDH_ECDSA, 和 ECDH_RSA 只是历史原因，这几个名字的后半部分中指定的算法，并不会被使用，即DH_DSS中的DSS并不会被使用，DH_RSA中并不会使用RSA做签名，ECDH_ECDSA并不会使用ECDSA算法。。。 如果服务器有多个证书，就必须从中选择一个，一般根据服务器的外网ip地址，SNI中指定的hostname，服务器配置来做选择。如果服务器只有一个证书，那么要确保这一个证书符

合这些条件。 要注意的是，存在一些证书使用了TLS目前不支持的 算法组合。例如，使用 RSASSA-PSS 签名公钥的证书（即证书的SubjectPublicKeyInfo字段是id-RSASSA-PSS）。由于TLS没有给这些算法定义对应的签名算法，这些证书不能在TLS中使用。 如果一个CipherSuite指定了新的TLS密钥交换算法，也会指定证书格式和要求的密钥编码方法。

6. handshake – Server Key Exchange

服务器会在 server Certificate 消息之后，立即发送 Server Key Exchange消息。（如果协商出的CipherSuite不需要做认证，即anonymous negotiation，会在ServerHello之后立即发送Server Key Exchange消息）

只有在server Certificate 消息没有足够的信息，不能让客户端完成premaster的密钥交换时，服务器才发送 server Key Exchange， 主要是对前向安全的几种密钥协商算法，列表如下：

1. DHE_DSS
2. DHE_RSA
3. DH_anon
4. ECDHE_ECDSA
5. ECDHE_RSA
6. ECDH_anon

对下面几种密钥交换方法，发送ServerKeyExchange消息是非法的：

1. RSA
2. DH_DSS
3. DH_RSA
4. ECDH_ECDSA
5. ECDH_RSA

需要注意的是，ECDH和ECDSA公钥的数据结构是一样的。所以，CA在签署一个证书的时候，可能要使用 X.509 v3 的 keyUsage 和 extendedKeyUsage 扩展来限定ECC公钥的使用方式。

ServerKeyExchange传递足够的信息给客户端，来让客户端交换premaster secret。一般要传递的是：一个 Diffie-Hellman 公钥，或者一个其他算法(例如RSA)的公钥。

在TLS实际部署中，我们一般只使用这4种：ECDHE_RSA, DHE_RSA, ECDHE_ECDSA，RSA

其中RSA密钥协商（也可以叫密钥传输）算法，由于没有前向安全性，在TLS 1.3里面已经被废除了。参见： Confirming Consensus on removing RSA key Transport from TLS 1.3

消息格式：

```
enum { dhe_dss, dhe_rsa, dh_anon, rsa, dh_dss, dh_rsa,
ec_diffie_hellman } KeyExchangeAlgorithm; struct { opaque
dh_p<1..2^16-1>; opaque dh_g<1..2^16-1>; opaque dh_Ys<1..2^16-1>;
} ServerDHParams; /* Ephemeral DH parameters */ dh_p Diffie-
Hellman密钥协商计算的大质数模数。 dh_g Diffie-Hellman 的生成元， dh_Ys 服
务器的Diffie-Hellman公钥 (g^X mod p). struct { opaque point <1..2^8-
1>; } ECPoint; enum { explicit_prime (1), explicit_char2 (2),
named_curve (3), reserved(248..255) } ECCurveType; struct {
ECCurveType curve_type; select (curve_type) { case named_curve:
NamedCurve namedcurve; }; } ECParameters; struct { ECParameters
curve_params; ECPoint public; //ECDH的公钥 } ServerECDHParams;
struct { select (KeyExchangeAlgorithm) { case dh_anon:
ServerDHParams params; case dhe_dss: case dhe_rsa: ServerDHParams
params; digitally-signed struct { opaque client_random[32]; opaque
server_random[32]; ServerDHParams params; } signed_params; case
ec_diffie_hellman: ServerECDHParams params; Signature
signed_params; case rsa: case dh_dss: case dh_rsa: struct {} ; /*
message is omitted for rsa, dh_dss, and dh_rsa */ /* may be
extended, e.g., for ECDH -- see [TLSECC] */ }; }
ServerKeyExchange; params 服务器的密钥交换参数。 signed_params 对需要认
证的（即非anonymous的）密钥交换，对服务器的密钥交换参数的数字签名。
```

ECParameters 结构比较麻烦，其中ECCurveType是支持3种曲线类型的，可以自行指定椭圆曲线的多项式系数，基点等参数。但是，我们基本不会用到这种功能，因为一般部署都是使用 NamedCurve，即参数已经预先选定，各种密码学库普遍都支持的一组曲线，其中目前用的最广的是 secp256r1 （还被称为 P256，或 prime256v1）

NamedCurve 列表中比较重要的曲线(在TLS1.3中，只保留了这几条曲线。)，定义如下：

```
enum { ... secp256r1 (23), secp384r1 (24), secp521r1 (25), reserved (0xFE00..0xFEFF), (0xFFFF) } NamedCurve;
```

ECDHE_RSA 密钥交换算法的 SignatureAlgorithm 是 rsa 。 ECDHE_RSA 密钥交换算法的 SignatureAlgorithm 是 ecdsa。

如果客户端提供了 “signature_algorithms” 扩展， 则签名算法和hash算法必须是列在扩展中的算法。 要注意的是，这个地方可能有不一致，例如客户端可能提供了 DHE_DSS 密钥交换，但是 “signature_algorithms"扩展中没有DSA算法，在这类情况下，为了正确地协商，服务器必须确保满足自己选择的CipherSuite满足 “signature_algorithms” 的限制。这不优雅，但是是为了把对原来的CipherSuite协商的设计的改动减到最小，而做的妥协。

并且，hash和签名算法，必须和服务器的证书里面的公钥兼容。

7. handshake – Certificate Request

TLS规定了一个可选功能：服务器可以认证客户端的身份，这通过服务器要求客户端发送一个证书实现，服务器应该在ServerKeyExchange之后立即发送CertificateRequest消息。

消息结构：

```
enum { rsa_sign(1), dss_sign(2), rsa_fixed_dh(3),dss_fixed_dh(4),
rsa_ephemeral_dh_RESERVED(5),dss_ephemeral_dh_RESERVED(6), fortezza_dms_RESERVED(20),
ecdsa_sign(64), rsa_fixed_ecdh(65), ecdsa_fixed_ecdh(66), (255) }
ClientCertificateType; opaque DistinguishedName<1..2^16-1>; struct {
ClientCertificateType certificate_types<1..2^8-1>; SignatureAndHashAlgorithm
supported_signature_algorithms<2^16-1>; DistinguishedName
certificate_authorities<0..2^16-1>; } CertificateRequest;
```

- **certificate_types**

  客户端可以提供的证书类型。

- **rsa_sign 包含RSA公钥的证书。**
- **dss_sign 包含DSA公钥的证书。**
- **rsa_fixed_dh 包含静态DH公钥的证书。**
- **dss_fixed_dh 包含静态DH公钥的证书。**

  supported_signature_algorithms : 服务器支持的 hash/signature 算法的列表。

  certificate_authorities : 服务器可以接受的CA(certificate_authorities)的 distinguished names 的列表 DER编码格式.

这些 distinguished names 可能为root CA或者次级CA指定了想要的 distinguished name ，因此，这个消息可以用来描述已知的root，或者希望的授权空间。 如果 certificate_authorities 列表是空的，那么客户端可以发送任何适当的 ClientCertificateType 类型的证书，如果没有别的限制的话。

certificate_types 和 supported_signature_algorithms 字段的交叉选择很复杂。
certificate_types 这个字段从SSLv3时代就定义了，但是一直都没有详细定义，其大多数功能都被 supported_signature_algorithms 代替了。 有如下规则：

- 客户端提供的任何证书，必须用一个supported_signature_algorithms 中出现过的 hash/signature 算法对 签名.
- 客户端提供的末端证书必须提供一个和 certificate_types 兼容的key。 如果这个key是一个签名key，那必须能和 supported_signature_algorithms 中提供的某个 hash/signature 算法对配合使用。
- 由于历史原因，某些客户端证书类型的名字，包含了证书的签名算法，例如，早期版本的TLS中，rsa_fixed_dh 意思是一个被RSA算法签署，并且包含一个固定DH密钥的证书。在TLS1.2中，这个功能被 supported_signature_algorithms 淘汰，并且证书类型不再限制用来签署证书的算法。例如，如果服务器发送了 dss_fixed_dh 证书类型，和 { {sha1, dsa}, {sha1,rsa} } 签名类型，客户端可以回复一个 包含静态DH密钥，用RSA-sha1签署的证书。
- 如果协商出来的是匿名CipherSuite，服务器不能要求客户端认证。

8. handshake – Server Hello Done

在 ServerHello和相关消息已经处理结束后，服务器发送ServerHelloDone。在发送 ServerHelloDone后，服务器开始等待客户端的响应。

ServerHelloDone消息表示，服务器已经发送完了密钥协商需要的消息，并且客户端可以开始客户端的密钥协商处理了。

收到ServerHelloDone后，客户端应该确认服务器提供了合法的证书，并且确认服务器的ServerHello消息里面的参数是可以接受的。

消息格式：

```
1  struct { } ServerHelloDone;
```

9. handshake – Client Certificate

ClientCertificate消息是客户端收到ServerHelloDone后，可以发送的第一条消息。仅当服务器要求了一个证书的情况下，客户端才发送ClientCertificate消息，如果没有可用的合适证书，客户端必须发送一条不包含任何证书的ClientCertificate消息（即 certificate_list 结构长度为0）。

如果客户端没有发送任何证书，服务器自行决定，可以放弃要求客户端认证，继续握手；或者发送一条 fatal handshake_failure的alert消息，断开连接。并且，如果证书链的某些方面是不能接受的（比如证书没有被可信任的CA签署），服务器可以自行决定，是继续握手（放弃要求客户端认证），或者发送一条fatal的alert。

客户端证书使用和ServerCertificate相同的结构发送。

ClientCertificate把客户端的证书链发送给服务器。服务器会使用证书链来验证CertificateVerify 消息（如果使用基于签名的客户端认证），或者来计算premaster secret（对于非短暂的 DH）。证书必须和协商出来的CipherSuite的密钥交换算法配套，并和任何协商的扩展配套。

尤其是：

- **证书必须是X.509v3 类型的。**
- **客户端的末级证书的公钥必须和CertificateRequest里列出的证书类型兼容。**

客户端证书类型	证书公钥类型
rsa_sign	RSA公钥；证书必须允许公钥用于certificateVerify消息中的数字签名和hash算法
dss_sign	DSA 公钥；证书必须允许密钥使用CertificateVerify中的hahs函数做签名；
ecdsa_sign	可以用作 ECDSA 的公钥；证书必须允许 公钥用 CertificateVerify中的hash函数做签名；公钥必须使用服务器支持的曲线，和点格式；
rsa_fixed_dh / dss_fixed_dh	Diffie-Hellman 公钥; 必须使用和服务器key相同的参数。
rsa_fixed_ecdh / ecdsa_fixed_ecdh	可以用作 ECDH 的公钥。必须和服务器的公钥使用同样的曲线，同样的点格式

- **如果 certificate_authorities 列表不是空的，客户端证书链中的某一个证书必须是CA中的某一个签署的。**
- **证书必须使用 服务器可以接受的 hash/signature 算法对。**

类似于Server Certificate，有一些证书目前无法在TLS中使用。

10. handshake – Client Key Exchange

客户端必须在客户端的Certificate消息之后，立即发送ClientKeyExchange消息。 或者必须在ServerHelloDone后立即发送ClientKeyExchange消息。

ClientKeyExchange消息中，会设置premaster secret，通过发送 RSA公钥加密premaster secret的密文，或者发送允许双方得出相同的premaster secret的Diffie-Hellman参数。

当客户端使用短暂的 Diffie-Hellman 密钥对时，ClientKeyExchange包含客户端的 Diffie-Hellman 公钥。如果客户端发送一个包含静态 Diffie-Hellman 指数的证书（比如，在使用固定DH的客户端认证），那么这条消息必须被发送，并且必须为空。

消息结构： 消息的选择取决于选择的密钥交换算法。

```
struct { select (KeyExchangeAlgorithm) { case rsa:
EncryptedPreMasterSecret; case dhe_dss: case dhe_rsa: case dh_dss:
case dh_rsa: case dh_anon: ClientDiffieHellmanPublic; case
ec_diffie_hellman: ClientECDiffieHellmanPublic; } exchange_keys; }
ClientKeyExchange;
```

(1). RSA 加密的 Premaster Secret 消息

如果用RSA做密钥协商和认证，客户端生成 48字节的 premaster secret，使用服务器证书里面的公钥加密，然后把密文EncryptedPreMasterSecret发送给服务器，结构定义如下：

```
struct { ProtocolVersion client_version; opaque random[46]; }
PreMasterSecret; client_version 客户端支持的最新协议版本号，这个字段用来检测中间人版本回退攻击。T random 46 字节的，安全生成的随机值。 struct { public-key-encrypted PreMasterSecret pre_master_secret; }
EncryptedPreMasterSecret; pre_master_secret 这个随机值由客户端生成，用于生成master secret。
```

注：PreMasterSecret里面的 client_version 是 ClientHello.client_version，而不是协商的到的版本号，这个特性用来阻止版本回退攻击。不幸的是，有些不正确的老的代码使用了协商得到的版本号，导致检查client_version字段的时候，和正确的实现无法互通。

客户端实现必须在PreMasterSecret中发送正确的版本号。如果 ClientHello.client_version 的版本号是 TLS 1.1 或者更高，服务器实现必须如下检查版本号。如果版本号是 TLS 1.0 或者更早，服务器必须检查版本号，但是可以通过配置项关闭检查。

要注意的是，如果版本号检查失败了，PreMasterSecret 应该像下面描述的那样填充成随机数。

TLS中的RSA使用的是 PKCS1-V1.5 填充(PKCS1-V1.5也是openssl库RSA的默认填充方式)。Bleichenbacher 在1998年发表了一种针对 PKCS1-V1.5 的选择密文攻击， Klima在2003年发现 PKCS1-V1.5 中 PreMasterSecret 版本号检查的一个侧通道攻击。只要TLS 服务器暴露一条特定的消息是否符合PKCS1-V1.5格式，或暴露PreMasterSecret解密后结构是否合法，或版本号是否合法，就可以用上面2种方法攻击。

Klima 还提出了完全避免这类攻击的方法：对格式不正确的消息，版本号不符的情况，要做出和完全正确的RSA块一样的响应，要让客户端区分不出这3种情况。 具体地说，要如下：

1. 生成 46 字节的密码学安全随机值 R
2. 解密消息，获得明文 M
3. 如果 PKCS#1 填充不正确，或者 PreMasterSecret 消息的长度不是48字节，则 pre_master_secret = ClientHello.client_version || R 或者如果 ClientHello.client_version <= TLS 1.0，并且明确禁止了版本号检查，则 pre_master_secret = ClientHello.client_version

|| M[2..47]

注意：明确地用 ClientHello.client_version 构造 pre_master_secret 时，当客户端在原来的 pre_master_secret 中发送了错误的 客户端版本值时，会产生一个不合法的 master_secret 。

另一种解决问题的方法是，把版本号不符，当成 PKCS-1 格式错误来对待，并且完全随机填充 premaster secret。

1. 生成 48 字节的密码学安全随机值 R
2. 解密 PreMasterSecret 恢复出明文 M
3. 如果 PKCS#1 填充不正确，或者消息的长度不是48字节，则 pre_master_secret = R 或者如果 ClientHello.client_version <= TLS 1.0，并且 明确禁止了版本号检查，则 pre_master_secret = M 或者如果 M[0..1] != CleintHello.client_version pre_master_secret = R 或者 pre_master_secret = M

尽管实践中，还没有发现针对这种结构的攻击，Klima 在论文中描述了几种理论上的攻击方式，因此推荐上述的第一种结构。

在任何情况下，一个 TLS 服务器绝对不能在:1. 处理 RSA 加密的 premaster 消息失败, 2.或者版本号检查失败 时产生alert消息。当遇到这两种情况时，服务器必须用随机生成的 premaster 值继续握手。服务器可以把造成失败的真实原因log下来，用于调查问题，但是必须小心确保不能把这种信息泄漏给攻击者（比如通过时间侧通道，log文件，或者其它通道等泄漏）。

RSAES-OAEP 加密体制，更能抵抗 Bleichenbacher 发表的攻击，然而，为了和早期的TLS版本最大程度保持兼容，TLS 仍然规定使用 RSAES-PKCS1-v1_5 体制。只要遵守了上面列出的建议，目前还没有 Bleichenbacher 的变化形式能攻破 TLS 。

实现的时候要注意：公钥加密的数据用 字节数组 <0..2^16-1> 的形式表示。因此，ClientKeyExchange中的 RSA加密的PreMasterSecret 前面有2个字节用来表示长度。这2个字节在使用RSA做密钥协商时，是冗余的，因为此时 EncryptedPreMasterSecret 是 ClientKeyExchange 中的唯一字段，因此可以无歧义地得出 EncryptedPreMasterSecret 的长度。因此更早的 SSLv3 规范没有明确规定 public-key-encrypted 数据的编码格式，因此有一些SSLv3的实现没有包含 长度字段，这些实现直接把 RSA 加密的数据放入了 ClientKeyExchange消息里面。 TLS规范要求 EncryptedPreMasterSecret 字段包含长度字段。因此得出的结果会和一些 SSLv3 的实现不兼容。实现者从 SSLv3 升级到 TLS 时，必须修改自己的实现，以接受并且生成带长度的格式。如果一个实现要同时兼容 SSLv3 和 TLS，那就应该根据协议版本确定自己的行为。

注意：根据 Boneh 等在2003年USENIX Security Symposium上发表的论文 “Remote timing attacks are practical”，针对 TLS RSA密钥交换的远程时间侧通道攻击，是实际可行的，起码当客户端和服务器在同一个LAN里时是可行的。因此，使用静态 RSA 密钥的实现，必须使用 RSA blinding，或者Boneh论文中提到的，其他抵抗时间侧通道攻击的技术。

openssl中的RSA blinding，参见：http://linux.die.net/man/3/rsa_blinding_on

(2). 客户端 Diffie-Hellman 公钥

这条消息把客户端的 Diffie-Hellman 公钥 (Yc) 发送给服务器。

Yc的编码方式由 PublicValueEncoding 决定。

消息的结构：

```
enum { implicit, explicit } PublicValueEncoding; implicit 如果客户端已经发送了一个包含合适的 DH 公钥的证书（即 fixed_dh 客户端认证方式），那么Yc已经隐式包含了，不需要再发送。这种情况下，ClientKeyExchange消息必须发送，并且必须是空的。 explicit 表示Yc需要发送。 struct { select (PublicValueEncoding) { case implicit: struct { }; case explicit: opaque dh_Yc<1..2^16-1>; } dh_public; } ClientDiffieHellmanPublic; dh_Yc 客户端的 Diffie-Hellman 公钥 Yc.
```

(3). 客户端 EC Diffie-Hellman 公钥

```
struct { select (PublicValueEncoding) { case implicit: struct { }; case explicit: ECPoint ecdh_Yc; } ecdh_public; } ClientECDiffieHellmanPublic;
```

Diffie-Hellman 推广到椭圆曲线群上，就是 EC Diffie-Hellman ，简称 ECDH，其它的计算，和一般的 DH 计算类似。

ECDH 是目前最重要的密钥协商算法 http://vincent.bernat.im/en/blog/2011-ssl-perfect-forward-secrecy.html

11. handshake – Cerificate Verify

当需要做客户端认证时，客户端发送CertificateVerify消息，来证明自己确实拥有客户端证书的私钥。这条消息仅仅在客户端证书有签名能力的情况下发送（就是说，除了含有固定 Diffie-Hellman 参数的证书以外的证书）。CertificateVerify必须紧跟在ClientKeyExchange之后发送。

消息结构： Structure of this message:

```
struct { digitally-signed struct { opaque handshake_messages[handshake_messages_length]; } } CertificateVerify;
```

此处， handshake_messages 表示所有发送或者接收的握手消息，从client hello开始，一直到CertificateVerify之前的所有消息，包括handshake消息的type和length字段，这是之前所有握手结构体的拼接。要注意，这要求双方在握手过程中，都得缓存所有消息，或者在握手过程中，用每一种可能的hash算法计算到CeritificateVerify为止的hash值。

signature中用的hash和签名算法必须是 CertificateRequest 的 supported_signature_algorithms 中的某一种。另外，hash和签名算法必须和客户端的证书的算法兼容。 RSA公钥可能被用于任何允许的hash函数，只要遵循证书中的限制。

12. handshake – Finished

在 ChangeCipherSpec 消息之后，应该立即发送 Finished 消息，来确认密钥交换和认证过程已经成功了。ChangeCipherSpec 必须在其它握手消息和 Finished 消息之间。

Finished 消息是第一条用刚刚协商出来的参数保护的消息。接收方必须确认Finished消息的内容是正确的。一旦某一方发送了，并且确认了对端发来的Finished消息，就可以开始在连接上发送和接收应用数据了。

消息结构：

```
1
2 3
4 5
6 7
8 9
10
11
```

```
struct { opaque verify_data[verify_data_length]; } Finished;
verify_data PRF(master_secret,
finished_label,Hash(handshake_messages)) [0..verify_data_length-1];
finished_label 对客户端发的Finished消息来说，固定是字符串 "client finished".
对服务器发的Finished消息来说，固定是字符串 "server finished".
```

Hash表示握手消息的hash。hash函数是前文 PRF 的hash 函数。或者 CipherSuite 规定的用于 Finished 计算的hash函数。

在TLS的之前版本中，verify_data 总是 12 字节。在TLS 1.2中，这取决于CipherSuite。如果CipherSuite没有显式规定 verify_data_length ，就当成12字节处理。将来的CipherSuite可能会规定别的长度，但是不能小于12字节。

Finished 消息必须跟在 ChangeCipherSpec 消息之后，如果顺序错乱，就是 fatal error.

handshake_message 的内容包含从 ClientHello开始，直到 本条Finished之前的所有消息，只包含handshake层的消息体，不包含record层的几个消息头字段。包括CertificateVerify 消息。同时，对客户端和服务器来说，handshake_message 的内容不同， 后发送者必须包含前发送者的 Finished 消息。

注意：ChangeCipherSpec 消息，alert，和其它的record 类型不是握手消息，不包含在 hash计算中。同时，HelloRequest 消息也不算在内。

13. handshake – NewSessionTicket

SessionTicket 定义在 RFC5077 标准里面，2008年发布。

SessionTicket是一种不需要服务器端状态的，恢复TLS session的方式。 SessionTicket可以用于任何CipherSuite。 TLS 1.0, TLS 1.1, TLS 1.2 都适用。

在下面这些场景下，尤其有用：

用户量巨大，session id的方式耗费服务器内存过多 服务器希望长时间缓存session 服务器有多台，不希望服务器间有共享状态 服务器内存不足 客户端在 ClientHello中设置一个 SessionTicket 扩展来标识自己支持 SessionTicket。如果客户端本地没有存之前收到的ticket，就把这个扩展设为空。

如果服务器希望使用 SessionTicket 机制，服务器把本地的 session 状态存入一个ticket中，ticket会被加密，并被MAC保护，无法篡改，加密和算MAC用的key只有服务器知道。 加密并MAC过的ticket用 NewSessionTicket 消息分发给客户端，NewSessionTicket 消息应该在 ChangeCipherSpec 消息之前，在服务器验证通过客户端的Finished消息之后发送。

```
1 2 3
4 5 6
7 8 9
10 11
12 13
14 15
16 17
18 19
20 21
```

```
Client Server ClientHello (empty SessionTicket extension)-------> ServerHello (empty SessionTicket extension) Certificate* ServerKeyExchange* CertificateRequest* <-------- ServerHelloDone Certificate* ClientKeyExchange CertificateVerify* [ChangeCipherSpec] Finished --------> NewSessionTicket [ChangeCipherSpec] <-------- Finished Application Data <-------> Application Data Figure 1: Message flow for full handshake issuing new session ticket
```

客户端把收到的ticket和master secret等其它与当前session有关的参数一起，缓存起来。 单客户端希望恢复会话时，就把ticket包含在 ClientHello 的 SessionTicket 扩展中发给服务器。 服务器收到后，解密ticket，算MAC确认ticket没有被篡改过，然后从解密的内容里面，获取session 状态，用来恢复会话。如果服务器成功地验证了ticket，可以在 ServerHello 之后返回一个 NewSessionTicket 消息来更新ticket。

显然，这种情况下，相比完整握手，可以省掉1个RTT。如下图：

```
1 2
3 4
5 6
7 8
9 10
11
12
13
14
```

```
Client Server ClientHello (SessionTicket extension) --------> ServerHello (empty SessionTicket extension) NewSessionTicket [ChangeCipherSpec] <-------- Finished [ChangeCipherSpec] Finished --------> Application Data <-------> Application Data Figure 2: Message flow for abbreviated handshake using new session ticket
```

如果服务器不能，或者不想使用客户端发来的ticket，那服务器可以忽略ticket，启动一个完整的握手流程。

如果服务器此时不希望下发新的ticket，那就可以不回复 SessionTicket 扩展，或者不回复 NewSessionTicket 消息。 此时除了 ClientHello里面的 SessionTicket扩展，就和一般的TLS流程一样了。

如果服务器拒绝收到的ticket，服务器可能仍然希望在完整的握手之后，下发新的ticket。 此时流程和全新 ticket 生成下发的区别，就是ClientHello的SessionTicket不是空的。

NewSessionTicket 消息 服务器在握手过程中，发ChangeCipherSpec之前发送NewSessionTicket消息。 如果服务器在ServerHello中包含了一个SessionTicket扩展，那就必须发送NewSessionTicket消息。 如果服务器没有包含SessionTicket扩展，那绝对不能发送NewSessionTicket消息。 如果服务器在包含了SessionTicket扩展之后，不想发送ticket，那可以发送一个长度为0的NewSessionTicket消息。

在完整握手的情况下，客户端必须在确认服务器的Finished消息正确之后，才能认为NewSessionTicket 里面的ticket合法。

服务器可以NewSessionTicket消息中更新 ticket。

ticket_lifetime_hint 字段包含一个服务器的提示，提示客户端本ticket应该存多长时间就失效。单位是秒，网络字节序。当时间到期时，客户端应该删掉ticket和关联的状态。客户端也可以提前删除。服务器端也可以提前认为ticket失效。

```
1 2 3 4
```

```
struct { uint32 ticket_lifetime_hint; opaque ticket<0..2^16-1>; } NewSessionTicket;
```

SessionTicket 和 Session ID 之间的关系比较繁琐。感兴趣的自行去看RFC吧。

对于客户端来说，ticket就是一块二进制buffer，客户端并不管里面的内容。所以ticket具体怎么加密加MAC服务器可以为所欲为，无需顾及客户端的感受。

RFC5077中推荐了一种ticket的加密保护方法： 服务器使用2个key，一个 aes-128-cbc的key，一个 HMAC-SHA-256 的key。

ticket的格式像这样：

```
1 2 3 4 5 6
```

```
struct { opaque key_name[16]; opaque iv[16]; opaque encrypted_state<0..2^16-1>; opaque mac[32]; } ticket;
```

其中，key_name 用来标识一组key，这样服务器端就可以使用多组key。

加密过程，首先随机生成IV，然后用 aes-128-cbc 加密 session 的序列化结果， 然后用 HMAC-SHA-256 对 key_name,IV,encrypted_data 的长度（2字节），encrypted_data 计算MAC。 最好把各个字段填入上面ticket结构体。 显然，此处是 Encrypt-then-MAC的方式，是最安全的。

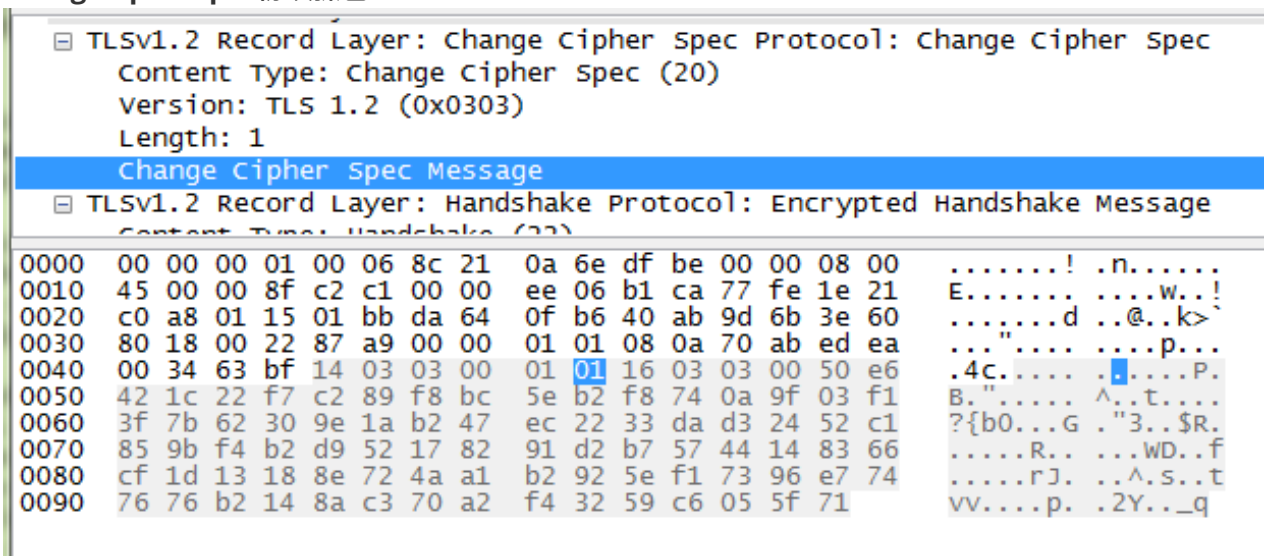
实际在openssl 中的session，用asn1格式序列化保存了下面这些字段：

```
typedef struct ssl_session_asn1_st { ASN1_INTEGER version;
ASN1_INTEGER ssl_version; ASN1_OCTET_STRING cipher; ASN1_OCTET_STRING
master_key; ASN1_OCTET_STRING session_id; ASN1_OCTET_STRING
session_id_context; ASN1_INTEGER time; ASN1_INTEGER timeout;
ASN1_INTEGER verify_result; ASN1_OCTET_STRING tlsext_hostname;
ASN1_INTEGER tlsext_tick_lifetime; ASN1_OCTET_STRING tlsext_tick; }
SSL_SESSION_ASN1;
```

6. ChangeCipherSpec 协议

ChangeCipherSpec用来通知对端，开始启用协商好的Connection State做对称加密，内容只有1个字节。 这个协议是冗余的，在TLS 1.3里面直接被删除了。

changeCipherSpec协议抓包：



7. Alert 协议

一种返回码机制，简单

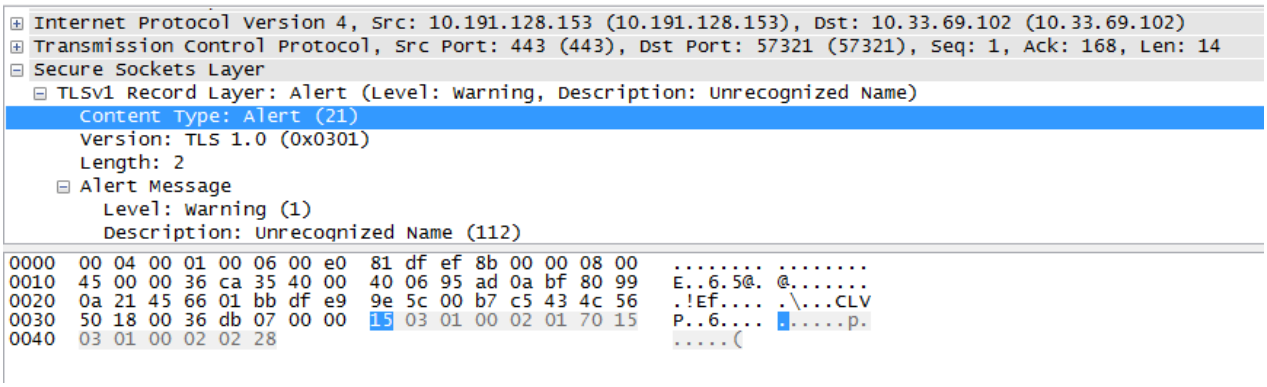
```
enum { warning(1), fatal(2), (255) } AlertLevel;

struct {
    AlertLevel level;
    AlertDescription description;
} Alert;
```

其中level是等级，不同等级要求不同的处理。

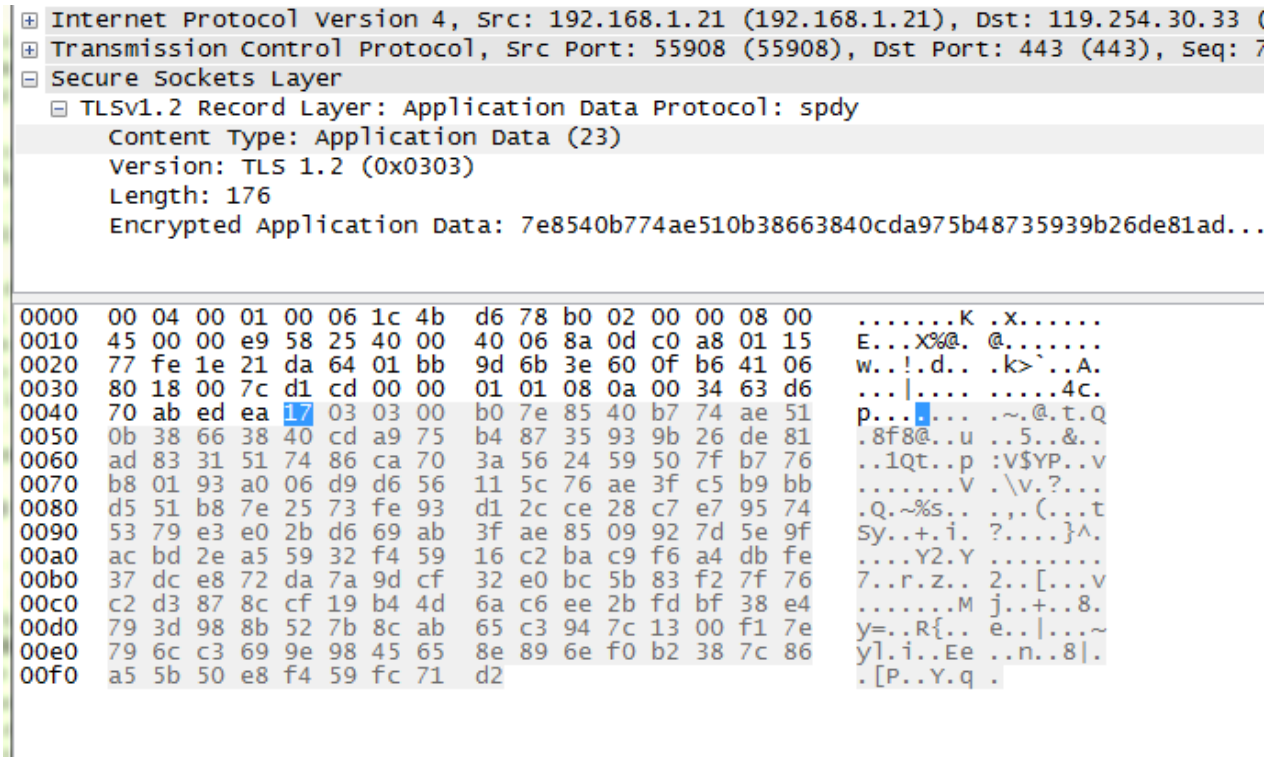
其中有一种：close_notify，用来通知对端，我不会再发送更多数据了。这个可以让对端主动close fd，这样可以减少我方tcp timewait状态的socket 量。

alert协议：



8. application data协议

application data协议，就是把应用数据直接输入record层，做分段，算MAC，加密，传输。 抓包举例如下：



8. TLS协议的安全分析

安全分析，重中之重，也是大家最关心的。

安全分析的第一步是建立攻击模型，TLS的攻击模式是：

- 攻击者有充足的计算资源
- 攻击者无法得到私钥，无法得到客户端和服务器内存里面的密钥等保密信息
- 攻击者可以抓包，修改包，删除包，重放包，篡改包。

这个模型其实就是密码学里面一般假定的攻击模型。

好了，在这个模型下，TLS的安全性分析如下：

1. 认证和密钥交换 的安全性

TLS有三种认证模式：双方认证，服务器认证，无认证。 只要包含了有服务器端认证，就可以免疫 man-in-the-middle 攻击。但是完全匿名的会话是可以被 MITM 攻击的。匿名的服务器不能认证客户端。

密钥交换的目的，是产生一个只有通信双方知道的共享密钥 pre_master_secret 。 pre_master_secret 用来生成 master_secret 。 master_secret 用来生成 Finished 消息，加密 key，和MAC key。通过发送正确的Finished消息，双方可以证明自己知道正确的 pre_master_key。

1. 匿名密钥交换

匿名密钥交换是一种历史遗留的不安全方式。 匿名密钥交换缺失认证(Authentication)，所以绝大多数场景下，我们应该禁用这种方式。

2. RSA 密钥交换和认证

当使用RSA的时候，合并了密钥交换 和 服务器端认证。 RSA公钥包含在服务器证书中。要注意的是，一旦服务器证书的RSA私钥泄露，之前用该证书保护的所有流量都会变成可以破解的，即没有前向安全性(Perfect Forward Secrecy)。 需要前向安全性的TLS用户，应该使用 DHE 或者 EC TLS users desiring Perfect Forward Secrecy should use DHE 类的CcipherSuite。这样，如果私钥泄露，只需要更换私钥和证书就行，不会有更大的损失。

RSA密钥交换和认证的安全性基于，在验证了服务器的证书之后，客户端用服务器的公钥加密一个 pre_master_secret 。成功地解密 pre_master_secret 并产生正确地 Finished 消息之后，就可以确信服务器拥有证书对应的私钥。

如果使用了客户端认证，通过 CertificateVerify 消息来认证客户端。客户端会签署一个之前所有握手消息的hash值，这些握手消息包括 服务器的证书，ServerHello.random 。其中服务器证书确保客户端签署了和本服务器有关的绑定(即不能重放和别的服务器的握手)，ServerHello.random 确保签名和当前握手流程绑定(即不能重放)。

3. Diffie-Hellman 密钥交换和认证

当使用 DH 密钥交换的时候，服务器:

1. 或者发送包含固定 DH参数的证书
2. 或者发送一组临时DH参数，并用 ECDSA/RSA/DSA 证书的私钥签署。而且在签署之前，临时DH参数和 hello.random 都参与hash计算，来确保攻击者不能重放老的签名值。

无论如何，客户端都可以通过验证证书，或者验证签名，来确保收到的DH参数确实来自真正的服务器。

如果客户端有一个包含固定 Diffie-Hellman 参数的证书，则证书包含完成密钥交换所需的参数。要注意的是，这种情况下，客户端和服务器每次都会协商出相同的 DH 结果(就是 pre_master_secret)。 为了尽可能减少 pre_master_secret 存在在内存里面的时间，当不再需要的时候，尽快将其清除，pre_master_secret 应该尽早转换成 master_secret 的形式。 为了进行密钥交换，客户端发送的 Diffie-Hellman 参数必须和服务器发送的兼容。

如果客户端有一个标准的 DSA 或者 RSA 证书，或者 客户端没有被认证，那么客户端在ClientKeyExchange中发送一组临时参数，或者可选地发送一个CertificateVerify消息来证明自己的身份。

如果相同的 DH 密钥对，被多次用于握手协商，不管是由于客户端或服务器使用了固定DH密钥的证书，还是服务器在重用 DH 密钥，都必须小心避免 small subgroup 攻击。实现都必须遵循 rfc2785 中的标准。

最简单避免 small subgroup 攻击的方法是使用一种 DHE CipherSuite，并且每次都握手都生成一个新的 DH 私钥 X。如果选择了一个合适的base（例如2），g^X mod p 的计算可以非常快，因而性能开销会最小化。并且每次都使用一个新的DH密钥，可以提供前向安全性。当使用 DHE 类的CipherSuite 时，实现必须每次握手都生成一个全新的DH私钥（即 X ）。

由于TLS允许服务器提供任意的 DH 群，客户端必须确认服务器提供的DH 群的大小适合本地策略。 客户端必须确认 DH 公共指数有足够的大小。 如果DH群已知的话，客户端做简单比对就行了，因此服务器可以使用一个已知的群，来方便客户端的检查。

2. 版本回退攻击

由于 TLS 历史上出现过多个版本，服务器端实现可能会兼容多个版本的协议，而像 SSL 2.0 这样的版本是有严重安全问题的，因此攻击者可能会尝试诱骗客户端和服务器，来使TLS连接回退到 SSL 2.0这种老版本。

TLS 对此的解决办法，就是PreMasterSecret里面包含版本号。

3. 针对握手过程的攻击

攻击者可能会尝试影响握手过程，来使双方选择不安全的加密算法。

对这种攻击的解决办法是，如果握手消息被篡改了，那么在Finished消息里，客户端和服务器都会计算握手消息的hash，如果攻击者篡改了握手消息，那么双方得出的hash就不一样，这样Finished消息就会验证不过。就会发现攻击。

4. 针对 Resuming Sessions 的攻击

当使用 session resuming的时候，会产生新的 ClientHello.random 和 ServerHello.random ，并和session的 master_secret 一同被hash。只要master_secret没有泄漏，并且PRF中用来生成加密key和MAC key的hash算法是安全的，连接就是安全的，并且独立于前一个连接(被恢复的前一个连接)。

只有在客户端和服务器都同意的情况下，才会做session resuming。只要有任意一方怀疑 session 泄漏，或者证书过期/被吊销，就可以要求对端做完整的握手。 一个session的生命周期建议定位24小时。由于如果攻击者获得了 master_secret 就可以在session ID过期之前伪装成被泄漏者，所以要加一个生命期限制。 运行在不安全环境的应用程序，不应该把session ID写入持久存储。

5. 针对应用数据保护的攻击

master_secret 和 ClientHello.random 及 ServerHello.random 一起做 hash，来生成每个连接唯一的加密key和MAC key（就算是session resuming得到的连接，也是不同的）。

在CBC和stream cipher的情况下， 发送出去的数据，在发送前用MAC保护，来避免消息重放，避免篡改。 MAC根据 MAC key，序列号，消息长度，消息内容，固定字符串算出。 消息类型字段（content type）是必须的，来确保握手消息，ChangeCipherSpec消息，应用数据消息不会被混淆。 序列号用来确保删除包或者打乱包顺序的攻击无法得逞。 由于序列号是64位的，可以认为不会回绕。 从一方发给对端的消息，不能被插入对端发来的字节流中，这是用于两端使用不同的 MAC key。类似地，server write key 和 client write key相互独立。因此stream cipher的key只使用了一次，避免了类似问题。

如果攻击者获取了加密key，那么就可以解密所有的消息。 类似地，泄漏MAC key，会使攻击者可以篡改消息。

AEAD就简单了。

6. 显式 IV的安全性

如前文所述，TLS 1.0是把前一条消息的最后一个block，当作下一条消息的第一个IV的，这促成了2004年公开的 BEAST 攻击，后来就改成这种显式IV的更安全的方式了。

7. 加密和MAC组合模式的安全性

前文介绍CBC和AEAD时已有分析，此处略过。

8. DOS 攻击下的安全性

TLS容易遭受某些 DoS 攻击。例如，攻击者创建很多TCP连接，就可以让服务器忙于做 RSA 解密计算。然而，由于TLS运行在TCP之上，只要操作系统TCP栈的 SYN-ACK里seqnum是随机的，攻击者就无法隐藏自己的ip，这样就可以和一般的TCP连接一样做DOS防御。

由于TLS运行在TCP上，每个独立的连接都可能遭受一系列DOS攻击。尤其是，攻击者可以伪造RST包，来中断一条TCP+TLS连接。或者伪造部分TLS记录，导致连接阻塞挂起。不过这些攻击都是任何TCP协议都有问题，不是TLS特有的。

9.Session Ticket 的安全分析

Ticket必须: 1.有MAC （即 authenticated，不可篡改），2.加密（即保密）。

下面分析在各种攻击方法下的安全性。

1. 无效的Session

TLS协议要求当发现错误的时候，把TLS session变为无效。

这不会影响到ticket的安全性。

2. 窃取 Tickets

攻击者或者中间人，可能会窃取到ticket，并且尝试用来和server建立会话。 然而，由于ticket是加密过的，并且攻击者不知道密钥，窃取到的ticket无法使攻击者恢复会话。 TLS服务器必须使用强加密和MAC算法，来保护ticket。

3. 伪造 Tickets

一个恶意用户可能会伪造，或者篡改一个ticket，来恢复一个会话，来延长ticket的生命周期，或者假装成另一个用户。

然而，由于服务器使用了强的校验保护算法，比如带密码的 HMAC-SHA1 ，因此无法得逞。

4. DoS 攻击

推荐ticket 格式中的 key_name 字段帮助服务器有效地拒绝不是自己签发的票据。 因此，一个攻击者可能发送大量的ticket，让服务器忙于验证ticket。 然而，只要服务器使用了高效的加密和MAC算法，就不会有问题。（现实中，加密和MAC算法效率都极高，这根本不是问题）

5. 加密 Ticket 的key 的管理

加密ticket的key的管理，推荐的做法：

- key 应该用密码学安全的随机数生成器生成，按照RFC4086。
- key 和加密算法最少应该是 128 比特安全程度的。
- key 除了加密和解密ticket以外，不应该有其他用途。
- key 应该定期更换
- 当ticket格式更换，或者算法更换时，应该更换key

6. Ticket 的有效期

TLS服务器控制ticket的生命周期。服务器根据配置来决定可以接受的ticket生命周期。 ticket的生命周期可能会长于24小时。TLS客户端可能会接受到一个ticket生命周期的提示，当然，客户端本地的策略最终决定ticket保存多久。

7. 其他的 Ticket 格式和分发方法

如果没使用推荐的ticket格式，那必须小心地分析方案的安全性。尤其是，如果保密数据比如保密密钥传输给了客户端，那必须用加密方式传输，来防止泄露或篡改。

8. Identity Privacy, Anonymity, and Unlinkability

ticket的加密和加MAC，就保证了敏感信息不会泄露。

由于在ticket解密之前的TLS握手，无法隐藏客户端的特征，因此中间人可能根据相同的ticket被复用，发现相同的ticket属于相同的用户。TLS对这种情况不提供保证。

9. TLS扩展:

https://tools.ietf.org/html/rfc6066

几个比较重要的TLS扩展：

1. Server Name Indication (SNI) 由于在SNI提出之前，tls握手过程中没有字段标明客户端希望连接服务器端的哪个域名，这样如果一个服务器端口上有多个域名，服务器就无法给出正确的证书。随着ipv4地址空间紧张，这个问题越发突出。因此提出了SNI。
2. TLSEXT_ALPN 上层协议协商，就是在握手过程中，标明TLS里面是什么协议，例如 http2就是 h2
3. Maximum Fragment Length Negotiation 主要用于嵌入式环境，需要客户端发送。
4. Session Ticket Session Ticket，就是把session cache加密后存入客户端，这样服务器端就不需要任何存储了。

5. **TLSEXT_SAFE_RENEGOTIATION 重协商**
6. **Certificate Status Request: OCSP ，OCSP 主要是为了减少客户端查询 证书撤销列表（Ceritificate Revoke List)的网络调用，而提出的。**

10. TLS的配套： PKI体系

1. X.509 证书

X.509是PKI的一个标准，其中内容包括：

- **公钥证书**
- **证书撤销列表，CRL**
- **证书路径验证算法（CA/证书 链的格式）**

X.509使用ASN.1语法做序列化/反序列化

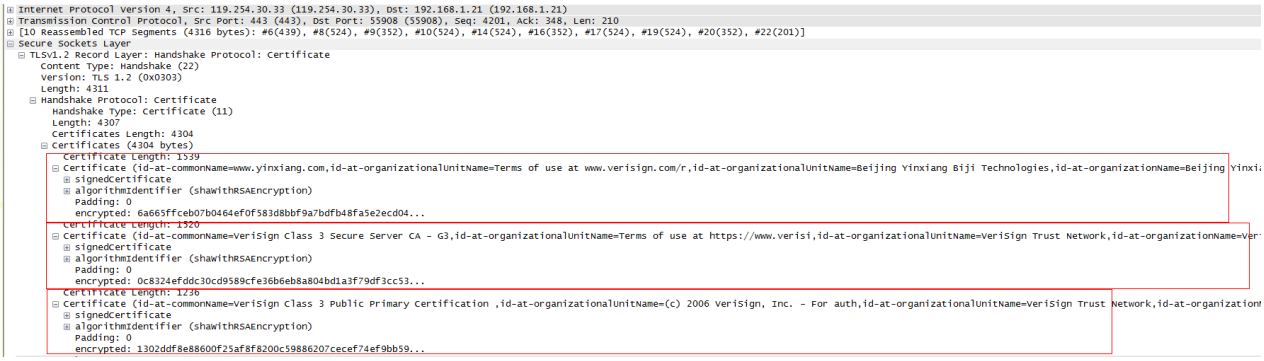
ASN1 就是一个数据序列化/反序列化格式，跟 protobuf 差不多，可以算作竞争对手。

DER 就是用 ASN1 序列化某些数据结构的格式。

PEM 就是 DER做base64，加上一些其他字段。

证书链，以一个或多个CA证书开头的证书的列表，其中:

- **每一个证书的 Issuer 和下一个证书的 Subject 相同**
- **每一个证书都被下一个证书的私钥签署**
- **最后一个证书是 根证书(“root CA”)，在TLS握手中不会被发送**



证书里面包含公钥，和其它一些字段（比如证书用途，有效期，签发者等等） x509.v3证书的字段：

mozilla的ca证书列表 https://www.mozilla.org/en-US/about/governance/policies/security-group/certs/

https://www.apple.com/certificateauthority/ca_program.html 苹果对CA提的要求：

1.CA必须取得完整的 WebTrust for Certification Authorities audit （WebTrust CA审计：http://www.webtrust.org/） 2.你的root CA证书必须为apple平台的用户提供广泛的商业价值。例如，一个组织内内部使用的证书不能被接受为root证书。 3.你签的证书必须含有可以公开访问的CRL地址。

Webtrust审计介绍： Webtrust是由世界两大著名注册会计师协会（美国注册会计师协会，AICPA和加拿大注册会计师协会，CICA）制定的安全审计标准，主要对申请对象的系统及业务运作逻辑安全性、保密性等共计七项内容进行近乎严苛的审查和鉴证。只有通过Webtrust国际安全审计认证，才有可能成为全球主流浏览器根信任的证书签发机构。

https://www.geotrust.com/ 的网站上右下角，有个图标: 

点开就可以看到 KPMG 对 geotrust 公司的 webtrust 审计报告: https://cert.webtrust.org/SealFile?seal=1567&file=pdf

2011年 荷兰CA公司DigiNotar颁发假google，Facebook，微软证书被发现，后发现被入侵，导致该公司破产。 http://www.cnbeta.com/articles/154375.htm

https://news.ycombinator.com/item?id=530600 CA公司签署一个证书的成本是0 。 CA公司的主要成本构成：审核 ，验证CSR成本，支持成本，法律成本(保险费用，担保费用)。 要进入各个浏览器的根证书列表，CA公司每年必须过 WebTrust 年度审计，是很大的开销。 一些浏览器厂商还会对植入根证书列表的CA收费。 基础设施开销，CRL 和 OCSP 服务器成本。 验证CSR：就是提交证书申请后，CA要做多项验证，越是高级的证书（比如EV）验证越麻烦。不固定开销，有些要花费很多人力和时间来完成。 法律开销：CA公司得买保险，保险费跑不了。 CA链费用：新开的CA公司要等5-10年，才会被普遍信任，才能广泛进入根证书链。要想加快点，就得给别的大牌CA公司掏钱，买次级证书。

2.现有PKI体系暴露出的问题

http://googleonlinesecurity.blogspot.com/2015/03/maintaining-digital-certificate-security.html

https://blog.mozilla.org/security/2015/04/02/distrusting-new-cnnic-certificates/

https://www.dfn-cert.de/dokumente/workshop/2013/FolienSmith.pdf

https://www.cs.utexas.edu/~shmat/shmat_ccs12.pdf

解决方案：

1. public key pin

https://developer.mozilla.org/en-US/docs/Web/Security/Public_Key_Pinning

2. HSTS

http://www.chromium.org/hsts 收录进chrome的默认HSTS列表： https://hstspreload.appspot.com/

11. TLS协议历史上出现过的漏洞，密码学常见陷阱

1. TLS的漏洞

漏洞分析很耗时间，这里总结一些资料，有兴趣的自己看吧。

虽然TLS的设计已经尽可能的严密，但是随着技术进步的滚滚车轮，历史上还是出现过很多漏洞， 可以参看这个rfc，做了总结：

Summarizing Known Attacks on Transport Layer Security (TLS) and Datagram TLS (DTLS)

还有这个文档： The Sorry State Of SSL

http://hyperelliptic.org/internetcrypto/OpenSSLPresentation.pdf

TLS 协议最近一些年被爆出过的设计缺陷，尤其是在用的最多的 AES-CBC 和 RC4 上。

AES-CBC 发现了:

1. padding oracle 攻击
2. BEAST 攻击
3. Lucky 13 攻击
4. TIME 攻击
5. POODLE攻击

2013 年, AlFardan发表了对 RC4 的一个攻击分析，展示如何恢复 RC4 传输的连接上的数据。这种恢复攻击利用了RC4的一些已知弱点，例如RC4最初的一些字节的显著统计特征。

最近几年，TLS的代码实现引起了安全研究者的关注，这导致了新漏洞不断发现。 2014年，OpenSSL 库爆出了好几个漏洞，例如 HeartBleed，还有 CVE-2014-6321 (Microsoft SChannel 的实现漏洞) 等.

TLS的问题:

• 很多问题是由于TLS使用了一些"史前时代"的密码学算法(- Eric Rescorla) • CBC 和 Mac-Pad-then-Encrypt • RSA-PKCS#1v1.5 的 RSA padding • RC4 的任何使用 • 很蠢的设计：临时 RSA 密钥协商，GOST 类CipherSuite，Snap Start 等 • 可怕的向后兼容要求，导致迟迟不能废弃一些老算法。

The Most Dangerous Code in the World: Validating SSL Certificates in Non-Browser Software

http://crypto.stanford.edu/~dabo/pubs/abstracts/ssl-client-bugs.html

https://www.cs.utexas.edu/~shmat/shmat_ccs12.pdf

Why Eve and Mallory Love Android An Analysis of Android SSL (In)Security

2. 密码学常见陷阱

先举几个加密协议被破解的例子，给大家助兴：

- 人人网使用256比特RSA加密登录密码，3分钟可破
- Flickr length extension attack 漏洞
- 分析whatsapp协议缺陷的一个文章

- 卫星电话的私有gmr-1/gmr-2加密算法被逆向并破解
- http://cryptofails.blogspot.ca/2013/07/cakephp-using-iv-as-key.html
- http://cryptofails.blogspot.ca/2013/07/saltstack-rsa-e-d-1.html

网上有一些资料，有兴趣自己看吧:

- https://www.schneier.com/essays/archives/1998/01/security_pitfalls_in.html
- https://www.schneier.com/essays/archives/1999/03/cryptography_the_imp.html
- http://www.lauradhamilton.com/10-cryptography-mistakes-amateurs-make
- http://www.cryptofails.com/
- http://cryptofails.blogspot.ca/

密码学常见应用错误 http://security.stackexchange.com/questions/2202/lessons-learned-and-misconceptions-regarding-encryption-and-cryptology

- 不要自己发明加密算法。Don't roll your own crypto.
- 不要使用不带MAC的加密 Don't use encryption without message authentication.
- 在拼接多个字符串做hash之前，要特别小心 Be careful when concatenating multiple strings, before hashing.
- 要特别小心使用的随机数生成器，确保有足够的熵 Make sure you seed random number generators with enough entropy.
- 不要重用 nonce 或者。IV Don't reuse nonces or IVs.
- 加密和MAC不要使用同样的key，非对称加密和签名不要使用相同的key Don't use the same key for both encryption and authentication. Don't use the same key for both encryption and signing.
- 不要使用ECB模式做对称加密 Don't use a block cipher with ECB for symmetric encryption
- Kerckhoffs定律，一个密码学系统的安全性必须建立在密码保密的基础上，其他都是公开的。Kerckhoffs's principle: A cryptosystem should be secure even if everything about the system, except the key, is public knowledge
- 不要把用户产生的密码作为加密的key。Try to avoid using passwords as encryption keys.
- 在密码学协议中，任何2条消息的密文都不应该一样。In a cryptographic protocol: Make every authenticated message recognisable: no two messages should look the same
- 不要把相同的key用在通信的2个方向上。Don't use the same key in both directions.
- 不要使用不安全的key长度。Don't use insecure key lengths.
- ...

13. 下一代TLS: TLS 1.3

tls 1.3的草案在 http://tlswg.github.io/tls13-spec/ 相比tls 1.2, 1.3改动巨大，这些改动对加密通信协议的一般设计也有重要启发。

TLS 1.3 的改动 值得关注的重大改进有：

- 0-RTT支持
- 1-RTT握手支持
- 改为使用HKDF做密钥拓展
- 彻底禁止RC4
- 彻底禁止压缩
- 彻底禁止aead以外的其他算法
- 去除aead的显式IV

- 去除了AEAD的AD中的长度字段
- 去除ChangeCipherSpec
- 去除重协商
- 去除静态RSA和DH密钥协商

移动互联网兴起之后，rtt延迟变得更重要，可以看到，tls 1.3 的各项改进，主要就是针对移动互联网场景的。

TLS 1.3 去掉了 ChangeCipherSpec ，这样record之上有3个协议：handshake，alert，application data

1. record层的密码学保护的改动

由于只保留了aead，所以不需要MAC key了。

aead的具体参数用法也有调整，前文有。

KDF 换成了标准的HKDF，有2种 tls_kdf_sha256, tls_kdf_sha384

2.handshake协议的改动

鉴于session ticket如此之好用，简直人见人爱，所以 TLS 1.3 直接把session ticket内置了，并改名叫 PSK

要注意的是，此 PSK 和 tls 1.2中一个很生僻的psk(见 rfc4279)并不是一回事。

综合考虑了 session resuming ，session ticket后， TLS 1.3 提出了3种handshake模式：

1. Diffie-Hellman （ 包含 DH 和 ECDH 两种，下文说到 ECDH 的地方，请自行脑补成 “ECDH/DH”）.
2. A pre-shared symmetric key (PSK) ，预先共享的对称密钥，此处用统一的模型来处理session resuming 和 rfc4279的psk
3. A combination of a symmetric key and Diffie-Hellman ，前两者合体

3.1-RTT 握手

首先，TLS 1.2 的握手有2个rtt，第一个rtt是 ClientHello/ServerHello，第二个rtt是 ClientKeyExchange/ServerKeyExchange， 之所以KeyExchange要放在第二个rtt，是由于tls1.2要支持多种密钥交换算法，和各种不同参数(比如 DH还是ECDH还是RSA，ECDHE用什么曲线，DH用什么群生成元，用什么模数，等等)，这些算法和参数都依赖第一个rtt去协商出来， TLS1.3大刀阔斧地砍掉了各种自定义DH群，砍掉了ECDH的自定义曲线，砍掉了RSA协商，密钥协商的算法只剩下不多几个，而且其实大家实际应用中基本都用 ECDH P-256，也没啥人用别的，所以干脆让客户端缓存服务器上一次用的是啥协商算法，把 KeyExchange直接和入第一个rtt，客户端在第一个rtt里直接就用缓存的这个算法发KeyExchange的公钥，如果服务器发现客户端发上来的算法不对，那么再告诉正确的，让客户端重试好了。 这样，就引入了 HelloRetryRequest 这个消息。

这样，基本没有副作用，就可以降到 1-RTT。 这是TLS 1.3 的完整握手。

显然，如果一个协议只有一种密钥协商算法，比如定死为 ECDH P-256，那一定可以做到 1-RTT

4. 有副作用的 0-RTT握手

0-RTT应该是受Google的QUIC协议的启发， 如果服务器把自己的 ECDH 公钥长期缓存在客户端，那么客户端就可以用缓存里的ECDHE公钥，构造一个电子信封，在第一个RTT里，直接就发送应用层数据了。 这个长期缓存在客户端的ECDH公钥，称为 半静态 ECDH 公钥（ semi-static (EC)DH share ）ECDH公钥通过 ServerConfiguration 消息发送给客户端。

这个0-rtt优化是有副作用的：

1. 0-RTT发送的应用数据没有前向安全性。
2. 跨连接可以重放0-RTT里的应用数据（任何服务器端无共享状态的协议，都无法做到跨连接防重放）
3. 如果服务器端 半静态 ECDH公钥对应的私钥泄露了，攻击者就可以伪装成客户端随意篡改数据了。

服务器在 ServerConfiguration 消息里把半静态 ECDH 公钥发送给客户端。 ServerConfiguration 值得关注一下：

```
struct { opaque configuration_id<1..2^16-1>; uint32 expiration_date;
NamedGroup group; opaque server_key<1..2^16-1>; EarlyDataType
early_data_type; ConfigurationExtension extensions<0..2^16-1>; }
ServerConfiguration;
```

其中的 expiration_date 是本 ServerConfiguration 最后的有效期限。 这个值绝对不允许大于7天。客户端绝对不允许存储 ServerConfiguration 大于7天，不管服务器怎么填这个值。

0-RTT 中的应用数据，放在 EarlyDataIndication 中发送，

TLS 1.3 还特意给 EarlyDataIndication 定义了一种 ContentType : early_handshake （共四种 alert(21), handshake(22), application_data(23), early_handshake(25) ）

5. Resumption 和 PSK

TLS 1.3 里面，把session resumption/session ticket 恢复出来的key，和 psk (rfc4279)， 统一在一个 handshake PSK 模式下处理。

PSK CipherSuite可以 把PSK和ECDHE结合起来用，这样是有前向安全性的。 也可以仅仅使用PSK，这样就没有前向安全性。

6. Key Schedule 过程的改动

TLS 1.3 中，综合考虑的 session ticket的各种情况后，提出了 ES，SS 两个概念，统一处理密钥协商的各种情况。 在各种handshake模式下，ES和SS的取值来源不同。

- Ephemeral Secret (ES)

每个连接新鲜的 ECDHE 协商得出的值。凡是从 ES 得出的值，都是前向安全的（当然，在 PSK only模式下，不是前向安全的）。

- **Static Secret (SS)**

  从静态，或者半静态key得出的值。例如psk，或者服务器的半静态 ECDH 公钥。

在各种 handshake 模式下：

Key Exchange	Static Secret (SS)	Ephemeral Secret (ES)
(EC)DHE (完整握手)	Client ephemeral w/ server ephemeral	Client ephemeral w/ server ephemeral
(EC)DHE (w/ 0-RTT)	Client ephemeral w/ server static	Client ephemeral w/ server ephemeral
PSK	Pre-Shared Key	Pre-shared key
PSK + (EC)DHE	Pre-Shared Key	Client ephemeral w/ server ephemeral
如上表所示：		

1. 完整的 1-RTT握手的时候， SS 和 ES 都是用的 ephemeral key ，这样是一定有前向安全性的。
2. 使用 0-RTT 的握手的时候，使用客户端的 ephemeral key 和 服务器端的半静态 ECDH 公钥生成 SS，
3. 纯 PSK，这种场景完全没有前向安全性，应该避免。
4. PSK + ECDHE，这种场景比较有意思，SS是用的Pre-Shared Key，没有前向安全性，ES 用的 ephemeral key，有前向安全性。

可以看到，相比 TLS 1.2 的 session ticket，TLS 1.3 中 的 PSK + ECDHE，是结合了 ES 的，这样就有了前向安全性，相对更安全。

和 TLS 1.2 不同的是，TLS 1.3的 master_secret 是使用 ES和SS 两个得出的。

```
HKDF-Expand-Label(Secret, Label, HashValue, Length) = HKDF-Expand(Secret, Label + '\0' + HashValue, Length) 1. xSS = HKDF(0, SS, "extractedSS", L) 2. xES = HKDF(0, ES, "extractedES", L) 3. master_secret = HKDF(xSS, xES, "master secret", L) 4. finished_secret = HKDF-Expand-Label(xSS, "finished secret", handshake_hash, L)
```

Traffic Key Calculation

加密流量用的key，在 TLS 1.3 里面称为 Traffic Key，由于多引入了一种ContentType，在不同的ContentType下，Traffic Key 并不相同。 如下表：

Record Type	Secret	Label	Handshake Hash
Early data	xSS	“early data key expansion”	ClientHello
Handshake	xES	“handshake key expansion”	ClientHello... ServerKeyShare
Application	master secret	“application data key expansion”	All handshake messages but Finished

要关注的是， Early Data 的 Traffic Key 是用 xSS 算出来的。也就是说，是用 Pre-Shared Key决定的。因此是没有前向安全性的。

在一个TLS 连接中，究竟是用哪种 handshake 模式，是由 CipherSuite 协商决定的。

三. TLS协议的代码实现

TLS的主要实现：

- **OpenSSL**
- **libressl**
- **boringssl(Google)**
- **libressl**
- **s2n(Amazon)**
- **nss(Mozilla)**
- **polarssl**
- **botan**
- **gnutls(gpl)**
- **cyassl**
- **go.crypto**

openssl 的 tls 协议实现有 6W 行，libressl 3.68W行， polarssl 1.29 W行， Botan 1.13 W行

openssl是其中代码最糟糕的（没有之一）。 openssl提供了的api都太过于底层，api设计的也很费解，而且严重匮乏文档。 请参考 《令人作呕的OpenSSL》

不幸的是，OpenSSL是用的最广泛的，是事实上的标准。

boringssl Google’s OpenSSL fork by Adam Langley (@agl__)

https://github.com/sweis/crypto-might-not-suck

四. TLS协议的部署与优化

这个方面网上的文章还是不少的，本文就简略一点。

全站https时代正在到来!， 移动互联网对人们生活的介入越来越深人，用户越来越多的隐私数据和支付数据通过网络传输，人们的隐私意识安全意识不断提高；运营商流量劫持，强行插入广告越来越引起反感。因此，各互联网大厂都开始切换到https。

例如，2015年3月百度全站切换到https，百度运维部的介绍文章：《全站 https 时代的号角：大型网站的 https 实践系列》。

不久后淘宝切了全站https，https://www.taobao.com/ http://velocity.oreilly.com.cn/2015/index.php?func=session&id=8

国外：由Snowden爆料，美国人发现NSA在大范围深度地监听互联网; 还有openssl连续被爆多个严重安全漏洞。之后近2年，各种加密通信协议，软件，项目开始热门，各大厂商开始关注密码协议，做数据加密，信息安全。(openssl资助，pfs被重视，)

Google的性能数据：

> "In January this year (2010), Gmail switched to using HTTPS for everything by default. .. In order to do this we had to deploy no additional machines and no special hardware. On our production frontend machines, SSL accounts for < 1% of the CPU load, < 10 KB of memory per connection, and < 2% of network overhead...
>
> If you stop reading now you only need to remember one thing: SSL is not computationally expensive any more."
>
> – Overclocking SSL blog post by Adam Langley (Google https://www.imperialviolet.org/2010/06/25/overclocking-ssl.html)

google的优化： https://bit.ly/gottls https://www.imperialviolet.org/2010/06/25/overclocking-ssl.html https://istlsfastyet.com/ https://www.ssllabs.com/downloads/SSL_TLS_Deployment_Best_Practices.pdf http://chimera.labs.oreilly.com/books/1230000000545/ch04.html

baidu的经验： http://op.baidu.com/2015/04/https-index/

aws的配置 http://docs.aws.amazon.com/ElasticLoadBalancing/latest/DeveloperGuide/elb-https-load-balancers.html

可以参考byron之前给出的一个介绍nginx配置的文章 Nginx下配置高性能，高安全性的https TLS服务，本人提供售后咨询服务，哈哈。

CipherSuite配置(Mozilla的权威配置) https://wiki.mozilla.org/Security/Server_Side_TLS

hardenedlinux的这个文档：SSL/TLS部署最佳实践v1.4: http://hardenedlinux.org/jekyll/update/2015/07/28/ssl-tls-deployment-1.4.html

全站切https，值得关注的一个点是cdn切https，如果cdn资源不使用cdn提供商的域名的话，之前会有私钥必须得交给cdn提供商的安全风险，但是幸运的是cloudflare提出了keyless ssl方案，解决了这个问题 https://github.com/cloudflare/keyless**，cdn切https应该可以借鉴。**

有时候我们会用wireshark之类的工具抓包，来调试http协议，但是切换到https后，都变成二进制密文了，直接抓包是行不通了，那怎么调试协议呢？ 有个简单的解决办法：小技巧:如何在wireshark里查看https的明文数据 参考 https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSS/Key_Log_Format

五. 更多的加密通信协议case：QUIC, iMessage，TextSecure, otr, ios HomeKit，libsodium

时间有限，下面有些协议就没有做详细的分析了，读者自己去看吧。

1. QUIC

QUIC = TCP+TLS+SPDY https://www.chromium.org/quic

其中的 crypto design文档是本文关注的。

http://network.chinabyte.com/162/13361162.shtml http://blog.chromium.org/2015/04/a-quic-update-on-googles-experimental.html 截止2015.04，从Chrome到Google server的流量的大概50% 是走的QUIC协议，而且还在不断增加。 据说减少了YouTube的30%的卡顿。

https://github.com/devsisters/libquic

QUIC值得借鉴的地方有：crypto算法选择，0-RTT的实现方法，证书压缩省流量

QUIC的crypto算法选择： 密钥交换算法只有2种：

```
// Key exchange methods const QuicTag kP256 = TAG('P', '2', '5', '6');
// ECDH, Curve P-256 const QuicTag kC255 = TAG('C', '2', '5', '5'); //
ECDH, Curve25519
```

对称加密只使用AEAD，并且只有2种：

```
// AEAD algorithms const QuicTag kNULL = TAG('N', 'U', 'L', 'N'); //
null algorithm const QuicTag kAESG = TAG('A', 'E', 'S', 'G'); // AES128
+ GCM-12 const QuicTag kCC12 = TAG('C', 'C', '1', '2'); // ChaCha20 +
Poly1305
```

证书类型2种，RSA证书， 和 RSA/ECDSA双证书

```
// Proof types (i.e. certificate types) const QuicTag kX509 = TAG('X', '5', '0', '9'); // X.509 certificate, all key types const QuicTag kX59R = TAG('X', '5', '9', 'R'); // X.509 certificate, RSA keys only
```

handshake的结果是为了协商出来下面这些参数：

```
// Parameters negotiated by the crypto handshake. struct NET_EXPORT_PRIVATE QuicCryptoNegotiatedParameters { // Initializes the members to 0 or empty values. QuicCryptoNegotiatedParameters(); ~QuicCryptoNegotiatedParameters(); QuicTag key_exchange; QuicTag aead; std::string initial_premaster_secret; std::string forward_secure_premaster_secret; // subkey_secret is used as the PRK input to the HKDF used for key extraction. std::string subkey_secret; CrypterPair initial_crypters; CrypterPair forward_secure_crypters; // Normalized SNI: converted to lower case and trailing '.' removed. std::string sni; std::string client_nonce; std::string server_nonce; // hkdf_input_suffix contains the HKDF input following the label: the // ConnectionId, client hello and server config. This is only populated in the // client because only the client needs to derive the forward secure keys at a // later time from the initial keys. std::string hkdf_input_suffix; // cached_certs contains the cached certificates that a client used when // sending a client hello. std::vector<std::string> cached_certs; // client_key_exchange is used by clients to store the ephemeral KeyExchange // for the connection. scoped_ptr<KeyExchange> client_key_exchange; // channel_id is set by servers to a ChannelID key when the client correctly // proves possession of the corresponding private key. It consists of 32 // bytes of x coordinate, followed by 32 bytes of y coordinate. Both values // are big-endian and the pair is a P-256 public key. std::string channel_id; // Used when generating proof signature when sending server config updates. bool x509_ecdsa_supported; // Used to generate cert chain when sending server config updates. std::string client_common_set_hashes; std::string client_cached_cert_hashes; };
```

37 38 39 40 41 42	

可以看到：QUIC内置支持sni 而且区分 initial_premaster_secret 和 forward_secure_premaster_secret。

先这样吧，后续再分析。

2. apple ios iMessage

iOS Security Guide : https://www.apple.com/business/docs/iOS_Security_Guide.pdf

Apple 的 iMessage系统的密码学安全机制设计，端到端加密，前向安全(PFS)，签名使用ECDSA P-256，非对称加密使用RSA 1280 bit，苹果自己维护一个 用户名–》公钥 的目录服务。

iMessage在注册时，给每个用户生成一对 RSA-1280 密钥用作非对称加密，一对 NIST P-256 ECDSA 密钥用作签名，2个私钥本地保存，公钥上传给Apple的目录服务器(IDS)。

当要发送消息的时候，根据接收方的用户名，从IDS里面找到RSA公钥 和 APNS 地址。然后随机生成 128 比特密钥，用 AES-CTR-128 加密要发送的消息，用接收方的 RSA 1280 公钥，使用 OAEP 填充加密 128比特aes密钥。然后拼接 aes密文和rsa密文，对结果使用发送方的 ECDSA 私钥，用sha1算一次数字签名。 然后把aes密文，rsa密文，数字签名拼接起来，发给 APNS 投递给接收方。

如果要发送大文件，就生成一个key，用 aes-ctr-256 加密文件，并计算一个sha1，然后把key和sha1放入消息里面发送。

> Apple iMessage is a messaging service for iOS devices and Mac computers. iMessage supports text and attachments such as photos, contacts, and locations.Apple does not log messages or attachments, and their contents are protected by end-to-end encryption so no one but the sender and receiver can access them. Apple cannot decrypt the data. ... When a user turns on iMessage on a device, the device generates two pairs of keys for use with the service: an RSA 1280-bit key for encryption and an ECDSA 256-bit key on the NIST P-256 curve for signing. The private keys for both key pairs are saved in the device's keychain and the public keys are sent to Apple's directory service (IDS), where they are associated with the user's phone number or email address, along with the device's APNs address. ...

> Users start a new iMessage conversation by entering an address or name. If the user enters a name, the device first utilizes the user's Contacts app to gather the phone numbers and email addresses associated with that name, then gets the public keys and APNs addresses from the IDS. The user's outgoing message is individually encrypted for each of the receiver's devices. The public RSA encryption keys of the receiving devices are retrieved from IDS. For each receiving device, the sending device generates a random 128-bit key and encrypts the message with it using AES in CTR

> mode. This per-message AES key is encrypted using RSA-OAEP to the public key of the receiving device. The combination of the encrypted message text and the encrypted message key is then hashed with SHA-1, and the hash is signed with ECDSA using the sending device's private signing key. The resulting messages, one for each receiving device, consist of the encrypted message text, the encrypted message key, and the sender's digital signature. They are then dispatched to the APNs for delivery. Metadata, such as the timestamp and APNs routing information, is not encrypted. Communication with APNs is encrypted using a forwardsecret TLS channel.
>
> APNs can only relay messages up to 4 KB or 16 KB in size, depending on iOS version. If the message text is too long, or if an attachment such as a photo is included, the attachment is encrypted using AES in CTR mode with a randomly generated 256-bit key and uploaded to iCloud. The AES key for the attachment, its URI (Uniform Resource Identifier), and a SHA-1 hash of its encrypted form are then sent to the recipient as the contents of an iMessage, with their confidentiality and integrity protected through normal iMessage encryption,

3. apple ios HomeKit

iOS Security Guide : https://www.apple.com/business/docs/iOS_Security_Guide.pdf

Apple的HomeKit，是 WWDC2014 上提出的 iot 智能家居开发平台 （iot啊，目前最火的概念啊，各种高大上啊）。 可以看到 HomeKit 作为一个全新的协议， 抛弃了历史遗留算法，直接采用了目前最先进的算法

HomeKit 密码学安全机制的设计： 使用Ed25519做 公钥签名/验证，使用 SRP(3072 bit) 做来在iOS设备和HomeKit配件之间交换密码并做认证，使用 ChaCha20-Poly1305做对称加密， 使用HKDF-SHA512做密钥拓展。每个session的开始用Station-to-Station 协议做密钥协商和认证， 随后使用Curve25519做密钥协商，生成共享key。

> HomeKit provides a home automation infrastructure that utilizes iCloud and iOS security to protect and synchronize private data without exposing it to Apple.... HomeKit identity and security are based on Ed25519 public-private key pairs. An Ed25519 key pair is generated on the iOS device for each user for HomeKit, which becomes his or her HomeKit identity. It is used to authenticate communication between iOS devices, and between iOS devices and accessories.... Communication with HomeKit accessories HomeKit accessories generate their own Ed25519 key pair for use in communicating with iOS devices. To establish a relationship between an iOS device and a HomeKit accessory, keys are exchanged using Secure Remote Password (3072-bit) protocol, utilizing an 8-digit code provided by the accessory's manufacturer and entered on the iOS device by the user, and then encrypted using ChaCha20-Poly1305 AEAD with HKDF-SHA-512-derived keys. The accessory's MFi certification is also verified during setup. When the iOS device and the HomeKit accessory communicate during use, each authenticates the other utilizing the keys exchanged in the above process. Each session is established using the Station-to-Station protocol and is encrypted with HKDF-SHA-512 derived keys based on per-session Curve25519 keys. This applies to both IP-based and Bluetooth Low Energy accessories.

4. TextSecure

TextSecure是一个端到端im加密通信协议，由WhisperSystem公司设计，目前whatsapp和WhisperSystem公司有合作，看网上资料，2014年11月开始，whatsapp已经开始使用TextSecure协议来做端到端加密(消息来源: https://whispersystems.org/blog/whatsapp/ http://www.wired.com/2014/11/whatsapp-encrypted-messaging/)。

TextSecure V2 协议： https://github.com/WhisperSystems/TextSecure/wiki/ProtocolV2 https://github.com/trevp/axolotl/wiki https://whispersystems.org/blog/advanced-ratcheting/

The TextSecure encrypted messaging protocol 是otr的一个衍生协议，主要有2个不同点: 1.ECDSA代替DSA 2.某些数据结构压缩

5. otr 协议

标准文档见：https://otr.cypherpunks.ca/Protocol-v3-4.0.0.html

open kullo协议 https://www.kullo.net/protocol/

Choice of algorithms Whenever we write about symmetric or asymmetric encryption or signatures, we mean the following algorithms, modes and parameters:

symmetric encryption: AES-256 in GCM mode asymmetric encryption: RSA-4096 with OAEP(SHA-512) padding asymmetric signatures: RSA-4096 with PSS(SHA-512) padding

6. libsodium/NaCL

libsodium/NaCL 值得重点介绍，大力推广 。 新的没有兼容包袱的系统，都值得考虑用 NaCL来代替 openssl。 libsodium是对NaCL的封装，NaCL大有来头，作者 DJB 是密码学领域的权威人物，chacha20，Curve25519 的作者 。 没有历史包袱的项目，强烈建议使用 libsodium/NaCL。

这篇文章介绍了NaCL和openssl相比的各方面改进 http://cr.yp.to/highspeed/coolnacl-20120725.pdf https://cryptojedi.org/peter/data/tenerife-20130121.pdf http://nacl.cr.yp.to/

7. Tox.im

一款实用NaCL的端到端加密im https://github.com/irungentoo/toxcore/blob/master/docs/updates/Crypto.md

8. CurveCP

CurveCP值得重点介绍， http://curvecp.org/security.html

> CurveCP的安全考量： Confidentiality and integrity server authentication? client authentication? replay attacks? man-in-the-middle attacks? passive forward secrecy? active forward secrecy? against traffic analysis? internet destination, exact timing, and approximate length of each packet that you send.

> Availability availability, i.e., to make denial-of-service attacks more difficult. Blind amplification Unauthenticated memory consumption CPU consumption

> Efficiency CPU overhead Network overhead without packet loss Latency without packet loss

> Decongestion

9. tcpcrypt

http://tcpcrypt.org/

10.noise

https://github.com/trevp/noise/wiki

11.tcpcrypt

http://tcpcrypt.org/

12. netflix MSL

http://techblog.netflix.com/2014/10/message-security-layer-modern-take-on.html

http://www.infoq.com/cn/news/2014/11/netflix-safe-communication

12.Amazon KMS 密钥管理服务 白皮书

https://d0.awsstatic.com/whitepapers/KMS-Cryptographic-Details.pdf

值得注意和借鉴的点：

- 对称加密算法选择了 AES-GCM-256
- 数字签名有2种：ECDSA，RSA，ECDSA 的曲线选择了 secp384r1 （P384），hash 算法选择了 SHA384RSA 选择2048位，签名体制选择 RSASSA-PSS，hash 算法选择了 SHA256
- 密钥协商，使用ECDH，选择曲线 secp384r1 (P384)，有2种用法one-pass ECDH.ECDHE
- 电子信封加密，KMS内置了电子信封。

电子信封就是，你预先知道对方的长期公钥，你有一个消息要发送给对方，所以你生成一个随机的 msgKey,然后 ciphertext = Encrypt(msgKey, message), 并且用对方的公钥加密 msgKey： encKey = Encrypt(k, msgKey), 最后把(encKey, ciphertext) 发给对方，这样，只有公钥对应私钥的拥有者才能打开信封。典型应用比如 OpenPGP。

其中的 one-pass ECDH，大概意思是： 发起方有一对长期使用的签名密钥对，发起方生成一对临时的 ECDH 密钥，用自己的长期签名密钥签署 临时ECDH公钥。对端有一对长期 ECDH 密钥，收到发起方发来的 ECDH 公钥后，验证签名，并且用自己的长期ECDH私钥和收到的公钥协商出共享密钥。 整个过程中，只是用了一对临时ECDH密钥，2对长期密钥。

ECDHE就是比较典型的ECDHE了，和TLS用法一样：双方都持有一对长期使用的签名密钥对，并拥有对方的签名公钥，然后分别生成一对临时ECDH密钥，用自己的签名私钥签署ECDH公钥，把得出的签名和ECDH公钥发给对方， 双方收到对方的ECDH公钥后，验证签名，通过后用对方的ECDH公钥和自己的ECDH私钥协商出共享密钥。DONE。

白皮书中还举了几个例子，

六. TLS协议给我们的启发 – 现代加密通信协议设计

在看了这么多的分析和案例之后，我们已经可以归纳出加密通信协议设计的普遍问题，和常见设计决策，

设计决策点：

1. 四类基础算法 加密/MAC/签名/密钥交换 如何选择？ 对称加密目前毫无疑问应该直接用aead，最佳选择就是 aes-128-gcm/aes-256-gcm/chacha20-poly1305了 数字签名/验证方案，如果是移动互联网，应该考虑直接放弃 RSA，考虑 P-256 的 ECDSA 公钥证书，或者更进一步的 ed25519 公钥证书。 密钥交换算法，目前最佳选择就是 curve25519，或者 P-256。
2. 对称加密算法+认证算法，如何选择？或者直接用aead?
3. 签名算法如何选择？RSA or ECDSA or Ed25519?
4. 考虑将来的算法调整，要加版本号机制吗？ 建议是加上，起码在密钥协商的步骤，要加上版本号。便于将来更新算法。
5. RSA用作密钥交换是一个好的选择吗？考虑PFS 建议直接放弃RSA，RSA服务器端性能比ECDSA更差，签名更大费流量，而且没有前向安全性，给私钥保管带来更大风险。
6. 自建PKI，是个好的选择吗？crl如何解决？ 自建PKI可以做到更安全，比如简单的客户端内置数字签名公钥。可是当需要紧急吊销一个证书的时候，只能通过紧急发布新版客户端来解决。
7. 必须用糟糕的openssl吗？or something better？crypto++,botan, nacl/libsodium, polarssl？libsodium: ed25519+curve2519+chacha20+poly1305
8. 重放攻击如何解决？某种seq？或者nonce如何生成?
9. 握手过程被中间人篡改的问题怎么解决?
10. 性能：私钥运算的cpu消耗可以承受吗？加上某种cache？ 要解决私钥运算的高cpu消耗，必然就需要 session ticket/session id 这种cache机制。显然session ticket 更好
11. 延迟：密钥协商需要几个rtt？最少多少？加上cache后？和tcp对比如何
12. TLS的性能(主要指服务器cpu消耗)还有空间可以压榨吗？我能设计一个性能更牛逼的吗?

七. 附录：密码学基础概念

本文已经很长了，基础概念的内容更多，再展开介绍就太长了，下面就列一下点，贴一下参考资料，就先这样,以后再说吧。

当然，最好的资料是下面列的书。

1. 块加密算法 block cipher

AES 等

《AES后分组密码的研究现状 及发展趋势》 http://www.ccf.org.cn/resources/1190201776262/2010/04/15/019026.pdf

aead的介绍（作者是大神） https://www.imperialviolet.org/2015/05/16/aeads.html

3种组合方式之争 http://www.thoughtcrime.org/blog/the-cryptographic-doom-principle/

CBC模式+MAC-then-encrypt的padding oracle 攻击, tls POODLE 漏洞 http://drops.wooyun.org/papers/3194 https://defuse.ca/blog/recovering-cbc-mode-iv-chosen-ciphertext.html

128 bit 和 256 bit key size之争 https://www.schneier.com/blog/archives/2009/07/another_new_aes.html

nist 对 aes gcm 的技术标准，官方权威文档： http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/proposedmodes/gcm/gcm-spec.pdf

一个gcm的调用范例 https://github.com/facebook/conceal/blob/master/native/crypto/gcm_util.c

DES 1天之内破解DES（2008年） http://www.sciengines.com/company/news-a-events/74-des-in-1-day.html

iPhone 5S开始，A7芯片也有了aes硬件指令 (ARMv8 指令集)，有825%的性能提升： http://www.anandtech.com/show/7335/the-iphone-5s-review/4

2. 流加密算法 stream cipher

RC4，ChaCha20 等

序列密码发展现状 http://www.ccf.org.cn/resources/1190201776262/2010/04/15/019018.pdf

rc4 : http://www.rc4nomore.com/

[RC4加密已不再安全，破解效率极高（含视频）] http://www.freebuf.com/news/72622.html

3. Hash函数 hash funtion

MD5，sha1，sha256，sha512 , ripemd 160，poly1305 等

MD5被碰撞: http://natmchugh.blogspot.com/2014/10/how-i-created-two-images-with-same-md5.html

http://blog.avira.com/md5-the-broken-algorithm/

4. 消息验证码函数 message authentication code

HMAC-sha256，AEAD 等

为什么要用MAC http://www.happybearsoftware.com/you-are-dangerously-bad-at-cryptography.html

Flickr的漏洞案例： http://netifera.com/research/flickr_api_signature_forgery.pdf

http://www.ietf.org/rfc/rfc2104.txt

5. 密钥交换 key exchange

DH，ECDH，RSA，PFS方式的（DHE，ECDHE）等

https://blog.cloudflare.com/a-relatively-easy-to-understand-primer-on-elliptic-curve-cryptography/

关于 前向安全性(Perfect Forward Secrecy) http://vincent.bernat.im/en/blog/2011-ssl-perfect-forward-secrecy.html

http://www.cryptopp.com/wiki/Elliptic_Curve_Cryptography

google对openssl里面的椭圆曲线的优化： http://static.googleusercontent.com/media/research.google.com/en//pubs/archive/37376.pdf

http://www.math.brown.edu/~jhs/Presentations/WyomingEllipticCurve.pdf

ripple从nistp256k1曲线迁移到ed25519 https://ripple.com/uncategorized/curves-with-a-twist/

openssh 6.5 开始支持 ed25519, curve25519, chacha20-poly1305 http://www.openssh.org/txt/release-6.5

6. 公钥加密 public-key encryption

RSA，rabin-williams 等

RSA入门必读（斯坦福，普渡的课件）： http://crypto.stanford.edu/~dabo/courses/cs255_winter07/rsa.ppt https://engineering.purdue.edu/kak/compsec/NewLectures/Lecture12.pdf

PKCS1 标准，应用RSA必读： https://www.ietf.org/rfc/rfc3447

RSA 的公钥为什么比AES的key长？ http://crypto.stackexchange.com/questions/8687/security-strength-of-rsa-in-relation-with-the-modulus-size

http://cryptofails.blogspot.ca/2013/07/saltstack-rsa-e-d-1.html

使用什么padding？ OAEP，为什么不要用PKCS V1.5

http://stackoverflow.com/questions/2991603/pkcs1-v2-0-encryption-is-usually-called-oaep-encryption-where-can-i-confirm-i

http://crypto.stackexchange.com/questions/12688/can-you-explain-bleichenbachers-cca-attack-on-pkcs1-v1-5 http://en.wikipedia.org/wiki/Adaptive_chosen-ciphertext_attack

PKCS #1 – #15标准协议官方网站： http://www.emc.com/emc-plus/rsa-labs/standards-initiatives/public-key-cryptography-standards.htm http://arxiv.org/pdf/1207.5446v1.pdf

blinding 一种实现上的技术，用来解决 timing 侧通道攻击的问题 https://en.wikipedia.org/wiki/Blinding_(cryptography) http://crypto.stanford.edu/~dabo/papers/ssl-timing.pdf

Twenty Years of Attacks on the RSA Cryptosystem: http://crypto.stanford.edu/~dabo/papers/RSA-survey.pdf

电子信封(digital envelope) http://www.emc.com/emc-plus/rsa-labs/standards-initiatives/what-is-a-digital-envelope.htm

在openssl的evp接口中有直接支持： https://wiki.openssl.org/index.php/EVP_Asymmetric_Encryption_and_Decryption_of_an_Envelope

7. 数字签名算法 signature algorithm

RSA，DSA，ECDSA (secp256r1 , ed25519) 等

三大公钥体制：RSA，DSA，ECDSA RSA目前是主流，占据绝大多数市场份额 DSA已经被废弃 ECDSA是未来的趋势，例如bitcoin就用ECDSA https://blog.cloudflare.com/ecdsa-the-digital-signature-algorithm-of-a-better-internet/ https://blog.cloudflare.com/a-relatively-easy-to-understand-primer-on-elliptic-curve-cryptography/

8. 密码衍生函数 key derivation function

TLS-12-PRF(SHA-256) , bcrypto，scrypto，pbkdf2 等

hkdf： http://tools.ietf.org/html/rfc5869 https://cryptography.io/en/latest/hazmat/primitives/key-derivation-functions/

9. 随机数生成器 random number generators

/dev/urandom 等

现代密码学实践指南[2015年]

八. 参考文献：

TLS/SSL 相关RFC及标准

- TLS 1.2
- TLS 1.3 draft specification
- AES GCM for TLS
- ECC cipher suites for TLS
- TLS extensions
- Application-Layer Protocol Negotiation Extension
- X.509 PKI
- X.509 PKI and CRLs
- 美国国家标准局NIST 的密码学标准
- NIST SP 800-90A
- nsa 的 SuiteB 密码学标准
- TLS on wikipedia

协议分析文章

- http://www.root.org/talks/TLS_Design20071129_2.pdf
- 20 Years of SSL/TLS Research An Analysis of the Internet's Security Foundation
- https://www.slideshare.net/yassl/securing-data-in-transit
- http://security.stackexchange.com/questions/20803/how-does-ssl-tls-work
- SSL/TLS in Detail
- SSL/TLS

- The Sorry State Of SSL
- What's the matter with TLS?
- http://blog.csdn.net/CaesarZou/article/details/9331993
- <tools.ietf.org/html/rfc4210>)
- X.509 PKI and CRLs
- Layman's Guide to ASN.1

实际部署调优相关

- https://bit.ly/gottls
- https://istlsfastyet.com/
- https://www.imperialviolet.org/
- https://letsencrypt.org/
- http://chimera.labs.oreilly.com/books/1230000000545/ch04.to/crypto.html
- RSA Conference 2015 : New Trends In Cryptographic Algorithm Suites Used For TLS Communications

密码学相关

- Stanford Cryptography open course
- crypto101，一本很棒的开源电子书
- 现代密码学理论与实践 - 毛文波
- 现代密码学:原理与协议 - Katz and Lindell
- "Modern Crypto: 15 Years of Advancement in Cryptography" – 2015 defcon 大会Steve Weis 演讲
- 强烈建议不要看90年代的书，普遍严重过时，比如《应用密码学：协议、算法与C源程序（原书第2版）》
- DJBs crypto page
- DJBs entropy attacks
- Cryptographic Right Answers
- http://www.slideshare.net/yassl/securing-data-in-transit
- Schneier 关于密码学2010年现状的评述
- http://security.stackexchange.com/questions/2202/lessons-learned-and-misconceptions-regarding-encryption-and-cryptology
- http://chargen.matasano.com/chargen/2009/7/22/if-youre-typing-the-letters-a-e-s-into-your-code-youre-doing.html
- http://kodu.ut.ee/~swen/publications/articles/laur-thesis-binded.pdf
- https://www.enisa.europa.eu/activities/identity-and-trust/library/deliverables/study-on-cryptographic-protocols
- https://github.com/sweis/crypto-might-not-suck
- Cryptographic Best Practices in the Post-Snowden Era
- Crypto War
- 52 Things People Should Know To Do Cryptography
- http://bristolcrypto.blogspot.com/
- https://www.schneier.com/
- https://www.imperialviolet.org/2015/05/16/aeads.html
- https://crypto.stanford.edu/~dabo/cryptobook/draft_0_2.pdf

- http://saweis.net/pdfs/weis-modern-crypto-defcon-2015.pdf

相关开源项目

- **GoTLS - go语言自己搞的 TLS 协议实现**
- **OpenSSL - 事实上的标准**
- **LibreSSL - OpenBSD搞的OpenSSL的分支，代码可读性大大提高**
- **BoringSSL - Google Security team 维护的OpenSSL分支**
- **NSS - Mozilla 维护的TLS协议实现**
- **s2n - Amazon搞的tls协议实现**
- **MiTLS , TLS Attacks**
- **NaCL and libsodium**
- **spiped**