深入理解SSL/TLS技术内幕

秦青　2016.12.15

内容

- 协议介绍
- 协议历史
- 协议栈
- 会话与连接
- 记录协议操作
- 子协议交互
- TLS扩展
- 密码套件
- 密码操作
- 协议版本间的差异
- 安全性分析

协议介绍

- 什么是SSL/TLS

SSL的全称是Secure Socket Layer（安全套接字层），由Netscape公司开发的主要用于Web的安全传输协议；TLS的全称是Transport Layer Security（传输层安全），是SSL的改进版本，由TLS工作组维护

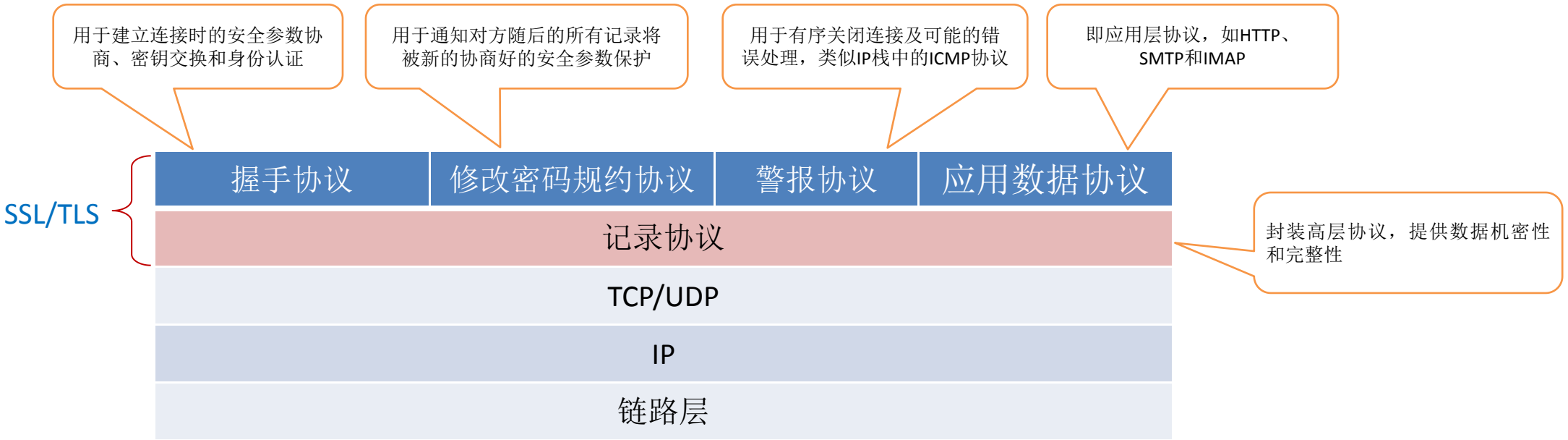
- 为什么需要SSL/TLS

互联网的核心是IP和TCP协议，但最初设计时，很少考虑到安全性，因此IP和TCP本质上是不安全的。为了在不安全的基础设施上提供安全通信，便产生了SSL/TLS。安全不是SSL/TLS的唯一目标，实际上有以下四个目标

协议历史


SSL 2.0
1994年11月发布，有严重的安全问题，失败的协议，已退出历史舞台
SSL 3.0
1995年年底发布，见RFC 6101，完全重新的设计，该设计沿用至今
1996年5月，TLS工作组成立
将SSL迁移至IETF，并更名为TLS
TLS 1.0
1999年1月发布，见RFC 2246，与SSL 3.0相比，修改不大
TLS 1.1
2006年4月发布，见RFC 4346，修复了一些关键的安全问题，重要改进是集成了2003年6月发布的TLS扩展
TLS 1.2
2008年8月发布，见RFC 5246，支持了已验证加密，删除了所有硬编码的安全基元，使协议完全弹性化
下一版本
正在开发中，目标是简化设计、除去安全性较弱或不太需要的功能，并且提升性能

协议栈



SSL/TLS处于协议栈的传输层TCP/UDP之上，由两层协议构成，下层为记录协议，上层为握手协议、修改密码规约协议和警告协议，这三种子协议基于记录协议实现，而应用数据协议则独立于记录协议

会话与连接

- 定义

会话（Session）：指客户端和服务器间的一个关联。由握手协议创建，它确定了一组安全参数，是有状态的。一个Session可被多个连接共享，从而避免为每个连接协商新的安全参数带来的昂贵开销。一个通信实体可以有多个并行的Session

连接：与传输层协议的端到端连接（如TCP连接）关联，每个连接都与一个Session关联

状态：实际上为一组参数的集合，每个Session和连接都有一个独立的状态

- 状态

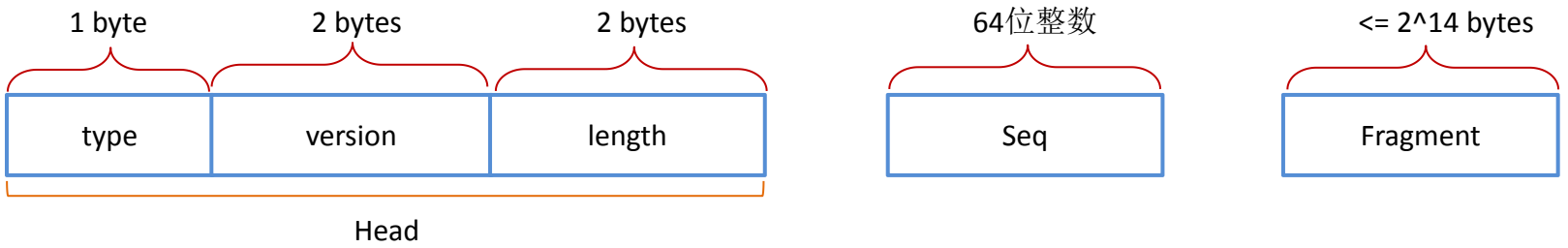
包括了下列参数，表现为当前状态和未决状态，每种又分读状态和写状态。当客户端或服务器收到一个**ChangeCihperSpec**消息时，则更新当前读状态为未决读状态；当发出一个**ChangeCihperSpec**消息时，则更新当前写状态为未决写状态

会话状态	Session ID	字节序列，由服务器产生用来唯一标识活跃或可恢复的会话状态的会话ID
	Peer certificate	对方的X509.v3证书，可能为空（例如匿名认证）
	Compression method	压缩数据的算法
	Cipher spec	密码套件，包括数据加密算法和MAC算法，以及算法相应的参数
	Master secret	在客户端和服务器共享的长48字节的主密钥
	Is resumable	一个标识，表示这个会话是否可用于初始化一个新的连接
连接状态	Server and client random	字节序列，由客户端和服务器为每个连接选择的随机数
	Server write MAC secret	服务器在发送数据时，用于计算MAC的密钥
	Client write MAC secret	客户端在发送数据时，用于计算MAC的密钥
	Server write key	服务器加密以及客户端解密数据的密钥
	Client write key	客户端加密以及服务器解密数据的密钥
	Initialization vectors	简称IV集，在CBC模式中用到的初始向量，每个密钥维持一个IV。由握手协议初始化，此后每个记录的最后一个密文块用作下个记录的IV

记录协议操作

- 操作数据
- 操作概要
- 流加密方式
- 分组加密方式
- 已验证的加密(AEAD)方式

- # 操作数据



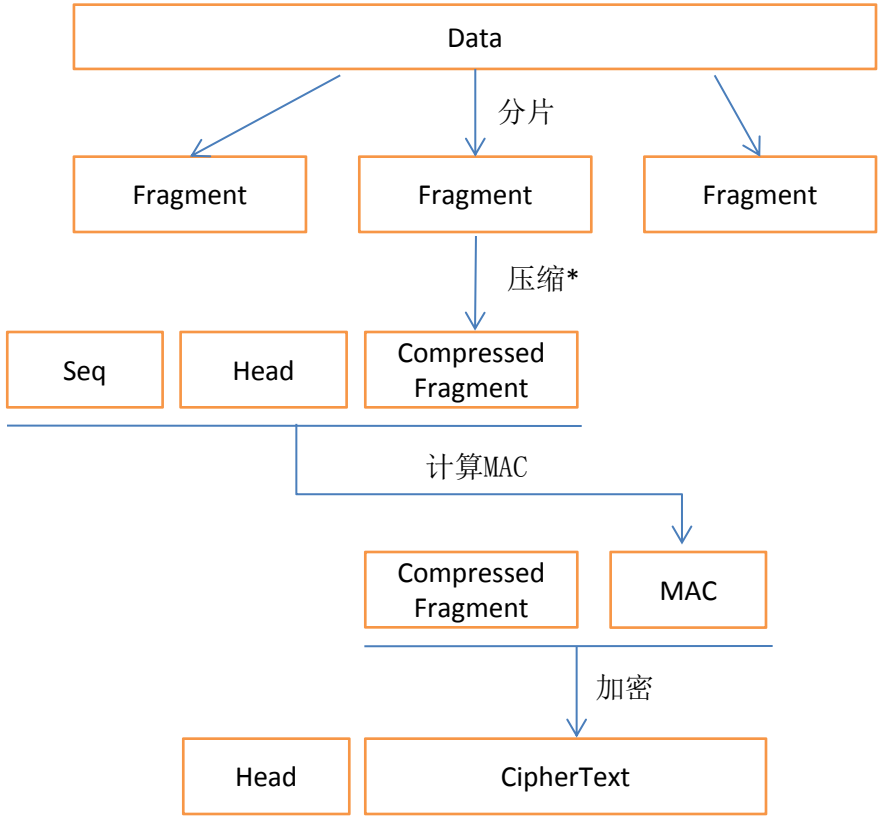
- **type**：含change_cipher_spec=20（改变密码规约）、alert=21（警报）、shandshark=22（握手）和application_data=23（应用数据）4种类型
- **version**：由高版本major和低版本minor组成，每个1字节
- **length**：数据净荷长度，不包括Head
- **Seq**：消息序列号，当每个连接上收到或发出消息时递增，若收到**ChangeCihperSpec**消息时，则重设为0
- **Fragment**：记录层分片，不应超过2^14字节

- # 操作概要

在发送SSL/TLS消息前，需经过分片、压缩、计算MAC和加密4个阶段，收到消息后则进行相应的逆操作即解密、验证MAC、解压和重组

- **分片**：若应用层数据长度大于记录层指定的分片长度，则进行分片
- **压缩**：按密码套件指定的压缩方法进行压缩，压缩后不应超过2^14 + 1024字节，一般应用时都没启用压缩
- **计算MAC**：按密码套件指定的MAC算法，传入Seq、Head和Fragment，其中Head中的净荷长度值为压缩后的大小
- **加密**：按密码套件指定的加密算法进行加密，加密后不应超过2^14+2048字节

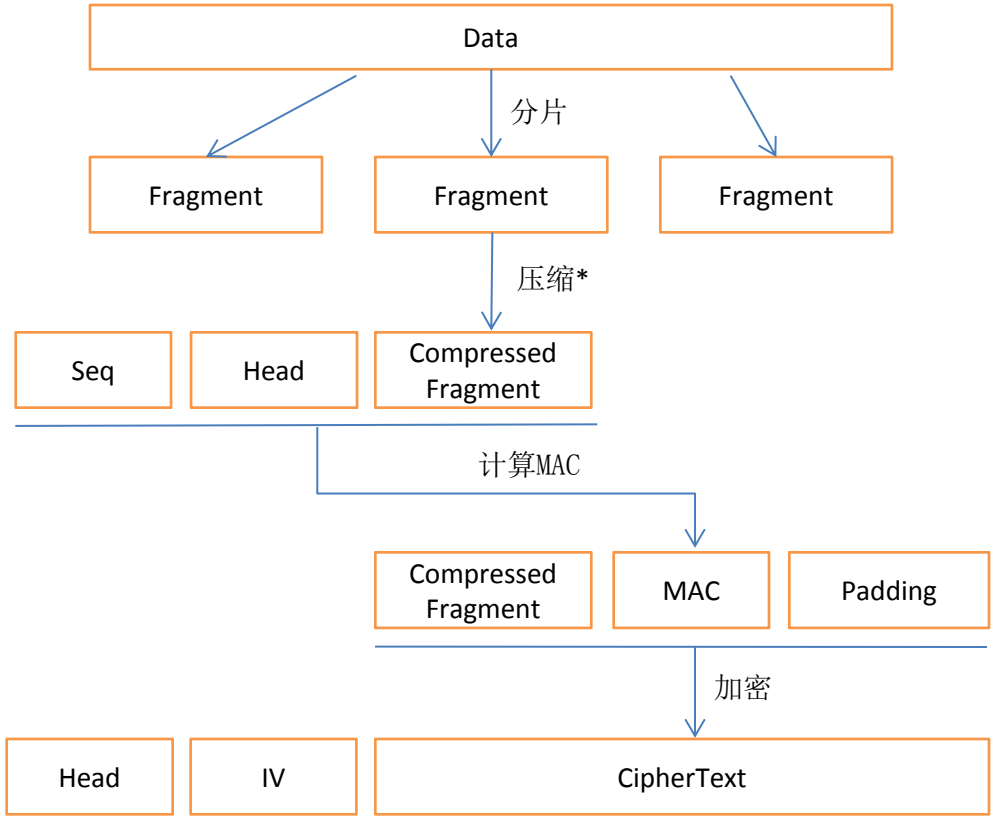
- 流加密方式



Data	应用层数据
CompressedFragment	压缩后的分片
CipherText	密文
*	可选

流加密不使用IV和填充，加密一个记录终止（最末位）时的内部状态决定了随后记录的加密。常用算法是RC4

- 分组加密方式

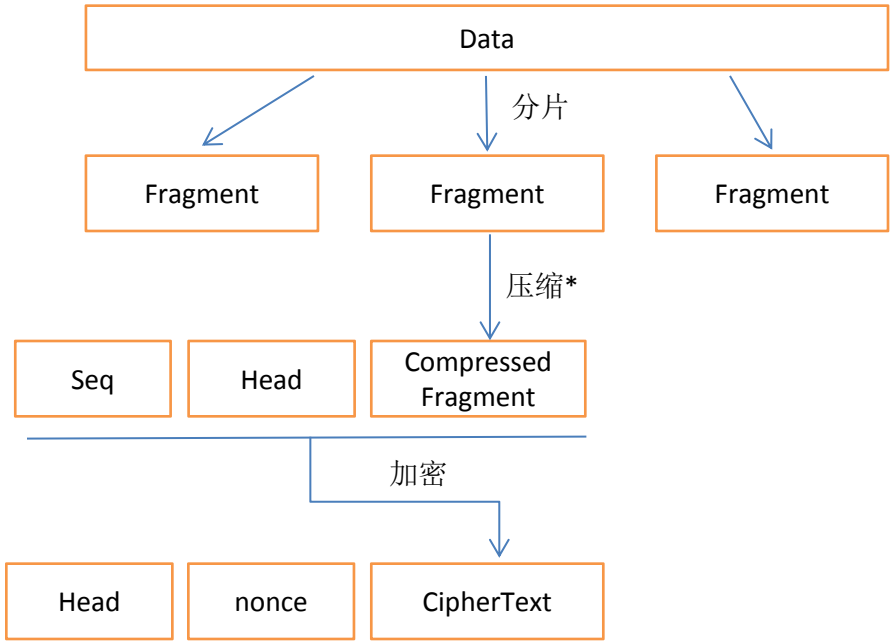


Data	应用层数据
CompressedFragment	压缩后的分片
Padding	填充
IV	初始向量
CipherText	密文
*	可选

先扩充消息使其长度为加密块大小（由具体加密算法决定）的整数倍（填充块最后1字节为所填充的字节数），再加密，最后在CipherText前附加Head和IV，其中Head中的净荷长度为IV和CipherText的总大小

• 已验证的加密(AEAD)方式

已验证的加密全名是使用关联数据的已验证加密(authenticated encryption with associated data)，简称AEAD，它将加密和完整性合二为一，是当前TLS中可用的加密模式中最好的一种



Data	应用层数据
CompressedFragment	压缩后的分片
CipherText	密文
*	可选

不用填充和IV，而用一个特殊唯一的64位值nonce（加密通信中仅使用一次的密钥），同时将Seq和Head作为完整性验证依据的额外数据交给算法，其中Head中净荷长度为压缩后分片的大小；加密完成后在nonce和CipherText前附加Head和nonce，其中Head中净荷长度为nonce和CipherText的总大小。AEAD避免了分组加密方式中的问题（填充预示攻击和可预测IV的明文选择攻击）

子协议交互

- 修改密码规约协议
- 警报协议
- 握手协议
- 连接关闭

• 修改密码规约协议

数据结构和交互过程描述如下

```
struct {
    enum { change_cipher_spec(1), (255) } type;
}ChangeCipherSpec;
```

新建会话时

1. 客户端在密钥交换和证书校验（若有）消息后，发送ChangeCipherSpec消息
2. 服务器收到后，若成功处理了密钥交换和证书认证（若有），则响应ChangeCipherSpec消息

恢复会话时

1. 服务器在响应Hello消息后、Finished消息前，发送ChangeCipherSpec消息
2. 客户收到后，若检验握手完整，则在Finished消息前响应ChangeCipherSpec消息

异常情况

若失序收到ChangeCipherSpec消息，则发送unexpected_message警报消息

• 警报协议

当一个SSL/TLS连接要关闭或检测到错误时，则发送相应的警报消息，数据结构描述如下

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

enum { close_notify(1),
       unexpected_message(10),
       bad_record_mac(20),
       ...
       (255)
     } AlertDescription;

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

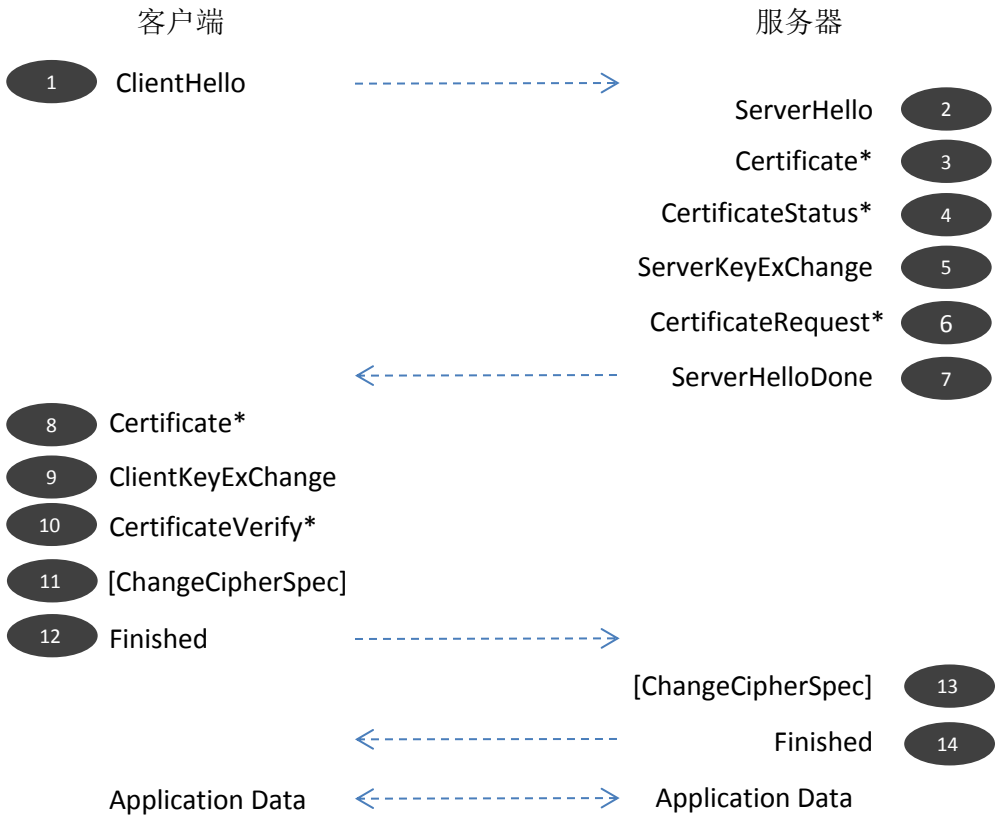
- 警报消息包括2个字节，第1个表示级别，有warning和fatat 两种；第2个表示描述
- fatal级别的alert消息导致连接立即终止，此时该会话的其它连接可能继续，但是Session ID应无效，以阻止失败的会话用于建立新的连接
- 对于warning消息，接收方可以自行处理；而对于fatal消息，一定要当做fatal消息处理

握手协议

- 完整握手
- 简短握手
- 客户端握手消息
- 服务器握手消息
- 共用的握手消息

• 完整握手

每一个SSL/TLS连接都以握手开始来协商会话。若客户端此前没有与服务器建立会话，则执行一次完整的握手



- 带*的表示该消息可选，仅当某时候才发送；带[]的表示ChangeCipherSpec协议消息，不属于握手协议中的消息
- 在所有握手消息中，12和14是密文，其它为明文
- 从上图可见，经2个RTT后，开始应用数据传输

1. 客户端发出握手请求，包括下列参数：支持的协议最高版本号、随机数、会话ID、密码套件列表（密钥交换和认证算法、对称加密算法、MAC算法）和压缩方法列表。在完整握手中，会话ID通常为0，表示新建会话，由服务器返回具体的会话ID

2. 服务器响应握手请求，包括两端都支持的协议最高版本号、一个不同于客户端的随机数、对应该连接的会话ID、从客户端密码套件列表中选定的某个密码套件、从客户端压缩方法列表选定的某个压缩方法

3. 服务器发送证书链，仅当需验证服务器时

4. 服务器响应证书吊销查询，仅当两端支持status_request扩展时，紧跟Certificate消息，该消息适用于TLS1.0以后的协议中

5. 服务器根据选择的密钥交换算法，发送生成主密钥所需的相关参数，对于RSA和DH算法，还包括一个对应参数的签名

6. 服务器发送证书请求，要求客户端提供证书，仅当不允许匿名客户端时

7. 服务器通知自己完成了协商过程，并等待客户端的响应

8. 客户端发送证书链，仅当服务器需验证客户端时，即服务器发送过CertificateRequest消息

9. 客户端根据协商好的密钥交换算法，发送生成主密钥所需的相关参数，注意与ServerKeyExchage消息不同，这没有对应参数的签名，当需要签名时，放在后面的CertificateVerify消息中

10. 客户端发送显式的证书校验消息，仅当需验证客户端并且客户端证书支持签名时，紧跟ClientKeyExChange消息，所签名的握手消息中不包括本消息

11. 客户端切换加密方式并通知服务器

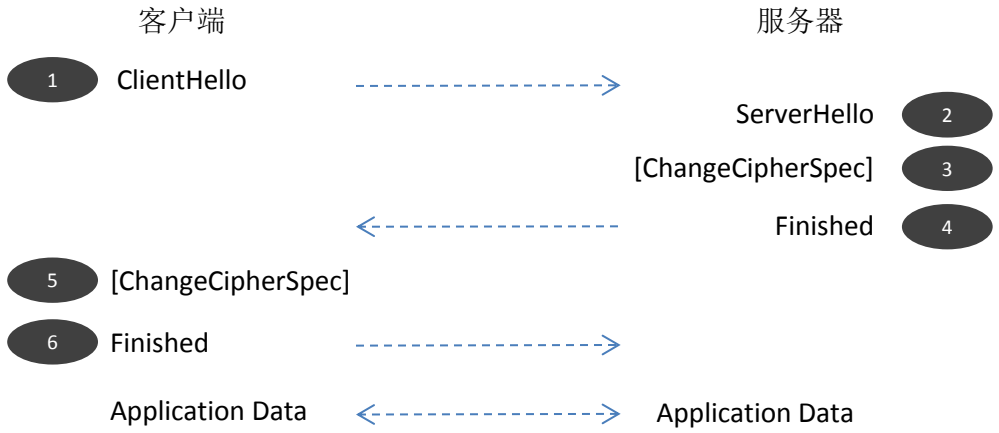
12. 客户端计算发送过和接收到的握手消息的MAC并发送，但不包括本消息，由于安全参数至此都已协商好，因此该消息是加密了的；然后等待服务器的响应

13. 服务器切换加密方式并通知客户端

14. 服务器计算发送和接收到的握手消息的MAC并发送，但不包括本消息，同理，该消息也是加密了的

简短握手

若已有会话，则可能恢复会话执行一次简短的握手



1. 客户端发出握手请求，该ClientHello消息中会话ID指定为期望被恢复的会话ID，也就是先前某次握手中服务器响应ServerHello消息中指定的会话ID

2. 服务收到请求后，先在会话缓存中查找指定的会话ID，若找到，则响应ServerHello消息，其中会话ID与指定的会话ID相同；若没找到，则生成一个新的会话ID，此后执行一个完整握手

3. 服务器切换加密方式并通知客户端

4. 服务器计算发送和接收到的握手消息的MAC并发送，但不包括本消息，由于安全参数至此都已协商好，因此该消息是加密了的

5. 客户端切换加密方式并通知服务器

6. 客户端计算发送和接收到的握手消息的MAC并发送，但不包括本消息，同理，该消息也是加密了的

- 在所有握手消息中，4和6是密文，其它为明文
- 从上图可见，简短握手经1个RTT后，就开始应用数据传输
- 服务器发送ServerHello消息后，使用之前协商的主密钥生成一套新的密钥，客户端收到已恢复的消息后，也进行同样的操作。虽共享的主密钥没变，但生成密钥需要两端的随机数，因每次握手时的随机数不同，故生成的密钥也不同

• 客户端握手消息

```
struct {
   ProtocolVersion  client_version;
   Random  random;
   SessionID  session_id;
   CipherSuite  cipher_suites[];
   CompressionMethod  compression_methods[];
   select  (extension_present) {
      case false:  struct { };
      case  true:  Extension  extensions[];
   };
}ClientHello;
```

- 必须被发送，当客户端发起握手或收到HelloRequest后
- 若服务器不支持对应的扩展功能，则客户端可能终止握手；否则按扩展格式解析扩展数据，若格式不匹配，则响应fatal 类型的decode error警报
- 客户端发送该消息后，期望服务端返回ServerHello，其它消息（HelloRequest除外）则当作fatal error处理

```
struct {
  ...
}ClientKeyExChange;
```

详见密钥交换与认证

```
struct {
   Signature  handshark_messages_signature;
}CertificateVerify;
```

- 若发送过Certificate，且证书支持签名，则发送该消息，用于向服务器验证自己的身份，表示拥有与之前所发送证书中的公钥对应的私钥
- 包含一个到该消息为止（但不包括该消息）的所有握手消息的签名

• 服务器握手消息

```
struct {
   ProtocolVersion  client_version;
   Random  random;
   SessionID  session_id;
   CihperSuite  cipher_suite;
   CompressionMethod  compression_method;
   select  (extension_present) {
      case false:  struct { };
      case  true:  Extension  extensions[];
   };
}ServerHello;
```

- 必须被发送，当收到ClientHello后，且选定了可接受的密码套件
- 若服务器不支持与客户端相同的版本，可提供某个版本以期待客户端接受
- 该消息与ClientHello类似，只是CipherSuite和CompressionMethod字段为一个值

```
struct {
  ...
}ServerKeyExChange;
```

详见密钥交换与认证

```
struct {
  ClientCertificateType  certificate_types[];
  SignatureAndHashAlgorithm  signature_hash_algorithms[];
  DistinguishedName   certificate_authorities[];
}CertificateRequest;
```

- 用于请求验证客户端身份
- 包含可接受的公钥和签名算法、证书颁发机构列表，这些机构都由可分辨的名称标识

```
struct {

}ServerHelloDone;
```

- 必须被发送，表示ServerHello及相关消息的结束。此后服务器等待客户端的消息
- 客户端收到该消息后，可以校验服务器证书，检查ServerHello参数

- # 共用的握手消息

```
opaque  ASN.1Cert;
struct {
  ASN.1Cert  certificate_list[];
}Certificate;
```

- 用于携带X.509证书链，证书链中的证书以ASN.1 DER编码
- 主证书在最前面，颁发者的证书跟随其后，根证书可以省略，内置在客户端
- 服务器必须保证选定的密码套件与发送的证书一致

```
struct {
  opaque  verify_data[];
}Finished;
```

- 表示握手完成，消息内容被加密，以安全地验证握手的完整性
- verify_data的值是握手过程中所有消息（但不包括该消息）的散列值
- 散列值：SSL 3用MD5 + SHA1计算，长36字节；TLS用PRF计算，默认12字节，TLS 1.2支持密码套件使用更长的值

- # 连接关闭

close_notify警报消息用于有序关闭连接，即当本端不再发送更多数据或数据发完时，先发送该消息。若已经发送过fatal alert消息，则不应发送该消息

1. 客户端发出关闭请求，此时必须等待服务器的确认

2. 服务收到请求后，丢弃所有未写出的数据，并发出自己的close_notify消息，再关闭连接（执行TCP关闭）；客户端收到确认后，再关闭连接（执行TCP关闭）。警报到来后的任何消息都将被忽略

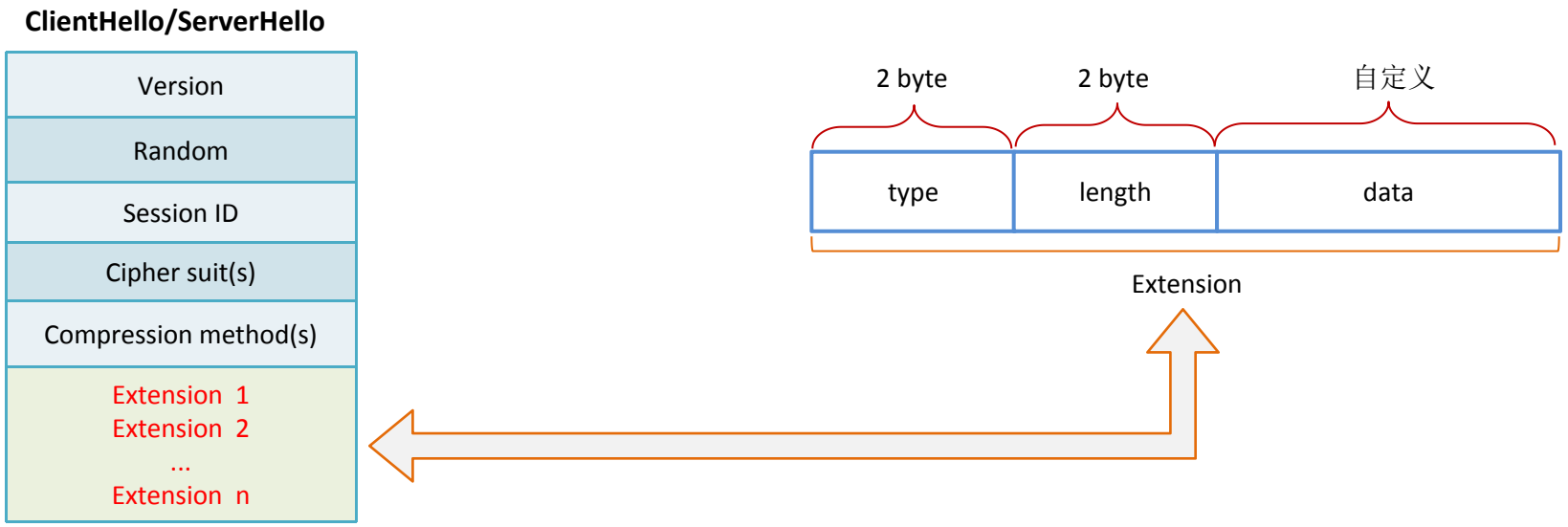
这个关闭协议虽然简单，但可以避免连接截断攻击，如果没有关闭协议，通信双方就无法确认是遭到攻击还是通信真正结束。在TLS 1.1前，任何连接上的错误都会导致TLS会话信息被丢弃，这意味着客户端在后续的连接上要重新执行完整握手，但TLS 1.1对于非正常关闭的连接删除了这个要求

TLS扩展

- 定义和格式
- 主要扩展

- # 定义和格式

SSL不支持扩展，TLS扩展是一种通用目的的扩展机制，使用它可在不修改协议本身的情况下，增加功能。作为单独的规范首次出现在RFC 3546中。扩展以扩展块的形式增加在ClientHello和ServerHello消息的末尾，扩展块由一个个扩展组成，如下所示



扩展的格式和期望的行为由每个扩展自己决定，在实践中，扩展通常用于通知支持某些新功能，以及用于在握手阶段传递所需的额外数据

主要扩展

服务器名称指示

- 通过server_name扩展实现，客户端利用它可以连接到某个虚拟服务器（比如某个站点），又称安全虚拟主机机制（思考为什么）
- 服务器收到后，匹配相应的证书发给客户端验证；若没匹配到，则应响应严重级别的unrecognized_name警报或继续握手
- 如果没有这种机制，那么每个IP只能部署一张证书

OSCP stapling

- 客户端使用status_request扩展指示支持OSCP stapling，而支持的服务器在ServerHello中返回一个空的status_request扩展
- 服务器使用这个特性发送最新的证书吊销信息给客户端，并且能包含多个证书的吊销状态（RFC 6961支持）

椭圆曲线功能

RFC 4492提出了两个扩展，可以在握手时通告客户端的EC功能。分别是elliptic_curves和ec_point_formats扩展

- **elliptic_curves：** 客户端在ClientHello中列出支持的曲线名称，服务器从中选择一条双方都支持的曲线名称。RFC 4492定义了很多主要的曲线，但目前只有secp256r1和secp384r1这两种得到了广泛支持
- **ec_point_formats：** 在协商时对椭圆曲线顶点进行可选压缩。理论上，经压缩的顶点格式能在受限的环境下节省宝贵的带宽资源，但在实际使用中节省的量不大（比如为256位的曲线节约64字节），因此压缩格式一般未使用

其它扩展

应用层协议协商（RFC 7301）、心跳（RFC 6520）、安全重新协商、会话票证（RFC 5077）、签名算法

密码套件

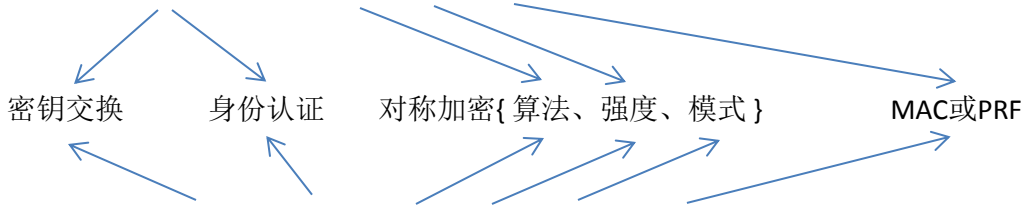
- 描述性名称
- 密钥交换与认证
- RSA密钥交换
- DH密钥交换
- 对称加密及模式
- MAC计算

- # 描述性名称

一般地，SSL套件使用SSL_前缀，TLS套件使用TLS_前缀

完整套件

SSL_RSA_WITH_RC4_128_MD5



TLS_ECDHE_ECDSA_AES_256_GCM_SHA384

部分套件

SSL_RSA_WITH_NULL_MD5

TLS_RSA_WITH_NULL_SHA256

NULL套件

SSL_NULL_WITH_NULL_NULL

TLS_NULL_WITH_NULL_NULL

- 在MAC或PRF前，若密钥交换、身份认证和对称加密之一没使用，则用NULL表示；NULL套件仅用于ClientHello和ServerHello握手消息
- 有的算法既能用于密钥交换，也能用于身份验证，如RSA；有的仅能用于身份验证，如DSA
- 仅分组加密具有对应的模式，而模式又决定是否使用及使用的MAC或PRF算法，GCM套件使用名称的最后一段指明为PRF而非MAC算法
- SSL使用MAC；TLS使用PRF，TLS 1.0和TLS 1.1基于HMAC-MD5/HMAC-SHA1，TLS 1.2默认基于HMAC-SHA256，允许套件定义PRF，SHA384 GCM套件则使用HMAC- SHA384

• 密钥交换与认证

密钥交换是握手过程中最引人入胜的部分。在SSL/TLS中，会话安全性取决于称为主密钥（master secret）的48字节共享密钥。密钥交换的目的是计算另一个值，即预主密钥（premaster secret），这个值是组成主密钥的来源

为什么密钥交换要结合身份认证？这可能是考虑到密码操作开销很大，若服务器被仿冒，则后面的密码操作就没意义了，为了避免无效费时的密码操作，因此密钥交换与身份认证结合在一起（注意Finished消息中的完整性检验不能解决身份仿冒的问题）

密钥交换有多种算法，最根本主要的是RSA、DHE_RSA、ECDHE_RSA和ECDHE_ECDSA，DHE为临时DH，与DH原理相同；ECDHE是基于椭圆曲线实现的DHE。DH和DHE、ECDHE与ECDH的算法相同，不同的是使用时机，前者临时生成参数，而后者是长久静态的，通常固定在证书或程序中。SSL不支持ECDHE算法

密钥交换使用ServerKeyExchange和ClientKeyExchange消息来交换参数，前者由服务器发送，后者由客户端发送。

```
struct {
    select (KeyExchangeAlgorithm) {
      case  rsa:
          ServerRSAParams params;
          Signature  signed_params;
      case  dhe_rsa:
          ServerDHParams params;
          Signature  signed_params;
      case  ecdhe_rsa:
      case  ecdhe_ecdsa:
          ServerECDHParams  params;
          Signature  signed_params;
    };
}ServerKeyExChange;
```

```
struct {
    select (KeyExchangeAlgorithm) {
      case  rsa:
          EncryptedPreMasterSecret;
       case  dhe_rsa:
          ClientDiffieHellmanPublic;
      case  ecdhe_rsa:
      case  ecdhe_ecdsa:
          ClientDiffieHellmanPublic;
     } exchange_keys;

}ClientKeyExChange;
```

ServerKeyExchange仅当服务器Ceritificate消息不包含让客户端交换预主密钥所需足够信息（例如仅签名证书）的时候才发送，对于RSA、DH_DSS和DH_DSA类型的密钥交换算法，发送该消息是不合法的。该消息不仅有算法参数，还有对应参数的签名用于身份验证。使用签名，客户端就能确认它正在与持有私钥对应证书中的公钥的实体进行通信

ClientKeyExchange在收到ServerHelloDone后必须被发送，若服务器需验证客户端，则先发送Certificate，再发送它；若服务器证书包含静态的DH参数，则消息为空

RSA密钥交换

RSA密钥交换十分直接简单，涉及的数据结构和交换过程描述如下

```
struct {
    opaque rsa_modulus;
    opaque rsa_exponent;
}ServerRSAParams;
```

```
struct {
    ProtocolVersion  client_version;
    opaque  random[46];
}PreMasterSecret;
```

```
struct {
    public-key-encrypted PreMasterSecret pre_master_secret;
}EncryptedPreMasterSecret;
```

{ rsa_modulus, rsa_exponent } 为公开密钥，其中rsa_modulus为模数，rsa_exponent为加密指数

1. 服务器生成一个临时公钥发即ServerRSAParams发给客户端
2. 客户端收到后，先生成一个46字节的随机数、加上它支持的最高版本号，再用临时公钥加密（运算如下），最后发给服务器

   EncryptedPreMasterSecret = (client_version+random) ^ rsa_exponent mod rsa_modulus　（+ 表示串联）
3. 当服务器收到后，解密EncryptedPreMasterSecret消息即可取出预主密钥

为什么预主密钥包含了client_version？这个client_version与ClientHello消息中的版本号不一定相同，若两者不同，说明协议版本发生了回退，这可能是由于遭遇了一个协议降级攻击。

在正常的RSA握手过程中，ServerKeyExchange是不允许发送的，但若使用了出口密码套件或有问题的SSL库，则会继续处理这个消息，进而为密钥交换提供了弱RSA密钥（512位），使暴力破解成为可能。

- # DH密钥交换

DH算法是Diffie-Hellman算法的简称，它是第一个公开密钥算法，也是一种密钥协定的协议，使两个实体在不安全的信道上生成共享密钥成为可能，它的安全性源于在有限域上计算离散对数的困难性。涉及的数据结构和交换过程描述如下

```
struct {
      opaque  dh_p;
      opaque  dh_g;
      opaque  dh_Ys;
}ServerDHParams;
```

```
struct {
      select (PublicValueEncoding) {
            case  implicit:  struct { }
            case  explicit: opaque  dh_Yc;
      }dh_public
}ClientDiffieHellmanPublic;
```

1. 服务器选取两个大素数dh_p、dh_g，一个大随机整数x，其中dh_g为dh_p的生成元，计算dh_Ys = dh_g ^ x mod dh_p，将它们即ServerDHParams发给客户端
2. 客户收到后，选取一个大随机整数y，计算dh_Yc = dh_g ^ y mod dh_p，将它即ClientDiffieHellmanPublic发给服务器
3. 客户端计算pre_master_secret = dh_Ys ^ y mod dh_p = dh_g ^ (xy) mod dh_p，服务器计算pre_master_secret = dh_Yc ^ x = dh_g ^ (xy) mod dh_p，由此可见两端计算的预主密钥相同

临时DH密钥交换中没有任何参数被重复使用，与之相对，在一些DH密钥交换方式中，某些参数是静态的，并被嵌入到服务器和客户端的证书中（此时ClientDiffieHellmanPublic为空）。这样一来，密钥交换的结果是一直不变的共享密钥，就无法提供前向保密了

- # 对称加密及模式

对称加密有两种基本类型：序列（或称流）加密（stream）和分组（或称块）加密（block），加密模式通常由基本密码、一些反馈和简单运算构成

流加密：　　在明文和密文数据序列上的1位或1字节，有时甚至是32位的字，它的安全性完全依赖于随机密钥流生成器的内部机制

分组加密：　　在明文分组与密文分组上进行运算，一般分组为64位，但有时更长

分组加密模式：主要有ECB、CBC、CFB和OFB 四种，SSL/TLS实现采用的是CBC，这是因为ECB不能隐藏明文数据的模式，在密钥一定的情况下，加密相同的明文总是得到相同的密文，这在某些环境下可能造成严重安全问题；而CFB和OFB一次只能加密j位的明文数据（通常j < 分组大小），所以综合安全和性能，选择了CBC。在CBC即密码分组链接模式中，每个明文块先与前一个密文块异或（第一个明文块与IV异或），再加密；每个密文块解密后，再与前一个密文块异或得到明文块（第一个密文块与IV异或）

下表列举了常用的加密模式及相关属性

Cipher 加密算法	Type 加密类型	Key Material 密钥原始大小	IV Size 初始向量大小	Block Size 加密块大小
NULL	stream	0	0	N/A
RC4_128	stream	16	0	N/A
DES_CBC	block	8	8	8
3DES_EDE_CBC	block	24	8	8
AES_128_CBC	block	16	16	16
AES_256_CBC	block	32	16	16

NULL表示未使用加密，用于明文传输阶段，一般是握手过程中Finished消息前的所有消息。Block Size仅用于分组加密算法，3DES由于增加了密钥长度，因此比DES安全；AES比3DES更安全且高效，有128位和256位两种类型。由于CBC使用了IV和前个密文块作为反馈，这个特点使它易受BEAST攻击——一种基于可预测IV的明文选择攻击，而RC4是流加密算法，对该攻击免疫

- # MAC计算

MAC用于验证消息的完整性，即检测消息是否被修改、添加、删除。它的计算发生在记录协议操作中，压缩后（若有）加密前。以SSL 3.0和TLS 1.2为代表，计算公式如下，+表示串联

MAC =

hash(MAC_write_secret + pad_2 + hash(MAC_write_secret + pad_1 + seq_num + SSLCompressed.type + SSLCompressed.length + SSLCompressed.fragment))　　协议为SSL 3.0

HMAC_hash(MAC_write_secret, seq_num + TLSCompressed.type + TLSCompressed.version + TLSCompressed.length + TLSCompressed.fragment))　　协议为TLS 1.2

hash代指哈希算法，HMAC_hash代指带密钥的哈希算法，seq_num为消息的序列号，MAC_write_secret为MAC写密钥，具体算法及参数如下表所示

	MAC algorithm MAC算法	**mac_length MAC输出长度**	**MAC_write_secret_length MAC写密钥长度**	**padding size 填充大小**	**pad_1 填充特殊值**	**pad_2 填充特殊值**
共用	NULL	0	0	N/A	N/A	N/A
SSL	MD5	16	16	48	48个0x36	48个0x5c
	SHA1	20	20	40	40个0x36	40个0x5c
TLS	HMAC-MD5	16	16	N/A	N/A	N/A
	HMAC-SHA1	20	20	N/A	N/A	N/A
	HMAC-SHA256	32	32	N/A	N/A	N/A
	HMAC-SHA384	48	48	N/A	N/A	N/A

NULL表示未使用MAC，这发生在SSL_NULL_WITH_NULL_NULL或TLS_NULL_WITH_NULL_NULL套件中

密码操作

- 伪随机函数
- 主密钥计算
- 密钥生成

- # 伪随机函数

在TLS中，伪随机函数（pseudorandom function，PRF）用于生成任意数量的伪随机数据，PRF使用一条秘密、一颗种子和一个唯一标签。从TLS 1.2起，所有密码套件都需要明确指定它们的PRF，TLS 1.2指定的PRF则是基于HMAC-SHA256

TLS 1.2定义的PRF基于数据扩展函数P_hash，该函数使用了HMAC和一个任意散列函数

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

A(i)函数定义如下

A(1) = **HMAC_hash**(secret, seed)

A(2) = **HMAC_hash**(secret, A(1))

...

A(3) = **HMAC_hash**(secret, A(i - 1))

PRF则是结合标签和种子对P_hash的封装

PRF = **P_hash**(secret, label + seed)

- # 主密钥计算

前面已经讲过，密钥交换的目的是计算预主密钥，因为不同的密钥交换方法，得到的预主密钥长度可能不同，所以需要执行主密钥的计算，主密钥总是固定长48字节（384位）。以SSL 3.0和TLS 1.2为代表，计算公式如下

master_secret =

- **MD5**(pre_master_secret + SHA('A' + pre_master_secret + ClientHello.random + ServerHello.randhom)) +
  MD5(pre_master_secret + SHA('BB' + pre_master_secret + ClientHello.random + ServerHello.randhom)) +
  MD5(pre_master_secret + SHA('CCC' + pre_master_secret + ClientHello.random + ServerHello.randhom))　　协议为SSL 3.0
- **PRF**(pre_master_secret, “master secret”, ClientHello.random + ServerHello.random)　　协议为TLS 1.2

查前面的加密表和哈希表可知，MD5输出为16字节，HMAC-SHA256输出为32字节，所以MD5迭代了3次，而PRF迭代了2次（忽略最后的16字节）。因为计算引入了客户端和服务器的随机数，所以主密钥也是随机的，且与协商握手绑定。一旦主密钥计算完成，预主密钥应从内存删除

- 密钥生成

计算公式如下

key_block =

MD5(master_secret + SHA('A' + master_secret + ClientHello.random + ServerHello.randhom)) +
MD5(master_secret + SHA('BB' + master_secret + ClientHello.random + ServerHello.randhom)) +
MD5(master_secret + SHA('CCC' + master_secret + ClientHello.random + ServerHello.randhom)) + [...]

协议为SSL 3.0

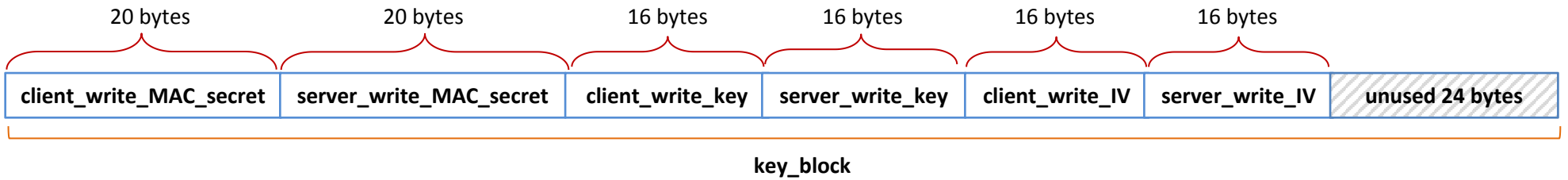
PRF(master_secret, “key expansion”, ClientHello.random + ServerHello.random)

协议为TLS 1.2

密钥生成将主密钥扩展成足够的序列即密钥块，从左到右按序分割为下面6个密钥（流加密不使用IV，AEAD套件不使用MAC密钥）

客户端MAC密钥、服务器MAC密钥、客户端加密密钥、服务器加密密钥、客户端加密IV、服务器加密IV

密钥块的长度是6个密钥长度的总和。具体的长度则由协商选定的密码套件决定，以套件TLS_RSA_WITH_AES_128_CBC_SHA为例，查前面的加密表和哈希表可知，key_block输出应为2*16 + 2*20 + 2*16 = 104，又因HMAC-SHA256输出为32字节，所以PRF需迭代(104 + 32 - 1)/32 = 4次 （输出为32的倍数），生成的各密钥分布如下图



协义版本间的差异

下表总结了各协议(SSL 3.0 ~ TLS 1.2)间的主要差异，-表示删除，+表示增加

	Finished校验	密钥交换与认证	对称加密及模式	MAC计算	密码操作
SSL 3.0	MD5 + SHA1 36字节	RSA DH_RSA, DHE_RSA DH_DSS, DHE_DSS FORTEZZA_KEA	RC4_40, RC4_128 IDEA_CBC, DES_CBC 3DES_EDE_CBC	MD5 SHA1	MD5
TLS 1.0	PRF 12字节	同SSL 3.0，但 - FORTEZZA_KEA + ECDHE_RSA, ECDHE_ECDSA	同SSL 3.0，但 - RC4_40 + AES_128_CBC, AES_256_CBC	HMAC-MD5 HMAC-SHA1	PRF
TLS 1.1	同TLS 1.0	同TLS 1.0	同TLS 1.0	同TLS 1.0	PRF
TLS 1.2	PRF 长度自定义	同TLS 1.1	同TLS 1.1，但 - IDEA, DES_CBC +AES_128_GCM,AES_256_GCM + AEAD	同TLS 1.1，但 + HMAC-SHA256 + HMAC-SHA384	PRF

- 密码操作即主密钥计算和密钥生成，SSL使用MD5迭代，而TLS用PRF迭代
- TLS比SSL最大的变化是支持了椭圆曲线密钥交换及认证、AES加密、HMAC及PRF，TLS 1.2又增加了AEAD加密及GCM套件
- 安全性TLS比SSL强

安全性分析

- 数据保护机制
- 协议降级攻击
- FREAK攻击
- Dos攻击
- 其它攻击

- # 数据保护机制

机密性

每个连接使用由主密钥生成的唯一MAC secrets、加密keys和IVs。当通过恢复会话新建一个连接时，虽然master secret相同，但因两端的Random不同，故生成的MAC secrets、加密keys和IVs不同。攻击者不可能在不打破安全hash操作的情况下，通过已知的MAC secrets或加密keys来获得master secret，所以如果这个会话的master secret是安全的，并且hash操作也是安全的，那么这个连接是安全的且独立于以前的连接。但仍建议一个Session ID的生存周期最长为24小时，因为获得了master secret的攻击者可能在Session ID改变之前假冒受攻击的一方

完整性

攻击者可能尝试修改握手消息，使双方选择不同寻常的密码套件，因为许多实现支持40位的出口加密套件，甚至不用加密或MAC算法。若一个或多个握手消息被修改，则客户端和服务器计算出不同的handshake message hashes，导致双方不接受彼此的Finished消息。没有主密钥，攻击者就无法修复Finished消息，因此攻击将被检测

应用协议层的数据由MAC进行保护，MAC计算不仅包括数据，还包括序列号和MAC密钥，序列号使消息的重放、重排和删除攻击被检测，而MAC密钥阻止一个实体的消息被注入到另一个实体，因为每个实体有唯一的MAC密钥。如果数据被修改，则验证MAC失败，发出fatal级别的bad_record_mac警报，导致连接立即关闭，从而攻击失败

有效性

身份认证发生在握手阶段的密钥交换或证书验证中，服务器和客户端皆可验证对方

• 协议降级攻击

协议漏洞

在SSL 2.0中，没有握手完整性检验机制，且在最坏情况下，RSA密钥（RSA是SSL 2.0中唯一的密钥交换与认证机制）只有40位

攻击原理

攻击者作为中间人企图修改握手过程中的连接参数，迫使使用一个低等级的协议或低强度的密码套件

缓解方法

SSL 3.0和TLS提供了2种检测方法

1. 当客户端被降级为SSL 2.0时，则使用0x03（正常方式是随机值）填充RSA加密的PKCS#1块结尾的8个字节，这样服务器就能检验它们是否全为0x03，从而发现攻击结束握手；仅支持SSL 2.0的服务器，则忽略并继续握手

正常填充方式	密钥	?	?	?	?	?	?	?	?
降级填充方式	密钥	3	3	3	3	3	3	3	3

若中间人伪装成SSL 2.0服务器并仅提供40位的套件，再暴力破解RSA私钥并解密，最后构造新的包含相同密钥但使用降级填充方式的ENCRYPTED-KEY-DATA消息，则该检测方法无效

2. RSA密钥交换中的PreMasterSecret包含一个客户端支持的最高版本号，这样服务器就能与之前收到的版本号比较，从而发现攻击结束握手。该方法即使在散列函数安全性被破坏的情况下同样有效

- # FREAK攻击

漏洞利用

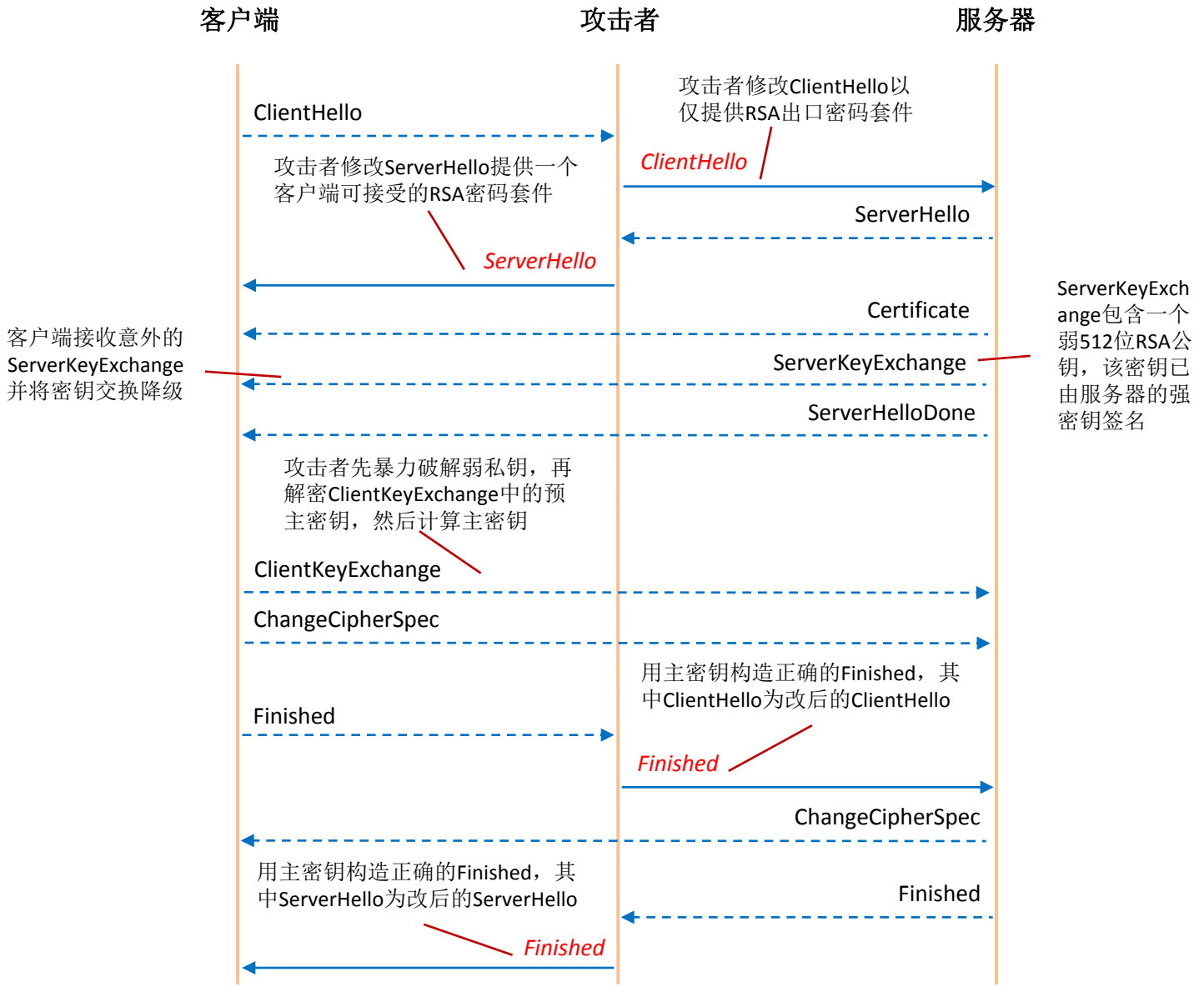
针对支持出口RSA密码套件的服务器，而且客户端提供的密码套件列表中至少有一个支持RSA密钥交换，因为这样才能协商选定RSA密钥交换，导致ClientKeyExchange消息使用弱512位RSA公钥加密预主密钥

攻击原理

攻击者作为中间人拦截客户端和服务器间的SSL/TLS连接，先修改ClientHello、ServerHello和Finished握手消息，再发回，如右图

缓解方法

服务器禁止支持出口套件，如果一定要支持，那么在通信时实时生成512位的公钥，或者提前生成一批密钥，在使用一段时间后更换。总之，要保证密钥使用的周期不超过512位密钥分解需要的时间



- # Dos攻击

漏洞利用

SSL/TLS握手中的RSA密钥交换，解密比加密更耗CPU（平均2048位的密钥，解密开销为加密的4倍）

客户端发起的重新协商，每次重新协商执行一次新的握手以协商不同的安全参数

攻击原理

使用僵尸网络发起大量的虚假连接，在单个连接上又多次重新协商，以自己最少的成本（优化过的），消耗压垮服务器的CPU

缓解方法

1. 限制连接的数量和速率，对于Linux服务器，可采用开源的netfilter加固方案（使用特征码识别SSL/TLS握手）
2. 禁止用RSA作为密钥交换算法，仅用于身份认证
3. 关闭重新协商

- # 其它攻击

Logjam

攻击原理与FREAK类似，但针对的是弱安全的出口DH密钥交换（512位密钥），使用NFS（数域筛子）法暴力计算离散对数，从而获得预主密钥

BEAST

针对CBC模式中IV为前个密文块的漏洞，截获可能包含敏感信息M的密文块C[i]和它的前一个密文块C[i-1]，构造明文块P = C[i-1]⊕C[i]⊕M'，M'为猜测敏感信息的明文， 注入到敏感块后面，观察加密发送后的结果并与C[i]比较，如果相同，那么猜测成功，否则继续猜测尝试

POOLDE

针对SSL 3.0中CBC模式的填充预示漏洞而进行的攻击，用于破解cookie或密码等小段数据，先破解1字节，至多尝试256次，再循环破解其它字节

参考资料

- 协议标准

RFC 6101：SSL 3.0
RFC 2246：TLS 1.0
RFC 4346：TLS 1.1
RFC 5246：TLS 1.2
RFC 6066：目前最新的TLS扩展规范
RFC 2104：基于密钥的MAC

- 书籍

《应用密码学 -- 协议、算法与C源程序》
《HTTPS权威指南》