摘 要

本协议基于 AES-GCM 和 x25519 实现了类似 TSL1.3 协议的功能，在仅两方知道密码又不要求证书的情况下，仅需 1-RTT 做到内容安全、端点可靠认证，无法通过主动探测、重放攻击、机器学习等方法区分本协议与真实 TLS1.3。并且为了提高机器和网络性能，在不影响安全的前提下，可以选择性地加密和使用网络多路复用。

关键词：TLS1.3；信道安全；伪造 TLS

Abstract

This protocol implements functions similar to TSL1.3 protocol based on AES-GCM and x25519. When only two parties know the password and do not require a certificate, only 1-RTT is required to achieve content security and reliable endpoint authentication. It is impossible to distinguish this protocol from real TLS1.3 through active detection, Replay attack, machine learning and other methods. And in order to improve machine and network performance, selective encryption and network multiplexing can be used without affecting security.

Keywords: TLS1.3; Channel Security

目 录

符号和缩略语说明

Server	服务端
Client	客户端
TLS	传输层安全性协议 (Transport Layer Security)
ECDHE	椭圆曲线迪菲-赫尔曼金钥交换 (Elliptic-curve Diffie–Hellman)

第 1 章　协议现状

最新版本为 V1，最后更新于 2023 年 6 月 27 日，处于征集讨论阶段，无协议其他变种。唯一官网：https://github.com/JimmyHuang454/JLS。文档语言：中文简体。

目前支持 JLS 协议的软件工具有：

- https://github.com/JimmyHuang454/RRS

1.1　Copyright

使用最宽松、最自由的版权协议 WTFPL。

1.2　贡献与反馈

请移步到 github 官网，根据需要提出问题或贡献。所有贡献代码都应符合 WTFPL。

1.3　Change Log

1.3.1　V1

2023 年 6 月 24 日，初始版本

第 2 章　Specification V1

本章描述了 JLS 协议第 1 版的具体内容（不含正确性和安全性验证），另付一个基于 Dart 编程语言的具体实现。

2.1　约定

- 数组相加有先后顺序，如无特殊标注，一律以大端字节序存储
- 如无特殊标注，本文所指 TLS 均为 TLS 版本 1.3[1]

2.2　背景

对代理协议内容加密和认证是一项广泛的需求，基于应用层的代理协议如 HTTP、SOCK5 等都不支持内容加密①，因此出现了如 ShadowSocket 等加密代理协议，随着 TLS 的广泛应用，为了解决维护难、缺少安全性验证、主动探测等问题，又出现了以 TLS 为基础的协议，如 Trojan 协议。

Trojan 可以说是一个完美的协议，但必须要求购买域名，申请域名证书。在某些极端环境，仅允许特定域名通行，Trojan 失效。但预估 IP 白名单模式②很快来临，届时我们只能祈祷。

2.3　基本原理

首先考虑一个简化模型：现有一个客户端、一个服务端和一个仅双方知道的一个密码 PWD；如何保证：

- 1. 客户端和服务端之间的数据原文没被篡改
- 2. 不能被除了客户端和服务端以外的第三方解密
- 3. 客户端（服务端）能够识别接收到的数据是否真的来自服务端（客户端）
- 4. 收到的通讯数据是否已经接收过（即防止重放攻击）

为了实现第 1、2 、3 点，可以采用 AES-GCM 对称加密算法；AES 保证了第 2 点，GCM 保证了第 1 和第 3 点；

① 虽然有认证，但形同虚设
② 仅限指定 IP 通行的模式

为了实现第 4 点，我们可以在 AES-GCM 的基础上往数据原文中加上一个时间戳，但是这种方式有延迟，比如说 vmess 协议，只要攻击速度够快，90 秒内可以无限让客户端或者服务端重复接收已经接收过的数据。

还有一种方法是让客户端和服务端之间进行一次握手，客户端先生成一个随机数 N1，通过 AES-GCM 加密发送 N1，服务端接收后可以解密出来，再生成一个随机数 N2，同样地加密发送给客户端；

完成握手后，就约定以后发送的数据的秘钥都是 N=N1+N2+PWD，而且每发一个数据包都要在原文中附上序号（第一个包、第二包……第 x 个包）。这种会增加一次 RTT，但我们可以稍微修改一下 TLS 协议，把其中的随机数（random）换成我们给出的随机数 N1 和 N2，达到交换随机数的目的。

这种方法没有延迟，两个随机数共同组成本次对话的 ID，所以该随机数应该要足够随机才能保证能防止重放攻击（Replay Attacks）。

根据 GCM 的认证功能，如果不知道 PWD，就无法解出密文，所以我们就可以检测出客户端/服务端是否假冒，从而把数据转发到我们想要的任何地址。

但为了避免服务器被探测到，只能支持全程加密的 TLS1.3；同样的，如果服务端检测到不是 TLS1.3，直接转发数据，不作处理。

2.4　准备

Client 和 Server 都必须准备一个仅双方知道的密码 PWD 和一个随机数 IV。PWD 和 IV 长度无限制，但都不应该小于 32 字节并应该足够随机。

Server 端必须准备一个伪装站（Fallback Website）地址，伪装站应该优先选取延迟低的、使用 TLS1.3 的、使用 HTTP2 的、不能是 HTTP3、DDOS 防御没那么敏感的。

2.5　Handshake（握手）

JLS 的握手包跟 TLS 所规定的数据结构和时序完全一致。

2.5.1　Client Hello 包

功能：向服务器发送建立安全信道的请求

- 如果 Client 支持 TLS1.3，那么在 Client Hello 的 supported_version 拓展必须包含 TLS1.3 字段。JLS 是建立在 TLS1.3 的基础上，所以 JLS 的 Client Hello 包必须要有此字段
- Client Hello 中的 random 字段，必须是通过算法2.1所生成的 FakeRandom，通

过代码2.1还会生成一个随机数 N1，N1 必须要足够随机，用于防止已知明文攻击（Known Plaintext Attack）

- Client Hello 中的 session ID 字段，应该是随机的
- Client Hello 包必须包含 key_share 字段和交换秘钥所用算法。Client 必须自行生成正确的 key_share 和所用算法（即 supported_groups）发送给 Server
- 开发者可以在不修改或缺少 JLS 必要信息的情况下伪造 Client Hello 头特征，比如说 chrome 浏览器的 cipher suits 列表等，用于伪造 Client Hello 指纹。需要特别注意的是：如果开发者错误地伪造 Client Hello，也就是 JLS 能够正常使用，但 TLS 不能使用的情况下，会导致明显的特征，比如说 Client Hello 缺少了 cipher suits 等必要信息，TLS Server 应该发送 HelloRetryRequest，但 JLS 认为这是合法 Client，所以是不会发送 HelloRetryRequest
- Client Hello 不应该使用 early data 或 pre_shared_key，虽然可以做到 0-RTT，但是会导致重放攻击等安全问题
- 使用 padding extension 是为了希望避免发送长度为 256-511 字节的 Client hello[2]，因为有不少 TLS server 拒绝接收长度小于 512 字节的 Client hello，因此开发者在伪装 Client hello 时，应该尽可能使用长度（不含 server_name 扩展）大于 512 字节的 Client hello

算法 2.1　生成 Client Hello

```
bytes,bytes buildFakeRandom(iv, pwd) {
  iv = sha512.convert(iv);
  pwd = sha256.convert(pwd);

  // n 长度为16字节
  n = buildRandomBytes(16);

  // mac, cipherText均为默认的16字节
  mac, cipherText = AES_GCM_256.encrypt(n, pwd, iv);

  fakeRandom = mac + cipherText;
  return fakeRandom, n;
}


bytes buildClientHello() {
  // clientHello是将要发送给server的clientHello
  // 由于未生成fakeRandom,
  // 所以此时clientHello中的random 32 字节应该全部填充为0
  clientHello['random'] = 0
  clientHello['pre_shared_key'] = buildDH();
  iv = utf8.encode(userIV) + clientHello;

  clientFakeRandom, N1 = buildFakeRandom(iv, utf8.encode(userPwd));
```

```
25
26    clientHello['random'] = clientFakeRandom;
27    return clientHello;
28  }
```

2.5.2　Server 验证

功能：验证收到的握手是否来自有效的 Client

- Server 必须通过算法2.2判断是否为有效 Client
- 如果不是有效 Client，直接把接收到的所有数据转发到伪装站，不作其他处理
- Server FakeRandom 的后 8 字节不能是

$$44, 4F, 57, 4E, 47, 52, 44, 01$$

和

$$44, 4F, 57, 4E, 47, 52, 44, 00$$

，如果是，应该不断重新生成一个 Server FakeRandom ，直到合规后再发送 Server Hello

算法 2.2　检查是否有效 Client

```
1  bool, bytes parseFakeRandom(clientFakeRandom, iv, pwd){
2    iv = sha512.convert(iv);
3    pwd = sha256.convert(pwd);
4
5    mac = clientFakeRandom[0:16];
6    cipherText = clientFakeRandom[16:32];
7    isValid, n = AES_GCM_256.decrypt(cipherText, mac, pwd, iv);
8    return isValid, n;
9  }
10
11 bool serverCheck(clientHello){
12   clientFakeRandom = clientHello['random'];
13   // 32字节全置为0
14   clientHello['random'] = 0;
15   // 防止Client Hello被修改
16   iv = utf8.encode(userIV) + clientHello;
17   pwd = utf8.encode(userPwd);
18
19   isValid, N1 = parseFakeRandom(clientFakeRandom, iv, pwd);
20   return isValid;
21 }
```

2.5.3　发送 Server Hello 和证书

功能：发送 Server Hello，并发送伪装站证书

- 首先要确认是否有效 Client，然后通过算法2.3生成 Server Hello.
- 如果是有效 Client，则可以得出来自 Client 的随机数 N1，然后 Server 要生成自己的随机数 N2 和 Server FakeRandom
- Server 必须要根据 Client 的 key_share 和 supported_groups 得出共同秘钥 S1，并把 S1 也作为最终秘钥之一来加密数据，以保证前向安全性
- 因为 Server 已经验证了 Client 的有效性，所以 Server 证书可以传输随意内容，但包长度应该要伪装站返回的证书一致，建议开发者在软件初始化时获取伪装站的真实证书包；为了方便，返回差不多长度的 Server 证书包也是被允许的，因为这不会影响安全性，但可能成为特征；Client 无需验证该证书是否有效
- Session ID 应与 Client Hello 中的一致
- support_group 应只使用 x25519

算法 2.3　生成 Server Hello

```
bytes handleDH(clientHello) {
  algorithm = clientHello['supported_groups'];
  serverKeyPair = newServerKeyPair;
  sharedSecretKey = algorithm.sharedSecretKey(
        keyPair: serverKeyPair,
        remotePublicKey: clientHello['pre_shared_key'],
      );
  return serverKeyPair.publicKey, sharedSecretKey;
}

bytes buildServerHello(clientHello) {
  if (!serverCheck(clientHello)) {
    forwardFallbackWebsite();
    return;
  }
  serverPublicKey, S1 = handleDH(clientHello);
  serverHello['pre_shared_key'] = serverPublicKey;

  // 此时 serverHello 中的 random
  // 也同样全置为0
  serverHello['random'] = 0;
  // 此时 clientHello 中的random 不置0
  iv = utf8.encode(userIV) + clientHello + serverHello;
  pwd = utf8.encode(userPwd) + S1;

  serverFakeRandom, N2 = buildFakeRandom(iv, pwd);
  serverHello['random'] = serverFakeRandom
  return serverFakeRandom;
```

```
}
```

2.5.4　Client 验证

功能：验证是否有效 Server

- 根据算法2.4得出验证是否有效 Server，并且得出共同秘钥 S1。如果不是有效 Server，则 Client 完全按照 TLS1.3 流程处理，即验证证书，并协商出 TLS 最终随机数（秘钥），最后发送合规的 http 请求即可
- 如果是有效 Server，则无需验证来自 Server 的证书。

算法 2.4　验证 server

```
bool clientCheck(serverHello){
  serverFakeRandom = serverHello['random'];
  serverHello['random'] = 0;
  // 此时 clientHello 中的random 不置0
  serverPublicKey, S1 = handleDH(serverHello);
  iv = utf8.encode(userIV) + clientHello + serverHello;
  pwd = utf8.encode(userPwd) + S1;
  isValid, N2 = parseFakeRandom(serverFakeRandom, iv, pwd);
  return isValid;
}
```

2.6　Application Data

使用经过 x25519 算法得出共享密钥 S1，最后得出的秘钥 finalPWD = PWD + S1 + N1 + N2，通过 AES_GCM_256 加密发送数据。包结构与 TLS 一致。Server 和 Client 都必须各自维护一个自增 ID，用于记录已接收和已发送包数量，按照算法2.5得出实际发送数据。目前 GCM 的 MAC 的长度为默认的 16 字节，每一个 Application Data 都必须在密文前加上 MAC。如果 Client 或 Server 收到验证失败的 Application data，必要按照正常 TLS 流程处理。

算法 2.5　加密数据

```
iv = utf8.encode(userIV);
pwd = sha256.convert(utf8.encode(userPWD) + S1 + N1 + N2);

// 从0开始，8字节大端存储
sendCount = 0;

bytes,bytes encrypt(data){
  packetIV = sha512.convert(iv + sendCount);
  mac, cipherText = AES_GCM_256.encrypt(data, pwd, packetIV);
  sendCount += 1;
  return mac + cipherText;
}
```

算法 2.6　解密数据

```
iv = utf8.encode(userIV);
// N1来自Client，N2是来自Server，S1是x25519的sharedSecretKey
pwd = sha256.convert(utf8.encode(userPWD) + S1 + N1 + N2);

// 从0开始，8字节大端存储
// 应与加密中的sendCount相同
receiveCount = 0;

bytes encrypt(data) {
  packetIV = sha512.convert(iv + receiveCount);
  mac = data[0:16]; // 前16字节
  cipherText = date[16:];
  mac, cipherText = AES_GCM_256.decrypt(cipherText, pwd, packetIV,
      mac);
  receiveCount += 1;
  return plainText;
}
```

2.7　Dart 编程语言实现

RRS 是一个 Trojan 和 JLS 的具体实现。详见地址：

https://github.com/JimmyHuang454/RRS/tree/master/lib/transport/jls

2.8　节点分享格式

示例：

算法 2.7　分享格式

```
// 全小写
[
  {
    "tag": "node 1", // 选填，自定义名字
    "address": "jls.github.com", // 必填，地址
    "port": 443, // 必填，端口
    "sni": "jls.github.com", // 选填，server name 所用域名
    "password": "sou7230894hfksadjhfosdf", // 必填，随机密码
    "transport": "tcp", // 选填，默认TCP
    "timeout": 10, // 选填，默认10秒
    "fingerprint": "", // 选填,留空由软件随机选择
    "random": "2345kjgku2345234kj235kj23h4" // 必填，随机数
  }
]
```

参考文献

[1] Rescorla E. The transport layer security (tls) protocol version 1.3[R]. 2018.

[2] Langley A. Rfc 7685: A transport layer security (tls) clienthello padding extension[M]. RFC Editor, 2015.