Machine Translated by Google

Grupo de Trabajo de Ingeniería de Internet (IETF)
Solicitud de comentarios: 8446
Obsoletas: 5077, 5246, 6961
Actualizaciones: 5705, 6066
Categoría: Pista de estándares
ISSN: 2070-1721

E. Rescorla
Mozilla

Agosto 2018

Protocolo de seguridad de la capa de transporte (TLS) versión 1.3

Resumen

Este documento especifica la versión 1.3 del protocolo Transport Layer Security (TLS). TLS permite que las aplicaciones cliente/servidor se comuniquen a través de Internet de una manera diseñada para evitar escuchas ilegales, manipulación y falsificación de mensajes.

Este documento actualiza los RFC 5705 y 6066 y obsoleta los RFC 5077, 5246 y 6961. Este documento también especifica nuevos requisitos para las implementaciones de TLS 1.2.

Estado de este memorándum

Este es un documento de seguimiento de estándares de Internet.

Este documento es un producto del Grupo de Trabajo de Ingeniería de Internet (IETF). Representa el consenso de la comunidad IETF. Ha recibido revisión pública y ha sido aprobado para su publicación por el Grupo Directivo de Ingeniería de Internet (IESG). Más información sobre los Estándares de Internet está disponible en la Sección 2 de RFC 7841.

Puede obtener información sobre el estado actual de este documento, cualquier fe de erratas y cómo proporcionar comentarios al respecto en https://www.rfc-editor.org/info/rfc8446.

Machine Translated by Google

Aviso de copyright

Copyright (c) 2018 IETF Trust y las personas identificadas como los autores del documento. Todos los derechos reservados.

Este documento está sujeto a BCP 78 y las Disposiciones legales relacionadas con los documentos de IETF de IETF Trust (https://trustee.ietf.org/license-info) vigentes en la fecha de publicación de este documento. Revise estos documentos detenidamente, ya que describen sus derechos y restricciones con respecto a este documento. Los Componentes de Código extraídos de este documento deben incluir el texto de la Licencia BSD Simplificada como se describe en la Sección 4.e de las Disposiciones Legales de Fideicomiso y se proporcionan sin garantía como se describe en la Licencia BSD Simplificada.

Este documento puede contener material de Documentos IETF o Contribuciones IETF publicados o puestos a disposición del público antes del 10 de noviembre de 2008. Es posible que las personas que controlan los derechos de autor de parte de este material no hayan otorgado a IETF Trust el derecho de permitir modificaciones de dicho material. fuera del proceso de estándares del IETF.

Sin obtener una licencia adecuada de la(s) persona(s) que controla(n) los derechos de autor de dichos materiales, este documento no puede modificarse fuera del Proceso de estándares de la IETF, y no se pueden crear obras derivadas fuera del Proceso de estándares de la IETF, excepto para formatearlo para publicación como un RFC o para traducirlo a otros idiomas además del inglés.

Machine Translated by Google

Tabla de contenido

1. Introducción ....6 1.1. Convenciones y Terminología .....7 1.2. Principales diferencias con TLS 1.2 .....8 1.3. Actualizaciones que afectan a TLS 1.2 .....9 2. Descripción general del protocolo ....10 2.1. Porcentaje incorrecto de DHE .....14 2.2. Reanudación y clave precompartida (PSK) .....15 2.3. 0-Datos RTT ....17 3. Idioma de presentación .....19 3.1. Tamaño de bloque básico .....19 3.2. Misceláneos .....20 3.3. Números ....20 3.4. Vectores ....20 3.5. Enumerados .....21 3.6 . Tipos Construidos .....22 3.7. constantes ..... 23 3.8. variantes ..... .23 4. Protocolo de apretón de manos ..... .24 4.1. Mensajes de intercambio de claves .....25 4.1.1. Negociación criptográfica .....26 4.1.2. Hola al cliente .....27 4.1.3. Servidor Hola .....31 4.1.4. Solicitud de reintento de saludo .....33 4.2. Extensiones .....35 4.2.1. Versiones compatibles .....39 4.2.2. Galleta .....40 4.2.3 . Algoritmos de Firma .....41 4.2.4. Autoridades de Certificación .....45 4.2.5. Filtros OID .....45 4.2.6. Autenticación de cliente posterior al protocolo de enlace .....47 4.2.7. Grupos Soportados .....47 4.2.8. Clave compartida .....48 4.2.9. Modos de intercambio de claves precompartidas .....51 4.2.10. Indicación Temprana de Datos .....52 4.2.11. Extensión de clave precompartida .....55 4.3. Parámetros del servidor .....59 4.3.1. Extensiones encriptadas .....60 4.3.2. Solicitud de Certificado .....60 4.4. Mensajes de Autenticación .....61 4.4.1. El Hash de Transcripción .....63 4.4.2. Certificado .....64 4.4.3. Verificación de certificado .....69 4.4.4. Terminado .....71 4.5. Fin de los primeros datos 72

Machine Translated by Google

4.6. Mensajes Post-Handshake .....73 4.6.1. Mensaje de ticket de nueva sesión 73
4.6.2. Autenticación Post-Handshake .....75 4.6.3. Actualización de clave y vector de inicialización .....76 5. Protocolo de registro ....77 5.1. Capa de
grabación .....78 5.2 . Protección de la carga útil de registros ..... 80 5.3. Nonce por registro .....82 5.4. Relleno de registro ..... 83 5.5. Límites
en el uso de claves .....84 6. Protocolo de alerta .. ..85 6.1. Alertas
de Cierre .....87 6.2. Alertas de error .....88 7 Cálculos criptográficos .....90 7.1. Programación de claves .....91 7.2 .
Actualización de secretos de tráfico .....94 7.3. Cálculo de clave de tráfico 95
7.4. Cálculo del secreto compartido (EC)DHE .....95 7.4.1. Campo Finito Diffie-Hellman .....95 7.4.2. Curva Elíptica Diffie-Hellman .....96 7.5.
Exportadores ..... 97 8. 0-RTT y Anti-Replay ....98 8.1.
Billetes de un solo uso .....99 8.2. Grabación de saludo del cliente .....99 8.3.
Comprobaciones de frescura .....101 9. Requisitos de cumplimiento 102
9.1. Conjuntos de Cifrado Obligatorio de Implementar .....102 9.2. Extensiones Obligatorias de
Implementar .....103 9.3. Invariantes del protocolo ..... 104 10. Consideraciones
de seguridad ....106 11. Consideraciones de la IANA ....106 12.
Referencias ....109 12.1. Referencias Normativas 109
12.2. Referencias Informativas .....112 Apéndice A. Máquina de
Estados ....120 A.1. Cliente ....120 A.2.
servidor ....121 Apéndice B. Estructuras de datos de protocolo y valores constantes .....122 B.1. Capa de grabación .....122 B .2. Mensajes de
Alerta .....123 B.3 . Protocolo de apretón de manos .....124 B.3.1.
Mensajes de intercambio de claves .....125 B.3.2. Mensajes de Parámetros del
Servidor .....131 B.3.3. Mensajes de Autenticación .....132 B.3.4. Establecimiento de Billetes .....132 B.3.5. Actualización de Claves .....133 B.4. Conjuntos de
Cifrado 133

Machine Translated by Google

Apéndice C. Notas de implementación .....134 C.1. Generación y Siembra de Números Aleatorios .....134 C.2. Certificados y Autenticación .....135 C.3. Dificultades en la implementación .....135 C.4. Prevención de seguimiento de clientes ..... 137 C.5. Operación no autenticada .....137 Apéndice D. Compatibilidad con versiones anteriores ....138 D.1. Negociar con un servidor antiguo .....139 D.2. Negociar con un cliente antiguo .....139 D.3. Compatibilidad con versiones anteriores de 0-RTT .....140 D.4. Modo de compatibilidad de Middlebox .....140 D.5. Restricciones de seguridad relacionadas con la compatibilidad con versiones anteriores ..141 Apéndice E. Descripción general de las propiedades de seguridad .....142 E.1. Apretón de manos ..... 142 E.1.1. Derivación de clave y HKDF .....145 E.1.2. Autenticación de clientes .....146 E.1.3. 0-RTT ....146 E.1.4. Independencia del Exportador .....146 E.1.5. Seguridad Posterior al Compromiso .....146 E.1.6. Referencias Externas .....147 E.2. Capa de grabación .....147 E .2.1. Referencias Externas .....148 E.3. Análisis de tráfico .....148 E.4. Ataques de canal lateral ..... 149 E.5. Ataques de repetición en 0-RTT .....150 E.5.1. Replay y Exportadores .....151 E.6. Exposición de la identidad de PSK .....152 E.7. Compartir PSK .....152 S .8. Ataques a Static RSA .....152 Colaboradores ....153 Dirección del autor 160

Machine Translated by Google

1. Introducción

El objetivo principal de TLS es proporcionar un canal seguro entre dos pares que se comunican; el único requisito del transporte subyacente es un flujo de datos fiable y en orden. Específicamente, el canal seguro debe proporcionar las siguientes propiedades:

- Autenticación: El lado del servidor del canal siempre es autenticado; el lado del cliente se autentica opcionalmente. La autenticación puede ocurrir a través de criptografía asimétrica (p. ej., RSA [RSA], el algoritmo de firma digital de curva elíptica (ECDSA) [ECDSA], o el algoritmo de firma digital Edwards-Curve (EdDSA) [RFC8032]) o una clave precompartida simétrica (PSK).

- Confidencialidad: los datos enviados a través del canal después del establecimiento solo son visibles para los puntos finales. TLS no oculta la longitud de los datos que transmite, aunque los puntos finales pueden rellenar registros TLS para ocultar longitudes y mejorar la protección contra las técnicas de análisis de tráfico.

- Integridad: los datos enviados a través del canal después del establecimiento no pueden ser modificado por atacantes sin detección.

Estas propiedades deberían ser verdaderas incluso frente a un atacante que tiene el control total de la red, como se describe en [RFC3552]. Consulte el Apéndice E para obtener una declaración más completa de las propiedades de seguridad relevantes.

TLS consta de dos componentes principales:

- Un protocolo de protocolo de enlace (Sección 4) que autentica el partes que se comunican, negocia modos y parámetros criptográficos y establece material de clave compartido. El protocolo de apretón de manos está diseñado para resistir la manipulación; un atacante activo no debería poder obligar a los pares a negociar parámetros diferentes de los que harían si la conexión no estuviera bajo ataque.

- Un protocolo de registro (Sección 5) que utiliza los parámetros establecidos por el protocolo de negociación para proteger el tráfico entre los pares que se comunican. El protocolo de registro divide el tráfico en una serie de registros, cada uno de los cuales está protegido de forma independiente mediante las claves de tráfico.

Machine Translated by Google

TLS es independiente del protocolo de aplicación; los protocolos de nivel superior pueden superponerse a TLS de forma transparente. Sin embargo, el estándar TLS no especifica cómo los protocolos agregan seguridad con TLS; cómo iniciar el protocolo de enlace TLS y cómo interpretar los certificados de autenticación intercambiados se deja a criterio de los diseñadores e implementadores de protocolos que se ejecutan sobre TLS.

Este documento define la versión 1.3 de TLS. Si bien TLS 1.3 no es directamente compatible con versiones anteriores, todas las versiones de TLS incorporan un mecanismo de control de versiones que permite a los clientes y servidores negociar de manera interoperable una versión común si ambos admiten una.

colegas.

Este documento reemplaza y deja obsoletas las versiones anteriores de TLS, incluida la versión 1.2 [RFC5246]. También deja obsoleto el mecanismo de ticket TLS definido en [RFC5077] y lo reemplaza con el mecanismo definido en la Sección 2.2. Debido a que TLS 1.3 cambia la forma en que se derivan las claves, actualiza [RFC5705] como se describe en la Sección 7.5. También cambia la forma en que se transportan los mensajes del Protocolo de estado de certificado en línea (OCSP) y, por lo tanto, actualiza [RFC6066] y obsoleta [RFC6961] como se describe en la Sección 4.4.2.1.

1.1. Convenciones y Terminología

Las palabras clave "DEBE", "NO DEBE", "REQUERIDO", "DEBE", "NO DEBE", "DEBE", "NO DEBE", "RECOMENDADO", "NO RECOMENDADO", "PUEDE" y "OPCIONAL " en este documento deben interpretarse como se describe en BCP 14 [RFC2119] [RFC8174] cuando, y solo cuando, aparecen en mayúsculas, como se muestra aquí.

Se utilizan los siguientes términos:

cliente: el extremo que inicia la conexión TLS.

conexión: una conexión de capa de transporte entre dos puntos finales.

punto final: el cliente o el servidor de la conexión.

apretón de manos: una negociación inicial entre el cliente y el servidor que establece los parámetros de sus interacciones posteriores dentro de TLS.

par: Un punto final. Cuando se analiza un criterio de valoración en particular, "par" se refiere al criterio de valoración que no es el tema principal de discusión.

Machine Translated by Google

receptor: un punto final que está recibiendo registros.

remitente: Un punto final que está transmitiendo registros.

servidor: el extremo que no inició la conexión TLS.

1.2. Principales diferencias con TLS 1.2

La siguiente es una lista de las principales diferencias funcionales entre TLS 1.2 y TLS 1.3. No pretende ser exhaustivo, y hay muchas diferencias menores.

- La lista de algoritmos de cifrado simétrico compatibles ha sido podado de todos los algoritmos que se consideran heredados. Los que quedan son todos los algoritmos de cifrado autenticado con datos asociados (AEAD). El concepto de conjunto de cifrado se ha cambiado para separar los mecanismos de autenticación e intercambio de claves del algoritmo de protección de registros (incluida la longitud de la clave secreta) y un hash que se utilizará tanto con la función de derivación de claves como con el código de autenticación de mensajes de protocolo de enlace (MAC).

- Se agregó un modo de tiempo de ida y vuelta cero (0-RTT), ahorrando un viaje de ida y vuelta en la configuración de la conexión para algunos datos de la aplicación, a costa de ciertas propiedades de seguridad.

- Se han eliminado los conjuntos de cifrado Static RSA y Diffie-Hellman; todos los mecanismos de intercambio de claves basados en claves públicas ahora proporcionan secreto.

- Todos los mensajes de protocolo de enlace después de ServerHello ahora están encriptados. El mensaje EncryptedExtensions recientemente introducido permite que varias extensiones previamente enviadas sin cifrar en ServerHello también disfruten de protección de confidencialidad.

- Se han rediseñado las funciones de derivación clave. El nuevo diseño permite un análisis más sencillo por parte de los criptógrafos debido a sus propiedades mejoradas de separación de claves. La función de derivación de clave de extracción y expansión basada en HMAC (HKDF) se utiliza como una primitiva subyacente.

- La máquina de estado de apretón de manos se ha reestructurado significativamente para sea más consistente y elimine los mensajes superfluos como ChangeCipherSpec (excepto cuando sea necesario para la compatibilidad con middlebox).

- Los algoritmos de curva elíptica ahora están en la especificación básica y se incluyen nuevos algoritmos de firma, como EdDSA. TLS 1.3 eliminó la negociación de formato de punto a favor de un formato de punto único para cada curva.

Machine Translated by Google

- Se realizaron otras mejoras criptográficas, incluido el cambio del relleno RSA para usar el RSA Probabilistic Signature Scheme (RSASSA-PSS) y la eliminación de la compresión, el algoritmo de firma digital (DSA) y los grupos personalizados Ephemeral Diffie-Hellman (DHE).

- El mecanismo de negociación de la versión de TLS 1.2 ha quedado obsoleto en favor de una lista de versiones en una extensión. Esto aumenta la compatibilidad con los servidores existentes que implementaron incorrectamente la negociación de versiones.

- La reanudación de la sesión con y sin estado del lado del servidor, así como los conjuntos de cifrado basados en PSK de versiones anteriores de TLS, se han reemplazado por un solo intercambio de PSK nuevo.

- Las referencias se han actualizado para señalar las versiones actualizadas de los RFC, según corresponda (por ejemplo, RFC 5280 en lugar de RFC 3280).

1.3. Actualizaciones que afectan a TLS 1.2

Este documento define varios cambios que afectan opcionalmente las implementaciones de TLS 1.2, incluidas aquellas que no son compatibles con TLS 1.3:

- Un mecanismo de protección de versión anterior se describe en Sección 4.1.3.

- Los esquemas de firma RSASSA-PSS se definen en la Sección 4.2.3.

- La extensión ClientHello "supported_versions" se puede usar para negociar la versión de TLS que se usará, en lugar del campo legacy_version de ClientHello.

- La extensión "signature_algorithms_cert" permite que un cliente indique qué algoritmos de firma puede validar en certificados X.509.

Además, este documento aclara algunos requisitos de cumplimiento para versiones anteriores de TLS; consulte la Sección 9.3.

Machine Translated by Google

2. Descripción general del protocolo

Los parámetros criptográficos utilizados por el canal seguro son producidos por el protocolo de negociación TLS. El cliente y el servidor utilizan este subprotocolo de TLS cuando se comunican entre sí por primera vez. El protocolo de negociación permite a los pares negociar una versión de protocolo, seleccionar algoritmos criptográficos, opcionalmente autenticarse entre sí y establecer material de claves secretas compartidas. Una vez que se completa el protocolo de enlace, los pares usan las claves establecidas para proteger el tráfico de la capa de aplicación.

Una falla del protocolo de enlace u otro error de protocolo desencadena la terminación de la conexión, precedida opcionalmente por un mensaje de alerta (Sección 6).

TLS admite tres modos básicos de intercambio de claves:

- (EC)DHE (Diffie-Hellman sobre campos finitos o elípticos) curvas)

- Solo PSK

- PSK con (CE) Y

Machine Translated by Google

La Figura 1 a continuación muestra el protocolo de enlace TLS completo básico:

```
       Cliente                                                  Servidor

Llave  ^ ClienteHola
Intercambio | + clave_compartida*
            | + algoritmos_de_firma*
            | + psk_key_exchange_modes*
            v + pre_shared_key *          -------->
                                                       ServidorHola  ^ Llave
                                                  + clave_compartida* | Intercambio
                                            + pre_shared_key * v
                                       {Extensiones cifradas} ^          Servidor
                                       {CertificateRequest*} v Parámetros
                                                 {Certificado*} ^
                                        {CertificadoVerificar*} | Autorización
                                                   {Terminado} v
                                      <-------- [Datos de la aplicación*]
            ^ {Certificado*}
Autorización | {CertificadoVerificar*}
            {Terminado}               -------->
              [Datos de la aplicación] <-------> [Datos de la aplicación]

                  + Indica ampliaciones destacables enviadas en el
                        mensaje anotado anteriormente.

                  * Indica opcional o dependiente de la situación
                        mensajes/extensiones que no siempre se envían.

                  {} Indica mensajes protegidos mediante claves
                        derivado de un [remitente]_handshake_traffic_secret.

                  [] Indica mensajes protegidos mediante claves
                        derivado de [remitente]_application_traffic_secret_N.
```

Figura 1: Flujo de mensajes para el protocolo de enlace TLS completo

Se puede pensar que el apretón de manos tiene tres fases (indicadas en el diagrama de arriba):

- Intercambio de claves: establezca material de claves compartido y seleccione el parámetros criptográficos. Todo después de esta fase es encriptado.

- Parámetros del servidor: establecer otros parámetros de protocolo de enlace (si el cliente está autenticado, protocolo de capa de aplicación apoyo, etc).

Machine Translated by Google

- Autenticación: autentica el servidor (y, opcionalmente, el cliente) y proporciona confirmación de clave e integridad de protocolo de enlace.

En la fase de intercambio de claves, el cliente envía el mensaje ClientHello (Sección 4.1.2), que contiene un nonce aleatorio (ClientHello.random); sus versiones de protocolo ofrecidas; una lista de pares de hash HKDF/cifrado simétrico; ya sea un conjunto de claves compartidas Diffie-Hellman (en la extensión "key_share" (Sección 4.2.8)), un conjunto de etiquetas de claves previamente compartidas (en la extensión "pre_shared_key" (Sección 4.2.11)), o ambos; y potencialmente extensiones adicionales.

También pueden estar presentes campos y/o mensajes adicionales para compatibilidad con middlebox.

El servidor procesa ClientHello y determina los parámetros criptográficos apropiados para la conexión. Luego responde con su propio ServerHello (Sección 4.1.3), que indica los parámetros de conexión negociados. La combinación de ClientHello y ServerHello determina las claves compartidas. Si el establecimiento de clave (EC)DHE está en uso, entonces ServerHello contiene una extensión "key_share" con el recurso compartido efímero Diffie-Hellman del servidor; el recurso compartido del servidor DEBE estar en el mismo grupo que uno de los recursos compartidos del cliente. Si el establecimiento de la clave PSK está en uso, ServerHello contiene una extensión "pre_shared_key" que indica cuál de las PSK ofrecidas por el cliente se seleccionó. Tenga en cuenta que las implementaciones pueden usar (EC)DHE y PSK juntos, en cuyo caso se proporcionarán ambas extensiones.

El servidor luego envía dos mensajes para establecer el servidor Parámetros:

EncryptedExtensions: respuestas a las extensiones de ClientHello que son no se requiere para determinar los parámetros criptográficos, distintos de los que son específicos de los certificados individuales.
[Sección 4.3.1]

CertificateRequest: si se desea la autenticación de cliente basada en certificados, los parámetros deseados para ese certificado. Este mensaje se omite si no se desea la autenticación del cliente.

[Sección 4.3.2]

Machine Translated by Google

Finalmente, el cliente y el servidor intercambian mensajes de autenticación. TLS usa el mismo conjunto de mensajes cada vez que se necesita la autenticación basada en certificados. (La autenticación basada en PSK ocurre como un efecto secundario del intercambio de claves). Específicamente:

Certificado: el certificado del extremo y cualquier extensión por certificado. Este mensaje lo omite el servidor si no se autentica con un certificado y el cliente si el servidor no envió CertificateRequest (lo que indica que el cliente no debe autenticarse con un certificado). Tenga en cuenta que si se utilizan claves públicas sin procesar [RFC7250] o la extensión de información en caché [RFC7924], este mensaje no contendrá un certificado sino algún otro valor correspondiente a la clave a largo plazo del servidor. [Sección 4.4.2]

CertificateVerify: una firma sobre todo el protocolo de enlace utilizando la clave privada correspondiente a la clave pública en el mensaje del Certificado. Este mensaje se omite si el extremo no se autentica a través de un certificado. [Sección 4.4.3]

Terminado: un MAC (Código de autenticación de mensajes) en todo el apretón de manos. Este mensaje proporciona confirmación de clave, vincula la identidad del extremo a las claves intercambiadas y, en el modo PSK, también autentica el protocolo de enlace. [Sección 4.4.4]

Al recibir los mensajes del servidor, el cliente responde con sus mensajes de autenticación, a saber, Certificate y CertificateVerify (si se solicita) y Finished.

En este punto, el protocolo de enlace está completo y el cliente y el servidor derivan el material de clave requerido por la capa de registro para intercambiar datos de la capa de aplicación protegidos mediante cifrado autenticado.

Los datos de la aplicación NO DEBEN enviarse antes de enviar el mensaje Finalizado, excepto como se especifica en la Sección 2.3. Tenga en cuenta que si bien el servidor puede enviar datos de la aplicación antes de recibir los mensajes de autenticación del cliente, todos los datos enviados en ese momento, por supuesto, se envían a un par no autenticado.

Machine Translated by Google

2.1. Incorrecto Y Compartir

Si el cliente no ha proporcionado una extensión "key_share" suficiente (por ejemplo, incluye solo grupos DHE o ECDHE inaceptables o no compatible con el servidor), el servidor corrige la discrepancia con un HelloRetryRequest y el cliente necesita reiniciar el protocolo de enlace con una extensión "key_share" apropiada, como se muestra en la Figura 2. Si no se pueden negociar parámetros criptográficos comunes, el servidor DEBE cancelar el apretón de manos con una alerta adecuada.

```
Cliente                                                    Servidor

ClienteHola
+ clave_compartida          -------->
                                                   HolaReintentarSolicitud
                            <--------                + clave_compartida
ClienteHola
+ clave_compartida          -------->
                                                   ServidorHola
                                                   + clave_compartida
                                         {Extensiones cifradas}
                                         {Solicitud de certificado *}
                                                   {Certificado*}
                                         {CertificadoVerificar*}
                                                   {Finalizado}
                            <--------    [Datos de la aplicación*]
{Certificado*}
{CertificadoVerificar*}
{Finalizado}                -------->
[Datos de la aplicación]    <------->    [Datos de la aplicación]
```

Figura 2: Flujo de mensajes para un apretón de manos completo con Parámetros no coincidentes

Nota: La transcripción del apretón de manos incorpora la inicial ClientHello/HelloRetryRequest intercambio; no se reinicia con el nuevo clienteHola.

TLS también permite varias variantes optimizadas del protocolo de enlace básico, como descrito en las siguientes secciones.

Machine Translated by Google

2.2. Reanudación y clave precompartida (PSK)

Aunque los PSK de TLS se pueden establecer fuera de banda, los PSK también se pueden establecer en una conexión anterior y luego usarse para establecer una nueva conexión ("reanudación de la sesión" o "reanudación" con un PSK). Una vez que se ha completado un protocolo de enlace, el servidor puede enviar al cliente una identidad PSK que corresponde a una clave única derivada del protocolo de enlace inicial (consulte la Sección 4.6.1). Luego, el cliente puede usar esa identidad de PSK en futuros protocolos de enlace para negociar el uso del PSK asociado. Si el servidor acepta el PSK, el contexto de seguridad de la nueva conexión se vincula criptográficamente a la conexión original y la clave derivada del protocolo de enlace inicial se utiliza para iniciar el estado criptográfico en lugar de un protocolo de enlace completo. En TLS 1.2 y versiones anteriores, esta funcionalidad la proporcionaban los "ID de sesión" y los "tickets de sesión" [RFC5077]. Ambos mecanismos están obsoletos en TLS 1.3.

Los PSK se pueden usar con el intercambio de claves (EC)DHE para proporcionar confidencialidad directa en combinación con claves compartidas, o se pueden usar solos, a costa de perder la confidencialidad directa de los datos de la aplicación.

Machine Translated by Google

La figura 3 muestra un par de apretones de manos en los que el primer apretón de manos establece un PSK y el segundo apretón de manos lo usa:

```
       Cliente                                                  Servidor

Apretón de manos inicial:
       ClienteHola
       + clave_compartida          -------->
                                                       ServidorHola
                                                      + clave_compartida
                                             {Extensiones cifradas}
                                             {Solicitud de certificado *}
                                                   {Certificado*}
                                              {CertificadoVerificar*}
                                                       {Finalizado}
                                   <--------  [Datos de la aplicación*]
       {Certificado*}
       {CertificadoVerificar*}
       {Finalizado}                -------->
                                   <--------      [Nuevo boleto de sesión]
       [Datos de la aplicación]    <------->      [Datos de la aplicación]


Apretón de manos posterior:
       ClienteHola
       + clave_compartida*
       + pre_shared_key            -------->
                                                       ServidorHola
                                                 + pre_shared_key
                                                + clave_compartida*
                                             {Extensiones cifradas}
                                                       {Finalizado}
                                   <--------  [Datos de la aplicación*]
       {Finalizado}                -------->
       [Datos de la aplicación]    <------->      [Datos de la aplicación]
```

Figura 3: Flujo de mensajes para reanudación y PSK

Como el servidor se está autenticando a través de un PSK, no envía un Certificado o un mensaje CertificateVerify. Cuando un cliente ofrece reanudación a través de un PSK, también DEBERÍA proporcionar una extensión "key_share" al servidor para permitir que el servidor rechace la reanudación y retroceda a un apretón de manos completo, si es necesario. El servidor responde con un Extensión "pre_shared_key" para negociar el uso de la clave PSK establecimiento y puede (como se muestra aquí) responder con un "key_share" extensión para hacer (EC) DHE establecimiento clave, proporcionando así adelante secreto.

Machine Translated by Google

Cuando los PSK se aprovisionan fuera de banda, también DEBEN proporcionarse la identidad del PSK y el algoritmo hash de KDF que se utilizará con el PSK.

Nota: Cuando se utiliza un secreto precompartido aprovisionado fuera de banda, un una consideración crítica es usar suficiente entropía durante la generación de claves, como se explica en [RFC4086]. Derivar un secreto compartido de una contraseña u otras fuentes de baja entropía no es seguro. Un secreto o contraseña de baja entropía está sujeto a ataques de diccionario basados en el enlace PSK. La autenticación PSK especificada no es un intercambio de clave autenticado basado en contraseña fuerte, incluso cuando se usa con el establecimiento de clave Diffie-Hellman. Específicamente, no evita que un atacante que pueda observar el protocolo de enlace realice un ataque de fuerza bruta en la contraseña/clave precompartida.

2.3. 0-RTT datos

Cuando los clientes y los servidores comparten un PSK (obtenido externamente o mediante un protocolo de enlace anterior), TLS 1.3 permite a los clientes enviar datos en el primer vuelo ("datos iniciales"). El cliente usa el PSK para autenticar el servidor y cifrar los primeros datos.

Como se muestra en la Figura 4, los datos 0-RTT solo se agregan al protocolo de enlace 1-RTT en el primer vuelo. El resto del protocolo de enlace utiliza los mismos mensajes que para un protocolo de enlace de 1 RTT con reanudación de PSK.

Machine Translated by Google


```
Cliente                                                    Servidor

ClienteHola
+ datos_tempranos
+ clave_compartida*
+ psk_key_exchange_modes
+ pre_shared_key
 (Datos de la aplicación*)              -------->
                                                       ServidorHola
                                                  + pre_shared_key
                                                      + clave_compartida*
                                             {Extensiones cifradas}
                                                  + datos_tempranos*
                                                       {Finalizado}
                                        <--------  [Datos de la aplicación*]
 (EndOfEarlyData)
{Finalizado}                            -------->
[Datos de la aplicación]                <-------> [Datos de la aplicación]
```

+ Indica ampliaciones destacables enviadas en el mensaje anotado anteriormente.

* Indica opcional o dependiente de la situación mensajes/extensiones que no siempre se envían.

() Indica mensajes protegidos mediante claves derivado de un client_early_traffic_secret.

{} Indica mensajes protegidos mediante claves derivado de un [remitente]_handshake_traffic_secret.

[] Indica mensajes protegidos mediante claves derivado de [remitente]_application_traffic_secret_N.

Figura 4: Flujo de mensajes para un protocolo de enlace 0-RTT

Machine Translated by Google

NOTA IMPORTANTE: Las propiedades de seguridad de los datos 0-RTT son más débiles que las de otros tipos de datos TLS. Específicamente:

1. Estos datos no son secretos directos, ya que están encriptados únicamente bajo claves derivadas usando el PSK ofrecido.

2. No hay garantías de no reproducción entre conexiones.
   La protección contra la reproducción para datos TLS 1.3 1-RTT ordinarios se proporciona a través del valor aleatorio del servidor, pero los datos 0-RTT no dependen de ServerHello y, por lo tanto, tienen garantías más débiles.
   Esto es especialmente relevante si los datos se autentican con la autenticación de cliente TLS o dentro del protocolo de la aplicación. Las mismas advertencias se aplican a cualquier uso de early_exporter_master_secret.

Los datos 0-RTT no se pueden duplicar dentro de una conexión (es decir, el servidor no procesará los mismos datos dos veces para la misma conexión), y un atacante no podrá hacer que los datos 0-RTT parezcan datos 1-RTT (porque está protegido con claves diferentes). El Apéndice E.5 contiene una descripción de posibles ataques y la Sección 8 describe los mecanismos que el servidor puede usar para limitar el impacto de la reproducción.

3. Idioma de presentación

Este documento trata sobre el formateo de datos en una representación externa. Se utilizará la siguiente sintaxis de presentación muy básica y un tanto informal.

3.1. Tamaño de bloque básico

La representación de todos los elementos de datos se especifica explícitamente. El tamaño básico del bloque de datos es de un byte (es decir, 8 bits). Los elementos de datos de varios bytes son concatenaciones de bytes, de izquierda a derecha y de arriba a abajo. A partir del flujo de bytes, se forma un elemento de varios bytes (un número en el siguiente ejemplo) (utilizando la notación C) mediante:

```
valor = (byte[0] << 8*(n-1)) | (byte[1] << 8*(n-2)) |
                ... | byte[n-1];
```

Este orden de bytes para valores de varios bytes es el orden de bytes de red común o formato big-endian.

Machine Translated by Google

3.2. Diverso

Los comentarios comienzan con "/*" y terminan con "*/".

Los componentes opcionales se indican encerrándolos entre "[[ ]]" (corchetes dobles).

Las entidades de un solo byte que contienen datos no interpretados son de tipo opaco.

Un alias de tipo T' para un tipo T existente se define mediante:

```
TT';
```

3.3. Números

El tipo de datos numérico básico es un byte sin signo (uint8). Todos los tipos de datos numéricos más grandes se construyen a partir de una serie de bytes de longitud fija concatenados como se describe en la Sección 3.1 y tampoco están firmados.
Los siguientes tipos numéricos están predefinidos.

```
uint8 uint16[2]; uint8
uint24[3]; uint8 uint32[4];
uint8 uint64[8];
```

Todos los valores, aquí y en otras partes de la especificación, se transmiten en orden de bytes de red (big-endian); el uint32 representado por los bytes hexadecimales 01 02 03 04 es equivalente al valor decimal 16909060.

3.4. Vectores

Un vector (arreglo unidimensional) es un flujo de elementos de datos homogéneos. El tamaño del vector puede especificarse en el momento de la documentación o dejarse sin especificar hasta el tiempo de ejecución. En cualquier caso, la longitud declara el número de bytes, no el número de elementos, en el vector. La sintaxis para especificar un nuevo tipo, T', que es un vector de longitud fija de tipo T es

```
TT'[n];
```

Aquí, T' ocupa n bytes en el flujo de datos, donde n es un múltiplo del tamaño de T. La longitud del vector no está incluida en el flujo codificado.

Machine Translated by Google

En el siguiente ejemplo, Datum se define como tres bytes consecutivos que el protocolo no interpreta, mientras que Data son tres Datum consecutivos que consumen un total de nueve bytes.

```
fecha opaca[3];          /* tres bytes no interpretados */ /* tres vectores
Datos de fecha [9];      consecutivos de 3 bytes */
```

Los vectores de longitud variable se definen especificando un subrango de longitudes legales, inclusive, utilizando la notación <suelo..techo>. Cuando estos están codificados, la longitud real precede al contenido del vector en el flujo de bytes. La longitud tendrá la forma de un número que consumirá tantos bytes como sea necesario para contener la longitud máxima (techo) especificada del vector. Un vector de longitud variable con un campo de longitud real de cero se denomina vector vacío.

```
T T'<suelo..techo>;
```

En el siguiente ejemplo, "obligatorio" es un vector que debe contener entre 300 y 400 bytes de tipo opaco. Nunca puede estar vacío.
El campo de longitud real consume dos bytes, un uint16, que es suficiente para representar el valor 400 (consulte la Sección 3.3). De manera similar, "más largo" puede representar hasta 800 bytes de datos, o 400 elementos uint16, y puede estar vacío. Su codificación incluirá un campo de longitud real de dos bytes antepuesto al vector. La longitud de un vector codificado debe ser un múltiplo exacto de la longitud de un solo elemento (p. ej., un vector de 17 bytes de uint16 sería ilegal).

```
opaco obligatorio<300..400>; /* el campo de
        longitud es de dos bytes, no puede estar vacío */ uint16 long<0..800>; /* cero a
400 enteros de 16 bits sin signo */
```

3.5. enumerados

Está disponible un tipo de datos dispersos adicional, llamado "enumerado" o "enumerado". Cada definición es de un tipo diferente. Sólo se pueden asignar o comparar enumerados del mismo tipo. A cada elemento de un enumerado se le debe asignar un valor, como se demuestra en el siguiente ejemplo. Dado que los elementos de los enumerados no están ordenados, se les puede asignar cualquier valor único, en cualquier orden.

```
enumeración { e1(v1), e2(v2), ... ,          en(vn) [[, (n)]] } Te;
```

Futuras extensiones o adiciones al protocolo pueden definir nuevos valores.
Las implementaciones deben poder analizar e ignorar valores desconocidos a menos que la definición del campo indique lo contrario.

Machine Translated by Google

Un enumerado ocupa tanto espacio en el flujo de bytes como lo haría su valor ordinal máximo definido. La siguiente definición haría que se usara un byte para transportar campos de tipo Color.

```
enum { rojo (3), azul (5), blanco (7) } Color;
```

Opcionalmente, se puede especificar un valor sin su etiqueta asociada para forzar la definición de ancho sin definir un elemento superfluo.

En el siguiente ejemplo, Taste consumirá dos bytes en el flujo de datos pero solo puede asumir los valores 1, 2 o 4 en la versión actual del protocolo.

```
enum { dulce (1), agrio (2), amargo (4), (32000) } Sabor;
```

Los nombres de los elementos de una enumeración están dentro del ámbito del tipo definido. En el primer ejemplo, una referencia completa al segundo elemento de la enumeración sería Color.blue. No se requiere tal calificación si el objetivo de la tarea está bien especificado.

```
Color color = Color.azul; Color color =      /* sobreespecificado, legal */ /* correcto,
azul;                                        tipo implícito */
```

Los nombres asignados a los enumerados no necesitan ser únicos. El valor numérico puede describir un rango en el que se aplica el mismo nombre. El valor incluye los valores inclusivos mínimo y máximo en ese rango, separados por dos puntos. Esto es principalmente útil para reservar regiones del espacio.

```
enum { triste(0), meh(1..254), feliz(255) } Estado de ánimo;
```

3.6. Tipos construidos

Los tipos de estructura pueden construirse a partir de tipos primitivos por conveniencia. Cada especificación declara un tipo nuevo y único. La sintaxis utilizada para las definiciones es muy parecida a la de C.

```
estructura {
        T1 f1;
        T2 f2;
         ...
        Tnfn;
}T;
```

Los campos vectoriales de longitud fija y variable se permiten mediante la sintaxis de vectores estándar. Las estructuras V1 y V2 en el ejemplo de variantes (Sección 3.8) demuestran esto.

Machine Translated by Google

Los campos dentro de una estructura se pueden calificar usando el nombre del tipo, con una sintaxis muy parecida a la disponible para enumerados. Por ejemplo, T.f2 se refiere al segundo campo de la declaración anterior.

3.7. constantes

A los campos y variables se les puede asignar un valor fijo usando "=", como en:

```
estructura
        { T1 f1 = 8; /* T.f1 siempre debe ser 8 */ T2 f2; }T;
```

3.8. variantes

Las estructuras definidas pueden tener variantes basadas en algún conocimiento que esté disponible en el entorno. El selector debe ser un tipo enumerado que define las posibles variantes que define la estructura. Cada rama de la selección (a continuación) especifica el tipo de campo de esa variante y una etiqueta de campo opcional. El mecanismo por el cual se selecciona la variante en tiempo de ejecución no está prescrito por el lenguaje de presentación.

```
estructura {
        T1 f1;
        T2 f2;
         ....
        Tnfn;
        seleccione (E)
                { caso e1: Te1 [[fe1]]; caso e2:
                Te2 [[fe2]];
                 ....
                case en: Ten [[fen]];
        };
} Televisor;
```

Machine Translated by Google

Por ejemplo:

```
enum { manzana(0), naranja(1) } VariantTag;

estructura
        { uint16 número;
        cadena opaca<0..10>; /* longitud variable */ } V1;


estructura
        {uint32 numero;
        cadena opaca[10];                    /* longitud fija */
} V2;

estructura
        { Tipo de etiqueta variante;
        seleccione (VariantRecord.type) { case apple:
                V1; caja naranja: V2;

        };
} RegistroVariante;
```

4. Protocolo de apretón de manos

El protocolo de negociación se utiliza para negociar los parámetros de seguridad de una conexión. Los mensajes de protocolo de enlace se envían a la capa de registro TLS, donde se encapsulan dentro de una o más estructuras TLSPlaintext o TLSCiphertext que se procesan y transmiten según lo especificado por el estado de conexión activo actual.

Machine Translated by Google


```
enumeración {
        cliente_hola(1),
        servidor_hola(2),
        boleto_nueva_sesión(4),
        fin_de_los_datos_antiguos(5),
        extensiones_cifradas(8),
        certificado(11),
        solicitud_certificado(13),
        certificado_verificar(15),
        terminado(20),
        actualización_clave(24),
        mensaje_hash(254),
        (255)
} Tipo de apretón de manos;

estructura {
        HandshakeType msg_type; uint24          /* tipo de apretón de manos */
        longitud; seleccione (Apretón de        /* bytes restantes en el mensaje */
        manos.msg_type) {
                caso cliente_hola: ClienteHola;
                case servidor_hola: ServidorHola;
                case end_of_early_data: EndOfEarlyData;
                caso extensiones_cifradas: extensiones_cifradas;
                case certificate_request: CertificateRequest;
                caso certificado: caso                  Certificado;
                certificado_verificar: caso finalizado: CertificadoVerificar;
                caso nuevo_ticket_sesión: caso          Finalizado;
                clave_actualización:                    NewSessionTicket;
                                                        Actualización de clave;
        };
} Apretón de manos;
```

Los mensajes de protocolo DEBEN enviarse en el orden definido en la Sección 4.4.1 y se muestra en los diagramas en la Sección 2. Un compañero que recibe un mensaje de apretón de manos en un orden inesperado DEBE abortar el apretón de manos con una alerta de "mensaje_inesperado".

La IANA asigna nuevos tipos de mensajes de protocolo de enlace como se describe en Sección 11.

4.1. Mensajes de intercambio de claves

Los mensajes de intercambio de claves se utilizan para determinar la seguridad capacidades del cliente y el servidor y establecer secretos, incluidas las claves de tráfico utilizadas para proteger el resto de los apretón de manos y los datos.

Machine Translated by Google

4.1.1. Negociación criptográfica

En TLS, la negociación criptográfica procede cuando el cliente ofrece los siguientes cuatro conjuntos de opciones en su ClientHello:

- Una lista de conjuntos de cifrado que indica el algoritmo AEAD/HKDF pares de hash que admite el cliente.

- Una extensión "supported_groups" (Sección 4.2.7) que indica los grupos (EC)DHE que admite el cliente y una "key_share" (Sección 4.2.8) extensión que contiene acciones (EC)DHE para algunos o todos estos grupos.

- Una extensión "signature_algorithms" (Sección 4.2.3) que indica los algoritmos de firma que el cliente puede aceptar. También se puede agregar una extensión "signature_algorithms_cert" (Sección 4.2.3) para indicar algoritmos de firma específicos del certificado.

- Una extensión "pre_shared_key" (Sección 4.2.11) que contiene una lista de identidades de claves simétricas conocidas por el cliente y una extensión "psk_key_exchange_modes" (Sección 4.2.9) que indica los modos de intercambio de claves que se pueden usar con PSK.

Si el servidor no selecciona un PSK, las primeras tres de estas opciones son completamente ortogonales: el servidor selecciona de forma independiente un conjunto de cifrado, un grupo (EC)DHE y un recurso compartido de claves para el establecimiento de claves, y un par de certificado/algoritmo de firma para autenticar mismo al cliente. Si no hay superposición entre los "supported_groups" recibidos y los grupos admitidos por el servidor, entonces el servidor DEBE cancelar el protocolo de enlace con una alerta de "handshake_failure" o "insuficient_security".

Si el servidor selecciona un PSK, también DEBE seleccionar un modo de establecimiento de clave del conjunto indicado por la extensión "psk_key_exchange_modes" del cliente (actualmente, PSK solo o con (EC)DHE). Tenga en cuenta que si el PSK se puede utilizar sin (EC)DHE, entonces la no superposición en los parámetros "supported_groups" no tiene por qué ser fatal, como lo es en el caso de no PSK discutido en el párrafo anterior.

Si el servidor selecciona un grupo (EC)DHE y el cliente no ofreció una extensión "key_share" compatible en el ClientHello inicial, el servidor DEBE responder con un mensaje HelloRetryRequest (Sección 4.1.4).

Machine Translated by Google

Si el servidor selecciona correctamente los parámetros y no requiere un HelloRetryRequest, indica los parámetros seleccionados en el ServerHello de la siguiente manera:

- Si se usa PSK, el servidor enviará una "pre_shared_key" extensión que indica la tecla seleccionada.

- Cuando (EC)DHE está en uso, el servidor también proporcionará una extensión "key_share". Si no se utiliza PSK, siempre se utilizan (EC)DHE y la autenticación basada en certificados.

- Al autenticarse a través de un certificado, el servidor enviará el Mensajes de Certificate (Sección 4.4.2) y CertificateVerify (Sección 4.4.3). En TLS 1.3 como se define en este documento, siempre se usa un PSK o un certificado, pero no ambos. Los documentos futuros pueden definir cómo usarlos juntos.

Si el servidor no puede negociar un conjunto de parámetros compatibles (es decir, no hay superposición entre los parámetros del cliente y del servidor), DEBE abortar el protocolo de enlace con una alerta fatal "handshake_failure" o "insuficient_security" (consulte la Sección 6).

4.1.2. cliente hola

Cuando un cliente se conecta por primera vez a un servidor, se REQUIERE enviar ClientHello como su primer mensaje TLS. El cliente también enviará un ClientHello cuando el servidor haya respondido a su ClientHello con un HelloRetryRequest. En ese caso, el cliente DEBE enviar el mismo ClientHello sin modificaciones, excepto lo siguiente:

- Si se proporcionó una extensión "key_share" en HelloRetryRequest, reemplazando la lista de recursos compartidos con una lista que contiene una sola KeyShareEntry del grupo indicado.

- Eliminación de la extensión "early_data" (Sección 4.2.10) si estaba presente. Los primeros datos no están permitidos después de una HelloRetryRequest.

- Incluir una extensión de "cookie" si se proporcionó una en el HelloRetryRequest.

Machine Translated by Google

- Actualización de la extensión "pre_shared_key", si está presente, volviendo a calcular los valores de "ofuscated_ticket_age" y binder y (opcionalmente) eliminando cualquier PSK que sea incompatible con el conjunto de cifrado indicado del servidor.

- Opcionalmente agregar, quitar o cambiar la longitud de la extensión "relleno" [RFC7685].

- Otras modificaciones que pueda permitir una prórroga definida en el futuro y el presente en HelloRetryRequest.

Debido a que TLS 1.3 prohíbe la renegociación, si un servidor ha negociado TLS 1.3 y recibe un ClientHello en cualquier otro momento, DEBE finalizar la conexión con una alerta de "mensaje_inesperado".

Si un servidor estableció una conexión TLS con una versión anterior de TLS y recibe un TLS 1.3 ClientHello en una renegociación, DEBE conservar la versión anterior del protocolo. En particular, NO DEBE negociar TLS 1.3.

Estructura de este mensaje:

```
uint16 Versión de protocolo; Aleatorio
opaco[32];

uint8 CipherSuite[2];                    /* Selector de suite criptográfica */

struct
        { ProtocolVersion legacy_version = 0x0303; Aleatorio aleatorio;          /* TLS v1.2 */
        opaco legacy_session_id<0..32>; CipherSuite
        cipher_suites<2..2^16-2>; métodos_compresión_heredados
        opacos<1..2^8-1>; Extensión extensiones<8..2^16-1>; }
        ClienteHola;
```

Machine Translated by Google

legacy_version: en versiones anteriores de TLS, este campo se usaba para la negociación de versiones y representaba el número de versión más alto admitido por el cliente. La experiencia ha demostrado que muchos servidores no implementan correctamente la negociación de versiones, lo que lleva a una "intolerancia a la versión" en la que el servidor rechaza un ClientHello aceptable con un número de versión superior al que admite. En TLS 1.3, el cliente indica sus preferencias de versión en la extensión "supported_versions" (Sección 4.2.1) y el campo legacy_version DEBE establecerse en 0x0303, que es el número de versión para TLS 1.2. Los ClientHellos de TLS 1.3 se identifican con una versión heredada de 0x0303 y una extensión de versiones admitidas presente con 0x0304 como la versión más alta indicada allí.

(Consulte el Apéndice D para obtener detalles sobre la compatibilidad con versiones anteriores).

aleatorio: 32 bytes generados por un generador seguro de números aleatorios. Ver Apéndice C para obtener información adicional.

legacy_session_id: las versiones de TLS anteriores a TLS 1.3 admitían un función de "reanudación de la sesión" que se ha fusionado con claves previamente compartidas en esta versión (consulte la Sección 2.2). Un cliente que tiene un ID de sesión en caché establecido por un servidor anterior a TLS 1.3 DEBE establecer este campo en ese valor. En el modo de compatibilidad (consulte el Apéndice D.4), este campo NO DEBE estar vacío, por lo que un cliente que no ofrezca una sesión anterior a TLS 1.3 DEBE generar un nuevo valor de 32 bytes. Este valor no necesita ser aleatorio, pero DEBE ser impredecible para evitar que las implementaciones se fijen en un valor específico (también conocido como osificación). De lo contrario, DEBE establecerse como un vector de longitud cero (es decir, un campo de longitud de byte único de valor cero).

cipher_suites: una lista de las opciones de cifrado simétrico admitidas por el cliente, específicamente el algoritmo de protección de registros (incluida la longitud de la clave secreta) y un hash para usar con HKDF, en orden descendente de preferencia del cliente. Los valores se definen en el Apéndice B.4. Si la lista contiene conjuntos de cifrado que el servidor no reconoce, admite o no desea utilizar, el servidor DEBE ignorar esos conjuntos de cifrado y procesar los restantes como de costumbre. Si el cliente intenta establecer una clave PSK, DEBERÍA anunciar al menos un conjunto de cifrado que indique un Hash asociado con la PSK.

Machine Translated by Google

legacy_compression_methods: versiones de TLS anteriores a 1.3 admitidas compresión con la lista de métodos de compresión admitidos que se envían en este campo. Para cada TLS 1.3 ClientHello, este vector DEBE contener exactamente un byte, establecido en cero, que corresponde al método de compresión "nulo" en versiones anteriores de TLS. Si se recibe un TLS 1.3 ClientHello con cualquier otro valor en este campo, el servidor DEBE cancelar el protocolo de enlace con una alerta de "parámetro_illegal". Tenga en cuenta que los servidores TLS 1.3 pueden recibir TLS 1.2 o ClientHellos anteriores que contienen otros métodos de compresión y (si negocia una versión anterior) DEBEN seguir los procedimientos para la versión anterior adecuada de TLS.

Extensiones: los clientes solicitan una funcionalidad ampliada de los servidores mediante el envío de datos en el campo de extensiones. El formato real de "Extensión" se define en la Sección 4.2. En TLS 1.3, el uso de ciertas extensiones es obligatorio, ya que la funcionalidad se trasladó a las extensiones para preservar la compatibilidad de ClientHello con versiones anteriores de TLS. Los servidores DEBEN ignorar las extensiones no reconocidas.

Todas las versiones de TLS permiten que un campo de extensiones siga opcionalmente al campo de métodos de compresión. Los mensajes TLS 1.3 ClientHello siempre contienen extensiones (mínimamente "versiones_compatibles", de lo contrario, se interpretarán como mensajes TLS 1.2 ClientHello). Sin embargo, los servidores TLS 1.3 pueden recibir mensajes ClientHello sin un campo de extensiones de versiones anteriores de TLS. La presencia de extensiones se puede detectar determinando si hay bytes después del campopression_methods al final de ClientHello. Tenga en cuenta que este método de detección de datos opcionales difiere del método TLS normal de tener un campo de longitud variable, pero se usa para la compatibilidad con TLS antes de que se definieran las extensiones. Los servidores TLS 1.3 deberán realizar esta verificación primero y solo intentar negociar TLS 1.3 si la extensión "supported_versions" está presente. Si se negocia una versión de TLS anterior a la 1.3, un servidor DEBE verificar que el mensaje no contenga datos después de legacy_compression_methods o que contenga un bloque de extensiones válido sin datos a continuación. De lo contrario, DEBE abortar el protocolo de enlace con una alerta de "decode_error".

En el caso de que un cliente solicite una funcionalidad adicional utilizando extensiones y esta funcionalidad no sea proporcionada por el servidor, el cliente PUEDE abortar el protocolo de enlace.

Después de enviar el mensaje ClientHello, el cliente espera un mensaje ServerHello o HelloRetryRequest. Si se están utilizando los primeros datos, el cliente puede transmitir los primeros datos de la aplicación (Sección 2.3) mientras espera el siguiente mensaje de protocolo de enlace.

Machine Translated by Google

4.1.3. servidor hola

El servidor enviará este mensaje en respuesta a un mensaje ClientHello para continuar con el protocolo de enlace si puede negociar un conjunto aceptable de parámetros de protocolo de enlace basado en ClientHello.

Estructura de este mensaje:

```
struct
    { ProtocolVersion legacy_version = 0x0303; Aleatorio aleatorio;      /* TLS v1.2 */
    opaco legacy_session_id_echo<0..32>; CipherSuite cipher_suite;
    uint8 método_compresión_heredado = 0; Extensión
    extensiones<6..2^16-1>; } ServidorHola;
```

legacy_version: en versiones anteriores de TLS, este campo se usaba para la negociación de versiones y representaba el número de versión seleccionado para la conexión. Desafortunadamente, algunos cuadros intermedios fallan cuando se les presentan nuevos valores. En TLS 1.3, el servidor TLS indica su versión mediante la extensión "supported_versions" (Sección 4.2.1), y el campo legacy_version DEBE establecerse en 0x0303, que es el número de versión para TLS 1.2. (Consulte el Apéndice D para obtener detalles sobre la compatibilidad con versiones anteriores).

aleatorio: 32 bytes generados por un generador seguro de números aleatorios. Consulte el Apéndice C para obtener información adicional. Los últimos 8 bytes DEBEN sobrescribirse como se describe a continuación si se negocia TLS 1.2 o TLS 1.1, pero los bytes restantes DEBEN ser aleatorios. Esta estructura es generada por el servidor y DEBE generarse independientemente de ClientHello.random.

legacy_session_id_echo: El contenido del cliente campo legacy_session_id. Tenga en cuenta que este campo se repite incluso si el valor del cliente correspondía a una sesión anterior a TLS 1.3 almacenada en caché que el servidor decidió no reanudar. Un cliente que recibe un campo legacy_session_id_echo que no coincide con lo que envió en ClientHello DEBE abortar el protocolo de enlace con una alerta de "parámetro_illegal".

cipher_suite: El único conjunto de cifrado seleccionado por el servidor de la lista en ClientHello.cipher_suites. Un cliente que recibe un conjunto de cifrado que no se ofreció DEBE abortar el protocolo de enlace con una alerta de "parámetro_illegal".

legacy_compression_method: Un solo byte que DEBE tener el valor 0.

Machine Translated by Google

extensiones: Una lista de extensiones. ServerHello DEBE incluir únicamente las extensiones necesarias para establecer el contexto criptográfico y negociar la versión del protocolo. Todos los mensajes TLS 1.3 ServerHello DEBEN contener la extensión "supported_versions". Los mensajes actuales de ServerHello contienen además la extensión "pre_shared_key" o la extensión "key_share", o ambas (cuando se usa un PSK con establecimiento de clave (EC)DHE). Otras extensiones (consulte la Sección 4.2) se envían por separado en el mensaje EncryptedExtensions.

Por motivos de compatibilidad con versiones anteriores de middleboxes (consulte el Apéndice D.4), el mensaje HelloRetryRequest utiliza la misma estructura que ServerHello, pero con Random establecido en el valor especial de SHA-256 de "HelloRetryRequest":

```
CF 21 AD 74 E5 9A 61 11 BE 1D 8C 02 1E 65 B8 91
C2 A2 11 16 7A BB 8C 5E 07 9E 09 E2 C8 A8 33 9C
```

Al recibir un mensaje con el tipo server_hello, las implementaciones DEBEN examinar primero el valor Random y, si coincide con este valor, procesarlo como se describe en la Sección 4.1.4).

TLS 1.3 tiene un mecanismo de protección de degradación integrado en el valor aleatorio del servidor. Los servidores TLS 1.3 que negocian TLS 1.2 o inferior en respuesta a un ClientHello DEBEN establecer los últimos 8 bytes de su valor Random especialmente en su ServerHello.

Si negocia TLS 1.2, los servidores TLS 1.3 DEBEN establecer los últimos 8 bytes de su valor aleatorio en los bytes:

```
44 4F 57 4E 47 52 44 01
```

Si negocia TLS 1.1 o inferior, los servidores TLS 1.3 DEBEN, y los servidores TLS 1.2 DEBEN, establecer los últimos 8 bytes de su valor ServerHello.Random en los bytes:

```
44 4F 57 4E 47 52 44 00
```

Los clientes TLS 1.3 que reciben un ServerHello que indica TLS 1.2 o inferior DEBEN verificar que los últimos 8 bytes no sean iguales a ninguno de estos valores. Los clientes TLS 1.2 también DEBERÍAN verificar que los últimos 8 bytes no sean iguales al segundo valor si ServerHello indica TLS 1.1 o inferior. Si se encuentra una coincidencia, el cliente DEBE cancelar el protocolo de enlace con una alerta de "parámetro_illegal". Este mecanismo proporciona una protección limitada contra los ataques de degradación por encima de lo que proporciona el intercambio

Finalizado: debido a que ServerKeyExchange, un mensaje presente en TLS 1.2 y versiones anteriores, incluye una firma sobre ambos valores aleatorios, no es posible que un atacante activo modifique los

Machine Translated by Google

valores aleatorios sin detección siempre que se utilicen cifrados efímeros. No proporciona protección de degradación cuando se utiliza RSA estático.

Nota: Este es un cambio de [RFC5246], por lo que, en la práctica, muchos clientes y servidores TLS 1.2 no se comportarán como se especificó anteriormente.

Un cliente TLS heredado que realiza una renegociación con TLS 1.2 o anterior y que recibe un TLS 1.3 ServerHello durante la renegociación DEBE cancelar el protocolo de enlace con una alerta "protocol_version". Tenga en cuenta que la renegociación no es posible cuando se ha negociado TLS 1.3.

4.1.4. Hola Solicitud de reintento

El servidor enviará este mensaje en respuesta a un mensaje ClientHello si puede encontrar un conjunto aceptable de parámetros pero ClientHello no contiene información suficiente para continuar con el protocolo de enlace. Como se discutió en la Sección 4.1.3, HelloRetryRequest tiene el mismo formato que un mensaje ServerHello, y los campos legacy_version, legacy_session_id_echo, cipher_suite y legacy_compression_method tienen el mismo significado. Sin embargo, por conveniencia, tratamos "HelloRetryRequest" a lo largo de este documento como si fuera una

mensaje.

Las extensiones del servidor DEBEN contener "supported_versions". Además, DEBERÍA contener el conjunto mínimo de extensiones necesarias para que el cliente genere un par ClientHello correcto. Al igual que con ServerHello, una HelloRetryRequest NO DEBE contener ninguna extensión que no haya sido ofrecida primero por el cliente en su ClientHello, con la excepción opcional de la extensión "cookie" (consulte la Sección 4.2.2).

Al recibir una HelloRetryRequest, el cliente DEBE verificar legacy_version, legacy_session_id_echo, cipher_suite y legacy_compression_method como se especifica en la Sección 4.1.3 y luego procesar las extensiones, comenzando con la determinación de la versión usando "supported_versions". Los clientes DEBEN cancelar el protocolo de enlace con una alerta de "parámetro ilegal" si HelloRetryRequest no genera ningún cambio en ClientHello. Si un cliente recibe un segundo HelloRetryRequest en la misma conexión (es decir, donde ClientHello estaba en respuesta a un HelloRetryRequest), DEBE abortar el protocolo de enlace con una alerta de "mensaje_inesperado".

Machine Translated by Google

De lo contrario, el cliente DEBE procesar todas las extensiones en HelloRetryRequest y enviar un segundo ClientHello actualizado. Las extensiones HelloRetryRequest definidas en esta especificación son:

- versiones_soportadas (ver Sección 4.2.1)

- cookie (ver Sección 4.2.2)

- key_share (ver Sección 4.2.8)

Un cliente que recibe un conjunto de cifrado que no se ofreció DEBE abortar el protocolo de enlace. Los servidores DEBEN asegurarse de que negocian el mismo conjunto de cifrado cuando reciben un ClientHello actualizado conforme (si el servidor selecciona el conjunto de cifrado como primer paso en la negociación, esto sucederá automáticamente). Al recibir ServerHello, los clientes DEBEN comprobar que el conjunto de cifrado suministrado en ServerHello sea el mismo que el de HelloRetryRequest y, de lo contrario, cancelar el protocolo de enlace con una alerta de "parámetro_illegal".

Además, en su ClientHello actualizado, el cliente NO DEBE ofrecer ninguna clave precompartida asociada con un hash que no sea el del conjunto de cifrado seleccionado. Esto permite que el cliente evite tener que calcular transcripciones de hash parciales para múltiples hash en el segundo ClientHello.

El valor de selected_version en la extensión HelloRetryRequest "supported_versions" DEBE conservarse en ServerHello, y un cliente DEBE cancelar el protocolo de enlace con una alerta de "illegal_parameter" si el valor cambia.

Machine Translated by Google

4.2. Extensiones

Varios mensajes TLS contienen extensiones codificadas de valor de longitud de etiqueta estructuras

```
estructura {
        ExtensionType extension_type;
        extension_data opaca<0..2^16-1>;
} Extensión;

enumeración {
        nombre_servidor(0),                                  /*RFC6066*/
        longitud_máx_fragmento(1),                           /*RFC6066*/
        estado_solicitud(5),                                 /*RFC6066*/
        grupos_soportados(10),                               /* RFC 8422, 7919 */
        algoritmos_de_firma(13),                             /* RFC 8446 */
        usar_srtp(14),                                       /* RFC 5764 */
        latido del corazón(15),                              /*RFC6520*/
        application_layer_protocol_negotiation(16), /* RFC 7301 */
        firmado_certificado_marca de tiempo (18),            /*RFC6962*/
        tipo_certificado_cliente(19),                        /*RFC7250*/
        tipo_certificado_servidor(20),                       /*RFC7250*/
        relleno(21),                                         /*RFC7685*/
        clave_pre_compartida(41),                            /* RFC 8446 */
        datos_tempranos(42),                                 /* RFC 8446 */
        versiones_soportadas(43),                            /* RFC 8446 */
        galleta (44),                                        /* RFC 8446 */
        psk_key_exchange_modes(45),                          /* RFC 8446 */
        autoridades_de_certificación(47),                    /* RFC 8446 */
        filtros_oid(48),                                     /* RFC 8446 */
        post_apretón de manos_autenticación(49),             /* RFC 8446 */
        algoritmos_de_firma_cert(50),                        /* RFC 8446 */
        clave_compartida(51),                                /* RFC 8446 */
        (65535)
} tipo de extensión;
```

Machine Translated by Google

Aquí:

- "extension_type" identifica el tipo de extensión particular.

- "extension_data" contiene información específica del tipo de extensión en particular.

La IANA mantiene la lista de tipos de extensión como se describe en la Sección 11.

Las extensiones generalmente se estructuran en forma de solicitud/respuesta, aunque algunas extensiones son solo indicaciones sin una respuesta correspondiente. El cliente envía sus solicitudes de extensión en el mensaje ClientHello y el servidor envía sus respuestas de extensión en los mensajes ServerHello, EncryptedExtensions, HelloRetryRequest y Certificate. El servidor envía solicitudes de extensión en el mensaje CertificateRequest al que un cliente PUEDE responder con un mensaje de Certificado. El servidor también PUEDE enviar extensiones no solicitadas en NewSessionTicket, aunque el cliente no responde directamente a ellas.

Las implementaciones NO DEBEN enviar respuestas de extensión si el extremo remoto no envió las solicitudes de extensión correspondientes, con la excepción de la extensión "cookie" en HelloRetryRequest. Al recibir dicha extensión, un punto final DEBE abortar el protocolo de enlace con una alerta de "extensión no admitida".

La siguiente tabla indica los mensajes en los que puede aparecer una determinada extensión, utilizando la siguiente notación: CH (ClientHello), SH (ServerHello), EE (EncryptedExtensions), CT (Certificate), CR (CertificateRequest), NST (NewSessionTicket) y HRR (HolaReintentarSolicitud). Si una implementación recibe una extensión que reconoce y que no está especificada para el mensaje en el que aparece, DEBE abortar el protocolo de enlace con una alerta de "parámetro_illegal".

Machine Translated by Google

Extensión	TLS 1.3
nombre_servidor [RFC6066]	CH, EE
max_fragment_length [RFC6066]	CH, EE
estado_solicitud [RFC6066]	CH, CR, CT
grupos_soportados [RFC7919]	CH, EE
algoritmos_de_firma (RFC 8446)	CR, CR
utilizar_srtp [RFC5764]	CH, EE
latido del corazón [RFC6520]	CH, EE
application_layer_protocol_negotiation [RFC7301]	CH, EE
firmado_certificado_marca de tiempo [RFC6962]	CH, CR, CT
tipo_certificado_cliente [RFC7250]	CH, EE
tipo_certificado_servidor [RFC7250]	CH, EE
relleno [RFC7685]	CH
clave_compartida (RFC 8446)	, CH, SH, HRR
pre_shared_key (RFC 8446)	CH, SH
psk_key_exchange_modos (RFC 8446)	CH
datos_anteriores (RFC 8446)	CH, EE, NST
galleta (RFC 8446)	CH, TRH
versiones_compatibles (RFC 8446)	, CH, SH, HRR
autoridades_de_certificación (RFC 8446)	CR, CR
† filtros_oid (RFC 8446)	CR
post_handshake_auth (RFC 8446)	CH
signature_algorithms_cert (RFC 8446)	CR, CR

Machine Translated by Google

Cuando hay varias extensiones de diferentes tipos, las extensiones PUEDEN aparecer en cualquier orden, con la excepción de "pre_shared_key" (Sección 4.2.11), que DEBE ser la última extensión en ClientHello (pero puede aparecer en cualquier parte del bloque de extensiones ServerHello) . NO DEBE haber más de una extensión del mismo tipo en un bloque de extensión determinado.

En TLS 1.3, a diferencia de TLS 1.2, las extensiones se negocian para cada protocolo de enlace, incluso cuando se está en modo PSK de reanudación. Sin embargo, los parámetros 0-RTT son los negociados en el protocolo de enlace anterior; las discrepancias pueden requerir el rechazo de 0-RTT (consulte la Sección 4.2.10).

Hay interacciones sutiles (y no tan sutiles) que pueden ocurrir en este protocolo entre las funciones nuevas y las funciones existentes que pueden resultar en una reducción significativa en la seguridad general. Las siguientes consideraciones deben tenerse en cuenta al diseñar nuevas extensiones:

- Algunos casos en los que un servidor no acepta una extensión son condiciones de error (p. ej., el protocolo de enlace no puede continuar), y algunos son simplemente rechazos para admitir características particulares. En general, se deben usar alertas de error para las primeras y un campo en la respuesta de la extensión del servidor para las segundas.

- Las extensiones deben, en la medida de lo posible, diseñarse para evitar cualquier ataque que fuerce el uso (o no uso) de una característica particular mediante la manipulación de mensajes de protocolo de enlace. Este principio debe seguirse independientemente de si se cree que la característica causa un problema de seguridad. A menudo, el hecho de que los campos de la extensión se incluyan en las entradas de los hash del mensaje Finalizado será suficiente, pero se necesita mucho cuidado cuando la extensión cambia el significado de los mensajes enviados en la fase de reconocimiento. Los diseñadores e implementadores deben ser conscientes del hecho de que hasta que se haya autenticado el protocolo de enlace, los atacantes activos pueden modificar mensajes e insertar, eliminar o reemplazar extensiones.

Machine Translated by Google

4.2.1. Versiones compatibles

```
struct { select
        (Handshake.msg_type) { case client_hello:
                Versiones de ProtocolVersion<2..254>;


                case server_hello: /* y HelloRetryRequest */
                        ProtocolVersion versión_seleccionada;
        };
} Versiones admitidas;
```

El cliente usa la extensión "supported_versions" para indicar qué versiones de TLS admite y el servidor para indicar qué versión está usando. La extensión contiene una lista de versiones admitidas en orden de preferencia, con la versión preferida primero.

Las implementaciones de esta especificación DEBEN enviar esta extensión en ClientHello que contiene todas las versiones de TLS que están preparadas para negociar (para esta especificación, eso significa como mínimo 0x0304, pero si se permite negociar versiones anteriores de TLS, DEBEN estar presentes también).).

Si esta extensión no está presente, los servidores que cumplen con esta especificación y que también son compatibles con TLS 1.2 DEBEN negociar TLS 1.2 o anterior como se especifica en [RFC5246], incluso si ClientHello.legacy_version es 0x0304 o posterior. Los servidores PUEDEN anular el protocolo de enlace al recibir un ClientHello con legacy_version 0x0304 o posterior.

Si esta extensión está presente en ClientHello, los servidores NO DEBEN usar el valor ClientHello.legacy_version para la negociación de la versión y DEBEN usar solo la extensión "supported_versions" para determinar las preferencias del cliente. Los servidores DEBEN seleccionar solo una versión de TLS presente en esa extensión y DEBEN ignorar cualquier versión desconocida que esté presente en esa extensión. Tenga en cuenta que este mecanismo permite negociar una versión anterior a TLS 1.2 si un lado admite un rango disperso. Las implementaciones de TLS 1.3 que opten por admitir versiones anteriores de TLS DEBERÍAN admitir TLS 1.2. Los servidores DEBEN estar preparados para recibir ClientHellos que incluyen esta extensión pero no incluyen 0x0304 en la lista de versiones.

Un servidor que negocia una versión de TLS anterior a TLS 1.3 DEBE configurar ServerHello.version y NO DEBE enviar la extensión "supported_versions". Un servidor que negocia TLS 1.3 DEBE responder enviando una extensión "supported_versions" que contiene el valor de la versión seleccionada (0x0304). DEBE establecer el campo ServerHello.legacy_version en 0x0303 (TLS 1.2). Los clientes DEBEN verificar esta extensión antes de procesar el resto de ServerHello (aunque tendrán que

Machine Translated by Google

analizar ServerHello para leer la extensión). Si esta extensión está presente, los clientes DEBEN ignorar el valor ServerHello.legacy_version y DEBEN usar solo la extensión "supported_versions" para determinar la versión seleccionada. Si la extensión "supported_versions" en ServerHello contiene una versión que no ofrece el cliente o contiene una versión anterior a TLS 1.3, el cliente DEBE cancelar el protocolo de enlace con una alerta de "illegal_parameter".

4.2.2. Galleta

```
estructura
        {cookie opaca<1..2^16-1>; } Galleta;
```

Las cookies tienen dos propósitos principales:

- Permitir que el servidor obligue al cliente a demostrar Accesibilidad en su dirección de red aparente (lo que proporciona una medida de protección DoS). Esto es principalmente útil para transportes no orientados a la conexión (consulte [RFC6347] para ver un ejemplo de esto).

- Permitir que el servidor descargue el estado al cliente, lo que le permite enviar un HelloRetryRequest sin almacenar ningún estado. El servidor puede hacer esto almacenando el hash de ClientHello en la cookie HelloRetryRequest (protegido con algún algoritmo de protección de integridad adecuado).

Al enviar una HelloRetryRequest, el servidor PUEDE proporcionar una extensión de "cookie" al cliente (esta es una excepción a la regla habitual de que las únicas extensiones que se pueden enviar son las que aparecen en ClientHello). Al enviar el nuevo ClientHello, el cliente DEBE copiar el contenido de la extensión recibida en HelloRetryRequest en una extensión de "cookie" en el nuevo ClientHello. Los clientes NO DEBEN utilizar cookies en su ClientHello inicial en conexiones posteriores.

Cuando un servidor funciona sin estado, puede recibir un registro desprotegido de tipo change_cipher_spec entre el primer y el segundo ClientHello (consulte la Sección 5). Dado que el servidor no está almacenando ningún estado, este aparecerá como si fuera el primer mensaje que se recibe. Los servidores que funcionan sin estado DEBEN ignorar estos registros.

Machine Translated by Google

4.2.3. Algoritmos de firma

TLS 1.3 proporciona dos extensiones para indicar qué algoritmos de firma se pueden usar en las firmas digitales. La extensión "signature_algorithms_cert" se aplica a las firmas en los certificados, y la extensión "signature_algorithms", que apareció originalmente en TLS 1.2, se aplica a las firmas en los mensajes de CertificateVerify. Las claves que se encuentran en los certificados también DEBEN ser del tipo apropiado para los algoritmos de firma con los que se utilizan. Este es un problema particular para las claves RSA y las firmas PSS, como se describe a continuación. Si no está presente la extensión "signature_algorithms_cert", entonces la extensión "signature_algorithms" también se aplica a las firmas que aparecen en los certificados. Los clientes que deseen que el servidor se autentique mediante un certificado DEBEN enviar la extensión "signature_algorithms". Si un servidor se está autenticando a través de un certificado y el cliente no ha enviado una extensión "signature_algorithms", entonces el servidor DEBE abortar el protocolo de enlace con una alerta de "missing_extension" (consulte la Sección 9.2).

Se agregó la extensión "signature_algorithms_cert" para permitir implementaciones que admiten diferentes conjuntos de algoritmos para certificados y en TLS mismo para señalar claramente sus capacidades.

Las implementaciones de TLS 1.2 también DEBERÍAN procesar esta extensión.
Las implementaciones que tienen la misma política en ambos casos PUEDEN omitir la extensión "signature_algorithms_cert".

Machine Translated by Google

El campo "extension_data" de estas extensiones contiene un
Valor de SignatureSchemeList:

```
enumeración {
        /* Algoritmos RSASSA-PKCS1-v1_5 */
        rsa_pkcs1_sha256(0x0401), rsa_pkcs1_sha384(0x0501),
        rsa_pkcs1_sha512(0x0601),


        /* Algoritmos ECDSA */
        ecdsa_secp256r1_sha256(0x0403),
        ecdsa_secp384r1_sha384(0x0503),
        ecdsa_secp521r1_sha512(0x0603),

        /* Algoritmos RSASSA-PSS con clave pública OID rsaEncryption */ rsa_pss_rsae_sha256(0x0804),
        rsa_pss_rsae_sha384(0x0805), rsa_pss_rsae_sha512(0x0806),



        /* Algoritmos EdDSA */
        ed25519(0x0807), ed448(0x0808),


        /* Algoritmos RSASSA-PSS con clave pública OID RSASSA-PSS */ rsa_pss_pss_sha256(0x0809),
        rsa_pss_pss_sha384(0x080a), rsa_pss_pss_sha512(0x080b),



        /* Algoritmos heredados */
        rsa_pkcs1_sha1(0x0201),
        ecdsa_sha1(0x0203),

        /* Puntos de código reservados */
        private_use(0xFE00..0xFFFF), (0xFFFF)

} Esquema de firma;

struct
        { SignatureScheme support_signature_algorithms<2..2^16-2>;
} Lista de esquemas de firmas;
```

Nota: Esta enumeración se denomina "SignatureScheme" porque ya existe un tipo "SignatureAlgorithm" en TLS 1.2, que reemplaza. Usamos el término "algoritmo de firma" en todo el texto.

Machine Translated by Google

Cada valor de SignatureScheme enumera un único algoritmo de firma que el cliente desea verificar. Los valores se indican en orden descendente de preferencia. Tenga en cuenta que un algoritmo de firma toma como entrada un mensaje de longitud arbitraria, en lugar de un resumen.

Los algoritmos que tradicionalmente actúan en un resumen deben definirse en TLS para primero aplicar un hash a la entrada con un algoritmo hash específico y luego proceder como de costumbre. Los grupos de puntos de código enumerados anteriormente tienen los siguientes significados:

Algoritmos RSASSA-PKCS1-v1_5: indica un algoritmo de firma que utiliza RSASSA-PKCS1-v1_5 [RFC8017] con el algoritmo hash correspondiente según se define en [SHS]. Estos valores se refieren únicamente a las firmas que aparecen en los certificados (consulte la Sección 4.4.2.2) y no están definidos para su uso en mensajes de protocolo de enlace TLS firmados, aunque PUEDEN aparecer en "signature_algorithms" y "signature_algorithms_cert" para compatibilidad con versiones anteriores de TLS 1.2.

Algoritmos ECDSA: indica un algoritmo de firma que utiliza ECDSA [ECDSA], la curva correspondiente como se define en ANSI X9.62 [ECDSA] y FIPS 186-4 [DSS], y el algoritmo hash correspondiente como se define en [SHS]. La firma se representa como una estructura ECDSA-Sig-Value codificada en DER [X690].

Algoritmos RSASSA-PSS RSAE: indica un algoritmo de firma que utiliza RSASSA-PSS [RFC8017] con función de generación de máscaras 1. El resumen utilizado en la función de generación de máscaras y el resumen que se firma son el algoritmo hash correspondiente, tal como se define en [SHS].
La longitud de Salt DEBE ser igual a la longitud de la salida del algoritmo de resumen. Si la clave pública se incluye en un certificado X.509, DEBE utilizar el OID de rsaEncryption [RFC5280].

Algoritmos EdDSA: Indica un algoritmo de firma que usa EdDSA como se define en [RFC8032] o sus sucesores. Tenga en cuenta que estos corresponden a los algoritmos "PureEdDSA" y no a las variantes "prehash".

RSASSA-PSS Algoritmos PSS: indica un algoritmo de firma que utiliza RSASSA-PSS [RFC8017] con función de generación de máscaras 1. El resumen utilizado en la función de generación de máscaras y el resumen que se firma son el algoritmo hash correspondiente, tal como se define en [SHS].
La longitud de Salt DEBE ser igual a la longitud del algoritmo de resumen. Si la clave pública se lleva en un certificado X.509, DEBE utilizar el RSASSA-PSS OID [RFC5756]. Cuando se utilizan en firmas de certificados, los parámetros del algoritmo DEBEN codificarse con DER. Si los parámetros de la clave pública correspondiente están presentes, los parámetros de la firma DEBEN ser idénticos a los de la clave pública.

Machine Translated by Google

Algoritmos heredados: indica algoritmos que están en desuso porque usan algoritmos con debilidades conocidas, específicamente SHA-1 que se usa en este contexto con (1) RSA usando RSASSA-PKCS1-v1_5 o (2) ECDSA. Estos valores se refieren únicamente a las firmas que aparecen en los certificados (consulte la Sección 4.4.2.2) y no están definidos para su uso en mensajes de protocolo de enlace TLS firmados, aunque PUEDEN aparecer en "signature_algorithms" y "signature_algorithms_cert" para compatibilidad con versiones anteriores de TLS 1.2. Los puntos finales NO DEBEN negociar estos algoritmos, pero se les permite hacerlo únicamente por compatibilidad con versiones anteriores. Los clientes que ofrecen estos valores DEBEN enumerarlos como la prioridad más baja (enumerados después de todos los demás algoritmos en SignatureSchemeList).

Los servidores TLS 1.3 NO DEBEN ofrecer un certificado firmado SHA-1 a menos que no se pueda producir una cadena de certificados válida sin él (consulte la Sección 4.4.2.2).

Las firmas de los certificados que son autofirmados o los certificados que son anclajes de confianza no se validan, ya que comienzan una ruta de certificación (consulte [RFC5280], Sección 3.2). Un certificado que comienza una ruta de certificación PUEDE usar un algoritmo de firma que no se anuncia como compatible en la extensión "signature_algorithms".

Tenga en cuenta que TLS 1.2 define esta extensión de manera diferente. Las implementaciones de TLS 1.3 que deseen negociar TLS 1.2 DEBEN comportarse de acuerdo con los requisitos de [RFC5246] al negociar esa versión. En particular:

- TLS 1.2 ClientHellos PUEDE omitir esta extensión.

- En TLS 1.2, la extensión contenía pares hash/firma. los los pares están codificados en dos octetos, por lo que los valores de SignatureScheme se han asignado para alinearse con la codificación de TLS 1.2. Algunos pares heredados se dejan sin asignar. Estos algoritmos están obsoletos a partir de TLS 1.3. NO DEBEN ser ofrecidos o negociados por ninguna implementación. En particular, NO DEBEN usarse MD5 [SLOTH], SHA-224 y DSA.

- Los esquemas de firma ECDSA se alinean con los pares de hash/firma ECDSA de TLS 1.2. Sin embargo, la semántica antigua no restringía la curva de firmas. Si se negocia TLS 1.2, las implementaciones DEBEN estar preparadas para aceptar una firma que use cualquier curva que anunciaron en la extensión "supported_groups".

- Implementaciones que anuncian soporte para RSASSA-PSS (que es obligatorio en TLS 1.3) DEBE estar preparado para aceptar una firma usando ese esquema incluso cuando se negocie TLS 1.2. En TLS 1.2, RSASSA-PSS se usa con conjuntos de cifrado RSA.

Machine Translated by Google

4.2.4. Autoridades de certificación

La extensión "certificate_authorities" se usa para indicar las autoridades de certificación (CA) que admite un punto final y que DEBERÍA utilizar el punto final receptor para guiar la selección del certificado.

El cuerpo de la extensión "certificate_authorities" consta de una estructura CertificateAuthoritiesExtension.

```
nombre Distinguido opaco<1..2^16-1>;

struct
        { Nombre Distinguido autoridades<3..2^16-1>; } Extensión de las
 autoridades del certificado;
```

autoridades: una lista de nombres distinguidos [X501] de autoridades de certificación aceptables, representadas en formato codificado DER [X690].
Estos nombres distinguidos especifican un nombre distinguido deseado para un ancla de confianza o CA subordinada; por lo tanto, este mensaje se puede usar para describir anclas de confianza conocidas, así como una autorización deseada
espacio.

El cliente PUEDE enviar la extensión "certificate_authorities" en el mensaje ClientHello. El servidor PUEDE enviarlo en el mensaje CertificateRequest.

La extensión "trusted_ca_keys" [RFC6066], que tiene un propósito similar pero es más complicada, no se usa en TLS 1.3 (aunque puede aparecer en los mensajes ClientHello de clientes que ofrecen versiones anteriores de TLS).

4.2.5. Filtros OID

La extensión "oid_filters" permite a los servidores proporcionar un conjunto de pares OID/valor con los que desea que coincida el certificado del cliente. Esta extensión, si la proporciona el servidor, solo DEBE enviarse en el mensaje CertificateRequest.

```
struct
        { certificado opaco_extensión_oid<1..2^8-1>; valores_extensión_certificado
         opaco<0..2^16-1>; } FiltroOID;


estructura
         { filtros OIDFilter<0..2^16-1>; } OIDFilterExtension;
```

Machine Translated by Google

filtros: una lista de OID de extensión de certificado [RFC5280] con sus valores permitidos y representados en formato codificado en DER [X690].
Algunos OID de extensión de certificado permiten varios valores (p. ej., uso extendido de clave). Si el servidor ha incluido una lista de filtros no vacíos, el certificado del cliente incluido en la respuesta DEBE contener todos los OID de extensión especificados que el cliente reconoce. Para cada OID de extensión reconocido por el cliente, todos los valores especificados DEBEN estar presentes en el certificado del cliente (pero el certificado PUEDE tener también otros valores). Sin embargo, el cliente DEBE ignorar y omitir cualquier OID de extensión de certificado no reconocido. Si el cliente ignoró algunos de los OID de extensión de certificado requeridos y proporcionó un certificado que no satisface la solicitud, el servidor PUEDE, a su discreción, continuar la conexión sin la autenticación del cliente o cancelar el protocolo de enlace con una alerta de "certificado no admitido". Cualquier OID dado NO DEBE aparecer más de una vez en la lista de filtros.

Los RFC de PKIX definen una variedad de OID de extensión de certificado y sus tipos de valores correspondientes. Según el tipo, los valores de extensión de certificado coincidentes no son necesariamente iguales a nivel de bits. Se espera que las implementaciones de TLS se basen en sus bibliotecas PKI para realizar la selección de certificados utilizando OID de extensión de certificado.

Este documento define reglas coincidentes para dos extensiones de certificado estándar definidas en [RFC5280]:

- La extensión de uso de clave en un certificado coincide con la solicitud cuando todos los bits de uso de clave declarados en la solicitud también se afirman en la extensión de certificado de uso de clave.

- La extensión Extended Key Usage en un certificado coincide con la solicitud cuando todos los OID de propósito clave presentes en la solicitud también se encuentran en la extensión del certificado de uso extendido de clave. El OID especial anyExtendedKeyUsage NO DEBE utilizarse en la solicitud.

Las especificaciones separadas pueden definir reglas coincidentes para otras extensiones de certificado.

Machine Translated by Google

4.2.6. Autenticación de cliente posterior al protocolo de enlace

La extensión "post_handshake_auth" se usa para indicar que un cliente está dispuesto a realizar la autenticación posterior al protocolo de enlace (Sección 4.6.2).
Los servidores NO DEBEN enviar un CertificateRequest posterior al protocolo de enlace a los clientes que no ofrecen esta extensión. Los servidores NO DEBEN enviar esta extensión.

```
estructura {} PostHandshakeAuth;
```

El campo "extension_data" de la extensión "post_handshake_auth" tiene longitud cero.

4.2.7. Grupos admitidos

Cuando la envía el cliente, la extensión "supported_groups" indica los grupos con nombre que el cliente admite para el intercambio de claves, ordenados de mayor a menor preferencia.

Nota: En las versiones de TLS anteriores a TLS 1.3, esta extensión se denominaba "elliptic_curves" y solo contenía grupos de curvas elípticas. Ver [RFC8422] y [RFC7919]. Esta extensión también se utilizó para negociar curvas ECDSA. Los algoritmos de firma ahora se negocian de forma independiente (consulte la Sección 4.2.3).

El campo "extension_data" de esta extensión contiene un
Valor de "Lista de grupos con nombre":

```
enumeración {

        /* Grupos de curvas elípticas (ECDHE) */ secp256r1(0x0017),
        secp384r1(0x0018), secp521r1(0x0019), x25519(0x001D), x448(0x001E),


        /* Grupos de campos finitos (DHE) */
        ffdhe2048(0x0100), ffdhe3072(0x0101), ffdhe4096(0x0102), ffdhe6144(0x0103),
        ffdhe8192(0x0104),

        /* Puntos de código reservados */
        ffdhe_private_use(0x01FC..0x01FF),
        ecdhe_private_use(0xFE00..0xFEFF), (0xFFFF)

} grupo con nombre;

struct
        { NamedGroup lista_de_grupos_nombrados<2..2^16-1>; } Lista
de grupos con nombre;
```

Machine Translated by Google

Grupos de curvas elípticas (ECDHE): indica soporte para el curva nombrada correspondiente, definida en FIPS 186-4 [DSS] o [RFC7748]. Los valores 0xFE00 a 0xFEFF están reservados para uso privado [RFC8126].

Grupos de campos finitos (DHE): indica soporte para el grupo de campos finitos correspondiente, definido en [RFC7919]. Los valores 0x01FC a 0x01FF están reservados para uso privado.

Los elementos en named_group_list se ordenan de acuerdo con las preferencias del remitente (la opción más preferida primero).

A partir de TLS 1.3, los servidores pueden enviar la extensión "supported_groups" al cliente. Los clientes NO DEBEN actuar sobre la información que se encuentra en "supported_groups" antes de completar con éxito el protocolo de enlace, pero PUEDEN usar la información aprendida de un protocolo de enlace completado con éxito para cambiar qué grupos usan en su extensión "key_share" en conexiones posteriores. Si el servidor tiene un grupo que prefiere a los de la extensión "key_share" pero aún está dispuesto a aceptar ClientHello, DEBERÍA enviar "supported_groups" para actualizar la vista del cliente de sus preferencias; esta extensión DEBERÍA contener todos los grupos que admite el servidor, independientemente de si el cliente los admite actualmente.

4.2.8. Clave compartida

La extensión "key_share" contiene los parámetros criptográficos del punto final.

Los clientes PUEDEN enviar un vector client_shares vacío para solicitar la selección de grupo del servidor, al costo de un viaje de ida y vuelta adicional (consulte la Sección 4.1.4).

```
struct { grupo
        NamedGroup; intercambio
        de clave opaca<1..2^16-1>; } KeyShareEntry;
```

grupo: el grupo con nombre para la clave que se intercambia.

key_exchange: Información de intercambio de claves. El contenido de este campo está determinado por el grupo especificado y su definición correspondiente. Los parámetros de campo finito Diffie-Hellman [DH76] se describen en la Sección 4.2.8.1; Los parámetros Diffie-Hellman de la curva elíptica se describen en la Sección 4.2.8.2.

Machine Translated by Google

En el mensaje ClientHello, el campo "extension_data" de esta extensión contiene un valor "KeyShareClientHello":

```
estructura
        { KeyShareEntry client_shares<0..2^16-1>;
} KeyShareClientHola;
```

client_shares: una lista de valores KeyShareEntry ofrecidos en orden descendente de preferencia del cliente.

Este vector PUEDE estar vacío si el cliente solicita una HelloRetryRequest. Cada valor de KeyShareEntry DEBE corresponder a un grupo ofrecido en la extensión "supported_groups" y DEBE aparecer en el mismo orden. Sin embargo, los valores PUEDEN ser un subconjunto no contiguo de la extensión "supported_groups" y PUEDEN omitir los grupos más preferidos. Tal situación podría surgir si los grupos más preferidos son nuevos y es poco probable que reciban apoyo en suficientes lugares para que la generación previa de recursos compartidos clave para ellos sea eficiente.

Los clientes pueden ofrecer tantos valores de KeyShareEntry como la cantidad de grupos admitidos que ofrece, cada uno de los cuales representa un conjunto único de parámetros de intercambio de claves. Por ejemplo, un cliente puede ofrecer acciones para varias curvas elípticas o múltiples grupos FFDHE. Los valores key_exchange para cada KeyShareEntry DEBEN generarse de forma independiente. Los clientes NO DEBEN ofrecer múltiples valores KeyShareEntry para el mismo grupo. Los clientes NO DEBEN ofrecer valores KeyShareEntry para grupos que no figuran en la extensión "supported_groups" del cliente.

Los servidores PUEDEN verificar si hay violaciones de estas reglas y anular el protocolo de enlace con una alerta de "parámetro_illegal" si se viola alguna.

En un mensaje HelloRetryRequest, el campo "extension_data" de esta extensión contiene un valor KeyShareHelloRetryRequest:

```
estructura {
        grupo_nombrado grupo_seleccionado;
} KeyShareHelloRetryRequest;
```

grupo_seleccionado: el grupo con apoyo mutuo que el servidor pretende negociar y para el que solicita un ClientHello/KeyShare reintentado.

Al recibir esta extensión en una HelloRetryRequest, el cliente DEBE verificar que (1) el campo selected_group corresponde a un grupo que se proporcionó en la extensión "supported_groups" en el ClientHello original y (2) el campo selected_group no corresponde a un grupo que se proporcionó en la extensión "key_share" en el ClientHello original. Si cualquiera de estas comprobaciones falla, entonces el cliente DEBE abortar el protocolo de enlace con una alerta de "parámetro_illegal". De lo contrario, al enviar el nuevo ClientHello, el cliente DEBE

Machine Translated by Google

reemplace la extensión "key_share" original con una que contenga solo una KeyShareEntry nueva para el grupo indicado en el campo selected_group de la activación HelloRetryRequest.

En un mensaje de ServerHello, el campo "extension_data" de esta extensión contiene un valor de KeyShareServerHello:

```
estructura {
        KeyShareEntry server_share;
} KeyShareServerHola;
```

server_share: un único valor KeyShareEntry que está en el mismo grupo que uno de los recursos compartidos del cliente.

Si usa el establecimiento de claves (EC)DHE, los servidores ofrecen exactamente una KeyShareEntry en ServerHello. Este valor DEBE estar en el mismo grupo que el valor KeyShareEntry ofrecido por el cliente que el servidor ha seleccionado para el intercambio de claves negociado. Los servidores NO DEBEN enviar una KeyShareEntry para ningún grupo que no esté indicado en la extensión "supported_groups" del cliente y NO DEBEN enviar una KeyShareEntry cuando utilicen el PskKeyExchangeMode "psk_ke". Si se utiliza el establecimiento de clave (EC)DHE y el cliente recibió una HelloRetryRequest que contiene una extensión "key_share", el cliente DEBE verificar que el NamedGroup seleccionado en ServerHello sea el mismo que el de HelloRetryRequest. Si esta verificación falla, el cliente DEBE abortar el protocolo de enlace con una alerta de "parámetro_illegal".

4.2.8.1. Parámetros de Diffie-Hellman

Los parámetros Diffie-Hellman [DH76] para clientes y servidores están codificados en el campo key_exchange opaco de KeyShareEntry en una estructura KeyShare. El valor opaco contiene el valor público de Diffie-Hellman (Y = g^X mod p) para el grupo especificado (ver [RFC7919] para definiciones de grupos) codificado como un entero big-endian y rellenado a la izquierda con ceros del tamaño de p en bytes.

Nota: Para un grupo Diffie-Hellman dado, el relleno da como resultado que todas las claves públicas tengan la misma longitud.

Los pares DEBEN validar la clave pública Y de los demás asegurándose de que 1 < Y < p-1. Esta comprobación garantiza que el par remoto se comporte correctamente y no fuerce al sistema local a formar parte de un pequeño subgrupo.

Machine Translated by Google

4.2.8.2. Parámetros de ECDHE

Los parámetros ECDHE para clientes y servidores están codificados en el campo key_exchange opaco de KeyShareEntry en una estructura KeyShare.

Para secp256r1, secp384r1 y secp521r1, el contenido es el valor serializado de la siguiente estructura:

```
estructura
        { uint8 forma_heredada = 4; X
        opaco[longitud_coordenada]; opaco
        Y[longitud_coordenada];
} Representación de puntos sin comprimir;
```

X e Y, respectivamente, son las representaciones binarias de los valores de x e y en orden de bytes de red. No hay marcadores de longitud internos, por lo que cada representación numérica ocupa tantos octetos como implican los parámetros de la curva. Para P-256, esto significa que cada uno de X e Y usa 32 octetos, rellenados con ceros a la izquierda si es necesario. Para P-384, toman 48 octetos cada uno. Para P-521, toman 66 octetos cada uno.

Para las curvas secp256r1, secp384r1 y secp521r1, los pares DEBEN validar el valor público Q de cada uno asegurándose de que el punto sea un punto válido en la curva elíptica. Los procedimientos de validación apropiados se definen en la Sección 4.3.7 de [ECDSA] y, alternativamente, en la Sección 5.6.2.3 de [ACUERDO CLAVE]. Este proceso consta de tres pasos: (1) verificar que Q no es el punto en el infinito (O), (2) verificar que para Q = (x, y) ambos enteros x e y están en el intervalo correcto, y (3) asegúrese de que (x, y) sea una solución correcta de la ecuación de la curva elíptica. Para estas curvas, los implementadores no necesitan verificar la pertenencia al subgrupo correcto.

Para X25519 y X448, el contenido del valor público son las entradas y salidas de cadenas de bytes de las funciones correspondientes definidas en [RFC7748]: 32 bytes para X25519 y 56 bytes para X448.

Nota: Las versiones de TLS anteriores a la 1.3 permitían la negociación del formato de puntos; TLS 1.3 elimina esta característica a favor de un formato de punto único para cada curva.

4.2.9. Modos de intercambio de claves previamente compartidas

Para usar PSK, los clientes también DEBEN enviar una extensión "psk_key_exchange_modes". La semántica de esta extensión es que el cliente solo admite el uso de PSK con estos modos, lo que restringe tanto el uso de PSK ofrecidos en este ClientHello como los que el servidor podría proporcionar a través de NewSessionTicket.

Machine Translated by Google

Un cliente DEBE proporcionar una extensión "psk_key_exchange_modes" si ofrece una extensión "pre_shared_key". Si los clientes ofrecen "pre_shared_key" sin una extensión "psk_key_exchange_modes", los servidores DEBEN cancelar el protocolo de enlace. Los servidores NO DEBEN seleccionar un modo de intercambio de claves que no esté en la lista del cliente. Esta extensión también restringe los modos de uso con la reanudación de PSK. Los servidores NO DEBEN enviar NewSessionTicket con tickets que no sean compatibles con los modos anunciados; sin embargo, si un servidor lo hace, el impacto será que los intentos de reanudación del cliente fallarán.

El servidor NO DEBE enviar una extensión "psk_key_exchange_modes".

```
enum {psk_ke (0), psk_dhe_ke (1), (255)} PskKeyExchangeMode;

estructura
        { PskKeyExchangeMode ke_modes<1..255>; }
 PskKeyExchangeModes;
```

psk_ke: establecimiento de clave solo de PSK. En este modo, el servidor NO DEBE proporcionar un valor "key_share".

psk_dhe_ke: PSK con establecimiento de clave (EC)DHE. En este modo, el cliente y el servidor DEBEN proporcionar valores de "key_share" como se describe en la Sección 4.2.8.

Cualquier valor futuro que se asigne debe garantizar que los mensajes de protocolo transmitidos identifiquen sin ambigüedades qué modo seleccionó el servidor; en la actualidad, esto se indica mediante la presencia de "key_share" en ServerHello.

4.2.10. Indicación de datos tempranos

Cuando se usa un PSK y se permiten datos iniciales para ese PSK, el cliente puede enviar datos de aplicación en su primer tramo de mensajes. Si el cliente opta por hacerlo, DEBE proporcionar las extensiones "pre_shared_key" y "early_data".

El campo "extension_data" de esta extensión contiene un valor "EarlyDataIndication".

Machine Translated by Google


```
estructura {} Vacío;

estructura
        { seleccionar (Apretón de manos.msg_type) {
                case new_session_ticket: uint32 max_early_data_size; caso cliente_hola:
                                                                Vacío;
                case encrypted_extensions: Vacío;
        };
} Indicación de datos tempranos;
```

Consulte la Sección 4.6.1 para obtener detalles sobre el uso del campo max_early_data_size.

Los parámetros para los datos 0-RTT (versión, conjunto de cifrado simétrico, protocolo de negociación de protocolo de capa de aplicación (ALPN) [RFC7301], etc.) son los asociados con el PSK en uso. Para los PSK aprovisionados externamente, los valores asociados son los aprovisionados junto con la clave. Para los PSK establecidos a través de un mensaje NewSessionTicket, los valores asociados son los que se negociaron en la conexión que estableció el PSK. El PSK utilizado para cifrar los primeros datos DEBE ser el primer PSK enumerado en la extensión "pre_shared_key" del cliente.

Para los PSK provistos a través de NewSessionTicket, un servidor DEBE validar que la antigüedad del boleto para la identidad de PSK seleccionada (calculada al restar ticket_age_add de PskIdentity.obfuscated_ticket_age módulo 2^32) está dentro de una pequeña tolerancia del tiempo desde que se emitió el boleto (consulte la Sección 8). Si no es así, el servidor DEBERÍA continuar con el protocolo de enlace pero rechazar 0-RTT, y NO DEBERÍA tomar ninguna otra acción que suponga que este ClientHello es nuevo.

Los mensajes 0-RTT enviados en el primer tramo tienen los mismos tipos de contenido (encriptados) que los mensajes del mismo tipo enviados en otros tramos (intercambio de señales y datos de aplicación), pero están protegidos con claves diferentes. Después de recibir el mensaje Finalizado del servidor, si el servidor ha aceptado datos anteriores, se enviará un mensaje EndOfEarlyData para indicar el cambio de clave. Este mensaje se cifrará con las claves de tráfico 0-RTT.

Machine Translated by Google

Un servidor que recibe una extensión "early_data" DEBE comportarse de una de estas tres maneras:

- Ignora la extensión y devuelve una respuesta regular de 1-RTT. Luego, el servidor salta los datos anteriores al intentar desproteger los registros recibidos mediante la clave de tráfico de protocolo de enlace, descartando los registros que fallan en la desprotección (hasta el max_early_data_size configurado). Una vez que un registro se desprotege con éxito, se trata como el comienzo del segundo vuelo del cliente y el servidor procede como con un protocolo de enlace 1-RTT normal.

- Solicitar que el cliente envíe otro ClientHello respondiendo con un HelloRetryRequest. Un cliente NO DEBE incluir la extensión "early_data" en su ClientHello de seguimiento. Luego, el servidor ignora los datos iniciales al omitir todos los registros con un tipo de contenido externo de "datos_de_aplicación" (lo que indica que están encriptados), hasta el tamaño máximo_de_datos_tempranos configurado.

- Devuelve su propia extensión "early_data" en EncryptedExtensions, indicando que tiene la intención de procesar los primeros datos. No es posible que el servidor acepte solo un subconjunto de los primeros mensajes de datos. Aunque el servidor envía un mensaje aceptando los primeros datos, es posible que los primeros datos en sí ya estén en tránsito cuando el servidor genera este mensaje.

Para aceptar los primeros datos, el servidor DEBE haber aceptado un conjunto de cifrado PSK y haber seleccionado la primera clave ofrecida en la extensión "pre_shared_key" del cliente. Además, DEBE verificar que los siguientes valores sean los mismos que los asociados con el PSK seleccionado:

- El número de versión de TLS

- El conjunto de cifrado seleccionado

- El protocolo ALPN [RFC7301] seleccionado, si lo hay

Estos requisitos son un superconjunto de los necesarios para realizar un protocolo de enlace 1-RTT utilizando el PSK en cuestión. Para los PSK establecidos externamente, los valores asociados son los proporcionados junto con la clave.
Para los PSK establecidos a través de un mensaje NewSessionTicket, los valores asociados son los negociados en la conexión durante la cual se estableció el ticket.

Las extensiones futuras DEBEN definir su interacción con 0-RTT.

Machine Translated by Google

Si alguna de estas comprobaciones falla, el servidor NO DEBE responder con la extensión y debe descartar todos los datos del primer vuelo utilizando uno de los dos primeros mecanismos enumerados anteriormente (recurriendo así a 1-RTT o 2-RTT). Si el cliente intenta un protocolo de enlace 0-RTT pero el servidor lo rechaza, el servidor generalmente no tendrá las claves de protección de registro 0-RTT y en su lugar debe usar el descifrado de prueba (ya sea con las claves de protocolo de enlace 1-RTT o buscando un ClientHello de texto claro en el caso de HelloRetryRequest) para encontrar el primer mensaje que no sea 0-RTT.

Si el servidor elige aceptar la extensión "early_data", DEBE cumplir con los mismos requisitos de manejo de errores especificados para todos los registros al procesar los primeros registros de datos. Específicamente, si el servidor no puede descifrar un registro 0-RTT luego de una extensión "early_data" aceptada, DEBE terminar la conexión con una alerta "bad_record_mac" según la Sección 5.2.

Si el servidor rechaza la extensión "early_data", la aplicación del cliente PUEDE optar por retransmitir los Datos de la aplicación enviados previamente en los primeros datos una vez que se haya completado el protocolo de enlace. Tenga en cuenta que la retransmisión automática de datos anteriores podría generar suposiciones incorrectas sobre el estado de la conexión. Por ejemplo, cuando la conexión negociada selecciona un protocolo ALPN diferente del que se usó para los primeros datos, es posible que una aplicación necesite construir mensajes diferentes. De manera similar, si los primeros datos asumen algo sobre el estado de la conexión, es posible que se envíen por error después de que se complete el protocolo de enlace.

Una implementación de TLS NO DEBE reenviar automáticamente los primeros datos; las aplicaciones están en una mejor posición para decidir cuándo es apropiada la retransmisión. Una implementación de TLS NO DEBE reenviar automáticamente los primeros datos a menos que la conexión negociada seleccione el mismo protocolo ALPN.

4.2.11. Extensión de clave precompartida

La extensión "pre_shared_key" se usa para negociar la identidad de la clave precompartida que se usará con un protocolo de enlace dado en asociación con el establecimiento de la clave PSK.

Machine Translated by Google

El campo "extension_data" de esta extensión contiene un
Valor de "PreSharedKeyExtension":

```
struct
        { identidad opaca<1..2^16-1>; uint32
         ofuscado_ticket_age; } PskIdentidad;


opaco PskBinderEntry<32..255>;

estructura
         { PskIdentity identidades<7..2^16-1>; Carpetas
         PskBinderEntry<33..2^16-1>; } Paquetes Ofrecidos;


estructura
         { seleccionar (Apretón de manos.msg_type) {
                 caso client_hello: OfferedPsks; case server_hello:
                 uint16 selected_identity;
         };
} PreSharedKeyExtension;
```

identidad: Una etiqueta para una clave. Por ejemplo, un ticket (como se define en el Apéndice B.3.4) o una etiqueta para una clave precompartida establecida externamente.

obfuscated_ticket_age: una versión ofuscada de la edad de la clave. La Sección 4.2.11.1 describe cómo formar este valor para las identidades establecidas a través del mensaje NewSessionTicket. Para las identidades establecidas externamente, se DEBERÍA utilizar un ofuscated_ticket_age de 0, y los servidores DEBEN ignorar el valor.

identidades: Una lista de las identidades que el cliente está dispuesto a negociar con el servidor. Si se envía junto con la extensión "early_data" (consulte la Sección 4.2.10), la primera identidad es la utilizada para los datos 0-RTT.

aglutinantes: una serie de valores HMAC, uno para cada valor en el lista de identidades y en el mismo orden, calculado como se describe a continuación.

Identidad_seleccionada: la identidad elegida por el servidor expresada como un índice (basado en 0) en las identidades de la lista del cliente.

Cada PSK está asociado con un solo algoritmo Hash. Para los PSK establecidos a través del mecanismo de ticket (Sección 4.6.1), este es el algoritmo KDF Hash en la conexión donde se estableció el ticket.

Para PSK establecidos externamente, el algoritmo Hash DEBE establecerse cuando

Machine Translated by Google

el PSK se establece o se establece de forma predeterminada en SHA-256 si no se define dicho algoritmo. El servidor DEBE asegurarse de seleccionar un PSK compatible (si lo hay) y un conjunto de cifrado.

En las versiones de TLS anteriores a TLS 1.3, el valor de Identificación del nombre del servidor (SNI) estaba destinado a estar asociado con la sesión (Sección 3 de [RFC6066]), y el servidor debe exigir que el valor SNI asociado con la sesión coincida con el uno especificado en el apretón de manos de reanudación. Sin embargo, en realidad las implementaciones no eran consistentes en cuál de los dos valores de SNI proporcionados usarían, lo que llevó a que los clientes aplicaran de facto el requisito de coherencia. En TLS 1.3, el valor de SNI siempre se especifica explícitamente en el protocolo de enlace de reanudación y no es necesario que el servidor asocie un valor de SNI con el ticket. Sin embargo, los clientes DEBERÍAN almacenar el SNI con el PSK para cumplir con los requisitos de la Sección 4.6.1.

Nota del implementador: cuando la reanudación de la sesión es el caso de uso principal de PSK, la forma más sencilla de implementar los requisitos de coincidencia de PSK/conjunto de cifrado es negociar primero el conjunto de cifrado y luego excluir cualquier PSK incompatible. Cualquier PSK desconocido (p. ej., los que no están en la base de datos de PSK o están cifrados con una clave desconocida) DEBEN simplemente ignorarse. Si no se encuentran PSK aceptables, el servidor DEBERÍA realizar un protocolo de enlace que no sea PSK si es posible. Si la compatibilidad con versiones anteriores es importante, los PSK proporcionados por el cliente y establecidos externamente DEBERÍAN influir en la selección del conjunto de cifrado.

Antes de aceptar el establecimiento de la clave PSK, el servidor DEBE validar el valor de enlace correspondiente (consulte la Sección 4.2.11.2 a continuación). Si este valor no está presente o no se valida, el servidor DEBE abortar el protocolo de enlace. Los servidores NO DEBEN intentar validar múltiples carpetas; más bien, DEBERÍAN seleccionar un solo PSK y validar únicamente el aglutinante que corresponde a ese PSK. Consulte la Sección 8.2 y el Apéndice E.6 para conocer la justificación de seguridad de este requisito. Para aceptar el establecimiento de la clave PSK, el servidor envía una extensión "pre_shared_key" que indica la identidad seleccionada.

Los clientes DEBEN verificar que la identidad_seleccionada del servidor se encuentra dentro del rango proporcionado por el cliente, que el servidor seleccionó un conjunto de cifrado que indica un Hash asociado con el PSK y que una extensión "key_share" del servidor está presente si la extensión ClientHello "psk_key_exchange_modes" lo requiere . Si estos valores no son consistentes, el cliente DEBE abortar el protocolo de enlace con una alerta de "parámetro_illegal".

Machine Translated by Google

Si el servidor proporciona una extensión "early_data", el cliente DEBE verificar que la identidad_seleccionada del servidor sea 0. Si se devuelve cualquier otro valor, el cliente DEBE cancelar el protocolo de enlace con una alerta de "parámetro_illegal".

La extensión "pre_shared_key" DEBE ser la última extensión en ClientHello (esto facilita la implementación como se describe a continuación).
Los servidores DEBEN verificar que sea la última extensión y, de lo contrario, fallar el protocolo de enlace con una alerta de "parámetro_illegal".

4.2.11.1. Edad del billete

La vista del cliente de la antigüedad de un ticket es el tiempo transcurrido desde la recepción del mensaje NewSessionTicket. Los clientes NO DEBEN intentar usar boletos que tengan una antigüedad superior al valor "ticket_lifetime" que se proporcionó con el boleto. El campo "obfuscated_ticket_age" de cada PskIdentity contiene una versión ofuscada de la antigüedad del ticket formada al tomar la antigüedad en milisegundos y agregar el valor "ticket_age_add" que se incluyó con el ticket (consulte la Sección 4.6.1), módulo 2^32. Esta adición evita que los observadores pasivos correlacionen las conexiones a menos que se reutilicen los boletos.

Tenga en cuenta que el campo "ticket_lifetime" en el mensaje NewSessionTicket está en segundos, pero "ofuscated_ticket_age" está en milisegundos.
Debido a que la vida útil de los boletos está restringida a una semana, 32 bits son suficientes para representar cualquier edad plausible, incluso en milisegundos.

4.2.11.2. Carpeta PSK

El valor del enlace de PSK forma un vínculo entre un PSK y el protocolo de enlace actual, así como un vínculo entre el protocolo de enlace en el que se generó el PSK (si es a través de un mensaje NewSessionTicket) y el protocolo de enlace actual. Cada entrada en la lista de carpetas se calcula como un HMAC sobre un hash de transcripción (consulte la Sección 4.4.1) que contiene un ClientHello parcial hasta el campo PreSharedKeyExtension.identities incluido. Es decir, incluye todo ClientHello pero no la lista de carpetas en sí. Los campos de longitud del mensaje (incluida la longitud total, la longitud del bloque de extensiones y la longitud de la extensión "pre_shared_key") se configuran como si estuvieran presentes carpetas de las longitudes correctas.

PskBinderEntry se calcula de la misma manera que el mensaje Finalizado (Sección 4.4.4), pero BaseKey es el binder_key derivado a través del programa de claves del PSK correspondiente que se ofrece (consulte la Sección 7.1).

Machine Translated by Google

Si el protocolo de enlace incluye HelloRetryRequest, ClientHello y HelloRetryRequest iniciales se incluyen en la transcripción junto con el nuevo ClientHello. Por ejemplo, si el cliente envía ClientHello1, su carpeta se calculará sobre:

```
Transcripción-Hash(Truncar(ClientHello1))
```

Donde Truncate() elimina la lista de carpetas de ClientHello.

Si el servidor responde con HelloRetryRequest y el cliente envía ClientHello2, su carpeta se calculará sobre:

```
Transcripción-Hash(ClientHello1,
                   HolaReintentarSolicitud,
                   Truncar (ClienteHola2))
```

El ClientHello1/ClientHello2 completo se incluye en todos los demás cálculos de hash de protocolo de enlace. Tenga en cuenta que en el primer tramo, Truncate(ClientHello1) se aplica hash directamente, pero en el segundo tramo, ClientHello1 se aplica hash y luego se reinyecta como un mensaje "message_hash", como se describe en la Sección 4.4.1.

4.2.11.3. Orden de procesamiento

Los clientes pueden "transmitir" datos 0-RTT hasta que reciben el mensaje Finalizado del servidor, solo luego envían el mensaje EndOfEarlyData, seguido del resto del protocolo de enlace. Para evitar interbloqueos, al aceptar "early_data", los servidores DEBEN procesar el ClientHello del cliente y luego enviar inmediatamente su tramo de mensajes, en lugar de esperar el mensaje EndOfEarlyData del cliente antes de enviar su ServerHello.

4.3. Parámetros del servidor

Los siguientes dos mensajes del servidor, EncryptedExtensions y CertificateRequest, contienen información del servidor que determina el resto del protocolo de enlace. Estos mensajes se cifran con claves derivadas de server_handshake_traffic_secret.

Machine Translated by Google

4.3.1. Extensiones encriptadas

En todos los protocolos de enlace, el servidor DEBE enviar el mensaje EncryptedExtensions inmediatamente después del mensaje ServerHello. Este es el primer mensaje que se cifra con claves derivadas de server_handshake_traffic_secret.

El mensaje EncryptedExtensions contiene extensiones que pueden protegerse, es decir, cualquiera que no sea necesaria para establecer el contexto criptográfico pero que no esté asociada con certificados individuales. El cliente DEBE verificar EncryptedExtensions para detectar la presencia de extensiones prohibidas y, si encuentra alguna, DEBE abortar el protocolo de enlace con una alerta de "parámetro_illegal".

Estructura de este mensaje:

```
estructura
        { Extensión extensiones<0..2^16-1>;
} Extensiones cifradas;
```

extensiones: Una lista de extensiones. Para obtener más información, consulte el tabla en la Sección 4.2.

4.3.2. Solicitud de certificado

Un servidor que se está autenticando con un certificado PUEDE solicitar opcionalmente un certificado del cliente. Este mensaje, si se envía, DEBE seguir a EncryptedExtensions.

Estructura de este mensaje:

```
struct
        { certificado opaco_request_context<0..2^8-1>; Extensión
         extensiones<2..2^16-1>; } Solicitud de certificado;
```

Machine Translated by Google

Certificate_request_context: una cadena opaca que identifica la solicitud de certificado y que se repetirá en el mensaje de certificado del cliente. El context_request_context DEBE ser único dentro del alcance de esta conexión (evitando así la reproducción de los mensajes CertificateVerify del cliente). Este campo DEBERÁ tener una longitud de cero a menos que se utilice para los intercambios de autenticación posteriores al protocolo de enlace descritos en la Sección 4.6.2. Al solicitar la autenticación posterior al protocolo de enlace, el servidor DEBERÍA hacer que el contexto sea impredecible para el cliente (por ejemplo, generándolo aleatoriamente) para evitar que un atacante que tenga acceso temporal a la clave privada del cliente calcule previamente los mensajes CertificateVerify válidos.

extensiones: un conjunto de extensiones que describen los parámetros del certificado que se solicita. La extensión "signature_algorithms" DEBE especificarse, y otras extensiones pueden incluirse opcionalmente si se definen para este mensaje. Los clientes DEBEN ignorar las extensiones no reconocidas.

En versiones anteriores de TLS, el mensaje CertificateRequest incluía una lista de algoritmos de firma y autoridades de certificación que aceptaría el servidor. En TLS 1.3, el primero se expresa enviando las extensiones "signature_algorithms" y, opcionalmente, "signature_algorithms_cert". Esto último se expresa mediante el envío de la extensión "certificate_authorities" (ver Sección 4.2.4).

Los servidores que se autentican con un PSK NO DEBEN enviar el mensaje CertificateRequest en el protocolo de enlace principal, aunque PUEDEN enviarlo en la autenticación posterior al protocolo de enlace (consulte la Sección 4.6.2) siempre que el cliente haya enviado la extensión "post_handshake_auth" (consulte la Sección 4.2). .6).

4.4. Mensajes de autenticación

Como se discutió en la Sección 2, TLS generalmente usa un conjunto común de mensajes para la autenticación, la confirmación de la clave y la integridad del protocolo de enlace: Certificate, CertificateVerify y Finished. (Los enlazadores de PSK también realizan la confirmación de la clave, de manera similar). Estos tres mensajes siempre se envían como los últimos mensajes en su vuelo de protocolo de enlace. Los mensajes Certificate y CertificateVerify solo se envían en determinadas circunstancias, como se define a continuación. El mensaje Finalizado siempre se envía como parte del Bloque de autenticación.

Estos mensajes se cifran con claves derivadas de [sender]_handshake_traffic_secret.

Machine Translated by Google

Todos los cálculos para los mensajes de autenticación se llevan a cabo de manera uniforme. las siguientes entradas:

- El certificado y clave de firma a utilizar.

- Un Handshake Context que consiste en el conjunto de mensajes a ser incluido en el hash de la transcripción.

- Una clave base que se utilizará para calcular una clave MAC.

Según estas entradas, los mensajes contienen:

Certificado: El certificado que se utilizará para la autenticación, y cualquier certificados de respaldo en la cadena. Tenga en cuenta que basado en certificado la autenticación del cliente no está disponible en los flujos de protocolo de enlace de PSK (incluyendo 0-RTT).

CertificateVerify: una firma sobre el valor Transcripción-Hash (Contexto de protocolo de enlace, Certificado).

Terminado: un MAC sobre el valor Transcript-Hash(Handshake Context, Certificate, CertificateVerify) usando una clave MAC derivada del Clave básica.

La siguiente tabla define el contexto del protocolo de enlace y la clave base MAC para cada escenario:

Modo	Contexto de apretón de manos	Clave básica
Servidor	ClienteHola ... más tarde de extensiones cifradas/ Solicitud de certificado	server_handshake_traffic_ secreto
Cliente	ClienteHola ... más tarde de servidor Terminado/FinDeDatosEarly	client_handshake_traffic_ secreto
Post-apretón de manos	ClienteHola... cliente Terminado + Solicitud de certificado	client_application_traffic_ secreto_N

Machine Translated by Google

4.4.1. El hash de la transcripción

Muchos de los cálculos criptográficos en TLS utilizan un hash de transcripción. Este valor se calcula mediante el hash de la concatenación de cada mensaje de reconocimiento incluido, incluido el encabezado del mensaje de reconocimiento que contiene los campos de tipo y longitud del mensaje de reconocimiento, pero sin incluir los encabezados de la capa de registro. Es decir,

```
Transcripción-Hash(M1, M2, ... Mn) = Hash(M1 || M2 || ... || Mn)
```

Como excepción a esta regla general, cuando el servidor responde a ClientHello con HelloRetryRequest, el valor de ClientHello1 se reemplaza con un mensaje de reconocimiento sintético especial del tipo de reconocimiento "message_hash" que contiene Hash(ClientHello1). Es decir,

```
Transcript-Hash(ClientHello1, HelloRetryRequest, ... Mn) = Hash(message_hash || 00 00
    Hash.length || /* Longitud del mensaje de protocolo de enlace (bytes) */ Hash(ClientHello1)
        || /* Hash de ClientHello1 */ HelloRetryRequest || ... || Mn)
```

El motivo de esta construcción es permitir que el servidor realice una HelloRetryRequest sin estado almacenando solo el hash de ClientHello1 en la cookie, en lugar de exigirle que exporte todo el estado hash intermedio (consulte la Sección 4.2.2).

Para ser más concretos, el hash de la transcripción siempre se toma de la siguiente secuencia de mensajes de protocolo de enlace, comenzando en el primer ClientHello e incluyendo solo los mensajes que se enviaron: ClientHello, HelloRetryRequest, ClientHello, ServerHello, EncryptedExtensions, server CertificateRequest, server Certificate, server CertificateVerify, servidor Finalizado, EndOfEarlyData, Certificado de cliente, CertificateVerify de cliente, Cliente Finalizado.

En general, las implementaciones pueden implementar la transcripción manteniendo un valor hash de transcripción en ejecución basado en el hash negociado. Tenga en cuenta, sin embargo, que las autenticaciones posteriores al protocolo de enlace no se incluyen entre sí, solo los mensajes hasta el final del protocolo de enlace principal.

Machine Translated by Google

4.4.2. Certificado

Este mensaje transmite la cadena de certificados del extremo al par.

El servidor DEBE enviar un mensaje de Certificado cada vez que el método de intercambio de claves acordado utilice certificados para la autenticación (esto incluye todos los métodos de intercambio de claves definidos en este documento excepto PSK).

El cliente DEBE enviar un mensaje de Certificado si y solo si el servidor ha solicitado la autenticación del cliente a través de un mensaje CertificateRequest (Sección 4.3.2). Si el servidor solicita la autenticación del cliente pero no hay un certificado adecuado disponible, el cliente DEBE enviar un mensaje de Certificado que no contenga certificados (es decir, con el campo "certificate_list" de longitud 0). Se DEBE enviar un mensaje Finalizado independientemente de si el mensaje Certificado está vacío.

Estructura de este mensaje:

```
enumeración {
        X509(0),
        RawPublicKey(2), (255)

} Tipo de certificado;

struct { select
        (tipo_certificado) { case RawPublicKey: /* De
                RFC 7250 ASN.1_subjectPublicKeyInfo
                    */ opaco ASN1_subjectPublicKeyInfo<1..2^24-1>;


                caso X509:
                    opaco cert_data<1..2^24-1>;
        };
        Extensión extensiones<0..2^16-1>; } Entrada de
certificado;

struct
        { certificado opaco_request_context<0..2^8-1>; CertificateEntry
        lista_certificados<0..2^24-1>; } Certificado;
```

Machine Translated by Google

Certificate_request_context: si este mensaje es en respuesta a un CertificateRequest, el valor de certificate_request_context en ese mensaje. De lo contrario (en el caso de la autenticación del servidor), este campo DEBERÁ tener una longitud cero.

Certificate_list: una secuencia (cadena) de estructuras CertificateEntry, cada una de las cuales contiene un solo certificado y un conjunto de extensiones.

extensiones: un conjunto de valores de extensión para CertificateEntry. El formato de "Extensión" se define en la Sección 4.2. Las extensiones válidas para los certificados de servidor en la actualidad incluyen la extensión de estado OCSP [RFC6066] y la extensión SignedCertificateTimestamp [RFC6962]; También se pueden definir extensiones futuras para este mensaje. Las extensiones en el mensaje de certificado del servidor DEBEN corresponder a las del mensaje ClientHello. Las extensiones en el mensaje de certificado del cliente DEBEN corresponder a las extensiones en el mensaje CertificateRequest del servidor. Si una extensión se aplica a toda la cadena, DEBERÍA incluirse en el primer CertificateEntry.

Si la extensión de tipo de certificado correspondiente ("server_certificate_type" o "client_certificate_type") no se negoció en EncryptedExtensions, o se negoció el tipo de certificado X.509, entonces cada CertificateEntry contiene un certificado X.509 codificado en DER. El certificado del remitente DEBE aparecer en el primer CertificateEntry de la lista. Cada certificado siguiente DEBERÍA certificar directamente el inmediatamente anterior. Debido a que la validación de certificados requiere que los anclajes de confianza se distribuyan de forma independiente, un certificado que especifique un anclaje de confianza PUEDE omitirse de la cadena, siempre que se sepa que los pares admitidos poseen cualquier certificado omitido.

Nota: Antes de TLS 1.3, el pedido de "certificate_list" requería que cada certificado certificara el inmediatamente anterior; sin embargo, algunas implementaciones permitieron cierta flexibilidad. Los servidores a veces envían un intermedio actual y obsoleto con fines de transición, y otros simplemente se configuran incorrectamente, pero estos casos, no obstante, se pueden validar correctamente. Para una máxima compatibilidad, todas las implementaciones DEBERÍAN estar preparadas para manejar certificados potencialmente extraños y ordenaciones arbitrarias de cualquier versión de TLS, con la excepción del certificado de entidad final que DEBE ser el primero.

Si se negoció el tipo de certificado RawPublicKey, entonces la lista de certificados DEBE contener no más de una CertificateEntry, que contiene un valor ASN1_subjectPublicKeyInfo como se define en [RFC7250], Sección 3.

Machine Translated by Google

El tipo de certificado OpenPGP [RFC6091] NO DEBE utilizarse con TLS 1.3.

La lista de certificados del servidor DEBE estar siempre no vacía. Un cliente enviará una lista de certificados vacía si no tiene un certificado adecuado para enviar en respuesta a la solicitud de autenticación del servidor.

4.4.2.1. Estado OCSP y Extensiones SCT

[RFC6066] y [RFC6961] proporcionan extensiones para negociar el envío del servidor de respuestas OCSP al cliente. En TLS 1.2 y versiones anteriores, el servidor responde con una extensión vacía para indicar la negociación de esta extensión y la información OCSP se transporta en un mensaje CertificateStatus. En TLS 1.3, la información OCSP del servidor se transporta en una extensión en CertificateEntry que contiene el certificado asociado. Específicamente, el cuerpo de la extensión "status_request" del servidor DEBE ser una estructura CertificateStatus como se define en [RFC6066], que se interpreta como se define en [RFC6960].

Nota: La extensión status_request_v2 [RFC6961] está obsoleta.
Los servidores TLS 1.3 NO DEBEN actuar sobre su presencia o información en él al procesar mensajes de ClientHello; en particular, NO DEBEN enviar la extensión status_request_v2 en los mensajes EncryptedExtensions, CertificateRequest o Certificate. Los servidores TLS 1.3 DEBEN poder procesar mensajes ClientHello que lo incluyan, ya que PUEDEN ser enviados por clientes que deseen usarlo en versiones anteriores del protocolo.

Un servidor PUEDE solicitar que un cliente presente una respuesta OCSP con su certificado enviando una extensión "status_request" vacía en su mensaje CertificateRequest. Si el cliente opta por enviar una respuesta OCSP, el cuerpo de su extensión "status_request" DEBE ser una estructura CertificateStatus como se define en [RFC6066].

De manera similar, [RFC6962] proporciona un mecanismo para que un servidor envíe una marca de tiempo de certificado firmado (SCT) como una extensión en ServerHello en TLS 1.2 y versiones anteriores. En TLS 1.3, la información SCT del servidor se transporta en una extensión en CertificateEntry.

Machine Translated by Google

4.4.2.2. Selección de certificado de servidor

Las siguientes reglas se aplican a los certificados enviados por el servidor:

- El tipo de certificado DEBE ser X.509v3 [RFC5280], a menos que sea explícitamente negociado de otra manera (por ejemplo, [RFC7250]).

- La clave pública del certificado de entidad final del servidor (y restricciones) DEBE ser compatible con el algoritmo de autenticación seleccionado de la extensión "signature_algorithms" del cliente (actualmente RSA, ECDSA o EdDSA).

- El certificado DEBE permitir que la clave se use para firmar (es decir, el bit digitalSignature DEBE establecerse si la extensión Key Usage está presente) con un esquema de firma indicado en las extensiones "signature_algorithms"/"signature_algorithms_cert" del cliente (consulte la Sección 4.2.3).

- El "nombre_del_servidor" [RFC6066] y las "autoridades_del_certificado" las extensiones se utilizan para guiar la selección de certificados. Como los servidores PUEDEN requerir la presencia de la extensión "server_name", los clientes DEBERÍAN enviar esta extensión, cuando corresponda.

Todos los certificados proporcionados por el servidor DEBEN estar firmados por un algoritmo de firma anunciado por el cliente si puede proporcionar dicha cadena (consulte la Sección 4.2.3). Los certificados que son autofirmados o los certificados que se espera que sean anclajes de confianza no se validan como parte de la cadena y, por lo tanto, PUEDEN firmarse con cualquier algoritmo.

Si el servidor no puede producir una cadena de certificados que esté firmada solo a través de los algoritmos compatibles indicados, DEBERÍA continuar con el protocolo de enlace enviando al cliente una cadena de certificados de su elección que puede incluir algoritmos que el cliente no sabe que son compatibles. Esta cadena de respaldo NO DEBE usar el algoritmo hash SHA-1 en desuso en general, pero PUEDE hacerlo si el anuncio del cliente lo permite, y NO DEBE hacerlo de otra manera.

Si el cliente no puede construir una cadena aceptable utilizando los certificados provistos y decide cancelar el protocolo de enlace, DEBE cancelar el protocolo de enlace con una alerta relacionada con el certificado correspondiente (de forma predeterminada, "certificado_no admitido"; consulte la Sección 6.2 para obtener más información).

Si el servidor tiene varios certificados, elige uno de ellos en función de los criterios mencionados anteriormente (además de otros criterios, como el punto final de la capa de transporte, la configuración local y las preferencias).

Machine Translated by Google

4.4.2.3. Selección de certificado de cliente

Las siguientes reglas se aplican a los certificados enviados por el cliente:

- El tipo de certificado DEBE ser X.509v3 [RFC5280], a menos que sea explícitamente negociado de otra manera (por ejemplo, [RFC7250]).

- Si la extensión "certificate_authorities" en el El mensaje CertificateRequest estaba presente, al menos uno de los certificados en la cadena de certificados DEBERÍA ser emitido por una de las CA enumeradas.

- Los certificados DEBEN estar firmados con una firma aceptable algoritmo, como se describe en la Sección 4.3.2. Tenga en cuenta que esto relaja las restricciones sobre los algoritmos de firma de certificados que se encuentran en versiones anteriores de TLS.

- Si el mensaje CertificateRequest contenía un mensaje no vacío extensión "oid_filters", el certificado de entidad final DEBE coincidir con los OID de extensión que reconoce el cliente, como se describe en la Sección 4.2.5.

4.4.2.4. Recibir un mensaje de certificado

En general, los procedimientos detallados de validación de certificados están fuera del alcance de TLS (consulte [RFC5280]). Esta sección proporciona requisitos específicos de TLS.

Si el servidor proporciona un mensaje de certificado vacío, el cliente DEBE cancelar el protocolo de enlace con una alerta de "decode_error".

Si el cliente no envía ningún certificado (es decir, envía un mensaje de Certificado vacío), el servidor PUEDE, a su discreción, continuar con el protocolo de enlace sin autenticación del cliente o cancelar el protocolo de enlace con una alerta de "certificado_requerido". Además, si algún aspecto de la cadena de certificados fuera inaceptable (por ejemplo, no fue firmado por una CA conocida y confiable), el servidor PUEDE, a su discreción, continuar con el protocolo de enlace (considerando que el cliente no está autenticado) o cancelar el protocolo de enlace.

Cualquier punto final que reciba un certificado que necesitaría validar usando cualquier algoritmo de firma usando un hash MD5 DEBE abortar el protocolo de enlace con una alerta "bad_certificate". SHA-1 está en desuso, y se RECOMIENDA que cualquier punto final que reciba un certificado que necesite validar usando cualquier algoritmo de firma que use un hash SHA-1 cancele el protocolo de enlace con una alerta "bad_certificate". Para mayor claridad, esto significa que los puntos finales pueden aceptar estos algoritmos para certificados autofirmados o anclas de confianza.

Machine Translated by Google

Se RECOMIENDA que todos los puntos finales hagan la transición a SHA-256 o mejor lo antes posible para mantener la interoperabilidad con las implementaciones que actualmente están en proceso de eliminar el soporte de SHA-1.

Tenga en cuenta que un certificado que contiene una clave para un algoritmo de firma PUEDE firmarse con un algoritmo de firma diferente (por ejemplo, una clave RSA firmada con una clave ECDSA).

4.4.3. Certificado Verificar

Este mensaje se utiliza para proporcionar una prueba explícita de que un punto final posee la clave privada correspondiente a su certificado. El mensaje CertificateVerify también proporciona integridad para el protocolo de enlace hasta este punto. Los servidores DEBEN enviar este mensaje al autenticarse a través de un certificado. Los clientes DEBEN enviar este mensaje cada vez que se autentican a través de un certificado (es decir, cuando el mensaje del certificado no está vacío). Cuando se envía, este mensaje DEBE aparecer inmediatamente después del mensaje Certificado e inmediatamente antes del mensaje Finalizado.

mensaje.

Estructura de este mensaje:

```
struct
        {Algoritmo SignatureScheme; firma
        opaca<0..2^16-1>;
} CertificadoVerificar;
```

El campo de algoritmo especifica el algoritmo de firma utilizado (ver la Sección 4.2.3 para la definición de este tipo). La firma es una firma digital que utiliza ese algoritmo. El contenido que cubre la firma es la salida hash como se describe en la Sección 4.4.1, a saber:

Transcripción-Hash (Contexto de protocolo de enlace, Certificado)

A continuación, la firma digital se calcula sobre la concatenación de:

- Una cadena que consta del octeto 32 (0x20) repetido 64 veces

- La cadena de contexto

- Un solo byte 0 que sirve como separador

- El contenido a firmar.

Machine Translated by Google

Esta estructura está destinada a evitar un ataque a versiones anteriores de TLS en las que el formato ServerKeyExchange significaba que los atacantes podían obtener una firma de un mensaje con un prefijo de 32 bytes elegido (ClientHello.random). El pad inicial de 64 bytes borra ese prefijo junto con ServerHello.random controlado por el servidor.

La cadena de contexto para una firma de servidor es "TLS 1.3, servidor CertificateVerify". La cadena de contexto para una firma de cliente es "TLS 1.3, cliente CertificateVerify". Se utiliza para proporcionar separación entre firmas realizadas en diferentes contextos, lo que ayuda contra posibles ataques entre protocolos.

Por ejemplo, si el hash de la transcripción fuera 32 bytes de 01 (esta longitud tendría sentido para SHA-256), el contenido cubierto por la firma digital para un servidor CertificateVerify sería:

```
2020202020202020202020202020202020202020202020202020202020202020202020202020202020202020202020202020202020202020202020202020202020
```

Del lado del remitente, el proceso para calcular el campo de firma del mensaje CertificateVerify toma como entrada:

- El contenido amparado por la firma digital

- La clave de firma privada correspondiente al certificado enviado en el mensaje anterior

Si el mensaje CertificateVerify es enviado por un servidor, el algoritmo de firma DEBE ser uno que se ofrezca en la extensión "signature_algorithms" del cliente, a menos que no se pueda generar una cadena de certificados válida sin algoritmos no admitidos (consulte la Sección 4.2.3).

Si lo envía un cliente, el algoritmo de firma utilizado en la firma DEBE ser uno de los presentes en el campo support_signature_algorithms de la extensión "signature_algorithms" en el mensaje CertificateRequest.

Además, el algoritmo de firma DEBE ser compatible con la clave del certificado de entidad final del remitente. Las firmas RSA DEBEN usar un algoritmo RSASSA-PSS, independientemente de si los algoritmos RSASSA-PKCS1-v1_5 aparecen en "signature_algorithms". El algoritmo SHA-1 NO DEBE usarse en ninguna firma de mensajes CertificateVerify.

Machine Translated by Google

Todos los algoritmos de firma SHA-1 en esta especificación están definidos únicamente para su uso en certificados heredados y no son válidos para firmas CertificateVerify.

El receptor de un mensaje CertificateVerify DEBE verificar el campo de la firma. El proceso de verificación toma como entrada:

- El contenido amparado por la firma digital

- La clave pública contenida en el certificado de entidad final que se encuentra en el mensaje de certificado asociado

- La firma digital recibida en el campo de firma del CertificadoVerificar mensaje

Si la verificación falla, el receptor DEBE finalizar el protocolo de enlace con una alerta "decrypt_error".

4.4.4. Finalizado

El mensaje Finalizado es el mensaje final en el Bloque de autenticación. Es esencial para proporcionar la autenticación del protocolo de enlace y de las claves calculadas.

Los destinatarios de los mensajes finalizados DEBEN verificar que el contenido sea correcto y, si es incorrecto, DEBEN finalizar la conexión con una alerta "decrypt_error".

Una vez que un lado ha enviado su mensaje Finalizado y ha recibido y validado el mensaje Finalizado de su par, puede comenzar a enviar y recibir Datos de la aplicación a través de la conexión. Hay dos configuraciones en las que se permite enviar datos antes de recibir el Finalizado del par:

1. Clientes que envían datos 0-RTT como se describe en la Sección 4.2.10.

2. Los servidores PUEDEN enviar datos después de enviar su primer vuelo, pero debido a que el apretón de manos aún no está completo, no tienen seguridad de la identidad del par o de su actividad (es decir, ClientHello podría haber sido reproducido).

Machine Translated by Google

La clave utilizada para calcular el mensaje Finalizado se calcula a partir de la Clave base definida en la Sección 4.4 mediante HKDF (consulte la Sección 7.1).
Específicamente:

```
finish_key = HKDF-
        Expand-Label(BaseKey, "finished", "", Hash.length)
```

Estructura de este mensaje:

```
struct { opaco
        verificar_datos[Hash.longitud];
} Finalizado;
```

El valor de verificar_datos se calcula de la siguiente manera:

```
verificar_datos =
        HMAC(terminado_clave,
                Transcripción-Hash (Contexto de apretón de manos,
                                        Certificado*, CertificadoVerificar*))

* Solo incluido si está presente.
```

HMAC [RFC2104] utiliza el algoritmo Hash para el protocolo de enlace. Como se señaló anteriormente, la entrada HMAC generalmente se puede implementar mediante un hash en ejecución, es decir, solo el hash de protocolo de enlace en este punto.

En versiones anteriores de TLS, verificar_datos siempre tenía una longitud de 12 octetos. En TLS 1.3, es el tamaño de la salida HMAC para el Hash utilizado para el protocolo de enlace.

Nota: Las alertas y cualquier otro tipo de registro que no sea de protocolo de enlace no son mensajes de protocolo de enlace y no se incluyen en los cálculos de hash.

Todos los registros que siguen a un mensaje Finalizado DEBEN cifrarse con la clave de tráfico de la aplicación adecuada, como se describe en la Sección 7.2. En particular, esto incluye cualquier alerta enviada por el servidor en respuesta a los mensajes Certificate y CertificateVerify del cliente.

4.5. Fin de los primeros datos

```
estructura {} EndOfEarlyData;
```

Si el servidor envió una extensión "early_data" en EncryptedExtensions, el cliente DEBE enviar un mensaje EndOfEarlyData después de recibir el mensaje Finalizado del servidor. Si el servidor no envía una extensión "early_data" en EncryptedExtensions, entonces el cliente NO DEBE enviar un mensaje EndOfEarlyData. Este mensaje indica que todos los mensajes de datos de aplicación 0-RTT, si los hay, se han transmitido y que el

Machine Translated by Google

Los siguientes registros están protegidos por claves de tráfico de protocolo de enlace.
Los servidores NO DEBEN enviar este mensaje, y los clientes que lo reciben DEBEN finalizar la conexión con una alerta de "mensaje_inesperado". Este mensaje está cifrado con claves derivadas de client_early_traffic_secret.

4.6. Mensajes posteriores al apretón de manos

TLS también permite enviar otros mensajes después del protocolo de enlace principal.
Estos mensajes utilizan un tipo de contenido de protocolo de enlace y se cifran con la clave de tráfico de la aplicación adecuada.

4.6.1. Nuevo mensaje de ticket de sesión

En cualquier momento después de que el servidor haya recibido el mensaje Finalizado del cliente, PUEDE enviar un mensaje NewSessionTicket. Este mensaje crea una asociación única entre el valor del ticket y un PSK secreto derivado del secreto maestro de reanudación (consulte la Sección 7).

El cliente PUEDE usar este PSK para futuros protocolos de enlace al incluir el valor del ticket en la extensión "pre_shared_key" en su ClientHello (Sección 4.2.11). Los servidores PUEDEN enviar múltiples boletos en una sola conexión, ya sea inmediatamente después de otro o después de eventos específicos (consulte el Apéndice C.4). Por ejemplo, el servidor podría enviar un nuevo ticket después de la autenticación posterior al protocolo de enlace para encapsular el estado de autenticación del cliente adicional. Los boletos múltiples son útiles para los clientes para una variedad de propósitos, que incluyen:

- Apertura de múltiples conexiones HTTP paralelas.

- Realización de carreras de conexión a través de interfaces y direcciones.
  familias a través de (por ejemplo) Happy Eyeballs [RFC8305] o técnicas relacionadas.

Cualquier ticket DEBE reanudarse únicamente con un conjunto de cifrado que tenga el mismo algoritmo hash KDF que el utilizado para establecer la conexión original.

Los clientes DEBEN reanudar solo si el nuevo valor de SNI es válido para el certificado de servidor presentado en la sesión original y DEBERÍAN reanudar solo si el valor de SNI coincide con el utilizado en la sesión original. Este último es una optimización del rendimiento: normalmente, no hay razón para esperar que diferentes servidores cubiertos por un solo certificado puedan aceptar los boletos de los demás; por lo tanto, intentar la reanudación en ese caso desperdiciaría un boleto de un solo uso. Si se proporciona dicha indicación (externamente o por cualquier otro medio), los clientes PUEDEN reanudar con un valor de SNI diferente.

Machine Translated by Google

En la reanudación, si informa un valor de SNI a la aplicación que llama, las implementaciones DEBEN utilizar el valor enviado en la reanudación ClientHello en lugar del valor enviado en la sesión anterior. Tenga en cuenta que si una implementación de servidor rechaza todas las identidades de PSK con diferentes valores de SNI, estos dos valores son siempre los mismos.

Nota: Aunque el secreto maestro de reanudación depende del segundo vuelo del cliente, un servidor que no solicita la autenticación del cliente PUEDE calcular el resto de la transcripción de forma independiente y luego enviar un NewSessionTicket inmediatamente después de enviar su Finalizado en lugar de esperar a que el cliente Finalice. Esto podría ser apropiado en los casos en los que se espera que el cliente abra varias conexiones TLS en paralelo y se beneficiaría de la sobrecarga reducida de un protocolo de enlace de reanudación, por ejemplo.

```
estructura
        { uint32 ticket_lifetime; uint32
        ticket_age_add; ticket_nonce
        opaco<0..255>; ticket opaco<1..2^16-1>;
        Extensión extensiones<0..2^16-2>; }
        NewSessionTicket;
```

ticket_lifetime: Indica el tiempo de vida en segundos como un 32-bit entero sin signo en el orden de bytes de la red desde el momento de la emisión del billete. Los servidores NO DEBEN utilizar ningún valor superior a 604800 segundos (7 días). El valor de cero indica que el boleto debe descartarse inmediatamente. Los clientes NO DEBEN almacenar los boletos en caché durante más de 7 días, independientemente de la duración del boleto, y PUEDEN eliminar los boletos antes según la política local. Un servidor PUEDE tratar un boleto como válido por un período de tiempo más corto que el establecido en ticket_lifetime.

ticket_age_add: un valor aleatorio de 32 bits generado de forma segura que es se utiliza para ocultar la antigüedad del ticket que el cliente incluye en la extensión "pre_shared_key". A este valor se suma la edad del ticket del lado del cliente módulo 2^32 para obtener el valor que transmite el cliente. El servidor DEBE generar un valor nuevo para cada ticket que envía.

ticket_nonce: un valor por boleto que es único en todos los boletos emitido en esta conexión.

Machine Translated by Google

ticket: El valor del ticket que se utilizará como identidad PSK. El billete en sí es una etiqueta opaca. PUEDE ser una clave de búsqueda en la base de datos o un valor autocifrado y autoautenticado.

extensiones: un conjunto de valores de extensión para el ticket. los
El formato de "Extensión" se define en la Sección 4.2. Los clientes DEBEN ignorar las extensiones no reconocidas.

La única extensión definida actualmente para NewSessionTicket es "early_data", lo que indica que el ticket se puede usar para enviar datos 0-RTT (Sección 4.2.10). Contiene el siguiente valor:

max_early_data_size: la cantidad máxima de datos 0-RTT que el
el cliente puede enviar cuando usa este ticket, en bytes. Solo se cuenta la carga útil de los datos de la aplicación (es decir, el texto sin formato pero no el relleno o el byte de tipo de contenido interno). Un servidor que recibe más de max_early_data_size bytes de datos 0-RTT DEBERÍA terminar la conexión con una alerta de "mensaje_inesperado". Tenga en cuenta que los servidores que rechazan los primeros datos debido a la falta de material criptográfico no podrán diferenciar el relleno del contenido, por lo que los clientes NO DEBEN depender de poder enviar grandes cantidades de relleno en los primeros registros de datos.

El PSK asociado con el boleto se calcula como:

```
HKDF-Expand-Label(resumen_maestro_secreto,
                  "reanudación", ticket_nonce, Hash.length)
```

Debido a que el valor ticket_nonce es distinto para cada mensaje NewSessionTicket, se derivará un PSK diferente para cada ticket.

Tenga en cuenta que, en principio, es posible continuar emitiendo nuevos boletos que extiendan indefinidamente la vida útil del material de clave derivado originalmente de un protocolo de enlace inicial que no sea PSK (que probablemente estaba vinculado al certificado del par). Se RECOMIENDA que las implementaciones establezcan límites en la vida útil total de dicho material clave; estos límites deben tener en cuenta la vigencia del certificado del par, la probabilidad de revocación intermedia y el tiempo transcurrido desde la firma CertificateVerify en línea del par.

4.6.2. Autenticación posterior al protocolo de enlace

Cuando el cliente ha enviado la extensión "post_handshake_auth" (consulte la Sección 4.2.6), un servidor PUEDE solicitar la autenticación del cliente en cualquier momento después de que se haya completado el protocolo de enlace mediante el envío de un mensaje CertificateRequest. El cliente DEBE responder con los mensajes de Autenticación apropiados (consulte la Sección 4.4). Si el cliente elige autenticarse, DEBE enviar Certificate, CertificateVerify,

Machine Translated by Google

y terminó. Si se niega, DEBE enviar un mensaje de Certificado que no contenga certificados seguido de Finalizado. Todos los mensajes del cliente para una respuesta dada DEBEN aparecer consecutivamente en el cable sin mensajes intermedios de otros tipos.

Un cliente que recibe un mensaje CertificateRequest sin haber enviado la extensión "post_handshake_auth" DEBE enviar una alerta fatal "unexpected_message".

Nota: Debido a que la autenticación del cliente podría implicar solicitar al usuario, los servidores DEBEN estar preparados para algún retraso, incluida la recepción de una cantidad arbitraria de otros mensajes entre el envío de CertificateRequest y la recepción de una respuesta. Además, los clientes que reciben varias solicitudes de certificado seguidas PUEDEN responderlas en un orden diferente al que se recibieron (el valor del contexto de solicitud de certificado permite que el servidor elimine la ambigüedad de las respuestas).

4.6.3. Actualización de clave y vector de inicialización

El mensaje de protocolo de enlace KeyUpdate se utiliza para indicar que el remitente está actualizando sus claves criptográficas de envío. Este mensaje puede ser enviado por cualquiera de los pares después de haber enviado un mensaje Finalizado. Las implementaciones que reciben un mensaje KeyUpdate antes de recibir un mensaje Finalizado DEBEN finalizar la conexión con una alerta de "mensaje_inesperado". Después de enviar un mensaje KeyUpdate, el remitente DEBE enviar todo su tráfico utilizando la próxima generación de claves, calculadas como se describe en la Sección 7.2. Al recibir KeyUpdate, el receptor DEBE actualizar sus claves de recepción.

```
enumeración { actualización_no_solicitada(0), actualización_solicitada(1), (255)
} Solicitud de actualización de clave;

estructura {
        KeyUpdateRequest request_update;
} Actualización de clave;
```

request_update: indica si el destinatario de KeyUpdate debe responder con su propia KeyUpdate. Si una implementación recibe cualquier otro valor, DEBE terminar la conexión con una alerta de "parámetro_illegal".

Si el campo request_update se establece en "update_requested", entonces el receptor DEBE enviar una KeyUpdate propia con request_update establecido en "update_not_requested" antes de enviar su siguiente registro de datos de aplicación. Este mecanismo permite que cualquiera de los lados fuerce una actualización de toda la conexión, pero provoca una implementación que recibe

Machine Translated by Google

múltiples KeyUpdates mientras está en silencio para responder con una sola actualización. Tenga en cuenta que las implementaciones pueden recibir una cantidad arbitraria de mensajes entre el envío de KeyUpdate con request_update establecido en "update_requested" y la recepción de KeyUpdate del par, porque es posible que esos mensajes ya estén en tránsito. Sin embargo, debido a que las claves de envío y recepción se derivan de secretos de tráfico independientes, conservar el secreto de tráfico de recepción no amenaza la confidencialidad de los datos enviados antes de que el remitente cambiara las claves.

Si las implementaciones envían de forma independiente sus propias KeyUpdates con request_update establecido en "update_requested" y se cruzan en vuelo, cada lado también enviará una respuesta, con el resultado de que cada lado se incrementa en dos generaciones.

Tanto el remitente como el receptor DEBEN cifrar sus mensajes KeyUpdate con las claves antiguas. Además, ambas partes DEBEN exigir que se reciba una actualización de clave con la clave anterior antes de aceptar cualquier mensaje cifrado con la nueva clave. El no hacerlo puede permitir ataques de truncamiento de mensajes.

5. Protocolo de registro

El protocolo de registro TLS toma mensajes para transmitir, fragmenta los datos en bloques manejables, protege los registros y transmite el resultado. Los datos recibidos se verifican, descifran, vuelven a ensamblar y luego se envían a clientes de nivel superior.

Los registros TLS se escriben, lo que permite multiplexar múltiples protocolos de nivel superior sobre la misma capa de registro. Este documento especifica cuatro tipos de contenido: protocolo de enlace, application_data, alerta y change_cipher_spec. El registro change_cipher_spec se usa solo con fines de compatibilidad (consulte el Apéndice D.4).

Una implementación puede recibir un registro sin cifrar de tipo change_cipher_spec que consta del valor de un solo byte 0x01 en cualquier momento después de que se haya enviado o recibido el primer mensaje ClientHello y antes de que se haya recibido el mensaje Finalizado del par y DEBE simplemente descartarlo sin más procesamiento. Tenga en cuenta que este registro puede aparecer en un punto del protocolo de enlace donde la implementación espera registros protegidos, por lo que es necesario detectar esta condición antes de intentar desproteger el registro. Una implementación que recibe cualquier otro valor change_cipher_spec o que recibe un registro change_cipher_spec protegido DEBE abortar el protocolo de enlace con una alerta de "mensaje_inesperado". Si una implementación detecta un registro change_cipher_spec recibido antes del primer mensaje ClientHello o después del mensaje Finalizado del par, DEBE tratarse como un tipo de registro inesperado (aunque es posible que los servidores sin estado no puedan distinguir estos casos de los casos permitidos).

Machine Translated by Google

Las implementaciones NO DEBEN enviar tipos de registro no definidos en este documento a menos que se negocien por alguna extensión. Si una implementación de TLS recibe un tipo de registro inesperado, DEBE finalizar la conexión con una alerta de "mensaje_inesperado". La IANA asigna nuevos valores de tipo de contenido de registro en el registro TLS ContentType como se describe en la Sección 11.

5.1. Capa de registro

La capa de registro fragmenta los bloques de información en registros TLSPlaintext que transportan datos en fragmentos de 2^14 bytes o menos. Los límites de los mensajes se manejan de manera diferente según el tipo de contenido subyacente. Cualquier tipo de contenido futuro DEBE especificar las reglas apropiadas. Tenga en cuenta que estas reglas son más estrictas que las aplicadas en TLS 1.2.

Los mensajes de protocolo de enlace PUEDEN fusionarse en un solo registro de texto sin formato TLSP o fragmentarse en varios registros, siempre que:

- Los mensajes de apretón de manos NO DEBEN intercalarse con otro registro tipos Es decir, si un mensaje de protocolo de enlace se divide en dos o más registros, NO DEBE haber ningún otro registro entre ellos.

- Los mensajes de protocolo de enlace NO DEBEN abarcar cambios de clave. Implementaciones DEBE verificar que todos los mensajes que preceden inmediatamente a un cambio de clave se alineen con un límite de registro; si no, DEBEN terminar la conexión con una alerta de "mensaje_inesperado". Debido a que los mensajes ClientHello, EndOfEarlyData, ServerHello, Finished y KeyUpdate pueden preceder inmediatamente a un cambio de clave, las implementaciones DEBEN enviar estos mensajes alineados con un límite de registro.

Las implementaciones NO DEBEN enviar fragmentos de tipo Handshake de longitud cero, incluso si esos fragmentos contienen relleno.

Los mensajes de alerta (Sección 6) NO DEBEN fragmentarse entre registros, y los mensajes de alerta múltiples NO DEBEN fusionarse en un solo registro de texto sin formato TLSP. En otras palabras, un registro con un tipo de Alerta DEBE contener exactamente un mensaje.

Los mensajes de datos de aplicación contienen datos que son opacos para TLS.
Los mensajes de datos de aplicación siempre están protegidos. Se PUEDEN enviar fragmentos de datos de aplicación de longitud cero, ya que son potencialmente útiles como contramedida de análisis de tráfico.
Los fragmentos de datos de la aplicación PUEDEN dividirse en varios registros o fusionarse en un solo registro.

Machine Translated by Google


```
enum
        { invalid(0),
         change_cipher_spec(20), alert(21),
         handshake(22), application_data(23),
         (255)


} Tipo de contenido;

estructura
        { tipo de tipo de contenido;
         ProtocolVersion legacy_record_version; uint16 longitud;
         fragmento opaco[TLSPlaintext.length]; } TLSTexto sin formato;
```

tipo: El protocolo de nivel superior utilizado para procesar el adjunto fragmento.

legacy_record_version: DEBE establecerse en 0x0303 para todos los registros generado por una implementación TLS 1.3 que no sea un ClientHello inicial (es decir, uno no generado después de HelloRetryRequest), donde también PUEDE ser 0x0301 por motivos de compatibilidad. Este campo está en desuso y DEBE ignorarse a todos los efectos.

Las versiones anteriores de TLS usarían otros valores en este campo en algunas circunstancias.

longitud: la longitud (en bytes) del siguiente TLSPlaintext.fragment. La longitud NO DEBE exceder los 2^14 bytes. Un punto final que recibe un registro que excede esta longitud DEBE terminar la conexión con una alerta "record_overflow".

fragmento: Los datos que se transmiten. Este valor es transparente y se trata como un bloque independiente para ser tratado por el protocolo de nivel superior especificado por el campo de tipo.

Este documento describe TLS 1.3, que utiliza la versión 0x0304. Este valor de versión es histórico y se deriva del uso de 0x0301 para TLS 1.0 y 0x0300 para SSL 3.0. Para maximizar la compatibilidad con versiones anteriores, un registro que contenga un ClientHello inicial DEBE tener la versión 0x0301 (que refleje TLS 1.0) y un registro que contenga un segundo ClientHello o ServerHello DEBE tener la versión 0x0303 (que refleje TLS 1.2). Al negociar versiones anteriores de TLS, los terminales siguen el procedimiento y los requisitos proporcionados en el Apéndice D.

Machine Translated by Google

Cuando aún no se ha activado la protección de registros, las estructuras TLSPlaintext se escriben directamente en el cable. Una vez que ha comenzado la protección de registros, los registros TLSPlaintext se protegen y envían como se describe en la siguiente sección. Tenga en cuenta que los registros de datos de la aplicación NO DEBEN escribirse en el cable sin protección (consulte la Sección 2 para obtener más detalles).

5.2. Protección de carga útil de registro

Las funciones de protección de registros traducen una estructura TLSPlaintext en una estructura TLSCiphertext. Las funciones de desprotección invierten el proceso. En TLS 1.3, a diferencia de las versiones anteriores de TLS, todos los cifrados se modelan como "Cifrado autenticado con datos asociados" (AEAD) [RFC5116]. Las funciones de AEAD proporcionan una operación unificada de encriptación y autenticación que convierte el texto sin formato en texto cifrado autenticado y viceversa. Cada registro cifrado consta de un encabezado de texto sin formato seguido de un cuerpo cifrado, que a su vez contiene un tipo y un relleno opcional.

```
struct
        { contenido opaco[TLSPlaintext.length]; Tipo de tipo de
         contenido; uint8 ceros[longitud_de_relleno]; } TLSInnerPlaintext;
```

```
struct
        { ContentType opaque_type = application_data; /* 23 */ ProtocolVersion
         legacy_record_version = 0x0303; /* TLS v1.2 */ uint16 longitud; registro_cifrado opaco[TLSCiphertext.length]; }
         TLSCiphertext;
```

contenido: El valor TLSPlaintext.fragment, que contiene el byte codificación de un apretón de manos o un mensaje de alerta, o los bytes sin procesar de los datos de la aplicación para enviar.

tipo: El valor TLSLaintext.type que contiene el tipo de contenido del registro.

ceros: puede aparecer una serie de bytes de valor cero de longitud arbitraria en el texto claro después del campo de tipo. Esto brinda una oportunidad para que los remitentes completen cualquier registro TLS en una cantidad elegida, siempre que el total se mantenga dentro de los límites de tamaño del registro. Consulte la Sección 5.4 para obtener más detalles.

Machine Translated by Google

opaque_type: el campo opaque_type exterior de un registro de TLSCiphertext siempre se establece en el valor 23 (application_data) para la compatibilidad externa con middleboxes acostumbrados a analizar versiones anteriores de TLS. El tipo de contenido real del registro se encuentra en TLSInnerPlaintext.type después del descifrado.

legacy_record_version: El campo legacy_record_version siempre es 0x0303. TLS 1.3 Los TLSCiphertexts no se generan hasta que se ha negociado TLS 1.3, por lo que no hay problemas de compatibilidad histórica donde se pueden recibir otros valores. Tenga en cuenta que el protocolo de negociación, incluidos los mensajes ClientHello y ServerHello, autentica la versión del protocolo, por lo que este valor es redundante.

longitud: la longitud (en bytes) del siguiente TLSCiphertext.encrypted_record, que es la suma de las longitudes del contenido y el relleno, más uno para el tipo de contenido interno, más cualquier expansión agregada por el algoritmo AEAD. La longitud NO DEBE exceder 2^14 + 256 bytes. Un punto final que recibe un registro que excede esta longitud DEBE terminar la conexión con una alerta "record_overflow".

encriptado_registro: La forma encriptada AEAD de la estructura TLSInnerPlaintext serializada.

Los algoritmos AEAD toman como entrada una sola clave, un nonce, un texto sin formato y "datos adicionales" que se incluirán en la verificación de autenticación, como se describe en la Sección 2.1 de [RFC5116]. La clave es client_write_key o server_write_key, el nonce se deriva del número de secuencia y client_write_iv o server_write_iv (consulte la Sección 5.3), y la entrada de datos adicional es el encabezado del registro.

Es decir,

```
datos_adicionales = TLSCiphertext.opaque_type ||
                    TLSCiphertext.legacy_record_version || TLSCiphertext.length
```

La entrada de texto sin formato al algoritmo AEAD es la estructura TLSInnerPlaintext codificada. La derivación de claves de tráfico se define en la Sección 7.3.

La salida de AEAD consiste en la salida de texto cifrado de la operación de cifrado de AEAD. La longitud del texto sin formato es mayor que el TLSPlaintext.length correspondiente debido a la inclusión de TLSInnerPlaintext.type y cualquier relleno proporcionado por el remitente. La longitud de la salida AEAD generalmente será mayor que el texto sin formato, pero en una cantidad que varía con el algoritmo AEAD.

Machine Translated by Google

Dado que los cifrados pueden incorporar relleno, la cantidad de sobrecarga puede variar con diferentes longitudes de texto sin formato. Simbólicamente,

```
AEADEncrypted =
        AEAD-Encrypt (write_key, nonce, added_data, texto sin formato)
```

El campo encrypted_record de TLSCiphertext se establece en AEADEncrypted.

Para descifrar y verificar, el cifrado toma como entrada la clave, el nonce, los datos adicionales y el valor AEADEncrypted. El resultado es el texto sin formato o un error que indica que el descifrado falló. No hay una verificación de integridad separada. Simbólicamente,

```
texto sin formato del registro_cifrado =
        AEAD-Descifrar (peer_write_key, nonce,
                        datos_adicionales, AEADEncrypted)
```

Si el descifrado falla, el receptor DEBE terminar la conexión con una alerta "bad_record_mac".

Un algoritmo AEAD utilizado en TLS 1.3 NO DEBE producir una expansión superior a 255 octetos. Un punto final que recibe un registro de su par con TLSCiphertext.length mayor que 2^14 + 256 octetos DEBE terminar la conexión con una alerta "record_overflow". Este límite se deriva de la longitud máxima de TLSInnerPlaintext de 2^14 octetos + 1 octeto para ContentType + la expansión máxima de AEAD de 255 octetos.

5.3. Nonce por registro

Se mantiene un número de secuencia de 64 bits por separado para leer y escribir registros. El número de secuencia apropiado se incrementa en uno después de leer o escribir cada registro. Cada número de secuencia se establece en cero al comienzo de una conexión y cada vez que se cambia la clave; el primer registro transmitido bajo una clave de tráfico particular DEBE usar el número de secuencia 0.

Debido a que el tamaño de los números de secuencia es de 64 bits, no deberían ajustarse. Si una implementación de TLS necesitaría envolver un número de secuencia, DEBE volver a codificar (Sección 4.6.3) o terminar la conexión.

Machine Translated by Google

Cada algoritmo AEAD especificará un rango de longitudes posibles para el nonce por registro, desde N_MIN bytes hasta N_MAX bytes de entrada [RFC5116].
La longitud del nonce por registro de TLS (iv_length) se establece en el mayor de 8 bytes y N_MIN para el algoritmo AEAD (consulte [RFC5116], Sección 4). Un algoritmo AEAD en el que N_MAX es inferior a 8 bytes NO DEBE utilizarse con TLS. El nonce por registro para la construcción de AEAD se forma de la siguiente manera:

1. El número de secuencia de registro de 64 bits está codificado en byte de red order y rellenado a la izquierda con ceros hasta iv_length.

2. El número de secuencia rellenado se aplica XOR con client_write_iv o server_write_iv estáticos (según el rol).

La cantidad resultante (de longitud iv_longitud) se utiliza como nonce por registro.

Nota: Esta es una construcción diferente de la de TLS 1.2, que especificaba un nonce parcialmente explícito.

5.4. Registro de relleno

Todos los registros TLS cifrados se pueden rellenar para inflar el tamaño del TLSCiphertext. Esto permite que el remitente oculte el tamaño del tráfico de un observador.

Al generar un registro de texto cifrado TLSC, las implementaciones PUEDEN optar por rellenar. Un registro sin relleno es solo un registro con una longitud de relleno de cero. El relleno es una cadena de bytes con valor cero que se agrega al campo ContentType antes del cifrado. Las implementaciones DEBEN establecer los octetos de relleno en ceros antes del cifrado.

Los registros de datos de aplicación pueden contener un TLSInnerPlaintext.content de longitud cero si el remitente lo desea. Esto permite la generación de tráfico de cobertura de tamaño plausible en contextos donde la presencia o ausencia de actividad puede ser sensible. Las implementaciones NO DEBEN enviar registros de protocolo de enlace y alerta que tengan un TLSInnerPlaintext.content de longitud cero; si se recibe tal mensaje, la implementación receptora DEBE terminar la conexión con una alerta de "mensaje_inesperado".

Machine Translated by Google

El relleno enviado es verificado automáticamente por el mecanismo de protección de registros; tras el descifrado exitoso de un TLSCiphertext.encrypted_record, la implementación receptora escanea el campo desde el final hacia el principio hasta que encuentra un octeto distinto de cero. Este octeto distinto de cero es el tipo de contenido del mensaje. Se seleccionó este esquema de relleno porque permite el relleno de cualquier registro TLS cifrado en un tamaño arbitrario (desde cero hasta los límites de tamaño del registro TLS) sin introducir nuevos tipos de contenido. El diseño también aplica octetos de relleno de ceros, lo que permite una detección rápida de errores de relleno.

Las implementaciones DEBEN limitar su escaneo al texto claro devuelto por el descifrado AEAD. Si una implementación receptora no encuentra un octeto distinto de cero en el texto claro, DEBE terminar la conexión con una alerta de "mensaje_inesperado".

La presencia de relleno no cambia las limitaciones generales del tamaño del registro: el TLSInnerPlaintext codificado completo NO DEBE exceder los 2^14 + 1 octetos. Si se reduce la longitud máxima del fragmento, como, por ejemplo, por la extensión record_size_limit de [RFC8449], entonces el límite reducido se aplica al texto sin formato completo, incluido el tipo de contenido y el relleno.

Seleccionar una política de relleno que sugiera cuándo y cuánto rellenar es un tema complejo y está más allá del alcance de esta especificación. Si el protocolo de capa de aplicación sobre TLS tiene su propio relleno, puede ser preferible rellenar los registros TLS de datos de aplicación dentro de la capa de aplicación. Sin embargo, el relleno para registros cifrados de protocolo de enlace o alerta aún debe manejarse en la capa TLS. Los documentos posteriores pueden definir algoritmos de selección de relleno o definir un mecanismo de solicitud de política de relleno a través de extensiones TLS o algún otro medio.

5.5. Límites en el uso de claves

Existen límites criptográficos en la cantidad de texto sin formato que se puede cifrar de forma segura con un conjunto determinado de claves. [AEAD-LIMITS] proporciona un análisis de estos límites bajo el supuesto de que la primitiva subyacente (AES o ChaCha20) no tiene debilidades.

Las implementaciones DEBERÍAN realizar una actualización clave como se describe en la Sección 4.6.3 antes de alcanzar estos límites.

Para AES-GCM, se pueden cifrar hasta 2^24,5 registros de tamaño completo (alrededor de 24 millones) en una conexión determinada mientras se mantiene un margen de seguridad de aproximadamente 2^-57 para la seguridad del cifrado autenticado (AE). Para ChaCha20/Poly1305, el número de secuencia de registro terminaría antes de alcanzar el límite de seguridad.

Machine Translated by Google

6. Protocolo de alerta

TLS proporciona un tipo de contenido de alerta para indicar información de cierre y errores. Al igual que otros mensajes, los mensajes de alerta se cifran según lo especificado por el estado de conexión actual.

Los mensajes de alerta transmiten una descripción de la alerta y un campo heredado que transmitía el nivel de gravedad del mensaje en versiones anteriores de TLS. Las alertas se dividen en dos clases: alertas de cierre y alertas de error. En TLS 1.3, la gravedad está implícita en el tipo de alerta que se envía y el campo "nivel" se puede ignorar de forma segura. La alerta "close_notify" se utiliza para indicar el cierre ordenado de una dirección de la conexión. Al recibir dicha alerta, la implementación de TLS DEBERÍA indicar el final de los datos a la aplicación.

Las alertas de error indican el cierre abortado de la conexión (consulte la Sección 6.2). Al recibir una alerta de error, la implementación de TLS DEBE indicar un error a la aplicación y NO DEBE permitir que se envíen o reciban más datos en la conexión. Los servidores y clientes DEBEN olvidar los valores secretos y las claves establecidas en las conexiones fallidas, con la excepción de los PSK asociados con los tickets de sesión, que DEBERÍAN descartarse si es posible.

Todas las alertas enumeradas en la Sección 6.2 DEBEN enviarse con AlertLevel=fatal y DEBEN tratarse como alertas de error cuando se reciben independientemente del AlertLevel en el mensaje. Los tipos de alerta desconocidos DEBEN tratarse como alertas de error.

Nota: TLS define dos alertas genéricas (consulte la Sección 6) para usar en caso de falla al analizar un mensaje. Los pares que reciben un mensaje que no se puede analizar de acuerdo con la sintaxis (por ejemplo, tienen una longitud que se extiende más allá del límite del mensaje o contienen una longitud fuera de rango) DEBEN terminar la conexión con una alerta de "decode_error".

Pares que reciben un mensaje que es sintácticamente correcto pero semánticamente inválido (p. ej., un DHE compartido de p - 1, o una enumeración inválida) DEBE terminar la conexión con una alerta "illegal_parameter".

Machine Translated by Google


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

enum
        { cerrar_notificar(0),
         mensaje_inesperado(10),
         registro_malo_mac(20),
         desbordamiento_de_registro(22),
         error_de apretón de manos(40),
         certificado_incorrecto(42),
         certificado_no admitido(43),
         certificado_revocado(44),
         certificado_caducado(45),
         certificado_desconocido(46) ) ),
         extensión_faltante(109),
         extensión_no_soportada(110),
         nombre_no_reconocido(112),
         respuesta_estado_de_certificado_incorrecto(113),
         identidad_psk_desconocida(115),
         certificado_requerido(116),
         sin_protocolo_de_aplicación(120), (255)

} Descripción de alerta;

estructura {
        nivel de AlertLevel;
        AlertDescription descripción;
} Alerta;
```

Machine Translated by Google

6.1. Alertas de cierre

El cliente y el servidor deben compartir el conocimiento de que la conexión está finalizando para evitar un ataque de truncamiento.

close_notify: esta alerta notifica al destinatario que el remitente no enviará más mensajes en esta conexión. Cualquier dato recibido después de que se haya recibido una alerta de cierre DEBE ser ignorado.

user_canceled: esta alerta notifica al destinatario que el remitente está cancelando el protocolo de enlace por algún motivo no relacionado con una falla del protocolo. Si un usuario cancela una operación después de que se completa el protocolo de enlace, es más apropiado simplemente cerrar la conexión enviando un "close_notify". Esta alerta DEBE ir seguida de un "close_notify". Esta alerta generalmente tiene AlertLevel=warning.

Cualquiera de las partes PUEDE iniciar un cierre de su lado de escritura de la conexión mediante el envío de una alerta "close_notify". Cualquier dato recibido después de que se haya recibido una alerta de cierre DEBE ser ignorado. Si se recibe un cierre de nivel de transporte antes de un "close_notify", el receptor no puede saber que se han recibido todos los datos que se enviaron.

Cada parte DEBE enviar una alerta "close_notify" antes de cerrar su lado de escritura de la conexión, a menos que ya haya enviado alguna alerta de error.
Esto no tiene ningún efecto en su lado de lectura de la conexión.
Tenga en cuenta que este es un cambio con respecto a las versiones de TLS anteriores a TLS 1.3 en las que se requería que las implementaciones reaccionaran a un "close_notify" descartando las escrituras pendientes y enviando una alerta propia inmediata de "close_notify". Ese requisito anterior podría causar truncamiento en el lado de lectura. Ambas partes no necesitan esperar a recibir una alerta de "cerrar_notificación" antes de cerrar su lado de lectura de la conexión, aunque hacerlo podría presentar la posibilidad de truncamiento.

Si el protocolo de aplicación que usa TLS establece que cualquier dato puede transportarse a través del transporte subyacente después de que se cierra la conexión TLS, la implementación de TLS DEBE recibir una alerta "close_notify" antes de indicar el fin de los datos a la capa de aplicación. Ninguna parte de este estándar debe tomarse para dictar la manera en que un perfil de uso para TLS gestiona su transporte de datos, incluso cuando se abren o cierran las conexiones.

Nota: Se supone que cerrar el lado de escritura de una conexión entrega de manera confiable los datos pendientes antes de destruir el transporte.

Machine Translated by Google

6.2. Alertas de errores

El manejo de errores en TLS es muy simple. Cuando se detecta un error, la parte que lo detecta envía un mensaje a su par. Tras la transmisión o recepción de un mensaje de alerta fatal, ambas partes DEBEN cerrar inmediatamente la conexión.

Cada vez que una implementación encuentra una condición de error fatal, DEBERÍA enviar una alerta fatal adecuada y DEBE cerrar la conexión sin enviar ni recibir ningún dato adicional. En el resto de esta especificación, cuando las frases "terminar la conexión" y "abortar el protocolo de enlace" se utilizan sin una alerta específica, significa que la implementación DEBERÍA enviar la alerta indicada por las descripciones a continuación. Las frases "terminar la conexión con una alerta X" y "abortar el protocolo de enlace con una alerta X" significan que la implementación DEBE enviar la alerta X si envía cualquier alerta. Todas las alertas definidas a continuación en esta sección, así como todas las alertas desconocidas, se consideran universalmente fatales a partir de TLS 1.3 (consulte la Sección 6). La implementación DEBERÍA proporcionar una forma de facilitar el registro del envío y recepción de alertas.

Se definen las siguientes alertas de error:

mensaje_inesperado: un mensaje inapropiado (p. ej., el error mensaje de protocolo de enlace, datos de aplicación prematuros, etc.). Esta alerta nunca debe observarse en la comunicación entre implementaciones adecuadas.

bad_record_mac: esta alerta se devuelve si se recibe un registro que no se puede desproteger. Debido a que los algoritmos AEAD combinan descifrado y verificación, y también para evitar ataques de canal lateral, esta alerta se usa para todas las fallas de desprotección. Esta alerta nunca debe observarse en la comunicación entre implementaciones adecuadas, excepto cuando los mensajes se corrompieron en la red.

record_overflow: se recibió un registro TLSCiphertext que tenía una longitud superior a 2^14 + 256 bytes, o un registro descifrado a un registro TLSPlaintext con más de 2^14 bytes (o algún otro límite negociado). Esta alerta nunca debe observarse en la comunicación entre implementaciones adecuadas, excepto cuando los mensajes se corrompieron en la red.

handshake_failure: la recepción de un mensaje de alerta "handshake_failure" indica que el remitente no pudo negociar un conjunto aceptable de parámetros de seguridad dadas las opciones disponibles.

bad_certificate: un certificado estaba dañado, contenía firmas que no verificó correctamente, etc.

Machine Translated by Google

unsupported_certificate: un certificado era de un tipo no admitido.

certificate_revoked: un certificado fue revocado por su firmante.

certificate_expired: un certificado ha caducado o no está actualmente válido.

certificate_unknown: surgió algún otro problema (no especificado) al procesar el certificado, lo que lo hizo inaceptable.

ilegal_parameter: un campo en el protocolo de enlace era incorrecto o inconsistente con otros campos. Esta alerta se utiliza para errores que se ajustan a la sintaxis del protocolo formal pero que, por lo demás, son incorrectos.

unknown_ca: se recibió una cadena de certificados válida o una cadena parcial, pero el certificado no se aceptó porque no se pudo ubicar el certificado de CA o no se pudo hacer coincidir con un ancla de confianza conocida.

access_denied: se recibió un certificado válido o PSK, pero cuando se aplicó el control de acceso, el remitente decidió no continuar con la negociación.

decode_error: no se pudo decodificar un mensaje porque algún campo estaba fuera del rango especificado o la longitud del mensaje era incorrecta. Esta alerta se utiliza para errores en los que el mensaje no se ajusta a la sintaxis del protocolo formal. Esta alerta nunca debe observarse en la comunicación entre implementaciones adecuadas, excepto cuando los mensajes se corrompieron en la red.

decrypt_error: un apretón de manos (no capa de registro) criptográfico la operación falló, incluida la imposibilidad de verificar correctamente una firma o validar un mensaje Finalizado o un archivador PSK.

protocol_version: La versión del protocolo que el par ha intentado negociar se reconoce pero no se admite (consulte el Apéndice D).

insuficiente_seguridad: Devuelto en lugar de "handshake_failure" cuando una negociación ha fallado específicamente porque el servidor requiere parámetros más seguros que los soportados por el cliente.

internal_error: Un error interno no relacionado con el par o el la corrección del protocolo (como una falla en la asignación de memoria) hace que sea imposible continuar.

inapproup_fallback: enviado por un servidor en respuesta a un reintento de conexión no válido de un cliente (ver [RFC7507]).

Machine Translated by Google

perdido_extensión: Enviado por puntos finales que reciben un apretón de manos mensaje que no contiene una extensión que es obligatorio enviar para la versión TLS ofrecida u otros parámetros negociados.

unsupported_extension: Enviado por puntos finales que reciben cualquier mensaje de reconocimiento que contiene una extensión que se sabe que está prohibida para su inclusión en el mensaje de reconocimiento dado, o que incluye cualquier extensión en un ServerHello o Certificado que no se ofreció primero en el ClientHello o CertificateRequest correspondiente.

unrecognized_name: Enviado por servidores cuando no existe un servidor identificado por el nombre proporcionado por el cliente a través de la extensión "server_name" (ver [RFC6066]).

bad_certificate_status_response: enviado por los clientes cuando el servidor proporciona una respuesta OCSP inválida o inaceptable a través de la extensión "status_request" (consulte [RFC6066]).

unknown_psk_identity: enviado por los servidores cuando se desea establecer la clave PSK pero el cliente no proporciona una identidad PSK aceptable.
Enviar esta alerta es OPCIONAL; los servidores PUEDEN optar por enviar una alerta "decrypt_error" para indicar simplemente una identidad de PSK no válida.

certificate_required: Enviado por los servidores cuando se solicita un certificado de cliente. deseado pero ninguno fue proporcionado por el cliente.

no_application_protocol: Enviado por servidores cuando un cliente La extensión "application_layer_protocol_negotiation" solo anuncia protocolos que el servidor no admite (consulte [RFC7301]).

La IANA asigna nuevos valores de Alerta como se describe en la Sección 11.

7. Cálculos criptográficos

El protocolo de enlace TLS establece uno o más secretos de entrada que se combinan para crear el material de clave de trabajo real, como se detalla a continuación. El proceso de derivación de claves incorpora tanto los secretos de entrada como la transcripción del protocolo de enlace. Tenga en cuenta que debido a que la transcripción del protocolo de enlace incluye los valores aleatorios de los mensajes de saludo, cualquier protocolo de enlace dado tendrá diferentes secretos de tráfico, incluso si se usan los mismos secretos de entrada, como es el caso cuando se usa el mismo PSK para varias conexiones.

Machine Translated by Google

7.1. Calendario clave

El proceso de obtención de claves hace uso de HKDF-Extract y Funciones HKDF-Expand tal como se definen para HKDF [RFC5869], así como las funciones definidas a continuación:

```
HKDF-Expand-Label (Secreto, Etiqueta, Contexto, Longitud) =
        HKDF-Expand (Secreto, HkdfLabel, Longitud)

Donde HkdfLabel se especifica como:

estructura
        { uint16 longitud = longitud; etiqueta
        opaca<7..255> = "tls13"                    + Etiqueta;
        contexto opaco<0..255> = Contexto; } HkdfEtiqueta;


Derive-Secret(Secreto, Etiqueta, Mensajes) =
        HKDF-Expand-Label (Secreto, Etiqueta,
                                   Transcripción-Hash(Mensajes), Hash.length)
```

La función Hash utilizada por Transcript-Hash y HKDF es el algoritmo hash del conjunto de cifrado.
Hash.length es su longitud de salida en bytes.
Mensajes es la concatenación de los mensajes de protocolo de enlace indicados, incluidos el tipo de mensaje de protocolo de enlace y los campos de longitud, pero sin incluir los encabezados de la capa de registro. Tenga en cuenta que, en algunos casos, se pasa un Contexto de longitud cero (indicado por "") a HKDF-Expand-Label.
Las etiquetas especificadas en este documento son todas cadenas ASCII y no incluyen un byte NUL final.

Nota: con las funciones hash comunes, cualquier etiqueta de más de 12 caracteres requiere una iteración adicional de la función hash para calcular.
Todas las etiquetas de esta especificación han sido elegidas para ajustarse a este límite.

Machine Translated by Google

Las claves se derivan de dos secretos de entrada mediante las funciones HKDF-Extract y Derive-Secret. El patrón general para agregar un nuevo secreto es usar HKDF-Extract con Salt como el estado secreto actual y Input Keying Material (IKM) como el nuevo secreto que se agregará.

En esta versión de TLS 1.3, los dos secretos de entrada son:

- PSK (una clave precompartida establecida externamente o derivada del valor resumption_master_secret de una conexión anterior)

- (EC) Y secreto compartido (Sección 7.4)

Esto produce un programa completo de derivación de claves que se muestra en el siguiente diagrama. En este diagrama, se aplican las siguientes convenciones de formato:

- HKDF-Extract se dibuja tomando el argumento Salt de la parte superior y el argumento IKM de la izquierda, con su salida en la parte inferior y el nombre de la salida a la derecha.

- El argumento Secreto de Derive-Secret se indica mediante la flecha entrante. Por ejemplo, Early Secret es el secreto para generar client_early_traffic_secret.

- "0" indica una cadena de bytes Hash.length establecida en cero.

Machine Translated by Google


```
                    0
                    |
                    v
PSK -> Extracto de HKDF = Secreto inicial
                    |
                    +-----> Derive-Secret(., "carpeta externa" | "carpeta res", "")
                    | = carpeta_clave
                    |
                    +-----> Derive-Secret(., "tráfico ce", ClientHello)
                    | = client_early_traffic_secret
                    |
                    +-----> Derive-Secret(., "e exp master", ClientHello)
                    | = Early_exporter_master_secret
                    v
        Derive-Secret(., "derivado", "")
                    |
                    v
 (EC) Y -> Extracto de HKDF = Apretón de manos secreto
                    |
                    +-----> Derive-Secret(., "c hs tráfico",
                                          ClienteHola...ServidorHola)
                                          = cliente_apretón de manos_tráfico_secreto
                    |
                    +-----> Derive-Secret(., "s hs tráfico",
                                          ClienteHola...ServidorHola)
                                          = server_handshake_traffic_secret
                    v
        Derive-Secret(., "derivado", "")
                    |
                    v
0 -> Extracto de HKDF = Secreto maestro
                    |
                    +-----> Derive-Secret(., "tráfico máximo",
                    | ClientHello... servidor terminado)
                    | = client_application_traffic_secret_0
                    |
                    +-----> Derive-Secret(., "tráfico sap",
                    | ClientHello... servidor terminado)
                    | = server_application_traffic_secret_0
                    |
                    +-----> Derive-Secret(., "exp maestro",
                    | ClientHello... servidor terminado)
                    | = exportador_maestro_secreto
                    |
                    +-----> Derive-Secret(., "maestro res",
                                          ClienteHola... cliente Finalizado)
                                          = reanudación_maestro_secreto
```

Machine Translated by Google

El patrón general aquí es que los secretos que se muestran en el lado izquierdo del diagrama son solo entropía sin contexto, mientras que los secretos en el lado derecho incluyen contexto de protocolo de enlace y, por lo tanto, se pueden usar para derivar claves de trabajo sin contexto adicional. Tenga en cuenta que las diferentes llamadas a Derive-Secret pueden tomar diferentes argumentos de Mensajes, incluso con el mismo secreto. En un intercambio 0-RTT, Derive-Secret se llama con cuatro transcripciones distintas; en un intercambio de solo 1-RTT, se llama con tres transcripciones distintas.

Si un secreto dado no está disponible, entonces se usa el valor 0 que consiste en una cadena de bytes Hash.length establecidos en ceros. Tenga en cuenta que esto no significa saltarse rondas, por lo que si PSK no está en uso, Early Secret seguirá siendo HKDF-Extract(0, 0). Para el cálculo de binder_key, la etiqueta es "ext binder" para los PSK externos (aquellos provistos fuera de TLS) y "res binder" para los PSK de reanudación (aquellos provistos como el secreto maestro de reanudación de un protocolo de enlace anterior). Las diferentes etiquetas impiden la sustitución de un tipo de PSK por otro.

Existen varios valores potenciales de Early Secret, según el PSK que seleccione el servidor en última instancia. El cliente deberá calcular uno para cada PSK potencial; si no se selecciona PSK, entonces deberá calcular el Early Secret correspondiente al PSK cero.

Una vez que se han calculado todos los valores que se van a derivar de un secreto dado, ese secreto DEBE borrarse.

7.2. Actualización de secretos de tráfico

Una vez que se completa el protocolo de enlace, cualquiera de los lados puede actualizar sus claves de tráfico de envío utilizando el mensaje de protocolo de enlace KeyUpdate definido en la Sección 4.6.3. La siguiente generación de claves de tráfico se calcula generando client_/server_application_traffic_secret_N+1 a partir de client_/ server_application_traffic_secret_N como se describe en esta sección y luego derivando nuevamente las claves de tráfico como se describe en la Sección 7.3.

El application_traffic_secret de próxima generación se calcula como:

```
application_traffic_secret_N+1 =
        HKDF-Expand-Label(application_traffic_secret_N,
                                        "actualización de tráfico", "", Hash.length)
```

Una vez que se han calculado client_/server_application_traffic_secret_N+1 y sus claves de tráfico asociadas, las implementaciones DEBERÍAN eliminar client_/server_application_traffic_secret_N y sus claves de tráfico asociadas.

Machine Translated by Google

7.3. Cálculo de clave de tráfico

El material de clave de tráfico se genera a partir de la siguiente entrada valores:

- Un valor secreto

- Un valor de propósito que indica el valor específico que se está generando.

- La longitud de la clave que se genera

El material de clave de tráfico se genera a partir de un secreto de tráfico de entrada valor usando:

```
[remitente]_write_key = HKDF-Expand-Label(Secret, "key", "", key_length)
[remitente]_write_iv = HKDF-Expand-Label(Secreto, "iv", "", iv_longitud)
```

[remitente] denota el lado emisor. El valor de Secret para cada tipo de registro se muestra en la siguiente tabla.

Tipo de registro	Secreto
Aplicación 0-RTT	client_early_traffic_secret
apretón de manos	[remitente]_handshake_traffic_secret
Datos de la aplicación	[remitente]_application_traffic_secret_N

Todo el material de clave de tráfico se vuelve a calcular cada vez que el subyacente Cambios secretos (p. ej., al cambiar del apretón de manos a Aplicación Claves de datos o tras una actualización de clave).

7.4. (EC) Y cálculo de secreto compartido

7.4.1. Campo finito Diffie-Hellman

Para grupos de campos finitos, un Diffie-Hellman convencional [DH76] se realiza el cómputo. La clave negociada (Z) se convierte en una cadena de bytes mediante codificación en formato big-endian y relleno a la izquierda con ceros hasta el tamaño de la prima. Esta cadena de bytes se utiliza como la compartida secret en el programa clave como se especifica anteriormente.

Tenga en cuenta que esta construcción difiere de las versiones anteriores de TLS que eliminó los ceros iniciales.

Machine Translated by Google

7.4.2. Curva Elíptica Diffie-Hellman

Para secp256r1, secp384r1 y secp521r1, los cálculos de ECDH (incluida la generación de parámetros y claves, así como el cálculo de secreto compartido) se realizan de acuerdo con [IEEE1363] utilizando el esquema ECKAS-DH1 con el mapa de identidad como la función de derivación de clave (KDF), de modo que el secreto compartido es la coordenada x del punto de la curva elíptica del secreto compartido ECDH representado como una cadena de octetos.

Tenga en cuenta que esta cadena de octetos ("Z" en la terminología IEEE 1363) como salida de FE2OSP (la primitiva de conversión de elemento de campo a cadena de octetos) tiene una longitud constante para cualquier campo dado; los ceros iniciales que se encuentran en esta cadena de octetos NO DEBEN truncarse.

(Tenga en cuenta que este uso de la identidad KDF es un tecnicismo. La imagen completa es que ECDH se emplea con un KDF no trivial porque TLS no usa directamente este secreto para otra cosa que no sea calcular otros secretos).

Para X25519 y X448, los cálculos ECDH son los siguientes:

- La clave pública para poner en KeyShareEntry.key_exchange
  La estructura es el resultado de aplicar la función de multiplicación escalar ECDH a la clave secreta de la longitud adecuada (en la entrada escalar) y el punto de base público estándar (en la entrada del punto de coordenadas u).

- El secreto compartido ECDH es el resultado de aplicar el escalar ECDH
  función de multiplicación a la clave secreta (en la entrada escalar) y la clave pública del par (en la entrada del punto de coordenadas u). La salida se utiliza sin procesar, sin procesamiento.

Para estas curvas, las implementaciones DEBERÍAN utilizar el enfoque especificado en [RFC7748] para calcular el secreto compartido de Diffie-Hellman.
Las implementaciones DEBEN verificar si el secreto compartido Diffie-Hellman calculado es el valor de todos cero y abortar si es así, como se describe en la Sección 6 de [RFC7748]. Si los implementadores utilizan una implementación alternativa de estas curvas elípticas, DEBERÍAN realizar las comprobaciones adicionales especificadas en la Sección 7 de [RFC7748].

Machine Translated by Google

7.5. Exportadores

[RFC5705] define exportadores de material de claves para TLS en términos de la función pseudoaleatoria (PRF) de TLS. Este documento reemplaza el PRF con HKDF, por lo que requiere una nueva construcción. La interfaz del exportador sigue siendo la misma.

El valor del exportador se computa como:

```
TLS-Exportador (etiqueta, valor_contexto, longitud_clave) =
        HKDF-Expand-Label(Derive-Secret(Secret, label, ""), "exportador", Hash(context_value),
                                                key_length)
```

Donde Secret es early_exporter_master_secret o exporter_master_secret. Las implementaciones DEBEN utilizar exporter_master_secret a menos que la aplicación lo especifique explícitamente. El early_exporter_master_secret se define para su uso en configuraciones donde se necesita un exportador para datos 0-RTT. Se RECOMIENDA una interfaz separada para el primer exportador; esto evita que el usuario del exportador use accidentalmente un exportador temprano cuando desea uno regular o viceversa.

Si no se proporciona ningún contexto, context_value tiene una longitud cero.
En consecuencia, no proporcionar contexto calcula el mismo valor que proporcionar un contexto vacío. Este es un cambio con respecto a las versiones anteriores de TLS donde un contexto vacío producía una salida diferente a la de un contexto ausente. A partir de la publicación de este documento, no se utiliza ninguna etiqueta de exportador asignado con y sin contexto. Las especificaciones futuras NO DEBEN definir un uso de exportadores que permitan tanto un contexto vacío como ningún contexto con la misma etiqueta. Los nuevos usos de los exportadores DEBERÍAN proporcionar un contexto en todos los cálculos del exportador, aunque el valor podría estar vacío.

Los requisitos para el formato de las etiquetas del exportador se definen en la Sección 4 de [RFC5705].

Machine Translated by Google

8. 0-RTT y Anti-Repetición

Como se indica en la Sección 2.3 y el Apéndice E.5, TLS no proporciona protecciones de reproducción inherentes para datos 0-RTT. Hay dos amenazas potenciales de las que preocuparse:

- Atacantes de red que montan un ataque de repetición simplemente duplicando un vuelo de datos 0-RTT.

- Los atacantes de red que se aprovechan del comportamiento de reintento del cliente para hacer arreglos para que el servidor reciba varias copias de un mensaje de aplicación. Esta amenaza ya existe hasta cierto punto porque los clientes que valoran la solidez responden a los errores de la red intentando reintentar las solicitudes. Sin embargo, 0-RTT agrega una dimensión adicional para cualquier sistema de servidor que no mantenga un estado de servidor globalmente consistente. Específicamente, si un sistema de servidor tiene varias zonas donde los boletos de la zona A no se aceptarán en la zona B, entonces un atacante puede duplicar un ClientHello y los primeros datos destinados a A tanto para A como para B. En A, los datos se aceptarán en 0-RTT, pero en B el servidor rechazará los datos 0-RTT y, en su lugar, forzará un protocolo de enlace completo. Si el atacante bloquea el ServerHello de A, entonces el cliente completará el protocolo de enlace con B y probablemente volverá a intentar la solicitud, lo que provocará la duplicación en el sistema del servidor como un todo.

La primera clase de ataque se puede prevenir compartiendo el estado para garantizar que los datos 0-RTT se acepten como máximo una vez. Los servidores DEBERÍAN proporcionar ese nivel de seguridad de reproducción mediante la implementación de uno de los métodos descritos en esta sección o por medios equivalentes. Sin embargo, se entiende que, debido a preocupaciones operativas, no todos los despliegues mantendrán el estado en ese nivel. Por lo tanto, en el funcionamiento normal, los clientes no sabrán cuál de estos mecanismos, si es que alguno, implementan realmente los servidores y, por lo tanto, DEBEN enviar solo los primeros datos que consideren seguros para su reproducción.

Además de los efectos directos de las repeticiones, existe una clase de ataques en los que incluso las operaciones normalmente consideradas idempotentes podrían ser explotadas por un gran número de repeticiones (ataques de sincronización, agotamiento del límite de recursos y otros, como se describe en el Apéndice E.5). Se pueden mitigar asegurándose de que cada carga útil de 0-RTT se pueda reproducir solo una cantidad limitada de veces. El servidor DEBE asegurarse de que cualquier instancia del mismo (ya sea una máquina, un subproceso o cualquier otra entidad dentro de la infraestructura de servicio relevante) acepte 0-RTT para el mismo protocolo de enlace 0-RTT como máximo una vez; esto limita la cantidad de reproducciones a la cantidad de instancias de servidor en la implementación. Dicha garantía se puede lograr mediante el registro local de datos de ClientHellos recibidos recientemente y el rechazo de repeticiones, o mediante cualquier otro método que

Machine Translated by Google

proporciona la misma garantía o una más fuerte. La garantía de "como máximo una vez por instancia de servidor" es un requisito mínimo; los servidores DEBERÍAN limitar más las repeticiones de 0-RTT cuando sea factible.

La segunda clase de ataque no se puede prevenir en la capa TLS y DEBE ser tratada por cualquier aplicación. Tenga en cuenta que cualquier aplicación cuyos clientes implementen algún tipo de comportamiento de reintento ya necesita implementar algún tipo de defensa anti-reproducción.

8.1. Entradas de un solo uso

La forma más simple de defensa contra la reproducción es que el servidor solo permita que cada ticket de sesión se use una vez. Por ejemplo, el servidor puede mantener una base de datos de todos los boletos válidos pendientes, eliminando cada boleto de la base de datos a medida que se usa. Si se proporciona un ticket desconocido, el servidor recurriría a un protocolo de enlace completo.

Si los boletos no son autónomos, sino claves de base de datos, y los PSK correspondientes se eliminan al usarlos, las conexiones establecidas mediante PSK disfrutan de confidencialidad hacia adelante. Esto mejora la seguridad para todos los datos 0-RTT y el uso de PSK cuando PSK se usa sin (EC)DHE.

Debido a que este mecanismo requiere compartir la base de datos de la sesión entre los nodos del servidor en entornos con múltiples servidores distribuidos, puede ser difícil lograr altas tasas de conexiones PSK 0-RTT exitosas en comparación con los tickets autocifrados. A diferencia de las bases de datos de sesión, los vales de sesión pueden realizar con éxito el establecimiento de sesión basado en PSK incluso sin un almacenamiento constante, aunque cuando se permite 0-RTT, aún requieren un almacenamiento constante para evitar la reproducción de datos 0-RTT, como se detalla en la siguiente sección.

8.2. Grabación de saludo del cliente

Una forma alternativa de anti-reproducción es registrar un valor único derivado de ClientHello (generalmente el valor aleatorio o la carpeta PSK) y rechazar los duplicados. Grabar todos los ClientHellos hace que el estado crezca sin límites, pero un servidor puede registrar ClientHellos dentro de un período de tiempo determinado y usar "obfuscated_ticket_age" para asegurarse de que los boletos no se reutilicen fuera de ese período.

Para implementar esto, cuando se recibe un ClientHello, el servidor primero verifica el enlazador de PSK como se describe en la Sección 4.2.11.

Luego calcula el tiempo de llegada esperado como se describe en la siguiente sección y rechaza 0-RTT si está fuera de la ventana de grabación, recurriendo al protocolo de enlace 1-RTT.

Machine Translated by Google

Si la hora_de_llegada_esperada está en la ventana, entonces el servidor verifica si ha registrado un ClientHello coincidente. Si se encuentra uno, aborta el protocolo de enlace con una alerta de "parámetro_illegal" o acepta el PSK pero rechaza 0-RTT. Si no se encuentra ClientHello coincidente, entonces acepta 0-RTT y luego almacena ClientHello mientras la hora de llegada esperada esté dentro de la ventana. Los servidores PUEDEN también implementar almacenes de datos con falsos positivos, como los filtros Bloom, en cuyo caso DEBEN responder a la reproducción aparente rechazando 0-RTT pero NO DEBEN abortar el protocolo de enlace.

El servidor DEBE derivar la clave de almacenamiento solo de las secciones validadas de ClientHello. Si ClientHello contiene varias identidades PSK, un atacante puede crear múltiples ClientHellos con diferentes valores de enlace para la identidad menos preferida suponiendo que el servidor no la verificará (como se recomienda en la Sección 4.2.11). Es decir, si el cliente envía PSK A y B pero el servidor prefiere A, entonces el atacante puede cambiar la carpeta de B sin afectar la carpeta de A. Si la carpeta de B es parte de la clave de almacenamiento, este ClientHello no aparecerá. como un duplicado, lo que hará que se acepte ClientHello y puede causar efectos secundarios como la contaminación de la caché de reproducción, aunque los datos 0-RTT no se podrán descifrar porque usarán claves diferentes. Si se utiliza el archivador validado o ClientHello.random como clave de almacenamiento, este ataque no es posible.

Debido a que este mecanismo no requiere almacenar todos los tickets pendientes, puede ser más fácil de implementar en sistemas distribuidos con altas tasas de reanudación y 0-RTT, a costa de una defensa antirrepetición potencialmente más débil debido a la dificultad de almacenar y recuperar de manera confiable el recibió mensajes ClientHello. En muchos de estos sistemas, no es práctico tener un almacenamiento coherente a nivel mundial de todos los ClientHello recibidos. En este caso, la mejor protección contra la reproducción se obtiene al tener una sola zona de almacenamiento autorizada para un boleto determinado y rechazar 0-RTT para ese boleto en cualquier otra zona.

Este enfoque evita la reproducción simple por parte del atacante porque solo una zona aceptará datos 0-RTT. Un diseño más débil es implementar almacenamiento separado para cada zona pero permitir 0-RTT en cualquier zona. Este enfoque limita el número de repeticiones a una vez por zona. La duplicación de mensajes de aplicaciones, por supuesto, sigue siendo posible con cualquiera de los dos diseños.

Cuando las implementaciones recién comienzan, DEBERÍAN rechazar 0-RTT siempre que cualquier parte de su ventana de grabación se superponga al tiempo de inicio. De lo contrario, corren el riesgo de aceptar repeticiones enviadas originalmente durante ese período.

Machine Translated by Google

Nota: si el reloj del cliente funciona mucho más rápido que el del servidor, es posible que se reciba un ClientHello que esté fuera de la ventana en el futuro, en cuyo caso podría aceptarse para 1-RTT, lo que provocaría un reintento del cliente y luego sería aceptable. para 0-RTT. Esta es otra variante de la segunda forma de ataque descrita en la Sección 8.

8.3. Controles de frescura

Debido a que ClientHello indica la hora a la que el cliente lo envió, es posible determinar de manera eficiente si un ClientHello probablemente se envió hace poco tiempo y solo acepta 0-RTT para dicho ClientHello, de lo contrario recurre a un apretón de manos de 1-RTT. Esto es necesario para el mecanismo de almacenamiento de ClientHello descrito en la Sección 8.2 porque, de lo contrario, el servidor necesita almacenar una cantidad ilimitada de ClientHellos, y es una optimización útil para los boletos autónomos de un solo uso porque permite el rechazo eficiente de ClientHellos que no se pueden usar para 0-RTT.

Para implementar este mecanismo, un servidor necesita almacenar la hora en que el servidor generó el ticket de sesión, compensada por una estimación del tiempo de ida y vuelta entre el cliente y el servidor. Es decir,

tiempo_de_creación_ajustado = tiempo_de_creación + RTT_estimado

Este valor se puede codificar en el ticket, evitando así la necesidad de mantener el estado de cada ticket pendiente. El servidor puede determinar la vista del cliente sobre la antigüedad del ticket restando el valor "ticket_age_add" del ticket del parámetro "ofuscated_ticket_age" en la extensión "pre_shared_key" del cliente. El servidor puede determinar la hora_de_llegada_esperada de ClientHello como:

hora_de_llegada_prevista = hora_de_creación_ajustada + edad_del_ticket_de_clientes

Cuando se recibe un ClientHello nuevo, la hora_de_llegada_esperada se compara con la hora actual del reloj de pared del servidor y, si difieren en más de una cierta cantidad, se rechaza 0-RTT, aunque se puede permitir que se complete el protocolo de enlace 1-RTT.

Machine Translated by Google

Hay varias fuentes potenciales de error que pueden causar discrepancias entre la hora_de_llegada_esperada y la hora medida.
Es probable que las variaciones en las frecuencias de reloj del cliente y del servidor sean mínimas, aunque es posible que los tiempos absolutos se desvíen en valores grandes.
Los retrasos en la propagación de la red son las causas más probables de una discrepancia en los valores legítimos del tiempo transcurrido. Los mensajes NewSessionTicket y ClientHello pueden retransmitirse y, por lo tanto, retrasarse, lo que TCP puede ocultar. Para los clientes en Internet, esto implica ventanas del orden de diez segundos para dar cuenta de errores en los relojes y variaciones en las medidas; otros escenarios de implementación pueden tener necesidades diferentes. Las distribuciones de sesgo de reloj no son simétricas, por lo que la compensación óptima puede implicar un rango asimétrico de valores de desajuste permisibles.

Tenga en cuenta que la verificación de actualización por sí sola no es suficiente para evitar las repeticiones porque no las detecta durante la ventana de error que, según el ancho de banda y la capacidad del sistema, podría incluir miles de millones de repeticiones en entornos reales. Además, esta verificación de actualización solo se realiza en el momento en que se recibe ClientHello y no cuando se reciben los primeros registros posteriores de datos de la aplicación. Una vez que se aceptan los primeros datos, los registros pueden continuar transmitiéndose al servidor durante un período de tiempo más prolongado.

9. Requisitos de cumplimiento

9.1. Conjuntos de cifrado obligatorios para implementar

A falta de una norma de perfil de aplicación que especifique lo contrario:

Una aplicación compatible con TLS DEBE implementar el conjunto de cifrado TLS_AES_128_GCM_SHA256 [GCM] y DEBERÍA implementar los conjuntos de cifrado TLS_AES_256_GCM_SHA384 [GCM] y TLS_CHACHA20_POLY1305_SHA256 [RFC8439] (consulte el Apéndice B.4).

Una aplicación compatible con TLS DEBE admitir firmas digitales con rsa_pkcs1_sha256 (para certificados), rsa_pss_rsae_sha256 (para CertificateVerify y certificados) y ecdsa_secp256r1_sha256. Una aplicación compatible con TLS DEBE admitir el intercambio de claves con secp256r1 (NIST P-256) y DEBERÍA admitir el intercambio de claves con X25519 [RFC7748].

Machine Translated by Google

9.2. Extensiones Obligatorias de Implementar

En ausencia de un estándar de perfil de aplicación que especifique lo contrario, una aplicación compatible con TLS DEBE implementar lo siguiente Extensiones TLS:

- Versiones compatibles ("supported_versions"; Sección 4.2.1)

- Cookie ("cookie"; Sección 4.2.2)

- Algoritmos de firma ("signature_algorithms"; Sección 4.2.3)

- Certificado de Algoritmos de Firma ("signature_algorithms_cert"; Apartado 4.2.3)

- Grupos negociados ("supported_groups"; Sección 4.2.7)

- Clave compartida ("key_share"; Sección 4.2.8)

- Indicación del nombre del servidor ("server_name"; Sección 3 de [RFC6066])

Todas las implementaciones DEBEN enviar y utilizar estas extensiones cuando ofrezcan funciones aplicables:

- Se REQUIERE "supported_versions" para todos los mensajes ClientHello, ServerHello y HelloRetryRequest.

- Se REQUIERE "signature_algorithms" para la autenticación de certificados.

- "supported_groups" es OBLIGATORIO para los mensajes ClientHello usando DHE o intercambio de claves ECDHE.

- Se REQUIERE "key_share" para el intercambio de claves DHE o ECDHE.

- Se REQUIERE "pre_shared_key" para el acuerdo de clave de PSK.

- Se REQUIERE "psk_key_exchange_modes" para el acuerdo de clave de PSK.

Machine Translated by Google

Se considera que un cliente intenta negociar utilizando esta especificación si ClientHello contiene una extensión "supported_versions" con 0x0304 en su cuerpo. Dicho mensaje ClientHello DEBE cumplir con los siguientes requisitos:

- Si no contiene una extensión "pre_shared_key", DEBE contener una extensión "signature_algorithms" y una extensión "supported_groups".

- Si contiene una extensión "supported_groups", también DEBE contener una extensión "key_share" y viceversa. Se permite un vector KeyShare.client_shares vacío.

Los servidores que reciben un ClientHello que no cumple con estos requisitos DEBEN cancelar el protocolo de enlace con una alerta de "extensión faltante".

Además, todas las implementaciones DEBEN admitir el uso de la extensión "server_name" con aplicaciones capaces de usarla.
Los servidores PUEDEN requerir que los clientes envíen una extensión válida de "nombre_servidor". Los servidores que requieren esta extensión DEBERÍAN responder a un ClientHello que carece de una extensión "server_name" al finalizar la conexión con una alerta de "missing_extension".

9.3. Invariantes de protocolo

En esta sección, se describen las invariantes que DEBEN seguir los puntos finales y los middleboxes de TLS. También se aplica a versiones anteriores de TLS.

TLS está diseñado para ser extensible de forma segura y compatible. Los clientes o servidores más nuevos, cuando se comunican con pares más nuevos, deben negociar los parámetros comunes preferidos. El protocolo de enlace TLS proporciona protección de degradación: los middleboxes que pasan tráfico entre un cliente más nuevo y un servidor más nuevo sin terminar TLS no deberían poder influir en el protocolo de enlace (consulte el Apéndice E.1). Al mismo tiempo, las implementaciones se actualizan a diferentes velocidades, por lo que un cliente o servidor más nuevo PUEDE continuar admitiendo parámetros más antiguos, lo que le permitiría interoperar con puntos finales más antiguos.

Machine Translated by Google

Para que esto funcione, las implementaciones DEBEN manejar correctamente los campos extensibles:

- Un cliente que envía un ClientHello DEBE admitir todos los parámetros anunciado en el mismo. De lo contrario, es posible que el servidor no pueda interoperar al seleccionar uno de esos parámetros.

- Un servidor que recibe un ClientHello DEBE ignorar correctamente todos conjuntos de cifrado no reconocidos, extensiones y otros parámetros. De lo contrario, es posible que no pueda interoperar con clientes más nuevos. En TLS 1.3, un cliente que recibe un CertificateRequest o NewSessionTicket también DEBE ignorar todas las extensiones no reconocidas.

- Un middlebox que termina una conexión TLS DEBE comportarse como un servidor TLS compatible (para el cliente original), lo que incluye tener un certificado que el cliente esté dispuesto a aceptar, y también como cliente TLS compatible (para el servidor original), incluida la verificación del certificado del servidor original. En particular, DEBE generar su propio ClientHello que contenga solo los parámetros que comprende, y DEBE generar un nuevo valor aleatorio de ServerHello, en lugar de reenviar el valor del punto final.

  Tenga en cuenta que los requisitos de protocolo y el análisis de seguridad de TLS solo se aplican a las dos conexiones por separado. La implementación segura de un terminador TLS requiere consideraciones de seguridad adicionales que están más allá del alcance de este documento.

- Un middlebox que reenvía los parámetros ClientHello que no entender NO DEBE procesar ningún mensaje más allá de ClientHello. DEBE reenviar todo el tráfico subsiguiente sin modificar. De lo contrario, es posible que no pueda interoperar con clientes y servidores más nuevos.

  Los ClientHellos reenviados pueden contener anuncios de funciones no compatibles con el middlebox, por lo que la respuesta puede incluir futuras adiciones de TLS que el middlebox no reconoce. Estas adiciones PUEDEN cambiar arbitrariamente cualquier mensaje más allá de ClientHello. En particular, los valores enviados en ServerHello pueden cambiar, el formato de ServerHello puede cambiar y el formato de TLSCiphertext puede cambiar.

El diseño de TLS 1.3 estaba limitado por cajas intermedias de TLS no compatibles ampliamente implementadas (consulte el Apéndice D.4); sin embargo, no relaja las invariantes. Esos middleboxes continúan sin cumplir.

Machine Translated by Google

10. Consideraciones de seguridad

Los problemas de seguridad se tratan a lo largo de este memorando, especialmente en los Apéndices C, D y E.

11. Consideraciones de la IANA

Este documento utiliza varios registros que se crearon originalmente en [RFC4346] y se actualizaron en [RFC8447]. IANA los ha actualizado para hacer referencia a este documento. Los registros y sus políticas de asignación se encuentran a continuación:

- Registro TLS Cipher Suites: los valores con el primer byte en el rango 0-254 (decimal) se asignan a través de Specification Required [RFC8126]. Los valores con el primer byte 255 (decimal) están reservados para uso privado [RFC8126].

  IANA ha agregado los conjuntos de cifrado enumerados en el Apéndice B.4 al registro. Las columnas "Valor" y "Descripción" se toman de la tabla. Las columnas "DTLS-OK" y "Recomendado" están marcadas como "Y" para cada nuevo conjunto de cifrado.

- Registro TLS ContentType: los valores futuros se asignan a través de Standards Action [RFC8126].

- Registro de alertas TLS: los valores futuros se asignan a través de Standards Action [RFC8126]. IANA ha llenado este registro con los valores del Apéndice B.2. La columna "DTLS-OK" está marcada como "Y" para todos estos valores. Los valores marcados como "_RESERVADOS" tienen comentarios que describen su uso anterior.

- Registro TLS HandshakeType: los valores futuros se asignan a través de Acción de estándares [RFC8126]. IANA actualizó este registro para cambiar el nombre del elemento 4 de "NewSessionTicket" a "new_session_ticket" y completó este registro con los valores del Apéndice B.3. La columna "DTLS-OK" está marcada como "Y" para todos estos valores. Los valores marcados como "_RESERVADOS" tienen comentarios que describen su uso anterior o temporal.

Machine Translated by Google

Este documento también utiliza el registro TLS ExtensionType Values creado originalmente en [RFC4366]. IANA lo ha actualizado para hacer referencia a este documento. Los cambios en el registro son los siguientes:

- IANA ha actualizado la política de registro de la siguiente manera:

  Los valores con el primer byte en el rango 0-254 (decimal) se asignan mediante Especificación requerida [RFC8126]. Los valores con el primer byte 255 (decimal) están reservados para uso privado [RFC8126].

- IANA actualizó este registro para incluir las extensiones "key_share", "pre_shared_key", "psk_key_exchange_modes", "early_data", "cookie", "supported_versions", "certificate_authorities", "oid_filters", "post_handshake_auth" y "signature_algorithms_cert". con los valores definidos en este documento y el valor "Recomendado" de "Y".

- IANA ha actualizado este registro para incluir una columna "TLS 1.3" que enumera los mensajes en los que puede aparecer la extensión. Esta columna se completó inicialmente a partir de la tabla en la Sección 4.2, con cualquier extensión que no aparezca marcada como "-" para indicar que TLS 1.3 no la usa.

Este documento actualiza una entrada en el registro de tipos de certificados TLS creado originalmente en [RFC6091] y actualizado en [RFC8447]. IANA actualizó la entrada para el valor 1 para que tenga el nombre "OpenPGP_RESERVED", el valor "N" "Recomendado" y el comentario "Usado en versiones de TLS anteriores a la 1.3".

Este documento actualiza una entrada en el registro de tipos de estado de certificado TLS creado originalmente en [RFC6961]. IANA actualizó la entrada para el valor 2 para que tenga el nombre "ocsp_multi_RESERVED" y el comentario "Usado en versiones de TLS anteriores a la 1.3".

Este documento actualiza dos entradas en el registro de grupos compatibles con TLS (creado con un nombre diferente por [RFC4492]; ahora mantenido por [RFC8422]) y actualizado por [RFC7919] y [RFC8447]. Las entradas para los valores 29 y 30 (x25519 y x448) se han actualizado para hacer referencia también a este documento.

Machine Translated by Google

Además, este documento define dos nuevos registros que mantiene la IANA:

- Registro TLS SignatureScheme: los valores con el primer byte en el rango 0-253 (decimal) se asignan a través de la especificación requerida [RFC8126]. Los valores con el primer byte 254 o 255 (decimal) están reservados para uso privado [RFC8126]. Los valores con el primer byte en el rango 0-6 o con el segundo byte en el rango 0-3 que no están actualmente asignados están reservados para la compatibilidad con versiones anteriores.

  Este registro tiene una columna "Recomendado". El registro se ha rellenado inicialmente con los valores descritos en la Sección 4.2.3.
  Los siguientes valores están marcados como "recomendados":
  ECDSA_SECP256R1_SHA256, ECDSA_SECP384R1_SHA384,
  RSA_PSS_RSAE_SHA256, RSA_PSS_RSAE_SHA384, RSA_PSSS_RSAE_SHA512,
  RSA_PSS_PSS_SHA256, RSA_PSS_PSS_SHA384, RSA_PSSS_SSSS_SHA512 y ED25519. A la columna "Recomendado" se le asigna un valor de "N" a menos que se solicite explícitamente, y agregar un valor con un valor "Recomendado" de "Y" requiere Acción de estándares [RFC8126].

  Se REQUIERE la aprobación de IESG para una transición Y->N.

- Registro TLS PskKeyExchangeMode: valores en el rango 0-253 (decimal) se asignan a través de Especificación requerida [RFC8126].
  Los valores 254 y 255 (decimales) están reservados para Uso Privado [RFC8126]. Este registro tiene una columna "Recomendado". El registro se ha rellenado inicialmente con psk_ke (0) y psk_dhe_ke (1). Ambos están marcados como "Recomendados". A la columna "Recomendado" se le asigna un valor de "N" a menos que se solicite explícitamente, y agregar un valor con un valor "Recomendado" de "Y" requiere Acción de estándares [RFC8126]. Se REQUIERE la aprobación de IESG para una transición Y->N.

Machine Translated by Google

12. Referencias

12.1. Referencias normativas

[DH76] Diffie, W. y M. Hellman, "Nuevas direcciones en criptografía", IEEE Transactions on Information Theory, vol. 22 No. 6, pp. 644-654, DOI 10.1109/TIT.1976.1055638, noviembre de 1976.

[ECDSA] Instituto Nacional Estadounidense de Estándares, "Criptografía de clave pública para la industria de servicios financieros: el algoritmo de firma digital de curva elíptica (ECDSA)", ANSI ANS X9.62-2005, noviembre de 2005.

[MCG] Dworkin, M., "Recomendación para modos de operación de cifrado de bloques: modo Galois/Contador (GCM) y GMAC", publicación especial NIST 800-38D, DOI 10.6028/NIST.SP.800-38D, noviembre de 2007.

[RFC2104] Krawczyk, H., Bellare, M. y R. Canetti, "HMAC: Keyed- Hashing for Message Authentication", RFC 2104, DOI 10.17487/RFC2104, febrero de 1997, <https://www.rfc-editor .org/info/rfc2104>.

[RFC2119] Bradner, S., "Palabras clave para usar en RFC para indicar Niveles de requisitos", BCP 14, RFC 2119, DOI 10.17487/RFC2119, marzo de 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC5116] McGrew, D., "Una interfaz y algoritmos para el cifrado autenticado", RFC 5116, DOI 10.17487/RFC5116, enero de 2008, <https://www.rfc-editor.org/info/rfc5116>.

[RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R. y W. Polk, "Perfil de lista de revocación de certificados (CRL) y certificado de infraestructura de clave pública X.509 de Internet", RFC 5280, DOI 10.17487/RFC5280, mayo de 2008, <https://www.rfc-editor .org/info/rfc5280>.

[RFC5705] Rescorla, E., "Exportadores de materiales clave para la seguridad de la capa de transporte (TLS)", RFC 5705, DOI 10.17487/RFC5705, marzo de 2010, <https://www.rfc-editor.org/info/rfc5705>.

[RFC5756] Turner, S., Brown, D., Yiu, K., Housley, R. y T. Polk, "Actualizaciones para parámetros de algoritmo RSAES-OAEP y RSASSA-PSS", RFC 5756, DOI 10.17487/RFC5756, enero de 2010, <https://www.rfc-editor.org/info/rfc5756>.

Machine Translated by Google

[RFC5869] Krawczyk, H. y P. Eronen, "Función de derivación de clave de extracción y expansión basada en HMAC (HKDF)", RFC 5869, DOI 10.17487/RFC5869, mayo de 2010, <https://www.rfc-editor .org/info/rfc5869>.

[RFC6066] Eastlake 3rd, D., "Seguridad de la capa de transporte (TLS) Extensiones: definiciones de extensión", RFC 6066, DOI 10.17487/ RFC6066, enero de 2011, <https://www.rfc-editor.org/info/rfc6066>.

[RFC6655] McGrew, D. y D. Bailey, "AES-CCM Cipher Suites for Transport Layer Security (TLS)", RFC 6655, DOI 10.17487/RFC6655, julio de 2012, <https://www.rfc-editor.org /info/ rfc6655>.

[RFC6960] Santesson, S., Myers, M., Ankney, R., Malpani, A., Galperin, S. y C. Adams, "X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP", RFC 6960, DOI 10.17487/RFC6960, junio de 2013, <https:// www.rfc-editor.org/ información/rfc6960>.

[RFC6961] Pettersen, Y., "La seguridad de la capa de transporte (TLS) Extensión de solicitud de estado de certificado múltiple", RFC 6961, DOI 10.17487/ RFC6961, junio de 2013, <https://www.rfc-editor.org/info/rfc6961>.

[RFC6962] Laurie, B., Langley, A. y E. Kasper, "Certificado Transparencia", RFC 6962, DOI 10.17487/RFC6962, junio de 2013, <https://www.rfc-editor.org/info/rfc6962>.

[RFC6979] Pornin, T., "Uso determinista de la firma digital (DSA) y algoritmo de firma digital de curva elíptica (ECDSA)", RFC 6979, DOI 10.17487/RFC6979, agosto de 2013, <https://www.rfc-editor.org/info/rfc6979>.

[RFC7301] Friedl, S., Popov, A., Langley, A. y E. Stephan, "Extensión de negociación del protocolo de capa de aplicación de seguridad de la capa de transporte (TLS), RFC 7301, DOI 10.17487/RFC7301, julio de 2014, <https://www.rfc-editor.org/info/rfc7301>.

[RFC7507] Moeller, B. y A. Langley, "Cifrado de señalización de respaldo TLS Suite Value (SCSV) para prevenir ataques de degradación de protocolo", RFC 7507, DOI 10.17487/RFC7507, abril de 2015, <https://www.rfc-editor.org/info/ rfc7507>.

[RFC7748] Langley, A., Hamburg, M. y S. Turner, "Curvas elípticas para la seguridad", RFC 7748, DOI 10.17487/RFC7748, enero de 2016, <https://www.rfc-editor.org/info /rfc7748>.

Machine Translated by Google

[RFC7919] Gillmor, D., "Diffie-Hellman de campo finito negociado Parámetros efímeros para la seguridad de la capa de transporte (TLS)", RFC 7919, DOI 10.17487/ RFC7919, agosto de 2016, <https://www.rfc-editor.org/info/rfc7919>.

[RFC8017] Moriarty, K., Ed., Kaliski, B., Jonsson, J. y A. Rusch, "PKCS #1: RSA Cryptography Specifications Version 2.2", RFC 8017, DOI 10.17487/RFC8017, noviembre de 2016, < https://www.rfc-editor.org/info/ rfc8017>.

[RFC8032] Josefsson, S. e I. Liusvaara, "Algoritmo de firma digital de curva de Edwards (EdDSA)", RFC 8032, DOI 10.17487/RFC8032, enero de 2017, <https://www.rfc-editor.org/info/ rfc8032>.

[RFC8126] Cotton, M., Leiba, B. y T. Narten, "Pautas para Redacción de una sección de consideraciones de la IANA en los RFC", BCP 26, RFC 8126, DOI 10.17487/RFC8126, junio de 2017, <https://www.rfc-editor.org/info/rfc8126>.

[RFC8174] Leiba, B., "Ambigüedad de mayúsculas y minúsculas en palabras clave de RFC 2119", BCP 14, RFC 8174, DOI 10.17487/RFC8174, mayo de 2017, <https://www.rfc-editor.org/ info/ rfc8174>.

[RFC8439] Nir, Y. y A. Langley, "ChaCha20 y Poly1305 para IETF Protocolos", RFC 8439, DOI 10.17487/RFC8439, junio de 2018, <https://www.rfc-editor.org/ info/rfc8439>.

[SHS] Dang, Q., "Secure Hash Standard (SHS)", informe del Instituto Nacional de Estándares y Tecnología, DOI 10.6028/NIST.FIPS.180-4, agosto de 2015.

[X690] UIT-T, "Tecnología de la información: reglas de codificación ASN.1: especificación de reglas de codificación básicas (BER), reglas de codificación canónicas (CER) y reglas de codificación distinguidas (DER)", ISO/IEC 8825-1:2015, noviembre de 2015 .

Machine Translated by Google

12.2. Referencias Informativas

[AEAD-LÍMITES]
Luykx, A. y K. Paterson, "Límites del uso de cifrado autenticado en TLS", agosto de 2017, <http://www.isg.rhul.ac.uk/~kp/TLS-AEbounds.pdf>.

[BBFGKZ16]
Bhargavan, K., Brzuska, C., Fournet, C., Green, M., Kohlweiss, M. y S. Zanella-Beguelin, "Downgrade Resilience in Key-Exchange Protocols", Actas del simposio IEEE sobre seguridad y privacidad (San José), DOI 10.1109/SP.2016.37, mayo de 2016.

[BBK17] Bhargavan, K., Blanchet, B. y N. Kobeissi, "Modelos verificados e implementaciones de referencia para el candidato estándar TLS 1.3", Actas del Simposio IEEE sobre seguridad y privacidad (San José), DOI 10.1109/SP.2017.26, mayo 2017.

[BDFKPPRSZZ16]
Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Kohlweiss, M., Pan, J., Protzenko, J., Rastogi, A., Swamy, N., Zanella-Beguelin, S. y J. Zinzindohoue, "Implementación y prueba de la capa de registro TLS 1.3", Actas del Simposio IEEE sobre seguridad y privacidad (San José), mayo de 2017, <https://eprint.iacr.org/2016/1178>.

[Ben17a] Benjamin, D., "Presentación ante el TLS WG en IETF 100", noviembre de 2017, <https://datatracker.ietf.org/meeting/100/materials/ slides-100-tls-sessa-tls13/> .

[Ben17b] Benjamin, D., "Resultados adicionales de TLS 1.3 de Chrome", mensaje a la lista de correo de TLS, 18 de diciembre de 2017, <https://www.ietf.org/mail-archive/web/tls/current/msg25168.html>.

[Blei98] Bleichenbacher, D., "Ataques de texto cifrado elegidos contra protocolos basados en el estándar de cifrado RSA PKCS #1", Actas de CRYPTO '98, 1998.

[BMMRT15] Badertscher, C., Matt, C., Maurer, U., Rogaway, P. y B. Tackmann, "Canales seguros aumentados y el objetivo de la capa de registro TLS 1.3", ProvSec 2015, septiembre de 2015, <https://eprint.iacr.org/2015/394>.

Machine Translated by Google

[BT16] Bellare, M. y B. Tackmann, "La seguridad multiusuario del cifrado autenticado: AES-GCM en TLS 1.3", Actas de CRYPTO 2016, julio de 2016, <https://eprint.iacr.org/2016/564> .

[CCG16] Cohn-Gordon, K., Cremers, C. y L. Garratt, "On Post-compromise Security", Simposio de fundamentos de seguridad informática de IEEE, DOI 10.1109/CSF.2016.19, julio de 2015.

[CHEQUEO] Checkoway, S., Maskiewicz, J., Garman, C., Fried, J., Cohney, S., Green, M., Heninger, N., Weinmann, R., Rescorla, E. y H. Shacham, "Un análisis sistemático del incidente Juniper Dual EC", Actas de la Conferencia ACM SIGSAC de 2016 sobre seguridad informática y de las comunicaciones - CCS '16, DOI 10.1145/2976749.2978395, octubre de 2016.

[CHHSV17] Cremers, C., Horvat, M., Hoyland, J., Scott, S. y T. van der Merwe, "Apretón de manos incómodo: posible discrepancia de la vista del cliente/servidor en la autenticación del cliente en el modo posterior al apretón de manos en Revisión 18", mensaje a la lista de correo de TLS, 10 de febrero de 2017, <https://www.ietf.org/mail-archive/web/tls/current/msg22382.html>.

[CHSV16] Cremers, C., Horvat, M., Scott, S. y T. van der Merwe, "Análisis y verificación automatizados de TLS 1.3: 0-RTT, reanudación y autenticación retrasada", Actas del Simposio IEEE sobre seguridad y privacidad (San José), DOI 10.1109/SP.2016.35, mayo de 2016, <https://ieeexplore.ieee .org/document/7546518/>.

[CK01] Canetti, R. y H. Krawczyk, "Análisis de los protocolos de intercambio de claves y su uso para crear canales seguros", Actas de Eurocrypt 2001, DOI 10.1007/3-540-44987-6_28, abril de 2001.

[CLÍNICA] Miller, B., Huang, L., Joseph, A. y J. Tygar, "I Know Por qué fue a la clínica: riesgos y realización del análisis de tráfico HTTPS", Privacy Enhancing Technologies, págs. 143-163, DOI 10.1007 / 978-3-319-08506-7_8, 2014.

[DFGS15] Dowling, B., Fischlin, M., Guenther, F. y D. Stebila, "Un análisis criptográfico de los candidatos del protocolo de protocolo de enlace TLS 1.3", Actas de ACM CCS 2015, octubre de 2015, <https://eprint.iacr.org/2015/914>.

Machine Translated by Google

[DFGS16] Dowling, B., Fischlin, M., Guenther, F. y D. Stebila, "Un análisis criptográfico del protocolo de protocolo de enlace de clave precompartida y completa de TLS 1.3", TRON 2016, febrero de 2016, <https://eprint.iacr.org/2016/081>.

[DOW92] Diffie, W., van Oorschot, P. y M. Wiener, "Autenticación e intercambios de claves autenticadas", Diseños, códigos y criptografía, DOI 10.1007/BF00124891, junio de 1992.

[DSS] Instituto Nacional de Estándares y Tecnología, EE. UU. Departamento de Comercio, "Digital Signature Standard (DSS)", NIST FIPS PUB 186-4, DOI 10.6028/NIST.FIPS.186-4, julio de 2013.

[FG17] Fischlin, M. y F. Guenther, "Ataques de repetición en tiempo de ida y vuelta cero: el caso de los candidatos del protocolo de enlace TLS 1.3", Actas de EuroS&P 2017, abril de 2017, <https://eprint.iacr.org/2017/ 082>.

[FGSW16] Fischlin, M., Guenther, F., Schmidt, B. y B. Warinschi, "Confirmación de claves en el intercambio de claves: un tratamiento formal e implicaciones para TLS 1.3", Actas del simposio IEEE sobre seguridad y privacidad (San José), DOI 10.1109/SP.2016.34, mayo de 2016, <https://ieeexplore.ieee.org/document/7546517/>.

[FW15] Weimer, F., "Factorización de claves RSA con TLS Perfect Forward Secrecy", septiembre de 2015.

[HCJC16] Husak, M., Cermak, M., Jirsik, T. y P. Celeda, "Análisis de tráfico HTTPS e identificación de clientes mediante huellas dactilares SSL/TLS pasivas", EURASIP Journal on Information Security, vol. 2016, DOI 10.1186/s13635-016-0030-7, febrero de 2016.

[HGFS15] Hlauschek, C., Gruber, M., Fankhauser, F. y C. Schanes, "Prying Open Pandora's Box: KCI Attacks against TLS", Actas del taller de USENIX sobre tecnologías ofensivas, agosto de 2015.

[IEEE1363] IEEE, "Especificaciones estándar de IEEE para criptografía de clave pública", IEEE Std. 1363-2000, DOI 10.1109/IEEESTD.2000.92292.

Machine Translated by Google

[JSS15] Jager, T., Schwenk, J. y J. Somorovsky, "Sobre la seguridad de TLS 1.3 y QUIC contra las debilidades en el cifrado PKCS#1 v1.5", Procedimientos de ACM CCS 2015, DOI 10.1145/2810103.2813657, octubre de 2015, <https://www.nds.rub.de/media/nds/veroeffentlichungen/2015/08/21/Tls13QuicAttacks.pdf>.

[ACUERDO CLAVE] Barker, E., Chen, L., Roginsky, A., Vassilev, A. y R. Davis, "Recomendación para esquemas de establecimiento de claves por pares mediante criptografía de logaritmos discretos", Instituto Nacional de Estándares y Tecnología, DOI 10.6028/NIST.SP.800-56Ar3, abril de 2018.

[Kraw10] Krawczyk, H., "Extracción criptográfica y clave Derivation: The HKDF Scheme", Actas de CRYPTO 2010, agosto de 2010, <https://eprint.iacr.org/2010/264>.

[Kraw16] Krawczyk, H., "Una autenticación unilateral a mutua Compilador para intercambio de claves (con aplicaciones para la autenticación de clientes en TLS 1.3), Actas de ACM CCS 2016, octubre de 2016, <https://eprint.iacr.org/2016/711>.

[KW16] Krawczyk, H. and H. Wee, "The OPTLS Protocol and TLS 1.3", Actas de EuroS&P 2016, marzo de 2016, <https://eprint.iacr.org/2015/978>.

[LXZFH16] Li, X., Xu, J., Zhang, Z., Feng, D. y H. Hu, "Multiple Handshakes Security of TLS 1.3 Candidates", Actas del Simposio IEEE sobre seguridad y privacidad (San José), DOI 10.1109/SP.2016.36, mayo de 2016, <https://ieeexplore.ieee.org/document/7546519/>.

[Mac17] MacCarthaigh, C., "Revisión de seguridad de TLS1.3 0-RTT", marzo de 2017, <https://github.com/tlswg/tls13-spec/issues/1001>.

[PS18] Patton, C. y T. Shrimpton, "Canales parcialmente especificados: la capa de registro TLS 1.3 sin elisión", 2018, <https://eprint.iacr.org/2018/634>.

[PSK-FINALIZADO] Scott, S., Cremers, C., Horvat, M. y T. van der Merwe, "Revisión 10: posible ataque si se permite la autenticación del cliente durante PSK", mensaje a la lista de correo de TLS, 31 de octubre de 2015, <https ://www.ietf.org/mail-archive/web/tls/current/msg18215.html>.

Machine Translated by Google

[VOLVER A CLAVE] Abdalla, M. and M. Bellare, "Increasing the Lifetime of a Key: A Comparative Analysis of the Security of Re-keying Techniques", ASIACRYPT 2000, DOI 10.1007/3-540-44448-3_42, octubre de 2000.

[Res17a] Rescorla, E., “Datos preliminares sobre Firefox TLS 1.3 Experimento de Middlebox", mensaje a la lista de correo de TLS, 5 de diciembre de 2017, <https://www.ietf.org/mail-archive/web/tls/current/msg25091.html>.

[Res17b] Rescorla, E., "Más resultados de medición de compatibilidad", mensaje a la lista de correo de TLS, 22 de diciembre de 2017, <https://www.ietf.org/mail-archive/web/tls/current/msg25179.html>.

[RFC3552] Rescorla, E. y B. Korver, "Pautas para escribir texto RFC sobre consideraciones de seguridad", BCP 72, RFC 3552, DOI 10.17487/RFC3552, julio de 2003, <https://www.rfc-editor.org/información/rfc3552>.

[RFC4086] Eastlake 3rd, D., Schiller, J. y S. Crocker, "Requisitos de aleatoriedad para la seguridad", BCP 106, RFC 4086, DOI 10.17487/RFC4086, junio de 2005, <https://www.rfc-editor.org/info/rfc4086>.

[RFC4346] Dierks, T. y E. Rescorla, "La seguridad de la capa de transporte (TLS) Protocolo versión 1.1", RFC 4346, DOI 10.17487/RFC4346, abril de 2006, <https://www.rfc-editor.org/info/rfc4346>.

[RFC4366] Blake-Wilson, S., Nystrom, M., Hopwood, D., Mikkelsen, J. y T. Wright, "Seguridad de la capa de transporte (TLS) Extensiones", RFC 4366, DOI 10.17487/RFC4366, abril de 2006, <https://www.rfc-editor.org/info/rfc4366>.

[RFC4492] Blake-Wilson, S., Bolyard, N., Gupta, V., Hawk, C. y B. Moeller, "Suites de cifrado de criptografía de curva elíptica (ECC) para seguridad de la capa de transporte (TLS)", RFC 4492, DOI 10.17487/RFC4492, mayo de 2006, <https://www.rfc-editor.org/info/rfc4492>.

[RFC5077] Salowey, J., Zhou, H., Eronen, P. y H. Tschofenig, "Reanudación de la sesión de seguridad de la capa de transporte (TLS) sin estado del lado del servidor", RFC 5077, DOI 10.17487/RFC5077, enero de 2008, <https://www.rfc-editor.org/info/rfc5077>.

Machine Translated by Google

[RFC5246] Dierks, T. y E. Rescorla, "La seguridad de la capa de transporte (TLS) Protocolo versión 1.2", RFC 5246, DOI 10.17487/RFC5246, agosto de 2008, <https://www.rfc-editor.org/info/rfc5246>.

[RFC5764] McGrew, D. y E. Rescorla, "Capa de transporte de datagramas Security (DTLS) Extension to Establish Keys for the Secure Real-time Transport Protocol (SRTP)", RFC 5764, DOI 10.17487/RFC5764, mayo de 2010, <https://www.rfc-editor.org/info/rfc5764>.

[RFC5929] Altman, J., Williams, N. y L. Zhu, "Channel Bindings for TLS", RFC 5929, DOI 10.17487/RFC5929, julio de 2010, <https://www.rfc-editor.org/info /rfc5929>.

[RFC6091] Mavrogiannopoulos, N. y D. Gillmor, "Uso de claves OpenPGP para la autenticación de seguridad de la capa de transporte (TLS), RFC 6091, DOI 10.17487/RFC6091, febrero de 2011, <https://www.rfc-editor.org/info/rfc6091>.

[RFC6101] Freier, A., Karlton, P. y P. Kocher, "Protocolo de capa de conexión segura (SSL) versión 3.0", RFC 6101, DOI 10.17487/RFC6101, agosto de 2011, <https://www.rfc-editor.org/info/rfc6101>.

[RFC6176] Turner, S. y T. Polk, "Prohibición de la capa de sockets seguros (SSL) Versión 2.0", RFC 6176, DOI 10.17487/RFC6176, marzo de 2011, <https://www.rfc-editor.org/info/rfc6176>.

[RFC6347] Rescorla, E. y N. Modadugu, "Capa de transporte de datagramas Versión de seguridad 1.2", RFC 6347, DOI 10.17487/RFC6347, enero de 2012, <https://www.rfc-editor.org/info/rfc6347>.

[RFC6520] Seggelmann, R., Tuexen, M. y M. Williams, "Transporte Layer Security (TLS) y Datagram Transport Layer Security (DTLS) Heartbeat Extension", RFC 6520, DOI 10.17487/RFC6520, febrero de 2012, <https://www.rfc-editor.org/info/rfc6520>.

[RFC7230] Fielding, R., ed. y J. Reschke, Ed., "Protocolo de transferencia de hipertexto (HTTP/1.1): enrutamiento y sintaxis de mensajes", RFC 7230, DOI 10.17487/RFC7230, junio de 2014, <https://www.rfc-editor.org/info/ rfc7230>.

Machine Translated by Google

[RFC7250] Wouters, P., Ed., Tschofenig, H., Ed., Gilmore, J., Weiler, S. y T. Kivinen, "Uso de claves públicas sin procesar en la seguridad de la capa de transporte (TLS) y la seguridad de la capa de transporte de datagramas (DTLS)", RFC 7250, DOI 10.17487/RFC7250, junio de 2014, <https://www. rfc-editor.org/info/ rfc7250>.

[RFC7465] Popov, A., "Prohibición de suites de cifrado RC4", RFC 7465, DOI 10.17487/RFC7465, febrero de 2015, <https://www.rfc-editor.org/info/rfc7465>.

[RFC7568] Barnes, R., Thomson, M., Pironti, A. y A. Langley, "Desuso de la versión 3.0 de Secure Sockets Layer", RFC 7568, DOI 10.17487/RFC7568, junio de 2015, <https://www.rfc-editor.org/info/rfc7568>.

[RFC7627] Bhargavan, K., Ed., Delignat-Lavaud, A., Pironti, A., Langley, A. y M. Ray, "Seguridad de la capa de transporte (TLS) Session Hash and Extended Master Secret Extension", RFC 7627, DOI 10.17487/ RFC7627, septiembre de 2015, <https://www.rfc-editor.org/info/rfc7627>.

[RFC7685] Langley, A., "Un cliente de seguridad de la capa de transporte (TLS)Hello Extensión de relleno", RFC 7685, DOI 10.17487/RFC7685, octubre de 2015, <https:// www.rfc-editor.org/info/rfc7685>.

[RFC7924] Santesson, S. y H. Tschofenig, "Transport Layer Security (TLS) Cached Information Extension", RFC 7924, DOI 10.17487/RFC7924, julio de 2016, <https://www.rfc-editor.org/info/ rfc7924>.

[RFC8305] Schinazi, D. y T. Pauly, "Happy Eyeballs Version 2: Better Connectivity Using Concurrency", RFC 8305, DOI 10.17487/RFC8305, diciembre de 2017, <https://www.rfc-editor.org/ info/ rfc8305>.

[RFC8422] Nir, Y., Josefsson, S. y M. Pegourie-Gonnard, "Suites de cifrado de criptografía de curva elíptica (ECC) para seguridad de la capa de transporte (TLS) versiones 1.2 y anteriores", RFC 8422, DOI 10.17487/RFC8422, Agosto de 2018, <https://www.rfc-editor.org/info/rfc8422>.

[RFC8447] Salowey, J. y S. Turner, "Actualizaciones del registro IANA para TLS y DTLS", RFC 8447, DOI 10.17487/RFC8447, agosto de 2018, <https://www.rfc-editor.org/ info/rfc8447>.

[RFC8449] Thomson, M., "Extensión de límite de tamaño de registro para TLS", RFC 8449, DOI 10.17487/RFC8449, agosto de 2018, <https://www.rfc-editor.org/info/rfc8449>.

Machine Translated by Google

[RSA] Rivest, R., Shamir, A. y L. Adleman, "Un método para Obtención de Firmas Digitales y Clave Pública Criptosistemas", Comunicaciones de la ACM, Vol. 21 No. 2, págs. 120-126, DOI 10.1145/359340.359342, febrero de 1978.

[SIGMA] Krawczyk, H., "SIGMA: El enfoque 'SIGn-and-MAc' para Diffie-Hellman autenticado y su uso en el IKE Protocolos", Actas de CRYPTO 2003, DOI 10.1007 / 978-3-540-45146-4_24, agosto de 2003.

[RANURA] Bhargavan, K. y G. Leurent, "Transcript Collision Ataques: romper la autenticación en TLS, IKE y SSH", Seguridad de redes y sistemas distribuidos Simposio (NDSS 2016), DOI 10.14722/ndss.2016.23418, febrero de 2016.

[SSL2] Hickman, K., "El protocolo SSL", febrero de 1995.

[TIMING] Boneh, D. y D. Brumley, "Los ataques de sincronización remota son Práctico", Simposio de seguridad de USENIX, agosto de 2003.

[TLS13-TRAZAS] Thomson, M., "Ejemplo de seguimientos de protocolo de enlace para TLS 1.3", Trabajo en curso, draft-ietf-tls-tls13-vectors-06, julio de 2018.

[X501] UIT-T, "Tecnología de la información - Sistemas abiertos Interconexión - El Directorio: Modelos", ITU-T X.501, Octubre de 2016, <https://www.itu.int/rec/T-REC-X.501/en>.

Machine Translated by Google

Apéndice A. Máquina de estado

Este apéndice proporciona un resumen de las transiciones de estado legal para los apretones de manos del cliente y del servidor. Nombres de los estados (en mayúsculas, ej., INICIO) no tienen un significado formal, pero se proporcionan para facilitar su comprensión. Acciones que se toman solo en ciertas circunstancias se indican en []. La notación "K_{send,recv} = foo" significa "establecer la clave de enviar/recibir a la clave dada".

A.1. Cliente

```
                                   COMENZAR <----+
            Enviar ClienteHola |                 | Recv HelloRetryRequest
      [K_send = datos iniciales] |                 |
                                 v                 |
 /|                        ESPERA_SH ----+
 |                              | Servidor de recepciónHola
                                | K_recv = apretón de manos
    puede |                     V
   enviar |                  ESPERA_EE
temprano |                      | Recv EncryptedExtensions
   datos | | | |      +--------+--------+
            Usando |                      | Usando certificado
      prostituta |                       v
                 |                 ESPERA_CERT_CR
                 |            recibido |          | Solicitud de certificado de recepción
                 | Certificado | | | | | |        v
                                          ESPERA_CERT
                                                | Certificado de recepción
                                   v            v
                                        ESPERA_CV
                 |                              | Certificado de recepciónVerificar
                 +> ESPERA_TERMINADO <+
                                | Recepción finalizada
 | | | | | | | | \              | [Enviar EndOfEarlyData]
                                | K_send = apretón de manos
                                | [Enviar certificado [+ CertificateVerify]]
Puede enviar                    | Enviar terminado
datos de la aplicación --       | K_send = K_recv = aplicación
> después de aquí               v
                            CONECTADO
```

Tenga en cuenta que con las transiciones que se muestran arriba, los clientes pueden enviar alertas que se derivan de mensajes post-ServerHello en claro o con las primeras claves de datos. Si los clientes necesitan enviar dichas alertas, Primero DEBE volver a introducir las teclas de protocolo de enlace si es posible.

Machine Translated by Google

A.2. Servidor

```
                                   COMENZAR <------+
                Recv ClientHola |                  | Enviar HelloRetryRequest
                                v                  |
                             RECVD_CH ----+
                                  | Seleccionar parámetros
                                  v
                             NEGOCIADO
                                  | Enviar servidorHola
                                  | K_send = apretón de manos
                                  | Enviar extensiones cifradas
                                  | [Enviar solicitud de certificado]
Puede enviar                      | [Enviar certificado + Verificar certificado]
datos de la                       | Enviar terminado
aplicación después                | K_send = aplicación
de --> aquí          +--------+--------+
            Sin 0-RTT | |                      | 0-RTT
                                                |
  K_recv = apretón de manos |                   | K_recv = datos iniciales
[Omitir errores de descifrado] |     +------> ESPERA_EOED -+
                                     "                recibido |          | Recv EndOfEarlyData
                                     | | primeros datos |                 | K_recv = apretón de manos
                                     +------------+
                                     "
                                     "
                                     +> ESPERA_VUELO2 <--------+
                                                |
                                     +--------+--------+
                Sin autenticación | | |                | autenticación de cliente
                                     |                  |
                                                        v
                                                  ESPERA_CERT
                                     |            recibido |          | Certificado de recepción
                                     | vacío |                        v
                                     | Certificado |                ESPERA_CV
                                     "                                | recibido
                                     |                  v             | CertificadoVerificar
                                     +-> ESPERAR_TERMINADO <---+
                                                | Recepción finalizada
                                                | K_recv = aplicación
                                                v
                                           CONECTADO
```

Machine Translated by Google

Apéndice B. Estructuras de datos de protocolo y valores constantes

Este apéndice proporciona los tipos de protocolos normativos y las definiciones de constantes. Los valores enumerados como "_RESERVADOS" se usaron en versiones anteriores de TLS y se enumeran aquí para completar.
Las implementaciones de TLS 1.3 NO DEBEN enviarlos, pero pueden recibirlos de implementaciones de TLS anteriores.

B.1. Capa de registro

```
enum
        { invalid(0),
         change_cipher_spec(20), alert(21),
         handshake(22), application_data(23),
         heartbeat(24), /* RFC 6520 */ (255)



} Tipo de contenido;

estructura
         { tipo de tipo de contenido;
         ProtocolVersion legacy_record_version; uint16 longitud;
         fragmento opaco[TLSPlaintext.length]; } TLSTexto sin formato;



struct
        { contenido opaco[TLSPlaintext.length]; Tipo de tipo de
         contenido; uint8 ceros[longitud_de_relleno]; } TLSInnerPlaintext;



struct
        { ContentType opaque_type = application_data; /* 23 */ ProtocolVersion
         legacy_record_version = 0x0303; /* TLS v1.2 */ uint16 longitud; registro_cifrado opaco[TLSCiphertext.length]; }
         TLSCiphertext;
```

Machine Translated by Google

B.2. Mensajes de alerta

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

enum
        { close_notify(0),
         mensaje_inesperado(10),
         bad_record_mac(20),
         decryption_failed_RESERVED(21),
         record_overflow(22),
         descompression_failure_RESERVED(30),
         handshake_failure(40), no_certificate_RESERVED(41),
         bad_certificate(42), unsupported_certificate(43) ),
         certificado_revocado(44), certificado_caducado(45),
         certificado_desconocido(46), parámetro_ilegal(47),
         ca_desconocida(48), acceso_denegado(49),
         error_descifrado(50), error_descifrado(51),
         restricción_exportación_RESERVADA(60),
         versión_protocolo(70) ) ),
         bad_certificate_hash_value_RESERVED(114),
         unknown_psk_identity(115), certificate_required(116),
         no_application_protocol(120), (255)

} Descripción de alerta;

estructura {
        nivel de AlertLevel;
        AlertDescription descripción;
} Alerta;
```

Machine Translated by Google

B.3. Protocolo de apretón de manos

```
enumeración {
        hola_solicitud_RESERVADO(0),
        cliente_hola(1),
        servidor_hola(2),
        hola_verificar_solicitud_RESERVADO(3),
        boleto_nueva_sesión(4),
        fin_de_los_datos_antiguos(5),
        hola_reintentar_solicitud_RESERVADO(6),
        extensiones_cifradas(8),
        certificado(11),
        server_key_exchange_RESERVADO(12),
        solicitud_certificado(13),
        servidor_hola_hecho_RESERVADO(14),
        certificado_verificar(15),
        cliente_intercambio_clave_RESERVADO(16),
        terminado(20),
        URL_certificado_RESERVADO(21),
        estado_certificado_RESERVADO(22),
        datos_suplementarios_RESERVADOS(23),
        actualización_clave(24),
        mensaje_hash(254),
        (255)
} Tipo de apretón de manos;

estructura {
        HandshakeType msg_type; uint24          /* tipo de apretón de manos */
        longitud; seleccione (Apretón de        /* bytes en el mensaje */
        manos.msg_type) {
                caso cliente_hola: ClienteHola;
                case servidor_hola: ServidorHola;
                case end_of_early_data: EndOfEarlyData;
                caso extensiones_cifradas: extensiones_cifradas;
                case certificate_request: CertificateRequest;
                caso certificado: caso                  Certificado;
                certificado_verificar: caso finalizado: CertificadoVerificar;
                caso nuevo_ticket_sesión: caso          Finalizado;
                clave_actualización:                    NewSessionTicket;
                                                        Actualización de clave;
        };
} Apretón de manos;
```

Machine Translated by Google

B.3.1. Mensajes de intercambio de claves

```
uint16 Versión de protocolo;
Aleatorio opaco[32];

uint8 CipherSuite[2];                    /* Selector de suite criptográfica */

estructura {
        Versión de protocolo versión_heredada = 0x0303; Aleatorio aleatorio;        /* TLS v1.2 */

        opaco legacy_session_id<0..32>;
        CipherSuite cipher_suites<2..2^16-2>;
        métodos_compresión_heredados opacos<1..2^8-1>;
        Extensión extensiones<8..2^16-1>;
} ClienteHola;

estructura {
        Versión de protocolo versión_heredada = 0x0303; Aleatorio aleatorio;        /* TLS v1.2 */

        opaco legacy_session_id_echo<0..32>;
        CipherSuite cipher_suite;
        uint8 método_compresión_heredado = 0;
        Extensión extensiones<6..2^16-1>;
} ServidorHola;
```

Machine Translated by Google


```
estructura {
        ExtensionType extension_type;
        extension_data opaca<0..2^16-1>;
} Extensión;

enumeración {
        nombre_servidor(0),                                  /*RFC6066*/
        longitud_máx_fragmento(1),                           /*RFC6066*/
        estado_solicitud(5),                                 /*RFC6066*/
        grupos_soportados(10),                               /* RFC 8422, 7919 */
        algoritmos_de_firma(13),                             /* RFC 8446 */
        usar_srtp(14),                                       /* RFC 5764 */
        latido del corazón(15),                              /*RFC6520*/
        application_layer_protocol_negotiation(16), /* RFC 7301 */
        firmado_certificado_marca de tiempo (18),            /*RFC6962*/
        tipo_certificado_cliente(19),                        /*RFC7250*/
        tipo_certificado_servidor(20),                       /*RFC7250*/
        relleno(21),                                         /*RFC7685*/
        RESERVADO(40),                                       /* Usado pero nunca
                                                                asignado */
        pre_shared_key(41),                                  /* RFC 8446 */
        early_data(42),                                      /* RFC 8446 */
        support_versions(43), cookie(44),                    /* RFC 8446 */
        psk_key_exchange_modes(45),                          /* RFC 8446 */
                                                             /* RFC 8446 */
        RESERVADO(46),                                       /* Usado pero nunca
                                                                asignado */
        Certificate_authorities(47), oid_filters(48),        /* RFC 8446 */
        post_handshake_auth(49),                             /* RFC 8446 */
        signature_algorithms_cert(50), key_share(51),        /* RFC 8446 */
        (65535)                                              /* RFC 8446 */
                                                             /* RFC 8446 */

} tipo de extensión;

estructura {
        grupo NamedGroup;
        intercambio de clave opaca<1..2^16-1>;
} KeyShareEntry;

estructura {
        KeyShareEntry client_shares<0..2^16-1>;
} KeyShareClientHola;

estructura {
        grupo_nombrado grupo_seleccionado;
} KeyShareHelloRetryRequest;
```

Machine Translated by Google


```
estructura {
        KeyShareEntry server_share;
} KeyShareServerHola;

estructura
        { uint8 forma_heredada = 4; X
        opaco[longitud_coordenada]; opaco
        Y[longitud_coordenada];
} Representación de puntos sin comprimir;

enum {psk_ke (0), psk_dhe_ke (1), (255)} PskKeyExchangeMode;

estructura
        { PskKeyExchangeMode ke_modes<1..255>; }
PskKeyExchangeModes;

estructura {} Vacío;

estructura
        { seleccionar (Apretón de manos.msg_type) {
                case new_session_ticket: uint32 max_early_data_size; caso cliente_hola:
                                                                Vacío;
                case encrypted_extensions: Vacío;
        };
} Indicación de datos tempranos;

struct
        { identidad opaca<1..2^16-1>; uint32
        ofuscado_ticket_age; } PskIdentidad;


opaco PskBinderEntry<32..255>;

estructura
        { PskIdentity identidades<7..2^16-1>; Carpetas
        PskBinderEntry<33..2^16-1>; } Paquetes Ofrecidos;


estructura
        { seleccionar (Apretón de manos.msg_type) {
                caso client_hello: OfferedPsks; case server_hello:
                uint16 selected_identity;
        };
} PreSharedKeyExtension;
```

Machine Translated by Google

B.3.1.1. Extensión de versión

```
estructura {
    seleccione (Apretón de manos.msg_type) {
        caso cliente_hola:
            Versiones de ProtocolVersion<2..254>;

        case server_hello: /* y HelloRetryRequest */
            ProtocolVersion versión_seleccionada;
    };
} Versiones admitidas;
```

B.3.1.2. Extensión de cookies

```
estructura {
    cookie opaca<1..2^16-1>;
} Galleta;
```

Machine Translated by Google

B.3.1.3. Extensión del algoritmo de firma

```
enumeración {
        /* Algoritmos RSASSA-PKCS1-v1_5 */
        rsa_pkcs1_sha256(0x0401), rsa_pkcs1_sha384(0x0501),
        rsa_pkcs1_sha512(0x0601),


        /* Algoritmos ECDSA */
        ecdsa_secp256r1_sha256(0x0403),
        ecdsa_secp384r1_sha384(0x0503),
        ecdsa_secp521r1_sha512(0x0603),

        /* Algoritmos RSASSA-PSS con clave pública OID rsaEncryption */ rsa_pss_rsae_sha256(0x0804),
        rsa_pss_rsae_sha384(0x0805), rsa_pss_rsae_sha512(0x0806),



        /* Algoritmos EdDSA */
        ed25519(0x0807), ed448(0x0808),


        /* Algoritmos RSASSA-PSS con clave pública OID RSASSA-PSS */ rsa_pss_pss_sha256(0x0809),
        rsa_pss_pss_sha384(0x080a), rsa_pss_pss_sha512(0x080b),



        /* Algoritmos heredados */
        rsa_pkcs1_sha1(0x0201),
        ecdsa_sha1(0x0203),

        / * * Puntos de código reservados /
        obsolete_RESERVED (0x0000..0x0200),
        dsa_sha1_RESERVED (0x0202), obsolete_RESERVED
        (0x0204..0x0400), dsa_sha256_RESERVED (0x0402),
        obsolete_RESERVED (0x0404..0x0500),
        dsa_sha384_RESERVED (0x0502), obsolete_RESERVED
        (0x0504. .0x0600), dsa_sha512_RESERVED(0x0602),
        obsoleto_RESERVED(0x0604..0x06FF),
        private_use(0xFE00..0xFFFF), (0xFFFF)



} Esquema de firma;

struct
        { SignatureScheme support_signature_algorithms<2..2^16-2>;
} Lista de esquemas de firmas;
```

Machine Translated by Google

B.3.1.4. Extensión de grupos admitidos

```
enumeración { no asignado_RESERVADO (0x0000),

        /* Grupos de curvas elípticas (ECDHE) */
        obsolete_RESERVED(0x0001..0x0016), secp256r1(0x0017),
        secp384r1(0x0018), secp521r1(0x0019), obsolete_RESERVED(0x001A..0x001C),
        x25519(0x001D), x041E(0) ,


        /* Grupos de campos finitos (DHE) */
        ffdhe2048(0x0100), ffdhe3072(0x0101), ffdhe4096(0x0102), ffdhe6144(0x0103),
        ffdhe8192(0x0104),

        /* Puntos de código reservados */
        ffdhe_private_use(0x01FC..0x01FF),
        ecdhe_private_use(0xFE00..0xFEFF),
        obsolete_RESERVED(0xFF01..0xFF02), (0xFFFF)

} grupo con nombre;

struct
        { NamedGroup lista_de_grupos_nombrados<2..2^16-1>; } Lista
de grupos con nombre;
```

Los valores dentro de los rangos "obsolete_RESERVED" se usan en versiones anteriores de TLS y NO DEBEN ser ofrecidos ni negociados por las implementaciones de TLS 1.3. Las curvas obsoletas tienen varias debilidades conocidas/teóricas o han tenido muy poco uso, en algunos casos solo debido a problemas de configuración del servidor no intencionales. Ya no se consideran apropiados para uso general y se debe suponer que son potencialmente inseguros. El conjunto de curvas especificado aquí es suficiente para la interoperabilidad con todas las implementaciones de TLS actualmente implementadas y configuradas correctamente.

Machine Translated by Google

B.3.2. Mensajes de parámetros del servidor

```
nombre Distinguido opaco<1..2^16-1>;

struct
        { Nombre Distinguido autoridades<3..2^16-1>; } Extensión de las
 autoridades del certificado;

struct
        { certificado opaco_extensión_oid<1..2^8-1>; valores_extensión_certificado
         opaco<0..2^16-1>; } FiltroOID;


estructura
         { filtros OIDFilter<0..2^16-1>; } OIDFilterExtension;


estructura {} PostHandshakeAuth;

estructura
         { Extensión extensiones<0..2^16-1>;
} Extensiones cifradas;

struct
        { certificado opaco_request_context<0..2^8-1>; Extensión
         extensiones<2..2^16-1>; } Solicitud de certificado;
```

Machine Translated by Google

B.3.3. Mensajes de autenticación

```
enumeración {
        X509(0),
        AbrirPGP_RESERVADO(1),
        RawPublicKey(2), (255)

} Tipo de certificado;

struct { select
        (tipo_certificado) { case RawPublicKey: /* De
                RFC 7250 ASN.1_subjectPublicKeyInfo
                    */ opaco ASN1_subjectPublicKeyInfo<1..2^24-1>;


                caso X509:
                    opaco cert_data<1..2^24-1>;
        };
        Extensión extensiones<0..2^16-1>; } Entrada de
certificado;

struct
        { certificado opaco_request_context<0..2^8-1>; CertificateEntry
        lista_certificados<0..2^24-1>; } Certificado;


struct
        {Algoritmo SignatureScheme; firma
        opaca<0..2^16-1>;
} CertificadoVerificar;

struct { opaco
        verificar_datos[Hash.longitud];
} Finalizado;
```

B.3.4. Establecimiento de boletos

```
estructura
        { uint32 ticket_lifetime; uint32
        ticket_age_add; ticket_nonce
        opaco<0..255>; ticket opaco<1..2^16-1>;
        Extensión extensiones<0..2^16-2>; }
        NewSessionTicket;
```

Machine Translated by Google

B.3.5. Actualización de claves

```
estructura {} EndOfEarlyData;

enumeración {
    actualización_no_solicitada(0), actualización_solicitada(1), (255)
} Solicitud de actualización de clave;

estructura {
    KeyUpdateRequest request_update;
} Actualización de clave;
```

B.4. Paquetes de cifrado

Un conjunto de cifrado simétrico define el par del algoritmo AEAD y algoritmo hash para ser utilizado con HKDF. Los nombres de los conjuntos de cifrado siguen el convenio de denominación:

```
CipherSuite TLS_AEAD_HASH = VALOR;

+-----------+------------------------------------------------- -----------+
| Componente | Contenido                                                    |
+-----------+------------------------------------------------- -----------+
| TLS |          | La cadena "TLS" |
|           |
| AEAD         | El algoritmo AEAD utilizado para la protección de registros |
|           |
| picadillo    | El algoritmo hash utilizado con HKDF |
|           |
| VALOR        | El ID de dos bytes asignado para este conjunto de cifrado |
+-----------+------------------------------------------------- -----------+
```

Esta especificación define los siguientes conjuntos de cifrado para su uso con TLS 1.3.

Descripción	Valor
TLS_AES_128_GCM_SHA256	{0x13,0x01}
TLS_AES_256_GCM_SHA384	{0x13,0x02}
TLS_CHACHA20_POLY1305_SHA256	{0x13,0x03}
TLS_AES_128_CCM_SHA256	{0x13,0x04}
TLS_AES_128_CCM_8_SHA256	{0x13,0x05}

Machine Translated by Google

Los algoritmos AEAD correspondientes AEAD_AES_128_GCM, AEAD_AES_256_GCM y AEAD_AES_128_CCM se definen en [RFC5116].
AEAD_CHACHA20_POLY1305 se define en [RFC8439]. AEAD_AES_128_CCM_8 se define en [RFC6655]. Los algoritmos hash correspondientes se definen en [SHS].

Si bien TLS 1.3 usa el mismo espacio de conjuntos de cifrado que las versiones anteriores de TLS, los conjuntos de cifrado de TLS 1.3 se definen de manera diferente, solo especifican los cifrados simétricos y no se pueden usar para TLS 1.2.
Del mismo modo, los conjuntos de cifrado para TLS 1.2 y versiones anteriores no se pueden usar con TLS 1.3.

La IANA asigna nuevos valores de conjunto de cifrado como se describe en la Sección 11.

Apéndice C. Notas de implementación

El protocolo TLS no puede evitar muchos errores de seguridad comunes. Este apéndice proporciona varias recomendaciones para ayudar a los implementadores.
[TLS13-TRACES] proporciona vectores de prueba para protocolos de enlace TLS 1.3.

C.1. Generación y siembra de números aleatorios

TLS requiere un generador de números pseudoaleatorios criptográficamente seguro (CSPRNG). En la mayoría de los casos, el sistema operativo proporciona una función adecuada, como /dev/urandom, que se debe usar en ausencia de otras preocupaciones (por ejemplo, rendimiento). Se RECOMIENDA utilizar una implementación CSPRNG existente en lugar de crear una nueva. Muchas bibliotecas criptográficas adecuadas ya están disponibles bajo términos de licencia favorables. En caso de que resulten insatisfactorios, [RFC4086] proporciona orientación sobre la generación de valores aleatorios.

TLS utiliza valores aleatorios (1) en campos de protocolo público, como los valores aleatorios públicos en ClientHello y ServerHello y (2) para generar material de claves. Con un CSPRNG que funcione correctamente, esto no presenta un problema de seguridad, ya que no es factible determinar el estado de CSPRNG a partir de su salida. Sin embargo, con un CSPRNG dañado, es posible que un atacante use la salida pública para determinar el estado interno del CSPRNG y, por lo tanto, predecir el material de clave, como se documenta en [CHECKOWAY]. Las implementaciones pueden proporcionar seguridad adicional contra esta forma de ataque mediante el uso de CSPRNG separados para generar valores públicos y privados.

Machine Translated by Google

C.2. Certificados y Autenticación

Las implementaciones son responsables de verificar la integridad de los certificados y, en general, deben admitir mensajes de revocación de certificados. En ausencia de una indicación específica de un perfil de aplicación, los certificados siempre deben verificarse para garantizar la firma adecuada por parte de una autoridad de certificación (CA) de confianza. La selección y adición de anclas de confianza debe hacerse con mucho cuidado. Los usuarios deberían poder ver información sobre el certificado y el ancla de confianza.

Las aplicaciones DEBERÍAN también imponer tamaños de clave mínimos y máximos. Por ejemplo, las rutas de certificación que contienen claves o firmas más débiles que RSA de 2048 bits o ECDSA de 224 bits no son apropiadas para aplicaciones seguras.

C.3. Errores de implementación

La experiencia de implementación ha demostrado que ciertas partes de las especificaciones TLS anteriores no son fáciles de entender y han sido una fuente de problemas de interoperabilidad y seguridad. Muchas de estas áreas se han aclarado en este documento, pero este apéndice contiene una breve lista de las cosas más importantes que requieren una atención especial por parte de los implementadores.

Problemas con el protocolo TLS:

- ¿Maneja correctamente los mensajes de protocolo de enlace que están fragmentados en varios registros TLS (consulte la Sección 5.1)? ¿Maneja correctamente los casos de esquina como un ClientHello que se divide en varios fragmentos pequeños? ¿Fragmentas mensajes de protocolo de enlace que exceden el tamaño máximo de fragmento? En particular, los mensajes de protocolo de enlace Certificate y CertificateRequest pueden ser lo suficientemente grandes como para requerir fragmentación.

- ¿Ignora el número de versión de la capa de registro TLS en todos los registros TLS sin cifrar (consulte el Apéndice D)?

- ¿Se ha asegurado de que todo el soporte para cifrado SSL, RC4, EXPORTACIÓN, y MD5 (a través de la extensión "signature_algorithms") se eliminó por completo de todas las configuraciones posibles que admiten TLS 1.3 o posterior, y que los intentos de usar estas capacidades obsoletas fallan correctamente (consulte el Apéndice D)?

- ¿Maneja correctamente las extensiones TLS en ClientHellos, incluidas las extensiones desconocidas?

Machine Translated by Google

- Cuando el servidor ha solicitado un certificado de cliente pero no hay ningún certificado adecuado disponible, ¿envía correctamente un mensaje de Certificado vacío, en lugar de omitir el mensaje completo (consulte la Sección 4.4.2)?

- Al procesar el fragmento de texto sin formato producido por AEAD-Decrypt y escanear desde el final para el ContentType, ¿evita escanear más allá del inicio del texto sin cifrar en caso de que el par haya enviado un texto sin formato con formato incorrecto de todos ceros?

- ¿Ignora correctamente conjuntos de cifrado no reconocidos (Sección 4.1.2), extensiones de saludo (Sección 4.2), grupos con nombre (Sección 4.2.7), claves compartidas (Sección 4.2.8), versiones admitidas (Sección 4.2.1) y algoritmos de firma (Sección 4.2.3) en ClientHello?

- Como servidor, ¿envía un HelloRetryRequest a los clientes que admite un grupo DHE compatible (EC) pero no lo predice en la extensión "key_share"? Como cliente, ¿maneja correctamente una HelloRetryRequest del servidor?

Detalles criptográficos:

- ¿Qué contramedidas usas para prevenir ataques de tiempo? [SINCRONIZACIÓN]?

- Al utilizar el intercambio de claves Diffie-Hellman, ¿conserva correctamente los ceros iniciales en la clave negociada (consulte la Sección 7.4.1)?

- ¿Su cliente TLS verifica que los parámetros Diffie-Hellman enviados por el servidor son aceptables (ver Sección 4.2.8.1)?

- ¿Utiliza un aleatorio fuerte y, lo más importante, correctamente sembrado? generador de números (consulte el Apéndice C.1) al generar valores privados Diffie-Hellman, el parámetro ECDSA "k" y otros valores críticos para la seguridad? Se RECOMIENDA que las implementaciones implementen "ECDSA determinista" como se especifica en [RFC6979].

- ¿Ajusta a cero los valores de clave pública de Diffie-Hellman y comparte secretos del tamaño del grupo (ver Sección 4.2.8.1 y Sección 7.4.1)?

- ¿Verifica las firmas después de hacerlas, para protegerse contra filtraciones de claves RSA-CRT [FW15]?

Machine Translated by Google

C.4. Prevención de seguimiento de clientes

Los clientes NO DEBEN reutilizar un boleto para múltiples conexiones. La reutilización de un ticket permite a los observadores pasivos correlacionar diferentes conexiones.
Los servidores que emiten boletos DEBERÍAN ofrecer al menos tantos boletos como el número de conexiones que un cliente podría usar; por ejemplo, un navegador web que utilice HTTP/1.1 [RFC7230] podría abrir seis conexiones a un servidor. Los servidores DEBERÍAN emitir nuevos boletos con cada conexión.

Esto asegura que los clientes siempre puedan usar un nuevo ticket al crear una nueva conexión.

C.5. Operación no autenticada

Las versiones anteriores de TLS ofrecían conjuntos de cifrado explícitamente no autenticados basados en Diffie-Hellman anónimo. Estos modos han quedado obsoletos en TLS 1.3. Sin embargo, aún es posible negociar parámetros que no proporcionen una autenticación de servidor verificable mediante varios métodos, entre ellos:

- Claves públicas sin procesar [RFC7250].

- Utilizar una clave pública contenida en un certificado pero sin validación de la cadena de certificados o cualquiera de sus contenidos.

Cualquiera de las dos técnicas usadas solas es vulnerable a los ataques de intermediarios y, por lo tanto, no es segura para el uso general. Sin embargo, también es posible vincular dichas conexiones a un mecanismo de autenticación externo a través de la validación fuera de banda de la clave pública del servidor, confianza en el primer uso o un mecanismo como enlaces de canal (aunque los enlaces de canal descritos en [RFC5929] no están definidos para TLS 1.3). Si no se utiliza tal mecanismo, entonces la conexión no tiene protección contra el ataque de intermediario activo; las aplicaciones NO DEBEN usar TLS de tal manera que no exista una configuración explícita o un perfil de aplicación específico.

Machine Translated by Google

Apéndice D. Compatibilidad con versiones anteriores

El protocolo TLS proporciona un mecanismo integrado para la negociación de versiones entre puntos finales que posiblemente admitan diferentes versiones de TLS.

TLS 1.x y SSL 3.0 usan mensajes ClientHello compatibles. Los servidores también pueden manejar clientes que intentan usar versiones futuras de TLS siempre que el formato ClientHello siga siendo compatible y haya al menos una versión de protocolo compatible tanto con el cliente como con el servidor.

Las versiones anteriores de TLS usaban el número de versión de la capa de registro (TLSPlaintext.legacy_record_version y TLSCiphertext.legacy_record_version) para varios propósitos. A partir de TLS 1.3, este campo está obsoleto. Todas las implementaciones DEBEN ignorar el valor de TLSPlaintext.legacy_record_version. El valor de TLSCiphertext.legacy_record_version se incluye en los datos adicionales para la desprotección pero, de lo contrario, PUEDE ignorarse o PUEDE validarse para que coincida con el valor constante fijo.

La negociación de la versión se realiza utilizando solo las versiones de protocolo de enlace (ClientHello.legacy_version y ServerHello.legacy_version, así como las extensiones ClientHello, HelloRetryRequest y ServerHello "supported_versions"). Para maximizar la interoperabilidad con puntos finales más antiguos, las implementaciones que negocian el uso de TLS 1.0-1.2 DEBERÍAN establecer el número de versión de la capa de registro en la versión negociada para ServerHello y todos los registros posteriores.

Para lograr la máxima compatibilidad con comportamientos previamente no estándar e implementaciones mal configuradas, todas las implementaciones DEBERÍAN admitir la validación de las rutas de certificación en función de las expectativas de este documento, incluso cuando se manejen protocolos de enlace de versiones anteriores de TLS (consulte la Sección 4.4.2.2).

TLS 1.2 y versiones anteriores admitían una extensión de "Secreto maestro extendido" [RFC7627] que digería gran parte de la transcripción del protocolo de enlace en el secreto maestro. Debido a que TLS 1.3 siempre genera hash en la transcripción hasta que el servidor finaliza, las implementaciones que admiten tanto TLS 1.3 como versiones anteriores DEBERÍAN indicar el uso de la extensión Extended Master Secret en sus API cada vez que se usa TLS 1.3.

Machine Translated by Google

D.1. Negociar con un servidor antiguo

Un cliente TLS 1.3 que desea negociar con servidores que no admiten TLS 1.3 enviará un ClientHello TLS 1.3 normal que contiene 0x0303 (TLS 1.2) en ClientHello.legacy_version pero con las versiones correctas en la extensión "supported_versions". Si el servidor no es compatible con TLS 1.3, responderá con un ServerHello que contiene un número de versión anterior. Si el cliente acepta utilizar esta versión, la negociación procederá según corresponda para el protocolo negociado.

Un cliente que utilice un ticket para la reanudación DEBERÍA iniciar la conexión utilizando la versión que se negoció previamente.

Tenga en cuenta que los datos 0-RTT no son compatibles con servidores más antiguos y NO DEBEN enviarse si no se sabe que el servidor es compatible con TLS 1.3.
Ver Apéndice D.3.

Si la versión elegida por el servidor no es compatible con el cliente (o no es aceptable), el cliente DEBE cancelar el protocolo de enlace con una alerta de "protocol_version".

Se sabe que algunas implementaciones de servidores heredados no implementan la especificación TLS correctamente y pueden anular las conexiones al encontrar extensiones o versiones de TLS que no conocen.

La interoperabilidad con servidores con errores es un tema complejo que va más allá del alcance de este documento. Es posible que se requieran varios intentos de conexión para negociar una conexión compatible con versiones anteriores; sin embargo, esta práctica es vulnerable a los ataques de degradación y NO SE RECOMIENDA.

D.2. Negociar con un cliente antiguo

Un servidor TLS también puede recibir un ClientHello que indica un número de versión más pequeño que su versión compatible más alta. Si la extensión "supported_versions" está presente, el servidor DEBE negociar usando esa extensión como se describe en la Sección 4.2.1. Si la extensión "supported_versions" no está presente, el servidor DEBE negociar el mínimo de ClientHello.legacy_version y TLS 1.2. Por ejemplo, si el servidor admite TLS 1.0, 1.1 y 1.2, y legacy_version es TLS 1.0, el servidor procederá con TLS 1.0 ServerHello. Si la extensión "supported_versions" está ausente y el servidor solo admite versiones superiores a ClientHello.legacy_version, el servidor DEBE cancelar el protocolo de enlace con una alerta de "protocol_version".

Tenga en cuenta que las versiones anteriores de TLS no especificaban claramente el valor del número de versión de la capa de registro en todos los casos (TLSPlaintext.legacy_record_version). Los servidores recibirán varias versiones de TLS 1.x en este campo, pero su valor siempre DEBE ignorarse.

Machine Translated by Google

D.3. Compatibilidad con versiones anteriores de 0-RTT

Los datos 0-RTT no son compatibles con servidores más antiguos. Un servidor anterior responderá a ClientHello con un ServerHello anterior, pero no omitirá correctamente los datos 0-RTT y no podrá completar el protocolo de enlace. Esto puede causar problemas cuando un cliente intenta usar 0-RTT, particularmente en implementaciones de varios servidores. Por ejemplo, una implementación podría implementar TLS 1.3 gradualmente con algunos servidores implementando TLS 1.3 y algunos implementando TLS 1.2, o una implementación de TLS 1.3 podría degradarse a TLS 1.2.

Un cliente que intente enviar datos 0-RTT DEBE fallar en la conexión si recibe un ServerHello con TLS 1.2 o anterior. Luego puede volver a intentar la conexión con 0-RTT deshabilitado. Para evitar un ataque de degradación, el cliente NO DEBE deshabilitar TLS 1.3, solo 0-RTT.

Para evitar esta condición de error, las implementaciones de varios servidores DEBERÍAN garantizar una implementación uniforme y estable de TLS 1.3 sin 0-RTT antes de habilitar 0-RTT.

D.4. Modo de compatibilidad de caja intermedia

Las mediciones de campo [Ben17a] [Ben17b] [Res17a] [Res17b] han encontrado que una cantidad significativa de cajas intermedias se comportan mal cuando un par de cliente/servidor TLS negocia TLS 1.3. Las implementaciones pueden aumentar la posibilidad de realizar conexiones a través de esos middleboxes al hacer que el protocolo de enlace TLS 1.3 se parezca más a un protocolo de enlace TLS 1.2:

- El cliente siempre proporciona una ID de sesión no vacía en el ClientHello, como se describe en la sección legacy_session_id de la Sección 4.1.2.

- Si no ofrece datos anticipados, el cliente envía un dummy registro change_cipher_spec (ver el tercer párrafo de la Sección 5) inmediatamente antes de su segundo vuelo. Esto puede ser antes de su segundo ClientHello o antes de su vuelo de protocolo de enlace cifrado.

  Si ofrece datos iniciales, el registro se coloca inmediatamente después del primer ClientHello.

- El servidor envía un registro ficticio change_cipher_spec inmediatamente después de su primer mensaje de reconocimiento. Esto puede ser después de un ServerHello o un HelloRetryRequest.

Cuando se juntan, estos cambios hacen que el protocolo de enlace TLS 1.3 se asemeje a la reanudación de sesión de TLS 1.2, lo que mejora la posibilidad de conectarse con éxito a través de middleboxes. Este "modo de compatibilidad" se negocia parcialmente: el cliente puede optar por proporcionar una identificación de sesión o no, y el servidor debe repetirla. Cualquiera de los lados puede enviar

Machine Translated by Google

change_cipher_spec en cualquier momento durante el protocolo de enlace, ya que el par debe ignorarlos, pero si el cliente envía una ID de sesión no vacía, el servidor DEBE enviar change_cipher_spec como se describe en este apéndice.

D.5. Restricciones de seguridad relacionadas con la compatibilidad con versiones anteriores

Las implementaciones que negocien el uso de versiones anteriores de TLS DEBERÍAN preferir los conjuntos de cifrado AEAD y secreto directo, cuando estén disponibles.

La seguridad de los conjuntos de cifrado RC4 se considera insuficiente por las razones citadas en [RFC7465]. Las implementaciones NO DEBEN ofrecer ni negociar conjuntos de cifrado RC4 para ninguna versión de TLS por ningún motivo.

Las versiones antiguas de TLS permitían el uso de cifrados de muy baja potencia.
Los cifrados con una fuerza inferior a 112 bits NO DEBEN ofrecerse ni negociarse para ninguna versión de TLS por ningún motivo.

La seguridad de SSL 3.0 [RFC6101] se considera insuficiente por las razones enumeradas en [RFC7568], y NO DEBE negociarse para
cualquier razón.

La seguridad de SSL 2.0 [SSL2] se considera insuficiente por las razones enumeradas en [RFC6176], y NO DEBE negociarse para
cualquier razón.

Las implementaciones NO DEBEN enviar un CLIENTE-HOLA compatible con SSL versión 2.0. Las implementaciones NO DEBEN negociar TLS 1.3 o posterior utilizando un CLIENTE-HOLA compatible con SSL versión 2.0. NO SE RECOMIENDAN las implementaciones para aceptar un CLIENTE-HOLA compatible con la versión 2.0 de SSL para poder negociar versiones anteriores de TLS.

Las implementaciones NO DEBEN enviar ClientHello.legacy_version o ServerHello.legacy_version establecidos en 0x0300 o menos. Cualquier extremo que reciba un mensaje de saludo con ClientHello.legacy_version o ServerHello.legacy_version establecido en 0x0300 DEBE cancelar el protocolo de enlace con una alerta de "protocol_version".

Las implementaciones NO DEBEN enviar ningún registro con una versión inferior a 0x0300. Las implementaciones NO DEBEN aceptar registros con una versión inferior a 0x0300 (pero pueden hacerlo sin darse cuenta si el número de versión del registro se ignora por completo).

Las implementaciones NO DEBEN utilizar la extensión HMAC truncada, definida en la Sección 7 de [RFC6066], ya que no es aplicable a los algoritmos AEAD y se ha demostrado que es insegura en algunos escenarios.

Machine Translated by Google

Apéndice E. Descripción general de las propiedades de seguridad

Un análisis de seguridad completo de TLS está fuera del alcance de este documento. En este apéndice, proporcionamos una descripción informal de las propiedades deseadas, así como referencias a trabajos más detallados en la literatura de investigación que proporciona definiciones más formales.

Cubrimos las propiedades del apretón de manos por separado de las de la capa de registro.

E.1. Apretón de manos

El protocolo de enlace TLS es un protocolo de intercambio de claves autenticadas (AKE) que está diseñado para proporcionar una funcionalidad autenticada unidireccional (solo servidor) y mutuamente autenticada (cliente y servidor). Al finalizar el apretón de manos, cada lado muestra su vista de los siguientes valores:

- Un conjunto de "claves de sesión" (los diversos secretos derivados de la secreto maestro) del que se puede derivar un conjunto de claves de trabajo.

- Un conjunto de parámetros criptográficos (algoritmos, etc.).

- Las identidades de los comunicantes.

Suponemos que el atacante es un atacante de red activo, lo que significa que tiene control total sobre la red utilizada para comunicarse entre las partes [RFC3552]. Incluso en estas condiciones, el apretón de manos debe proporcionar las propiedades que se enumeran a continuación. Tenga en cuenta que estas propiedades no son necesariamente independientes, sino que reflejan las necesidades de los consumidores del protocolo.

Establecimiento de las mismas claves de sesión: el protocolo de enlace debe generar el mismo conjunto de claves de sesión en ambos lados del protocolo de enlace, siempre que se complete correctamente en cada punto final (consulte [CK01], Definición 1, parte 1).

Secreto de las claves de sesión: Las claves de sesión compartidas deben ser conocidas solo por las partes que se comunican y no por el atacante (ver [CK01], Definición 1, parte 2). Tenga en cuenta que en una conexión autenticada unilateralmente, el atacante puede establecer sus propias claves de sesión con el servidor, pero esas claves de sesión son distintas de las establecidas por el cliente.

Autenticación de pares: la vista del cliente de la identidad de pares debe reflejar la identidad del servidor. Si el cliente está autenticado, la vista del servidor de la identidad del par debe coincidir con la identidad del cliente.

Machine Translated by Google

Unicidad de las claves de sesión: dos apretones de manos distintos deben producir claves de sesión distintas y no relacionadas. Las claves de sesión individuales producidas por un apretón de manos también deben ser distintas e independientes.

Protección de downgrade: Los parámetros criptográficos deben ser los mismo en ambos lados y debe ser el mismo que si los compañeros se hubieran estado comunicando en ausencia de un ataque (ver [BBFGKZ16], Definiciones 8 y 9).

Secreto de reenvío con respecto a las claves a largo plazo: si el el material de clave (en este caso, las claves de firma en los modos de autenticación basados en certificados o el PSK externo/de reanudación en PSK con modos (EC)DHE) se ve comprometido una vez que se completa el protocolo de enlace, esto no compromete la seguridad de la clave de sesión (ver [DOW92]), siempre que se haya borrado la propia clave de sesión.

La propiedad de confidencialidad directa no se cumple cuando se usa PSK en el PskKeyExchangeMode "psk_ke".

Resistencia a la suplantación de compromiso clave (KCI): En una conexión autenticada con certificados, comprometer el secreto a largo plazo de un actor no debería romper la autenticación de ese actor de su par en la conexión dada (consulte [HGFS15]). Por ejemplo, si la clave de firma de un cliente se ve comprometida, no debería ser posible hacerse pasar por servidores arbitrarios para ese cliente en protocolos de enlace posteriores.

Protección de identidades de punto final: la identidad del servidor (certificado) debe estar protegido contra atacantes pasivos. La identidad del cliente debe estar protegida contra atacantes tanto pasivos como activos.

De manera informal, los modos basados en firmas de TLS 1.3 permiten el establecimiento de una clave única, secreta y compartida establecida mediante un intercambio de claves (EC)DHE y autenticada por la firma del servidor sobre la transcripción del protocolo de enlace, así como vinculada a la identidad del servidor. por un MAC. Si el cliente está autenticado por un certificado, también firma la transcripción del protocolo de enlace y proporciona un MAC vinculado a ambas identidades. [SIGMA] describe el diseño y análisis de este tipo de protocolo de intercambio de claves. Si se utilizan claves (EC)DHE nuevas para cada conexión, las claves de salida son secretas hacia adelante.

El PSK externo y el PSK de reanudación se inician desde un secreto compartido a largo plazo en un conjunto único por conexión de claves de sesión a corto plazo.
Este secreto puede haberse establecido en un apretón de manos anterior. Si se utiliza PSK con establecimiento de clave (EC)DHE, estas claves de sesión también serán secretas hacia adelante. El PSK de reanudación se ha diseñado para que el secreto maestro de reanudación calculado por la conexión N y necesario para formar la conexión N+1 sea independiente de las claves de tráfico utilizadas por

Machine Translated by Google

conexión N, proporcionando así confidencialidad directa entre las conexiones.
Además, si se establecen varios boletos en la misma conexión, se asocian con diferentes claves, por lo que el compromiso del PSK asociado con un boleto no implica el compromiso de las conexiones establecidas con los PSK asociados con otros boletos.

Esta propiedad es más interesante si los tickets se almacenan en una base de datos (y, por lo tanto, se pueden eliminar) en lugar de si están autocifrados.

El valor del enlace PSK forma un enlace entre un PSK y el protocolo de enlace actual, así como entre la sesión en la que se estableció el PSK y la sesión actual. Este enlace incluye transitivamente la transcripción del protocolo de enlace original, porque esa transcripción se digiere en los valores que producen el secreto maestro de reanudación. Esto requiere que tanto el KDF utilizado para producir el secreto maestro de reanudación como el MAC utilizado para calcular el enlazador sean resistentes a colisiones. Consulte el Apéndice E.1.1 para obtener más información al respecto. Nota: El binder no cubre los valores de binder de otros PSK, aunque están incluidos en el MAC terminado.

Actualmente, TLS no permite que el servidor envíe un mensaje de solicitud de certificado en protocolos de enlace no basados en certificados (por ejemplo, PSK). Si esta restricción se flexibilizara en el futuro, la firma del cliente no cubriría directamente el certificado del servidor.

Sin embargo, si el PSK se estableció a través de un NewSessionTicket, la firma del cliente cubriría transitivamente el certificado del servidor a través del enlace PSK. [PSK-FINISHED] describe un ataque concreto a las construcciones que no se unen al certificado del servidor (ver también [Kraw16]). No es seguro utilizar la autenticación de cliente basada en certificados cuando el cliente podría compartir potencialmente el mismo par PSK/key-id con dos terminales diferentes. Las implementaciones NO DEBEN combinar PSK externos con autenticación basada en certificados del cliente o del servidor, a menos que alguna extensión lo negocie.

Si se utiliza un exportador, produce valores que son únicos y secretos (porque se generan a partir de una clave de sesión única).
Los exportadores calculados con diferentes etiquetas y contextos son computacionalmente independientes, por lo que no es factible calcular uno a partir de otro o el secreto de sesión a partir del valor exportado.
Nota: Los exportadores pueden producir valores de longitud arbitraria; si los exportadores se van a utilizar como enlaces de canal, el valor exportado DEBE ser lo suficientemente grande como para proporcionar resistencia a la colisión. Los exportadores proporcionados en TLS 1.3 se derivan de los mismos contextos de protocolo de enlace que las claves de tráfico anteriores y las claves de tráfico de la aplicación, respectivamente, y por lo tanto tienen propiedades de seguridad similares. Tenga en cuenta que no incluyen el certificado del cliente; Es posible que las aplicaciones futuras que deseen vincularse al certificado del cliente deban definir un nuevo exportador que incluya la transcripción completa del protocolo de enlace.

Machine Translated by Google

Para todos los modos de protocolo de enlace, el MAC finalizado (y, si está presente, la firma) evita los ataques de degradación. Además, el uso de ciertos bytes en los nonces aleatorios como se describe en la Sección 4.1.3 permite la detección de versiones anteriores de TLS. Consulte [BBFGKZ16] para obtener más detalles sobre TLS 1.3 y la versión anterior.

Tan pronto como el cliente y el servidor hayan intercambiado suficiente información para establecer claves compartidas, el resto del protocolo de enlace se cifra, lo que brinda protección contra atacantes pasivos, incluso si la clave compartida calculada no está autenticada. Debido a que el servidor se autentica antes que el cliente, el cliente puede asegurarse de que si se autentica en el servidor, solo revela su identidad a un servidor autenticado. Tenga en cuenta que las implementaciones deben utilizar el mecanismo de relleno de registro proporcionado durante el protocolo de enlace para evitar la fuga de información sobre las identidades debido a la longitud. Las identidades PSK propuestas por el cliente no están cifradas, ni tampoco la que selecciona el servidor.

E.1.1. Derivación de claves y HKDF

La derivación de claves en TLS 1.3 usa HKDF como se define en [RFC5869] y sus dos componentes, HKDF-Extract y HKDF-Expand. El fundamento completo de la construcción HKDF se puede encontrar en [Kraw10] y el fundamento de la forma en que se utiliza en TLS 1.3 en [KW16]. A lo largo de este documento, cada aplicación de HKDF-Extract va seguida de una o más invocaciones de HKDF-Expand. Siempre se debe seguir este orden (incluso en futuras revisiones de este documento); en particular, uno NO DEBE usar una salida de HKDF-Extract como entrada a otra aplicación de HKDF-Extract sin un HKDF-Expand en el medio.

Se permiten múltiples aplicaciones de HKDF-Expand a algunas de las mismas entradas siempre que se diferencien a través de la clave y/o las etiquetas.

Tenga en cuenta que HKDF-Expand implementa una función pseudoaleatoria (PRF) con entradas y salidas de longitud variable. En algunos de los usos de HKDF en este documento (p. ej., para generar exportadores y resuming_master_secret), es necesario que la aplicación de HKDF-Expand sea resistente a colisiones; es decir, no debería ser factible encontrar dos entradas diferentes para HKDF-Expand que generen el mismo valor. Esto requiere que la función hash subyacente sea resistente a colisiones y que la longitud de salida de HKDF-Expand tenga un tamaño de al menos 256 bits (o tanto como sea necesario para que la función hash evite encontrar colisiones).

Machine Translated by Google

E.1.2. Autenticación del cliente

Un cliente que ha enviado datos de autenticación a un servidor, ya sea durante el protocolo de enlace o en la autenticación posterior al protocolo de enlace, no puede estar seguro de si el servidor luego considera que el cliente está autenticado o no. Si el cliente necesita determinar si el servidor considera que la conexión se autentica unilateral o mutuamente, la capa de aplicación debe proporcionarlo.

Consulte [CHHSV17] para obtener más detalles. Además, el análisis de la autenticación posterior al protocolo de enlace de [Kraw16] muestra que el cliente identificado por el certificado enviado en la fase posterior al protocolo de enlace posee la clave de tráfico. Por lo tanto, esta parte es el cliente que participó en el protocolo de enlace original o aquel en quien el cliente original delegó la clave de tráfico (suponiendo que la clave de tráfico no se haya visto comprometida).

E.1.3. 0-RTT

El modo de operación 0-RTT generalmente brinda propiedades de seguridad similares a las de los datos 1-RTT, con las dos excepciones de que las claves de cifrado 0-RTT no brindan total confidencialidad directa y que el servidor no puede garantizar la unicidad del protocolo de enlace. (no rejugabilidad) sin mantener cantidades de estado potencialmente indebidas. Consulte la Sección 8 para conocer los mecanismos para limitar la exposición a la reproducción.

E.1.4. Independencia del exportador

exporter_master_secret y early_exporter_master_secret se derivan para ser independientes de las claves de tráfico y, por lo tanto, no representan una amenaza para la seguridad del tráfico cifrado con esas claves. Sin embargo, debido a que estos secretos se pueden usar para calcular cualquier valor de exportador, DEBERÍAN borrarse lo antes posible. Si se conoce el conjunto total de etiquetas del exportador, las implementaciones DEBERÍAN precalcular la etapa interna Derive-Secret del cálculo del exportador para todas esas etiquetas, luego borrar el [early_]exporter_master_secret, seguido de cada valor interno tan pronto como se conozca que no se volverá a necesitar.

E.1.5. Seguridad posterior al compromiso

TLS no proporciona seguridad para los protocolos de enlace que tienen lugar después de que se compromete el secreto a largo plazo del par (clave de firma o PSK externo). Por lo tanto, no proporciona seguridad posterior al compromiso [CCG16], a veces también denominada secreto hacia atrás o futuro.

Esto contrasta con la resistencia de KCI, que describe las garantías de seguridad que tiene una parte después de que su propio secreto a largo plazo se ha visto comprometido.

Machine Translated by Google

E.1.6. Referencias externas

El lector debe consultar las siguientes referencias para el análisis del protocolo de enlace TLS: [DFGS15], [CHSV16], [DFGS16], [KW16], [Kraw16], [FGSW16], [LXZFH16], [FG17] y [BBK17].

E.2. Capa de registro

La capa de registro depende del protocolo de enlace que produce secretos de tráfico sólidos que se pueden usar para derivar claves de cifrado bidireccionales y nonces. Suponiendo que eso sea cierto y que las claves no se utilicen para más datos que los indicados en la Sección 5.5, entonces la capa de registro debería proporcionar las siguientes garantías:

Confidencialidad: Un atacante no debe ser capaz de determinar la contenido de texto plano de un registro dado.

Integridad: un atacante no debería poder crear un nuevo registro que es diferente de un registro existente que será aceptado por el receptor.

Orden de protección/no repetibilidad: un atacante no debe poder hacer que el receptor acepte un registro que ya ha aceptado o hacer que el receptor acepte el registro N+1 sin haber procesado primero el registro N.

Ocultación de longitud: dado un registro con una longitud externa dada, el atacante no debería poder determinar la cantidad del registro que es contenido versus relleno.

Secreto de reenvío después del cambio de clave: si la clave de tráfico se actualiza Si se utiliza el mecanismo descrito en la Sección 4.6.3 y se elimina la clave de la generación anterior, un atacante que comprometa el punto final no debería poder descifrar el tráfico cifrado con la clave anterior.

De manera informal, TLS 1.3 proporciona estas propiedades mediante la protección AEAD del texto sin formato con una clave segura. El cifrado AEAD [RFC5116] proporciona confidencialidad e integridad a los datos. La no repetibilidad se proporciona mediante el uso de un nonce separado para cada registro, derivándose el nonce del número de secuencia del registro (Sección 5.3), y el número de secuencia se mantiene de forma independiente en ambos lados; por lo tanto, los registros que se entregan desordenados dan como resultado fallas en la desprotección de AEAD. Para evitar el criptoanálisis masivo cuando diferentes usuarios cifran repetidamente el mismo texto sin formato con la misma clave (como suele ser el caso de HTTP), el nonce se forma mezclando

Machine Translated by Google

el número de secuencia con un vector secreto de inicialización por conexión derivado junto con las claves de tráfico. Ver [BT16] para el análisis de esta construcción.

La técnica de cambio de clave en TLS 1.3 (consulte la Sección 7.2) sigue la construcción del generador en serie como se explica en [REKEY], que muestra que el cambio de clave puede permitir que las claves se usen para una mayor cantidad de cifrados que sin el cambio de clave. Esto se basa en la seguridad de la función HKDF-Expand-Label como función pseudoaleatoria (PRF). Además, siempre que esta función sea realmente unidireccional, no es posible calcular las claves de tráfico antes de un cambio de clave (secreto hacia adelante).

TLS no brinda seguridad para los datos que se comunican en una conexión después de que se compromete un secreto de tráfico de esa conexión.
Es decir, TLS no proporciona seguridad posterior al compromiso/secreto futuro/secreto hacia atrás con respecto al secreto de tráfico. De hecho, un atacante que aprende un secreto de tráfico puede calcular todos los secretos de tráfico futuros en esa conexión. Los sistemas que desean tales garantías necesitan hacer un nuevo apretón de manos y establecer una nueva conexión con un intercambio (EC)DHE.

E.2.1. Referencias externas

El lector debe consultar las siguientes referencias para el análisis de la capa de registro TLS: [BMMRT15], [BT16], [BDFKPPRSZZ16], [BBK17] y [PS18].

E.3. Análisis de tráfico

TLS es susceptible a una variedad de ataques de análisis de tráfico basados en la observación de la longitud y el tiempo de los paquetes cifrados [CLINIC]
[HCJC16]. Esto es particularmente fácil cuando hay un pequeño conjunto de posibles mensajes a distinguir, como para un servidor de video que aloja un corpus fijo de contenido, pero aún proporciona información utilizable incluso en escenarios más complicados.

TLS no proporciona ninguna defensa específica contra esta forma de ataque, pero incluye un mecanismo de relleno para que lo usen las aplicaciones: El texto sin formato protegido por la función AEAD consta de contenido más relleno de longitud variable, lo que permite que la aplicación produzca registros cifrados de longitud arbitraria. así como el tráfico de cobertura de solo relleno para ocultar la diferencia entre los períodos de transmisión y los períodos de silencio. Debido a que el relleno está encriptado junto con el contenido real, un atacante no puede determinar directamente la longitud del relleno, pero puede medirlo indirectamente mediante el uso de canales de tiempo expuestos durante el procesamiento de registros (es decir, ver cuánto tiempo lleva procesar un registro). o goteando en registros para ver

Machine Translated by Google

cuáles obtienen una respuesta del servidor). En general, no se sabe cómo eliminar todos estos canales porque incluso una función de eliminación de relleno de tiempo constante probablemente alimentará el contenido a funciones dependientes de datos. Como mínimo, un servidor o cliente de tiempo completamente constante requeriría una estrecha cooperación con la implementación del protocolo de la capa de aplicación, lo que incluye hacer que ese protocolo de nivel superior sea de tiempo constante.

Nota: Es probable que las sólidas defensas de análisis de tráfico conduzcan a un rendimiento inferior debido a los retrasos en la transmisión de paquetes y al aumento del volumen de tráfico.

E.4. Ataques de canal lateral

En general, TLS no tiene defensas específicas contra ataques de canal lateral (es decir, aquellos que atacan las comunicaciones a través de canales secundarios como el tiempo), dejándolos a la implementación de las primitivas criptográficas relevantes. Sin embargo, ciertas características de TLS están diseñadas para facilitar la escritura de código resistente al canal lateral:

- A diferencia de las versiones anteriores de TLS que usaban una estructura compuesta MAC-then-encrypt, TLS 1.3 solo usa algoritmos AEAD, lo que permite que las implementaciones usen implementaciones autónomas en tiempo constante de esas primitivas.

- TLS utiliza una alerta uniforme "bad_record_mac" para todos los errores de descifrado, que tiene como objetivo evitar que un atacante obtenga una visión parcial de partes del mensaje. Se proporciona resistencia adicional al terminar la conexión en tales errores; una nueva conexión tendrá material criptográfico diferente, previniendo ataques contra las primitivas criptográficas que requieren múltiples intentos.

La fuga de información a través de canales laterales puede ocurrir en capas por encima de TLS, en los protocolos de aplicación y en las aplicaciones que los utilizan.
La resistencia a los ataques de canal lateral depende de las aplicaciones y los protocolos de aplicación por separado, lo que garantiza que la información confidencial no se filtre inadvertidamente.

Machine Translated by Google

E.5. Reproducir ataques en 0-RTT

Los datos 0-RTT reproducibles presentan una serie de amenazas de seguridad para las aplicaciones que usan TLS, a menos que esas aplicaciones estén diseñadas específicamente para ser seguras en la reproducción (mínimamente, esto significa idempotentes, pero en muchos casos también pueden requerir otras condiciones más fuertes, como constante- tiempo de respuesta). Los posibles ataques incluyen:

- Duplicación de acciones que provocan efectos secundarios (p. ej., comprar un artículo o transferir dinero) para que se dupliquen, perjudicando así al sitio o al usuario.

- Los atacantes pueden almacenar y reproducir mensajes 0-RTT para reordenarlos con respecto a otros mensajes (p. ej., mover una eliminación después de una creación).

- Explotar el comportamiento del tiempo de caché para descubrir el contenido de los mensajes 0-RTT al reproducir un mensaje 0-RTT en un nodo de caché diferente y luego usar una conexión separada para medir la latencia de la solicitud, para ver si las dos solicitudes se dirigen al mismo recurso.

Si los datos se pueden reproducir una gran cantidad de veces, es posible realizar ataques adicionales, como realizar mediciones repetidas de la velocidad de las operaciones criptográficas. Además, pueden sobrecargar los sistemas de limitación de velocidad. Para obtener una descripción más detallada de estos ataques, consulte [Mac17].

En última instancia, los servidores tienen la responsabilidad de protegerse contra los ataques que emplean la replicación de datos 0-RTT. Los mecanismos descritos en la Sección 8 están destinados a evitar la reproducción en la capa TLS, pero no brindan una protección completa contra la recepción de múltiples copias de los datos del cliente. TLS 1.3 recurre al protocolo de enlace 1-RTT cuando el servidor no tiene ninguna información sobre el cliente, por ejemplo, porque está en un clúster diferente que no comparte el estado o porque el ticket se eliminó como se describe en la Sección 8.1. Si el protocolo de capa de aplicación retransmite datos en esta configuración, entonces es posible que un atacante induzca la duplicación de mensajes enviando ClientHello tanto al clúster original (que procesa los datos inmediatamente) como a otro clúster que recurrirá a 1-RTT. y procesar los datos en la reproducción de la capa de aplicación. La escala de este ataque está limitada por la voluntad del cliente de volver a intentar transacciones y, por lo tanto, solo permite una cantidad limitada de duplicación, y cada copia aparece como una nueva conexión en el servidor.

Machine Translated by Google

Si se implementan correctamente, los mecanismos descritos en las Secciones 8.1 y 8.2 evitan que un ClientHello reproducido y sus datos 0-RTT asociados sean aceptados varias veces por cualquier clúster con un estado consistente; para servidores que limitan el uso de 0-RTT a un clúster para un solo ticket, entonces un ClientHello determinado y sus datos 0-RTT asociados solo se aceptarán una vez. Sin embargo, si el estado no es completamente coherente, un atacante podría aceptar varias copias de los datos durante la ventana de replicación. Debido a que los clientes no conocen los detalles exactos del comportamiento del servidor, NO DEBEN enviar mensajes en datos iniciales cuya reproducción no sea segura y que no estarían dispuestos a volver a intentar a través de múltiples conexiones 1-RTT.

Los protocolos de aplicación NO DEBEN utilizar datos 0-RTT sin un perfil que defina su uso. Ese perfil debe identificar qué mensajes o interacciones son seguros para usar con 0-RTT y cómo manejar la situación cuando el servidor rechaza 0-RTT y recurre a 1-RTT.

Además, para evitar el uso indebido accidental, las implementaciones de TLS NO DEBEN habilitar 0-RTT (ya sea para enviar o aceptar) a menos que la aplicación lo solicite específicamente y NO DEBEN reenviar automáticamente los datos 0-RTT si el servidor los rechaza, a menos que así lo indique la aplicación. Es posible que las aplicaciones del lado del servidor deseen implementar un procesamiento especial para datos 0-RTT para algunos tipos de tráfico de aplicaciones (p. ej., anular la conexión, solicitar que los datos se vuelvan a enviar en la capa de la aplicación o retrasar el procesamiento hasta que se complete el protocolo de enlace). Para permitir que las aplicaciones implementen este tipo de procesamiento, las implementaciones de TLS DEBEN proporcionar una forma para que la aplicación determine si el protocolo de enlace se ha completado.

E.5.1. Reproducción y Exportadores

Las repeticiones de ClientHello producen el mismo exportador inicial, lo que requiere un cuidado adicional por parte de las aplicaciones que utilizan estos exportadores.
En particular, si estos exportadores se utilizan como un enlace de canal de autenticación (por ejemplo, firmando la salida del exportador), un atacante que comprometa el PSK puede trasplantar autenticadores entre conexiones sin comprometer la clave de autenticación.

Además, el exportador temprano NO DEBE usarse para generar claves de cifrado de servidor a cliente porque eso implicaría la reutilización de esas claves. Esto es paralelo al uso de las primeras claves de tráfico de aplicaciones solo en la dirección de cliente a servidor.

Machine Translated by Google

E.6. Exposición de identidad de PSK

Debido a que las implementaciones responden a un enlace de PSK no válido abortando el protocolo de enlace, es posible que un atacante verifique si una identidad de PSK determinada es válida. Específicamente, si un servidor acepta protocolos de enlace de PSK externos y protocolos de enlace basados en certificados, una identidad de PSK válida dará como resultado un protocolo de enlace fallido, mientras que una identidad no válida simplemente se omitirá y dará como resultado un protocolo de enlace de certificado exitoso. Los servidores que solo admiten protocolos de enlace de PSK pueden resistir esta forma de ataque al tratar los casos en los que no hay una identidad de PSK válida y en los que hay una identidad pero tiene un enlace no válido de manera idéntica.

E.7. Compartir PSK

TLS 1.3 adopta un enfoque conservador para los PSK al vincularlos a un KDF específico. Por el contrario, TLS 1.2 permite usar PSK con cualquier función hash y TLS 1.2 PRF. Por lo tanto, cualquier PSK que se use tanto con TLS 1.2 como con TLS 1.3 debe usarse con un solo hash en TLS 1.3, lo que no es óptimo si los usuarios desean aprovisionar un solo PSK.

Las construcciones en TLS 1.2 y TLS 1.3 son diferentes, aunque ambos se basan en HMAC. Si bien no existe una forma conocida en la que el mismo PSK pueda producir resultados relacionados en ambas versiones, solo se ha realizado un análisis limitado. Las implementaciones pueden garantizar la seguridad de la salida relacionada con protocolos cruzados al no reutilizar PSK entre TLS 1.3 y TLS 1.2.

E.8. Ataques a Static RSA

Aunque TLS 1.3 no utiliza el transporte de claves RSA y, por lo tanto, no es directamente susceptible a ataques de tipo Bleichenbacher [Blei98], si los servidores TLS 1.3 también admiten RSA estático en el contexto de versiones anteriores de TLS, entonces es posible suplantar el servidor. para conexiones TLS 1.3 [JSS15]. Las implementaciones de TLS 1.3 pueden prevenir este ataque al deshabilitar la compatibilidad con RSA estático en todas las versiones de TLS. En principio, las implementaciones también podrían ser capaces de separar certificados con diferentes bits de uso de clave para el descifrado RSA estático y la firma RSA, pero esta técnica se basa en que los clientes se nieguen a aceptar firmas usando claves en certificados que no tienen el bit digitalSignature establecido, y muchos clientes sí lo tienen. no hacer cumplir esta restricción.

Machine Translated by Google

Colaboradores

Martín Abadi
Universidad de California, Santa Cruz inmortal@cs.ucsc.edu

Christopher Allen (coeditor
de TLS 1.0)
Alacrity Ventures
ChristopherA@AlacrityManagement.com

Richard Barnes Cisco
rlb@ipv.sx

Steven M. Bellovin Universidad
de Columbia smb@cs.columbia.edu

David Benjamín Google
davidben@google.com

Benjamin Beurdouche INRIA y
Microsoft Research benjamin.beurdouche@ens.fr

Karthikeyan Bhargavan (editor de
[RFC7627])
INRIA
karthikeyan.bhargavan@inria.fr

Simon Blake-Wilson (coautor
de [RFC4492])
BCI
sblakewilson@bcisse.com

Nelson Bolyard (coautor
de [RFC4492])
Sun Microsystems, Inc.
nelson@bolyard.com

Ran Canetti
IBM
canetti@watson.ibm.com

Machine Translated by Google

Matt Caswell
OpenSSL
matt@openssl.org

Stephen Checkoway
Universidad de Illinois en Chicago sfc@uic.edu

Pete Chown
Skygate Technology Ltd
pc@skygate.co.uk

Katriel Cohn-Gordon Universidad
de Oxford me@katriel.co.uk

Cas Cremers
Universidad de Oxford
cas.cremers@cs.ox.ac.uk

Antoine Delignat-Lavaud (coautor de
[RFC7627])
INRIA
antdl@microsoft.com

Tim Dierks
(coautor de TLS 1.0, coeditor de TLS 1.1 y 1.2)
Independiente
tim@dierks.org

Roelof DuToit
Symantec Corporation
roelof_dutoit@symantec.com

Taher Elgamal
Securify
taher@securify.com

Pasi Eronen Nokia

pasi.eronen@nokia.com

Cedric FournetMicrosoft
fournet@microsoft.com

Machine Translated by Google

Anil Gangolli en
anil@busybuddha.org

David M. Garrett
dave@nulldereference.com

Illya Gerasymchuk
Independiente
illya@iluxonchik.me

Alessandro Ghedini Cloudflare
Inc. alessandro@cloudflare.com

Daniel Kahn Gillmor
ACLU
dkg@fifthhorseman.net

Matthew Green
Universidad Johns Hopkins
mgreen@cs.jhu.edu

Jens Gubala
ETAS
jens.guballa@etas.com

Felix Guenther TU
Darmstadt
mail@felixguenther.info

Vipul Gupta
(coautor de [RFC4492])
Laboratorios Sun Microsystems
vipul.gupta@sun.com

Chris Hawk
(coautor de [RFC4492])
Corriente Networks LLC
chris@corriente.net

kipp hickman

Alfredo Hoenes

David Hopwood
Consultor independiente
david.hopwood@blueyonder.co.uk

Machine Translated by Google

Marko Horvat
MPI-SWS
mhorvat@mpi-sws.org

Jonathan Hoyland Royal
Holloway, Universidad de Londres jonathan.hoyland@gmail.com

Subodh Iyengar
Facebook
subodh@fb.com

Benjamin Kaduk Akamai
Technologies kaduk@mit.edu

Hubert Kario Red
Hat Inc.
hkario@redhat.com

Phil Karlton (coautor
de SSL 3.0)

Leon Klingele
Independiente
mail@leonklingele.de

Paul Kocher
(coautor de SSL 3.0)
Investigación criptográfica
paul@cryptography.com

Hugo Krawczyk
IBM
hugokraw@us.ibm.com

Adam Langley
(coautor de [RFC7627])
Google
agl@google.com

olivier levillain
ANSSI
olivier.levillain@ssi.gouv.fr

Machine Translated by Google

Xiaoyin Liu
Universidad de Carolina del Norte en Chapel Hill xiaoyin.l@outlook.com

Ilari Liusvaara
Independiente
ilariliusvaara@welho.com

Atul Luykx KU
Lovaina
atul.luykx@kuleuven.be

Colm MacCarthaigh Amazon
Web Services colm@allcosts.net

Carlos Mehner
USAA
carl.mehner@usaa.com

Jan Mikkelsen
Transactionware
janm@transactionware.com

Bodo Moeller
(coautor de [RFC4492])
Google
bodo@acm.org

Kyle Nekritz
Facebook
knekritz@fb.com

Erik Nygren
Akamai Technologies
erik+ietf@nygren.org

Magnus Nystrom
Microsoft
mnystrom@microsoft.com

Kazuho Oku
DeNA Co., Ltd.
kazuhooku@gmail.com

Machine Translated by Google

Kenny Paterson Royal
Holloway, Universidad de Londres kenny.paterson@rhul.ac.uk

Christopher Patton Universidad de
Florida cjpatton@ufl.edu

Alfredo Pironti (coautor de
[RFC7627])
INRIA
alfredo.pironti@inria.fr

Andréi Popov Microsoft
andrei.popov@microsoft.com

Marsh Ray
(coautor de [RFC7627])
Microsoft
maray@microsoft.com

Robert Relyea
Comunicaciones de Netscape
confiarea@netscape.com

Kyle Rose
Akamai Technologies krose@krose.org

jim roskind
Amazon
jroskind@amazon.com

miguel sabin

Joe Salowey Tableau
Software joe@salowey.net

Rich Salt Akamai
rsalz@akamai.com

David Schinazi Apple Inc.
dschinazi@apple.com

Machine Translated by Google

Sam Scott
Royal Holloway, Universidad de Londres me@samjs.co.uk

Thomas Shrimpton
Universidad de Florida
teshrim@ufl.edu

Dan Simon
Microsoft, Inc.
dansimon@microsoft.com

Brian Smith
Independiente
brian@briansmith.org

Brian Sniffen Akamai
Technologies ietf@bts.evenmere.org

Nick Sullivan
Cloudflare Inc.
nick@cloudflare.com

Bjoern Tackmann
Universidad de California, San Diego btackmann@eng.ucsd.edu

Tim Taubert
Mozilla
ttaubert@mozilla.com

Martín Thomson Mozilla
mt@mozilla.com

Hannes Tschofenig Arm
Limited
Hannes.Tschofenig@arm.com

Sean Turner sn3rd
sean@sn3rd.com

Steven Valdez Google
svaldez@google.com

Machine Translated by Google

Filippo Valsorda Cloudflare
Inc. filippo@cloudflare.com

Thyla van der Merwe Royal
Holloway, Universidad de Londres tjvdmerwe@gmail.com

Víctor Vasiliev Google
vasilvv@google.com

Hoeteck Wee
Ecole Normale Supérieure, París hoeteck@alum.mit.edu

tom weinstein

David Wong
Grupo NCC
david.wong@nccgroup.trust

Christopher A. Wood Apple Inc.
cawood@apple.com

Tim Wright
Vodafone
timothy.wright@vodafone.com

Peter Wu
Independiente
peter@lekensteyn.nl

Iniciativa de Internet
de Kazu Yamamoto Japón Inc. kazu@iij.ad.jp

Dirección del autor

Eric Rescorla
Mozilla

Correo electrónico: ekr@rtfm.com