tls

package standard library

Version: go1.20.1 Latest | Published: Feb 14, 2023 | License: BSD-3-Clause | Imports: 40 | Imported by: 101,484

Details ✓ Valid go.mod file ✓ Redistributable license ✓ Tagged version ✓ Stable version

Learn more

Repository cs.opensource.google/go/go

Links Report a Vulnerability

Documentation

Documentation

Rendered for linux/amd64

Overview

Package tls partially implements TLS 1.2, as specified in RFC 5246, and TLS 1.3, as specified in RFC 8446.

Index

Examples

Constants

View Source

```
const (
	// TLS 1.0 - 1.2 cipher suites.
	TLS_RSA_WITH_RC4_128_SHA                      uint16 = 0x0005
	TLS_RSA_WITH_3DES_EDE_CBC_SHA                 uint16 = 0x000a
	TLS_RSA_WITH_AES_128_CBC_SHA                  uint16 = 0x002f
	TLS_RSA_WITH_AES_256_CBC_SHA                  uint16 = 0x0035
	TLS_RSA_WITH_AES_128_CBC_SHA256               uint16 = 0x003c
	TLS_RSA_WITH_AES_128_GCM_SHA256               uint16 = 0x009c
	TLS_RSA_WITH_AES_256_GCM_SHA384               uint16 = 0x009d
	TLS_ECDHE_ECDSA_WITH_RC4_128_SHA              uint16 = 0xc007
	TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA          uint16 = 0xc009
	TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA          uint16 = 0xc00a
	TLS_ECDHE_RSA_WITH_RC4_128_SHA                uint16 = 0xc011
	TLS_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA           uint16 = 0xc012
	TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA            uint16 = 0xc013
	TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA            uint16 = 0xc014
	TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256       uint16 = 0xc023
	TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256         uint16 = 0xc027
	TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256         uint16 = 0xc02f
	TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256       uint16 = 0xc02b
	TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384         uint16 = 0xc030
	TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384       uint16 = 0xc02c
	TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256   uint16 = 0xcca8
	TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256 uint16 = 0xcca9

	// TLS 1.3 cipher suites.
	TLS_AES_128_GCM_SHA256       uint16 = 0x1301
	TLS_AES_256_GCM_SHA384       uint16 = 0x1302
	TLS_CHACHA20_POLY1305_SHA256 uint16 = 0x1303

	// TLS_FALLBACK_SCSV isn't a standard cipher suite but an indicator
	// that the client is doing version fallback. See RFC 7507.
	TLS_FALLBACK_SCSV uint16 = 0x5600

	// Legacy names for the corresponding cipher suites with the correct _SHA256
	// suffix, retained for backward compatibility.
	TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305   = TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA25
	TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305 = TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA
)
```

A list of cipher suite IDs that are, or have been, implemented by this package.

See https://www.iana.org/assignments/tls-parameters/tls-parameters.xml

View Source

```
const (
    VersionTLS10 = 0x0301
    VersionTLS11 = 0x0302
    VersionTLS12 = 0x0303
    VersionTLS13 = 0x0304

    // Deprecated: SSLv3 is cryptographically broken, and is no longer
    // supported by this package. See golang.org/issue/32716.
    VersionSSL30 = 0x0300
)
```

Variables

This section is empty.

Functions

func CipherSuiteName

added in go1.14

```
func CipherSuiteName(id uint16) string
```

CipherSuiteName returns the standard name for the passed cipher suite ID (e.g. "TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256"), or a fallback representation of the ID value if the cipher suite is not implemented by this package.

func Listen

```
func Listen(network, laddr string, config *Config) (net.Listener, error)
```

Listen creates a TLS listener accepting connections on the given network address using net.Listen. The configuration config must be non-nil and must include at least one certificate or else set GetCertificate.

func NewListener

```
func NewListener(inner net.Listener, config *Config) net.Listener
```

NewListener creates a Listener which accepts connections from an inner Listener and wraps each connection with Server. The configuration config must be non-nil and must include at least one certificate or else set GetCertificate.

Types

type Certificate

```
type Certificate struct {
    Certificate [][]byte
    // PrivateKey contains the private key corresponding to the public key in
    // Leaf. This must implement crypto.Signer with an RSA, ECDSA or Ed25519 PublicKey.
    // For a server up to TLS 1.2, it can also implement crypto.Decrypter with
    // an RSA PublicKey.
    PrivateKey crypto.PrivateKey
    // SupportedSignatureAlgorithms is an optional list restricting what
    // signature algorithms the PrivateKey can be used for.
    SupportedSignatureAlgorithms []SignatureScheme
    // OCSPStaple contains an optional OCSP response which will be served
    // to clients that request it.
    OCSPStaple []byte
    // SignedCertificateTimestamps contains an optional list of Signed
    // Certificate Timestamps which will be served to clients that request it.
    SignedCertificateTimestamps [][]byte
    // Leaf is the parsed form of the leaf certificate, which may be initialized
    // using x509.ParseCertificate to reduce per-handshake processing. If nil,
    // the leaf certificate will be parsed as needed.
    Leaf *x509.Certificate
}
```

A Certificate is a chain of one or more certificates, leaf first.

func LoadX509KeyPair

```
func LoadX509KeyPair(certFile, keyFile string) (Certificate, error)
```

LoadX509KeyPair reads and parses a public/private key pair from a pair of files. The files must contain PEM encoded data. The certificate file may contain intermediate certificates following the leaf certificate to form a certificate chain. On successful return, Certificate.Leaf will be nil because the parsed form of the certificate is not retained.

▶ Example

func X509KeyPair

```
func X509KeyPair(certPEMBlock, keyPEMBlock []byte) (Certificate, error)
```

X509KeyPair parses a public/private key pair from a pair of PEM encoded data. On successful return, Certificate.Leaf will be nil because the parsed form of the certificate is not retained.

▶ Example

▶ Example (HttpServer)

type CertificateRequestInfo

added in go1.8

```
type CertificateRequestInfo struct {
    // AcceptableCAs contains zero or more, DER-encoded, X.501
    // Distinguished Names. These are the names of root or intermediate CAs
    // that the server wishes the returned certificate to be signed by. An
    // empty slice indicates that the server has no preference.
    AcceptableCAs [][]byte

    // SignatureSchemes lists the signature schemes that the server is
    // willing to verify.
    SignatureSchemes []SignatureScheme

    // Version is the TLS version that was negotiated for this connection.
    Version uint16
    // contains filtered or unexported fields
}
```

CertificateRequestInfo contains information from a server's CertificateRequest message, which is used to demand a certificate and proof of control from a client.

func (*CertificateRequestInfo) Context

added in go1.17

```
func (c *CertificateRequestInfo) Context() context.Context
```

Context returns the context of the handshake that is in progress. This context is a child of the context passed to HandshakeContext, if any, and is canceled when the handshake concludes.

func (*CertificateRequestInfo) SupportsCertificate

added in go1.14

```
func (cri *CertificateRequestInfo) SupportsCertificate(c *Certificate) error
```

SupportsCertificate returns nil if the provided certificate is supported by the server that sent the CertificateRequest. Otherwise, it returns an error describing the reason for the incompatibility.

type CertificateVerificationError

added in go1.20

```
type CertificateVerificationError struct {
    // UnverifiedCertificates and its contents should not be modified.
    UnverifiedCertificates []*x509.Certificate
    Err                    error
}
```

CertificateVerificationError is returned when certificate verification fails during the handshake.

func (*CertificateVerificationError) Error

added in go1.20

```
func (e *CertificateVerificationError) Error() string
```

func (*CertificateVerificationError) Unwrap

added in go1.20

```
func (e *CertificateVerificationError) Unwrap() error
```

type CipherSuite

added in go1.14

```
type CipherSuite struct {
	ID   uint16
	Name string

	// Supported versions is the list of TLS protocol versions that can
	// negotiate this cipher suite.
	SupportedVersions []uint16

	// Insecure is true if the cipher suite has known security issues
	// due to its primitives, design, or implementation.
	Insecure bool
}
```

CipherSuite is a TLS cipher suite. Note that most functions in this package accept and expose cipher suite IDs instead of this type.

func CipherSuites

added in go1.14

```
func CipherSuites() []*CipherSuite
```

CipherSuites returns a list of cipher suites currently implemented by this package, excluding those with security issues, which are returned by InsecureCipherSuites.

The list is sorted by ID. Note that the default cipher suites selected by this package might depend on logic that can't be captured by a static list, and might not match those returned by this function.

func InsecureCipherSuites

added in go1.14

```
func InsecureCipherSuites() []*CipherSuite
```

InsecureCipherSuites returns a list of cipher suites currently implemented by this package and which have security issues.

Most applications should not use the cipher suites in this list, and should only use those returned by CipherSuites.

type ClientAuthType

```
type ClientAuthType int
```

ClientAuthType declares the policy the server will follow for TLS Client Authentication.

```
const (
	// NoClientCert indicates that no client certificate should be requested
	// during the handshake, and if any certificates are sent they will not
	// be verified.
	NoClientCert ClientAuthType = iota
```

```
    // RequestClientCert indicates that a client certificate should be requested
    // during the handshake, but does not require that the client send any
    // certificates.
    RequestClientCert
    // RequireAnyClientCert indicates that a client certificate should be requested
    // during the handshake, and that at least one certificate is required to be
    // sent by the client, but that certificate is not required to be valid.
    RequireAnyClientCert
    // VerifyClientCertIfGiven indicates that a client certificate should be requested
    // during the handshake, but does not require that the client sends a
    // certificate. If the client does send a certificate it is required to be
    // valid.
    VerifyClientCertIfGiven
    // RequireAndVerifyClientCert indicates that a client certificate should be requeste
    // during the handshake, and that at least one valid certificate is required
    // to be sent by the client.
    RequireAndVerifyClientCert
)
```

func (ClientAuthType) String

added in go1.15

```
func (i ClientAuthType) String() string
```

type ClientHelloInfo

added in go1.4

```
type ClientHelloInfo struct {
    // CipherSuites lists the CipherSuites supported by the client (e.g.
    // TLS_AES_128_GCM_SHA256, TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256).
    CipherSuites []uint16

    // ServerName indicates the name of the server requested by the client
    // in order to support virtual hosting. ServerName is only set if the
    // client is using SNI (see RFC 4366, Section 3.1).
    ServerName string

    // SupportedCurves lists the elliptic curves supported by the client.
    // SupportedCurves is set only if the Supported Elliptic Curves
    // Extension is being used (see RFC 4492, Section 5.1.1).
    SupportedCurves []CurveID

    // SupportedPoints lists the point formats supported by the client.
    // SupportedPoints is set only if the Supported Point Formats Extension
    // is being used (see RFC 4492, Section 5.1.2).
    SupportedPoints []uint8

    // SignatureSchemes lists the signature and hash schemes that the client
    // is willing to verify. SignatureSchemes is set only if the Signature
    // Algorithms Extension is being used (see RFC 5246, Section 7.4.1.4.1).
    SignatureSchemes []SignatureScheme

    // SupportedProtos lists the application protocols supported by the client.
    // SupportedProtos is set only if the Application-Layer Protocol
```

```
	// Negotiation Extension is being used (see RFC 7301, Section 3.1).
	//
	// Servers can select a protocol by setting Config.NextProtos in a
	// GetConfigForClient return value.
	SupportedProtos []string

	// SupportedVersions lists the TLS versions supported by the client.
	// For TLS versions less than 1.3, this is extrapolated from the max
	// version advertised by the client, so values other than the greatest
	// might be rejected if used.
	SupportedVersions []uint16

	// Conn is the underlying net.Conn for the connection. Do not read
	// from, or write to, this connection; that will cause the TLS
	// connection to fail.
	Conn net.Conn
	// contains filtered or unexported fields
}
```

ClientHelloInfo contains information from a ClientHello message in order to guide application logic in the GetCertificate and GetConfigForClient callbacks.

func (*ClientHelloInfo) Context

added in go1.17

```
func (c *ClientHelloInfo) Context() context.Context
```

Context returns the context of the handshake that is in progress. This context is a child of the context passed to HandshakeContext, if any, and is canceled when the handshake concludes.

func (*ClientHelloInfo) SupportsCertificate

added in go1.14

```
func (chi *ClientHelloInfo) SupportsCertificate(c *Certificate) error
```

SupportsCertificate returns nil if the provided certificate is supported by the client that sent the ClientHello. Otherwise, it returns an error describing the reason for the incompatibility.

If this ClientHelloInfo was passed to a GetConfigForClient or GetCertificate callback, this method will take into account the associated Config. Note that if GetConfigForClient returns a different Config, the change can't be accounted for by this method.

This function will call x509.ParseCertificate unless c.Leaf is set, which can incur a significant performance cost.

type ClientSessionCache

added in go1.3

```
type ClientSessionCache interface {
	// Get searches for a ClientSessionState associated with the given key.
	// On return, ok is true if one was found.
	Get(sessionKey string) (session *ClientSessionState, ok bool)
	// Put adds the ClientSessionState to the cache with the given key. It might
	// get called multiple times in a connection if a TLS 1.3 server provides
	// more than one session ticket. If called with a nil *ClientSessionState,
	// it should remove the cache entry.
	Put(sessionKey string, cs *ClientSessionState)
}
```

ClientSessionCache is a cache of ClientSessionState objects that can be used by a client to resume a TLS session with a given server. ClientSessionCache implementations should expect to be called concurrently from different goroutines. Up to TLS 1.2, only ticket-based resumption is supported, not SessionID-based resumption. In TLS 1.3 they were merged into PSK modes, which are supported via this interface.

func NewLRUClientSessionCache

added in go1.3

```
func NewLRUClientSessionCache(capacity int) ClientSessionCache
```

NewLRUClientSessionCache returns a ClientSessionCache with the given capacity that uses an LRU strategy. If capacity is < 1, a default capacity is used instead.

type ClientSessionState

added in go1.3

```
type ClientSessionState struct {
	// contains filtered or unexported fields
}
```

ClientSessionState contains the state needed by clients to resume TLS sessions.

type Config

```
type Config struct {
	// Rand provides the source of entropy for nonces and RSA blinding.
	// If Rand is nil, TLS uses the cryptographic random reader in package
	// crypto/rand.
	// The Reader must be safe for use by multiple goroutines.
	Rand io.Reader

	// Time returns the current time as the number of seconds since the epoch.
	// If Time is nil, TLS uses time.Now.
	Time func() time.Time

	// Certificates contains one or more certificate chains to present to the
	// other side of the connection. The first certificate compatible with the
	// peer's requirements is selected automatically.
	//
	// Server configurations must set one of Certificates, GetCertificate or
	// GetConfigForClient. Clients doing client-authentication may set either
	// Certificates or GetClientCertificate.
	//
	// Note: if there are multiple Certificates, and they don't have the
```

```
    // optional field Leaf set, certificate selection will incur a significant
    // per-handshake performance cost.
    Certificates []Certificate

    // NameToCertificate maps from a certificate name to an element of
    // Certificates. Note that a certificate name can be of the form
    // '*.example.com' and so doesn't have to be a domain name as such.
    //
    // Deprecated: NameToCertificate only allows associating a single
    // certificate with a given name. Leave this field nil to let the library
    // select the first compatible chain from Certificates.
    NameToCertificate map[string]*Certificate

    // GetCertificate returns a Certificate based on the given
    // ClientHelloInfo. It will only be called if the client supplies SNI
    // information or if Certificates is empty.
    //
    // If GetCertificate is nil or returns nil, then the certificate is
    // retrieved from NameToCertificate. If NameToCertificate is nil, the
    // best element of Certificates will be used.
    //
    // Once a Certificate is returned it should not be modified.
    GetCertificate func(*ClientHelloInfo) (*Certificate, error)

    // GetClientCertificate, if not nil, is called when a server requests a
    // certificate from a client. If set, the contents of Certificates will
    // be ignored.
    //
    // If GetClientCertificate returns an error, the handshake will be
    // aborted and that error will be returned. Otherwise
    // GetClientCertificate must return a non-nil Certificate. If
    // Certificate.Certificate is empty then no certificate will be sent to
    // the server. If this is unacceptable to the server then it may abort
    // the handshake.
    //
    // GetClientCertificate may be called multiple times for the same
    // connection if renegotiation occurs or if TLS 1.3 is in use.
    //
    // Once a Certificate is returned it should not be modified.
    GetClientCertificate func(*CertificateRequestInfo) (*Certificate, error)

    // GetConfigForClient, if not nil, is called after a ClientHello is
    // received from a client. It may return a non-nil Config in order to
    // change the Config that will be used to handle this connection. If
    // the returned Config is nil, the original Config will be used. The
    // Config returned by this callback may not be subsequently modified.
    //
    // If GetConfigForClient is nil, the Config passed to Server() will be
    // used for all connections.
    //
    // If SessionTicketKey was explicitly set on the returned Config, or if
    // SetSessionTicketKeys was called on the returned Config, those keys will
    // be used. Otherwise, the original Config keys will be used (and possibly
```

```go
	// rotated if they are automatically managed).
	GetConfigForClient func(*ClientHelloInfo) (*Config, error)

	// VerifyPeerCertificate, if not nil, is called after normal
	// certificate verification by either a TLS client or server. It
	// receives the raw ASN.1 certificates provided by the peer and also
	// any verified chains that normal processing found. If it returns a
	// non-nil error, the handshake is aborted and that error results.
	//
	// If normal verification fails then the handshake will abort before
	// considering this callback. If normal verification is disabled by
	// setting InsecureSkipVerify, or (for a server) when ClientAuth is
	// RequestClientCert or RequireAnyClientCert, then this callback will
	// be considered but the verifiedChains argument will always be nil.
	//
	// verifiedChains and its contents should not be modified.
	VerifyPeerCertificate func(rawCerts [][]byte, verifiedChains [][]*x509.Certificate)

	// VerifyConnection, if not nil, is called after normal certificate
	// verification and after VerifyPeerCertificate by either a TLS client
	// or server. If it returns a non-nil error, the handshake is aborted
	// and that error results.
	//
	// If normal verification fails then the handshake will abort before
	// considering this callback. This callback will run for all connections
	// regardless of InsecureSkipVerify or ClientAuth settings.
	VerifyConnection func(ConnectionState) error

	// RootCAs defines the set of root certificate authorities
	// that clients use when verifying server certificates.
	// If RootCAs is nil, TLS uses the host's root CA set.
	RootCAs *x509.CertPool

	// NextProtos is a list of supported application level protocols, in
	// order of preference. If both peers support ALPN, the selected
	// protocol will be one from this list, and the connection will fail
	// if there is no mutually supported protocol. If NextProtos is empty
	// or the peer doesn't support ALPN, the connection will succeed and
	// ConnectionState.NegotiatedProtocol will be empty.
	NextProtos []string

	// ServerName is used to verify the hostname on the returned
	// certificates unless InsecureSkipVerify is given. It is also included
	// in the client's handshake to support virtual hosting unless it is
	// an IP address.
	ServerName string

	// ClientAuth determines the server's policy for
	// TLS Client Authentication. The default is NoClientCert.
	ClientAuth ClientAuthType

	// ClientCAs defines the set of root certificate authorities
	// that servers use if required to verify a client certificate
```

```
    // by the policy in ClientAuth.
    ClientCAs *x509.CertPool

    // InsecureSkipVerify controls whether a client verifies the server's
    // certificate chain and host name. If InsecureSkipVerify is true, crypto/tls
    // accepts any certificate presented by the server and any host name in that
    // certificate. In this mode, TLS is susceptible to machine-in-the-middle
    // attacks unless custom verification is used. This should be used only for
    // testing or in combination with VerifyConnection or VerifyPeerCertificate.
    InsecureSkipVerify bool

    // CipherSuites is a list of enabled TLS 1.0–1.2 cipher suites. The order of
    // the list is ignored. Note that TLS 1.3 ciphersuites are not configurable.
    //
    // If CipherSuites is nil, a safe default list is used. The default cipher
    // suites might change over time.
    CipherSuites []uint16

    // PreferServerCipherSuites is a legacy field and has no effect.
    //
    // It used to control whether the server would follow the client's or the
    // server's preference. Servers now select the best mutually supported
    // cipher suite based on logic that takes into account inferred client
    // hardware, server hardware, and security.
    //
    // Deprecated: PreferServerCipherSuites is ignored.
    PreferServerCipherSuites bool

    // SessionTicketsDisabled may be set to true to disable session ticket and
    // PSK (resumption) support. Note that on clients, session ticket support is
    // also disabled if ClientSessionCache is nil.
    SessionTicketsDisabled bool

    // SessionTicketKey is used by TLS servers to provide session resumption.
    // See RFC 5077 and the PSK mode of RFC 8446. If zero, it will be filled
    // with random data before the first server handshake.
    //
    // Deprecated: if this field is left at zero, session ticket keys will be
    // automatically rotated every day and dropped after seven days. For
    // customizing the rotation schedule or synchronizing servers that are
    // terminating connections for the same host, use SetSessionTicketKeys.
    SessionTicketKey [32]byte

    // ClientSessionCache is a cache of ClientSessionState entries for TLS
    // session resumption. It is only used by clients.
    ClientSessionCache ClientSessionCache

    // MinVersion contains the minimum TLS version that is acceptable.
    //
    // By default, TLS 1.2 is currently used as the minimum when acting as a
    // client, and TLS 1.0 when acting as a server. TLS 1.0 is the minimum
    // supported by this package, both as a client and as a server.
    //
    // The client-side default can temporarily be reverted to TLS 1.0 by
    // including the value "x509sha1=1" in the GODEBUG environment variable.
    // Note that this option will be removed in Go 1.19 (but it will still be
    // possible to set this field to VersionTLS10 explicitly).
    MinVersion uint16

    // MaxVersion contains the maximum TLS version that is acceptable.
    //
    // By default, the maximum version supported by this package is used,
    // which is currently TLS 1.3.
    MaxVersion uint16

    // CurvePreferences contains the elliptic curves that will be used in
    // an ECDHE handshake, in preference order. If empty, the default will
    // be used. The client will use the first preference as the type for
    // its key share in TLS 1.3. This may change in the future.
    CurvePreferences []CurveID

    // DynamicRecordSizingDisabled disables adaptive sizing of TLS records.
    // When true, the largest possible TLS record size is always used. When
    // false, the size of TLS records may be adjusted in an attempt to
    // improve latency.
    DynamicRecordSizingDisabled bool

    // Renegotiation controls what types of renegotiation are supported.
    // The default, none, is correct for the vast majority of applications.
    Renegotiation RenegotiationSupport

    // KeyLogWriter optionally specifies a destination for TLS master secrets
    // in NSS key log format that can be used to allow external programs
    // such as Wireshark to decrypt TLS connections.
    // See https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSS/Key_Log_Format
    // Use of KeyLogWriter compromises security and should only be
    // used for debugging.
    KeyLogWriter io.Writer
    // contains filtered or unexported fields
}
```

A Config structure is used to configure a TLS client or server. After one has been passed to a TLS function it must not be modified. A Config may be reused; the tls package will also not modify it.

▶ Example (KeyLogWriter)

▶ Example (VerifyConnection)

func (*Config) BuildNameToCertificate DEPRECATED Show

func (*Config) Clone

added in go1.8

```
func (c *Config) Clone() *Config
```

Clone returns a shallow clone of c or nil if c is nil. It is safe to clone a Config that is being used concurrently by a TLS client or server.

func (*Config) SetSessionTicketKeys

added in go1.5

```
func (c *Config) SetSessionTicketKeys(keys [][32]byte)
```

SetSessionTicketKeys updates the session ticket keys for a server.

The first key will be used when creating new tickets, while all keys can be used for decrypting tickets. It is safe to call this function while the server is running in order to rotate the session ticket keys. The function will panic if keys is empty.

Calling this function will turn off automatic session ticket key rotation.

If multiple servers are terminating connections for the same host they should all have the same session ticket keys. If the session ticket keys leaks, previously recorded and future TLS connections using those keys might be compromised.

type Conn

```
type Conn struct {
    // contains filtered or unexported fields
}
```

A Conn represents a secured connection. It implements the net.Conn interface.

func Client

```
func Client(conn net.Conn, config *Config) *Conn
```

Client returns a new TLS client side connection using conn as the underlying transport. The config cannot be nil: users must set either ServerName or InsecureSkipVerify in the config.

func Dial

```
func Dial(network, addr string, config *Config) (*Conn, error)
```

Dial connects to the given network address using net.Dial and then initiates a TLS handshake, returning the resulting TLS connection. Dial interprets a nil configuration as equivalent to the zero configuration; see the documentation of Config for the defaults.

▶ Example

func DialWithDialer

added in go1.3

```
func DialWithDialer(dialer *net.Dialer, network, addr string, config *Config) (*Conn,
error)
```

DialWithDialer connects to the given network address using dialer.Dial and then initiates a TLS handshake, returning the resulting TLS connection. Any timeout or deadline given in the dialer apply to connection and TLS handshake as a whole.

DialWithDialer interprets a nil configuration as equivalent to the zero configuration; see the documentation of Config for the defaults.

DialWithDialer uses context.Background internally; to specify the context, use Dialer.DialContext with NetDialer set to the desired dialer.

func Server

```
func Server(conn net.Conn, config *Config) *Conn
```

Server returns a new TLS server side connection using conn as the underlying transport. The configuration config must be non-nil and must include at least one certificate or else set GetCertificate.

func (*Conn) Close

```
func (c *Conn) Close() error
```

Close closes the connection.

func (*Conn) CloseWrite

added in go1.8

```
func (c *Conn) CloseWrite() error
```

CloseWrite shuts down the writing side of the connection. It should only be called once the handshake has completed and does not call CloseWrite on the underlying connection. Most callers should just use Close.

func (*Conn) ConnectionState

```
func (c *Conn) ConnectionState() ConnectionState
```

ConnectionState returns basic TLS details about the connection.

func (*Conn) Handshake

```
func (c *Conn) Handshake() error
```

Handshake runs the client or server handshake protocol if it has not yet been run.

Most uses of this package need not call Handshake explicitly: the first Read or Write will call it automatically.

For control over canceling or setting a timeout on a handshake, use HandshakeContext or the Dialer's DialContext method instead.

func (*Conn) HandshakeContext

added in go1.17

```
func (c *Conn) HandshakeContext(ctx context.Context) error
```

HandshakeContext runs the client or server handshake protocol if it has not yet been run.

The provided Context must be non-nil. If the context is canceled before the handshake is complete, the handshake is interrupted and an error is returned. Once the handshake has completed, cancellation of the context will not affect the connection.

Most uses of this package need not call HandshakeContext explicitly: the first Read or Write will call it automatically.

func (*Conn) LocalAddr

```
func (c *Conn) LocalAddr() net.Addr
```

LocalAddr returns the local network address.

func (*Conn) NetConn

added in go1.18

```
func (c *Conn) NetConn() net.Conn
```

NetConn returns the underlying connection that is wrapped by c. Note that writing to or reading from this connection directly will corrupt the TLS session.

func (*Conn) OCSPResponse

```
func (c *Conn) OCSPResponse() []byte
```

OCSPResponse returns the stapled OCSP response from the TLS server, if any. (Only valid for client connections.)

func (*Conn) Read

```
func (c *Conn) Read(b []byte) (int, error)
```

Read reads data from the connection.

As Read calls Handshake, in order to prevent indefinite blocking a deadline must be set for both Read and Write before Read is called when the handshake has not yet completed. See SetDeadline, SetReadDeadline, and SetWriteDeadline.

func (*Conn) RemoteAddr

```
func (c *Conn) RemoteAddr() net.Addr
```

RemoteAddr returns the remote network address.

func (*Conn) SetDeadline

```
func (c *Conn) SetDeadline(t time.Time) error
```

SetDeadline sets the read and write deadlines associated with the connection. A zero value for t means Read and Write will not time out. After a Write has timed out, the TLS state is corrupt and all future writes will return the same error.

func (*Conn) SetReadDeadline

```
func (c *Conn) SetReadDeadline(t time.Time) error
```

SetReadDeadline sets the read deadline on the underlying connection. A zero value for t means Read will not time out.

func (*Conn) SetWriteDeadline

```
func (c *Conn) SetWriteDeadline(t time.Time) error
```

SetWriteDeadline sets the write deadline on the underlying connection. A zero value for t means Write will not time out. After a Write has timed out, the TLS state is corrupt and all future writes will return the same error.

func (*Conn) VerifyHostname

```
func (c *Conn) VerifyHostname(host string) error
```

VerifyHostname checks that the peer certificate chain is valid for connecting to host. If so, it returns nil; if not, it returns an error describing the problem.

func (*Conn) Write

```
func (c *Conn) Write(b []byte) (int, error)
```

Write writes data to the connection.

As Write calls Handshake, in order to prevent indefinite blocking a deadline must be set for both Read and Write before Write is called when the handshake has not yet completed. See SetDeadline, SetReadDeadline, and SetWriteDeadline.

type ConnectionState

```
type ConnectionState struct {
    // Version is the TLS version used by the connection (e.g. VersionTLS12).
```

```
	Version uint16

	// HandshakeComplete is true if the handshake has concluded.
	HandshakeComplete bool

	// DidResume is true if this connection was successfully resumed from a
	// previous session with a session ticket or similar mechanism.
	DidResume bool

	// CipherSuite is the cipher suite negotiated for the connection (e.g.
	// TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256, TLS_AES_128_GCM_SHA256).
	CipherSuite uint16

	// NegotiatedProtocol is the application protocol negotiated with ALPN.
	NegotiatedProtocol string

	// NegotiatedProtocolIsMutual used to indicate a mutual NPN negotiation.
	//
	// Deprecated: this value is always true.
	NegotiatedProtocolIsMutual bool

	// ServerName is the value of the Server Name Indication extension sent by
	// the client. It's available both on the server and on the client side.
	ServerName string

	// PeerCertificates are the parsed certificates sent by the peer, in the
	// order in which they were sent. The first element is the leaf certificate
	// that the connection is verified against.
	//
	// On the client side, it can't be empty. On the server side, it can be
	// empty if Config.ClientAuth is not RequireAnyClientCert or
	// RequireAndVerifyClientCert.
	//
	// PeerCertificates and its contents should not be modified.
	PeerCertificates []*x509.Certificate

	// VerifiedChains is a list of one or more chains where the first element is
	// PeerCertificates[0] and the last element is from Config.RootCAs (on the
	// client side) or Config.ClientCAs (on the server side).
	//
	// On the client side, it's set if Config.InsecureSkipVerify is false. On
	// the server side, it's set if Config.ClientAuth is VerifyClientCertIfGiven
	// (and the peer provided a certificate) or RequireAndVerifyClientCert.
	//
	// VerifiedChains and its contents should not be modified.
	VerifiedChains [][]*x509.Certificate

	// SignedCertificateTimestamps is a list of SCTs provided by the peer
	// through the TLS handshake for the leaf certificate, if any.
	SignedCertificateTimestamps [][]byte

	// OCSPResponse is a stapled Online Certificate Status Protocol (OCSP)
	// response provided by the peer for the leaf certificate, if any.
```

```
    OCSPResponse []byte

    // TLSUnique contains the "tls-unique" channel binding value (see RFC 5929,
    // Section 3). This value will be nil for TLS 1.3 connections and for all
    // resumed connections.
    //
    // Deprecated: there are conditions in which this value might not be unique
    // to a connection. See the Security Considerations sections of RFC 5705 and
    // RFC 7627, and https://mitls.org/pages/attacks/3SHAKE#channelbindings.
    TLSUnique []byte
    // contains filtered or unexported fields
}
```

ConnectionState records basic TLS details about the connection.

func (*ConnectionState) ExportKeyingMaterial

added in go1.11

```
func (cs *ConnectionState) ExportKeyingMaterial(label string, context []byte, length i
nt) ([]byte, error)
```

ExportKeyingMaterial returns length bytes of exported key material in a new slice as defined in RFC 5705. If context is nil, it is not used as part of the seed. If the connection was set to allow renegotiation via Config.Renegotiation, this function will return an error.

type CurveID

added in go1.3

```
type CurveID uint16
```

CurveID is the type of a TLS identifier for an elliptic curve. See https://www.iana.org/assignments/tls-parameters/tls-parameters.xml#tls-parameters-8.

In TLS 1.3, this type is called NamedGroup, but at this time this library only supports Elliptic Curve based groups. See RFC 8446, Section 4.2.7.

```
const (
    CurveP256 CurveID = 23
    CurveP384 CurveID = 24
    CurveP521 CurveID = 25
    X25519    CurveID = 29
)
```

func (CurveID) String

added in go1.15

```
func (i CurveID) String() string
```

type Dialer

added in go1.15

```
type Dialer struct {
    // NetDialer is the optional dialer to use for the TLS connections'
    // underlying TCP connections.
    // A nil NetDialer is equivalent to the net.Dialer zero value.
    NetDialer *net.Dialer

    // Config is the TLS configuration to use for new connections.
    // A nil configuration is equivalent to the zero
    // configuration; see the documentation of Config for the
    // defaults.
    Config *Config
}
```

Dialer dials TLS connections given a configuration and a Dialer for the underlying connection.

func (*Dialer) Dial

added in go1.15

```
func (d *Dialer) Dial(network, addr string) (net.Conn, error)
```

Dial connects to the given network address and initiates a TLS handshake, returning the resulting TLS connection.

The returned Conn, if any, will always be of type *Conn.

Dial uses context.Background internally; to specify the context, use DialContext.

func (*Dialer) DialContext

added in go1.15

```
func (d *Dialer) DialContext(ctx context.Context, network, addr string) (net.Conn, err
or)
```

DialContext connects to the given network address and initiates a TLS handshake, returning the resulting TLS connection.

The provided Context must be non-nil. If the context expires before the connection is complete, an error is returned. Once successfully connected, any expiration of the context will not affect the connection.

The returned Conn, if any, will always be of type *Conn.

type RecordHeaderError

added in go1.6

```
type RecordHeaderError struct {
    // Msg contains a human readable string that describes the error.
    Msg string
    // RecordHeader contains the five bytes of TLS record header that
    // triggered the error.
    RecordHeader [5]byte
    // Conn provides the underlying net.Conn in the case that a client
    // sent an initial handshake that didn't look like TLS.
    // It is nil if there's already been a handshake or a TLS alert has
    // been written to the connection.
    Conn net.Conn
}
```

RecordHeaderError is returned when a TLS record header is invalid.

func (RecordHeaderError) Error

added in go1.6

```
func (e RecordHeaderError) Error() string
```

type RenegotiationSupport

added in go1.7

```
type RenegotiationSupport int
```

RenegotiationSupport enumerates the different levels of support for TLS renegotiation. TLS renegotiation is the act of performing subsequent handshakes on a connection after the first. This significantly complicates the state machine and has been the source of numerous, subtle security issues. Initiating a renegotiation is not supported, but support for accepting renegotiation requests may be enabled.

Even when enabled, the server may not change its identity between handshakes (i.e. the leaf certificate must be the same). Additionally, concurrent handshake and application data flow is not permitted so renegotiation can only be used with protocols that synchronise with the renegotiation, such as HTTPS.

Renegotiation is not defined in TLS 1.3.

```
const (
    // RenegotiateNever disables renegotiation.
    RenegotiateNever RenegotiationSupport = iota

    // RenegotiateOnceAsClient allows a remote server to request
    // renegotiation once per connection.
    RenegotiateOnceAsClient

    // RenegotiateFreelyAsClient allows a remote server to repeatedly
    // request renegotiation.
    RenegotiateFreelyAsClient
)
```

type SignatureScheme

added in go1.8

```
type SignatureScheme uint16
```

SignatureScheme identifies a signature algorithm supported by TLS. See RFC 8446, Section 4.2.3.

```
const (
    // RSASSA-PKCS1-v1_5 algorithms.
    PKCS1WithSHA256 SignatureScheme = 0x0401
    PKCS1WithSHA384 SignatureScheme = 0x0501
    PKCS1WithSHA512 SignatureScheme = 0x0601

    // RSASSA-PSS algorithms with public key OID rsaEncryption.
    PSSWithSHA256 SignatureScheme = 0x0804
    PSSWithSHA384 SignatureScheme = 0x0805
    PSSWithSHA512 SignatureScheme = 0x0806

    // ECDSA algorithms. Only constrained to a specific curve in TLS 1.3.
    ECDSAWithP256AndSHA256 SignatureScheme = 0x0403
    ECDSAWithP384AndSHA384 SignatureScheme = 0x0503
    ECDSAWithP521AndSHA512 SignatureScheme = 0x0603

    // EdDSA algorithms.
    Ed25519 SignatureScheme = 0x0807

    // Legacy signature and hash algorithms for TLS 1.2.
    PKCS1WithSHA1 SignatureScheme = 0x0201
    ECDSAWithSHA1 SignatureScheme = 0x0203
)
```

func (SignatureScheme) String

added in go1.15

```
func (i SignatureScheme) String() string
```

Notes

Bugs

- The crypto/tls package only implements some countermeasures against Lucky13 attacks on CBC-mode encryption, and only on SHA1 variants. See http://www.isg.rhul.ac.uk/tls/TLStiming.pdf and https://www.imperialviolet.org/2013/02/04/luckythirteen.html.

Source Files

View all

alert.go	conn.go	key_agreement.go
auth.go	handshake_client.go	key_schedule.go
cache.go	handshake_client_tls13.go	notboring.go
cipher_suites.go	handshake_messages.go	prf.go
common.go	handshake_server.go	ticket.go
common_string.go	handshake_server_tls13.go	tls.go

Google