TLS Working Group P.Gutmann
Internet-Draft University of Auckland
Expires: April 2004 October 2003

Use of Shared Keys in the TLS Protocol
draft-ietf-tls-sharedkeys-02

1. Abstract

The TLS handshake requires the use of CPU-intensive public-key algorithms with a considerable overhead in resource-constrained environments or ones such as mainframes where users are charged for CPU time. This document describes a means of employing TLS using symmetric keys or passwords shared in advance among communicating parties. As an additional benefit, this mechanism provides cryptographic authentication of both client and server without requiring the transmission of passwords or the use of certificates. No modifications or alterations to the TLS protocol are required for this process.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document (in uppercase, as shown) are to be interpreted as described in [RFC 2119].

2. Problem analysis

TLS is frequently used with devices with little CPU power available, for example mobile and embedded devices. In these situations the initial TLS handshake can take as long as half a minute with a 1Kbit RSA key. In many cases a fully general public-key-based handshake is unnecessary, since the device is only syncing to a host PC or contacting a fixed base station, which would allow a pre-shared symmetric key to be used instead. An example of this kind of use is using 3GPP cellular mechanisms to establish keys used to secure a TLS tunnel to a mobile device.

In a slight variation of this case, CPU power is available but is too expensive to devote to public-key operations. This situation is common in mainframe environments, where users are charged for CPU time. As with mobile devices, mainframe-to-mainframe or client-to-mainframe communications are generally fixed in advance, allowing shared symmetric keys to be employed.

In order to solve these problems, we require a means of eliminating the expensive public-key operations in the TLS handshake, while providing an equivalent level of security using shared symmetric keys. The solution is fairly straightforward. Observe that after the initial handshake phase, TLS is operating with a quantity of symmetric keying material derived from the information exchanged during the initial handshake. Using shared symmetric keys involves explicitly deriving the TLS master secret from the shared key, rather than sharing it implicitly via the public-key-based key agreement process. TLS already contains such a mechanism built into the protocol in the form of the session cacheing mechanism, which allows a TLS session to be resumed without requiring a full public-key-based re-handshake.

The solution to the problem then is obvious: We need to seed the TLS session cache with the shared symmetric key. When the client connects, the session cacheing mechanism takes over and the client and server "resume" the phantom session created by seeding the cache. This mechanism requires an absolute minimum of code changes to existing TLS implementations (it can be bolted onto any existing TLS engine without needing to change the engine itself), and no changes to the TLS protocol itself.

2.1 Design considerations

In order to work within the existing TLS protocol design, we require a means of identifying a particular session (the session ID in TLS terminology), and the keying material required to protect the session. The { ID, key } combination is analogous to the { user name, password } combination traditionally used to secure access to computer systems.

In TLS, the session ID is a variable-length value of up to 32 bytes, but is typically 32 or less frequently 16 bytes long. For our use we don't really care about its form. A (somewhat unsound) practice would be to use the user name as the session ID. A more secure alternative would be to employ a value derived from the user name in such a way that it can't be directly connected to it, for example a MAC of the user name.

Normally the exact format of the session ID is determined explicitly by the server and remembered by the client for use during session resumption.

However, when "resuming" a phantom session in the manner described here, both the client and the server must be able to implicitly generate identical session ID values in order to identify the phantom session to be resumed. To create a canonical session ID value, we pad the variable-length value out to a fixed length by appending zero bytes.

The TLS master secret is a 48-byte value, which is unlikely to correspond to the value of the shared symmetric key or password, which would typically be a 128-bit key or a text password/passphrase. In order to transform this into the type of keying material required by TLS, we need to apply the TLS pseudorandom function (PRF) to produce the master secret with which we seed the session cache. The shared secret thus takes the place of the 48-byte premaster secret usually used to derive the master secret. As with the variable-length session ID, we need to canonicalise the variable-length secret.

The obvious way to do this would be to by zero-pad it to the standard 48-byte length usually used for the premaster secret, as for the session ID. Unfortunately this straightforward approach doesn't work. Unlike the SSL PRF, which uses the full secret for both the MD5 and SHA-1 halves, the TLS PRF isn't a pure black-box design because it splits the secret into two halves before using it. This would result in the second (SHA-1) half in most cases end up with only the zero padding bytes as its "secret". The reasoning behind this splitting of the secret was that there might be some interaction between the two algorithms that could cause security problems.

As a result, it's necessary to be aware of the PRF's internal structure and pre-process the input in a way that negates what the PRF does. Some of the possible options to fix the problem are:

1. Synthesise a new PRF from HMAC to pre-PRF the input to the TLS PRF. Apart from just being an awful approach, this violates the minimal code-change requirement for TLS implementations that the shared-keys mechanism is supposed to provide. Instead of simply feeding data in via a standard mechanism, implementors would now need to extend their TLS implementation to introduce new crypto mechanisms.

2. Repeat the input (or some variant thereof) to fill the 48-byte secret value. This is problematic in that it creates key equivalence classes, for example "ABCD" == "ABCDABCD".

3. Unsplit the input, so that instead of arranaging it as 1 x 48 bytes it's done as 2 x 24 bytes. This limits the overall key size, and is specific to the PRF being used - a future PRF design may not split the input in this manner, negating the un-splitting step.

The least ugly solution is a variation of 2, prepending a single length byte to the secret, then repeating it to fill 48 bytes, to fix the problem of key equivalence classes. This is the approach used here.

Currently the shared-key mechanism always uses the TLS PRF (even if it's used with SSL, since this is purely a TLS mechanism). If in the future a new PRF is introduced, it will be necessary to provide some means of switching over to the new PRF if both it and the current one are in active use. Presumably the only reason to introduce a new, incompatible PRF would be a successful attack on the current one, in which case the point is moot. However, if for some reason it's necessary to keep both PRFs in active use at the same time, then some mechanism such as adding the session ID and shared key in the standard manner using the TLS PRF and some transformation of the session ID and the shared key using the new PRF can be adopted. Since the details of a possible PRF switch are impossible to predict (it may entail a complete protocol overhaul for example), this document does not attempt to guess at the details beyond providing this implementation hint.

Finally, we need a means of injecting the resulting session ID and master secret into the session cache. This is the only modification required to existing TLS implementations. Once the cache is seeded, all further details are handled automatically by the TLS protocol.

It should be noted that this mechanism is best suited for situations where a small number of clients/servers are communicating. While seeding a session cache with IDs and keys for 10,000 different users is certainly possible, this is rather wasteful of server resources, not to mention the accompanying key management nightmare involved in handling such a large number of shared symmetric keys.

3. TLS using shared keys

[Note: The following is phrased fairly informally, since it's really an application note rather than a standards-track RFC]

Before any exchange takes place, the client and server session caches are seeded with a session ID identifying the user/session, and a master secret derived from the shared secret key or password/passphrase. The exact form of the data used to create the session ID is application specific (but see the comment in the security considerations). The data used to create the session ID is zero-padded to 16 bytes (128 bits) if necessary to meet the requirements given in section 2.1. In C this may be expressed as:

```
  memset( session_id, 0, 16 );
  memcpy( session_id, input_data, min( input_data_length, 16 ) );
```

The master secret used to seed the cache is computed in the standard manner using the TLS PRF:

```
  master_secret = PRF(shared_secret, "shared secret", "")[0..47];
```

The shared secret or password/passphrase takes the place of the premaster secret that is normally used at this point, arranged as follows: First, the shared secret/password has a single length byte prepended to it. The length + secret value is then repeated as required to fill the standard 48 bytes. In C this may be expressed as:

```
  for( premaster_index = 0; premaster_index < 48; )
    {
    int i;

    premaster_secret[ premaster_index++ ] = shared_secret_length;
    for( i = 0; i < shared_secret_length && premaster_index < 48; i++ )
      premaster_secret[ premaster_index++ ] = shared_secret[ i ];
    }
```

This formats the shared secret in a manner that allows it to be used directly in place of the standard premaster secret derived from the public-key-based key agreement process.

The 'seed' component of the calculation (normally occupied by the client and server nonces) is empty in this case, however applications may choose to use an application or system-specific value to ensure that the same shared secret used with another application or system yields a different master secret. When the 'seed' component is non-empty, it should not contain information computed from the shared_secret value [SIGMA]. Note that the use of the client and server nonces will always produce different keys for each session, even if the same master secret is employed.

The final step involves injecting the session ID and master secret into the session cache. This is an implementation-specific issue beyond the scope of this document. All further steps are handled automatically by the TLS protocol, which will "resume" the phantom session created by the above steps without going through the full public-key-based handshake.

Session cache entries are normally expired after a given amount of time, or overwritten on an LRU basis. In order to prevent shared secret-based entries from vanishing after a certain amount of time, these cache entries will need to be distinguished from standard cache entries and made more persistent then the latter, for example by giving them a longer expiry time when they are added or periodically touching them to update their last-access time. Again, this is an implementation issue beyond the scope of this document.

3.1 Use of shared keys with SSLv3

If this key management mechanism is used with an implementation that supports SSLv3 alongside TLS (as most do), the TLS PRF must be used for both SSLv3 and TLS. This is required in order to allow the mechanism to function for both SSLv3 and TLS, since using different PRFs would require a different session ID for each PRF used.

3.2 Test vectors

The following test vectors are derived from the transformation of the password "test" into a master_secret value to be added to the session cache:

```
  Shared secret:
     74 65 73 74   ("test")
```

Shared secret expanded to 48-byte premaster secret size:

```
     04 74 65 73 74 04 74 65
     73 74 04 74 65 73 74 04
     ...
```

Master secret added to session cache:

```
     F5 CE 30 92 B8 09 70 D9
     22 D5 A1 2C EB 7C 43 FA
     9C 46 A8 83 EA 6E EF 98
     EB A5 15 12 FD B1 B6 5A
     5A 47 B8 C4 C5 63 5B 30
     86 96 F4 FC FB D5 45 78
```

4. Security considerations

The session ID used to identify sessions is visible to observers. While using a user name as the session ID is the most straightforward option, it may lead to problems with traffic analysis, with an attacker being able to track the identities of communicating parties. In addition since the session ID is reused over time, traffic analysis may eventually allow an attacker to identify parties even if an opaque session ID is used. [RFC 2246] contains a similar warning about the contents of session IDs with TLS in general. It should be noted though that even a worse-case non-opaque session ID results in no more exposure than the use of client certificates during a handshake.

As with all schemes involving shared keys, special care should be taken to protect the shared values and to limit their exposure over time. Documents covering other shared-key protocols such as Kerberos [RFC 1510] contain various security suggestions in this regard.

Use of a fixed shared secret of limited entropy (for example a password) allows an attacker to perform an online password-guessing attack by trying to resume a session with a master secret derived from each possible password. This results in a fatal decrypt_error alert (or some equivalent such as handshake_failure or bad_record_mac) which makes the session non-resumable (that is, it clears the phantom session from the session cache). Implementations should limit the enthusiasm with which they re-seed the session cache after such an event; standard precautions against online password-guessing attacks apply.

This mechanism is purely a shared-key session establishment mechanism and does not provide perfect forward secrecy (PFS) by negotiating additional new keying material for each session. Users requiring PFS can either use a shared-key mechanism that also provides PFS such as SRP [SRP], or perform a rehandshake using a standard PFS-providing mechanism over the shared key-protected channel. Note though that both of these mechanisms negate the two main advantages of the shared-key mechanism, requiring both considerable re-engineering of an existing TLS implementation and considerable CPU time to perform the PFS cryptographic operations.

Since it does not contain an innate cryptographic mechanism to provide PFS, the shared-key mechanism is vulnerable to an offline password-guessing attack as follows: An attacker who records all of the handshake messages and knows the plaintext for at least one encrypted message can perform the TLS key-derivation using a selection of guessed passwords, perform the cryptographic operations required to process the TLS handshake exchange, and then apply the resulting cryptographic keys to the known-plaintext message. Such an attack consumes considerable time and CPU resources, but is nevertheless possible.

There are three possible defences against this type of attack, the first two of which are standard defences against password-guessing attacks:

1. Don't use weak, easily-guess passwords or keys.

2. Perform iterated pre-processing of the password/key before adding it to the session cache. This has the disadvantage that it negates the shared-key advantage of low CPU consumption during the handshake phase, however the preprocessing can be performed offline on a one-off basis and only the preprocessed key stored by the two communicating parties. An attacker can, however, also generate a dictionary of pre-processed keys offline, given sufficient CPU and storage space. The use of a per-server diversifier ('seed' in the PRF process) makes use of a precomputed dictionary impractical, and a secret diversifier makes a general offline attack considerably more difficult through to impossible depending on the circumstances.

3. Use a mechanism that allows for the use of shared keys but also provides PFS, with the advantages and disadvantages described earlier.

Note that the two password-guessing attacks possible against the shared-key mechanism, while superficially similar, have quite different requirements on the attacker's side. An online attack merely requires that the attacker know the URL of the server that they wish to attack. An offline attack requires that an attacker both know the URL of the server that they wish to attack and be able to record complete sessions between the client and the server in order to provide the material required for the offline attack.

The TLS specification requires that when a session is resumed, the resumed session use the same cipher suite as the original one. Since with a shared-secret session there is no actual session being resumed, it's not possible to meet this requirement. Two approaches are possible to resolve this:

1. When the session cache is seeded on the server, a cipher suite acceptable to the server is specified for the resumed session. This complies with the requirements, but requires that the server know that the client is capable of supporting this particular suite. In closed environments (for example syncing to a host PC or a fixed base station, or in a mainframe

environment) this is likely to be the case.

2. The requirements are relaxed to allow the client and server to negotiate a cipher suite in the usual manner. In order to subvert this, an attacker would have to be able to perform a real-time simultaneous break of both HMAC-MD5 and HMAC-SHA1. In particular the attacker would need to be able to subvert:

```
HMAC( secret, PRF( secret, MD5+SHA1 hash ) )
```

in the Finished message, which expands to:

```
HMAC( secret, HMAC-MD5^HMAC-SHA1( secret, MD5+SHA1 hash ) )
```

Because of the unlikeliness of this occurring (an attacker capable of doing this can subvert any TLS session, with or without shared secrets), it appears safe to relax the requirement for resuming with the same cipher suite.

References (Normative)

[RFC 2119] "Key words for use in RFCs to Indicate Requirement Levels", Scott Bradner, RFC 2119, March 1997.

[RFC 2246] "The TLS Protocol", RFC 2246, Tim Dierks and Christopher Allen, January 1999.

References (Informative)

[RFC 1510] "The Kerberos Network Authentication Service (V5)", RFC 1510, John Kohl and B. Clifford Neuman, September 1993.

[SIGMA] "SIGMA: the `SIGn-and-MAc' Approach to Authenticated Diffie-Hellman and its Use in the IKE Protocols", Hugo Krawczyk, Proceedings of of Crypto'03, Springer-Verlag Lecture Notes in Computer Science No.2729, p.399.

[SRP] "Using SRP for TLS Authentication", David Taylor, IETF draft, November 2002.

Author's Address

Peter Gutmann
University of Auckland
Private Bag 92019
Auckland, New Zealand

Email: pgut001@cs.auckland.ac.nz

Full Copyright Statement

Acknowledgement

Funding for the RFC Editor function is currently provided by the Internet Society.