Fast-Track Session Establishment for TLS

Hovav Shacham
hovav@cs.stanford.edu

Dan Boneh
dabo@cs.stanford.edu

Abstract

We propose a new, "fast-track" handshake mechanism for TLS. A fast-track client caches a server's public parameters and negotiated parameters in the course of an initial, enabling handshake. These parameters need not be resent on subsequent handshakes. The new mechanism reduces both network traffic and the number of round trips, and requires no additional server state. These savings are most useful in high latency environments such as wireless networks. We include a rollback mechanism to allow a server to gracefully revert to an ordinary TLS handshake when needed. Our design is fully backwards compatible: fast-track clients can interoperate with servers unaware of fast-track and vise versa. We have implemented our proposal to demonstrate the savings in network traffic and round trips.

1. Introduction

TLS is a widely deployed protocol for securing network traffic. It is commonly used for protecting web traffic and some e-mail protocols such as IMAP and POP. Variants of TLS, such as WTLS [6], are used for securing wireless communication. In this paper we consider a modification to the TLS (and WTLS) handshake protocol that makes the protocol more efficient in terms of bandwidth and number of round trips. Improving the handshake protocol is especially relevant in bandwidth-constrained environments, such as wireless communications, where latency is high and small payload transfers are common. We hope that these extensions will promote the use of TLS in high latency networks and discourage the development of ad-hoc security protocols for such networks.

Recall that the TLS protocol [3] incorporates two types of handshake mechanisms: a full handshake, and a resume handshake protocol. The resume handshake protocol is used to reinstate a previously negotiated TLS session between a client and a server. Compared to a full handshake, the resume mechanism significantly reduces handshake network traffic and computation on both ends. However, a session can only be resumed if the old session is present in the server's session cache. Unfortunately, heavily loaded servers can only store a session for a relatively short time before the session is evicted from the cache. As a result, a full handshake is often needed even though the client may be willing to resume an old session.

In contrast, clients rarely connect to numerous TLS servers, and could cache information about servers for a longer time. We propose a new, "fast-track" handshake mechanism for TLS that exploits this situation. The fast-track mechanism improves the full TLS handshake protocol. Fast-track clients maintain a cache of long-lived server information, such as the server certificate, and long-lived negotiated information, such as the preferred cipher suite. The long-lived cached information allows a reduction in handshake bandwidth: the handshake messages by which a server communicates this information to the client are obviated by the cache, and omitted from the fast-track handshake. Moreover, the remaining messages are reordered, so a fast-track handshake has three flows rather than four. Hence, our fast-track mechanism reduces both network traffic and round trips in the TLS handshake protocol.

By a flow we mean an uninterrupted sequence of messages from one participant in a connection to the other. An ordinary TLS handshake has four flows; our fast-track handshake has three. Because of the design of the TLS protocol, multiple consecutive handshake messages can be coalesced into a single TLS transport-layer message. Thus, when network latency is high, a savings in flows can translate into a savings in time.

The use of fast-track, along with the particular fast-track parameters, is negotiated between clients and servers by means of TLS extensions [1]. Care is taken to ensure interoperability with non-fast-track clients and servers, and to allow graceful fallback to ordinary TLS handshakes when required.

The use of fast-track session establishment gives savings in handshake bandwidth and flows, but does not provide a significant computational speedup relative to ordinary TLS handshakes. It is most useful for bandwidth-constrained, high-latency situations, and those in which application message payloads are small. Thus fast-track,

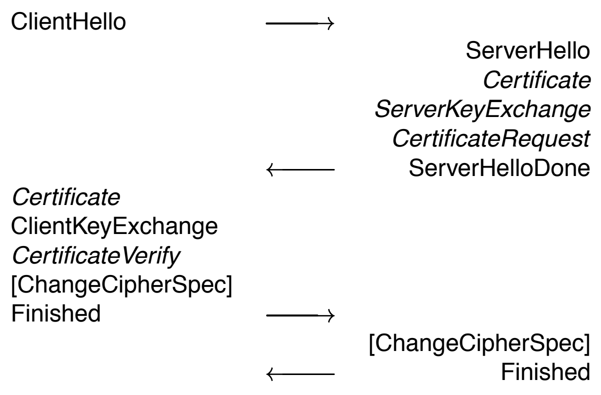

Figure 1. TLS Handshake Message Diagram

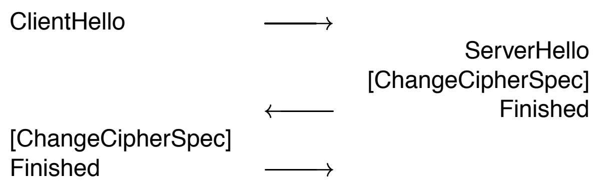

Figure 2. Message Diagram for a TLS Session Resume

via a relatively simple, and fully backwards-compatible change to the TLS protocol, improves performance and makes TLS more usable in wireless environments. Client-side caching could be applied to other mainstream security protocols to yield similar results.

We enumerate the long-lived, cacheable items and describe the manner in which they are used in Section 3. We discuss some design criteria in Section 4. We describe the fast-track handshake protocol in Section 5. We then discuss performance, implementation, and security consideration in Sections 6, 7, and 8. Finally, we briefly consider other client-side caching strategies for TLS in Section 9.

2. TLS handshake overview

A TLS handshake has three objectives: (1) to negotiate certain session parameters; (2) to authenticate the server to the client, and optionally the client to the server; and (3) to establish a shared cryptographic secret. The session parameters include the protocol version, the cipher suite, and the compression method. Authentication makes use of a certificate-based public key infrastructure (PKI): servers and clients identify themselves through certificate chains terminating in well-known Certification Authority (CA) certificates.

The standard TLS handshake is summarized in Figure 1. Messages sent by the client are on the left; by the server, the right. Messages appearing in slanted type are only sent in certain configurations; messages in brackets are sent out-of-band. The handshake proceeds, in four flows, as follows.

A client initiates a handshake by sending a ClientHello message. This message includes a suggested protocol version, a list of acceptable cipher suites and compression methods, a client random value used in establishing the shared secret, and (when TLS extensions [1] are used) other extension-specific parameters.

The server replies with a ServerHello message, which selects a protocol version, cipher suite, and compression method, and includes the server random, and extension-specific parameters. The server then sends its certificate chain in the Certificate message. In certain cases, it sends a ServerKeyExchange message with additional information required for establishing the shared secret. (For example, the 512-bit export-grade RSA key for RSA export key-exchange.) If the server wishes that the client authenticate itself, it sends a CertificateRequest message listing acceptable certificate types and CA names for the client's certificate chain. Finally, it sends a ServerHelloDone message to signal the end of the flow.

If the server requests client authentication, the client begins its response with a Certificate message that includes its certificate chain, and after the ClientKeyExchange message, a CertificateVerify message that includes its signature on a digest of the handshake messages to that point. The ClientKeyExchange message includes the information necessary to determine the shared secret. (For example, in RSA key exchange, it includes the encryption of a "premaster secret" that is used to calculate the secret.)

Finally, the client sends a ChangeCipherSpec message (which is not a handshake message), signaling its switch to the newly-negotiated parameters and secret key, and sends an encrypted and compressed Finished message that includes a digest of the handshake messages.

The server, in its turn, also sends a ChangeCipherSpec message and a Finished message that includes a digest of the handshake messages (up to the client's Finished message). After this, the client and server can exchange application data over the encrypted, authenticated, and possibly compressed link that has been established.

A server can identify a particular connection by a "session ID," a field in the ServerHello message. By mutual consent, a client and server can resume a connection. The client includes the ID of the session it wishes to resume in its hello message, and the server accepts by including the same ID in its hello. The client and server proceed directly to the ChangeCipherSpec and Finished messages (with the previously-agreed-upon parameters and secrets). This exchange is summarized in Figure 2.

Relative to establishing a new session, resuming a previously-negotiated session saves bandwidth, flows, and computation (since the handshake's expensive cryptographic operations are avoided). However, heavily loaded servers typically keep session IDs in their session cache for only a relatively short while.

We note that our fast-track optimization only applies to the full handshake protocol (not the session resume handshake). Hence, fast-track is most effective in environments where short lived TLS sessions are common so that frequent handshakes are not resumes.

3. Cacheable handshake parameters

The savings we achieve through fast-track depend on a client's caching certain long-lived handshake parameters. "Long-lived," in this context, means, first, that they do not change between handshakes (as does, e.g., the server-random), and, second, that they are expected not to change except when either the server or client is reconfigured. A client collects these parameters in the course of an ordinary TLS handshake. In the course of a fast-track handshake, it uses these parameters to craft its messages.

The particular values which a client uses in a fast-track handshake are called the determining parameters for that connection. A server uses information in the client hello message and its own configuration to come up with its own version of the determining parameters for the connection. The two versions must match for the handshake to be successful. Therefore, a fast-track-initiating hello message includes a hash of the determining parameters to allow the server to check this, as described in Section 5.2.

The long-lived parameters fall into two general categories: (1) those that are features of the server's configuration alone, and (2) those that properly depend on the interaction of the server's configuration with the client's. In the first category, we include:

- The server's certificate chain;
- The server's Diffie-Hellman group, if any; and
- Whether client authentication is required; if so,
  - Acceptable client certificate types; and
  - Acceptable certificate authorities.

These features of a TLS server's configuration are assumed to change infrequently and thus to be capable of being cached on the client.
In the second category, we include parameters such as:

- The preferred client-server cipher suite; and
- The preferred client-server compression method.

(The cipher suite comprises a key-exchange algorithm, a bulk encryption algorithm, and a MAC algorithm.) These are a function of both the server and client configurations, and are negotiated in a TLS handshake: the client proposes a list for each, and the server chooses.

A client in possession of the above information knows enough to be able to compute a key-exchange message, without any additional input from the server (with one exception discussed below). It is this fact that allows the reordering of the handshake messages.

To participate in ephemeral Diffie-Hellman (EDH) key exchange, a client needs to know the group modulus and generator relative to which the Diffie-Hellman exchange will operate. The description of this group is part of the ServerKeyExchange message when EDH is used. It is assumed that the server will not often change its EDH group, so a fast-track client can cache the group parameters and use them to send a ClientKeyExchange message during a fast-track handshake. By contrast, a server employing temporary RSA keys for key exchange, in the RSA "export" cipher suites, will typically change its export RSA key quite often. The temporary RSA key, which a client would need for its fast-track key exchange, can be cached only briefly. Accordingly, fast-track explicitly does not support RSA export authentication. Since the RSA export mechanism is being phased out, this does not seem like a serious constraint.

4. Design considerations

With significant deployment of legacy TLS clients, incompatible changes to the protocol are unlikely to be accepted. Accordingly, fast-track's design emphasizes interoperability and backwards-compatibility. Fast-track clients and servers must be able to interoperate with TLS servers and clients not capable of using fast-track; they must be able to discover which peers are capable of fast-track; and they must recover gracefully when configurations have changed, falling back on the ordinary TLS handshake protocol.

Through the use of TLS extensions [1], a client and server can, in an ordinary TLS handshake, negotiate the future use of fast-track. A subsequent fast-track connection uses another extension to allow the client and server to ascertain their both using the same unsent, client-cached parameters. Since a client must suggest, and a server must assent to the use of fast-track, the likelihood of a client's attempting to initiate a fast-track connection with a non-fast-track server is minimal.

If a client does attempt to initiate a fast-track connection with a non-fast-track server, it is important that it be alerted of its mistake quickly. A fast-track handshake is initiated through a message that TLS servers not implementing fast-track would reject as invalid. This minimizes confusion resulting from such a mismatch. For servers aware of fast-track, but not wishing to use it, we include a rollback mechanism to allow a server to gracefully revert to an ordinary TLS handshake if its configuration has changed.

5. The fast-track handshake

In this section, we describe the actual fast-track handshake protocol. There are two distinct phases. First, in the course of an ordinary TLS handshake, a client and server negotiate and agree on the future use of fast-track, and the client collects the parameters that will allow it to make that future handshake. Next, the client initiates a fast-track handshake with the server, using the determining parameters from earlier.

Fast-track also defines a mechanism whereby the server can deny the fast-track; it would do so, for example, when its configuration has changed, rendering the client's determining parameters obsolete. This mechanism is also used for session resumes.

5.1. Negotiation of fast-track

A client wishing to engage in a fast-track handshake with a server must first determine whether that server is capable of (and willing to use) fast-track. This is not a problem, since the client must also have completed an ordinary handshake with the server to have obtained the information it needs for the new, fast-track handshake.

The TLS Extensions mechanism [1] provides the machinery for the negotiation. A client proposing the prospective use of fast-track includes the fasttrack-capable extension in its hello; a server assenting to the prospective use includes the same extension in its hello. Such a handshake is referred to as "enabling."

Servers might be reconfigured to disable fast-track, and clients should be alerted of the configuration change as soon as possible; preferably, before they undertake the computationally-heavy early steps of the fast-track handshake.

Accordingly, a client is expected to include in each of its handshakes the fasttrack-capable extension, and attempt a fast-track handshake with a server only if their most recent successful handshake was an enabling one. (Per the specification, the extensions governing a resumed session are those negotiated in the original handshake for that session; a successful resume is therefore not considered a handshake for this purpose.)

5.2. Fast-track

To engage in a fast-track handshake, the client and server must agree on certain determining parameters (see Section 3). The client obtains these from a previous, enabling handshake. But it and the server must make sure that they expect to use the same parameters. Fast-track ensures this as follows. As part of its fast-track hello message, a client must include, in the fasttrack-hash extension, the SHA-1 hash of the determining parameters. The server builds its own version of the parameters, and ensures that the hashes match.

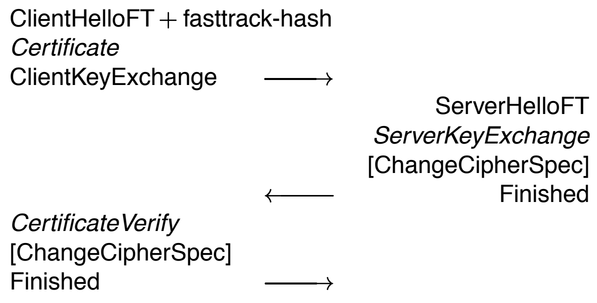

Figure 3. Message Diagram for an Accepted Fast-Track Handshake

Suppose a client initiates a fast-track handshake, and includes in its hello message both the fasttrack-capable extension and the fasttrack-hash extension, accompanying the latter with a hash of what it thinks are the determining parameters for the handshake. If the server's configuration has changed, but it still wishes to engage in fast-track in the future (with the new, correct parameters), it ought to deny the fast-track, but include the fasttrack-capable extension in its (ordinary) hello message. If, instead, the server's configuration has changed, and it no longer wishes to engage in fast-track in the future, it ought to deny the fast-track, and ought not to include the fasttrack-capable extension in its hello.

The fast-track handshake is summarized in Figure 3. The notation is that employed in Figures 1 and 2, above. Note that the ClientHelloFT message must include the fasttrack-hash extension with a hash of the determining parameters; this requirement is indicated in the first line of the figure.

The exchange omits the server Certificate, CertificateRequest, and ServerHelloDone messages, and requires three flows rather than four. In an ordinary TLS handshake, the server has the last handshake flow; here, the client does. If the client sends the first application data — the typical situation — the savings in flows is magnified, since the client's first application-data flow can be coalesced with its last handshake flow.

The fast-track handshake calls for a nontrivial reordering of the TLS handshake messages. If a client were accidentally to attempt it with a server entirely unaware of fast-track, the client and server might entirely befuddle one another. In keeping with the design goal that the client and server should discover as expeditiously as possible whether fast-track is appropriate, the fast-track client

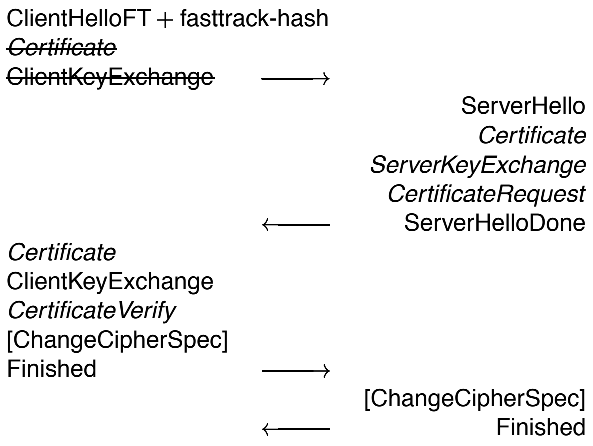

Figure 4. Message Diagram for a Denied Fast-Track Handshake

hello is made a different message type — ClientHelloFT rather than ClientHello — although the two message types have an identical format. A TLS server that is not aware of fast-track will alert the client immediately to the unexpected message type.

The client has enough information to create its key-exchange message without any additional server input, so this message can be sent in the first flow. Once the server has sent its server-random (in its hello) and potentially its key-exchange message, both sides have enough information to calculate the master secret and change cipher suites. The client must wait until it has seen a message from the server before sending its CertificateVerify message, to avoid replay attacks.

5.3. Denying fast-track

A server need not agree to engage in a fast-track handshake, even if it had previously assented to one through the fasttrack-capable extension. Fast-track includes a mechanism whereby the server denies an in-progress fast-track handshake, and the client and server revert to an ordinary handshake negotiation.

A server denies fast-track by responding to the client's first flow with a ServerHello message rather than a ServerHelloFT. Its response should be as though the client had initiated the connection through a ClientHello message with the same body as that of the ClientHelloFT message it actually had sent (except without the fasttrack-hash extension). From that point on, the parties carry on an ordinary TLS handshake, conforming to the rules given in the TLS specification. The other messages sent by the client as part of its first flow are ignored by both parties, and are not included in any handshake message digests.

Figure 4 presents the messages exchanged when fast-

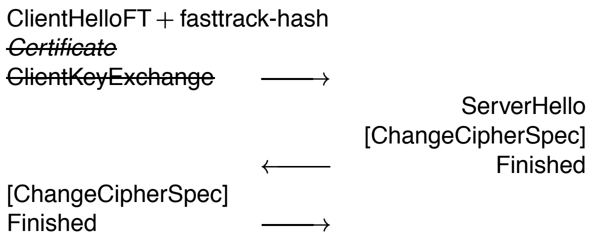

Figure 5. Message Diagram for a Session Resume, with Fast-Track Denied

track is denied. The notation is the same as employed in Figure 3, with the additional convention that messages printed with strike-through are not included in any handshake digests.

Finally, a server can deny fast-track but proceed with a session-resume if it wishes, and if the client sent a nonempty session-id in its fast-track hello message. Figure 5 gives the message flow in this case, using the same notational conventions as the previous figures. Session resumption provides less of a performance gain to fast-track clients, since they will have already engaged in the time-consuming ClientKeyExchange calculations when the server accepts the resume.

6. Performance considerations

The fast-track handshake mechanism reduces the protocol's communication requirements and round trips but has little effect on CPU load. We briefly discuss fast-track's effect on CPU load for both servers and clients A more extensive analysis of the performance of standard TLS in the Web environment is available [2].

The performance of servers employing fast-track is comparable to that of ordinary servers. Servers avoid sending as many as three messages (Certificate, CertificateRequest, and ServerHelloDone), but none of these involves any computationally-intensive operation; contrariwise, fast-track servers must verify the SHA-1 hash of the determining parameters.

Performance of fast-track clients is slightly improved, with a proper implementation. For example, once a client has validated a server's certificate chain, it need not revalidate it in the course of a fast-track handshake. Indeed, once it has computed the determining parameters hash which will later be sent to the server, the client may choose to discard the chain, maintaining only the server's public key. Thus, in a fast-track handshake, a client avoids the signature verifications of an ordinary handshake, with a long-term space overhead of only a few hundred bytes for the server key.

	RFC 2246	Fast-Track	Savings
TLS_RSA_WITH_3DES_EDE_CBC_SHA			
Client	322	291	10%
Server	1187	130	89%
Total	1509	421	72%
TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA			
Client	285	245	14%
Server	1461	404	72%
Total	1746	649	63%

Table 1. Handshake Bytes Sent for TLS Key Exchange Methods; No Client Authentication

	RFC 2246	Fast-Track	Savings
TLS_RSA_WITH_3DES_EDE_CBC_SHA, client auth			
Client	2519	2488	1%
Server	1196	130	89%
Total	3715	2618	30%
TLS_DHE_RSA_WITH_3DES_SHA, client auth			
Client	2482	2442	2%
Server	1472	404	73%
Total	3954	2846	28%

Table 2. Handshake Bytes Sent for TLS Key Exchange Methods; Client Authentication Required

7. Implementation

We have modified OpenSSL 0.9.6a to negotiate and perform fast-track handshakes. Since OpenSSL does not currently support TLS extensions, our implementation instead used TLS' version negotiation scheme: fast-track-capable clients and servers speak the fictitious TLS "Version 1.1."

We summarize our observed savings in bandwidth below. Aside from the bytes-sent measurements, our implementation also maintains the savings in flows that fast-track provides over ordinary TLS handshakes: three flows, rather than four.

Table 1 presents the number of bytes written across the wire by the client and by the server in both a standard (RFC 2246) TLS handshake [3], and a fast-track handshake. The first cipher suite, called "`TLS_RSA_WITH_3DES_EDE_CBC_SHA`" in RFC 2246 (and called "`DES-CBC3-SHA`" in OpenSSL), uses RSA for key exchange. It does not require a ServerKeyExchange message to be sent. The second cipher suite, "`TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA`" (called "`EDH-RSA-DES-CBC3-SHA`" in OpenSSL), employs Ephemeral Diffie-Hellman (EDH) for key exchange, with RSA authentication. A handshake using this cipher suite requires the server to send a ServerKeyExchange message. At present, EDH-based key exchange is not widely deployed in TLS environments, though support for it has been added in some recent browsers; accordingly, the first of the two settings in Table 1 is by far the more common.

The data in Table 1 show quite clearly that, in typical situations, the bandwidth cost of a TLS handshake is dominated by the server certificate chain. The server's key exchange message, when sent, is also a significant component. Note that the server here sends only its own certificate. Since the client must already have a copy of the self-signed CA certificate to assess the server's credentials, the CA certificate need not be transmitted. (This is permitted by the TLS specification [3, 7.4.2].)

Although the savings in bandwidth generated by the server is substantial, the savings in client bandwidth is quite modest. In fact, our implemented client does not (yet) send the determining-parameters hash to the server. These additional 22 bytes of extension to the client hello (required in a fully-conforming fast-track implementation) would largely negate the savings in client bytes-sent evident in Tables 1 and 2. The savings in server bytes-sent is unaffected. This underscores that, since fast-track does not assume a server-side cache, it can do little to reduce the information that a client must supply during a handshake. (The client bytes-sent savings are largely at the TLS transport layer, where the reduced number of flows allows greater consolidation of messages.)

Table 2 presents data in the same format as in Table 1, but in which the server requires that the client authenticate itself. Here, the dominant component is the client's certificate chain. Unlike the server, the client does send the CA certificate along with its own.

The limited gains in bytes-sent seen in Table 2 again reflect fast-track's inability to do away with the sending of client information to the server. The specific problem of client certificates can be alleviated via a different mechanism, complementary to fast-track: the TLS Extensions document defines a client-certificate-url extension [1, 3.3]. With this extension, a client sends the URL where its certificate may be found, along with a hash of the certificate, rather than the certificate itself.

The number of bytes which a server writes depends on its certificate chain; similarly for a client when client authentication is required. Since certificates vary in length, a limit is placed on the accuracy of bytes-sent measurements. This limit is made more severe by the presence at several points in the TLS handshake of arbitrary-length lists: the client's supported cipher suites; the client's supported compression methods; and the server's acceptable certificate types and acceptable CA names (for client authentication).

8. Security analysis

In this section we argue that fast-track is no less secure than the ordinary TLS handshake protocol. Unfortunately, a formal argument about the security of fast-track as a handshake protocol is extremely difficult, especially in the absence of a comprehensive formal analysis of TLS [4]. Nor is a rigorous reduction of fast-track security to TLS security feasible – the message order is changed between the two protocols, so an attacker on one would not necessarily be able to create messages for the other without breaking the hash functions used in the finished-message digests. In light of these limitations, we present common arguments about the security of fast-track.

Fast-track is negotiated in the course of an ordinary TLS handshake, using the fasttrack-capable extension (Section 5.1). The extension itself contains no sensitive data, and the negotiation is protected by the same mechanisms that protect other negotiated extensions.

A client should store determining parameters for use in a future fast-track handshake only after verifying that the server has a valid certificate, and the parameters come from an ordinary handshake, so these parameters should not be open to tampering. Furthermore, if the client and server determining parameters differ, the mismatch will be detected in the course of the handshake, since some messages will be incomprehensible. Thus, determining parameter mismatch is not a security problem, and the SHA-1 hash should be sufficient to provide collision-resistance for robustness. (The exception is if the client has obtained an adversary's certificate for the server's distinguished name, a situation that could allow for a man-in-the-middle attack. But this would require a compromise of the public key infrastructure.)

All the same information exchanged in a standard handshake is exchanged in a fast-track handshake, except for the determining parameters, for which a cryptographic hash is exchanged. The handshake digest hashes in the Finished messages should thus provide the same security as in ordinary TLS.

The ordering of the server and client Finished messages is opposite of that in ordinary TLS handshakes, but TLS session resumes also use this reversed ordering.

The server response message (ServerHello or ServerHelloFT) is included in the final hashes regardless of whether fast-track is denied, so rollback attacks should be impossible.

The only message not verified by both the client and server finished-message hashes is the client CertificateVerify message. It is included in the client finished-message hash, so the server should be able to detect its having been modified and abort the connection.

In any case, the client certificate itself is included in both finished-message hashes, and is presumably no more open to tampering than in an ordinary TLS handshake. The client CertificateVerify message contains only a signature with the certificate's key, so opportunities for mischief through its modification are limited.

9. Other client cache strategies

It was noted earlier that fast-track is a strategy for exploiting client-side caching to reduce handshake overhead. There are other possible strategies.

One can keep actual session information on the client side, encrypted — perhaps with a secret server symmetric key — to defeat tampering. A client could later send the encrypted information to the server, initiating what might be termed a client-side session resume.

There are limitations to this approach. TLS servers usually agree to resume only sessions that satisfy certain properties. For example, OpenSSL refuses to resume sessions that were not properly shut down. A client-side resume scheme would have to be designed so the client does not obtain a session ticket until the server is convinced that it is willing for that session to be resumed. (The TLS specification also recommends that sessions persist for no longer than twenty-four hours, to limit the damage from a compromise of the session's master secret.)

It should be possible, however, for a TLS server to maintain a cache of resumable session IDs, while obligating the client to hold on to an opaque ticket that contains the remainder of the information the server maintains about sessions.

If the initial handshake were unsatisfactory, the server could drop the session ID from its cache, and refuse a subsequent client-side resume even though the client had a supposedly valid ticket.

This approach essentially uses the client as secondary storage for the server's session cache, which is made to take less storage.

10. Conclusions

We proposed an improvement to the TLS handshake protocol that makes TLS easier to use in bandwidth-constrained environments. Our improvement is based on a fast-track handshake mechanism that exploits a small session-cache on the client. In a fast-track handshake there is no need to transmit server parameters such as the server certificate and preferred cipher suite. This enables us to reorder the messages in a fast-track handshake and effectively save two round trips. Hence, the fast-track handshake saves both network traffic and round trips over the standard full TLS handshake protocol. The reduction in number of round trips speeds up TLS session establishment in high latency networks. Overall, fast-track is a simple addition to TLS that makes it more friendly to use in constrained environments.

We implemented our proposal in a backwards compatible design. If either the client or the server does not support fast-track or do not wish to use fast-track they can revert to the standard full TLS handshake. Our experiments show that fast-track can save up to 72% of the handshake traffic. Our prototype implementation of fast-track and Internet draft are available for download [5].

Acknowledgments

We thanks Dan Simon and Dan Wallach for helpful conversations about this paper. This work was partially supported by an NSF CAREER grant.

References

[1] S. Blake-Wilson, M. Nystrom, D. H. J. Mikkelsen, and T. Wright. TLS Extensions. Internet-Draft: `draft-ietf-tls-extensions-01.txt`, Sept. 2001. Work in progress.

[2] C. Coarfa, P. Druschel, and D. Wallach. Performance Analysis of TLS Web Servers. In Proceedings of NDSS '02, 2002.

[3] T. Dierks and C. Allen. RFC 2246: The TLS Protocol, Version 1, Jan. 1999.

[4] J. Mitchell, V. Shmatikov, and U. Stern. Finite-State Analysis of SSL 3.0. In Seventh USENIX Security Symposium, pages 201–216, 1998.

[5] H. Shacham and D. Boneh. TLS Fast-Track Session Establishment. Internet Draft: `draft-shacham-tls-fasttrack-00.txt`, Aug. 2001. Work in progress.

[6] Wireless Application Forum. Wireless Transport Layer Security Specification. `http://www.wapforum.org`, 2000.