

Network Working Group
Internet-Draft
Expires: December 31, 2013

D. Balfanz
R. Hamilton
Google Inc
June 29, 2013

Transport Layer Security (TLS) Channel IDs
draft-balfanz-tls-channelid-01

Abstract

This document describes a Transport Layer Security (TLS) extension for identifying client machines at the TLS layer without using bearer tokens.

Status of this Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <http://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on December 31, 2013.

Copyright Notice

Copyright (c) 2013 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	3
2. Why not client certificates	4
3. Requirements Notation	6
4. Channel ID Client Keys	7
5. Channel ID Extension	8
6. Security Considerations	11
7. Use Cases	12
7.1. Channel-Bound Cookies	12
7.2. Channel-Bound OAuth Tokens	12
8. Privacy Considerations	13
9. IANA Considerations	14
10. References	15
10.1. Normative References	15
10.2. Informative References	15
Appendix A. Acknowledgements	16
Appendix B. History of Changes	17
B.1. Version 01	17
Authors' Addresses	18

1. Introduction

Many applications on the Internet use bearer tokens to authenticate clients to servers. The most prominent example is the HTTP-based World Wide Web, which overwhelmingly uses HTTP cookies to authenticate client requests. Other examples include OpenID or SAML assertions, and OAuth tokens. All these have in common that the bearer of the HTTP cookie or authentication token is granted access to a protected resource, regardless of the channel over which the token is presented, or who presented it.

As a result, an adversary that manages to steal a bearer token from a client can impersonate that client to services that require the token.

This document describes a light-weight mechanism for establishing a cryptographic channel between client and server. A server can choose to bind authentication tokens to this channel, thus rendering the theft of authentication tokens fruitless - tokens must be sent over the channel to which they are bound (i.e., by the client to which they were issued) or else they will be ignored.

This document does not prescribe how authentication tokens are bound to the underlying channel. Rather, it prescribes how a client can establish a long-lived channel with a server. Such a channel persists across HTTP requests, TLS connections, and even multiple TLS sessions, as long as the same client communicates with the same server.

The basic idea is that the client proves, during the TLS handshake, possession of a private key. The corresponding public key becomes the "Channel ID" that identifies this TLS connection. Clients should re-use the same private/public key pair across subsequent TLS connections to the same server, thus creating TLS connections that share the same Channel ID.

Using private/public key pairs to define a channel (as opposed to, say, an HTTP session cookie) has several advantages: One, the credential establishing the channel (the private key) is never sent from client to server, thus removing it from the reach of eavesdroppers in the network. Two, clients can choose to implement cryptographic operations in a secure hardware module, which further removes the private key from the reach of eavesdroppers residing on the client itself.

2. Why not client certificates

TLS already supports a means of identifying clients without using bearer tokens: client certificates. However, a number of problems with using client certificates motivated the development of an alternative.

Most importantly, it's not acceptable for a client identifier to be transmitted in the clear, because eavesdroppers in the network could use these identifiers to deanonymize TLS connections. Client certificates in TLS, however, are sent unencrypted. Although we could also define a change to the TLS state machine to move the client certificates under encryption, such changes eliminate most of the benefits of reusing something that's already defined.

TLS client certificates are also defined to be part of the session state. Even though the key material used for TLS client authentication might be protected from theft from compromised clients (for example, by employing hardware secure elements on the client), TLS session resumption information rarely is. Because client certificates are part of the session state, stolen session resumption information gives the attacker something equivalent to a stolen client private key. Our objective, however, is that attackers should not be able to give the impression that they can wield a private key unless they are actually in control of that private key.

Client-certificates typically identify a user, while we seek to identify machines. Since they are not, conceptually, mutually exclusive and as only a single client certificate can be provided in TLS, we don't want to consume that single slot and eliminate the possibility of also using existing client certificates.

Client certificates are implemented in TLS as X.509 certificates and we don't wish to require servers to parse arbitrary ASN.1. ASN.1 is a complex encoding that has been the source of several security vulnerabilities in the past and typical TLS servers can currently avoid doing ASN.1 parsing.

X.509 certificates always include a signature, which would be a self-signature in this case. Calculating and transmitting the self-signature is a waste of computation and network traffic in our use. Although we could define a null signature algorithm with an empty signature, such deviations from X.509 eliminate many of the benefits of reusing something that is already implemented.

Finally, client certificates trigger significant server-side processing by default and often need to be stored in their entirety for the duration of the connection. Since this design is intended to

be widely used, it allows servers to retain only a cryptographic hash of the client's public key after the handshake completes.

3. Requirements Notation

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

4. Channel ID Client Keys

For the purpose of this specification, a public key is a point " $Q = dG$ " on the P-256 curve [DSS] (where " d " is the ECC private key, and " G " is the curve base point). Clients SHOULD use a separate key pair " (d, Q) " for each server they connect to, and generate a new key pair if necessary according to appendix B.4 in FIPS-186-3 [DSS].

A public key " Q " has two affine coordinates " x, y ": " $Q = (x,y)$ ". The public key " Q " - or, in other words, the pair " x, y " - that a client uses for a specific server is that client's Channel ID for that server.

5. Channel ID Extension

A new extension type ("channel_id(TBD)") is defined and MAY be included by the client in its "ClientHello" message. If, and only if, the server sees this extension in the "ClientHello", it MAY choose to echo the extension in its "ServerHello". In both cases, the "extension_data" field MUST be empty.

```
enum {
    channel_id(TBD), (65535)
} ExtensionType;
```

A new handshake message type ("encrypted_extensions(TBD)") is defined. If the server included a "channel_id" extension in its "ServerHello" message, the client MUST verify that the selected cipher suite is sufficiently strong. If the cipher suite provides < 80-bits of security, the client MUST abort the handshake with a fatal "illegal_parameter" alert. Otherwise, the client MUST send an "EncryptedExtensions" message after its "ChangeCipherSpec" and before its "Finished" message.

```
enum {
    encrypted_extensions(TBD), (65535)
} HandshakeType;
```

Therefore a full handshake with "EncryptedExtensions" has the following flow (contrast with section 7.3 of RFC 5246 [RFC5246]):

Client	Server
ClientHello (ChannelID extension)	----->
	ServerHello
	(ChannelID extension)
	Certificate*
	ServerKeyExchange*
	CertificateRequest*
	<----- ServerHelloDone
Certificate*	
ClientKeyExchange	
CertificateVerify*	
[ChangeCipherSpec]	
EncryptedExtensions	
Finished	----->
	[ChangeCipherSpec]
	<----- Finished
Application Data	<-----> Application Data

An abbreviated handshake with "EncryptedExtensions" has the following

flow:

Client

Server

```

ClientHello (ChannelID extension)  ----->
                                     ServerHello
                                     (ChannelID extension)
                                     [ChangeCipherSpec]
                                     <----- Finished
[ChangeCipherSpec]
EncryptedExtensions
Finished                          ----->
Application Data                  <-----> Application Data

```

The "EncryptedExtensions" message contains a series of "Extension" structures (see section 7.4.1.4 of RFC 5246 [RFC5246])

If the server included a "channel_id" extension in its "ServerHello" message, the client MUST include, within an EncryptedExtensions message, an "Extension" with "extension_type" equal to "channel_id(TBD)". The "extension_data" of which has the following format:

```

struct {
    opaque x[32];
    opaque y[32];
    opaque r[32];
    opaque s[32];
} ChannelIDExtension;

```

The contents of each of "x", "y", "r" and "s" is a 32-byte, big-endian number. The "x" and "y" fields contain the affine coordinates of the client's Channel ID Q (i.e., a P-256 [DSS] curve point). The "r" and "s" fields contain an ECDSA [DSS] signature by the corresponding private key over this US-ASCII string (not including quotes, and where "\x00" represents an octet containing all zero bits):

"TLS Channel ID signature\x00"

followed by hashes of both the client-sent and server-sent handshake messages, as seen by the client, prior to the "EncryptedExtensions" message.

Unlike many other TLS extensions, this extension does not establish properties of the session, only of the connection. When session resumption or session tickets [RFC5077] are used, the previous contents of this extension are irrelevant and only the values in the

new handshake messages are considered.

6. Security Considerations

There are four classes of attackers against which we consider our security guarantees: passive network attackers, active network attackers, active network attackers with misissued certificates and attackers in possession of the legitimate server's private key.

First, we wish to guarantee that we don't disclose the Channel ID to passive or active network attackers. We do this by sending a constant-length Channel ID under encryption. However, since the Channel ID may be transmitted before the server's Finished message is received, it's possible that the server isn't in possession of the corresponding private key to the certificate that it presented. In this situation, an active attacker could cause a Channel ID to be transmitted under a random key in a cipher suite of their choosing. Therefore we limit the permissible cipher suites to those where decrypting the message is infeasible.

Even with this limit, an active attacker can cause the Channel ID to be transmitted in a non-forward-secure manner. Subsequent disclosure of the server's private key would allow previously recorded Channel IDs to be decrypted.

Second, we wish to guarantee that none of the first three attackers can terminate/hijack a TLS connection and impersonate a Channel ID from that connection when connecting to the legitimate server. We assume that TLS provides sufficient security to prevent these attackers from being able to hijack the TLS connection. An active attacker illegitimately in possession of a certificate for a server can successfully terminate a TLS connection destined for that server and decrypt the Channel ID. However, as the signature covers the handshake hashes, and therefore the server's certificate, it wouldn't be accepted by the true server.

Against an attacker with the legitimate server's private key we can provide the second guarantee only if the legitimate server uses a forward-secret cipher suite, otherwise the attacker can hijack the connection.

7. Use Cases

7.1. Channel-Bound Cookies

An HTTP application on the server can `_channel-bind_` its cookies by associating them with the Channel ID of the user-agent that the cookies are being set on. The server MAY then choose to consider cookies sent from the user-agent invalid if the Channel ID associated with the cookie does not match the Channel ID used by the user-agent when it sends the cookie back to the server.

Such a mismatch could occur when the cookie has been obtained from the legitimate user-agent and is now being sent by a client not in possession of the legitimate user-agent's Channel ID private key. The mismatch can also occur if the legitimate user-agent has changed the Channel ID it is using for the server, presumably due to the user requesting a Channel ID reset through the user-agent's user interface (see Section 8). Such a user intervention is analogous to the user's removal of cookies from the user-agent, but instead of removing cookies, the cookies are being rendered invalid (in the eyes of the server).

7.2. Channel-Bound OAuth Tokens

Similarly to cookies, a server may choose to channel-bind OAuth tokens (or any other kind of authorization tokens) to the clients to which they are issued. The mechanism on the server remains the same (it associates the OAuth token with the client's Channel ID either by storing this information in a database, or by suitably encoding the information in the OAuth token itself), but the application-level protocol may be different: In addition to HTTP, OAuth tokens are used in protocols such as IMAP and XMPP.

8. Privacy Considerations

The TLS layer does its part in protecting user privacy by transmitting the Channel ID public key under encryption. Higher levels of the stack must ensure that the same Channel ID is not used with different servers in such a way as to provide a linkable identifier. For example, a user-agent must use different Channel IDs for communicating with different servers. Because channel-bound cookies are an important use case for TLS Channel ID, and cookies can be set on top-level domains, it is RECOMMENDED that user-agents use the same Channel ID for servers within the same top-level domain, and different Channel IDs for different top-level domains. User-agents must also ensure that Channel ID state can be reset by the user in the same way as other identifiers, i.e. cookies.

However, there are some security concerns that could result in the disclosure of a client's Channel ID to a network attacker. This is covered in the Security Considerations section.

Clients that share an IP address can be disambiguated through their Channel IDs. This is analogous to protocols that use cookies (e.g., HTTP), which also allow disambiguation of user-agents behind proxies.

Channel ID has been designed to provide privacy equivalent to that of cookies. User-agents SHOULD continue to meet this design goal at higher layers of the protocol stack. For example, if a user indicates that they would like to block third-party cookies (or if the user-agent has some sort of policy around when it blocks third-party cookies by default), then the user agent SHOULD NOT use Channel ID on third-party connections (or other connections through which the user-agent would refuse to send or accept cookies).

9. IANA Considerations

This document requires IANA to update its registry of TLS extensions to assign an entry referred to here as "channel_id".

This document also requires IANA to update its registry of TLS handshake types to assign an entry referred to here as "encrypted_extensions".

10. References

10.1. Normative References

- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.
- [RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008.
- [DSS] National Institute of Standards and Technology, "FIPS 186-3: Digital Signature Standard".

10.2. Informative References

- [RFC5077] Salowey, J., Zhou, H., Eronen, P., and H. Tschofenig, "Transport Layer Security (TLS) Session Resumption without Server-Side State", RFC 5077, January 2008.

Appendix A. Acknowledgements

The following individuals contributed to this specification:

Dirk Balfanz, Wan-Teh Chang, Ryan Hamilton, Adam Langley, and Mayank Upadhyay.

Appendix B. History of Changes

B.1. Version 01

- o Some clarifications, mostly around the Channel ID and session state.
- o Added a section on Use Cases.
- o Expanded the Privacy Considerations sections to include discussion of third-party connections in HTTP user-agents.
- o Fixed some typos.

Authors' Addresses

Dirk Balfanz
Google Inc

Email: balfanz@google.com

Ryan Hamilton
Google Inc

Email: rch@google.com

Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: October 27, 2013

S.F. Friedl
Cisco Systems, Inc.
A. Popov
Microsoft Corp.
A. Langley
Google Inc.
E. Stephan
France Telecom - Orange
April 25, 2013

Transport Layer Security (TLS) Application Layer Protocol Negotiation
Extension
draft-ietf-tls-applayerprotoneg-01

Abstract

This document describes a Transport Layer Security (TLS) extension for application layer protocol negotiation within the TLS handshake. For instances in which the TLS connection is established over a well known TCP/IP port not associated with the desired application layer protocol, this extension allows the application layer to negotiate which protocol will be used within the TLS session.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <http://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on October 27, 2013.

Copyright Notice

Copyright (c) 2013 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents

(<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	2
2. Requirements Language	3
3. Application Layer Protocol Negotiation	3
3.1. The Application Layer Protocol Negotiation Extension . .	3
3.2. Protocol Selection	5
4. Design Considerations	5
5. Security Considerations	6
6. IANA Considerations	6
7. Acknowledgements	7
8. References	7
8.1. Normative References	7
8.2. Informative References	7
Authors' Addresses	7

1. Introduction

Increasingly, application layer protocols are encapsulated in the TLS security protocol [RFC5246]. This encapsulation enables applications to use the existing, secure communications links already present on port 443 across virtually the entire global IP infrastructure.

When multiple application protocols are supported on a single server-side port number, such as port 443, the client and the server need to negotiate an application protocol for use with each connection. It is desirable to accomplish this negotiation without adding network round-trips between the client and the server, as each round-trip will degrade an end-user's experience. Further, it would be advantageous to allow certificate selection based on the negotiated application protocol.

This document specifies a TLS extension which permits the application layer to negotiate protocol selection within the TLS handshake. This work was requested by the HTTPbis WG to address the negotiation of HTTP version ([RFC2616], [I-D.ietf-httpbis-http2]) over TLS, however ALPN facilitates negotiation of arbitrary application layer protocols.

With ALPN, the client sends the list of supported application protocols as part of the TLS ClientHello message. The server chooses a protocol and sends the selected protocol as part of the TLS ServerHello message. The application protocol negotiation can thus be accomplished within the TLS handshake, without adding network round-trips, and allows the server to associate a different certificate with each application protocol, if desired.

2. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

3. Application Layer Protocol Negotiation

3.1. The Application Layer Protocol Negotiation Extension

A new extension type ("application_layer_protocol_negotiation(16)") is defined and MAY be included by the client in its "ClientHello" message.

```
enum {  
    application_layer_protocol_negotiation(16), (65535)  
} ExtensionType;
```

The "extension_data" field of the ("application_layer_protocol_negotiation(16)") extension SHALL contain a "ProtocolNameList" value.

```
opaque ProtocolName<1..2^8-1>;
```

```
struct {  
    ProtocolName protocol_name_list<2..2^16-1>  
} ProtocolNameList;
```

"ProtocolNameList" contains the list of protocols advertised by the client, in descending order of preference. Protocols are named by IANA registered, opaque, non-empty byte strings, as described further in Section 6 "IANA Considerations" of this document. Implementations MUST ensure that an empty string is not included and that no byte strings are truncated.

Experimental protocol names, which are not registered by IANA, will start with the following sequence of bytes: 0x65, 0x78, 0x70 ("exp").

Servers that receive a client hello containing the "application_layer_protocol_negotiation" extension, MAY return a suitable protocol selection response to the client. The server will ignore any protocol name that it does not recognize. A new ServerHello extension type ("application_layer_protocol_negotiation(16)") MAY be returned to the client within the extended ServerHello message. The "extension_data" field of the ("application_layer_protocol_negotiation(16)") extension SHALL be structured the same as described above for the client "extension_data", except that the "ProtocolNameList" MUST contain exactly one "ProtocolName".

Therefore, a full handshake with the "application_layer_protocol_negotiation" extension in the ClientHello and ServerHello messages has the following flow (contrast with section 7.3 of [RFC5246]):

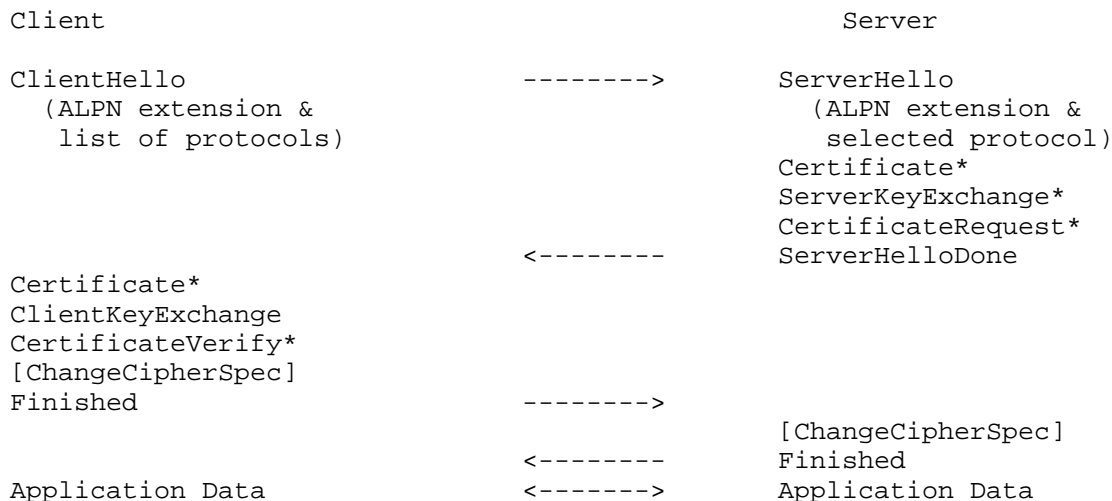
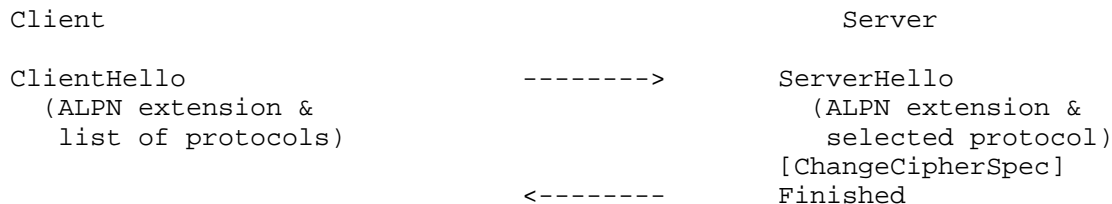


Figure 1

An abbreviated handshake with the "application_layer_protocol_negotiation" extension has the following flow:



```
[ChangeCipherSpec]
Finished           ----->
Application Data   <----->      Application Data
```

Figure 2

Unlike many other TLS extensions, this extension does not establish properties of the session, only of the connection. When session resumption or session tickets [RFC5077] are used, the previous contents of this extension are irrelevant and only the values in the new handshake messages are considered.

3.2. Protocol Selection

It is expected that a server will have a list of protocols that it supports, in preference order, and will only select a protocol if the client supports it. In that case, the server SHOULD select the most highly preferred protocol it supports which is also advertised by the client. In the event that the server supports no protocols that the client advertises, then the server SHALL respond with a fatal "no_application_protocol" alert.

```
enum {
    no_application_protocol(120),
    (255)
} AlertDescription;
```

The "no_application_protocol" fatal alert is only defined for the "application_layer_protocol_negotiation" extension and MUST NOT be sent unless the server has received a ClientHello message containing this extension.

The protocol identified in the "application_layer_protocol_negotiation" extension type in the ServerHello SHALL be definitive for the connection. The server SHALL NOT respond with a selected protocol and subsequently use a different protocol for application data exchange.

4. Design Considerations

The ALPN extension is intended to follow the typical design of TLS protocol extensions. Specifically, the negotiation is performed entirely within the client/server hello exchange in accordance with established TLS architecture. The "application_layer_protocol_negotiation" ServerHello extension is intended to be definitive for the connection and is sent in plaintext to permit network elements to provide differentiated service for the

connection when the TCP/IP port number is not definitive for the application layer protocol to be used in the connection. By placing ownership of protocol selection on the server, ALPN facilitates scenarios in which certificate selection or connection rerouting may be based on the negotiated protocol.

Finally, by managing protocol selection in the clear as part of the handshake, ALPN avoids introducing false confidence with respect to the the ability to hide the negotiated protocol in advance of establishing the connection. If hiding the protocol is required, then renegotiation after connection establishment, which would provide true TLS security guarantees, would be a preferred methodology.

A namespace will be assigned for experimental protocols, comprising byte strings which start with the following sequence of bytes: 0x65, 0x78, 0x70 ("exp"). Assignments in this namespace do not need IANA registration.

5. Security Considerations

The ALPN extension does not impact the security of TLS session establishment or application data exchange. ALPN serves to provide an externally visible marker for the application layer protocol associated with the TLS connection. Historically, the application layer protocol associated with a connection could be ascertained from the TCP/IP port number in use.

6. IANA Considerations

The IANA has updated its Registry of TLS ExtensionType Values to include the following entry:

- 16 application_layer_protocol_negotiation

This document also requires the IANA to create a registry of Application Layer Protocol Negotiation protocol byte strings, initially containing the following entries:

- "http/1.1": HTTP/1.1 [RFC2616];
- "http/2.0": HTTP/2.0 [I-D.ietf-httpbis-http2];
- "spdy/1": (obsolete) SPDY version 1;
- "spdy/2": SPDY version 2;
- "spdy/3": SPDY version 3.

We propose that this new registry be created in a new page entitled: "Application Layer Protocol Negotiation (ALPN) Protocol IDs" beneath the existing heading of "Transport Layer Security (TLS)".

7. Acknowledgements

This document benefitted specifically from the NPN extension draft authored by Adam Langley and from discussions with Tom Wesselman and Cullen Jennings both of Cisco.

8. References

8.1. Normative References

- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.
- [RFC2616] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2616, June 1999.
- [RFC3264] Rosenberg, J. and H. Schulzrinne, "An Offer/Answer Model with Session Description Protocol (SDP)", RFC 3264, June 2002.
- [RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008.
- [RFC6066] Eastlake, D., "Transport Layer Security (TLS) Extensions: Extension Definitions", RFC 6066, January 2011.

8.2. Informative References

- [I-D.ietf-httpbis-http2] Belshe, M., Peon, R., Thomson, M., and A. Melnikov, "Hypertext Transfer Protocol version 2.0", draft-ietf-httpbis-http2-02 (work in progress), April 2013.
- [I-D.mbelshe-httpbis-spdy] Belshe, M. and R. Peon, "SPDY Protocol", draft-mbelshe-httpbis-spdy-00 (work in progress), February 2012.
- [RFC5077] Salowey, J., Zhou, H., Eronen, P., and H. Tschofenig, "Transport Layer Security (TLS) Session Resumption without Server-Side State", RFC 5077, January 2008.

Authors' Addresses

Stephan Friedl
Cisco Systems, Inc.
170 West Tasman Drive
San Jose, CA 95134
USA

Phone: (720)562-6785
Email: sfriedl@cisco.com

Andrei Popov
Microsoft Corp.
One Microsoft Way
Redmond, WA 98052
USA

Email: andreipo@microsoft.com

Adam Langley
Google Inc.
USA

Email: agl@google.com

Emile Stephan
France Telecom - Orange
2 avenue Pierre Marzin
Lannion F-22307
France

Email: emile.stephan@orange.com

TLS
Internet-Draft
Intended status: Standards Track
Expires: September 29, 2013

S. Santesson
3xA Security AB
H. Tschofenig
Nokia Siemens Networks
March 28, 2013

Transport Layer Security (TLS) Cached Information Extension
draft-ietf-tls-cached-info-14.txt

Abstract

Transport Layer Security (TLS) handshakes often include fairly static information, such as the server certificate and a list of trusted Certification Authorities (CAs). This information can be of considerable size, particularly if the server certificate is bundled with a complete certificate path (including all intermediary certificates up to the trust anchor public key).

This document defines an extension that omits the exchange of already available information. The TLS client informs a server of cached information, for example from a previous TLS handshake, allowing the server to omit the already available information.

Status of this Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <http://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on September 29, 2013.

Copyright Notice

Copyright (c) 2013 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents

(<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	3
2. Terminology	4
3. Cached Information Extension	5
4. Exchange Specification	7
4.1. Omitting the Certificate Chain	7
4.2. Omitting the Trusted CAs	8
5. Example	9
6. Security Considerations	11
7. IANA Considerations	12
7.1. New Entry to the TLS ExtensionType Registry	12
7.2. New Registry for CachedInformationType	12
8. Acknowledgments	13
9. References	14
9.1. Normative References	14
9.2. Informative References	14
Authors' Addresses	15

1. Introduction

Transport Layer Security (TLS) handshakes often include fairly static information, such as the server certificate and a list of trusted Certification Authorities (CAs). This information can be of considerable size, particularly if the server certificate is bundled with a complete certificate path (including all intermediary certificates up to the trust anchor public key).

Optimizing the exchange of information to a minimum helps to improve performance in environments where devices are connected to a network with characteristics like low bandwidth, high latency and high loss rate. These types of networks exist, for example, when smart objects are connected using a low power IEEE 802.15.4 radio. For more information about the challenges with smart object deployments please see [RFC6574].

This specification defines a TLS extension that allows a client and a server to exclude transmission of cached information from the TLS handshake.

A typical example exchange may therefore look as follows. First, the client and the server executes the usual TLS handshake. The client may, for example, decide to cache the certificate provided by the server. When the TLS client connects to the TLS server some time in the future, without using session resumption, it then attaches the `cached_information` extension defined in this document to the client hello message to indicate that it had cached the certificate, and it provides the fingerprint of it. If the server's certificate had not changed then the TLS server does not need to send the full certificate to the client again. In case the information had changed, the certificate payload is transmitted to the client to allow the client to update it's state information.

2. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "MUST", "MUST NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

3. Cached Information Extension

This document defines a new extension type (`cached_information(TBD)`), which is used in client hello and server hello messages. The extension type is specified as follows.

```
enum {  
    cached_information(TBD), (65535)  
} ExtensionType;
```

The `extension_data` field of this extension, when included in the client hello, MUST contain the `CachedInformation` structure.

```
enum {  
    certificate_chain(1), trusted_cas(2) (255)  
} CachedInformationType;  
  
struct {  
    CachedInformationType type;  
    HashAlgorithm hash;  
    opaque hash_value<1..255>;  
} CachedObject;  
  
struct {  
    CachedObject cached_info<1..2^16-1>;  
} CachedInformation;
```

When the `CachedInformationType` identifies a `certificate_chain`, then the `hash_value` field MUST include the hash calculated over the `certificate_list` element of the `Certificate` payload provided by the TLS server in an earlier exchange, excluding the three length bytes of the `certificate_list` vector.

When the `CachedInformationType` identifies a `trusted_cas`, then the `hash_value` MUST include a hash calculated over the `certificate_authorities` element of the `CertificateRequest` payload provided by the TLS server in an earlier exchange, excluding the two length bytes of the `certificate_authorities` vector.

The hash algorithm used to calculate hash values is conveyed in the 'hash' field of the `CachedObject` element. The list of registered hash algorithms can be found in the TLS HashAlgorithm Registry, which was created by RFC 5246 [RFC5246]. The value zero (0) for 'none' is not an allowed choice for a hash algorithm and MUST NOT be used.

This document establishes a registry for `CachedInformationType` types and additional values can be added following the policy described in Section 7.

4. Exchange Specification

Clients supporting this extension MAY include the "cached_information" extension in the (extended) client hello, which MAY contain zero or more CachedObject attributes.

A server supporting this extension MAY include the "cached_information" extension in the (extended) server hello, which MAY contain one or more CachedObject attributes it supports. By returning the "cached_information" extension the server indicates that it supports caching of each present CachedObject that matches the specified hash value. The server MAY support other cached objects that are not present in the extension.

Note: If clients make use of the Server Name Indication [RFC6066] then clients may need to cache multiple data items for a single server since servers may host multiple 'virtual' servers at a single underlying network address.

Following a successful exchange of the "cached_information" extensions in the client and server hello, the server omits sending the corresponding handshake message. How information is omitted from the handshake message is defined per cached info type. Section 4.1 and Section 4.2 defines the syntax of the fingerprinted information.

The handshake protocol MUST proceed using the information as if it was provided in the handshake protocol. The Finished message MUST be calculated over the actual data exchanged in the handshake protocol. That is, the Finished message will be calculated over the information that was omitted from transmission by means of its present hash in the client hello and not through its presence in the handshake exchange.

The server MUST NOT include more than one fingerprint for a single information element, i.e., at maximum only one CachedObject structure per replaced information is provided.

4.1. Omitting the Certificate Chain

When an object of type 'certificate_chain' is provided in the client hello, the server MAY replace the sequence of certificates with an empty sequence with an actual length field of zero (=empty vector).

The original handshake message syntax is defined in RFC 5246 [RFC5246] and has the following structure:

```
opaque ASN.1Cert<1..2^24-1>;

struct {
    ASN.1Cert certificate_list<0..2^24-1>;
} Certificate;
```

Note that [I-D.ietf-tls-oob-pubkey] allows the certificate payload to contain only the SubjectPublicKeyInfo instead of the full information typically found in a certificate. Hence, when this specification is used in combination with [I-D.ietf-tls-oob-pubkey] and the negotiated certificate type is a raw public key then the TLS server omits sending a Certificate payload that contains an ASN.1Cert structure of the SubjectPublicKeyInfo.

4.2. Omitting the Trusted CAs

When a fingerprint for an object of type 'trusted_cas' is provided in the client hello, the server MAY send a DistinguishedName in the Certificate Request message with an actual length field of zero (=empty vector).

The original handshake message syntax is defined in RFC 5246 [RFC5246] and has the following structure:

```
opaque DistinguishedName<1..2^16-1>;

struct {
    ClientCertificateType certificate_types<1..2^8-1>;
    SignatureAndHashAlgorithm
        supported_signature_algorithms<2^16-1>;
    DistinguishedName certificate_authorities<0..2^16-1>;
} CertificateRequest;
```

5. Example

Figure 1 illustrates an example exchange using the TLS cached info extension. In the normal TLS handshake exchange shown in flow (A) the TLS server provides its certificate in the Certificate payload to the client, see step [1]. This allows the client to store the certificate for future use. After some time the TLS client again interacts with the same TLS server and makes use of the TLS cached info extension, as shown in flow (B). The TLS client indicates support for this specification via the `cached_information` extension, see [2], and indicates that it has stored the `certificate_chain` from the earlier exchange. With [3] the TLS server indicates that it also supports this specification and informs the client that it also supports caching of other objects beyond the `'certificate_chain'`, namely `'trusted_cas'` (also defined in this document), and the `'foo-bar'` extension (i.e., an imaginary extension that yet needs to be defined). With [4] the TLS server omits sending the certificate chain, as described in Section 4.1.

(A) Initial (full) Exchange

```
client_hello ->
               <-  server_hello,
                   certificate, // [1]
                   server_key_exchange,
                   server_hello_done

client_key_exchange,
change_cipher_spec,
finished
               ->
               <-  change_cipher_spec,
                   finished

Application Data    <----->    Application Data
```

(B) TLS Cached Extension Usage

```
client_hello,
cached_information=(certificate_chain) -> // [2]
               <-  server_hello,
                   cached_information= // [3]
                   (certificate_chain, trusted_cas, foo-bar)
                   certificate, // [4]
                   server_key_exchange,
                   server_hello_done

client_key_exchange,
change_cipher_spec,
finished
               ->
               <-  change_cipher_spec,
                   finished

Application Data    <----->    Application Data
```

Figure 1: Example Message Exchange

6. Security Considerations

This specification defines a mechanism to reference stored state using a fingerprint. Sending a fingerprint of cached information in an unencrypted handshake, as the client and server hello is, may allow an attacker or observer to correlate independent TLS exchanges. While some information elements used in this specification, such as server certificates, are public objects and usually not sensitive in this regard, others may be. Those who implement and deploy this specification should therefore make an informed decision whether the cached information is inline with their security and privacy goals. In case of concerns, it is advised to avoid sending the fingerprint of the data objects in clear.

The hash algorithm used in this specification is required to have reasonable random properties in order to provide reasonably unique identifiers. There is no requirement that this hash algorithm must have strong collision resistance.

7. IANA Considerations

7.1. New Entry to the TLS ExtensionType Registry

IANA is requested to add an entry to the existing TLS ExtensionType registry, defined in RFC 5246 [RFC5246], for `cached_information(TBD)` defined in this document.

7.2. New Registry for CachedInformationType

IANA is requested to establish a registry for TLS CachedInformationType values. The first entries in the registry are

- o `certificate_chain(1)`
- o `trusted_cas(2)`

The policy for adding new values to this registry, following the terminology defined in RFC 5226 [RFC5226], is as follows:

- o 0-63 (decimal): Standards Action
- o 64-223 (decimal): Specification Required
- o 224-255 (decimal): reserved for Private Use

8. Acknowledgments

We would like to thank the following persons for your detailed document reviews:

- o Paul Wouters and Nikos Mavrogiannopoulos (December 2011)
- o Rob Stradling (February 2012)
- o Ondrej Mikle (in March 2012)

Additionally, we would like to thank the TLS working group chairs, Eric Rescorla and Joe Salowey, as well as the security area directors, Sean Turner and Stephen Farrell, for their feedback and support.

9. References

9.1. Normative References

- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.
- [RFC3874] Housley, R., "A 224-bit One-way Hash Function: SHA-224", RFC 3874, September 2004.
- [RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008.
- [RFC6066] Eastlake, D., "Transport Layer Security (TLS) Extensions: Extension Definitions", RFC 6066, January 2011.

9.2. Informative References

- [I-D.ietf-tls-oob-pubkey] Wouters, P., Tschofenig, H., Gilmore, J., Weiler, S., and T. Kivinen, "Out-of-Band Public Key Validation for Transport Layer Security (TLS)", draft-ietf-tls-oob-pubkey-07 (work in progress), February 2013.
- [RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, May 2008.
- [RFC6574] Tschofenig, H. and J. Arkko, "Report from the Smart Object Workshop", RFC 6574, April 2012.

Authors' Addresses

Stefan Santesson
3xA Security AB
Scheelev. 17
Lund 223 70
Sweden

Email: sts@aaa-sec.com

Hannes Tschofenig
Nokia Siemens Networks
Linnoitustie 6
Espoo 02600
Finland

Phone: +358 (50) 4871445
Email: Hannes.Tschofenig@gmx.net
URI: <http://www.tschofenig.priv.at>

TLS
Internet-Draft
Intended status: Standards Track
Expires: August 19, 2013

P. Wouters, Ed.
Red Hat
H. Tschofenig, Ed.
Nokia Siemens Networks
J. Gilmore

S. Weiler
SPARTA, Inc.
T. Kivinen
AuthenTec
February 15, 2013

Out-of-Band Public Key Validation for Transport Layer Security (TLS)
draft-ietf-tls-oob-pubkey-07.txt

Abstract

This document specifies a new certificate type for exchanging raw public keys in Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) for use with out-of-band public key validation. Currently, TLS authentication can only occur via X.509-based Public Key Infrastructure (PKI) or OpenPGP certificates. By specifying a minimum resource for raw public key exchange, implementations can use alternative public key validation methods.

One such alternative public key validation method is offered by the DNS-Based Authentication of Named Entities (DANE) together with DNS Security. Another alternative is to utilize pre-configured keys, as is the case with sensors and other embedded devices. The usage of raw public keys, instead of X.509-based certificates, leads to a smaller code footprint.

This document introduces the support for raw public keys in TLS.

Status of this Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <http://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference

material or to cite them other than as "work in progress."

This Internet-Draft will expire on August 19, 2013.

Copyright Notice

Copyright (c) 2013 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	4
2. Terminology	4
3. New TLS Extension	5
4. TLS Handshake Extension	8
4.1. Client Hello	8
4.2. Server Hello	9
4.3. Certificate Request	9
4.4. Other Handshake Messages	9
4.5. Client authentication	9
5. Examples	10
6. Security Considerations	12
7. IANA Considerations	13
8. Acknowledgements	13
9. References	14
9.1. Normative References	14
9.2. Informative References	14
Appendix A. Example Encoding	15
Authors' Addresses	16

1. Introduction

Traditionally, TLS server public keys are obtained in PKIX containers in-band using the TLS handshake and validated using trust anchors based on a [PKIX] certification authority (CA). This method can add a complicated trust relationship that is difficult to validate. Examples of such complexity can be seen in [Defeating-SSL].

Alternative methods are available that allow a TLS client to obtain the TLS server public key:

- o The TLS server public key is obtained from a DNSSEC secured resource records using DANE [RFC6698].
- o The TLS server public key is obtained from a [PKIX] certificate chain from an Lightweight Directory Access Protocol (LDAP) [LDAP] server.
- o The TLS client and server public key is provisioned into the operating system firmware image, and updated via software updates.

Some smart objects use the UDP-based Constrained Application Protocol (CoAP) [I-D.ietf-core-coap] to interact with a Web server to upload sensor data at a regular intervals, such as temperature readings. CoAP [I-D.ietf-core-coap] can utilize DTLS for securing the client-to-server communication. As part of the manufacturing process, the embedded device may be configured with the address and the public key of a dedicated CoAP server, as well as a public key for the client itself. The usage of X.509-based PKIX certificates [PKIX] may not suit all smart object deployments and would therefore be an unnecessary burden.

The Transport Layer Security (TLS) Protocol Version 1.2 [RFC5246] provides a framework for extensions to TLS as well as guidelines for designing such extensions. This document registers a new value to the IANA certificate types registry for the support of raw public keys.

2. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

3. New TLS Extension

This section describes the changes to the TLS handshake message contents when raw public key certificates are to be used. Figure 4 illustrates the exchange of messages as described in the sub-sections below. The client and the server exchange make use of two new TLS extensions, namely 'client_certificate_type' and 'server_certificate_type', and an already available IANA TLS Certificate Type registry [TLS-Certificate-Types-Registry] to indicate their ability and desire to exchange raw public keys. These raw public keys, in the form of a SubjectPublicKeyInfo structure, are then carried inside the Certificate payload. The Certificate and the SubjectPublicKeyInfo structure is shown in Figure 1.

```
opaque ASN.1Cert<1..2^24-1>;

struct {
    select(certificate_type){
        // certificate type defined in this document.
        case RawPublicKey:
            opaque ASN.1_subjectPublicKeyInfo<1..2^24-1>;

        // X.509 certificate defined in RFC 5246
        case X.509:
            ASN.1Cert certificate_list<0..2^24-1>;

        // Additional certificate type based on TLS
        // Certificate Type Registry
    };
} Certificate;
```

Figure 1: TLS Certificate Structure.

The SubjectPublicKeyInfo structure is defined in Section 4.1 of RFC 5280 [PKIX] and does not only contain the raw keys, such as the public exponent and the modulus of an RSA public key, but also an algorithm identifier. The structure, as shown in Figure 2, is encoded in an ASN.1 format and therefore contains length information as well. An example is provided in Appendix A.


```

SubjectPublicKeyInfo ::= SEQUENCE {
    algorithm           AlgorithmIdentifier,
    subjectPublicKey     BIT STRING }

```

Figure 2: SubjectPublicKeyInfo ASN.1 Structure.

The algorithm identifiers are Object Identifiers (OIDs). RFC 3279 [RFC3279], for example, defines the following OIDs shown in Figure 3.

Key Type	Document	OID
RSA	Section 2.3.1 of RFC 3279	1.2.840.113549.1.1
.....
Digital Signature Algorithm (DSS)	Section 2.3.2 of RFC 3279	1.2.840.10040.4.1
.....
Elliptic Curve Digital Signature Algorithm (ECDSA)	Section 2.3.5 of RFC 3279	1.2.840.10045.2.1

Figure 3: Example Algorithm Identifiers.

The message exchange in Figure 4 shows the 'client_certificate_type' and 'server_certificate_type' extensions added to the client and server hello messages.

```

client_hello,
client_certificate_type
server_certificate_type  ->

                                <-  server_hello,
                                client_certificate_type,
                                server_certificate_type,
                                certificate,
                                server_key_exchange,
                                certificate_request,
                                server_hello_done

certificate,
client_key_exchange,
certificate_verify,
change_cipher_spec,
finished                                ->

                                <-  change_cipher_spec,
                                finished

Application Data          <----->          Application Data

```

Figure 4: Basic Raw Public Key TLS Exchange.

The semantic of the two extensions is defined as follows:

The 'client_certificate_type' and 'server_certificate_type' sent in the client hello, may carry a list of supported certificate types, sorted by client preference. It is a list in the case where the client supports multiple certificate types. These extension MUST be omitted if the client only supports X.509 certificates. The 'client_certificate_type' sent in the client hello indicates the certificate types the client is able to provide to the server, when requested using a certificate_request message. The 'server_certificate_type' in the client hello indicates the type of certificates the client is able to process when provided by the server in a subsequent certificate payload.

The 'client_certificate_type' returned in the server hello indicates the certificate type found in the attached certificate payload. Only a single value is permitted. The 'server_certificate_type' in the server hello indicates the type of certificates the client is requested to provide in a subsequent certificate payload. The value conveyed in the 'server_certificate_type' MUST be selected from one of the values provided in the 'server_certificate_type' sent in the client hello. If the server does not send a certificate_request payload

or none of the certificates supported by the client (as indicated in the 'server_certificate_type' in the client hello) match the server-supported certificate types the 'server_certificate_type' payload sent in the server hello is omitted.

The "extension_data" field of this extension contains the ClientCertTypeExtension or the ServerCertTypeExtension structure, as shown in Figure 5. The CertificateType structure is an enum with values from TLS Certificate Type Registry.

```
struct {  
    select(ClientOrServerExtension)  
        case client:  
            CertificateType client_certificate_types<1..2^8-1>;  
        case server:  
            CertificateType client_certificate_type;  
    }  
} ClientCertTypeExtension;  
  
struct {  
    select(ClientOrServerExtension)  
        case client:  
            CertificateType server_certificate_types<1..2^8-1>;  
        case server:  
            CertificateType server_certificate_type;  
    }  
} ServerCertTypeExtension;
```

Figure 5: CertTypeExtension Structure.

No new cipher suites are required to use raw public keys. All existing cipher suites that support a key exchange method compatible with the defined extension can be used.

4. TLS Handshake Extension

4.1. Client Hello

In order to indicate the support of out-of-band raw public keys, clients MUST include the 'client_certificate_type' and 'server_certificate_type' extensions extended client hello message. The hello extension mechanism is described in TLS 1.2 [RFC5246].

4.2. Server Hello

If the server receives a client hello that contains the 'client_certificate_type' and 'server_certificate_type' extensions and chooses a cipher suite then three outcomes are possible:

1. The server does not support the extension defined in this document. In this case the server returns the server hello without the extensions defined in this document.
2. The server supports the extension defined in this document and has at least one certificate type in common with the client. In this case it returns the 'server_certificate_type' and indicates the selected certificate type value.
3. The server supports the extension defined in this document but does not have a certificate type in common with the client. In this case the server terminate the session with a fatal alert of type "unsupported_certificate".

If the TLS server also requests a certificate from the client (via the certificate_request) it MUST include the 'client_certificate_type' extension with a value chosen from the list of client-supported certificates types (as provided in the 'client_certificate_type' of the client hello).

If the client indicated the support of raw public keys in the 'client_certificate_type' extension in the client hello and the server is able to provide such raw public key then the TLS server MUST place the SubjectPublicKeyInfo structure into the Certificate payload. The public key algorithm MUST match the selected key exchange algorithm.

4.3. Certificate Request

The semantics of this message remain the same as in the TLS specification.

4.4. Other Handshake Messages

All the other handshake messages are identical to the TLS specification.

4.5. Client authentication

Client authentication by the TLS server is supported only through authentication of the received client SubjectPublicKeyInfo via an out-of-band method.

5. Examples

Figure 6, Figure 7, and Figure 8 illustrate example exchanges.

The first example shows an exchange where the TLS client indicates its ability to receive and validate raw public keys from the server. In our example the client is quite restricted since it is unable to process other certificate types sent by the server. It also does not have credentials (at the TLS layer) it could send. The 'client_certificate_type' extension indicates this in [1]. When the TLS server receives the client hello it processes the 'client_certificate_type' extension. Since it also has a raw public key it indicates in [2] that it had chosen to place the SubjectPublicKeyInfo structure into the Certificate payload [3]. The client uses this raw public key in the TLS handshake and an out-of-band technique, such as DANE, to verify its validity.

```

client_hello,
server_certificate_type=(RawPublicKey) -> // [1]

      <- server_hello,
      server_certificate_type=(RawPublicKey), // [2]
      certificate, // [3]
      server_key_exchange,
      server_hello_done

client_key_exchange,
change_cipher_spec,
finished                                     ->

      <- change_cipher_spec,
      finished

Application Data          <----->      Application Data

```

Figure 6: Example with Raw Public Key provided by the TLS Server

In our second example the TLS client as well as the TLS server use raw public keys. This is a use case envisioned for smart object networking. The TLS client in this case is an embedded device that is configured with a raw public key for use with TLS and is also able to process raw public keys sent by the server. Therefore, it indicates these capabilities in [1]. As in the previously shown example the server fulfills the client's request, indicates this via the "RawPublicKey" value in the server_certificate_type payload, and

provides a raw public key into the Certificate payload back to the client (see [3]). The TLS server, however, demands client authentication and therefore a `certificate_request` is added [4]. The `certificate_type` payload in [2] indicates that the TLS server accepts raw public keys. The TLS client, who has a raw public key pre-provisioned, returns it in the Certificate payload [5] to the server.

```

client_hello,
client_certificate_type=(RawPublicKey) // [1]
server_certificate_type=(RawPublicKey) // [1]
->
    <-  server_hello,
        server_certificate_type=(RawPublicKey)//[2]
        certificate, // [3]
        client_certificate_type=(RawPublicKey)//[4]
        certificate_request, // [4]
        server_key_exchange,
        server_hello_done

certificate, // [5]
client_key_exchange,
change_cipher_spec,
finished
->
    <-  change_cipher_spec,
        finished

Application Data    <----->    Application Data

```

Figure 7: Example with Raw Public Key provided by the TLS Server and the Client

In our last example we illustrate a combination of raw public key and X.509 usage. The client uses a raw public key for client authentication but the server provides an X.509 certificate. This exchange starts with the client indicating its ability to process X.509 certificates provided by the server, and the ability to send raw public keys (see [1]). The server provides the X.509 certificate in [3] with the indication present in [2]. For client authentication the server indicates in [4] that it selected the raw public key format and requests a certificate from the client in [5]. The TLS client provides a raw public key in [6] after receiving and processing the TLS server hello message.

```

client_hello,
server_certificate_type=(X.509)
client_certificate_type=(RawPublicKey) // [1]
->
<-  server_hello,
    server_certificate_type=(X.509)//[2]
    certificate, // [3]
    client_certificate_type=(RawPublicKey)//[4]
    certificate_request, // [5]
    server_key_exchange,
    server_hello_done

certificate, // [6]
client_key_exchange,
change_cipher_spec,
finished
->

<-  change_cipher_spec,
    finished

Application Data      <----->      Application Data

```

Figure 8: Hybrid Certificate Example

6. Security Considerations

The transmission of raw public keys, as described in this document, provides benefits by lowering the over-the-air transmission overhead since raw public keys are quite naturally smaller than an entire certificate. There are also advantages from a codesize point of view for parsing and processing these keys. The cryptographic procedures for associating the public key with the possession of a private key also follows standard procedures.

The main security challenge is, however, how to associate the public key with a specific entity. This information will be needed to make authorization decisions. Without a secure binding, man-in-the-middle attacks may be the consequence. This document assumes that such binding can be made out-of-band and we list a few examples in Section 1. DANE [RFC6698] offers one such approach. If public keys are obtained using DANE, these public keys are authenticated via DNSSEC. Pre-configured keys is another out of band method for authenticating raw public keys. While pre-configured keys are not suitable for a generic Web-based e-commerce environment such keys are a reasonable approach for many smart object deployments where there is a close relationship between the software running on the device and the server-side communication endpoint. Regardless of the chosen

mechanism for out-of-band public key validation an assessment of the most suitable approach has to be made prior to the start of a deployment to ensure the security of the system.

7. IANA Considerations

IANA is asked to register a new value in the "TLS Certificate Types" registry of Transport Layer Security (TLS) Extensions [TLS-Certificate-Types-Registry], as follows:

Value: 2
Description: Raw Public Key
Reference: [[THIS RFC]]

This document asks IANA to allocate two new TLS extensions, "client_certificate_type" and "server_certificate_type", from the TLS ExtensionType registry defined in [RFC5246]. These extensions are used in both the client hello message and the server hello message. The new extension type is used for certificate type negotiation. The values carried in these extensions are taken from the TLS Certificate Types registry [TLS-Certificate-Types-Registry].

8. Acknowledgements

The feedback from the TLS working group meeting at IETF#81 has substantially shaped the document and we would like to thank the meeting participants for their input. The support for hashes of public keys has been moved to [I-D.ietf-tls-cached-info] after the discussions at the IETF#82 meeting.

We would like to thank the following persons for their review comments: Martin Rex, Bill Frantz, Zach Shelby, Carsten Bormann, Cullen Jennings, Rene Struik, Alper Yegin, Jim Schaad, Barry Leiba, Paul Hoffman, Robert Cragie, Nikos Mavrogiannopoulos, Phil Hunt, John Bradley, Klaus Hartke, Stefan Jucker, Kovatsch Matthias, Daniel Kahn Gillmor, and James Manger. Nikos Mavrogiannopoulos contributed the design for re-using the certificate type registry. Barry Leiba contributed guidance for the IANA consideration text. Stefan Jucker, Kovatsch Matthias, and Klaus Hartke provided implementation feedback regarding the SubjectPublicKeyInfo structure.

Finally, we would like to thank our TLS working group chairs, Eric Rescorla and Joe Salowey, for their guidance and support.

9. References

9.1. Normative References

- [PKIX] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, May 2008.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.
- [RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008.
- [TLS-Certificate-Types-Registry] "TLS Certificate Types Registry", February 2013, <<http://www.iana.org/assignments/tls-extensiontype-values#tls-extensiontype-values-2>>.

9.2. Informative References

- [ASN.1-Dump] Gutmann, P., "ASN.1 Object Dump Program", February 2013, <<http://www.cs.auckland.ac.nz/~pgut001/>>.
- [Defeating-SSL] Marlinspike, M., "New Tricks for Defeating SSL in Practice", February 2009, <<http://www.blackhat.com/presentations/bh-dc-09/Marlinspike/BlackHat-DC-09-Marlinspike-Defeating-SSL.pdf>>.
- [I-D.ietf-core-coap] Shelby, Z., Hartke, K., Bormann, C., and B. Frank, "Constrained Application Protocol (CoAP)", draft-ietf-core-coap-13 (work in progress), December 2012.
- [I-D.ietf-tls-cached-info] Santesson, S. and H. Tschofenig, "Transport Layer Security (TLS) Cached Information Extension", draft-ietf-tls-cached-info-13 (work in progress), September 2012.
- [LDAP] Sermersheim, J., "Lightweight Directory Access Protocol (LDAP): The Protocol", RFC 4511, June 2006.
- [RFC3279] Bassham, L., Polk, W., and R. Housley, "Algorithms and Identifiers for the Internet X.509 Public Key

Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 3279, April 2002.

[RFC6698] Hoffman, P. and J. Schlyter, "The DNS-Based Authentication of Named Entities (DANE) Transport Layer Security (TLS) Protocol: TLSA", RFC 6698, August 2012.

Appendix A. Example Encoding

For example, the following hex sequence describes a SubjectPublicKeyInfo structure inside the certificate payload:

	0	1	2	3	4	5	6	7	8	9
1	0x30,	0x81,	0x9f,	0x30,	0x0d,	0x06,	0x09,	0x2a,	0x86,	0x48,
2	0x86,	0xf7,	0x0d,	0x01,	0x01,	0x01,	0x05,	0x00,	0x03,	0x81,
3	0x8d,	0x00,	0x30,	0x81,	0x89,	0x02,	0x81,	0x81,	0x00,	0xcd,
4	0xfd,	0x89,	0x48,	0xbe,	0x36,	0xb9,	0x95,	0x76,	0xd4,	0x13,
5	0x30,	0x0e,	0xbf,	0xb2,	0xed,	0x67,	0x0a,	0xc0,	0x16,	0x3f,
6	0x51,	0x09,	0x9d,	0x29,	0x2f,	0xb2,	0x6d,	0x3f,	0x3e,	0x6c,
7	0x2f,	0x90,	0x80,	0xa1,	0x71,	0xdf,	0xbe,	0x38,	0xc5,	0xcb,
8	0xa9,	0x9a,	0x40,	0x14,	0x90,	0x0a,	0xf9,	0xb7,	0x07,	0x0b,
9	0xe1,	0xda,	0xe7,	0x09,	0xbf,	0x0d,	0x57,	0x41,	0x86,	0x60,
10	0xa1,	0xc1,	0x27,	0x91,	0x5b,	0x0a,	0x98,	0x46,	0x1b,	0xf6,
11	0xa2,	0x84,	0xf8,	0x65,	0xc7,	0xce,	0x2d,	0x96,	0x17,	0xaa,
12	0x91,	0xf8,	0x61,	0x04,	0x50,	0x70,	0xeb,	0xb4,	0x43,	0xb7,
13	0xdc,	0x9a,	0xcc,	0x31,	0x01,	0x14,	0xd4,	0xcd,	0xcc,	0xc2,
14	0x37,	0x6d,	0x69,	0x82,	0xd6,	0xc6,	0xc4,	0xbe,	0xf2,	0x34,
15	0xa5,	0xc9,	0xa6,	0x19,	0x53,	0x32,	0x7a,	0x86,	0x0e,	0x91,
16	0x82,	0x0f,	0xa1,	0x42,	0x54,	0xaa,	0x01,	0x02,	0x03,	0x01,
17	0x00,	0x01								

Figure 9: Example SubjectPublicKeyInfo Structure Byte Sequence.

The decoded byte-sequence shown in Figure 9 (for example using Peter's ASN.1 decoder [ASN.1-Dump]) illustrates the structure, as shown in Figure 10.

Offset	Length	Description
0	3+159:	SEQUENCE {
3	2+13:	SEQUENCE {
5	2+9:	OBJECT IDENTIFIER Value (1 2 840 113549 1 1 1)
	:	PKCS #1, rsaEncryption
16	2+0:	NULL
	:	}
18	3+141:	BIT STRING, encapsulates {
22	3+137:	SEQUENCE {
25	3+129:	INTEGER Value (1024 bit)
157	2+3:	INTEGER Value (65537)
	:	}
	:	}
	:	}

Figure 10: Decoding of Example SubjectPublicKeyInfo Structure.

Authors' Addresses

Paul Wouters (editor)
Red Hat

Email: paul@nohats.ca

Hannes Tschofenig (editor)
Nokia Siemens Networks
Linnoitustie 6
Espoo 02600
Finland

Phone: +358 (50) 4871445
Email: Hannes.Tschofenig@gmx.net
URI: <http://www.tschofenig.priv.at>

John Gilmore
PO Box 170608
San Francisco, California 94117
USA

Phone: +1 415 221 6524
Email: gnu@toad.com
URI: <https://www.toad.com/>

Samuel Weiler
SPARTA, Inc.
7110 Samuel Morse Drive
Columbia, Maryland 21046
US

Email: weiler@tislabs.com

Tero Kivinen
AuthenTec
Eerikinkatu 28
HELSINKI FI-00180
FI

Email: kivinen@iki.fi

Transport Layer Security
Internet-Draft
Intended status: Standards Track
Expires: July 25, 2013

D. Harkins, Ed.
Aruba Networks
D. Halasz, Ed.
Halasz Ventures
January 21, 2013

Secure Password Ciphersuites for Transport Layer Security (TLS)
draft-ietf-tls-pwd-00

Abstract

This memo defines several new ciphersuites for the Transport Layer Security (TLS) protocol to support certificate-less, secure authentication using only a simple, low-entropy, password. The ciphersuites are all based on an authentication and key exchange protocol that is resistant to off-line dictionary attack.

Status of this Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <http://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on July 25, 2013.

Copyright Notice

Copyright (c) 2013 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as

described in the Simplified BSD License.

Table of Contents

1. Background	3
1.1. The Case for Certificate-less Authentication	3
1.2. Resistance to Dictionary Attack	3
2. Keyword Definitions	4
3. Introduction	4
3.1. Notation	4
3.2. Discrete Logarithm Cryptography	5
3.2.1. Elliptic Curve Cryptography	5
3.2.2. Finite Field Cryptography	6
3.3. Instantiating the Random Function	7
3.4. Passwords	7
3.5. Assumptions	8
4. Specification of the TLS-PWD Handshake	8
4.1. Fixing the Password Element	9
4.1.1. Computing an ECC Password Element	10
4.1.2. Computing an FFC Password Element	11
4.2. Changes to Handshake Message Contents	12
4.2.1. Client Hello Changes	12
4.2.2. Server Key Exchange Changes	13
4.2.2.1. Generation of ServerKeyExchange	14
4.2.2.2. Processing of ServerKeyExchange	15
4.2.3. Client Key Exchange Changes	15
4.2.3.1. Generation of Client Key Exchange	16
4.2.3.2. Processing of Client Key Exchange	16
4.3. Computing the Premaster Secret	16
5. Ciphersuite Definition	17
6. Acknowledgements	18
7. IANA Considerations	18
8. Security Considerations	19
9. Implementation Considerations	22
10. References	23
10.1. Normative References	23
10.2. Informative References	23
Authors' Addresses	24

1. Background

1.1. The Case for Certificate-less Authentication

TLS usually uses public key certificates for authentication [RFC5246]. This is problematic in some cases:

- o Frequently, TLS [RFC5246] is used in devices owned, operated, and provisioned by people who lack competency to properly use certificates and merely want to establish a secure connection using a more natural credential like a simple password. The proliferation of deployments that use a self-signed server certificate in TLS [RFC5246] followed by a PAP-style exchange over the unauthenticated channel underscores this case.
- o A password is a more natural credential than a certificate (from early childhood people learn the semantics of a shared secret), so a password-based TLS ciphersuite can be used to protect an HTTP-based certificate enrollment scheme-- e.g. an [RFC5967] -style request and an [RFC5751] -style response-- to parlay a simple password into a certificate for subsequent use with any certificate-based authentication protocol. This addresses a significant "chicken-and-egg" dilemma found with certificate-only use of [RFC5246].
- o Some PIN-code readers will transfer the entered PIN to a smart card in clear text. Assuming a hostile environment, this is a bad practice. A password-based TLS ciphersuite can enable the establishment of an authenticated connection between reader and card based on the PIN.

1.2. Resistance to Dictionary Attack

It is a common misconception that a protocol that authenticates with a shared and secret credential is resistant to dictionary attack if the credential is assumed to be an N-bit uniformly random secret, where N is sufficiently large. The concept of resistance to dictionary attack really has nothing to do with whether that secret can be found in a standard collection of a language's defined words (i.e. a dictionary). It has to do with how an adversary gains an advantage in attacking the protocol.

For a protocol to be resistant to dictionary attack any advantage an adversary can gain must be a function of the amount of interactions she makes with an honest protocol participant and not a function of the amount of computation she uses. The adversary will not be able to obtain any information about the password except whether a single guess from a single protocol run which she took part in is correct or

incorrect.

It is assumed that the attacker has access to a pool of data from which the secret was drawn-- it could be all numbers between 1 and 2^N , it could be all defined words in a dictionary. The key is that the attacker cannot do a an attack and then enumerate through the pool trying potential secrets (computation) to see if one is correct. She must do an active attack for each secret she wishes to try (interaction) and the only information she can glean from that attack is whether the secret used with that particular attack is correct or not.

2. Keyword Definitions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

3. Introduction

3.1. Notation

The following notation is used in this memo:

password

a secret, and potentially low-entropy word, phrase, code or key used as a credential for authentication. The password is shared between the TLS client and TLS server.

$y = H(x)$

a binary string of arbitrary length, x , is given to a function H which produces a fixed-length output, y .

$a \mid b$

denotes concatenation of string a with string b .

$[a]b$

indicates a string consisting of the single bit " a " repeated " b " times.

$x \bmod y$

indicates the remainder of division of x by y . The result will be between 0 and y .

LSB(x)

returns the least-significant bit of the bitstring "x".

3.2. Discrete Logarithm Cryptography

The ciphersuites defined in this memo use discrete logarithm cryptography (see [SP800-56A]) to produce an authenticated and shared secret value that is an element in a group defined by a set of domain parameters. The domain parameters can be based on either Finite Field Cryptography (FFC) or Elliptic Curve Cryptography (EEC).

Elements in a group, either an FFC or EEC group, are indicated using upper-case while scalar values are indicated using lower-case.

3.2.1. Elliptic Curve Cryptography

The authenticated key exchange defined in this memo uses fundamental algorithms of elliptic curves defined over $GF(p)$ as described in [RFC6090].

Domain parameters for the ECC groups used by this memo are:

- o A prime, p , determining a prime field $GF(p)$. The cryptographic group will be a subgroup of the full elliptic curve group which consists points on an elliptic curve-- elements from $GF(p)$ that satisfy the curve's equation-- together with the "point at infinity" that serves as the identity element.
- o Elements a and b from $GF(p)$ that define the curve's equation. The point (x,y) in $GF(p) \times GF(p)$ is on the elliptic curve if and only if $(y^2 - x^3 - a*x - b) \bmod p$ equals zero (0).
- o A point, G , on the elliptic curve, which serves as a generator for the ECC group. G is chosen such that its order, with respect to elliptic curve addition, is a sufficiently large prime.
- o A prime, q , which is the order of G , and thus is also the size of the cryptographic subgroup that is generated by G .
- o A co-factor, f , defined by the requirement that the size of the full elliptic curve group (including the "point at infinity") is the product of f and q .

This memo uses the following ECC Functions:

- o $Z = \text{elem-op}(X,Y) = X + Y$: two points on the curve X and Y , are summed to produce another point on the curve, Z . This is the group operation for ECC groups.

- o $Z = \text{scalar-op}(x,Y) = x * Y$: an integer scalar, x , acts on a point on the curve, Y , via repetitive addition (Y is added to itself x times), to produce another ECC element, Z .
- o $Y = \text{inverse}(X)$: a point on the curve, X , has an inverse, Y , which is also a point on the curve, when their sum is the "point at infinity" (the identity for elliptic curve addition). In other words, $R + \text{inverse}(R) = "0"$.
- o $z = F(X)$: the x -coordinate of a point (x, y) on the curve is returned. This is a mapping function to convert a group element into an integer.

Only ECC groups over $GF(p)$ can be used with TLS-PWD. ECC groups over $GF(2^m)$ SHALL NOT be used by TLS-PWD. In addition, ECC groups with a co-factor greater than one (1) SHALL NOT be used by TLS-PWD.

A composite (x, y) pair can be validated as an a point on the elliptic curve by checking whether: 1) both coordinates x and y are greater than zero (0) and less than the prime defining the underlying field; 2) the x - and y - coordinates satisfy the equation of the curve; and 3) they do not represent the point-at-infinity "0". If any of those conditions are not true the (x, y) pair is not a valid point on the curve.

3.2.2. Finite Field Cryptography

Domain parameters for the FFC groups used by this memo are:

- o A prime, p , determining a prime field $GF(p)$, the integers modulo p . The FFC group will be a subgroup of $GF(p)^*$, the multiplicative group of non-zero elements in $GF(p)$.
- o An element, G , in $GF(p)^*$ which serves as a generator for the FFC group. G is chosen such that its multiplicative order is a sufficiently large prime divisor of $((p-1)/2)$.
- o A prime, q , which is the multiplicative order of G , and thus also the size of the cryptographic subgroup of $GF(p)^*$ that is generated by G .

This memo uses the following FFC Functions:

- o $Z = \text{elem-op}(X,Y) = (X * Y) \bmod p$: two FFC elements, X and Y , are multiplied modulo the prime, p , to produce another FFC element, Z . This is the group operation for FFC groups.

- o $Z = \text{scalar-op}(x,Y) = Y^x \bmod p$: an integer scalar, x , acts on an FFC group element, Y , via exponentiation modulo the prime, p , to produce another FFC element, Z .
- o $Y = \text{inverse}(X)$: a group element, X , has an inverse, Y , when the product of the element and its inverse modulo the prime equals one (1). In other words, $(X * \text{inverse}(X)) \bmod p = 1$.
- o $z = F(X)$: is the identity function since an element in an FFC group is already an integer. It is included here for consistency in the specification.

Many FFC groups used in IETF protocols are based on safe primes and do not define an order (q). For these groups, the order (q) used in this memo shall be the prime of the group minus one divided by two-- $(p-1)/2$.

An integer can be validated as being an element in an FFC group by checking whether: 1) it is between one (1) and the prime, p , exclusive; and 2) if modular exponentiation of the integer by the group order, q , equals one (1). If either of these conditions are not true the integer is not an element in the group.

3.3. Instantiating the Random Function

The protocol described in this memo uses a random function, H , which is modeled as a "random oracle". At first glance, one may view this as a hash function. As noted in [RANDOR], though, hash functions are too structured to be used directly as a random oracle. But they can be used to instantiate the random oracle.

The random function, H , in this memo is instantiated by using the hash algorithm defined by the particular TLS-PWD ciphersuite in HMAC mode with a key whose length is equal to block size of the hash algorithm and whose value is zero. For example, if the ciphersuite is TLS_ECCPWD_WITH_AES_128_GCM_SHA256 then H will be instantiated with SHA256 as:

$$H(x) = \text{HMAC-SHA256}([0]_{32}, x)$$

3.4. Passwords

The authenticated key exchange used in TLS-PWD requires each side to have a common view of a shared credential. To protect a database of stored passwords, though, the password SHALL be salted and the result, called the base, SHALL be used as the authentication credential.

The salting function is defined as:

```
base = HMAC-SHA256(salt, username | password)
```

The password used for generation of the base SHALL be represented as a UTF-8 encoded character string processed according to the rules of the [RFC4013] profile of [RFC3454] and the salt SHALL be a 32 octet random number. The server SHALL store a triplet of the form:

```
{ username, base, salt }
```

And the client SHALL generate the base upon receiving the salt from the server.

3.5. Assumptions

The security properties of the authenticated key exchange defined in this memo are based on a number of assumptions:

1. The random function, H , is a "random oracle" as defined in [RANDOR].
2. The discrete logarithm problem for the chosen group is hard. That is, given g , p , and $y = g^x \bmod p$, it is computationally infeasible to determine x . Similarly, for an ECC group given the curve definition, a generator G , and $Y = x * G$, it is computationally infeasible to determine x .
3. Quality random numbers with sufficient entropy can be created. This may entail the use of specialized hardware. If such hardware is unavailable a cryptographic mixing function (like a strong hash function) to distill entropy from multiple, uncorrelated sources of information and events may be needed. A very good discussion of this can be found in [RFC4086].

4. Specification of the TLS-PWD Handshake

The authenticated key exchange is accomplished by each side deriving a password-based element, PE , in the chosen group, making a "commitment" to a single guess of the password using PE , and generating the Premaster Secret. The ability of each side to produce a valid finished message authenticates itself to the other side.

The authenticated key exchange is dropped into the standard TLS message handshake by modifying some of the messages.

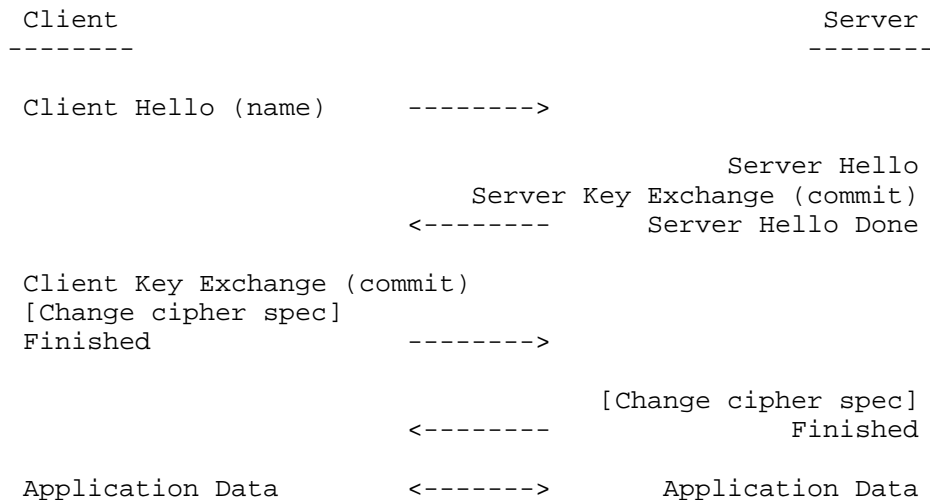


Figure 1

4.1. Fixing the Password Element

Prior to making a "commitment" both sides must generate a secret element, PE, in the chosen group using the common password-derived base. The server generates PE after it receives the Client Hello and chooses the particular group to use, and the client generates PE upon receipt of the Server Key Exchange.

Fixing the password element involves an iterative "hunting and pecking" technique using the prime from the negotiated group's domain parameter set and an ECC- or FFC-specific operation depending on the negotiated group.

To thwart side channel attacks which attempt to determine the number of iterations of the "hunting-and-pecking" loop are used to find PE for a given password, a security parameter, k , is used to ensure that at least k iterations are always performed.

First, an 8-bit counter is set to the value one (1). Then, H is used to generate a password seed from the a counter, the prime of the selected group, and the base (which is derived from the username, password, and salt):

$$\text{pwd-seed} = H(\text{base} \mid \text{counter} \mid p)$$

Then, using the technique from section B.5.1 of [FIPS186-3], the pwd-seed is expanded using the PRF to the length of the prime from the negotiated group's domain parameter set plus a constant sixty-four

(64) to produce an intermediate pwd-tmp which is modularly reduced to create pwd-value:

```
n = p + 64
pwd-tmp = PRF(pwd-seed, "TLS-PWD Hunting And Pecking",
              ClientHello.random | ServerHello.random) [0..n];
pwd-value = (pwd-tmp mod (p-1)) + 1
```

The pwd-value is then passed to the group-specific operation which either returns the selected password element or fails. If the group-specific operation fails, the counter is incremented, a new pwd-seed is generated, and the hunting-and-pecking continues. This process continues until the group-specific operation returns the password element. After the password element has been chosen, the base is changed to a random number, the counter is incremented and the hunting-and-pecking continues until the counter is greater than the security parameter, k .

The probability that one requires more than " n " iterations of the "hunting and pecking" loop to find an ECC PE is roughly $(q/2p)^n$ and to find an FFC PE is roughly $(q/p)^n$, both of which rapidly approach zero (0) as " n " increases. The security parameter, k , SHOULD be set sufficiently large such that the probability that finding PE would take more than k iterations is sufficiently small (see Section 8).

When PE has been discovered, pwd-seed, pwd-tmp, and pwd-value SHALL be irretrievably destroyed.

4.1.1.1. Computing an ECC Password Element

The group-specific operation for ECC groups uses pwd-value, pwd-seed, and the equation for the curve to produce PE. First, pwd-value is used directly as the x-coordinate, x , with the equation for the elliptic curve, with parameters a and b from the domain parameter set of the curve, to solve for a y-coordinate, y . If there is no solution to the quadratic equation, this operation fails and the hunting-and-pecking process continues. If a solution is found, then an ambiguity exists as there are technically two solutions to the equation and pwd-seed is used to unambiguously select one of them. If the low-order bit of pwd-seed is equal to the low-order bit of y , then a candidate PE is defined as the point (x, y) ; if the low-order bit of pwd-seed differs from the low-order bit of y , then a candidate PE is defined as the point $(x, p - y)$, where p is the prime over which the curve is defined. The candidate PE becomes PE, a random number is used instead of the base, and the hunting and pecking continues until it has looped through k iterations.

Algorithmically, the process looks like this:

```

found = 0
counter = 0
base = H(username | password | salt)
n = p + 64
do {
  counter = counter + 1
  pwd-seed = H(base | counter | p)
  pwd-tmp = PRF(pwd-seed, "TLS-PWD Hunting And Pecking",
                ClientHello.random | ServerHello.random) [0..n]
  pwd-value = (pwd-tmp mod (p-1)) + 1
  x = pwd-value
  if ( (y = sqrt(x^3 + ax + b)) != FAIL)
  then
    if (found == 0)
    then
      if (LSB(y) == LSB(pwd-seed))
      then
        PE = (x, y)
      else
        PE = (x, p-y)
      fi
      found = 1
      base = random()
    fi
  fi
} while ((found == 0) || (counter <= k))

```

Figure 2: Fixing PE for ECC Groups

4.1.2. Computing an FFC Password Element

The group-specific operation for FFC groups takes `pwd-value`, and the prime, `p`, and order, `q`, from the group's domain parameter set (see Section 3.2.2 when the order is not part of the defined domain parameter set) to directly produce a candidate password element, by exponentiating the `pwd-value` to the value $((p-1)/q)$ modulo the prime. If the result is greater than one (1), the candidate password element becomes PE, and the hunting and pecking terminates successfully.

Algorithmically, the process looks like this:


```

found = 0
counter = 0
base = H(username | password | salt)
n = p + 64
do {
  counter = counter + 1
  pwd-seed = H(base | counter | p)
  pwd-tmp = PRF(pwd-seed, "TLS-PWD Hunting And Pecking",
                ClientHello.random | ServerHello.random) [0..n]
  pwd-value = (pwd-tmp mod (p-1)) + 1
  PE = pwd-value ^ ((p-1)/q) mod p
  if (PE > 1)
  then
    found = 1
    base = random()
  fi
} while ((found == 0) || (counter <= k))

```

Figure 3: Fixing PE for FFC Groups

4.2. Changes to Handshake Message Contents

4.2.1. Client Hello Changes

The client is required to identify herself to the server by adding a PWD extension to the Client Hello message. The PWD extension uses the standard mechanism defined in [RFC5246]. The "extension data" field of the PWD extension SHALL contain a PWD_name which is used to identify the password shared between the client and server.

```

enum { pwd(TBD) } ExtensionType;

opaque PWD_name<1..2^8-1>;

```

The PWD_name SHALL be UTF-8 encoded character string processed according to the rules of the [RFC4013] profile of [RFC3454].

A client offering a PWD ciphersuite MUST include the PWD extension in her Client Hello.

If a server does not have a password identified by the PWD_name in the PWD extension of the Client Hello, the server SHOULD hide that fact by simulating the protocol-- putting random data in the PWD-specific components of the Server Key Exchange-- and then rejecting the client's finished message with a "bad_record_mac" alert. To properly effect a simulated TLS-PWD exchange, an appropriate delay SHOULD be inserted between receipt of the Client Hello and response of the Server Hello. Alternately, a server MAY choose to terminate

the exchange if a password identified by the PWD_name in the PWD extension of the Client Hello is not found.

The server decides on a group to use with the named user (see Section 9 and generates the password element, PE, according to Section 4.1.2.

4.2.2. Server Key Exchange Changes

The domain parameter set for the selected group MUST be specified in the ServerKeyExchange, either explicitly or, in the case of some elliptic curve groups, by name. In addition to the group specification, the ServerKeyExchange also contains the server's "commitment" in the form of a scalar and element, and the salt which was used to store the user's password.

Two new values have been added to the enumerated KeyExchangeAlgorithm to indicate TLS-PWD using finite field cryptography, ff_pwd, and TLS-PWD using elliptic curve cryptography, ec_pwd.

```
enum { ff_pwd, ec_pwd } KeyExchangeAlgorithms;

struct {
    opaque salt<1..2^8-1>;
    opaque pwd_p<1..2^16-1>;
    opaque pwd_g<1..2^16-1>;
    opaque pwd_q<1..2^16-1>;
    opaque ff_selement<1..2^16-1>;
    opaque ff_sscalar<1..2^16-1>;
} ServerFFPWDParams;

struct
    opaque salt<1..2^8-1>;
    ECPParameters curve_params;
    ECPoint ec_selement;
    opaque ec_sscalar<1..2^8-1>;
} ServerECPWDParams;

struct {
    select (KeyExchangeAlgorithm) {
        case ec_pwd:
            ServerECPWDParams params;
        case ff_pwd:
            ServerFFPWDParams params;
    };
} ServerKeyExchange;
```

4.2.2.1. Generation of ServerKeyExchange

The scalar and Element that comprise the server's "commitment" are generated as follows.

First two random numbers, called private and mask, between zero and the order of the group (exclusive) are generated. If their sum modulo the order of the group, q , equals zero the numbers must be thrown away and new random numbers generated. If their sum modulo the order of the group, q , is greater than zero the sum becomes the scalar.

$$\text{scalar} = (\text{private} + \text{mask}) \bmod q$$

The Element is then calculated as the inverse of the group's scalar operation (see the group specific operations in Section 3.2) with the mask and PE.

$$\text{Element} = \text{inverse}(\text{scalar-op}(\text{mask}, \text{PE}))$$

After calculation of the scalar and Element the mask SHALL be irretrievably destroyed.

4.2.2.1.1. ECC Server Key Exchange

ECC domain parameters are specified, either explicitly or named, in the ECPParameters component of the ECC-specific ServerKeyExchange as defined in [RFC4492]. The scalar SHALL become the `ec_sscalar` component and the Element SHALL become the `ec_selement` of the ServerKeyExchange. If the client requested a specific point format (compressed or uncompressed) with the Support Point Formats Extension (see [RFC4492]) in its Client Hello, the Element MUST be formatted in the `ec_selement` to conform to that request.

As mentioned in Section 3.2.1, elliptic curves over $\text{GF}(2^m)$, so called characteristic-2 curves, and curves with a co-factor greater than one (1) SHALL NOT be used with TLS-PWD.

4.2.2.1.2. FFC Server Key Exchange

FFC domain parameters sent in the ServerKeyExchange are for the group's prime, generator (which is only used for verification of the group specification), and the order of the group's generator. The scalar SHALL become the `ff_sscalar` component and the Element SHALL become the `ff_selement` in the FFC-specific ServerKeyExchange.

As mentioned in Section 3.2.2 if the prime is a safe prime and no order is included in the domain parameter set, the order added to the

ServerKeyExchange SHALL be the prime minus one divided by two--
(p-1)/2.

4.2.2.2. Processing of ServerKeyExchange

Upon receipt of the ServerKeyExchange, the client decides whether to support the indicated group or not. Named elliptic curves are easy to validate-- either they are supported or they are not, but care must be taken with FFC groups and explicitly specified ECC groups. As mentioned in Section 3.5, the discrete logarithm problem MUST be hard for any group used with this memo. The specific steps taken to come to this assurance for a particular group are outside the scope of this memo but they are the same steps to take when using the Diffie-Hellman key exchange with TLS. If the client decides not to support the group indicated in the ServerKeyExchange, she MUST abort the exchange.

If the client decides to support the indicated group the server's "commitment" MUST be validated by ensuring that: 1) the server's scalar value is greater than zero (0) and less than the order of the group, q ; and 2) that the Element is valid for the chosen group (see Section 3.2.2 and Section 3.2.1 for how to determine whether an Element is valid for the particular group. Note that if the Element is a compressed point on an elliptic curve it MUST be uncompressed before checking its validity).

If the group is acceptable, the client extracts the salt from the ServerKeyExchange and generates the password element, PE, according to Section 3.4 and Section 4.1.2.

4.2.3. Client Key Exchange Changes

When the value of KeyExchangeAlgorithm is either ff_pwd or ec_pwd, the ClientKeyExchange is used to convey the client's "commitment" to the server. It, therefore, contains a scalar and an Element.

```
struct {
    opaque ff_celement<1..2^16-1>;
    opaque ff_cscalar<1..2^16-1>;
} ClientFFPWDParams;

struct
    ECPoint ec_celement;
    opaque ec_cscalar<1..2^8-1>;
} ClientECPWDParams;

struct {
    select (KeyExchangeAlgorithm) {
        case ff_pwd: ClientFFPWDParams;
        case ec_pwd: ClientECPWDParams;
    } exchange_keys;
} ClientKeyExchange;
```

4.2.3.1. Generation of Client Key Exchange

The client's scalar and Element are generated in the manner described in Section 4.2.2.1.

For an FFC group, the scalar SHALL become the ff_cscalar component and the Element SHALL become the ff_celement in the FFC-specific ClientKeyExchange.

For an ECC group, the scalar SHALL become the ec_cscalar component and the Element SHALL become the ec_celement in the ECC-specific ClientKeyExchange. If the client requested a specific point format (compressed or uncompressed) with the Support Point Formats Extension in its ClientHello, then the Element MUST be formatted in the ec_celement to conform to its initial request.

4.2.3.2. Processing of Client Key Exchange

The server MUST validate the client's "commitment" by ensuring that: 1) the client's scalar value is greater than zero (0) and less than the order of the group, q ; and 2) that the Element is valid for the chosen group (see Section 3.2.2 and Section 3.2.1 for how to determine whether an Element is valid for a particular group. Note that if the Element is a compressed point on an elliptic curve it MUST be uncompressed before checking its validity.

4.3. Computing the Premaster Secret

The client uses her own scalar and Element, denoted here ClientKeyExchange.scalar and ClientKeyExchange.Element, the server's scalar and Element, denoted here as ServerKeyExchange.scalar and

ServerKeyExchange.Element, and the random private value, denoted here as client.private, she created as part of the generation of her "commit" to compute an intermediate value, z, as indicated:

```
z = F(scalar-op(client.private,
                element-op(ServerKeyExchange.Element,
                           scalar-op(ServerKeyExchange.scalar, PE))))
```

With the same notation as above, the server uses his own scalar and Element, the client's scalar and Element, and his random private value, denoted here as server.private, he created as part of the generation of his "commit" to compute the premaster secret as follows:

```
z = F(scalar-op(server.private,
                element-op(ClientKeyExchange.Element,
                           scalar-op(ClientKeyExchange.scalar, PE))))
```

The intermediate value, z, is then used as the premaster secret after any leading bytes of z that contain all zero bits have been stripped off.

5. Ciphersuite Definition

This memo adds the following ciphersuites:

```
CipherSuite TLS_FFCPWD_WITH_3DES_EDE_CBC_SHA = ( TBD, TBD );
CipherSuite TLS_FFCPWD_WITH_AES_128_CBC_SHA = (TBD, TBD );
CipherSuite TLS_ECCPWD_WITH_AES_128_CBC_SHA = (TBD, TBD );
CipherSuite TLS_ECCPWD_WITH_AES_128_GCM_SHA256 = (TBD, TBD );
CipherSuite TLS_ECCPWD_WITH_AES_256_GCM_SHA384 = (TBD, TBD );
CipherSuite TLS_FFCPWD_WITH_AES_128_CCM_SHA = (TBD, TBD );
CipherSuite TLS_ECCPWD_WITH_AES_128_CCM_SHA = (TBD, TBD );
CipherSuite TLS_ECCPWD_WITH_AES_128_CCM_SHA256 = (TBD, TBD );
CipherSuite TLS_ECCPWD_WITH_AES_256_CCM_SHA384 = (TBD, TBD );
```

Implementations conforming to this specification MUST support the TLS_ECCPWD_WITH_AES_128_CBC_SHA ciphersuite; they SHOULD support TLS_ECCPWD_WITH_AES_128_CCM_SHA, TLS_FFCPWD_WITH_AES_128_CCM_SHA,

TLS_ECCPWD_WITH_AES_128_GCM_SHA256,
TLS_ECCPWD_WITH_AES_256_GCM_SHA384; and MAY support the remaining
ciphersuites.

When negotiated with a version of TLS prior to 1.2, the Pseudo-Random Function (PRF) from that version is used; otherwise, the PRF is the TLS PRF [RFC5246] using the hash function indicated by the ciphersuite. Regardless of the TLS version, the TLS-PWD random function, H, is always instantiated with the hash algorithm indicated by the ciphersuite.

For those ciphersuites that use Cipher Block Chaining (CBC) [SP800-38A] mode, the MAC is HMAC [RFC2104] with the hash function indicated by the ciphersuite.

6. Acknowledgements

The authenticated key exchange defined here has also been defined for use in 802.11 networks, as an EAP method, and as an authentication method for IKE. Each of these specifications has elicited very helpful comments from a wide collection of people that have allowed the definition of the authenticated key exchange to be refined and improved.

The authors would like to thank Scott Fluhrer for discovering the "password as exponent" attack that was possible in an early version of this key exchange and for his very helpful suggestions on the techniques for fixing the PE to prevent it. The authors would also like to thank Hideyuki Suzuki for his insight in discovering an attack against a previous version of the underlying key exchange protocol. Special thanks to Lily Chen for helpful discussions on hashing into an elliptic curve. Rich Davis suggested the defensive checks that are part of the processing of the ServerKeyExchange and ClientKeyExchange messages, and his various comments have greatly improved the quality of this memo and the underlying key exchange on which it is based.

Martin Rex, Peter Gutmann, Marsh Ray, and Rene Struik, discussed the possibility of a side-channel attack against the hunting-and-pecking loop on the TLS mailing list. That discussion prompted the addition of the security parameter, k, to the hunting-and-pecking loop.

7. IANA Considerations

IANA SHALL assign a value for a new TLS extension type from the TLS ExtensionType Registry defined in [RFC5246] with the name "pwd". The

RFC editor SHALL replace TBD in Section 4.2.1 with the IANA-assigned value for this extension.

IANA SHALL assign nine new ciphersuites from the TLS Ciphersuite Registry defined in [RFC5246] for the following ciphersuites:

```
CipherSuite TLS_FFCPWD_WITH_3DES_EDE_CBC_SHA = ( TBD, TBD );  
  
CipherSuite TLS_FFCPWD_WITH_AES_128_CBC_SHA = (TBD, TBD );  
  
CipherSuite TLS_ECCPWD_WITH_AES_128_CBC_SHA = (TBD, TBD );  
  
CipherSuite TLS_ECCPWD_WITH_AES_128_GCM_SHA256 = (TBD, TBD );  
  
CipherSuite TLS_ECCPWD_WITH_AES_256_GCM_SHA384 = (TBD, TBD );  
  
CipherSuite TLS_FFCPWD_WITH_AES_128_CCM_SHA = (TBD, TBD );  
  
CipherSuite TLS_ECCPWD_WITH_AES_128_CCM_SHA = (TBD, TBD );  
  
CipherSuite TLS_ECCPWD_WITH_AES_128_CCM_SHA256 = (TBD, TBD );  
  
CipherSuite TLS_ECCPWD_WITH_AES_256_CCM_SHA384 = (TBD, TBD );
```

The RFC editor SHALL replace (TBD, TBD) in all the ciphersuites defined in Section 5 with the appropriate IANA-assigned values. The "DTLS-OK" column in the ciphersuite registry SHALL be set to "Y" for all ciphersuites defined in this memo.

8. Security Considerations

A passive attacker against this protocol will see the ServerKeyExchange and the ClientKeyExchange containing the server's scalar and Element, and the client's scalar and Element, respectively. The client and server effectively hide their secret private value by masking it modulo the order of the selected group. If the order is "q", then there are approximately "q" distinct pairs of numbers that will sum to the scalar values observed. It is possible for an attacker to iterate through all such values but for a large value of "q", this exhaustive search technique is computationally infeasible. The attacker would have a better chance in solving the discrete logarithm problem, which we have already assumed (see Section 3.5) to be an intractable problem.

A passive attacker can take the Element from either the ServerKeyExchange or the ClientKeyExchange and try to determine the random "mask" value used in its construction and then recover the

other party's "private" value from the scalar in the same message. But this requires the attacker to solve the discrete logarithm problem which we assumed was intractable.

Both the client and the server obtain a shared secret, the premaster secret, based on a secret group element and the private information they contributed to the exchange. The secret group element is based on the password. If they do not share the same password they will be unable to derive the same secret group element and if they don't generate the same secret group element they will be unable to generate the same premaster secret. Seeing a finished message along with the ServerKeyExchange and ClientKeyExchange will not provide any additional advantage of attack since it is generated with the unknowable premaster secret.

An active attacker impersonating the client can induce a server to send a ServerKeyExchange containing the server's scalar and Element. It can attempt to generate a ClientKeyExchange and send to the server but the attacker is required to send a finished message first so the only information she can obtain in this attack is less than the information she can obtain from a passive attack, so this particular active attack is not very fruitful.

An active attacker can impersonate the server and send a forged ServerKeyExchange after receiving the ClientHello. The attacker then waits until it receives the ClientKeyExchange and finished message from the client. Now the attacker can attempt to run through all possible values of the password, computing PE (see Section 4.1), computing candidate premaster secrets (see Section 4.3), and attempting to recreate the client's finished message.

But the attacker committed to a single guess of the password with her forged ServerKeyExchange. That value was used by the client in her computation of the premaster secret which was used to produce the finished message. Any guess of the password which differs from the one used in the forged ServerKeyExchange would result in each side using a different PE in the computation of the premaster secret and therefore the finished message cannot be verified as correct, even if a subsequent guess, while running through all possible values, was correct. The attacker gets one guess, and one guess only, per active attack.

Instead of attempting to guess at the password, an attacker can attempt to determine PE and then launch an attack. But PE is determined by the output of the random function, H, which is indistinguishable from a random source since H is assumed to be a "random oracle" (Section 3.5). Therefore, each element of the finite cyclic group will have an equal probability of being the PE. The

probability of guessing PE will be $1/q$, where q is the order of the group. For a large value of " q " this will be computationally infeasible.

The implications of resistance to dictionary attack are significant. An implementation can provision a password in a practical and realistic manner-- i.e. it MAY be a character string and it MAY be relatively short-- and still maintain security. The nature of the pool of potential passwords determines the size of the pool, D , and countermeasures can prevent an attacker from determining the password in the only possible way: repeated, active, guessing attacks. For example, a simple four character string using lower-case English characters, and assuming random selection of those characters, will result in D of over four hundred thousand. An attacker would need to mount over one hundred thousand active, guessing attacks (which will easily be detected) before gaining any significant advantage in determining the pre-shared key.

Countermeasures to deal with successive active, guessing attacks are only possible by noticing a certain username is failing repeatedly over a certain period of time. Attacks which attempt to find a password for a random user are more difficult to detect. For instance, if a device uses a serial number as a username and the pool of potential passwords is sufficiently small, a more effective attack would be to select a password and try all potential "users" to disperse the attack and confound countermeasures. It is therefore RECOMMENDED that implementations of TLS-pwd keep track of the total number of failed authentications regardless of username in an effort to detect and thwart this type of attack.

The benefits of resistance to dictionary attack can be lessened by a client using the same passwords with multiple servers. An attacker could re-direct a session from one server to the other if the attacker knew that the intended server stored the same password for the client as another server.

An adversary that has access to, and a considerable amount of control over, a client or server could attempt to mount a side-channel attack to determine the number of times it took for a certain password (plus client random and server random) to select a password element. Each such attack could result in a successive paring-down of the size of the pool of potential passwords, resulting in a manageably small set from which to launch a series of active attacks to determine the password. A security parameter, k , is used to normalize the amount of work necessary to determine the password element (see Section 4.1). The probability that a password will require more than k iterations is roughly $(q/2p)^k$ for ECC groups and $(q/p)^k$ for FFC groups, so it is possible to mitigate side channel attack at the

expense of a constant cost per connection attempt. But if a particular password requires more than k iterations it will leak k bits of information to the side-channel attacker, which for some dictionaries will uniquely identify the password. Therefore, the security parameter, k , needs to be set with great care. It is RECOMMENDED that an implementation set the security parameter, k , to a value of at least forty (40) which will put the probability that more than forty iterations are needed in the order of one in one trillion (1:1,000,000,000,000).

9. Implementation Considerations

The selection of the ciphersuite and selection of the particular finite cyclic group to use with the ciphersuite are divorced in this memo but they remain intimately close.

It is RECOMMENDED that implementations take note of the strength estimates of particular groups and to select a ciphersuite providing commensurate security with its hash and encryption algorithms. A ciphersuite whose encryption algorithm has a keylength less than the strength estimate, or whose hash algorithm has a blocksize that is less than twice the strength estimate SHOULD NOT be used.

For example, the elliptic curve named secp256r1 (whose IANA-assigned number is 23) provides an estimated 128 bits of strength and would be compatible with an encryption algorithm supporting a key of that length, and a hash algorithm that has at least a 256-bit blocksize. Therefore, a suitable ciphersuite to use with secp256r1 could be TLS_ECCPWD_WITH_AES_128_GCM_SHA256.

Resistance to dictionary attack means that the attacker must launch an active attack to make a single guess at the password. If the size of the pool from which the password was extracted was D , and each password in the pool has an equal probability of being chosen, then the probability of success after a single guess is $1/D$. After X guesses, and removal of failed guesses from the pool of possible passwords, the probability becomes $1/(D-X)$. As X grows so does the probability of success. Therefore it is possible for an attacker to determine the password through repeated brute-force, active, guessing attacks. Implementations SHOULD take note of this fact and choose an appropriate pool of potential passwords-- i.e. make D big. Implementations SHOULD also take countermeasures, for instance refusing authentication attempts by a particular username for a certain amount of time, after the number of failed authentication attempts reaches a certain threshold. No such threshold or amount of time is recommended in this memo.

10. References

10.1. Normative References

- [RFC2104] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, February 1997.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.
- [RFC3454] Hoffman, P. and M. Blanchet, "Preparation of Internationalized Strings ("stringprep")", RFC 3454, December 2002.
- [RFC4013] Zeilenga, K., "SASLprep: Stringprep Profile for User Names and Passwords", RFC 4013, February 2005.
- [RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008.
- [SP800-38A] National Institute of Standards and Technology, "Recommendation for Block Cipher Modes of Operation--Methods and Techniques", NIST Special Publication 800-38A, December 2001.

10.2. Informative References

- [FIPS186-3] National Institute of Standards and Technology, "Digital Signature Standard (DSS)", Federal Information Processing Standards Publication 186-3.
- [RANDOR] Bellare, M. and P. Rogaway, "Random Oracles are Practical: A Paradigm for Designing Efficient Protocols", Proceedings of the 1st ACM Conference on Computer and Communication Security, ACM Press, 1993.
- [RFC4086] Eastlake, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security", BCP 106, RFC 4086, June 2005.
- [RFC4492] Blake-Wilson, S., Bolyard, N., Gupta, V., Hawk, C., and B. Moeller, "Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS)", RFC 4492, May 2006.
- [RFC5751] Ramsdell, B. and S. Turner, "Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 3.2 Message

Specification", RFC 5751, January 2010.

[RFC5967] Turner, S., "The application/pkcs10 Media Type", RFC 5967, August 2010.

[RFC6090] McGrew, D., Igoe, K., and M. Salter, "Fundamental Elliptic Curve Cryptography Algorithms", RFC 6090, February 2011.

[SP800-56A]
Barker, E., Johnson, D., and M. Smid, "Recommendations for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography", NIST Special Publication 800-56A, March 2007.

Authors' Addresses

Dan Harkins (editor)
Aruba Networks
1322 Crossman Avenue
Sunnyvale, CA 94089-1113
United States of America

Email: dharkins@arubanetworks.com

Dave Halasz (editor)
Halasz Ventures
8401 Chagrin Road, Suite 10A
Chagrin Falls, OH 44023
United States of America

Email: david.e.halasz@gmail.com

Network Working Group
Internet-Draft
Intended status: Informational
Expires: October 11, 2013

S. Josefsson
SJD AB
J. Strombergson
Secworks Sweden AB
N. Mavrogiannopoulos
KU Leuven
April 9, 2013

The Salsa20 Stream Cipher for Transport Layer Security
draft-josefsson-salsa20-tls-02

Abstract

This document describe how the Salsa20 stream cipher can be used in the Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) protocols.

Status of this Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <http://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on October 11, 2013.

Copyright Notice

Copyright (c) 2013 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as

described in the Simplified BSD License.

Table of Contents

1. Introduction	3
2. Salsa20 Cipher Suites	4
2.1. Salsa20 Cipher Suites with HMAC-SHA1	4
2.2. Salsa20 Cipher Suites with UMAC-96	5
3. The TLS GenericStreamCipher	6
4. Acknowledgements	7
5. IANA Considerations	8
6. Security Considerations	9
7. References	10
7.1. Normative References	10
7.2. Informative References	10
Authors' Addresses	12

1. Introduction

This document describe how the Salsa20 stream cipher can be used in the Transport Layer Security (TLS) version 1.0 [RFC2246], TLS version 1.1 [RFC4346], and TLS version 1.2 [RFC5246] protocols, as well as in the Datagram Transport Layer Security (DTLS) versions 1.0 [RFC4347] and 1.2 [RFC6347]. It can also be used with Secure Sockets Layer (SSL) version 3.0 [RFC6101].

Salsa20 [SALSA20SPEC] is a stream cipher that has been designed for high performance in software implementations. The cipher has compact implementation and uses few resources and inexpensive operations that makes it suitable for implementation on a wide range of architectures. It has been designed to prevent leakage of information through side channel analysis, has a simple initialization sequence and provides good key agility and performance. Salsa20 is one of the ciphers selected as part of the eSTREAM portfolio of stream ciphers [ESTREAM].

Recent attacks [CBC-ATTACK] have indicated problems with CBC-mode cipher suites in TLS and DTLS as well as issues with the only supported stream cipher (RC4) [RC4-ATTACK]. While the existing AEAD ciphersuites address these issues, concerns about their performance, on general purpose CPUs, are sometimes raised [AEAD-PERFORMANCE].

Moreover, the RC4 cipher cannot be used in DTLS because it does not provide random access in the key stream. That allowed no choice of a fast stream cipher in the context of DTLS.

The purpose of this document is to provide an alternative stream cipher for both TLS and DTLS that is comparable to RC4 in speed on a wide range of platforms.

2. Salsa20 Cipher Suites

The following variants of Salsa20 are specified. The variants provide a range of performance and security that can be selected as appropriate.

ESTREAM_SALSA20: Salsa20 with 12 rounds and a 256 bit key. This cipher is the high performant eSTREAM Salsa20 with 256 bit key.

SALSA20: Salsa20 with 20 rounds and a 256 bit key. This is the original (conservative with respect to security) variant of Salsa20.

In the next sections different ciphersuites are defined that utilize the Salsa20 cipher combined with various MAC methods

In all cases, the pseudorandom function (PRF) for TLS 1.2 is the TLS PRF with SHA-256 as the hash function. When used with TLS versions prior to 1.2, the PRF is calculated as specified in the appropriate version of the TLS specification.

2.1. Salsa20 Cipher Suites with HMAC-SHA1

The following CipherSuites are defined:

TLS_RSA_WITH_ESTREAM_SALSA20_SHA1	= {0xTBD, 0xTBD}
TLS_RSA_WITH_SALSA20_SHA1	= {0xTBD, 0xTBD}
TLS_DHE_RSA_WITH_ESTREAM_SALSA20_SHA1	= {0xTBD, 0xTBD}
TLS_DHE_RSA_WITH_SALSA20_SHA1	= {0xTBD, 0xTBD}
TLS_ECDHE_RSA_WITH_ESTREAM_SALSA20_SHA1	= {0xTBD, 0xTBD}
TLS_ECDHE_RSA_WITH_SALSA20_SHA1	= {0xTBD, 0xTBD}
TLS_ECDHE_ECDSA_WITH_ESTREAM_SALSA20_SHA1	= {0xTBD, 0xTBD}
TLS_ECDHE_ECDSA_WITH_SALSA20_SHA1	= {0xTBD, 0xTBD}
TLS_PSK_WITH_ESTREAM_SALSA20_SHA1	= {0xTBD, 0xTBD}
TLS_PSK_WITH_SALSA20_SHA1	= {0xTBD, 0xTBD}
TLS_DHE_PSK_WITH_ESTREAM_SALSA20_SHA1	= {0xTBD, 0xTBD}
TLS_DHE_PSK_WITH_SALSA20_SHA1	= {0xTBD, 0xTBD}
TLS_RSA_PSK_WITH_ESTREAM_SALSA20_SHA1	= {0xTBD, 0xTBD}
TLS_RSA_PSK_WITH_SALSA20_SHA1	= {0xTBD, 0xTBD}
TLS_ECDHE_PSK_WITH_ESTREAM_SALSA20_SHA1	= {0xTBD, 0xTBD}
TLS_ECDHE_PSK_WITH_SALSA20_SHA1	= {0xTBD, 0xTBD}

Note that Salsa20 requires a 64-bit nonce. That nonce is updated on the encryption of every TLS record, and is set to be the 64-bit TLS record sequence number. In case of DTLS the 64-bit nonce is formed as the concatenation of the 16-bit epoch with the 48-bit sequence number.

The RSA, DHE_RSA, ECDHE_RSA, ECDHE_ECDSA, PSK, DHE_PSK, RSA_PSK, ECDHE_PSK key exchanges are performed as defined in [RFC5246], [RFC4492], and [RFC5489].

The MAC algorithm used in the ciphersuites above is HMAC-SHA1 [RFC6234].

2.2. Salsa20 Cipher Suites with UMAC-96

The following CipherSuites utilize Salsa20 in combination with UMAC-96 [RFC4418], a very fast MAC algorithm based on Universal Hashing.

TLS_RSA_WITH_ESTREAM_SALSA20_UMAC96	= {0xTBD, 0xTBD}
TLS_RSA_WITH_SALSA20_UMAC96	= {0xTBD, 0xTBD}
TLS_DHE_RSA_WITH_ESTREAM_SALSA20_UMAC96	= {0xTBD, 0xTBD}
TLS_DHE_RSA_WITH_SALSA20_UMAC96	= {0xTBD, 0xTBD}
TLS_ECDHE_RSA_WITH_ESTREAM_SALSA20_UMAC96	= {0xTBD, 0xTBD}
TLS_ECDHE_RSA_WITH_SALSA20_UMAC96	= {0xTBD, 0xTBD}
TLS_ECDHE_ECDSA_WITH_ESTREAM_SALSA20_UMAC96	= {0xTBD, 0xTBD}
TLS_ECDHE_ECDSA_WITH_SALSA20_UMAC96	= {0xTBD, 0xTBD}
TLS_PSK_WITH_ESTREAM_SALSA20_UMAC96	= {0xTBD, 0xTBD}
TLS_PSK_WITH_SALSA20_UMAC96	= {0xTBD, 0xTBD}
TLS_DHE_PSK_WITH_ESTREAM_SALSA20_UMAC96	= {0xTBD, 0xTBD}
TLS_DHE_PSK_WITH_SALSA20_UMAC96	= {0xTBD, 0xTBD}
TLS_RSA_PSK_WITH_ESTREAM_SALSA20_UMAC96	= {0xTBD, 0xTBD}
TLS_RSA_PSK_WITH_SALSA20_UMAC96	= {0xTBD, 0xTBD}
TLS_ECDHE_PSK_WITH_ESTREAM_SALSA20_UMAC96	= {0xTBD, 0xTBD}
TLS_ECDHE_PSK_WITH_SALSA20_UMAC96	= {0xTBD, 0xTBD}

Note that both Salsa20 and UMAC-96 are used with a 64-bit nonce. That nonce is set to be the 64-bit TLS record sequence number. In case of DTLS the 64-bit nonce is formed as the concatenation of the 16-bit epoch with the 48-bit sequence number.

The RSA, DHE_RSA, ECDHE_RSA, ECDHE_ECDSA, PSK, DHE_PSK, RSA_PSK, ECDHE_PSK key exchanges are performed as defined in [RFC5246], [RFC4492], and [RFC5489].

3. The TLS GenericStreamCipher

The ciphersuites defined in this document differ from the TLS RC4 ciphersuites that have been the basis for the definition of GenericStreamCipher. Unlike RC4, Salsa20 requires a nonce per record. This however, does not affect the description of the GenericStreamCipher if one assumes that a nonce is optional and depends on the cipher's characteristics (in that case RC4 uses a 0 byte nonce, and Salsa20 an 8-byte nonce).

Moreover, in order to accommodate MAC algorithms like UMAC that require a nonce as part of their operation, the document extends the MAC algorithm as specified in the TLS protocol. The extended MAC includes a nonce as a second parameter. MAC algorithms that do not require a nonce, such as HMAC, are assumed to ignore the nonce input value. The MAC in a GenericStreamCipher is then calculated as follows.

```
MAC(MAC_write_key, nonce,
    seq_num +
    TLSCompressed.type +
    TLSCompressed.version +
    TLSCompressed.length +
    TLSCompressed.fragment);
```

where "+" denotes concatenation.

nonce The nonce for this record. If the size of the nonce accepted by the MAC is 64-bits then nonce equals the sequence number (or the concatenation of the 16-bit epoch with the 48-bit sequence number in DTLS). Otherwise the MAC algorithm must specify how the nonce is formed.

seq_num The sequence number for this record.

MAC The MAC algorithm specified by SecurityParameters.mac_algorithm.

As specified in TLS [RFC5246] the MAC is computed before encryption and the stream cipher encrypts the entire block, including the MAC.

4. Acknowledgements

The authors would like to thank D. J. Bernstein, David McGrew, Wan-Teh Chang, and Adam Langley for discussion and suggestions.

5. IANA Considerations

IANA is requested to allocate the following numbers in the TLS Cipher Suite Registry:

TLS_RSA_WITH_ESTREAM_SALSA20_SHA1	= {0xTBD, 0xTBD}
TLS_RSA_WITH_SALSA20_SHA1	= {0xTBD, 0xTBD}
TLS_DHE_RSA_WITH_ESTREAM_SALSA20_SHA1	= {0xTBD, 0xTBD}
TLS_DHE_RSA_WITH_SALSA20_SHA1	= {0xTBD, 0xTBD}
TLS_ECDHE_RSA_WITH_ESTREAM_SALSA20_SHA1	= {0xTBD, 0xTBD}
TLS_ECDHE_RSA_WITH_SALSA20_SHA1	= {0xTBD, 0xTBD}
TLS_ECDHE_ECDSA_WITH_ESTREAM_SALSA20_SHA1	= {0xTBD, 0xTBD}
TLS_ECDHE_ECDSA_WITH_SALSA20_SHA1	= {0xTBD, 0xTBD}
TLS_PSK_WITH_ESTREAM_SALSA20_SHA1	= {0xTBD, 0xTBD}
TLS_PSK_WITH_SALSA20_SHA1	= {0xTBD, 0xTBD}
TLS_DHE_PSK_WITH_ESTREAM_SALSA20_SHA1	= {0xTBD, 0xTBD}
TLS_DHE_PSK_WITH_SALSA20_SHA1	= {0xTBD, 0xTBD}
TLS_RSA_PSK_WITH_ESTREAM_SALSA20_SHA1	= {0xTBD, 0xTBD}
TLS_RSA_PSK_WITH_SALSA20_SHA1	= {0xTBD, 0xTBD}
TLS_ECDHE_PSK_WITH_ESTREAM_SALSA20_SHA1	= {0xTBD, 0xTBD}
TLS_ECDHE_PSK_WITH_SALSA20_SHA1	= {0xTBD, 0xTBD}
TLS_RSA_WITH_ESTREAM_SALSA20_UMAC96	= {0xTBD, 0xTBD}
TLS_RSA_WITH_SALSA20_UMAC96	= {0xTBD, 0xTBD}
TLS_DHE_RSA_WITH_ESTREAM_SALSA20_UMAC96	= {0xTBD, 0xTBD}
TLS_DHE_RSA_WITH_SALSA20_UMAC96	= {0xTBD, 0xTBD}
TLS_ECDHE_RSA_WITH_ESTREAM_SALSA20_UMAC96	= {0xTBD, 0xTBD}
TLS_ECDHE_RSA_WITH_SALSA20_UMAC96	= {0xTBD, 0xTBD}
TLS_ECDHE_ECDSA_WITH_ESTREAM_SALSA20_UMAC96	= {0xTBD, 0xTBD}
TLS_ECDHE_ECDSA_WITH_SALSA20_UMAC96	= {0xTBD, 0xTBD}
TLS_PSK_WITH_ESTREAM_SALSA20_UMAC96	= {0xTBD, 0xTBD}
TLS_PSK_WITH_SALSA20_UMAC96	= {0xTBD, 0xTBD}
TLS_DHE_PSK_WITH_ESTREAM_SALSA20_UMAC96	= {0xTBD, 0xTBD}
TLS_DHE_PSK_WITH_SALSA20_UMAC96	= {0xTBD, 0xTBD}
TLS_RSA_PSK_WITH_ESTREAM_SALSA20_UMAC96	= {0xTBD, 0xTBD}
TLS_RSA_PSK_WITH_SALSA20_UMAC96	= {0xTBD, 0xTBD}
TLS_ECDHE_PSK_WITH_ESTREAM_SALSA20_UMAC96	= {0xTBD, 0xTBD}
TLS_ECDHE_PSK_WITH_SALSA20_UMAC96	= {0xTBD, 0xTBD}

6. Security Considerations

The security of Salsa20 is discussed in the Salsa20 security [SALSA20-SECURITY] paper. At the time of writing this document, there are no known significant security problems with the eSTREAM variant of Salsa20, nor with the original 20 round variant. As of early 2013, the best cryptanalysis breaks 8 out of 20 rounds to recover the 256-bit secret key in 2^{251} operations, using 2^{31} keystream pairs (see [SALSA20-ATTACK]). For more background, see the eSTREAM report [ESTREAM].

There are no ciphersuites defined in this document that utilize the variant of Salsa20 with 128-bit key material, because (due to the design of Salsa20) they provide no performance advantage over the 256-bit variant.

The ciphersuites that utilize UMAC-96 use a short MAC (96-bits), to be consistent with the MAC size used in the TLS Finished messages, which is also 96-bits, and thus allow room for more data in TLS records. The security considerations of [RFC4418] also apply.

This document should not introduce any other security considerations than those that directly follow from any use of the stream cipher Salsa20 and those that directly follow from introducing any set of stream cipher suites into TLS and DTLS.

7. References

7.1. Normative References

- [RFC2246] Dierks, T. and C. Allen, "The TLS Protocol Version 1.0", RFC 2246, January 1999.
- [RFC4346] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.1", RFC 4346, April 2006.
- [RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008.
- [RFC4347] Rescorla, E. and N. Modadugu, "Datagram Transport Layer Security", RFC 4347, April 2006.
- [RFC4418] Krovetz, T., "UMAC: Message Authentication Code using Universal Hashing", RFC 4418, March 2006.
- [RFC4492] Blake-Wilson, S., Bolyard, N., Gupta, V., Hawk, C., and B. Moeller, "Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS)", RFC 4492, May 2006.
- [RFC5489] Badra, M. and I. Hajjeh, "ECDHE_PSK Cipher Suites for Transport Layer Security (TLS)", RFC 5489, March 2009.
- [RFC6347] Rescorla, E. and N. Modadugu, "Datagram Transport Layer Security Version 1.2", RFC 6347, January 2012.
- [RFC6234] Eastlake, D. and T. Hansen, "US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)", RFC 6234, May 2011.
- [SALSA20SPEC] Bernstein, D., "Salsa20 specification", WWW <http://cr.yp.to/snuffle/spec.pdf>, April 2005.

7.2. Informative References

- [RFC6101] Freier, A., Karlton, P., and P. Kocher, "The Secure Sockets Layer (SSL) Protocol Version 3.0", RFC 6101, August 2011.
- [SALSA20-SECURITY] Bernstein, D., "Salsa20 security", WWW <http://cr.yp.to/snuffle/security.pdf>, April 2005.
- [ESTREAM] Babbage, S., DeCanniere, C., Cantenaut, A., Cid, C., Gilbert, H., Johansson, T., Parker, M., Preneel, B.,

Rijmen, V., and M. Robshaw, "The eSTREAM Portfolio (rev. 1)", WWW <http://www.ecrypt.eu.org/stream/finallist.html>, September 2008.

[CBC-ATTACK]

AlFardan, N. and K. Paterson, "Lucky Thirteen: Breaking the TLS and DTLS Record Protocols", IEEE Symposium on Security and Privacy , 2013.

[RC4-ATTACK]

ISOBE, T., OHIGASHI, T., WATANABE, Y., and M. MORII, "Full Plaintext Recovery Attack on Broadcast RC4", International Workshop on Fast Software Encryption , 2013.

[AEAD-PERFORMANCE]

Krovetz, T. and P. Rogaway, "The Software Performance of Authenticated-Encryption Modes", International Workshop on Fast Software Encryption , 2011.

[SALSA20-ATTACK]

Aumasson, J-P., Fischer, S., Khazaei, S., Meier, W., and C. Rechberger, "New Features of Latin Dances: Analysis of Salsa, ChaCha, and Rumba", WWW <http://eprint.iacr.org/2007/472.pdf>, 2007.

Authors' Addresses

Simon Josefsson
SJD AB

Email: simon@josefsson.org
URI: <http://josefsson.org/>

Joachim Strombergson
Secworks Sweden AB

Email: joachim@secworks.se
URI: <http://secworks.se/>

Nikos Mavrogiannopoulos
KU Leuven

Email: nikos.mavrogiannopoulos@esat.kuleuven.be

Network Working Group
Internet-Draft
Expires: January 30, 2014

A. Pironti
INRIA Paris-Rocquencourt
N. Mavrogiannopoulos
KU Leuven
July 29, 2013

Length Hiding Padding for the Transport Layer Security Protocol
draft-pironti-tls-length-hiding-01

Abstract

This memo proposes length hiding methods of operation for the TLS protocol. It defines a TLS extension to allow arbitrary amount of padding in any TLS ciphersuite, and it presents guidelines and a reference implementation of record fragmentation and padding so that the length of the exchanged messages is effectively concealed within a given range of lengths. The latter guidelines also apply to the standard TLS padding allowed by the TLS block ciphers.

Status of this Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <http://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on January 30, 2014.

Copyright Notice

Copyright (c) 2013 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must

include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	3
2. Terminology	4
3. TLS Extension: Extended Record Padding	5
3.1. Extension Negotiation	5
3.2. Record Payload	5
4. A Length Hiding Mechanism for TLS	8
4.1. Range Splitting	8
4.1.1. Fragmenting Plaintext into Records	10
4.1.2. Adding the Length Hiding Padding	11
4.1.3. A Length Hiding API	11
4.2. Applicability	12
5. Security Considerations	13
5.1. Length Hiding with extended record padding	13
5.2. Length Hiding with standard TLS block ciphers	13
5.3. Mitigating Denial of Service	14
6. IANA Considerations	15
7. Normative References	16
Appendix A. Acknowledgements	17
Authors' Addresses	18

1. Introduction

When using CBC block ciphers, the TLS protocol [RFC5246] provides means to frustrate attacks based on analysis of the length of exchanged messages, by adding extra pad to TLS records. However, the TLS specification does not define a length hiding (LH) method for applications that require it. In fact, current implementations of eager fragmentation strategies or random padding strategies have been showed to be ineffective against this kind of traffic analysis [LH-PADDING].

By design, in the standard TLS block cipher mode, only a limited amount of extra padding can be carried with each record fragment, and this can potentially require extra fragmentation to carry all required padding. Moreover, no LH can be implemented for stream ciphers. To overcome these limitations, the TLS extension proposed in this document enables efficient LH both for block and stream ciphers.

In addition, it presents guidelines and a reference implementation of record fragmentation and padding so that the length of the exchanged messages is effectively concealed within a range of lengths provided by the user of the TLS record protocol.

The proposed extension also eliminates padding oracles (both in errors and timing) that have been plaguing standard TLS block ciphers [CBCTIME] [DTLS-ATTACK].

The goals of LH for TLS are the following:

1. Length-Hiding: use message fragmentation and the allowed extra padding for block ciphers to conceal the real length of the exchanged message within a range of lengths chosen by the user of the TLS record protocol. All messages sent with the same range use the same network bandwidth, regardless of the real size of the message itself.
2. Efficiency: the minimum required amount of extra padding is used, and the minimum number of required fragments is sent.

To maximize interoperability, this document also includes guidelines to implement LH by using the limited amount of padding provided by existing block ciphers. This variant of LH is backward compatible, in that an implementation sending length-hidden messages correctly interoperates with non LH-aware implementations of TLS, but leads to a less efficient LH implementation.

2. Terminology

This document uses the same notation and terminology used in the TLS Protocol specification [RFC5246].

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

3. TLS Extension: Extended Record Padding

The TLS extended record padding is a variant of the TLS record protocol where every record can be padded up to 2^{14} bytes, regardless of the cipher being used.

3.1. Extension Negotiation

In order to indicate the support of the extended record padding, clients MUST include an extension of type "extended_record_padding" to the extended client hello message. The "extended_record_padding" TLS extension is assigned the value of TDB-BY-IANA from the TLS ExtensionType registry. This value is used as the extension number for the extensions in both the client hello message and the server hello message. The hello extension mechanism is described in [RFC5246].

This extension carries no payload and indicates support for the extended record padding. The "extension_data" field of this extension are of zero length in both the client and the server.

The negotiated record padding applies for the duration of the session, including session resumption. A client wishing to resume a session where the extended record padding was negotiated SHOULD include the "extended_record_padding" extension in the client hello.

3.2. Record Payload

The translation of the TLSCompressed structure into TLSCiphertext remains the same as in [RFC5246]. When the cipher is BulkCipherAlgorithm.null, the 'fragment' structure of TLSCiphertext also remains unchanged. That is, for the TLS_NULL_WITH_NULL_NULL ciphersuite and for MAC-only ciphersuites this extension has no effect. For all other ciphersuites, the 'fragment' structure of TLSCiphertext is modified as follows.


```
stream-ciphered struct {
    opaque pad<0..2^14>;
    opaque content[TLSCompressed.length];
    opaque MAC[SecurityParameters.mac_length];
} GenericStreamCipher;

struct {
    opaque IV[SecurityParameters.record_iv_length];
    block-ciphered ciphered struct {
        opaque pad<0..2^14>;
        opaque content[TLSCompressed.length];
        opaque MAC[CipherSpec.hash_size];
    };
} GenericBlockCipher;

struct {
    opaque nonce_explicit[SecurityParameters.record_iv_length];
    aead-ciphered struct {
        opaque pad<0..2^14>;
        opaque content[TLSCompressed.length];
    };
} GenericAEADCipher;
```

The padding can be filled with arbitrary data, and it is authenticated as part of the MAC. For block ciphers, the length of the pad MUST be such that the total length (i.e., the pad, the content and the MAC) are a multiple of the block size.

For the various ciphers the data are authenticated as follows.

Standard Stream Ciphers:

```
MAC(MAC_write_key, seq_num +
    TLSCompressed.type +
    TLSCompressed.version +
    TLSCompressed.length +
    TLSCiphertext.fragment.GenericStreamCipher.pad +
    TLSCompressed.fragment);
```

Block Ciphers:

```
MAC(MAC_write_key, seq_num +
    TLSCompressed.type +
    TLSCompressed.version +
    TLSCompressed.length +
    TLSCiphertext.fragment.GenericBlockCipher.pad +
    TLSCompressed.fragment);
```

AEAD Ciphers:

```
AEADEncrypted = AEAD-Encrypt(write_key, nonce,  
                               pad + plaintext,  
                               additional_data);
```

Implementation note: With block and stream ciphers, in order to avoid padding oracles, decryption, MAC verification and payload decoding MUST be executed in the following order.

1. Decrypt TLSCiphertext.fragment.
2. Verify the MAC.
3. Split plaintext from pad.

4. A Length Hiding Mechanism for TLS

In order to send length-hidden messages, a user of a LH-TLS implementation provides the plaintext to be sent together with a range (low,high), meaning that an attacker can at most learn that the real plaintext length is between low and high.

The LH mechanism described in the rest of this document applies both to standard TLS block ciphers and the extended record padding of Section 3.

4.1. Range Splitting

Not all user-provided ranges can be conveyed in a single TLS record fragment. A LH-TLS implementation uses a fragmentation algorithm that takes a message with a desired length range (low,high) and breaks it up into n suitably sized ranges each of which can be conveyed in a single TLS record fragment. The Range and FragmentRange are defined as follows.

```
struct {  
    uint32 low;  
    uint32 high;  
} Range;  
  
struct {  
    uint16 low;  
    uint16 high;  
} FragmentRange;
```

If the difference between Range.high and Range.low is greater than the maximum allowed padding size for a single fragment, or if their value is greater than the maximum fragment size, the given range must be split into multiple smaller FragmentRange structures each of which can be conveyed into a single TLS record.

Range.low MUST be less or equal to Range.high. Declaring Range.low and Range.high as uint32 allows to send messages of length up to 2^{32} : TLS implementations MAY use larger data types for these fields. A FragmentRange, that can be conveyed in one record, MUST have both values of FragmentRange.low and FragmentRange.high not exceeding 2^{14} (or the negotiated maximum value of TLSPlaintext.length [RFC6066]).

A TLS implementation applies the range splitting algorithm starting from the user-provided Range structure, resulting into a sequence of FragmentRange structures. For each FragmentRange structure, it transmits a TLS record adhering into the limits of the corresponding FragmentRange. When a block cipher is in use, on each record the

implementation computes n bytes of minimal padding (the minimum amount of padding required to get block alignment) pretending the length of the plaintext is `FragmentRange.high`. The total padding added to the current fragment is finally n plus the difference between `Range.high` and the real plaintext length.

This document does not mandate any specific algorithm to split a `Range` into multiple `FragmentRange` ranges. The only constraint is that the sum of the obtained sequence of ranges equals the range given as input. Implementations may use non-deterministic splitting algorithms to change the shape of the traffic each time messages with the same range are exchanged.

A reference range splitting algorithm is provided in the following.

```
// The maximum allowed TLSPlaintext.length
uint16 FS = 2^14;
// Maximum padding size:
// p = 255 for standard TLS block ciphers;
// p = 2^14 for extended record padding
uint16 PS = p;
// Length of the padlen:
// pl = 1 for standard TLS block ciphers;
// pl = 2 for extended record padding
uint8 PL = pl;
// Note: Block size is 0 for stream ciphers
uint8 BS = SecurityParameters.block_length;
// MAC size
uint8 MS = SecurityParameters.mac_length;

/* Returns the maximum pad that can be added for a fragment,
 * given that at least 'len' bytes of plaintext will be
 * transferred.
 */
uint16 max_lh_pad(uint16 len)
{
    uint16 this_pad = min(PS, FS-len);
    if (BS == 0) {
        return this_pad;
    } else {
        uint8 overflow = (len + this_pad + MS + PL) % BS;
        if (overflow > this_pad) {
            return this_pad;
        } else {
            return this_pad - overflow;
        }
    }
}
```

```
FragmentRange split_range(Range *total)
{
    FragmentRange f;

    if (total->high == total->low) {
        // "Point" range, no real LH to do:
        // just implement standard fragmentation.
        uint16 len = min(total->high, FS);
        f->low = len;
        f->high = len;
        total->low -= len;
        total->high -= len;
    } else if (total->low >= FS) {
        // More bytes to send than a fragment can handle:
        // send as many bytes as possible.
        f->low = FS;
        f->high = FS;
        total->low -= FS;
        total->high -= FS;
    } else {
        // We are LH: add as much padding as necessary
        // in the current fragment.
        uint16 all_pad = max_lh_pad(total->low);
        all_pad = min(all_pad, total->high - total->low);
        f->low = total->low;
        f->high = total->low + all_pad;
        total->low = 0;
        total->high -= total->low + all_pad;
    }

    return f;
}
```

If invoked multiple times, this algorithm creates a list of `FragmentRange` structures, carrying all the payload up to `Range->low`, followed by a sequence of fragments carrying either padding or the remaining part of the message that exceeds `Range->low`.

4.1.1. Fragmenting Plaintext into Records

There are many ways to fragment the message content across a sequence of `FragmentRanges`. This document does not mandate any fragmentation algorithm. In the following, a fragmentation algorithm that tries to put as many bytes as possible in the first fragments is provided.

```
/* len: The total real plaintext length to be sent.
 * r0: a range that can be conveyed in one fragment,
 *     as returned by split_range.
 * r1: the remaining range used to send the remaining data
 * Returns: the number of bytes of plaintext to be sent
 *          in the next fragment with range r0.
uint16 fragment(uint32 len, FragmentRange r0, Range r1)
{
    return min(r0.high, len - r1.low);
}
```

4.1.2. Adding the Length Hiding Padding

If 'len' is the real plaintext length to be sent in a record fragment with range `FragmentRange`, a LH-TLS implementation **MUST** add at least `FragmentRange.high - len` bytes of padding to that record fragment (plus, if needed, some additional padding required to get block alignment).

If the `split_range` and `fragment` functions above are used, then the difference `FragmentRange.high - len` is always smaller than the maximum available padding size (including further block alignment padding).

4.1.3. A Length Hiding API

Finally, a LH-aware TLS implementation **MAY** use the algorithms described in Section 4.1 and Section 4.1.1 to offer a LH TLS API similar to the following, where it is assumed that a `TLS_send(data, len, target_length)` function sends a single TLS record fragment adding the necessary padding to match the `target_length`, as explained in Section 4.1.2.

```
uint32 message_send(opaque data, Range total)
{
    FragmentRange current;
    uint16 current_len, sent = 0;

    while (total.high != 0) {
        current = split_range(&total);
        next_len = fragment(data.length - sent, current, total);
        sent += TLS_send(&data[sent], next_len, current.high);
    }

    return sent;
}
```

This interface requires the TLS implementation to internally buffer

the entire application message. Alternatively, a LH TLS implementation MAY directly expose the `split_range` and `fragment` functions to the user, to avoid internal buffering. Note that it is only necessary to know the desired plaintext range to execute the `split_range` function, not the real plaintext size nor its content.

4.2. Applicability

If a TLS-LH mechanism is used in a TLS session, then TLS record protocol compression MUST be disabled. Compression is known to leak substantial information about the plaintext, including its length [COMPLEAK], which defeats the purpose of LH. Moreover, since in TLS compression happens after fragmentation, and the compression ratio is not known a priori, it is impossible to define a precise fragmentation strategy when compression is in place.

Length hiding can only work when some padding can be added before encryption, so that an attacker cannot distinguish whether the encrypted data are padding or application data. Hence, LH can only be used with block ciphers in standard TLS, and with any cipher when the extended record padding is used. In any case, length hiding MUST NOT be used with TLS_NULL_WITH_NULL_NULL or MAC-only ciphersuites.

5. Security Considerations

The LH scheme described in this document is effective in hiding the length of the exchanged messages, when an attacker observes the total bandwidth exchanged by a client and server using TLS. Crucially, the `split_range` algorithm, which determines the traffic shape and total bandwidth, MUST NOT depend on the real message length, but only on the `Range.low` and `Range.high` values, which are public.

Similarly, only the application knows when the recipient of the message is expected to react, upon receiving the message. For example, a web browser may start loading a hyperlink contained in an HTML file, as soon as the hyperlink is received, before the HTML file has been fully parsed. By using a callback for the implementation of the fragment function, a LH-aware application using a TLS-LH library can decide how much data to send in each fragment. An application should consider the TLS LH mechanism effective only to conceal the length of the message exchanged over the network.

Yet, an application on top of TLS could easily leak the message length, by performing visible actions after a known amount of bytes has been received. Hiding the length of the message at the application level is outside the scope of this document, and is a complex information flow property that should carefully be considered when designing a LH-aware implementation. Even the way the bytes are transferred from the TLS library to the application could leak information about their length.

5.1. Length Hiding with extended record padding

Since the padding is always included in the MAC computation, attacks that utilize the current CBC-padding timing channel (e.g., [DTLS-ATTACK]) are not applicable.

In a way, the extended record padding can be seen as a special way of encoding application data before encryption (where application data given by the user are prefixed by some padding). Hence, previous security results on standard TLS block and stream ciphers still apply to the extended record padding.

5.2. Length Hiding with standard TLS block ciphers

Section 6.2.3.2, Implementation note, of [RFC5246] acknowledges a small timing channel, due to the MAC timing depending on the length of each `TLSCiphertext.content`. Usage of large ranges with the LH scheme amplifies this timing channel, up to make it exploitable [LH-PADDING], because shorter messages within a range will be processed faster than longer messages in the same range.

Implementations supporting the LH scheme SHOULD implement a MAC algorithm whose execution time depends on the length of the `TLSCiphertext.content` plus the length of the padding, thus eliminating this timing channel.

5.3. Mitigating Denial of Service

The TLS protocol allows zero-length fragments of Application data, and these are exploited by the TLS length-hiding mechanism proposed in this document. For implementations that notify the application of such zero-length fragments, this poses no denial of service (DoS) issues. However, some TLS implementations will keep reading for the next fragment if a zero-length fragment is received. This exposes such implementations (especially server-side ones) to distributed DoS attacks, where a network of attackers connects to the same host and sends a sequence of zero-length fragments, keeping the host busy in processing them. This issue gets amplified when the "extended_record_padding" extension is used, because MAC computation includes a possibly large amount of padding.

Implementations that keep reading for the next fragment when a zero-length one is received, and that are concerned by such DoS attacks, MAY implement a DoS countermeasure. For example, they could accept 'n' zero-length fragments in a row, before notifying the application or returning an error. This conflicts with the requirements of a length-hiding mechanism, where zero-length fragments are used to conceal the real plaintext length. The value of 'n' SHOULD be chosen such that it is the smallest number of fragments that can convey the application-required LH padding. Usually, this value is application specific, so TLS implementations that implement this DoS mitigation SHOULD let 'n' be set by the application.

6. IANA Considerations

This document defines a new TLS extension, "extended_record_padding", assigned a value of TBD-BY-IANA (the value 48015 is suggested) from the TLS ExtensionType registry defined in [RFC5246]. This value is used as the extension number for the extensions in both the client hello message and the server hello message.

7. Normative References

- [RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.
- [RFC6066] Eastlake, D., "Transport Layer Security (TLS) Extensions: Extension Definitions", RFC 6066, January 2011.
- [DTLS-ATTACK]
Nadhem, N. and K. Paterson, "Plaintext-recovery attacks against datagram TLS.", Network and Distributed System Security Symposium , 2012.
- [LH-PADDING]
Pironti, A., Strub, P., and K. Bhargavan, "Identifying Website Users by TLS Traffic Analysis: New Attacks and Effective Countermeasures.", INRIA Research Report 8067 , 2012.
- [CBCTIME] Canvel, B., Hiltgen, A., Vaudenay, S., and M. Vuagnoux, "Password Interception in a SSL/TLS Channel", Advances in Cryptology -- CRYPTO , 2003.
- [COMPLEAK]
Kelsey, K., "Compression and information leakage of plaintext", Fast software encryption , 2002.

Appendix A. Acknowledgements

The authors wish to thank Kenny Paterson for his suggestions on improving this document.

Authors' Addresses

Alfredo Pironti
INRIA Paris-Rocquencourt
23, Avenue d'Italie
Paris, 75214 CEDEX 13
France

Email: alfredo.pironti@inria.fr

Nikos Mavrogiannopoulos
Dept. of Electrical Engineering ESAT/COSIC KU Leuven - iMinds
Kasteelpark Arenberg 10, bus 2446
Leuven-Heverlee, B-3001
Belgium

Email: nikos.mavrogiannopoulos@esat.kuleuven.be

