TLS Working Group
INTERNET-DRAFT
May 2, 2002
Expires November 2, 2002
Intended Category: Standards track

Simon Blake-Wilson, Certicom
Magnus Nystrom, RSA Security
David Hopwood, Independent Consultant
Jan Mikkelsen, Transactionware
Tim Wright, Vodafone

Transport Layer Security (TLS) Extensions

<draft-ietf-tls-extensions-04.txt>

Status of this Memo

Abstract

This document describes extensions that may be used to add functionality to TLS. It provides both generic extension mechanisms for the TLS handshake client and server hellos, and specific extensions using these generic mechanisms.

The extensions may be used by TLS clients and servers. The extensions are backwards compatible - communication is possible between TLS 1.0 clients that support the extensions and TLS 1.0 servers that do not support the extensions, and vice versa.

This document is based on discussions within the TLS working group and within the WAP security group.

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 [KEYWORDS].

Please send comments on this document to the TLS mailing list.

Table of Contents

1. Introduction

This document describes extensions that may be used to add functionality to TLS. It provides both generic extension mechanisms for the TLS handshake client and server hellos, and specific extensions using these generic mechanisms.

TLS is now used in an increasing variety of operational environments - many of which were not envisioned when the original design criteria for TLS were determined. The extensions introduced in this document are designed to enable TLS to operate as effectively as possible in new environments like wireless networks.

Wireless environments often suffer from a number of constraints not commonly present in wired environments - these constraints may include bandwidth limitations, computational power limitations, memory limitations, and battery life limitations.

The extensions described here focus on extending the functionality provided by the TLS protocol message formats. Other issues, such as

the addition of new cipher suites, are deferred.

Specifically, the extensions described in this document are designed to:

- Allow TLS clients to provide to the TLS server the name of the server they are contacting. This functionality is desirable to facilitate secure connections to servers that host multiple 'virtual' servers at a single underlying network address.

- Allow TLS clients and servers to negotiate the maximum fragment length to be sent. This functionality is desirable as a result of memory constraints among some clients, and bandwidth constraints among some access networks.

- Allow TLS clients and servers to negotiate the use of client certificate URLs. This functionality is desirable in order to conserve memory on constrained clients.

- Allow TLS clients to indicate to TLS servers which CA root keys they possess. This functionality is desirable in order to prevent multiple handshake failures involving TLS clients that are only able to store a small number of CA root keys due to memory limitations.

- Allow TLS clients and servers to negotiate the use of truncated MACs. This functionality is desirable in order to conserve bandwidth in constrained access networks.

- Allow TLS clients and servers to negotiate that the server sends the client certificate status information (e.g. an OCSP [OCSP] response) during a TLS handshake. This functionality is desirable in order to avoid sending a CRL over a constrained access network and therefore save bandwidth.

In order to support the extensions above, general extension mechanisms for the client hello message and the server hello message are introduced.

The extensions described in this document may be used by TLS 1.0 clients and TLS 1.0 servers. The extensions are designed to be backwards compatible - meaning that TLS 1.0 clients that support the extensions can talk to TLS 1.0 servers that do not support the extensions, and vice versa.

Backwards compatibility is primarily achieved via two considerations:

- Clients typically request the use of extensions via the extended client hello message described in Section 2.1. TLS 1.0 [TLS] requires servers to accept extended client hello messages, even if the server does not "understand" the extension.

- For the specific extensions described here, no mandatory server response is required when clients request extended functionality.

Note however, that although backwards compatibility is supported, some constrained clients may be forced to reject communications with servers that do not support the extensions as a result of the limited capabilities of such clients.

The remainder of this document is organized as follows. Section 2 describes general extension mechanisms for the client hello and server hello handshake messages. Section 3 describes specific extensions to TLS 1.0. Section 4 describes new error alerts for use with the TLS extensions. The final sections of the document address IPR, security considerations, registration of the application/pkix-pkipath MIME type, acknowledgements, and references.

2. General Extension Mechanisms

This section presents general extension mechanisms for the TLS handshake client hello and server hello messages.

These general extension mechanisms are necessary in order to enable clients and servers to negotiate whether to use specific extensions, and how to use specific extensions. The extension formats described are based on [MAILING LIST].

Section 2.1 specifies the extended client hello message format, Section 2.2 specifies the extended server hello message format, and Section 2.3 describes the actual extension format used with the extended client and server hellos.

2.1. Extended Client Hello

Clients MAY request extended functionality from servers by sending the extended client hello message format in place of the client hello message format. The extended client hello message format is:

```
struct {
    ProtocolVersion client_version;
    Random random;
    SessionID session_id;
    CipherSuite cipher_suites<2..2^16-1>;
    CompressionMethod compression_methods<1..2^8-1>;
    Extension client_hello_extension_list<0..2^16-1>;
} ClientHello;
```

Here the new "client_hello_extension_list" field contains a list of extensions. The actual "Extension" format is defined in Section 2.3.

In the event that clients request additional functionality using the extended client hello, and this functionality is not supplied by the server, clients MAY abort the handshake.

Note that [TLS], Section 7.4.1.2, allows additional information to be added to the client hello message. Thus the use of the extended client hello defined above should not "break" existing TLS 1.0 servers.

A server that supports the extensions mechanism MUST accept only client hello messages in either the original or extended ClientHello format, and (as for all other messages) MUST check that the amount of data in the message precisely matches one of these formats; if not then it MUST send a fatal "decode_error" alert. This overrides the "Forward compatibility note" in [TLS].

2.2. Extended Server Hello

The extended server hello message format MAY be sent in place of the server hello message when the client has requested extended functionality via the extended client hello message specified in Section 2.1. The extended server hello message format is:

```
struct {
    ProtocolVersion server_version;
    Random random;
    SessionID session_id;
    CipherSuite cipher_suite;
    CompressionMethod compression_method;
    Extension server_hello_extension_list<0..2^16-1>;
} ServerHello;
```

Here the new "server_hello_extension_list" field contains a list of extensions. The actual "Extension" format is defined in Section 2.3.

Note that the extended server hello message is only sent in response to an extended client hello message. This prevents the possibility that the extended server hello message could "break" existing TLS 1.0 clients.

2.3. Hello Extensions

The extension format for extended client hellos and extended server hellos is:

```
struct {
    ExtensionType extension_type;
    opaque extension_data<0..2^16-1>;
} Extension;
```

Here:

- "extension_type" identifies the particular extension type.

- "extension_data" contains information specific to the particular

extension type.

The extension types defined in this document are:

```
enum {
    server_name(0), max_fragment_length(1),
    client_certificate_url(2), trusted_ca_keys(3),
    truncated_hmac(4), status_request(5), (65535)
} ExtensionType;
```

Note that for all extension types (including those defined in future), the extension type MUST NOT appear in the extended server hello unless the same extension type appeared in the corresponding client hello. Thus clients MUST abort the handshake if they receive an extension type in the extended server hello that they did not request in the associated (extended) client hello.

Nonetheless "server initiated" extensions may be provided in the future within this framework by requiring the client to first send an empty extension to indicate that it supports a particular extension.

Also note that when multiple extensions of different types are present in the extended client hello or the extended server hello, the extensions may appear in any order. There MUST NOT be more than one extension of the same type.

Finally note that all the extensions defined in this document are relevant only when a session is initiated. However, a client that requests resumption of a session does not in general know whether the server will accept this request, and therefore it SHOULD send an extended client hello if it would normally do so for a new session. If the resumption request is denied, then a new set of extensions will be negotiated as normal. If, on the other hand, the older session is resumed, then the server MUST ignore extensions appearing in the client hello, and send a server hello containing no extensions; in this case the extension functionality negotiated during the original session initiation is applied to the resumed session.

2.4. Extensions to the handshake protocol

This document suggests the use of two new handshake messages, "CertificateURL" and "CertificateStatus". These messages are described in Section 3.3 and Section 3.6, respectively. The new handshake message structure therefore becomes:

```
enum {
    hello_request(0), client_hello(1), server_hello(2),
    certificate(11), server_key_exchange (12),
    certificate_request(13), server_hello_done(14),
    certificate_verify(15), client_key_exchange(16),
```

```
        finished(20), certificate_url(21), certificate_status(22),
        (255)
    } HandshakeType;

    struct {
        HandshakeType msg_type;    /* handshake type */
        uint24 length;             /* bytes in message */
        select (HandshakeType) {
            case hello_request:       HelloRequest;
            case client_hello:        ClientHello;
            case server_hello:        ServerHello;
            case certificate:         Certificate;
            case server_key_exchange: ServerKeyExchange;
            case certificate_request: CertificateRequest;
            case server_hello_done:   ServerHelloDone;
            case certificate_verify:  CertificateVerify;
            case client_key_exchange: ClientKeyExchange;
            case finished:            Finished;
            case certificate_url:     CertificateURL;
            case certificate_status:  CertificateStatus;
        } body;
    } Handshake;
```

3. Specific Extensions

This section describes the specific TLS extensions specified in this document.

Note that any messages associated with these extensions that are sent during the TLS handshake MUST be included in the hash calculations involved in "Finished" messages.

Section 3.1 describes the extension of TLS to allow a client to indicate which server it is contacting. Section 3.2 describes the extension to provide maximum fragment length negotiation. Section 3.3 describes the extension to allow client certificate URLs. Section 3.4 describes the extension to allow a client to indicate which CA root keys it possesses. Section 3.5 describes the extension to allow the use of truncated HMAC. Section 3.6 describes the extension to support integration of certificate status information messages into TLS handshakes.

3.1. Server Name Indication

[TLS] does not provide a mechanism for a client to tell a server the name of the server it is contacting. It may be desirable for clients to provide this information to facilitate secure connections to servers that host multiple 'virtual' servers at a single underlying network address.

In order to provide the server name, clients MAY include an extension

of type "server_name" in the (extended) client hello. The "extension_data" field of this extension SHALL contain "ServerNameList" where:

```
    struct {
        NameType name_type;
        select (name_type) {
            case host_name: HostName;
        } name;
    } ServerName;

    enum {
        host_name(0), (255)
    } NameType;

    opaque HostName<1..2^16-1>;

    struct {
        ServerName server_name_list<1..2^16-1>
    } ServerNameList;
```

Currently the only server names supported are DNS hostnames, however this does not imply any dependency of TLS on DNS, and other name types may be added in the future (by an RFC that Updates this document). TLS MAY treat provided server names as opaque data and pass the names and types to the application.

"HostName" contains the fully qualified DNS hostname of the server, as understood by the client. The hostname is represented as a byte string using UTF-8 encoding [UTF8], without a trailing dot. (Note that the use of UTF-8 here for encoding internationalized hostnames is independent of the choice of encoding for these names in the DNS protocol. The latter has yet to be decided by the IETF Internationalized Domain Name Working Group [IDNWG].)

If the server needs to match the HostName against names that contain non-ASCII characters (that is, if it has one or more internationalized virtual host names), it MUST take account of name equivalence rules that will be defined by the IDN Working Group. If the server only needs to match the HostName against names containing exclusively ASCII characters, it MUST NOT treat a HostName that contains a byte value >= 128 as matching any ASCII name, and it MUST compare ASCII names case-insensitively.

Literal IPv4 and IPv6 addresses are not permitted in "HostName".

It is RECOMMENDED that clients include an extension of type "server_name" in the client hello whenever they locate a server by a supported name type.

A server that receives a client hello containing the "server_name" extension, MAY use the information contained in the extension to

guide its selection of an appropriate certificate to return to the client, and/or other aspects of security policy. In this event, the server SHALL include an extension of type "server_name" in the (extended) server hello. The "extension_data" field of this extension SHALL be empty.

If the server understood the client hello extension but does not recognize the server name, it SHOULD send an "unrecognized_name" alert (which MAY be fatal).

If an application negotiates a server name using an application protocol, then upgrades to TLS, and a server_name extension is sent, then the extension SHOULD contain the same name that was negotiated in the application protocol. If the server_name is established in the TLS session handshake, the client SHOULD NOT attempt to request a different server name at the application layer.

3.2. Maximum Fragment Length Negotiation

[TLS] specifies a fixed maximum plaintext fragment length of 2^14 bytes. It may be desirable for constrained clients to negotiate a smaller maximum fragment length due to memory limitations or bandwidth limitations.

In order to negotiate smaller maximum fragment lengths, clients MAY include an extension of type "max_fragment_length" in the (extended) client hello. The "extension_data" field of this extension SHALL contain:

```
enum{
    2^9(1), 2^10(2), 2^11(3), 2^12(4), (255)
} MaxFragmentLength;
```

whose value is the desired maximum fragment length. The allowed values for this field are: 2^9, 2^10, 2^11, and 2^12.

Servers that receive an extended client hello containing a "max_fragment_length" extension, MAY accept the requested maximum fragment length by including an extension of type "max_fragment_length" in the (extended) server hello. The "extension_data" field of this extension SHALL contain "MaxFragmentLength" whose value is the same as the requested maximum fragment length.

If a server receives a maximum fragment length negotiation request for a value other than the allowed values, it MUST abort the handshake with an "illegal_parameter" alert. Similarly, if a client receives a maximum fragment length negotiation response that differs from the length it requested, it MUST also abort the handshake with an "illegal_parameter" alert.

Once a maximum fragment length other than 2^14 has been successfully

negotiated, the client and server MUST immediately begin fragmenting messages (including handshake messages), to ensure that no fragment larger than the negotiated length is sent. Note that TLS already requires clients and servers to support fragmentation of handshake messages.

The negotiated length applies for the duration of the session including session resumptions.

The negotiated length limits the input that the record layer may process without fragmentation (that is, the maximum value of TLSPlaintext.length; see [TLS] section 6.2.1). Note that the output of the record layer may be larger. For example, if the negotiated length is 2^9=512, then for currently defined cipher suites (those defined in [TLS], [KERB], and planned AES cipher suites), the record layer output can be at most 793 bytes: 5 bytes of headers, 512 bytes of application data, 256 bytes of padding, and 20 bytes of MAC. That means that in this event a TLS record layer peer receiving a TLS record layer message larger than 793 bytes may discard the message and send a "record_overflow" alert, without decrypting the message.

3.3. Client Certificate URLs

[TLS] specifies that when client authentication is performed, client certificates are sent by clients to servers during the TLS handshake. It may be desirable for constrained clients to send certificate URLs in place of certificates, so that they do not need to store their certificates and can therefore save memory.

In order to negotiate to send certificate URLs to a server, clients MAY include an extension of type "client_certificate_url" in the (extended) client hello. The "extension_data" field of this extension SHALL be empty.

(Note that it is necessary to negotiate use of client certificate URLs in order to avoid "breaking" existing TLS 1.0 servers.)

Servers that receive an extended client hello containing a "client_certificate_url" extension, MAY indicate that they are willing to accept certificate URLs by including an extension of type "client_certificate_url" in the (extended) server hello. The "extension_data" field of this extension SHALL be empty.

After negotiation of the use of client certificate URLs has been successfully completed (by exchanging hellos including "client_certificate_url" extensions), clients MAY send a "CertificateURL" message in place of a "Certificate" message:

```
enum {
    individual_certs(0), pkipath(1), (255)
} CertChainType;
```

```
    enum {
        false(0), true(1)
    } Boolean;

    struct {
        CertChainType type;
        URLAndOptionalHash url_and_hash_list<1..2^16-1>;
    } CertificateURL;

    struct {
        opaque url<1..2^16-1>;
        Boolean hash_present;
        select (hash_present) {
            case false: struct {};
            case true: SHA1Hash;
        } hash;
    } URLAndOptionalHash;

    opaque SHA1Hash[20];
```

Here "url_and_hash_list" contains a sequence of URLs and optional hashes.

When X.509 certificates are used, there are two possibilities:

- if CertificateURL.type is "individual_certs", each URL refers to a single DER-encoded X.509v3 certificate, with the URL for the client's certificate first, or

- if CertificateURL.type is "pkipath", the list contains a single URL referring to a DER-encoded certificate chain, using the type PkiPath described in Section 8.

When any other certificate format is used, the specification that describes use of that format in TLS should define the encoding format of certificates or certificate chains, and any constraint on their ordering.

The hash corresponding to each URL at the client's discretion is either not present or is the SHA-1 hash of the certificate or certificate chain (in the case of X.509 certificates, the DER-encoded certificate or the DER-encoded PkiPath).

Note that when a list of URLs for X.509 certificates is used, the ordering of URLs is the same as that used in the TLS Certificate message (see [TLS] Section 7.4.2), but opposite to the order in which certificates are encoded in PkiPath. In either case, the self-signed root certificate MAY be omitted from the chain, under the assumption that the server must already possess it in order to validate it.

Servers receiving "CertificateURL" SHALL attempt to retrieve the client's certificate chain from the URLs, and then process the

certificate chain as usual. Servers that support this extension MUST support the http: URL scheme for certificate URLs, and MAY support other schemes.

If the protocol used to retrieve certificates or certificate chains returns a MIME formatted response (as HTTP does), then the following MIME Content-Types SHALL be used: when a single X.509v3 certificate is returned, the Content-Type is "application/pkix-cert" [PKIOP], and when a chain of X.509v3 certificates is returned, the Content-Type is "application/pkix-pkipath" (see Section 8).

If a SHA-1 hash is present for an URL, then the server MUST check that the SHA-1 hash of the contents of the object retrieved from that URL (after decoding any MIME Content-Transfer-Encoding) matches the given hash. If any retrieved object does not have the correct SHA-1 hash, the server MUST abort the handshake with a "bad_certificate_hash_value" alert.

Note that clients may choose to send either "Certificate" or "CertificateURL" after successfully negotiating the option to send certificate URLs. The option to send a certificate is included to provide flexibility to clients possessing multiple certificates.

If a server encounters an unreasonable delay in obtaining certificates in a given CertificateURL, it SHOULD time out and signal a "certificate_unobtainable" error alert.

3.4. Trusted CA Indication

Constrained clients that, due to memory limitations, possess only a small number of CA root keys, may wish to indicate to servers which root keys they possess, in order to avoid repeated handshake failures.

In order to indicate which CA root keys they possess, clients MAY include an extension of type "trusted_ca_keys" in the (extended) client hello. The "extension_data" field of this extension SHALL contain "TrustedAuthorities" where:

```
    struct {
        TrustedAuthority trusted_authorities_list<0..2^16-1>;
    } TrustedAuthorities;

    struct {
        IdentifierType identifier_type;
        select (identifier_type) {
            case pre_agreed: struct {};
            case key_sha1_hash: SHA1Hash;
            case x509_name: DistinguishedName;
            case cert_sha1_hash: SHA1Hash;
        } identifier;
```

```
    } TrustedAuthority;

    enum {
        pre_agreed(0), key_sha1_hash(1), x509_name(2),
        cert_sha1_hash(3), (255)
    } IdentifierType;

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

Here "TrustedAuthorities" provides a list of CA root key identifiers that the client possesses. Each CA root key is identified via either:

- "pre_agreed" - no CA root key identity supplied.

- "key_sha1_hash" - contains the SHA-1 hash of the CA root key. For DSA and ECDSA keys, this is the hash of the "subjectPublicKey" value. For RSA keys, the hash is of the byte string representation of the modulus without any initial 0-valued bytes. (This copies the key hash formats deployed in other environments.)

- "x509_name" - contains the DER-encoded X.509 DistinguishedName of the CA.

- "cert_sha1_hash" - contains the SHA-1 hash of a DER-encoded Certificate containing the CA root key.

Note that clients may include none, some, or all of the CA root keys they possess in this extension.

Note also that it is possible that a key hash or a Distinguished Name alone may not uniquely identify a certificate issuer - for example if a particular CA has multiple key pairs - however here we assume this is the case following the use of Distinguished Names to identify certificate issuers in TLS.

The option to include no CA root keys is included to allow the client to indicate possession of some pre-defined set of CA root keys.

Servers that receive a client hello containing the "trusted_ca_keys" extension, MAY use the information contained in the extension to guide their selection of an appropriate certificate chain to return to the client. In this event, the server SHALL include an extension of type "trusted_ca_keys" in the (extended) server hello. The "extension_data" field of this extension SHALL be empty.

3.5. Truncated HMAC

Currently defined TLS cipher suites use the MAC construction HMAC with either MD5 or SHA-1 [HMAC] to authenticate record layer communications. In TLS the entire output of the hash function is used as the MAC tag. However it may be desirable in constrained

environments to save bandwidth by truncating the output of the hash function to 80 bits when forming MAC tags.

In order to negotiate the use of 80-bit truncated HMAC, clients MAY include an extension of type "truncated_hmac" in the extended client hello. The "extension_data" field of this extension SHALL be empty.

Servers that receive an extended hello containing a "truncated_hmac" extension, MAY agree to use a truncated HMAC by including an extension of type "truncated_hmac" in the extended server hello.

Note that if new cipher suites are added that do not use HMAC, and the session negotiates one of these cipher suites, this extension will have no effect. It is strongly recommended that any new cipher suites using other MACs consider the MAC size as an integral part of the cipher suite definition, taking into account both security and bandwidth considerations.

If HMAC truncation has been successfully negotiated during a TLS handshake, and the negotiated cipher suite uses HMAC, both the client and the server pass this fact to the TLS record layer along with the other negotiated security parameters. Subsequently during the session, clients and servers MUST use truncated HMACs, calculated as specified in [HMAC]. That is, CipherSpec.hash_size is 10 bytes, and only the first 10 bytes of the HMAC output are transmitted and checked. Note that this extension does not affect the calculation of the PRF as part of handshaking or key derivation.

The negotiated HMAC truncation size applies for the duration of the session including session resumptions.

3.6. Certificate Status Request

Constrained clients may wish to use a certificate-status protocol such as OCSP [OCSP] to check the validity of server certificates, in order to avoid transmission of CRLs and therefore save bandwidth on constrained networks. This extension allows for such information to be sent in the TLS handshake, saving roundtrips and resources.

In order to indicate their desire to receive certificate status information, clients MAY include an extension of type "status_request" in the (extended) client hello. The "extension_data" field of this extension SHALL contain "CertificateStatusRequest" where:

```
struct {
    CertificateStatusType status_type;
    select (status_type) {
        case ocsp: OCSPStatusRequest;
    } request;
} CertificateStatusRequest;
```

```
    enum { ocsp(1), (255) } CertificateStatusType;

    struct {
        ResponderID responder_id_list<0..2^16-1>;
        Extensions  request_extensions;
    } OCSPStatusRequest;

    opaque ResponderID<1..2^16-1>;
    opaque Extensions<0..2^16-1>;
```

In the OCSPStatusRequest, the "ResponderIDs" provides a list of OCSP responders that the client trusts. A zero-length "responder_id_list" sequence has the special meaning that the responders are implicitly known to the server - e.g. by prior arrangement. "Extensions" is a DER encoding of OCSP request extensions.

Both "ResponderID" and "Extensions" are DER-encoded ASN.1 types as defined in [OCSP].

Servers that receive a client hello containing the "status_request" extension, MAY return a suitable certificate status response to the client along with their certificate. If OCSP is requested, they SHOULD use the information contained in the extension when selecting an OCSP responder, and SHOULD include request_extensions in the OCSP request.

Servers return a certificate response along with their certificate by sending a "CertificateStatus" message immediately after the "Certificate" message (and before any "ServerKeyExchange" or "CertificateRequest" messages). If a server returns a "CertificateStatus" message, then the server MUST have included an extension of type "status_request" with empty "extension_data" in the extended server hello.

```
    struct {
        CertificateStatusType status_type;
        select (status_type) {
            case ocsp: OCSPResponse;
        } response;
    } CertificateStatus;

    opaque OCSPResponse<1..2^24-1>;
```

An "ocsp_response" contains a complete, DER-encoded OCSP response (using the ASN.1 type OCSPResponse defined in [OCSP]). Note that only one OCSP response may be sent.

The "CertificateStatus" message is conveyed using the handshake message type "certificate_status".

Note that a server MAY also choose not to send a "CertificateStatus" message, even if it receives a "status_request" extension in the

client hello message.

Note in addition that servers MUST NOT send the "CertificateStatus" message unless it received a "status_request" extension in the client hello message.

Clients requesting an OCSP response, and receiving an OCSP response in a "CertificateStatus" message MUST check the OCSP response and abort the handshake if the response is not satisfactory.

4. Error Alerts

This section defines new error alerts for use with the TLS extensions defined in this document.

The following new error alerts are defined. To avoid "breaking" existing clients and servers, these alerts MUST NOT be sent unless the sending party has received an extended hello message from the party they are communicating with.

- "unsupported_extension" - this alert is sent by clients that receive an extended server hello containing an extension that they did not put in the corresponding client hello (see Section 2.3). This message is always fatal.

- "unrecognized_name" - this alert is sent by servers that receive a server_name extension request, but do not recognize the server name. This message MAY be fatal.

- "certificate_unobtainable" - this alert is sent by servers who are unable to retrieve a certificate chain from the URL supplied by the client (see Section 3.3). This message MAY be fatal - for example if client authentication is required by the server for the handshake to continue and the server is unable to retrieve the certificate chain, it may send a fatal alert.

- "bad_certificate_status_response" - this alert is sent by clients that receive an invalid certificate status response (see Section 3.6). This message is always fatal.

- "bad_certificate_hash_value" - this alert is sent by servers when a certificate hash does not match a client provided certificate_hash. This message is always fatal.

These error alerts are conveyed using the following syntax:

```
    enum {
        close_notify(0),
        unexpected_message(10),
        bad_record_mac(20),
        decryption_failed(21),
        record_overflow(22),
```

```
        decompression_failure(30),
        handshake_failure(40),
        /* 41 is not defined, for historical reasons */
        bad_certificate(42),
        unsupported_certificate(43),
        certificate_revoked(44),
        certificate_expired(45),
        certificate_unknown(46),
        illegal_parameter(47),
        unknown_ca(48),
        access_denied(49),
        decode_error(50),
        decrypt_error(51),
        export_restriction(60),
        protocol_version(70),
        insufficient_security(71),
        internal_error(80),
        user_canceled(90),
        no_renegotiation(100),
        unsupported_extension(110),           /* new */
        certificate_unobtainable(111),        /* new */
        unrecognized_name(112),               /* new */
        bad_certificate_status_response(113), /* new */
        bad_certificate_hash_value(114),      /* new */
        (255)
    } AlertDescription;
```

5. Procedure for Defining New Extensions

Traditionally for Internet protocols, the Internet Assigned Numbers Authority (IANA) handles the allocation of new values for future expansion, and RFCs usually define the procedure to be used by the IANA. However, there are subtle (and not so subtle) interactions that may occur in this protocol between new features and existing features which may result in a significant reduction in overall security.

Therefore, requests to define new extensions (including assigning extension and error alert numbers) should be forwarded to the IETF TLS Working Group for discussion.

The following considerations should be taken into account when designing new extensions:

- All of the extensions defined in this document follow the convention that for each extension that a client requests and that the server understands, the server replies with an extension of the same type.

- Some cases where a server does not agree to an extension are error conditions, and some simply a refusal to support a particular feature. In general error alerts should be used for

the former, and a field in the server extension response for the latter.

- Extensions should as far as possible be designed to prevent any attack that forces use (or non-use) of a particular feature by manipulation of handshake messages. This principle should be followed regardless of whether the feature is believed to cause a security problem.

  Often the fact that the extension fields are included in the inputs to the Finished message hashes will be sufficient, but extreme care is needed when the extension changes the meaning of messages sent in the handshake phase. Designers and implementors should be aware of the fact that until the handshake has been authenticated, active attackers can modify messages and insert, remove, or replace extensions.

- It would be technically possible to use extensions to change major aspects of the design of TLS; for example the design of cipher suite negotiation. This is not recommended; it would be more appropriate to define a new version of TLS - particularly since the TLS handshake algorithms have specific protection against version rollback attacks based on the version number, and the possibility of version rollback should be a significant consideration in any major design change.

6. Security Considerations

Security considerations for the extension mechanism in general, and the design of new extensions, are described in the previous section. A security analysis of each of the extensions defined in this document is given below.

In general, implementers should continue to monitor the state of the art, and address any weaknesses identified.

Additional security considerations are described in the TLS 1.0 RFC [TLS].

6.1. Security of server_name

If a single server hosts several domains, then clearly it is necessary for the owners of each domain to ensure that this satisfies their security needs. Apart from this, server_name does not appear to introduce significant security issues.

Implementations MUST ensure that a buffer overflow does not occur whatever the values of the length fields in server_name.

6.2. Security of max_fragment_length

The maximum fragment length takes effect immediately, including for handshake messages. However, that does not introduce any security complications that are not already present in TLS, since [TLS] requires implementations to be able to handle fragmented handshake messages.

Note that as described in section 3.2, once a non-null cipher suite has been activated, the effective maximum fragment length depends on the cipher suite, as well as on the negotiated max_fragment_length. This must be taken into account when sizing buffers, and checking for buffer overflow.

6.3. Security of client_certificate_url

The major issue with this extension is whether or not clients should include certificate hashes when they send certificate URLs.

When client authentication is used *without* the client_certificate_url extension, the client certificate chain is covered by the Finished message hashes. The purpose of including hashes and checking them against the retrieved certificate chain, is to ensure that the same property holds when this extension is used - i.e. that all of the information in the certificate chain retrieved by the server is as the client intended.

On the other hand, omitting certificate hashes enables functionality that is desirable in some circumstances - for example clients can be issued daily certificates that are stored at a fixed URL and need not be provided to the client. Clients that choose to omit certificate hashes should be aware of the possibility of an attack in which the attacker obtains a valid certificate on the client's key that is different from the certificate the client intended to provide.

Also note that HTTP caching proxies are common on the Internet, and some proxies do not check for the latest version of an object correctly. If a request using HTTP (or another caching protocol) goes through a misconfigured or otherwise broken proxy, the proxy may return an out-of-date response.

Although TLS uses both MD5 and SHA-1 hashes in several other places, this was not believed to be necessary here. The property required of SHA-1 is second pre-image resistance.

Support for client_certificate_url involves the server acting as a client in another protocol (usually HTTP, but other URL schemes are not prohibited). It is therefore subject to many of the same security considerations that apply to a publicly accessible HTTP proxy server. This includes the possibility that an attacker might use the server to indirectly attack another host that is vulnerable to some security flaw. It also includes potentially increased exposure to denial of service attacks: an attacker can make many connections, each of which

results in the server making an HTTP request.

It is RECOMMENDED that the client_certificate_url extension should have to be specifically enabled by a server administrator, rather than being enabled by default.

As discussed in [URI], URLs that specify ports other than the default may cause problems, as may very long URLs (which are more likely to be useful in exploiting buffer overflow bugs).

6.4. Security of trusted_ca_keys

It is possible that which CA root keys a client possesses could be regarded as confidential information. As a result, the CA root key indication extension should be used with care.

The use of the SHA-1 certificate hash alternative ensures that each certificate is specified unambiguously. As for the previous extension, it was not believed necessary to use both MD5 and SHA-1 hashes.

6.5. Security of truncated_hmac

It is possible that truncated MACs are weaker than "un-truncated" MACs. However, no significant weaknesses are currently known or expected to exist for HMAC with MD5 or SHA-1, truncated to 80 bits. Note that the output length of a MAC need not be as long as the length of a symmetric cipher key, since forging of MAC values cannot be done off-line: in TLS, a single failed MAC guess will cause the immediate termination of the TLS session.

Since the MAC algorithm only takes effect after the handshake messages have been authenticated by the hashes in the Finished messages, it is not possible for an active attacker to force negotiation of the truncated HMAC extension where it would not otherwise be used (to the extent that the handshake authentication is secure). Therefore, in the event that any security problem were found with truncated HMAC in future, if either the client or the server for a given session were updated to take into account the problem, they would be able to veto use of this extension.

6.6. Security of status_request

If a client requests an OCSP response, it must take into account that an attacker's server using a compromised key could (and probably would) pretend not to support the extension. A client that requires OCSP validation of certificates SHOULD either contact the OCSP server directly in this case, or abort the handshake.

Use of the OCSP nonce request extension (id-pkix-ocsp-nonce) may improve security against attacks that attempt to replay OCSP responses; see section 4.4.1 of [OCSP] for further details.

7. Internationalization Considerations

None of the extensions defined here directly use strings subject to localization. DNS hostnames are encoded using UTF-8. If future extensions use text strings, then internationalization should be considered in their design.

8. IANA Considerations

The MIME type "application/pkix-pkipath" is to be registered with the following template:

To: ietf-types@iana.org
Subject: Registration of MIME media type application/pkix-pkipath

MIME media type name: application

MIME subtype name: pkix-pkipath

Required parameters: none

Optional parameters: version (default value is "1")

Encoding considerations:

This MIME type is a DER encoding of the ASN.1 type PkiPath, defined as follows:

```
PkiPath ::= SEQUENCE OF Certificate
```

PkiPath is used to represent a certification path. Within the sequence, the order of certificates is such that the subject of the first certificate is the issuer of the second certificate, etc.

This is identical to the definition that will be published in [X509-4th-TC1]; note that it is different from that in [X509-4th].

All Certificates MUST conform to [PKIX] (an update to [PKIX] is in preparation, and should be followed when it is published). DER (as opposed to BER) encoding MUST be used. If this type is sent over a 7-bit transport, base64 encoding SHOULD be used.

Security considerations:

The security considerations of [X509-4th] and [PKIX] (or any updates to them) apply, as well as those of any protocol that uses this type (e.g. TLS).

Note that this type only specifies a certificate chain that can be assessed for validity according to the relying party's existing configuration of trusted CAs; it is not intended to be used to specify any change to that configuration.

Interoperability considerations:
   No specific interoperability problems are known with this type,
   but for recommendations relating to X.509 certificates in general,
   see [PKIX].

Published specification: <draft-ietf-tls-extensions-04.txt> and
   [PKIX].

Applications which use this media type: TLS. It may also be used by
   other protocols, or for general interchange of PKIX certificate
   chains.

Additional information:
   Magic number(s): DER-encoded ASN.1 can be easily recognised.
      Further parsing is required to distinguish from other ASN.1
      types.
   File extension(s): .pkipath
   Macintosh File Type Code(s): not specified

Person & email address to contact for further information:
   Magnus Nystrom <magnus@rsasecurity.com>

Intended usage: COMMON

Author/Change controller:
   Magnus Nystrom <magnus@rsasecurity.com>

9. Intellectual Property Rights

The IETF invites any interested party to bring to its attention any
copyrights, patents or patent applications, or other proprietary
rights which may cover technology that may be required to practice
this document. Please address the information to the IETF Executive
Director.

10. Acknowledgments

The authors wish to thank the TLS Working Group and the WAP Security
Group. This document is based on discussion within these groups.

11. Normative References

[HMAC] H. Krawczyk, M. Bellare, and R. Canetti, "HMAC: Keyed-hashing
for message authentication," IETF RFC 2104, February 1997.

[HTTP] J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T.
Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1," IETF RFC
2616, June 1999.

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

[OCSP] M. Myers, R. Ankney, A. Malpani, S. Galperin, and C. Adams, "Internet X.509 Public Key Infrastructure: Online Certificate Status Protocol - OCSP," IETF RFC 2560, June 1999.

[PKIOP] R. Housley and P. Hoffman, "Internet X.509 Public Key Infrastructure - Operation Protocols: FTP and HTTP," IETF RFC 2585, May 1999.

[PKIX] R. Housley, W. Ford, W. Polk, and D. Solo, "Internet Public Key Infrastructure: Part I: X.509 Certificate and CRL Profile", IETF RFC 2459, January 1999.

[TLS] T. Dierks and C. Allen, "The TLS Protocol - Version 1.0," IETF RFC 2246, January 1999.

[URI] T. Berners-Lee, R. Fielding, and L. Masinter, "Uniform Resource Identifiers (URI): Generic Syntax," IETF RFC 2396, August 1998.

[UTF8] F. Yergeau, "UTF-8, a transformation format of ISO 10646," IETF RFC 2279, January 1998.

[X509-4th] ITU-T Recommendation X.509 (2000) | ISO/IEC 9594-8:2001, "Information Systems - Open Systems Interconnection - The Directory: Public key and attribute certificate frameworks."

[X509-4th-TC1] ITU-T Recommendation X.509(2000) Corrigendum 1(2001) | ISO/IEC 9594-8:2001/Cor.1:2002, Technical Corrigendum 1 to ISO/IEC 9594:8:2001.

12. Informative References

[IDNWG] IETF Internationalized Domain Name Working Group, http://www.i-d-n.net/

[KERB] A. Medvinsky and M. Hur, "Addition of Kerberos Cipher Suites to Transport Layer Security (TLS)," IETF RFC 2712, October 1999.

[MAILING LIST] J. Mikkelsen, R. Eberhard, and J. Kistler, "General ClientHello extension mechanism and virtual hosting," ietf-tls mailing list posting, August 14, 2000.

13. Authors' Addresses

Simon Blake-Wilson
Certicom Corp.
sblake-wilson@certicom.com

Magnus Nystrom
RSA Security
magnus@rsasecurity.com

David Hopwood
Independent Consultant
david.hopwood@zetnet.co.uk

Jan Mikkelsen
Transactionware
janm@transactionware.com

Tim Wright
Vodafone
timothy.wright@vf.vodafone.co.uk