TLS Working Group
Internet-Draft
Intended status: Standards Track
Expires: April 26, 2017
P. Gutmann
University of Auckland
October 23, 2016

TLS 1.2 Update for Long-term Support
draft-gutmann-tls-lts-06

Abstract

This document specifies an update of TLS 1.2 for long-term support, one that incoporates as far as possible what's already deployed for TLS 1.2 but with the security holes and bugs fixed. This represents a stable, known-good version that can be deployed to systems that can't roll out ongoing patches and updates every time a new attack on standard TLS appears.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

TLS [2] and DTLS [5], by nature of their enormous complexity and the inclusion of large amounts of legacy material, contain numerous security issues that have been known to be a problem for many years and that keep coming up again and again in attacks (there are too many of these to provide references for in the standard manner, and in any case more will have been published by the time you read this). This document presents a minimal, known-good set of mechanisms that defend against all currently-known weaknesses in TLS, that would have defended against them ten years ago, and that have a good chance of defending against them ten years from now, providing the long-term stability that's required by many systems in the field. This long-term stability is particularly important in light of the fact that widespread mainstream adoption of new versions of TLS has been shown to take 15 years or more [21].

In particular, this document takes inspiration from numerous published analyses of TLS [11] [12] [13] [14] [15] [16] [17] [18] [19] [20] along with two decades of implementation and deployment experience to select a standard interoperable feature set that provides the best chance of long-term stability and resistance to attack. This is intended for use in systems that need to run in a fixed configuration for a long period of time after they're deployed,

with little or no ability to roll out patches every month or two when the next attack on TLS is published.

Unlike the full TLS 1.2, TLS-LTS is not meant to be all things to all people. It represents a fixed, safe solution that's appropriate for users who require a simple, secure, and long-term stable means of getting data from A to B. This represents the majority of the non-browser uses of TLS, particularly for embedded systems that are most in need of a long-term stable protocol definition.

[Note: There is currently a TLS 1.2 LTS test server running at https://82.94.251.205:8443. This uses the extension value 26 until a value is permanently assigned for LTS use. To connect, your implementation should accept whatever certificate is presented by the server or use PSK with name = "plc", password = "test". For embedded systems testing, note that the server talks HTTP and not DNP3 or ICCP, so you'll get an error if you try and connect with a PLC control centre that expects one of those protocols].

1.1. Conventions Used in This Document

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 [1].

2. TLS-LTS Negotiation

The use of TLS-LTS is negotiated via TLS/DTLS extensions as defined in TLS Extensions [4]. On connecting, the client includes the tls_lts extension in its Client Hello if it wishes to use TLS-LTS. If the server is capable of meeting this requirement, it responds with a tls_lts extension in its Server Hello. The "extension_type" value for this extension MUST be TBD (0xTBD) and the "extension_data" field of this extension is empty. The client and server MUST NOT use TLS-LTS unless both sides have successfully exchanged tls_lts extensions.

In the case of session resumption, once TLS-LTS has been negotiated implementations MUST retain the use of TLS-LTS across all subsequent resumed sessions. In other words if TLS-LTS is enabled for the current session then the resumed session MUST also use TLS-LTS. If a client attempts to resume a TLS-LTS session as a non-TLS-LTS session then the server MUST abort the handshake.

3. TLS-LTS

TLS-LTS specifies a few simple restrictions on the huge range of TLS suites, options and parameters to limit the protocol to a known-good subset, as well as making minor corrections to prevent or at least limit various attacks.

3.1. Encryption/Authentication

TLS-LTS restricts the more or less unlimited TLS 1.2 with its more than three hundred cipher suites, over forty ECC parameter sets, and zoo of supplementary algorithms, parameters, and parameter formats, to just two, one traditional one with DHE + AES-CBC + HMAC-SHA-256 + RSA-SHA-256/PSK and one ECC one with ECDHE-P256 + AES-GCM + HMAC-SHA-256 + ECDSA-P256-SHA-256/PSK with uncompressed points:

- o TLS-LTS implementations MUST support TLS_DHE_RSA_WITH_AES_128_CBC_SHA256, TLS_DHE_PSK_WITH_AES_128_CBC_SHA256, TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256 and TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA256. For these suites, SHA-256 is used in all locations in the protocol where a hash function is required, specifically in the PRF and per-packet MAC calculations (as indicated by the _SHA256 in the suite) and also in the client and server signatures in the CertificateVerify and ServerKeyExchange messages.

  [Note: There's a gap in the suites with TLS_ECDHE_PSK_WITH_AES_128_GCM_SHA256 missing, there's currently a draft in progress to fill the gap, draft-mattsson-tls-ecdhe-psk-aead, which can be used to replace the placeholder TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA256].

TLS-LTS only permits encrypt-then-MAC, not MAC-then-encrypt, fixing 20 years of attacks on this mechanism:

- o TLS-LTS implementations MUST implement encrypt-then-MAC [6] rather than the earlier MAC-then-encrypt.

TLS-LTS adds a hash of all messages leading up to the calculation of the master secret into the master secret to protect against the use of manipulated handshake parameters:

- o TLS-LTS implementations MUST implement extended master secret [8] to protect handshake and crypto parameters.

TLS-LTS drops the MAC truncation in the Finished message, which serves no obvious purpose and leads to security concerns:

o The length of verify_data (verify_data_length) in the Finished message MUST be equal to the length of the output of the hash function used for the PRF. For the mandatory TLS-LTS cipher suites this hash is always SHA-256, so the value of verify_data_length will be 32 bytes. For other suites, the size depends on the hash algorithm associated with the suite. For example for SHA-512 it would be 64 bytes.

TLS-LTS signs a hash of the client and server hello messages for the ServerKeyExchange rather than signing just the client and server nonces, avoiding various attacks that build on the fact that standard TLS doesn't authenticate previously-exchanged parameters when the ServerKeyExchange message is sent:

o When generating the ServerKeyExchange signature, the signed_params value is updated to replace the client_random and server_random with a hash of the full Client Hello and Server Hello using the hash algorithm for the chosen cipher suite. In other words the value being signed is changed from:

```
digitally-signed struct {
    opaque client_random[32];
    opaque server_random[32];
    ServerDHParams params;
    } signed_params;
```

to:

```
digitally-signed struct {
    opaque client_server_hello_hash;
    ServerDHParams params;
    } signed_params;
```

For the mandatory TLS-LTS cipher suites the hash algorithm is always SHA-256, so the length of the client_server_hello_hash is 32 bytes. For other suites, the size depends on the hash algorithm associated with the suite. For example for SHA-512 it would be 64 bytes.

(In terms of side-channel attack prevention, it would be preferable to include a non-public quantity into the data being signed since this reduces the scope of attack from a passive to an active one, with the attacker needing to initiate their own handshakes in order to carry out their attack. However no shared secret value has been established at this point so only public data can be signed).

The choice of key sizes is something that will never get any consensus because there are so many different worldviews involved.

TLS-LTS makes only general recommendations on best practices and leaves the choice of which key sizes are appropriate to implementers and policy makers:

o Implementations SHOULD choose public-key algorithm key sizes that are appropriate for the situation, weighted by the value of the information being protected, the probability of attack and capabilities of the attacker(s), any relevant security policies, and the ability of the system running the TLS implementation to deal with the computational load of large keys. For example a SCADA system being used to switch a ventilator on and off doesn't require anywhere near the keysize-based security of a system used to transfer classified data.

One way to avoid having to use very large public keys is to switch the keys periodically. For example for DH keys this can be done by regenerating DH parameters in a background thread and rolling them over from time to time. If this isn't possible, an alternative option is to pre-generate a selection of DH parameters and choose one set at random for each new handshake, or again roll them over from time to time from the pre-generated selection, so that an attacker has to attack multiple sets of parameters rather than just one.

3.2. Message Formats

TLS-LTS sends the full set of DH parameters, X9.42/FIPS 186 style, not p and g only, PKCS #3 style. This allows verification of the DH parameters, which the current format doesn't allow:

o TLS-LTS implementations MUST send the DH domain parameters as { p, q, g } rather than { p, g }. This makes the ServerDHParams field:

```
struct {
    opaque dh_p<1..2^16-1>;
    opaque dh_q<1..2^16-1>;
    opaque dh_g<1..2^16-1>;
    opaque dh_Ys<1..2^16-1>;
    } ServerDHParams;     /* Ephemeral DH parameters */
```

Note that this uses the standard DLP parameter order { p, q, g }, not the erroneous { p, g, q } order from the X9.42 DH specification.

o The domain parameters MUST either be compared for equivalence to a set of known-good parameters provided by an appropriate standards body or they MUST be verified as specified in FIPS 186 [9]. Examples of the former may be found in RFC 3526 [22].

Note that while other sources of DH parameters exist, these should be treated with a great deal of caution. For example RFC 5114 [23] provides no source for the values used, leading to suspicions that they may be trapdoored, and RFC 7919 [24] mandates fallback to RSA if the one specific DH parameter set for each key size specified in the standard isn't automatically chosen by both client and server.

Industry standards bodies may consider restricting domain parameters to only allow known-good values such as those referenced in the above standard, or ones generated by the standards body. This makes checking easier, but has the downside that restricting the choice to a small set of values makes them a more tempting target for well-resourced attackers. In addition it requires that the values be carefully generated, and the generation process well-documented, to produce a so-called NUMS (Nothing Up My Sleeve) number that avoids any suspicion of it having undesirable hidden properties (the standard mentioned above, RFC 5114 [23], does not contain NUMS values).

In any case signing the Client/Server Hello messages and the use of Extended Master Secret makes active attacks that manipulate the domain parameters on the fly far more difficult than they would be for standard TLS.

3.3. Miscellaneous

TLS-LTS drops the need to send the current time in the random data, which serves no obvious purpose and leaks the client/server's time to attackers:

- o TLS-LTS implementations SHOULD NOT include the time in the Client/Server Hello random data. The data SHOULD consist entirely of random bytes.

  [Note: A proposed downgrade-attack prevention mechanism may make use of these bytes, see section 3.6].

TLS-LTS drops compression and rehandshake, which have led to a number of attacks:

- o TLS-LTS implementations MUST NOT implement compression or rehandshake.

3.4. Implementation Issues

TLS-LTS requires that RSA signature verification be done as encode-then-compare, which fixes all known padding-manipulation issues:

- TLS-LTS implementations MUST verify RSA signatures by using encode-then-compare as described in PKCS #1 [10], meaning that they encode the expected signature result and perform a constant-time compare against the recovered signature data.

The constant-time compare isn't strictly necessary for security in this case, but it's generally good hygiene and is explicitly required when comparing secret data values:

- All operations on crypto- or security-related values SHOULD be performed in a manner that's as timing-independent as possible. For example compares of MAC values such as those used in the Finished message and data packets SHOULD be performed using a constant-time memcmp() or equivalent so as not to leak timing data to an attacker.

TLS-LTS recommends that implementations take measures to protect against side-channel attacks:

- Implementations SHOULD take steps to protect against timing attacks, for example by using constant-time implementations of algorithms and by using blinding for non-randomised algorithms like RSA.

- Implementations SHOULD take steps to protect against fault attacks, in particular for the extremely brittle ECC algorithms whose typical failure mode if a fault occurs is to leak the private key. One simple countermeasure is to use the public key to verify any signatures generated before they are sent over the wire.

Authentication mechanisms for protocols run over TLS typically have separate authentication procedures for the tunnelled protocol and the encapsulating TLS session. The leads to an issue known as the channel binding problem in which the tunnelled protocol isn't tied to the encapsulating TLS session and can be manipulated by an attacker once it passes the TLS endpoint. Channel binding ties the cryptographic protection offered by TLS to the protocol that's being run over the TLS tunnel:

- Implementations that require authentication for protocols run over TLS SHOULD consider using channel bindings to tie the application-level protocol to the TLS session, specifically the tls_unique binding, which makes use of the contents of the first TLS Finished message sent in an exchange to bind to the tunneled application-level protocol [3].

The original description of the tls_unique binding contains a long note detailing problems that arise due to rehandshake issues and how to deal with them. Since TLS-LTS doesn't allow rehandshakes, these problems don't exist, so no special handling is required.

The TLS protocol has historically and somewhat arbitrarily been described as a state machine, which has led to numerous implementation flaws when state transitions weren't very carefully considered and enforced [20]. A safer and more logical means of representing the protocol is as a ladder diagram, which hardcodes the transitions into the diagram and removes the need to juggle a large amount of state:

o Implementations SHOULD consider representing/implementing the protocol as a ladder diagram rather than a state machine, since the state-diagram form has led to numerous implementation errors in the past which are avoided through the use of the ladder diagram form.

TLS-LTS mandates the use of cipher suites that provide so-called Perfect Forward Secrecy (PFS), in which an attacker can't record sessions and decrypt them at a later date. The PFS property is however impacted by the TLS session cache and session tickets, which allow an attacker to decrypt old sessions. The session cache is relatively short-term and only allows decryption while a session is held in the cache, but the use of long-term keys in combination with session tickets means that an attacker can decrypt any session used with that key, defeating PFS:

o Implementations SHOULD consider the impact of using session caches and session tickets on PFS. Security issues in this area can be mitigated by using short session cache expiry times, and avoiding session tickets or changing the key used to encrypt them periodically.

Another form of cacheing that can affect security is the reuse of the supposedly-ephemeral value y = g^x mod p. Instead of computing a fresh value for each session, some servers compute the y value once and then reuse it across multiple TLS sessions. If this is done then an attacker can compute the discrete log value from one TLS session and reuse it to attack later sessions:

o Implementations SHOULD consider the impact of reusing the y = g^x mod p value across multiple TLS sessions, and avoid this reuse if possible. Where the reuse of y is unavoidable, it SHOULD be refreshed as often as is feasible. One way to do this is to compute it as a background task so that a fresh value is available when required.

TLS-LTS protects its handshake by including cryptographic integrity checks of preceding messages in subsequent messages, defeating attacks that build on the ability to manipulate handshake messages to compromise security. What's authenticated at various stages is a log of preceding messages in the exchange. The simplest way to implement this, if the underlying API supports it, is to keep a running hash of all messages (which will be required for the final Finished computation) and peel off a copy of the current hash state to generate the hash value required at various stages during the handshake. If only the traditional { Begin, [Update, Update, ...], Final } hash API interface is available then several parallel chains of hashing will need to be run in order to terminate the hashing at different points during the handshake.

3.5. Use of TLS Extensions

TLS-LTS is inspired by Grigg's Law that "there is only one mode and that is secure". Because it mandates the use of known-good mechanisms, much of the signalling and negotiation that's required in standard TLS to reach the same state becomes redundant. In particular, TLS-LTS removes the need to use the following extensions:

- o The signature_algorithms extension, since the use of SHA-256 with RSA or ECDSA is implicit in TLS-LTS.

- o The elliptic_curves and ec_point_formats extensions, since the use of P256 with uncompressed points is implicit in TLS-LTS.

- o The universally-ignored requirement that all certificates provided by the server must be signed by the algorithm(s) specified in the signature_algorithms extension is removed both implicitly by not sending the extension and explicitly by removing this requirement.

- o The encrypt_then_mac extension, since the use of encrypt-then-MAC is implicit in TLS-LTS.

- o The extended_master_secret extension, since the use of extended Master Secret is implicit in TLS-LTS.

TLS-LTS implementations that wish to communicate only with other TLS-LTS implementations MAY omit these extensions, with the presence of tls_lts implying signature_algorithms = RSA/ECDSA + SHA-256, elliptic_curves = P256, ec_point_formats = uncompressed, encrypt_then_mac = TRUE, and extended_master_secret = TRUE. Implementations that wish to communicate with legacy implementations and wish to use the capabilities described by the extensions outside of TLS-LTS MUST include these extensions in their Client Hello.

Conversely, although all of the above extensions are implied by TLS-LTS, if a client requests TLS-LTS in its Client Hello then it doesn't expect to see them returned in the Server Hello if TLS-LTS is indicated. The handling of extensions during the Client/Server Hello exchange is therefore as follows:

Client Hello	Server Chooses	Server Hello
TLS-LTS	TLS-LTS	TLS-LTS
TLS-LTS, EMS/EncThenMAC/...	TLS-LTS	TLS-LTS
TLS-LTS, EMS/EncThenMAC/...	EMS/EncThenMAC/...	EMS/EncThenMAC/...

Table 1: Use of TLS-LTS Extensions

TLS-LTS capabilities are indicated purely by the presence of the tls_lts extension, not the plethora of other extensions that it's comprised of. This allows an implementation that needs to be backwards-compatible with legacy implementations to specify individual options for use with non-TLS-LTS implementations via a range of extensions, and specify the use of TLS-LTS via the tls_lts extension.

3.6. Downgrade Attack Prevention

The use of the TLS-LTS improvements relies on an attacker not being able to delete the TLS-LTS extension from the Client/Server Hello messages. This is achieved through the SCSV [7] signalling mechanism.

[If SCSV is used then insert required boilerplate here, however this will also require banning weak cipher suites like export ones, which is a bit interesting in that it'll required banning something that in theory has already been extinct for 15 years. A better option is to refer to Karthikeyan Bhargavan's rather clever idea on anti-downgrade signalling, which is a more reliable mechanism than SCSV].

3.7. Rationale

This section addresses the question of why this document specifies a long-term support profile for TLS 1.2 rather than going to TLS 1.3. The reason for this is twofold. Firstly, we know that TLS, which has become more or less the universal substrate for secure communications

over the Internet, has extremely long deployment times. Much of this information is anecdotal (although there are a large number of these anecdotes), however one survey carried out in 2015 and 2016 illustrates the scope of the problem. This study found that the most frequently-encountered protocol (in terms of use in observed Internet connections) was the fifteen-year-old TLS 1.0, with the next most common, TLS 1.2, lagging well behind [21]. This was on the public Internet, in the non-public arena (where much of the anecdotal evidence comes from, since it's not possible to perform a public scan) the most common protocol appears to be TLS 1.0, with significant numbers of systems still using the twenty-year-old SSLv3.

Given that TLS 1.3 is almost a completely new protocol compared to the incremental changes from SSLv3 to TLS 1.2, and that the most widely-encountered protocol version from that branch is more than fifteen years old, it's likely that TLS 1.3 deployment outside of constantly-updated web browsers may take one to two decades, or may never happen at all given that a move to TLS 1.2 is an incremental change from TLS 1.0 while TLS 1.3 requires the implementation of a new protocol. This document takes the position that if a protocol from the TLS 1.0 - 1.2 branch will remain in use for decades to come, it should be the best form of TLS 1.2 available.

The second reason why this document exists has already been mentioned above, that while TLS 1.0 - 1.2 are all from the same fairly similar family, TLS 1.3 is an almost entirely new protocol. As such, it rolls back the 20 years of experience that we have with all the things that can go wrong in TLS and starts again from scratch with a new protocol based on bleeding-edge/experimental ideas, mechanisms, and algorithms. When SSLv3 was introduced, it used ideas that were 10-20 years old (DH, RSA, DES, and so on were all long-established algorithms, only SHA-1 was relatively new). These were mature algorithms with large amounts of research published on them, and yet we're still fixing issues with them 20 years later (the DH algorithm was published in 1976, SSLv3 dates from 1996, and the latest DH issue, Logjam, dates from 2015). With TLS 1.3 we currently have zero implementation and deployment experience, which means that we're likely to have another 10-20 years of patching holes and fixing protocol and implementation problems ahead of us.

It's for this reason that this specification uses the decades of experience we have with SSL and TLS and the huge deployed base of TLS 1.0 - 1.2 implementations to update TLS 1.2 into a known-good form that leverages about 15 years of analysis and 20 years of implementation experience, rather than betting on what's almost an entirely new protocol based on bleeding-edge/experimental ideas, mechanisms, and algorithms, and hoping that it can be deployed in less than a decade- or multi-decade time frame. The intent is to

create a long-term stable protocol specification that can be deployed once as a minor update to existing TLS implementations, not deployed as a new from-scratch implementation and then patched, updated, and fixed constantly for the lifetime of the equipment that it's used with.

4. Security Considerations

This document defines a minimal, known-good subset of TLS 1.2 that attempts to address all known weaknesses in the protocol, mostly by simply removing known-insecure mechanisms but also by updating the ones that remain to take advantage of many years of security research and implementation experience. As an example of its efficacy, several attacks on standard TLS that emerged after this document was first published were countered by the mechanisms specified here, with no updates or changes to TLS-LTS implementations being necessary to deal with them.

5. IANA Considerations

IANA has added the extension code point TBD (0xTBD) for the tls_lts extension to the TLS ExtensionType values registry as specified in TLS [2].

6. Acknowledgements

The author would like to thank the members of the TLS mailing list and contributors from various embedded systems vendors for their feedback on this document.

7. References

7.1. Normative References

[1] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[2] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008.

[3] Altman, J., Williams, N., and L. Zhu, "Channel Bindings for TLS", RFC 5929, July 2010.

[4] Eastlake 3rd, D., "Transport Layer Security (TLS) Extensions", RFC 6066, January 2011.

[5] Rescorla, E. and N. Modadugu, "Datagram Transport Layer Security Version 1.2", RFC 6347, January 2012.

[6] Gutmann, P., "Encrypt-then-MAC for Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)", RFC 7366, September 2014.

[7] Moeller, B. and A. Langley, "TLS Fallback Signaling Cipher Suite Value (SCSV) for Preventing Protocol Downgrade Attacks", RFC 7507, April 2015.

[8] Bhargavan, K., Delignat-Lavaud, A., Pironti, A., Langley, A., and M. Ray, "Transport Layer Security (TLS) Session Hash and Extended Master Secret Extension", RFC 7627, September 2015.

[9] "Digital Signature Standard (DSS)", FIPS 186, July 2013.

[10] Jonsson, J. and B. Kaliski, "Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1", RFC 3447, February 2003.

7.2. Informative References

[11] Bhargavan, K., Fournet, C., Kohlweiss, M., Pironti, A., Strub, P., and S. Zanella-Beguelin, "Proving the TLS handshake secure (as is)", Springer-Verlag LNCS 8617, August 2014.

[12] Brzuska, C., Fischlin, M., Smart, N., Warinschi, B., and S. Williams, "Less is more: relaxed yet compatible security notions for key exchange", IACR ePrint archive 2012/242, April 2012.

[13] Dowling, B. and D. Stebila, "Modelling ciphersuite and version negotiation in the TLS protocol", Springer-Verlag LNCS 9144, June 2015.

[14] Firing, T., "Analysis of the Transport Layer Security protocol", June 2010.

[15] Gajek, S., Manulis, M., Pereira, O., Sadeghi, A., and J. Schwenk, "Universally Composable Security Analysis of TLS", Springer-Verlag LNCS 5324, November 2008.

[16] Jager, T., Kohlar, F., Schaege, S., and J. Schwenk, "On the security of TLS-DHE in the standard model", Springer-Verlag LNCS 7417, August 2012.

[17] Giesen, F., Kohlar, F., and D. Stebila, "On the security of TLS renegotiation", ACM CCS 2013, November 2013.

[18] Meyer, C. and J. Schwenk, "Lessons Learned From Previous SSL/TLS Attacks - A Brief Chronology Of Attacks And Weaknesses", Cryptology ePrint Archive 2013/049, January 2013.

[19] Krawczyk, H., Paterson, K., and H. Wee, "On the security of the TLS protocol", Springer-Verlag LNCS 8042, August 2013.

[20] Beurdouche, B., Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Kohlweiss, M., Pironti, A., Strub, P., and J. Zinzindohoue, "A Messy State of the Union: Taming the Composite State Machines of TLS", IEEE Symposium on Security and Privacy 2015, May 2015.

[21] Holz, R., Amann, J., Mehani, O., Wachs, M., and M. Kaafar, "TLS in the Wild: An Internet-Wide Analysis of TLS-Based Protocols for Electronic Communication", Network and Distributed System Security Symposium 2016, February 2016.

[22] Kivinen, T. and M. Kojo, "More Modular Exponential (MODP) Diffie-Hellman groups for Internet Key Exchange (IKE)", RFC 3526, May 2003.

[23] Lepinski, M. and S. Kent, "Additional Diffie-Hellman Groups for Use with IETF Standards", RFC 5114, January 2008.

[24] Gillmor, D., "Negotiated Finite Field Diffie-Hellman Ephemeral Parameters for Transport Layer Security (TLS)", RFC 7919, August 2016.

Author's Address

Peter Gutmann
University of Auckland
Department of Computer Science
University of Auckland
New Zealand

Email: pgut001@cs.auckland.ac.nz