Secure Password-Based Cipher Suite for TLS*

Peter Buhler[1] Thomas Eirich[1] Michael Steiner[2†] Michael Waidner[1]

[1] IBM Zürich Research Laboratory
CH-8803 Rüschlikon, Switzerland
{bup,eir,wmi}@zurich.ibm.com

[2] Universität des Saarlandes
D-66123 Saarbrücken, Germany
steiner@cs.uni-sb.de

Abstract

SSL is the de facto standard today for securing end-to-end transport. While the protocol seems rather secure there are a number of risks which lurk in its use, e.g., in web banking. We motivate the use of password-based key exchange protocols by showing how they overcome some of these problems. We propose the integration of such a protocol (DH-EKE) in the TLS protocol, the standardization of SSL by IETF. The resulting protocol provides secure mutual authentication and key establishment over an insecure channel. It does not have to resort to a PKI or keys and certificates stored on the users computer. Additionaly the integration in TLS is as minimal and non-intrusive as possible. As a side-effect we also improve DH-EKE to provide semantic security assuming the hardness of the Decisional Diffie-Hellman Problem.

1. Introduction

The *Secure Socket Layer (SSL)* protocol [15] is todays de facto standard for securing end-to-end transport over the Internet. In particular the presence of SSL in virtually all web browsers led to a widespread use of SSL, also in application requiring a high level of security such as home banking. While early versions of SSL contained a number of flaws and shortcomings the analysis of the latest version 3.0 shows only a few minor anomalies [35, 28]. SSL was further refined in the *Transport Layer Security (TLS)* protocol [13], the standardization effort of the *Internet Engineering Task Force (IETF)*, and seems to provide a reasonable level of security[1].

Currently all standard methods for authentication in TLS rely on a *public-key infrastructure (PKI)*. While this is suitable for many cases it might not suit environments where the infrastructures is "light-weight" (e.g., diskless workstations, user-to-user authentication), times when a system has to be bootstrapped from scratch, or situations when user mobility is required.

Furthermore current cipher suites also have their own risks, most prominently illustrated in following example. Over the last years many banks have built home banking applications for the web. For their security they rely mainly on the web browser and the SSL built in. As reliably issuing client certificates is rather complicated most of these applications use SSL for server authentication only. They set up a secure channel from the browser to the server and then ask the user to authenticate herself by typing her password in a simple web form. However, in such a setup the user authentication is not tied to the channel and in fact the security cannot be guaranteed if the user does not explicitly verify the connection before entering her password. Verification does not only mean to verify that there is a secure connection by observing that the lock gets golden and closed. It also requires that the user makes sure that the connection is to the right entity by checking that the certificate identifies the right bank and is issued by an appropriate certification authority (CA). This is a non-trivial task, e.g., Netscape contains by default over 70 root certificates varying from high to virtually no assurance. To counter possible attacks[2] the user might even have to verify the fingerprint of the CA itself. If the user fails to do that properly she is highly susceptible to a man-in-the-middle attack. This seems to put too high a burden on the average user.

*Appeared in Proceedings of Network and Distributed Systems Security Symposium (NDSS 2000), San Diego, California, February 2000. ©Internet Society.

†On leave of absence from the IBM Zürich Research Laboratory

[1]Note that the risk of the recent, very practical attack of Bleichenbacher [12] on the RSA-based cipher suites can be reduced through careful implementations. The adoption of Version 2.0 of PKCS #1 [21] and its new encoding method `EME-OAEP` based on work by Bellare and Rogaway [6] should dwarf that attack completely.

However, as illustrated by [32], the attack clearly demonstrates the importance of careful protocol design when treating crypto systems as black boxes. Either we have to carefully specify the requirements on the black boxes or we have to use the strongest available crypto systems when instantiating the black boxes. A failure to do so clearly led to this attack.

[2]See Section 6.7.3 for further discussion on problems lurking in the certificate management of web browsers.

Use of one-time-use transaction authorization numbers (TAN) only marginally improves this situation. Using client side certificates helps but besides complicating the setup it requires proper protection of the client's keys, which is difficult given the (in)security of common operating systems available today.

Above problems related to a PKI apply mainly to multi-purpose applications such as a web browser. Multiple different trust domains (CAs) co-exist and the application cannot know and enforce which policies are appropriate to particular contexts. However, these issues are not intrinsic problems of SSL and will not appear with the password-based protocols presented in the following, regardless of the applications.

The proposition to add cipher suites based on Kerberos [26, 22] would get rid of the requirement of a PKI. Unfortunately Kerberos is not really light-weight (e.g., there is no real structural difference from a PKI) and, even more importantly, it is vulnerable to *dictionary attacks* when weak passwords are used [37, 9, 29, 16]. Given the human nature, this cannot be excluded. Proactive password checking [11] can help only to a limited degree. On the one hand the choice of passwords has to be easy and unrestricted enough to make it possible for humans to remember (and prevent them from writing down the password!). This limits the possible entropy in such passwords. On the other hand computing power grows still drastically and makes dictionary attacks possible on larger and larger classes of passwords.

Luckily, there is a class of authenticated key-exchange protocols based on human-memorizeable weak passwords which are resistant to (off-line) dictionary attacks. They do not have to be backed by any infrastructure such as a PKI. Assuming proper handling of online dictionary attacks which are usually detectable, these systems are at least as secure as other systems based on strong public or shared keys. To substantiate the "at least" we note that in reality most of these other systems rely also on passwords somewhere at the user end: the key ring in PGP [38] and keys for browsers are password encrypted and are vulnerable even to undetectable off-line dictionary attacks once stolen! The security of password-based key exchange protocols relies only on two assumptions: The integrity of the underlying machine, and the availability of a reasonably good source of randomness. But this is in essence the minimal requirement for any other system as well.

Therefore it seems quite useful to enrich the set of current TLS cipher suites with a password-based protocol and reduce the risks explained above. In the following we describe the integration of an improved version of the *Diffie-Hellman Encrypted Key Exchange (DH-EKE)* [8] into TLS. The new cipher suite provides mutual authentication and key establishment with *perfect forward secrecy* over an insecure channel and limits the damage in case an attacker gains access to the server's databases. The integration into TLS is as non-intrusive as possible and with some optimizations retains the 4-round handshake overhead of TLS.

The structure of the remainder of the paper is as follows. In Section 2 we give a brief overview of the flows of TLS and we state some criteria for the integration of a new cipher suite. In Section 3 we introduce a new cipher suite based on DH-EKE. Before presenting the details of the protocol in Section 5 we have to dig into some cryptographic preliminaries in Section 4. We then give rationales for our choices in Section 6 and conclude in Section 7.

2. TLS

2.1. Overview

TLS is composed of two layers: the *TLS Record Protocol* and the *TLS Handshake Protocol*. The Record Protocol encapsulates higher level protocols (such as HTTP [10]) and cares about the reliability, confidentiality and compression of the messages exchanged over the connection. The TLS Handshake Protocol is responsible for setting up the secure channel between server and client and provides the keys and algorithm information to the Record Protocol. The changes required in our integration of password based protocols are not relevant to the Record Protocol. Therefore we will omit further discussion of it.

Figure 1 gives an overview of the flows of the Handshake Protocol. The main purpose of the first message, `ClientHello`, is to send a random challenge to guarantee freshness and to tell the server which cryptographic algorithms are supported by the client.

Based on this proposal the server will pick a set of algorithms, the *cipher suite*. As an illustrative example let us assume the choice was the cipher suite `TLS_DHE_DSS_WITH_DES_CBC_SHA`. This means that the session key will be based on a *Diffie-Hellman key exchange* [14] using ephemeral parameters, DSA is the signature algorithm used and the security on the record layer will be based on DES in CBC mode and SHA-1. The chosen cipher suite is stored in the `ServerHello` message together with another random challenge to help assuring the server of the freshness of the protocol run. If server authentication is required the server sends the own certificate in `Certificate`. Depending on the chosen cipher suite the server also sends the `Server-KeyExchange` message. This message contains keying data required for the key exchange. In our example it would hold the server's ephemeral Diffie-Hellman

Client		**Server**
propose acceptable cipher suites		
	`ClientHello` →	
		choose cipher suite and prepare key exchange parameters
	← `ServerHello`, `Certificate*`, `ServerKeyExchange*`, `CertificateRequest*`, `ServerHelloDone`	
verify parameters and prepare own key exchange and verification parameters		
	`Certificate*`, `ClientKeyExchange`, `CertificateVerify*`, `[ChangeCipherSpec]`, `Finished` →	
		complete authentication of client (if possible)
	← `[ChangeCipherSpec]`, `Finished`	
complete authentication of server		
	← *ApplicationData* →	

Figure 1. Overview of TLS flows. (Situation-dependent messages are flagged with a "*").

half-key h^x signed with the server's signing key. Furthermore a list of accepted certificate types and CAs is send as part of the `CertificateRequest` if client authentication is required. Finally the server marks the end of the turn by sending the `ServerHelloDone`.

In the next step the client verifies the received data. The client prepares the own input to the key generation, e.g., the Diffie-Hellman half-key h^y, stores them in `ClientKeyExchange` and derives from this and the server's input contained in `ServerKeyExchange` the *premaster secret*. In our example this would mean computing the Diffie-Hellman key h^{xy}. The premaster secret is then hashed together with two previously exchanged challenges to form the *master secret*. The master secret is, as its name implies, the main session key and all cryptographic keys used further on are derived from this master secret. The client sends now the `Client-KeyExchange` and, if required by the cipher suite, also `CertificateVerify` and `Certificate` for client authentication to the server. The client issues then a `ChangeCipherSpec` to the Record Protocol instructing it to use keys and algorithms newly negotiated. Finally the client sends the `Finished` message, a message authentication code (MAC) on the previously sent messages using a newly derived key.

The server derives the premaster and master secret from the data contained in `ClientKeyExchange` and the own inputs. Verifying the `Finished` message will assure the server now of the freshness of the request and of the authenticity of the client if client authentication was enabled. The server then sends a similar `Finished` message to the client. This allows the client to verify the authenticity of the server and the freshness of the keys used.

2.2. Adding New Cipher Suites

Before presenting the integration of DH-EKE let us look first at the requirements and constraints of the integration of a new cipher suite in general. The TLS specifications [13] do not mention explicitly what is allowed or what is not for the integration of a new cipher suite. But it is obvious that such an integration should be as least intrusive as possible. Looking closer at the defined data structures reveals that the ideal places to adjust TLS for new cipher suites are the `ServerKeyExchange` and `ClientKeyExchange` messages. They are already variant records and can be rather transparently extended with a new element. We can also approach the problem from the other side and look on the hard constraints. It is quite clear that for compatibility reasons we should not alter messages which are sent before an agreement on a cipher suite has been reached. This means in particular that we should refrain from modifying `ClientHello`. As we will see later this unfortunately has important consequences. Further desirable properties are to minimize setup time by keeping the number of flows and the computation costs low.

3. DH-EKE/TLS: An overview

3.1. Exponential key exchange

In 1992 Steve Bellovin and Michael Merritt, of Bell Labs, published a family of methods called *Encrypted Key Exchange (EKE)* [8]. These methods provide key exchange with mutual authentication based on weak secrets (e.g., passwords). They are very careful designed to prevent the leakage of weak secrets and withstand dictionary attacks which are most often possible on protocols involving secrets with low entropy.

The simplest and most elegant of the methods is DH-EKE. In DH-EKE a weak secret P is used to encrypt the elements of a Diffie-Hellman key exchange, i.e., $g^x \pmod p$ and $g^y \pmod p$. The protocol is shown in Figure 2.

The session key that Alice and Bob compute is $g^{xy} \pmod p$. The key is cryptographically strong if x and y are cryptographically strong random numbers, regardless of the strength of the password.

Various ways exist to optimize the number of flows as well as the number of encryptions. However, these optimizations, as design of the encryption process and the choice of the algebraic group, has to be done very carefully to prevent various attacks [8, 34, 19, 30].

The cipher suite presented in the following will be based on the optimized protocol presented by Steiner et. al. [34]. As a second line of defense we also integrate B-EKE [20], a technique to reduce the risks caused by loss or theft of user databases from the server's machine.

3.2. Integration of DH-EKE in TLS

Let us now turn our attention to the concrete integration of DH-EKE. Figure 3 gives an overview of the flows assuming DH-EKE/TLS was among the proposed cipher suites in the `ClientHello` and got selected by the server. The arguments of messages contain the security critical protocol information in abstracted and simplified form where $\mathcal{H}_i$'s are different pseudo-random functions and g is a key derivation function.

The protocol looks very similar to the case given as an example in the previous section with two main differences: The client's h^y is encrypted with the password instead of being accompanied by a signature and the sending of the client's and server's `Finished` messages are swapped. The first difference helps to authenticate each other based on the common knowledge of the password. The second change is due to the problems of transferring

Alice		**Bob**
(weak secret P) choose $x \in_R \mathbb{Z}_q$		(weak secret$P$)
	$\xrightarrow{E_P(g^x \pmod p)}$	
		choose $y \in_R \mathbb{Z}_q$, compute $K = g^{xy} \pmod p$, choose challenge C_1
	$\xleftarrow{E_P(g^x \pmod p)), E_K(C_1)}$	
compute $K = g^{xy} \pmod p$, chooses challenge C_2		
	$\xrightarrow{E_K(C_1, C_2)}$	
		verify response
	$\xleftarrow{E_K(C_2)}$	
verify response		

Figure 2. DH-EKE

identity information and the subtle issues of dictionary attacks. Note that it is of paramount importance that the client does not use any key derived from the premaster secret pms before the client has successfully received and verified the server's `Finished` message. However, the changes in the overall protocol state-machine should be kept to a minimum. Note also that there is no penalty in communication delay due to the fifth flow: The client can start to send application data immediately after sending the `Finished` message.

Our integration is actually optimal in respect to the number of flows as we will show in Section 6.1.

The server's `Certificate` and `CertificateRequest` messages and the client's `Certificate` and `CertificateVerify` messages are omitted in Figure 3 for obvious reasons (no PKI). Note that these messages are specified as optional in the TLS protocol; therefore, omitting them is permissible.

4. Cryptographic Preliminaries

4.1. Multiplicative Group $\mathbb{Z}_p^*$

The cryptographic operations in DH-EKE are performed in the multiplicative group $\mathbb{Z}_p^*$ with p prime and q a large prime divisor of $\phi(p) = (p-1)$. Let $n = \lceil \log_2 p \rceil$ and $m = \lceil \log_2 q \rceil$ be the number of bits of p and q respectively. Typical values are 768, 1024 or 2048 for n and 160 or 320 for m. We also need an (arbitrary) primitive root g of the group $\mathbb{Z}_p^*$ and a generator of the (unique) subgroup G of order q computed as $h = g^{(p-1)/q}$. For algorithms on finding primitive roots and efficiently computing group operations in multiplicative groups we refer the reader to other sources, e.g., [27].

4.2. Group Verification

The group parameters p, q, g and h should preferably be fixed at system startup. Otherwise, they may be chosen by the server and passed to the client in `ServerKeyExchange`. In this case, the client has to verify them. Of particular importance is to make sure that p and q are prime and n and m are sufficiently large. As in the ephemeral case the parameter might be chosen by an adversary, it is not possible to use optimization techniques which drastically reduce the number of Miller-Rabin tests such as the one described in Table 4.3 of [27]. Instead we can only rely on $1/4^t$ as the upper bound of the probability that a candidate is prime after t Miller-Rabin tests: Therefore at least 40-50 tests per prime, i.e., q and p, are required to make the probability negligible that we accept a composite number falsely as prime. The test bases a should be chosen at random and not be predictable by the adversary.

These tests are rather expensive, in particular if we assume light-weight clients. A more efficient way of verification is to let the server send further verification information together with the group parameters. This can help proving the correctness of the parameters more efficiently. The approach chosen here is quite simple. To show the randomness of the prime selection the server sends together with the prime also a pre-image of that number taken from a one-way function. This would require only a small change in the server's prime

Client		**Server**
(password pwd)		($v = h^{\mathcal{H}_1(pwd, client\_name)}$, $auth\_key = \mathcal{H}_2(pwd, client\_name)$)
	`ClientHello` $\longrightarrow$	choose $x, x' \in_R \mathbb{Z}_q$
	$\longleftarrow$ `ServerHello`, `ServerKeyExchange(`$h^x, h^{x'}$`)`, `ServerHelloDone`	
derive $auth\_key$ from pwd and choose $y \in_R \mathbb{Z}_q$	`ClientKeyExchange(`$E_{auth\_key}(h^y)$`)` $\longrightarrow$	calculate premaster secret as $pms = \mathcal{H}_3((h^y)^x, v^{x'})$
	$\longleftarrow$ `[ChangeCipherSpec]`, `Finished(`$\mathcal{H}_4(g(pms), h^x, \ldots)$`)`	
calculate premaster secret as $pms = \mathcal{H}_3((h^x)^y, (h^{x'})^{\mathcal{H}_1(pwd, client\_name)})$ and accept session if `Finished` can be verified	`[ChangeCipherSpec]`, `Finished(`$\mathcal{H}_5(g(pms), h^x, \ldots)$`)` $\longrightarrow$	Accept session if `Finished` can be verified
	$\longleftarrow$ $ApplicationData$ $\longrightarrow$	

Figure 3. Overview of (simplified) flows of DH-EKE/TLS.

generation process but it should restrict an adversary from choosing weak or special primes. Therefore this randomization allows the use of the optimization techniques described in [27] and the number of Miller-Rabin tests on the client side can be reduced down to at most 5 tests with the given range of n as defined above.

4.3. Encryption using Weak Secrets

In addition to exponentiations in multiplicative groups we also need a shared-key encryption function $E_P(z)$ to transport the client's Diffie-Hellman half-key. On input P, a weak secret, and z, an element of $\mathbb{Z}_p^*$ we perform the following steps:

Key Derivation We derive the encryption key k as $\mathcal{H}_0(P, salt)$. The input parameters are the weak secret P and the concatenation of the two challenges found in `ClientHello` and `ServerHello` as $salt$. The function $\mathcal{H}_0(z, w)$ is computed as the first $keylength$ bits of $PRF(z, \text{``}Password - derived\ key\text{''}, w)$. The *pseudo-random function* PRF is defined in Section 5 of [13], takes as input a secret, an identifying label and a seed and produces an output of arbitrary length. $keylength$ equals to 8 for DES, 16 for 3DES, IDEA and RC4-128, 5 for RC2. For DES (3DES) the key should be considered as a 64-bit (192-bit) encoding of a 56-bit (168-bit) DES key with parity bits ignored.

Expansion To prevent dictionary attacks on the encrypted elements (see Section 6.3 for more details) we uniformly expand the element z from an n-bit number to a $n + \alpha$-bit number. We form a block b of $n + \alpha$ bits as follows:

$\alpha = 50$;
$n = \lceil \log_2 p \rceil$;
$\alpha' := \lfloor (2^{\alpha+n})/p \rfloor$;
$c \in_{\mathcal{R}} \{0, \ldots, \alpha' - 1\}$;
$b := z + cp$; {Note that this calculation is in $\mathbb{Z}$ and not in $\mathbb{Z}_p^*$, e.g., no reduction!}

Padding If the block length len of the encryption scheme does not divide $\log b$ then b is padded with $(len - (\log b \pmod{len}))$ random bits to form b'.

Encryption The b' is encrypted using the derived key k. The used shared-key cipher is defined by the agreed cipher suite. It is encoded in the cipher suite name after `TLS_DH_EKE_` and is basically the agreed session encryption cipher if existing (e.g., we would encrypt with RC4/128 if the agreed upon cipher suite is `TLS_DH_EKE_RC4_128_WITH_RC4_128_MD5`). See Figure 5 for the proposed cipher suites. For block ciphers in chaining mode the IV will be set to all 0.

Decryption An encrypted value is decrypted using the key k derived as defined above in "Key derivation". From the decrypted text, the random padding (if existing) is removed and the resulting value is then reduced $\pmod p$ to undo the expansion.

5. Protocol Flow Processing

Before describing the processing of the flows let us first look at the setup of the system. The client first chooses a password pwd. Then the client derives a *password authentication key* $auth\_key = \mathcal{H}_2(pwd, client\_name)$ which is later used to authenticate the session key. Finally the client computes the *password verifier* $v = h^{\mathcal{H}_1(pwd, client\_name)}$. The value v will allow the server later to verify that the user really knows the password in a way that the server does not has to get or store the password itself; this way we can limit the damage if the server is corrupted or the database is leaked. The functions $\mathcal{H}_1(z, w)$ and $\mathcal{H}_2(z, w)$ are computed as the first m bits of $PRF(z, \text{``}password\ verifier\text{''}, w)$ and $PRF(z, \text{``}password\ authentication\ key\text{''}, w)$ respectively. v and $auth\_key$ are then sent securely to the server and stored together with the client's name in the server's user database.

In the following we assume the client proposes in `ClientHello` some of the DH-EKE cipher suites and the server agrees on one of them. We also omit all standard processing as defined in TLS and refer the reader to [13].

1. Client → Server The client prepares the `ClientHello` as usual.

2. Server → Client The server chooses $x \in_R \mathbb{Z}_q$ and computes $h^x \pmod p$. Additionally the server also chooses $x' \in_R \mathbb{Z}_q$ and computes $h^{x'} \pmod p$.

The server completes the `ServerDHEKEParams` field in `ServerKeyExchange` with h^x and $h^{x'}$. If the server's group parameters are not a priori fixed, the server also prepares `ServerDHParamsProof` to allow optimized parameter verification for the client as described in Section 4.2. The server sends the `ServerHello`, `ServerKeyExchange` and `ServerHelloDone` messages to the client.

3. Client → Server The Client verifies the parameters of the group: if they are not installed and well-defined, the client performs the tests as outlined in Section 4.2.

The client then verifies that the h^x and $h^{x'}$ contained in `ServerKeyExchange` are of the right size and order ($(h^x)^q \pmod p) = 1 \wedge h^x \neq 1$). The client aborts if above conditions are not fulfilled.

The client (software) ask the user for her password and derives the authentication key as $auth\_key = \mathcal{H}_2(pwd, , client\_name)$. The client chooses $y \in_R \mathbb{Z}_p$ and computes $g^y \pmod p$. Then the client encrypts g^y as defined in Section 4.3, enters the resulting value $E_{auth\_key}(g^y)$ as well as the user's identity in the `ClientDHEKEParams` field of the `ClientKeyExchange` message and sends the message to the server.

4. Server → Client The server extracts the identity of the client from the `ClientKeyExchange` message and retrieves the client's password context. The server verifies that the account is not locked and decrypts the client's half-key g^y as defined in Section 4.3 using the authentication key $auth\_key$ stored in the context.

The server computes the premaster secret pms as $\mathcal{H}_3((g^y)^{(x(p-1/q))}, v^{x'})$ (with $\mathcal{H}_3(z, w)$ defined as $PRF(z, \text{``}DH\ premaster\ secret'', w)$ and generates the server's `Finished` message as defined in the TLS specifications, e.g., the function $\mathcal{H}_4$ in Figure 3 is a MAC over all previously sent handshake messages. The server performs a `Change-CipherSpec` and sends the `Finished` message to the client.

5. Client → Server The client computes $\mathcal{H}_3((h^x)^{(y \pmod q)}, (h^{x'})^{\mathcal{H}_1(pwd, client\_name)})$ to get the premaster secret pms and verifies the server's `Finished` message. If the verification fails, the client aborts.

The client generates the `Finished` message ($\mathcal{H}_5$ in Figure 3 is again a MAC over all previously exchanged handshake messages), proceeds with the `ChangeCipherSpec` and sends the `Finished` message to the server. Note that contrary to the standard case the client can start to send data immediately after the `Finished` (and basically retains the original 4-flow handshake overhead).

6. Server → Client The server verifies the client's `Finished` message. If the verification fails, the server aborts, increments the 'potential online attack' counter in the client's password context and locks the account if the 'potential online attack' counter reaches a threshold (a reasonable number for the threshold might be 5. Note that more elaborate policies with exponential retry delays might be used in addition). If the verification is OK, the 'potential online attack' counter is updated (exact procedure depends on local policy: possibilities are setting it to 0, decrementing it by 1, etc.).

Note: To reduce risk of password exposure implementors are advised to throw away (zero out) all traces from the password and all used critical random values (e.g., the Diffie-Hellman parameters x, y, h^x and the premaster secret) as soon as possible.

6. Rationales and Explanations

The above proposed protocol takes into account all known attacks ([8, 34, 19, 30]). In addition, it provides for semantic security and at the same time it improves performance. Find in the following a few more detailed rationales and explanation of certain choices taken during the protocol design.

6.1. Flows

The client cannot carry its identity information in the `ClientHello` message[3]. Therefore the server cannot encrypt its value as in the optimal protocol [34]. However, to prevent dictionary attacks, the party which encrypts with the password should be very careful. In no case should the client use derived session keys before it knows that the server confirmed knowledge of the password explicitly by proving knowledge of the key or implicitly by encrypting its own half-key with the password.

This rules out using the standard TLS flows. The client, which is the first party to be able to encrypt with the password, cannot send the `Finished` before getting a "proof of knowledge of password" from the server. Any other approach would have increased the number of flows and would have deviated even further from the standard TLS messages.

If we exclude altering or misusing `ClientHello` we can actually extend this reasoning and show that it is completely impossible to build a secure mutually authenticated key-exchange in four flows which relies only on weak secrets. The server, not knowing the client's identity after the first flow, can not produce any sort of implicit or explicit proof of knowledge of the password in the second flow. Consequently the client cannot send any key confirmation in the third flow and the only way to complete client authentication is to send such a mes-

[3] At least if we like to stay compatible to standard TLS and do not resort to changes of `ClientHello` or unacceptable ad-hoc measures such as encoding the identity in the nonce-field of the `ClientHello`

sage in an additional fifth flow. Henceforth our protocol is optimal in number of flows.

6.2. Subgroup Confinement

An attacker might send elements of small order to either reduce the possible key-space for impersonations or attacks on the password. (e.g., if the attacker sends 1 instead of h^x then the key will be 1 regardless of what the other (honest) party chooses as random exponent!). This attack can be prevented when the receiver of the unencrypted half-key h^x verifies the order of the element. However, note that verification of the order for decrypted value is not necessary: An attacker can either guess a password and encrypt an element of small order or send an arbitrary random value. In the unlikely case that the password guess was correct then obviously there was no point of encrypting an element of small order in the first place. Otherwise, given the pseudo-random nature of the encryption function, a decryption will yield a random element regardless if the attacker has chosen a wrong password or an arbitrary value. But if $\phi(p)$ has large prime factors it is highly unlikely that a randomly selected value decrypts to an element of small order.

If we choose $\mathbb{Z}_p^*$ such that $\phi(p)$ would contain only prime factors of sufficient size (e.g., they are all of at least m bits) we could improve the check for elements of small order even further. In such groups it is sufficient to test that $x^2 \pmod p \neq 1$ to verify that x has larger order. Although this seems to be sufficient, the overall security of this method needs further study [23] and it is not immediately clear if we can retain semantic security.

6.3. Encryption

The security in the encryption process $E_P(z)$ described in Section 4.3 relies on two properties to prevent an adversary from verifying candidate passwords using an encrypted element z: First, the encryption function should produce cipher-texts containing no redundancy and the range of the encryption function has to be the same regardless of the chosen key. Second, the plaintext has to be close to uniformly distributed. The first condition is fulfilled by stream-ciphers and by block-ciphers performing a permutation on the input block. The second condition is fulfilled by encrypting a (random) element of the group and not of the subgroup (see also Section 6.4), by random padding (for block ciphers) and proper expansion.

The expansion is necessary to prevent dictionary attacks on the encrypted elements. If we omit expansion an attacker has a probability of $(1 - r)$ (where r is the ratio of size of the valid range over the size of the possible range, i.e., $(p - 1)/2^n$) to reject a wrong password guess by decrypting observed encryptions with the guess and finding an element in the illegal range $\{0, p, p + 1, \ldots, 2^n - 1\}$. Taking into account that the attacker can observe t runs of the protocol the probability of successfully rejecting a password guess becomes $(1 - r^t)$ and approaches 1 very quickly, even for small t. Note that $0.5 < r < 1$ always holds by definition of n.

In average if we expand with 1 bit we decrease the proportion of the invalid range in respect to the complete range by half. Therefore we also reduce the chances of an attacker by half. Let us define t_{max} as an upper bound for the number of protocol runs with a given password and 2^{-k} as the maximally tolerable probability that an attacker can reject an (incorrect) password guess after having observed some (i.e, at most t_{max}) protocol runs. Then the number of required expansion bits is $\alpha = -\log_2(1 - (1 - 2^{-k})^{1/t_{max}}) \approx k + log_2(t_{max})$. If we take as 30 for k and 2^{20} for t_{max} we get the $\alpha = 50$ required in Section 4.3. Note that with these values we have a wide safety margin in all practical applications: On the one hand no user will enter his password and connect to the server more than 2^{20} times and the server which tracks failed connection request in his 'potential online attack' counter will foil all attempts to get more samples with active attacks. On the other hand already $k = 1$ means that an attacker reduces the number of possible passwords by half which would be acceptable already in most cases.

The key derivation mechanism reuses basic building blocks of TLS and approximates also the upcoming PKCS #5 Version 2.0 [31]. The salt guarantees that for each protocol run we get independent keys to address concerns about interactions between multiple usage of the same key.

6.4. Choice Of Group

As mentioned above there should be no structure in decryption otherwise we might be open to attacks. In previous papers on DH-EKE it was commonly assumed that this means that we cannot operate in a (more efficient) cyclic subgroup G but have to work in the whole group $\mathbb{Z}_p^*$ (e.g., we need a primitive root as base for the exponentiations). Encrypting elements of the subgroup would lead to following attack: The attacker chooses a candidate password, decrypts with it an encrypted half-key g^y observed on the wire and rejects the password if the decrypted element is not an element of the subgroup. If the password guess was wrong the likelihood that the decrypted element is not an element of the subgroup is high and therefore the attack will be quite effective.

However, we run into a problem if we like to achieve semantic security in the sense of the indistinguishability of a valid session key from a random key. If we don't resort to random oracles [5], the weakest cryptographic assumption we can rely on is the hardness of the Decisional Diffie-Hellman Problem (DDH). It is also obvious that this cannot be done in $\mathbb{Z}_p^*$[4] but only in a prime order subgroup of $\mathbb{Z}_p^*$. This means that the proof of the security as found in the appendix of [34] does not work for DH-EKE as originally proposed in [8]. To make the proof work we have to modify the protocol such that they operate in a subgroup.

Luckily, the first observation that we cannot operate in subgroups is not completely correct: While it is true that we cannot encrypt elements of the subgroup with the password it nevertheless does not prevent us from computing in the subgroup. The trick is simple. Instead of encrypting an element of the subgroup we send randomly one of the $(p-1)/q$)-th roots contained in group. Assuming uniformly and randomly chosen exponents and roots we will get a uniform distribution over $\mathbb{Z}_p^*$. Even better, as the sender actually chooses the element there is no need to compute roots and randomly select one of them: Just selecting a random element in $\mathbb{Z}_p^*$ and letting the receiver construct the group element by raising it to the power of $(p-1)/q$ is sufficient (Note that following equality holds $g^{y((p-1)/q)} = (g^{(p-1)/q})^y = h^y = h^{y \pmod q}$). Therefore not only can we retain semantic security but we also improve efficiency as now only two of the four exponentiations require long exponents. Further performance improvements can be obtained if we choose g and/or h to be small. This will speed up exponentiations without any loss of security.

Instead of $\mathbb{Z}_p^*$ we could also choose the alternative multiplicative group $GF(2^m)^*$. Computation is rather efficient and additionally the encryption problem discussed in Section 6.3 disappear. The cardinality of $GF(2^m)^*$ is $\phi(2^m) = 2^{m-1}$ and this can be efficiently mapped to $m-1$ bits. This means that with proper encoding a decryption of a random value and a random password guess will always produce a legal value and cannot serve as basis for dictionary attacks. However, further study is necessary to find concrete parameters and compare the security and performance with the solution for $\mathbb{Z}_p^*$.

6.5. Verifiable Parameter generation

The verification of ephemeral group parameter is based on heuristics. There still remains some degree of freedom for the opponent to find (pseudo) primes through pre-computational search. A safer alternative might be to use provable primes generated from Maurer's provable prime number generation [25]. The server generates p based on Maurer's algorithm. The primality of q can then be shown as part of the primality proof for p. One drawback of this approach is that messages get bigger and the code gets more complicated (the current approach can be built on components already existing most TLS toolkits). Additionally we can expect a considerable performance impact for this approach.

6.6. Reducing the Risk of Stolen Server Databases

As additional measure of precaution we also reduce the risks caused by loss or theft of user databases from the server's machine. In the original proposal by Bellovin and Merrit the server had to store the password. This meant that an attacker getting access to the server's database could masquerade as both client and server right away. Extensions to EKE such as A-EKE [7], B-EKE [20] or SRP [36] reduce the risk of stolen server databases to–unavoidable in such situations–dictionary attacks as only a (salted) hash of the password is stored. While we argue that dictionary attacks are always feasible and therefore the password will eventually be revealed such a second line of defense is nevertheless desirable. For this reason we used the idea of B-EKE in our protocol with the inclusion of v and $auth\_key$ and the computation of the premaster secret as the hash of the two DH-keys. Using the strong DH-key h^{xy} as key to the pseudo-random function should completely hide any information on the password, even if the premaster secret is available to an attacker. We consider the additional costs of the additional exponentiations (note that all are with small exponents) worthwhile but it would be straightforward to make the use of B-EKE optional and allow performance critical environments to trade the risk of stolen server databases with improved performance.

6.7. Why EKE?

We also investigated alternatives to EKE. While many of them do have various advantages over DH-EKE none could match DH-EKE with its minimal impact on TLS: Two additions in `ClientKeyExchange` and `ServerKeyExchange` and a minimal and unavoidable change in the protocol state machine (reversion of the two finished flows) seems to be the smallest change possible to integrate secure password based protocols. Find below some more detailed explanations why we rejected the other protocols.

6.7.1. SPEKE An alternative protocol is the Simple Password Encrypted Key Exchange (SPEKE) [19]. The protocol is also based on a Diffie-Hellman key exchange but instead of encrypting the half-keys with the password it uses the password to derive a generator for a large prime-order subgroup.

[4] The order of elements in $\mathbb{Z}_p^*$ leaks too much information. This leads easily to an algorithm which distinguishes with high probability between (g^x, g^y, g^{xy}) and a triple of random elements.

It has two main advantages over DH-EKE. On the one hand the problem due to non-uniform distribution of encrypted elements does not occur and on the other hand there is a possibility to improve performance by computing on elliptic curves.[5]

Unfortunately integrating SPEKE into TLS is not straightforward: As previously explained the `ClientHello` message cannot carry identity information and as identity of the peer to be known before anybody can start the protocol we require more radical changes in the flows, in particular it would require two more messages and/or changes in `Finished` messages.

6.7.2. SRP Yet another prominent proposal is the secure remote password protocol (SRP) [36]. While it seems the most efficient system which reduces also the risk when the server database is stolen it has similar problems with integration as SPEKE. The protocol cannot be started in flow 2 which means that the handshake would require an additional request response pair. Taking into account current network delays and performance of todays computers lead us to trade performance for reduced flows.

6.7.3. What about Protocols relying on Server Public Keys? The responder side in TLS is quite often a stand-alone server capable of keeping strong public key pairs. You might wonder if this cannot be exploited to achieve easier and more efficient protocols. Indeed, various protocols [18, 17] show how to do this in a provably secure and arguably simpler manner. While these protocols are definitely suitable in many applications there is one major drawback: The client has to get the proper public key of the server. One solution is to ask the user for confirmation of a fingerprint as suggested in [18]. While this is definitely preferable over fixing the public key in the software it is quite cumbersome for the user. You might argue now that current web-browsers already manage root-certificates and adding one more is not a big deal. While this is true there is the problem of key revocation. Additionally one should not ignore the fact that it is not too hard to trick ignorant users in installing bogus root keys to their key ring: Generate your own root CA, build a fancy web site and require https using certificates relying on your own root CA to access it. The likelihood that some user will install this key is rather large. Even worse you can tell who has installed your root CA certificate if you track user access to the site and the certificate and then you can target that user for a man-in-the-middle attack. In fact a similar man-in-the-middle attack has happened mid-1998 to a Dutch web banking site. As EKE-like protocols rely less on the user's awareness of the such involved risks they are clearly a more secure approach.

6.7.4. Others Further protocols we considered where the EKE variant due to Lucks [24] and protocols based on collisionful hash [1, 3]. However, none of their feature could outweigh the simplicity of the integration of DH-EKE in TLS.

7. Conclusion

We outlined a number of situations where the current cipher suites of TLS are not completely satisfactory, e.g. home banking over the web. Secure password based authenticated key-exchange protocols can improve the situation and can be integrated into TLS in an efficient and non-intrusive manner. We validated our approach by integrating the cipher suite into a in-house toolkit providing the complete SSL3.0 protocol suite. Due to our careful protocol design with a reliance on existing building blocks and the non-intrusive integration of the protocol flows we had to adapt the protocol engine only with few and small changes. Measurements of the performance showed that our cipher suite compares well with other cipher suites. DH-EKE outperformed comparable cipher suites providing mutual authentication and perfect forward secrecy by a factor of up to two (`SSL_DHE_DSS_WITH_DES_CBC_SHA`) and was only slightly slower than the commonly used cipher suite `SSL_RSA_WITH_RC4_128_SHA`.

In a modification to the original DH-EKE protocol we showed further that the session keys not only can but also should be computed in subgroups of prime order: We achieve better security and as a side-effect also improve the performance of DH-EKE. In line with the security analysis as found in the appendix of [34] we get reasonable assurance that the security of our protocols can be founded on the hardness of DDH. However, in the light of recent development in the formalization of the security of key agreement protocols [4, 33] it's an open question if the protocol could also be formally proven secure in these stronger and more rigid models.

Acknowledgments

We would like to thank Victor Shoup, Luke O'Connor and Birgit Pfitzmann for their invaluable discussions on cryptographic and algorithmic issues related to this document and N. Asokan and Ahmad-Reza Sadeghi for their detailed comments.

[5] Using elliptic curves for DH-EKE seems rather hard as we would have to bijectively map the elements on the curve onto a range of $\mathbb{Z}$.

References

[1] R. J. Anderson and T. M. A. Lomas. Fortifying key negotiation schemes with poorly chosen passwords. *Electronics Letters*, 30(13):1040–1041, June 1994.

[2] V. Ashby, editor. *1st ACM Conference on Computer and Communications Security*, Fairfax, Virginia, Nov. 1993. ACM Press.

[3] S. Bakhtiari, R. Safavi-Naini, and J. Pieprzyk. On password-based authenticated key exchange using collisionful hash functions. In *1st Australasian Conference on Information Security and Privacy (ACISP '96)*, number 1172 in Lecture Notes in Computer Science, pages 299–310. Springer-Verlag, Berlin Germany, 1996.

[4] M. Bellare, R. Canetti, and H. Krawczyk. A modular approach to the design and analysis of authentication and key exchange protocols. In *Proceedings of th 30th Annual Symposium on the Theory of Computing*, 1998.

[5] M. Bellare and P. Rogaway. Random oracles are practical: A pardigm for designing efficient protocols. In Ashby [2]. also appeared (in identical form) as IBM RC 19619 (87000) 6/22/94.

[6] M. Bellare and P. Rogaway. Optimal asymmetric encryption – how to encrypt with RSA. In I. Damgard, editor, *Advances in Cryptology – EUROCRYPT '94*, Lecture Notes in Computer Science, pages 92–111. Springer-Verlag, Berlin Germany, 1994. Final (revised) version appeared November 19, 1995.

[7] S. Bellovin and M. Merrit. Augmented encrypted key exchange: a password-based protocol secure against dictionary atttacks and password file compromise. In Ashby [2], pages 244–250.

[8] S. M. Bellovin and M. Merrit. Encrypted key exchange: Password-based protocols secure against dictionary attacks. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, May 1992.

[9] S. M. Bellovin and M. Merritt. Limitations of the Kerberos authentication system. In *USENIX Conference Proceedings*, pages 253–267, Dallas, TX, Winter 1991. USENIX. An earlier version of this paper was published in the October, 1990 issue of Computer Communications Review.

[10] T. Berners-Lee, R. T. Fielding, H. F. Nielsen, J. Gettys, and J. Mogul. Hypertext transfer protocol – HTTP/1.1. Internet Request for Comment RFC 2068, Jan. 1997.

[11] M. Bishop and D. V. Klein. Improving system security via proactive password checking. *Computers & Security*, 14(3):233–249, 1995.

[12] D. Bleichenbacher. Chosen ciphertext attacks against protocols based on the rsa encryption standard pkcs. In H. Krawczyk, editor, *Advances in Cryptology – CRYPTO '98*, number 1462 in Lecture Notes in Computer Science, pages 1–12. Springer-Verlag, Berlin Germany, Aug. 1998.

[13] T. Dierks and C. Allen. The TLS protocol version 1.0. Internet Request for Comment RFC 2246, Jan. 1999. Proposed Standard.

[14] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, Nov. 1976.

[15] A. O. Freier, P. Kariton, and P. C. Kocher. The SSL protocol: Version 3.0. Technical report, Internet Draft, 1996. Will be eventually replaced by TLS.

[16] L. Gong, M. Lomas, R. Needham, and J. Saltzer. Protecting poorly chosen secrets from guessing attacks. In *Proceedings of the Twelth ACM Symposium on Operating Systems Principles*. The Wigwam, Litchfield Park, Arizona, Dec. 1989. A revised journal version appeared as [17].

[17] L. Gong, M. Lomas, R. Needham, and J. Saltzer. Protecting poorly chosen secrets from guessing attacks. *IEEE Journal on Selected Areas in Communications*, 11(5):648–656, June 1993.

[18] S. Halevi and H. Krawczyk. Public-key cryptography and password protocols. In *5th ACM Conference on Computer and Communications Security*, San Francisco, California, Nov. 1998. ACM Press. Revised version available as Theory of Cryptography Library: Record 99-04.

[19] D. P. Jablon. Strong password-only authenticated key exchange. *Computer Communication Review*, 26(5):5–26, Sep 1996.

[20] D. P. Jablon. Extended password key exchange protocols immune to dictionary attack. In *Proceedings of the WET-ICE'97 Workshop on Enterprise Security*, Cambridge, MA, USA, June 1997.

[21] B. Kaliski and J. Staddon. PKCS #1: RSA cryptography specifications. Technical report, RSA Laboratories, Sept. 1998. Version 2.0. Published in October 1998 as Internet RFC 2437.

[22] J. T. Kohl and B. C. Neuman. The Kerberos network authentication service (V5). Internet Request for Comment RFC 1510, Project Athena, MIT, 1993.

[23] C. H. Lim and P. J. Lee. A key recovery attack on discrete log-based schemes using a prime order subgroup. In B. S. K. Jr., editor, *Advances in Cryptology – CRYPTO '97*, number 1294 in Lecture Notes in Computer Science, pages 249–263. Springer-Verlag, Berlin Germany, Aug. 1997.

[24] S. Lucks. Open key exchange: How to defeat dictionary attacks without encrypting public keys. In *Security Protocol Workshop'97*, Ecole Normale Suprieure, Paris, Apr. 1997.

[25] U. M. Maurer. Fast generation of prime numbers and secure public-key cryptographic parameters. *Journal of Cryptology*, 8(3):123–155, 1995.

[26] A. Medvinsky and M. Hur. Addition of kerberos cipher suites to transport layer security (TLS). Internet Draft, Aug. 1999. Expires January 22, 2000.

[27] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of applied cryptography*. CRC Press series on discrete mathematics and its applications. CRC Press, 1997. ISBN 0-8493-8523-7.

[28] J. Mitchell, V. Shmatikov, and U. Stern. Finite-state analysis of SSL 3.0. In *7th USENIX Security Symposium*, 1998.

[29] R. Morris and K. Thompson. Password security: A case history. *Commun. ACM*, 22(11), Nov. 1979.

[30] S. Patel. Number theoretic attacks on secure password schemes. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, Research in Security and Privacy, pages 236–247, Oakland, CA, May 1997. IEEE Computer Society Press.

[31] RSA. PKCS #5: Password-based encryption standard. Technical report, RSA Laboratories, Feb. 1999. Version 2.0; Third Draft.

[32] V. Shoup. Why chosen ciphertext security matters. Research Report RZ 3076 (#93122), IBM Research, Nov. 1998.

[33] V. Shoup. On formal models for secure key exchange. Research Report RZ 3120 (#93166), IBM Research, Apr. 1999.

[34] M. Steiner, G. Tsudik, and M. Waidner. Refinement and extension of *encrypted key exchange*. *ACM Operating Systems Review*, 29(3):22–30, July 1995.

[35] D. Wagner and B. Schneier. Analysis of the SSL 3.0 protocol. In *Second USENIX Workshop on Electronic Commerce*, pages 29–40, Oakland, California, Nov. 1996. USENIX.

[36] T. Wu. The secure remote password protocol. In *Symposium on Network and Distributed Systems Security (NDSS '98)*, pages 97–111, San Diego, California, Mar. 1998. Internet Society.

[37] T. Wu. A real-world analysis of kerberos password security. In *Symposium on Network and Distributed Systems Security (NDSS '99)*, San Diego, CA, Feb. 1999. Internet Society.

[38] P. R. Zimmermann. *The Official PGP User's Guide*. MIT Press, Cambridge, MA, USA, 1995. ISBN 0-262-74017-6.

Appendix: Data Structures and Definitions

Figure 4 defines the necessary additional data structures for the `ClientKeyExchange` and `ServerKeyExchange` messages. For a standardization of TLS extension one also would have to define the corresponding cipher suite codes. Figure 5 proposes possible cipher suites for the DH-EKE protocol but leaves for obvious reasons the codes blank.

```
struct {
    select (KeyExchangeAlgorithm) {
        case dh_eke:      /* new option */
            ServerDHEKEParams   params;
        case diffie_hellman:
            ServerDHParams   params;
            Signature   signed_params;
        case rsa:
            ServerRSAParams   params;
            Signature   signed_params;
        } ;
} ServerKeyExchange;

struct {
    ServerDHParams   key_params;
    ServerDHParams   verifier_params;
    ServerDHParamsProof   proof;    /* optional */
} ServerDHEKEParams;     /* new type */

struct {
    seed <0 .. 2^16 - 1>;
} ServerDHParamsProof;     /* new type */

struct {
    select (KeyExchangeAlgorithm) {
        case dh_eke:      /* new option */
            ClientDHEKEParams   params;
        case rsa:
            EncryptedPreMasterSecret;
        case diffie_hellman:
            ClientDiffieHellmanPublic;
    } exchange_keys;
} ClientKeyExchange;

struct {
    String   clientIdentity;
    EncryptedDHParams   params;
} ClientDHEKEParams;     /* new type */

struct {
    password-encrypted   dh_Xs<1 .. 2^16 - 1>;
} EncryptedDHParams;     /* new addition */
```

Figure 4. Adding DH-EKE/TLS to data structures of TLS.

```
CipherSuite   TLS_DH_EKE_DES_CBC_WITH_NULL_SHA                  = { , };
CipherSuite   TLS_DH_EKE_RC4_128_WITH_NULL_MD5                  = { , };
CipherSuite   TLS_DH_EKE_DES_CBC_WITH_DES_CBC_SHA               = { , };
CipherSuite   TLS_DH_EKE_3DES_EDE_CBC_WITH_3DES_EDE_CBC_SHA     = { , };
CipherSuite   TLS_DH_EKE_RC4_128_WITH_RC4_128_MD5               = { , };
CipherSuite   TLS_DH_EKE_IDEA_CBC_WITH_IDEA_CBC_SHA             = { , };
CipherSuite   TLS_DH_EKE_RC4_128_WITH_NULL_SHA                  = { , };
CipherSuite   TLS_DH_EKE_DES_CBC_WITH_NULL_MD5                  = { , };
CipherSuite   TLS_DH_EKE_DES_CBC_WITH_DES_CBC_MD5               = { , };
CipherSuite   TLS_DH_EKE_3DES_EDE_CBC_WITH_3DES_EDE_CBC_MD5     = { , };
CipherSuite   TLS_DH_EKE_RC4_128_WITH_RC4_128_SHA               = { , };
CipherSuite   TLS_DH_EKE_IDEA_CBC_WITH_IDEA_CBC_MD5             = { , };
```

Figure 5. Proposed Cipher Suites for DH-EKE/TLS.