SSL/TLS Session-Aware User Authentication Revisited

Rolf Oppliger[1], Ralf Hauser[2], and David Basin[3]

[1] eSECURITY Technologies Rolf Oppliger
Beethovenstrasse 10, CH-3073 Gümligen, Switzerland
Phone/Fax: +41 79 654 8437
E-mail: rolf.oppliger@esecurity.ch
[2] PrivaSphere AG
Fichtenstrasse 61, CH-8032 Zürich, Switzerland
Phone: +41 43 299 5588, Fax: +41 44 382 7762
E-mail: hauser@privasphere.com
[3] Department of Computer Science, ETH Zurich
Haldeneggsteig 4, CH-8092 Zürich, Switzerland
Phone: +41 44 632 7245, Fax: +41 44 632 1172
E-Mail: basin@inf.ethz.ch

Abstract. Man-in-the-middle (MITM) attacks pose a serious threat to SSL/TLS-based e-commerce applications. In [OHB06], we introduced the notion of SSL/TLS session-aware user authentication to protect SSL/TLS-based e-commerce applications against MITM attacks and we proposed an implementation based on impersonal authentication tokens. In this paper, we present a number of extensions of the basic idea. These include multi-institution tokens, possibilities for changing the PIN, and, most importantly, different ways of making several popular and widely deployed user authentication systems SSL/TLS session-aware.

Keywords. Electronic commerce, security, phishing, man-in-the-middle attack, SSL/TLS protocol, SSL/TLS session-aware user authentication

1 Introduction

Man-in-the-middle (MITM) attacks [Bur02] pose a serious threat to SSL/TLS-based e-commerce applications, such as Internet banking. In [OHB06], we introduced the notion of SSL/TLS session-aware user authentication (TLS-SA) to protect SSL/TLS-based e-commerce applications against MITM attacks. The main idea behind TLS-SA is to make user authentication depend not only on the user's credentials, such as his password or personal identification number, but also on state information related to the SSL/TLS session in which the credentials are transferred to the server. The rationale is that the server should have the possibility to determine whether the SSL/TLS session in which it receives the credentials is the same as the one the user employed when he sent out the credentials in the first place. If the two sessions are the same, then the session is directly established between the user and the server, whereas if they are different, then a MITM attack is likely taking place.

Using TLS-SA, the user authenticates himself by providing a user authentication code (UAC) that depends on both his credentials and the SSL/TLS session, in particular, on information from the SSL/TLS session state. A MITM may still be able to acquire the UAC, but he can no longer misuse it simply by retransmitting it. The key point is that the UAC is bound to a particular SSL/TLS session and if the UAC is submitted on another SSL/TLS session, then the server can recognize this and drop the session.

As outlined in [OHB06], there are a number of possibilities to implement TLS-SA and we distinguish between hardware-based and software-based implementations. The former employ hardware tokens, which may be physically connected to the client.[4] The latter utilize some form of software tokens. Independent of whether a token is a hard-token or a soft-token, it can be personal or impersonal, and it may support a cryptographic token interface standard, such as the public key cryptography standard (PKCS) #11 [RSA04] or Microsoft's cryptographic application programming interface (CAPI).

In the remainder of this paper, we present a number of extensions and variations of TLS-SA. To make the paper self-contained, we summarize our original token-based proposal in Section 2. In Sections 3–5, we address multi-institution tokens, possibilities for changing the personal identification number (PIN), and, most importantly, different ways of making several popular and widely deployed user authentication systems SSL/TLS session-aware. In Section 6, we draw conclusions and provide an outlook. For the purpose of this paper, we assume that the reader is familiar with the SSL/TLS protocol [DR06].

2 Token-Based Approach

The implementation of TLS-SA originally proposed in [OHB06] is based on impersonal authentication tokens. The following entities play a role:

- A user U;
- An impersonal authentication token T with a (small) display;
- A client (i.e., browser) C that is used by U to access an SSL/TLS-based application;
- An SSL/TLS-enabled server S that hosts the application. Note that we do not differentiate between S and the application it hosts. Conceptually, one may think of S as a dedicated server, i.e., a server that exclusively hosts only the application under consideration.

The user U is identified with ID_U and holds PIN_U, a secret PIN that U shares with S. T is identified by a publicly-known serial number SN_T that may,

[4] We say that a token is *physically connected*, or just *connected*, if there is a direct communication path in place between the token and the client system. This includes, for example, galvanic connections, as well as connections based on wireless technologies, such as Bluetooth or infrared.

for example, be imprinted on the back side of the token. Furthermore, T is equipped with both a public key pair[5] (k, k^{-1}) and a secret token key K_T that is shared with S. The keys k and k^{-1} are the same for all tokens (which is why the tokens are impersonal), whereas K_T is unique and specific to T. Note, however, that K_T is not specific to the user, and hence the tokens need not be personalized. This is the main advantage of using impersonal tokens in the first place. K_T can be generated randomly, or pseudorandomly, using a master key MK, which is typically held exclusively by S:[6]

$$K_T = E_{MK}(SN_T)$$

In order to access S, U directs C to S. C and S then try to establish an SSL/TLS session using the SSL/TLS handshake protocol. As part of this protocol, S authenticates itself using a public key certificate (note that this certificate is seldom verified and validated by the user). S is configured in a way that it always requires certificate-based client authentication by sending out a CertificateRequest message to C. When C receives this message, it knows that it must authenticate itself by returning a Certificate and a properly signed CertificateVerify message to S. The CertificateVerify message comprises a digitally signed hash value of all messages previously exchanged during the execution of the SSL/TLS handshake protocol (this hash value is further referred to as $Hash$). Part of these messages is the server's Certificate message, which comprises the server's public key certificate and hence the server's public key, included in this certificate. Consequently, the CertificateVerify message is logically bound to the server's public key. The digital signature is generated by T using its private key k^{-1}. Since T is an impersonal token, the CertificateVerify message neither authenticates the token nor the client. It only ensures that C uses a token to establish an SSL/TLS session with S and that the token can access $Hash$. Alternatively speaking, the token represents a trusted observer for $Hash$ and for enforcing a proper execution of the TLS-SA.

In addition to providing a properly signed CertificateVerify message to S, the token T also renders a shortened version of

$$N_T = E_{K_T}(Hash) \tag{1}$$

on its display, for example, in decimal notation. In this case, E represents an encryption function that is keyed with K_T. Alternatively, N_T may also represent a message authentication code (MAC) computed with a keyed one-way hash function, where K_T is again the key.[7] For example, the HMAC construction

[5] It is possible to have the token only hold the private key.

[6] In the first case (where K_T is generated randomly), all token keys must be stored on the server S. In the second case (where K_T is derived from SN_T), however, there is no need to centrally store all token keys on S. Instead, K_T can be generated dynamically from SN_T and MK.

[7] In some circumstances, the use of a MAC is advantageous because keyed one-way hash functions are generally more efficient than symmetric encryption systems and because there are usually no regulations restricting the use of (keyed) one-way hash functions, in contrast to the use of encryption systems.

[KBC97] can be used to generate

$$N_T = HMAC_{K_T}(Hash). \tag{2}$$

In either case, N_T can be shortened to the length of PIN_U (e.g., 8 decimal digits) by truncating it. Care must be taken so that collisions occur sufficiently seldom. Note, however, that an adversary would have to know K_T to actively look for colliding values of $Hash$. Anyway, N_T must then be combined with PIN_U to generate a UAC that is valid for exactly one SSL/TLS session initiated by U. If f represents a function that combines N_T and PIN_U in some appropriate way, then the UAC can be expressed as

$$UAC = f(N_T, PIN_U)\,. \tag{3}$$

In general, there are many ways to define an appropriate function f. One possibility, adopted from [MT93], is digit-wise addition, modulo 10. In this case, the UAC is the digit-wise sum, modulo 10, of N_T and PIN_U, which is a function simple enough for users to compute themselves, in their heads. Another possibility is to use PIN_U to select specific digits of N_T. This construction is, for example, employed by PINsafe of Swivel.[8] If the token comprises a keypad for entering PIN_U, then the token can also be used to compute f. Of course, in this case, f can then be more complex.

After a server-authenticated SSL/TLS session is successfully established between C and S, S authenticates U by asking him to provide ID_U, SN_T, and a valid UAC for the SSL/TLS session in current use. Note that the user need not enter SN_T if this value is included in the public key certificate for T's private key k^{-1}. In this case, S can retrieve SN_T from the certificate, but the certificate is then token-specific. Alternatively, one could also provide for the user to temporarily register with a specific token. In this case, S can retrieve ID_U from its registration database and set it as a default value in the user authentication process. Note, however, that the binding between SN_T and ID_U is still weak. On the server side, S can verify the UAC because it knows f and PIN_U, and because it can reconstruct N_T (since it knows $Hash$ and the master key MK that is used to dynamically generate K_T). There is nothing fundamentally different if the server knows only a one-way hash value of PIN_U (instead of PIN_U). This is a well-known security mechanism to protect PINs (or passwords) from malicious system administrators or users that have gained access to the file that comprises the PINs [MT79]. In either case, the server authentication module must have access to the SSL/TLS session cache in order to retrieve $Hash$.

Note that it is technically feasible to transfer the UAC as part of the SSL/TLS CertificateVerify message so that the user authentication can be handled entirely by the token. This would free the user from entering anything into a Web form. There are at least three possibilities here.

1. T can digitally sign both the $Hash$ value and the UAC.

[8] In Swivel's terminology, PIN_U refers to the user PIN, N_T refers to the security string, and UAC refers to the one-time code.

2. T can digitally sign a keyed hash value (instead of the hash value $Hash$), where the UAC represents the key, $Hash$ represents the argument, and the HMAC construction is used to key the hash function.
3. T can only send the keyed hash value (instead of the digital signature). This is similar to the second possibility; the only difference is that the keyed hash value is not digitally signed. Consequently, there is no need to have a private signing key on the token.

All of these possibilities require changes to the server side of the SSL/TLS handshake protocol and the way the SSL/TLS CertificateVerify message is used in the protocol. However, this is less disadvantageous than it appears at first sight because the browser (and all other application clients layered on top of SSL/TLS) remains unaffected and hence need not be altered.

3 Multi-institution Tokens

Multi-institution tokens are briefly mentioned in [OHB06]. As the name suggests, the idea is to allow multiple institutions to use the same token T for user authentication. The rationale behind multi-institution tokens is an economic one: it may be cost effective for multiple institutions to share a single token.

A simple and straightforward possibility for implementing multi-institution tokens is to replace the master key MK with a set of institution-specific master keys MK_I, one for each institution I. The token key is also then replaced with a set of institution-specific token keys K_{IT} that can be generated pseudorandomly by

$$K_{IT} = E_{MK_I}(SN_T).$$

Furthermore, each such key can be used to generate institution-specific values for

$$N_{IT} = E_{K_{IT}}(Hash)$$

and

$$UAC = f(N_{IT}, PIN_{IU}).$$

The personal identification code PIN_{IU} is used to authenticate the user U to the institution I, i.e., PIN_{IU} is user-specific and also institution-specific. The resulting UAC can be employed by the user U to authenticate to (the server of) the institution I.

There is another possibility for implementing multi-institution tokens. This possibility does not require the token to store K_{IT} for each institution I. Instead, the token only stores a master key MK_T that is shared with an authentication server (AS). If the token T is used to authenticate user U to institution I, then T randomly generates a nonce N_T and generates the following pair of UACs:

$$\begin{aligned} UAC_I &= f'(N_T, PIN_{IU}) \\ UAC_{AS} &= E_{MK_T}(N_T) \end{aligned}$$

The pair of UACs is then submitted to I. I, in turn, forwards UAC_{AS} to AS for decryption and AS sends back N_T to I. Here, all communications between I and AS must take place over a secure channel, e.g., over a SSL/TLS session. Finally, I can use N_T to verify PIN_{IU}. In this scheme, the function f' must be designed in such a way that it is computationally infeasible to recover PIN_{IU} from UAC_I and UAC_{AS}.

Both approaches to implementing multi-institution tokens are useful in a practical setting. If the set of institutions is more-or-less static, then the first possibility is advantageous since it does not require an authentication server. If, however, the set of institutions changes often, then the second possibility seems more appropriate.

4 Changing the PIN

The security of a token-based implementation of TLS-SA largely depends on the fact that users never enter their PIN into the client system, for example, in a Web form. Instead, they either use their PIN to compute the UAC in their head or they directly enter their PIN in the authentication token in use. If one weakens this constraint, then users are vulnerable to a doppelganger window attack [JM05]. In this attack, the adversary pops up a fake window to request user credentials. Since users cannot typically distinguish between original and fake windows, they are likely to enter their credentials into any window that asks for them. As of this writing, there is no technology that fully protects against this type of attack—this includes, for example, Delayed Password Disclosure (DPD) proposed in [JM05].

Ideally users are taught to never to enter their PIN into a client system. But this then complicates changing their PIN. Normally, one would implement a Web form in which a user can change his PIN interactively. Since we discourage, or even disallow, users from entering PINs into Web forms, we must provide other means to allow PIN changes. In either case, there must be some mechanism in place that allows a user to signal to the server that he wants to change his PIN and to protect the new PIN PIN_U^{new} with the old PIN PIN_U^{old}.

Let us assume that the user has authenticated himself using TLS-SA and that he has signaled to the server that he wants to change his PIN (using, for example, a Web form, or another mechanism for other application protocols layered on top of SSL/TLS). The server can then establish an auxiliary SSL/TLS session and send back to the browser a Web form in which the user is requested to enter a PIN change code (PCC). Again, the token displays N_T for the auxiliary SSL/TLS session and the user (or token) computes a PCC (instead of a UAC) as

$$PCC = f'(N_T, PIN_U^{old}, PIN_U^{new}).$$

Here, f' represents an arbitrary (but appropriately chosen) function that allows the server to recover PIN_U^{new} from the PCC. This excludes, for example, the use of hash functions. In the reference example of [OHB06], f represents the digit-wise addition modulo 10 and hence f' can be defined as

$$f'(N_T, PIN_U^{old}, PIN_U^{new}) = f(f(N_T, PIN_U^{old}), PIN_U^{new}).$$

Let, for example, $N_T = 123$, $PIN_U^{old} = 345$, and $PIN_U^{new} = 781$. In this case, $f(N_T, PIN_U^{old}) = 468$ and $f(f(N_T, PIN_U^{old}), PIN_U^{new}) = 149$. Consequently, the server can retrieve PIN_U^{new} from the PCC 149 submitted by the user. Note that the PCC can also be transferred as part of the SSL/TLS CertificateVerify message as discussed at the end of Section 2 (again, this requires the server to properly interpret this SSL/TLS protocol message). Also note that the PCC is sent over an SSL/TLS session. Since the SSL/TLS protocol protects the integrity of all messages (by sending a MAC at the end of each SSL/TLS record), it is infeasible for an adversary to modify the PCC, e.g., by flipping bits.

5 Making User Authentication Systems SSL/TLS Session-aware

There are many user authentication systems that can be employed in an SSL/TLS setting and almost all of them are susceptible to MITM attacks. In the remainder of this paper, we elaborate on how to make some popular and widely deployed user authentication systems SSL/TLS session-aware in a way that provides protection against MITM attacks. We distinguish between challenge-response (C/R) and one-time password (OTP) systems.

5.1 C/R Systems

A C/R system is an authentication system in which the entity that is authenticated (typically the client) is provided with a challenge for which it must return an appropriate response to the entity that is authenticating it (typically the server). C/R systems are often implemented in hardware and the corresponding (hardware) tokens may be connected to the client systems using some cryptographic token interface standard, such as PKCS #11 or CAPI.

On the conceptual level, there is a simple and straightforward possibility to make a C/R system SSL/TLS session-aware: instead of having the server provide a challenge to the client, the client and the server simultaneously employ N_T as a challenge. Note that N_T is cryptographically protected using the token key K_T as a shared secret. The formula (3) can be used to compute the UAC that then represents the response. We distinguish between two cases, depending on whether or not the token is connected to the client system.

1. If the token is connected, then it is straightforward to make the user authentication SSL/TLS session-aware. In this case, $Hash$ is sent to the token and the token can use it to compute N_T according to the formula (1) or (2). N_T can either be displayed so that the user can compute the UAC from it

or the user must additionally input PIN_U, so that the token can compute the UAC (this is the preferred choice). In either case, this protects against visual spoofing of the locally calculated challenge by a MITM and has the advantage that human copying errors can be excluded. The response can be copied by the user from a device display as is common practice. Alternatively, the response can also be directly returned by the device interface inside a pseudo-signature that will be put into the CertificateVerify message by the authenticating client.

2. If the token is not connected, then the situation is more involved. In this case, there is no direct communication path between the client and the token and hence there must be another possibility to communicate SSL/TLS state information from the browser to the token. We sketch one such possibility below.

The main advantages of the first case, where the token is connected, are that the user interaction is simple and that one can use N_T and possibly the UAC in its entire length and thereby provide better security. The main disadvantage is the necessity of having a free port available to connect the device to and to install a token driver on the client system.

The main advantage of the second case, where tokens are not connected, is that there is no need for a physical token interface, and hence, a token driver need not be installed. The main disadvantage is that one must have another possibility to communicate SSL/TLS state information from the browser to the token. One possibility is to employ the user as an active communication medium: the browser displays[9] $Hash$ and the user enters the displayed value into the token. This possibility has the disadvantage that $Hash$ is quite long (i.e., 36 bytes), and that it is unacceptable for the user to enter it into the token. So we need a way of shortening $Hash$ to only a few bits. This step is tricky. If we work with a deterministically shortened version of $Hash$, then we are vulnerable to a specific MITM attack. The MITM can wait for a first SSL/TLS session to be established by the client system. He can then set up a second SSL/TLS session to the origin server and modify the ServerHello message to be returned in the first session in a way that the compressed or truncated $Hash$ value matches the $Hash$ value of the second session (more specifically, the MITM looks for a second-preimage of the $Hash$ value for the shortening function in use). Since the challenge is determined by this value, the MITM can simply forward the user's response to authenticate to the origin server.

Due to the feasibility of this attack, one must be very careful to properly use the $Hash$ value. Instead of working with the entire $Hash$ value, for example, the browser can pseudorandomly select and work with only a subset of its bits to form the challenge. More specifically, the browser can pseudorandomly select a few position indices and bit sequences that begin at these positions from $Hash$.

[9] A problem here is how to make browsers display the needed information without having to alter them. In the short term, one may employ browser extensions, such as plugins. In the long term, however, it should not be necessary to extend the browser and the browser should have the possibility to access the information natively.

The values can be concatenated to form a challenge and thereby to add the requirement that the MITM needs to contact the server to validate the guesses for the pre-image or compressed hash respectively. The number and lengths of the bit sequences depend on the maximum length of the challenge (and it is a challenging engineering task to find optimal solutions). Anyway, the challenge can be displayed by the browser and the user can enter it into his token. The token can then generate an appropriate response for the challenge. If the token implements a block cipher and holds a token key K_T, then this key can be used to encrypt the challenge according to

$$Response = E_{K_T}(Challenge||Padding). \tag{4}$$

Note that *Challenge* is typically shorter than the block length of the block cipher and hence one can introduce some padding. Also note that the block length is important because one must avoid the case in which the output of the block cipher is too long (since the user must manually enter this value into a Web form).

In a typical setting, *Challenge* may comprise 4 to 8 decimal characters, whereas *Response* may comprise up to 13 alphanumerical characters (e.g., uppercase letters and decimal characters). If $l_{Challenge}$ is the maximum length of the challenge (in decimal digits), then one requires

$$b = \lfloor \log_2(10^{l_{Challenge}} - 1) \rfloor$$

bits to represent the challenge. For $l_{Challenge} = 4$, this corresponds to $b = 13$, and for $l_{Challenge} = 8$, this corresponds to $b = 26$. So if the challenge is 8 decimal digits long, then one can employ 26 bits to encode the challenge.

According to equation (4), the response of the C/R system refers to one ciphertext block. This basically means that the length of the response is equal to the block length of the symmetric encryption system in use. For DES and 3DES, for example, this is 64 bits. If we employ the Base-32 encoding scheme [Jos03] that comprises 32 characters encoded in 5 bits each, then we need $\lceil 64/5 \rceil = 13$ alphanumeric characters to encode the response. If, however, we employ the Base-64 encoding scheme [Jos03] that comprises 64 characters encoded in 6 bits each, then we can reduce the length of the response to the more realistic size of $\lceil 64/6 \rceil = 11$ characters.

Figure 1 illustrates the situation for $l_{Challenge} = 8$ and $b = 26$ with two 8-bit position indices (i.e., r_1 and r_2) and two corresponding 5-bit sequences starting at these positions. In this case, $2 \cdot 5 = 10$ out of 26 bits are taken to form the subset of the *Hash* value's bits. The bit utilization rate (BUR) is $10/26 \approx 0.38$.

As illustrated in Figure 2, the BUR can be improved by only considering 1 position index and one bit sequence of $26 - 8 = 18$ bits. In this case, the BUR is $18/26 \approx 0.69$. Furthermore, it may be possible to code the position index in less than one byte (by reducing, for example, the number of possible position indices). The goal is to have the BUR approach 1 as close as possible. Another possibility is to use a pseudorandom bit generator (PRBG) seeded with the

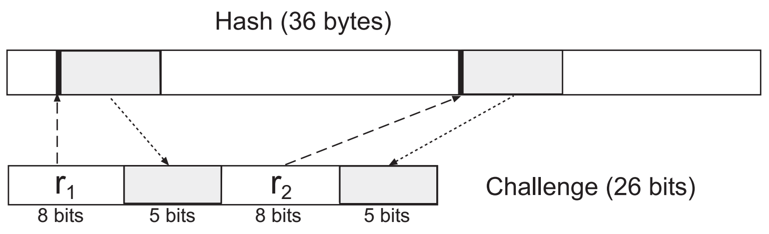

Figure 1 The construction of the challenge with two position indices

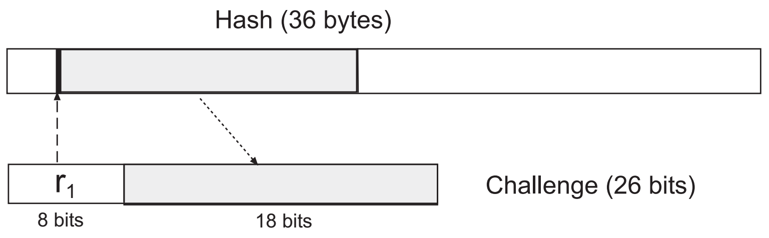

Figure 2 The construction of the challenge with only one position index

$Hash$ value to generate a bit sequence from which the challenge is constructed. We do not, however, consider this possibility further.

Last but not least, we note that connectivity may be seen as an added value for C/R tokens. It is possible to build C/R tokens that are not connected by default, but can be connected to provide better security.

5.2 OTP Systems

In contrast to C/R systems, OTP systems do not work with challenges. Instead, the entity that is authenticated provides an OTP to the entity that is authenticating it. No interaction or handshake takes place between the client and the server. Roughly speaking, there are three classes of OTP systems.

1. *Physical lists of OTPs:* Examples include scratch lists and access cards, as well as lists of transaction authentication numbers (TANs) and indexed TANs (iTANs).
2. *Software-based OTP systems:* Examples include Lamport-style [Lam81] OTP systems, such as Bellcore's S/Key [Hal95] and the one-time passwords in everything (OPIE) system [HM96].
3. *Hardware-based OTP systems:* Examples include SecurID, SafeWord, and SecOVID tokens, as well as integrated tokens, such as ICT Tokens of InCard Technologies. Note that most hardware-based OTP systems are not connected to the client systems. This simplifies the deployment of the tokens and makes them resistant to many malware attacks.

In [OHB06], we briefly mentioned how to use impersonal authentication tokens to complement hardware-based OTP systems in the sense that the resulting (combined) authentication system is SSL/TLS session-aware. In short, U employs the OTP as input for f (instead of PIN_U) in formula (3). Consequently, the UAC computed is

$$UAC = f(N_T, OTP).$$

Everything else (including, for example, the construction of N_T) remains unchanged. The disadvantage of this approach is that the user must have two tokens—the original OTP token and the impersonal authentication token—or the tokens must be integrated into one. The first possibility is likely to be unacceptable in practice, whereas the second possibility requires some time before integrated tokens appear on the marketplace. Consequently, the use of software-based OTP systems seems to be more promising, at least in the short term.

Among the classes of OTP systems enumerated above, the first class is definitively the most difficult one to make SSL/TLS session-aware. If we only have a physical (paper) list of OTPs, then a technique similar to the one used to make C/R systems SSL/TLS session-aware may be employed. If a SSL/TLS session between the browser and the server is established, then the user selects the next OTP not yet used and enters it into the browser. The browser, in turn, interprets this value as an index into the $Hash$ value. A specific number of bits is then extracted from the $Hash$ value and this bit sequence represents the UAC. The browser may display the UAC and the user enters it in a Web form, or it can be transferred as part of the SSL/TLS CertificateVerify message as discussed at the end of Section 2. In either case, the user must be taught to never enter an OTP directly into a Web form.

6 Conclusions and Outlook

Man-in-the-middle (MITM) attacks pose a serious threat to SSL/TLS-based e-commerce applications, such as Internet banking. In [OHB06], we introduced the notion of SSL/TLS session-aware user authentication (TLS-SA) and we proposed an implementation based on impersonal authentication tokens. In this paper, we presented a number of extensions and variations, which we believe are important for implementing TLS-SA and deploying it in practice. This is particularly true for soft-tokens and hard-tokens that are not physically connected to the client systems. These tend to be less secure, but they are much simpler to deploy than their connected counterparts. Again, we note that it is possible to design tokens that can either be connected or not, depending on the security requirements of the application one has in mind.

We believe that TLS-SA provides a lightweight alternative to the deployment and rollout of a public key infrastructure (PKI). In the short term, we are developing a TLS-SA proof of concept implementation for demonstration purposes. On the client side, we employ unconnected C/R tokens and a plugin for Microsoft Internet Explorer. On the server side, we employ (and adapt) a

Web portal that comprises a secure reverse proxy and an authentication server. The results we have achieved so far are promising and are reported elsewhere [O+07]. In the medium and long term, we plan to develop turnkey solutions that can be used to secure e-commerce applications against MITM attacks. For some authentication systems, this requires the cooperation of browser manufacturers. Nevertheless, we hope that we will be ready by the time the first wave of large-scale MITM attacks occur on the Internet. Initial, small-scale attacks are already taking place.

References

[Bur02] Burkholder, P., "SSL Man-in-the-Middle Attacks," SANS Reading Room, February 2002.

[DR06] Dierks, T., and E. Rescorla, "The TLS Protocol Version 1.1," RFC 4346, April 2006.

[Hal95] Haller, N., *The S/KEY One-Time Password System*, Request for Comments 1760, February 1995.

[HM96] Haller, N., and C. Metz, *A One-Time Password System*, Request for Comments 1938, May 1996.

[JM05] Jakobsson, M., and S. Myers, "Stealth Attacks and Delayed Password Disclosure," `http://www.informatics.indiana.edu/markus/stealth-attacks.htm`.

[Jos03] Josefsson, S., Ed., *The Base16, Base32, and Base64 Data Encodings*, Request for Comments 3548, July 2003.

[KBC97] Krawczyk, H., M. Bellare, and R. Canetti, *HMAC: Keyed-Hashing for Message Authentication*, Request for Comments 2104, February 1997.

[Lam81] Lamport, L., "Password Authentication with Insecure Communication," *Communications of the ACM*, Vol. 24, 1981, pp. 770–772.

[MT79] Morris, R., and K. Thompson, "Password security: a case history," *Communications of the ACM*, Vol. 22, Issue 11, November 1979, pp. 594–597.

[MT93] Molva, R., and G. Tsudik, "Authentication Method with Impersonal Token Cards," *Proceedings of IEEE Symposium on Research in Security and Privacy*, IEEE Press, May 1993.

[OHB06] Oppliger, R., Hauser, R., and D. Basin, "SSL/TLS Session-Aware User Authentication—Or How to Effectively Thwart the Man-in-the-Middle," *Computer Communications*, Vol. 29, Issue 12, August 2006, pp. 2238–2246.

[O+07] Oppliger, R., et al., "A Proof of concept Implementation of SSL/TLS Session-Aware User Authentication," *Proceedings of the 15th GI/ITG Conference on "Kommunikation in Verteilten Systemen" (KiVS '07)*, Berne (Switzerland), Springer-Verlag, Berlin, LNCS, February 26 - March 2, 2007, pp. 225–236.

[RSA04] RSA Laboratories, "PKCS #11 v2.20: Cryptographic Token Interface Standard," June 28, 2004.