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 (0)79 654 8437
E-mail: rolf.oppliger@esecurity.ch
[2] PrivaSphere AG
Fichtenstrasse 61, CH-8032 Zürich, Switzerland
Phone: +41 (0)43 299 5588, Fax: +41 (0)1 382 2133
E-mail: hauser@privasphere.com
[3] Department of Computer Science, ETH Zurich
Haldeneggsteig 4, CH-8092 Zürich, Switzerland
Phone: +41 (0)44 632 7245, Fax: +41 (0)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, and there are only a few technologies available to mitigate the risks. In [OHB05], 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 and variations of SSL/TLS session-aware user authentication. More specifically, we address multi-institution tokens, possibilities for changing the PIN, and possibilities for making several popular and widely deployed user authentication systems be SSL/TLS session-aware. Furthermore, we also investigate the technical feasibility and the security implications of software-based implementations of SSL/TLS session-aware user authentication.

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

1 Introduction

Man-in-the-middle (MITM) attacks pose a serious threat to SSL/TLS-based e-commerce applications, such as Internet banking, and there are only a few technologies that can be used to mitigate the risks. In [OHB05], we introduced the notion of SSL/TLS session-aware user authentication to protect SSL/TLS-based e-commerce applications against MITM attacks. The main idea is to make the user authentication depend not only on the user's (secret) credentials, such as a

password or personal identification number (PIN), but also on state information related to the SSL/TLS session in which the credentials are being transferred to the server. The rationale behind this idea 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 user employed when he sent out the credentials in the first place.

- If the two sessions are the same, then there is probably no MITM involved.
- If the two sessions are different, then something abnormal is going on. It is likely that a MITM is located between the user's client system and the server.

Using SSL/TLS session-aware user authentication, the user authenticates himself by providing a user authentication code (UAC[4]) that depends on both the credentials and the SSL/TLS session (in particular, information from the SSL/TLS session state). A MITM who gets hold of the UAC can no longer misuse it by simply 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 session, then the server can easily recognize this fact and drop the session. As such, SSL/TLS session-aware user authentication provides a lightweight alternative to the deployment and rollout of a public key infrastructure (PKI) to protect against MITM attacks.[5]

There are a number of possibilities to implement SSL/TLS session-aware user authentication. In [OHB05], we argued (i) that software-based implementations are inherently vulnerable, (ii) that one should therefore pursue hardware-based implementations in the first place, and (iii) that a particularly promising possibility is the use of hardware tokens, preferrably in the form of impersonal PKCS #11-compliant authentication tokens. While we are still in basic agreement with these points, we are less strict in this paper. In fact, we broaden the scope of our ideas and also consider possibilities to implement (parts of) SSL/TLS session-aware user authentication in software, and investigate the security implications of doing this. Consequently, we make a distinction between hardware-based and software-based implementations of SSL/TLS session-aware user authentication.

- In the first case, we are talking about hardware tokens (i.e., hard-tokens). Such a token may be (physically) connected or not.[6]
- In the second case, we are talking about software tokens (i.e., soft-tokens).

In either case, an authentication token can be personal or impersonal. Furthermore, it can be consistent with a cryptographic token interface standard,

[4] In [OHB05], a UAC is referred to as a transaction authorization number (TAN).

[5] Note, however, that the deployment and rollout of a PKI typically may have other objectives, e.g., the ability to provide nonrepudiation services.

[6] We say that the 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.

such as PKCS #11 [RSA04] or Microsoft's cryptographic application programming interface (CAPI). We assume that a standard CAPI driver is preinstalled on newer versions of the Windows operating system, and that this driver can be used by the Microsoft Internet Explorer to drive the token in some transparent way (from the user's viewpoint). On all other platforms, we assume that some PKCS #11 driver software must be installed to drive the token. This includes, for example, the case in which a Mozilla browser is employed on a Windows platform.

For the remainder of this paper, we present a number of extensions and variations of SSL/TLS session-aware user authentication. To make the paper self-contained, we summarize the originally proposed token-based approach in Section 2. We then address multi-institution tokens, possibilities for changing the PIN, and possibilities for making several popular and widely deployed user authentication systems be SSL/TLS session-aware in Sections 3 – 5. In Section 6, we address the question whether (parts of) SSL/TLS session-aware user authentication can be reasonably implemented in software. Finally, we draw conclusions and provide an outlook in Section 7.

2 Token-Based Approach

The implementation of SSL/TLS session-aware user authentication proposed in [OHB05] is based on impersonal authentication tokens. Each token holds a public key pair,[7] of which the private key is used to digitally sign CertificateVerify messages in the SSL/TLS handshake protocol. Each CertificateVerify message, in turn, represents a digitally signed hash value of all messages previously exchanged during the execution of the SSL/TLS handshake protocol. Part of these messages is the server's Certificate message, which comprises the server's public key certificate (the server's public key is included in this certificate). Consequently, the CertificateVerify message is logically bound to the server's public key.

The following entities play a role in the originally proposed token-based approach:

- 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.[8]

The user U is identified with identifier 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.

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

[8] 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 serial number may, for example, be imprinted on the back side of the token. Furthermore, T is equipped with both a public key pair (k, k^{-1})—of which the private key k^{-1} is used to digitally sign the CertificateVerify messages—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 it is not specific to the user). K_T can be generated randomly or pseudo-randomly using a master key MK:

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

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 by anybody who holds MK. MK, in turn, is typically held exclusively by S.

When U wants to access S, he 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 (at this point in time, we do not assume that the user properly verifies this certificate). S is configured in a way that it always requires client authentication by sending out a CertificateRequest message to C. When C receives the message, it knows that it must authenticate itself by returning a Certificate and a properly signed CertificateVerify message to S. As mentioned above, the CertificateVerify message must comprise a digitally signed hash value of all previously exchanged messages (the hash value is further referred to as $Hash$). The digital signature is generated by T using its private key k^{-1}. Due to the fact that T is an impersonal token, the CertificateVerify message neither authenticates the token nor the client. Instead, the CertificateVerify message only ensures that C uses a token to establish an SSL/TLS session with S, and that the SSL/TLS session-aware user authentication mechanism gets access to $Hash$. Alternatively speaking, the token acts as a trusted observer.

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

$$N_T = E_{K_T}(Hash)$$

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 could also represent a message authentication code (MAC) computed with a keyed one-way hash function where K_T is the key.[9] For example, the HMAC construction [KBC97] can be used to generate

$$N_T = HMAC_{K_T}(Hash).$$

[9] 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 on the use of (keyed) one-way hash functions (in contrast to the use of encryption systems).

In either case, N_T can be shortened to the length of PIN_U (e.g., 8 decimal digits) by truncating it. This value 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 follows:

$$UAC = f(N_T, PIN_U) \tag{1}$$

In general, there are many ways to define an appropriate function f. One possibility adopted from [MT93] is the digit-wise addition modulo 10. In this case, the UAC is the digit-wise modulo 10 sum of N_T and PIN_U, which is a function simple enough for users to compute themselves (e.g., in their heads). If the token comprises a keypad (to enter PIN_U), then the token can also be used to compute f. In this case, f can be a much more complex function.

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.[10] 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 a one-way hash value of PIN_U (instead of PIN_U). This is a well-known security mechanism to protect the PINs (or passwords, respectively) 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 be designed in a way that it can access the SSL/TLS cache to retrieve $Hash$.

Note that the token-based approach described so far neither requires synchronized clocks nor that nonces are sent back and forth between the entities involved in the user authentication. Instead, the approach employs a new idea, namely using nonces derived from the symmetrically encrypted hash values used by the SSL/TLS protocol.

Also 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 (and hence the user does not have to enter anything in a Web form). There are at least three possibilities here:

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

[10] 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, the server S can retrieve SN_T from the certificate. Similarly, 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 only weak.

3. T can only send the keyed hash value (instead of the digital signature). This possibility 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 in the SSL/TLS handshake protocol and the way the SSL/TLS CertificateVerify message is used in the protocol. This is disadvantageous. However, a major advantage is that C (i.e., the browser) then remains unaffected and need not be altered in one way or another. This simplifies deployment considerably (because the server and the tokens are typically under the control of the service provider). Furthermore, the property that the client need not be altered obviously applies to all application protocols that may be layered on top of SSL/TLS (in addition to HTTP or HTTPS, respectively).

3 Multi-institution Tokens

The notion of a multi-institution token was briefly mentioned in [OHB05]. As the name suggests, the idea is to allow T to be used by multiple institutions I, i.e., multiple institutions can use the same token for user authentication. The rationale behind multi-institution tokens is an economic one: it may be cost effective to have multiple institutions 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, and the token key with a set of institution-specific token keys K_{IT} (where K_{IT} refers to the secret token key that T shares with I). Similar to the single-institution case, the token key K_{IT} can be generated dynamically according to

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

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

There is another possibility to implement 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:

$$UAC_I = f'(N_T, PIN_{IU})$$
$$UAC_{AS} = E_{MK_T}(N_T)$$

The pair of UACs is then submitted to I. I, in turn, forwards UAC_{AS} to AS for decryption, and AS returns back N_T to I. It goes without saying that 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}.

Either possibility to implement multi-institution tokens is useful in practice. 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 is more appropriate.

4 Changing the PIN

The security of a token-based implementation of SSL/TLS session-aware user authentication largely depends on the fact that users never have to enter their PIN into the client system (e.g., in a Web form). Instead, they either use the PIN to compute the UAC in their head or they directly enter the PIN in the authentication token in use. If one weakens this constraint, then users are vulnerable to the "doppelganger window" attack [JM05]. In such an attack, the adversary pops up a faked window to request user credentials. Since a user is typically not able to distinguish original and faked windows, it is likely that he enters his credentials into any window that asks for them. As of this writing, there is no technology that fully protects against this attack (this includes, for example, Delayed Password Disclosure (DPD) proposed in [JM05]). This situation is unsatisfactory and we return to this problem in Section 6 (when we elaborate on soft-tokens).

The fact that users are preferrably taught to never enter their PIN into a client system considerably complicates changing the PIN. Normally, one would implement a Web form in which the user can change his PIN interactively. Since we disallow 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 an SSL/TLS session with an UAC, 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) can compute 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 [OHB05], 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 tranferred as part of the SSL/TLS CertificateVerify message as discussed at the end of Section 2 (again, this requires the server to properly interprete 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.[11] In the remainder of this paper, we elaborate on how to make some popular and widely deployed user authentication systems be SSL/TLS session-aware in a way that provides protection against MITM attacks. We distinguish between one-time password (OTP) and challenge-response systems. In the following section, we then elaborate on the technical feasibility and the security implications of software-based implementations of SSL/TLS session-aware user authentication.

5.1 OTP Systems

There are basically 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).

[11] A proof-of-concept for indexed TANs (iTANs) was developed by the "Arbeitsgruppe Identitätsschutz im Internet e.V." at the University of Bochum in Germany (cf. https://www.a-i3.org/content/view/411/28/).

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 and SecOVID tokens. Note that most hardware-based OTP systems are not connected to the client systems. This makes the enrollment and deployment of these tokens considerably simpler (than the ones that are connected to the client systems). It also makes them resistant to malware attacks.

In [OHB05], we briefly mentioned how to use impersonal authentication tokens to complement hardware-based OTP systems, such as SecurID tokens, 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 (1). Consequently, the UAC computed is

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

Everything else (including, for example, the construction of N_T) remains the same. The disadvantage of this approach is that the user must have two tokens (i.e., the original OTP token and the impersonal authentication token). This is likely to be unacceptable in practice and therefore the use of a software-based OTP system seems to be appropriate. In this case, the OTP system is implemented in software and is complemented by a hard-token. Again, the use of soft-tokens is addressed in the following section.

A simple trick is required to make lists of OTPs be SSL/TLS session-aware. Namely, a compression function *compress* implemented in the token can be used to compress $Hash$ in a way that the result, i.e., $compress(Hash)$, may serve as an index in the list of OTPs. If, for example, $compress(Hash) = 37$, then the 37^{th} entry in the list represents the OTP that must be used. Due to the fact that collisions are likely to occur, a collision resolution strategy is needed. Since the server can keep track of previously used values, we can require that the server initiates an SSL/TLS session renegotiation, or that the client and the server both compute an offset value (for the list of OTPs). In the second case, the offset value must be computed pseudo-randomly from the SSL/TLS session state. In either case, it is obvious that a list of OTPs should be replaced long before all OTPs on the list are used (otherwise, collisions become frequent and the performance decreases considerably).[12]

Once the appropriate OTP is found (in the list of OTPs), it can be entered together with PIN_U in the token. The token, in turn, uses the $Hash$ value and an appropriately chosen function f'' to compute a UAC:

$$UAC = f''(PIN_U, OTP, Hash) \tag{2}$$

[12] The question when to replace the lists of OTP is an optimization problem. In either case, a replacement must be requested and initiated by the server.

In this formula, $Hash$ is required to protect against a trivial MITM attack. If the list of OTPs is relatively short and the MITM needs a specific OTP to authenticate to the server (on behalf of the user), then the MITM can simply initiate a series of SSL/TLS session renegotiation or computation of offset values until the appropriate OTP is found. If $Hash$ is included in the UAC computation, then it is no longer sufficient for the MITM to have the appropriate OTP. Furthermore, $Hash$ provides a salt value that makes it computationally more expensive to precompute tables of valid PIN_U-OTP pairs for specific UAC values. This, in turn, makes off-line guessing attacks more difficult to launch.

In either case, the UAC can be entered by the user in a Web form, or it can be tranferred as part of the SSL/TLS CertificateVerify message as discussed at the end of Section 2. In the second case, one may also use a nonce N (instead of $Hash$) to randomize the UAC.[13] For example, N may be added bitwise modulo 2 to the UAC, and N may be encrypted with a public key of the server S (i.e, k_S). Both values—$UAC \oplus N$ and $E_{k_S}(N)$—are then transferred as part of the SSL/TLS CertificateVerify message to S. S, in turn, can use its private key k_S^{-1} to decrypt N and extract the UAC from $UAC \oplus N$. Alternatively, one may also work with 2 nonces. In this case, one of the nonces may be used for mutual authentication (i.e., to authenticate the server to the client).

5.2 Challenge-Response Systems

Informally speaking, a challenge-response (CR) system is an authentication system in which the entity that is authenticated (typically the client) is provided with a challenge for which it must compute an appropriate response to the entity that is authenticating (typically the server). In contrast to OTP systems, CR systems are often implemented in hardware. The corresponding (hardware) tokens may or may not be connected to the client systems using some cryptographic token interface standard , such as PKCS #11 or CAPI.

There is a simple and straightforward possibility to make a CR system be SSL/TLS session-aware: instead of having the server provide a challenge to the client, the client and the server use N_T as a challenge, which is cryptographically protected using the token key K_T as a shared secret. The resulting UAC is the response that is computed according to formula (1). We distinguish between two cases depending on whether the token is connected to the client system or not.

1. If the token is connected to the client system, then it is simple and straightforward to make the user authentication be SSL/TLS session-aware. In this case, $Hash$ is sent to the token, and the token can use it to compute N_T. The user must additionally input PIN_U, so that the token can compute the UAC according to formula (1).
2. If the token is not connected to the client system, then there is no direct communications path between the client and the token. This means that

[13] In this case, the formula (2) to compute the UAC only takes PIN_U and the OTP as arguments. This simplifies things considerably from the user's point of view.

there must be a possibility to communicate SSL/TLS state information from the browser to the token. One possibility, which is further addressed in the following section, is to have the browser display some digits of $Hash$ in the graphical user interface (GUI) in some authentic way. The user can read these digits and enter them—together with PIN_U—in the token. The token, in turn, can again use formula (1) to compute the UAC.

The major advantage of the first case, where the token is connected, is that the user interaction is straightforward. Furthermore, one can use N_T in its entire length and thereby provide better protection against PIN guessing attacks. The major disadvantage is the necessity to install a token driver on the client system (unless one uses only Windows-based client systems and platforms). The major advantage of the second case, where tokens are not connected, is that there is no need for a token interface, and hence, a token driver need not be installed. The major disadvantage is that the browser must be modified so that its GUI display some digits of $Hash$. This point is further addressed in the following section.

In either case, the UAC must be displayed on the token and the user must enter the UAC in the appropriate Web form. Alternatively, the UAC can also be tranferred as part of the SSL/TLS CertificateVerify message as discussed at the end of Section 2.

6 Software-based Implementations

So far, we have assumed that all tokens are hardware-based, meaning that we only have to deal with hard-tokens. We now weaken this assumption and elaborate on the technical feasibility and the security implications of software-based implementations of SSL/TLS session-aware user authentication.

In a software-based implementation, the hard-token is replaced with a soft-token, meaning that the token's functionality is simulated in software, and that the token's display is emulated on the display of the client system. The soft-token may still be compliant to PKCS #11, CAPI, or any other cryptographic token interface standard. In this case, the basic functionality and interface of the soft-tokens remain essentially the same (as compared to the hard-tokens). On the one hand, soft-tokens are flexible and less expensive than hardware-based solutions. On the other hand, however, soft-tokens have to deal with (at least) two security problems:

1. Soft-tokens are inherently vulnerable to malware and keylogger attacks. Malware can do many things, including, for example, reading out cryptographic keys. Keylogger attacks typically try to retrieve the user credentials when they are typed in.
2. Soft-tokens are vulnerable to visual spoofing attacks.

Both problems are difficult to solve. Keylogger attacks can be partially solved by displaying a keyboard on the client' screen and having the user type in his credentials using this keyboard. The second problem is particularly tricky. One

has to find means to have the soft-token's GUI display authentic information. As of this writing, there are only a few (visual) technologies that can be used for this purpose (e.g.,[YS02, DT05]).

In one way or another, a software-based implementation of SSL/TLS session-aware user authentication must have access to some SSL/TLS state information, in particular the $Hash$ value.

- If the soft-token is consistent with PKCS #11 or CAPI, then it has immediate access to this information (similar to the hard-token). In this case, the implementation of the soft-token is essentially the same. This includes, for example, the necessity to install driver software on the client system.
- If, however, the soft-token is not consistent with PKCS #11 or CAPI, then it has no immediate access to this information. In this case, the situation is slightly more involved and one must employ other means to access the $Hash$ value. One possibility is to modify the browser in a way that it is able to render and display the first digits of the $Hash$ (or $compress(Hash)$, respectively) value as it appears in the execution of the SSL/TLS handshake protocol. These digits can, for example, be displayed near the closed padlock icon that marks the SSL/TLS session (typically at the bottom right of the bowser window). The character set and length of $compress(Hash)$ may be configurable, and thereby meet the requirements of different user authentication mechanisms and systems.

In the second case, the users can be equipped with a (typically small) program that implements a UAC calculator. If, for example, we work with lists of OTPs, then the user can employ the first digits of $Hash$ as an index in the list of OTPs. The OTP found in the list must be entered by the user—together with PIN_U and $Hash$—into the UAC calculator, and the UAC calculator must compute and display the currently valid UAC according to formula (2). This value must then be entered by the user in a Web form or transferred to the server S as part of the SSL/TLS CertificateVerify message. In the second case, we have the possibility to work with nonces encrypted with a server public key instead of $Hash$ (cf. Section 5.1). The big advantage we see in this case is that there is no secret key that must be stored on the token.

In either case, the user must have the assurance that the first digits of $Hash$ displayed by the soft-token are authentic and can somehow be verified. Otherwise, an attacker can fake the digits and use the UAC to launch a PIN guessing attack. We already mentioned the fact that this problem is not trivial and that there are only a few technologies available.

7 Conclusions and Outlook

Man-in-the-middle (MITM) attacks pose a serious threat to SSL/TLS-based e-commerce applications, such as Internet banking, and there are only a few technologies that can be used to mitigate the corresponding risks. In [OHB05],

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 presented a number of extensions and variations of SSL/TLS session-aware user authentication. More specifically, we proposed multi-institution tokens, possibilities for changing the PIN, and possibilities for making several popular and widely deployed user authentication systems be SSL/TLS session-aware. In addition, we have also described the technical feasibility and the security implications of software-based implementations. We think that these extensions and variations are important to implement SSL/TLS session-aware user authentication in a practical (real-world) setting. 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 simpler to deploy than their connected counterparts.

Our next steps will be to prototype the above-mentioned possibilities to make lists of OTPs (e.g., access cards) and EMV cards that implement Mastercard's chip authentication program (CAP) or Visa's dynamic passcode authentication be SSL/TLS session-aware. In the first case, we intend to employ soft-tokens, whereas in the second case, we intend to integrate the functionality into existing hard-tokens. In either case, we think that both possibilities have a large potential, mainly because the corresponding authentication systems are (and will be) widely deployed in practice. In fact, many banks are using lists of OTPs and consider the replacement of these lists with EMV cards in the future.

References

[**DT05**] Dhamija, R., and J.D. Tygar, "The Battle Against Phishing: Dynamic Security Skins," *Proceedings of the 2005 ACM Symposium on Usable Security and Privacy (SOUPS 2005)*, ACM Press, July 2005, pp. 77–88.

[**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`.

[**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.

[**OHB05**] Oppliger, R., Hauser, R., and D. Basin, "SSL/TLS Session-Aware User Authentication—Or How to Effectively Thwart the Man-in-the-Middle," submitted for publication

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

[YS02] Ye, Z.E., and S. Smith, "Trusted Paths for Browsers," *Proceedings of the USENIX Security Symposium*, 2002, pp. 263–279.