Poster: Save our passwords

Martin Boonk
Department of Computer Science
University of Paderborn
33098 Paderborn, Germany
mboonk@mail.upb.de

Ronald Petrlic
Department of Computer Science
University of Paderborn
33098 Paderborn, Germany
ronald.petrlic@upb.de

Christoph Sorge
Department of Computer Science
University of Paderborn
33098 Paderborn, Germany
christoph.sorge@upb.de

I. Introduction

Passwords, despite the problems they entail in terms of security, are widely used for user *authentication* towards web services, mail boxes, etc. [1] They are often used improperly:

- users often use short and easy to remember passwords
- the same password is used for different services
- passwords are stored in unsafe locations (on the computer) or are written down on paper

Many services use *Transport Layer Security* (TLS) [2], providing *confidentiality*, *integrity*, and *authenticity*, to secure the communication between the user's host and the service provider's server, thus securing passwords during transmission.

Even if using secure transmissions, passwords are still unprotected on the user's host. This host may be compromised, i.e. controlled by an attacker (due to malware infections or other reasons). Techniques like certificate-based client authentication as part of the TLS handshake—with the private key stored on the user's smartcard—solve the problem, but they are not popular in practice as they mostly entail changes on the server side. Few service providers offer such functionality.

A. Contribution

We come up with a solution that protects users' passwords from being spied out by an untrusted user's host during the password-based authentication to a server. The main advantage of our solution is that it does not require any modifications on the server-side. While smartcard producers have presented cards that support TLS, our approach has been tested on cards that are available at low cost to end customers.

We assume that the *attacker* we are dealing with is only interested in passwords. The attacker has control over the user's host, i.e. he can arbitrarily compromise the software. Moreover, we assume that the attacker can mount a hardware key logger. Users might face such an attacker in an Internet Cafe or at other publicly available terminals.

We require the server to support passwords with a sufficient amount of characters consisting of letters/numbers/special characters and which cannot be found in a dictionary.

II. Approach

Our basic approach is as follows. The user is in possession of a *smartcard* (SC) and a smartcard reader (*card acceptance device*, CAD) that includes a PIN pad[1]. In a first step, the user initializes the SC in a "trusted environment", e.g. on his own computer that is (assumed to be) uncompromised. Intialization means that the user sets the PIN for his SC and further stores passwords (and settings[2]) for services that he will be using in the future on the SC. As a more convenient alternative, we suggest to develop a distributed system, in which a subset of users performs the configuration for each website, and this configuration is automatically adopted by other users' smartcards. However, the decisions on other users' trustworthiness in such a system are not straightforward, so the initial version depends on manual configuration.

The *authentication protocol* works as follows. The user first authenticates with his PIN (entered on his CAD) towards the SC. Then, the SC initializes a TLS handshake with the service provider. The server's certificate is validated (by checking for equality with the server certificate stored on the SC, or by validating a certificate chain). If this check succeeds, both parties negotiate a pre-master secret via RSA key exchange as defined in the TLS standard [2] and then derive the master secret. When the server sends the website that requires the client authentication via password over the TLS-secured channel, the SC submits the values for the username and password fields—which are part of the TLS settings stored on the SC—to the server. The client is now authenticated and for performance reasons, the TLS session is transferred from the SC to the host. This transfer is performed by the SC by initiating a new TLS handshake—using the resumption feature of the TLS protocol, i.e. using the previous session ID. This results in new key material—based on hashing the previous master secret and new random values exchanged during the current handshake. The new key material is provided to the host. The host can now communicate with the server over the TLS-protected channel. On the host, a browser extension handles the handover of the secure communication session.

III. Discussion

We briefly discuss our concept in terms of security and give an overview of the overall performance of the implementation.

[1]Note that SCs including a PIN entry facility exist—in which case a separate CAD would not be needed.

[2]Settings include the preferred cipher suite as well as the server's certificate or trusted root certificates.

A. Evaluation

Our solution achieves the required properties. The user does not need to remember his passwords and thus is able to create good passwords uniquely for each service during initialization of the SC. The user only needs to remember the SC's PIN. Moreover, our solution does not require any changes to be done on the service providers's side, which allows for an easy roll-out of our solution.

In terms of security, we achieve our requirement that an attacker as depicted above—controlling either the used host or eavesdropping on the communication channel as *man in the middle* (MITM)—cannot access the user's passwords. Even though the TLS session is transferred from the SC to the host, the attacker cannot retrieve the password as new key material is generated during the TLS session transfer. The SC's internal random number generator (RNG) is used to create random numbers for the pre-master secret—which yields "good" random numbers with high entropy.

Due to the low throughput of the SC, as covered next, our solution does not allow transmission of large amounts of data between the SC and the server. Besides authentication, the solution could be used for other scenarios as well, though. For example, some very sensitive (small amounts of) data could be stored on the SC in a trusted environment, e.g. on a SIM card[3] by the mobile phone, and the data can be securely transmitted in an Internet Cafe rather than via using some mobile carrier network in foreign countries to avoid roaming costs.

B. Results

Our used Javacard 2.2.2 SC has the following features:

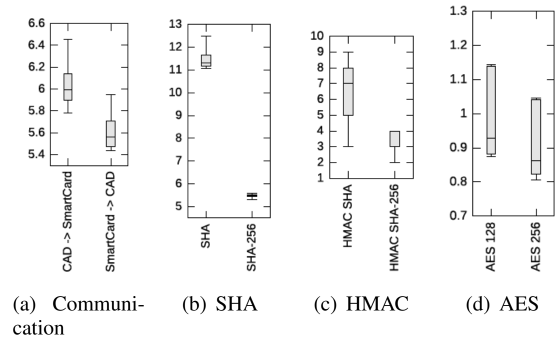
- 200 KB ROM
- 80 KB EEPROM
- 6 KB RAM (3.5 KB usable)
- 16 Bit processor
- Crypto co-processors for RSA, ECC, DES and AES
- Hardware RNG

The standard Sun/Oracle JavaCard SDK 2.2.2 (2006) and the OpenJDK 1.7 (SmartcardIO) were used.

Besides implementing the TLS 1.2 handshake with RSA key exchange and the *TLS_RSA_WITH_AES_CBC_128_SHA* cipher suite specified in [2], we implemented the pseudo-random number generation (PRNG) function as well as the HMAC algorithm. The HMAC algorithm is indeed part of the *JavaCard* specification and could be implemented natively on SCs, however, it is not implemented on our used SC. Due to the card's limited performance, we have not implemented certificate chains validation, but rely on stored server certificates.

The achieved throughput is shown in Fig. 1 and is as follows. The throughput of data from the SC to the CAD is 6 KB/sec. and from CAD to SC is 5.6 KB/sec. (Fig. 1(a))—this is the hardware's upper bound for the throughput of our solution. *SHA-1*, used for integrity validation, achieves a throughput of 11 KB/sec. (Fig. 1(b)) and *SHA-256*, used as part of *HMAC* for the PRNG (Fig. 1(c)), 5.5 KB/sec. The smartcard's AES implementation provides a surprisingly low throughput of only 0.9 KB/sec. (Fig. 1(d)) The running times of our implemented TLS PRNG are shown in Fig. 2.

[3]Note that a SIM card is a smartcard as well and could thus easily be used in our scenario.



(a) Communication (b) SHA (c) HMAC (d) AES

Fig. 1. Throughput in KB/sec.

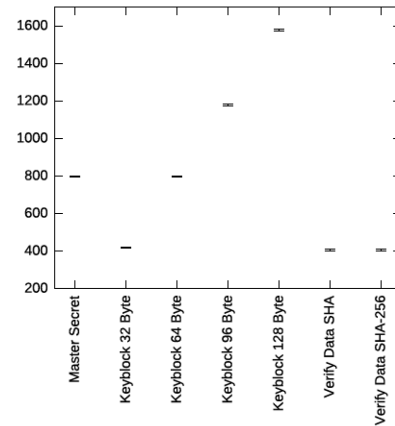


Fig. 2. Running times in ms.

The TLS handshake with the cipher suite *TLS_RSA_WITH_AES_CBC_128_SHA* performed between the SC and the server—up to the first transmission of application data takes 11 seconds. The computation of the *master secret*, *keyblock* and *verify data* with the pseudo-random function takes 2.8 seconds. The following data transmission achieves a throughput of 0.8 KB/sec.

We published part of our implementation—a library that performs the smartcard-enabled TLS handshake—as open source project at https://github.com/halemmerich/opentlssc. Furthermore, we would like to test our solution on more efficient SCs as we believe that the solution will achieve a much better throughput on newer SCs.

References

[1] J. Bonneau, C. Herley, P. C. v. Oorschot, and F. Stajano, "The quest to replace passwords: A framework for comparative evaluation of web authentication schemes," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, ser. SP '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 553–567.

[2] T. Dierks and E. Rescorla, *The Transport Layer Security (TLS) Protocol, Version 1.2*, Internet Engineering Task Force (IETF) Request For Comment (RFC) 5246, Aug. 2008, Proposed Standard.