Lecture Notes for CS-578 Cryptography (SS2006) Prof. Michael Backes

12. Authentication Methods, SSL, and other Important Protocols

Lecture 17 Saarland University

This chapter investigates various methods for letting a user authenticate itself to a server. Furthermore we will discuss several important protocols that are frequently used in practice, especially concentrating on SSL/TLS.

12.1 Authentication

Authentication is one of the crucial aspects of computer security, especially since it is used very often in daily life, e.g., one authenticates when logging into a computer, when checking emails, when doing online banking, and so on.

Let us first note that authentication is typically complemented with a key exchange protocol as otherwise an attacker can simply hijack a session, i.e., take over your session after your authentication to the server has been completed, e.g., by cutting off your connection and taking over your IP.

12.1.1 Password Authentication

The probably most common method used for authentication are passwords. They are easy to handle, easy to carry with you, and the principle is easy to understand. On the other hand, they often do not provide sufficiently strong security guarantees: common passwords in practice tend to have low entropy ("`password`" as a password is much more common than "`k34-hN(a1W*`"), and some protocols even further decrease entropy by not allowing special characters, by truncating the password, or by even converting the whole password to uppercase letters.

The most intuitive way to realize password authentication is to store the password on the server's harddisk and to test if the password provided by the user matches the stored password. While this solution seems natural, it bears the risk that an attacker that successfully gains access to the server's harddisk can simply read the password and use it for later authenticating himself as the user.

Instead of storing the password in clear, servers typically store a list containing entries of the form $(\mathit{Alice}, \mathsf{H}(P_{\mathit{Alice}}))$, where P_{Alice} is Alice's password and H is a one-way (hash) function. Then a password presented by a user P is verified by first applying H and then comparing $\mathsf{H}(P)$ with the stored hash of the password password. Thus even if an attackers gets access to the password file, it still needs to find a password P that maps to a fixed value $\mathsf{H}(P_{\mathit{Alice}})$ under H, thus an attacker needs to break one-wayness of the function H.

Early versions of Windows (before Windows 2000 SP3) used a password verification called LANMAN. It accepted 14 bytes for each password P, converted these bytes into uppercase letters, then split them into two halves of equal size $P = P_1 \parallel P_2$, and used both halves as DES keys to encrypt a fixed message IV. This method had various weaknesses: Most importantly, converting all letters to upper-case significantly reduces the passwords entropy, thus paving the way to successful dictionary attacks. Additionally, truncating passwords to 14 bytes is not good, but much worse is splitting the password into two independent halves. Thus an attacker can attack both halves independently, which makes successful dictionary or brute-force attacks much easier to mount. Nowadays, Windows relies on MD4 hashes, which constitutes a much better primitive for storing password compared to LANMAN.

Early versions of Linux allowed passwords of 7 bytes only (but it additionally uses salting to protect against dictionary attacks, see below), and uses the password as key for several consecutive DES encryptions of a fixed message IV.

Dictionary attacks A generic attack against hashed passwords stored in a file are so-called *dictionary attacks.* The attack does not depend on the concrete hash function used, as long as the function is publicly known. The adversary starts with a dictionary of common words $W_1, W_2, \ldots$ (such dictionary are freely available on the net) and simply hashes each of them, i.e., he computes $\mathsf{H}(W_1), \mathsf{H}(W_2), \ldots$, until he found a hash H which is stored in the file. If this does not succeed then he tries to modify these words as a user might have done, i.e., by appending or inserting numbers and special characters, by reversing the order, and so on.

This is of course a heuristic attack, which however turns out to be successful in practice if an attacker is satisfied with finding *any* password stored in the file (in practice even most passwords, depending on the sophistication of the users to choose passwords, of course).

This attack can be made much more difficult to mount by *salting* the passwords: instead of hashing the password P_A, one picks a random $salt_A \in \{0,1\}^l$ for every user A (a common choice is $l = 12$), and stores

$$Alice, salt_A, \mathsf{H}(salt_A \parallel P_A)$$

in the file. Verifying the correctness of a given password can easily be done, as the salt is known to the server, but an attacker cannot attack a large number of passwords simultaneously, as they use different salts. So he has to attack each password individually, resulting in a much longer computation.

The idea of adding some form of uncertainty to the hash value can be further extended by using a so-called *secret salt* or *pepper.* One chooses $salt_A \in \{0,1\}^l$ and $salt^*_A \in \{0,1\}^k$ (a common choice is $k = 8$), and stores

$$Alice, salt_A, \mathsf{H}(salt_A \parallel salt^*_A \parallel P_A).$$

To verify a password the server has to try all 2^k values for $salt^*_A$. If the server is given a correct password, this only increases the computation time from micro-seconds to milli-seconds, which is usually doable. The attacker's time, however, is as well increased by a factor of 2^k; however this is typically is from, say, one day to 2^k days.

12.1.2 Biometric Passwords

Biometric authentication methods are a pretty novel research and application area. One can for example use finger prints, iris scans, or speech-recognition to identify a person. Using biometrics as passwords however suffers from two main problems: First, the biometric passwords are not secret, e.g., copies of a person's fingerprints turn out to be easily achievable in practice. Secondly, there is no possibility to revoke a copied fingerprint (except by using surgery of course).

12.1.3 One-time Passwords

The following authentication scheme, which is due to Lamport, exploits so-called one-time passwords. Alice generates a password P_A, and computes $W_A := \mathsf{H}^n(P_A) := \mathsf{H}(\mathsf{H}(\ldots(\mathsf{H}(P_A))\ldots))$ for some one-way function H. The server stores W_A, Alice stores P_A and sets $cnt := n$. If Alice wants to authenticate to the server, she decreases $cnt := cnt - 1$, sends $A := \mathsf{H}^{cnt}(P_A)$ to the server. The server verifies that $\mathsf{H}(A) = W_A$, and if so, sets $W_A := A$. This scheme in particular avoids eavesdropping attacks since merely listening on the channel will only reveal passwords that would have been accepted by the server before he changed its state and hence the value of W_A. Moreover, breaking into the server does not weaken the scheme since no secrets are stored on the server. The drawback however is that Alice can only authenticate herself a limited number of times, namely n times. After that, the chain originating at her password has been used up so that she has to generate a new passwords, hash it n times, and let the server know the new value W_A.

Another system relying on one-time passwords is SecureID, where the client owns a smartcard. Here the server and the smartcard start with the same random value K_0, and every minute, they compute a new value $K_{i+1} := \mathsf{H}(K_i)$. If Alice would like to authenticate to the server, the smartcard outputs $\mathsf{DES}(K_i, ct)$, where ct denotes the current time. While this system allows for unlimited password usage, the drawback is that the server is now required to store a secret; moreover, the smartcard has to be tamper-resistant as otherwise stealing the smartcard, copying its memory and handing it back unnotedly would cause a severe threat to this kind of system. Note that the protocol is secure against eavesdropping attacks as well.

12.1.4 Challenge-Response Protocols

Another common authentication method is to use challenge-response protocols.

Based on Symmetric keys Alice and the server share a common key K. The authentication procedure constitutes a three-round protocol: Alice initiates the protocol by sending a hello message to the server. The server picks a random r – the challenge – and sends it to Alice. Alice picks a random r' and sends $\mathsf{E}(K,\ r \parallel r')$ to the server. The server decrypts this encryption, and if the first component matches the previously sent challenge r, it considers Alice authenticated. This protocol resists eavesdropping attacks but requires the server to stores secrets.

Based on Public-key Cryptography The protocol works exactly as the one before, but together with the hello message, Alice also sends her certificate for her public signature key. Later instead of encryption the message with a secret key, she signs the message with her secret signature key. The server then only tests the validity of the signature, which frees the server from storing secret information.

The Station-to-Station Protocol (STS) The STS protocol constitutes a widely known protocol for authentication and key exchange. It relies on a large prime p and a generator g of $\mathbb{Z}_p^*$ as publicly known parameters. Alice selects signature keys pk_A, sk_A, and obtains a certificate $cert_A$ for pk. Alice and Bob pick $x, y \in \{1, \ldots, p-1\}$, respectively. Then Alice sends $cert_A, g^x \bmod p$ to Bob, which in turn sends $cert_S, g^y, \mathsf{E}(K, \mathsf{S}(sk_S, (g^x, g^y)))$ where $K := g^{xy}$. Alice answers with $\mathsf{E}(K, \mathsf{S}(sk_A, (g^x, g^y)))$. K can be computed by both parties and is the shared key. Based on this shared key, Alice and Bob can subsequently authenticate themselves to each other, e.g., by running the aforementioned challenge-response protocol.

12.1.5 Encrypted Key Exchange (EKE)

The encrypted key exchange protocol (EKE) works as follows. Both Alice and the server share a password pwd. They choose $x, y \in \{1, \dots, p-1\}$, respectively. Alice sends $h := \mathsf{AES}(pwd, g^x)$ to the server, which in turn sends $i := \mathsf{AES}(pwd, g^y)$ to Alice. Then both compute $K := g^{xy}$ as usual. Then Alice sends r to the server, and the server answers with $\mathsf{E}(K, r \parallel r')$ as for the challenge-response protocol based on symmetric keys.

$$\begin{tikzcd}
\text{Client } (pwd) & & \text{Server } (pwd) \\
x \leftarrow_{\mathcal{R}} \{1, \dots, p-1\} & & y \leftarrow_{\mathcal{R}} \{1, \dots, p-1\} \\
{} \arrow[rr, "\mathsf{AES}(pwd, g^x)"] & & {} \\
{} & & {} \arrow[ll, "\mathsf{AES}(pwd, g^y)"'] \\
K = g^{xy} & & K = g^{xy}
\end{tikzcd}$$

Jablon proposes a variation of EKE in 1996 which works as follows. Let pwd be the password, and let Alice and the server choose $x, y \leftarrow_{\mathcal{R}} \{1, \dots, p-1\}$ respectively, Alice sends pwd^x to the server which answers with the message pwd^y. Then both share the key pwd^{xy}.

$$\begin{tikzcd}
\text{Client } (pwd) & & \text{Server } (pwd) \\
x \leftarrow_{\mathcal{R}} \{1, \dots, p-1\} & & y \leftarrow_{\mathcal{R}} \{1, \dots, p-1\} \\
{} \arrow[rr, "pwd^x"] & & {} \\
{} & & {} \arrow[ll, "pwd^y"'] \\
K = pwd^{xy} & & K = pwd^{xy}
\end{tikzcd}$$

12.2 Protocols

Next we will see in some detail how authentication is done in practice.

12.2.1 SSL/TLS

SSL/TLS is a protocol for providing secure (i.e., secret and authenticated) communications over the Internet. SSL 1.0 was first proposed in 1994 by Netscape Communications and subsequently improved. In 1999 the current version 3.0 was renamed (with some very small changes) to TLS 1.0, causing some confusion with names and versions. In the following, we use the term SSL, unless stated otherwise.

The SSL protocol first exchanges a symmetric encryption key using slow public key operations, then the payload data is encrypted using the faster symmetric encryption with the key they both parties previously agreed on. Depending on the protocol needs, SSL can provide single-sided authentication or mutual authentication. This is achieved using certificates; thus authenticated parties are required to maintain certificates. SSL supports, amongst other algorithms, PKCS#1 public-key encryption and digital signatures, Diffie-Hellman key exchange, the symmetric ciphers AES and 3DES, and the hash function SHA-1. It also supports primitives that are nowadays considered outdated such as (single) DES and MD5, but one can (and should) avoid using them. The SSL protocol has three phases:

1. Negotiation of supported algorithms,
2. Exchange of a session key by means of public-key encryption and authentication based on certificates, and
3. Payload encryption based on a symmetric cipher and the session key.

Overall Protocol A client starts a session by sending a client_hello message including a random string *c-rand* of length 28 bits, a session id, and specifying the cryptographic primitives it supports in *cipher-specs*. The server responds with a server_hello message, including another random string *s-rand*, the session id, and specifying the primitives that will be used in the sequel from the ones specified by the client.

$$\text{Client} \xrightarrow{(\mathsf{client\_hello}, \textit{session-id}, \textit{c-rand}, \textit{cipher-specs})} \text{Server}$$
$$\text{Client} \xleftarrow{(\mathsf{server\_hello}, \textit{session-id}, \textit{s-rand}, \textit{cipher-spec})} \text{Server}$$

To be able to verify a certificate the parties need to support the cryptographic primitives that where used, e.g., RSA signatures or DSA signatures. Thus, it might be necessary to have a variety of certificates using different primitives. Now that both parties agreed on the parameters, they exchange a suitable certificate. For most applications it is sufficient that only the server authenticates to the user, in which case only the server sends its certificate to the user. Currently all certificates follow the X.509 standard, but there exists a draft for using OpenPGP-based certificates.

$$\text{Client} \xleftarrow{(\mathsf{server\_cert}, \textit{cert})} \text{Server}$$

In the next step the key-exchange takes place. There are different options how to exchange a key; two of them we will discuss later. In either case, after finishing this step both parties share a so-called *pre-master secret* (PMS), a 40 bit secret key which is then used to derive the master key.

$$\text{Client} \xleftrightarrow{\text{“\textit{key-exchange}”}} \text{Server}$$

This pre-master secret is then used to derive a so-called master-secret as $\mathsf{H}(PMS \parallel \textit{c-rand} \parallel \textit{s-rand})$. This master-secret is then used to derive the (symmetric) encryption and authentication keys as needed. Finally, both parties acknowledge the successful key-exchange.

$$\text{Client} \xleftarrow{(\mathsf{finish})} \text{Server}$$
$$\text{Client} \xrightarrow{(\mathsf{finish})} \text{Server}$$

Key Exchange I: Encrypted Secret A very simple variant for realizing the key exchange is the following: the client chooses a random bit string of 48 bits, encrypts it with the servers public key (which is known from the certificate), and sends the resulting ciphertext to the server.

This is reasonable fast, as only one public-key operation needs to be performed by each party. However, it does not provide forward secrecy, i.e., if at some point the servers key is compromised, an attacker can use it to obtain all past session keys and thus to read any communication of the server with any client. This is of course not optimal, and there are better ways to do it.

$$\text{Client} \xrightarrow{(\mathsf{client\text{-}key\text{-}exchange}, c)} \text{Server}$$

Key Exchange II: Ephemeral Diffie-Hellman (EDH) The following key-exchange provides forward secrecy, i.e., if at some time the servers key gets compromised, then an adversary can only read future communication using this key. Hence, if the attack was noticed, one can simply avoid using this key and almost no harm is done. This improvement, however, comes along with a running time which is three times slower than the previous method.

First, the server chooses a random 1024 bit prime p and a generator g of $\mathbb{Z}_p^*$. It furthermore chooses $a \leftarrow_{\mathcal{R}} \{1, \ldots, p-1\}$, and computes $z_1 = g^a \bmod p$. Then it signs (p, g, z_1) yielding *sig*, and sends this to the client. The client in turn verifies the signature *sig*, picks a random $b \in \{1, \ldots, p-1\}$, computes $z_2 = z_1^b \bmod p$, and sends it back to the server

$$\text{Client} \xleftarrow{\quad (\mathsf{server\text{-}key\text{-}exchange}, p, g, z_1, sig) \quad} \text{Server}$$

$$\text{Client} \xrightarrow{\quad (\mathsf{client\text{-}key\text{-}exchange}, z_2) \quad} \text{Server}$$

Then, both the client and the server can compute the pre-master secret $PMS := g^{ab}$. This is almost a Diffie-Hellman key exchange, but the server's signature prevents an attacker from doing a man-in-the-middle attack, as he cannot sign the value z_1.

Improving Performance: SSL Resume A nice feature of SSL is the so-called *SSL resume*, which improves performance by avoiding unnecessary key-exchange operations. If a client and a server have communicated before using the session id *CSID*, then the client simply sends a *client-hello* message with this session id:

$$\text{Client} \xrightarrow{\quad (\mathsf{client\_hello}, \mathit{CSID}, \mathit{c\text{-}rand}, \mathit{cipher\text{-}specs}) \quad} \text{Server}$$

If the server can find this session in his session cache, then he answers with the same *CSID*, otherwise with a fresh session id.

$$\text{Client} \xleftarrow{\quad (\mathsf{server\_hello}, \mathit{CSID}, \mathit{s\text{-}rand}, \mathit{cipher\text{-}spec}) \quad} \text{Server}$$

$$\text{Client} \xleftarrow{\quad (\mathsf{server\_hello}, \mathit{new\text{-}session\text{-}id}, \mathit{s\text{-}rand}, \mathit{cipher\text{-}spec}) \quad} \text{Server}$$

If resume was successful, both the user and the server reuse their previous PMS and send an acknowledge message.

$$\text{Client} \xleftarrow{\quad (\mathsf{finish}) \quad} \text{Server}$$

$$\text{Client} \xrightarrow{\quad (\mathsf{finish}) \quad} \text{Server}$$

The master-secret, which is then used to actually derive keys from, is generated as $\mathsf{H}(\mathit{PMS} \parallel \mathit{c\text{-}rand} \parallel \mathit{s\text{-}rand})$.

SSL in Practice Let us briefly talk about networking problems of SSL that occur in practice. Commonly, big web sites are designed as follows: First, a server side cache tries to answer common queries to static data. Then a load balancer balances traffic between different web servers; the web servers are finally connecting to a database.

First of all, for using SSL one needs to install the keys on all web servers, which is not optimal from a security point of view, as this increases the risk of key exposure. Furthermore, the prevalent choice of RSA encryption is not optimal, as the web servers have to perform RSA decryption operations that are more time-consuming than RSA encryption operations. Currently, there are strong efforts in speeding up SSL, e.g., by using specialized hardware etc.

Another point is that caching as described is not possible if the content is encrypted, as the cache cannot see what is inside the packets. This, however, cannot be avoided with encrypted content.

One more problem occurs for web servers that host different domains at the same IP address. In this setting, the web server distinguishes the domains by the HTTP headers. However, with SSL the user and server exchange certificates before any HTTP request was made, i.e., the server cannot know the certificate for which domain it should provide. This was fixed with TLS 1.1 by including the host address in the client_hello message.

Another problem arises when encrypted communication passes a proxies server. Here get-requests are first sent to a proxy, which forwards them to the server. If SSL is turned on, the proxy does not know where to connect to, as the server's name is encrypted. For this reason HTTPS permits an additional first message which contains the target hostname as parameter; then normal SSL messages are exchanged.

User Authentication If user authentication is desired, the protocol slightly deviates from what was stated above: In the server-hello message the server includes a directive cert-req and a list of accepted certificates.

$$\text{Client} \xleftarrow{\quad(\mathsf{server\_hello}, \ldots, \mathsf{cert\text{-}req},\ \textit{list of accepted CAs})\quad} \text{Server}$$

Then the client sends the appropriate certificate to the server.

$$\text{Client} \xrightarrow{\quad(\mathsf{client\_cert}, \textit{cert})\quad} \text{Server}$$

Then, after the remaining protocol, the client signs all protocol data with his secret key and sending it to the server, which verifies the signature.

$$\text{Client} \xrightarrow{\quad(\mathsf{cert\_verify}, \mathsf{S}(sk, \textit{All protocol data}))\quad} \text{Server}$$

12.2.2 Kerberos

Kerberos is a well-known system for symmetric key exchange over a central authority. It originates from the "Athena" project at MIT for secure distributed systems, from around 1987, and is more or less deprecated nowadays. It is the name for a protocol standard as well as for a library of commands. It uses the term "tickets" for server messages, that can be used to derive session keys for the users.

12.2.3 Pretty Good Privacy (PGP)

Probably the best-known program for encrypting mails and files in the private sector, as it is free-of-charge for private use. It has various applications for encrypting emails and files, and also provides transparent encryption of the file system (PGP-Disk).

It was published first in 1991 by Phil Zimmermann. At that time strict US export regulations prevented the export of cryptography with keys larger than 40 bits. To circumvent this restriction he printed a book which contained the whole source-code of the program, and distributed this book around the world, which surprisingly was legal. Then other people scanned in the source code and re-compiled the program. Fortunately, nowadays the export regulations are gone, and the program is available freely. PGP can be used with a variety of algorithms, including RSA and ElGamal for asymmetric encryption, RSA and DSA digital signatures, symmetric encryption algorithms IDEA and 3DES, and hash functions MD5, SHA-1, and SHA-256.

12.2.4 IPSEC (for IPv6)

IPSEC is a cryptographic protocol for securing IP communication and key exchange. It operates on the network layer, i.e., lower than the aforementioned protocols, so it is more generally usable. Even the usual headers can be authenticated. However, header fields such as hop-count may change when a packet is sent trough the Internet; these fields are set to 0 before headers are authenticated. Two modes are supported by IPSEC: *Transport mode*, where only payload data is encrypted, and *tunnel mode*, where the whole package is encrypted, even the header, and new headers are added.

12.2.5 SSH ("Secure Shell")

The first version of SSH, also called SSH-1, was designed in 1995 by Tatu Ylönen from the University of Helsinki as a response to a password-sniffing attack at his university. It provides a replacement for protocols such as telnet and rlogin, which transmit passwords in cleartext and which are thus highly vulnerable to sniffing attacks. In 1996 SSH-1 was improved significantly, yielding the currently used version SSH-2. Some of these changes, however, caused both versions to be incompatible. However, given that successful attacks against SSH-1 have been discovered, this version should not be used any more anyway. While SSH is known best for its secure shell, it also provides other tools such as SCP as a replacement for the insecure "remote copy" (rcp) command, and options for tunneling, e.g., X11 sessions.