Protocols I

Computer Security Lecture 7

Mike Just[1]

School of Informatics
University of Edinburgh

1st February 2010

[1] Based on original lecture notes by David Aspinall

Outline

Introducing protocols

Simple authentication
Password security

Authentication with shared keys
Simple shared-key authentication
Challenge and response
Timestamps

Protocols and attacks

- A **security protocol** is a sequence of communications that two or more *principals* undertake to achieve a security objective.
  - Protocols may be **1-pass** or **multi-pass**.
- Principals: people, organizations, systems, . . .
- The objective may be **authentication**, **exchange of secrets**, or some larger task.
  - Authentication may be **unilateral** or **mutual**.
- Protocols have been one of the richest areas of study in computer security research (design, verification, and breaking).
- Part of the interest comes because protocols can be carefully designed, yet still have surprising flaws. A “flaw” means that the protocol can be **attacked** in a way that the designer did not intend or imagine.
- This lecture introduces some simple protocols and common flaws.

Understanding Protocols

- To understand a protocol, you need to *think carefully* about the underlying assumptions, the initial setup and what happens at each stage.
- Usual assumptions include:
  - **secrets** are known only by those intended
  - the **Dolev-Yao** model of an attacker: may read, delete, copy and synthesise messages (but not, e.g., break encryption)
- At each step in the protocol, the **beliefs of participants** change. If something goes wrong, the protocol is aborted.
- This reasoning can be made formal with specialised logics and calculi for reasoning about protocol correctness. Formal protocol analysis has been a big success, uncovering flaws in real protocols that had been hidden for many years.

Authentication Protocols

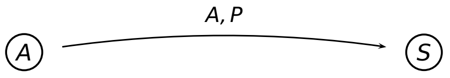
- Authentication protocols are a common type of protocol familiar to most users
- The protocols are often characterized based upon the *thing* used to achieve successful authentication
  1. “Something you are” including **biometrics** such as fingerprints, iris scans, face recognition, key typing behaviour, . . .
  2. “Something you have” including **hard tokens** such as smartcards and mobile phones, as well as **soft tokens**, or files of information used for authentication
  3. “Something you know” including passwords, PINs, passphrases, challenge questions, . . .
- When multiple (independent) methods are used simultaneously, it is called a **multi-factor** authentication protocol

Password authentication

- The most common protocol most users know is *logging in* to a computer system, by giving a **username** and **password**.
- There are two principals involved: Alice (A) and the server (S). Alice sends the server her login name `alice` and password `b1aZfa9s`. In protocol notation,

  $$A \rightarrow S: \quad A, \;\; P$$

  Alice’s (login) name is also written as A, and P stands for her password. Diagrammatically:



- The server then verifies Alice’s password, and if it is correct, it lets her in to the system.

Password authentication - Points of Attack

- At Alice
  - Shoulder surfing (visual/video or auditory/audio)
  - Social engineering (e.g., phishing)
  - Server impersonation to Alice (phony web site, or fake login device)
- On communication channel
  - Eavesdropping
- At the Server
  - Impersonation (online guessing)
  - Denial of Service (DoS)
- At the Server's database
  - Theft
  - Offline guessing
  - Alter information
  - Delete information

Understanding password authentication

- Initial assumptions: Alice's password is a *shared secret*: it is known only to Alice and the server and neither party reveal the secret to anyone else.
- Provided the protocol is secure so that this confidentiality is maintained while she logs in, then an attacker cannot ever learn Alice's password.
- Therefore, a principal demonstrating knowledge of Alice's password to the server will authenticate themselves as Alice to the server.
- This is the only step in this one-message protocol.
- But, there are questions:
  - But how does the server verify her password?
  - And how does her password get sent to the server?

Password Security

- The server may keep a *password file* of user names and plaintext passwords, and use lookup:

  ```
  alice   b1aZfa9s
  ```

  Vulnerability: file stolen $\Longrightarrow$ all passwords break.
- Improvement: use a **one-way hash function** h. This is a cryptographic primitive that acts as a "digital fingerprint".
- The server stores hashes of passwords, not plain texts. To verify Alice's password, the server checks $h(P)$ against the stored hash $h(P_0)$. If the two match, then (almost certainly), $P = P_0$.

  ```
  alice   VUhUKC1OTUzKSVUoKE3Ky
  ```

- This is more secure than before: anyone who reads the file does not immediately learn all passwords.

Better password security

- Password files containing hashes are still vulnerable. For an 8 character ASCII password, brute-force attack needs to check about 2^{53} (10^{16}) combinations. Inconvenient but not infeasible.
- More convenient are **dictionary attacks**, a form of *intelligent search* which reduces search space to mere millions (2^{20}). Work by exploiting the propensity of users to pick passwords related to common words, personal details, etc. Nowadays, *password crackers* are used to prevent a variety bad choices.
- Dictionary attacks which precompute many hash values can be thwarted by **adding salt** to passwords. Salt is a random number that is combined with the password before applying the hash, and stored along with the result. Still doesn't stop a determined attack on a single password.

Securely Communicating the Key

- Password authentication has another vulnerability: the plaintext password must either be sent along a **secure channel** before it is verified, or be **unique** at each protocol instance to prevent eavesdroppers learning it.
- **One-time passwords** can be used to provide uniqueness as per the following three options
  1. The user could be assigned a sheet of paper with 100 one-time passwords: $P_1, P_2, \ldots, P_{100}$.
  2. (Lamport *or* S/Key) Server initially stores $h^{100}(P)$. At *round* i, the server will have $h^{n-i+1}(P)$, and Alice submits $h^{n-i}(P)$. The server verifies with the store value by computing $h(h^{n-i}(P))$ and if correct, stores $h^{n-i}(P)$.
  3. (SecureID) Uses a token device which computes a new "password" every minute by computing a hash of the current time (to a granularity of minutes), and a secret key stored on the device.

Simple shared-key authentication

- With an unsecured channel, we may instead use *shared keys*.
- Suppose that we have an in-car device C, which is a congestion-charging transmitter in a car window screen. C wishes to identify itself to a server S in an overhead charging point, which clocks cars passing. Suppose that C and S share a secret key K_{cs}. The in-car device might send a message with its name (a secret serial number) and an encrypted copy of its name, perhaps with some additional relevant data R (e.g., the time since it last passed a charge-point).

Protocol for shared-key authentication

$$C \to S: \quad C, \{ C, R \}_{K_{cs}}$$

(the notation $\{ C, R \}_{K_{cs}}$ stands for the combination of C and the rest R encrypted under the key K_{cs}).

- How does this work?
  1. S uses plaintext name C to find the key it shares with C, K_{cs}.
  2. S attempts to decrypt the rest of the message using K_{cs}. If successful, S concludes that C is the device it is claiming to be.
- Why is C duplicated in the message?
  - This prevents a **reflection attack**. If the protocol worked the other way around as well, it would prevent message being re-used immediately by an adversary, on C.
- Are there any other problems?
  - It is vulnerable to **replay attacks**. A device which captures and replays messages from windscreen beamers, they could rack up huge charges on another bill!

The need for nonces

- To prevent a replay attack, we need a method to ensure that messages are **fresh**.
- We can do this using **nonces** (“number used once”).
- A nonce is a random number or a sequence number. The server S maintains a list of messages it has seen (or if the nonce is a sequence number, just the last value), and ignores those that have gone before.

Remembering nonces

With a nonce N in the protocol, we now have:

$$C \to S: \quad C, \{ C, N, R \}_{K_{cs}}$$

- This works, but has engineering drawbacks. The server S must remember a reasonable history of past messages, or the last value of the counter.
- But the counter may be distributed or may get incremented several times during faulty transmissions, etc. Can we remove the need to remember nonces?
- A solution is to introduce a two-way communication, based on **challenge and response**.

Challenge and response

Now the nonce is generated randomly by the server, and neither side needs to keep any (long-term) state:

Message 1. $S \to C: \quad N$
Message 2. $C \to S: \quad C, \{ C, N, R \}_{K_{cs}}$

- Many protocols are based on this basic challenge-response idea, using nonces to guarantee freshness.
- But challenge-response protocols are open to another form of attack, the **man-in-the-middle attack** (or to be politically correct, the **middleperson attack**).

Man-in-the-middle attacks

- In the car congestion charging scenario, suppose somebody builds a device which attaches to their back windscreen and charges the car behind them as they pass the barrier, simply by passing the communications back and forth.
- To show this explicitly, let M be the middleperson:

Message 1. $S \to M: \quad N$
Message 1’. $M \to C: \quad N$
Message 2. $C \to M: \quad C, \{ C, N, R \}_{K_{cs}}$
Message 2’. $M \to S: \quad C, \{ C, N, R \}_{K_{cs}}$

- The charges are passed to the car behind!
- Notice that M here is particularly stupid, and needs to understand nothing in the transmitted messages.

Foiling man-in-the-middle

- Man-in-the-middle attacks as passive and direct as the simple case above can be difficult to foil: the server on the overhead charging point may have no way of telling that it is not talking directly to the car that’s actually passing.
- One approach: **timestamps** instead of nonces, and check that messages are sent within tight time constraints. But in this case we would probably rely on other techniques, e.g. secondary authentication by number plate recognition, or at the least, good recording mechanisms so that *accountability* is maintained (somebody later questions their bill).
- Other middleperson attacks are more sophisticated, e.g., typically the middle person taking an active role in decrypting and re-encrypting messages. Some of these other attacks do have defences in protocols.

Timestamps

- Using **timestamps** in place of nonces provides timeliness and uniqueness guarantees, preventing replay. Additionally, they may provide time-limited access privileges, or detect forced delays.
- General method: A generates a timestamp T_a from her local clock, and binds it cryptographically in a message sent to B. On receipt, B compares the time against his own local clock, and accepts the message if (1) the difference is within some *acceptance window*, and optionally (2) no identical timestamp has previously been received.
- Pros: reduced number of messages; no requirement to maintain (possibly pairwise) state information.
- Cons: clocks are required to be (“loosely”) synchronized; state required for storing observed timestamps; synchronization itself may require secure authenticated protocols. . .

References

Interesting treatments of security protocols are given in Chapter 2 of Anderson and Chapters 3–5 of Schneier.

Ross Anderson.
Security Engineering: A Comprehensive Guide to Building Dependable Distributed Systems.
Wiley & Sons, 2001.

Bruce Schneier.
Applied Cryptography.
John Wiley & Sons, second edition, 1996.

Recommended Reading

Chapter 2 of Anderson.