INF226 – Software Security

Håkon Robbestad Gylterud

2019–09-18

Authentication

Authentication

Authentication is the act of verifying the identity of actors in the system.

Authentication

Authentication is the act of verifying the identity of actors in the system.

1. `A` is communicating with `B`. (Un authenticated)
2. `A` knows that they are communicating with `B`. (Authenticated)

Authentication

Authentication is the act of verifying the identity of actors in the system.

1. `A` is communicating with `B`. (Un authenticated)
2. `A` knows that they are communicating with `B`. (Authenticated)

Examples

- **Certificates** issued by a Certificate Authority (CA) authenticates websites visted over HTTPS by the browser.
- **Passwords** authenticate the user when logging in to a computer.
- A **shibboleth** authenticates a member of a group, in social settings.

Passwords

Passwords

What are the pros and cons of passwords as an authentication mechanism?

Passwords

What are the pros and cons of passwords as an authentication mechanism?

What were Dr. Cranor's conclusions?

Password alphabet size vs. password length

A quick "back of the napkin" calculation shows that there is not much to gain by increasing the size alphabet:

- Incresing the alphabet is equivalent to increasing the length by a constant factor which *grows logarithmically in the size in the size of the alphabet*.

BUT: There is also no reason to make artificial restrictions on what character's the users are *allowed* to use.

NIST guidelines for passwords

The complicated requirements mentioned by Dr. Cranor have been deprecated in the latest guidelines, in favour of a more simpler:

- Require a minimum password length.
- The minimum length requirement must be 8 characters or greater.
- Allow at least 64 characters.
- Check against a list of known bad passwords. For instance:
  - Dictionary words.
  - Repetitive or sequential characters (e.g. ‘aaaaaa’, ‘1234abcd’).
  - Context-specific words, such as the name of the service, the username, and derivatives thereof.
  - Passwords obtained from previous breach corpuses.

Storing passwords

Have I been pwned?

`https://haveibeenpwned.com/`

Cryptographic hash functions

`foobar` $\rightarrow$ `aec070645fe...`
`foobat` $\rightarrow$ `c7f0f45765b...`

Requirements of a cryptographic hash function

- **One-way**: Given y, difficult to find x such that $h(x) = y$.
- **Collision free**: Difficult to find x and x' such that $h(x) = h(x')$.
- A small change in input yield a large difference in output.
- Quick to compute.

Cryptographic hash functions

`foobar` $\rightarrow$ `aec070645fe...`
`foobat` $\rightarrow$ `c7f0f45765b...`

Requirements of a cryptographic hash function

- **One-way**: Given y, difficult to find x such that $h(x) = y$.
- **Collision free**: Difficult to find x and x' such that $h(x) = h(x')$.
- A small change in input yield a large difference in output.
- Quick to compute.

Examples:

- MD5, and SHA1 has known collisions
- SHA256/512 and SHA3 has no known collisions

Uses of hash functions

- Checksumming transferred data
- Data identifier
- **Hashing passwords**
- Signature generation/verification
- Building other cryptographic primitives

Hashing passwords

Easy (but not recommended) way to verify passwords without storing the password itself:

- Given password x, store $h(x)$.
- When the user logs in with password y, check that $h(y) = h(x)$ and conclude $x = y$.

If the application database is leaked, only hashes of passwords are disclosed.

Issues with this strategy?

Issues with hashing password

- If same password is reused, hashes will be the same.
- Hashes can be computed efficiently for a dictionary of passwords.
- An attacker can use the hash to *brute force* the password

Rainbow tables

Rainbow tables refers to a time–space-tradeoff when creating a **lookup table for hash values** $\rightarrow$ **plaintext**.

Expositions:

- https://en.wikipedia.org/wiki/Rainbow_table
- http://kestas.kuliukas.com/RainbowTables/

Salting

Efficient solution to make rainbow tables / hash dictionaries infeasible.

In stead of storing $h(x)$, generate a random byte-string s and store $s, h(h(x) \oplus s)$.

How much salt?

Contemporary unix-like systems use 128-bits salts.

Salting does not help against a brute-force attack on a single password.

Key derivation functions

Fact of life: Users chose passwords with low entropy.

Idea: What if we made computing the hash really expensive?

If each attempt as guessing is expensive, it will be more difficult to guess the password.

Key derivation functions

Requirements for key derivation functions:

- One-way
- Collision free
- A small change in input yield a large difference in output.
- **CPU intensive**
- **Memory expensive**
- **Sequential** (difficult to parallelize)

A naïve key derivation scheme

In stead of storing $h(x)$, generate two random byte-strings s_1 and s_2 and store $s_1, h(h(h(x) \oplus s_2) \oplus s_1)$.

Now both an attacker and a legitimate login function must guess s_2.

The length of s_2 works as a cost parameter. s_1 is just regular salt.

Problem with this approach?

SCrypt

Introduced by Colin Percival in 2009, for his Tarsnap back-up service.

Sources:

- RFC 7914
- https://www.tarsnap.com/scrypt.html

(Compare to Argon2)

SCrypt

The previous key derivation scheme is trivially computed in parallel, at no additional memory cost.

SCrypt is a key derivation function which is maximally memory hard.

However, its use in crypto-currencies means that there has been developed quite fast specialized circuits for scrypt.

SCrypt

Parameters:

- r block size parameter
- N CPU/Memory cost parameter (a power of two)
- p parallelism parameter (affects CPU cost, not memory)

Other password guessing prevention measures

- Rate-limiting password attempts
- Proof-of-work form the client

Two-factor authentication

Two-factor authentication

The idea: Introduce an additional authentication mechanisms in addition to passwords.

Two-factor authentication

The idea: Introduce an additional authentication mechanisms in addition to passwords.

Examples:

- SMS codes (considered insecure: Example Reddit developers hacked via SMS intercept)
- Print-out with one-time codes.
- A device with time-based, one-time passwords (TOTP)
- Approval from an already authenticated device (Example: Keybase)
- Public key cryptography (U2F / FIDO , WebAuthn).

Two-factor authentication

Current status:

- More and more services use multiple factors.
- Many two-factor systems vulnereable to phishing $\rightarrow$ malicious proxy attacks (Modlishka is one such proxy).
- Public-key systems integrated with the browser can (in theory) prevent proxy attack.
- WebAuthn is a new (March 2019) W3C standard.

Password recovery

What about when the user forgets their password? Or looses their second factor?

Public key cryptography and authentication

Public key cryptography

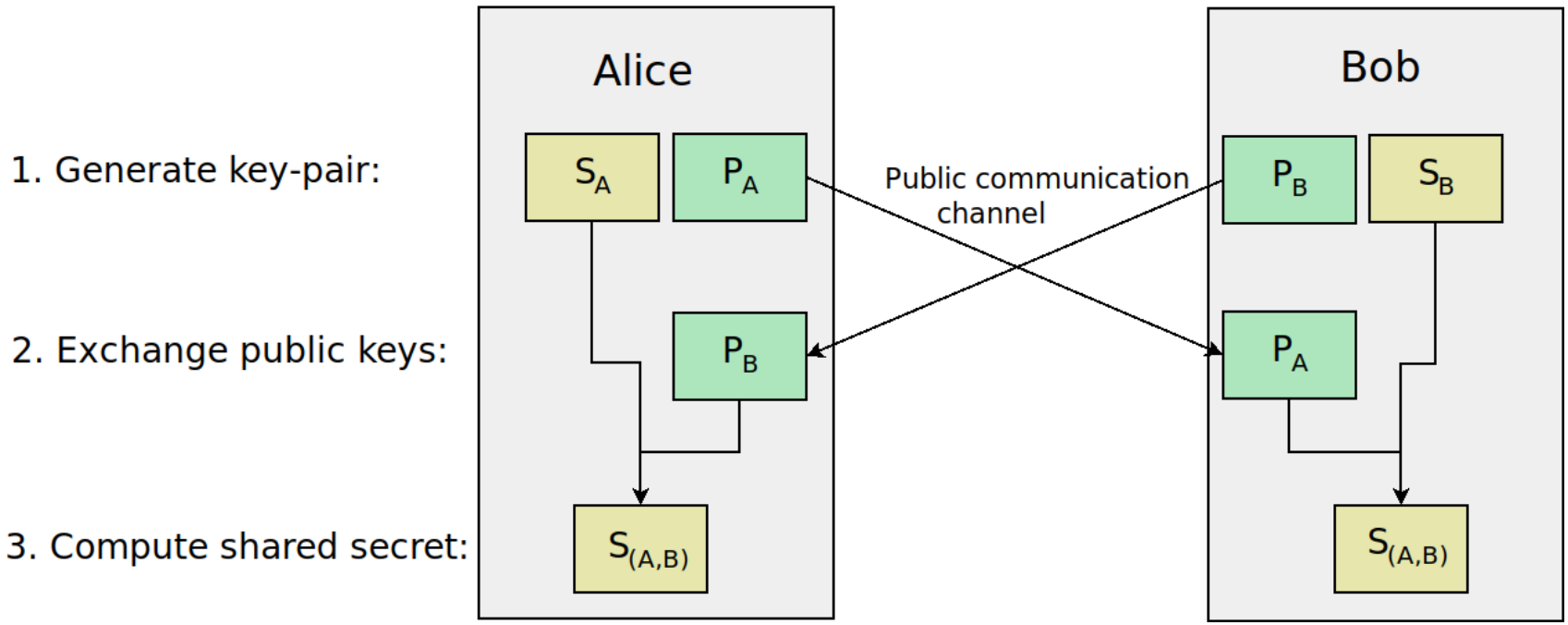

Figure 1: Public key cryptography

Man-in-the-middle attacks

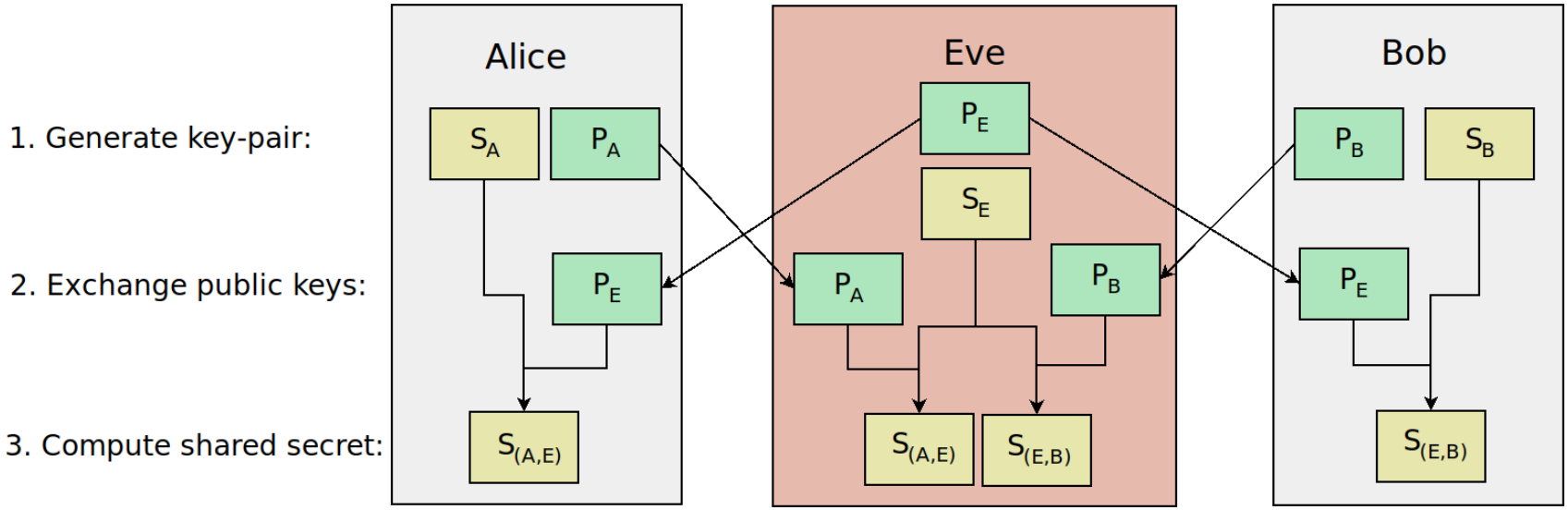

Figure 2: Public key cryptography

Trust upon first use

Assumption: The man in the middle does not strike the first time.

Trust upon first use

Assumption: The man in the middle does not strike the first time.

Mechanism: Trust the public key used in first session. Use that for authentication of later sessions.

Trust upon first use

Assumption: The man in the middle does not strike the first time.

Mechanism: Trust the public key used in first session. Use that for authentication of later sessions.

Works well for long-lasting trust-relationships. Or when no existing trust relationship exists (i.e. web-site registration).

Centralized Certificate Authorities

Assumption: We trust a central authority to verfiy public keys for us.

Mechanims: Central authority verifies identity and issues certificates on public keys.

Examples:

- Browsers ship with a list of public keys of trusted Certificate Authorities.
- Organsations can distribute their own certificates for internal use.

Other schemes

For peer-to-peer authentication:

- one can use preexisting shared secrets (Example: Socialist Millionaire protocol)
- out-of-band communication (verfication of key fingerprints)

Logged in, and then what?

Logged in, and then what?

- User actions are often given in separate requests from the authentication request.
- How do we ensure that each request comes from a valid user?

Example: Webmail

1. `/login`
   - User requests login form, and enters password
2. `/inbox`
   - User posts login details to the inbox page
   - Server responds with inbox, listing messages, after checking password
3. `/delete?messageid=123`
   - User requests a message deleted
   - **How can the server know the user is the same?**

Session IDs

The standard way solution is to use a **session ID**, which identifies the user in the following session.

Requires:

- Entropy: Session ID must not be guessable (random, 128 bits)
- Secrecy: Session ID must not be leaked:
  - HTTPS
  - Debugging modes often leak session IDs
  - Cross-site-scripting (Cookies: `HttpOnly`, `SameSite`).

Common pitfall: Lacking entropy

Special care needs to be taken when generating random salts or secret keys.

- Entropy is a finite resource on any system.
- Not all random number generators are suitable for cryptographic use.

Use the recommended source of randomness on your system!

java.util.Random

- java.util.Random is a Linear Congruential Generator (LCG).

Using `java.util.Random` is a very insecure source of cryptographic randomness:

- By observing only a few bytes of output from an LCG, one can completely determine the rest of the sequence.

(LCGs are well suited for statistical work and Monte-Carlo simulations)

Generating secure random bytes in Java

You can use SecureRandom as a general purpose source of entropy:

Code

```
import java.security.SecureRandom;
...
SecureRandom random = new SecureRandom();

final byte[] token = new byte[32];
random.nextBytes(token);
```

Generating secure random keys in Java

Different ciphers have different `KeyGenerator` implementations in Java. For instance AES:

```
javax.crypto.KeyGenerator;
javax.crypto.SecretKey;
...
KeyGenerator keyGen = KeyGenerator.getInstance("AES");
keyGen.init(256); // Specifying the key-size
SecretKey secretKey = keyGen.generateKey();
```

Structure of a user authentication scheme based on passwords

1. Provide a way for user to authenticate server (ex: HTTPS w/valid certificate)
2. Establish a secure communication channel (ex: HTTPS)
3. User transmits password
4. Server verifies password:
   - Salted (128 bit)
   - Run through an expensive key derivation function (ex: SCrypt)
5. Server responds with a secure session ID
6. Client program stores session ID as securely as possible

Comments:

- Are there alternatives to sending the password to the server?
- Two factor would be better