Security in Networked Computer Systems
OpenSSL Lab Session #2

Pericle Perazzo
pericle.perazzo@iet.unipi.it
http://www.iet.unipi.it/p.perazzo/teaching/

18th March 2015
Hash Functions
The logical representation of a hash algorithm is a function, taking a variable-sized plaintext as input, and returning a fixed-sized digest as output.

Implementing encryption and decryption in this way is not efficient neither practical, for the same reasons explained in the last lesson for symmetric encryption. The majority of cryptographic libraries uses instead incremental functions, which update a hashing context step-by-step. This is done in higher-level languages as well, for example Java and C#.
Hash Functions

Hashing (pseudo-code):

```
1  md_ctx = context_initialize(hash_algorithm);
2  cycle:
3     context_update(md_ctx, plaintext_fragment);
4  end
5  digest = context_finalize(md_ctx);
```

Hash verifying:

```
1  computed_digest = <in the same way as above>
2  compare(computed_digest, received_digest);
```

This slide shows the pseudo-code of an incremental hashing operation. It is very similar to the incremental encrypting operation, except that the context_update function does not return any data. The context_finalize function returns the digest.

The hash verification simply re-computes the digest, and then compares it to the received one. The verification is positive if they are equal.
EVP Hashing Context

- `#include <openssl/evp.h>`
  High-level OpenSSL API functions.
- **EVP_MD_CTX** (data structure)
  A hashing context.
- Context creation:
  ```c
  EVP_MD_CTX* md_ctx;
  md_ctx = malloc(sizeof(EVP_MD_CTX));
  EVP_MD_CTX_init(md_ctx);
  ```
- Hashing:
  ```c
  EVP_DigestInit(md_ctx, EVP_md5());
  EVP_DigestUpdate(md_ctx, inbuf, inlen);
  EVP_DigestFinal(md_ctx, outbuf, &outlen);
  ```
- Context destruction:
  ```c
  EVP_MD_CTX_cleanup(md_ctx);
  free(md_ctx);
  ```

These OpenSSL API functions (in `<openssl/evp.h>`) realize the incremental hash. The buffer passed to EVP_DigestFinal will receive the digest, so it must be sized accordingly.
Utility Functions

- Utility functions for hashing:
  - `EVP_MD_size(EVP_md5());`
    
    Returns the size (in bytes) of a digest (16 bytes for MD5).
  - `int CRYPTO_memcmp(computed_digest, received_digest, digest_len);`

    Compares two portions of memory in constant time. Returns 0 if they are equal. Defined in `<openssl/crypto.h>`.

The `CRYPTO_memcmp()` function (defined in `<openssl/crypto.h>`) is useful for digest checking.

It is NOT safe to use the standard `memcmp()` function to compare two digests, because it makes the system vulnerable to timing attacks. In fact, the runtime of `memcmp()` depends on the inputs: if they differ on the first bytes, the runtime will be short; if they differ on the last bytes, it will be long. An adversary can make the system check several (wrong) digests, and by measuring the runtime each time, she can learn the value of the correct digest. On the contrary, `CRYPTO_memcmp()` has a constant runtime, and it is recommended to check digests.
Other Hash Algorithms

- **EVP_MD** (data structure)
  A hash algorithm.

- **EVP_MD** \* EVP_md5();

- **EVP_MD** \* EVP_sha1();

- **EVP_MD** \* EVP_sha256();
  SHA-256 hash algorithm (part of SHA-2 family). 256-bit digests. Good security.

This slide shows the most common hash algorithms used in OpenSSL. It is recommended to use SHA-256, the other algorithms are obsolete or will become.

MD5 (=Message Digest 5) is an obsolete algorithm, completely broken from the security point of view. A 2013 research showed how to find colliding texts (birthday attack) for MD5 in <1sec of processing time on a common PC. (Theoretical) preimage attacks are known too, even if not realized in practice yet.

SHA-1 (=Secure Hash Algorithm 1) offers medium security, meaning that no attack has been developed in practice, but some theoretical attacks do exist. Microsoft, Google and Mozilla Foundation announced that their software will not accept any more certificates signed with SHA-1 by 2017.

SHA-256 (part of the SHA-2 family) offers good security. No practical neither theoretical attacks are known.
Keyed Hash Functions

logical representation:

plaintext

\[ H_k() \]

digest

- Every hash algorithm can be modified to be keyed.
  - There are HMAC-MD5, HMAC-SHA1, HMAC-SHA256, etc.
  - The HMAC digests have the same size of the basic hash algorithm digests.
- There are no constraints on the key length, but keys of the same length of the digests are recommended.
  - Shorter keys are discouraged, as they make the authentication weaker.
  - Longer keys do not significantly increase security.

Within security applications, **keyed hash algorithms** (HMAC) are more useful than “pure” ones, because they are used for authenticating communications. The logical representation of a keyed hash algorithm is a function, taking a key and a variable-sized plaintext as input, and returning a fixed-size digest as output.

The majority of cryptographic libraries uses incremental functions for keyed hash algorithms as well.

Note that HMAC algorithms do not impose constraints on the key length. However, keys of the same size of the digests are implicitly recommended by the HMAC RFC (rfc2104). This is because shorter keys make the authentication weaker, whereas longer keys do not significantly increase security.
Keyed Hash Functions

Keyed-hashing (pseudo-code):

```
1  hmac_ctx = contextInitialize(hash_algorithm, key);
2  cycle:
3      contextUpdate(hmac_ctx, plaintext_fragment);
4  end
5  digest = contextFinalize(hmac_ctx);
```

Keyed-hash verifying:

```
1  computed_digest = <in the same way as above>
2  compare(computed_digest, received_digest);
```

This slide shows the pseudo-code of an incremental keyed hash operation. Note that we have to pass the key to the contextInitialize function.
EVP Keyed-Hashing Context

- `#include <openssl/hmac.h>`
  OpenSSL API functions for keyed-hash.
- `HMAC_CTX` (data structure)
  A keyed-hashing context.
- Context creation:
  ```c
  HMAC_CTX* hmac_ctx;
  hmac_ctx = malloc(sizeof(HMAC_CTX));
  HMAC_CTX_init(hmac_ctx);
  ```
- Keyed-hashing:
  ```c
  HMAC_Init(hmac_ctx, key, keylen, EVP_md5());
  HMAC_Update(hmac_ctx, inbuf, inlen);
  HMAC_Final(hmac_ctx, outbuf, &outlen);
  ```
- Context destruction:
  ```c
  HMAC_CTX_cleanup(hmac_ctx);
  free(hmac_ctx);
  ```

These OpenSSL API functions realize the incremental keyed hash. Note that you have to include `<openssl/hmac.h>`, since HMAC functionalities are not included in the `<openssl/evp.h>` header.
**EVP Keyed-Hashing Context**

- Compute an HMAC on-the-fly (without context):
  
  ```c
  HMAC(EVP_md5(), key, keylen, inbuf, inlen, outbuf, &outlen);
  ```

There is also a function to compute an HMAC on-the-fly, without initializing and destroying the context. This function is useful to simplify the code when the message to be authenticated has a short and fixed size (for example a *nonce*).
Final Exercise

- The client reads a file, and sends it to the server, following the schema:
  \[ E_k(m, H_k(m)), \]
  where:
  - \( m \) is the content of the file
  - \( k \) is a shared key
  - \( E_k() \) is a symmetric encryption function (DES in CBC mode)
  - \( H_k() \) is a keyed hash function (HMAC-MD5)

- The server decrypts the message received from the client, verifies its authenticity, and saves it on a local file.

- The same key must be used for authenticating and encrypting
  - The key must be as long as an MD5 digest: 16 bytes.
  - For DES encryption, only the first 8 bytes of the key must be used.
**Useful Socket Snippets**

- **Include’s:**
  ```
  #include <arpa/inet.h>
  #include <sys/socket.h>
  ```

- **General:**
  ```
  sockaddr_in addr;
  memset(&addr, 0, sizeof(addr));
  addr.sin_family = AF_INET; // IPv6
  addr.sin_port = htons(4444);
  inet_pton(AF_INET, "127.0.0.1", &addr.sin_addr);
  ```

- **Client-side:**
  ```
  sk = socket(AF_INET, SOCK_STREAM, 0);
  ret = connect(sk, (sockaddr*) &srv_addr, sizeof(srv_addr));
  if(ret == -1) ... // connection error
  ret = send(sk, buffer, buffer_len, 0);
  if(ret != buffer_len) ... // communication error
  close(sk);
  ```
Useful Socket Snippets

- Server-side:

```c
lst_sk = socket(AF_INET, SOCK_STREAM, 0);
optval = 1;
setsockopt(lst_sk, SOL_SOCKET, SO_REUSEADDR, &optval, sizeof(optval));
ret = bind(lst_sk, (sockaddr*)&my_addr, sizeof(my_addr));
if (ret == -1) // port opening error
    ...
ret = listen(lst_sk, BACKLOG_SIZE);
if (ret == -1) // strange error
    ...
client_addr_len = sizeof(client_addr);
com_sk = accept(lst_sk, (sockaddr*)&client_addr, &len);
if (com_sk == -1) // communication error
    ...
ret = recv(com_sk, buffer, buffer_len, MSG_WAITALL);
if (ret != <expected number of bytes>) // communication error
    ...
close(com_sk);
close(lst_sk);
```
Replay attack

- The solution adopted in the exercise leaves space for a simple attack called *replay attack*.
- The adversary eavesdrops the communication and then replays it afterwards, pretending to be the legitimate client.

By checking the HMAC, the server is sure that the legitimate client has produced it, but he does not know *when*. It could actually be an old HMAC. In other words, the *freshness* of the HMAC is not guaranteed. This leaves space for a simple attack called *replay attack*. The adversary eavesdrops (i.e. intercepts) the communication and then replays it afterwards, pretending to be the legitimate client.
Replay attack

- To defend against replay attacks, the server must be sure about the freshness of the HMAC.
- A simple (and not too secure) way to do that is to send a timestamp, and include it in the HMAC, with this schema:
  Client → Server: $E_k(m, \text{timestamp}, H_k(m, \text{timestamp}))$.
- The server must check that the timestamp is not too old.

A simple way to guarantee the freshness of an HMAC is to include a timestamp in it. The server checks that the timestamp is not too old, for example (max) 2 minutes ago (timestamp tolerance). In this way, the adversary can replay the communication only after (max) 2 minutes from the legitimate one.

The timestamp-based countermeasure is not very secure nor practical, because it relies too much on the configurations of the server and the client machines. If the server clock and the client clock are misaligned of more than the timestamp tolerance, the protocol will fail. On the other hand, relaxing too much the timestamp tolerance reduces the security.

A better solution requires the generation of a nonce (=number used once) by the server. The server generates a nonce (usually a random quantity) and sends it in clear to the client. The client must include the nonce in the HMAC. Hence, the server is sure of the freshness of the HMAC.
Final Exercise - Extension

• Add a timestamp-based defense against replay attack to the client-server application.

• Use `<time.h>` for the timestamp functions:
  • `time_t timestamp;`
    Data structure representing a timestamp.
  • `timestamp = time(NULL);`
    System clock.
  • `double difftime(timestamp1, timestamp2);`
    Difference between two timestamps (in seconds).