Security in Networked Computer Systems
OpenSSL Lab Session #3

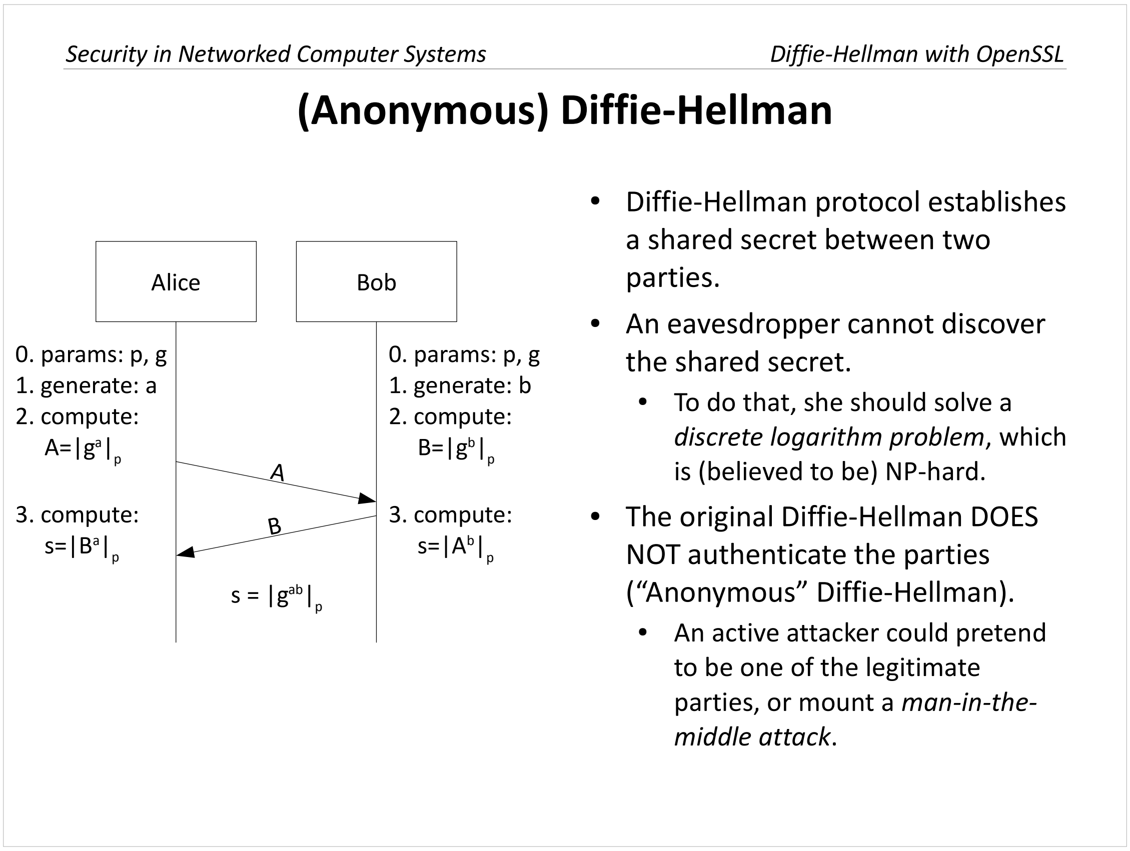
Pericle Perazzo

pericle.perazzo@iet.unipi.it

http://www.iet.unipi.it/p.perazzo/teaching/

25th March 2015

Diffie-Hellman Key Agreement



Security in Networked Computer Systems *Diffie-Hellman with OpenSSL*

(Anonymous) Diffie-Hellman

Alice

0. params: p, g
1. generate: a
2. compute: $A=|g^a|_p$
3. compute: $s=|B^a|_p$

Bob

0. params: p, g
1. generate: b
2. compute: $B=|g^b|_p$
3. compute: $s=|A^b|_p$

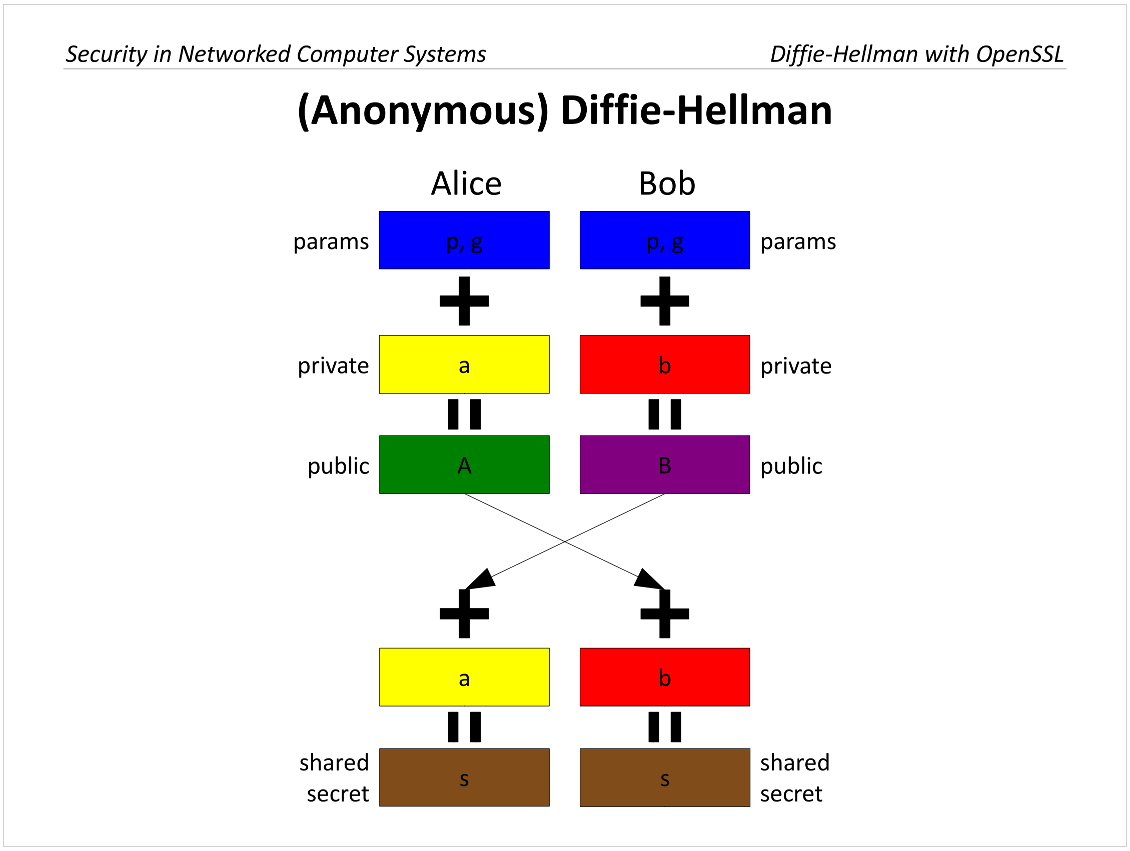
$s = |g^{ab}|_p$

- Diffie-Hellman protocol establishes a shared secret between two parties.
- An eavesdropper cannot discover the shared secret.
  - To do that, she should solve a *discrete logarithm problem*, which is (believed to be) NP-hard.
- The original Diffie-Hellman DOES NOT authenticate the parties (“Anonymous” Diffie-Hellman).
  - An active attacker could pretend to be one of the legitimate parties, or mount a *man-in-the-middle attack*.

Diffie-Hellman protocol (or Diffie-Hellman-Merkle protocol) is a technique which allows two parties to establish a shared secret over a public channel, without using any pre-shared secret. It was published by Whitfield Diffie and Martin Hellman (inspired by the work of Ralph Merkle) in 1976. It has been the first example of public-key cryptography.

In Diffie-Hellman (DH), first the parties agree on two parameters: p (the modulus, a large prime number) and g (the generator, usually 2 or 5). Then, they both generate a pair of quantities, one *private* and one *public*. Alice generates a (private quantity) and $A=|g^a|_p$ (public quantity). Bob generates b (private quantity) and $B=|g^b|_p$ (public quantity). They send the public quantity to the other party. Finally, Alice and Bob independently compute the secret, by $s=|B^a|_p$ (Alice) and $s=|A^b|_p$ (Bob). Diffie-Hellman resists against an eavesdropper adversary, wanting to discover the shared secret. The security is based on the NP-hardness of the discrete logarithm problem.

The original version of Diffie-Hellman (also called “Anonymous” Diffie-Hellman) does not provide for the authentication of the parties. This means that an active adversary can pretend to be one of the parties. Moreover, an adversary could mount a *man-on-the-middle attack*, performing two distinct Diffie-Hellman protocols, one with Alice and another with Bob. She could then transparently decrypt and re-encrypt the messages that Alice and Bob send to each other, thus intercepting the whole communication. These attacks can be avoided by using authenticated versions of Diffie-Hellman, which are currently used in state-of-the-art security protocols (TLS, IPsec, etc.).



Security in Networked Computer Systems *Diffie-Hellman with OpenSSL*

(Anonymous) Diffie-Hellman

Alice Bob

params p, g p, g params

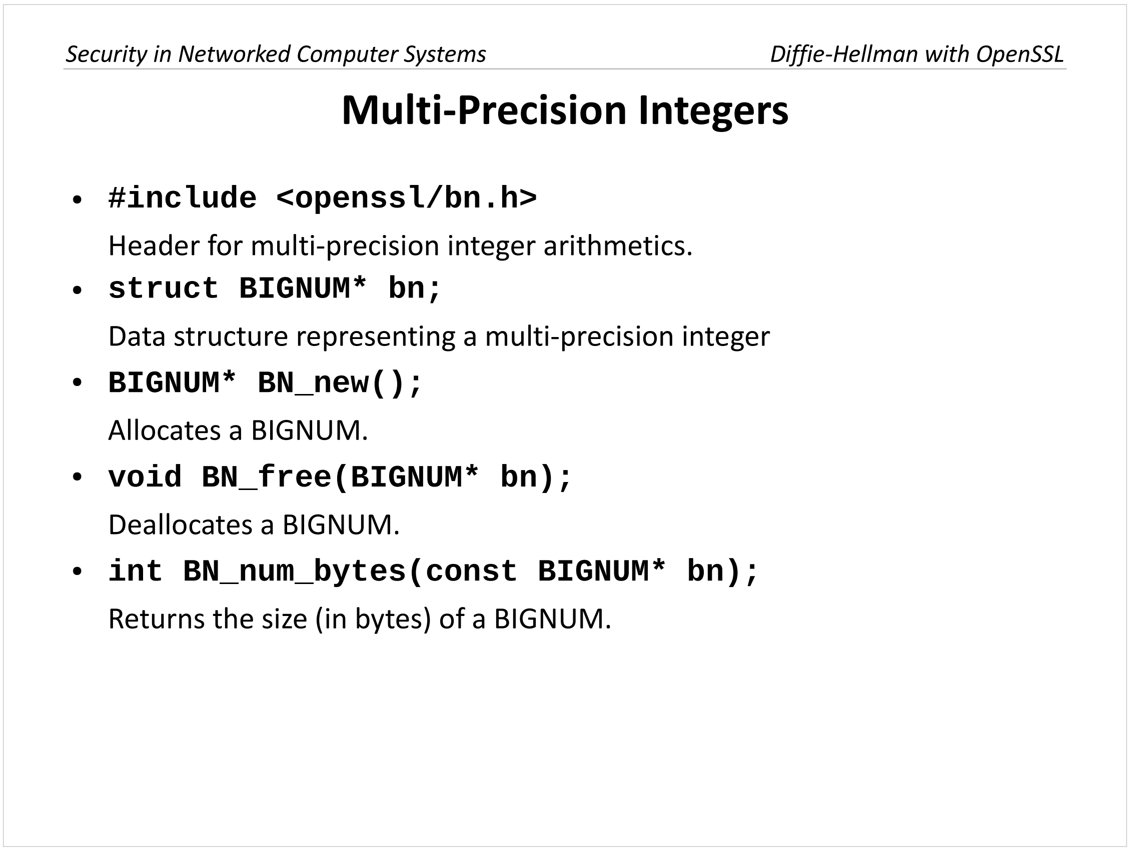
private a b private

public A B public

a b

shared secret s s shared secret

The slide shows a color-based analogy of Diffie-Hellman technique. At the end, the two parties both knows the shared secret (brown color), without knowing the other party's private quantity (yellow and red colors).



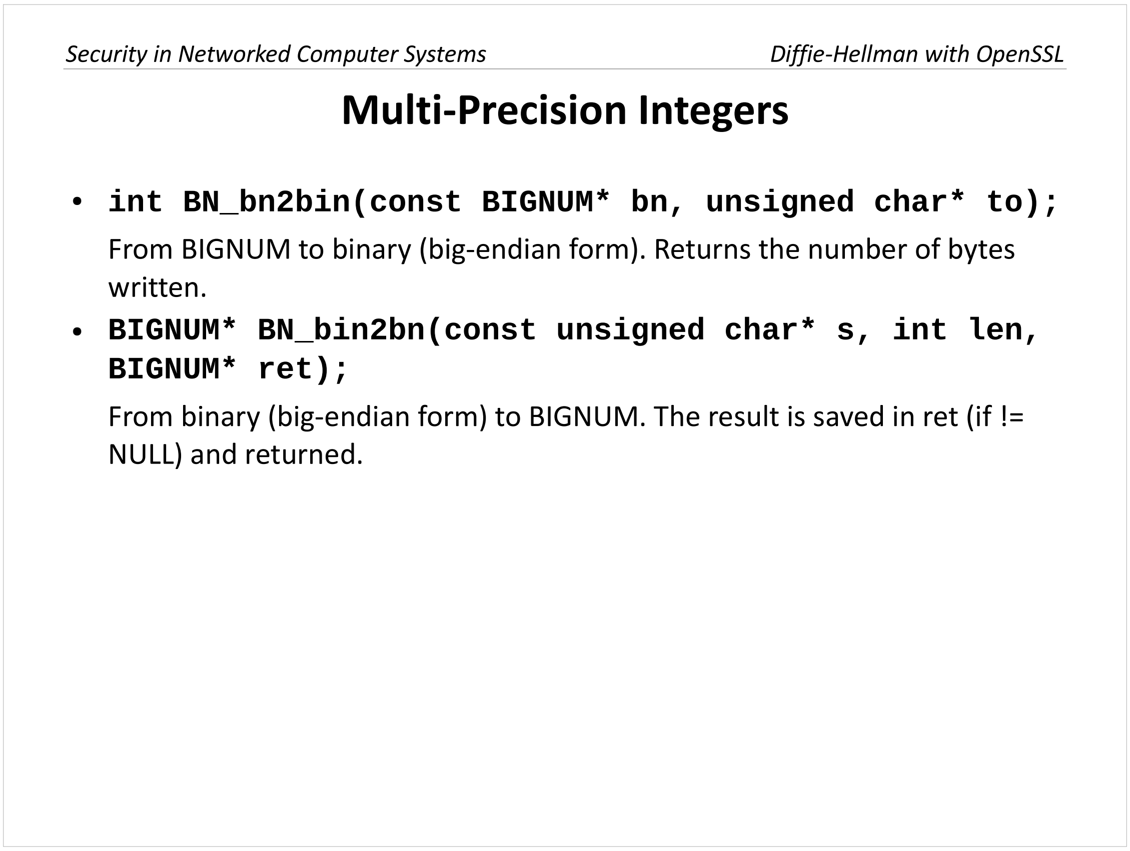
Security in Networked Computer Systems *Diffie-Hellman with OpenSSL*

Multi-Precision Integers

- `#include <openssl/bn.h>`
  Header for multi-precision integer arithmetics.
- `struct BIGNUM* bn;`
  Data structure representing a multi-precision integer
- `BIGNUM* BN_new();`
  Allocates a BIGNUM.
- `void BN_free(BIGNUM* bn);`
  Deallocates a BIGNUM.
- `int BN_num_bytes(const BIGNUM* bn);`
  Returns the size (in bytes) of a BIGNUM.

In order to implement Diffie-Hellman with (low-level) OpenSSL API, we need to use the OpenSSL implementation of *multi-precision integers*. A multi-precision integer is a (signed) integer stored on a variable number of bytes. It can represent an integer of arbitrary size. A multi-precision integer is represented by a BIGNUM data structure (`#include<openssl/bn.h>`).

These API functions perform the basic allocation/deallocation operations on a BIGNUM.



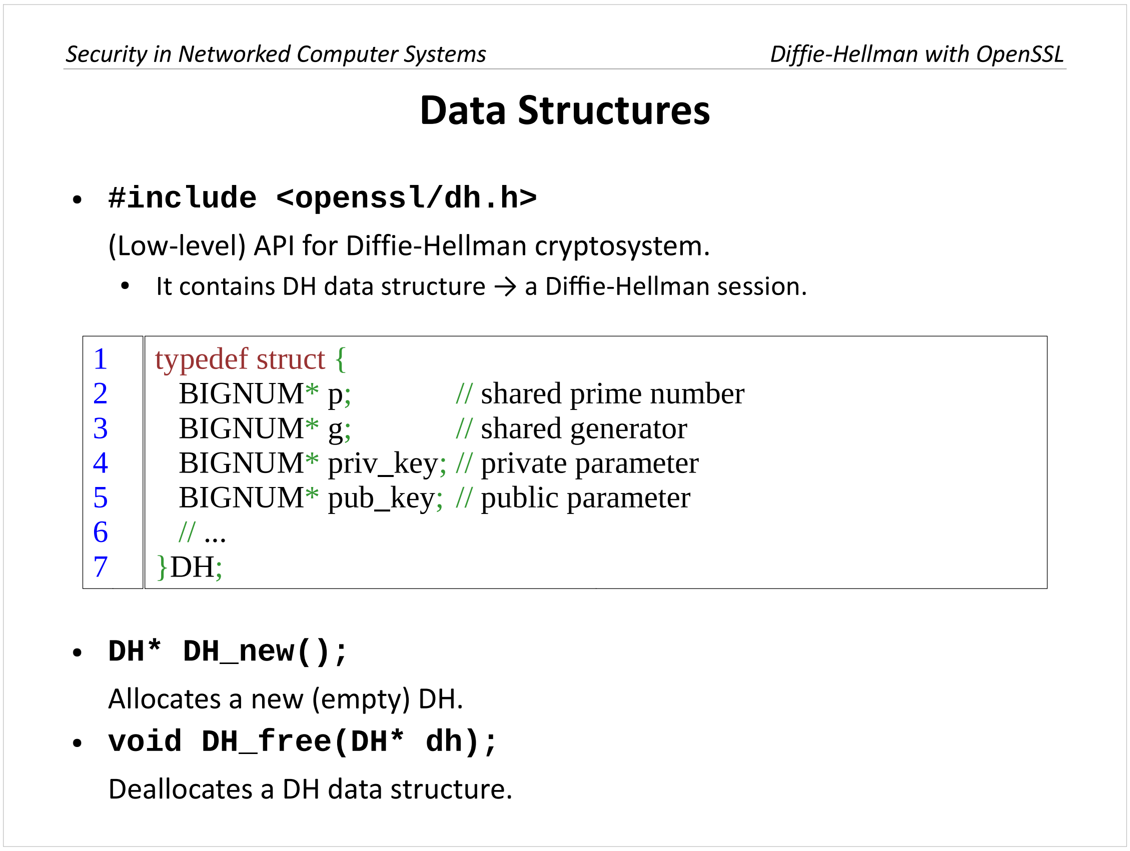
Multi-Precision Integers

- **`int BN_bn2bin(const BIGNUM* bn, unsigned char* to);`**

  From BIGNUM to binary (big-endian form). Returns the number of bytes written.
- **`BIGNUM* BN_bin2bn(const unsigned char* s, int len, BIGNUM* ret);`**

  From binary (big-endian form) to BIGNUM. The result is saved in ret (if != NULL) and returned.

These API functions convert from BIGNUM to byte buffer (big-endian form) and vice versa. They are useful for sending BIGNUM's over sockets.



Security in Networked Computer Systems *Diffie-Hellman with OpenSSL*

Data Structures

- **`#include <openssl/dh.h>`**

  (Low-level) API for Diffie-Hellman cryptosystem.

  - It contains DH data structure → a Diffie-Hellman session.

```
typedef struct {
   BIGNUM* p;           // shared prime number
   BIGNUM* g;           // shared generator
   BIGNUM* priv_key; // private parameter
   BIGNUM* pub_key;  // public parameter
   // ...
}DH;
```

- **`DH* DH_new();`**

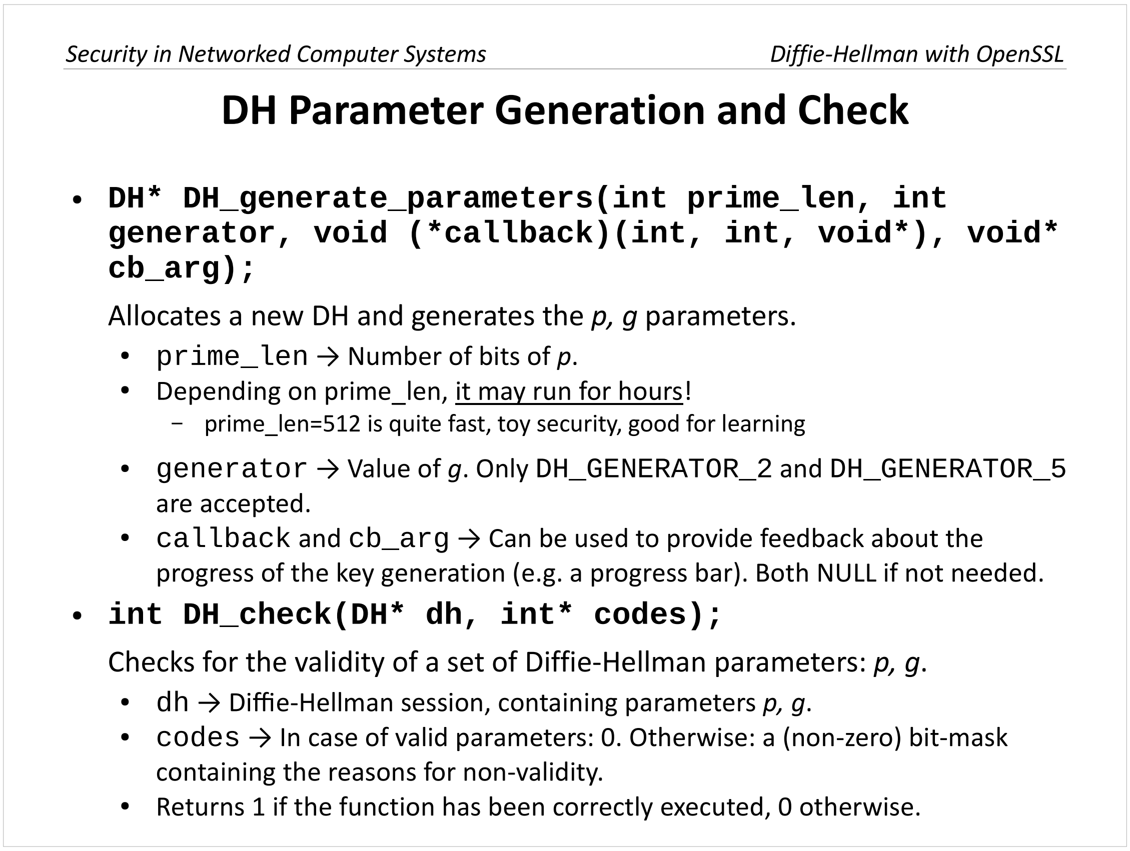
  Allocates a new (empty) DH.

- **`void DH_free(DH* dh);`**

  Deallocates a DH data structure.

A DH data structure (`#include<openssl/dh.h>`) represents a Diffie-Hellman protocol session. It contains all the quantities needed to run a protocol, under the form of BIGNUM's. priv_key and pub_key fields represent a party's private and public quantity, respectively.

These API functions allocate and deallocate a DH data structure.



Security in Networked Computer Systems *Diffie-Hellman with OpenSSL*

DH Parameter Generation and Check

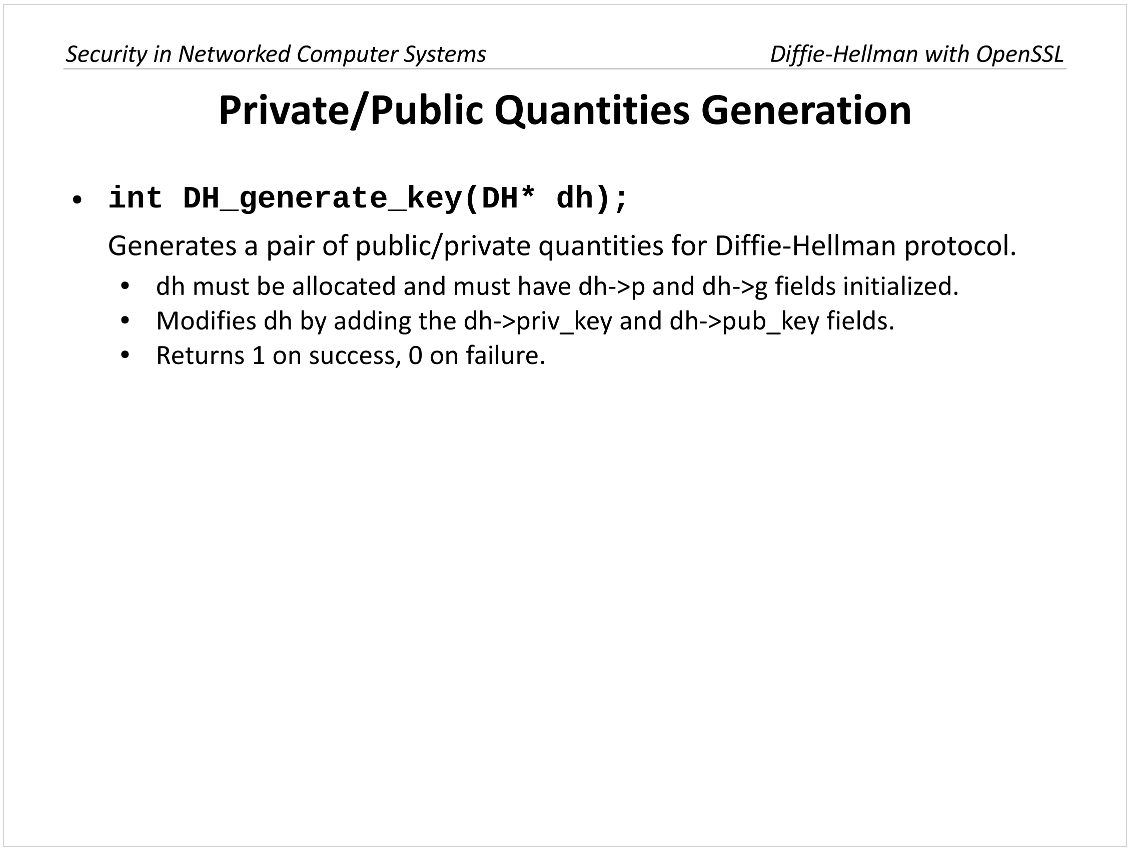
- **`DH* DH_generate_parameters(int prime_len, int generator, void (*callback)(int, int, void*), void* cb_arg);`**

  Allocates a new DH and generates the *p, g* parameters.
  - `prime_len` → Number of bits of *p*.
  - Depending on prime_len, it may run for hours!
    - prime_len=512 is quite fast, toy security, good for learning
  - `generator` → Value of *g*. Only `DH_GENERATOR_2` and `DH_GENERATOR_5` are accepted.
  - `callback` and `cb_arg` → Can be used to provide feedback about the progress of the key generation (e.g. a progress bar). Both NULL if not needed.
- **`int DH_check(DH* dh, int* codes);`**

  Checks for the validity of a set of Diffie-Hellman parameters: *p, g*.
  - `dh` → Diffie-Hellman session, containing parameters *p, g*.
  - `codes` → In case of valid parameters: 0. Otherwise: a (non-zero) bit-mask containing the reasons for non-validity.
  - Returns 1 if the function has been correctly executed, 0 otherwise.

The function `DH_generate_parameters()` allocates a DH data structure and generates randomly the parameter *p* and *g* (the PRNG must be properly seeded). Depending on the number of bits of *p* (prime_len) it may run for a long time. From simple tests, parameter generation requires ~2.7sec for 512 bits, ~1min for 1024 bits, ~15min for 2048 bits (on a 1.66GHz Intel processor). The resistance of a Diffie-Hellman protocol (against eavesdropper) is roughly the same of an RSA cryptosystem with the same number of bits. Thus: 512 bits are toy security (good for learning); 1024 bits are quite poor security (almost obsolete nowadays), 3072 bits are good security (fine for 99% real-life applications), 7680 bits are TOP SECRET security (equivalent to 384-bit Elliptic-Curve Cryptography, which is approved by NSA).

The function `DH_check()` checks for the validity of a set of Diffie-Hellman parameters *p, g*. Note that the return value IS NOT the outcome of the check, which is returned by the `codes` parameter. If the parameters are not valid, the function will return 1 and `codes` will be non-zero.



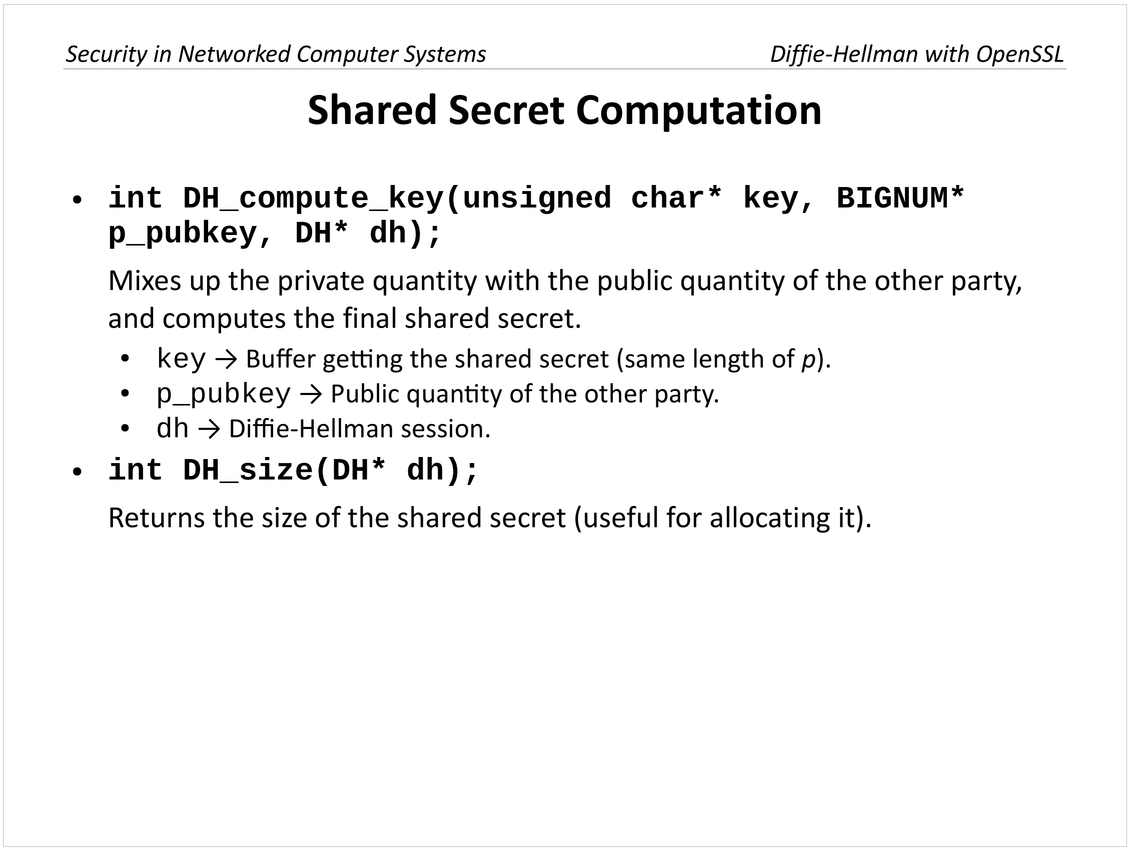
Private/Public Quantities Generation

- **`int DH_generate_key(DH* dh);`**

  Generates a pair of public/private quantities for Diffie-Hellman protocol.

  - dh must be allocated and must have dh->p and dh->g fields initialized.
  - Modifies dh by adding the dh->priv_key and dh->pub_key fields.
  - Returns 1 on success, 0 on failure.

The function `DH_generate_key()` generates a pair of private/public quantities to be used in a Diffie-Hellman protocol. The private quantity must be kept secret, whereas the public one (dh->pub_key) must be sent to the other party.



Shared Secret Computation

- **`int DH_compute_key(unsigned char* key, BIGNUM* p_pubkey, DH* dh);`**

  Mixes up the private quantity with the public quantity of the other party, and computes the final shared secret.

  - `key` → Buffer getting the shared secret (same length of *p*).
  - `p_pubkey` → Public quantity of the other party.
  - `dh` → Diffie-Hellman session.

- **`int DH_size(DH* dh);`**

  Returns the size of the shared secret (useful for allocating it).

The function `DH_compute_key()` computes the final shared secret, given a Diffie-Hellman session and the public quantity received from the other party. The buffer key must be sized properly to accommodate the shared secret. The function `DH_size()` returns the size of the shared secret. Note that in Diffie-Hellman protocol, the final shared secret has the same size of the parameter *p*.

Final Exercise

- Client-server application which establishes a Diffie-Hellman confidential channel and exchanges a file.
- The server:
  - Generates DH parameters (512 bits) and sends them to the client.
  - Generates server's private and public quantities.
  - Exchanges the public quantity with the client.
- The client:
  - Receives DH parameters and checks their validity.
  - Generates client's private and public quantities.
  - Exchanges the public quantity with the server.
- Both server and client:
  - Compute the shared secret.
- The client uses the shared secret to encrypt a file (DES in CBC mode) and sends the ciphertext to the server, the server receives the ciphertext, decrypts it, and saves it on a local file.

Final Exercise - Extension

- Generate once the Diffie-Hellman parameters and store them in a file. Load them when you need to perform Diffie-Hellman.
- To generate Diffie-Hellman parameters, use the command-line tool:

  ```
  openssl dhparam -out dh.pem 1024
  ```

  It generates 1024-bit parameters (takes about 1 minute) and stores them in “dh.pem” (in PEM format, which is human-readable ASCII).
- In the C code, load the “dh.pem” file with:

  ```
  DH* PEM_read_DHparams(FILE* fp, NULL, NULL, NULL);
  ```

  It allocates a DH data structure and loads the parameters *p, g* from the file `fp` (opened with `fopen()`). It returns the DH data structure (NULL if error).