

Lecture 3

Programming with OpenSSL

Patrick P. C. Lee

Roadmap

➤ OpenSSL

➤ Why Cryptosystems Fail?

SSL and OpenSSL

- SSL is the security protocol behind secure HTTP (HTTPS). It can secure any protocol that works over TCP
 - Will cover the theoretical details in “Web Security” module
- OpenSSL is a full-featured implementation of SSL, including TLS (transport layer security)

Overview of OpenSSL

- **OpenSSL** is an open-source implementation of cryptographic functions. It includes **executable commands with cryptographic functions**, and a **library of APIs** with which programmers can use to develop cryptographic applications.
- Its design follows the object-oriented principle. Each cryptographic algorithm is built upon a **context**, an object that holds the necessary parameters.

Installation of OpenSSL

➤ How to install:

- Get the latest stable source from <http://www.openssl.org>, and run (as root):

```
# ./config  
# make  
# make test  
# make install
```

➤ How to compile (as normal users):

```
% gcc -Wall test_openssl.c -o test_openssl -lcrypto
```

BIGNUM – Arbitrary Precision Math

- Public key cryptography handles very large integers. Standard C data types are not enough
- **BIGNUM** package has virtually no limits on upper bounds of numbers
- Header file: `#include <openssl/bn.h>`
- Reference:
 - <http://www.openssl.org/docs/crypto/bn.html>

Initializing & Destroying BIGNUMs

- A BIGNUM is an object (or context) that contains dynamically allocated memory

```
BIGNUM static_bn, *dynamic_bn;

/* initialize a static BIGNUM */
BN_init(&static_bn);

/* allocate a dynamic BIGNUM */
dynamic_bn = BN_new();

/* free the BIGNUMs */
BN_free(dynamic_bn);
BN_free(&static_bn);
```

Copying BIGNUMs

- **Deep copy** is required when you copy a BIGNUM object

```
BIGNUM a, b, *c;

/* wrong way */
a = b;
*c = b;

/* right away */
BN_copy(&a, &b);    /* copies b to a */
c = BN_dup(&b);     /* creates c and initialize it to same value as b */
```


BIGNUM to Binary

- Sometimes you need to convert a BIGNUM into binary representation for:
 - Store it in a file
 - Send it via a socket connection
- Similarly, we can convert a BIGNUM into a decimal or hexadecimal representation

BIGNUM to Binary

➤ How to convert?

```
BIGNUM* num;

/* converting from BIGNUM to binary */
len = BN_num_bytes(num)
buf = (unsigned char*)calloc(len, sizeof(unsigned char));
len = BN_bn2bin(num, buf);

/* converting from binary to BIGNUM */
BN_bin2bn(buf, len, num);
num = BN_bin2bn(buf, len, NULL);
```

BIGNUM to Binary

- **Pitfall** – if you don't pay attention to the actual length of the binary string.
- Suppose I work on 1024-bit RSA

```
const int RSA_SIZE = 128;    /* in bytes */
BIGNUM* num;

/* .. set num to 3 as public key */

/* converting from BIGNUM to binary */
buf = (unsigned char*)calloc(RSA_SIZE, sizeof(unsigned char));
BN_bn2bin(num, buf);

/* converting from binary to BIGNUM (use 128 bytes directly) */
BN_bin2bn(buf, RSA_SIZE, num);
num = BN_bin2bn(buf, RSA_SIZE, NULL);

/* WRONG RESULT: num <> 3 */
```

BIGNUM to Binary

- The binary representation of BIGNUM is in **big-endian format**.
- After BN_bn2bin(),

Address	0	1	2	...	127
	3	0	0		0

- After BN_bin2bn(), $\text{num} = 3 * 2^{127 * 8}$

BIGNUM to Binary

- You can store the actual length separately.
Or, to save the overhead of storing the length, move the significant string to the right

```
const int RSA_SIZE = 128;
BIGNUM* num;

/* .. set num to 3 as public key */

/* converting from BIGNUM to binary */
buf = (unsigned char*)calloc(RSA_SIZE, sizeof(unsigned char));
BN_bn2bin(num, buf + RSA_SIZE - BN_num_bytes(num));
```

Math Operations of BIGNUM

- Many operations are included in BIGNUM package, including modular arithmetic.
- Example: BN_mod_exp()

```
BIGNUM *r, *g, *x, *p;  
BN_CTX* ctx = BN_CTX_new();    /* store temporary results */  
  
/* .. call BN_new() on r, g, x, p */  
  
/* r = g^x mod p */  
BN_mod_exp(r, g, x, p, ctx);  
  
/* when done, free r, g, x, p, and ctx */
```

- BN_CTX stores temporary values of operations so as to improve the performance.

Generating Prime BIGNUM

- `BN_generate_prime()` generates pseudorandom prime numbers
- Instead of factoring, check if a number is prime after a number of primality tests
 - The generation is quite efficient

```
BIGNUM* prime = BN_generate_prime(NULL, bits, safe, NULL, NULL,  
                                   callback, NULL);
```

- Safe primes – p is prime and $(p-1)/2$ is also prime.

Symmetric Key Crypto in OpenSSL

- We choose AES with CBC
- We need to pad the last block if the plaintext length is not a multiple of block size (i.e., 128 bits / 16 bytes)
 - In general, you can add a few more junk blocks to decouple the plaintext length and ciphertext length, with a tradeoff of longer ciphertext

AES with CBC in OpenSSL

- Find the actual length of the encrypted message.
Make it a multiple of 128 bits

```
#include <openssl/aes.h>
...
unsigned int message_len = strlen((char*)input_string) + 1; // including '\0'
unsigned encrypt_len = (message_len % AES_BLOCK_SIZE == 0) ?
    message_len : (message_len / AES_BLOCK_SIZE + 1) * AES_BLOCK_SIZE;
```

- Define the key (assuming 128 bits)

```
unsigned char key[16];
AES aes;
int ret = AES_set_encrypt_key(key, 128, &aes); // ret < 0 → error
```

- Define the IV:

```
unsigned char iv[AES_BLOCK_SIZE];
memset(iv, 0, AES_BLOCK_SIZE);
```

AES with CBC in OpenSSL

- Encrypt the plaintext (note that **iv** will be updated)

```
AES_cbc_encrypt(input_string, encrypt_string, encrypt_len, &aes, iv,  
                AES_ENCRYPT);
```

- Decrypt the ciphertext. The decryption side must synchronize on the **iv** and **key**.
 - **iv** can be sent in plain, but **key** must be sent securely

```
AES_set_decrypt_key(key, 128, &aes);  
memset(iv, 0, AES_BLOCK_SIZE);  
...  
AES_cbc_encrypt(encrypt_string, decrypt_string, encrypt_len, &aes, iv,  
                AES_DECRYPT);
```

Public Key Crypto in OpenSSL

➤ Focus on:

- RSA for public key encryption
- Diffie-Hellman for key management
- DSA for digital signatures

➤ Public key crypto operates on BIGNUMs

```
typedef struct {  
    BIGNUM *n; // public modulus  
    BIGNUM *e; // public exponent  
    BIGNUM *d; // private exponent  
    BIGNUM *p; // secret prime factor  
    BIGNUM *q; // secret prime factor  
    // ...  
} RSA;
```

RSA in OpenSSL

- Bob first generates the RSA keys, e.g., 1024 bits and exponent 3.

```
#include <openssl/rsa.h>
RSA* rsa = RSA_generate_key(1024, 3, NULL, NULL);
```

- Bob passes public keys to Alice

```
#include <openssl/bn.h>
unsigned char* n_b = (unsigned char*)calloc(RSA_size(rsa), sizeof(unsigned char));
unsigned char* e_b = (unsigned char*)calloc(RSA_size(rsa), sizeof(unsigned char));
int n_size = BN_bn2bin(rsa->n, n_b);
int b_size = BN_bn2bin(rsa->e, e_b);
```

- Alice constructs the RSA context from public params

```
RSA* encrypt_rsa = RSA_new();
encrypt_rsa->n = BN_bin2bn(n_b, n_size, NULL);
encrypt_rsa->e = BN_bin2bn(e_b, b_size, NULL);
```

RSA in OpenSSL

- Alice can now encrypt data.

```
unsigned char* encrypt_string = (unsigned char*) calloc(RSA_size(encrypt_rsa),  
    sizeof(unsigned char));  
int encrypt_size = RSA_public_encrypt(strlen((char*)input_string),  
    input_string, encrypt_string, encrypt_rsa, RSA_PKCS1_OAEP_PADDING);
```

- RSA_PKCS1_OAEP_PADDING is recommended. The size of the input message block must be smaller than $\text{RSA}(\text{size}) - 41$.
- Bob can then decrypt

```
unsigned char* decrypt_string = (unsigned char*)calloc(RSA_size(rsa),  
    sizeof(unsigned char));  
int decrypt_size = RSA_private_decrypt(encrypt_size, encrypt_string,  
    decrypt_string, rsa, RSA_PKCS1_OAEP_PADDING);
```

RSA in OpenSSL

- Padding encodes packets before encryption
- Padding schemes in RSA
 - `RSA_PKCS1_PADDING`:
 - plaintext length smaller than `RSA_size(rsa) - 11`
 - `RSA_PKCS1_OAEP_PADDING`:
 - plaintext length smaller than `RSA_size(rsa) - 41`
 - `RSA_SSLV23_PADDING`:
 - rarely used
 - `RSA_NO_PADDING`:
 - Assumes the caller performs padding. Plaintext length must equal to `RSA_size(rsa)`, not recommended.

RSA in OpenSSL

- Instead of generating parameters in a program, you can do it in command line and save parameters into a **PEM** file for permanent use
 - PEM file is a base64-encoded file format that represents keys in a file
 - E.g., generate 1024-bit parameters

```
% openssl genrsa -out rsa1024.pem 1024
```

- The private key contains public key info as well.
- In program, call PEM_read_RSAPrivateKey()

```
RSA* rsa = PEM_read_RSAPrivateKey(fp, NULL, NULL, NULL);
```

Diffie-Hellman in OpenSSL

- Header file: `#include <openssl/dh.h>`
- A DH struct:

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


Diffie-Hellman in OpenSSL

- Generate DH parameters in command line (e.g., prime p and generator g)

```
% openssl dhparam -out dh1024.pem 1024
```

- Read parameters from the file

```
#include <openssl/dh.h>
#include <openssl/pem.h>

FILE* fp = fopen("dh1024.pem", "r");
DH* dh1 = PEM_read_DHparams(fp, NULL, NULL, NULL);
```

Diffie-Hellman in OpenSSL

- No keys are in the file. You need to generate keys separately

```
DH_generate_key(dh1);
```

- Compute the secret key, assuming public key was sent over network and reconstructed into BIGNUM object

```
unsigned char* key1 = (unsigned char*)calloc(DH_size(dh1),  
        sizeof(unsigned char));  
unsigned char* key2 = (unsigned char*)calloc(DH_size(dh2),  
        sizeof(unsigned char));  
DH_compute_key(key1, dh2_pub_key, dh1);  
DH_compute_key(key2, dh1_pub_key, dh2);
```

DSA in OpenSSL

- Sender generates DSA parameters (e.g., 1024 bits)

```
#include <openssl/dsa.h>
...
DSA* dsa = DSA_generate_parameters(1024, NULL, 0, NULL,
    NULL, NULL);
```

- And generate the keys

```
DSA_generate_key(dsa);
```

DSA in OpenSSL

➤ To sign a message:

```
unsigned char* sign_string = (unsigned char*)calloc(DSA_size(dsa),  
    sizeof(unsigned char));  
int ret = DSA_sign(0, input_string, strlen((char*)input_string),  
    sign_string, &sig_len, dsa);
```

➤ To verify a message

```
int is_valid = DSA_verify(0, input_string,  
    strlen((char*)input_string), sign_string, sig_len, dsa);
```

- is_valid = 1 means verified, 0 means wrong signature

Hash Functions in OpenSSL

➤ Based on three functions:

- *Init(): initialize the hash structure
- *Update(): keep adding messages to be hashed (can be called multiple times)
- *Final(): compute the hash value

➤ Example: md5

```
#include <openssl/md5.h>

MD5_CTX hash_ctx;
MD5_Init(&hash_ctx);    // initialize

char input_string[100];
strcpy(input_string, "abcdedfg");
MD5_Update(&hash_ctx, input_string, strlen(input_string)); // update

unsigned char hash_ret[16];
MD5_Final(hash_ret, &hash_ctx);    // compute the hash vlaue
```

Certificates in OpenSSL

- Goal: verify a signature using a public key certificate issued by a certificate authority
- Main idea: generate certificates in command line, and call the certs from programs through APIs

Certificates in OpenSSL

- Step 1: we need a CA. First, create a **self-signed certificate**

```
% openssl req -x509 -newkey rsa -out cacert.pem -outform PEM -days 365 \  
-key cakey.pem
```

- Let's use "5470" as the passphrase.
- Specify a one-year expiration period using "-days"
- The Certificate is encoded in PEM format. You can see the content with the command:

```
% openssl x509 -in cacert.pem -text -noout
```

- Default configuration is used for certificate generation. Can provide our own configuration.
- Two files created:
 - cakey.pem – private key of CA
 - cacert.pem – certificate of CA

Certificates in OpenSSL

- Step 2: When a user wants to apply for a certificate, the CA will generate a new public/private key pair and the corresponding **certificate request**.

```
% openssl genpkey -algorithm RSA -out key.pem -aes-128-cbc \  
-pkeyopt rsa_keygen_bits:1024  
% openssl req -new -key key.pem -keyform PEM -out req.pem \  
-outform PEM
```

The above command will encrypt the private key file (key.pem) with a passphrase when the certificate is issued. It will prompt for a passphrase.

Let's use **"5470"** for the passphrase.

Certificates in OpenSSL

- Step 3: The CA will then generate a certificate based on the request.

```
% openssl ca -in req.pem -out cert.pem -config ca.conf
```

See ca.conf for the format. Note that before calling the command, we must have prepared two files: index.txt (which could be empty) and serial (which stores a number, e.g., 01). The private key file (key.pem) and the certificate (cert.pem) will be given to the user.

How should I sign/verify?

➤ People who sign:

- Use the private key to sign
- Issue the public key certificate

➤ People who verify:

- Use the public key in the certificate to verify

➤ Here, we demonstrate the use of **EVP** API:

- EVP API provides a common interface to every cipher OpenSSL exports
- When you change to a new cipher algorithm, you only register the new algorithm during initialization

Signing with Cert in OpenSSL

➤ Step 1: load the private key file.

Since the file is encrypted, we need to specify which encryption algorithm is used. To save troubles, we just call OpenSSL `add_all_algorithms()` to include all possible cipher and digest algorithms.

```
#include <openssl/evp.h>
#include <openssl/pem.h>
...
OpenSSL_add_all_algorithms();
...
// load the private key
FILE* fp = fopen("key.pem", "r") == NULL);
EVP_PKEY* priv_key = PEM_read_PrivateKey(fp, NULL, NULL, (char*)"5470");
fclose(fp);
if (priv_key == NULL) {
    fprintf(stderr, "cannot read private key.\n");
    exit(-1);
}
```

Signing with Cert in OpenSSL

- Step 2: sign the message digest
You don't need to sign the whole message.
Just sign the (shorter) message digest.

```
int sig_len = 128; // 1024-bit key
unsigned char sign_string[128];
EVP_MD_CTX evp_md_ctx;

EVP_SignInit(&evp_md_ctx, EVP_sha1());
EVP_SignUpdate(&evp_md_ctx, input_string, strlen((char*)input_string));
if (EVP_SignFinal(&evp_md_ctx, sign_string, &sig_len, priv_key) == 0) {
    fprintf(stderr, "Unable to sign.\n");
    exit(-1);
}
```

Verifying with Cert in OpenSSL

- Step 1: Before we can verify a message, we need to first obtain the certificate of the signer. The certificate also contains the public key.

```
X509* cert;
EVP_PKEY* pub_key;

FILE* fp = fopen("cert.pem", "r");
if ((cert = PEM_read_X509(fp, NULL, NULL, NULL)) == NULL) {
    fprintf(stderr, "cannot read cert file\n");
    exit(-1);
}
fclose(fp);
if ((pub_key = X509_get_pubkey(cert)) == NULL) {
    fprintf(stderr, "cannot read x509's public key\n");
    exit(-1);
}
```

Verifying with Cert in OpenSSL

➤ Step 2: verify the message digest

```
EVP_VerifyInit(&evp_md_ctx, EVP_sha1());  
EVP_VerifyUpdate(&evp_md_ctx, input_string, strlen((char*)input_string));  
if (EVP_VerifyFinal(&evp_md_ctx, sign_string, sig_len, pub_key)) {  
    printf("Verified\n");  
} else {  
    printf("Wrong\n");  
}
```

Roadmap

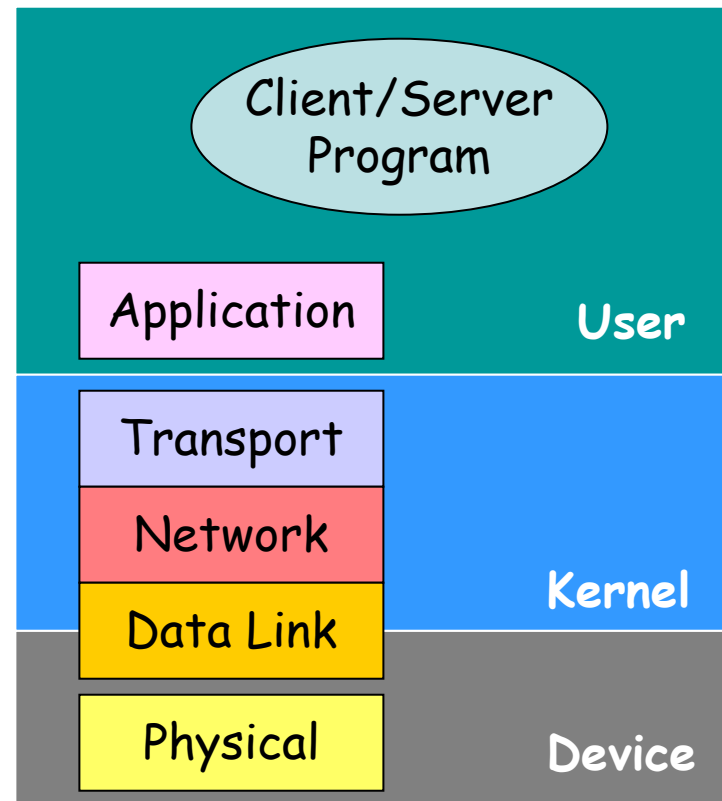
➤ OpenSSL

- SSL/TLS Programming

➤ Why Cryptosystems Fail?

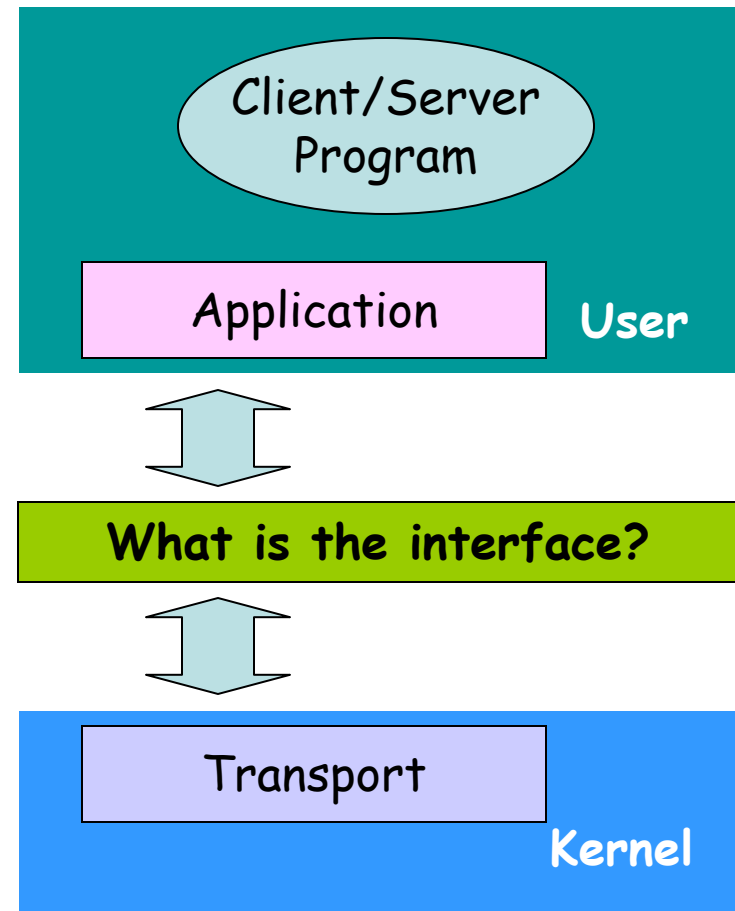
How to implement a network application?

- Implementation of each protocol layer is located in either the user-space, kernel-space, or the device (hardware)



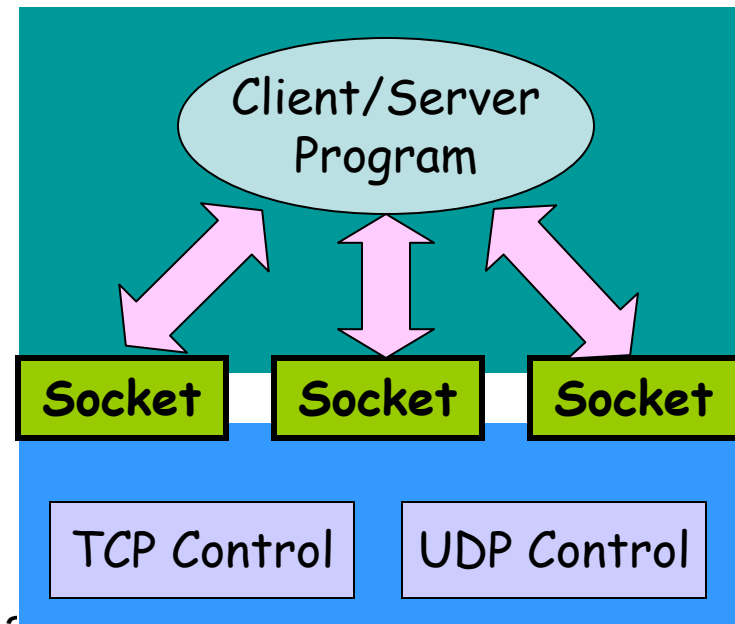
How to implement a network application?

- We need a “door” so that a network application can send/receive messages to/from the network
- The door should appear between the user space and kernel space so that details of the kernel space can be hidden
- Socket is the door!



How to implement a network application?

- A **socket** is a *host-local, application-created, OS-controlled* interface (a “door”) into which application process can **both send and receive** messages to/from another application process
 - Similar to a **file descriptor**, which interfaces between an application and a file
- One socket is tied to one application process (or thread)
 - An application can create many processes (and hence sockets)



Socket programming

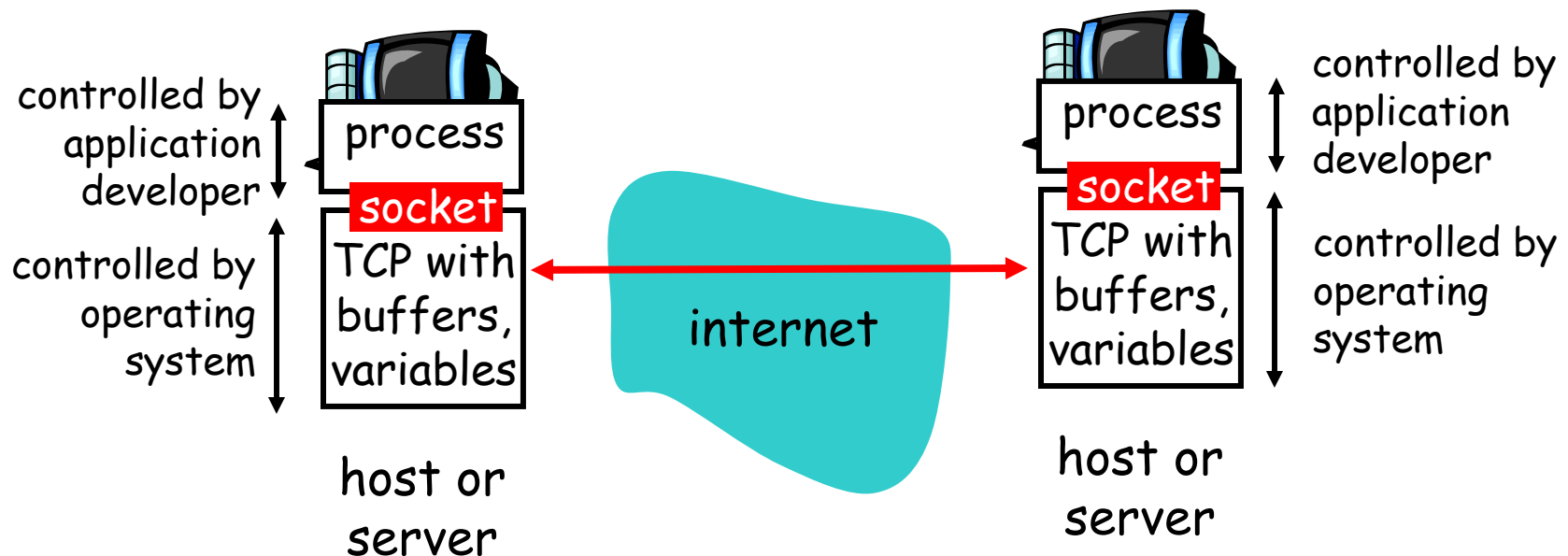
Socket programming is to build client/server application that communicate using sockets

Socket API

- introduced in BSD4.1 UNIX, 1981
- explicitly created, used, released by apps
- client/server paradigm
- two types of transport service via socket API:
 - unreliable datagram
 - reliable, byte stream-oriented

Socket-programming using TCP

- TCP service: reliable transfer of bytes from one process to another.
- An application may view TCP a reliable, in-order **pipe** (or **stream**).



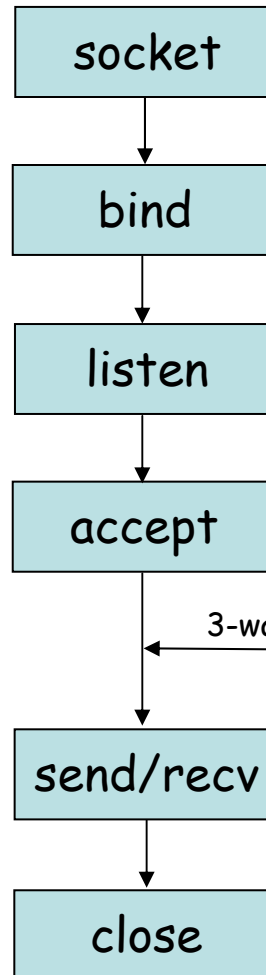
Socket programming using TCP

- Before client contacts server:
 - server process must first be running
 - server must have created socket (door) that welcomes client's contact
- Client contacts server by:
 - creating client-local TCP socket
 - specifying **IP address, port number** of server process
 - When **client creates socket**: client TCP establishes connection to server TCP
- When contacted by client, server creates new TCP socket for server process to communicate with client
 - allows server to talk with multiple clients
 - **source port numbers used to distinguish clients**

Socket programming with TCP

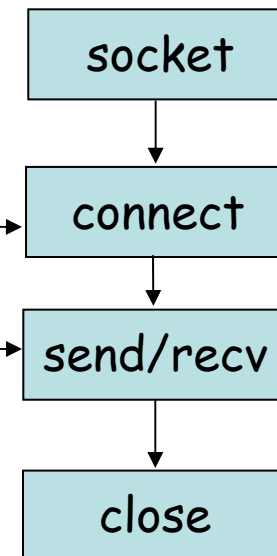
- When a TCP server creates a socket, it needs to specify:
 - Identifier of the socket
 - Connection mode (TCP/UDP)
- Analogous to when you open a file in C, you need to specify:
 - location of the file
 - access mode (e.g., read-only, write-only)

TCP server



Operations of socket programming in C/C++

TCP client



3-way handshake

share data

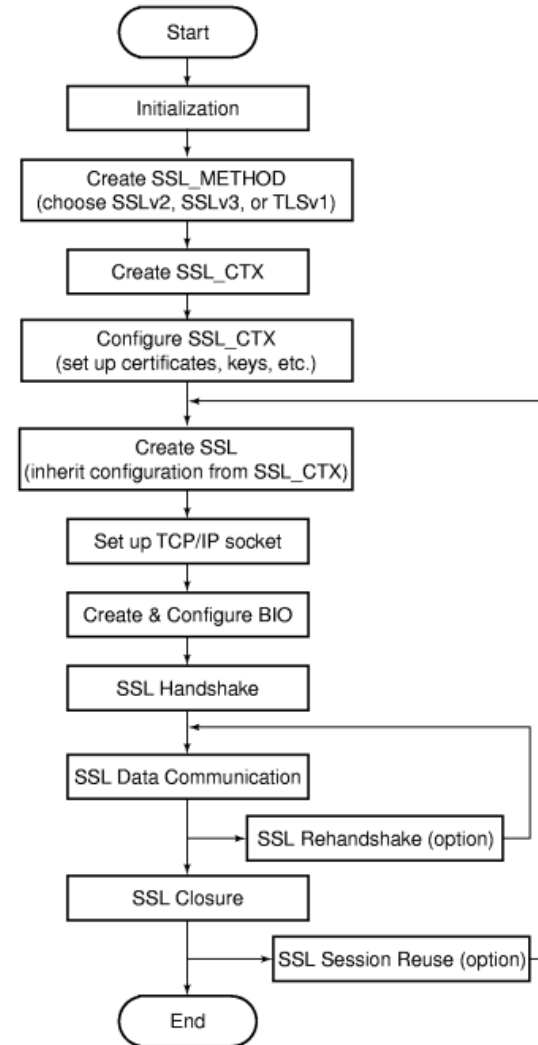
SSL/TLS Programming

- Our goal is to enhance socket programming with Secure Sockets Layer (SSL) and Transport Layer Security (TLS)
- We only provide templates here, with many subtleties ignored.

SSL/TLS Programming

➤ Flow of SSL programming

- Can be viewed as extensions of socket programming



Steps of SSL/TLS Programming

- Step 1: Initialize the SSL library to register all cipher and hash algorithms
- Step 2: Create the SSL context structure
 - e.g., specify the SSL versions
- Step 3: Set up certificates and keys
 - SSL server
 - server's own certificate (mandatory)
 - CA certificate (optional)
 - SSL client
 - CA's certificate (mandatory) for verifying server's cert
 - client's own certificate (optional)

Steps of SSL/TLS Programming

- Step 4: Set up certificate verification
 - can specify the chain length (verify_depth)
 - Client can set SSL_VERIFY_PEER to verify server's certificate is indeed issued by CA's cert
- Step 5: Create SSL structure and TCP/IP sockets, and bind them together

Steps of SSL/TLS Programming

- Step 6: SSL handshake
 - invoked when client calls `SSL_connect()`
 - if certificate verification is specified before, the actual verification will be carried out in this phase
- Step 7: Transmit SSL data
- Step 8: Shutdown SSL structure
- **See source code**

Summary on OpenSSL

- Low-level design: BIGNUM
 - Use BIGNUM to build your own cryptographic primitive
- Mid-level design: Cryptographic primitives
 - Use different combinations of cryptographic primitives to design a cryptosystem
- High-level design: SSL programming
 - Follow the SSL implementation

Roadmap

➤ OpenSSL

➤ Why Cryptosystems Fail?

Why Cryptosystems Fail?

[Anderson 1993]

- In practice, most frauds were not caused by cryptanalysis, but by implementation errors and management failures
- Public feedback about how cryptosystems fail is limited
- Let's use ATM as a case study
 - You provide your bank card and PIN to an ATM machine, and you access your account

Case Study – ATM

➤ How ATM fraud takes place?

- Insider knowledge
 - by bank clerks who issue bank cards
 - by technical staff who record customers' PINs
- Outsider attacks
 - observe the PINs entered by customers at ATM queues, and pick up discarded ATM tickets that have full account numbers
- PINs may not be fully random
 - e.g., digit 1 + digit 3 is equal to digit 2 + digit 4 so that offline ATMs and point-of-sale devices can recognize

Case Study - ATM

- How to keep keys secret is a major problem in cryptosystem design
- Standard approach: store keys in a tamper-resistant **security module** (e.g., trusted computing module)
- Yet, not all banks use security modules to protect keys (as of 1993)
 - too expensive, too difficult to install, etc.

Open Issues to Equipment Vendors

- Security products provide raw encryption capabilities, and leave application designers to worry about protocols and integrate into their systems
- Problems on application designers' side:
 - Lack of proper skills
 - Security functions neglected
 - Changing threats
 - Sloppy quality control

Implications for Equipment Vendors

- Three courses of actions:
- to design products that can be integrated and maintained by real experts
 - to train and certify client personnel
 - to supply their own personnel to implement, maintain, and manage the system

Wrong Threat Models

- Threats are misjudged.
 - Assuming criminals have the expertise
 - Assuming customer sites have the expertise in building secure systems
- Shift to new security paradigm?

Conclusions

- Cryptosystem designers have limited information of how their system fail, and hence accept wrong threat models
- As a result, security failures are mainly due to implementation and management errors

References

- John Viega et al., “Network Security with OpenSSL”, O’Reilly, 2002
- HP, “SSL Programming Tutorial”,
<http://h71000.www7.hp.com/doc/83final/ba55490007/ch04s03.html>
- Ross Anderson, “Why Cryptosystems Fail”, ACM CCS 1993