

A Client-Server Secure Communication Approach

EVANGELOS MOYROYTSOS, 2016

In cryptography, encryption is the process of encoding messages or information in such a way that only authorized parties can read it. Encryption does not of itself prevent interception, but denies the message content to the interceptor. In an encryption scheme, the intended communication information or message, referred to as plaintext, is encrypted using an encryption algorithm, generating ciphertext that can only be read if decrypted. An encryption scheme usually uses an encryption key generated by an algorithm, it is in principle possible to decrypt the message without possessing the key, but, for a well-designed encryption scheme, large computational resources and skill are required. An authorized recipient can easily decrypt the message with the key provided by the originator to recipients, but not to unauthorized interceptors.

Additional Key Words and Phrases: Challenge-Response, Public key encryption, Symmetric Encryption , Secure communication, Cipher Block Chaining, Hybrid Encryption, Client-Server, Threaded messaging

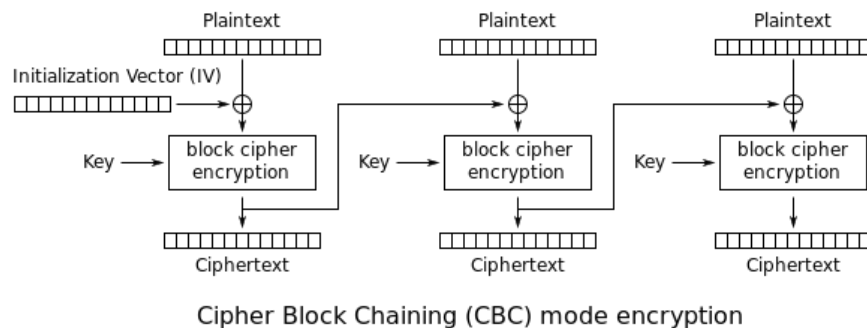
1. INTRODUCTION

Secure communications is a hot topic these days. In this white paper we implement a state of the art client-server hybrid model encryption system. Although some project details may not be optimal, an effort has been made to follow the latest security and cryptography standards and terminology. While this approach isn't ground breaking it applies some new concepts to cryptography such as RSA encryption with CBC block chaining and a hybrid challenge response protocol implementation. Our goal is to have a client server model for secure communication and deliver messages between parties.

2. HYBRID PUBLIC KEY ENCRYPTION

2.1 Cipher Block Chaining

Ehrtam, Meyer, Smith and Tuchman invented the Cipher Block Chaining (CBC) mode of operation in 1976. In CBC mode, each block of plaintext is XORed with the previous ciphertext block before being encrypted. This way, each ciphertext block depends on all plaintext blocks processed up to that point. To make each message unique, an initialization vector must be used in the first block.



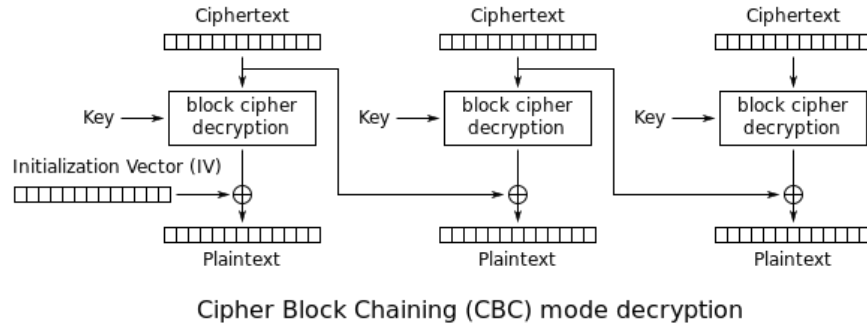


Fig. 0. Cipher Block Chaining

If the first block has index 1, the mathematical formula for CBC encryption is

$$C_i = E_k(P_i \text{ xor } C_{i-1}), C_0 = IV$$

while the mathematical formula for CBC decryption is

$$P_i = D_k(C_i) \text{ xor } C_{i-1}, C_0 = IV$$

Its main drawbacks are that encryption is sequential and that the message must be padded to a multiple of the cipher block size. Decrypting with the incorrect IV causes the first block of plaintext to be corrupt but subsequent plaintext blocks will be correct. This is because each block is XORed with the ciphertext of the previous block, not the plaintext, so one does not need to decrypt the previous block before using it as the IV for the decryption of the current one. This means that a plaintext block can be recovered from two adjacent blocks of ciphertext. Note that a one-bit change to the ciphertext causes complete corruption of the corresponding block of plaintext, and inverts the corresponding bit in the following block of plaintext, but the rest of the blocks remain intact.

2.2 RSA Cipher Block Chaining

We propose a hybrid encryption scheme to be used by each node, which is an extension of the well known RSA crypto system with extended support encryption for long input sequences and block ciphers. RSA encryption despite being a very good encryption scheme has some disadvantages one of them is the length of data that can be encrypted is analogous to the length of encryption key. By adding block ciphers processing in RSA we can encrypt arbitrary length of data without any limitation in the key size during encryption. In our approach we use RSA key sizes of 2048 bits and data block size of 128 bits. Effectively keys with 2048 bits produce ciphers of 256 bytes and the max length of input is $256 - 11 = 246$ bytes without the padding applied, so in reality we can use block sizes of up to 246 bytes. For more security against cryptanalysis the extension does use ECB but uses CBC which more secure. A good thumb of rule is to use block sizes of power of 2 and the biggest available block size available. ALGORITHM 1 shows the encryption process and ALGORITHM 2 shows the decryption process of RSA with CBC. Before any data encryption a padding scheme PKCS1 is applied in order to extend input sequence length to a multiple of block size,

again after decryption the inverse padding scheme is applied to the plain text obtained.

ALGORITHM 1. RSA CBC Encrypt

Input: Message m to encrypt, and the public key k 's of the recipient

Output: The Concatenation of cipher texts and IV.

$block_size = 128$;

$m = padd_PKCS1(m)$

$c = []$

for $i = 0$; $i < len(m)$; $i = i + block_size$

$c.append(m[i:i+block_size])$

end

$IV_first = IV = randint32bit()$

$ciphers = []$

for i in c

$dx = xor(i, IV)$

$cipher_t = rsa_enc(dx, key)$

$IV = cipher_t[0: block_size]$

$ciphers.append(cipher_t)$

end

$ciphers.append(IV_first)$

$return\ concat(ciphers)$

end;

ALGORITHM 2. RSA CBC Decrypt

Input: Encrypted message m and the private key k 's of the recipient

Output: unencrypted text plaintext.

$block_size = 128$;

$IV = message[-block_size:]$

$cipher_text = message[:- block_size]$

$c = []$

for $i = 0$; $i < len(cipher_text)$; $i = i + 2*block_size$

$c.append(cipher_text [i:i+2*block_size])$

end

$texts = []$

for d in c

$dx = rsa_dec(d, key)$

$plain_t = xor(dx, IV)$

$IV = d[0: block_size]$

$texts.append(plain_t)$

end

$m = concat(texts)$

$m = unpadd_PKCS1(m)$

```

return m
end;

```

3. CHALLENGE RESPOCE PROTOCOL

3.1 Protocol Encrypted Packets

The implementation of a secure authentication protocol using public key cryptography and signature verification for mutual authentication on both sides has 4 stages each one of them may not complete successfully and as a result the authentication attempt to be dropped from either of the two parties involved. Every data send over the network is double encrypted. First data is encrypted with the issuing party private key and then data encrypted by the destinations party public key so the encryption equation is a follows:

$$c = E_{Cu}(E_{Sr}(m), m)$$

Equation 1.

m denotes message for transmission
c denotes ciphertext produced
Cu, Cs denotes public private key client
Su, Sr denotes public private key server

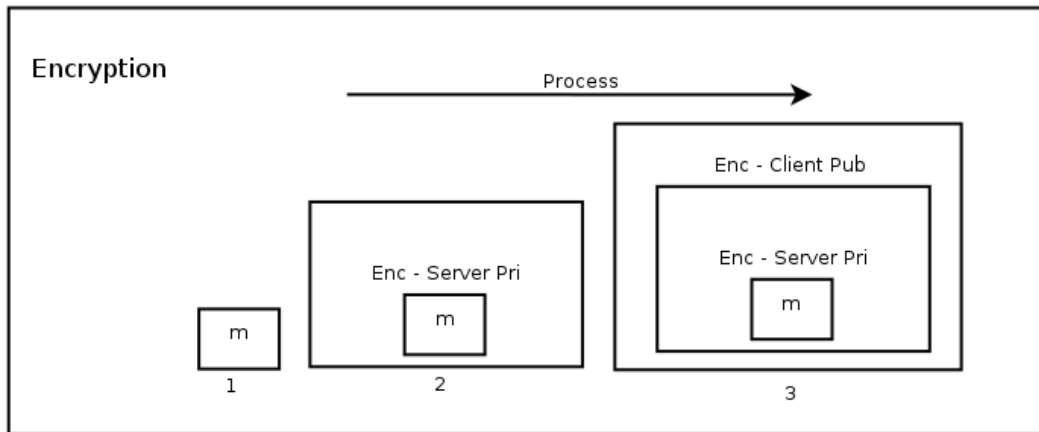


Fig. 1. Public key encryption with Equation 1, m denotes message

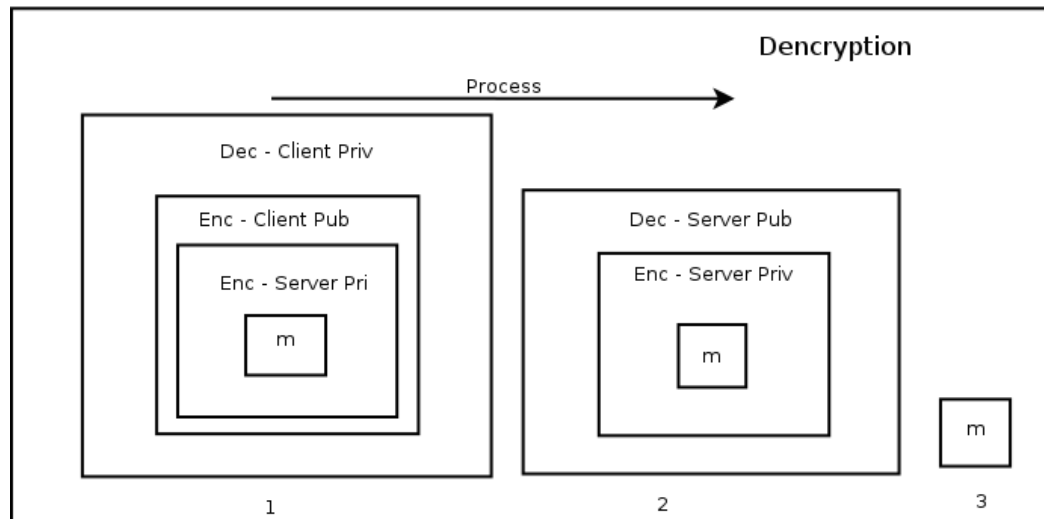


Fig. 2. Public key decryption with Equation 1, m denotes message

3.2 Protocol Implementation

The protocol messages use the encryption scheme from chapter 3.1 in order to securely exchange encrypted messages with both parties. On our example the client will start the challenge response protocol by sending the magic serial number to the server. The protocol is designed to deliver great security between both parties with mutual authentication and mechanisms to detect network attacks from untrusted parties. First the protocol structure will be explained which consists from 4 stages. The protocol assumes that public keys are known to both parties or delivered with a PKI.

Protocol Stages:

- (1) Client generates a 32 bit random number R_a (serial_a) and sent R_a along with the signature of R_a to the server.
- (2) Server verifies R_a with signature. Then server generates a 32 bit random number R_b (serial_b) and sends back to the client R_a and R_b and their signature.
- (3) Client notices if $R_a^* = R_a$ and signature match then the server identity is verified and proceeding by sending back R_b to the sever with it's signature. If signatures don't match client terminates the connection.
- (4) Server verifies client identity if $R_b^* = R_b$ and signature match if so the mutual authentication is completed successfully. If signatures don't match server terminates the connection.

The Challenge response now is completed successfully and both parties are trusted and verified. In the final step of the process the server generates a symmetric session key S_m and sends it to the client. The communication between client and sever from now on will be with symmetric cryptography using AES. Figure 3 shows the messages send between client and server in order authenticate and to agree on session key.

3.3 Challenge Response Packet Format

In this section we discuss packet format of the Challenge Response protocol discussed in the previews section.

Challenge Response Packet Format

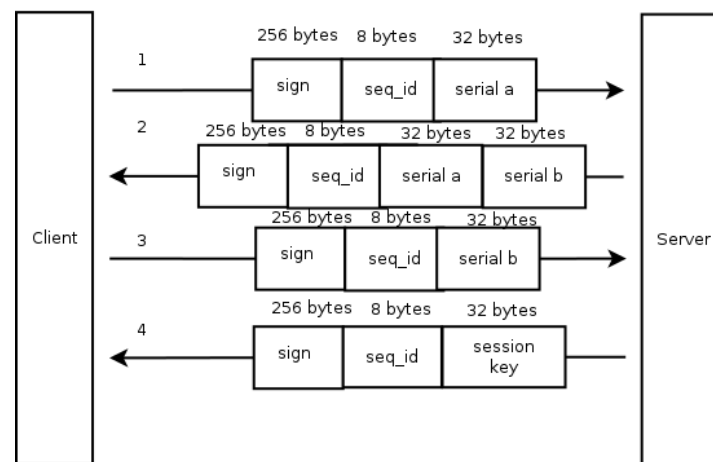


Fig. 3. Challenge response protocol packet formatting and message exchange

The protocol uses sequence numbers in order to verify messages exchanges come in order and avoid replay attack or man in the middle attacks. Also the protocol can delay attackers by applying time delay between the messages during stage 1 – 4. Every message reply is send with one second time delay so in total 4 seconds of time delay for every try. Also the protocol is designed to verify the integrity and the authentication of every message exchange through public key signing. Although we have to mention that it's more secure to have different keys for encryption and signing.

4. SECURE COMMUNICATION AND SESSION

The client opens a socket and try to connect to the server port. The server accepts the connection and starts a challenge response procedure in order to have mutual authentication both the client and the server for the opposite party of the connection. When authenticity is verified in both sides the server creates a session key and delivers it to the client. From now on all the messages until the termination of the connection will be encrypted with this session key and asymmetric encryption AES as implemented in AES module. When client has transmitted all the valuable data to the server and no longer wants to maintain a connection to the server, informs the server that the connection is going to be terminated with a special message content containing the keyword “secure-close” and data will be no longer transmitted. During the key establishment or message transmission if one of the parties can't verify the data or data delivery is out of order or sequence numbers don't match the connection is dropped and the user or server is notified with a message that it might be a possible man in the middle attack.

Typical Client/Server Session

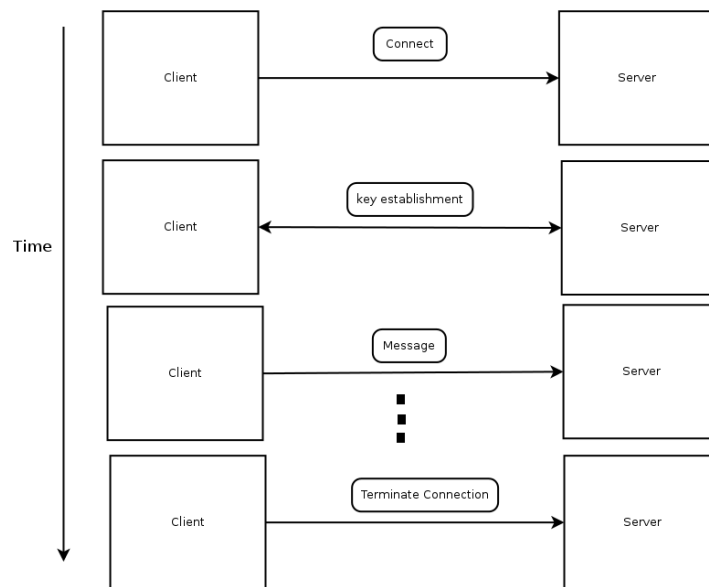


Fig. 4. High Level Block diagram of a client server session initiation until session termination.

5. CLIENT SERVER MODEL

The Server is implemented with threading support so that many users can connect concurrently and perform secure messaging. The server can hold logs for every message transmitted from the clients connected in the file “connection.log”. The server runs in a while loop in order to terminate the server process a special exception must be triggered with CTRL+C to signal the server to shutdown gracefully. The client when is connected to the server and authenticated prompts the user to enter a message to send to the server. If the user want to terminate the connection he must press key “Enter” enter as an input an empty line. Then the client sends a reserved keyword to the server to notify the server that the connection will be terminated as described in chapter 4. The packet format the client and server exchange is the following.

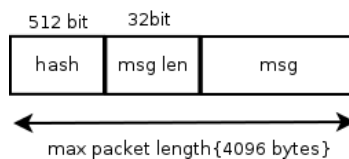


Fig. 5. Package transmitted from the client to the server

The first 512 bits of every packet is the SHA512 digest of the symmetric encryption key, followed by the plaintext message length represented in a 32 bit unsigned integer and finally follows the plaintext. Every packet then is encrypted with the session symmetric key AES cipher with PKCS5 padding scheme and send to the server. The maximum length for packet is defined as 4096 bytes in order to avoid fragmentation of the TCP/IP stack during data transmissions.

6. IMPLEMENTATION AND CODE STRUCTURE

For the implementation of the client and server the programming language Python 2.7.11 was used. In order to meet the requirements cryptographic libraries were used when it was applicable. The public key cryptography is based in the cryptographic library Cryptography and the symmetric cryptography is based on the PyCrypto library implementation. No other external dependencies were used.

The source code is kept to minimum number of modules and is structured as follows:

- client_threaded.py
- server_threaded.py
- aes_module.py
- rsa_module.py
- digest_module.py

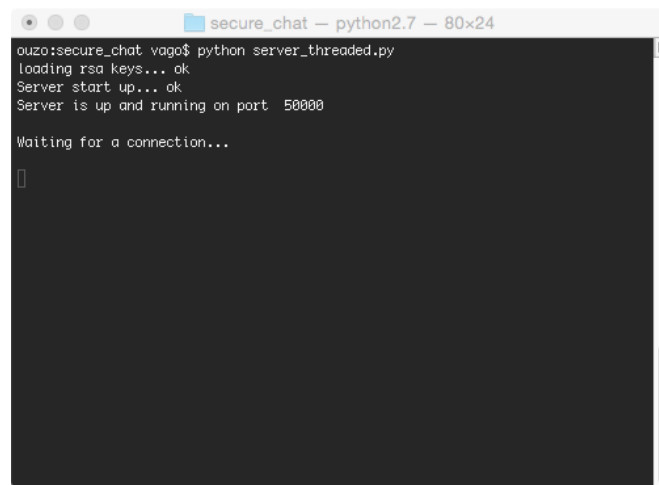
The Client and the server are implemented with a modular architecture based on modules which provide an API for cryptographic schemes and protocols from the modules: RSA module, AES module, Digest module.

Cryptogrquaphy Library: <https://cryptography.io/en/latest/>

PyCrypto Library: <https://www.dlitz.net/software/pycrypto/>

7. USAGE EXAMPLE

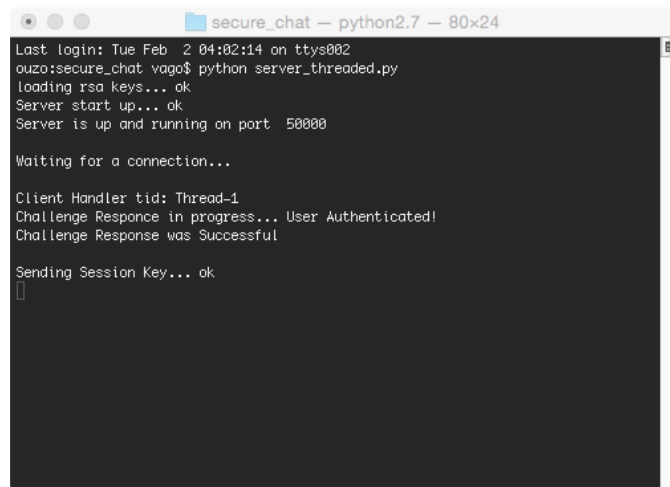
First start the server then. Start a client(s)

A screenshot of a terminal window titled "secure_chat - python2.7 - 80x24". The terminal shows the execution of the command "python server_threaded.py". The output of the script is as follows: "loading rsa keys... ok", "Server start up... ok", "Server is up and running on port 50000", and "Waiting for a connection...". A cursor is visible on the line "Waiting for a connection...".

```
secure_chat - python2.7 - 80x24
ouzo@secure_chat vago$ python server_threaded.py
loading rsa keys... ok
Server start up... ok
Server is up and running on port 50000
Waiting for a connection...
█
```

Fig. 5. Server start up message

A Secure Communication Client Server approach 39:9



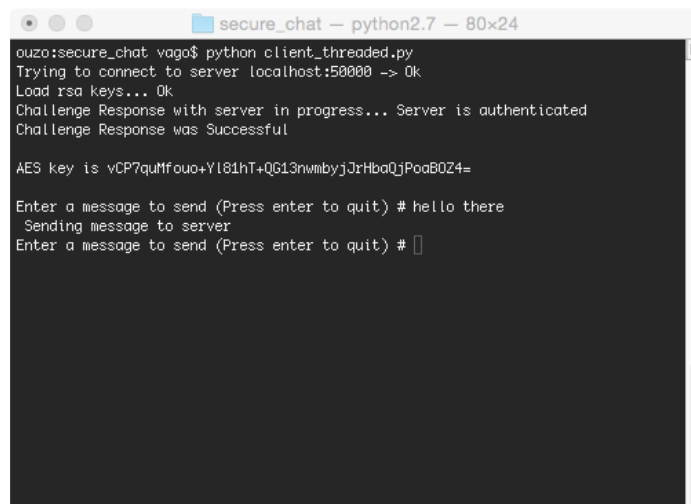
```
secure_chat — python2.7 — 80x24
Last login: Tue Feb  2 04:02:14 on ttys002
ouzo:secure_chat vago$ python server_threaded.py
loading rsa keys... ok
Server start up... ok
Server is up and running on port 50000

Waiting for a connection...

Client Handler tid: Thread-1
Challenge Response in progress... User Authenticated!
Challenge Response was Successful

Sending Session Key... ok
█
```

Fig. 6. New Connection to server

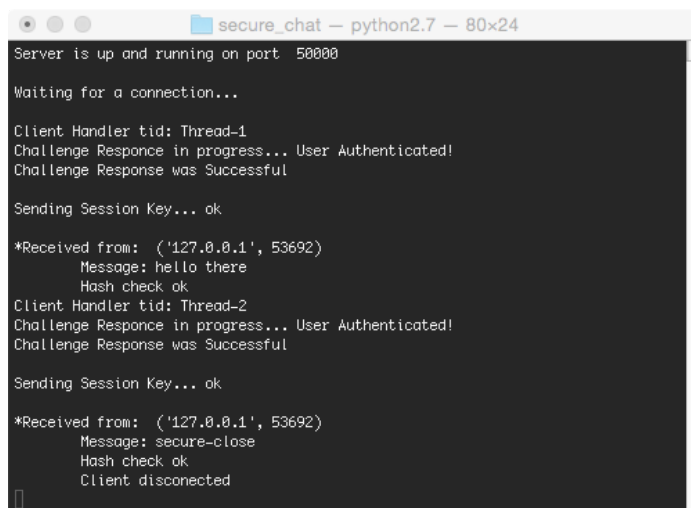


```
secure_chat — python2.7 — 80x24
ouzo:secure_chat vago$ python client_threaded.py
Trying to connect to server localhost:50000 -> Ok
Load rsa keys... Ok
Challenge Response with server in progress... Server is authenticated
Challenge Response was Successful

AES key is vCP7quMfouo+Yl81hT+Q613nmbyjJrHbaQjPoaB0Z4=

Enter a message to send (Press enter to quit) # hello there
Sending message to server
Enter a message to send (Press enter to quit) # █
```

Fig. 7. Authenticated client sends a message to server



```
secure_chat — python2.7 — 80x24
Server is up and running on port 50000

Waiting for a connection...

Client Handler tid: Thread-1
Challenge Response in progress... User Authenticated!
Challenge Response was Successful

Sending Session Key... ok

*Received from: ('127.0.0.1', 53692)
  Message: hello there
  Hash check ok
Client Handler tid: Thread-2
Challenge Response in progress... User Authenticated!
Challenge Response was Successful

Sending Session Key... ok

*Received from: ('127.0.0.1', 53692)
  Message: secure-close
  Hash check ok
  Client disconnected
█
```

Fig. 8. Server receives message from authenticated client 1, Second client 2 connects concurrently to the server , Client 1 terminates the connection to the server securely with “secure-close”

ACKNOWLEDGMENTS

Nope.

REFERENCES

- https://en.wikipedia.org/wiki/Challenge-response_authentication
- <https://www.digitalocean.com/community/tutorials/understanding-the-ssh-encryption-and-connection-process>
- <http://www.hcs.wisc.edu/reading/chalresp.txt>
- http://world.std.com/~dtd/sign_encrypt/sign_encrypt7.html
- <http://cacr.uwaterloo.ca/hac/>
- <https://www.dlitz.net/software/pycrypto/api/current/Crypto-module.html>
- [https://en.wikipedia.org/wiki/RSA_\(cryptosystem\)#Padding_schemes](https://en.wikipedia.org/wiki/RSA_(cryptosystem)#Padding_schemes)
- <http://www.di-mgt.com.au/cryptopad.html#RFC1423>