CS114 - Homework 1, part 3[*]

Due 11:59pm on March 28th, 2023

Prof. Daniel Votipka

This programming assignment has two distinct parts, each requiring writing a separate program.

1 Diffie-Hellman {50 points}

First, you will create a Python program, called `dh.py`, that performs a Diffie-Hellman (DH) Key Exchange over a network. The program operates in either client or server mode. The former initiates a connection to the latter; otherwise, their functionality is identical.

The program should work as follows:

1. The client instance of `dh.py` opens a TCP connection to the server instance of `dh.py`. The latter listens on TCP port 9999 for the client's connection.
2. Once connected, the client chooses a number a uniformly at random from the range $[1, p)$ and computes $A = g^a \bmod p$. It then sends A to the server.
3. The server similarly and independently chooses a number b uniformly at random from the range $[1, p)$ and computes $B = g^b \bmod p$. It then sends B to the client.
4. The client then computes $K = B^a \bmod p$ and prints K to standard output.
5. The server then computes $K = A^b \bmod p$ and prints K to standard output.

Note that the order of the last two steps is immaterial.

There are lots of additional requirements. They are listed on the following page. Please read them carefully.

[*]Last revised on January 16, 2024.

Additional requirements

- Your program **must** be called `dh.py` (in lowercase).
- Your program should have the following command-line options:

  ```
  dh.py --s|--c hostname
  ```

  where the `--s` argument indicates that the program should wait for an incoming TCP/IP connection on port 9999; the `--c` argument (with its required `hostname` parameter) indicates that the program should connect to the machine `hostname` (over TCP/IP on port 9999).

- You must hardcode the DH parameters. Set $g = 2$ — that is, the generator should be two. Use the following 1024-bit prime (p):

  ```
  0x00cc81ea8157352a9e9a318aac4e33
    ffba80fc8da3373fb44895109e4c3f
    f6cedcc55c02228fccbd551a504feb
    4346d2aef47053311ceaba95f6c540
    b967b9409e9f0502e598cfc71327c5
    a455e2e807bede1e0b7d23fbea054b
    951ca964eaecae7ba842ba1fc6818c
    453bf19eb9c5c86e723e69a210d4b7
    2561cab97b3fb3060b
  ```

  That is, you should literally write, in your program:

  ```
  p = 0x00cc81ea8157352a9e...   # need to put the above on a single line
  ```

- Your program will be tested against a solution developed by the teaching staff. Your solution and our solution must be able to interoperate. Thus, we must agree on a message format for steps 2 and 3 above (that is, for the transmissions of A and B). The message format is simply the value of A represented in base 10 as a string, followed by a newline.

  For example, suppose that the value of A was stored in a Python variable also called A, and that `s` was a connected socket to the other instance of `dh`. Then to send A, your code would be:

  ```
  s.send( bytes(str(A) + '\n','utf8'))
  ```

  The *str*$(\cdot)$ function converts a number to a string; it's necessary for the string concatenation with the newline. The *bytes*$(\cdot, 'utf8')$ function converts the string into a set of bytes, allowing it to be transmitted.

  The value of B would be sent in the analogous manner.

- Your program should only print K (plus a newline) to standard output. It shouldn't print anything else – this risks confusing the autograder.

Hints

For modular exponentiation, use Python's pow() function with the optional third argument. For example, to compute $2^{5000000000} \bmod 23$, do:

```
Z = pow(2,5000000000,23)    # YAY!  QUICK!
print Z
```

Python knows how to do modular exponentiation in $\mathbb{Z}_n$ (i.e., modulo some number n) very efficiently. The above is far, far faster than:

```
Y = pow(2,5000000000)     # THIS WILL TAKE A WICKED LONG TIME
Z = Y % 23
print Z
```

You can convert from a string back to an int (on the receiver side) using Python's built-in *int()* function.

Rubric

Your code will be autograded. Your grade for the assignment will be the grade of the final submission. The following is a rough rubric for evaluating `dh.py`.

Tests	**Scoring**
Computes K	+50
Deductions	
Compilation / interpreter errors	-50
Server doesn't accept client connection	-40
Client can't initiate connection	-40
Non-conformant command-line options (hinders automated testing)	-15
Includes unnecessary prompts (hinders automated testing)	-10

Table 1: Grading rubric. Note that this grading rubric is not intended to be comprehensive.

2 Signed Messages {90 points}

For the second programming assignment, you will write a program that (1) generates an RSA keypair, (2) writes the public portion of the key to a file, and (3) sends a message over a network, followed by its signature.

- Your program **must** be called `signer.py` (in lowercase).
- Your program should have the following command-line options:

  ```
  signer.py --genkey | --c hostname --m message
  ```

  That is, the program works in two modes. If the `--genkey` option is specified, your program should generate a new RSA keypair. The public key must be stored in a file in the current working directory called `mypubkey.pem` (Example here). That is, you should save the key as the contents of this file. Note that `mypubkey.pem` must not contain the private key. (You may want to store the private key in another file because you'll need it to sign messages.)

  If the `-c` option is specified, then `signer.py` should open a TCP connection to port 9998 (not 9999) of `hostname` and send via that connection a **signed** copy of `message`.
- You should generate 4096 bit keys.
- When `--genkey` is issued, you should store the public key in PEM format. If `key` is the keypair generated using the Crypto.PublicKey.RSA module of PyCryptodome, then you can get the public key portion of that key via:

  ```
  pubkey_pem = key.publickey().export_key()
  ```
- Your signatures must be in PKCS#1 v1.5 format. This is done for you, so long as you use the appropriate PyCrypto Signature module (you'll really want to visit that URL), you should be fine.
- If you are implementing RSA yourself, you're doing it wrong.
- You **MUST** send signed messages in EXACTLY the following format:
  - the length of the message, in bytes, as a padded 4-byte string. You can generate this via:

    ```
    def mypad(somenum):
        return '0' * (4-len(str(somenum))) + str(somenum)
    ```

    For example, *mypad(123)* would return the string "0123".
  - the message itself (This should be unencrypted; there's no confidentiality here.);
  - the length of the signature (as specified below), in bytes, as a padded 4-byte string;
  - the signature, after being hexified via *binascii.hexlify()*, of the **SHA256** hash of the message. For example, if h is the SHA256 hash of the message, s is the connected socket, and *signature* is the signature over h, then you would do:

```
signature_hex = binascii.hexlify(signature)
s.send(signature_hex)
```

As an example, the string "Hello world" would be sent as:

```
0011Hello world10244efbedcc34d8b8653bf485e3aa3d3c43b2321a2b9d
4beae740062c03edff25e2bcf29fcc20949d74d5d6895a11745ba2481de70
7479a2930ad7776d19e22f5d9f4c80ec2777139f8c684dfcd35cd8fad9e63
a8e0072c0bdc70d26d54d4fb5f215372a4a727f6b71c3606a0a6707b0e857
2bbbedf05bdf64d8fd4583d4cdf63629dd4fb7848da38e763b50b084067bc
08171dd9cb54b334897d85e79716d0152cd91587d066582d4ca951999ae43
9b5e5c4e38728197d964a96974616ffab5435357ac2ce714c14e19380fa92
cf5bb8bd556d9c2324906ccd555448b9b82bf439e2bd41585ba4120d1997e
850c68aa4d9a14465792762fba317f0ffa6c10162b2a32864e30c125ab575
a568bb04afe7388fe9db398f9930f6f10f5d7470e7328722f3652ca364394
24e07c3a2900fcbb5b3ba32a23a81fa57c33621f0d0ca2ff846891e43b3c8
af4cdefce6dd4aa5c22874297125293b5f3ace66b86850021ffde457b4eb0
1a55ca7c2b3f64b7bf4ac175ac5729362c4c8e4ffedf73811a74ddfb123c8
d7fce77ed9af5a7ba5054ff7372715d561da4a6be09afb64119f92b8b5cc2
827837ae507ef1c83a6c31a6f8f8d957f40265bc9e93351d4314175c00dd7
9b51f0d5e867eaed79dbfd07e22015c1910ef97122a4aeb70f47cbf175fb9
fc365075b9afa455182e9c5030cfeb089edcf16e47a5637b35e586bbf3373
33978d
```

(above, the newlines are added for readability; your code should not send newlines unless they are part of the message.)

Note the format of the above: the length of the string ("0011"; i.e., 11), padded to 4 bytes; the string itself; the length of the signature ("1024"; i.e., 1024), padded to 4 bytes; and the signature of the hash of the string, in hex. Note also that there are no delimiters. This is why we have the padded lengths in there.

Hints

- If you are confused about any of the above, don't guess. Post a request for clarification to Piazza.
- You are not required to build your own server for this problem. The autograder will provide the server that you submit the signature to for evaluation. For testing purposes, you can use *nc -l 127.0.0.1 9999* to set up a listening socket on port 9998 that will print to any data sent to that socket to the command line. This will allow you to see whether everything is being sent as expected.

Rubric

The following is a rough rubric for evaluating `signer.py`.

Deductions	**Scoring**
Doesn't correctly produce signature / signature does not validate	-45
`mypubkey.pem` doesn't contain the public key	-45
`mypubkey.pem` contains both the public and private key key	-20
Compilation / interpreter errors	-90
Server doesn't accept client connection	-75
Client can't initiate connection	-75
Non-conformant command-line options (hinders automated testing)	-15
Includes unnecessary prompts (hinders automated testing)	-10

Table 2: Grading rubric. Note that this grading rubric is not intended to be comprehensive.

Submission Instructions

Submit your solutions as two separate files (*dh.py* and *signer.py*) to Gradescope.

Upload your assignment before 11:59pm on March 28th.

Please post questions (especially requests for clarification) about this homework to Piazza.