Technische Universität München
Lehrstuhl Informatik VIII
Prof. Dr.-Ing. Georg Carle
Dr. Heiko Niedermayer
Cornelius Diekmann, M.Sc.
Dr. Ralph Holz
Stefan Liebald, B.Sc.

Network Security WS14/15
Challenge 06

Task 1 Implementing Simple RSA

This time we will implement an inefficient (and not particularly secure) version of the RSA algorithm on our client and exchange some messages with our server. Have a look at the lecture slides, we will use the same notation for keys, plaintext, etc. For example, **e** is our public key, **d** the private key.

Protocol Flow. Once again we will use dictionaries and JSON for the communication between client and server; the **fat** keywords are the names of the JSON dictionary keys.
In general we have kind of an authentication protocol here.[1] To authenticate, you need to prove that you have the private key corresponding to a public key you register at the server. Once authentication is successful, your can send your tag.

- Client ⇒ Server: connect to server, server will send the first message.
- Client ⇐ Server: Welcome message (**welcome**).
- Client ⇒ Server: The Client registers her RSA public key and username at the server (**username**, **e**, **n**).
- Client ⇐ Server: The Server sends the Client a nonce encrypted with the server's own private key, together with its own *public key* (**nonce_enc**, **e**, **n**).
- Client ⇒ Server: The Client decrypts the nonce, re-encrypts it with her private key and sends it back (**nonce2_enc**) (so nonce2=nonce but encrypted with the client's key).
- Client ⇐ Server: The Server sends a **success** message if nonce2 was correct.
- Client ⇒ Server: The Client can now send her **tag**, which will be entered in our database (no encryption needed).
- Client ⇐ Server: The Server sends a final confirmation message (**success**).
- Whenever an error occurs, the server will send you the **Error** message instead of the usually expected next message.

[1]As you send your public unauthenticated, this protocol is totally vulnerable to many active attacks. However, in the absence of an active attacker, it is sufficient to verify that your RSA implementation is correct.

General instructions

- You can find the client framework to use on the lecture's home page (`netsec06_client.py`).
- There is also a framework for the RSA implementation which will be imported in the client (`rsa_skeleton.py`).
- We won't use a cryptographic library like pycrypto this time. The goal of this challenge is to implement RSA by our own. However, we restrict our implementation to primes $50 < p, q < 600$ to save ourselves some computation time. Be aware that real RSA needs way larger primes in order to be secure.
- As you learned during the lecture, padding is important when using RSA. However, since we want to focus on the RSA basics, we provide you with a (simple, primitive, and not secure) padding function. You can use it as a simple black box function, no need to change anything here. Our padding function only works if the text to pad is at least 1 bit smaller then the RSA module n. Since the only thing we encrypt (and therefore pad) in our protocol are the nonces, you don't need to worry about this, the server creates and pads the nonce accordingly. However, your encrypt (decrypt) functions should call pad (unpad).
- You can test your RSA implementation by simply executing `rsa_skeleton.py`, which has a built-in test functionality.

Specific instructions

- First we will implement RSA in `rsa_skeleton.py`[2]:
  - Begin by creating a key pair:
    - You will need to implement the Extended Euclidean Algorithm, EEA[3].
    - A function `gcd()` can use EEA's output.
    - Find e with `gcd`$(e, \phi(n)) = 1$ by simply looping over numbers $50 < e < 600$ and testing with `gcd()`. For our testcode included in `rsa_skeleton.py` we chose the smallest possible e, other values will also work, but cannot be verified using our tests.
  - Next, compute the private key. Write a function `computePrivKey(e,` $\phi(n)$`)` that makes use of the EEA. The output is an integer d.
  - Now write the functions for encryption and decryption. Don't forget padding before encryption and removing the padding after decryption.
- When you're done with the RSA implementation, fully implement the protocol flow explained above in `netsec06_client.py`.

[2] Have a look at the slides for details how RSA works.
[3] http://en.wikipedia.org/wiki/Extended_Euclidean_algorithm