Exercise Sheet Cryptography

Advanced Computer Networks and Security, DI8001, HT20, LP1

1. We discussed the One-Time-Pad cipher and showed that Eve's guesses for the key are futile as she can get any reasonably looking result. Check that the second key she guessed indeed decrypts the ciphertext to "LATER". If it does, what should be the key stream to give "NEVER" instead. If it does not, correct the key so that you do get "LATER". You can (of course) write a computer program in the language of your choice to help with this.

2. Implement a simple OTP cipher (in Python or any other language) that uses XOR for encryption. That is, your program should be able to read key material from a file and combine it with the plaintext data from another file to produce ciphertext, and vice versa (ciphertext to plaintext). Otherwise, design, implementation or interface details are up to you.

3. Consider the RSA cipher. Assume that the following two prime numbers are selected, P=7 and Q=11. Compute N and W. Select a valid public exponent E. Given the selected public exponent E, compute the private exponent D. What are the public key and private key here? Show an example encryption and decryption with these keys.

4. During the lectures we have seen how the generic RSA encryption and decryption can be done in Java using the BigInteger numbers. Big integers are the default in Python, implement the same generic RSA algorithms in Python. For this you will need the modular power operation and modular inverse. For the former, you can use `pow(x,y,z)` which gives you $x^y \bmod z$, for the latter use Google to find a suitable implementation of the Euclidean algorithm in Python.

5. Following the same principles, implement the generic Diffie-Hellman scheme in Python as it was done in Java during the lectures.

6. For RSA, the generic algorithm (see ex. 4) that works on (large) integers is in essence the same as the crypto library implementation. The library implementation only differs from the generic one in that it:

   - pads the data automatically to provide input of a fixed bit length, and
   - transforms the character/byte input into a corresponding (large) integer.

   Write a Python (or Java if you prefer) program that verifies that this is indeed one and the same thing by using one method (the generic one that you developed in exercise 4) to encrypt and the other (crypto library) to decrypt some message. To make this work you need to provide your own padding (and consequently use no-padding option with the crypto library) and represent character strings as large integers. For the latter, the following link explains how: https://stackoverflow.com/questions/8730927/convert-python-long-int-to-fixed-size-byte-array

How to get M2Crypto Python library to work

Windows

Every year students complain that it is almost impossible to get M2Crypto in Python to work, predominantly on Windows. The process and "secret" steps are indeed not 100% obvious. This is how I got to work on Windows 7 64bit (your mileage will vary depending on your version I guess):

1. Install OpenSSL for Windows matching your bit architecture. I followed the advice from: https://tecadmin.net/install-openssl-on-windows/. I did not bother with setting up environment variables, but if you plan to use the OpenSSL shell you may want to do that.
2. Install **Python2.7** from https://www.python.org/downloads/windows/. Concretely, I used the file from https://www.python.org/ftp/python/2.7.16/python-2.7.16.amd64.msi. **Important!** During installation change the option to include Python executable in PATH (it is not on by default).
3. If it did not come with one of your .NET installations or software requiring it, you also need VC Redistributable 2010. I got mine from here: https://www.microsoft.com/en-us/download/details.aspx?id=13523.
4. It is a good time to reboot at this point, just to make sure.
5. Open your command prompt / powershell and say:
    a. **`pip install wheel`**
    b. **`pip install M2CryptoWin64`**
    (The package name for win32 architecture if you happen to be there is **`M2CryptoWin32`**. In any case, **do not use** the "M2Crypto" for the package name, it will try to install the library from the sources which will 99.9% end up in a huge mess.)
6. If all that was successful (hopefully) you should be able to run the example code from the lecture, for example: **`python sha1.py`**
7. This **does not seem to work** for Python 3.7, perhaps yet another version of something (VCredist?) is needed. However, the example files I provided are both versions friendly if you happen to succeed with 3.7.

Linux (Ubuntu)

It is much simpler here, OpenSSL should come preinstalled, then install Python 2.7 (package name is **`python`**) from the *Synaptic Package Manager* (this one itself you may have to install first from the Ubuntu Software Center). Then find the package called **`python-m2crypto`** in *Synaptic* and tick it to install and apply changes, this will ask you to pull in all the required dependencies, you should be all set after this.

For Python 3 the process is not so straightforward, but possible. It requires running **`pip`** to compile M2Crypto from sources, but first also installing several development libraries. I do not provide instructions for this on purpose, for two reasons: (a) if you really need / want to do it, it means that you are strong enough to figure everything out (and it is a good exercise in itself to find out how), (b) on Ubuntu you can easily run two Python versions in parallel, so just use the 2.7 Python recipe. When having two versions in parallel, Python 3 is called with **`python3`** while just **`python`** calls Python 2.7.