CSc 466/566 Computer Security

Assignment 3

Due 23:59, Nov 4, 2014
Worth 15% (ugrads), 15% (grads)

Christian Collberg
Department of Computer Science, University of Arizona

1. Introduction

In this assignment we'll examine symmetric and public key cryptography. You should work in a team of two students.

- Implement your programs in Java. If you wish, you may use the
  - `java.util.BitSet` and
  - `java.math.BigInteger`
  - `java.security.SecureRandom`
  - `java.util.Base64.Encoder`

  packages to perform bitwise operations (useful for DES), arithmetic on large integers (useful for RSA), generation of random numbers (useful for key generation), and encoding of binary data.
- You cannot use the rest of the `java.security` package, nor any other code not written by yourselves to implement this program.
- You may get inspiration for how to implement the client-server part for part C from here: `http://way2java.com/networking/client-to-server-one-way-communication`.
- You will be evaluated on the correctness and quality of your code. I.e., the code must be readable and well documented.

NOTE: In this assignment we rely on the honor code: You cannot look at any DES/RSA implementations. I'm sure there must be billions of implementations on the web (in C, perl, Visual Basic, etc.) and in textbooks, but you cannot look at any of them. You may, of course, read about the algorithms themselves — you just can't look at any code.

2. Part A: Symmetric Key Cryptography

Implement the DES algorithm to encrypt/decrypt a file. /40

1. `java DES -h`
   - This should list out all the command line options supported by your program.
2. `java DES -k`:

- This should generate a DES key, encoded in hex, printed on the command line.
- Each time this mode is executed, a different key must be generated, i.e., you must extract some entropy from the environment.
- You should *not* generate a weak key.

3. `java DES -e <64_bit_key_in_hex> -i <input_file> -o <output_file>`:
   - This should encrypt the file `<input_file>` using `<64_bit_key_in_hex>` and store the encrypted file in `<output_file>`. Use ECB mode. Each encrypted block should be printed as 16 ascii hex characters, separated by newlines. The last block should be padded appropriately.
   - There is no restriction on the size of the input file.
4. `java DES -d <64_bit_key_in_hex> -i <input_file> -o <output_file>`:
   - This should decrypt the file `<input_file>` using `<64_bit_key_in_hex>` and store the plain text file in `<output_file>`.

NOTE: You can find DES' tables here: `http://en.wikipedia.org/wiki/DES_supplementary_material`.

NOTE: We will only test with an ASCII text file as input. You can choose to encode the input file in any way you want.

3. Part B: Public-Key Cryptography

Implement the RSA algorithm to generate key pairs, encrypt a file, and decrypt a file. /40

Your program should support the following operations:

1. `java RSA -h`
   - This should list out all the command line options supported by your program.
2. `java RSA -k <key_file> -b <bit_size>`:
   - This should generate two text files `<key_file>.public` and `<key_file>.private` containing the public and private keys, respectively, encoded in hex.
   - The size of the key should be `<bit_size>` bits. You should support at least 1024 bit keys.
   - In case `-b` option is not specified, the default bit size should be 1024.
   - Each time this mode is executed, different key pairs must be generated, i.e., you must extract some entropy from the environment.
3. `java RSA -e <key_file>.public -i <input_file> -o <output_file>`:
   - This should encrypt the file `<input_file>` using `<key_file>.public` and store the encrypted file in `<output_file>`.
   - There is no restriction on the size of the input file. You should break the input file into appropriate sized blocks (same as the key size), apply appropriate padding, and encrypt each block in ECB mode.
4. `java RSA -d <key_file>.private -i <input_file> -o <output_file>`:
   - This should decrypt the file `<input_file>` using `<key_file>.private` and store the plain text file in `<output_file>`.

NOTE: You can use `java.util.Base64.Encoder` **to encode binary data.**

4. Part C: A Hybrid Protocol

Use the code from part A and part B to implement a simple chat program using a hybrid protocol to set up encrypted communication. /20

1. `java CHAT -h`
   - This should list out all the command line options supported by your program.
2. `java CHAT -alice -e bob.public -d alice.private -p port -a address`:
   - This is what Alice would run. She knows Bob's public key and the port and IP address on which to talk to him.
   - She starts up first, and lays down to wait for Bob to come online.
3. `java CHAT -bob -e alice.public -d bob.private -p port -a address`:
   - This is what Bob would run. He knows Alice's public key and the port and IP address on which to talk to her.
   - When Bob starts up, he initializes the hybrid protocol, by generating a DES key, encrypting it with Alice's public key, and sending her the encrypted package.
   - Alice returns the string "OK" (encrypted) to Bob.
   - Once they are done with this simple handshake, they take turns exchanging one line messages read from the command line, encrypted with DES in ECB mode.

Here's an example:

Alice	Bob
`java CHAT -alice -e bob.public -d alice.private -p 3000 -a 127.0.0.1`	
Alice waits.	
	`java CHAT -bob -e alice.public -d bob.private -p 3000 -a 127.0.0.1`
	Bob generates a DES key 0x12345678, encrypts it with alice.public, *sends the package to Alice. He waits.*
Alice receives the package, decrypts it with her private key, sends ''OK'' (encrypted with the key she just received) to Bob. She waits.	
	Bob receives the package and decrypts. If it says ''OK'', he continues, otherwise he exits the program. He starts typing on the command line.
	Hi Alice! Nice to talk to you! \n.
	The line is ecnrypted with DES in ECB mode and sent to Alice. He waits.
Alice receives the package, decrypts with DES, the message is printed on the command line.	
Hi Alice! Nice to talk to you! \n.	
She starts typing.	
Hi Bob, s'up? \n	