Socket programming

A Brief Introduction Network Programming and Encryption Rich Lundeen Introduction This paper is the report of an exercise in socket programming and encryption. The objectives of this project were as follows: ? ? ? ? To create two programs, a client and a server The server broadcasts its public key to all who connect A client attaches to the server and exchanges a session key using public key encryption A message is passed between the client and the server encrypted with a symmetric key There are weaknesses in this scheme. There is no real authentication to speak of.

On the other hand, something like this could be considered useful, especially if the server's public key were verified with a fingerprint locally. It would also be fairly trivial to extend the already trivial program so that it authenticates with a special and specific CA. There are many similarities with this scheme and how ssl or other " real" cryptographic techniques work. Many times, public key encryption is only used to transfer a key to be used with symmetric cyphers which are much more computationally fast. Not all source code is included here (although most the interesting source code is).

To view the complete source code, download it from http://webstersprodigy. net/media/code/networkproject. tar. gz . Specifications For a socket API, I chose the Berkely model since it seems to be the standard (although Java Sockets and Winsock are fairly popular as well, they resemble the Berkely model). As a programming language I chose Python. At this level efficiency was not nearly as important to me as readability and programming ease. I chose to implement RSA as the public key algorithm, and wanted to do this from scratch.

I made the decision to write the cryptography algorithms from scratch anticipating that truly random numbers and big integers would provide important insight into some of the facets of cryptography, and I was not mistaken. I chose RSA (as opposed to other probably superior cryptographic algorithms like ECC) because it is intuitively simple and extremely popular. However, it is much easier (and better) to import modules from a library, so for the symmetric cypher I chose to use the built in Python cryptographic libraries.

Because of the abstraction here, it would be trivial to change algorithms, but I chose AES because of its strength (as opposed to DSA variants) and popularity. These programs were only tested on a Debian Linux platform, but because of the portability of Python they should be portable to most other Operating Systems with some minor changes. Most notably this program makes use of /dev/urandom which is only available in Linux (Unix?). Necessary dependencies for this to run are python and python-crypto. This program is in no way secure whatsoever. For one thing, the keys are not yet generated properly.

This program was written a learning exercise. Although I wrote most of these programs together, I will program in the order of the libraries used, the server, and then the client. Detailed Execution Overview Execute the server on the server (. /server. py). The server listens for connections to port 51424 (which is considered a 'standard' port). It then sends out it's public key. A session key is then sent by the client encrypted with the server's public key using rsa. The server then encrypts a secret message with the session key using aes. The client then decrypts the message.

Cryptography Here is rsa. py, where the rsa class is implemented. It is fairly simple especially because of the simplifications involved (they are mostly commented below). #! /usr/bin/env python #This is my (probably inefficient and insecure) version of rsa. #For real security, the cryptography module should be included #It takes two random numbers as init parameters. This means #randomness must be provided elsewhere #it also prints out nice messages for the sake of understandability import random,