**Encrypted Message Exchange Simulator**

By Chris Maguschak

CS463 at ODU

Purpose: The purpose of this program is to simulate encrypted exchanges between users, as well as how secure/insecure their messages are depending on what encryption methods are used.

In this simulation, there are 3 users, Alice, Bob, and Charlie. Alice and Bob are supposed to be the 2 users trying to communicate, and Charlie is supposed to take on the role of the hacker. However, anyone can have any role, and you can even add or subtract users from the simulation. For the purposes of the simulation, Alice and Bob are simply trying to have a secure conversation with each other, while Charlie is trying to snoop in on their conversation.

At the beginning of the simulation, you are given options for which encryption method you’d like to use as well as which block cipher mode of operation you’d like to use and whether or not you’d like to sign your messages. For encryption, you are given the option between RSA encryption or ElGamal encryption, as well as, just for fun, the Caesar cipher, Atbash cipher. Unencrypted plaintext was also added, to demonstrate how dangerous it is to transmit messages through unencrypted media such as SMS. For block cipher modes of operation (which are often abbreviated to BMOP in the code), you have the choice of ECB, CBC, OFB, and CFB. Due to how ElGamal encrypts messages (requiring a different random number be generated each time), OFB and CFB don’t quite work in combination with ElGamal encryption, but you’re more than welcome to give it a try, anyway. Also, due to last minute difficulties, CFB just flat out does not work. I don’t know why, it just doesn’t seem to decrypt properly, even when encryption is disabled. In retrospect, I probably bit off way more than I could chew with this project, as the amount of cases I have to bug test for seems to far exceed the amount of time I have left for this project. So, if you run into a bug with this simulation…sorry…

For all message exchanges, the messages are NEVER stored as plaintext (unless the “no encryption” method was chosen, though even then, the system still considers the messages encrypted using a cipher that just so happens to not change anything). They are always stored as encrypted text, and the only plaintext version of that message is a personal copy the sender keeps for themselves, which nobody but said user is able to access. As a result, for Charlie to read Alice or Bob’s messages, he has to crack their encryption schemes. In some situations, this is rather easy. For example, using ECB, it’s simply a matter of solving a cryptoquip puzzle, or if it uses the Caesar cipher, finding the private key is as simple as taking the public key and subtracting it from 26. However, for more secure methods, such as RSA encryption with CBC, this is much more difficult. In fact, in a real world cryptosystem with keys with thousands of bits, this would be near impossible, but luckily for Charlie, this simulation is just for demonstration purposes, and only uses 32-bit keys. It is therefore a lot easier to find the private key from the public key.

There’s a good reason that 32-bit keys were used, and that’s that, in order to accurately multiply two n-bit numbers mod a third n-bit number, you need to store their (pre-modulo) product as a 2n-bit number. The largest integers that are officially supported on C++ std is 64-bit numbers. Going any bigger would require using a third party library for big integers, which I felt a. was a bit overkill and b. might mean this program isn’t compatible with all devices. So, I had to stick to 32-bits. Unfortunately, it wasn’t until very late in the game that I realized that this restricted me from implementing DES, which uses 64-bit blocks, and AES, which uses 128-bit blocks. So my only real options, unless I found a workaround (which I didn’t, at least not before submitting this) was to only use RSA and ElGamal (Elliptic curve cryptography was also an option, but I ran out of time). I also added a few others just for fun. Ultimately, though, since AES and DES weren’t implemented, the only practical tools I had were both asymmetric key cryptography, hence, I didn’t get the chance to implement any feature whereby two parties exchange their keys via a Diffie Hellman exchange or 3 pass protocol.

Detailed mechanics: Initially, you choose between 4 methods of encryption, then between 4 methods of block cipher modes of operation, then you choose whether or not you want the messages to be signed. You are then logged in as Alice. You are now in the home menu.

1. Send message: Choose who you send it to, then type your message, then press enter to send it. When you type out messages, spaces are temporarily disabled as delimiters, meaning that you can use as many spaces as you want in a message. And letters, and numbers, and punctuation, and symbols. The only thing you can’t include in messages is endlines, since the system doesn’t know you’re done typing until you hit enter. When the message is sent, it an offline, unencrypted copy is saved to the current user’s system, and an encrypted version of the message is sent to the system’s simulated internet, where anybody can read the message, but nobody can understand it without the decryption key.

2. Read messages: First, choose which conversation you want to view. Do you want to view your conversation with Bob, Charlie, who? It’ll also display how many unread messages you have, if any. Since you have the decryption key, the messages to you will display just as they were typed, completely unencrypted. As for the messages sent by you, you will be shown an unencrypted copy saved to your device.

3. Change role: Change which user you want to play as. After all, how else can Bob send a message back to Alice?

Just to clarify, for the purpose of the simulation, you aren’t one person switching between multiple devices, but instead you are taking on the role of multiple different people in the simulation (like one person playing multiple people in a play). That means that, if you start out as Alice, then switch users to Charlie, then view one of Charlie’s messages to Bob, then switch back to Alice, that doesn’t count as Alice snooping in on Bob and Charlie’s conversation. Furthermore, if you start out as Charlie, then switch to Alice, then view Alice’s private key, then switch back to Charlie and save her key, that’s cheating, because there’s no way Charlie could have legitimately obtained the key unless he was looking over her shoulder or he just happened to guess it.

4. View all messages (encrypted): This allows you to view all the messages that have been sent since the start of the simulation. However, they’re all encrypted, so they all look like nonsense. If you go to settings, you can change whether the raw text is displayed as strings, decimal numbers, or hex dumps.

5. View public keys: You can view all the keys that have been made publicly available. You can also view your own private key. Keep in mind, just to keep the simulation honest and accurate, the function which retrieves private keys will throw an exception unless the current user is also the person whom the keys belong to.

6. Hacking tools: This takes you to another menu where you are provided with all sorts of hacking tools for the purpose of the simulation. More on that later

7. Admin: This allows you to edit the simulation. From adding users to removing users to changing users’ names. For the purposes of the simulation, nobody is actually performing the actions here, users are simply being willed into/out of existence. Or, you can pretend that they always/never existed.

8. Settings: A set of miscellaneous settings. Namely, here you can change how raw, encrypted text is formatted, and you can change whether the public key screen displays your private key

9. Exit: Self explanatory, it exits the simulation.

Hacking tools: Here, you’re provided with all sorts of tools that a nefarious third party might use to interfere with others’ conversations. Since this simulation only uses 32-bit numbers, that makes it much easier to determine the private key given the public key, and hence, you’re provided with a slew of math tools allowing you to crack their codes.

1. <- Go Back: Go back to the home menu.

2. Snoop: Allows you to snoop in on others’ conversations. Any conversations that you are a part of will automatically be decrypted. Likewise, if you manage to find and save anybody else’s public keys, this will automatically decrypt those messages as well. Any other messages, however, will be displayed as raw text.

3. Spoof: Send a message to one person pretending to be someone else. Since all you need to encrypt a message and send it to someone is their public key, it’s very easy to just do so pretending to be someone else. However, if signatures are enabled, you have to be careful, as it’ll warn the recipient that this message was spoofed. However, if you learn the private key of the person you’re spoofing, you can bypass this by using their private key to sign it.

4. Save/Edit keys: Once you learn someone’s private key, you can save it here. Likewise, if you save it wrong, you can go here and edit their key. After that, all messages they receive or have received will be automatically decrypted when you use the snoop feature, allowing you to freely access their messages. It should be noted that if you save an incorrect key, snooping will likely just show you a bunch of nonsense. This is not the case for OFB or CFB block cipher modes of operation, as those methods can be decrypted using the public key. However, due to the way it works, snooping won’t try to decrypt a message unless it has a key saved for that person (even if it’s wrong). This even applies for when the encryption method is plaintext (i.e. no encryption).

5. View keys: View all the keys you’ve saved. You might notice that your own private key is automatically saved. This is done for convenience, as well as making things easier to program.

6. Remove keys: Just as you can save keys, you can also remove them. Removing a key will prevent you from snooping on the person in question’s messages, and will instead just show them as encrypted nonsense.

7. Math tools: Several math tools that make sniffing out peoples’ private keys easier. More on that later.

Math tools: There are several math tools included, such as prime factorization, Euler’s totient, discrete logarithms, modular inversion, etc. These tools allow for a hacker to easily compute the private keys given the public keys. In a real world cryptosystem, finding such keys would most likely not be this easy, but with the advent of quantum computing, a new way of computing will emerge that will allow several NP hard problems to be solvable in polynomial time, including the problem of factoring and discrete logarithms. Hence, these tools and their integration into the simulation demonstrate how hackers will be able to crack certain encryption methods once such algorithms become readily available.

Many of these are functions I actually had written prior to this project, such as the prime factorization tool and the modular inverse and modular exponentiation functions.

1. <- Go Back: Go back to the hacking menu

2. Prime Factor: Gives you the prime factorization of the inputted integer. Special case: factoring 0 is just 0, factoring 1 is “Empty product”. Negative numbers don’t register propery, since it takes unsigned integer inputs. This works by taking all numbers up until the square root of the input, and finding if this number divides it. Due to clever/tedious progression, it skips all numbers that are divisible by 2, 3, 5, and 7 (except, of course, 2, 3, 5, and 7), thus cutting out over 75% of all the numbers that would have to be checked. In addition, every time it divides by a number, it recalculates the square root to the square root of what’s leftover. For most numbers, this would be a huge speedup, but in the cases this is primarily used for, i.e. the product of 2 primes, it’s not much of a speedup.

3. Euler’s Totient: Computes Euler’s totient function. This is not to be confused with the Carmichael totient, which I did not have time to implement for all integers, and which the RSA algorithm uses as its totient (though usually they both equal). For the product of two primes, the Euler’s totient is the product between those primes minus 1, while the Carmichael totient is the LCM between those primes minus 1.

4. Discrete Logarithm: Computes the discrete logarithm. Originally, this used the Pollard’s rho algorithm for logarithms (O(mod1/2)), but since I couldn’t get it to work properly, I just wrote it to find it via brute force, searching through all numbers [0,mod-1) and returning whichever ones satisfy ab mod m = c. As a result, it takes quite a few seconds to return the result.

5. Modular Inverse: Computes the modular inverse. Not all algorithms here are NP hard, but some of them, such as the modular inverse, aren’t readily available on most calculators, and hence their inclusion here comes in handy. This computation is performed by the Extended Euclidean Algorithm, which computes the solution to Bezout’s identity and, if the GCD is 1, spits out the modular inverse of both numbers with respect to each other.

6. Modular Exponent: Computes one number raised to another number modulo a third number. It does so using exponentiation by squaring.

7. Modular Product: Computes the products between two numbers modulo a third number.

8. GCD: Computes the greatest common denominator. This actually doesn’t use the Extended Euclidean Algorithm, but uses another, even faster algorithm I learned on StackExchange a few years back and modified over the years. It basically reduces both inputs using only addition, subtraction, and bitwise right shifts, until one of them is 0, hence making the GCD just the other number.

9. LCM: Computes the least common multiple. It does so by multiplying both numbers and dividing by the GCD.

Admin Tools: Here, you’re not exactly taking the role of one specific person, but instead taking the role as the controller of the simulation, giving you better control over the things that can be simulated. Here, you can add users, remove users, change their name, or switch users (though you could already do that last one). Changes you make here (or anywhere in the simulation, really) do not carry over to the next time you run the simulation.

1. <- Go Back: Takes you back to the main menu.

2. Switch user: Takes you to the switch user menu.

3. Add user: Add a user

4. Remove user: Remove a user (be careful!)

5. Change usernames: Change the name of users (for example, if you want to change Charlie to Carol)

Encryption algorithms:

There are 2 primary algorithms implemented here, RSA and ElGamal. They’re both asymmetric algorithms, whereby the sender uses the receiver’s private key to encrypt the message, then the receiver uses their private key to decrypt it. Likewise, signature algorithms are implemented to help prevent users from spoofing other users. For RSA, it simply encrypts the message’s hash using the sender’s private key, then it can be decrypted using their public key to verify authenticity. However, for ElGamal, another algorithm was developed specifically for signing ElGamal messages, and it was programmed in here.

In the ElGamal method, encryption consists of using the public key, as well as generating a random integer y mod the prime modulus. In typical use, a different y is generated for each block of encrypted text, resulting in 2 blocks of output for every one block of input. However implementing this turned out to be much, much, much more difficult than expected, and ultimately the idea was nixed in favor of having y computed once per message and the same value of y being reused for each block.

For Caesar and Atbash ciphers (which, again, were just added for fun), there isn’t any real signature being used, it just hashes the message and doesn’t even bother encrypting it. The same goes for plaintext.

Hacking:

In order to effectively use these hacking tools, it’s important to understand what the public keys are and how they determine the private keys.

RSA: The public keys are the modulo and the exponent, respectively, while the private key is the modular inverse of the exponent MOD the modulo’s Carmichael totient.

To compute the private key, you must first factor the modulus. Then, you take the two prime factors, subtract 1 from each of them, and take their least common multiple. From here, you now have the Carmichael totient of the modulus. Then, you take the inverse of the exponent MOD the totient, and you now have the private key. Save that as this user’s private key, and you can now snoop on all messages sent to this individual.

ElGamal: The public keys are the modulo m, the generator g, and h = gx, respectively. Meanwhile, the private key is the exponent x.

To compute the private key, you must take the base g logarithm of h, modulo m.

Caesar: The public key is how much each letter is shifted by the cipher. You can probably figure this out for yourself, but the private key is just the negative of the public key, modulo 26. You can probably even figure it out in your head.

Atbash, Plaintext: Neither atbash nor plaintext uses a key. More specifically, their public key is an array of length 0, and their private key is an array of length 0. All you have to do to successfully snoop on users is to register their non-existent keys.

Grouping: When text is converted into a stream of integers, each set of 2 characters is turned into a 4 byte integer. This might seem wasteful, but it was originally implemented as a way of preventing the input number from being larger than the modulo when working with RSA and ElGamal encryption, and I suppose I just forgot to change it for Caesar, Atbash, and plaintext. To be fair, the way it’s implemented, I could very easily have 3 chars per 4 byte integer, but I suppose I just never got around to doing that.

Much to my dismay, while I made sure to comment as I went along, in the end I feel I was not able to provide nearly as many comments as I should have when making this project, and I fear some parts of it might not be as clear as others.