###### Experiment Number: 01

**TITLE:** Pseudorandom number generator for generating the long-term private key.

**PROBLEM STATEMENT:** Write a program to generate a pseudorandom number generator for generating the long-term private key and the ephemeral keys used for each signing based on SHA-1 using Python/Java/C++. Disregard the use of existing pseudorandom number generators available.

**OBJECTIVES:**

1. To generate a pseudorandom numbers that can be used to generate a long term private key.
2. Share the key over network between a client and a server using Diffie-Hellman key exchange.
3. Perform encryption-decryption using the generated key.

**THEORY:**

A **pseudorandom number generator** (**PRNG**) is an algorithm for generating a sequence of numbers whose properties approximate the properties of sequences of random numbers. The PRNG-generated sequence is not truly random, because it is completely determined by a relatively small set of initial values, called the PRNG's seed*.*



Fig: Generating Random Number using PRNG Function

A **seed** is a number (or vector) used to initialize a pseudorandom number generator. For a seed to be used in a pseudorandom number generator, it does not need to be random. Because of the nature of number generating algorithms, so long as the original seed is ignored, the rest of the values that the algorithm generates will follow probability distribution in a pseudorandom manner. The choice of a good random seed is crucial in the field of computer security. When a secret encryption key is pseudorandomly generated, having the seed will allow one to obtain the key. If the same is deliberately shared, it becomes a secret key, so two or more systems using matching pseudorandom number algorithms and matching seeds can generate matching sequences of non-repeating numbers which can be used to synchronize remote systems.

Seeds are often generated from the state of the computer system (such as the time), a cryptographically secure pseudorandom number generator or from a hardware random number generator.

Various Types of PRNG’s:

1. **Linear Congruential Generator**

A linear congruential generator is an algorithm that yields a sequence of pseudo-randomized numbers calculated with a discontinuous piecewise linear equation. The theory behind them is relatively easy to understand, and they are easily implemented and fast, especially on computer hardware which can provide modulo arithmetic by storage-bit truncation.

The generator is defined by the recurrence relation:

X_{n+1} = \left( a X_n + c \right)~~\bmod~~m

where Xis the sequence of pseudorandom values, and

 m,\, 0<m – the "modulus"

 a,\,0 < a < m– the "multiplier"

 c,\,0 \le c < m– the "increment"

 X_0,\,0 \le X_0 < m– the "seed" or "start value"

1. **Blum Blum Shub Generator**

Blum Blum Shub takes the form:

x_{n+1} = x_n^2 \bmod M,

Where,

*M* = *pq* is the product of two large primes *p* and *q*. At each step of the algorithm, some output is derived from *xn*+1; the output is commonly either the bit parity of *xn*+1 or one or more of the least significant bits of *xn*+1*.*

The seed *x*0 should be an integer that is co-prime to *M* (i.e. *p* and *q* are not factors of *x*0) and not 1 or 0.

The two primes, *p* and *q*, should both be congruent to 3 (mod 4) (this guarantees that each quadratic residue has one square root which is also a quadratic residue) and gcd(*[φ](https://en.wikipedia.org/wiki/Euler%27s_totient_function" \o "Euler's totient function)*(*p* − 1), *φ*(*q* − 1)) should be small (this makes the cycle length large).

An interesting characteristic of the Blum Blum Shub generator is the possibility to calculate any *xi* value directly (via Euler's Theorem):

x_i = \left( x_0^{2^i \bmod \lambda(M)} \right) \bmod M,

Where, \lambda is the Carmichael function.

**Diffie–Hellman key exchange** (**D–H**) is a specific method of securely exchanging cryptographic keys over a public channel and was one of the first public-key protocols. The Diffie–Hellman key exchange method allows two parties that have no prior knowledge of each other to jointly establish a shared secret key over an insecure channel. This key can then be used to encrypt subsequent communications using a symmetric key cipher.

Let the users be named Alice and Bob. First, they agree on two prime numbers g and p, where pis large and gis a primitive root modulo p. The numbers g and p need not be kept secret from other users. Now Alice chooses a large random number aas her private key and Bob similarly chooses a large number *b*. Alice then computes A=g^a (mod p), which she sends to Bob, and Bob computes B=g^b (mod p), which he sends to Alice.

Now both Alice and Bob compute their shared key K=g^(ab) (mod p), which Alice computes as

|  |
| --- |
| K=B^a (mod p)=(g^b)^a (mod p) |

and Bob computes as

|  |
| --- |
| K=A^b (mod p)=(g^a)^b (mod p). |

Alice and Bob can now use their shared key Kto exchange information without worrying about other users obtaining this information.

**MATHEMATICAL MODEL:**

*Let U={I,O,F,S,T}*

I= Input {I1,I2}

I1= {Seed}

I2= {Message}

O= Output {Decrypted Message}

S= Case of success {message is successfully encrypted at client end, sent over the network, and then correctly decrypted by server.}

F= Case of failure {Message is incorrectly decrypted by the server.}

T= Functions used{ random\_number\_generator()}

Seed PRNG(seed) Random No

**IMPLEMENTATION DETAILS / DESIGN LOGIC:**

Algorithm for Random Number Generation:

1. Take input of seed from the user(or assume it).
2. Assume a, b, c.
3. Multiply seed by a.
4. Add b to the product,
5. Perform sum mod c. The result is the desired random number.



Fig: Process Flow Diagram of Random Number generation

Algorithm for Diffe-Hellman Key Exchange:

1. Generate X and Y as servers and clients random numbers using PRNG function.
2. Assume shared\_base and shared\_prime which is known to both server and client.
3. At server, calculate K1(sever key) using:

*K1 = ((shared\_base)X)% shared\_prime*

At client, Calculate K2(client key) using:

*K2 = ((shared\_base)Y)% shared\_prime*

1. Exchange K1 and K2 between server and client.
2. At Server, calculate common key using:

K = (K1)X % shared\_prime

At Client, calculate common key using:

K = (K2)Y % shared\_prime

1. K is the desired common Key to be used for Encryption.

Algorithm for Encryption:

1. Accept the input message from the user.
2. Convert each character of the message into its ASCII value.
3. Add Common Key K to each ASCII value obtained in step 2.

*E = ASCII(Msg) + K*

1. Convert the ASCII values into a list of strings and send it over the network.

Algorithm for Decryption:

1. Read the encrypted message and convert its data type to int.
2. Subtract each number by Common Key K.

*D = E - K*

1. The result of step 2 is the ASCII value corresponding to each character of the message. Convert ASCII to String.
2. Message is displayed as output.

**Execute the program with the following commands:**

Python server.py

Python client.py

**TEST CASES:**

|  |  |
| --- | --- |
| **Test case Input** | **Test case Output** |
| Generate Random Numbers using PRNG function | Random Numbers generated successfully |
| Exchange Keys using Diffie Hellman key exchange to compute a common key. | Common key generated successfully |

**CONCLUSION:**

We have successfully generated pseudorandom numbers to generate a long term private key.

**COURSE OUTCOMES ACHIEVED:**

|  |  |
| --- | --- |
| **COURSE OUTCOME** | **ACHEIVED** |
| To solve problems using mathematical modeling | **√** |
| To use software design methods and testing |  |
| To solve problems for multicore or distributed, concurrent/parallel environments | **√** |

**FAQ’s**

1. What are Pseudorandom Numbers?
2. What are the design considerations for stream cipher?
3. What is TRNG (True Random Number Generator)?
4. What is PRF (Pseudorandom function)?
5. What are the two forms of unpredictability required in stream of pseudorandom numbers?
6. Are preudorandom numbers always random?
7. Describe working of existing pseudorandom number generators.
8. Explain Man in the Middle Attack in Diffie-Hellman Key Exchange.