Министерство образования Республики Беларусь

Учреждение образования

«Брестский Государственный технический университет»

Кафедра ИИТ

**Лабораторная работа №1**

По дисциплине «Криптографические методы защиты информации»

Тема: «Алгоритмы обмена ключами»

**Выполнил:**

Студент 3 курса

Группы ИИ-21

Кирилович А. А.

**Проверил:**

Хацкевич А. С.

Брест 2023

**Цель:** изучить алгоритмы обмена ключами. Практически реализовать алгоритмы обмена ключами.

**Ход работы:**

**Вариант**:Расширенный EKE , RC-4

Код сервера:

mod eke;

use eke::{generate\_prime, generate\_private\_key, compute\_public\_key, compute\_shared\_secret};

use num\_bigint::BigUint;

use std::io::{Read, Write};

use std::net::{TcpListener, TcpStream};

use std::thread;

use byteorder::{BigEndian, ReadBytesExt, WriteBytesExt};

use rc4::Cipher;

fn main() {

let listener = TcpListener::bind("127.0.0.1:8081").expect("Failed to bind");

for stream in listener.incoming() {

match stream {

Ok(mut client) => {

let prime = generate\_prime(512);

let generator = BigUint::from(2u32);

send\_biguint(&mut client, &prime);

send\_biguint(&mut client, &generator);

let client\_public\_key = receive\_biguint(&mut client);

let server\_private\_key = generate\_private\_key(&prime);

let server\_public\_key = compute\_public\_key(&generator, &prime, &server\_private\_key);

send\_biguint(&mut client, &server\_public\_key);

let shared\_secret = compute\_shared\_secret(&client\_public\_key, &prime, &server\_private\_key);

println!("Shared secret: {}", shared\_secret);

let mut rc4 = rc4::Cipher::new(&shared\_secret.to\_bytes\_be()).unwrap();

thread::spawn(move || {

handle\_client(&mut rc4, &mut client);

});

}

Err(e) => {

eprintln!("Error accepting connection: {}", e);

}

}

}

}

fn handle\_client(rc4: &mut Cipher, client: &mut TcpStream) {

let mut buffer = [0; 1024];

loop {

match client.read(&mut buffer) {

Ok(0) => {

println!("The client closed the connection");

break;

}

Ok(bytes\_read) => {

let mut dst: Vec<u8> = vec![0; bytes\_read];

rc4.xor(&buffer[0..bytes\_read], &mut dst);

if let Ok(string\_result) = std::str::from\_utf8(&dst) {

println!("Received from client: {}", string\_result);

} else {

println!("Conversion to String failed");

}

}

Err(e) => {

eprintln!("Error reading data: {:?}", e);

break;

}

}

}

}

fn send\_biguint(stream: &mut dyn Write, num: &BigUint) {

let num\_bytes = num.to\_bytes\_be();

stream.write\_u64::<BigEndian>(num\_bytes.len() as u64).unwrap();

stream.write\_all(&num\_bytes).unwrap();

}

fn receive\_biguint(stream: &mut dyn Read) -> BigUint {

let size = stream.read\_u64::<BigEndian>().unwrap() as usize;

let mut num\_bytes = vec![0; size];

stream.read\_exact(&mut num\_bytes).unwrap();

let num = BigUint::from\_bytes\_be(&num\_bytes);

num

}

Код клиента:

mod eke;

use eke::{generate\_private\_key, compute\_public\_key, compute\_shared\_secret};

use num\_bigint::BigUint;

use std::io::{self, BufRead, Write, Read};

use std::net::TcpStream;

use byteorder::{BigEndian, ReadBytesExt, WriteBytesExt};

use rc4::Cipher;

fn main() {

match TcpStream::connect("127.0.0.1:8081") {

Ok(mut server) => {

let prime = receive\_biguint(&mut server);

let generator = receive\_biguint(&mut server);

let client\_private\_key = generate\_private\_key(&prime);

let client\_public\_key = compute\_public\_key(&generator, &prime, &client\_private\_key);

send\_biguint(&mut server, &client\_public\_key);

let server\_public\_key = receive\_biguint(&mut server);

let shared\_secret = compute\_shared\_secret(&server\_public\_key, &prime, &client\_private\_key);

println!("Shared secret: {}", shared\_secret);

let mut rc4 = rc4::Cipher::new(&shared\_secret.to\_bytes\_be()).unwrap();

handle\_server(&mut rc4, &mut server);

}

Err(e) => {

eprintln!("Error connecting to server: {}", e);

}

}

}

fn handle\_server(rc4: &mut Cipher, server: &mut TcpStream) {

let stdin = io::stdin();

let mut reader = stdin.lock();

let mut buffer = String::new();

loop {

reader.read\_line(&mut buffer).expect("Error reading line from console");

let mut dst: Vec<u8> = vec![0; buffer.len()];

rc4.xor(buffer.as\_bytes(), &mut dst);

server.write\_all(&dst).expect("Error sending data");

buffer.clear();

}

}

fn send\_biguint(stream: &mut dyn Write, num: &BigUint) {

let num\_bytes = num.to\_bytes\_be();

stream.write\_u64::<BigEndian>(num\_bytes.len() as u64).unwrap();

stream.write\_all(&num\_bytes).unwrap();

}

fn receive\_biguint(stream: &mut dyn Read) -> BigUint {

let size = stream.read\_u64::<BigEndian>().unwrap() as usize;

let mut num\_bytes = vec![0; size];

stream.read\_exact(&mut num\_bytes).unwrap();

let num = BigUint::from\_bytes\_be(&num\_bytes);

num

}

EKE:

use num\_bigint::BigUint;

use num\_traits::{One, Zero};

use rand::Rng;

use num\_integer::Integer;

// Random Prime number generation

pub fn generate\_prime(bit\_size: usize) -> BigUint {

let mut rng = rand::thread\_rng();

loop {

let prime\_candidate = generate\_random(bit\_size);

if is\_prime(&prime\_candidate) {

return prime\_candidate;

}

}

}

// Random number generation dimension bit\_size bit

fn generate\_random(bit\_size: usize) -> BigUint {

let mut rng = rand::thread\_rng();

let random\_bytes: Vec<u8> = (0..(bit\_size + 7) / 8)

.map(|\_| rng.gen())

.collect();

BigUint::from\_bytes\_be(&random\_bytes)

}

// Verify prime number with Miller-Rabin test

fn is\_prime(n: &BigUint) -> bool {

if n == &BigUint::zero() || n == &BigUint::one() {

return false;

}

let mut rng = rand::thread\_rng();

let num\_trials = 10;

for \_ in 0..num\_trials {

let a = BigUint::from(2u32 + rng.gen\_range(0..100));

if !miller\_rabin(n, &a) {

return false;

}

}

true

}

// Miller-Rabin test implementation prime number

fn miller\_rabin(n: &BigUint, a: &BigUint) -> bool {

let one = BigUint::one();

let d = n - &one;

let mut s = 0;

let mut d = d.clone();

while d.is\_even() {

d >>= 1;

s += 1;

}

let mut x = a.modpow(&d, n);

if x == one || x == n - &one {

return true;

}

for \_ in 0..s - 1 {

x = x.modpow(&2u32.into(), n);

if x == one {

return false;

}

if x == n - &one {

return true;

}

}

false

}

// Private key random generation

pub fn generate\_private\_key(prime: &BigUint) -> BigUint {

let mut rng = rand::thread\_rng();

loop {

let private\_key = generate\_random(prime.bits().try\_into().unwrap());

if private\_key > BigUint::zero() && private\_key < \*prime {

return private\_key;

}

}

}

// Public key computation from private key

pub fn compute\_public\_key(generator: &BigUint, prime: &BigUint, private\_key: &BigUint) -> BigUint {

generator.modpow(private\_key, prime)

}

// Secret shared key generation from public and private key

pub fn compute\_shared\_secret(public\_key: &BigUint, prime: &BigUint, private\_key: &BigUint) -> BigUint {

public\_key.modpow(private\_key, prime)

}

**Вывод:** в ходе лабораторной работы я изучил алгоритмы обмена ключами, а также реализовал клиент-серверное приложение, позволяющее обмениваться зашифрованными сообщениями.