МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМЕНИ Н.Г. ЧЕРНЫШЕВСКОГО»

Кафедра теоретических основ компьютерной безопасности и криптографии

Схемы аутентификации

ОТЧЁТ
ПО ДИСЦИПЛИНЕ
«КРИПТОГРАФИЧЕСКИЕ ПРОТОКОЛЫ»

студента 5 курса 531 группы
специальности 10.05.01 Компьютерная безопасность
факультета компьютерных наук и информационных технологий
Серебрякова Алексея Владимировича

Преподаватель
аспирант _____ Р. А. Фарахутдинов
подпись, дата

Саратов 2023

1 Постановка задачи

Цель работы:

- Изучение протокола аутентификации Шнорра (Схемы Шнорра) и его программная реализация.

Задачи работы:

- Изучить протокол Шнорра, его сильные и слабые стороны;
- Привести программную реализацию протокола.

2 Теоретические сведения

Схемы аутентификации и электронной подписи — одни из наиболее важных и распространенных типов криптографических протоколов, которые обеспечивают целостность информации.

Понять назначение протоколов аутентификации можно легко на следующем примере. Предположим, что у нас есть информационная система, в которой необходимо разграничить доступ к различным данным. Также в данной системе присутствует администратор, который хранит все идентификаторы пользователей с сопоставленным набором прав, с помощью которого происходит разграничение доступа к ресурсам. Одному пользователю может быть одновременно разрешено читать один файл, изменять второй и в то же время закрыт доступ к третьему. В данном примере под обеспечением целостности информации понимается предотвращение доступа к системе лиц, не являющихся её пользователями, а также предотвращение доступа пользователей к тем ресурсам, на которые у них нет полномочий. Наиболее распространенный метод разграничения доступа, парольная защита, имеет массу недостатков, поэтому перейдем к криптографической постановке задачи.

Схема Шнорра (англ. schnorr scheme) — одна из наиболее эффективных и теоретически обоснованных схем аутентификации. Безопасность схемы основывается на трудности вычисления дискретных логарифмов. Предложенная Клаусом Шнорром схема является модификацией схем Эль-Гамаля (1985) и Фиата-Шамира (1986), но имеет меньший размер подписи. Схема Шнорра лежит в основе стандарта Республики Беларусь СТБ 1176.2-99 и южнокорейских стандартов KCDSA и EC-KCDSA. Она была покрыта патентом США 4999082, который истек в феврале 2008 года.

В протоколе имеются два участника — Алиса, которая хочет подтвердить свой идентификатор, и Боб, который должен проверить это подтверждение. У Алисы имеется два ключа — K_1, открытый (общедоступный), и K_2 — закрытый (приватный) ключ, известный только

Алисе. Фактически, Боб должен проверить, что Алиса знает свой закрытый ключ K_2, используя только K_2.

Схема Шнорра — одна из наиболее эффективных среди практических протоколов аутентификации, реализующая данную задачу. Она минимизирует зависимость вычислений, необходимых для создания подписи, от сообщения. В этой схеме основные вычисления могут быть сделаны во время простоя процессора, что позволяет увеличить скорость подписания. Как и DSA, схема Шнорра использует подгруппу порядка q в Z_p^*. Также данный метод использует хеш-функцию $h : \{0,1\}^* \to Z_p$.

Генерация ключей для схемы подписи Шнорра происходит так же, как и генерация ключей для DSA, кроме того, что не существует никаких ограничений по размерам. Заметим также, что модуль p может быть вычислен автономно.

1. Выбирается простое число p , которое по длине обычно равняется 1024 битам.
2. Выбирается другое простое число q таким, чтобы оно было делителем числа $p - 1$. Или, другими словами, должно выполняться $p - 1 \equiv 1 \ (mod\ q)$. Размер для числа q принято выбирать равным 160 битам.
3. Выбирается число g, отличное от 1, такое, что $g^q \equiv 1 \ (mod\ p)$.
4. Пегги выбирает случайное целое число w, меньшее q.
5. Пегги вычисляет $y = g^{q-w} \ (mod\ p)$.
6. Общедоступный ключ Пегги - (p, q, g, y), секретный ключ Пегги - w.

Рассмотрим алгоритм работы протокола:

1. Предварительная обработка. Алиса выбирает случайное число r, меньшее q, и вычисляет $x = g^r \ (mod\ p)$. Эти вычисления являются предварительными и могут быть выполнены задолго до появления Боба.
2. Инициирование. Алиса посылает x Бобу.
3. Боб выбирает случайное число e из диапазона от 0 до $2^t - 1$ и отправляет его Алисе.

4. Алиса вычисляет $s = r + we\ (mod\ q)$ и посылает s Бобу.
5. Подтверждение. Боб проверяет что $x = g^s y^e (mod\ p)$.

Безопасность алгоритма зависит от параметра t. Сложность вскрытия алгоритма примерно равна 2^t. Шнорр советует использовать t около 72 битов, для $p \geq 2^{512}$ и $q \geq 2^{140}$. Для решения задачи дискретного логарифма, в этом случае, требуется по крайней мере 2^{72} шагов известных алгоритмов.

3 Тестирование программы

На рисунках 1-3 представлены результаты работы программы, эмулирующей работу протокола аутентификации Шнорра между двумя субъектами (Alice и Bob).

```
PS F:\all\crypto_protokols\n4> ruby .\go.rb

STEP 0 - Adding & Generation of start params

Enter p bit length
15

Enter q bit length
14

Enter t parameter
13

Alice is generating start params
Generated p: 11827
Generated q: 3
Generated g: 3740
Generated w: 2
Generated y: 3740
Generated r: 3
Generated x: 1

Bob is generating start params
Generated p: 12539
Generated q: 2
Generated g: 12538
Generated w: 0
Generated y: 1
Generated r: 2
Generated x: 1
{:name=>"Alice", :send=>[], :get=>[], :calc=>[], :open_key=>{:p=>11827, :q=>3, :g=>3740, :y=>3740}, :secret_key=>{:w=>2}, :process=>{}, :prep=>{:r=>3, :x=>1}}

{:name=>"Bob", :send=>[], :get=>[], :calc=>[], :open_key=>{:p=>12539, :q=>2, :g=>12538, :y=>1}, :secret_key=>{:w=>0}, :process=>{}, :prep=>{:r=>2, :x=>1}}
```

Рисунок 1 - Результат работы программы

```
STEP 1 - Alice send x value to Bob
{:name=>"Alice",
 :send=>[{:src=>"Alice", :msg=>{:x=>1}}],
 :get=>[],
 :calc=>[],
 :open_key=>{:p=>11827, :q=>3, :g=>3740, :y=>3740},
 :secret_key=>{:w=>2},
 :process=>{},
 :prep=>{:r=>3, :x=>1}}

{:name=>"Bob",
 :send=>[],
 :get=>[{:src=>"Alice", :msg=>{:x=>1}}],
 :calc=>[],
 :open_key=>{:p=>12539, :q=>2, :g=>12538, :y=>1},
 :secret_key=>{:w=>0},
 :process=>{:x=>1},
 :prep=>{:r=>2, :x=>1}}

STEP 2 - Bob generates 0 <= e <= 2^t - 1, sends e to Alice
{:name=>"Alice",
 :send=>[{:src=>"Alice", :msg=>{:x=>1}}],
 :get=>[{:src=>"Bob", :msg=>{:e=>2995}}],
 :calc=>[],
 :open_key=>{:p=>11827, :q=>3, :g=>3740, :y=>3740},
 :secret_key=>{:w=>2},
 :process=>{:e=>2995},
 :prep=>{:r=>3, :x=>1}}

{:name=>"Bob",
 :send=>[{:src=>"Bob", :msg=>{:e=>2995}}],
 :get=>[{:src=>"Alice", :msg=>{:x=>1}}],
 :calc=>[],
 :open_key=>{:p=>12539, :q=>2, :g=>12538, :y=>1},
 :secret_key=>{:w=>0},
 :process=>{:x=>1, :e=>2995},
 :prep=>{:r=>2, :x=>1}}
```

Рисунок 2 - Результат работы программы

Рисунок 3 - Результат работы программы

ПРИЛОЖЕНИЕ А
Код программы go.rb

```ruby
require './steps.rb'

@steps = Steps.new

@steps.step0
@steps.step1
@steps.step2
@steps.step3
@steps.step4
```

ПРИЛОЖЕНИЕ Б
Код программы keygen.rb

```ruby
require 'prime'
require 'openssl'

class Keygen

def initialize

end

def gen_keys
  # Шаг 1: Генерация простого числа p (битовая длина 1024)
  p = Prime.each(2**(1024-1), 2**1024 - 1).first
  puts p
  # Шаг 2: Генерация простого числа q (битовый размер 160 битов)
  q = Prime.each(2**(160-1), 2**160 - 1).first
  puts q
  # Шаг 3: Генерация случайного числа g, такого что g^q = 1 (mod p)
  g = find_random_g(p, q)
  puts g
  # Шаг 4: Генерация случайного числа w < q
  w = SecureRandom.random_number(q)
  puts w
  # Шаг 5: Вычисление y = g^(q-w) (mod p)
  y = g.mod_pow(q - w, p)
  puts y
  # Шаг 6: Сохранение ключей
  open_key = { p: p, q: q, g: g, y: y }
  private_key = { w: w }

  {open_key: open_key, private_key: private_key}
end

private

def find_random_g(p, q)
  loop do
    g = SecureRandom.random_number(p-2) + 2
    return g if g.mod_pow(q, p) == 1
  end
end

end
```

ПРИЛОЖЕНИЕ В
Код программы methods.rb

```ruby
require 'securerandom'
require 'prime'

class Methods

  def initialize(params = {})
    @p_bit_length = params.dig(:p_bit_length) || 1024
    @q_bit_length = params.dig(:q_bit_length) || 160
    @t =            params.dig(:t) || 72

    @debug_mode = params.dig(:debug_mode) == :all
  end

  def make_keys(name = 'Client')
    puts "\n#{name} is generating start params"
    p = gen_large_p
    puts "Generated p: #{p}" if @debug_mode
    q = gen_del_q(p)
    puts "Generated q: #{q}" if @debug_mode
    g = gen_step_g(p, q)
    puts "Generated g: #{g}" if @debug_mode
    w = gen_rand_w(q)
    puts "Generated w: #{w}" if @debug_mode
    y = find_y(g, q, w, p)
    puts "Generated y: #{y}" if @debug_mode

    keys = {
      open_key:{
        p: p,
        q: q,
        g: g,
        y: y
      },
      secret_key:{
        w: w
      },
      process:{
      }
    }

    keys[:prep] = prep_enum(keys)
    puts "Generated r: #{keys[:prep][:r]}" if @debug_mode
    puts "Generated x: #{keys[:prep][:x]}" if @debug_mode

    return keys
  end
```

```ruby
  private

  def gen_large_p
    loop do
      p = rand(2 ** @p_bit_length)
      return p if p.prime?
    end
  end

  def gen_del_q(p)
    loop do
      q = rand(2..Math.sqrt(p-1).to_i)
      return q if q.prime? && (p-1) % q == 0
    end
  end

  def gen_step_g(p, q)
    loop do
      g = rand(p)
      return g if g != 1 && g.pow(q, p) == 1 && g != 0
    end
  end

  def gen_rand_w(q)
    return rand(q)
  end

  def find_y(g, q, w, p)
    return g.pow(q - w, p)
  end

  def gen_rand_r(q)
    return rand(2..q)
  end

  def find_x(g, r, p)
    return g.pow(r, p)
  end

  def prep_enum(keys)
    r = gen_rand_r(keys[:open_key][:q])
    x = find_x(keys[:open_key][:g], r, keys[:open_key][:p])

    {
      r: r,
      x: x
    }
  end
end
```

ПРИЛОЖЕНИЕ Г
Код программы process.rb

```ruby
require './keygen.rb'

@keygen = Keygen.new

puts @keygen.gen_keys
```

ПРИЛОЖЕНИЕ Д
Код программы steps.rb

```ruby
require './methods.rb'
require 'securerandom'

class Steps
  def initialize()
    @alice = nil
    @bob = nil
  end

  def step0
    puts "\nSTEP 0 - Adding & Generation of start params"

    puts "\nEnter p bit length"
    @p_bit_length = gets.strip.to_i

    puts "\nEnter q bit length"
    @q_bit_length = gets.strip.to_i

    puts "\nEnter t parameter"
    @t = gets.strip.to_i

    params = {
      p_bit_length: @p_bit_length,
      q_bit_length: @q_bit_length,
      t: @t,
      debug_mode: :all
    }

    @methods = Methods.new(params)

    @alice = {name: 'Alice', send:[], get:[],
calc:[]}.merge(@methods.make_keys('Alice'))
    @bob = {name:'Bob', send:[], get:[],
calc:[]}.merge(@methods.make_keys('Bob'))

    pp @alice
    puts "\n"
    pp @bob
  end

  def step1
    puts "\nSTEP 1 - Alice send x value to Bob"

    @alice[:send] << make_msg(@alice[:name], {x: @alice[:prep][:x]})
    @bob[:get] << @alice[:send].last
    @bob[:process][:x] = @bob[:get].last[:msg][:x]

    pp @alice
```

```
    puts "\n"
    pp @bob
  end

  def step2
    puts "\nSTEP 2 - Bob generates 0 <= e <= 2^t - 1, sends e to Alice"

    @bob[:process][:e] = gen_e

    @bob[:send] << make_msg(@bob[:name], {e: @bob[:process][:e]})
    @alice[:get] << @bob[:send].last
    @alice[:process][:e] = @alice[:get].last[:msg][:e]

    pp @alice
    puts "\n"
    pp @bob
  end

  def step3
    puts "\nSTEP 3 - Alice calculates s = r + we, sends s to Bob"

    @alice[:process][:s] = find_s(@alice)

    @alice[:send] << make_msg(@alice[:name], {s: @alice[:process][:s]})
    @bob[:get] << @alice[:send].last
    @bob[:process][:s] = @bob[:get].last[:msg][:s]

    pp @alice
    puts "\n"
    pp @bob
  end


  def step4
    puts "\nSTEP 4 - Bob calculates z = g^s * y^e mod p, validates it with
Alice's x"

    @bob[:process][:z] = client_b_check_x(@alice, @bob)

    pp @alice
    puts "\n"
    pp @bob

    puts "\nBob checks owned x and calculated x:"
    puts "\tGot from Alice: #{@bob[:process][:x]}"
    puts "\tCalculated val: #{@bob[:process][:z]}"

    puts "\nResult is:"
    puts (@bob[:process][:x] == @bob[:process][:z] ? "\tALL GOOD" : "\tSOMETHING
GONE WRONG")
  end
```

```ruby
  private

  def make_msg(src, msg)
    {
      # uid:  SecureRandom.uuid,
      # time: Time.now,
      src:  src,
      msg:  msg
    }
  end

  def gen_e
    return rand(2 ** @t)
  end

  def find_s(client)
    return (client[:prep][:r] + client[:secret_key][:w] * client[:process][:e]) %
client[:open_key][:q]
  end

  def client_b_check_x(client_a, client_b)
    (client_a[:open_key][:g].pow(client_b[:process][:s], client_a[:open_key][:p])
*
    client_a[:open_key][:y].pow(client_b[:process][:e], client_a[:open_key][:p]))
% client_a[:open_key][:p]
  end
end
```