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

**Протоколы передачи секретного ключа по открытому каналу**

ОТЧЁТ

ПО ДИСЦИПЛИНЕ

«КРИПТОГРАФИЧЕСКИЕ ПРОТОКОЛЫ»

студента 5 курса 531 группы

специальности 10.05.01 Компьютерная безопасность

факультета компьютерных наук и информационных технологий

Бородина Артёма Горовича

|  |  |  |
| --- | --- | --- |
| Преподаватель  аспирант | \_\_\_\_\_\_\_\_\_\_ | Р. А. Фарахутдинов |
|  | подпись, дата |  |

Саратов 2023

**1 Постановка задачи**

Необходимо реализовать протокол передачи секретного ключа по открытому каналу Encrypted Key Exchange (EKE) на базе алгоритма RSA.

**2 Теоретические сведения**

Протокол обмена зашифрованными ключами (Encrypted Key Exchange, EKE) был разработан Стивом Белловином и Майклом Мерритом. Он обеспечивает безопасность и проверку подлинности в компьютерных сетях, по-новому используя и симметричную криптографию, и криптографию с открытыми ключами: общий секретный ключ используется для шифрования сгенерированного случайным образом открытого ключа.

**Базовый протокол EKE**

Алиса и Боб имеют общий пароль . Используя следующий протокол, они могут проверить подлинность друг друга и сгенерировать общий сеансовый ключ .

1. Алиса случайным образом генерирует пару открытый ключ / закрытый ключ. Она шифрует открытый ключ с помощью симметричного алгоритма, используя в качестве ключа: . Она посылает Бобу:
2. Боб знает Он расшифровывает сообщение, получая . Затем он генерирует случайный сеансовый ключ , шифрует его открытым ключом, который он получил от Алисы, а затем используя в качестве ключа. Он посылает Алисе
3. Алиса расшифровывает сообщение, получая . Она генерирует случайную строку шифрует её с помощью и посылает Бобу:
4. Боб расшифровывает сообщение, получая . Он генерирует другую случайную строку, , шифрует обе строки ключом и посылает Алисе результат:
5. Алиса расшифровывает сообщение, получая и . Если строка , полученная от Боба, – это та самая строка, которую она послала Бобу на этапе (3), она, используя , шифрует и посылает её Бобу.
6. Боб расшифровывает сообщение, получая . Если строка полученная от Алисы, – это та самая строка, которую он послал ей на этапе (4), протокол успешно завершается. Тогда обе стороны могут обмениваться информацией, используя в качестве сеансового ключа. Иначе выполнение протокола не удалось.

EKE может быть реализован множеством алгоритмов с открытыми ключами: RSA, ElGamal, Diffie-Hellman.

**Реализация EKE с помощью RSA**

Алгоритм RSA хорошо подходит для реализации EKE. Авторы рекомендуют шифровать на этапе (1) только показатель степени, посылая модуль в открытую.

**3 Практическая реализация**

**3.1 Описание программы**

Язык программной реализации – Common Lisp. Выполнение программы разбито на шаги и подшаги. Каждый шаг протокола описывается во время его выполнения. Также все генерируемые на текущем шаге значения отображаются в шестнадцатеричной системе счисления. В качестве симметричного алгоритма, использующегося в протоколе, применяется AES-128. Некоторые значения шифртекстов на скриншотах сокращены для удобства представления.

**3.2 Результаты тестирования программы**

Рассмотрим процесс выполнения программы при вводе длины модуля RSA, равной 128.

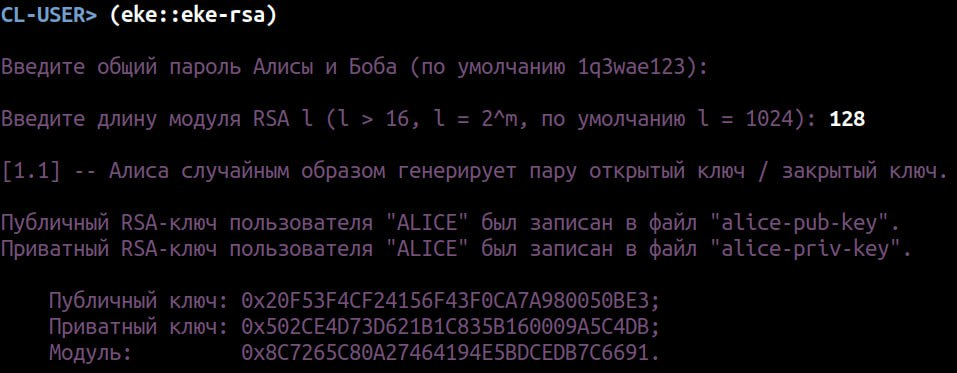


Рисунок 1 – Выполнение подшага 1 шага 1 протокола

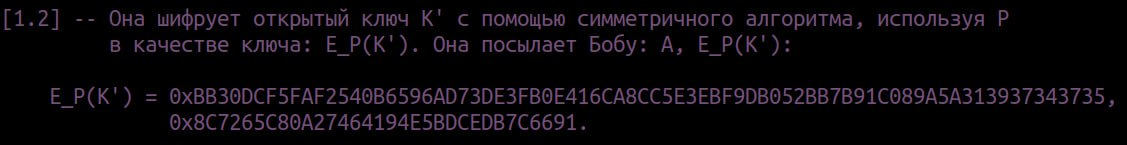


Рисунок 2 – Выполнение подшага 2 шага 1 протокола

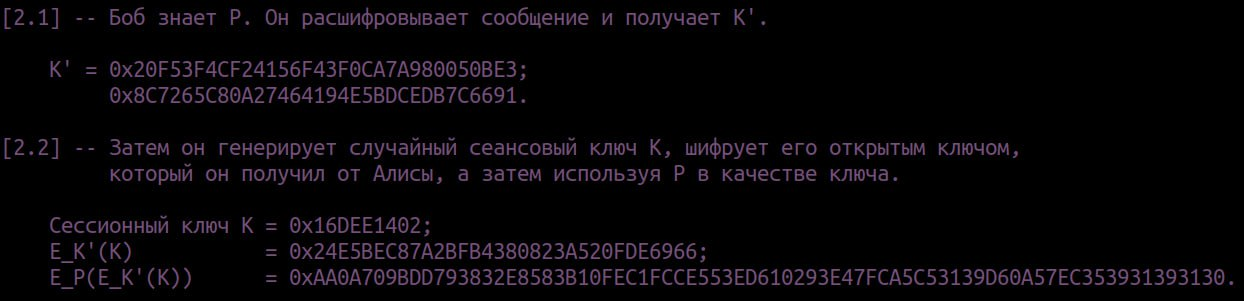


Рисунок 3 – Выполнение подшагов шага 2 протокола

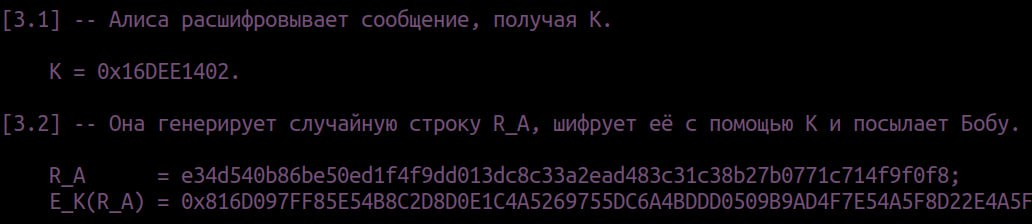


Рисунок 4 – Выполнение подшагов шага 3 протокола

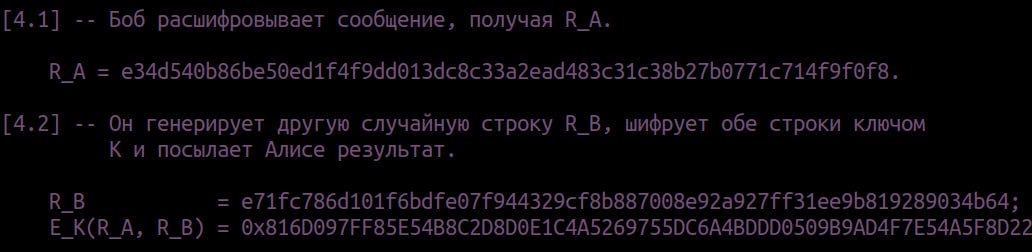


Рисунок 5 – Выполнение подшагов шага 4 протокола

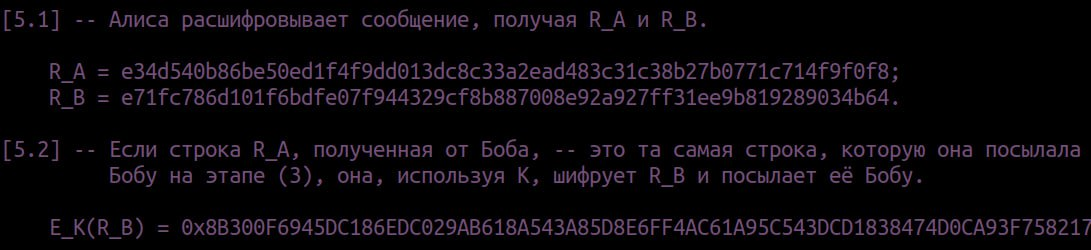


Рисунок 6 – Выполнение подшагов шага 5 протокола

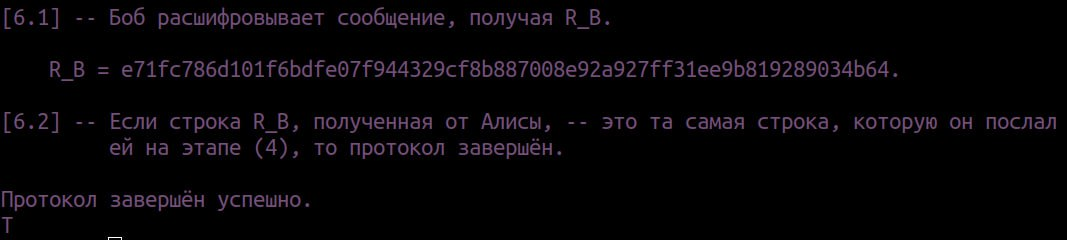


Рисунок 7 – Выполнение подшагов шага 6 протокола

**Листинг программы**

(defpackage :crypt

(:use :common-lisp)

(:export :aes-encrypt :aes-decrypt

:random-string :gen-session-key))

(in-package :crypt)

(defun ripemd128 (str)

(ironclad:byte-array-to-hex-string

(ironclad:digest-sequence

:ripemd-128

(ironclad:ascii-string-to-byte-array str))))

(defun get-cipher (key)

(ironclad:make-cipher :aes

:mode :ecb

:key (ironclad:ascii-string-to-byte-array (ripemd128 key))))

(defun aes-encrypt (plaintext key)

(let ((cipher (get-cipher key))

(msg (ironclad:ascii-string-to-byte-array plaintext)))

(ironclad:encrypt-in-place cipher msg)

(ironclad:octets-to-integer msg)))

(defun aes-decrypt (ciphertext-int key)

(let ((cipher (get-cipher key))

(msg (ironclad:integer-to-octets ciphertext-int)))

(ironclad:decrypt-in-place cipher msg)

(coerce (mapcar #'code-char (coerce msg 'list)) 'string)))

(defun random-string (&optional (len 32))

(ironclad:byte-array-to-hex-string

(ironclad:random-data len)))

(defun gen-session-key (len-mod)

(reduce #'+ (mapcar #'\* (loop for digit from 1 to (floor len-mod 4)

collect (1+ (random 9)))

(loop for pow from 1 to (floor len-mod 4)

collect (expt 10 pow)))))

(defpackage #:aux

(:use :cl))

(in-package #:aux)

(defmacro while (condition &body body)

`(loop while ,condition

do (progn ,@body)))

(defun write-to-file (data filename)

(with-open-file (out filename :direction :output :if-exists :supersede

:if-does-not-exist :create)

(dolist (param data)

(format out "~a~%" param))))

(defun n-elts (elt n)

(if (> n 1)

(list n elt)

elt))

(defun compr (elt n lst)

(if (null lst)

(list (n-elts elt n))

(let ((next (car lst)))

(if (eql next elt)

(compr elt (1+ n) (cdr lst))

(cons (n-elts elt n) (compr next 1 (cdr lst)))))))

(defun compress (x)

(if (consp x)

(compr (car x) 1 (cdr x))

x))

(defun is-pow-of-2? (num)

(zerop (logand num (1- num))))

(defun mod-expt (base power modulo)

(setq base (mod base modulo))

(do ((product 1)) ((zerop power) product)

(do () ((oddp power))

(setq base (mod (\* base base) modulo)

power (ash power -1)))

(setq product (mod (\* product base) modulo)

power (1- power))))

(defun miller-rabin (n &optional (k 10))

(when (or (= 2 n) (= 3 n)) (return-from miller-rabin t))

(when (or (< n 2) (= 0 (logand n 1))) (return-from miller-rabin))

(let\* ((n-pred (1- n)) (bound (- n-pred 2)) (t-val n-pred) (s 0) (round 0) (x))

(while (= 0 (logand t-val 1)) (setq s (1+ s) t-val (ash t-val -1)))

(do () (nil)

(tagbody next-iteration

(when (= k round) (return-from miller-rabin t))

(setq x (mod-expt (+ 2 (random bound)) t-val n))

(when (or (= 1 x) (= n-pred x))

(incf round) (go next-iteration))

(do ((iter 0 (1+ iter))) ((= iter (1- s)) (return-from miller-rabin))

(setq x (mod (\* x x) n))

(when (= 1 x) (return-from miller-rabin))

(when (= n-pred x)

(incf round) (go next-iteration)))))))

(defparameter \*base-primes\*

(remove-if-not #'(lambda (prime?) (miller-rabin prime? 12))

(loop for prime? from (1+ (ash 1 15)) to (1- (ash 1 16)) by 2

collect prime?)))

(defun ext-gcd (a b)

(let ((s 0) (old-s 1) (r b) (old-r a)

(quotient) (bezout-t))

(while (not (zerop r))

(setq quotient (floor old-r r))

(psetq old-r r r (- old-r (\* quotient r))

old-s s s (- old-s (\* quotient s))))

(if (zerop b) (setq bezout-t 0)

(setq bezout-t (floor (- old-r (\* old-s a)) b)))

(list old-r old-s bezout-t)))

(defun rho-pollard-machinerie (n x-0 &optional (c 1) (rounds 1000))

(when (miller-rabin n) (return-from rho-pollard-machinerie 'PRIME))

(let ((mapping (lambda (x) (mod (+ c (\* x x)) n)))

(a x-0) (b x-0) (round 0) (q))

(tagbody map

(incf round)

(when (> round rounds) (return-from rho-pollard-machinerie 'GEN-NEW))

(setq a (funcall mapping a)

b (funcall mapping (funcall mapping b))

q (gcd (- a b) n))

(cond ((< 1 q n) (return-from rho-pollard-machinerie

(list q (miller-rabin q))))

((= n q) (return-from rho-pollard-machinerie))

(t (go map))))))

(defun rho-pollard-wrapper (n x-0)

(let ((c 1) (head) (factor) (factors))

(while (zerop (logand n 1))

(setq factors (cons 2 factors) n (ash n -1)))

(setq x-0 (mod x-0 n))

(while (/= 1 n)

(setq factor (rho-pollard-machinerie n x-0 c))

(cond ((eql 'PRIME factor) (setq factors (cons n factors) n 1))

((eql 'GEN-NEW factor) (return))

((cadr factor) (setq factors (cons (setq head (car factor)) factors)

n (/ n head)))

((null factor) (while (= (- n 2) (setq c (1+ (random (1- n)))))))

(t (setq n (/ n (setq head (car factor)))

factors (append factors

(rho-pollard-wrapper head (random head)))))))

factors))

(defun rho-pollard (n x-0)

(let\* ((factors (rho-pollard-wrapper n x-0)))

(when (null factors) (return-from rho-pollard))

(when (= n (apply #'\* factors))

(compress (sort (rho-pollard-wrapper n x-0) #'<)))))

(defun get-bit-len ()

(let ((bit-len))

(tagbody try-again

(setq bit-len (read))

(when (and (integerp bit-len) (is-pow-of-2? bit-len) (> bit-len 16))

(return-from get-bit-len bit-len))

(format t "Некорректный ввод! Попробуйте снова: ")

(go try-again))))

(defun find-g (p)

(when (not (miller-rabin p)) (return-from find-g))

(let ((phi (1- p)) (factors) (g?) (bound (- p 2)))

(setq factors (rho-pollard phi (random p)))

(when (null factors) (return-from find-g))

(setq factors (mapcar #'(lambda (factor) (cond ((atom factor) factor)

(t (cadr factor)))) factors)

factors (mapcar #'(lambda (factor) (floor phi factor)) factors))

(tagbody try-again

(setq g? (+ 2 (random bound)))

(when (= 1 (mod-expt g? (car factors) p)) (go try-again))

(when (remove-if-not #'(lambda (pow) (= 1 (mod-expt g? pow p)))

factors) (go try-again))) g?))

(defun generate-even (target-len)

(apply #'+ (ash 1 (1- target-len))

(mapcar #'(lambda (bit pow) (\* bit (ash 1 pow)))

(append (loop for bit from 0 to (- target-len 3)

collect (random 2)) '(0))

(loop for pow from (- target-len 2) downto 0 collect pow))))

(defun generate-prime (target-len)

(when (not (is-pow-of-2? target-len))

(return-from generate-prime))

(when (= 16 target-len)

(return-from generate-prime (nth (random (length \*base-primes\*))

\*base-primes\*)))

(let ((prime) (s) (prime?) (req-len (- target-len 16)))

(tagbody pick-prime

(setq prime (nth (random (length \*base-primes\*)) \*base-primes\*))

(when (not (miller-rabin prime)) (go pick-prime)))

(tagbody try-again

(setq s (generate-even req-len)

prime? (1+ (\* prime s)))

(if (and (= 1 (mod-expt 2 (1- prime?) prime?))

(/= 1 (mod-expt 2 s prime?))

(zerop (logxor (length (write-to-string prime? :base 2))

target-len)))

(return-from generate-prime prime?)

(go try-again)))))

(defun gen-p&g (bit-len)

(let ((p) (g))

(tagbody gen-prime

(setq p (generate-prime bit-len)

g (find-g p))

(when (null g) (go gen-prime)))

(list p g)))

(defpackage #:rsa

(:use :cl))

(in-package #:rsa)

(defun rsa-machinerie (key-length &optional (num-users 1))

(when (not (aux::is-pow-of-2? key-length))

(return-from rsa-machinerie))

(let ((p) (q) (n) (phi) (e) (d) (res))

(tagbody try-again

(setq p (aux::generate-prime (ash key-length -1))

q (aux::generate-prime (ash key-length -1))

n (\* p q))

(when (not (zerop (logxor (length (write-to-string n :base 2))

key-length)))

(go try-again)))

(do ((i 0 (1+ i))) ((= num-users i) res)

(setq phi (\* (1- p) (1- q)))

(aux::while (/= 1 (gcd (setq e (random phi)) phi)))

(setq d (mod (cadr (aux::ext-gcd e phi)) phi)

res (cons (list e d n) res)))))

(defun rsa-generate-keys (key-length prefixes)

(let ((keys (rsa-machinerie key-length (length prefixes)))

(prefix) (name) (pub-filename) (priv-filename))

(do ((i 0 (1+ i))) ((= (length prefixes) i))

(destructuring-bind (e d n) (nth i keys)

(setq prefix (nth i prefixes)

name (string-upcase (nth i prefixes))

pub-filename (concatenate 'string prefix "-pub-key")

priv-filename (concatenate 'string prefix "-priv-key"))

(aux::write-to-file (list name e n) pub-filename)

(format t "Публичный RSA-ключ пользователя ~s был записан в файл ~s.~%"

name pub-filename)

(aux::write-to-file (list name d n) priv-filename)

(format t "Приватный RSA-ключ пользователя ~s был записан в файл ~s.~%"

name priv-filename)

(format t "~% Публичный ключ: 0x~x;

Приватный ключ: 0x~x;

Модуль: 0x~x.~%" e d n) (terpri)))))

(defpackage :eke

(:use :common-lisp))

(in-package :eke)

(defun stop () (read-line))

(defun read-parse (filename &optional (at 0))

(parse-integer (uiop:read-file-line filename :at at)))

(defun step-1-aux ()

(let ((key-len))

(format t "~%Введите длину модуля RSA l (l > 16, l = 2^m, по умолчанию l = 1024): ")

(tagbody try-again

(setq key-len (parse-integer (read-line) :junk-allowed t))

(when (not (integerp key-len))

(setq key-len 1024))

(when (or (null key-len) (not (and (zerop (logand key-len (1- key-len)))

(> key-len 16))))

(format t "~%Некорректное значение l! Введите l снова: ")

(go try-again))) key-len))

(defun step-1 ()

(let ((key-len (step-1-aux)) (shared-password) (encrypted) (pub-key))

(format t "~%[1.1] -- Алиса случайным образом генерирует пару открытый ключ / закрытый ключ.~2%")

(rsa::rsa-generate-keys key-len '("alice"))

(setq pub-key (cdr (uiop:read-file-lines "alice-pub-key"))

shared-password (uiop:read-file-line "shared-password"))

(format t "[1.2] -- Она шифрует открытый ключ K' с помощью симметричного алгоритма, используя P

в качестве ключа: E\_P(K'). Она посылает Бобу: A, E\_P(K'): ")

(setq encrypted (cons (crypt:aes-encrypt (car pub-key) shared-password)

(cdr pub-key)))

(aux::write-to-file encrypted "step-1-message")

(format t "~2%~4tE\_P(K') = 0x~x,

0x~x.~%" (car encrypted) (parse-integer (cadr encrypted)))))

(defun step-2 ()

(let ((shared-password) (decrypted) (modulo) (session-key) (encrypted))

(format t "~%[2.1] -- Боб знает P. Он расшифровывает сообщение и получает K'.")

(setq shared-password (uiop:read-file-line "shared-password")

decrypted (crypt:aes-decrypt (read-parse "step-1-message")

shared-password)

modulo (uiop:read-file-line "step-1-message" :at 1))

(format t "~2%~4tK' = 0x~x;

0x~x." (setq decrypted (parse-integer decrypted)) (parse-integer modulo))

(format t "~2%[2.2] -- Затем он генерирует случайный сеансовый ключ K, шифрует его открытым ключом,

который он получил от Алисы, а затем используя P в качестве ключа.")

(setq session-key (crypt:gen-session-key (length modulo))

encrypted (aux::mod-expt session-key decrypted (parse-integer modulo)))

(aux::write-to-file (list session-key) "session-key")

(format t "~2%~4tСессионный ключ K = 0x~x;

~4tE\_K'(K) = 0x~x;" session-key encrypted)

(setq encrypted (crypt:aes-encrypt (write-to-string encrypted) shared-password))

(format t "~%~4tE\_P(E\_K'(K)) = 0x~x.~%" encrypted)

(aux::write-to-file (list encrypted) "step-2-message")))

(defun step-3 ()

(let ((shared-password) (priv-key) (modulo)

(decrypted) (random-string) (encrypted))

(format t "~%[3.1] -- Алиса расшифровывает сообщение, получая K.")

(setq shared-password (uiop:read-file-line "shared-password")

priv-key (read-parse "alice-priv-key" 1)

modulo (read-parse "alice-priv-key" 2)

decrypted (aux::mod-expt

(parse-integer (crypt:aes-decrypt (read-parse "step-2-message")

shared-password))

priv-key modulo))

(format t "~2%~4tK = 0x~x." decrypted)

(setq random-string (crypt:random-string))

(aux::write-to-file (list random-string) "alice-random-string")

(setq encrypted (crypt:aes-encrypt random-string (write-to-string decrypted)))

(format t "~2%[3.2] -- Она генерирует случайную строку R\_A, шифрует её с помощью K и посылает Бобу.

~%~4tR\_A = ~a;~%~4tE\_K(R\_A) = 0x~x.~%" random-string encrypted)

(aux::write-to-file (list encrypted) "step-3-message")))

(defun step-4 ()

(let ((session-key) (decrypted) (random-string) (encrypted))

(format t "~%[4.1] -- Боб расшифровывает сообщение, получая R\_A.")

(setq session-key (uiop:read-file-line "session-key")

decrypted (read-parse "step-3-message")

decrypted (crypt:aes-decrypt decrypted session-key))

(format t "~2%~4tR\_A = ~a." decrypted)

(setq random-string (crypt:random-string)

encrypted (crypt:aes-encrypt (concatenate 'string decrypted " " random-string)

session-key))

(aux::write-to-file (list random-string) "bob-random-string")

(aux::write-to-file (list encrypted) "step-4-message")

(format t "~2%[4.2] -- Он генерирует другую случайную строку R\_B, шифрует обе строки ключом

K и посылает Алисе результат.~2%~4tR\_B = ~a;~%~4tE\_K(R\_A, R\_B) = 0x~x.~%"

random-string encrypted)))

(defun step-5 ()

(let ((session-key) (alice-str) (decrypted) (encrypted))

(format t "~%[5.1] -- Алиса расшифровывает сообщение, получая R\_A и R\_B.")

(setq session-key (uiop:read-file-line "session-key")

alice-str (uiop:read-file-line "alice-random-string")

decrypted (uiop:split-string (crypt:aes-decrypt (read-parse "step-4-message")

session-key)

:separator " "))

(when (not (equal alice-str (car decrypted)))

(format t "~2%Строка R\_A, полученная от Боба, не совпадает со строкой, посланной на 3-м этапе! Экстренное завершение протокола.~%")

(return-from step-5))

(format t "~2%~4tR\_A = ~a;~%~4tR\_B = ~a." (car decrypted) (cadr decrypted))

(format t "~2%[5.2] -- Если строка R\_A, полученная от Боба, -- это та самая строка, которую она посылала

Бобу на этапе (3), она, используя K, шифрует R\_B и посылает её Бобу.")

(setq encrypted (crypt:aes-encrypt (cadr decrypted) session-key))

(aux::write-to-file (list encrypted) "step-5-message")

(format t "~2%~4tE\_K(R\_B) = 0x~x.~%" encrypted) t))

(defun step-6 ()

(let ((session-key) (bob-str) (decrypted))

(format t "~%[6.1] -- Боб расшифровывает сообщение, получая R\_B.")

(setq session-key (uiop:read-file-line "session-key")

bob-str (uiop:read-file-line "bob-random-string")

decrypted (crypt:aes-decrypt (read-parse "step-5-message") session-key))

(when (not (equal bob-str decrypted))

(format t "~2%Строка R\_B, полученная от Алисы, не совпадает со строкой, посланной на 4-ом этапе! Экстренное завершение протокола.~%")

(return-from step-6))

(format t "~2%~4tR\_B = ~a." decrypted)

(format t "~2%[6.2] -- Если строка R\_B, полученная от Алисы, -- это та самая строка, которую он послал

ей на этапе (4), то протокол завершён. ") t))

(defun eke-rsa ()

(let ((shared-password))

(format t "~%Введите общий пароль Алисы и Боба (по умолчанию 1q3wae123): ")

(setq shared-password (read-line))

(when (zerop (length shared-password))

(setq shared-password "1q3wae123"))

(aux::write-to-file (list shared-password) "shared-password"))

(step-1) (stop) (step-2) (stop) (step-3) (stop)

(step-4) (stop)

(when (not (step-5)) (return-from eke-rsa)) (stop)

(when (not (step-6)) (return-from eke-rsa))

(format t "~2%Протокол завершён успешно.") t)