ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ   
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«СИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ТЕЛЕКОММУНИКАЦИЙ И ИНФОРМАТИКИ»

**Расчётно-графическая работа**

по дисциплине “Защита информации”

на тему

**Доказательство с нулевым знанием**

Вариант №3

|  |  |
| --- | --- |
| Выполнил студент | Дьяченко Даниил Вадимович |
|  | Ф.И.О. |

|  |  |
| --- | --- |
| Группы | ИВ-621 |
|  |  |

|  |  |  |
| --- | --- | --- |
| Работу приняла |  | ассистент кафедры ПМиК Я.В. Петухова |
|  | подпись |  |

|  |  |  |  |
| --- | --- | --- | --- |
| Защищена |  | Оценка |  |
|  |  |  |  |

Новосибирск – 2019 г.

Оглавление

[1 Постановка задачи 3](#_Toc27002262)

[2 Теоретические сведения 4](#_Toc27002263)

[ПРИЛОЖЕНИЕ 9](#_Toc27002264)

[1. Исходный код 9](#_Toc27002265)

1. Постановка задачи

В рамках расчётно-графического задания необходимо написать прог-рамму, реализующую протокол доказательства с нулевым знанием Фиата-Шамира.

При установлении подлинности пароля Алиса должна передать свой секрет (пароль) верификатору; это может привести к перехвату информации Евой. Кроме того, нечестный верификатор может показать пароль другим или использовать его, чтобы исполнить роль претендента.

При установлении подлинности объекта методом вызова-ответа секрет претендента не передают верификатору. Претендент применяет некоторую функцию для обработки вызова, которая передана верификатором, но при этом включает свой секрет. В некоторых методах "вызова-ответа" верификатор фактически знает секрет претендента, при этом он может неправильно использоваться нечестной верификацией. В других методах верификатор может извлечь некоторую информацию о секрете претендент а, выбирая заранее запланированное множество вызовов.

В установлении подлинности с нулевым разглашением претендент не раскрывает ничего, что могло бы создать угрозу конфиденциальности секрета. Претендент доказывает верификатору, что он знает секрет, не раскрывая и не показывая его. В таком случае взаимодействие разработано так, чтобы не привести к раскрытию или предположению о содержании секрета. После обмена сообщениями верификатор только знает, что претендент имеет или не имеет секрета - и ничего больше. В этой ситуации результат - да/нет. Это единственный бит информации

1. Теоретические сведения

Протокол Фиата — Шамира — это один из наиболее известных протоколов идентификации с нулевым разглашением (Zero-knowledge protocol). Протокол был предложен Амосом Фиатом (англ. Amos Fiat) и Ади Шамиром (англ. Adi Shamir)

Пусть А знает некоторый секрет s. Необходимо доказать знание этого секрета некоторой стороне В без разглашения какой-либо секретной информации. Стойкость протокола основывается на сложности извлечения квадратного корня по модулю достаточно большого составного числа n, факторизация которого неизвестна.

A доказывает B знание s в течение t раундов. Раунд называют также аккредитацией. Каждая аккредитация состоит из 3х этапов.

**Предварительные действия:**

* Доверенный центр Т выбирает и публикует модуль , где , — простые и держатся в секрете
* Каждый претендент A выбирает взаимно-простое с , где . Затем вычисляется . регистрируется Т в качестве открытого ключа А

**Передаваемые сообщения (этапы каждой аккредитации):**

**Основные действия:**

Следующие действия последовательно и независимо выполняются t раз. В считает знание доказанным, если все t раундов прошли успешно.

* А выбирает случайное число , такое, что и отсылает стороне B (доказательство)
* B случайно выбирает бит ( или ) и отсылает его A (вызов)
* А вычисляет и отправляет его обратно к B. Если , то , иначе (ответ)
* Если , то B отвергает доказательство или, другими словами, А не удалось доказать знание . В противном случае, сторона B проверяет, действительно ли и, если это так, то происходит переход к следующему раунду протокола

Выбор е из множества предполагает, что если сторона А действительно знает секрет, то она всегда сможет правильно ответить, вне зависимости от выбранного . Допустим, что А хочет обмануть B. В этом случае А, может отреагировать только на конкретное значение . Например, если А знает, что получит , то А следует действовать строго по инструкции и В примет ответ. В случае, если А знает, что получит , то А выбирает случайное и отсылает на сторону В, в результате получаем нам нужное . Проблема заключается в том, что А изначально не знает какое он получит и поэтому не может со 100 % вероятностью выслать на сторону В нужные для обмана и ( при и при ). Поэтому вероятность обмана в одном раунде составляет 50 %. Чтобы снизить вероятность жульничества (она равна )) выбирают достаточно большим (). Таким образом, B удостоверяется в знании А тогда и только тогда, когда все раундов прошли успешно.

Пример работы программы

Вывод логов сервера за одну сессию при подключении клиента:

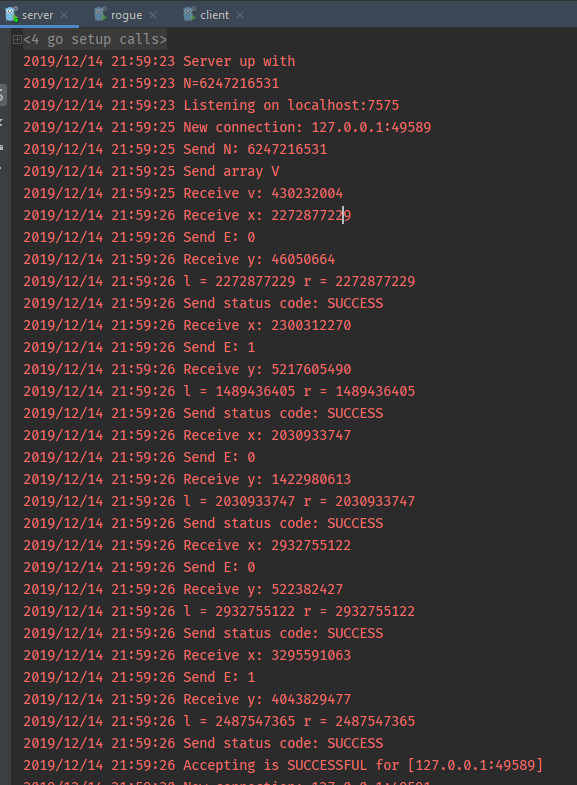


Рисунок 1 - Пример вывода программы

В логах можно увидеть как создается открытый ключ N, создается соединение с пользователем и начинается сессия из пяти раундом путем передачи открытых ключей N и V, приеме пользовательских открытых ключей V и X, передачи случайного Е, приеме сгенерированного пользователем Y, проверке правильности принятого Y и отправка кода статуса ответа (либо SUCCESSFUL, либо ERROR)

Дальше вывод логов пользователя при той же сессии, что была показана выше:

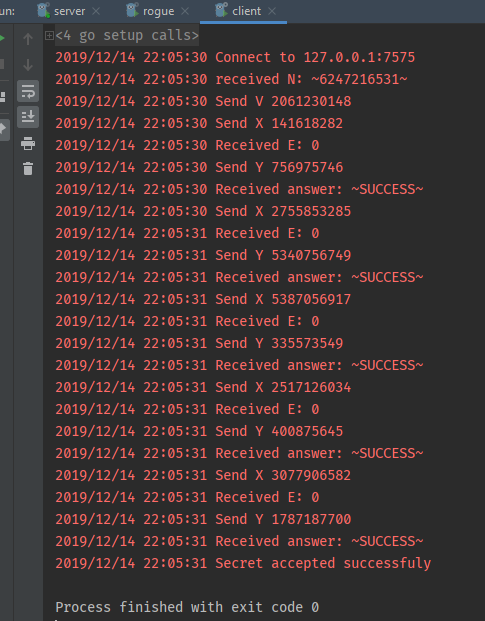
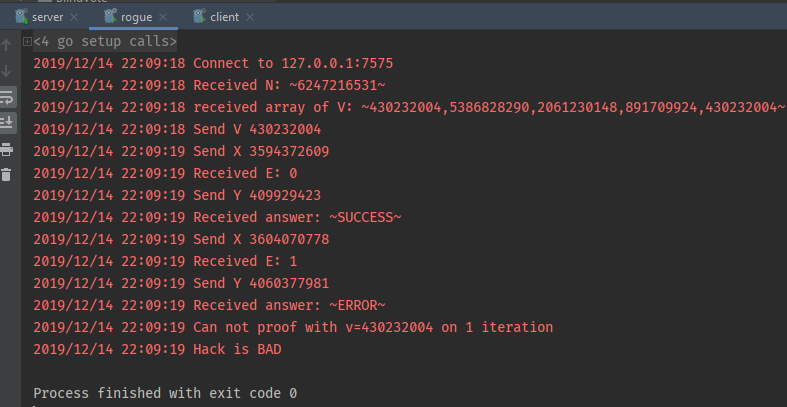


Рисунок 2 - Пример логов пользователя

В логах пользователя можно увидеть соединение с сервером, приеме открытых ключей N и V, передаче сгенерированного открытого ключа V, на основе закрытого ключа, который не показан в логах по понятным причинам, отправке открытого ключа X, сгенерированного на основе случайного большого числа R, приеме параметра E, отправке ответного ключа Y и получение кода статуса ответа (в данном случае это SUCCESSFUL). В случае, если хотя бы в одном из раундом сервер ответит кодом ERROR, авторизация не проходит и клиент отключается.

Дальше вывод логов мошенника при той же сессии, что была показана выше:

Рисунок 3 – Пример логов мошенника

В данных логах можно увидеть, как мошенник подключается к серверу, получает открытые ключи N и V, получает параметр Е и пытается подобрать ключи Y и X. Как видно при Е равным 0 у мошенника получается обмануть сервер, так как по алгоритму Фиата-Шамира в данном случае клиенту необходимо передать в ответе ключ , следовательно, так как ключ V известен мошеннику, то и обмануть сервер получается со 100% вероятностью. Но в следующем раунде мошеннику не везет и в качестве Е выпадет 1, и в этот раз сервер уже не удается обмануть.

ПРИЛОЖЕНИЕ

1. Исходный код
2. client/main.go
4. package main
5. import "cryptocrouse/src/go/FiatShamirProtocol/client/clnt"
6. func main() {
7. client := clnt.Client{}
8. client.ConnectToServer()
9. client.StartProof()
10. }
12. client/client.go
14. type Client struct {
15. conn net.Conn
16. reader \*bufio.Reader
17. writer \*bufio.Writer
18. data \*ClientData
19. }
20. type ClientData struct {
21. S \*big.Int
22. V \*big.Int
23. N \*big.Int
24. E int
25. Y \*big.Int
26. R \*big.Int
27. X \*big.Int
28. arrayV []\*big.Int
29. }
30. func (c \*Client) ConnectToServer() {
31. arguments := os.Args
32. if len(arguments) == 1 {
33. fmt.Println("Please provide host:port.")
34. return
35. }
36. connect := arguments[1]
37. conn, err := net.Dial("tcp", connect)
38. if err != nil {
39. fmt.Println(err)
40. return
41. }
42. log.Printf("Connect to %s\n", connect)
43. c.conn = conn
44. c.data = &ClientData{}
45. c.setupConnections()
46. }
47. func (c\* Client) setupConnections() {
48. c.reader = bufio.NewReader(c.conn)
49. c.writer = bufio.NewWriter(c.conn)
50. }
51. func (c \*Client) StartProof() {
52. c.receiveN()
53. c.receiveV()
54. c.generateS()
55. c.computeV()
56. c.sendV()
57. for i := 0; i < 5; i++ {
58. answerCode := c.round()
59. if answerCode == false {
60. log.Fatalf("Can not proof on %d iteration\n", i)
61. return
62. }
63. }
64. log.Printf("Secret accepted successfuly")
65. c.sendEnd()
66. }
67. func (c \*Client) round() bool {
68. c.generateR()
69. c.computeX()
70. c.sendX()
71. c.receiveE()
72. c.computeY()
73. c.sendY()
74. return c.getAnswer()
75. }
76. func (c \*Client) receiveN() {
77. time.Sleep(50 \* time.Millisecond)
78. msg, err := c.reader.ReadString('\n')
79. if err != nil {
80. log.Fatal(err)
81. }
82. msg = strings.TrimSuffix(msg, "\n")
83. log.Printf("received N: ~%s~\n", msg)
84. var flag bool
85. c.data.N, flag = big.NewInt(0).SetString(msg, 10)
86. if flag == false {
87. log.Fatal("Received N is bad")
88. }
89. }
90. func (c \*Client) generateS() {
91. for {
92. c.data.S = Fingerprints.GetBigRandomWithLimit(c.data.N)
93. if c.data.S.Cmp(big.NewInt(1)) == 0 {
94. continue
95. }
96. GCD := big.NewInt(0).GCD(
97. nil,
98. nil,
99. c.data.S,
100. c.data.N)
101. if GCD.Cmp(big.NewInt(1)) == 0 {
102. break
103. }
104. }
105. }
106. func (c \*Client) computeV() {
107. c.data.V = big.NewInt(0).Exp(c.data.S, big.NewInt(2), c.data.N)
108. }
109. func (c \*Client) receiveE() {
110. time.Sleep(50 \* time.Millisecond)
111. msg, \_ := c.reader.ReadString('\n')
112. msg = strings.TrimSuffix(msg, "\n")
113. log.Printf("Received E: %s\n", msg)
114. c.data.E, \_ = strconv.Atoi(msg)
115. }
116. func (c \*Client) computeY() {
117. switch c.data.E {
118. case 0:
119. c.data.Y = big.NewInt(0).Mod(c.data.R, c.data.N)
120. case 1:
121. c.data.Y = big.NewInt(0).Mod(
122. big.NewInt(0).Mul(
123. c.data.S,
124. c.data.R),
125. c.data.N)
126. }
127. }
128. func (c \*Client) generateR() {
129. for {
130. c.data.R = Fingerprints.GetBigRandomWithLimit(c.data.N)
131. if c.data.R.Cmp(big.NewInt(1)) > 0 && c.data.R.Cmp(c.data.N) < 0 {
132. break
133. }
134. }
135. }
136. func (c \*Client) computeX() {
137. c.data.X = big.NewInt(0).Exp(c.data.R, big.NewInt(2), c.data.N)
138. }
139. func (c \*Client) sendX() {
140. time.Sleep(50 \* time.Millisecond)
141. \_, \_ = c.writer.WriteString(c.data.X.Text(10) + "\n")
142. \_ = c.writer.Flush()
143. log.Printf("Send X %s\n", c.data.X.Text(10))
144. }
145. func (c \*Client) sendY() {
146. time.Sleep(50 \* time.Millisecond)
147. \_, \_ = c.writer.WriteString(c.data.Y.Text(10) + "\n")
148. \_ = c.writer.Flush()
149. log.Printf("Send Y %s\n", c.data.Y.Text(10))
150. }
151. func (c \*Client) sendV() {
152. time.Sleep(50 \* time.Millisecond)
153. \_, \_ = c.writer.WriteString(c.data.V.Text(10) + "\n")
154. \_ = c.writer.Flush()
155. log.Printf("Send V %s\n", c.data.V.Text(10))
156. }
157. func (c \*Client) getAnswer() bool {
158. msg, err := c.reader.ReadString('\n')
159. if err != nil {
160. log.Fatal(err)
161. }
162. msg = strings.TrimSuffix(msg, "\n")
163. log.Printf("Received answer: ~%s~\n", msg)
164. switch msg {
165. case FiatShamirProtocol.COMMAND\_ANSWER\_CODE\_SUCCESS:
166. return true
167. case FiatShamirProtocol.COMMAND\_ANSWER\_CODE\_ERROR:
168. return false
169. default:
170. return false
171. }
172. }
173. func (c \*Client) sendEnd() {
174. \_, \_ = c.writer.WriteString(FiatShamirProtocol.COMMAND\_END + "\n")
175. \_ = c.writer.Flush()
176. }
177. func (c \*Client) receiveV() {
178. time.Sleep(50 \* time.Millisecond)
179. \_, err := c.reader.ReadString('\n')
180. if err != nil {
181. log.Fatal(err)
182. }
183. }
184. server/main.go
186. package main
187. import "cryptocrouse/src/go/FiatShamirProtocol/server/srvr"
188. func main() {
189. server := srvr.ServerInit()
190. server.Run()
191. }
193. server/server.go
195. package srvr
196. const (
197. CONN\_HOST = "localhost"
198. CONN\_PORT = "7575"
199. CONN\_TYPE = "tcp"
200. )
201. var (
202. MIN\_P = big.NewInt(0).Exp(big.NewInt(2), big.NewInt(16), nil)
203. MAX\_P = big.NewInt(0).Exp(big.NewInt(2), big.NewInt(32), nil)
204. )
206. import (
207. "bufio"
208. "cryptocrouse/src/go/FiatShamirProtocol"
209. "cryptocrouse/src/go/Fingerprints"
210. "log"
211. "math/big"
212. "net"
213. "strconv"
214. "strings"
215. "time"
216. )
218. type Server struct {
219. data ServerData
220. }
221. type ServerData struct {
222. p \*big.Int
223. q \*big.Int
224. N \*big.Int
225. V []\*big.Int
226. }
227. func ServerInit() \*Server {
228. data := ServerData{
229. p: big.NewInt(0),
230. q: big.NewInt(0),
231. N: big.NewInt(0),
232. V: make([]\*big.Int, 0),
233. }
234. return &Server{
235. data: data,
236. }
237. }
238. func (s \*Server) Run() {
239. s.serverPrepare()
240. log.Printf("Server up with\n")
241. log.Printf("N=%s\n", s.data.N.Text(10))
242. s.serverListen()
243. }
244. func (s \*Server) serverPrepare() {
245. s.data.generateP()
246. s.data.computeN()
247. }
248. func (s \*Server) serverListen() {
249. l, err := net.Listen(CONN\_TYPE, CONN\_HOST+ ":" +CONN\_PORT)
250. if err != nil {
251. log.Fatalf("Error listening:", err.Error())
252. }
253. defer l.Close()
254. log.Println("Listening on " + CONN\_HOST + ":" + CONN\_PORT)
255. for {
256. conn, err := l.Accept()
257. if err != nil {
258. log.Fatalf("Error accepting: %s\n", err.Error())
259. }
260. log.Printf("New connection: %s\n", conn.RemoteAddr())
261. go s.startRound(conn)
262. }
263. }
264. func (s \*Server) startRound(conn net.Conn) {
265. r := bufio.NewReader(conn)
266. w := bufio.NewWriter(conn)
267. var x \*big.Int
268. var y \*big.Int
269. s.sendN(w)
270. s.sendV(w)
271. v := s.receiveV(r)
272. if v != nil {
273. s.data.V = append(s.data.V, v)
274. }
275. for t := 0; t < 5; t++ {
276. x = s.receiveX(r)
277. e := generateE()
278. s.sendE(w, e)
279. y = s.receiveY(r)
280. if x == nil || v == nil || y == nil {
281. return
282. }
283. statusCode := s.computeY(y, x, v, w, e)
284. s.sendAnswerCode(w, statusCode)
285. if statusCode == FiatShamirProtocol.COMMAND\_ANSWER\_CODE\_ERROR {
286. log.Printf("Accepting is BAD for [%d]\n", conn.RemoteAddr())
287. return
288. }
289. }
290. log.Printf("Accepting is SUCCESSFUL for [%v]\n", conn.RemoteAddr())
291. }
292. func (s \*Server) sendN(w \*bufio.Writer) {
293. log.Printf("Send N: %s\n", s.data.N.Text(10))
294. \_, \_ = w.WriteString(s.data.N.Text(10) + "\n")
295. \_ = w.Flush()
296. }
297. func (s \*Server) receiveX(r \*bufio.Reader) \*big.Int {
298. msg, \_ := r.ReadString('\n')
299. msg = strings.TrimSuffix(msg, "\n")
300. x, \_ := big.NewInt(0).SetString(msg, 10)
301. log.Printf("Receive x: %s\n", msg)
302. return x
303. }
304. func (s \*Server) receiveV(r \*bufio.Reader) \*big.Int {
305. msg, \_ := r.ReadString('\n')
306. msg = strings.TrimSuffix(msg, "\n")
307. v, \_ := big.NewInt(0).SetString(msg, 10)
308. log.Printf("Receive v: %s\n", msg)
309. return v
310. }
311. func (s \*Server) sendE(w \*bufio.Writer, e int) {
312. \_, \_ = w.WriteString(strconv.Itoa(e) + "\n")
313. \_ = w.Flush()
314. log.Println("Send E: " + strconv.Itoa(e))
315. }
316. func (s \*Server) receiveY(r \*bufio.Reader) \*big.Int {
317. msg, \_ := r.ReadString('\n')
318. msg = strings.TrimSuffix(msg, "\n")
319. y, \_ := big.NewInt(0).SetString(msg, 10)
320. log.Printf("Receive y: %s\n", msg)
321. return y
322. }
323. func (s \*Server) computeY(y \*big.Int, x \*big.Int, v \*big.Int, w \*bufio.Writer, e int) string {
324. if y.Cmp(big.NewInt(0)) == 0 {
325. \_, \_ = w.WriteString(FiatShamirProtocol.COMMAND\_ANSWER\_CODE\_ERROR)
326. \_ = w.Flush()
327. return FiatShamirProtocol.COMMAND\_ANSWER\_CODE\_ERROR
328. }
329. l := big.NewInt(0).Exp(y, big.NewInt(2), s.data.N)
330. var r \*big.Int
331. switch e {
332. case 0:
333. r = big.NewInt(0).Mod(x, s.data.N)
334. case 1:
335. r = big.NewInt(0).Mod(
336. big.NewInt(0).Mul(
337. x,
338. v),
339. s.data.N)
340. }
341. log.Printf("l = %s r = %s\n", l.Text(10), r.Text(10))
342. code := ""
343. if l.Cmp(r) == 0 {
344. code = FiatShamirProtocol.COMMAND\_ANSWER\_CODE\_SUCCESS
345. } else {
346. code = FiatShamirProtocol.COMMAND\_ANSWER\_CODE\_ERROR
347. }
348. return code
349. }
350. func (s \*Server) sendAnswerCode(w \*bufio.Writer, statusCode string) {
351. \_, err := w.WriteString(statusCode + "\n")
352. if err != nil {
353. log.Fatal(err)
354. }
355. err = w.Flush()
356. if err != nil {
357. log.Fatal(err)
358. }
359. log.Println("Send status code: " + statusCode)
360. time.Sleep(50 \* time.Millisecond)
361. }
362. func (s \*Server) sendV(w \*bufio.Writer) {
363. vString := ""
364. for \_, v := range s.data.V {
365. vString += v.Text(10) + ","
366. }
367. vString = strings.TrimSuffix(vString, ",")
368. \_, \_ = w.WriteString(vString + "\n")
369. \_ = w.Flush()
370. log.Printf("Send array V %s\n", vString)
371. }
372. func generateE() int {
373. rand := Fingerprints.GetBigRandom()
374. answer, \_ := strconv.Atoi(big.NewInt(0).Mod(rand, big.NewInt(2)).Text(10))
375. return answer
376. }
377. func (data \*ServerData) generateQ() {
378. data.q = Fingerprints.GenerateBigPrimeNumberWithLimit(MIN\_P)
379. }
380. func (data \*ServerData) generateP() {
381. data.p = big.NewInt(0)
382. for {
383. data.generateQ()
384. data.p.Add(
385. big.NewInt(0).Mul(
386. big.NewInt(2),
387. data.q),
388. big.NewInt(1))
389. if Fingerprints.IsPrimeRef(data.p) {
390. if data.p.Cmp(MIN\_P) > 0 && data.p.Cmp(MAX\_P) < 0 {
391. break
392. }
393. }
394. }
395. }
396. func (data \*ServerData) computeN() {
397. data.N = big.NewInt(0).Mul(data.p, data.q)
398. }
400. package rg
401. import (
402. "bufio"
403. "cryptocrouse/src/go/FiatShamirProtocol"
404. "cryptocrouse/src/go/Fingerprints"
405. "fmt"
406. "log"
407. "math/big"
408. "net"
409. "os"
410. "strconv"
411. "strings"
412. "time"
413. )
414. type Rogue struct {
415. conn net.Conn
416. reader \*bufio.Reader
417. writer \*bufio.Writer
418. data \*RogueData
419. }
420. type RogueData struct {
421. singleV \*big.Int
422. V []\*big.Int
423. N \*big.Int
424. R \*big.Int
425. X \*big.Int
426. E int
427. Y \*big.Int
428. S \*big.Int
429. }
430. func InitRogue() \*Rogue {
431. data := &RogueData{
432. V: make([]\*big.Int, 0),
433. N: big.NewInt(0),
434. }
435. return &Rogue{
436. data: data,
437. }
438. }
439. func (r \*Rogue) ConnectToServer() {
440. arguments := os.Args
441. if len(arguments) == 1 {
442. fmt.Println("Please provide host:port.")
443. return
444. }
445. connect := arguments[1]
446. conn, err := net.Dial("tcp", connect)
447. if err != nil {
448. fmt.Println(err)
449. return
450. }
451. log.Printf("Connect to %s\n", connect)
452. r.conn = conn
453. r.data = &RogueData{}
454. r.setupConnections()
455. }
456. func (r \*Rogue) setupConnections() {
457. r.reader = bufio.NewReader(r.conn)
458. r.writer = bufio.NewWriter(r.conn)
459. }
460. func (r \*Rogue) TryToAcceptSecret() {
461. r.receiveOpenKeys()
462. r.generateS()
463. var v \*big.Int
464. for i := 0; i < len(r.data.V); i++ {
465. v = r.data.V[i]
466. if v != nil {
467. r.data.singleV = v
468. break
469. }
470. }
471. flag := r.hackSecret(v)
472. if flag {
473. log.Println("Hack is SUCCESSFUL")
474. return
475. }
476. log.Println("Hack is BAD")
477. }
478. func (r \*Rogue) receiveOpenKeys() {
479. r.receiveN()
480. r.receiveV()
481. }
482. func (r \*Rogue) receiveN() {
483. time.Sleep(50 \* time.Millisecond)
484. msg, err := r.reader.ReadString('\n')
485. if err != nil {
486. log.Fatal(err)
487. }
488. msg = strings.TrimSuffix(msg, "\n")
489. log.Printf("Received N: ~%s~\n", msg)
490. var flag bool
491. r.data.N, flag = big.NewInt(0).SetString(msg, 10)
492. if flag == false {
493. log.Fatal("Received N is bad")
494. }
495. }
496. func (r \*Rogue) receiveV() {
497. time.Sleep(50 \* time.Millisecond)
498. msg, err := r.reader.ReadString('\n')
499. if err != nil {
500. log.Fatal(err)
501. }
502. msg = strings.TrimSuffix(msg, "\n")
503. log.Printf("received array of V: ~%s~\n", msg)
504. vString := strings.Split(msg, ",")
505. for \_, vstr := range vString {
506. var flag bool
507. v, flag := big.NewInt(0).SetString(vstr, 10)
508. if flag == false {
509. log.Fatal("Received N is bad")
510. }
511. r.data.V = append(r.data.V, v)
512. }
513. }
514. func (r \*Rogue) hackSecret(v \*big.Int) bool {
515. r.sendV()
516. for i := 0; i < 5; i++ {
517. if !r.round() {
518. log.Printf("Can not proof with v=%s on %d iteration\n", v.Text(10), i)
519. return false
520. }
521. }
522. log.Printf("Secret accepted successfuly")
523. r.sendEnd()
524. return true
525. }
526. func (r \*Rogue) round() bool {
527. r.generateR()
528. r.computeX()
529. r.sendX()
530. r.receiveE()
531. r.computeY()
532. r.sendY()
533. return r.getAnswer()
534. }
535. func (r \*Rogue) generateR() {
536. for {
537. r.data.R = Fingerprints.GetBigRandomWithLimit(r.data.N)
538. if r.data.R.Cmp(big.NewInt(1)) > 0 && r.data.R.Cmp(r.data.N) < 0 {
539. break
540. }
541. }
542. }
543. func (r \*Rogue) computeX() {
544. r.data.X = big.NewInt(0).Exp(r.data.R, big.NewInt(2), r.data.N)
545. }
546. func (r \*Rogue) sendX() {
547. time.Sleep(50 \* time.Millisecond)
548. \_, \_ = r.writer.WriteString(r.data.X.Text(10) + "\n")
549. \_ = r.writer.Flush()
550. log.Printf("Send X %s\n", r.data.X.Text(10))
551. time.Sleep(50 \* time.Millisecond)
552. }
553. func (r \*Rogue) receiveE() {
554. time.Sleep(50 \* time.Millisecond)
555. msg, \_ := r.reader.ReadString('\n')
556. msg = strings.TrimSuffix(msg, "\n")
557. log.Printf("Received E: %s\n", msg)
558. r.data.E, \_ = strconv.Atoi(msg)
559. }
560. func (r \*Rogue) computeY() {
561. switch r.data.E {
562. case 0:
563. r.data.Y = r.data.R
564. case 1:
565. r.data.Y = big.NewInt(0).Mod(
566. big.NewInt(0).Mul(
567. r.data.S,
568. r.data.R),
569. r.data.N)
570. }
571. }
572. func (r \*Rogue) generateS() {
573. for {
574. r.data.S = Fingerprints.GetBigRandomWithLimit(r.data.N)
575. if r.data.S.Cmp(big.NewInt(1)) == 0 {
576. continue
577. }
578. GCD := big.NewInt(0).GCD(
579. nil,
580. nil,
581. r.data.S,
582. r.data.N)
583. if GCD.Cmp(big.NewInt(1)) == 0 {
584. break
585. }
586. }
587. }
588. func (r \*Rogue) sendY() {
589. time.Sleep(50 \* time.Millisecond)
590. \_, \_ = r.writer.WriteString(r.data.Y.Text(10) + "\n")
591. \_ = r.writer.Flush()
592. log.Printf("Send Y %s\n", r.data.Y.Text(10))
593. time.Sleep(50 \* time.Millisecond)
594. }
595. func (r \*Rogue) getAnswer() bool {
596. msg, err := r.reader.ReadString('\n')
597. if err != nil {
598. log.Fatal(err)
599. }
600. msg = strings.TrimSuffix(msg, "\n")
601. log.Printf("Received answer: ~%s~\n", msg)
602. switch msg {
603. case FiatShamirProtocol.COMMAND\_ANSWER\_CODE\_SUCCESS:
604. return true
605. case FiatShamirProtocol.COMMAND\_ANSWER\_CODE\_ERROR:
606. return false
607. default:
608. return false
609. }
610. }
611. func (r \*Rogue) sendEnd() {
612. \_, \_ = r.writer.WriteString(FiatShamirProtocol.COMMAND\_END + "\n")
613. \_ = r.writer.Flush()
614. }
615. func (r \*Rogue) computeV() {
616. r.data.singleV = big.NewInt(0).Exp(r.data.S, big.NewInt(2), r.data.N)
617. }
618. func (r \*Rogue) sendV() {
619. time.Sleep(50 \* time.Millisecond)
620. \_, \_ = r.writer.WriteString(r.data.singleV.Text(10) + "\n")
621. \_ = r.writer.Flush()
622. log.Printf("Send V %s\n", r.data.singleV.Text(10))
623. }