实验一：SHA1加密算法实现

1. **实验原理与步骤**

1. SHA-1算法概述

SHA-1是一种密码散列函数，产生一个160位（20字节）的哈希值。它被广泛应用于数据完整性验证和数字签名等领域。

2. 初始哈希值

SHA-1算法使用五个初始哈希值（h0, h1, h2, h3, h4），这些值是固定的常数：

h0 = 0x67452301

h1 = 0xEFCDAB89

h2 = 0x98BADCFE

h3 = 0x10325476

h4 = 0xC3D2E1F0

1. 常量定义

SHA-1算法中使用的四个常量（k1, k2, k3, k4）：

k1 = 0x5A827999

k2 = 0x6ED9EBA1

k3 = 0x8F1BBCDC

k4 = 0xCA62C1D6

1. 左旋转操作

左旋转操作用于在哈希计算过程中改变位的位置，增强算法的安全性。

1. 消息预处理

消息预处理包括两步：

1. 填充：在消息末尾添加一个1位，然后添加足够数量的0位，使得消息长度变为512位的倍数减去64位。
2. 长度附加：将原始消息的长度（以位为单位）作为64位整数附加到消息末尾
3. 消息块处理

消息被分割成512位的块，每个块进一步分为16个32位的字。对于每个块，执行以下步骤：

1. 扩展：将16个字扩展到80个字。
2. 主循环：执行80轮迭代，每轮计算新的哈希值。根据轮次选择不同的逻辑函数f和常量k。计算中间结果，并更新哈希值a, b, c, d, e。
3. 哈希值更新

每次处理完一个块后，将当前的哈希值与初始哈希值相加，得到新的哈希值。

1. 输出结果

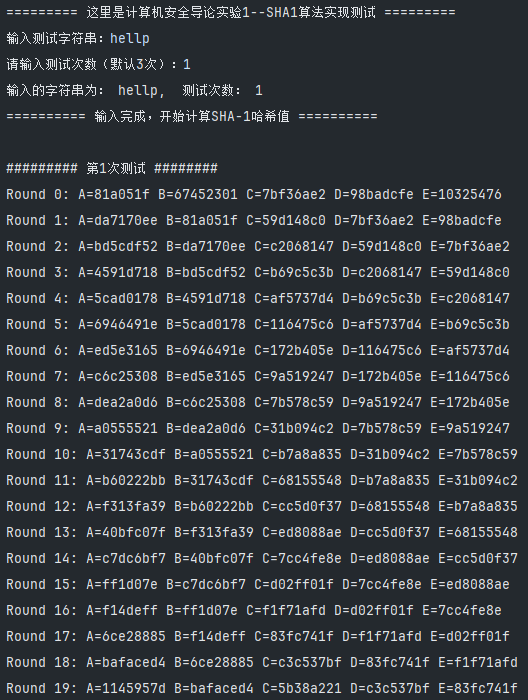
最终的哈希值经过转换，以字节数组的形式输出。

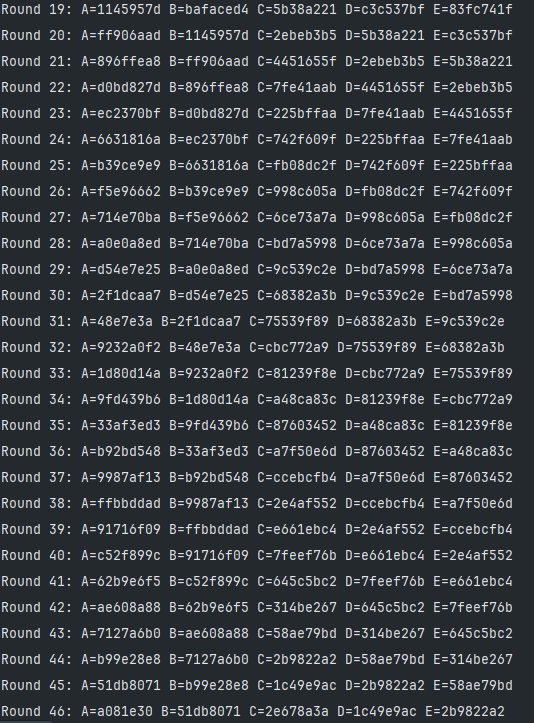
**二、 实验结果**

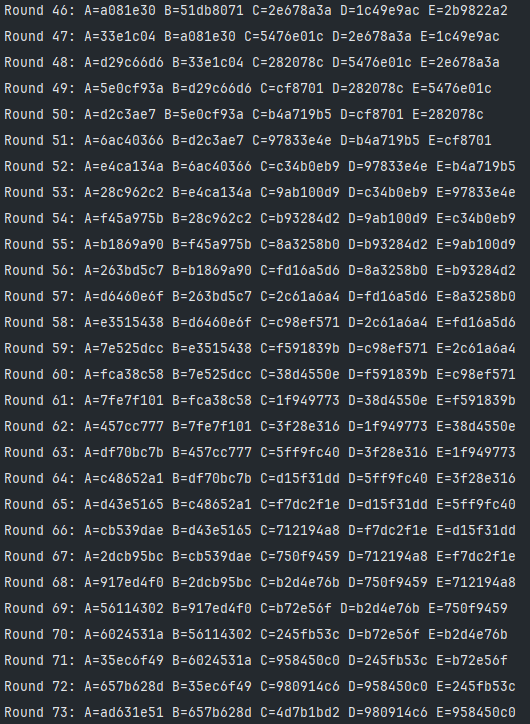
1. 输入测试字符串：用户输入需要计算SHA-1哈希值的字符串。

1. 设置测试次数：用户可以选择测试次数，默认为3次。
2. 计算SHA-1哈希值：对输入字符串进行SHA-1哈希计算，并输出结果。
3. 多次测试：重复上述过程指定次数，输出每次的哈希值。

结果如图：







**三、 实验总结与收获**

1. 理解哈希算法原理

通过实现SHA-1算法，深入理解了哈希算法的工作原理，特别是消息预处理和消息块处理的过程。

2. 提升编程技能

在实现过程中，熟练掌握了Go语言的常用库函数，如encoding/binary用于字节序转换，以及基本的数据结构操作。

3. 加强安全性意识

了解了哈希算法在数据完整性验证和数字签名中的应用，增强了对信息安全的认识。

4. 调试和测试能力

通过多次测试和调试，提高了代码调试和测试的能力，确保代码的正确性和稳定性。

通过本次实验，不仅实现了SHA-1哈希算法，还提升了编程能力和信息安全意识。未来将继续学习更多关于密码学的知识，进一步提升自己的技术水平。

**四、 附加材料**

源代码（Go语言）：

|  |
| --- |
| 1. package main 3. import ( 4. "encoding/binary" // 编码 5. "fmt"             // 流处理、打印信息 6. ) 8. **const** ( 9. // SHA-1初始哈希值 10. h0 = 0x67452301 11. h1 = 0xEFCDAB89 12. h2 = 0x98BADCFE 13. h3 = 0x10325476 14. h4 = 0xC3D2E1F0 16. // SHA-1常量 17. k1 = 0x5A827999 18. k2 = 0x6ED9EBA1 19. k3 = 0x8F1BBCDC 20. k4 = 0xCA62C1D6 21. ) 23. // 左旋转操作 24. func leftRotate(value uint32, shift uint) uint32 { 25. **return** (value << shift) | (value >> (32 - shift)) 26. } 28. // 消息预处理 29. func preprocess(message []byte) []byte { 30. // 计算消息的长度（以位为单位） 31. ml := uint64(len(message) \* 8) 33. // 填充消息，使长度为512位的倍数 34. message = append(message, 0x80) // 添加一个1位 35. **for** len(message)%64 != 56 { 36. message = append(message, 0x00) // 填充0直到满足条件 37. } 39. // 添加消息的长度（64位） 40. mlBytes := make([]byte, 8) 41. binary.BigEndian.PutUint64(mlBytes, ml) 42. message = append(message, mlBytes...) 44. **return** message 45. } 47. // 处理消息块 48. func processBlock(block []byte, h []uint32) { 49. // 将块分成16个32位的字 50. var w [80]uint32 51. **for** i := 0; i < 16; i++ { 52. w[i] = binary.BigEndian.Uint32(block[i\*4 : (i+1)\*4]) 53. } 55. // 扩展到80个字 56. **for** i := 16; i < 80; i++ { 57. w[i] = leftRotate(w[i-3]^w[i-8]^w[i-14]^w[i-16], 1) 58. } 60. a, b, c, d, e := h[0], h[1], h[2], h[3], h[4] 62. // 主循环 63. **for** i := 0; i < 80; i++ { 64. var f, k uint32 65. **if** i < 20 { 66. f = (b & c) | ((^b) & d) 67. k = k1 68. } **else** **if** i < 40 { 69. f = b ^ c ^ d 70. k = k2 71. } **else** **if** i < 60 { 72. f = (b & c) | (b & d) | (c & d) 73. k = k3 74. } **else** { 75. f = b ^ c ^ d 76. k = k4 77. } 79. // 计算中间结果 80. temp := leftRotate(a, 5) + f + e + k + w[i] 81. e = d 82. d = c 83. c = leftRotate(b, 30) 84. b = a 85. a = temp 87. // 输出每一轮的中间结果 88. fmt.Printf("Round %d: A=%x B=%x C=%x D=%x E=%x\n", i, a, b, c, d, e) 89. } 91. // 更新哈希值 92. h[0] += a 93. h[1] += b 94. h[2] += c 95. h[3] += d 96. h[4] += e 97. } 99. // SHA-1算法主函数 100. func sha1(message []byte) [20]byte { 101. // 初始化哈希值 102. h := [5]uint32{h0, h1, h2, h3, h4} 104. // 预处理消息 105. message = preprocess(message) 107. // 处理每一个512位的块 108. **for** i := 0; i < len(message); i += 64 { 109. block := message[i : i+64] 110. processBlock(block, h[:]) 111. } 113. // 将哈希值转换为字节数组 114. var hash [20]byte 115. **for** i := 0; i < 5; i++ { 116. binary.BigEndian.PutUint32(hash[i\*4:(i+1)\*4], h[i]) 117. } 119. **return** hash 120. } 122. func main() { 123. fmt.Printf("========= 这里是计算机安全导论实验1--SHA1算法实现测试 =========\n") 124. fmt.Print("输入测试字符串：") 125. var input string 126. \_, err := fmt.Scanln(&input) 127. **if** err != nil { 128. fmt.Printf("输入错误: %v\n", err) 129. **return** 130. } 131. fmt.Print("请输入测试次数（默认3次）：") 132. count := 3 133. \_, err = fmt.Scanf("%d", &count) 134. **if** err != nil { 135. fmt.Printf("使用默认值") 136. } 137. fmt.Printf("输入的字符串为： %s,  测试次数： %d \n", input, count) 138. fmt.Println("========== 输入完成，开始计算SHA-1哈希值 ==========") 139. sha1Res := make([][20]byte, count) 140. **for** i := 1; i <= count; i++ { 141. fmt.Printf("\n######### 第%d次测试 ########\n", i) 142. message := []byte(input) 143. hash := sha1(message) 144. fmt.Printf("---------- SHA-1计算结果: %x\n", hash) 145. sha1Res[i-1] = hash 146. } 147. fmt.Printf("\n各次加密结果：\n") 148. **for** i, hash := range sha1Res { 149. fmt.Printf("第%d次 ：%x\n", i+1, hash) 150. } 151. fmt.Println("\n========== 测试结束 ==========") 152. } |

实验二：公钥证书的颁发与使用

1. **实验原理与步骤**

1. SSH概述

SSH（Secure Shell）是一种网络协议，用于安全地远程登录到其他计算机。传统的SSH登录方式依赖于用户名和密码，这种方式存在被暴力破解的风险。为了提高安全性，可以使用SSH密钥对进行身份验证。

2. SSH密钥对

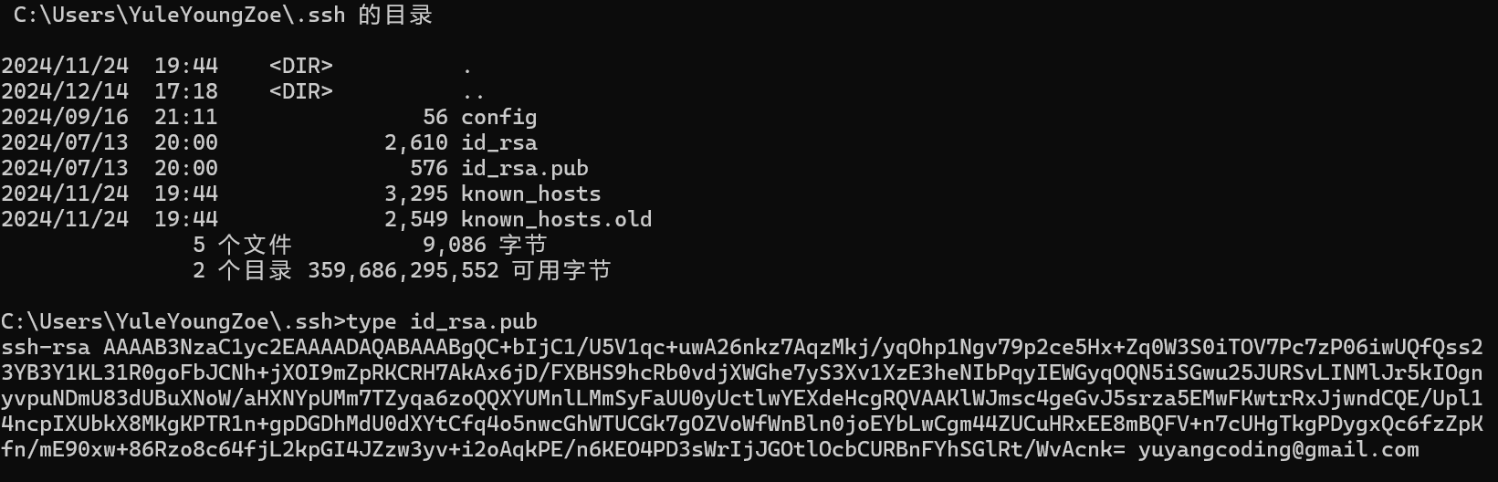
a. 公钥：存储在服务器，用于验证客户端的身份。

b. 私钥：存储在客户端，用于加密发送发送给服务器的数据

3. 配置步骤

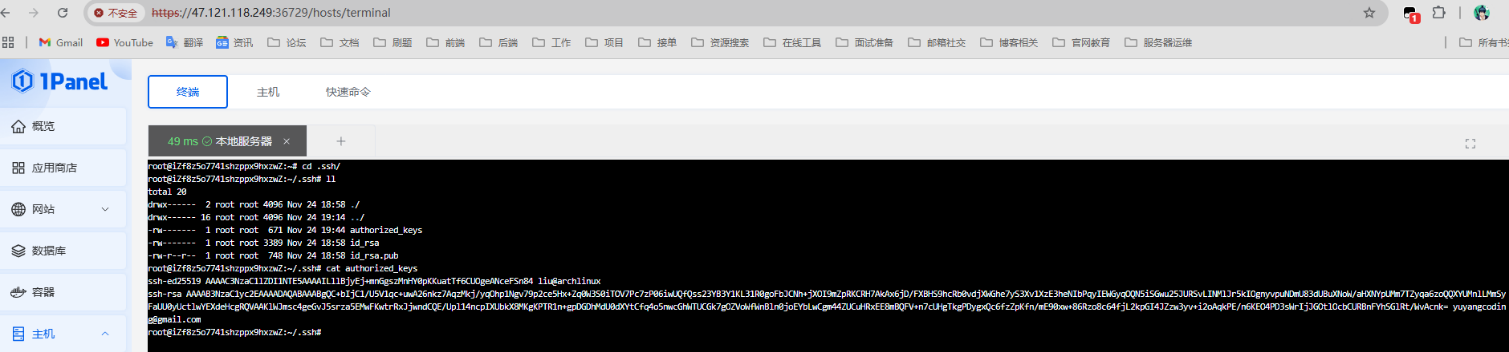
a. 生成SSH密钥对：

本地打开命令行，使用指令 ssh-keygen -t rsa -b 4096 -C "yuyangcoding@gmail.com", 在默认路径 ~/.ssh下生成密钥



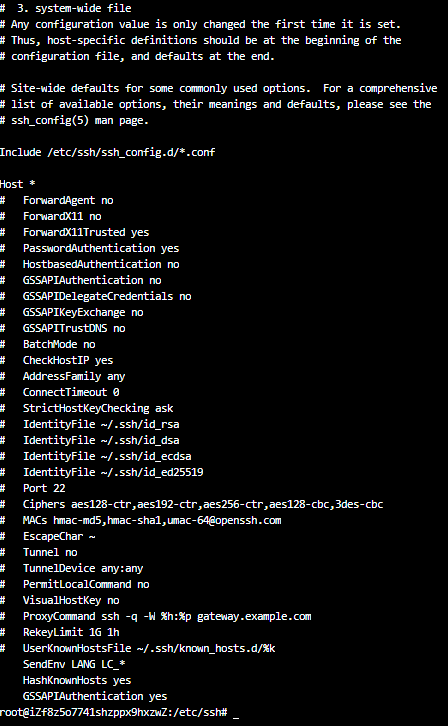
b. 将公钥复制到服务器

远程主机安装了1panel，这里直接使用其登录控制面板，并将公钥存放在root/.ssh/authorized\_keys文件中



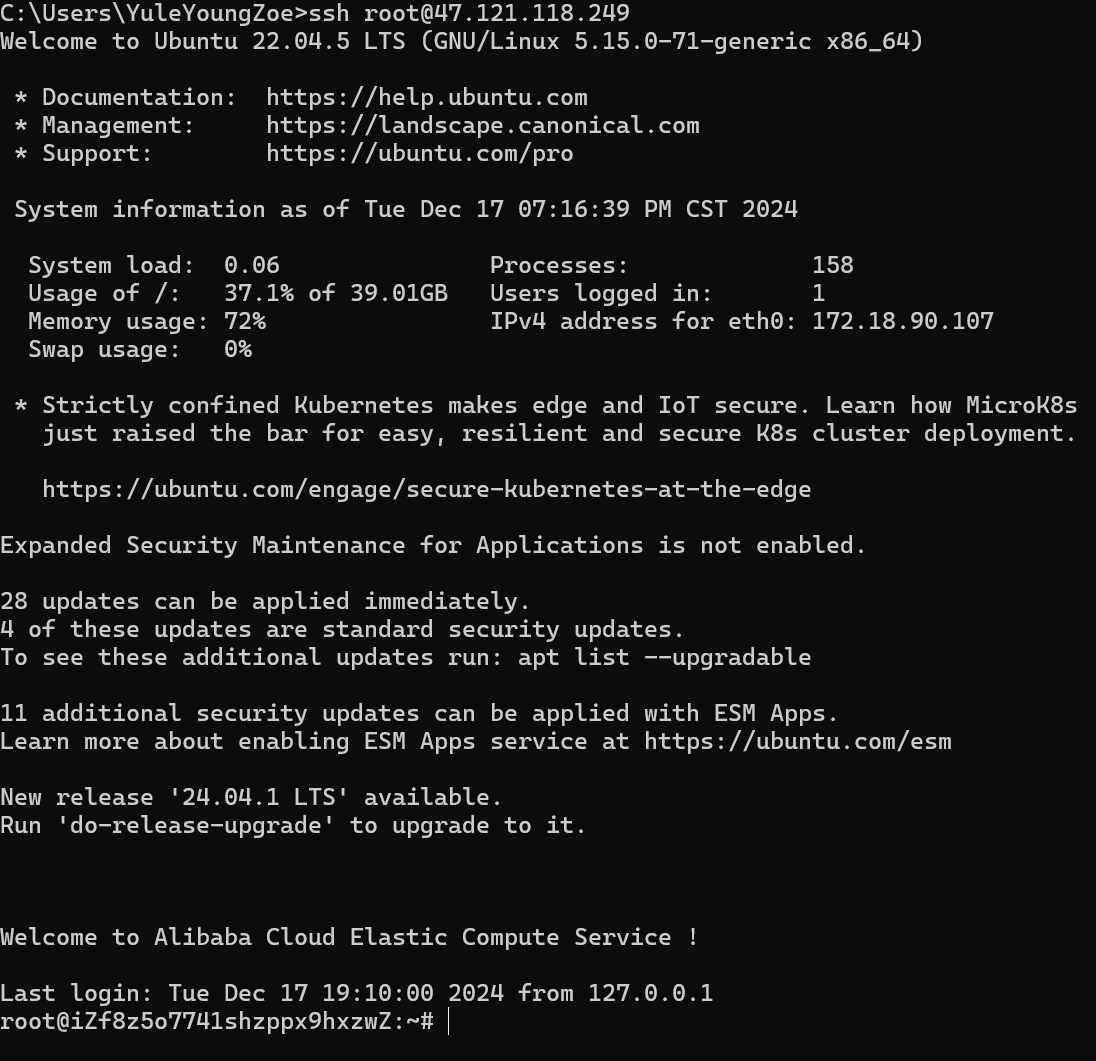
1. 配置SSH服务

编辑/etc/ssh/sshd\_config文件并重启SSH服务



1. **实验结果**

测试连接

****

1. **实验总结与收获**

1. 安全性提升：通过使用SSH密钥登录，大大提高了服务器的安全性，避免了密码被暴力破解的风险。

2. 操作便捷：密钥登录无需每次输入密码，简化了登录过程，提高了工作效率。

3. 配置细节：在配置过程中，注意文件权限的设置，确保 .ssh 目录和 authorized\_keys 文件的权限正确，这是配置成功的关键。

4. 故障排除：在配置过程中遇到问题时，查看日志文件 /var/log/auth.log 可以帮助定位问题，及时解决配置错误。

实验三：TLS协议解析

1. **实验原理与步骤**

TLS（Transport Layer Security）协议是用于在网络上提供安全通信的协议。它通过加密数据、验证通信双方的身份以及保护数据完整性来确保通信的安全性。TLS握手过程是TLS协议的核心部分，主要分为以下几个步骤：

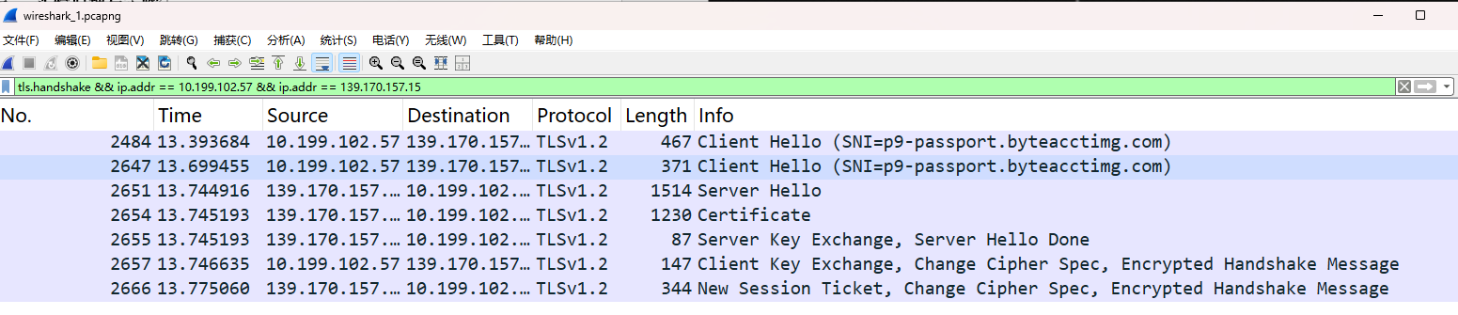
1. ClientHello：客户端向服务器发送一个 ClientHello 报文，包含支持的TLS版本、加密套件、随机数等信息。
2. ServerHello：服务器回应客户端，选择一个加密套件，返回服务器随机数等信息。
3. Certificate：服务器发送其证书链，用于身份验证。
4. ServerKeyExchange（可选）：服务器发送额外的密钥交换参数（某些加密套件需要）。
5. ServerHelloDone：服务器通知客户端握手消息发送完毕。
6. ClientKeyExchange：客户端发送密钥交换信息。
7. ChangeCipherSpec：客户端和服务器分别发送 ChangeCipherSpec 报文，通知对方从明文切换到加密模式。
8. Finished：客户端和服务器分别发送 Finished 报文，表示握手完成，可以开始传输应用数据。

本次实验使用Wireshark进行数据抓包和分析，具体步骤：

* + - 1. 打开Wireshark监听Wlan并开启捕获
      2. 在浏览器中打开某个网页进行资源请求，（我这里访问的是字节旗下的掘金官网）
      3. 在Wireshark中设置过滤器并应用（这里的过滤条件是 tls.handshake && ip.addr == 10.199.102.57 && ip.addr == 139.170.157.15）
      4. 检查捕获到的数据包并分析

1. **实验结果**

捕获到的数据包如下：



可以看到流程：

ClientHello 🡪 ServerHello 🡪 Certificate 🡪 ServerKeyExchange 🡪 ServerHelloDone 🡪 ClientKeyExchange 🡪 Change Cipher Spec

符合我们的预期设想

1. **实验总结与收获**

1. 理解TLS协议：通过实验，深入理解了TLS协议的基本工作原理和握手过程。

2. 掌握Wireshark使用：学会了如何使用Wireshark捕获和分析网络流量，特别是TLS握手过程。

实验四：Web用户口令解析

1. **实验原理与步骤**

本实验通过使用Go语言编写一个简单的Web应用，实现用户登录功能，并利用Wireshark抓取网络数据包，分析HTTP请求与响应过程中的数据传输情况。主要涉及的技术点包括：

1. Go语言：用于开发Web服务端，处理客户端发送的登录请求。
2. Gin框架：作为Go语言的Web框架，简化了路由、模板渲染等操作。
3. HTML/CSS：用于构建前端界面，提供用户输入用户名和密码的表单。
4. Wireshark：一款强大的网络协议分析工具，用于监听网络流量，捕获并分析数据包。

1. 环境搭建：

1. 安装Go语言环境。
2. 安装Gin框架。

c. 安装Wireshark。

2. 代码编写：

a. 编写main.go文件，设置路由处理登录请求。

b. 创建login.html文件，设计登录页面。

c. 运行服务：

d. 启动Go服务，监听8080端口。

3. Wireshark与网络配置：

a. 将本机ip加入回环监听

b. 打开Wireshark，选择合适的网络接口进行监听。

c. 设置过滤器（例如：http.request.method == "POST"），以便只捕获登录相关的POST请求。

4. 测试登录：

a. 在浏览器中访问登录页面。

b. 输入用户名和密码，提交登录请求。

c. 观察Wireshark中捕获的数据包，分析请求与响应的内容。

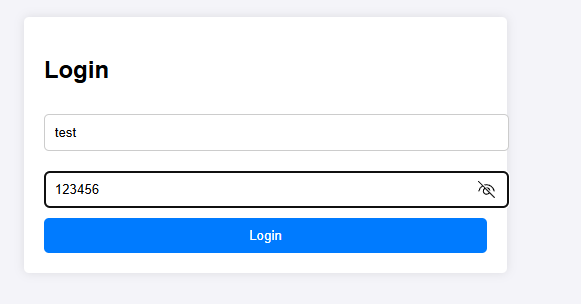
5. 结果分析：

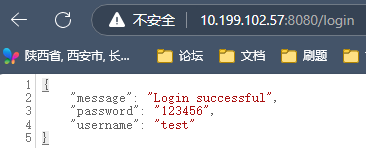
a. 检查Wireshark中捕获的HTTP请求是否包含正确的用户名和密码。

b. 分析服务器返回的响应内容，确认登录成功的信息。

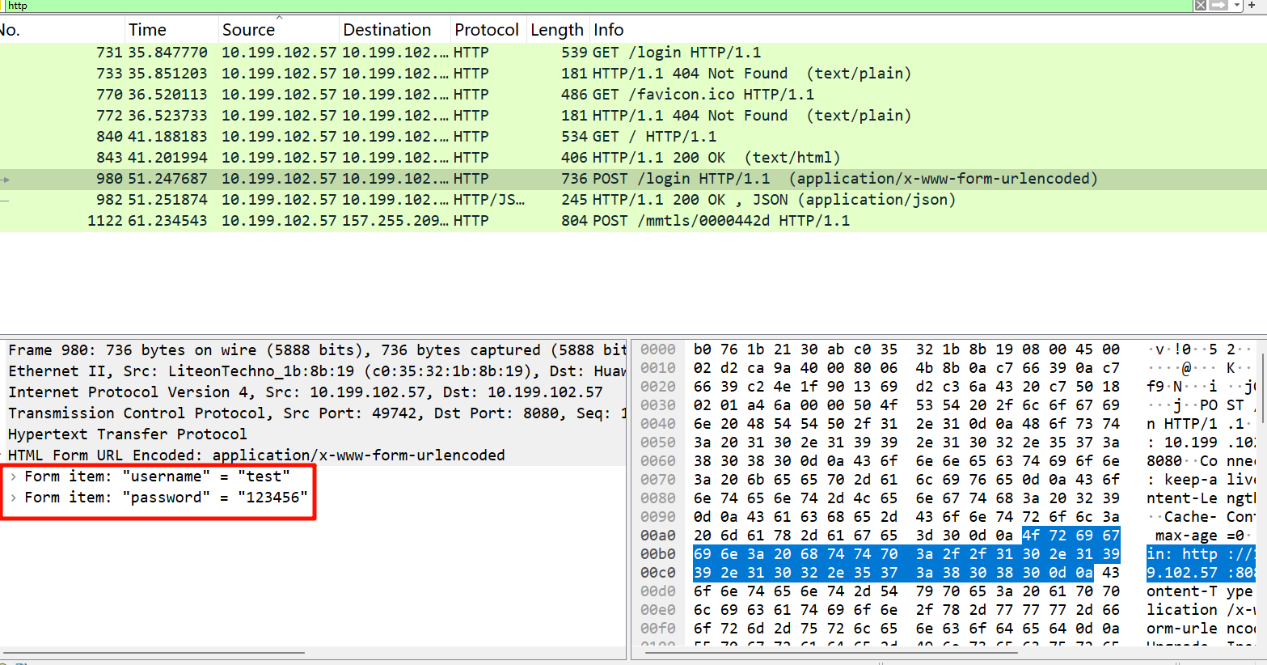
1. **实验结果**

用户页面：





Wireshark抓包结果：



1. **实验总结与收获**

通过本次实验，我深刻理解了Web应用的工作原理，特别是HTTP协议的请求与响应机制。同时，我也学会了如何使用Wireshark这一强大的工具来分析网络流量，这对于网络安全和性能优化都有很大的帮助。此外，我还熟悉了Go语言和Gin框架的基本用法，这为我今后的学习和工作打下了坚实的基础。