|  |
| --- |
| **Github（或者Coding）账号：https://github.com/tsrigo/xdu\_crypto\_exps/tree/main/exp3** |
| **个人博客关于密码学实验的链接：[密码学实验3](https://blog.csdn.net/weixin_45574854/article/details/134655088?spm=1001.2014.3001.5501)** |
| **实验题目（中文）：**   1. **RSA大礼包（密码挑战赛赛题三）** |
| **实验摘要（中文）：**  **本实验针对一个RSA加解密软件的加密数据进行了分析和破解。实验包括了对加密数据的截获以及尝试恢复RSA体制的相关信息。加密过程中使用了分片方法，每次加密最多8个明文字符，并对分片明文进行了512比特的填充，其中包括了64比特的标志位和32比特的通信序号。实验过程分为五个步骤：预处理、共模攻击、低指数攻击、费马分解法和Pollard p-1分解法，通过这些方法尝试破解加密信息。每个步骤都使用了Python代码来实现相应的算法，并最终得到了破解结果。** |
| **题目描述（清楚描述题目中文，写出自己的理解，请勿复制原题目）**   1. **有人制作了一个 RSA 加解密软件，发送了一些加密数据，我们对此进行了截获，并据此尽可能恢复出相关RSA体制信息。** 2. **对文本加密：每次加密最多 8 个明文字符；明文超过 8 个字符时，对明文分片，每个分片不超过 8 个字符。**  分片明文填充为 512 比特消息后再进行加密，填充规则为高位添加 64 比特标志位，随后加上 32 比特通信序号，再添加若干个 0，最后 64 比特为明文分片字符对应的 ASCII 码。  1. **数据帧格式：其中数据都是 16 进制表示，**     **。**   1. **Alice 初次使用该软件，可能会重复发送某一明文分片。** |
| **过程（包括背景，原理：必要的公式，图表；步骤，如有必要画出流程图，给出主要实现步骤代码）**  第一步： 预处理  第二步：共模攻击  在实现 RSA 时，为方便，我们可能给每一用户相同的模数，虽然加解密密钥不同，然而这样做是不行的。  设两个用户的公开钥分别为和 ，且和互素(一般情况都成立)，明文消息是m ，密文分别是    敌手截获密文后，可如下恢复m。用推广的 Euclid 算法求出满足    的两个整数r和 s,其中一个为负，设为r，再次用推广欧几里得算法求出，由此得    代码如下：   1. **from** Crypto.Util.number **import** long\_to\_bytes 3. **print**("扫描潜在的共模攻击情况，其中N被重用但e不同:") 5. # 初始化已扫描模数列表和具有相同模数的对象列表 6. scanned = set() 7. same\_module\_objs = [] 9. # 遍历所有帧，以识别共模攻击候选者 10. **for** i **in** range(num\_of\_frames): 11. **if** ns[i] **not** **in** scanned: 12. scanned.add(ns[i]) 13. attack\_candidate = [i] 14. **for** j **in** range(i + 1, num\_of\_frames): 15. **if** ns[i] == ns[j] **and** es[i] != es[j]: 16. attack\_candidate.append(j) 17. **if** len(attack\_candidate) > 1: 18. same\_module\_objs.append(attack\_candidate) 20. **print**("满足条件 ", same\_module\_objs) 22. # 扩展欧几里得算法 23. **def** egcd(a, b): 24. **if** a == 0: 25. **return** b, 0, 1 26. **else**: 27. g, y, x = egcd(b % a, a) 28. **return** g, x - (b // a) \* y, y 30. # 破解共模攻击的函数 31. **def** crack\_on\_same\_module(same\_module\_objs): 32. crack\_results = {} 33. **for** indices **in** same\_module\_objs: 34. index0, index1 = indices[0], indices[1] 35. g, r, s = egcd(es[index0], es[index1]) 36. # 确保找到的r和s满足条件 37. **assert** r \* es[index0] + s \* es[index1] == g 38. # 根据找到的r和s解密消息 39. int\_m = pow(cs[index0], r, ns[index0]) \* pow(cs[index1], s, ns[index0]) % ns[index0] 40. str\_m = long\_to\_bytes(int\_m)[-8:].decode() 41. crack\_results[index0] = str\_m 42. crack\_results[index1] = str\_m 43. **print**(f"破解帧 {indices} -> m = {str\_m}") 44. **return** crack\_results 46. # 执行攻击 47. crack\_results = crack\_on\_same\_module(same\_module\_objs)   结果为    第三步：低指数攻击  RSA低指数攻击主要是针对公钥指数$e$特别小的情况，特别是当$e$为3时，这种攻击尤为有效。这类攻击通常包括以下几种情况：  1. 低加密指数广播攻击（Håstad's broadcast attack）：  当同一个消息被发送给多个接收者，并且每个接收者都有不同的公钥模数，但是公钥指数相同（且较小）时，攻击者可以使用中国剩余定理（Chinese Remainder Theorem, CRT）来恢复原始消息。如果攻击者截获了至少个加密后的消息（），那么可以通过解同余方程组来找到一个的值，它是所有的公共解。如果小于所有的乘积，那么可以直接对开次方根来得到消息。  公式表示为：    使用CRT找到一个解，满足，其中。如果，则可以通过计算次方根来得到。  2. 低加密指数重复内容攻击：  如果攻击者知道原始消息的一部分，并且该部分的信息足够多以至于的剩余部分不足以产生足够的熵来抵抗次方根攻击，那么攻击者可以尝试对剩余部分进行暴力破解，因为的值会非常小。  3. 低加密指数选择明文攻击：  如果攻击者可以让系统加密特定构造的消息，那么可以选择特定的消息，使得非常小，从而能够直接计算出。  在实际应用中，为了防止这类攻击，通常会选择一个较大的，例如，并且在加密之前对消息进行适当的填充，以确保加密后的消息足够大，不会低于任何模数。  代码   1. **import** gmpy2 2. **from** itertools **import** combinations 4. # 需要提前定义或传入的全局变量 5. # num\_of\_frames, es, cs, ns, crack\_results 7. # 打印信息，指出正在检查是否存在低加密指数 8. **print**("遍历所有指数e，如果存在e小于10，则可能成为低加密指数广播的攻击目标") 9. low\_power\_objs = {} 10. **for** i **in** range(num\_of\_frames): 11. **if** es[i] <= 10: 12. low\_power\_objs.setdefault(es[i], []).append(i) 13. **print**("满足条件：", low\_power\_objs) 15. **def** egcd(a, b): 16. # 扩展欧几里得算法实现 17. **if** a == 0: 18. **return** (b, 0, 1) 19. **else**: 20. g, y, x = egcd(b % a, a) 21. **return** (g, x - (b // a) \* y, y) 23. **def** long\_to\_bytes(value): 24. # 将长整数转换为字节串 25. result = bytearray() 26. **while** value: 27. result.append(value & 0xff) 28. value //= 256 29. **return** bytes(result[::-1]) 31. **def** chinese\_remainder\_theorem(a\_list, m\_list): 32. # 使用中国剩余定理解同余方程组 33. **for** (x, y) **in** combinations(m\_list, 2): 34. **assert** gmpy2.gcd(x, y) == 1, "模数必须两两互质" 35. m = 1 36. result = 0 37. **for** mi **in** m\_list: 38. m \*= mi 39. **for** i **in** range(len(m\_list)): 40. Mi = m // m\_list[i] 41. \_, Mi\_re, \_ = egcd(Mi, m\_list[i]) 42. result += Mi \* Mi\_re \* a\_list[i] 43. **return** result % m 45. **def** crack\_on\_low\_power(low\_power\_objs): 46. # 对低加密指数进行破解 47. **for** power, seqs **in** low\_power\_objs.items(): 48. c\_list = [cs[i] **for** i **in** seqs] 49. n\_list = [ns[i] **for** i **in** seqs] 50. m\_pow = chinese\_remainder\_theorem(c\_list, n\_list) 51. int\_m = gmpy2.iroot(gmpy2.mpz(m\_pow), power) 52. **if** int\_m[1]: # 检查是否是完全的e次方根 53. str\_m = long\_to\_bytes(int\_m[0])[-8:].decode() 54. **for** i **in** seqs: 55. crack\_results[i] = str\_m 56. **print**("破解帧 {} -> m = {}".format(seqs, str\_m)) 57. **else**: 58. **print**("无法找到完全的e次方根") 59. **return** 61. # 假设crack\_results是一个全局字典，用于存储破解结果 62. crack\_results = {} 64. # 开始破解低加密指数 65. crack\_on\_low\_power(low\_power\_objs)   结果如下：    第四步：费马分解法  **原理：**  **费马分解法是一种基于数学原理的算法，用于将一个数分解为两个因数。在RSA加密中，如果选取的两个大素数（我们称之为p和q）非常接近，那么它们的乘积（n = p \* q）可以通过费马分解法被有效地分解。**  **费马分解法的工作原理是寻找两个平方数，它们之间的差刚好等于n。这个方法是基于差平方的恒等式，即(x + y) \* (x - y) = x的平方 - y的平方。如果我们能找到两个数x和y，使得x的平方 - y的平方等于n，那么我们就找到了n的两个因数x + y和x - y。**  **在实际操作中，费马分解法从大约n的平方根开始尝试，并逐渐增加x的值，直到找到一个y，使得x的平方 - n等于y的平方。一旦找到这样的x和y，就可以计算出n的两个因数。**  **这种方法之所以能够攻击RSA，是因为RSA加密的安全性部分依赖于将n分解为p和q是困难的。如果p和q选得太接近，费马分解法就能够快速找到它们，从而破解RSA。因此，在实际应用中，应该避免选择相近的p和q来保证RSA的安全性。**  **代码：**   1. **import** gmpy2 2. **from** Crypto.Util.number **import** long\_to\_bytes 4. **def** crack\_fermat\_method(num\_of\_frames, ns, es, cs): 5. """ 6. 使用费马分解法破解RSA加密的消息。 8. :param num\_of\_frames: 需要破解的帧数 9. :param ns: 包含每个帧的模数 n 的列表 10. :param es: 包含每个帧的公钥指数 e 的列表 11. :param cs: 包含每个帧的密文 c 的列表 12. :return: None 13. """ 14. crack\_results = [None] \* num\_of\_frames  # 存储破解结果的列表 16. **for** i **in** range(num\_of\_frames): 17. # 初始化变量 18. root\_n = gmpy2.isqrt(ns[i]) + 1  # 计算 n 的平方根并向上取整 19. y = gmpy2.isqrt(root\_n \* root\_n - ns[i]) 20. count = 0 21. max\_iterations = 100000 23. # 使用费马方法尝试分解 n 24. **while** (root\_n \* root\_n - ns[i] != y \* y) **and** count < max\_iterations: 25. root\_n += 1 26. y = gmpy2.isqrt(root\_n \* root\_n - ns[i]) 27. count += 1 29. # 检查是否成功分解 30. **if** root\_n \* root\_n - ns[i] == y \* y: 31. p = (root\_n + y) % ns[i] 32. q = (root\_n - y) % ns[i] 33. phi\_n = (p - 1) \* (q - 1) 34. d = gmpy2.invert(es[i], phi\_n) 35. int\_m = pow(cs[i], d, ns[i]) 37. **try**: 38. # 尝试将解密的整数转换为字节串，然后解码为字符串 39. str\_m = long\_to\_bytes(int\_m)[-8:].decode() 40. crack\_results[i] = str\_m 41. **print**(f"破解帧 {i} -> m = {str\_m}") 42. **except** UnicodeDecodeError: 43. **print**(f"错误帧 {i} 解码失败。") 45. **return** crack\_results 47. crack\_results = crack\_fermat\_method(num\_of\_frames, ns, es, cs)   **结果如下：**    步骤五：Pollard p-1 分解  原理：  Pollard's p-1 算法是一种用于寻找大合数的非平凡因子的算法。其基本原理是利用了数论中的一个事实：如果一个合数 n 的一个因子 p-1 的所有质因子的幂都很小，那么这个因子 p 可以相对容易地被找到。原理如下：  1. 选择一个基数 a，通常可以选取2。  2. 选择一个边界值 B，这个值决定了算法尝试寻找的因子的"平滑度"。  3. 对于每个整数 k，从2开始到B结束，计算 a 的 k 阶乘的幂（即 ）。  4. 计算 ，如果这个值大于1且小于n，那么它就是n的一个因子。  5. 如果在B的范围内没有找到因子，可以增加B的值重复步骤3和4。  算法的关键在于，如果n有一个因子p，使得p-1的所有质因子都小于或等于B，那么在步骤3中a^(k!)会变得非常接近于1 mod p。这是因为如果k大于p-1的最大质因子，k!将包含p-1的所有质因子的幂，从而使得成为p的倍数。这样就会给出p。  Pollard's p-1算法适用于那些p-1是"B-平滑"的数，即p-1的所有质因子都不大于B。如果p-1包含大的质因子，这个方法可能会失败。  代码：   1. **import** gmpy2 2. **from** Crypto.Util.number **import** long\_to\_bytes 4. # 使用 Pollard's rho 算法来分解质数 5. **def** pollard\_rho(n): 6. m = 2   # 保证 gcd(2, n) = 1 7. max\_k = 200000  # 设置最大迭代次数 8. **for** k **in** range(1, max\_k + 1): 9. m = pow(m, k, n)  # 计算 m = 2^(k!) mod n 10. gcd\_value = gmpy2.gcd(m - 1, n)  # 计算 gcd(m - 1, n) 11. **if** gcd\_value != 1:  # 如果 gcd 不为 1，则找到一个因子 12. **return** gcd\_value 13. **return** None  # 如果没有找到因子，返回 None 15. # 解决 Pollard's rho 算法的问题 16. **def** crack\_with\_pollard\_rho(): 17. **for** i **in** range(num\_of\_frames):  # 遍历所有帧 18. p = pollard\_rho(ns[i])  # 对每个 ns[i] 使用 Pollard's rho 算法 19. **if** p **is** **not** None:  # 如果 p 不为 None，则找到了一个因子 20. q = ns[i] // p  # 计算另一个因子 21. varphi\_n = (p - 1) \* (q - 1)  # 计算欧拉函数 φ(n) 22. d = gmpy2.invert(es[i], varphi\_n)  # 计算私钥 d 23. int\_m = pow(cs[i], d, ns[i])  # 解密得到消息 m 24. str\_m = long\_to\_bytes(int\_m)[-8:].decode()  # 将消息转换为字符串 25. crack\_results[i] = str\_m  # 存储解密结果 26. **print**("[破解] 帧 {} -> m = {}".format(i, str\_m))  # 打印解密结果 27. # 调用破解函数 28. crack\_with\_pollard\_rho()   结果：    最终结果： |
| **总结（完成心得与其它，主要自己碰到的问题和解决问题的方法）**  **实验心得**  **通过本次实验，我深刻体会到了密码学理论与实践的紧密结合。RSA加密作为一种广泛使用的非对称加密算法，其安全性依赖于大数分解问题的计算难度。然而，本实验让我认识到，即使是RSA这样成熟的加密体系，也可能因为实现上的疏忽或参数选择不当而变得脆弱。**  **实验中的每一步都是对理论知识的实践检验。共模攻击让我了解到，不同用户应使用不同的模数N来避免潜在的安全风险。低指数攻击则揭示了公钥指数e的选择需要谨慎，过低的e值会使得加密系统容易受到特定类型攻击的威胁。费马分解法和Pollard p-1分解法的应用则让我认识到了选择合适的大素数对于RSA安全性的重要性。**  **编写和运行Python代码实现这些攻击算法，不仅加强了我的编程技能，也提高了我对于算法有效性的直观理解。这次实验体现了理论知识与实践能力的结合，强化了我对密码学原理的理解，同时也提醒了我在设计安全系统时需要考虑的各种潜在风险。此外，实验也激发了我对于密码学更深层次问题的好奇心，比如如何设计更加安全的加密算法，以及如何评估一个加密体系的安全性。**  **总之，这次实验不仅是对加密技术的学习，也是对安全思维的一次锻炼。它让我更加明白，安全性是一个动态的目标，需要不断的学习和适应新的威胁，以保护信息不受未授权访问的侵害。。**  **在实验过程中，我还尝试了使用编程实现RSA加密算法。我学习了如何生成随机素数，如何计算欧拉函数，以及如何进行模运算。我还尝试了将字符串转换为十六进制数进行加密，这是一个非常实用的技巧。**  **总的来说，这次实验让我对RSA加密有了更深的理解。** |
| **参考文献（包括参考的书籍，论文，URL等，很重要）**   1. **Rivest, R. L., Shamir, A., & Adleman, L. (1978). A method for obtaining digital signatures and public-key cryptosystems. Communications of the ACM, 21(2), 120-126.** 2. **Boneh, D. (1999). Twenty years of attacks on the RSA cryptosystem. Notices of the AMS, 46(2), 203-213.** 3. **Menezes, A., van Oorschot, P., & Vanstone, S. (1996). Handbook of Applied Cryptography. CRC Press.** 4. **Håstad, J. (1986). Solving simultaneous modular equations of low degree. SIAM Journal on Computing, 15(2), 229-243.** 5. **Pollard, J. M. (1974). Theorems on Factorization and Primality Testing. Proceedings of the Cambridge Philosophical Society, 76(3), 521-528.** 6. **May, A. (2004). Computing the RSA secret key is deterministic polynomial time equivalent to factoring. In CRYPTO 2004. Springer, Berlin, Heidelberg.** 7. **Lenstra, A. K., & Weger, B. D. (2000). On the possibility of constructing meaningful hash collisions for public keys. In ACISP 2000 (pp. 267-279). Springer, Berlin, Heidelberg.** 8. **Ferguson, N., Schneier, B., & Kohno, T. (2010). Cryptography Engineering: Design Principles and Practical Applications. Wiley Publishing.** 9. **Stinson, D. R., & Paterson, M. (2019). Cryptography: Theory and Practice. CRC Press.** 10. **Silverman, R. D. (1987). A Cost-Based Security Analysis of Symmetric and Asymmetric Key Lengths. RSA Laboratories Bulletin, 13.** 11. [**GitHub - jkup64/HomeworkOfCryptography**](https://github.com/jkup64/HomeworkOfCryptography) |