公钥基础设施（PKI, Public Key Infrastructure），是一种遵循既定标准的密钥管理平台,它能够为所有网络应用提供加密和数字签名等密码服务及所必需的密钥和证书管理体系。简单来说，PKI就是利用公钥理论和技术建立的提供安全服务的基础设施。

数字证书是指在互联网通讯中标志通讯各方身份信息的一个数字认证，人们可以在网上用它来识别对方的身份。在PKI体系中，建有证书管理机构CA（Certificate Authority）。CA中心的公钥是公开的，因此由CA中心签发的内容均可以验证。

密钥的生存周期包括：密钥的产生和登记、密钥分发、密钥更新、密钥撤销、密钥销毁等。在产生密钥后，公钥需要在PKI中登记，并通过CA中心的私钥签名后形成公钥证书。由于CA中心的公钥公开，用户可以方便地对公钥证书进行验证，并通过公钥证书来互相交换自己的公钥。进而，PKI作为安全基础设施，能够提供身份认证、数据完整性、数据保密性、数据公正性、不可抵赖性和时间戳六种安全服务。

PKI的应用非常广泛，为网上金融、网上银行、网上证券、电子商务、电子政务等网络中的数据交换提供了完备的安全服务功能。

OpenSSL库提供了相关的基本功能支撑，下面提供一个数字签名及认证的简单示例。

**实验2-2** 在OpenSSL中进行数据签名及验证。

基于第2.3.3节的示例，在OpenSSL中进行数据签名及验证的实验如下所示。

1）使用OpenSSL命令签名并验证

① 生成2048位密钥，存储到文件id\_rsa.key

openssl genrsa -out id\_rsa.key 2048

② 根据私钥文件，导出公钥文件id\_rsa.pub

openssl rsa -in id\_rsa.key -out id\_rsa.pub -pubout

③ 使用私钥对文件message.txt进行签名，输出签名到message.sha256

openssl dgst -sign id\_rsa.key -out rsa\_signature.bin -sha256 message.txt

④ 使用公钥验证签名

openssl dgst -verify id\_rsa.pub -signature rsa\_signature.bin -sha256 message.txt

若验证成功，会输出Verified OK字段。

表2-2为语法：genrsa [options] numbits的参数解释；表2-3为语法：rsa [options]的参数解释；表2-4为语法：dgst [options] [file...]的参数解释。

表2-2 语法解释（语法：genrsa [options] numbits）

|  |  |
| --- | --- |
| 选项 | 作用 |
| -out | 指定输出文件 |
| numbits | 密钥长度，存在默认值 |

表2-3 语法解释（语法：rsa [options]）

|  |  |
| --- | --- |
| 选项 | 作用 |
| -in | 指定输入文件 |
| -out | 指定输出文件 |
| -pubout | 输出公钥 |

表2-4 语法解释（语法：dgst [options] [file...]）

|  |  |
| --- | --- |
| 选项 | 作用 |
| -sign val | 生成签名，同时指定私钥 |
| -versify val | 使用公钥验证签名 |
| -prverify val | 使用私钥验证签名 |
| -out outfile | 输出到文件 |
| -signature infile | 指定签名文件 |
| -sha256 | 使用sha256算法摘要 |
| file | 消息文件 |

2）数字签名程序

（1）编写程序文件signature.cpp

1. #include <stdio.h>
2. #include <string.h>
3. #include <openssl/evp.h>
4. #include <openssl/rsa.h>
5. #include <openssl/pem.h>
6. // 公钥文件名
7. #define PUBLIC\_KEY\_FILE\_NAME "public.pem"
8. // 私钥文件名
9. #define PRIVATE\_KEY\_FILE\_NAME "private.pem"
10. // RSA生成公私钥，存储到文件
11. bool genrsa(int numbit)
12. {
13. EVP\_PKEY\_CTX \*ctx = EVP\_PKEY\_CTX\_new\_id(EVP\_PKEY\_RSA, NULL);
14. if (!ctx)
15. return false;
16. EVP\_PKEY \*pkey = NULL;
17. bool ret = false;
18. int rt;
19. FILE \*prif = NULL, \*pubf = NULL;
20. if (EVP\_PKEY\_keygen\_init(ctx) <= 0)
21. goto err;
22. // 设置密钥长度
23. if (EVP\_PKEY\_CTX\_set\_rsa\_keygen\_bits(ctx, numbit) <= 0)
24. goto err;
25. // 生成密钥
26. if (EVP\_PKEY\_keygen(ctx, &pkey) <= 0)
27. goto err;
28. prif = fopen(PRIVATE\_KEY\_FILE\_NAME, "w");
29. if (!prif)
30. goto err;
31. // 输出私钥到文件
32. rt = PEM\_write\_PrivateKey(prif, pkey, NULL, NULL, 0, NULL, NULL);
33. fclose(prif);
34. if (rt <= 0)
35. goto err;
36. pubf = fopen(PUBLIC\_KEY\_FILE\_NAME, "w");
37. if (!pubf)
38. goto err;
39. // 输出公钥到文件
40. rt = PEM\_write\_PUBKEY(pubf, pkey);
41. fclose(pubf);
42. if (rt <= 0)
43. goto err;
44. ret = true;
45. err:
46. EVP\_PKEY\_CTX\_free(ctx);
47. return ret;
48. }
49. // 生成数据签名
50. bool gensign(const uint8\_t \*in, unsigned int in\_len, uint8\_t \*out, unsigned int \*out\_len)
51. {
52. FILE \*prif = fopen(PRIVATE\_KEY\_FILE\_NAME, "r");
53. if (!prif)
54. return false;
55. // 读取私钥
56. EVP\_PKEY \*pkey = PEM\_read\_PrivateKey(prif, NULL, NULL, NULL);
57. fclose(prif);
58. if (!pkey)
59. return false;
60. bool ret = false;
61. EVP\_MD\_CTX \*ctx = EVP\_MD\_CTX\_new();
62. if (!ctx)
63. goto ctx\_new\_err;
64. // 初始化
65. if (EVP\_SignInit(ctx, EVP\_sha256()) <= 0)
66. goto sign\_err;
67. // 输入消息，计算摘要
68. if (EVP\_SignUpdate(ctx, in, in\_len) <= 0)
69. goto sign\_err;
70. // 生成签名
71. if (EVP\_SignFinal(ctx, out, out\_len, pkey) <= 0)
72. goto sign\_err;
73. ret = true;
74. sign\_err:
75. EVP\_MD\_CTX\_free(ctx);
76. ctx\_new\_err:
77. EVP\_PKEY\_free(pkey);
78. return ret;
79. }
80. // 使用公钥验证数字签名，结构与签名相似
81. bool verify(const uint8\_t \*msg, unsigned int msg\_len, const uint8\_t \*sign, unsigned int sign\_len)
82. {
83. FILE \*pubf = fopen(PUBLIC\_KEY\_FILE\_NAME, "r");
84. if (!pubf)
85. return false;
86. // 读取公钥
87. EVP\_PKEY \*pkey = PEM\_read\_PUBKEY(pubf, NULL, NULL, NULL);
88. fclose(pubf);
89. if (!pkey)
90. return false;
91. bool ret = false;
92. EVP\_MD\_CTX \*ctx = EVP\_MD\_CTX\_new();
93. if (!ctx)
94. goto ctx\_new\_err;
95. // 初始化
96. if (EVP\_VerifyInit(ctx, EVP\_sha256()) <= 0)
97. goto sign\_err;
98. // 输入消息，计算摘要
99. if (EVP\_VerifyUpdate(ctx, msg, msg\_len) <= 0)
100. goto sign\_err;
101. // 验证签名
102. if (EVP\_VerifyFinal(ctx, sign, sign\_len, pkey) <= 0)
103. goto sign\_err;
104. ret = true;
105. sign\_err:
106. EVP\_MD\_CTX\_free(ctx);
107. ctx\_new\_err:
108. EVP\_PKEY\_free(pkey);
109. return ret;
110. }
111. int main()
112. {
113. // 生成长度为2048的密钥
114. genrsa(2048);
115. const char \*msg = "Hello World!";
116. const unsigned int msg\_len = strlen(msg);
117. // 存储签名
118. uint8\_t sign[256] = {0};
119. unsigned int sign\_len = 0;
120. // 签名
121. if (!gensign((uint8\_t \*)msg, msg\_len, sign, &sign\_len))
122. {
123. printf("签名失败\n");
124. return 0;
125. }
126. // 验证签名
127. if (verify((uint8\_t \*)msg, msg\_len, sign, sign\_len))
128. printf("验证成功\n");
129. else
130. printf("验证失败\n");
131. return 0;
132. }

（2）编译并运行

g++ signature.cpp -o signature -lcrypto  
./signature