南开大学

Nankai University

南　　开　　大　　学

网络空间安全学院

数据安全实验报告

实验 1：数据签名及验证

姓名：付政烨

学号：2113203

年级：2021 级

专业：信息安全、法学

2024 年 3 月 29 日

目录

一、 实验要求

参照教材 2.3.6，实现在 OpenSSL 中进行数据签名及验证这一实验。

二、 实验内容

基于第 2.2.4 节的示例，在 OpenSSL 中进行数据签名及验证的实验如下所示。

(一) 使用 OpenSSL 命令签名并验证

- 生成 2048 位密钥，并存储到文件 `id_rsa.key` 中:

```
openssl genrsa -out id_rsa.key 2048
```

- 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
```

- 使用私钥对文件 `message.txt` 进行签名，并将签名输出到文件 `rsa_signature.bin`:

```
openssl dgst -sign id_rsa.key -out rsa_signature.bin -sha256 message.txt
```

- 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。以下为部分语法解释:

指令: genrsa [options] numbits	
选项	作用
-out	指定输出文件
指令: 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
指令: dgst [options] [file...]	
选项	作用
-sign val	生成签名，同时指定私钥
-verify 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	源文件

表 1: OpenSSL 命令参数说明

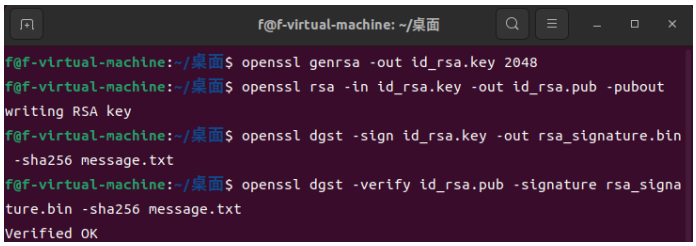

图 1: 实验结果 (a)

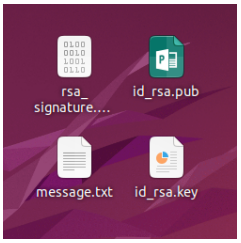

图 2: 实验结果 (b)

(二) 数字签名程序

- 编写程序文件 signature.cpp（详见附件：代码一）
- 编译并运行

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

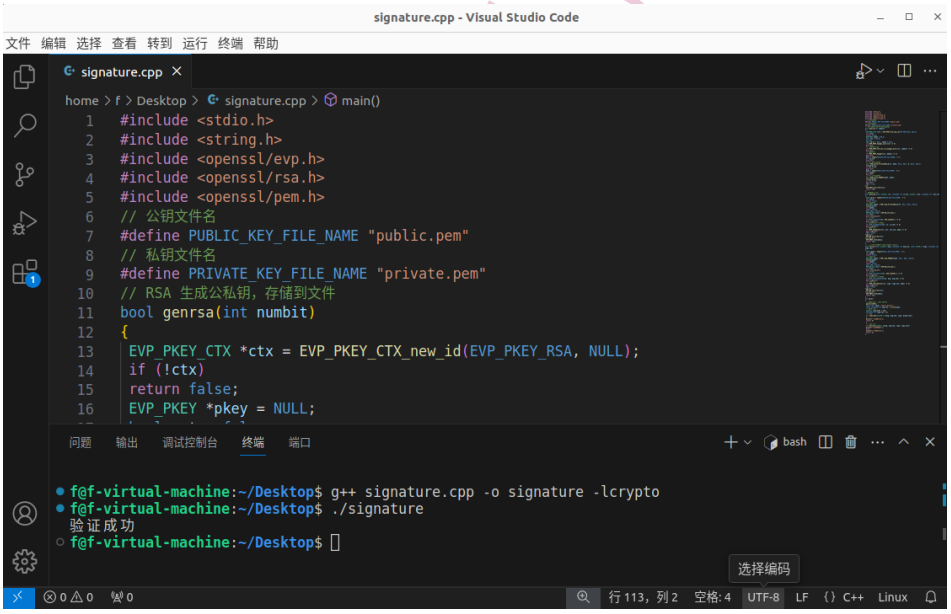

图 3: 验证成功

三、 探究实验：使用 OpenSSL 实现双因素身份验证

(一) 实验简介

在数字世界中，信息的安全性和真实性是至关重要的。数据签名及验证技术在确保信息未被篡改以及验证信息来源方面发挥着关键作用。使用 OpenSSL 进行命令行签名和验证是理解这一过程的基础。然而，在现实世界的应用中，仅依靠单一的签名和验证机制并不足以提供全面的安全保障。例如，如果私钥被未经授权的人员访问，那么该私钥签名的所有信息都可能处于风险之中。这种情况下，仅仅依赖于密钥本身的保护显然是不够的。因此，引入双因素身份验证成为了增强安全防护的必要步骤。

双因素身份验证（2FA）要求用户提供两种身份验证因素：一样是他们所知道的（如密码或 PIN 码），另一样是他们所拥有的（如手机或安全令牌）。

在这小节的实验中，我探索了如何将传统的 OpenSSL 签名和验证机制与一个额外的安全层结合，**即要求用户在使用私钥签名信息时，还必须提供一个密码。这不仅保留了基本的数字签名功能，还增加了一个物理因素的认证步骤，提高了整个验证过程的安全性**。通过这种方式，即便私钥文件被盗，没有对应的密码和身份验证，攻击者也无法滥用这个私钥。

(二) 实验过程

双因素身份验证结合了两种不同类型的认证方法来提高安全性。在本实验中，我们将结合使用密码和数字签名来验证用户的身份。使用 OpenSSL，我们可以实现这一过程，具体步骤和指令如下：

- **生成私钥并设置密码保护**：为私钥文件设置密码保护，可以在生成密钥时使用 `-aes256`（或其他加密算法，如 `aes128`, `aes192`, `des`, `des3` 等）来实现。这确保了每次使用私钥时都需要输入密码。

```
openssl genrsa -aes256 -out id_rsa_protected.key 2048
```

这条命令会要求输入一个密码来加密私钥。之后每次使用这个私钥时都需要提供这个密码。

- **导出公钥**：从已加密的私钥文件中导出公钥。这一步不需要密码，因为公钥不需要保密。

```
openssl rsa -in id_rsa_protected.key -outform PEM -pubout -out id_rsa.pub
```

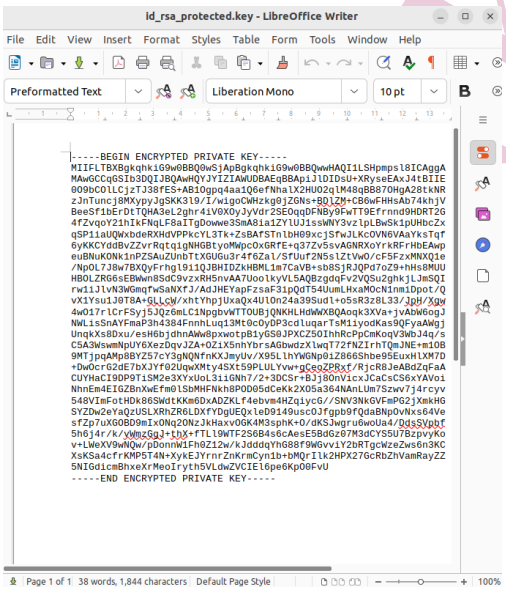

图 4: id_rsa_protected.key

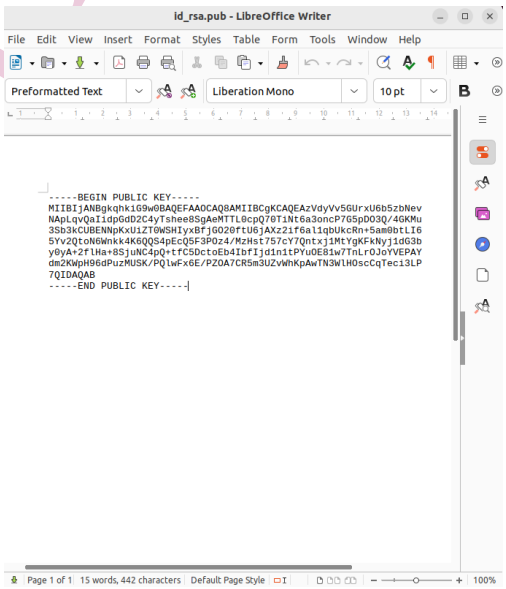

图 5: id_rsa.pub

这条命令会生成一个名为 `id_rsa.pub` 的文件，里面存储着公钥。

- **创建一个待签名的消息文件**：这个步骤只是创建一个示例文件，称为 `message.txt`，其中包含要签名的消息。

```
echo "This is a secret message." > message.txt
```

- **使用私钥进行签名**：使用加密的私钥对 `message.txt` 文件进行签名。在这个过程中，系统会提示输入之前设置的密码。

```
openssl dgst -sha256 -sign id_rsa_protected.key -out message.sha256 message.txt
```

这将创建一个签名文件 `message.sha256`，该文件包含了消息的数字签名。

- **验证签名**：使用公钥验证消息文件的签名是否有效。此步骤确保消息确实是使用匹配的私钥签名的。

```
openssl dgst -sha256 -verify id_rsa.pub -signature message.sha256 message.txt
```

如果签名验证成功，将输出 `Verified OK`。

图 6: 实验结果 (a)

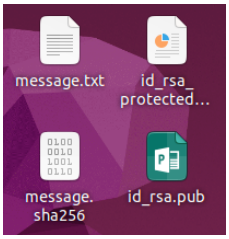

图 7: 实验结果 (b)

- **实现双因素身份验证**：

现在，每次使用私钥时，用户都需要输入密码。为了增加双因素认证，我们还可以在脚本或应用程序层面增加一个额外的身份验证步骤，比如发送一个一次性验证码到用户的手机或电子邮件，并要求用户输入这个验证码才能继续。

四、 实验心得与体会

通过这次实验，我深刻理解了数据签名及验证的重要性和实现方式，尤其是在保障信息传输安全和验证信息真实性方面的应用。实验的第一部分让我通过 OpenSSL 命令行工具亲手操作了生成密钥、对数据进行签名和验证签名的全过程。这一过程不仅加深了我对理论知识的理解，还提升了我在实际操作中解决问题的能力。特别是在探究实验中，通过结合双因素身份验证（2FA）的概念，我学习到了如何增强数据签名和验证的安全性。通过要求用户提供两种认证因素，一是他们知道的（如密码），二是他们拥有的（如密钥），极大地增加了安全性，即使私钥被泄露，没有密码也无法滥用。

这次实验也让我意识到，无论技术多么高级，安全性的核心仍然在于人。即使是使用了高度安全的加密技术，如果操作不当或密码管理不善，同样会导致安全漏洞。因此，在实际应用中，除了技术层面的保护，还需要加强个人和组织的安全意识和操作规范。

附 件

代码一

```
#include <stdio.h>
#include <string.h>
#include <openssl/evp.h>
#include <openssl/rsa.h>
#include <openssl/pem.h>
// 公钥文件名
#define PUBLIC_KEY_FILE_NAME "public.pem"
// 私钥文件名
#define PRIVATE_KEY_FILE_NAME "private.pem"
// RSA 生成公私钥，存储到文件
bool genrsa(int numbit)
{
        EVP_PKEY_CTX* ctx = EVP_PKEY_CTX_new_id(EVP_PKEY_RSA, NULL);
        if (!ctx)
                return false;
        EVP_PKEY* pkey = NULL;
        bool ret = false;
        int rt;
        FILE* prif = NULL, * pubf = NULL;
        if (EVP_PKEY_keygen_init(ctx) <= 0)
                goto err;
        // 设置密钥长度
        if (EVP_PKEY_CTX_set_rsa_keygen_bits(ctx, numbit) <= 0)
                goto err;
        // 生成密钥
        if (EVP_PKEY_keygen(ctx, &pkey) <= 0)
                goto err;
        prif = fopen(PRIVATE_KEY_FILE_NAME, "w");
        if (! prif )
                goto err;
        // 输出私钥到文件
        rt = PEM_write_PrivateKey(prif, pkey, NULL, NULL, 0, NULL, NULL);
        fclose ( prif );
        if (rt <= 0)
                goto err;
        pubf = fopen(PUBLIC_KEY_FILE_NAME, "w");
        if (!pubf)
                goto err;
        // 输出公钥到文件
        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
82          sign_len)
83  {
84          FILE* pubf = fopen(PUBLIC_KEY_FILE_NAME, "r");
```

```
85          if (!pubf)
86                  return false;
87          //读取公钥
88          EVP_PKEY* pkey = PEM_read_PUBKEY(pubf, NULL, NULL, NULL);
89          fclose(pubf);
90          if (!pkey)
91                  return false;
92          bool ret = false;
93          EVP_MD_CTX* ctx = EVP_MD_CTX_new();
94          if (!ctx)
95                  goto ctx_new_err;
96          //初始化
97          if (EVP_VerifyInit(ctx, EVP_sha256()) <= 0)
98                  goto sign_err;
99          //输入消息，计算摘要
100         if (EVP_VerifyUpdate(ctx, msg, msg_len) <= 0)
101                 goto sign_err;
102         //验证签名
103         if (EVP_VerifyFinal(ctx, sign, sign_len, pkey) <= 0)
104                 goto sign_err;
105         ret = true;
106 sign_err:
107         EVP_MD_CTX_free(ctx);
108 ctx_new_err:
109         EVP_PKEY_free(pkey);
110         return ret;
111 }
112 int main()
113 {
114         //生成长度为 2048 的密钥
115         genrsa(2048);
116         const char* msg = "Hello World!";
117         const unsigned int msg_len = strlen(msg);
118         //存储签名
119         uint8_t sign[256] = { 0 };
120         unsigned int sign_len = 0;
121         //签名
122         if (!gensign((uint8_t*)msg, msg_len, sign, &sign_len))
123         {
124                 printf("签名失败\n");
125                 return 0;
126         }
127         //验证签名
128         if ( verify ((uint8_t*)msg, msg_len, sign, sign_len))
```

```
129                 printf ("验证成功\n");
130         else
131                 printf ("验证失败\n");
132         return 0;
133 }
```