* 1. OpenSSL安装

一般情况下，Ubuntu系统存在预装的OpenSSL，可使用openssl version查看版本号。

* + 1. 使用APT软件包管理工具安装

您可使用以下命令安装或更新OpenSSL·，需要保证处于root账户下或使用sudo命令提权

[sudo] apt install openssl

若需使用OpenSSL开发组件，则需要额外安装libssl-dev包

[sudo] apt install libssl-dev

* + 1. 使用源码编译安装

1. 卸载预装版本

在安装新版本前，您可根据需要，使用如下命令卸载预装版本。

[sudo] apt autoremove openssl

1. 下载源码包，下载链接可从官网（openssl.org）获取

wget https://www.openssl.org/source/openssl-<版本号>.tar.gz

1. 解压并进入到该目录

tar -zxvf openssl-<版本号>.tar.gz  
cd openssl-<版本号>

1. 使用config生成Makefile

./config

若想修改安装地址，可使用

./config --prefix=/opt/openssl --openssldir=/usr/local/ssl

--prefix 指定安装的根目录，默认地址为/usr/local；

--openssldir指定配置文件地址，以及证书和密钥的默认存放目录，默认地址为/usr/local/ssl。

1. 编译并安装

make  
[sudo] make install

1. 解决动态链接库查找问题

安装完成后，可能会出现动态连接库相关错误。这是因为openssl默认放置动态连接库的地址并非系统搜寻目录，可通过以下命令添加。

[sudo] ldconfig /usr/local/lib #openssl 1.1.\*  
[sudo] ldconfig /usr/local/lib64 #openssl 3.0.\*

* 1. 对称加密及工作模式
     1. 使用OpenSSL命令加解密文件

openssl enc对称密码命令可使用各种块和流密码对数据进行加密或解密，通过openssl enc -help可查看全部选项。

1. 文件加密

使用aes-128-cbc对message.txt文件进行加密并使用base64编码，输出到ciphertext.txt。假设128位密钥为a3171d177d1ce97ebc644ea3ff826b4e，初始向量为8bc65f2f883f95eea10b6f940cc805f6。

openssl enc -e -aes-128-cbc -in message.txt -out ciphertext.txt -K a3171d177d1ce97ebc644ea3ff826b4e -iv 8bc65f2f883f95eea10b6f940cc805f6 -base64

1. 文件解密

对ciphertext.txt进行base64解码并解密，结果输出到plaintext.txt。

openssl enc -d -aes-128-cbc -in ciphertext.txt -out plaintext.txt -K a3171d177d1ce97ebc644ea3ff826b4e -iv 8bc65f2f883f95eea10b6f940cc805f6 -base64

以下为部分语法解释。

语法：enc [options]

|  |  |
| --- | --- |
| 选项 | 作用 |
| -e | 加密 |
| -d | 解密 |
| -in infile | 指定输入文件 |
| -out outfile | 指定输出文件 |
| -base64 | 加密时在加密后进行base64编码，或在解密时首先对密文进行解码 |
| -K val | 指定密钥 |
| -iv val | 提供初始向量 |

* + 1. 加解密程序

1. 编写程序文件aes-128-cbc.cpp

#include <stdio.h>

#include <string.h>

#include <openssl/evp.h>

// aes-128-cbc加密函数

bool aes\_128\_cbc\_encrypt(const uint8\_t \*in, int in\_len, uint8\_t \*out, int \*out\_len, const uint8\_t \*key, const uint8\_t \*iv)

{

// 创建上下文

EVP\_CIPHER\_CTX \*ctx = EVP\_CIPHER\_CTX\_new();

if (!ctx)

return false;

bool ret = false;

// 初始化加密模块

if (EVP\_EncryptInit\_ex(ctx, EVP\_aes\_128\_cbc(), NULL, key, iv) <= 0)

goto err;

int update\_len;

// 向缓冲区写入数据，同时将以对齐的数据加密并返回

if (EVP\_EncryptUpdate(ctx, out, &update\_len, in, in\_len) <= 0)

goto err;

int final\_len;

// 结束加密，填充并返回最后的加密数据

if (EVP\_EncryptFinal\_ex(ctx, out + update\_len, &final\_len) <= 0)

goto err;

\*out\_len = update\_len + final\_len;

ret = true;

err:

EVP\_CIPHER\_CTX\_free(ctx);

return ret;

}

// aes-128-cbc解密函数，结构与加密相似

bool aes\_128\_cbc\_decrypt(const uint8\_t \*in, int in\_len, uint8\_t \*out, int \*out\_len, const uint8\_t \*key, const uint8\_t \*iv)

{

EVP\_CIPHER\_CTX \*ctx = EVP\_CIPHER\_CTX\_new();

if (!ctx)

return false;

bool ret = false;

if (EVP\_DecryptInit\_ex(ctx, EVP\_aes\_128\_cbc(), NULL, key, iv) <= 0)

goto err;

int update\_len;

if (EVP\_DecryptUpdate(ctx, out, &update\_len, in, in\_len) <= 0)

goto err;

int final\_len;

if (EVP\_DecryptFinal\_ex(ctx, out + update\_len, &final\_len) <=0)

goto err;

\*out\_len = update\_len + final\_len;

ret = true;

err:

EVP\_CIPHER\_CTX\_free(ctx);

return ret;

}

int main()

{

// 密钥

uint8\_t key[] = {35, 31, 71, 44, 34, 42, 76, 16, 86, 27, 93, 59, 26, 62, 4, 19};

// 初始化向量

uint8\_t iv[] = {91, 66, 51, 17, 14, 40, 65, 38, 4, 60, 89, 44, 87, 63, 67, 32};

const char \*msg = "Hello World!";

const int msg\_len = strlen(msg);

// 存储密文

uint8\_t ciphertext[32] = {0};

int ciphertext\_len;

// 加密

aes\_128\_cbc\_encrypt((uint8\_t \*)msg, msg\_len, ciphertext, &ciphertext\_len, (uint8\_t \*)key, (uint8\_t \*)iv);

// 存储解密后的明文

uint8\_t plaintext[32] = {0};

int plaintext\_len;

// 解密

aes\_128\_cbc\_decrypt((uint8\_t \*)ciphertext, ciphertext\_len, plaintext, &plaintext\_len, (uint8\_t \*)key, (uint8\_t \*)iv);

// 输出解密后的内容

printf("%s\n", plaintext);

return 0;

}

1. 编译并运行

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

* 1. 数字签名及验证
     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

1. 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

1. 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

1. 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。

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 |

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 |

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 |

* + 1. 数字签名程序

1. 编写程序文件signature.cpp

#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);

fclose(pubf);

if (rt <= 0)

goto err;

ret = true;

err:

EVP\_PKEY\_CTX\_free(ctx);

return ret;

}

// 生成数据签名

bool gensign(const uint8\_t \*in, unsigned int in\_len, uint8\_t \*out, unsigned int \*out\_len)

{

FILE \*prif = fopen(PRIVATE\_KEY\_FILE\_NAME, "r");

if (!prif)

return false;

// 读取私钥

EVP\_PKEY \*pkey = PEM\_read\_PrivateKey(prif, NULL, NULL, NULL);

fclose(prif);

if (!pkey)

return false;

bool ret = false;

EVP\_MD\_CTX \*ctx = EVP\_MD\_CTX\_new();

if (!ctx)

goto ctx\_new\_err;

// 初始化

if (EVP\_SignInit(ctx, EVP\_sha256()) <= 0)

goto sign\_err;

// 输入消息，计算摘要

if (EVP\_SignUpdate(ctx, in, in\_len) <= 0)

goto sign\_err;

// 生成签名

if (EVP\_SignFinal(ctx, out, out\_len, pkey) <= 0)

goto sign\_err;

ret = true;

sign\_err:

EVP\_MD\_CTX\_free(ctx);

ctx\_new\_err:

EVP\_PKEY\_free(pkey);

return ret;

}

// 使用公钥验证数字签名，结构与签名相似

bool verify(const uint8\_t \*msg, unsigned int msg\_len, const uint8\_t \*sign, unsigned int sign\_len)

{

FILE \*pubf = fopen(PUBLIC\_KEY\_FILE\_NAME, "r");

if (!pubf)

return false;

// 读取公钥

EVP\_PKEY \*pkey = PEM\_read\_PUBKEY(pubf, NULL, NULL, NULL);

fclose(pubf);

if (!pkey)

return false;

bool ret = false;

EVP\_MD\_CTX \*ctx = EVP\_MD\_CTX\_new();

if (!ctx)

goto ctx\_new\_err;

// 初始化

if (EVP\_VerifyInit(ctx, EVP\_sha256()) <= 0)

goto sign\_err;

// 输入消息，计算摘要

if (EVP\_VerifyUpdate(ctx, msg, msg\_len) <= 0)

goto sign\_err;

// 验证签名

if (EVP\_VerifyFinal(ctx, sign, sign\_len, pkey) <= 0)

goto sign\_err;

ret = true;

sign\_err:

EVP\_MD\_CTX\_free(ctx);

ctx\_new\_err:

EVP\_PKEY\_free(pkey);

return ret;

}

int main()

{

// 生成长度为2048的密钥

genrsa(2048);

const char \*msg = "Hello World!";

const unsigned int msg\_len = strlen(msg);

// 存储签名

uint8\_t sign[256] = {0};

unsigned int sign\_len = 0;

// 签名

if (!gensign((uint8\_t \*)msg, msg\_len, sign, &sign\_len))

{

printf("签名失败\n");

return 0;

}

// 验证签名

if (verify((uint8\_t \*)msg, msg\_len, sign, sign\_len))

printf("验证成功\n");

else

printf("验证失败\n");

return 0;

}

1. 编译并运行

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