RSA 公钥加密与签名实验

1 概述

RSA （Rivest–Shamir–Adleman）是最早的公钥密码系统之一，被广泛用于安全通信。RSA 算法首先生成两个大的随机素数，然后使用它们生成公、私钥对。公私钥对可用于执行加密、解密、数字签名的生成与验证。RSA 算法是建立在数论之上的，可以利用已有的库函数来实现。

本实验的学习目标是让学生获得 RSA 算法的动手经验。通过课堂学习，学生应该已经了解 RSA 算法的理论部分，知道在数学上如何生成公钥、私钥以及如何执行加密、解密和签名生成、验证。通过使用具体数字做一遍 RSA 算法的每一个步骤，学生可以应用课堂上学习到的理论，加深对 RSA 的理解。学生将使用 C 程序语言实现 RSA 算法。该实验室涵盖以下与安全性有关的主题：

- 公钥密码学
- RSA 算法和密钥生成
- 大数运算
- RSA 加密与解密
- 数字签名
- X.509 证书

阅读材料 有关公钥密码学的详细信息，请阅读以下内容：

- SEED 书, *Computer & Internet Security: A Hands-on Approach*, 3rd Edition, by Wenliang Du. 详情请见 `https://www.handsonsecurity.net`.

实验环境 本实验在我们预先构建好的 Ubuntu 20.04 VM（可以从我们的 SEED 网站当中下载）当中测试可行。

致谢 该实验是在 Syracuse 大学电气工程与计算机科学系的研究生 Shatadiya Saha 的帮助下开发的。

2 实验背景

RSA 算法涉及大整数的计算。这些计算不能直接使用简单的算术运算符，因为这些运算符只能对原始数据类型（例如 32 位整数和 64 位长整数类型）进行运算。RSA 算法中涉及的数字通常长于 512 位。对于两个 32 位整数 `a` 和 `b` ，我们只需要在程序中使用 `a * b`。但是，如果数很大，我们就不能再这样做了，我们需要使用算法来计算其乘积。

有几个函数库可以实现对任意大小的整数执行算术运算。在本实验中，我们将使用 `openssl` 提供的 Big Number 库。要使用此库，我们将每个大数定义为 `BIGNUM` 类型，然后使用该库提供的 API 进行各种操作，例如加法，乘法，乘幂，模运算等。

2.1 BIGNUM APIs

所有的 BIGNUM API 都可以从 `https://linux.die.net/man/3/bn` 中找到。在下文中，我们介绍一下此实验所需的一些 API 。

- 有些库函数需要临时变量，它们的内存需要动态分配。但如果每次调用时都去动态分配内存的代价很高。常用的方法是使用 `BN_CTX` 结构来保存库函数使用的临时变量。我们先创建这样一个结构，然后传递给需要使用它的函数。

```
BN_CTX *ctx = BN_CTX_new()
```

- 初始化 BIGNUM 变量。

```
BIGNUM *a = BN_new()
```

- 给 BIGNUM 变量赋值的方法有很多。

```
// 用 10 进制数字字符串赋值
BN_dec2bn(&a, "12345678901112231223");

// 用 16 进制数字字符串赋值
BN_hex2bn(&a, "2A3B4C55FF77889AED3F");

// 生成 128 位的随机数
BN_rand(a, 128, 0, 0);

// 生成 128 位的随机质数
BN_generate_prime_ex(a, 128, 1, NULL, NULL, NULL);
```

- 输出大整数。

```
void printBN(char *msg, BIGNUM * a)
{
   // 把 BIGNUM 转换成 10 进制数字字符串
   char * number_str = BN_bn2dec(a);

   // 打印数字字符串
   printf("%s %s\n", msg, number_str);

   // 释放分配的内存
   OPENSSL_free(number_str);
}
```

- 计算 res = $a - b$ 和 res = $a + b$。

```
BN_sub(res, a, b);
BN_add(res, a, b);
```

- 计算 res = $a * b$。注意这个 API 需要 BN_CTX。

```
BN_mul(res, a, b, ctx)
```

- 计算 res = $a * b$ mod n：

```
BN_mod_mul(res, a, b, n, ctx)
```

- 计算 res = a^c mod n：

```
BN_mod_exp(res, a, c, n, ctx)
```

- 计算模逆元（modular inverse），也就是给定 a，找到一个 b，使得 $a * b$ mod n = 1，b 被称为 a 模 n 的逆元。

```
BN_mod_inverse(b, a, n, ctx);
```

2.2 一个完整的例子

接下来我们展示一个完整的例子。在这个例子中，我们初始化三个 BIGNUM 变量 a、b、和 n，然后计算 $a * b$ 和 (a^b mod n).

```
/* bn_sample.c */
#include <stdio.h>
#include <openssl/bn.h>

#define NBITS 256

void printBN(char *msg, BIGNUM * a)
{
   /* Use BN_bn2hex(a) for hex string
    * Use BN_bn2dec(a) for decimal string */
   char * number_str = BN_bn2hex(a);
   printf("%s %s\n", msg, number_str);
   OPENSSL_free(number_str);
}

int main ()
{
  BN_CTX *ctx = BN_CTX_new();
```

```
  BIGNUM *a = BN_new();
  BIGNUM *b = BN_new();
  BIGNUM *n = BN_new();
  BIGNUM *res = BN_new();


  // Initialize a, b, n
  BN_generate_prime_ex(a, NBITS, 1, NULL, NULL, NULL);
  BN_dec2bn(&b, "273489463796838501848592769467194369268");
  BN_rand(n, NBITS, 0, 0);

  // res = a*b
  BN_mul(res, a, b, ctx);
  printBN("a * b = ", res);

  // res = a^b mod n
  BN_mod_exp(res, a, b, n, ctx);
  printBN("a^c mod n = ", res);

  return 0;
}
```

编译 我们可以使用下面的命令来编译 `bn_sample.c` （`-` 后面的字符是字母 ℓ，不是数字 1。它告诉编译器使用 `crypto` 库）。

```
$ gcc bn_sample.c -lcrypto
```

3 实验任务

为避免错误，请不要手动输入实验任务中使用的数字，而应复制、粘贴此 PDF 文件中的数字。

提交要求。 在报告中，你应该描述你的步骤，并附上你的代码和运行结果。

3.1 任务 1: 生成私钥

`p`、`q` 和 `e` 为三个素数，`n = p*q`。我们使用 `(e, n)` 作为公钥。请计算私钥 `d`。`p`、`q` 和 `e` 的十六进制如下。注意：尽管本任务中的 `p` 和 `q` 非常的大，但它们还没有大到足以保证安全。为了简单，我们有意让这些数比较小。在实践中，这些数应当至少有 512 位，而这里使用的只有 128 位。

```
p = F7E75FDC469067FFDC4E847C51F452DF
q = E85CED54AF57E53E092113E62F436F4F
e = 0D88C3
```

3.2 任务 2: 加密一条消息

请将 (e, n) 作为公钥，加密消息 "A top secret!"（不包括引号）。我们需要将 ACSII 字符串转换为十六进制字符串，然后用 BN_hex2bn() 将十六进制字符串转换为 BIGNUM。下面的 python 命令可以用于将原始的 ASCII 字符串转换为十六进制字符串。

```
$ python  -c 'print("A top secret!".encode("hex"))'
4120746f702073656372657421
```

公钥如下所示（十六进制）。这里还提供了私钥 d 来帮助你验证你的加密结果。

```
n = DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5
e = 010001 (这个 16 进制数的值是 65537)
M = A top secret!
d = 74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D
```

3.3 任务 3: 解密一条消息

本任务中使用的公私钥对与任务 2 中的相同。请解密下面的密文 C，然后转换回 ASCII 字符串。

```
C = 8C0F971DF2F3672B28811407E2DABBE1DA0FEBBBDFC7DCB67396567EA1E2493F
```

你可以使用下面的 python 命令将一个十六进制字符串转换为 ASCII 字符串。

```
$ python  -c 'print("4120746f702073656372657421".decode("hex"))'
A top secret!
```

3.4 任务 4: 数字签名

本任务中使用的公私钥对与任务 2 中的相同。请为下面的消息生成一个数字签名（直接为这条消息签名，而不是它的 Hash ）:

```
M = I owe you $2000.
```

稍微改动一下 M，例如将 $2000 改为 $3000，然后为修改后的消息签名。比较两个签名，并描述你的发现。

3.5 任务 5: 验证数字签名

Bob 从 Alice 那里收到一条消息 M 和她对消息的数字签名 S。我们知道 Alice 的公钥是 (e, n)。请验证这个签名是否确实是 Alice 生成的。公钥和签名（十六进制）如下所示:

```
M = Launch a missile.
S = 643D6F34902D9C7EC90CB0B2BCA36C47FA37165C0005CAB026C0542CBDB6802F
e = 010001
n = AE1CD4DC432798D933779FBD46C6E1247F0CF1233595113AA51B450F18116115
```

假设上面的签名被破坏了，例如签名的最后一个字节从 2F 变成了 3F，也就是说只改变了一位。请重复这个实验，描述验证过程中发生了什么。

3.6 任务 6: 手动验证一张 X.509 证书

在此任务中，我们将使用程序手动验证 X.509 证书。X.509 包含有关公钥的数据以及颁发者对该数据的签名。我们将从 Web 服务器上下载真实的 X.509 证书，获取其颁发者的公钥，然后使用该公钥来验证证书上的签名。

第 1 步: 从真实的 Web 服务器上下载一张证书。 在这份文档中，我们使用 www.example.org 服务器。学生应当选择一个其他的 Web 服务器，获取与文档中不同的证书。(注意 www.example.com 和 www.example.org 的证书是一样的)。我们可以用浏览器或者使用下面的命令下载证书:

```
$ openssl s_client -connect www.example.org:443 -showcerts

Certificate chain
 0 s:/C=US/ST=California/L=Los Angeles/O=Internet Corporation for Assigned
     Names and Numbers/OU=Technology/CN=www.example.org
   i:/C=US/O=DigiCert Inc/OU=www.digicert.com/CN=DigiCert SHA2 High Assurance
     Server CA
   -----BEGIN CERTIFICATE-----
   MIIF8jCCBNqgAwIBAgIQDmTF+8I2reFLFyrrQceMsDANBgkqhkiG9w0BAQsFADBw
   MQswCQYDVQQGEwJVUzEVMBMGA1UEChMMRGlnaUNlcnQgSW5jMRkwFwYDVQQLExB3
   ......
   wDSiIIWIWJiJGbEeIOOTIFwEVWTOnbNl/faPXpk5IRXicapqiII=
   -----END CERTIFICATE-----
 1 s:/C=US/O=DigiCert Inc/OU=www.digicert.com/CN=DigiCert SHA2 High
     Assurance Server CA
   i:/C=US/O=DigiCert Inc/OU=www.digicert.com/CN=DigiCert High Assurance
     EV Root CA
   -----BEGIN CERTIFICATE-----
   MIIEsTCCA5mgAwIBAgIQBOHnpNxc8vNtwCtCuF0VnzANBgkqhkiG9w0BAQsFADBs
   MQswCQYDVQQGEwJVUzEVMBMGA1UEChMMRGlnaUNlcnQgSW5jMRkwFwYDVQQLExB3
   ......
   cPUeybQ=
   -----END CERTIFICATE-----
```

命令的结果包含两张证书。证书的 subject（s: 开头的一项）是 www.example.org，也就是说这张证书是 www.example.org 的证书。颁发者 issuer（i: 开头）提供了颁发者的信息。第二张证书的 subject 与第一张证书的颁发者相同，也就是说第二章证书是一个中间 CA 的。在本任务中，我们使用 CA 的证书验证服务器的证书。

如果上面的命令只返回了一张证书，就说明你获得的证书是根 CA 签发的。根 CA 的证书可以从我们的虚拟机中安装的 Firefox 浏览器里获得。点击 Edit → Preferences → Privacy 然后点击 Security → View Certificates。搜索颁发者的名字，然后下载它的证书。

复制每张证书（在 "Begin CERTIFICATE" 和 "END CERTIFICATE" 的两行中间的文本，包括这两行）到一个文件中。第一个文件称为 c0.pem ，第二个称为 c1.pem。

第 2 步：从颁发者的证书中（e，n）提取公钥。 Openssl 提供了从 X.509 证书中提取特定属性的命令。我们可以使用 -modulus 参数来提取 n 的值。没有特定参数的指令可以用来提取 e，但是我们可以输出所有的域，从中很容易就能找到 e 的值.

```
For modulus (n):
$ openssl x509 -in c1.pem -noout -modulus

打印所有的域，从中找到 e:
$ openssl x509 -in c1.pem -text -noout
```

第 3 步：从服务器的证书中提取签名。 openssl 没有具体命令来提取签名，但是我们可以输出所有的域，然后复制签名到一个文件中（注意：如果证书使用的签名算法不是基于 RSA 的，你可以找一张其他的证书）。

```
$ openssl x509 -in c0.pem -text -noout
...
Signature Algorithm: sha256WithRSAEncryption
  84:a8:9a:11:a7:d8:bd:0b:26:7e:52:24:7b:b2:55:9d:ea:30:
  89:51:08:87:6f:a9:ed:10:ea:5b:3e:0b:c7:2d:47:04:4e:dd:
  ......
  5c:04:55:64:ce:9d:b3:65:fd:f6:8f:5e:99:39:21:15:e2:71:
  aa:6a:88:82
```

我们需要从数据中删除空格和冒号，这样就能获得可以输入到我们的程序的十六进制字符串。下面的命令可以实现这个目的。tr 命令是一个 Linux 用来处理字符串的工具。在本例中，-d 选项用来从数据中删除 ":" 和 "space" (空格)。

```
$ cat signature | tr -d '[:space:]:'
84a89a11a7d8bd0b267e52247bb2559dea30895108876fa9ed10ea5b3e0bc7
......
5c045564ce9db365fdf68f5e99392115e271aa6a8882
```

第 4 步：提取服务器的证书主体。 证书颁发机构（CA）为一张服务器证书生成签名时，首先需要计算证书的 Hash ，然后对 Hash 签名。为了验证证书，我们也需要从证书中生成一个 Hash。由于 Hash 是在计算签名之前生成的，我们需要排除证书的签名块，然后再计算 Hash。如果没有对证书格式很好的理解，找到证书中用于生成 Hash 的部分是很有挑战性的。

X.509 证书使用 ASN.1 (Abstract Syntax Notation One) 标准编码，所以如果我们能解析 ASN.1 结构，我们就能很容易从整数中提取任何一个域。Openssl 有一个用于解析 ASN.1 结构的 asn1parse 命令，这里我们用来解析 X.509 证书。

```
$ openssl asn1parse -i -in c0.pem
    0:d=0  hl=4 l=1522 cons: SEQUENCE
    4:d=1  hl=4 l=1242 cons:  SEQUENCE          ❶
    8:d=2  hl=2 l=   3 cons:   cont [ 0 ]
   10:d=3  hl=2 l=   1 prim:    INTEGER           :02
   13:d=2  hl=2 l=  16 prim:   INTEGER           :0E64C5FBC236ADE14B172AEB41C78CB0
   ... ...
 1236:d=4  hl=2 l=  12 cons:     SEQUENCE
 1238:d=5  hl=2 l=   3 prim:      OBJECT            :X509v3 Basic Constraints
 1243:d=5  hl=2 l=   1 prim:      BOOLEAN           :255
 1246:d=5  hl=2 l=   2 prim:      OCTET STRING      [HEX DUMP]:3000
 1250:d=1  hl=2 l=  13 cons:  SEQUENCE          ❷
 1252:d=2  hl=2 l=   9 prim:   OBJECT            :sha256WithRSAEncryption
 1263:d=2  hl=2 l=   0 prim:   NULL
 1265:d=1  hl=4 l= 257 prim:  BIT STRING
```

从 ❶ 开始的部分是用于生成 Hash 的证书主体，从 ❷ 开始的部分是签名块。每行开头的数字是它们的偏移（Offset）。在本例中，证书主体的偏移是 4 到 1249，而签名块是从 1250 到文件的末尾。对于 X.509 证书，开头的偏移总是一样的（也就是 4），但结尾的位置依赖于证书内容的长度。我们可以使用 `-strparse` 选项获取起始偏移为 4 的域，也就是证书不带签名的主体部分。

```
$ openssl asn1parse -i -in c0.pem -strparse 4 -out c0_body.bin -noout
```

获得了证书的主体之后，我们就可以用下面的命令计算它的 Hash：

```
$ sha256sum c0_body.bin
```

第 5 步: 验证签名 现在我们有了全部的信息，包括 CA 的公钥、CA 的签名以及服务器证书的主体。我们可以执行自己的程序来验证签名是否有效。Openssl 提供了一个命令来为我们验证证书，但是这里要求学生使用他们自己的程序来实现。否则，这个任务给 0 分。

4 递交

你需要提交一份带有截图的详细实验报告来描述你所做的工作和你观察到的现象。你还需要对一些有趣或令人惊讶的观察结果进行解释。请同时列出重要的代码段并附上解释。只是简单地附上代码不加以解释不会获得学分。