Secure Message Software

Xi Liu, Zheyuan Gao, Zicheng Hao, Xavier Huang

Overview

Target

Build an encrypted real-time chat application mainly using RSA encryption method via web sockets.

Design Security

Web sockets with an encryption layer

End-to-end encryption

Implementing Security

Programming Languages: HTML/CSS, Javascript, Python

Implementing Cryptography: RSA

Testing Security

OWASP testing guide

Cryptographic failure:

Sending sensitive data in clear text, for example, using HTTP instead of HTTPS. HTTP is the protocol used to access the web, while HTTPS is the secure version of HTTP. Others can read everything you send over HTTP, but not HTTPS.

Relying on a weak cryptographic algorithm. One old cryptographic algorithm is to shift each letter by one. For instance, “TRY HACK ME” becomes “USZ IBDL NF.” This cryptographic algorithm is trivial to break.

Using default or weak keys for cryptographic functions. It won’t be challenging to break the encryption that used 1234 as the secret key.

Foundation Application

Message Application

- Using Python and socket programming
- Consist of two parts: Client and Server
- Features:
  - Different client can join different discussion group
  - Client can only see messages in the groups he joined
  - Client can retrieve history messages posted previously in the group
  - Client can enter commands to see the group users and group number

Encryption Feature of the Message Application

- The encryption algorithm can be embedded with the message application
- The message will be encrypted once been sent
- The message during transmitting will remain encrypted
- Message will be decrypted when the group user receives the message
- Security measures ensure the secure transfer of the message

Demonstration

Implementation

overview of RSA encryption

1. find prime numbers p, q
2. n := p * q
3. phi(n) = (p - 1) * (q - 1), # numbers < n that share no factor with n
4. choose encryption e s.t. 1 < e < phi(n), e is coprime with n, phi(n)
5. choose decryption d s.t. de = 1   mod phi(n)
6. c = (msg ^ e)   mod n
7. m = (c ^ d) = (msg ^ {de})   mod n

euclid gcd

kth step: find a quotient q_k, remainder r_k

$r_{k - 2} = q_k * r_{k - 1} + r_k$, $r_k = r_{k - 2} \% r_{k - 1}$

base step: $k = 0$, $r_{k - 2} = r_{-2} = a$, $r_{k - 1} = r_{-1} = b$

gcd($a = 1071$, $b = 462$)

step k:	equation	quotient, remainder
0	$r_{-2} = 1071 = q_0 * (r_{-1} = 462) + r_0$	$q_0 = 2$, $r_0 = 147$
1	$r_{-1} = 462 = q_1 * (r_0 = 147) + r_1$	$q_1 = 3$, $r_1 = 21$
2	$r_0 = 147 = q_2 * (r_1 = 21) + r_2$	$q_2 = 7$, $r_2 = 0$

```
int euclid_gcd(int a, int b)
{
    while(b)
    {
        int t = b;
        b = a % b; /* r_k = r_{k - 2} % r_{k - 1} */
        a = t;
    }
    return a > -a ? a : -a;
}
```

p = 1013; q = 1019; n = p * q; e = 2;

phi(i): number of integers less than i that do not share a common factor with i. since prime numbers have no factors greater than 1, if i is prime number, the i - 1 positive integers do not share a common factor with i

```
if(isprime(i))
    phi(i) = i - 1
```

2 justifications

1. let A, B, C be sets containing integers coprime to and < p, q, pq. |A| = phi(pq). there is bijection A x B and C by Chinese remainder theorem, phi is multiplicative function, phi(pq) = phi(p) * phi(q) = (p - 1) * (q - 1)

2. p, q are prime factors of pq, integer k is coprime to pq iff it is not a

multiple of p or q. in 0 < k < pq, there are p - 1 multiples of q

q - 1 multiples of p, so (p - 1) + (q - 1) integers share a factor with pq

phi(pq) = (#integers < pq) - (#integers not coprime)

= (pq - 1) - ((p - 1) + (q - 1))

= pq - 1 - p - q + 2

= pq - p - q + 1

= (p - 1)(q - 1)

phi = (p - 1) * (q - 1);

* RSA used for message encryption, digital signature, each message represented as a row in database

* encrypted video call, multi person collaboration session

* ElGamal scheme used for file or disk encryption and signature, which require numbers bigger than 32 or 64 bits int

* cellular automata graphics encryption

* secure database connections

* application make use of secure sockets layer when send and receive messages through the network

```c
#include <stdio.h>

#include <sqlite3.h>

int callback(void * data, int argc, char ** argv, char ** header){

    for(int i = 0; i < argc; ++i)

       printf("%s : %s\n", header[i], argv[i]);

    printf("\n");

    return 0;

}

int main(){

       char * er;

       sqlite3 * db;

       int r1 = sqlite3_open("msgdb", &db);

       if(r1) printf("open\n");

       int r2 = sqlite3_exec(db, "select * from tb1;", callback, 0, &er);

       if(r2) printf("exec\n");

       sqlite3_close(db);

}
```

```sql
create table tb1
(
     one text,
     two int
);
insert into tb1 values('hi', 10);
insert into tb1 values('bye', 20);
select * from tb1;
```

```
one : hi
two : 10

one : bye
two : 20
```

```c
#include <stdio.h>

#include <limits.h>

typedef long long _long;

int main()

{

    printf("llong_max = %lld\n", LLONG_MAX);

    printf("((_long)1 << 63) - 1 = %lld\n", ((_long)1 << 63) - 1); /* 2 ^ 63 - 1 =
9223372036854775807 */

    printf("int_max = %u\n", INT_MAX);

    printf("((unsigned)1 << 31) - 1 = %u\n", ((unsigned)1 << 31) - 1); /* 2 ^ 31 -
1 = 2147483647 */

}
```

```c
#include <stdio.h>

#include <gmp.h>

int main()

{

  mpz_t x, y, result;

  mpz_init_set_str(x, "7612058254738945", 10); mpz_init_set_str(y, "9263591128439081", 10); mpz_init(result);

  mpz_mul(result, x, y);

  gmp_printf("    %Zd\n"

             "*\n"

             "    %Zd\n"

             "--------------------\n"

             "%Zd\n", x, y, result);

  mpz_clear(x); mpz_clear(y); mpz_clear(result);

}
```

```
    7612058254738945
*
    9263591128439081
--------------------
70514995317761165008628990709545
```

```c
typedef struct
{
    char * digits;  /* char array that represent the number */
    int signbit;    /* 1 if positive, -1 if negative */
    int lastdigit; /* index of high-order digit, digits.cur_sz - 1 */
} mpz_t; /* multiple precision int */
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <string>
#include <gmp.h>
#include <gmpxx.h>
#define file FILE
#define int_min (1 << 31)
#define int_max (((unsigned)1 << 31) - 1)
typedef unsigned long int _long;
using namespace std;

/* convert a integer to string */
char * int_str(int a)
{
    int n_digit = log10(a) + 1,
    radix = 10;
    char * res = (char *)malloc(n_digit * sizeof(char));
    sprintf(res, "%d", a);
    return res;
}
```

```
/* convert a file to binary string */
char * file_str(file * fp)
{
    string a; /* file content in original char */
    char c;
    while((c = fgetc(fp)) != -1)
        a += c;
    fclose(fp);

    int a_sz = a.size(),
    res_sz = 8 * a_sz + 1;
    char * res = (char *)malloc(res_sz * sizeof(char)); /* each char is 1 byte, represented by 8 bits */
    for(int i = 0; i < a_sz; ++i)
    {
        char cur = a[i];
        for(int j = 0; j < 8; ++j)
        {
            res[8 * i + 7 - j] = '0' + (cur & 1);
            cur >>= 1;
        }
    }
    res[res_sz - 1] = '\0';
    return res;
}
```

```c
/* pk = <q, g, g ^ x>
   sk = <q, g, x> */
struct elgamal_key
{
    mpz_t q, g, x, gx;
};

void elgamal_key_gen(elgamal_key * res)
{
    #define q res->q
    #define g res->g
    #define x res->x
    #define gx res->gx

    time_t t;
    srand((unsigned)time(&t));
    mpz_init_set_str(q, int_str(rand() % (1 << 4)), 10);

    gmp_randstate_t state;
    gmp_randinit_default(state);

    mpz_init(g);
    mpz_urandomm(g, state, q);

    mpz_init(x);
    mpz_urandomm(x, state, q); /* rand int in [0, q - 1] */

    mpz_init(gx);
    mpz_pow_ui(gx, g, mpz_get_ui(x));
    gmp_printf("gx = %Zd\n", gx);

    #undef q
    #undef g
    #undef x
    #undef gx
}
```

```c
void elgamal_encrypt
(
    mpz_t c1,
    mpz_t c2,
    mpz_t q,
    mpz_t g,
    mpz_t gx,
    mpz_t m,
    bool write
)
{
    mpz_init(c1);
    mpz_init(c2);
    gmp_randstate_t state;
    gmp_randinit_default(state);

    mpz_t _y;
    mpz_urandomm(_y, state, q);
    mpz_cdiv_r_ui(_y, _y, int_max); /* avoid overflow: _y %= int_max */
    _long y = mpz_get_ui(_y);
    mpz_pow_ui(c1, g, y); /* c1 = g ^ y */
    mpz_pow_ui(c2, gx, y); /* c2 = (g ^ x) ^ y */
    mpz_mul(c2, c2, m); /* c2 *= m */

    if(write)
    {
        file * fp1, * fp2;
        fp1 = fopen("./c1.txt", "w+"); fp2 = fopen("./c2.txt", "w+");
        char * s1 = mpz_get_str(0, 10, c1),
        * s2 = mpz_get_str(0, 10, c2); /* (char * str, int base, const mpz_t op): str is null, allocate by library */
        fprintf(fp1, s1);
        fprintf(fp2, s2);
    }
}
```

```
s := (g ^ x) ^ y

decrypt:
c2 * s^{-1} = (m * s) * s^{-1} = m
```

```c
int main()
{
    file * fp = fopen("t.txt", "r");
    char * file1 = file_str(fp);
    printf("%s\n", file1);
    elgamal_key * res = (elgamal_key *)malloc(sizeof(elgamal_key));
    elgamal_key_gen(res);
    gmp_printf("gx = %Zd\n", res->gx);

    mpz_t c1, c2, m1;
    mpz_init_set_str(m1, file1, 10);
    elgamal_encrypt(c1, c2, res->q, res->g, res->gx, m1, 1);
}
```

encryption

Converting image to 2d matrix (M) and modifying each pixel value by using key function

M1 = Key(M), where M1 is the encrypted 2D matrix

Each pixel values of M matrix is modified by using the Key function h1(M) depending on whether the value of the pixel sent is odd or even

The modified pixel is then received and stored in M1.

Matrix M1 is converted to image and saved as an encrypted image.

decryption

Original pixels are retrieved by using the reverse technique. Encrypted image is taken as an input

M = Key(M1) is executed, where M is the decrypted 2D matrix.

Each pixel values of M1 matrix is modified by using the Key function h1(M) using its corresponding odd/even rule

The modified pixel is then received and stored in M1.

The M matrix is converted to image and saved as decrypted image.

key function

Here a 2D matrix is taken as an input and this key() function is called during encryption as well as decryption which modifies the input pixel value using cellular automata rule vector. The input pixel is converted into binary number of 8 blocks. Distinct cellular automata rules are applied per block. If input value is even, rule 90 and 30 are applied alternatively. If input value is odd, rule 90 is applied. The rule configuration is run till the initial input block repeats itself.

one dimensional cellular automata

assume an array of cells with an initial distribution of live and dead cells, and imaginary cells off the end of the array having fixed values

cells in the next generation of the array are calculated based on the value of the cell and its left and right nearest neighbours in the current generation

If, in the following table, a live cell is represented by 1 and a dead cell by 0 then to generate the value of the cell at a particular index in the array of cellular values you use the following table:

```
000 -> 0  #

001 -> 0  #

010 -> 0  # dies without enough neighbours

011 -> 1  # needs one neighbour to survive

100 -> 0  #

101 -> 1  # two neighbours giving birth

110 -> 1  # needs one neighbour to survive

111 -> 0  # starved to death
```

```c
#include <stdio.h>

#include <string.h>

char * map = "___#_##_";

#define idx(i) (cell[i] != '_')

int evolve(char * cell, char * buf, int n)

{

      int i, dif = 0;

      for(i = 0; i < n; ++i)

      {/* use left, self, right as binary number bits for map index */

            buf[i] = map[(idx(i - 1) << 2) + (idx(i) << 1) + idx(i + 1)];

            dif += (buf[i] != cell[i]);

      }

      strcpy(cell, buf);

      return dif;

}
```

```c
int main()

{

      char c[] = "_###_##_#_#_#_#__#__\n",

      b[] = "____________________\n";

      int n = sizeof(c) / sizeof(*c);

      do { printf("%s", c + 1); } while (evolve(c + 1, b + 1, n -
3));

}
```

```
###_##_#_#_#_#__#__
#_#####_#_#_#______
_##___##_#_#_______
_##___###_#________
_##___#_##_________
_##____###_________
_##____#_#_________
_##_____#__________
_##________________
```

rule 90 is an elementary cellular automaton. That means that it consists of a one-dimensional array of cells, each of which holds a single binary value, either 0 or 1. The automaton is given an initial configuration, and then progresses through other configurations in a sequence of discrete time steps. At each step, all cells are updated simultaneously. A pre-specified rule determines the new value of each cell as a function of its previous value and of the values in its two neighboring cells.

All cells obey the same rule, which may be given either as a formula or as a rule table that specifies the new value for each possible combination of neighboring values. In the case of Rule 90, each cell's new value is the exclusive or of the two neighboring values.

Equivalently, the next state of this particular automaton is governed by the following rule table:

current pattern	111	110	101	100	011	010	001	000
new state	0	1	0	1	1	0	1	0

$$(1011010)_2 = 2^6 + 2^4 + 2^3 + 2^1 = 64 + 16 + 8 + 2 = (90)_{10}$$

```c
void evolve(ull state, int rule)
{
	int i; ull st;
	printf("rule %d:\n", rule);
	do
	{
		st = state;
		for(i = n; --i; i)
		    putchar(st & b(i) ? '#' : '.');
		putchar('\n');

		for(state = i = 0; i < n; ++i)
			if(rule & b(7 & rot_left(st, i, n)))
				state |= b(i);
	}while (st != state);
}
```


```c
#include <stdio.h>
#define n 64 /* sizeof(ull) * CHAR_BIT */
#define b(x) (1ull << (x))
#define rot_left(a, i, n) (a << i | a >> (n - i))
#define print(i, n)\
{\
	for (i = n; --i; i)\
		putchar(st & b(i) ? '#' : '.');\
	putchar('\n');\
}
typedef unsigned long long ull;

int main(int argc, char ** argv)
{
	evolve(b(n / 2), 90);
	evolve(b(n / 4) | b(n - n / 4), 30);
}
```

Advanced Encryption Standard

The AES algorithm (also known as the rijndael algorithm) is a symmetric block cipher algorithm that takes a block size of 128 bits and converts them into ciphertext using keys of 128, 192, and 256 bits.

Features of AES

1. It uses Substitution and permutations, also called SP Networks.
2. A single key is expanded to be used in multiple rounds.
3. AES performs on byte data , instead of bit data.

The number of rounds during the encryption process depends on the key size that is being used

128-bit Key Length — 10 rounds

192-bit Key Length — 12 rounds

256-bit Key Length — 14 rounds

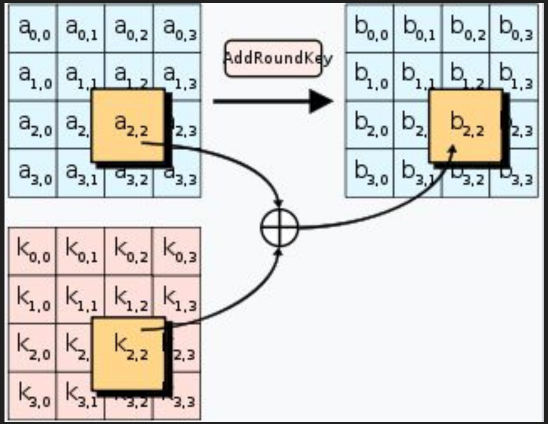
How does AES work

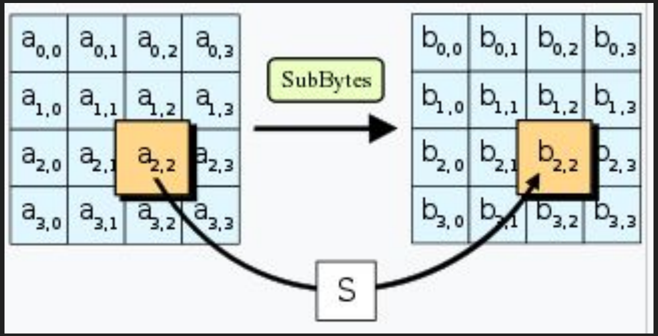
Everything is stored in a 4*4 matrix format

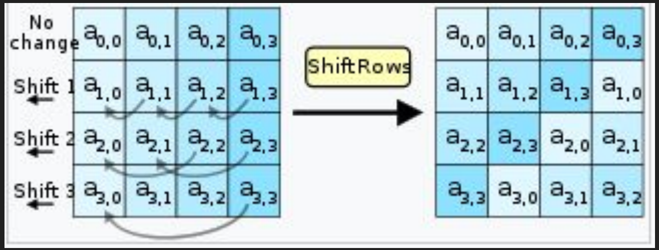
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

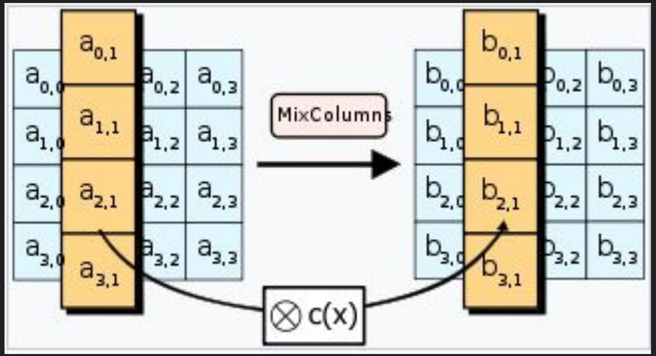
Four steps in each round

1. Add round key
2. Sub bytes
3. Shift Rows
4. Mix Columns









Thank You!