CS 265 CRYPTOGRAPHY PROJECT
REPORT

Project Tile: Not so Secret Message from Malwai – Part I

Crypto System: RSA

October 14, 2016

Team Members

Kabra, Rajat – 011425921

Bashyam, Yeshwanth - 011431017

Kabra, Rajat (rajat.kabra@sjsu.edu)
Bashyam, Yeshwanth (yeshwanthbashyam.balachander@sjsu.edu)

OBJECTIVE

The objective of the project is to exploit the vulnerability in the RSA cryptosystem, when the prime factors used by the system are close to each other by value. The project has implemented the RSA cryptosystem and has devised an RSA *Key Factorization attack* that will deduce the RSA private key value -*d*, from the RSA public key parameters -*N, e*.

PROBLEM STATEMENT

The problem adapted to demonstrate this project was proposed by Ed Schaefer in May, 2010 [1]

"For RSA to be safe, you need to pick your parameters n and e carefully. I forgot about that when I asked my friend Atipatsa in Malawi to encrypt a message for me using my RSA parameters

N = 316033...

and e = 17.

The cipher text he sent is

CT = 655373...

The complete parameters can be found in this additional file: parameters.txt

Find the plaintext number. Turn it into a binary string. The highest order bits are padding. Consider the 200 lowest order bits (this should be the same as dividing by 2200 and finding the remainder). That is the plaintext message encoded with ASCII, which is the code word."

Below is the information provided

Modulus - N:

```
3160332774263260970454747585057049809100379587193955605655712391008781
9295522849534318496830547730846019007640496755211064482229817971666968
9426595435572597197633507818204621591917460417859294285475630901332588
5454775521250470190221497465248435459237584253531030631345853752756382
57720039414711534847429265419
```

Cipher Text:

```
6553737221362549160467829576342303867230194078458614544455788969208029
4819860287924925474346680667131997413817580659610619950021078601544527
9985779597173650853676836877161679926166816253913491395741931064171382
1039539526627103122788149293368223969280354360846637109616466207418440
363036312473649455733021730
```

Public Exponent: 17

Kabra, Rajat (rajat.kabra@sjsu.edu)
Bashyam, Yeshwanth (yeshwanthbashyam.balachander@sjsu.edu)

CRYPTOSYSTEM - RSA

RSA algorithm is considered the gold standard of the public key cryptography because it was the first successful asymmetric cryptosystem i.e. public key crypto. RSA is named after its inventors, Ron Rivest, Adi Shamir and Leonard Adleman who first publicly described the algorithm in 1977. RSA is based on public key cryptology and uses a public key and a private key. The public key is used for encryption and is made public. The private key is used for decryption and is private to the owner of the key.

Public Key: (N, e)

Private Key: (d)

N, is called the modulus and is a product of two large prime number, *p* and *q*. Values chosen for *p* and *q* are critical to the strength of the system. *e*, the public exponent is chosen relatively prime to $\varphi(N)$.

The private exponent d, is chosen such that d and e are multiplicative inverse to $\varphi(N)$.

$$e.d = 1\ mod\ \varphi(N)$$

The operations performed in RSA are based on modular exponentiation. The encryption and decryption are performed using the below formula.

Encryption: $C = M^e\ mod\ N$

Decryption: $M = C^d\ mod\ N$

Kabra, Rajat (rajat.kabra@sjsu.edu)
Bashyam, Yeshwanth (yeshwanthbashyam.balachander@sjsu.edu)

CRYPTOSYSTEM IMPLEMENTATION

A sample RSA cryptosystem has been implemented which accepts ASCII plain text messages and performs RSA encryption for a given public key. The system also decrypts a cipher text when provided with the private key. The system represented below does not perform the attack to break the RSA key.

Encryption Module

```python
def encrypt_decrypt(data,exponent, modulus):
    result= pow(data,exponent,modulus)
    return result

def encrypt_module():
    plain_text = input("Enter Plain Text : ")
    key = int(input("Enter Public Key : "))
    exponenet = int(input("Enter Public Exponent : "))

    padded_text = plain_text
    padded_bytes = bytes(padded_text,"UTF-8")

    result =0
    for byte_data in padded_bytes:
        result = result*1000+ byte_data

    cipher_text = encrypt_decrypt(result,exponenet,key)
    print("\nResult\nCipher Text : ", cipher_text)
```

Decryption Module

```python
def decrypt_module():
    inp_cipher_text = int(input("Enter Cipher Text : "))
    pub_key = int(input("Enter Public Key : "))
    private_exp = int(input("Enter private exponent : "))

    plain_text = encrypt_decrypt(inp_cipher_text,private_exp,pub_key)

    ascii_list = []
    while plain_text !=0:
        rem = plain_text%1000
        ascii_list.insert(0,rem)
        plain_text = plain_text//1000

    print ("Decrypted Message : ", bytes(ascii_list).decode())
```

Encryption Result:

```
C:\Users\yeshwanth>py D:\SJSU\Year1\CS265\Project\RSACryptoSystem.py

RSA Crypto System
Select a choice
1. Encrypt
2. Decrypt
3. Exit
Choice : 1
Enter Plain Text : BANKACCESS:55697#7!@ASC
Enter Public Key : 3160332774263260970454747585057049809100379587193955605655712391008781929552284953431849683054773084601900764049675521106448222981797166696894265954355725971976335078182046215919174604178592942854756309013325885454775521250470190221497465248435459237584253531030631345853752756382577200394147115348474292654 19
Enter Public Exponent : 17

Result
Cipher Text :  9575066570610172367790089609287402988642492454557711361085194654995704984082365256680128144392367074329883971367623750186646955812992566749299314740184326280401861397837551488042725282460470624882187640447417947883703627068016842305197933556127693220018549775118385023369753453798590018814670246171845566884 0
```

Decryption Result:

```
RSA Crypto System
Select a choice
1. Encrypt
2. Decrypt
3. Exit
Choice : 2
Enter Cipher Text : 957506657061017236779008960928740298864249245455771136108519465499570498408236525668012814439236707432988397136762375018664695581299256674929931474018432628040186139783755148804272528246047062488218764044741794788370362706801684230519793355612769322001854977511838502336975345379859001881467024617184556688 40
Enter Public Key : 316033277426326097045474758505704980910037958719395560565571239100878192955228495343184968305477308460190076404967552110644822298179716669689426595435572597197633507818204621591917460417859294285475630901332588545477552125047019022149746524843545923758425353103063134585375275638257720039414711534847429265419
Enter private exponent : 929509639489194403074925760310897002676582231527634001663444820884935861633024986303485200898462671941735518838139859148955359700528578440263019398339919298949952606914443184161840836661990459044758615561986722727625589096164942263449554971986331590095186129747967243507113089951552045722381946310780117437 93
Decrypted Message :  BANKACCESS:55697#7!@ASC
```

Kabra, Rajat (rajat.kabra@sjsu.edu)
Bashyam, Yeshwanth (yeshwanthbashyam.balachander@sjsu.edu)

ATTACK ON THE SYSTEM

RSA cryptosystem, in general, is a very secure cryptosystem. The only attack that works on RSA is the brute force attack. But performing a brute force attack on a well thought RSA is highly infeasible considering the current computing power. But the strength of the RSA relies on the selection of the public key parameters. Vulnerabilities are induced in the system when the key parameters are not chosen well.

The public key is made up of two large prime numbers p and q, which is the heart of the RSA algorithm. An important thing to consider while choosing the public key is that the two factors *p* and *q* should be far away from each other, otherwise the key can be easily cracked with systematic factorization of *N*. Problem statement has stated a hint that the public key was not selected carefully. One of the major mistake could be that p and q are close to each other, making the algorithm vulnerable to factorization attacks. The project has implemented the factorization attack on the public modulus N, using *Fermat's Factorization method* [2] to arrive at the prime numbers *p* and *q*

This factorization method is based on the representation of an odd integer as the difference of two squares. So for a number N, it would look for two integer x and y such that $N = x^2 - y^2 = (x - y)(x + y)$. This gives us two factors, p and q, of N.

<table>
<tr><th>Algorithm</th><th>Implementation</th></tr>
<tr><td><pre>
FermatFactor(N) [2]:
    a ← ceil(sqrt(N))
    b2 ← a*a - N
    while b2 is not a square:
        b2 ← b2 + 2*a + 1
        a ← a + 1
    endwhile
    return a - sqrt(b2)
</pre></td><td><pre>
def fermat_factorization(number):
    xPart = int(math.ceil(Decimal(key).sqrt()))
    y2Part = xPart**2 - key
    while(isPerfectSquare(y2Part) ==False):
        y2Part = y2Part+2*xPart+1
        xPart = xPart+1
    yPart = int(Decimal(y2Part).sqrt())
    return xPart+yPart,xPart-yPart
</pre></td></tr>
</table>

After N is factorized into p and q, the totient of N is calculated which is the product of (p-1) and (q-1). Using the totient and e, we calculate the private key by *Extended Euclidean Algorithm* [3]. Once the private key is derived, the cipher text is decrypted using the RSA decryption equation. The plaintext that is calculated is in numeric form and is padded and the actual data residing in the 200 least significant bits. According to the problem statement, the numeric plaintext is first converted into binary form only the 200 lowest order bits are extracted. The extracted binary is converted to corresponding ASCII string.

Algorithm	Implementation

<pre>**extended_gcd**(a, b) [3]:
    s := 0;   old_s := 1
    t := 1;   old_t := 0
    r := b;   old_r := a
    while r ≠ 0
        quotient := old_r div r
            old_r := r
            old_s := s
            old_t := t
            r := old_r - quotient * r
            s := old_s - quotient * s
            t := old_t - quotient * t
return old_r</pre>
	<pre>#Calculates RSA private key from e(exponent) and totien(n)
	#Uses Extended Euclidean algorithm to find private_key (d)
	def private_key(e, totient):
	    z,x,c,v=0,1,1,0
	    while e != 0:
	        q = totient//e
	        r = totient%e
	        m = z-c*q
	        n = x-v*q
	        totient,e=e,r
	        z,x = c,v
	        c,v = m,n
	        gcd = totient
	    return z</pre>

ALGORITHM OF THE ATTACK

```
not-so-secret-rsa(ct, N, e):
begin
    p, q : = FermatFactor(N)
    N2 := p*q
    if N2 = N then
        Totient := (p-1)(q-1)
        d :=  extended-gcd(e, totient)

        padded_plain_text := rsa-decrypt(ct, d, N)
        bin_plain_text := extract last 200 bits of to_binary(padded_plain_text)
        plain_text = bin_to_ascii(bin_plain_text)
    else
        return failure
    return plain_text
```

All the methods used by the algorithm, listed below, are implemented in the project.

`isPerfectSquare`: Check if a number is a perfect square.

`private_key`: Calculated private key totient and e

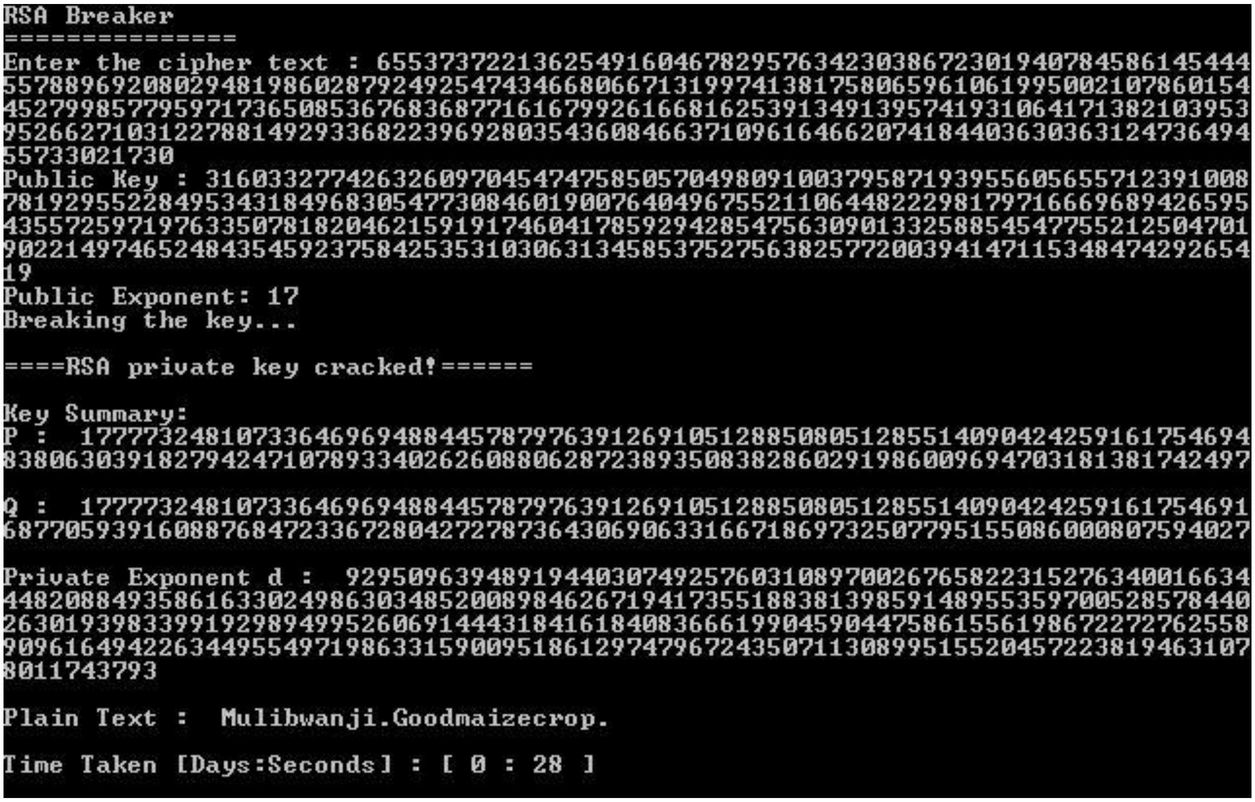
`decrypt_cipher_text`: Performs RSA decryption

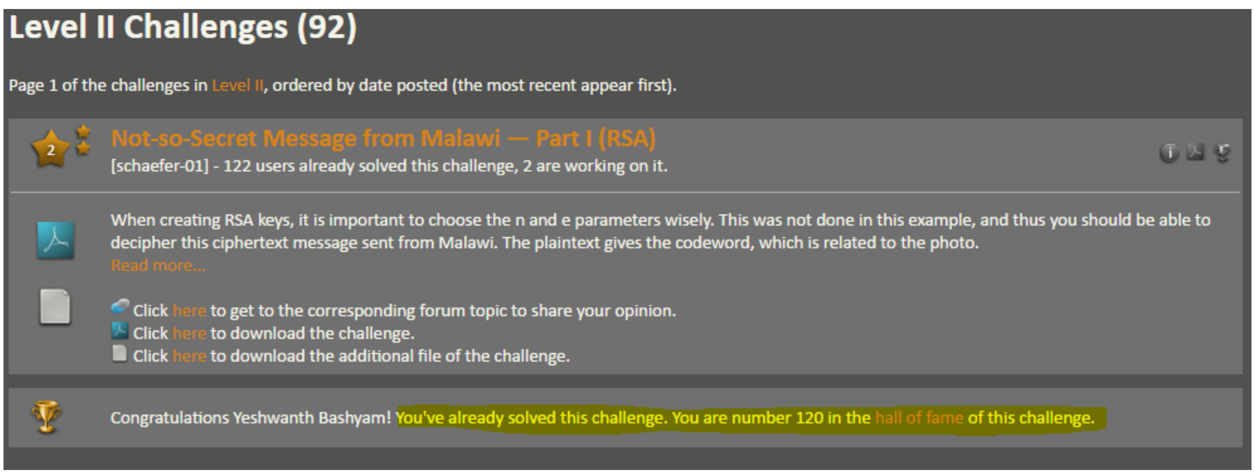
`bin_to_ascii`: Converts binary stream to ASCII string

`fermat_factorization`: Fermat Factorization Algorithm

Kabra, Rajat (rajat.kabra@sjsu.edu)
Bashyam, Yeshwanth (yeshwanthbashyam.balachander@sjsu.edu)

OUTPUT SNAPSHOT.





Kabra, Rajat (rajat.kabra@sjsu.edu)
Bashyam, Yeshwanth (yeshwanthbashyam.balachander@sjsu.edu)

WORK FACTOR ANALYSIS

The factorization approach used in the project is not a complete brute force. It performs a factorization of N with factors starting from $\sqrt{N}$. The time taken to break the key is directly proportional to the difference between p and q and does not depend on the size of N.

Considering $\Delta pq = p - q$. The work factor of the attack is Δpq. The closer the values of p and q are, the sooner the key is cracked.

REFERENCES

[1] – mysterytwisterc3 website. https://www.mysterytwisterc3.org/images/challenges/mtc3-schaefer-01-rsa-en.pdf

[2] – Fermat Factorization Method. https://en.wikipedia.org/wiki/Fermat%27s_factorization_method

[3] – Extended Euclidean Algorithm. https://en.wikipedia.org/wiki/Extended_Euclidean_algorithm

Kabra, Rajat (rajat.kabra@sjsu.edu)

Bashyam, Yeshwanth (yeshwanthbashyam.balachander@sjsu.edu)

PROJECT ATTRIBUTION SHEET

Deliverable	Owned By	Completed By	Comments
Write software to implement the system and attack it.	• Kabra, Rajat • Bashyam, Yeshwanth	• Kabra, Rajat (50%) • Bashyam, Yeshwanth (50%)	• Rajat focused on decrypting the private key, converting binary text into ASCII plain text and implementing encryption. • Yeshwanth focused on factorizing the key, removing the padding and implementing decryption.
Estimate and Document Work Factor for Attack	• Bashyam, Yeshwanth	• Bashyam, Yeshwanth (100%)	
Capture Screenshots	• Kabra, Rajat	• Kabra, Rajat (100%)	
Write report that includes a detailed description and analysis of your work and results	• Kabra, Rajat • Bashyam, Yeshwanth	• Kabra, Rajat (50%) • Bashyam, Yeshwanth (50%)	
Quality Assurance	• Kabra, Rajat • Bashyam, Yeshwanth	• Kabra, Rajat (50%) • Bashyam, Yeshwanth (50%)	• Yeshwanth focused on the time taken by the code to attack the system and did negative testing on the system. • Rajat focused on the efficiency of algorithms used and time efficiency.

Kabra, Rajat (rajat.kabra@sjsu.edu)
Bashyam, Yeshwanth (yeshwanthbashyam.balachander@sjsu.edu)