

Cryptography - Lab 5 - Asymmetric Crypto

Javier Antxon Garrues Apecechea

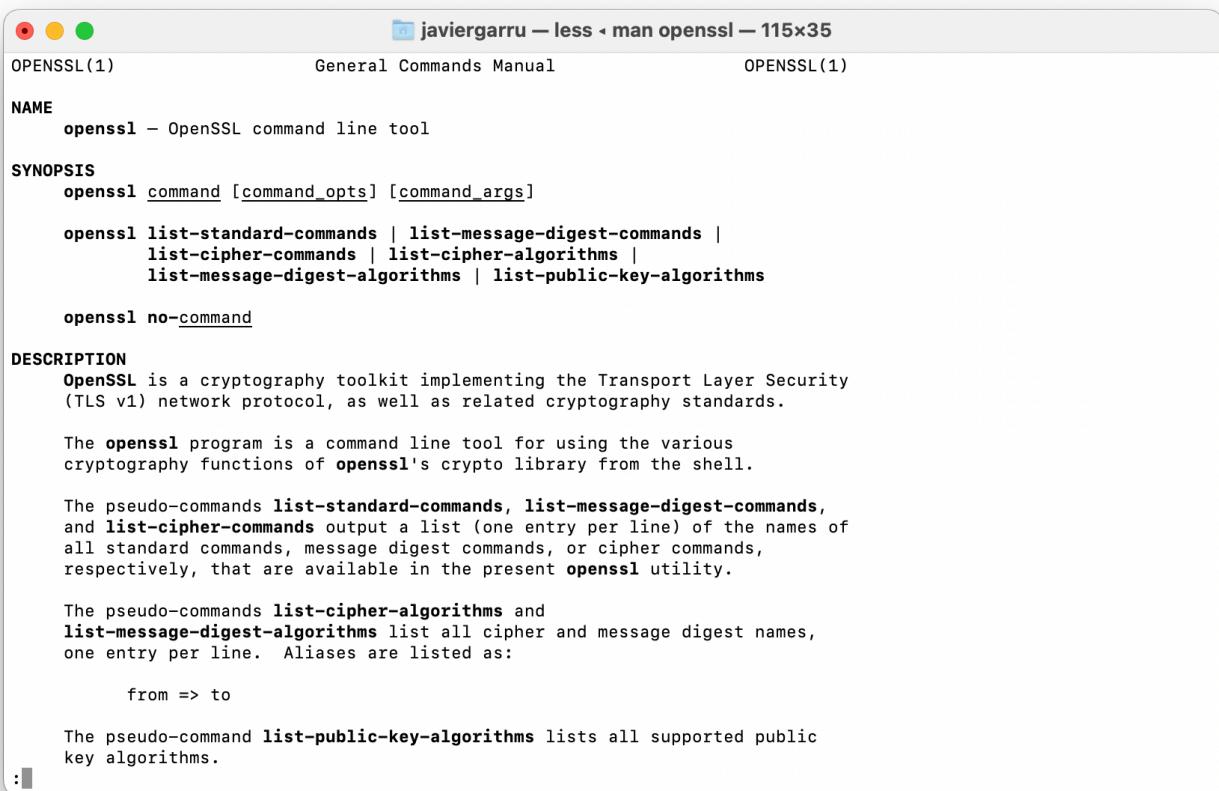
Student ID: 24647808

In this lab we will get familiar with Diffie Hellman Key Exchange and the RSA algorithm.

Part Two: openssl

In previous labs we had used this same tool so there is no need to download it again. We can have a look at the manual to check the command options:

```
man openssl
```



The screenshot shows a terminal window with the title "javiergarru — less - man openssl — 115x35". The window displays the man page for the OpenSSL command-line tool. The page is organized into sections: NAME, SYNOPSIS, DESCRIPTION, and others. The SYNOPSIS section shows examples of commands like "openssl command [command_opts] [command_args]", "openssl list-standard-commands | list-message-digest-commands | list-cipher-commands | list-cipher-algorithms | list-message-digest-algorithms | list-public-key-algorithms", and "openssl no-command". The DESCRIPTION section provides a brief overview of what OpenSSL is and how the program is used. The page continues with detailed descriptions of the pseudo-commands and supported public key algorithms.

```
OPENSSL(1)           General Commands Manual          OPENSSL(1)

NAME
    openssl - OpenSSL command line tool

SYNOPSIS
    openssl command [command_opts] [command_args]

    openssl list-standard-commands | list-message-digest-commands |
        list-cipher-commands | list-cipher-algorithms |
        list-message-digest-algorithms | list-public-key-algorithms

    openssl no-command

DESCRIPTION
    OpenSSL is a cryptography toolkit implementing the Transport Layer Security
    (TLS v1) network protocol, as well as related cryptography standards.

    The openssl program is a command line tool for using the various
    cryptography functions of openssl's crypto library from the shell.

    The pseudo-commands list-standard-commands, list-message-digest-commands,
    and list-cipher-commands output a list (one entry per line) of the names of
    all standard commands, message digest commands, or cipher commands,
    respectively, that are available in the present openssl utility.

    The pseudo-commands list-cipher-algorithms and
    list-message-digest-algorithms list all cipher and message digest names,
    one entry per line. Aliases are listed as:

        from => to

    The pseudo-command list-public-key-algorithms lists all supported public
    key algorithms.

:
```

Part Three: Diffie Hellman Key Exchange

We select a prime number from the following website <https://t5k.org/lists/small/10000.txt>. In our case, we have firstly selected the last one available **104729**, but it is too big and the next step takes longer than expected. For this reason, we changed and chose **977**.

Lab 5										Diffie-Hellman key exchange calculator			
												https://t5k.org/lists/small/10000.txt	
												Failed to open page	
98321	98323	98327	98347	98369	98377	98387	98389	98407	98411				
98419	98429	98443	98453	98459	98467	98473	98479	98491	98507				
98519	98533	98543	98561	98563	98573	98587	98621	98627	98639				
98641	98663	98669	98689	98711	98713	98717	98729	98731	98737				
98773	98779	98801	98807	98809	98837	98849	98867	98869	98873				
98887	98893	98897	98899	98909	98911	98927	98929	98939	98947				
98953	98963	98981	98993	98999	99013	99017	99023	99041	99053				
99079	99083	99089	99103	99109	99113	99131	99133	99137	99139				
99149	99173	99181	99191	99223	99233	99241	99251	99257	99259				
99277	99289	99317	99347	99349	99367	99371	99377	99391	99397				
99401	99409	99431	99439	99469	99487	99497	99523	99527	99529				
99551	99559	99563	99571	99577	99581	99607	99611	99623	99643				
99661	99667	99679	99689	99707	99709	99713	99719	99721	99733				
99761	99767	99787	99793	99809	99817	99823	99829	99833	99839				
99859	99871	99877	99881	99901	99907	99923	99929	99961	99971				
99981	99991	100003	100019	100043	100049	100057	100061	100103	100109				
100129	100151	100153	100169	100183	100193	100207	100213	100237	100261				
100267	100271	100279	100291	100297	100313	100333	100343	100357	100361				
100363	100379	100391	100393	100403	100411	100417	100447	100459	100469				
100483	100493	100501	100511	100517	100519	100523	100537	100547	100549				
100559	100591	100601	100613	100621	100649	100669	100673	100693	100699				
100703	100733	100741	100747	100769	100787	100799	100801	100811	100823				
100829	100847	100853	100897	100913	100927	100931	100937	100943	100957				
100981	100987	100994	101009	101021	101027	101051	101063	101081	101089				
101107	101111	101113	101117	101119	101141	101149	101153	101161	101173				
101182	101197	101203	101207	101209	101221	101267	101273	101279	101281				
101287	101293	101324	101333	101341	101347	101359	101363	101377	101383				
101399	101411	101419	101429	101449	101467	101477	101481	101489	101501				
101503	101513	101527	101531	101533	101537	101561	101571	101581	101599				
101603	101611	101627	101641	101653	101663	101681	101693	101701	101719				
101723	101737	101741	101747	101749	101771	101789	101797	101807	101833				
101837	101839	101861	101869	101873	101879	101891	101917	101921	101929				
101939	101957	101963	101977	101987	101994	102001	102013	102019	102023				
102031	102043	102050	102061	102071	102077	102079	102103	102107	102109				
102121	102139	102141	102161	102181	102191	102197	102199	102203	102217				
102229	102233	102241	102251	102253	102259	102293	102299	102301	102317				
102329	102337	102359	102367	102397	102407	102409	102433	102437	102451				
102461	102481	102497	102499	102503	102523	102533	102539	102547	102551				
102559	102563	102587	102593	102607	102611	102643	102647	102653	102667				
102673	102677	102679	102701	102761	102763	102769	102791	102797	102811				
102829	102841	102859	102871	102877	102881	102911	102913	102929	102931				
102954	102967	102981	103001	103007	103043	103049	103061	103069	103079				
103087	103091	103094	103099	103123	103141	103171	103177	103183	103217				
103231	103237	103281	103291	103307	103319	103333	103349	103357	103387				
103391	103393	103397	103409	103421	103423	103451	103457	103471	103483				
103511	103529	103541	103553	103561	103567	103573	103579	103583	103591				
103613	103619	103643	103651	103657	103669	103681	103689	103699	103703				
103723	103769	103781	103801	103811	103813	103837	103841	103843	103867				
103889	103903	103913	103919	103951	103963	103967	103969	103979	103981				
103991	103993	103997	104003	104009	104021	104033	104047	1040493	104059				
104087	104107	104109	104113	104119	104123	104141	104147	104161	104173				
104149	104183	104197	104231	104233	104239	104243	104261	104268	104287				
104309	104311	104323	104327	104347	104369	104381	104383	104393	104399				
104417	104459	104471	104473	104479	104491	104513	104527	104537	104543				
104549	104551	104561	104579	104593	104597	104623	104639	104651	104659				
104677	104681	104683	104693	104701	104707	104711	104717	104723	104729				

Then, we go to the following website <http://www.bluetulip.org/2014/programs/primitive.html> and introduce the chosen prime. This website returns all the primitive roots of our number. For 977 there are 480 primitive roots. Remember that a **primitive root of** a prime number **p** is another number **g** such that all the numbers **1,2,...,p-1** can be obtained through **gn mod p** with n being any number. For instance, 2 is a primitive root of 5:

- 1 = 2*8 mod 5
- 2 = 2*11 mod 5
- 3 = 2*9 mod 5
- 4 = 2*12 mod 5

Not Secure — bluetulip.org

Lab 5 | Diffie-Hellman key exchange calculator | https://t5k.org/lists/small/10000.txt | Primitive Roots Calculator

Blue Tulip Potpourri

Home Drawing Potpourri

Primitive Roots Calculator

Enter a prime number into the box, then click "submit." It will calculate the primitive roots of your number.

The first 10,000 primes, if you need some inspiration.

977 has 480 primitive roots, and they are 3, 5, 6, 10, 12, 13, 19, 20, 21, 23, 24, 26, 27, 33, 35, 37, 38, 40, 41, 42, 45, 46, 48, 51, 53, 54, 55, 59, 66, 70, 73, 74, 75, 76, 79, 82, 84, 85, 87, 90, 91, 92, 93, 96, 97, 102, 104, 105, 106, 113, 117, 118, 125, 129, 132, 133, 134, 135, 143, 145, 146, 147, 148, 149, 150, 151, 153, 155, 156, 159, 161, 163, 167, 171, 173, 175, 177, 181, 183, 185, 187, 189, 191, 192, 193, 195, 197, 199, 201, 204, 207, 208, 209, 211, 212, 213, 215, 216, 220, 221, 223, 226, 231, 233, 234, 235, 236, 239, 241, 242, 245, 249, 250, 251, 253, 257, 258, 259, 263, 264, 266, 267, 280, 281, 282, 283, 285, 286, 287, 290, 292, 293, 294, 296, 297, 298, 300, 302, 303, 304, 305, 307, 309, 310, 311, 315, 316, 320, 321, 322, 323, 325, 326, 327, 328, 331, 333, 334, 335, 336, 337, 340, 342, 345, 346, 347, 348, 355, 359, 361, 362, 363, 364, 366, 367, 368, 369, 371, 372, 373, 377, 378, 381, 382, 383, 384, 385, 388, 389, 391, 393, 394, 397, 398, 402, 404, 406, 408, 410, 412, 414, 416, 418, 420, 422, 423, 430, 432, 434, 436, 442, 443, 445, 446, 448, 451, 452, 459, 461, 463, 465, 467, 470, 471, 477, 478, 479, 481, 486, 487, 491, 495, 496, 499, 500, 502, 505, 507, 509, 511, 514, 515, 516, 518, 525, 526, 528, 532, 534, 536, 537, 545, 547, 551, 553, 555, 559, 560, 561, 562, 563, 564, 566, 569, 570, 572, 575, 579, 580, 583, 584, 586, 587, 588, 589, 592, 593, 594, 595, 596, 598, 600, 604, 605, 606, 608, 609, 610, 611, 613, 614, 615, 617, 618, 622, 629, 630, 631, 632, 635, 637, 640, 641, 642, 643, 646, 648, 649, 650, 651, 652, 654, 655, 656, 657, 661, 662, 666, 667, 668, 670, 672, 673, 674, 675, 677, 679, 680, 681, 683, 684, 685, 687, 690, 691, 692, 694, 695, 696, 697, 698, 713, 714, 715, 716, 717, 721, 722, 723, 724, 725, 726, 727, 728, 729, 730, 731, 732, 733, 734, 742, 743, 744, 745, 751, 754, 756, 757, 758, 764, 765, 766, 768, 769, 770, 773, 780, 782, 783, 785, 786, 788, 791, 793, 794, 795, 796, 797, 803, 804, 806, 807, 809, 810, 813, 814, 816, 817, 819, 822, 825, 826, 827, 828, 829, 830, 831, 832, 834, 836, 837, 844, 845, 848, 852, 859, 860, 864, 867, 869, 871, 873, 875,

Listen! First Current

And we **select** one from this list such as **12**. Now, we go to the following site

<https://www.irongeek.com/diffie-hellman.php> and introduce the selected values. Apart from those values, we also choose one value for Alice and another for Bob. We will choose **10** and **3**.

irongeek.com

Lab 5 | Diffie-Hellman key exchange calculator | https://t5k.org/lists/small/10000.txt | Primitive Roots Calculator | RSS feed | Follow @irongeek_adc | Donate | Subscribers or Patreon

Irongeek Security | Hacking Illustrated Videos | InfoSec Articles | Mobile Pen-testing Tools | Apps/Scripts | Reviews | RSS | Your IP | Podcasts | Hoosier Hackers | NewsCat | Links | Contact | Forums | Workout | Nutrition | Supplements | Humor | Irongeek Campuses | Fed Watch | Books | Store | About | Follow @irongeek_adc

Search irongeek.com: ENHANCED BY Google Search Select Language Powered by Google Translate Affiliates: irongeek.com Help irongeek.com pay for bandwidth and research equipment: Donate

Live 5-Day Course

Help irongeek.com pay for bandwidth and research equipment:

stayz Holiday homes near Search homes

Crappy PHP script for a simple Diffie-Hellman key exchange calculator. I guess I could have used Javascript instead of PHP, but I had rounding errors.

Set these two for everyone:
 $g: 12 \quad p: 977$

Alice Bob
 $a: 10 \quad b: 3$

$a = 10$
 $A = g^a \bmod p = 12^{10} \bmod 977 = 948$
 $b = 3$
 $B = g^b \bmod p = 12^3 \bmod 977 = 751$
Alice and Bob exchange A and B in view of Carl
 $\text{key}_a = B^a \bmod p = 751^{10} \bmod 977 = 36$
 $\text{key}_b = A^b \bmod p = 948^3 \bmod 977 = 36$

Hi all, the point of this game is to meet new people, and to learn about the Diffie-Hellman key exchange. Did you ever wonder how two parties can negotiate a cryptographic key in the presence of an observer, without the observer figuring out the key? My guess is not, but bear with me. This will be a simplified version of the Diffie-Hellman key exchange (in real life, better constants and larger variables should be chosen), in the form of a game. Enter as many times as you like.

Fixed numbers: $g=10, p=541$

Contestant steps:

- Find someone you do not know, and introduce yourself.
- One of you is Alice (a), and one is Bob (b). If genders don't match that's ok. one of you can be Alan and the other Barb for all I care.

stayz Holiday homes near Search homes

A summary of the process can be seen here:

- A prime and one primitive root of the prime are known
- Alice chooses a random number
- Bob chooses a random number
- Alice calculates the primitive root powered to her chosen number modulo the prime number
- Bob calculates the primitive root powered to his chosen number modulo the prime number
- Alice and Bob exchange their calculated values
- Alice powers Bob's value to her secret number and finds its modulo prime number
- Bob powers Alice's value to his secret number and finds its modulo prime number
- Alice and Bob now share the same secret key without having to exchange it

We can also check the process and calculations manually with python:

```
(base) javiergarru@MBP-de-Javier-2 ~ % python3
Python 3.9.12 (main, Apr  5 2022, 01:53:17)
[Clang 12.0.0 ] :: Anaconda, Inc. on darwin
Type "help", "copyright", "credits" or "license" for more information.
[>>> 12**3%977
751
[>>> 12**10%977
948
[>>> 751**10%977
36
[>>> 948**3%977
36
>>> █
```

Part Four: Manual RSA Encryption and Decryption

4.1. Deriving the private key:

We download the file **RSA.py** and note the values given in the lab document:

- $p = 191$
- $q = 281$
- $e = 33$
- $e \times d = 1 \text{ mod } (\phi(n))$
- $\phi(n) = (p-1)*(q-1)$

Public key: (e, n)

Private key: (d, n)

Question: what is the numerical values of the private and public key?

We execute the program and obtain the results:

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL JUPYTER
(base) javiergarru@MBP-de-Javier-2 w8 % python3 RSA.py
public key (33, 53671)
private key (12897, 53671)
encrypt [31644]
decrypt [200]
(base) javiergarru@MBP-de-Javier-2 w8 %

```

Public key: (33, 53671)

Private key: (12897, 53671)

Summary of the steps taken to obtain the keys:

1. Choose two prime numbers: p and q
2. Multiply the two numbers: $n = p \times q$
3. Calculate the totient of n: $\phi(n) = (p-1)(q-1)$
4. Select a random relative prime of the totient: $\text{gcd}(e, \phi(n)) = 1$
5. Find the multiplicative inverse modulo n of e: $d \times e \text{ mod } n = 1$
6. Generate public key (e, n) and private key (d, n)

4.2. Encrypting a message:

Question: what is the cipher-text of 100?

The ciphertext of 100 is 31644.

Question: try to encrypt a big number and describe your findings

We change the 100 to **10000** in the code and execute the program. In addition to this, we also change the ciphertext we want to decrypt to check that everything works correctly:

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL JUPYTER
81     print ("public key",pub)
82     print ("private key",pri)
83     print ("encrypt",encrypt(pub,[10000]))
84     print ("decrypt",decrypt(pri,[2889]))
85
(base) javiergarru@MBP-de-Javier-2 w8 % python3 RSA.py
public key (33, 53671)
private key (12897, 53671)
encrypt [2889]
decrypt [10000]
(base) javiergarru@MBP-de-Javier-2 w8 %

```

Let's now try with a bigger number like **100000**:

```
81     print ("public key",pub)
82     print ("private key",pri)
83     print ("encrypt",encrypt(pub,[100000]))
84     print ("decrypt",decrypt(pri,[47736]))
85
```

PROBLEMS OUTPUT DEBUG CONSOLE **TERMINAL** JUPYTER

```
(base) javiergarru@MBP-de-Javier-2 w8 % python3 RSA.py
public key (33, 53671)
private key (12897, 53671)
encrypt [47736]
decrypt [46329]
(base) javiergarru@MBP-de-Javier-2 w8 %
```

As we can see, the encryption seems to work well, but when we decrypt the obtained ciphertext (47736) we don't obtain the original value. The reason behind this problem is that we are working with a relatively low n (53671). The encryption process in RSA consists of powering the introduced number (100000) to the first value of the public key and finding its module n. The operation would look something like this:

Plaintext_value^e mod n = ciphertext_value

$100000^{33} \text{ mod } 53671 = 47736$

Because 100000 is greater than 53671 and, using the modulo properties, we know that $100000^{33} \text{ mod } 53671$ is the same as $(100000 \text{ mod } 53671)^{33} \text{ mod } 53671$, which is the same as $(46329)^{33} \text{ mod } 53671$. And that is how we would obtain the ciphertext associated to 46329, which means that the ciphertext that we obtained for 1000000 is indeed the one associated to another number. Let's see this with the program:

```

69  def encrypt(key, bstr):
70      return [rsa(key,c) for c in bstr]
71
72  #Not everything that can be counted counts, and not everyth
73  #Make everything as simple as possible, but not simpler." -
74  #There are only two ways to live your life. One is as thoug
75
76  # Decrypt. Takes a key and an array of numbers.
77  # Runs RSA on each of the values. Returns a byte string.
78  def decrypt(key, arr):
79      return [rsa(key,a) for a in arr]
80
81  print ("public key",pub)
82  print ("private key",pri)
83  print ("encrypt",encrypt(pub,[46329]))
84  print ("decrypt",decrypt(pri,[47736]))
85

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL JUPYTER

```

(base) javiergarru@MBP-de-Javier-2 w8 % python3 RSA.py
public key (33, 53671)
private key (12897, 53671)
encrypt [47736]
decrypt [46329]
(base) javiergarru@MBP-de-Javier-2 w8 %

```

As we can see the encryption and decryption of the value now works as expected and the ciphertext obtained by encrypting 46329 is the same as the one obtained when encrypting 100000.

4.3.Decrypting a message:

C = 11759

Question: what is the plain-text of C?

We execute the code again and see that the associated decryption is **200**:

```

75
76  # Decrypt. Takes a key and an array of numbers.
77  # Runs RSA on each of the values. Returns a byte string.
78  def decrypt(key, arr):
79      return [rsa(key,a) for a in arr]
80
81  print ("public key",pub)
82  print ("private key",pri)
83  print ("encrypt",encrypt(pub,[200]))
84  print ("decrypt",decrypt(pri,[11759]))
85

```

4.4.RSA key generation and encryption with an online tool:

Now, we can visit the following website <https://www.cryptool.org/en/cto/rsa-step-by-step.html> and introduce the same values and see how the different keys are generated:

only from the product n the two primes that yield the product. This decomposition is also called the factorization of n .

As a starting point for RSA choose two primes p and q .

1st prime p =
191

2nd prime q =
281

For the algorithm to work, the two primes must be different.

Public key

The product n is also called modulus in the RSA method.

$n = p \times q = 53671$ (16 bit)

For demonstration we start with small primes. To make the factorization difficult, the primes must be much larger. Currently, values of n with several thousand binary digits are used for secure communication.

The public key consists of the modulus n and an exponent e .

e =
33

This e may even be pre-selected and the same for all participants.

This website would like to use cookies for Google Analytics. [Learn more.](#) Accept Reject

Here we can see how p , q and e are introduced and n is calculated. Let's now see how the keys are generated with these values:

The screenshot shows a browser window with the CryptTool-Online homepage. The title bar includes tabs for "Lab 5", "Diffie-Hellman key ex...", "https://t5k.org/lists/s...", "Primitive Roots Calcul...", "Lab 5", "RSA (step-by-step) -...", "power a number - Bu...", and "Encryption of a numb...". The main content area has a green header "CrypTool-Online" with the subtitle "Cryptography for everybody". A navigation bar on the right includes a US flag, a user icon, and a search icon.

Secret key

RSA uses the Euler ϕ function of n to calculate the secret key. This is defined as

$$\phi(n) = (p - 1) \times (q - 1) = 53200$$

The prerequisite here is that p and q are different. Otherwise, the ϕ function would be calculated differently.

It is important for RSA that the value of the ϕ function is coprime to e (the largest common divisor must be 1).

$$\gcd(e, \phi(n)) = 1$$

To determine the value of $\phi(n)$, it is not enough to know n . Only with the knowledge of p and q we can efficiently determine $\phi(n)$.

The secret key also consists of a d with the property that $e \times d - 1$ is a multiple of $\phi(n)$.

Expressed in formulas, the following must apply:

$$e \times d = 1 \pmod{\phi(n)}$$

In this case, the mod expression means equality with regard to a residual class. It is $x = y \pmod{z}$ if and only if there is an integer a with $x - y = z \times a$.

For the chosen values of p , q , and e , we get d as:

$$d = 12897$$

This d can always be determined (if e was chosen with the restriction described above) — for example with the extended Euclidean algorithm.

Encryption and decryption

Internally, this method works only with numbers (no text), which are between 0 and $n - 1$.

A message m (number) is encrypted with the public key (n, e) by calculating:

$$m' := m^e \pmod{n}$$

This website would like to use cookies for Google Analytics. [Learn more.](#)

Accept

Reject

The euler function is used to calculate the number of relative primes smaller than the value n :

$$\phi(n) = \phi(p \times q) = \phi(p) \times \phi(q) = (p-1) \times (q-1) = 190 \times 280 = 53200$$

Then it is checked that $\gcd(e, \phi(n)) = 1 = \gcd(33, 53200)$

If $\phi(n)$ would have to be calculated without knowing p and q it would be significantly more complex and incredibly computationally heavy. This provides security to the algorithm.

Then d is found such that is the multiplicative inverse of e modulo n :

$$e \times d = 1 \pmod{\phi(n)} \text{ or in our case } 33 \times 12897 = 1 \pmod{53200}$$

This d is calculated with the help of the extended Euclidean algorithm.

The screenshot shows the homepage of CryptTool-Online. The title bar says "cryptool.org". Below it, there's a navigation bar with tabs: "Lab 5", "Diffie-Hellman key ex...", "https://t5k.org/lists/s...", "Primitive Roots Calcul...", "Lab 5", "RSA (step-by-step) -...", "power a number - Bu...", and "Encryption of a numb...". The main content area has a green header "CrypTool-Online" with the subtitle "Cryptography for everybody". There are also icons for a US flag, a person, and a search icon.

Encryption and decryption

Internally, this method works only with numbers (no text), which are between 0 and $n - 1$.

A message m (number) is encrypted with the public key (n, e) by calculating:

$$m' := m^e \pmod{n}$$

Decrypting with the private key (n, d) is done analogously with

$$m'' := (m')^d \pmod{n}.$$

This is

$$m'' = m^{e \times d} \pmod{n}.$$

RSA exploits the property that

$$x^a = x^b \pmod{n}$$

if

$$a = b \pmod{\phi(n)}$$

As e and d were chosen appropriately, it is

$$m'' = m.$$

The order does not matter. You could also first raise a message with the private key, and then power up the result with the public key — this is what you use with RSA signatures.

Messages

This website would like to use cookies for Google Analytics. [Learn more.](#)

Accept

Reject

Now, the website explain how encryption and decryption occurs and we can proceed to encrypt messages:

The screenshot shows the "Messages" page of CryptTool-Online. At the top, there's a cookie consent banner. Below it, the page title is "Messages". A text instruction says: "In the following two text boxes 'Plaintext' and 'Ciphertext', you can see how encryption and decryption work for concrete inputs (numbers)". There are two input fields: one for plain text ("d") and one for numbers ("100"). The number input field has a red arrow pointing to a ciphertext input field below it, which contains "31644". A grey arrow points down from the number input field to the ciphertext field.

Used library

This page uses the library [BigInteger.js](#) to work with big numbers.

As a result, you can calculate arbitrarily large numbers in JavaScript, even those that are actually used in RSA applications.

CTOAUTHORS: Timm Knappe (thanks to Bernhard Esslinger for the review), last update 2021-04-27

This website would like to use cookies for Google Analytics. [Learn more.](#)

Accept

Reject

When we try to encrypt a value greater than n , the program tells us that this cannot be done. Both python program and website allow us to do it but the results are not correct for the reasons given when answering the question:

The screenshot shows the CryptTool-Online interface. In the 'Plaintext (enter text)' field, the letter 'd' is entered. In the 'Plaintext (enter numbers, e.g. 6, 13, 111)' field, the number '100000' is entered. A red arrow points from this field to a red-bordered error message box containing the text: 'A plaintext number is too big. The maximum value is 53670 as n = 53671. Please choose bigger primes to get a bigger n.' Below these fields is a 'Ciphertext (enter numbers, e.g. 128, 52, 67)' field containing the value '47736'. A red arrow points from the error message box down to this ciphertext field.

Used library

This page uses the library [BigInteger.js](#) to work with big numbers.

As a result, you can calculate arbitrarily large numbers in JavaScript, even those that are actually used in RSA applications.

This website would like to use cookies for Google Analytics. [Learn more.](#)

Accept

Reject

Part Five: Attack RSA Keys

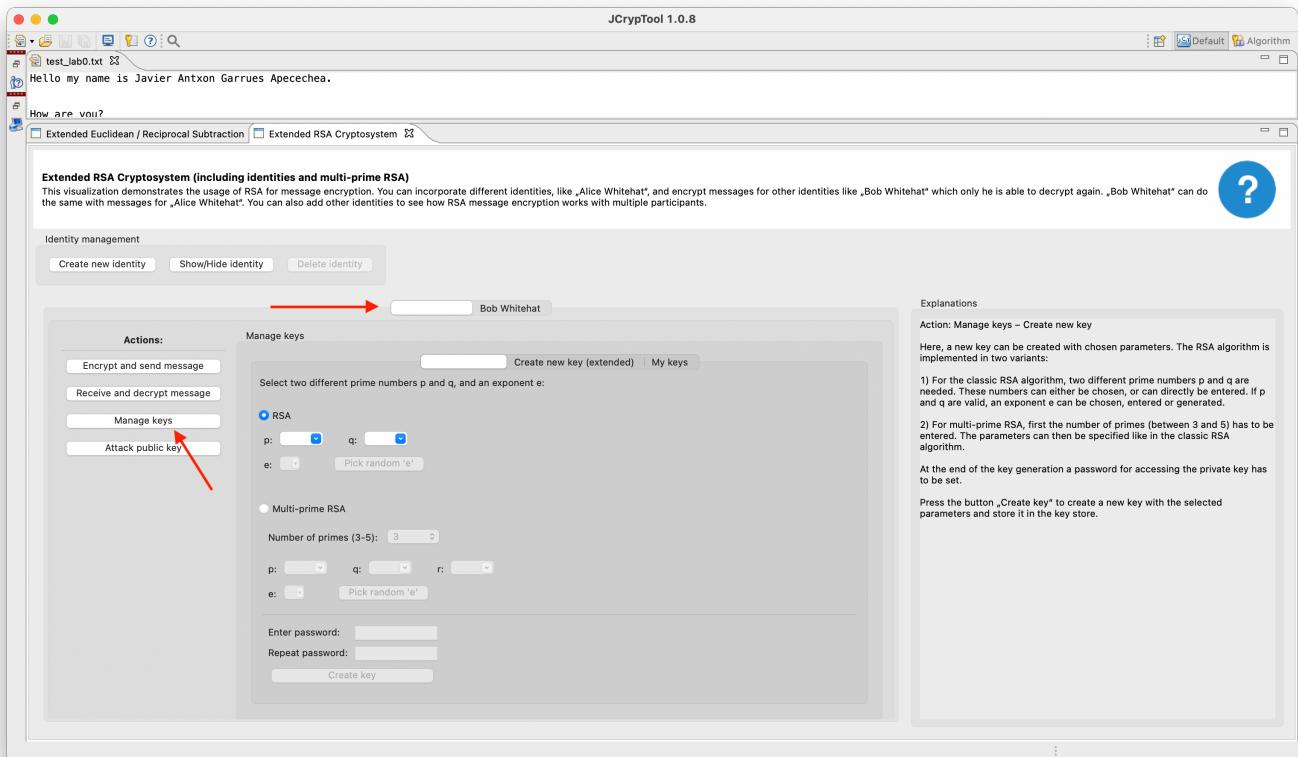
The strength of RSA is based on the hardness of the IFP (integer factorization problem). An efficient non-quantum solution to this problem when the numbers are large haven't been found, but it has been impossible to prove that this algorithm does not exist.

Now, we will see that RSA can be cracked in an exhaustive-way search.

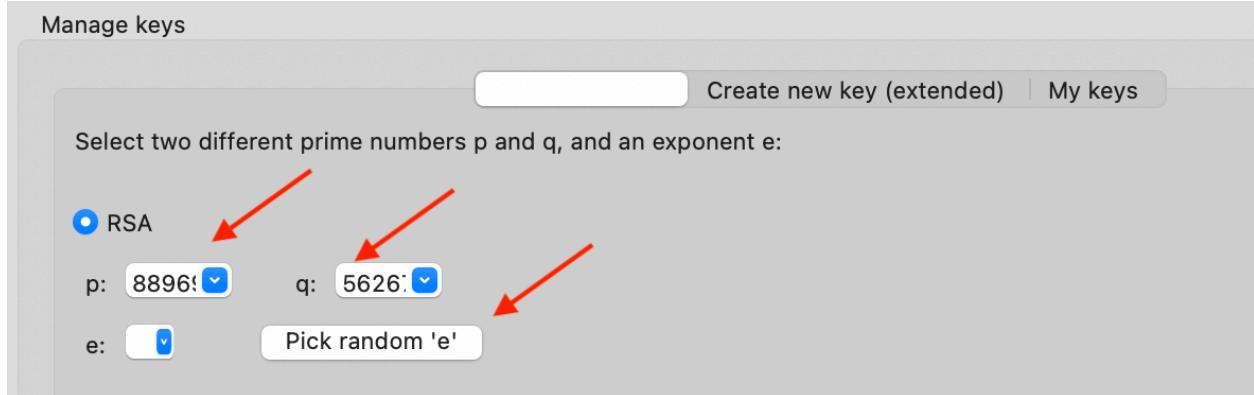
To do this task we will use JCrypTools. Once in there, we select the option `Visuals -> Extended RSA Cryptosystem`:

- Android Unlock Pattern (AUP)
- Ant Colony Optimization (ACO)
- ARC4 / Spritz
- Certificate Verification
- Chinese Remainder Theorem (CRT)
- Diffie-Hellman Key Exchange (EC)
- Elliptic Curve Calculations
- Extended Euclidean / Reciprocal Subtraction
- Extended RSA Cryptosystem
- Hash Sensitivity
- Homomorphic Encryption (HE)
- Huffman Coding
- Inner States of the Data Encryption Standard (DES)
- Kleptography
- McEliece Cryptosystem
- Merkle Signatures (XMSS^{MT})
- Merkle-Hellman Knapsack Cryptosystem
- Multipartite Key Exchange (BD II)
- Multivariate Cryptography
- Public-Key Infrastructure
- Redactable Signature Schemes (RSS)
- Shamir's Secret Sharing
- Shanks Babystep-Giantstep
- Signature Demonstration
- Signature Verification
- Simple Power Analysis / Square and Multiply
- SPHINCS Signature
- SPHINCS+ Signature
- SSL/TLS Handshake
- Verifiable Secret Sharing
- Winternitz OT-Signature (WOTS / WOTS+)
- Zero-Knowledge: Feige Fiat Shamir
- Zero-Knowledge: Fiat Shamir
- Zero-Knowledge: Graph Isomorphism
- Zero-Knowledge: Magic Door

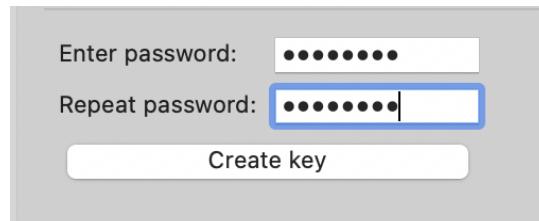
Then, we click on Alice whitehat and select the `Manage keys` option:



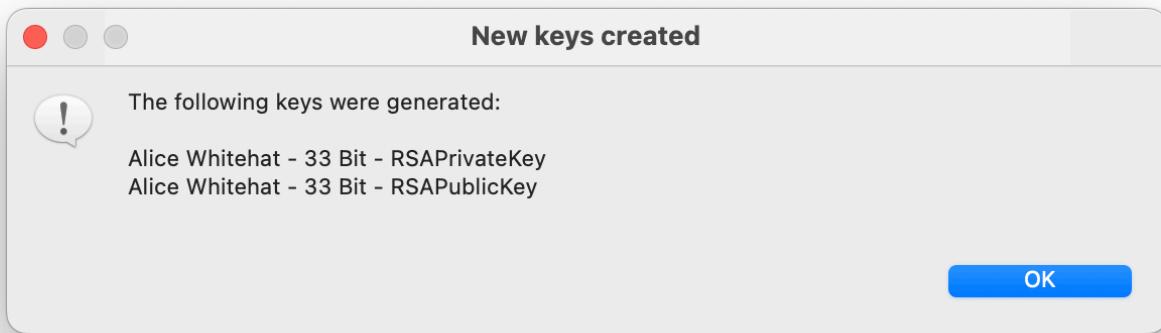
And proceed to enter the primes and select `Pick random 'e'`. The chosen primes are **88969** and **56267**:



Then a password is selected and we click on `Create key`:



The length of the key depends on the chosen prime numbers:



Once created, we can proceed to attack it:

Attack public key

Select a public key::

Alice Whitehat - 1024Bit - RSA PublicKey - KeyID: 5
Alice Whitehat - 1024Bit - RSA PublicKey - KeyID: 1
Bob Whitehat - 1024Bit - RSA PublicKey - KeyID: 2
Alice Whitehat - 33Bit - RSA PublicKey - KeyID: 3
Bob Whitehat - 1024Bit - RSA PublicKey - KeyID: 4

Attack key

The attack is successful and surprisingly fast due to the relatively *small* size of the chosen primes:

Attack public key

Select a public key::

Alice Whitehat - 33Bit - RSA PublicKey - KeyID: 3

Attack key

Reconstruct key

Press the button "Attack key" to try a factorizing attack on the modulus N of Alice Whitehat (bit length of N: 33 bit):

N: 5006018723

The factorization was successful. The following values were reconstructed in 0.093 seconds:

Parameter	Value
p	56267
q	88969
e	3551663305
d	4873146601

We can read the explanation provided by JCrypTools to understand what has happened:

Explanations

Action: Attack public key

In this tab you can try to attack a public key. If the modulus N is sufficiently short (less than 256 bit of length), it is feasible to factor N in reasonable time. The factorization results the prime numbers which had been used to calculate the parameters of the private key, and with these prime numbers the private key can be re-calculated. So you can impersonate the original key owner and read messages which are meant to be confidential for the original key owner.

Attention: The factorization can take a long time (depending on the size of N and your hardware). With actual hardware you can factorize the parameter N with a bit length of approximately 64 bit in one hour (tested on a notebook with Intel Core i5 2.4Ghz and 8GB RAM).

As the security of RSA keys is assumed to be based on the length of the keys, a key length of more than 1024 bit is recommended for encryption.

It says a key **length** of more than **1024 bits is recommended**, something that our key clearly didn't satisfy.

There are also other online tools that can be used to factor numbers in primes such as

<https://www.calculatorsoup.com/calculators/math/prime-factors.php>:

Prime Factors Calculator

Enter the Number to Factor

create a factorization tree
 also show me all factors

Clear **Calculate**

Answer:

The Prime Factorization is:
56267 x 88969

In Exponential Form:
56267¹ x 88969¹

CSV Format:
56267, 88969

Factorization in 2.187014e-3 seconds

Question: to factor a number N using the exhaustive-search, i.e., try factors in 1,2,3...N how many numbers at most should be tried?

Supposing the question means to find p and q such that $p \cdot q = N$ when it says factor a number N, then the maximum number of primes that should be tried is $\sqrt{N}/2$. The reason for this is that if N has a prime factor which is greater than its square root then it has prime factor that is smaller than its square root. This means that we would have found the smaller prime before reaching the bigger prime. The half comes due to the fact that all even numbers are not prime. If after trying $\sqrt{N}/2$ numbers we haven't obtained a prime then N would be prime which cannot be as $N = p \cdot q$ and p and q cannot be 1 as they are prime. The actual number varies depending on the value of N as the distribution of prime number is not homogeneous on the real line. It could be also used that approximately the number of primes smaller than N is $N/\ln(N)$.

Part Six: RSA Encryption and Decryption

6.1. Generate RSA private and public key pair:

Let's now work with openssl. First we generate a private key:

```
openssl genrsa -out private.pem 2048
cat private.pem
```

```

[base) javiergarru@MBP-de-Javier-2 w8 % openssl genrsa -out private.pem 2048
Generating RSA private key, 2048 bit long modulus (2 primes)
.....+++++
.e is 65537 (0x010001)
[base) javiergarru@MBP-de-Javier-2 w8 % cat private.pem
-----BEGIN RSA PRIVATE KEY-----
MIIEogIBAAKCAQEAvT65Un4/ZIV/rwlt2/Nviw1PloJprFI3hNHa2zrFXiu//yIM
6Cj7yDdOCb6bCSz8yudAMi2RDEilC3SZEAmKsKH1Q5EOVpsFirroUe/7U2m6ejpUp
Jgzc1MB/Gbbp408cfrJ6iqtJRy3V9812UmhpYYqbVmzbaXOcvDv42ByW/j+FNol
yOq2CL/YqHEBxj12oBV4N0D3AM8d0+i0FRh/6ky7uT8zV0Ven66977mIy/k1nX5S
HHC4nWgHTv69r7WMwzLrSuGaNq7ivaxUWpS8wqx5RhyH6fJe6dyRjRK90Bu1Gdo0
DTGi/eJqCX9gTsqHBfHAPKb2MdRA0Uod0ADQwIDAQABoIBACv00FGzXfNkH/mi
Aidvg/u5ehoKWHIGmkMJ1tL3G0dC03A6r9uyWQ6xFsUTgj6vA4wWCEFgVVJn8M2
G5Sl0NAwvbkFD9SjY+cDc/iXt2ty9CP4PW09pnndcc6/FAb48yy1Hj7phc4PYz+4
CEbtWq+0yV7IpAeI4XCK7QECJUXeZvywrgJZ/DWlchZs3xhHiTZCi+S6IbLejdW
F082b1/Tw6KGDX8b1frtl sAX1dGkMDU1SNvbRBCpr5xjB4TQoif/Yks/ADnKtdkv
0QsAR1Av1Sn1Y8GnjbGc35mo49h/LF+8ztFBsKdGBYb0AUPJN8QCK05KY2+0tBtK
WZTT03kCgYEAS5vGtZFDSeKw+BTWUbGzuPdDbo vV8w6zxw38LBuOnULB7QXvraEbj
U77DN6DgZQyHL4c7YSxGU30WPA+tNVovXRGWyxmJn5LOhbUt/cUiBah6b31+B+sR
x2y4mjWZf0zwtF2WeDLAk rwXLzHM9wERE A+569Zft3/ehzD8IWKwPs8CgYEAcbh
9Q+Bc5DnbJMmoW30EzJ+i2vmYtnDHF4LLFM7pjgPnisp+kBa jyF0YXFp7DCJblRm
tqs3Ea/5caTBh2RLIHPXP5X06wSek+5Cdt4Ng8edSRGb2wPMo/TFC24Trw3lHJis
ckNsY7soyMrHc9HgNzNZ/WGU9KYt4jcMgPowsU0CgYB1sZ6y6/mHILhpQTgF1bHI
cQMt1juiISYIglskcKj5FyeJYYsjXi nHkwU7+VHCTZMrZpidWgnxB4v9yeN0uCRB
IvCdSsN3PkKqoyXwGSHQj17vJpRTBrpFzqN5e21hk d1tnfBMC11WrmKGW7PgAXwn
mkwe415eTt0iOpwTk g7NnwKBgC+AUxokYLPChCTZu2QkEwoiu8XJ7wKg1vRD8ah
skgZobAfJgnef7/pW2kVPV72Pqh1BC0o5/dhWaKaJBh8ZWab1InZalldwms08Gfq
MDU+5ia9jgUx/wzBSD0mV96zzsWC7MmS6vfXEY0mHbPk011Fj4MIraoYVxt6S6KJ
BK2hAoGAEREGUpRSU6+uBB7okwxIDtZ9zA/hipVgyMKxgTgrn7NEDq9azw/5/Zo0
8798GG22Hor rjP9emZMyfWnz6kISz8311MGYdsrCP1bD0DpvGLihURIwcrMMWWQ3
1nMU34qLEFFLDuWVvf7Kg3ahOTDi7KHHrP+ys2YbuC2i07iwwCI=
-----END RSA PRIVATE KEY-----
(base) javiergarru@MBP-de-Javier-2 w8 %

```

The format is not what we were expecting due to us saving the key in a .pem file which follows a specific standard that encodes the keys. We can recover the values in an online site

http://certificate.fyicenter.com/2145_FYICenter_Public_Private_Key_Decoder_and_Viewer.html

We copy and paste the key:

Not Secure — cate.fyicenter.com

Key in PEM Format:

```
-----BEGIN RSA PRIVATE KEY-----  
MIIEogIBAAKCAQEAvT65Un4/ZIV/rwl...  
6Cj7yDdOCb6bCSz8yudAMi2RDEilC3SZEAmKsKH1Q5EOVpsFrroUe/7U2m6ejpUp  
Jgzc1MB/Gbbp4O8cfrJ6iqtJRy3V9812UmhpYYqbVm...  
yOq2CL/YqHEBxj12oBV4N0D3AM8d0+iORh/6ky7uT8zv0Ven66977mIy/k1nx5S  
HHC4nWgHTv69r7WMwzLrSuGa...  
DTGi/eJqCX9gTs...  
Aidvg/u5ehoKWHIGmk...  
-----END RSA PRIVATE KEY-----
```

Name your key: My Key

(All fields are required.)

Or select [RSA Public Key](#), [DSA Public Key](#), [DH Public Key](#) or [EC Public Key](#) to try sample public keys.

Or select [RSA Private Key](#), [DSA Private Key](#), [DH Private Key](#) or [EC Private Key](#) to try sample private keys.

FYIcenter.com

A FYIcenter.com Decoded Result:

Specified Key: **Valid ✓**

Private Key Detailed Information:

Key Details:

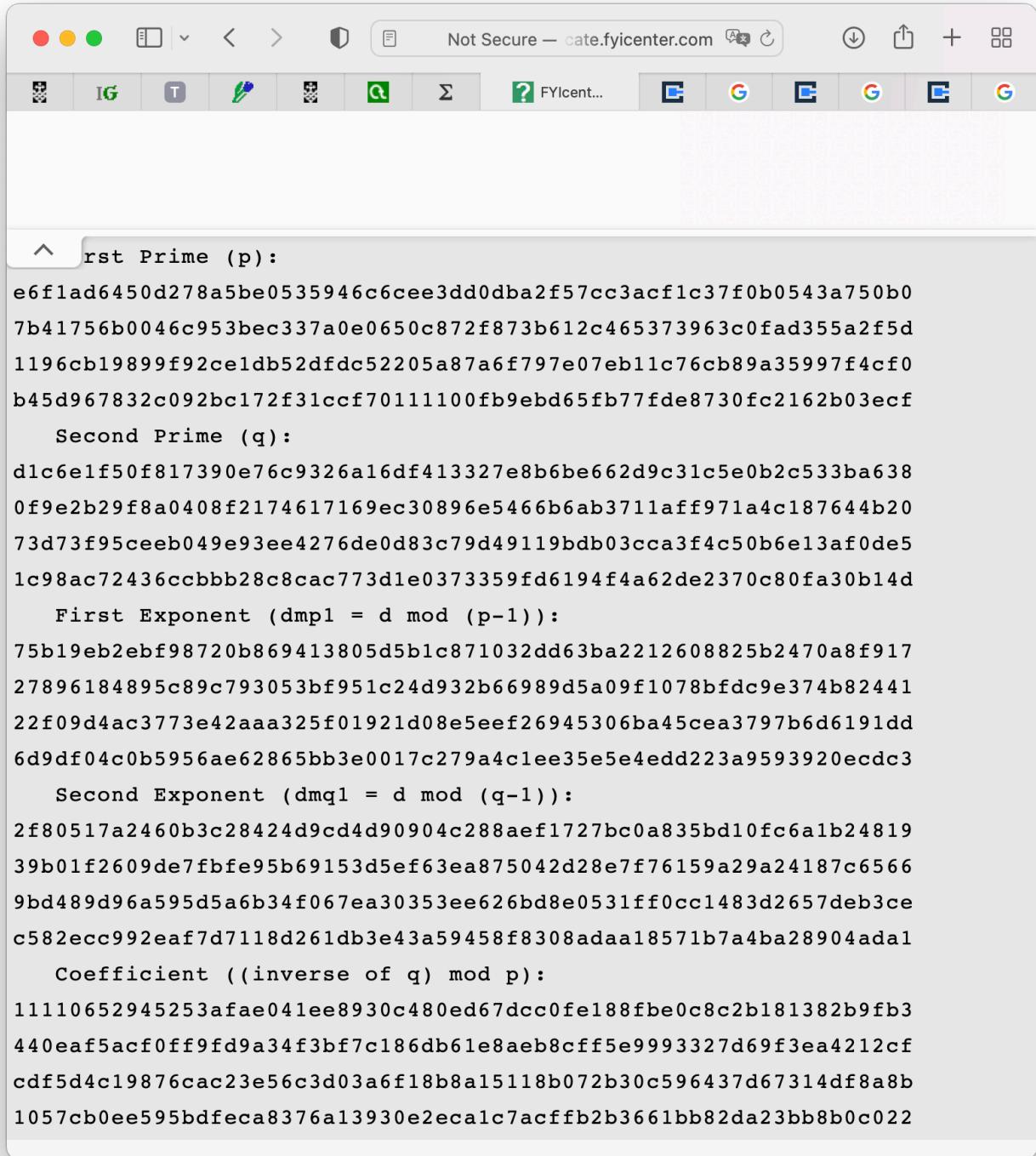
Type: RSA

Size (bits): 2048

And the read the generated values:

The screenshot shows a web browser window with the URL "Not Secure — cete.fyicenter.com". The main content area displays "Private Key Detailed Information" for a RSA key. The key details are as follows:

- Type:** RSA
- Size (bits):** 2048
- Modulus (n):**
bd3eb9527e3f64857faf096ddbf36f8b0d4f968269ac523784d1dadb3ac55e2b
bfff288ce828fbc8374e09be9b092cfccae740322d910c48a50b749910098ab0
a1f543910e569b05aeba147bfed4da6e9e8e9529260cdcd4c07f19b6e9e0ef1c
7eb27a8aab49472dd5f7c976526869618a9b5669d96da5ce72f0efe360725bf8
fe14da0bc8eab608bfd8a87101c63d76a015783740f700cf1d3be88e15187fea
4ccb93f3357455e9faebdefb988cbf9359d7e521c70b89d68074efebdafb58c
c332eb4ae19a36aee2bdac545a94bcc2ac79461c87e9f25ee9dc918d12bd381b
b519da340d31a2fde26a097f604eca870458403ca6f631d4403ba52877400343
- Public Exponent (e):** 65537 (0x010001)
- Private Exponent (d):**
2bcd051b35df3641ff9a202276f83fb97a1a0a5872069a4989d6d2f718e742
d3703aafdbb2590eb116c494b608fabcoe305821058155499fc3361b94a5d0d0
30bdb905643f52258f9c0dcfe25eddadcbd08fe0f5a8f699dd71cebf1406f8f3
2cb51e3ee985ce0f633fb80846ed5aafb4c95ec8a40788e170a42bb404089517
799bf2c2b80967f0d621c859b37c611e24d90a2f92e886cb7a377014ef366f5f
d3c3a2860d7f1b95faed96c017d5d1a430353548dbdb4410a9af9c630784d03a
27ff624b3f0039cab5d92fd10b0047502f9529f563c1a78db19cdf99a8e3d87f
2c5fbcced141b0a7460586ce0143c937c40290ee4a636fb4b41b4a5994d33b79
- First Prime (p):**



The screenshot shows a web browser window with the URL "Not Secure — cate.fyicenter.com". The page content displays a large string of hexadecimal digits, which are the components of an RSA key. The string is organized into sections:

- rst Prime (p):**
e6f1ad6450d278a5be0535946c6cee3dd0dba2f57cc3acf1c37f0b0543a750b0
7b41756b0046c953bec337a0e0650c872f873b612c465373963c0fad355a2f5d
1196cb19899f92ce1db52dfdc52205a87a6f797e07eb11c76cb89a35997f4cf0
b45d967832c092bc172f31ccf70111100fb9ebd65fb77fde8730fc2162b03ecf
- Second Prime (q):**
d1c6e1f50f817390e76c9326a16df413327e8b6be662d9c31c5e0b2c533ba638
0f9e2b29f8a0408f2174617169ec30896e5466b6ab3711aff971a4c187644b20
73d73f95ceeb049e93ee4276de0d83c79d49119bdb03cca3f4c50b6e13af0de5
1c98ac72436ccb8bb28c8cac773d1e0373359fd6194f4a62de2370c80fa30b14d
- First Exponent (dmp1 = d mod (p-1)):**
75b19eb2ebf98720b869413805d5b1c871032dd63ba2212608825b2470a8f917
27896184895c89c793053bf951c24d932b66989d5a09f1078bfd9e374b82441
22f09d4ac3773e42aaa325f01921d08e5eef26945306ba45cea3797b6d6191dd
6d9df04c0b5956ae62865bb3e0017c279a4c1ee35e5e4edd223a9593920ecdc3
- Second Exponent (dmq1 = d mod (q-1)):**
2f80517a2460b3c28424d9cd4d90904c288aef1727bc0a835bd10fc6a1b24819
39b01f2609de7fbfe95b69153d5ef63ea875042d28e7f76159a29a24187c6566
9bd489d96a595d5a6b34f067ea30353ee626bd8e0531ff0cc1483d2657deb3ce
c582ecc992eaf7d7118d261db3e43a59458f8308adaa18571b7a4ba28904ada1
- Coefficient ((inverse of q) mod p):**
11110652945253afae041ee8930c480ed67dcc0fe188fbe0c8c2b181382b9fb3
440eaf5acf0ff9fd9a34f3bf7c186db61e8aeb8cff5e9993327d69f3ea4212cf
cdf5d4c19876cac23e56c3d03a6f18b8a15118b072b30c596437d67314df8a8b
1057cb0ee595bdfeca8376a13930e2eca1c7acffb2b3661bb82da23bb8b0c022

We can also use the website <https://8gwifi.org/PemParserFunctions.jsp>. We type in the following key:

Public Key in PEM Format:

-----BEGIN PUBLIC KEY-----

MIIIBIjANBgkqhkiG9w0BAQEFAOCAQ8AMIIBCgKCAQEAvT65Un4/ZIV/rwl t2/Nv
iw1PloJprFI3hNHa2zrFXiu//yIM6Cj7yDdOCb6bCSz8yudAMi2RDEilC3SZEAmK
sKH1Q5EOVpsFrroUe/7U2m6ejpUpJgz c1MB/Gbbp408cfrJ6iqtJRy3V9812Umhp
YYqbVm nZbaXOcvDv42ByW/j+FNoLy0q2CL/YqHEBxj12oBV4N0D3AM8d0+iOFRh/
6ky7uT8zV0Ven66977mIy/k1nX5SHHC4nWgHTv69r7WMwzLrSuGa Nq7ivaxUWpS8
wqx5RhyH6fJe6dyRjRK9OBu1Gdo0DTGi/eJqCX9gTs qHBFhAPKb2MdRAO6Uod0AD
QwIDAQAB

-----END PUBLIC KEY-----

And click submit:

The screenshot shows a web browser window with the URL 8gwifi.org in the address bar. A blue banner at the top reads "Grab 9 Book (5 Cryptography + 4 Devops/Kubernetes) for JUST \$9 >". Below the banner, the page title is "8gwifi.org - Crypto Playground". A button on the right says "Follow @anish2good". A text input field labeled "Cert Password (if any)" is present. A "Submit" button is below it. A large text box contains RSA key information:

```
Algo RSA
Format X.509
ASN1 Dump
RSA Public Key [50:04:ea:17:2a:a3:e9:bb:4d:a1:d0:74:12:cf:74:4f:ae:57:b2:67]
    modulus:
bd3eb9527e3f64857faf096ddbf36f8b0d4f968269ac523784d1adb3ac55e2bfff288ce8
28fb8374e09be9b092cfccae740322d910c48a50b749910098ab0a1f543910e569b05aeb
a147bfed4da6e9e8e9529260cdcd4c07f19b6e9e0ef1c7eb27a8aab49472dd5f7c9765268
69618a9b5669d96da5ce72f0efe360725bf8fe14da0bc8eab608bfd8a87101c63d76a01578
3740f700cf1d3be88e15187fea4cbbb93f3357455e9faebdefb988cbf9359d7e521c70b89d
68074efebdafb58cc332eb4ae19a36aee2bdac545a94bcc2ac79461c87e9f25ee9dc918d1
2bd381bb519da340d31a2fde26a097f604eca870458403ca6f631d4403ba52877400343
    public exponent: 10001
```

Below the text box are social sharing buttons for Facebook, Twitter, Pinterest, Email, LinkedIn, and Google+.

Question: Try the following tools and confirm you can recover RSA components

From this website we can extract the RSA components. The values of n,d,p and q are immense and we display them here in their hexadecimal and decimal format:

e:

```
# IN DECIMAL
65537
```

d:

```
# IN DECIMAL
553023077299816581929082812905903390868704914919706256920642386559949931606011391054553
208286778086128938620913019849572637116543086216660423540643713219033018377917389630121
004788547412659996664867799624661916489792565225317423275711972891990186489420353882579
857489877009570136002453060140912787207817581007557985300127506902703986719913604617828
753248766080048956434689007395014022863621807321403948199093598334508066696854537520881
115204198013677612210825837223145314728957959027037137039448581936000340863396881856356
111005952562811083462137007744251672492748947702292492436887118840942797706252583590928
3396473
```

```
# IN HEX
2bc051b35df3641ff9a202276f83fbb97a1a0a5872069a4989d6d2f718e742
d3703aaafdbb2590eb116c494b608fab0e305821058155499fc3361b94a5d0d0
30bdb905643f52258f9c0dcfe25eddadcb08fe0f5a8f699dd71cebf1406f8f3
2cb51e3ee985ce0f633fb80846ed5aab4c95ec8a40788e170a42bb404089517
799bf2c2b80967f0d621c859b37c611e24d90a2f92e886cb7a377014ef366f5f
d3c3a2860d7f1b95faed96c017d5d1a430353548dbdb4410a9af9c630784d03a
27ff624b3f0039cab5d92fd10b0047502f9529f563c1a78db19cdf99a8e3d87f
2c5fbcced141b0a7460586ce0143c937c40290ee4a636fb4b41b4a5994d33b79
```

n:

```
# IN DECIMAL
238899699538580708785751106119663769872535192202839555466403929127805936771888264027040
100266894571423322459948431757177786030630045754276410108649179574423353275549218668441
370317230451410574131075347597399433267362305366631256800608632044178767727619416863770
589416083775467655416207014715279159813056099182399624432009184462379865590323793651905
385718220935679108680299506179579625608015218564646152849551311210212298528439865552969
974184658794204623508000091452666142405067745385038121527389604717322170905516249816872
839769994184065999971555700493940804965254994088328530345263160331033166760078774042346
21125443
```

```
# IN HEX
bd3eb9527e3f64857faf096ddb36f8b0d4f968269ac523784d1dad3ac55e2b
bfff288ce828fbc8374e09be9b092cfccae740322d910c48a50b749910098ab0
a1f543910e569b05aeba147bfed4da6e9e8e9529260cdcd4c07f19b6e9e0ef1c
7eb27a8aab49472dd5f7c976526869618a9b5669d96da5ce72f0efe360725bf8
fe14da0bc8eab608bfd8a87101c63d76a015783740f700cf1d3be88e15187fea
4cbbb93f3357455e9faebdefb988cbf9359d7e521c70b89d68074efebdafb58c
c332eb4ae19a36aee2bdac545a94bcc2ac79461c87e9f25ee9dc918d12bd381b
b519da340d31a2fde26a097f604eca870458403ca6f631d4403ba52877400343
```

p:

```
# IN DECIMAL
162174428444261123897833406675287093215145740839845914043759244991833062109813200715192
475680887384379020555080038015469458026049937325153929567808717842030507786289200269381
992955534694844897372446491764140886841665167011134562043574487815839313669940206236376
981687593830007920038694029093897844403886636751
```

```
# IN HEX
e6f1ad6450d278a5be0535946c6cee3dd0dba2f57cc3acf1c37f0b0543a750b0
7b41756b0046c953bec337a0e0650c872f873b612c465373963c0fad355a2f5d
1196cb19899f92ce1db52dfdc52205a87a6f797e07eb11c76cb89a35997f4cf0
b45d967832c092bc172f31ccf70111100fb9ebd65fb77fde8730fc2162b03ecf
```

q:

```
# IN DECIMAL
147310338522752886322414586894711475825736907401480012862258834254495365146784636478415
077952555158256448202848010654925491242349730736978063309233624654166308973300069462271
562644337934471045759378838742370756094046437041798372965903349269550839080479094763799
391114635320083007614178949547506966858174607693
```

```
# IN HEX
d1c6e1f50f817390e76c9326a16df413327e8b6be662d9c31c5e0b2c533ba638
0f9e2b29f8a0408f2174617169ec30896e5466b6ab3711aff971a4c187644b20
73d73f95ceeb049e93ee4276de0d83c79d49119bdb03cca3f4c50b6e13af0de5
1c98ac72436ccb828c8cac773d1e0373359fd6194f4a62de2370c80fa30b14d
```

The size of the numbers make impossible visually checking if n is indeed the product of p and q. We can use the following website to check it <https://www.dcode.fr/big-numbers-multiplication>:

As we can see the obtained value for n is the expected value.

Search for a tool

★ SEARCH A TOOL ON DCODE BY KEYWORDS:
e.g. type 'boolean'

★ BROWSE THE FULL DCODE TOOLS' LIST

Results

```
238899699538580708785751106119663769872535192
202839555466403929127805936771888264027040100
266894571423322459948431757177786030630045754
276410108649179574423353275549218668441370317
230451410574131075347597399433267362305366631
256800608632044178767727619416863770589416083
775467655416207014715279159813056099182399624
432009184462379865590323793651905385718220935
679108680299506179579625608015218564646152849
551311210212298528439865552969974184658794204
623508000091452666142405067745385038121527389
604717322170905516249816872839769994184065999
971555700493940804965254994088328530345263160
33103316676007877404234621125443
```

MULTIPLICATION
Mathematics > Arithmetics > Multiplication

NAVY **APPLY NOW >**

MULTIPLICATION OF 2 NUMBERS

★ NUMBER 1 162174428444261123897833406675287093215...
 ★ NUMBER 2 147310338522752886322414586894711475825...

► MULTIPLY

See also: Division – Exponentiation (Power)

MULTIPLY MANY NUMBERS

★ MULTIPLY MANY NUMBERS

	Numbers
1	...
2	...
3	...
...	

► MULTIPLY

CALCULATION WITH MULTIPLICATION

★ MATHEMATICAL EXPRESSION TO CALCULATE
1*23*456

★ RESULT FORMAT AUTOMATIC SELECTION
 EXACT VALUE (WHEN POSSIBLE)
 APPROXIMATE NUMERICAL VALUE
 SCIENTIFIC NOTATION

► MULTIPLY

stayz™

To conclude we can derive the public key from the private key:

```
openssl rsa -in private.pem -outform PEM -pubout -out public.pem
ls
cat public.pem
```

```
w8 -- zsh -- 116x17
[(base) javiergarru@MBP-de-Javier-2 w8 % openssl rsa -in private.pem -outform PEM -pubout -out public.pem
writing RSA key
[(base) javiergarru@MBP-de-Javier-2 w8 % ls
Cryptography - Lab 5 - Asymmetric Crypto.md      private.pem
RSA.py                                              public.pem
[(base) javiergarru@MBP-de-Javier-2 w8 % cat public.pem
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAvT65Un4/ZIV/rwlz2/Nv
iw1Pl0JprFI3hNHa2zrFXiu/yiM6Cj7yDd0cb6bCs8yudAMi2RDEilC3SZEAmK
sKH1Q5EOVpsFrroUe/7U2m6ejpUpJgzc1MB/GbbpA08cfriJ6lqtJRy3V98l2Umhp
YYqbVmnZbaXcvDv42ByW/j+FN0Ly0q2CL/YqHEBxj12oBV4N0D3AM8d0+i0FRh/
6ky7uT8zV0Ven66977mIy/k1nX5SHHC4nWgHTv69r7WMwzLrSuGaNq7ivaxUWpS8
wqx5RhYH6fJe6dyRjRK90Bu1Gdo0DTGi/eJqCX9gTsqHBFhAPKb2MdRA06Uod0AD
QwIDAQAB
-----END PUBLIC KEY-----
(base) javiergarru@MBP-de-Javier-2 w8 %
```

6.2. Encrypt a small file with the public key and decrypt with the private key:

We encrypt using the public key a file called secret and save it in another one called cipher:

```
echo "What's up blue kagaroo" > secret
openssl rsautl -encrypt -inkey public.pem -pubin -in secret -out cipher
ls
cat cipher
```

```
w8 -- zsh -- 116x33
[(base) javiergarru@MBP-de-Javier-2 w8 % echo "What's up blue kangaroo" > secret
[(base) javiergarru@MBP-de-Javier-2 w8 % openssl rsautl -encrypt -inkey public.pem -pubin -in secret -out cipher
[(base) javiergarru@MBP-de-Javier-2 w8 % ls
Cryptography - Lab 5 - Asymmetric Crypto.md      private.pem
RSA.py                                              public.pem
cipher                                              secret
[(base) javiergarru@MBP-de-Javier-2 w8 % cat cipher
$?9?TE??~RKMM?(2?_????pVt?sm-? ,?j?'?X?}r???,{x??j?u=H~??_p?'?dDd`?]?-E?? ?????
?????@?#w??3sE?????????????
1F???Pi?t|?
2$??7?M_ ?0?u(???N??f&y\b?"f
            3:[]??????
1?
<?I>?6???o,??#f??XWVE1?k?Q5?a??I[?(J?????
[ ?UK??8b??&#
(base) javiergarru@MBP-de-Javier-2 w8 % hexdump -C cipher
00000000  83 24 39 cd 54 45 f6 e4  7e 52 4b 4d 4d b4 28 7f |.$9?TE??~RKMM?(.|.
00000010  32 ed 5f e1 fd c9 ce 70  d1 b6 74 b0 da 8e 6d 2d |2?-????pVt??m-|
00000020  96 10 9d 20 dc b9 e0 e2  6a df 27 a0 58 b8 7d bc |...,??j?'?X?}?-|
00000030  72 a2 bc f1 12 2c 7b 78  90 1d f4 6a ed 75 16 3d |r???,,x..?j?u=|
00000040  d4 a8 7e 9c 97 5f 70 05  da 13 27 d6 b7 88 c7 64 |H~.._p.?,'?d|
00000050  44 64 60 00 b4 5d f1 91  2d 45 10 ac 02 da 20 b7 |Dd'?.?]-E?.? .? |
00000060  05 c3 f6 b5 c5 d7 be 4c  b8 ba 01 d5 87 23 77 8c |.?????@??.?..#w..|
00000070  91 33 73 45 97 c1 07 90  1d c8 ff b9 8e b4 8c 9a |.3sE?.?...???.?..|
00000080  ef 0b 6c 46 ba 3f 99 50  69 a3 74 7c 15 ee b2 32 |?1F??Pi?t|..22|
00000090  24 db db c5 4d 5f 16 09  14 a6 4f cc 75 28 85 8c |$??M_...?0?u(..|
000000a0  a9 4e bc ff 66 26 e5 79  5c 62 84 81 22 66 0c 33 |?N??f&y\b.."f..3|
000000b0  3a 5b 7d c0 03 be ae 0f  99 bc 0b 03 3c db 49 b3 |:[]?..?..?..<?I?|
000000c0  3e d2 36 cf fa cb 6f 11  2c 05 b0 9b 23 66 89 f8 |>?6????.,?.#f..?|
000000d0  58 57 56 45 0e 6c ce 6b  a0 1e 51 35 e3 84 61 b7 |XWVE.1?k?.Q5?.a?|
000000e0  8f 49 5b 02 ef bc 28 03  4a 9d de 37 f6 ab 0d 6c |.I[?.(J.???.1|
000000f0  d7 0c 89 e0 55 4b fc 8b  38 ac 62 07 fd b2 18 26 |?..?UK?.8?b.??.&|
00000100
(base) javiergarru@MBP-de-Javier-2 w8 %
```

Now, we can proceed to decrypt it using the private key:

```
openssl rsautl -decrypt -inkey private.pem -in cipher -out plain.txt  
ls  
cat plain.txt
```



w8 -- zsh -- 116x9

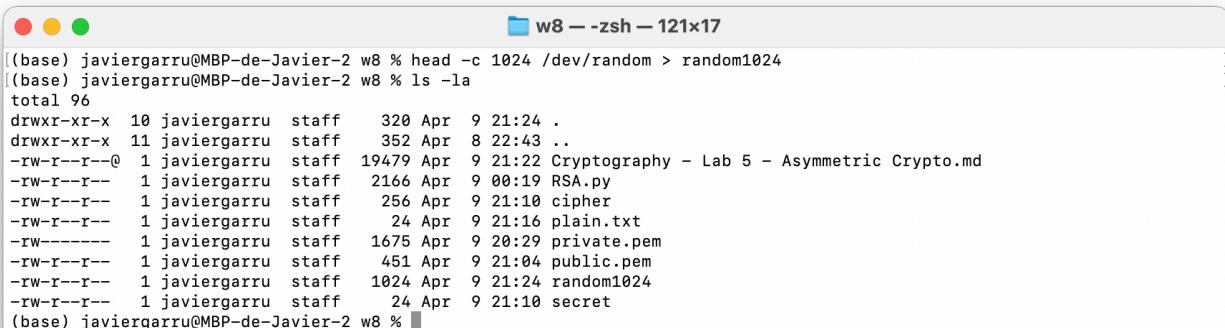
```
[(base) javiergarru@MBP-de-Javier-2 w8 % openssl rsautl -decrypt -inkey private.pem -in cipher -out plain.txt  
[(base) javiergarru@MBP-de-Javier-2 w8 % ls  
Cryptography - Lab 5 - Asymmetric Crypto.md      private.pem  
RSA.py                                              public.pem  
cipher                                              secret  
plain.txt  
[(base) javiergarru@MBP-de-Javier-2 w8 % cat plain.txt  
What's up blue kangaroo  
[(base) javiergarru@MBP-de-Javier-2 w8 % ]
```

As we can see the plaintext that was contained in the file has been correctly deciphered.

6.3.Size limitations of RSA encryption:

RSA is not usually used for encrypting large files. Encrypting a 1024 bytes file with a 2048-bit key will produce an error. We first create a file full of 1024 random bytes and call it random1024:

```
head -c 1024 /dev/random > random1024
```



w8 -- zsh -- 121x17

```
[(base) javiergarru@MBP-de-Javier-2 w8 % head -c 1024 /dev/random > random1024  
[(base) javiergarru@MBP-de-Javier-2 w8 % ls -la  
total 96  
drwxr-xr-x 10 javiergarru staff 320 Apr 9 21:24 .  
drwxr-xr-x 11 javiergarru staff 352 Apr 8 22:43 ..  
-rw-r--r--@ 1 javiergarru staff 19479 Apr 9 21:22 Cryptography - Lab 5 - Asymmetric Crypto.md  
-rw-r--r-- 1 javiergarru staff 2166 Apr 9 00:19 RSA.py  
-rw-r--r-- 1 javiergarru staff 256 Apr 9 21:10 cipher  
-rw-r--r-- 1 javiergarru staff 24 Apr 9 21:16 plain.txt  
-rw-r----- 1 javiergarru staff 1675 Apr 9 20:29 private.pem  
-rw-r--r-- 1 javiergarru staff 451 Apr 9 21:04 public.pem  
-rw-r--r-- 1 javiergarru staff 1024 Apr 9 21:24 random1024  
-rw-r--r-- 1 javiergarru staff 24 Apr 9 21:10 secret  
[(base) javiergarru@MBP-de-Javier-2 w8 % ]
```

Then we try to encrypt it using our 2048-bit key:

```
openssl rsautl -encrypt -inkey public.pem -pubin -in random1024 -out randomCipher1024
```



w8 -- zsh -- 133x5

```
(base) javiergarru@MBP-de-Javier-2 w8 % openssl rsautl -encrypt -inkey public.pem -pubin -in random1024 -out randomCipher1024
RSA operation error
8643737088:error:0406D06E:rsa routines:RSA_padding_add_PKCS1_type_2:data too large for key size:crypto/rsa/rsa_pk1.c:125:
(base) javiergarru@MBP-de-Javier-2 w8 %
```

And an error shows indicating that the dat is too large for our key. With smaller values like 100 this won't happen:



w8 -- zsh -- 157x18

```
(base) javiergarru@MBP-de-Javier-2 w8 % openssl rsautl -encrypt -inkey public.pem -pubin -in random100 -out randomCipher100
(base) javiergarru@MBP-de-Javier-2 w8 % ls -la
total 112
drwxr-xr-x 13 javiergarru staff 416 Apr 9 22:37 .
drwxr-xr-x 11 javiergarru staff 352 Apr 8 22:43 ..
-rw-r--r--@ 1 javiergarru staff 20442 Apr 9 22:37 Cryptography - Lab 5 - Asymmetric Crypto.md
-rw-r--r-- 1 javiergarru staff 2166 Apr 9 00:19 RSA.py
-rw-r--r-- 1 javiergarru staff 256 Apr 9 21:10 cipher
-rw-r--r-- 1 javiergarru staff 24 Apr 9 21:16 plain.txt
-rw----- 1 javiergarru staff 1675 Apr 9 20:29 private.pem
-rw-r--r-- 1 javiergarru staff 451 Apr 9 21:04 public.pem
-rw-r--r-- 1 javiergarru staff 100 Apr 9 21:29 random100
-rw-r--r-- 1 javiergarru staff 1024 Apr 9 21:24 random1024
-rw-r--r-- 1 javiergarru staff 256 Apr 9 22:37 randomCipher100
-rw-r--r-- 1 javiergarru staff 0 Apr 9 21:25 randomCipher1024
-rw-r--r-- 1 javiergarru staff 24 Apr 9 21:10 secret
(base) javiergarru@MBP-de-Javier-2 w8 %
```

Question: How many bytes at maximum can be encrypted with 2048-bit key RSA? Why?

The maximum encryptable size is the same length of the key, that is $256-11= 245$ bytes (this value changes depending on the padding algorithm being used). This is because before encrypting, the message must be converted into a number including 11 bytes of padding for security reasons. If the message is larger it cannot be converted to a number. As the documentation of openssl rsautl indicates, the RSA algorithm is directly used on the message (no blocking mode is used). If the message cannot be converted into a 245 byte-number, then when applying the module to the power of that number the encrypted value will match the encrypted value of another number too (this was seen when working with the script before). We can test that 11bytes is the exact padding value easily:



w8 -- zsh -- 157x8

```
(base) javiergarru@MBP-de-Javier-2 w8 % head -c 245 /dev/random > random245
(base) javiergarru@MBP-de-Javier-2 w8 % openssl rsautl -encrypt -inkey public.pem -pubin -in random245 -out randomCipher245
(base) javiergarru@MBP-de-Javier-2 w8 % head -c 246 /dev/random > random246
(base) javiergarru@MBP-de-Javier-2 w8 % openssl rsautl -encrypt -inkey public.pem -pubin -in random246 -out randomCipher246
RSA operation error
8644122112:error:0406D06E:rsa routines:RSA_padding_add_PKCS1_type_2:data too large for key size:crypto/rsa/rsa_pk1.c:125:
(base) javiergarru@MBP-de-Javier-2 w8 %
```

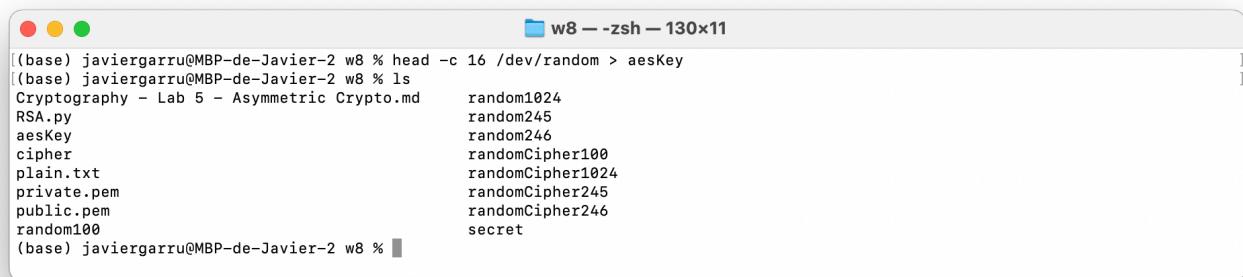
If we pay more attention to the specific error, we can see that it says it failed at *RSA_padding_add_PKCS1_type_2*, confirming that it fails when it isn't capable of adding the minimum padding necessary for security reasons.

6.3.Protect big files:

In order to avoid the above shown problem, a solution is proposed. This solution consists on generating a symmetric key, encrypting the file with a symmetric cipher mode, encrypying the key with RSA and sending the encrypted message and key to the receiver.

First, we generate a random encryption key:

```
head -c 16 /dev/random > aesKey
```



The screenshot shows a terminal window titled "w8 -- zsh -- 130x11". The command entered is "head -c 16 /dev/random > aesKey". The output shows the creation of a file named "aesKey" containing 16 bytes of random data. The terminal also lists several other files in the current directory, including RSA.py, aesKey, cipher, plain.txt, private.pem, public.pem, random100, random1024, random245, random246, randomCipher100, randomCipher1024, randomCipher245, randomCipher246, and secret.

Then, we encrypt the desired file with the generated key and the algorithm selected (aes-128-cbc):

```
openssl enc -aes-128-cbc -in uts.bmp -out utscipher -pass file:./aesKey -iv  
12345678901234567890123456789012
```



The screenshot shows a terminal window titled "w8 -- zsh -- 151x5". The command entered is "openssl enc -aes-128-cbc -in uts.bmp -out utscipher -pass file:./aesKey -iv 12345678901234567890123456789012". A warning message appears: "*** WARNING : deprecated key derivation used. Using -iter or -pbkdf2 would be better." The terminal also lists several other files in the current directory, including RSA.py, aesKey, cipher, plain.txt, private.pem, public.pem, random100, random1024, random245, random246, randomCipher100, randomCipher1024, randomCipher245, randomCipher246, and secret.

Now, we can encrypt the key using RSA:

```
openssl rsautl -encrypt -inkey public.pem -pubin -in aesKey -out aesKey.bin
```



The screenshot shows a terminal window titled "w8 -- zsh -- 155x10". The command entered is "openssl rsautl -encrypt -inkey public.pem -pubin -in aesKey -out aesKey.bin". The terminal also lists several other files in the current directory, including RSA.py, aesKey, cipher, plain.txt, private.pem, public.pem, random100, random1024, random245, random246, randomCipher100, randomCipher1024, randomCipher245, randomCipher246, and secret.

And end up deleting the key and the original file:

```
rm aesKey  
rm uts.bmp
```

A screenshot of a terminal window titled "w8 -- zsh -- 155x10". The window shows the command history and the current directory structure. The user has run commands to remove files and then lists the contents of the directory, which includes RSA-related files and random cipher keys.

```
(base) javiergarru@MBP-de-Javier-2 w8 % rm uts.bmp  
(base) javiergarru@MBP-de-Javier-2 w8 % rm aesKey  
(base) javiergarru@MBP-de-Javier-2 w8 % ls  
Cryptography - Lab 5 - Asymmetric Crypto.md      public.pem          randomCipher1024  
RSA.py                                         random100            randomCipher245  
aesKey.bin                                    random1024          randomCipher246  
cipher                                         random245           secret  
plain.txt                                     random246          utscipher  
private.pem                                    randomCipher100  
(base) javiergarru@MBP-de-Javier-2 w8 %
```

Now, supposed we are at the receiver side. We just need to use the RSA algorithm to decrypt the key (using the private key) and then the aes algorithm with the key to decrypt the file:

```
openssl rsautl -decrypt -inkey private.pem -in aesKey.bin -out aesKey  
openssl enc -aes-128-cbc -d -out utsR.bmp -i utscipher -pass file:./aesKey -iv  
12345678901234567890123456789012  
ls
```

A screenshot of a terminal window titled "w8 -- zsh -- 158x15". The user has run the decryption and encryption commands. The terminal shows the command history and the resulting file structure after the operations.

```
(base) javiergarru@MBP-de-Javier-2 w8 % openssl rsautl -decrypt -inkey private.pem -in aesKey.bin -out aesKey  
(base) javiergarru@MBP-de-Javier-2 w8 % openssl enc -aes-128-cbc -d -out utsR.bmp -i utscipher -pass file:./aesKey -iv 12345678901234567890123456789012  
*** WARNING : deprecated key derivation used.  
Using -iter or -pbkdf2 would be better.  
(base) javiergarru@MBP-de-Javier-2 w8 % ls  
Cryptography - Lab 5 - Asymmetric Crypto.md      public.pem          randomCipher245  
RSA.py                                         random100            randomCipher246  
aesKey                                         random1024          secret  
aesKey.bin                                    random245           utsR.bmp  
cipher                                         random246          utscipher  
plain.txt                                     randomCipher100  
private.pem                                    randomCipher1024  
(base) javiergarru@MBP-de-Javier-2 w8 %
```



Question: Why the private keys in asymmetric ciphers are often generated rather than specified?

To make the whole process more secure. If the keys were specified by the user, the chosen numbers being used wouldn't be (most of the times) as random as the ones a pseudorandom number generator based on a good source of entropy can be. For instance, users might choose primes that are small like 7, while the generator would usually generate big and random primes. The harder is to guess the value of the used primes, the strongest is the encryption algorithm used to encrypt the symmetric key using RSA. If this key was discovered, then the message could be also discovered.

Part Seven: Lab Summary

In this lab we have learnt how the Diffie Hellman Key Exchange algorithm works. We have also in depth study the RSA algorithm and what is the maximum length that can be encrypted depending on the key-size. We have seen how the keys are generated and the strength of using large primes due to the difficulty of factoring numbers using the exhaustive search. At last, we have learnt about the usual procedure that combines asymmetric and symmetric encryption techniques to *securely* transmit messages.