

A CTF Writeup of the RSA Cryptosystem

Norwegian University of Science and Technology
Ethical Hacking, specialization course
TTM4536

By: Alexander H. Bakken
Nickname: Th3Cyb3rChi3f
Email: alexander.h.bakken@ntnu.no

07.11.2019

Contents

1	Introduction	2
2	Methods	3
2.1	Life is too complex to compress	3
2.2	It smells like RSA	5
2.3	The hypothesis	6
2.4	Common factor attack	6
2.5	A solution to every puzzle	7
3	Discussion	10
4	Summary	10

Foreword

The main focus of this writeup is to solve the CTF with respect to the RSA public-key cryptosystem and one of its weaknesses we are soon to discover. We will begin with some introduction around the original RSA algorithm and its mathematical properties. In the methodology section we will solve the CTF step-by-step and wrap it all up in a discussion about the attack. We conclude this writeup with a short summary.

1 Introduction

The reader should have some basic knowledge of the RSA cryptosystem in order to make any sense of this writeup. Hence, we briefly go through the necessary knowledge here.

RSA is based on the fact that multiplying two primes $p \times q = N$ is easy, but factoring N is hard. That's to say; find p and q from N . Moreover, the RSA algorithm has its origins from the field of number theory and modular arithmetic. In short, this means that everything comes in the form of integers. The message(s) that we want to encrypt will of course be in a human-readable text format, but we can convert this text to a sequence of integers using a certain padding scheme. However, that is not our main concern in this writeup. Here, we provide the maths behind the original RSA encryption and decryption.

RSA algorithm:

Encryption: $c = m^e \pmod{n}$

Example: we choose $m = 7$, $e = 3$, $p = 5$ and $q = 11$.

We calculate; $n = p \times q = 5 \times 11 = 55$, such that;

$$c = 7^3 \pmod{55} = 343 \pmod{55} = 13 \pmod{55} \implies \underline{\underline{c = 13}}$$

Decryption: $m = c^d = (m^e)^d = m^{ed} \pmod{n}$

Example: so, we know that $c = 13$, $e = 3$ and $n = p \times q = 5 \times 11 = 55$.

We still need to calculate the private exponent (d). We get that

$$\phi(n) = (5 - 1)(11 - 1) = 4 \times 10 = 40, \text{ and}$$

$$3d \equiv 1 \pmod{40} \implies d = 27$$

$$\text{we find } m = 13^{27} \pmod{55} = 7 \pmod{55} \implies \underline{\underline{m = 7}}$$

Finding the private exponent: $ed \equiv 1 \pmod{\phi(n)}$

Finding the Euler's totient function: $\phi(n) = (p-1)(q-1) = n - (p+q-1)$

The ciphertext (c) is calculated by raising the message (m) to the power of the public exponent (e) and then taking the modulus of n (where $n = p \times q$). By taking the ciphertext (c) to the power of the private exponent (d) we get the decryption and the message (m) pops out again. Notice that some middle calculation in the decryption is left out for brevity. The $\phi(n)$

is Euler's totient function. The relation between the public (e) and the private (d) exponents is given by $\phi(n)$ that can only be calculated if you know p and q. Moreover, the public key is (n, e) and the private key is (n, d) and constitute the key-pair.

The concept of a public key comes from the world of public-key cryptosystems (PKC) where each communicating participant generates a key-pair; a private-key and its corresponding public-key. Usually, Bob and Alice would each generate such a key-pair and expose their public-keys to the world and keep the private-key (as a secret) to themselves. Whenever Bob wants to send a message, he encrypts the message with the public-key from Alice and sends it over the "insecure" network towards Alice. Then, upon reception, Alice decrypts the message with her secret private-key. Vice versa, Alice would encrypt with Bob's public-key, and Bob decrypts with his private-key. Notice that everyone can encrypt something with the public-key, but only the owner of the corresponding private-key is able to decrypt that message. With this in mind we are ready to get our hands (or keyboard) dirty. Let's get hacking!

2 Methods

In this section we go through the CTF step-by-step, using the list of tools located below.

List of tools:

- OS: Kali Linux 2
- Python version 2.7.16+
- Unzip
- OpenSSL
- RsaCtfTool

2.1 Life is too complex to compress

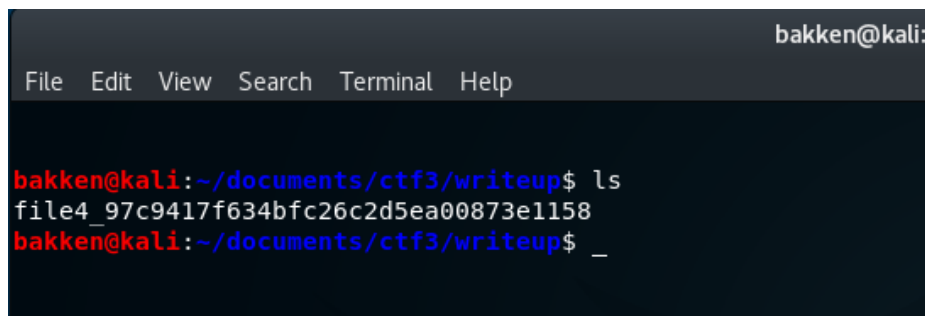
A screenshot of a terminal window with a dark background. The title bar at the top right says 'bakken@kali:'. Below the title bar is a menu bar with 'File', 'Edit', 'View', 'Search', 'Terminal', and 'Help'. The terminal shows a prompt 'bakken@kali:~/documents/ctf3/writeup\$' followed by the command 'ls'. The output of the command is 'file4_97c9417f634bfc26c2d5ea00873e1158'. The prompt is repeated on the next line: 'bakken@kali:~/documents/ctf3/writeup\$ _'.

FIGURE 1

We are given a mysterious file named file4_97c9417f634bfc26c2d5ea00873e1158 (fig1). It does not seem to have any file-ending, hence it could be anything.

```

bakken@kali: ~/documents/ctf3/writeup
File Edit View Search Terminal Help

bakken@kali:~/documents/ctf3/writeup$ ls
file4_97c9417f634bfc26c2d5ea00873e1158
bakken@kali:~/documents/ctf3/writeup$ file file4_97c9417f634bfc26c2d5ea00873e1158
file4_97c9417f634bfc26c2d5ea00873e1158: Zip archive data, at least v1.0 to extract
bakken@kali:~/documents/ctf3/writeup$ _

```

FIGURE 2

A nice place to start is to investigate what type of file this could be. We use the command-line tool called "file" (fig2), and discover that this is a .zip file, which is a lossless data compression archive format. Hence, it probably contains more files. Let's find out.

```

bakken@kali:~/documents/ctf3/writeup$ ls
file4_97c9417f634bfc26c2d5ea00873e1158
bakken@kali:~/documents/ctf3/writeup$ file file4_97c9417f634bfc26c2d5ea00873e1158
file4_97c9417f634bfc26c2d5ea00873e1158: Zip archive data, at least v1.0 to extract
bakken@kali:~/documents/ctf3/writeup$ mv file4_97c9417f634bfc26c2d5ea00873e1158 file4.zip
bakken@kali:~/documents/ctf3/writeup$ ls
file4.zip
bakken@kali:~/documents/ctf3/writeup$ whatis unzip
unzip (1) - list, test and extract compressed files in a ZIP archive
bakken@kali:~/documents/ctf3/writeup$ unzip file4.zip
Archive: file4.zip
  extracting: file1
  extracting: file2
  inflating: file3
bakken@kali:~/documents/ctf3/writeup$ ls
file1 file2 file3 file4.zip

```

FIGURE 3

We change the file-ending to .zip and make use of the tool "unzip" to unpack all the compressed files (fig3). We discover three new files named; file1, file2 and file3. Again, using the "file" tool we gain some knowledge of these. We discover that file1 and file2 are both raw-data files, while file3 is another .zip file. We use "unzip" for file3 and reveals itself to contain one folder with several files in a .pem format (fig4).

```

bakken@kali: ~/documents/ctf3/writeup/file3
File Edit View Search Terminal Help

bakken@kali:~/documents/ctf3/writeup$ ls
file1 file2 file3 file4.zip
bakken@kali:~/documents/ctf3/writeup$ cd file3/
bakken@kali:~/documents/ctf3/writeup/file3$ ls
'Aaron Blankenship.pem' 'Cheyanne Price.pem' 'Izabella Thornton.pem' 'Kenna Pugh.pem' 'Quinten Curtis.pem'
'Abram Contreras.pem' 'Chloe Craig.pem' 'Jacob Cantu.pem' 'Keon Shelton.pem' 'Raymond Cowan.pem'
'Adrian Graham.pem' 'Cierra Tapia.pem' 'Jamarion Jacobson.pem' 'Kiersten Cooley.pem' 'Renee Kim.pem'
'Adrien Roach.pem' 'Cristina Little.pem' 'Jasiah Burton.pem' 'Kolten Mata.pem' 'Ryker Orozco.pem'
'Alessandro Weaver.pem' 'Dalia Love.pem' 'Javon Hoover.pem' 'Korbin Avila.pem' 'Sadie Riggs.pem'
'Alfonso Wang.pem' 'Dania Cantu.pem' 'Jaylah Frye.pem' 'Lexi Herman.pem' 'Salma Mercado.pem'
'Amare Conrad.pem' 'Darrell Barnett.pem' 'Jayleen Yu.pem' 'Lily Summers.pem' 'Sammy Griffin.pem'
'Amare Walter.pem' 'Deandre McKay.pem' 'Jaylynn Ayala.pem' 'Lina Wall.pem' 'Samson Rasmussen.pem'
'Anabella Saunders.pem' 'Destiny Huff.pem' 'Jerry Petty.pem' 'Linda Walter.pem' 'Sanai Robles.pem'
'Arabella Leonard.pem' 'Ean Perez.pem' 'Joe Holden.pem' 'Lucia Walters.pem' 'Sawyer Henson.pem'
'Ashanti Andrade.pem' 'Elliana Burnett.pem' 'Johanna Klein.pem' 'Marilyn Park.pem' 'Sheldon Becker.pem'
'Ashanti Bowen.pem' 'Enrique Dominguez.pem' 'Jordin Armstrong.pem' 'Maximus Benitez.pem' 'Shirley Juarez.pem'
'Ashley Richardson.pem' 'Esther Branch.pem' 'Jovany Drake.pem' 'Melvin Gibbs.pem' 'Sidney Shannon.pem'
'Aubree Watkins.pem' 'Ezra Wyatt.pem' 'Julius Oconnor.pem' 'Mitchell Stanley.pem' 'Silas Leach.pem'
'Camden Garner.pem' 'Francesca Price.pem' 'Justice Ortiz.pem' 'Morgan Clark.pem' 'Sophie Huff.pem'
'Carolyn Castillo.pem' 'Gerardo Pineda.pem' 'Kassidy Huff.pem' 'Morgan Dougherty.pem' 'Tyler Murphy.pem'
'Chace Shea.pem' 'Harper Morris.pem' 'Kaya Hendricks.pem' 'Mya Hopkins.pem' 'Uriel Zhang.pem'
'Chana Beasley.pem' 'Hassan Murillo.pem' 'Keegan Summers.pem' 'Nelson Bean.pem' 'Winston Marquez.pem'
'Charlie Jensen.pem' 'India Lutz.pem' 'Kendall Burch.pem' 'Paxton Branch.pem' 'Zander Lynn.pem'
'Chelsea Clarke.pem' 'Ishaan Henson.pem' 'Kendra Calhoun.pem' 'Pranav Dean.pem' 'Zoey Reed.pem'
bakken@kali:~/documents/ctf3/writeup/file3$ _

```

FIGURE 4

These files are commonly associated with digital certificates. The Privacy Enhanced Email (PEM) file format is defined in RFC 1421 through 1424, and is a base64 encoding of the X509 ASN.1 keys. Let's have a closer look on one of them.

```

root@kali: /home/bakken/documents/ctf3/writeup/file3
File Edit View Search Terminal Help
root@kali: /home/bakken/documents/ctf3/writeup/file3# mv 'Chana Beasley.pem' chelsea.pem
root@kali: /home/bakken/documents/ctf3/writeup/file3# cat chelsea.pem
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEA5gA1m55CnaeCRghHzn76
yqXX2YTCmQ+VpJn9VHnCyC+p2VXVKo3S0kGzdvd0DPK230dQjMQDatorL7tqRm8o6
yCwLGkyUrIVyF9rsJWCz4tIizHm+hnB/R1WXR1mELx9qWbaysGFQlW8m69FyJ3Xo
wjF8xmZZ0nDxpikDs7EQxEs8u8aCcrwenbgSTSvYZBGuD8UtyeIkD6mN80JYf88F
I+VKQhMg2R2/jZK5S9VuMDg0fQrtMy3u06KSs0wdlAsmagu5+IuKmojcjit/C7q8
dLzYPQvckppRwFcKHXQvJpJjRSq26ipwulXzZARFmqGzpBSNpsjGr5VD0wzCXHI
BwIDAQAB
root@kali: /home/bakken/documents/ctf3/writeup/file3# _

```

FIGURE 5

First, we rename one of the files in the /file3 folder to "chelsea.pem" for brevity. We look into the .pem files and we see that all of them contain one public-key encoded in base64, just as expected (fig5).

2.2 It smells like RSA

Next, we want to know even more about the .pem files in greater detail. For this we can use the open source tool called "openssl" (fig6).

```

bakken@kali: ~/documents/ctf3/writeup/file3
File Edit View Search Terminal Help
Use "keytool -command name -help" for usage of command name.
Use the -conf <url> option to specify a pre-configured options file.
bakken@kali: ~/documents/ctf3/writeup/file3$ openssl rsa -inform PEM -pubin -in chelsea.pem -text -noout
RSA Public-Key: (2048 bit)
Modulus:
 00:d2:9f:44:16:34:db:ab:dc:58:a0:5f:b1:61:9b:
 e9:b1:c2:5c:8e:b1:b7:fc:f2:2f:0a:d0:d7:55:65:
 34:c6:22:2c:7f:5a:13:9c:d2:69:58:3b:9b:52:84:
 a1:eb:8f:38:fa:3a:00:29:5e:11:00:35:0b:03:ed:
 c3:dc:06:dd:8c:2c:06:65:c4:24:22:67:11:33:c7:
 ec:5a:72:eb:b0:eb:e2:72:ca:36:8d:23:7c:d6:47:
 36:40:97:04:06:29:b6:46:0a:b3:0d:4f:e3:0b:6c:
 5a:aa:36:58:ff:25:e7:4a:4f:b9:32:84:59:0b:39:
 9a:53:4b:7a:f5:e2:e7:76:5e:ab:0a:2f:df:77:86:
 bc:f4:0b:4d:8c:0e:ae:b2:22:03:7c:c7:13:1b:1d:
 fb:bd:bd:a6:1d:80:af:11:d0:04:6e:ba:71:b8:29:
 f9:f8:ed:b1:53:4d:94:ce:6b:31:d7:25:25:24:25:
 b2:a6:fe:55:84:10:51:98:31:c4:2b:c2:3f:eb:bb:
 1f:55:fc:5e:23:5b:ab:a2:0d:0a:7b:63:8c:dc:f3:
 e2:6c:b4:be:c8:63:ef:ac:e2:57:89:1a:a8:e6:0a:
 bd:a8:dd:68:0a:b4:07:f5:ff:4a:fd:12:2a:39:90:
 00:e5:c0:10:85:b9:57:46:0e:d0:3b:0e:4b:2e:54:
 e7:e1
Exponent: 65537 (0x10001)
bakken@kali: ~/documents/ctf3/writeup/file3$ _

```

FIGURE 6

The output (fig6) is telling us that we're dealing with the well-known public-key cryptosystem, **R**iverst-**S**hamir-**A**dleman (RSA), named after its three inventors. Furthermore, we can see that the public-key is 2048 bits long and the modulus and public exponent are given. Recall from the introduction, we now have (n, e), which of course is the public-key. The RSA algorithm operates securely under certain conditions, but vulnerabilities are discovered once these are not met. If we're in luck this might be the case.

2.3 The hypothesis

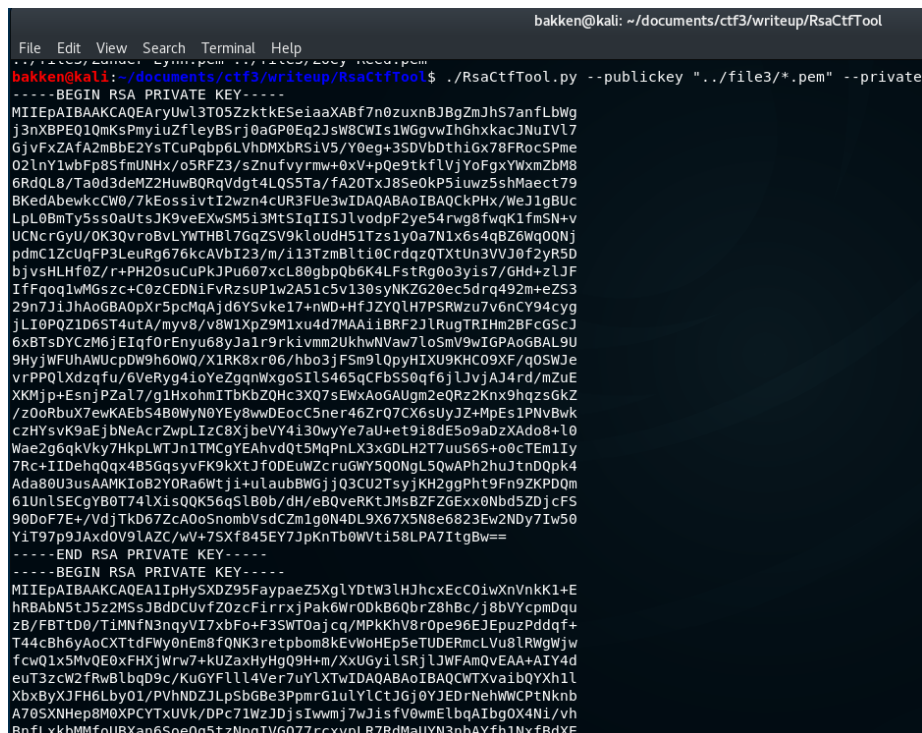
Using this knowledge, we can form our hypothesis:

Probably, file1 and file2 are encrypted with one of these public-keys. If we somehow could get hold of the private-key as well, then we could easily decrypt the messages and retrieve our flag. The numerous public-keys exposed to us might point in the direction of a possible common factor attack.

A *common factor attack* can derive the private key from two or more public keys and then use this private key to decrypt the message(s). This is possible because of a common factor in the modulus of the public-keys. It should be said that this is very rare to happen. More on this in the discussion.

2.4 Common factor attack

Lets put our hypothesis to the test! In order to perform a common factor attack on multiple public-keys as found in /file3 folder, we can use the RsaCtfTool (fig7).



```
bakken@kali: ~/documents/ctf3/writeup/RsaCtfTool
File Edit View Search Terminal Help
bakken@kali:~/documents/ctf3/writeup/RsaCtfTool$ ./RsaCtfTool.py --publickey "../file3/*.pem" --private
-----BEGIN RSA PRIVATE KEY-----
MIIEpAIBAAKCAQEAryUwL3T05ZzktkESeiaaXABf7n0zuxnBJBgZmJhS7anfLbWg
j3nXBPEQ1QmKsPmyIuZfleyBSrj0aGP0Eq2JSW8CWIIs1WGgvwIhGhxkacJNuIVL7
GjvFXZAfA2mBbE2YsTCuPqbp6LVhDMXBRS1V5/Y0eg+3SDVbDthiGx78FRocSPme
02lnY1wbFp8SfmuNHx/o5RFZ3/sZnuFvyrmw+0xV+pQe9tkfLvjYoFgxYwXmZbM8
6RdQL8/Ta0d3deM2ZHuBQRqVdgt4LQS5Ta/fA20TxJ8Se0kP5iuwz5shMaect79
BkedAbewkCW0/7KEossivtI2wzn4cUR3Fue3wIDAQABAoIBAQCkPHX/WeJ1gBuc
LpL0BmTy5s0aUtsJK9veEXwSM5i3MtSiQIISJlvodpF2ye54rwg8fwgK1fmsN+v
UCNcrGyU/OK3QvrbvLYWTHB17GqZSV9kLoUdH51TzslY0a7N1x6s4qBZ6Wq00Nj
pdmC1ZcUqFP3LeuRg676kcAVbI23/m/i13TzmBlti0Crdqz0TXtUn3VVJ0f2yR5D
bjvsHLHf0Z/r+PH20sucUPkJPu607xcL80gbpQb6K4LFstRg003yis7/GHd+ZLJF
IfFgoqlwMgszc+C0zCEDNiFvRzsUP1w2A51c5v130syNKZG20ec5drq492m+eZS3
29n7J1jha0GBA0pXr5pcMqAjd6YSvke17+nWD+HfJZYQ1H7PSRWzu7v6nCY94cyg
jLI0PQZ1D6S4uta/myv8/v8W1XPZ9M1xu4d7MAAiiBRF2JLRugTRIhm2BFC6ScJ
6xBTsDYCzM6jEIqf0rEnyu68yJa1r9rkiVmm2UkhwNVaw7LoSmV9wIGPAoGBAL9U
9HyjWfUhuAUCpDw9h60WQ/X1RK8xr06/hbo3jFsm9L0pyHIXU9KHC09XF/q0SWJe
vrPQLXdxzfu/6VeRyg4ioYeZgqnWxgoSIL5465qCFbS50qf6j1JvJAJ4rd/mZuE
XKMjp+EsngPZal7/g1HxohmITbKbZQHc3XQ7sEwxAoAUGm2e0Rz2Knx9hqzsGkZ
/z00RbuX7ewKAeb54B0Wyn0YEy8wwDEocC5ner46Zr07CX6sUyJZ+MpEs1PNvBwk
czHYsvK9AejbNeAcrZwpLIzC8XjbeVY4i30wyYe7aU+et9i8dE509aDzXAdo8+l0
Wae2G6qkVky7HkplWTJn1TMCgYEAhvdQtsMqPnLX3xGD1H2T7uuS6S+o0cTEm1Iy
7Rc+IIdheqQx4B5GqsyvFK9KxtJf0DeuWZcruGWY500NgL5QwAPH2huJtnD0pk4
Ada80U3usAAMKI0B2Y0Ra6Wtj1+u1aubBWGjjQ3CU2TsyjKH2ggPhT9Fn9ZKPDQm
61UnLSEcgYB0T74LXisQQK56qSLB0b/dH/eBQveRktJMsBZFZG6Exx0Nbd5ZDjcFS
90DoF7E+/VdjTkD67ZcA0oSnombVsdCzm1g0N4DL9X67X5N8e6823Ew2NDy7Iw50
Yi1T97p9JAXd0V9LAZC/wV+75Xf845EY7JpKnTb0Wvti58LPA7ItgBw==
-----END RSA PRIVATE KEY-----
-----BEGIN RSA PRIVATE KEY-----
MIIEpAIBAAKCAQEA1IpHySX0Z95FaypaeZ5XgLYDtW3LHJhcxEcC0iwxNvNkK1+E
hRBAbsNtJ5z2MSsJBdDCUvfZ0zcFirrxjPak6WroDk6QbrZ8hBc/j8bVYcpmDqu
zB/FBTtD0/TiMnfN3nqyVI7xbFo+F3SWT0ajcq/MPkKhV8r0pe96E3EpuzPddqf+
T44cBh6yAoCXTtdFWy0nEm8fQNK3retpbom8kEvWoHEP5eTUDERmclVU81RWgWjw
fcwQ1x5MvQE0xFOXjWrw7+kUZaxHyHg09H+m/XxUGy1LSRjLJWfAmQVEAA+AIY4d
euT3zcw2fRwBldqD9c/KuGYFLl4Ver7uYLTWIDAQABAoIBAQCWTXvaibQYXh1l
Xbx3yXJFH6Lby01/PVhNDZJLpSbGBE3PpmrG1uYlCtJGj0YJEDrNehWwCPTnKnB
A70SXNHep8M0XPCYtUVK/DPc71WzJDjsIwwmj7wJisfV0wmElbqA1Bg0X4Ni/vh
RnfiYkHMMfAIRXan6Sae0n5tzMnqTVG077rcxvnlR7RdMaliVn3nhAYfh1NvfrdYF
```

FIGURE 7

Note: the quotes around the .pem file is necessary in order to use the * wildcard. The wildcard will make sure that all the .pem files in the folder are checked for a possible common factor attack. Another thing, if you add -verbose to the RsaCtfTool command it will display the names of the vulnerable public-keys, if any.

As can be seen from Figure 7, we successfully extracted two private keys from the pool of public-keys (fig4). More on this and why this attack is possible, you will find in the discussion. Next, we want to store the keys in

separate files. We copy each private-key and store them in a separate file, named `privateKey1` and `privateKey2`. These files must be stored exactly as they are, meaning there cannot be any extra whitespace or symbols within the file. Its also important to include the `"-BEGIN RSA..."` and `"-END RSA..."` lines. In order to check or verify the syntax of both private-keys, use the `"openssl"` (fig8).

```

bakken@kali: ~/documents/ctf3/writeup
File Edit View Search Terminal Help
bakken@kali:~/documents/ctf3/writeup$ openssl rsa -in privateKeyOne -check
RSA key ok
writing RSA key
-----BEGIN RSA PRIVATE KEY-----
MIIEpAIBAAKCAQEArUwL3T05ZzktkESeiaaXABf7n0zuxnBJBgZmJhS7anfLbWg
j3nXBPEQ1QmKsPmyiuZfleyBSrj0aGP0Eq2Jsw8CWIslWGgvwIhGhxkacJNuIVl7
GjvFxAfA2mBbE2YsTCuPqbp6LVhDMXbRSiV5/Y0eg+3SDVbDthiGx78FRocSPme
02lnY1wbFp8SfmUNHx/o5RFZ3/sZnufvyrmw+0xV+pQe9tkfLvjYoFgxYwXmZbM8
6RdQL8/Ta0d3deMZ2HuwBQRqVdgt4LQ55Ta/fA20TxJ8SeOkP5iUwz5shMaect79
BKedAbewkCw0/7kEossivtI2wnzn4cUR3FUE3wIDAQABAoIBAQCkPHx/WeJlgbuc
LpL0BmTy5s0aUtsJK9veEXwSM5i3MtSIqIISJlvodpF2ye54rWG8fwqK1fmsN+v
UCNcrGyU/OK3QvrvBvLYWTHBL7GqZSV9kloUdH51TzslY0a7N1x6s4qBZ6Wq00Nj
pdmC1ZcUqFP3LeuRg676kCAVbI23/m/i13TzmBlti0CrdqzQTXtUn3VVJ0f2yR5D
bjvsHLHf0Z/r+PH20suCuPkJPu607xcl80gpbQb6K4LFstRg003yis7/GHd+zLJF
IffQoqIwMGszc+C0zCEDNiFvRzsUP1w2A51c5v130syNKZG20ec5drq492m+eZS3
29n7JiJhAoGBAOpXr5pcMqAjd6YSvke17+nWD+HfJZYQlH7PSRWzu7v6nCY94cyg
jLI0PQZ1D65T4utA/myv8/v8W1XpZ9M1xu4d7MAAiBRF2JlRugTRIhm2BFcGScJ
6xBTsDYCzMG6jEIqf0rEnyu68yJa1r9rkivmm2UkhWNvaw7LoSmV9wIGPAoGBAL9U
9HyjWfUHAUucpDW9h60WQ/X1RK8xr06/hbo3jFSm9lQpyHIXU9KHC09XF/q0SWJe
vrrPPQLxdzqfu/6VeRyg4ioYeZgqnWxgoSILS465qCFb5S0qf6jLjvAJ4rd/mZuE
XKMjp+EsnpZal7/g1HxohmITbKbZQHc3XQ7sEWxAoGAUgm2eQRz2Knx9hqzsGkZ
/z0oRbuX7ewKAebS4B0WkyN0Yey8wwDEocC5ner46ZrQ7CX6sUyJZ+MpEs1PNvBwk
czHYsvK9aEjbNeAcrZwplZc8XjbeVY4i30wyYe7aU+et9i8dE5o9aDzXAdo8+l0
Wae2q6qkVky7HkplWTJn1TMCgYEAhvdQt5MqPnLX3xGDH2T7uuS6S+o0CTEm1Iy
7Rc+IIDehqQqx4B5GqsyvFK9kXtJf0DEuWZcrUGWY5Q0NgL5QwAPh2huJtnDQpk4
Ada80U3usAAMKIoB2Y0Ra6wtji+ulaubBWGjjQ3CU2TsyjKH2ggPht9Fn9ZKPDQm
61UnlSECgYB0T74LXisQQK56qSLB0b/dH/eBQveRktJMsBZFZGExx0Nbd5ZDjcFS
90DoF7E+/VdjTkD67ZcA0oSnombsVsdCZm1g0N4DL9X67X5N8e6823Ew2NDy7Iw50
YiT97p9JAx0V9LAZC/wV+7SXf845EY7JpKnTb0WVti58LPA7ItgBw==
-----END RSA PRIVATE KEY-----
bakken@kali:~/documents/ctf3/writeup$ openssl rsa -in privateKeyTwo -check
RSA key ok
writing RSA key
-----BEGIN RSA PRIVATE KEY-----
MIIEpAIBAAKCAQEAlPhySXDZ95FaypaeZ5XglYDtW3lHJhcxEcC0iwXnVnkK1+E
hRBAbN5tJ52zMSsJBdDCUvfZ0zcFirrXjPak6Wr0dkB6QbrZ8hBc/j8bVYcpmDqu
zB/FBTd0/TimNfn3nqyVI7xbFo+F3SWT0ajcq/MPkKhV8r0pe96EJEpuZPddqf+
T44cBh6yAoCXTtdFWy0nEm8fQNK3retpbom8kEvWoHEp5eTUDERmCLVU8LRWgWjw
fcwQ1x5MvQEOxFHxjWrw7+kUZaxHyHgQ9H+m/XxUGyilSRjlJWFAMQvEAA+AIY4d
euT3zcw2fRwBlbqD9c/KuGYFlll4Ver7uYlXTwIDAQABAoIBAQCWTXvaibQYXh1l
XbxByXJFH6Lby01/PVhNDZJLpSbGBe3PpmrG1ulYlCtJGj0YJEDrNehWwCPtNknB
A70SXNHep8M0XPCYTXUVK/DPc71WzJDjsIwwmj7wJisfv0wmElbqAibgOX4Ni/vh
BnFLxkbMMfouBXan6SoeOq5tzNpgIVG077rcxvplR7RdMaUYN3nbAYfh1NxfBdXE
Sn0AUf4/BBR8w01lT6FBL0pv4qeD0t5dadQPf83lg2xKZBf9kthsUpIMP++Yf+ct
HnMDL2v612MoydUUI9SpCjfkMj3RfwoXWT1Us8b2/o2vI3uqwaLjDdf5naSrGm0

```

FIGURE 8

We get the following message; "RSA key ok", meaning that the syntax is good and the key is ready to be used. In case you wonder, `privateKey1` maps to 'Quinten Curtis.pem' and `privateKey2` maps to 'Maximus Benitez.pem' (fig4).

2.5 A solution to every puzzle

Alright, we know the private key! It turns out that both keys are in a so-called pkcs12 ASN.1 file format, defined in RFC 7292. This means that all values; modulus (n), primes (p, q) and exponents (e, d) are given within the file. By running the following commands in the terminal:

```

openssl rsa -inform PEM -in privateKey1 -text -noout
openssl rsa -inform PEM -in privateKey2 -text -noout

```


we will get an output verifying this type of format. Hence, we have everything we need in order to decrypt, lets create a python script to do so.

```

1
2 import base64
3 import gmpy
4 from Crypto.PublicKey import RSA
5
6 def decrypt(encFile, key):
7     pub_key = ""
8     with open(key) as f:
9         pub_key = f.read()
10
11     pub = RSA.importKey(pub_key)
12     n = long(pub.n)
13     p = long(pub.p)
14     q = long(pub.q)
15     assert(n == p*q)
16     e = long(pub.e)
17     d = long(gmpy.invert(e, (p-1)*(q-1) ))
18     key = RSA.construct( (n,e,d) )
19
20
21     with open(encFile) as f:
22         flag_enc = f.read()
23         C = base64.b64decode(flag_enc)
24         print key.decrypt(C)
25         #print "Decryption Key (d): " + str(d)
26
27
28 # DECRYPT
29 privateKey1 = "privateKey1"
30 privateKey2 = "privateKey2"
31
32 encryptedFile1 = "FileOneB64.enc"
33 encryptedFile2 = "FileTwoB64.enc"
34
35 decrypt(encryptedFile1, privateKey1)
36 decrypt(encryptedFile2, privateKey2)

```

Save it as "DecryptTheRSAFlag.py" without quotations. This python file uses native RSA encryption and decryption. The Crypto library will decrypt the input file with the given private-key, just like we did in the introduction. The python file will not be explained further, but the reader is encouraged to take a look and understand it. Before we can use the python file, we must convert the raw data-files (file1 and file2) into base64 encoding (fig9), like this:

```

openssl base64 -in file1 -out FileOneB64.enc
openssl base64 -in file2 -out FileTwoB64.enc

```

This is due to line 11 in the python file, where "RSA.importkey(input)", expects a base64 input.

```

bakken@kali: ~/documents/ctf3/writeup
File Edit View Search Terminal Help
bakken@kali:~/documents/ctf3/writeup$ ls
file1 file2 file3 file3.zip file4.zip MySolution privateKey1 privateKey2 RsaCtfTool
bakken@kali:~/documents/ctf3/writeup$ openssl base64 -in file1 -out FileOneB64.enc
bakken@kali:~/documents/ctf3/writeup$ ls
file1 file2 file3 file3.zip file4.zip FileOneB64.enc MySolution privateKey1 privateKey2 RsaCtfTool
bakken@kali:~/documents/ctf3/writeup$ cat FileOneB64.enc
cogRmEi/Wbv+cAh9hw3uxVkwWv863U8RvkYzIwF9Dm13/ZpfsKNvAmigJpo0VeZU
ZVnnLy1IHdR95jHxNYTHnPQLiUJWviEtn/4C0RJELVHst70Q8fSeTZvTLVeXnZ
HiNpzeXS50oq/6Ilqvrp+uoYX5EAYJhw1UZpEq5iU6RtnhWMT00X/h0lqShYHgRe
Dnje3ei0uYvMLbrCA2rJRLWvyymiYL2wzfe3J0kqECzCGaEy5YJsw9T74JnRachm
8S3L/icj8Alp88IWUx/Mcmmie9Tnya9e8BRHljR4jx0BiBgmMSvTWoCAMnWaoi
XEEct5fclpggbU5thkn+GA==
bakken@kali:~/documents/ctf3/writeup$ _

```

FIGURE 9

So far, we got the following files (fig10).

```

bakken@kali: ~/documents/ctf3/writeup/MySolution
File Edit View Search Terminal Help
bakken@kali:~/documents/ctf3/writeup/MySolution$ ls
DecryptTheRSAFlag.py FileOneB64.enc FileTwoB64.enc privateKey1 privateKey2
bakken@kali:~/documents/ctf3/writeup/MySolution$ _

```

FIGURE 10

A description of each file:

- *DecryptTheRSAFlag.py* will use the *privateKey1* and *privateKey2* in order to decrypt *FileOneB64.enc* and *FileTwoB64.enc*, respectively.
- *FileOneB64.enc* corresponds to file1 in base64 format.
- *FileTwoB64.enc* corresponds to file2 in base64 format.
- *privateKey1* and *privateKey2* is the private-keys extracted through common factor attack using the *RsaCtfTool*

All that remains to do is running the python script (fig11).

```

bakken@kali: ~/documents/ctf3/writeup/MySolution
File Edit View Search Terminal Help
bakken@kali:~/documents/ctf3/writeup/MySolution$ sudo python DecryptTheRSAFlag.py
ttm4536{'There is a way out of every box, a solution to every puzzle;
Decryption Key (d): 189739544324096910776606164329782763192425401105816764636202906333786424107362726794
970523446385230055576499149340848722922263191427907192254714480809552613352743505564506917580016094046350
36423080352875129358744654038850677161290733237403978142486098169600452376609608170057018860643744233305
144511944186282967905490926647261174840602485807174570759980216197906678455325700668573710366790999558843
733294316626566217411340626139093287620052338651781651420335119762715763718716280998955708550074545204523
029249118767395216844635666334898571231477612346657282685634465917221427357535658469945710640629963741147
1330049
it is just a matter of finding it.' - Captain Jean-Luc Picard}
Decryption Key (d): 2073290892961409818146578770235442728257676403394400324954944804019498012225455347647
429938277786341651350620357805273076792403621852568363139903779257448752973950406044779124752877377520678
611603958793085404330731602286737070206554481485584426761417632125444943363024429735872334213812422518211
062006242628134589121893937019836575620065953143825576369835877830565329494564354919990423808496941264574
322929855242741099825320845720993365223274389529078144544304619685373072410075882229370356032754156774136
790273522489652807725933702470563749811752874166803713036168129294996556543041063547612954262165539765273
3764193
bakken@kali:~/documents/ctf3/writeup/MySolution$ sudo vim DecryptTheRSAFlag.py
bakken@kali:~/documents/ctf3/writeup/MySolution$ sudo python DecryptTheRSAFlag.py
ttm4536{'There is a way out of every box, a solution to every puzzle;
it is just a matter of finding it.' - Captain Jean-Luc Picard}
bakken@kali:~/documents/ctf3/writeup/MySolution$ <3_

```

FIGURE 11

We got our flag!

```
ttm4536{'There is a way out of every box, a solution to every puzzle;
it is just a matter of finding it.' - Captain Jean-Luc Picard}
```

You can also see that the RSA decryption key, which indeed is a very long number, is printed out for each of the private-keys (fig11). Recall the introduction,

$$\text{Decryption: } m = c^d = (m^e)^d = m^{ed} \pmod{n}$$

The private-key (n, d) can be used to decrypt any message which is encrypted with the corresponding public-key.

3 Discussion

Here we discuss the common factor attack on the RSA algorithm. The reader should be aware of the *greatest common divisor* - what it is and how we can find it. The greatest common divisor (GCD) between two or more integers, which are not all zero, is the largest positive integer that divides each of the integers. The GCD is found by using Euclid's algorithm, which is not explained here. So, how is a common factor attack even possible? From the methods section above, we discovered two public-keys vulnerable to the common factor attack. This means that they have a shared prime factor in their modulus. Lets do the math.

Take for example the modulus of two different key-pairs (as found above), $N1 = a \times b$ and $N2 = c \times d$, where a , b , c and d are different prime numbers. Then they have no common factors, right? That means the greatest common divisor, $\text{GCD}(N1, N2) = 1$, thus $N1$ and $N2$ is only divisible with itself and 1. We say that $N1$ and $N2$ are co-prime (or relatively prime) to each other. If this is the case, then the common factor attack is not possible. Since, obviously there are no common factors.

Now, imagine that a random number generator failed doing its job or some hardware error occurred, such that bits were flipped in the $N2$ modulus. Now the situation is different: $N1 = a \times b$ and $N2 = c \times b$, then the common factor is **b** . Hence, **b** divides both $N1$ and $N2$. Since the attacker doesn't know the values of a , b nor c , but only $N1$ and $N2$, how can she find the common factor? You guessed it! By using Euclid's algorithm: $\text{GCD}(N1, N2) = \mathbf{b} \implies \text{finding } N1/b = \mathbf{a} \text{ and } N2/b = \mathbf{c}$. Now, the attacker knows the values of a , b , c , $N1$ and $N2$. In addition she knows the *public* exponent, **e** . The only missing integer is the private exponent, **d** . Recall from the introduction that;

Finding the private exponent: $ed \equiv 1 \pmod{\phi(n)}$

Well, she knows **e** , and can calculate $\phi(N1)$ or $\phi(N2)$, and thus finding **d** . Constructing both private keys, $(N1, d1)$ and $(N2, d2)$, are now very easy.

As we know, the RSA algorithm is used within multiple protocols such as OpenVPN, TLS, DSA and hybrid encryption schemes. So, you probably wonder, how likely is a common factor attack to happen in the real world? With modern key sizes and well developed random number generators, its astronomically unlikely for a common factor attack to happen. Now isnt that a relief? Lets conclude this writeup with a short summary.

Note: The *common factor attack* must not be confused with the *common modulus attack*. They are related, but they are not the same.

4 Summary

We have briefly seen how the encryption and decryption is done by the original RSA algorithm. The CTF was a good challenge, forcing us to the depths of a cryptic world of number theory and its applications. We

have learned that the RSA is vulnerable to the so-called "common factor attack", where two or more public-keys share the same prime number in their modulus. Luckily, its very rare to happen in everyday life. We conclude this writeup by hoisting the flag.

```
ttm4536{'There is a way out of every box, a solution to every puzzle;  
it is just a matter of finding it.' - Captain Jean-Luc Picard}
```