Lab Overview

RSA (Rivest–Shamir–Adleman) is a public-key cryptosystem widely used for secure data transmission. It is based on the mathematical fact that it's easy to multiply large prime numbers together but very difficult to factor their product.

1. **Key Generation**:
    - Choose two distinct prime numbers p and q. These values are kept secret.
    - Compute **n = p*q**. n is used as the modulus for both the public and private keys. Its length is the key length.
    - Compute the totient **ϕ(n) = (p-1)*(q-1).**
    - Choose an integer e such that **1 < e < ϕ(n) and gcd(e, ϕ(n)) = 1**. In other words, **e and ϕ(n)** are coprime. Often, e is chosen as **65537 (0x10001)** because of its mathematical properties that make it efficient to compute.
    - Determine d as **d ≡ e^(-1) mod ϕ(n)**. In other words, d is the modular multiplicative inverse of e modulo ϕ(n). This means **d*e ≡ 1 mod ϕ(n)**.
    - The **public key consists of the pair (n, e)**. The **private key is (n, d)**. The values of **p, q, and ϕ(n) must be kept secret**.
2. **Encryption**:
    - Given a plaintext message m and a public key (n, e), the ciphertext c is computed as **c ≡ m^e mod n**.
3. **Decryption**:
    - Given a ciphertext c and the private key (n, d), the plaintext message m can be recovered by computing **m ≡ c^d mod n**.

The security of RSA relies on the fact that given the public key (n, e), it's computationally infeasible to compute d or to factor n back into p and q without knowing ϕ(n). While e and n are both public and n is part of the private key as well, neither provides an efficient way to calculate ϕ(n).

This report summarises the use of the Big Number library provided by OpenSSL for performing arithmetic operations on integers of arbitrary size. As the RSA algorithm involves computations on large numbers, simple arithmetic operators in programs cannot be used for such calculations since the numbers involved are typically more than 512 bits long. To compute their products, we need to use a function, and several libraries can perform arithmetic operations on integers of arbitrary size. In this lab, we utilised the Big Number library provided by OpenSSL and defined each big number as a BIGNUM type. The APIs provided by the library were used for various operations, such as addition, multiplication, exponentiation, modular operations, etc.

We will complete the following action items:

1. Deriving the Private Key
2. Encrypting a Message
3. Decrypting a Message
4. Signing a Message
5. Verifying signature
6. Manually Verifying an X.509 Certificate

For the lab we will need only one VM, we will use the Attacker's machine, with ip of 10.0.2.19.

Also, each c file will be compiled with lcrypto (the OpenSSL library):
gcc filename.c **-o** filename **-lcrypto**

Task 1: Deriving the Private Key

This task involves calculating the private key (d) for a given public key (e, n), where n is the product of two prime numbers (p and q). The values of p, q, and e are provided in hexadecimal format. It should be noted that while the values of p and q used in this task are relatively small, in practice, they should be at least 512 bits long to ensure security.

The objective of this task is to demonstrate the process of generating a private key from a given public key, which is an essential aspect of the RSA algorithm used in public-key cryptography

The numbers given for the task:

- **p** = F7E75FDC469067FFDC4E847C51F452DF
- **q** = E85CED54AF57E53E092113E62F436F4F
- **e** = 0D88C3

The steps for calculating the private key are as follows:

1. Calculate the product of the two prime numbers: n = p * q.
2. Calculate Euler's totient function of n: φ(n) = (p-1) * (q-1).
3. Choose a value for the public exponent e such that 1 < e < φ(n) and e is coprime to φ(n), meaning that the greatest common divisor of e and φ(n) is 1. (in our case e is provided in advance)
4. Calculate the modular multiplicative inverse of e modulo φ(n), denoted by d, which is the private exponent. This means finding an integer d such that d*e mod φ(n) = 1.
5. The private key is (d, n).

To calculate d, we use the extended Euclidean algorithm or the algorithm for modular multiplicative inverse. The latter can be computed using the following steps:

1. Calculate the value of φ(n).
2. Choose a value for the public exponent e.
3. Use the extended Euclidean algorithm to find two integers x and y such that x * e + y * φ(n) = 1. The value of x will be the modular multiplicative inverse of e modulo φ(n).
4. If x is negative, add φ(n) to it to get a positive value.
5. The value of x will be the private exponent d.

- The code for this task:

```
#include <stdio.h>
#include <openssl/bn.h>
#define NBITS 256

void printBN(char *msg, BIGNUM * a)
{
  char * number_str = BN_bn2hex(a);
  printf("%s %s\n", msg, number_str);
  OPENSSL_free(number_str);
}


int main ()
{
  BN_CTX *ctx = BN_CTX_new();
  BIGNUM *p = BN_new();
  BIGNUM *q = BN_new();
  BIGNUM *e = BN_new();
  BIGNUM *n = BN_new();
  BIGNUM *etf_of_n = BN_new();
  BIGNUM *d = BN_new();
  BIGNUM *p_minus1 = BN_new();
  BIGNUM *q_minus1 = BN_new();
```

```
  // Initialize p, q, e
  BN_hex2bn(&p, "F7E75FDC469067FFDC4E847C51F452DF");
  BN_hex2bn(&q, "E85CED54AF57E53E092113E62F436F4F");
  BN_hex2bn(&e, "0D88C3");

  // Calculate n = p * q
  BN_mul(n, p, q, ctx);
  printBN("n = ", n);

  // Calculate Euler's totient function of n: φ(n) = (p-1) * (q-1)
  BN_sub(p_minus1, p, BN_value_one());
  BN_sub(q_minus1, q, BN_value_one());
  BN_mul(etf_of_n, p_minus1, q_minus1, ctx);
  printBN("φ(n) = ", etf_of_n);

  // Calculate the modular multiplicative inverse of e modulo φ(n): d
  BN_mod_inverse(d, e, etf_of_n, ctx);
  printBN("d = ", d);

  // Free the allocated memory
  BN_free(p);
  BN_free(q);
  BN_free(e);
  BN_free(n);
  BN_free(etf_of_n);
  BN_free(d);
  BN_free(p_minus1);
  BN_free(q_minus1);
  BN_CTX_free(ctx);

  return 0;
}
```

The code defines a function called printBN, which takes a string and a BIGNUM pointer as inputs and prints the string followed by the hexadecimal representation of the BIGNUM. The code also defines a function called BN_mod_inverse_extended_euclidean, which implements the extended Euclidean algorithm to find the modular multiplicative inverse of a BIGNUM modulo another BIGNUM.

In the main function, several BIGNUM variables are initialised, including p, q, and e. The code then calculates n, which is the product of p and q, and prints its value using the printBN function. The code also calculates Euler's totient function of n, which is (p-1)*(q-1), and prints its value using the printBN function. The code then uses the BN_mod_inverse function to calculate the modular multiplicative inverse of e modulo the totient function of n, and prints the result using the printBN function. Finally, the allocated memory is freed using the BN_free function, and the BN_CTX_free function is used to free the context.

- Compiling and running the code:

```
[04/29/23]seed@Attacker:~/tasks$ ./a.out
n =  E103ABD94892E3E74AFD724BF28E78366D9676BCCC70118BD0AA1968DBB143D1
φ(n) =  E103ABD94892E3E74AFD724BF28E78348D52298BD687C44DEB3A81065A7981A4
d =  3587A24598E5F2A21DB007D89D18CC50ABA5075BA19A33890FE7C28A9B496AEB
[04/29/23]seed@Attacker:~/tasks$
```

We can see printed all the **params we calculated to get d param**, and the **value of d param** itself. The private key, is going to be the pairing between the big number d and n -> (n,d).

- To calculate a pair of keys manually, we can generate P and Q. Using BN_generate_prime_ex() to generate a random Big Number.

```c
#include <stdio.h>
#include <stdlib.h>
#include <openssl/rand.h>
#include <openssl/bn.h>
#include <openssl/crypto.h>

int main(){
    BN_CTX *ctx = BN_CTX_new();
    BIGNUM *p_minus1 = BN_new();
    BIGNUM *q_minus1 = BN_new();
    BIGNUM *etf_of_n = BN_new();
    BIGNUM *p = BN_new();
    BIGNUM *q = BN_new();
    BIGNUM *e = BN_new();
    BIGNUM *n = BN_new();

// Generate a 128-bit safe prime
    BN_generate_prime_ex(p, 128, 1, NULL, NULL, NULL);
    BN_generate_prime_ex(q, 128, 1, NULL, NULL, NULL);
    BN_hex2bn(&e, "0D88C3");

    char *p_hex_str = BN_bn2hex(p);  // convert the prime to hexadecimal format
    char *q_hex_str = BN_bn2hex(q);  // convert the prime to hexadecimal format
    printf("Generated p: %s\n", p_hex_str);
    printf("Generated q: %s\n", q_hex_str);


// Calculate n = p * q
    BN_mul(n,p,q,ctx);
    char *n_hex_str = BN_bn2hex(n);  // convert the n to hexadecimal format
    printf("n: %s\n", n_hex_str);
```

```
// Calculate Euler's totient function of n = (p-1)*(q-1)
    BN_sub(p_minus1, p, BN_value_one());
    BN_sub(q_minus1, q, BN_value_one());
    BN_mul(etf_of_n, p_minus1, q_minus1, ctx);

    char *etf_hex_str = BN_bn2hex(etf_of_n);  // convert the etf to hexadecimal format
    printf("etf: %s\n", etf_hex_str);


    OPENSSL_free(etf_hex_str);
    OPENSSL_free(p_hex_str);
    OPENSSL_free(q_hex_str);
    OPENSSL_free(n_hex_str);

    BN_free(p_minus1);
    BN_free(q_minus1);
    BN_free(etf_of_n);
    BN_CTX_free(ctx);
    BN_free(p);
    BN_free(q);
    BN_free(e);
    BN_free(n);

    return 0;
}
```

- Let's break it down:
    - First we generate two Big Numbers with "BN_generate_prime_ex(BIGNUM *ret, int bits, int safe, const BIGNUM *add, const BIGNUM *rem, BN_GENCB *cb)" which it parameters are:
        - **ret**: a pointer to a BIGNUM that will hold the generated prime number. This parameter must be pre-allocated by the caller.
        - **bits**: the number of bits that the generated prime number should have. The generated prime number will be greater than or equal to 2^(bits-1), and less than 2^bits.
        - **safe**: a flag that determines whether the generated prime should be "safe". A safe prime is a prime number of the form p=2q+1, where q is also a prime number.

Setting this flag to a non-zero value ensures that the generated prime is safe. A safe prime is useful in some cryptographic applications, such as generating Diffie-Hellman parameters

- **add**: an optional BIGNUM that will be added to the generated prime number. This parameter can be used to ensure that the generated prime is congruent to a specific residue modulo some value.
- **rem**: an optional BIGNUM that will be subtracted from the generated prime number. This parameter can be used to ensure that the generated prime is congruent to a specific residue modulo some value.
- **cb**: an optional callback function that can be used to provide feedback on the progress of the prime generation process. This parameter can be set to NULL if no callback function is needed.

- The rest of the steps are exactly as we did before

Task 2: Encrypting a Message

The task requires the encryption of a given message "A top secret!" using RSA encryption. The public key is provided as (e, n). The message needs to be converted from an ASCII string to a hexadecimal string, and then converted to a BIGNUM using the BN_hex2bn() API provided by the OpenSSL library. The encrypted message can then be obtained by raising the hexadecimal representation of the message to the power e modulo n using the BN_mod_exp() function provided by the OpenSSL library.

The provided keys:

- **n** = DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5
- **e** = 010001 (this hex value equals to decimal 65537)
- **M** = A top secret!
- **d** = 74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D

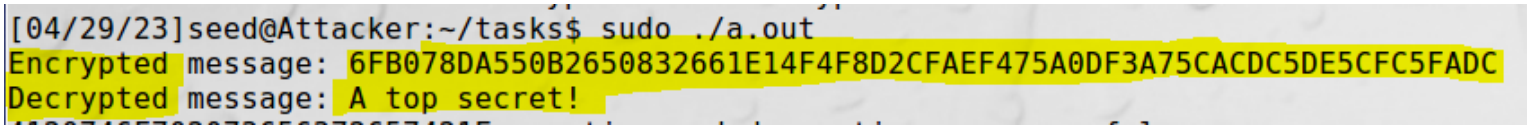
The code for this task:

```c
void decrypt(BIGNUM *C, BIGNUM *d, BIGNUM *n, char **result_str)
{
  BN_CTX *ctx = BN_CTX_new();
  BIGNUM *M = BN_new();

  // Decrypt the message using the private key (d, n)
  BN_mod_exp(M, C, d, n, ctx);

  // Convert the decrypted message to a hexadecimal string
  *result_str = BN_bn2hex(M);

  // Convert the hexadecimal string back to an ASCII string
  char *message_str = (char*) malloc(strlen(*result_str) / 2 + 1);
  int i;
  for (i = 0; i < strlen(*result_str) / 2; i++)
    sscanf(*result_str + i * 2, "%02X", message_str + i);
  message_str[strlen(*result_str) / 2] = '\0';

  // Print the decrypted message
  printf("Decrypted message: %s\n", message_str);

  // Free memory
  BN_free(M);
  BN_CTX_free(ctx);
  free(message_str);
}
```

```c
int main()
{
  BIGNUM *n = BN_new();
  BIGNUM *e = BN_new();
  BIGNUM *d = BN_new();
  BIGNUM *M = BN_new();
  BIGNUM *C = BN_new();
  BN_CTX *ctx = BN_CTX_new();

  // Initialize n, e, d to the given values
  BN_hex2bn(&n, "DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5");
  BN_hex2bn(&e, "010001");
  BN_hex2bn(&d, "74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D");

  // Convert the message to a hexadecimal string
  char *message_str = "A top secret!";
  char *hex_str = (char*) malloc(strlen(message_str) * 2 + 1);
  int i;
  for (i = 0; i < strlen(message_str); i++)
    sprintf(hex_str + i * 2, "%02X", message_str[i]);

  // Convert the hexadecimal string to a BIGNUM
  BN_hex2bn(&M, hex_str);
  free(hex_str);

  // Encrypt the message using the public key (e, n)
  BN_mod_exp(C, M, e, n, ctx);

  // Print the encrypted message as a hexadecimal string
  char *result_str = BN_bn2hex(C);
  printf("Encrypted message: %s\n", result_str);

  // Decrypt the message using the private key (d, n)
  decrypt(C, d, n, &result_str);


  // Free memory
  BN_free(n);
  BN_free(e);
  BN_free(d);
  BN_free(M);
  BN_free(C);
  return 0;
}
```

The program **first initialises** the public key (n and e) and private key (d) as BIGNUM values. The message to be encrypted is then **converted to a hexadecimal string and subsequently to a BIGNUM**. The message is encrypted using the public key by computing the modular exponentiation of the message with the public exponent (e) modulo n. The resulting ciphertext is then printed as a hexadecimal string. To decrypt the message, the **ciphertext is first converted to a BIGNUM**, and then the modular exponentiation is computed with the private exponent (d) modulo n. The decrypted message is then converted back to ASCII characters and printed. Finally, all allocated memory is freed. The function decrypt takes the ciphertext, private key, and modulus as input arguments and outputs the decrypted message as a string.

- Compiling and Running the code:



We can see in the result that the encrypted value really is a big number as expected, which shows that the task was successful. Also, the decrypted message is the original one, so indeed, the encrypted number is correct and the process works as expected.

Task 3: Decrypting a Message

In this task, the goal is to decrypt a given ciphertext using the private key that corresponds to the public key used in Task 2. The ciphertext is represented as a hexadecimal string, and needs to be converted to a BIGNUM before being decrypted using the RSA private key. The

decrypted message is then converted back to an ASCII string and printed to the console. The decryption process involves modular exponentiation, which is done using the BN_mod_exp() function from the OpenSSL library.

- The code for this task: (including decryption function from before)

```
int main()
{
    BIGNUM *n = BN_new();
    BIGNUM *e = BN_new();
    BIGNUM *d = BN_new();
    BIGNUM *M = BN_new();
    BIGNUM *C = BN_new();
    BN_CTX *ctx = BN_CTX_new();

    // Initialize n, e, d to the given values
    BN_hex2bn(&n, "DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5");
    BN_hex2bn(&e, "010001");
    BN_hex2bn(&d, "74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D");

    // Convert the ciphertext to a BIGNUM
    BN_hex2bn(&C, "8C0F971DF2F3672B28811407E2DABBE1DA0FEBBBDFC7DCB67396567EA1E2493F");

    // Decrypt the message using the private key (d, n)
    char *result_str;
    decrypt(C, d, n, &result_str);

    // Free memory
    BN_free(n);
    BN_free(e);
    BN_free(d);
    BN_free(M);
    BN_free(C);

    return 0;
}
```

The provided code is a C program that uses the OpenSSL library to decrypt an RSA encrypted message. It begins by defining a decrypt function that takes the ciphertext, private key components (d and n), and a pointer to the result string. Inside the function, the decryption is performed using the RSA algorithm, and the decrypted message is

converted to a hexadecimal string. Then, the hexadecimal string is converted back to an ASCII string and printed. In the main function, the public and private key components are initialised, and the ciphertext is provided as a hexadecimal string. The decrypt function is called with the appropriate arguments, and the decrypted message is displayed as output.

- The output after compilation and run:

```
[04/29/23]seed@Attacker:~/tasks$ sudo ./a.out
Decrypted message: Password is dees
```

We see the decrypted message, which is ‘Password is dees’ - meaning it is readable. Therefore, we were able to decrypt the message.

Task 4: Signing a Message

Signing a message refers to the process of creating a digital signature for a given piece of data or a message. This digital signature provides a way to verify the authenticity and integrity of the message. It allows a recipient to verify that the message was indeed sent by the claimed sender (authenticity) and that the message was not tampered with during transit (integrity).

The public/private keys used in this task are the same as the ones used in Task 2:

- **n** = DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5
- **e** = 010001 (this hex value equals to decimal 65537)
- **d** = 74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D

We have two similar message that we intend to sign and subsequently compare the two signatures:

- **M1** = I owe you $2000
- **M2** = I owe you $3000

- The code for this task:

```
void sign_and_verify_message(char *message_str, BIGNUM *d, BIGNUM *n, BIGNUM *e)
{
  BN_CTX *ctx = BN_CTX_new();
  BIGNUM *M = BN_new();
  BIGNUM *S = BN_new();

  // Convert the message to a hexadecimal string
  char *hex_str = (char*) malloc(strlen(message_str) * 2 + 1);
  int i;
  for (i = 0; i < strlen(message_str); i++)
    sprintf(hex_str + i * 2, "%02X", message_str[i]);

  // Convert the hexadecimal string to a BIGNUM
  BN_hex2bn(&M, hex_str);
  free(hex_str);
```

```
    // Sign the message using the private key (d, n)
    BN_mod_exp(S, M, d, n, ctx);

    // Print the signed message as a hexadecimal string
    char *signed_str = BN_bn2hex(S);
    printf("Signed message: %s\n", signed_str);

    // Verify the signature using the public key (e, n)
    BIGNUM *M2 = BN_new();
    BN_mod_exp(M2, S, e, n, ctx);

    // Convert the verified message to a hexadecimal string
    char *verified_str = BN_bn2hex(M2);

    // Convert the hexadecimal string back to an ASCII string
    char *message_str2 = (char*) malloc(strlen(verified_str) / 2 + 1);
    for (i = 0; i < strlen(verified_str) / 2; i++)
      sscanf(verified_str + i * 2, "%02X", message_str2 + i);
    message_str2[strlen(verified_str) / 2] = '\0';

    // Print the verified message
    printf("Verified message: %s\n", message_str2);

    // Free memory
    BN_free(M);
    BN_free(S);
    BN_free(M2);
    BN_CTX_free(ctx);
    free(message_str2);
  }

int main()
{
  BIGNUM *n = BN_new();
  BIGNUM *e = BN_new();
  BIGNUM *d = BN_new();

  // Initialize n, e, d to the given values
  BN_hex2bn(&n, "DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5");
  BN_hex2bn(&e, "010001");
  BN_hex2bn(&d, "74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D");

  // Sign and verify the messages
  sign_and_verify_message("I owe you $2000.", d, n, e);
  sign_and_verify_message("I owe you $3000.", d, n, e);

  // Free memory
  BN_free(n);
  BN_free(e);
  BN_free(d);

  return 0;
}
```

A breakdown of **sign_and_verify_message** function:

1. **BN_CTX_new**: This initializes a new BIGNUM context that holds temporary variables used by the BIGNUM library.
2. **BN_new**: This initializes new BIGNUM variables for the message M and the signature S.
3. The message string is converted to a hexadecimal string. This is necessary because the BIGNUM functions operate on large numbers, so the message string needs to be converted to a number. This is done by interpreting each character in the string as a hexadecimal number.
4. **BN_hex2bn**: This converts the hexadecimal string to a BIGNUM. This is the number that will be signed.
5. **BN_mod_exp**: This is the key step in the signing process. It raises the message M to the power of the private key d, modulo n. This creates the signature S.
6. **BN_bn2hex**: This converts the signature S to a hexadecimal string, which is printed as the signed message.
7. **BN_mod_exp**: This is the key step in the verification process. It raises the signature S to the power of the public key e, modulo n. This should recreate the original message M.
8. **BN_bn2hex**: This converts the verified message M to a hexadecimal string.
9. The hexadecimal string is converted back to an ASCII string. This is done by interpreting each pair of hexadecimal digits as an ASCII character.
10. The verified message is printed.

- Running the code:

```
[05/12/23]seed@Attacker:~/tasks$ task4
Signed message: 55A4E7F17F04CCFE2766E1EB32ADDBA890BBE92A6FBE2D785ED6E73CCB35E4CB
Verified message: I owe you $2000.
Signed message: BCC20FB7568E5D48E434C387C06A6025E90D29D848AF9C3EBAC0135D99305822
Verified message: I owe you $3000.
```

From the output of our program, we can observe the following:

1. The two signed messages are different. This is expected, because the two input messages are different ("I owe you $2000." and "I owe you $3000."). Even a small change in the input message will result in a completely different signature.
2. The verified messages match the original messages. This demonstrates that the signing and verification processes are working correctly. When the signature is verified using the public key, it produces the original message.

Task 5: Verifying a Signature

In this task, our objective is to verify the authenticity of a received message from Alice to Bob. The message, denoted as M, states "Launch a missile."

Along with the message, Alice has provided her signature, S. To perform the verification, we are provided with Alice's public key, represented as (e, n), where the public exponent, e, is given as 010001 (equivalent to decimal 65537), and the modulus, n, is given as

AE1CD4DC432798D933779FBD46C6E1247F0CF1233595113AA51B450F18116115.

Our verification process involves the following steps:

1. **Conversion**: We convert the hexadecimal signature, S, to its binary representation. Additionally, we convert the public key exponent, e, and modulus, n, from hexadecimal to BN.

2. **Hashing**: The original message, "Launch a missile.," is hashed using a cryptographic hash function (e.g., SHA-256) to generate a fixed-length hash value (this is in the real world but not in the scope of the task).

3. **Signature Verification**: We decrypt the signature, S, using Alice's public key (e, n) through the RSA decryption algorithm. The resulting decrypted value should match the hash of the original message M if the signature is valid.

To simulate a corrupted signature, we introduce a single-bit change in the last byte of the signature, changing it from 2F to 3F. We repeat the verification process to observe the effects of this alteration on the verification process.

Through this task, we aim to determine whether the received signature is indeed Alice's, thereby ensuring the integrity and authenticity of the message.

- The code used in this task:

```
void verify_signature(char *message_str, char *signature_str, BIGNUM *e, BIGNUM *n)
{
  BN_CTX *ctx = BN_CTX_new();
  BIGNUM *M = BN_new();
  BIGNUM *S = BN_new();

  // Convert the message to a hexadecimal string
  char *hex_str = (char*) malloc(strlen(message_str) * 2 + 1);
  int i;
  for (i = 0; i < strlen(message_str); i++)
    sprintf(hex_str + i * 2, "%02X", message_str[i]);

  // Convert the hexadecimal string to a BIGNUM
  BN_hex2bn(&M, hex_str);
  free(hex_str);

  // Convert the signature to a BIGNUM
  BN_hex2bn(&S, signature_str);

  // Verify the signature using the public key (e, n)
  BIGNUM *M2 = BN_new();
  BN_mod_exp(M2, S, e, n, ctx);

  // Convert the verified message to a hexadecimal string
  char *verified_str = BN_bn2hex(M2);

  // Convert the hexadecimal string back to an ASCII string
  char *message_str2 = (char*) malloc(strlen(verified_str) / 2 + 1);
  for (i = 0; i < strlen(verified_str) / 2; i++)
    sscanf(verified_str + i * 2, "%02X", message_str2 + i);
  message_str2[strlen(verified_str) / 2] = '\0';

  // Print the verified message
  printf("Verified message: %s\n", message_str2);

  // Check if the verified message matches the original
  if (strcmp(message_str, message_str2) == 0) {
    printf("The signature is valid.\n");
  } else {
    printf("The signature is invalid.\n");
  }

  // Free memory
  BN_free(M);
  BN_free(S);
  BN_free(M2);
  BN_CTX_free(ctx);
  free(message_str2);
```

```c
}

int main()
{
  BIGNUM *n = BN_new();
  BIGNUM *e = BN_new();

  // Initialize n, e to the given values
  BN_hex2bn(&n, "AE1CD4DC432798D933779FBD46C6E1247F0CF1233595113AA51B450F18116115");
  BN_hex2bn(&e, "010001");

  // Verify the signatures
  verify_signature("Launch a missile.", "643D6F34902D9C7EC90CB0B2BCA36C47FA37165C0005CAB026C0542CBDB6802F", e, n);
  verify_signature("Launch a missile.", "643D6F34902D9C7EC90CB0B2BCA36C47FA37165C0005CAB026C0542CBDB6803F", e, n);

  // Free memory
  BN_free(n);
  BN_free(e);

  return 0;
}
```

Code breakdown:

1. **Creating OpenSSL context and BIGNUM variables**: OpenSSL requires a BN_CTX structure to work with BIGNUMs, OpenSSL's arbitrary-precision integer type. We create a context with BN_CTX_new(), and then create several BIGNUM variables using BN_new(). M is for the original message, S is for the signature, and M2 is for the message obtained after verifying the signature.
2. **Converting the original message to a BIGNUM**: The message is initially a string of ASCII characters. We convert this to a hexadecimal string and then to a BIGNUM. This is necessary because the RSA operations work on numbers, not strings.

3. **Verifying the signature**: This is done with the BN_mod_exp() function, which calculates S to the power of e modulo n. If S is a valid signature for M, this operation will return M. Note that e and n are Alice's public key.
4. **Converting the verified message to a string**: We convert the BIGNUM M2 to a hexadecimal string, and then to an ASCII string. This is the reverse of the operation we did to convert the original message to a BIGNUM.
5. **Checking the verified message against the original**: We compare the ASCII string obtained in the previous step with the original message string. If they match, the signature is valid; if they don't, the signature is invalid.
6. **Freeing memory**: Finally, we free all the BIGNUM variables and the OpenSSL context to avoid memory leaks.

- Running the code:

```
[05/12/23]seed@Attacker:~/tasks$ task5
Verified message: Launch a missile.
The signature is valid.
The signature is invalid.
```

The result shows the output of the signature verification process for two different signatures.

1. **First Part**: The original message "Launch a missile." and the corresponding signature was used. The verified message matches the original message, which means the signature is valid. This is because when you use the correct signature with the provided public key (e, n), the decrypted signature equals the original

message. This is how digital signature verification works: if the decrypted signature equals the original message, the signature is considered valid.

2. **Second Part**: A corrupted signature was used. Even though only one bit was changed, the result of the verification process is completely different. The gibberish output "��,O�c���rm=f�:N����’" is due to the fact that a small change in the input signature results in a large change in the output, which is a characteristic of cryptographic systems. Because the verified message does not match the original message "Launch a missile.", the corrupted signature is considered invalid.

This demonstrates that if a signature is even slightly altered, it will fail verification, providing a strong guarantee of authenticity for the original, unaltered message. This is the core principle behind digital signatures and is vital for secure communication.

Task 6: Manually Verifying an X.509 Certificate

In this task, we will manually verify an X.509 certificate using our program. An X.509 contains data about a public key and an issuer's signature on the data. We will download a real X.509 certificate from a web server, get its issuer's public key, and then use this public key to verify the signature on the certificate.

Step 1: Download a certificate from a real web server

- We chose to download it from google.com:

```
[05/13/23]seed@Attacker:~/tasks$ openssl s_client -connect www.google.com:443 -showcerts
```

1. **openssl:** This is the command-line tool for the OpenSSL library, which supports a wide range of cryptographic operations.
2. **s_client**: This is an OpenSSL command that implements a generic SSL/TLS client which connects to a remote server using SSL/TLS. It's mainly used for debugging, or for creating a quick and dirty client to interact with a remote server.
3. **-connect www.google.com:443**: This tells the s_client where to connect. In this case, it's connecting to www.google.com on port 443, which is the standard port for HTTPS.
4. **-showcerts**: When the connection is made with the server, the server sends back its certificate chain. The -showcerts option tells OpenSSL to display the entire certificate chain as well (it's not shown by default).

So, in summary, this command uses OpenSSL to establish a SSL/TLS connection to www.google.com on port 443 and print out the certificate chain that the server presents during the SSL/TLS handshake.

- Analysing the output:

```
CONNECTED(00000003)
depth=3 C = BE, O = GlobalSign nv-sa, OU = Root CA, CN = GlobalSign Root CA
verify return:1
depth=2 C = US, O = Google Trust Services LLC, CN = GTS Root R1
verify return:1
depth=1 C = US, O = Google Trust Services LLC, CN = GTS CA 1C3
verify return:1
depth=0 CN = www.google.com
verify return:1
---
Certificate chain
 0 s:/CN=www.google.com
   i:/C=US/O=Google Trust Services LLC/CN=GTS CA 1C3
-----BEGIN CERTIFICATE-----
MIIFUzCCBDugAwIBAgIRAP5yxsYYmZMjCmiOza+SyOYwDQYJKoZIhvcNAQELBQAw
RjELMAkGA1UEBhMCVVMxIjAgBgNVBAoTGUdvb2dsZSBUcnVzdCBTZXJ2aWNlcyBM
TEMxEzARBgNVBAMTCkdUUyBDQSAxQzMwHhcNMjMwNDI0MTIwMTE2WhcNMjMwNzE3
MTIwMTE1WjAZMRcwFQYDVQQDEw53d3cuZ29vZ2xlLmNvbTCCASIwDQYJKoZIhvcN
AQEBBQADggEPADCCAQoCggEBAMchcbYTD0vL/9KuV4ooF8PwsI7vcOLDgC0dIjtD
8cBIslP9xcOax6pOPArBlyYHSOTVjX7Z9O3/GDZKWJlvzbxmg/c1eyYa1+kRyUvO
yVwZ/7B9/5E/L92wv1forijpfzkAeC0PY6VESa30lvwE2X28cZTJXRDsV6mklV4e
oSSz2nbpgwGzu9JbGzGqUsJasSznSrHfpxiq9ZQ4zB424t34Z/RuU5jOW1VdxKOV
lKaVqN3Ns1soQqgVDYvxmaaU8zqQ71e2+1ZH4lsPr91/R1zR1j81KdtMrdq5JXPk
NmHfmy54Ro/9RNG6ZQN+D2Y3EYC5tocFze3MlP/t6i74dK8CAwEAAaOCAmcwggJj
MA4GA1UdDwEB/wQEAwIFoDATBgNVHSUEDDAKBggrBgEFBQcDATAMBgNVHRMBAf8E
AjAAMB0GA1UdDgQWBBTRcy/CTzn/R/svYePK8jc3zrD6GzAfBgNVHSMEGDAWgBSK
dH+vhc3ulc09nNDiRhTzcTUdJzBqBggrBgEFBQcBAQReMFwwJwYIKwYBBQUHMAGG
G2h0dHA6Ly9vY3NwLnBraS5nb29nL2d0czFjMzAxBggrBgEFBQcwAoYlaHR0cDov
L3BraS5nb29nL3JlcG8vY2VydHMvZ3RzMWMzLmRlcjAZBgNVHREEEjAQgg53d3cu
Z29vZ2xlLmNvbTAhBgNVHSAEGjAYMAgGBmeBDAECATAMBgorBgEEAdZ5AgUDMDwG
A1UdHwQ1MDMwMaAvoC2GK2h0dHA6Ly9jcmxzLnBraS5nb29nL2d0czFjMy9RT3ZK
ME4xc1QyQS5jcmwwggEEBgorBgEEAdZ5AgQCBIH1BIHyAPAAdgCt9776fP8QyIud
PZwePhhqtGcpXc+xDCTKhYY069yCigAAAYezWq6LAAAEAwBHMEUCID/OmYkdM7IC
BZg4iC00JDn8rbZaHCxD13a/FHe51H7TAiEAo0JkuHeRP1G4YEk63xOFRrMDhLLF
p5se46d5VKqkNWYAdgCzc3cH4YRQ+GOG1gWp3BEJSnktsWcMC4fc8AMOeTalmgAA
AYezWq64AAAEAwBHMEUCIQCSovC4uPq4627Zt6NbAjsXx3K+2N70yDZR+8mxw+EK
bAIgAlHor/03fbe0pGe5Hm4EFUigEmetG8USBaY0+5srfLcwDQYJKoZIhvcNAQEL
BQADggEBAC3Dtlzl4jS6rPL5n2Rl9sGQu39Hvfkc9FY/CBsJ1Mm/cyMIs1fphLC1
Ww28AihjR/3QpiIYWWNHGLA0jRlaUytj8Tlt7U3JzClP23BOR0TY1gVBGGJDsDgT
6Pueejs25WerHYEPoWEBQuZTsxNZCu9rpd8PQca7fK8T01vaex7B6u9l+yYRi6Pd
ZfyKnmkPKRuZJIPtcVwzaZL/ulF0oHYq4KquLwKu68FjJrzdj06xfZ3v9LExfEVP
mprubeoXk7d5YAqprZN5VGqmG134o3yBBzfoyXZ5afcRlCBc3H+SsoDMtHRCJTxW
cmH/aLQzewNRuPkg3qhDn3S+DBUV9A4=
```

```
 1 s:/C=US/O=Google Trust Services LLC/CN=GTS CA 1C3
   i:/C=US/O=Google Trust Services LLC/CN=GTS Root R1
-----BEGIN CERTIFICATE-----
MIIFljCCA36gAwIBAgINAgO8U1lrNMcY9QFQZjANBgkqhkiG9w0BAQsFADBHMQsw
CQYDVQQGEwJVUzEiMCAGA1UEChMZR29vZ2xlIFRydXN0IFNlcnZpY2VzIExMQzEU
MBIGA1UEAxMLR1RTIFJvb3QgUjEwHhcNMjAwODEzMDAwMDQyWhcNMjcwOTMwMDAw
MDQyWjBGMQswCQYDVQQGEwJVUzEiMCAGA1UEChMZR29vZ2xlIFRydXN0IFNlcnZp
Y2VzIExMQzETMBEGA1UEAxMKR1RTIENBIDFDMzCCASIwDQYJKoZIhvcNAQEBBQAD
ggEPADCCAQoCggEBAPWI3+dijB43+DdCkH9sh9D7ZYIl/ejLa6T/belaI+KZ9hzp
kgOZE3wJCor6QtZeViSqejOEH9Hpabu5dOxXTGZok3c3VVP+ORBNtzS7XyV3NzsX
lOo85Z3VvMO0Q+sup0fvsEQRY9i0QYXdQTBIkxu/t/bgRQIh4JZCF8/ZK2VWNAcm
BA2o/X3KLu/qSHw3TT8An4Pf73WELnlXXPxXbhqW//yMmqaZviXZf5YsBvcRKgKA
gOtjGDxQSYflispfGStZloEAoPtR28p3CwvJlk/vcEnHXG0g/Zm0tOLKLnf9LdwL
tmsTDIwZKxeWmLnwi/agJ7u2441Rj72ux5uxiZ0CAwEAAaOCAYAwggF8MA4GA1Ud
DwEB/wQEAwIBhjAdBgNVHSUEFjAUBggrBgEFBQcDAQYIKwYBBQUHAwIwEgYDVR0T
AQH/BAgwBgEB/wIBADAdBgNVHQ4EFgQUinR/r4XN7pXNPZzQ4kYU83E1HScwHwYD
VR0jBBgwFoAU5K8rJnEaK0gnhS9SZizv8IkTcT4waAYIKwYBBQUHAQEEXDBaMCYG
CCsGAQUFBzABhhpodHRwOi8vb2NzcC5wa2kuZ29vZy9ndHNyMTAwBggrBgEFBQcw
AoYkaHR0cDovL3BraS5nb29nL3JlcG8vY2VydHMvZ3RzcjEuZGVyMDQGA1UdHwQt
MCswKaAnoCWGI2h0dHA6Ly9jcmwucGtpLmdvb2cvZ3RzcjEvZ3RzcjEuY3JsMFcG
A1UdIARQME4wOAYKKwYBBAHWeQIFAzAqMCgGCCsGAQUFBwIBFhxodHRwczovL3Br
aS5nb29nL3JlcG9zaXRvcnkvMAgGBmeBDAECATAIBgZngQwBAgIwDQYJKoZIhvcN
```

```
 2 s:/C=US/O=Google Trust Services LLC/CN=GTS Root R1
   i:/C=BE/O=GlobalSign nv-sa/OU=Root CA/CN=GlobalSign Root CA
-----BEGIN CERTIFICATE-----
MIIFYjCCBEqgAwIBAgIQd70NbNs2+RrqIQ/E8FjTDTANBgkqhkiG9w0BAQsFADBX
MQswCQYDVQQGEwJCRTEZMBcGA1UEChMQR2xvYmFsU2lnbiBudi1zYTEQMA4GA1UE
CxMHUm9vdCBDQTEbMBkGA1UEAxMSR2xvYmFsU2lnbiBSb290IENBMB4XDTIwMDYx
OTAwMDA0MloXDTI4MDEyODAwMDA0MlowRzELMAkGA1UEBhMCVVMxIjAgBgNVBAoT
GUdvb2dsZSBUcnVzdCBTZXJ2aWNlcyBMTEMxFDASBgNVBAMTC0dUUyBSb290IFIx
MIICIjANBgkqhkiG9w0BAQEFAAOCAg8AMIICCgKCAgEAthECix7joXebO9y/lD63
ladAPKH9gvl9MgaCcfb2jH/76Nu8ai6Xl6OMS/kr9rH5zoQdsfnFl97vufKj6bwS
iV6nqlKr+CMny6SxnGPb15l+8Ape62im9MZaRw1NEDPjTrETo8gYbEvs/AmQ351k
KSUjB6G00j0uYODP0gmHu81I8E3CwnqIiru6z1kZ1q+PsAewnjHxgsHA3y6mbWwZ
DrXYfiYaRQM9sHmklCitD38m5agI/pboPGiUU+6DOogrFZYJsuB6jC511pzrp1Zk
j5ZPaK49l8KEj8C8QMALXL32h7M1bKwYUH+E4EzNktMg6TO8UpmvMrUpsyUqtEj5
cuHKZPfmghCN6J3Cioj6OGaK/GP5Afl4/Xtcd/p2h/rs37EOeZVXtL0m79YB0esW
CruOC7XFxYpVq9Os6pFLKcwZpDIlTirxZUTQAs6qzkm06p98g7BAe+dDq6dso499
iYH6TKX/1Y7DzkvgtdizjkXPdsDtQCv9Uw+wp9U7DbGKogPeMa3Md+pvez7W35Ei
Eua++tgy/BBjFFFy3l3WFpO9KWgz7zpm7AeKJt8T11dleCfeXkkUAKIAf5qoIbap
sZWwpbkNFhHax2xIPEDgfg1azVY80ZcFuctL7TlLnMQ/0lUTbiSw1nH69MG6zO0b
9f6BQdgAmD06yK56mDcYBZUCAwEAAaOCATgwggE0MA4GA1UdDwEB/wQEAwIBhjAP
BgNVHRMBAf8EBTADAQH/MB0GA1UdDgQWBBTkrysmcRorSCeFL1JmLO/wiRNxPjAf
BgNVHSMEGDAWgBRge2YaRQ2XyolQL30EzTSo//z9SzBgBggrBgEFBQcBAQRUMFIw
JQYIKwYBBQUHMAGGGWh0dHA6Ly9vY3NwLnBraS5nb29nL2dzcjEwKQYIKwYBBQUH
MAKGHWh0dHA6Ly9wa2kuZ29vZy9nc3IxL2dzcjEuY3J0MDIGA1UdHwQrMCkwJ6Al
oCOGIWh0dHA6Ly9jcmwucGtpLmdvb2cvZ3NyMS9nc3IxLmNybDA7BgNVHSAENDAy
MAgGBmeBDAECATAIBgZngQwBAgIwDQYLKwYBBAHWeQIFAwIwDQYLKwYBBAHWeQIF
AwMwDQYJKoZIhvcNAQELBQADggEBADSkHrEoo9C0dhemMXoh6dFSPsjbdBZBiLg9
NR3t5P+T4Vxfq7vqfM/b5A3Ri1fyJm9bvhdGaJQ3b2t6yMAYN/olUazsaL+yyEn9
WprKASOshIArAoyZl+tJaox118fessmXn1hIVw41oeQa1v1vg4Fv74zPl6/AhSrw
9U5pCZEt4Wi4wStz6dTZ/CLANx8LZh1J7QJVj2fhMtfTJr9w4z30Z209fOU0iOMy
+qduBmpvvYuR7hZL6Dupszfnw0Skfths18dG9ZKb59UhvmaSGZRVbNQpsg3BZlvi
d0lIKO2d1xozclOzgjXPYovJJIultzkMu34qQb9Sz/yilrbCgj8=
-----END CERTIFICATE-----
```

```
Server certificate
subject=/CN=www.google.com
issuer=/C=US/O=Google Trust Services LLC/CN=GTS CA 1C3
---
No client certificate CA names sent
Peer signing digest: SHA256
Server Temp Key: ECDH, P-256, 256 bits
---
SSL handshake has read 4905 bytes and written 431 bytes
---
New, TLSv1/SSLv3, Cipher is ECDHE-RSA-AES128-GCM-SHA256
Server public key is 2048 bit
Secure Renegotiation IS supported
Compression: NONE
Expansion: NONE
No ALPN negotiated
```

```
SSL-Session:
    Protocol  : TLSv1.2
    Cipher    : ECDHE-RSA-AES128-GCM-SHA256
    Session-ID: 5CC886E7D33ADCE5C412F24EF7B8A634356A639F1902BCFB6C2319835FB3B81C
    Session-ID-ctx:
    Master-Key: 0934B42E38982DE28F1FF44CFF66F3BE6DF765B3E1F7AD06B232D65C3DC6127A5F3
    Key-Arg   : None
    PSK identity: None
    PSK identity hint: None
    SRP username: None
    TLS session ticket lifetime hint: 100800 (seconds)
    TLS session ticket:
    0000 - 02 1b 81 4a 01 05 f6 dc-6e 12 3e c8 f4 89 c5 ff   ...J....n.>.....
    0010 - af ae 7d 80 eb 8c ec dc-e9 9d 56 a2 4e ad 49 fe   ..}.......V.N.I.
    0020 - 41 e3 26 32 8d 53 c1 82-31 d3 1d 66 c3 88 dc 01   A.&2.S..1..f....
    0030 - 81 7b 4d 55 ae 39 e9 7e-72 cc 66 6f 1c 1f 97 a1   .{MU.9.~r.fo....
    0040 - b0 1d 1f 8c e2 dd da 14-99 9f 5d 15 1f 0e 13 6f   ..........]....o
    0050 - 3f 31 a6 d2 56 64 88 8a-94 da cc d9 63 40 eb 75   ?1..Vd......c@.u
    0060 - 0d 7c 4d 2b 11 23 81 02-02 31 50 f0 1e dd c0 6b   .|M+.#...1P....k
    0070 - 33 13 23 7a d7 25 11 75-74 27 03 b4 38 fa c4 72   3.#z.%.ut'..8..r
    0080 - 08 d9 dc ec f1 04 5e 7b-36 cd 92 98 5e 05 c8 b7   ......^{6...^...
    0090 - a6 af f6 2f 1e 53 f5 03-56 d5 2b 9b bf 56 95 73   .../.S..V.+..V.s
    00a0 - 2f 4e 5a e4 18 b4 e2 46-a5 ea 3a 0d 7f a7 c3 41   /NZ....F..:....A
    00b0 - f3 93 54 3e c2 a9 5b 86-53 ec 1a af 2d 93 ce 3a   ..T>..[.S...-..:
    00c0 - 93 4d 9e 08 a7 79 7c 3f-cc 34 4b 35 bb 33 2a 39   .M...y|?.4K5.3*9
    00d0 - d0 bc 65 44 d3 54 4c 56-a6 15 d8 75 ac d5 ae bb   ..eD.TLV...u....

    Start Time: 1683959437
    Timeout   : 300 (sec)
    Verify return code: 0 (ok)
---
```

The output shows several stages of information:

1. **CONNECTED(00000003)**: This indicates that a successful TCP connection was established with www.google.com on port 443 (HTTPS).

2. **Certificate chain verification**: The following lines show the certification path (also known as the chain of trust), from the root certificate to the end-entity certificate.

```
depth=3 C = BE, O = GlobalSign nv-sa, OU = Root CA, CN = GlobalSign Root CA
verify return:1
depth=2 C = US, O = Google Trust Services LLC, CN = GTS Root R1
verify return:1
depth=1 C = US, O = Google Trust Services LLC, CN = GTS CA 1C3
verify return:1
depth=0 CN = www.google.com
verify return:1
```

   The verify return:1 after each certificate means that the certificate passed the validation check. The depth starts from 0, which is the certificate of the site we are connecting to, up to the root certificate. Each step up the chain is a certification authority that has been used to sign the next certificate down.

3. **Certificate chain**: This section shows the actual certificates used in the chain. Each certificate is presented in PEM format, which is a Base64 encoded view of the actual certificate. We can decode this information to see the details of each certificate. The certificate chain starts from the server certificate (depth 0) up to the root certificate.

```
Certificate chain
 0 s:/CN=www.google.com
   i:/C=US/O=Google Trust Services LLC/CN=GTS CA 1C3
-----BEGIN CERTIFICATE-----
MIIFUzCCBDugAwIBAgIRAP5yxsYYmZMjCmiOza+SyOYwDQYJKoZIhvcNAQELBQAw
```

Each certificate has a subject (s:) and issuer (i:). The issuer of one certificate in the chain should be the subject of the next certificate up in the chain.

4. Breaking down the last part:

```
SSL handshake has read 4905 bytes and written 431 bytes
---
New, TLSv1/SSLv3, Cipher is ECDHE-RSA-AES128-GCM-SHA256
Server public key is 2048 bit
Secure Renegotiation IS supported
Compression: NONE
Expansion: NONE
No ALPN negotiated
SSL-Session:
    Protocol  : TLSv1.2
    Cipher    : ECDHE-RSA-AES128-GCM-SHA256
    Session-ID: 5CC886E7D33ADCE5C412F24EF7B8A634356A639F1902BCFB6C2319835FB3B81C
    Session-ID-ctx:
    Master-Key: 0934B42E38982DE28F1FF44CFF66F3BE6DF765B3E1F7AD06B232D65C3DC6127A
    Key-Arg   : None
    PSK identity: None
    PSK identity hint: None
    SRP username: None
    TLS session ticket lifetime hint: 100800 (seconds)
    TLS session ticket:
    0000 - 02 1b 81 4a 01 05 f6 dc-6e 12 3e c8 f4 89 c5 ff   ...J....n.>.....
    0010 - af ae 7d 80 eb 8c ec dc-e9 9d 56 a2 4e ad 49 fe   ..}.......V.N.I.
    0020 - 41 e3 26 32 8d 53 c1 82-31 d3 1d 66 c3 88 dc 01   A.&2.S..1..f....
    0030 - 81 7b 4d 55 ae 39 e9 7e-72 cc 66 6f 1c 1f 97 a1   .{MU.9.~r.fo....
    0040 - b0 1d 1f 8c e2 dd da 14-99 9f 5d 15 1f 0e 13 6f   ..........]....o
    0050 - 3f 31 a6 d2 56 64 88 8a-94 da cc d9 63 40 eb 75   ?1..Vd......c@.u
    0060 - 0d 7c 4d 2b 11 23 81 02-02 31 50 f0 1e dd c0 6b   .|M+.#...1P....k
    0070 - 33 13 23 7a d7 25 11 75-74 27 03 b4 38 fa c4 72   3.#z.%.ut'..8..r
    0080 - 08 d9 dc ec f1 04 5e 7b-36 cd 92 98 5e 05 c8 b7   ......^{6...^...
    0090 - a6 af f6 2f 1e 53 f5 03-56 d5 2b 9b bf 56 95 73   .../.S..V.+..V.s
    00a0 - 2f 4e 5a e4 18 b4 e2 46-a5 ea 3a 0d 7f a7 c3 41   /NZ....F..:....A
    00b0 - f3 93 54 3e c2 a9 5b 86-53 ec 1a af 2d 93 ce 3a   ..T>..[.S...-..:
    00c0 - 93 4d 9e 08 a7 79 7c 3f-cc 34 4b 35 bb 33 2a 39   .M...y|?.4K5.3*9
    00d0 - d0 bc 65 44 d3 54 4c 56-a6 15 d8 75 ac d5 ae bb   ..eD.TLV...u....

    Start Time: 1683959437
    Timeout   : 300 (sec)
    Verify return code: 0 (ok)
```

No client certificate CA names sent: This line tells us that the server did not request a client-side SSL certificate. In some configurations, servers can request a certificate from the client to verify its identity.

Peer signing digest: SHA256: This tells us that SHA256 is the cryptographic hash function used for creating the digital signature for this SSL/TLS connection.

Server Temp Key: ECDH, P-256, 256 bits: This line tells us that the server used Ephemeral Elliptic Curve Diffie-Hellman (ECDHE) for the key exchange with a 256-bit key. This provides forward secrecy.

SSL handshake has read 4905 bytes and written 431 bytes: This line tells us how much data was transferred during the SSL/TLS handshake.

New, TLSv1/SSLv3, Cipher is ECDHE-RSA-AES128-GCM-SHA256: This line tells us that a new session has been created using the TLSv1/SSLv3 protocol. The cipher suite used for this session is ECDHE-RSA-AES128-GCM-SHA256.

Server public key is 2048 bit: This tells us that the server's public key is 2048 bits long.

Secure Renegotiation IS supported: This tells us that the server supports secure renegotiation, which is a feature that helps prevent certain types of attacks.

Compression: NONE, Expansion: NONE: These lines tell us that there is no compression or expansion used in this SSL/TLS connection.

No ALPN negotiated: This tells us that there was no Application-Layer Protocol Negotiation (ALPN). ALPN allows the application layer to negotiate which protocol should be performed over a secure connection.

SSL-Session: This begins the section that contains details about the SSL/TLS session.

Protocol : TLSv1.2: This tells us that the protocol used for this session is TLSv1.2.

Cipher : ECDHE-RSA-AES128-GCM-SHA256: This is the cipher suite used for this session.

Session-ID, Session-ID-ctx, Master-Key, Key-Arg, PSK identity, PSK identity hint, SRP username: These lines provide additional details about the SSL/TLS session.

TLS session ticket lifetime hint: 100800 (seconds): This line gives the lifetime of the TLS session ticket, which can be used for session resumption.

TLS session ticket: This is the actual TLS session ticket. It allows the client and server to resume their session without needing to renegotiate the security parameters.

Start Time, Timeout, Verify return code: These lines provide additional details about the session. The "Start Time" is when the session started, the "Timeout" is how long the session will last before it times out, and the "Verify return code" tells you whether the verification process completed successfully (0 means there were no errors).

Important note:

1. **Issuer**: This field contains the name of the Certificate Authority (CA) that issued and signed the certificate. For a root certificate, the issuer is the certificate itself.
2. **Subject**: This field contains the name of the entity (person, organisation, server, etc.) the certificate is issued to. This is typically the owner of the public key that the certificate is certifying. For a root certificate, the subject is the certificate itself.

What we can learn from the output:

- We can see the entire certificate chain, from the web server's certificate up to the root certificate. This can be useful for diagnosing trust issues. If any certificate in the chain is not trusted, then the web server's certificate will not be trusted.
- We can see who issued each certificate in the chain.
- We can see the actual certificates, and could decode them to see all the details (validity period, public key, signature, etc.).

- Next, creating c0.pem, c1.pem and c2.pem

c0:

www.google.com

Identity: www.google.com
Verified by: GTS CA 1C3
Expires: 07/17/2023

Details

Subject Name

CN (Common Name):	www.google.com

Issuer Name

C (Country):	US
O (Organization):	Google Trust Services LLC
CN (Common Name):	GTS CA 1C3

Issued Certificate

Version:	3
Serial Number:	00 FE 72 C6 C6 18 99 93 23 0A 68 8E CD AF 92 C8 E6
Not Valid Before:	2023-04-24
Not Valid After:	2023-07-17

Certificate Fingerprints

SHA1:	29 CC BA E0 C8 4C 86 03 63 05 26 CE 7E 2D 8E 8B 7C 2D 97 73
MD5:	BB 92 8C EB 8A E0 03 61 65 0E F5 1A 53 83 E1 2E

Public Key Info

Key Algorithm:	RSA
Key Parameters:	05 00
Key Size:	2048
Key SHA1 Fingerprint:	76 CC 1D 21 64 B7 6E C1 19 EE B7 AE 8B 3D 46 E9 D1 B7 F3 D4
Public Key:	30 82 01 0A 02 82 01 01 00 C7 21 71 B6 13 0F 4B CB FF D2 AE 57 8 D9 F4 ED FF 18 36 4A 58 99 6F CD BC 66 83 F7 35 7B 26 1A D7 E9 1 71 94 C9 5D 10 EC 57 A9 A4 95 5E 1E A1 24 B3 DA 76 E9 83 01 B3 B A6 95 A8 DD CD B3 5B 28 42 A8 15 0D 8B F1 99 A6 94 F3 3A 90 EF 5 7E 0F 66 37 11 80 B9 B6 87 05 CD ED CC 94 FF ED EA 2E F8 74 AF 0

Key Usage

The certificate includes a lot of important information. Here's an explanation of the main components:

1. **www.google.com**
   Identity: www.google.com
   Verified by: GTS CA 1C3
   Expires: 07/17/2023

This shows the website or server that the certificate belongs to (www.google.com), who it was verified by (Google Trust Services CA - GTS CA 1C3), and when the certificate will expire.

2. **Subject Name**

   CN (Common Name): www.google.com

   The Subject Name is who the certificate is issued to. In this case, it is issued to www.google.com.

3. **Issuer Name**

   C (Country): US

   O (Organization): Google Trust Services LLC

   CN (Common Name): GTS CA 1C3

   The Issuer Name is the entity that has issued the certificate, which in this case is Google Trust Services LLC, a certificate authority (CA) based in the US.

4. **Issued Certificate**

   Not Valid Before: 2023-04-24

   Not Valid After: 2023-07-17

   These dates specify the period during which the certificate is valid.

5. **Certificate Fingerprints**

   SHA1: and MD5:

   These are fingerprints of the certificate. They're unique to each certificate and can be used to verify its integrity.

6. **Public Key Info**

Key Algorithm: RSA

Key Size: 2048

Public Key:

This section contains information about the public key associated with the certificate, including the algorithm used (RSA), the key size (2048 bits), and the actual public key.

7. **Key Usage**

   Usages: Digital signatureKey encipherment

   Critical: Yes

   Key usage defines the purposes for which the public key may be used. In this case, it can be used for digital signatures and key encipherment.

8. **Extended Key Usage**

   Allowed Purposes: Server Authentication

   Critical: No

   The Extended Key Usage field indicates additional purposes for which the certified public key may be used. Here, it can be used for server authentication.

9. **Basic Constraints**

   Certificate Authority: No

   Max Path Length: Unlimited

   Critical: Yes

   This indicates that the certificate is not a Certificate Authority (meaning it can't issue other certificates), and the maximum length of the certification path is unlimited.

10. Subject Key Identifier

Key Identifier:

This is an identifier for the public key in the certificate. It can help in the process of finding a specific key and is also used in various protocols.

c1:

GTS CA 1C3

Identity: GTS CA 1C3
Verified by: GTS Root R1
Expires: 09/30/2027

Details

Subject Name

C (Country):	US
O (Organization):	Google Trust Services LLC
CN (Common Name):	GTS CA 1C3

Issuer Name

C (Country):	US
O (Organization):	Google Trust Services LLC
CN (Common Name):	GTS Root R1

Issued Certificate

Version:	3
Serial Number:	02 03 BC 53 59 6B 34 C7 18 F5 01 50 66
Not Valid Before:	2020-08-13
Not Valid After:	2027-09-30

Certificate Fingerprints

SHA1:	1E 7E F6 47 CB A1 50 28 1C 60 89 72 57 10 28 78 C4 BD 8C DC
MD5:	17 8E F1 83 43 CC C9 E0 EC B0 E3 8D 9D EA 03 D8

Public Key Info

Key Algorithm:	RSA
Key Parameters:	05 00
Key Size:	2048
Key SHA1 Fingerprint:	34 69 BB FA 7C 26 15 12 8D 8D 9B D6 1E B3 14 A0 1D E2 29 DB
Public Key:	30 82 01 0A 02 82 01 01 00 F5 88 DF E7 62 8C 1E 37 F8 37 42 90 7F 6 84 1F D1 E9 69 BB B9 74 EC 57 4C 66 68 93 77 37 55 53 FE 39 10 4D B B7 F6 E0 45 02 21 E0 96 42 17 CF D9 2B 65 56 34 07 26 04 0D A8 FD 7 F7 11 2A 02 80 80 EB 63 18 3C 50 49 87 E5 8A CA 5F 19 2B 59 96 81 0 96 98 B9 F0 8B F6 A0 27 BB B6 E3 8D 51 8F BD AE C7 9B B1 89 9D 02 0

Key Usage

Usages:	Digital signature Revocation list signature
Critical:	Yes

Extended Key Usage

Allowed Purposes:	Server Authentication Client Authentication
Critical:	No

The differences between the first and second certificates are as follows:

1. **Identity and Issuer**: The first difference lies in the identity of the certificate and its issuer. The first certificate was 'GTS CA 1D4' and was issued by 'GTS Root R3'. The second certificate is 'GTS CA 1C3' and was issued by 'GTS Root R1'. This indicates that the certificates represent different entities within Google's infrastructure, and they were issued by different root certificates, which might suggest a change or evolution in Google's internal Certificate Authority (CA) hierarchy.

2. **Expiration Date and Validity Period**: The second certificate expires almost a year before the first one, which means it has a shorter lifespan. This could suggest a move towards shorter certificate lifetimes, which is a general industry trend for improving security.

3. **Key Usage**: The first certificate includes 'Key encipherment' in its usage, which is absent in the second certificate. This suggests that the second certificate is not intended to be used for encrypting keys, which could be a reflection of the specific security requirements of the entity it represents.

4. **Public Key Info**: The public keys in both certificates are different, and so are their fingerprints. This difference is not just unique identifiers; it's fundamental to how each certificate is used in secure communications. The entities that these certificates represent would have different key pairs, reflecting their distinct roles in Google's infrastructure.

c2:

GTS Root R1

Identity: GTS Root R1
Verified by: GlobalSign Root CA
Expires: 01/28/2028

Details

Subject Name

C (Country):	US
O (Organization):	Google Trust Services LLC
CN (Common Name):	GTS Root R1

Issuer Name

C (Country):	BE
O (Organization):	GlobalSign nv-sa
OU (Organizational Unit):	Root CA
CN (Common Name):	GlobalSign Root CA

Issued Certificate

Version:	3
Serial Number:	77 BD 0D 6C DB 36 F9 1A EA 21 0F C4 F0 58 D3 0D
Not Valid Before:	2020-06-19
Not Valid After:	2028-01-28

Certificate Fingerprints

SHA1:	08 74 54 87 E8 91 C1 9E 30 78 C1 F2 A0 7E 45 29 50 EF 36 F6
MD5:	36 82 B6 C0 EB 81 95 9E 4B 44 58 DF BB 65 D4 F7

Public Key Info

Key Algorithm:	RSA
Key Parameters:	05 00
Key Size:	4096
Key SHA1 Fingerprint:	85 56 53 49 BE EA 4A B1 86 49 B5 61 71 A9 95 CF 48 88 36 60
Public Key:	30 82 02 0A 02 82 02 01 00 B6 11 02 8B 1E E3 A1 77 9B 3B DC BF B1 F9 C5 97 DE EF B9 F2 A3 E9 BC 12 89 5E A7 AA 52 AB F8 23 27 29 25 23 07 A1 B4 D2 3D 2E 60 E0 CF D2 09 87 BB CD 48 F0 4D C2 94 28 AD 0F 7F 26 E5 A8 08 FE 96 E8 3C 68 94 53 EE 83 3A 88 2B 50 7F 84 E0 4C CD 92 D3 20 E9 33 BC 52 99 AF 32 B5 29 B3 25 2A 79 95 57 B4 BD 26 EF D6 01 D1 EB 16 0A BB 8E 0B B5 C5 C5 8A 55 89 81 FA 4C A5 FF D5 8E C3 CE 4B E0 B5 D8 B3 8E 45 CF 76 C0 ED DE 5D D6 16 93 BD 29 68 33 EF 3A 66 EC 07 8A 26 DF 13 D7 57 65 B9 CB 4B ED 39 4B 9C C4 3F D2 55 13 6E 24 B0 D6 71 FA F4 C1 BA

Key Usage

Usages:	Digital signature Revocation list signature
Critical:	Yes

Key points from the third certificate:

1. **Identity and Issuer**: The new certificate you provided is "GTS Root R1" issued by "GlobalSign Root CA". This is different from the previous one that was "GTS CA 1O1" issued by "GlobalSign".
2. **Expires**: This new certificate expires on January 28, 2028, as opposed to the previous one which expired on December 15, 2021.
3. **Public Key Info**: The new certificate has a 4096 bit RSA key, which is stronger than the 2048 bit RSA key used in the previous certificate.
4. **Key Usage**: The new certificate is used for "Digital signature" and "Revocation list signature", while the previous one was used for "Digital signature", "Key encipherment", "Client authentication", "Server authentication".
5. **Basic Constraints**: The new certificate is a Certificate Authority (CA) with unlimited path length, while the previous one was not a CA and had a path length of zero.
6. **Extensions**: The new certificate has a few more extensions than the previous one, like the Authority Information Access (1.3.6.1.5.5.7.1.1) which provides information about how to get the CA's certificate.

In the provided certificates, we observe a chain of digital trust, starting from the root authority GlobalSign Root CA, through Google's GTS Root R1, and down to the more specific GTS CA 1C3.

These certificates each hold key identity information and public keys essential for secure communication. They serve as electronic "passports", providing credibility to the entities they represent.

The root certificate, with its longer validity and stronger encryption, anchors this trust chain. As we go down the chain, the certificates have shorter validity periods and slightly weaker encryption, reflecting their closer proximity to end-users and higher replacement frequency.

Each certificate's key usage also varies, indicative of their roles in the digital trust architecture. Together, they facilitate a web of trust crucial for secure internet communication.

Step 2: Extract the public key (e, n) from the issuer’s certificate

The issuer's certificate is the one that is used to sign (i.e., authenticate) another certificate. In the chain of certificates we found, here's how it breaks down:

1. **GlobalSign Root CA**: This is the root certificate and it's the topmost authority in this chain. It can be considered an issuer's certificate because it signs and validates the next certificate in the chain, which is GTS Root R1 in this case.
2. **GTS Root R1**: This certificate is signed by the GlobalSign Root CA, making GlobalSign the issuer for this certificate. However, GTS Root R1 is also an issuer's certificate itself because it signs and validates the next certificate in the chain, which is GTS CA 1C3.
3. **GTS CA 1C3**: This is a leaf certificate (or end-entity certificate), and it's signed by GTS Root R1, making GTS Root R1 the issuer for this certificate. As a leaf certificate, it is typically not used to sign other certificates, so it wouldn't be considered an issuer's certificate.

In summary, both GlobalSign Root CA and GTS Root R1 can be considered issuer's certificates because they are used to sign and validate other certificates in this chain.

For the purpose of this task we will use the **GTS Root R1** which is c1.pem.

- Extracting modulus (n):

```
[05/14/23]seed@Attacker:~/cert$ openssl x509 -in c1.pem -noout -modulus
Modulus=F588DFE7628C1E37F83742907F6C87D0FB658225FDE8CB6BA4FF6DE95A23E299F61CE992
0399137C090A8AFA42D65E5624AA7A33841FD1E969BBB974EC574C66689377375553FE39104DB734
BB5F2577373B1794EA3CE59DD5BCC3B443EB2EA747EFB0441163D8B44185DD413048931BBFB7F6E0
450221E0964217CFD92B6556340726040DA8FD7DCA2EEFEA487C374D3F009F83DFEF75842E79575C
FC576E1A96FFFC8C9AA699BE25D97F962C06F7112A028080EB63183C504987E58ACA5F192B599681
00A0FB51DBCA770B0BC9964FEF7049C75C6D20FD99B4B4E2CA2E77FD2DDC0BB66B130C8C192B1796
98B9F08BF6A027BBB6E38D518FBDAEC79BB1899D
```

This command uses the openssl command-line tool, which is a powerful utility for managing and manipulating SSL certificates, generating keys, managing private keys, and more.

Here's a breakdown of what each part of the command does:

1. **openssl**: This is the main command for the OpenSSL cryptographic toolkit, which provides support for various secure communications protocols, including TLS and SSL.

2. **x509**: This is a sub-command of OpenSSL and is used for displaying and managing X.509 certificates, which are the type of certificates used in SSL/TLS and other secure communications protocols. X.509 is a standard defining the format of public key certificates.

3. **-in c1.pem**: This part of the command specifies the input file that the command should operate on. c1.pem is assumed to be a PEM-encoded X.509 certificate.

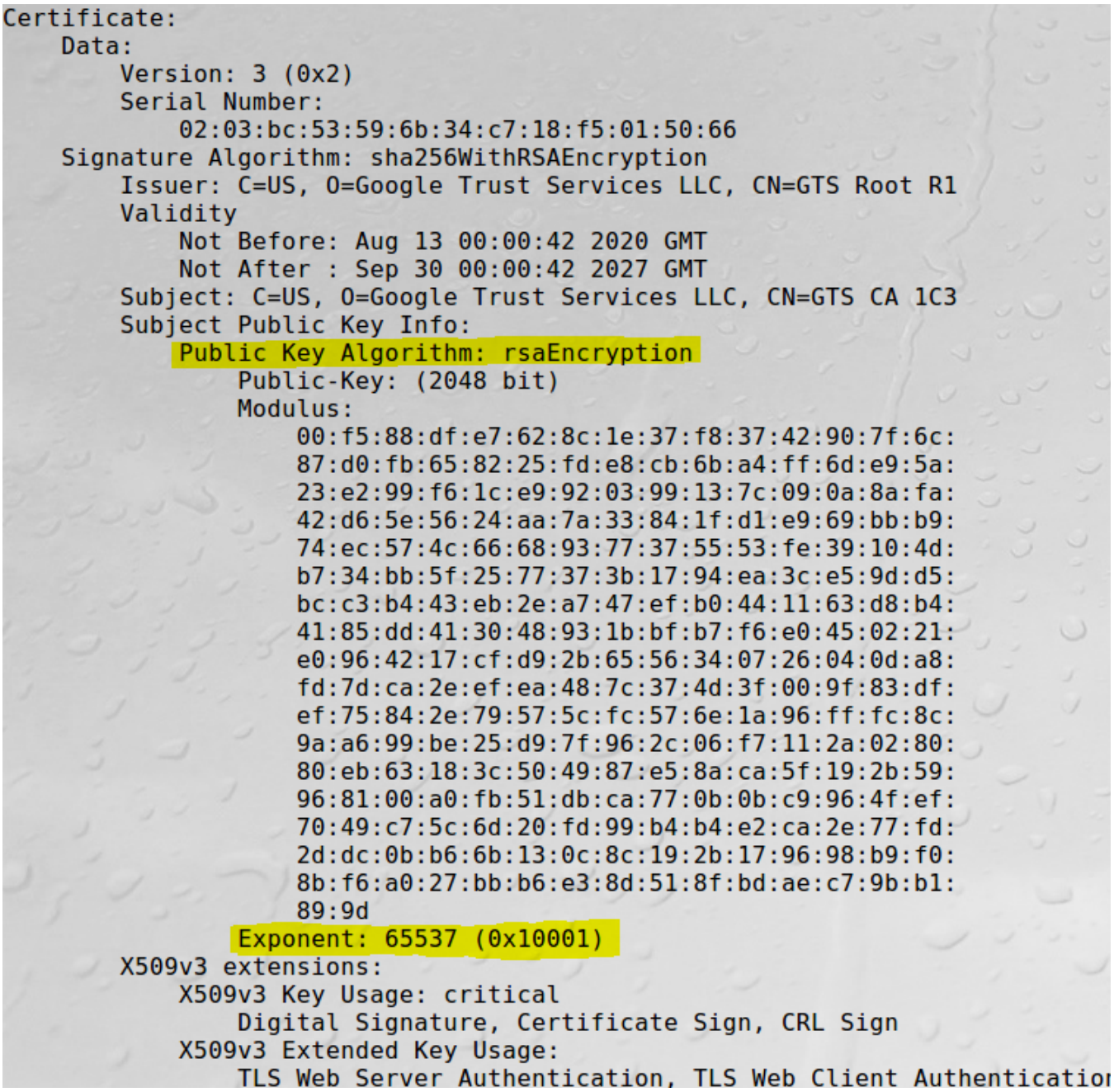
4. **-noout**: This option prevents output of the encoded version of the request. In other words, it suppresses the printing of the certificate in text form.

5. **-modulus**: This option prints the RSA public key modulus of the certificate. The modulus is a part of the public key in an RSA key pair. In the context of SSL/TLS, the modulus is used as part of the encryption and decryption process, as well as the creation and verification of digital signatures. It is a large integer value and is part of the public key.

The extracted modulus:
F588DFE7628C1E37F83742907F6C87D0FB658225FDE8CB6BA4FF6
DE95A23E299F61CE9920399137C090A8AFA42D65E5624AA7A33841
FD1E969BBB974EC574C66689377375553FE39104DB734BB5F25773
73B1794EA3CE59DD5BCC3B443EB2EA747EFB0441163D8B44185DD
413048931BBFB7F6E0450221E0964217CFD92B6556340726040DA8F
D7DCA2EEFEA487C374D3F009F83DFEF75842E79575CFC576E1A96
FFFC8C9AA699BE25D97F962C06F7112A028080EB63183C504987E5
8ACA5F192B59968100A0FB51DBCA770B0BC9964FEF7049C75C6D2
0FD99B4B4E2CA2E77FD2DDC0BB66B130C8C192B179698B9F08BF6
A027BBB6E38D518FBDAEC79BB1899D

- Finding the exponent (e):

```
seed@Attacker:~/cert$ openssl x509 -in c1.pem -text -noout
```



```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            02:03:bc:53:59:6b:34:c7:18:f5:01:50:66
    Signature Algorithm: sha256WithRSAEncryption
        Issuer: C=US, O=Google Trust Services LLC, CN=GTS Root R1
        Validity
            Not Before: Aug 13 00:00:42 2020 GMT
            Not After : Sep 30 00:00:42 2027 GMT
        Subject: C=US, O=Google Trust Services LLC, CN=GTS CA 1C3
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (2048 bit)
                Modulus:
                    00:f5:88:df:e7:62:8c:1e:37:f8:37:42:90:7f:6c:
                    87:d0:fb:65:82:25:fd:e8:cb:6b:a4:ff:6d:e9:5a:
                    23:e2:99:f6:1c:e9:92:03:99:13:7c:09:0a:8a:fa:
                    42:d6:5e:56:24:aa:7a:33:84:1f:d1:e9:69:bb:b9:
                    74:ec:57:4c:66:68:93:77:37:55:53:fe:39:10:4d:
                    b7:34:bb:5f:25:77:37:3b:17:94:ea:3c:e5:9d:d5:
                    bc:c3:b4:43:eb:2e:a7:47:ef:b0:44:11:63:d8:b4:
                    41:85:dd:41:30:48:93:1b:bf:b7:f6:e0:45:02:21:
                    e0:96:42:17:cf:d9:2b:65:56:34:07:26:04:0d:a8:
                    fd:7d:ca:2e:ef:ea:48:7c:37:4d:3f:00:9f:83:df:
                    ef:75:84:2e:79:57:5c:fc:57:6e:1a:96:ff:fc:8c:
                    9a:a6:99:be:25:d9:7f:96:2c:06:f7:11:2a:02:80:
                    80:eb:63:18:3c:50:49:87:e5:8a:ca:5f:19:2b:59:
                    96:81:00:a0:fb:51:db:ca:77:0b:0b:c9:96:4f:ef:
                    70:49:c7:5c:6d:20:fd:99:b4:b4:e2:ca:2e:77:fd:
                    2d:dc:0b:b6:6b:13:0c:8c:19:2b:17:96:98:b9:f0:
                    8b:f6:a0:27:bb:b6:e3:8d:51:8f:bd:ae:c7:9b:b1:
                    89:9d
                Exponent: 65537 (0x10001)
        X509v3 extensions:
            X509v3 Key Usage: critical
                Digital Signature, Certificate Sign, CRL Sign
            X509v3 Extended Key Usage:
                TLS Web Server Authentication, TLS Web Client Authenticatio
```

The openssl x509 -in c1.pem -text -noout command is used to view the details of a certificate file.

-text: This option tells OpenSSL to output the full details of the certificate in human-readable text form. It includes information like the version, serial number, signature algorithm, issuer name, validity period, subject name, subject public key info, and any extensions

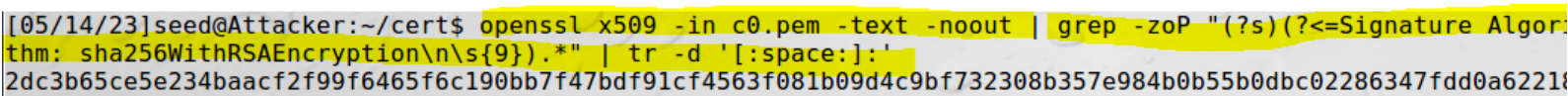
The exponent:
65537 (0x10001)

Step 3: Extract the signature from the server's certificate

Server certificates are typically issued to a specific domain or server, and the CN usually reflects this. They are used in the SSL/TLS handshake process to authenticate the server to the client and to establish the secure connection.

In our case, it is certificate c0 - **GTS CA 1C3**.

- Running this command to print the certificate data.



1. **openssl x509 -in c0.pem -text -noout**: This command reads the certificate file c0.pem and displays its textual representation with the -text option. The -noout option ensures that no additional information is outputted besides the certificate text.
2. **grep -zoP "(?s)(?<=Signature Algorithm**: **sha256WithRSAEncryption\n\s{9}).*"**: This command pipes the output of the previous command to grep for further processing. The options used with grep are as follows:
    - **-z** enables the null data format, allowing the matching to span across multiple lines.

    - **-o** only prints the matched content.
    - **-P** enables Perl-compatible regular expressions for more advanced pattern matching.
3. **The regular expression pattern (?s)(?<=Signature Algorithm: sha256WithRSAEncryption\n\s{9}).*** is used to match and extract the content following the "Signature Algorithm: sha256WithRSAEncryption" line, accounting for nine whitespace characters.
4. **tr -d '[:space:]:':** This command pipes the output of the previous command and uses tr (translate/delete) to remove any spaces and colons from the extracted signature. The '[:space:]:' character class specifies the characters to be deleted, including spaces and colons.

The signature:
2dc3b65ce5e234baacf2f99f6465f6c190bb7f47bdf91cf4563f081b09d4c9
bf732308b357e984b0b55b0dbc02286347fdd0a6221859634718b0348d1
95a532b63f1396ded4dc9cc294fdb704e4744d8d60541186243b03813e8
fb9e7a3b36e567ab1d810fa1610142e653b313590aef6ba5df0f41c6bb7c
af13d35bda7b1ec1eaef65fb26118ba3dd65fc8a9e690f291b992483ed71
5c336992ffba5174a0762ae0aaae2f02aeebc16326bcdd8f4eb17d9deff4b
1317c454f9a9aee6dea1793b779600aa9ad9379546aa61b5df8a37c8107
37e8c9767969f71194205cdc7f92b280ccb47442253c567261ff68b4337b
0351b8f920dea8439f74be0c1515f40e

Step 4: Extract the body of the server's certificate

In this task, we are focusing on extracting the body of a server's certificate. The signature of a server certificate is generated by a Certificate Authority (CA) using a two-step process:

1. computing the hash of the certificate
2. signing the hash.

To verify the signature, we also need to generate the hash from the certificate. However, since the hash is generated before the signature is computed, we must exclude the signature block from the certificate when computing the hash.

The challenge lies in determining which part of the certificate is used to generate the hash, as it requires a good understanding of the certificate's format. X.509 certificates are encoded using the ASN.1 (Abstract Syntax Notation.One) standard.
To easily extract any field from a certificate, we can parse the ASN.1 structure. OpenSSL provides a command called asn1parse, which enables us to parse an X.509 certificate and extract the desired fields. By utilising this command, we can effectively navigate the certificate's structure and extract the necessary information.

- Getting the body of the certificate and calculating it's hash:

```
acker:~/cert$ openssl asn1parse -i -in c0.pem -strparse 4 -out c0_body.bin -noout
[05/14/23]seed@Attacker:~/cert$ sha256sum c0_body.bin
30444d83ebcc1e1b932009d6610589e38b49bbcc0c83220e2f379469667cb61f  c0_body.bin
```

1. **openssl asn1parse -i -in c0.pem -strparse 4 -out c0_body.bin** -noout

- **openssl asn1parse** is a command-line tool that is part of OpenSSL. It is used to parse and display ASN.1 (Abstract Syntax Notation One) data. ASN.1 is a standard and notation that describes rules and structures for representing, encoding, transmitting, and decoding data in telecommunications and computer networking.
- **-i stands for 'indent'**. It causes the output to be indented to reflect the ASN.1 structure.
- **-in c0.pem** specifies the input file, which is the certificate 'c0.pem' in this case.
- **-strparse 4** tells OpenSSL to start parsing at offset 4 of the ASN.1 structure. This can be used to skip certain parts of the data. In this case, it's used to skip the headers of the certificate and get to the body of the certificate (this is typically where the actual certificate data starts in the ASN.1 structure).
- **-out c0_body.bin** specifies the output file for the parsed data. Here, the body of the certificate is being written to a binary file named 'c0_body.bin'.
- **-noout** prevents the parsed data from being printed to the terminal. Without this, asn1parse would print a text representation of the parsed data.

2. **sha256sum c0_body.bin**
    - **sha256sum** is a command-line utility that calculates and checks SHA256 (a cryptographic hash function) message digests.
    - **c0_body.bin is** the input file for the sha256sum command, which is the binary file output from the previous openssl asn1parse command.

- The output 30444d83ebcc1e1b932009d6610589e38b49bbcc0c83220e2f379469667cb61f c0_body.bin is the SHA256 digest of the 'c0_body.bin' file. The SHA256 hash function generates a 256-bit (32-byte) hash, which is typically rendered as a 64-character hexadecimal number. This digest is unique to the data in 'c0_body.bin'; even a small change in the data would result in a completely different digest. This can be used to verify the integrity of data.

The hash that we got:
30444d83ebcc1e1b932009d6610589e38b49bbcc0c83220e2f379469667cb61f

Step 5: Verifying a Signature

In this task, the objective is to verify the signature of a server's certificate using a custom program.
To complete the task, we will use a custom program. This program incorporates the CA's public key, the CA's signature, and the server's certificate body. The program performs the necessary steps to verify the signature's validity, ensuring its authenticity:

1. **Signature decryption**: Decrypt the CA's signature using the CA's public key. In the context of RSA, this decryption is done by calculating S^e mod n, where S is the CA's signature, and e and n

are parts of the CA's public key. The result of this decryption operation is a hash value.

2. **Signature validation**: Compare the hash value obtained from the signature decryption with the expected hash value. If these two values match, then the signature is valid, and the server's certificate can be trusted. If the values do not match, then the signature is not valid, indicating that the server's certificate may have been tampered with or may not originate from the claimed CA.

- The code we will use in the task:

```
#include <stdio.h>
#include <openssl/bn.h>

// Function to print Big Numbers
void printBN(char *msg, BIGNUM * a)
{
  char * number_str = BN_bn2hex(a);
  printf("%s %s\n", msg, number_str);
  OPENSSL_free(number_str);
}

int main()
{
  // Create a BN context
  BN_CTX *ctx = BN_CTX_new();

  // Create BIGNUM objects for the public exponent (e), the modulus (n),
the signature (S) and the message hash (M)
  BIGNUM *e = BN_new();
  BIGNUM *n = BN_new();
  BIGNUM *S = BN_new();
  BIGNUM *M = BN_new();

  // Initialize the values of e, n, and S from the given certificate
  BN_hex2bn(&e, "010001");   // Usually, e is set to 65537 (in hex,
010001)
  BN_hex2bn(&n,
"F588DFE7628C1E37F83742907F6C87D0FB658225FDE8CB6BA4FF6DE95A23E299F61CE992
  BN_hex2bn(&S,
"2dc3b65ce5e234baacf2f99f6465f6c190bb7f47bdf91cf4563f081b09d4c9bf732308b3
```

```
  // Compute the hash of the certificate body
  BN_hex2bn(&M,
"30444d83ebcc1e1b932009d6610589e38b49bbcc0c83220e2f379469667cb61f");

  // To verify the signature, we need to calculate S^e mod n
  BIGNUM *decrypted_S = BN_new();
  BN_mod_exp(decrypted_S, S, e, n, ctx);

  // Truncate hash value to 256 bits
  BN_mask_bits(decrypted_S, 256);

  // Print the decrypted signature and the hash of the message
  printBN("Decrypted signature =", decrypted_S);
  printBN("Hash of the message =", M);

  // Check if the decrypted signature matches the hash of the message
  if(BN_cmp(decrypted_S, M) == 0)
    printf("Signature Valid\n");
  else
    printf("Signature Invalid\n");

  return 0;
}
```

The RSA digital signature process involves two main steps:

1. **Signature generation**: The entity who signs the document (usually the certificate issuer) generates a hash of the document (in this case, the certificate body), then encrypts this hash using their private key. This encrypted hash is the signature.
2. **Signature verification**: The entity who verifies the signature (you, in this case) decrypts the signature using the issuer's public key, which should give the original hash of the document.

When the issuer creates the digital signature, they take a hash of the certificate body and then raise it to the power of d (their private exponent) modulo n (the modulus of their public key). The result is the signature S.

Mathematically, this is represented as:
S = Hash(cert_body)^d mod n

When we verify the signature, we take the signature S and raise it to the power of e (the public exponent of the issuer) modulo n (the same modulus as before).

This is represented as:
decrypted_S = S^e mod n

Given the properties of RSA, this operation should return the original hash of the certificate body. This is because (m^d)^e mod n is equal to m mod n, where m is the original message (or hash of the certificate body in this case).

If decrypted_S equals the actual hash of the certificate body, then the signature is verified. This is because only the holder of the private key could have generated a signature that can be decrypted into the correct hash with the public key. If the decrypted signature does not match the hash of the certificate body, then the signature is not valid, and the certificate may have been tampered with.

- running the code:

```
[05/16/23]seed@Attacker:~/tasks$ sudo ./task6
Decrypted signature = 30444D83EBCC1E1B932009D6610589E38B49BBCC0C83220E2F379469667C
B61F
Hash of the message = 30444D83EBCC1E1B932009D6610589E38B49BBCC0C83220E2F379469667C
B61F
Signature Valid
```

Our program successfully decrypted the provided signature using the public key of the Certificate Authority (CA). After performing the operation **S^e mod n**, we obtained a value that matched the hash of the server's certificate body that we had computed in Task 4. This successful match validates the authenticity of the certificate and demonstrates that it was indeed signed by the trusted CA.

Key Points and Learnings:

1. **Understanding RSA Signature Verification**: We learned the process of how RSA signatures are verified, which involved the use of a public key to decrypt the signature and compare it with the hash of the original message (or in this case, the server certificate body).
2. **Application of Cryptographic Libraries**: We explored how to use OpenSSL's BIGNUM library to handle the large numbers involved in RSA computations.
3. **Importance of Certificate Verification**: This task highlighted the importance of verifying server certificates in real-world cybersecurity. It's a critical step to ensure that the certificate indeed belongs to the claimed server and was not tampered with or replaced by an attacker.
4. **Practical Implementation**: Creating a custom program to verify a certificate offered hands-on experience in applying cryptographic concepts and reinforced the understanding of how secure communication protocols work under the hood.

In the real world, the task of validating digital signatures involves several more considerations, complexities and error checks than demonstrated in our simplified task. Here are some key differences:

1. **Dynamic Handling:** The program we wrote is statically configured for one specific certificate, whereas in real-world scenarios, the code should be capable of handling any certificate dynamically. It should be able to parse the certificate, extract the necessary information such as the public key, the signature, and the certificate body, and perform the verification process.
2. **Certificate Chain Validation:** In reality, certificates are often issued in a hierarchical manner, with a chain of trust leading back to a trusted root certificate authority. Our program only checks one level of this chain. A real-world validation program would need to check each certificate in the chain up to the root certificate.
3. **Revocation Checks:** Certificates can be revoked before they expire if they are compromised. A real-world program would also need to check the certificate's status against a Certificate Revocation List (CRL) or use the Online Certificate Status Protocol (OCSP) to ensure the certificate is still valid.
4. **Error Handling:** Our program does not handle errors or exceptions that could occur during the process. A robust program should handle errors gracefully and provide meaningful feedback or perform remediation steps as necessary.
5. **Time Validity Checks:** Certificates have a valid-from and valid-to timestamp. A real-world program would check whether the current time falls within this validity period.

6. **Certificate Policies and Extensions:** There are numerous optional fields in a certificate like Key Usage, Extended Key Usage, Certificate Policies etc., which impose certain constraints or provide additional information about a certificate. A real-world program would need to parse and validate these fields as per the application's requirement.
7. **Algorithmic Support:** Our program supports only one specific hash function and RSA for signature. A real-world program should ideally support multiple algorithms as certificates can be signed using different algorithms.
8. **Secure Coding Practices:** The program must follow secure coding practices to prevent potential vulnerabilities. For example, it should not leak sensitive information in error messages, and it should protect against buffer overflows or other potential security vulnerabilities.

Summary

In this laboratory exercise, we delved into the practical applications of the RSA Public-Key Cryptosystem, a fundamental encryption and decryption method in modern secure communication.

1. **Deriving the Private Key:** We started the lab by understanding the process of key generation in RSA. We chose two prime numbers, computed their product, and calculated the totient. We then selected a suitable public exponent and derived the corresponding private key. This exercise helped us appreciate the role of prime numbers and the computational difficulty of factoring in RSA's security.

2. **Encrypting a Message:** We used the public key to encrypt a plaintext message. The transformation involved raising the plaintext to the power of the public exponent, modulo the modulus. This demonstrated the fundamental operation of RSA encryption and its reliance on modular arithmetic.
3. **Decrypting a Message:** We then decrypted the ciphertext message using the private key. This involved a similar modular exponentiation operation, reinforcing the symmetry of encryption and decryption in RSA, albeit with different keys.
4. **Signing a Message:** We explored RSA's application in digital signatures. Here, we used the private key to 'encrypt' a message, or rather its hash, effectively creating a signature that can be used to verify the message's authenticity and integrity.
5. **Verifying a Signature:** We verified the signature by 'decrypting' it with the public key and comparing it to the hash of the message. This showed us how public keys can validate signatures without revealing the private key.
6. **Manually Verifying an X.509 Certificate:** Finally, we manually verified an X.509 certificate. This involved extracting the certificate's body, calculating its hash, and verifying it against the provided signature. We understood the structure of X.509 certificates and their role in establishing trust in secure communication.

Throughout this lab, we used OpenSSL's command-line tools and the OpenSSL library in a C program, gaining hands-on experience with these industry-standard tools. By manually performing operations typically handled by libraries or protocols, we deepened our understanding of RSA and public-key cryptography. We learned about

the mathematical underpinnings of RSA, the importance of key management, and RSA's dual functionality in encryption/decryption and signing/verification.

The Common Modulus Attack

This type of attack, while rare in real-world scenarios, is theoretically possible and useful for understanding RSA's potential vulnerabilities.

In the given scenario, a user (Bob) communicates with another user (Alice) using Alice's RSA public key (n, e_1). However, due to a computational error, a bit is flipped, and the message is encrypted with a faulty public key (n, e_2). Therefore, two different ciphertexts of the same message M are created, each encrypted with a different exponent but sharing a common modulus.

Under these conditions, an eavesdropper (Eve) can potentially recover the plaintext if $\gcd(e_1, e_2) = 1$ and $\gcd(ct_2, n) = 1$. The greatest common divisor (gcd) function here checks if the two numbers are co-prime, i.e., their gcd is 1, which means they share no other common factor than 1.

The attack exploits certain mathematical properties. First, it recalls that RSA encryption is performed using modular exponentiation. Secondly, it uses Bezout's Theorem, which states that for any non-zero integers a and b, there exist integers x and y such that $ax + by = \gcd(a, b)$. It also considers the property of multiplicative inverse in modular arithmetic, which requires the denominator and the modulus to be co-prime for the inverse to exist.

If gcd(e_1, e_2)=1, the integers x and y can be found using the Extended Euclidean algorithm. The plaintext can then be recovered by performing specific mathematical operations in the common modulus. However, this requires y to be a negative integer and ct_2 (the second ciphertext) to be invertible in mod n, necessitating gcd(ct_2, n) = 1.

In summary, this scenario illustrates a potential vulnerability in RSA, although it relies on very specific circumstances. It underscores the importance of maintaining the integrity of encryption keys and highlights the potential dangers of computational errors in cryptographic systems.

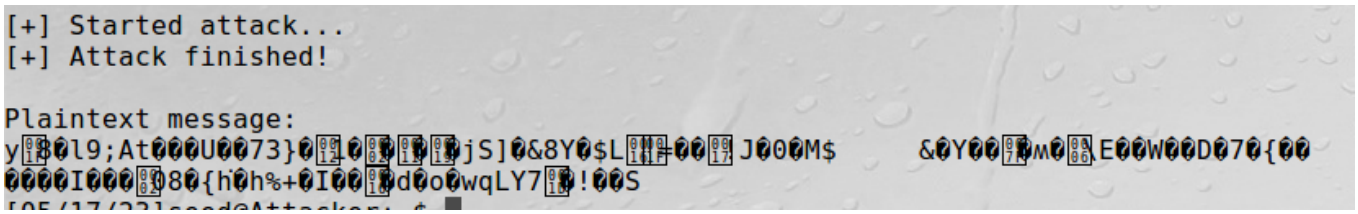
Code for this attack:

```
import argparse

from fractions import gcd

parser = argparse.ArgumentParser(description='RSA Common modulus attack')
required_named = parser.add_argument_group('required named arguments')
required_named.add_argument('-n', '--modulus', help='Common modulus',
type=long, required=True)
required_named.add_argument('-e1', '--e1', help='First exponent',
type=long, required=True)
required_named.add_argument('-e2', '--e2', help='Second exponent',
type=long, required=True)
required_named.add_argument('-ct1', '--ct1', help='First ciphertext',
type=long, required=True)
required_named.add_argument('-ct2', '--ct2', help='Second ciphertext',
type=long, required=True)

def egcd(a, b):
    if a == 0:
        return (b, 0, 1)
    else:
        g, y, x = egcd(b % a, a)
        return (g, x - (b // a) * y, y)

def modinv(a, m):
    g, x, y = egcd(a, m)
    if g != 1:
        raise ValueError('Modular inverse does not exist.')
    else:
        return x % m
```

```
def modinv(a, m):
    g, x, y = egcd(a, m)
    if g != 1:
        raise ValueError('Modular inverse does not exist.')
    else:
        return x % m

def attack(c1, c2, e1, e2, N):
    if gcd(e1, e2) != 1:
        raise ValueError("Exponents e1 and e2 must be coprime")
    s1 = modinv(e1,e2)
    s2 = (gcd(e1,e2) - e1 * s1) / e2
    temp = modinv(c2, N)
    m1 = pow(c1,s1,N)
    m2 = pow(temp,-s2,N)
    return (m1 * m2) % N

def main():
    args = parser.parse_args()
    print '[+] Started attack...'
    try:
        message = attack(args.ct1, args.ct2, args.e1, args.e2,
args.modulus)
        print '[+] Attack finished!'
        print '\nPlaintext message:\n%s' % format(message, 'x').decode
('hex')
    except Exception as e:
        print '[+] Attack failed!'
        print e.message

main()
```

breakdown of the code:

1. **Import Libraries:** The **argparse** module is used for writing user-friendly command-line interfaces. The fractions module provides support for rational number arithmetic.
2. **Argument Parsing:** The argparse.ArgumentParser function creates a new ArgumentParser object. Named arguments are then added to the parser through the add_argument function. These arguments will be the inputs for the RSA Common Modulus attack (modulus, exponents, and ciphertexts).
3. **Extended Euclidean Algorithm Function (egcd):** This function calculates the greatest common divisor (gcd) of two numbers, as well as the coefficients of Bézout's identity (which are used for finding the multiplicative inverse of a number modulo another number).
4. **Modular Inverse Function (modinv):** This function calculates the modular multiplicative inverse of a number modulo another number using the Extended Euclidean Algorithm.
5. **RSA Common Modulus Attack (attack):** This function executes the RSA Common Modulus attack. It first checks if the exponents are coprime (i.e., their gcd is 1). Then, it calculates the values of s1 and s2 using the Extended Euclidean Algorithm. After that, it calculates the original plaintext from the ciphertexts, the exponents, and the common modulus using the RSA Common Modulus attack equation.
6. **Main Function:** This function parses the arguments passed in the command line, executes the RSA Common Modulus attack using the parsed arguments, and prints the result.



We had issues in getting human readable output which we were unable to resolve yet.

Quantum Computer And 2048-Bit RSA Encryption

RSA encryption, a widely used encryption standard, is based on "trapdoor" mathematical functions. This method involves the use of large prime numbers and their difficulty to factorise, which in turn provides security. While it's easy to multiply large prime numbers, reverse-engineering that operation (i.e., factorization) is computationally intense, making it practically unbreakable by classical computers.

However, quantum computers have the potential to break this encryption system efficiently. Quantum computers operate using qubits, which unlike classical bits, can be in a state of superposition (i.e., both 0 and 1 simultaneously), enabling them to process information exponentially faster than classical computers.

The article mentions **Shor's algorithm**, a quantum algorithm that can factorise large numbers, and thus break RSA encryption, far more efficiently than any known classical algorithm. This algorithm would thus theoretically allow a sufficiently powerful quantum computer to break RSA encryption.

Previously, the expectation was that quantum computers powerful enough to break RSA encryption were decades away. This was because the resources and number of qubits required for such a task were

thought to be massive. In 2015, it was estimated that a quantum computer would need a billion qubits to factorise 2048-bit numbers reliably, far more than the capabilities of any existing quantum computers at that time.

However, researchers Craig Gidney and Martin Ekera have proposed an **optimization to Shor's algorithm**, which reduces the resources required to break RSA encryption significantly. Their method focuses on a **more efficient way to perform modular exponentiation**, the most resource-demanding operation in Shor's algorithm.

Gidney and Ekera's optimised method would require a quantum computer with **20 million** qubits to break 2048-bit RSA encryption, a dramatic reduction from the previously estimated **one billion qubits**. They estimate that such a quantum computer could complete the task in just **eight hours**.

This development highlights the need for post-quantum cryptography, which is designed to be secure even against quantum computers. Despite this, such cryptographic methods are not yet standard, leaving many current data transmissions potentially vulnerable to future quantum attacks.

The potential of quantum computers breaking RSA encryption in the foreseeable future presents an urgent concern for organisations such as governments, militaries, and banks, which need to store data securely for long periods.