DAT510 - Assignment 2

Frøydis Jørgensen

October 30, 2020

Abstract

In this assignment I have learned and implemented a RSA Digital Signatur system. I have used some packages, but explain step by step how RSA does work. I have used my RSA implementation in a staged scenario and by using two application where they send messages to each other with a digital signature where the other part have to verify the signature. I have also tested different bit lengths of the primes, finding out how it effects the execution time. Lastly I will discuss and answer some questions about SSL and Certificate Authority.

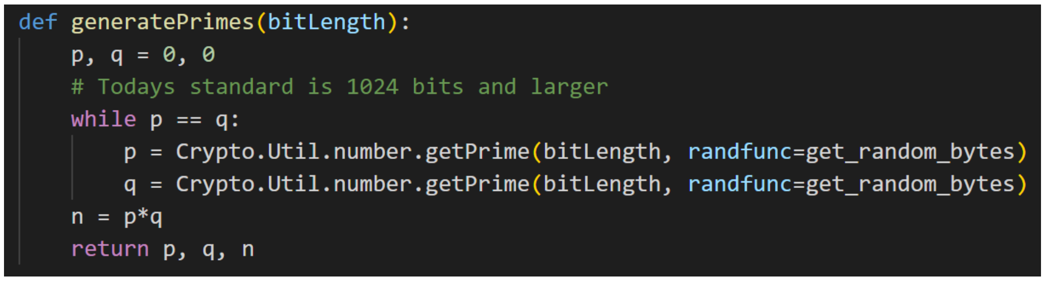
Introduction

Knowing that the message you got is the exact message the same message that was sent is important. We need to verify that the message is not tampered. We can verify that by using a Digital Signature. In this assignment I will implement a way to sign and validate data when it's being transferred by each side. I have chosen to use RSA and the hashing algorithm SHA256. I will also go through different SSL certificates and discuss Certificate Authority.

Design and Implementation

I have created both a digital signature scenario in the main with a predefined message where we can validate the signature on the message and two applications where Alice and Bob can send messages to each other with a signature and validate that the messages are correct. Both the main and the applications use the same methods and I will therefore first explain all the methods before I go into the applications.

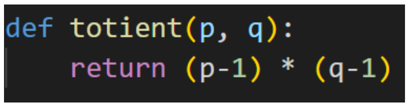
When implementing RSA, I think the most crucial thing is to find some good keys. We need two random large primes, **p** and **q**. By multiplying **p** and **q** we get **n**. The keys are generated as shown in Figure 1. To decide **p** and **q** I use a package called Crypto and uses their method *getPrime*, and to get a random primes I also uses Crypto's function *get_random_bytes*. Although the chances for p and q to be the same is very small, we should make sure that's not the case. I take in the bit length as an input to the function. There are major runtime differences that depend on the bit length, but I will get back to that.



```python
def generatePrimes(bitLength):
    p, q = 0, 0
    # Todays standard is 1024 bits and larger
    while p == q:
        p = Crypto.Util.number.getPrime(bitLength, randfunc=get_random_bytes)
        q = Crypto.Util.number.getPrime(bitLength, randfunc=get_random_bytes)
    n = p*q
    return p, q, n
```

Figure 1: Generates two large primes and multiplies them

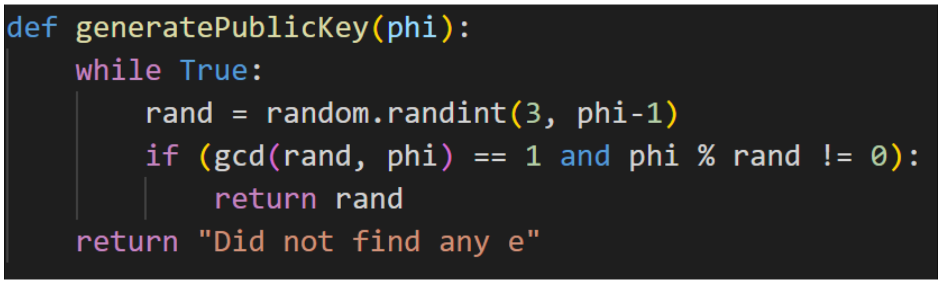
The next step is to find the totient of **n**, ϕ phi. As shown in Figure 2 that's an easy calculation using **p** and **q**.



```python
def totient(p, q):
    return (p-1) * (q-1)
```

Figure 2: Calculated the totient of n from p and q

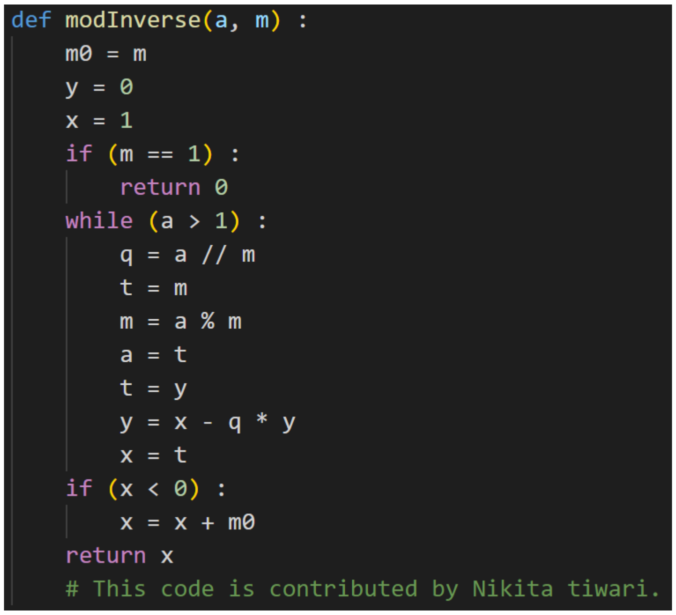
Together with **n** we have another public exponent **e**. This is a number that is the greatest common divisor of 1 with $\phi(n)$. The number in practical use is often chosen to be the Fermat number 65537. That is the largest known prime in the form of $2^{2^n} + 1, (n = 4)$. It is large enough to avoid attacks and quick on binary calculations. But we should always check that e is not a factor of the totient ϕ. If that is the case, we choose the next largest prime number, [1]. But in this assignment, we want to generate a new **e** for each time so that for testing. As shown in Figure 3 I use *randint* to get a random number in between 3 and phi and then check that it meets the requirement. I use *gcd* imported from the package math to calculate the greatest common divisor, and if the result from that is 1. We can use that number as our public key **e**. That means that **e** and ϕ are coprimes.



```python
def generatePublicKey(phi):
    while True:
        rand = random.randint(3, phi-1)
        if (gcd(rand, phi) == 1 and phi % rand != 0):
            return rand
    return "Did not find any e"
```

Figure 3: generate our public component e, by finding a coprime to phi

Using the public exponent **e** and ϕ we can create the private exponent **d**. The private exponent **d** is the inverse of the public exponent with respect to the totient ϕ. To calculate the inverse you can use the *Extended Euclidean Algorithm.* I have taken an implementation of that from geeksforgeeks, ref [2]. This method works when **e** and ϕ are coprimes, which we already know they are. The function to find **d** is shown in Figure 4



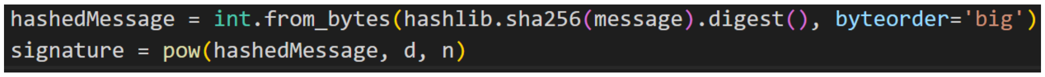
```python
def modInverse(a, m) :
    m0 = m
    y = 0
    x = 1
    if (m == 1) :
        return 0
    while (a > 1) :
        q = a // m
        t = m
        m = a % m
        a = t
        t = y
        y = x - q * y
        x = t
    if (x < 0) :
        x = x + m0
    return x
    # This code is contributed by Nikita tiwari.
```

Figure 4: Finding the inverse. Function from GeeksforGeeks

Now that we have the public keypair of **e** and **n** and the private key pair of **d** and **n**, we can start creating the digital signature on the message. Before we can run the RSA encryption calculation to create the signature we have to hash the original data (the message). I have imported *hashlib* and used SHA256 for hashing the message. So that means that no matter how long the message is, we will get a 256-bit block. Using the hash we can now create the signature by using RSA's encryption function. We will use our private key to create the signature as shown here in the mathematical expression.

$$hash^d \ mod \ n$$

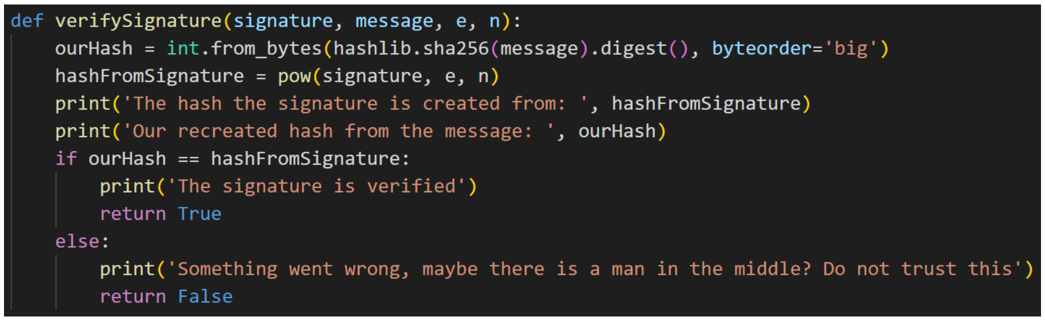
In the code, as shown in Figure 5, I use the function *pow* which does the same as the mathematical expression. As you can also see in the figure is that I convert the hash into an integer, and throughout the program, I only work with integers instead of bits.



```
hashedMessage = int.from_bytes(hashlib.sha256(message).digest(), byteorder='big')
signature = pow(hashedMessage, d, n)
```

Figure 5: Hashing the message and create signature

Now that we have the message and the digital signature we can send it and verify that the message is correct according to the signature and it has not been tampered by some man in the middle. To verify the signature we take the message and creates a hash using the same hashing method SHA256. This hash should match what we decrypt from the signature. To decrypt the signature we do the same mathematical operation as when it was created but instead of the message, we use the signature, and instead of the private key **d** we use the public key **e**. This works because both **e** and **n** is public keys. So in this case everyone can verify that the message was sent from f.ex Alice. Normally we would use some encryption on top of that since the message now is sent in plaintext. As you can see in Figure 6, we check that the hash we got from the message is the same that we got after decrypting the signature. If they are the same, we have verified that the message from Alice is the same as at the point Alice signed it and sent it.

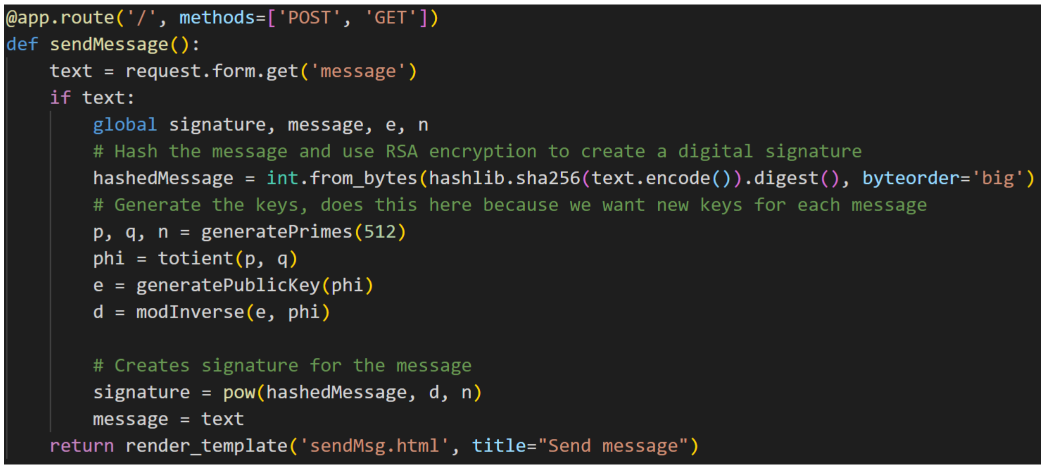


```python
def verifySignature(signature, message, e, n):
    ourHash = int.from_bytes(hashlib.sha256(message).digest(), byteorder='big')
    hashFromSignature = pow(signature, e, n)
    print('The hash the signature is created from: ', hashFromSignature)
    print('Our recreated hash from the message: ', ourHash)
    if ourHash == hashFromSignature:
        print('The signature is verified')
        return True
    else:
        print('Something went wrong, maybe there is a man in the middle? Do not trust this')
        return False
```

Figure 6: Verifying that the signature matches the message

In the application, we have two URLs. The start URL, where we can send a message and sign it. And /getMsg where we can see the message from the other part and see if the signature is correct.

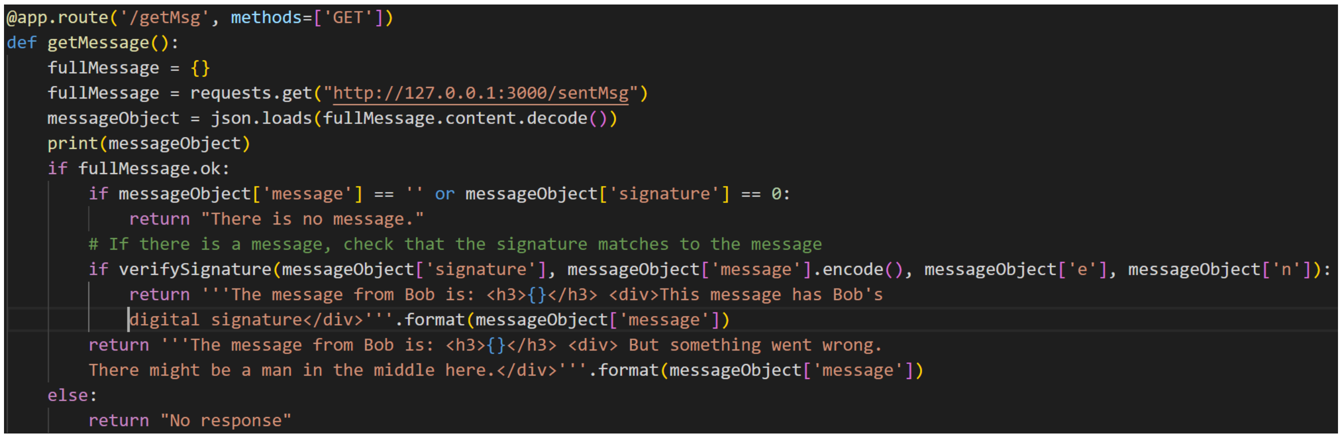
When a message is written and it's clicked send the keys will be created as explained previously and the signature will be created as shown in Figure 7. The fetching point for people that want to receive the message is /sentMsg. And the message is sent as a dictionary including the message, the signature, **e**, and **n**.



```python
@app.route('/', methods=['POST', 'GET'])
def sendMessage():
    text = request.form.get('message')
    if text:
        global signature, message, e, n
        # Hash the message and use RSA encryption to create a digital signature
        hashedMessage = int.from_bytes(hashlib.sha256(text.encode()).digest(), byteorder='big')
        # Generate the keys, does this here because we want new keys for each message
        p, q, n = generatePrimes(512)
        phi = totient(p, q)
        e = generatePublicKey(phi)
        d = modInverse(e, phi)

        # Creates signature for the message
        signature = pow(hashedMessage, d, n)
        message = text
    return render_template('sendMsg.html', title="Send message")
```

Figure 7: Running a template with input field and submit button to send message with a signature

To fetch messages you go to /getMsg. If there is no message it will be displayed "There is no message". If there is a message we have to verify the signature. We use the *verifySignature* method as shown in Figure 6. If the message and the signature correspond the message will be shown with a note that the signature is correct. If the signature is wrong there is a warning that someone might have tampered the message.



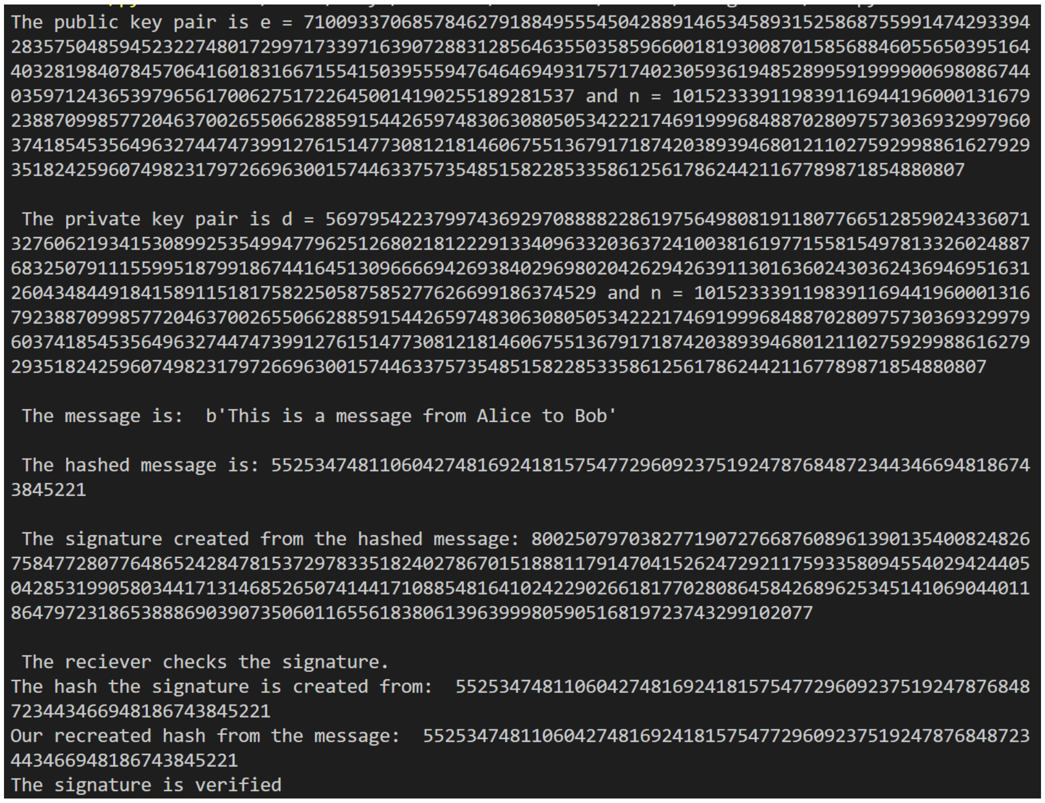
```
@app.route('/getMsg', methods=['GET'])
def getMessage():
    fullMessage = {}
    fullMessage = requests.get("http://127.0.0.1:3000/sentMsg")
    messageObject = json.loads(fullMessage.content.decode())
    print(messageObject)
    if fullMessage.ok:
        if messageObject['message'] == '' or messageObject['signature'] == 0:
            return "There is no message."
        # If there is a message, check that the signature matches to the message
        if verifySignature(messageObject['signature'], messageObject['message'].encode(), messageObject['e'], messageObject['n']):
            return '''The message from Bob is: <h3>{}</h3> <div>This message has Bob's
            digital signature</div>'''.format(messageObject['message'])
        return '''The message from Bob is: <h3>{}</h3> <div> But something went wrong.
        There might be a man in the middle here.</div>'''.format(messageObject['message'])
    else:
        return "No response"
```

Figure 8: Fetching the message and verifying the signature

The applications have a very similar set up as in assignment 2, and one is run on port 3000 and one on port 5000.

Test results

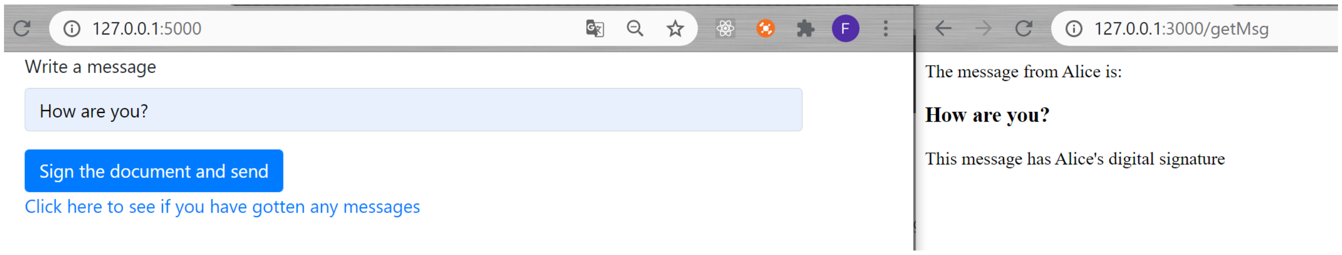
As in the previous assignment I created both a staged scenario and two applications to try out RSA digital signature. Running the main in RSA.py will print out the information we get while doing a digital signature and verifying it, in a staged scenario. As you can see in Figure 9, the first thing printed is the public and private keys. Then we can see the message sent from Alice to Bob and the hashed message. From the hashed message the signature is created. The signature will be sent to Bob from Alice with the message and the public keys. Then Bob recreates a hash from the message to see that it matches the hash we got by encrypting Alice's signature. And in this case, they are the same and therefore the signature is verified.



```
The public key pair is e = 7100933706857846279188495554504288914653458931525868755991474293394
2835750485945232274801729971733971639072883128564635503585966001819300870158568846055650395164
4032819840784570641601831667155415039555947646469493175717402305936194852899591999900698086744
0359712436539796561700627517226450014190255189281537 and n = 101523339119839116944196000131679
2388709985772046370026550662885915442659748306308050534222174691999684887028097573036932997960
3741854535649632744747399127615147730812181460675513679171874203893946801211027592998861627929
3518242596074982317972669630015744633757354851582285335861256178624421167789871854880807

 The private key pair is d = 56979542237997436929708888228619756498081911807766512859024336071
3276062193415308992535499477962512680218122291334096332036372410038161977155815497813326024887
6832507911155995187991867441645130966669426938402969802042629426391130163602430362436946951631
260434844918415891151817582250587585277626699186374529 and n = 1015233391198391169441960001316
7923887099857720463700265506628859154426597483063080505342221746919996848870280975730369329979
6037418545356496327447473991276151477308121814606755136791718742038939468012110275929988616279
293518242596074982317972669630015744633757354851582285335861256178624421167789871854880807

 The message is:  b'This is a message from Alice to Bob'

 The hashed message is: 5525347481106042748169241815754772960923751924787684872344346694818674
3845221

 The signature created from the hashed message: 8002507970382771907276687608961390135400824826
7584772807764865242847815372978335182402786701518881179147041526247292117593358094554029424405
0428531990580344171314685265074144171088548164102422902661817702808645842689625345141069044011
86479723186538886903907350601165561838061396399980590516819723743299102077

 The reciever checks the signature.
The hash the signature is created from:  55253474811060427481692418157547729609237519247876848
723443466948186743845221
Our recreated hash from the message:  55253474811060427481692418157547729609237519247876848723
443466948186743845221
The signature is verified
```

Figure 9: The print statements from the RSA.py main

Using this on the two applications will look as shown in Figure 10, when the signature is verified. If the signature does not match the message, the note under the message will tell you that.



Write a message

How are you?

Sign the document and send

Click here to see if you have gotten any messages

The message from Alice is:

How are you?

This message has Alice's digital signature

Figure 10: Alice sends a message and sign it. Bob receives it and verifies the signature

It seems like RSA usually uses keys of 2048 bits. Creating and working with that big numbers goes slow in my python program and to beware of computational problems I have used the package *timeit* to find out the different execution times depending on the bit length. You can see the result in the table here.

Bit length	Execution time
128	0.060
256	0.186
512	0.738
1024	4.894
2048	35.01

As you can see in the table, the execution time increases exponentially as we increase the bit length. But the important thing here is that if the product of p and q is bigger than the hashed message, which is 256-bits, or else this will not work. So when I tested with 128bit primes to create the key, the digital signature did not work. A way to solve that is to split up the keys or split up the message and create a longer hash, but I chose to not do that because the execution time of keys created from larger primes is not that much bigger.

Discussion

The keys are created and fulfill the requirements to work in RSA. The RSA digital signature does work as expected and I think the result is good. With the keys created I could also have used them for encrypting and decrypting messages. I have chosen to use a default of 512-bit primes as **p** and **q**, because that's small enough to test the application without having to wait for the program to run but large enough to provide some security and works without splitting the keys. I could have, and was thinking about splitting up the message, to get a lower execution time, but did not see that as a must in this case, since the messages were not that big anyway. I tried sending a string with 5000 characters, and it was slower, but nothing crucial. With more time I think it had been fun to try to implement SHA256 from the bottom, but that is for later when I have some extra time.
RSA does work because of Euler's Theorem. And it is hard to crack because it is much easier to multiply large numbers, rather than factor them apart.

I did go through the algorithm and why exactly RSA works. But if you are interested to see what I've read, you can take a look here [3].

Part 2

1. What are the different types of SSL's and how different they are in aspect of security? Why ?
SSL (Security Sockets Layer) does encrypt data that is transferred between users and a website. Because it ensures that malicious third parties can't intercept your personal information it is especially important if the website features facilities as logins, forms that capture personal data or Credit card transactions. There are different types of SSL certificate but all of them have the same level of encryption. The differences is the level of validation. Validation level refers to the extent of checks that a Certificate Authority (CA) does to verify the identify of a person or organization that owns a website. I will get back to the CA in the next question. There are three main types of SSL validation:
Domain Validated certificates (DV SSL)
DV SSL is the lowest level of validation of these three. When you get the certificate, CAs does not look into information about the person or company that is running the website. They just check that they have control over the domain that is want's to be SSL certified. As a user, there is therefore limited information about the website ownership. For the person or organization issuing the certificate this is a quick process which is generally online and automated. It is also the cheapest option.
Organization validated certificates (OV SSL)
OV SSL requires some background checks. The CAs has to verify the individual or business that own the domain and do a minor evaluation. This is more trustworthy that DV SSL because the users can click on the padlock display and get some information about the owners of the domain, such as name, address and country. The users do therefore know who they are giving their information to. It takes often several days to issue an OV SSL cert, since some checks have to be done.
Extended Validated certificates (EV SSL)
EV SSL is the highest level of SSL certification you can get. CAs does extensive background cheks on the owners, validating its ownership, legal existence, physical location, and more. This is very expensive and can often

take several weeks to issue. But with a EV cert the users will have no doubt about the site's trustworthiness and that it is a legitimate business.
Depending on how many domains you need there is different types of SSL certificates. Each SSL cert combines the level of validation with the number of domains. Based on how many domains you need there are four different types of SSL.
Single-domain SSL certificate
With this certificate a single domain and all the pages on that domain are protected. You can choose between all three different validation levels for this certificate.
Wildcard SSL certificate
With this certificate you can protect a single domain and unlimited subdomains for that domain. This certificate can be issued with DV and OS level of validation, but not EV.
Multi-Domain SSL certificate
With this certificate you can protect up to 100 different domains. Wildcard domains can also be protected with a multi-domain SSL certificate. All three validation levels are available for this certificate.
Unified Communications SSL certificate
This certificate are similar to multi-domain certificates and can secure up to 100 domains and subdomains on one certificate. But unified communication certificates are created specifically for environments that utilize Microsoft Exchange and Office Communications. They use the Subject Alternative Name (SAN) extension instead of different IP address to secure these domains.

By combining a validation level appropriate to the purposes, with the number of domains you need you can get an optimal SSL certificate. For answering this question I have used an article from namecheap.com and it's references, read it here [4].

2. Research about the the Certificate Authority Security concerns and explain.
A certificate authory (CA) is a company or organization that acts to validate the identities of entities such as websites, email addresses, companies, or individual persons and bind them to cryptographic keys through the issuance of electronic documents known as digital certificates. A digital certificate verifies the ownership of a public key by the named subject of the certificate. A CA acts as a trusted third part, which both the owner of the certificate

and the party relying upon the certificate must trust. But there are some concerns regarding trusting the CA, and I will go through some of them.
Certificate Revocation
Usually a certificate last for a few years, but in some cases the certificate should be revoked before it expires. The certificate revocation lists (CRL) are lists that contains serial number f revoked certificates. These are published by the CA responsible for issuing the certificate to be revoked. An issue with using CRLs is that in the event that the CRL server becomes unreachable, the client will be forced to either fail open or fail closed. When you choose to fail open the client will be left without protection from revoked certificates. If you choose to fail close the client will be left without the ability to connect to SSLs enabled endpoint.
Breaches
Since CAs hold important private keys and other information that can expose a lot of data, they are valuable targets. Because the trust to the CA are very important there are some cases where the CA will not own up to it and inform users if there is a breach. If the CA is trustworthy they will own up the the mistake and revoke the affected certificates. So a big concern is that the CA must be trustworthy so that you know that the certificate you use is not leaked so that you do actually have a secure connection.
CAs behaving badly
Since we have to trust the CAs we use there is extremely important that they behave as expected. There have been some cases where CAs works around the requirements and therefore does not give us the secure connection we think we have. There was an happening where a CA had several violations documented. But to ban a CA is a though decision and is a decision that browser vendors has to support, because they are the people who decides which CAs that are allowed in their browser. But in the case where many mistakes and violations was documented only Firefox and Apple decided that the CAs certificates would not be valid for a year. That means that in for example Chrome, the CA could go on as before with the bad behavior. There are also differences in what the requirements is to issue a EV SSL cert and OV SSL cert. Some CAs are more strict that others.
Single point of failure
It is not uncommon for a root store to contain several hundred root CAs. The browser treats all of the CAs equally, therefore it only takes one rouge or compromised CA for failure.
Lack of transparency

Another problem is that any CA can issue a certificate for a website without the domain owner to know about it. Browsers are starting to enforce certificate transparency, which means they will publish logs of all certificates issued. This log can be used in two different ways. First, to be sure that a certificate presented by a server is one that was properly issued, and second, that a CA did not issue a certificate for a domain that did not require it.
To answer this question I used an article from Fordham Center for Cybersecurity. You can find it here [5].

3. How does browsers identify secure CA's From another CA's and how is it measured ?
For the browser to identify secure CA's and verify the certificate there is some things that has to be checked. First the browser verifies the certificate's integrity. This is done by verifying the signature on the certificate using a normal public key cryptography. If the signature is valid the browser will move on and verify the certificate's validity. The validity period is the time interval during which the signing CA warrants that it will maintain information about its status. If the validity period is ending before or starting after the date and time of the validation check the browser will reject it.

If the validity period is correct the browser will move on and check the certificate's revocation status. Because the certificate is expected to be valid for the whole validity period, there can be circumstances, as explained previous that does cause the certificate to become invalid before it expires. If the CA has revoked the certificate, the certificate will be put on a Certificate Revocation List (CRL). If the certificate is on that list, the browser will reject it. There is also a possibility to send a request to the Online Certificate Status Protocol (OCSP) and get the revocation status from an online server.

The browser does also have to verify the issuer. X.509 certificates, which is the most common, are normally associated with two properties, the issuer and the subject. Browsers check that a certificate's issuer field is the same as the subject field of the previous certificate in the certificate path. Lastly the browser checks the certificate constrains, if these are defined by the CA. Most of this information is from this article [6].

Conclusion

In this assignment I have shown how to sign a message and how the other part can verify it by using RSA Digital Signature. I have explained how it's implemented in my program and how it works. I have taken a closer look into execution time using different sizes of my primes in RSA. I have also discussed SSL, looked into Certificate Authority and the concerns about it and how browsers checks certificates.

References

[1] 65,537, *Wikipedia - The number 65537*, `https://en.wikipedia.org/wiki/65,537`.

[2] GeeksforGeeks, *Modular inverse*, `https://www.geeksforgeeks.org/multiplicative-inverse-under-modulo-m/`.

[3] Medium - Pedro Matias de Araujo, *Why RSA Cryptography Works*, `https://medium.com/@pemtajo/why-rsa-criptography-works-ea8699f79779`.

[4] NameCheap, *Types of SSL Certificates*, `https://www.namecheap.com/security/ssl-certificate-types/`.

[5] Fordham Cnter of Cybersecurity, *Security Issues with Certificate Authorities*, `stamp.jsp?arnumber=8249081&fbclid=IwAR1y2WusyHwtP_3gjW20zbje_FkF5UCQCOEdiOAn8iA0kXVHessOgjJmIEc`.

[6] Venafi, *How Does a Browser Trust a Certificate*, `https://www.venafi.com/blog/how-does-browser-trust-certificate?fbclid=IwAR0tSCfi090XnS-RD5uQW6U5mHKJF3HuHFkKSk8Xv6r8erQVPzko7XJ7Efo`.