G53SEC Lab 3: Cryptography Part II

Lab Description

In this lab you'll be continuing to explore the features of the Cryptography library in Python. You'll gain experience of hash functions, on both short messages and longer files. We'll then look at the use of public-key cryptography – RSA – that when combined with hash functions provides message integrity and authenticity.

Finally, you'll combine all of the above techniques in order to verify a digital certificate chain for the real nottingham.ac.uk digital certificate.

This lab continues our work from the previous lab, if you've run through the previous lab document, you should know more than enough detail about Python and Cryptography to continue with this work.

Introduction

In the last lab we looked primarily at symmetric encryption. That is, using a single key to encrypt and decrypt messages. You gained experience using a modern cryptographic library in Python, but these techniques are very similar to those you'll see in other languages, and using other libraries. We addressed the issue of where we get a secret key using Diffie-Hellman, a public-key handshake. Finally, we looked a key derivation functions, and AEAD modes of encryption that provide message integrity as standard – much modern cryptography is conducted in this way. These functions often use hash functions as a primitive, we'll look in more detail at these today.

Another problem we haven't addressed is that Diffie-Hellman provides no protection against message tampering, a third party could act as a man-in-the-middle, perform handshakes with both parties, and routinely decrypt all of their traffic. This issue is addressed by RSA, or DSA, algorithms most commonly associated with the term public-key cryptography. As with AES and DH-KEX, RSA/DSA are vitally important for almost all modern communication.

Anyone can generate an RSA key pair, so it can't be used effectively without some measure of trust in a server's public key. Digital certificates fill this role, where a third-party authority verifies the server's ownership of a public key, and instills some trust in proceedings. Certificates themselves are complex, and require careful validation if they are to be used safely.

Python and Cryptography

It doesn't hurt to remind you that you shouldn't be coding up your own cryptographic algorithms. We'll continue using the Cryptography library (https://cryptography.io/) in Python for this lab, please refer to last week's document for more background. As a brief reminder:

- Cryptography separates the safer "black box" encryption algorithms from lower level primitives using its *Recipe*, and *Hazardous Materials* layers.
- Almost all communication between Python and Cryptography is done via the bytes class, which you can declare with the b character:

```
>>> byte_string = b'Secret Message'
```

- Cryptography wraps OpenSSL, which we usually pass as a backend parameter to functions:

```
>>> backend = default_backend()
>>> h = hmac.HMAC(key, hashes.SHA256(), backend=backend)
```

Server Communication

As with last week, many of these algorithms see most use over networks and the internet. Today we'll use the same class as before to emulate this process. This class is called EncryptionServer, and is imported and used like this:

```
>>> from servers import EncryptionServer
>>> server = EncryptionServer()
```

You can find the code for this server in the ./servers directory, it inherits from RSAServer, which you should feel free to look over if you'd like to see how it works. You'll find a table with the available functions we'll be using throughout the lab at the end of this document, I'll mention them as we go along.

The Lab Code

All of your code for this lab can be found within the ~/lab3 folder. Initial code for this exercise can be found in lab3.py. Open it in your preferred editor, you might also want to open a command prompt so that you can execute the file. The code begins by importing the necessary modules, importing the local server code, and then has headings where you'll put your own code.

HASH FUNCTIONS

Hash functions take a message of any length, and return a message of a fixed length. They perform a pivotal role in modern encryption, for example in message authentication codes, and as a basis for digital signatures. They operate in a similar way to hash functions you find in programing languages, such as those that drive hash tables or dictionaries. Cryptographic hash functions are traditionally longer, and have much stricter requirements. These are:

1. Given a hash of a message, you can't reverse it and extract the original message
2. Given a message and its hash, you can't find another message that hashes to the same thing.
3. You can't find any two messages that have the same hash.

In the Cryptography library, hash functions operate a lot like the other primitives, with one or more calls to `update()` before a call to `finalize()`. Unlike a cipher, because the hash function cannot return useful information before seeing the entire message, it is the call to `finalize()` that returns the hash.

To begin with, let's look at the so-called avalanche effect in SHA-256. Even a small change in a message causes a large change in the output. Begin by using the SHA-256 hash function to hash `message_one` as it appears in the lab3.py file. You can find an example of SHA-256 usage in the documentation (link). Next, output the final hash to the screen, for convenience you can print the string in HEX notation using `print(b16encode(hash))`.

Next, perform the same operation on `message_two`. You'll need to create a new instance of the hashing class, you can't use a hash function once it's been finalized. Once you've finished this, notice that the resulting hashes are vastly different. SHA-256 seems to be working correctly!

Now we'll calculate a hash on a much longer message – the entire works of Shakespeare. This code will load them into a bytes object for you:

```
with open('./data/shakespeare.txt', 'rb') as f:
    data = f.read()
```

Calculate the hash, then print it to the screen. If your code is right, the hash should be:

```
'C5601D266987AC885CA96522B1B4E439FEB7ECA39F0FE1111F7342B63B6468F3'
```

> OPTIONAL
> If you're interested, try changing a single character within the Shakespeare.txt source file. Does the avalanche effect still work?

We'll look at digital signatures shortly, these are effectively a signed (encrypted with a private key) hash, usually including some padding.

ASYMMETRIC CRYPTOGRAPHY

Public Key cryptography plays an important role in modern communications. It is usually used to sign and verify some components of protocol handshakes, to ensure that no man in the middle in involved in a conversation. For the rest of the lab we will look at the role that RSA plays, and how we manage RSA keys via digital certificates.

Recall from the lectures that each RSA key-pair contains a public key (e, n), and a private key (d). These keys perform inverse operations, which provides us with two use-cases:

1. We could encrypt a secret message using someone's public key, then only they could decrypt the message using their private key. Many people can talk to a single server securely this way.
2. Someone could sign a message with their private key. Anyone else can then decrypt the message with their public key, proving they provided the original signature.

Case (1) has a problem, in that if the private key is ever broken or stolen, suddenly everyone can decrypt everyone else's messages. This is why RSA and DSA aren't usually used for long term encryption. Case (2) is much more useful.

The `EncryptionServer` I've written has a private key, which is stored in `./servers/rsa_private.pem`. A PEM file is a standard format for public and private keys, as well as for parameters such as DH params. Since this private key is supposed to be secret, we'll pretend that it is, and only use the public one. This can be found in `./data/rsa_public.pem`.

We need to load the public key in order to use it, this is easily done in Cryptography. To save you the task of reading through the documentation on this one, here is the code, add it into your lab3.py file.

```
with open('./data/rsa_public.pem', 'r') as f:
    pem_data = f.read().encode("UTF-8")
    server_rsa_public_key = load_pem_public_key(data=pem_data,
                                                backend=backend)
```

If you want to examine the actual numbers stored in the key:

```
>>> server_rsa_public_key.public_numbers().e
65537
>>> server_rsa_public_key.public_numbers().n
28654777882514 ... 68910938507971
```

Now the key is loaded, we can either encrypt messages for the server (we won't do this now). Or we can have the server sign a message, and then verify their ownership of the private key. Think of it like a challenge-response, we challenge the server to verify our message, and when they do, we know they own the private key.

Create a message to have the server sign, it must be a bytes object, and can be any length, either random or a sentence. Once done, have the server sign the message like so:

```
>>> signed_message = server.sign_document(message)
```

This returns a digital signature - a hashed and padded message, encrypted with the server's private key. We can now run a verification process to confirm the server signed it correctly.

In Cryptography, verification either does nothing (for a pass) or raises an Exception. This is the code you need:

```
try:
    # Verify the signed message using public_key.verify()

    raise InvalidSignature("Replace this with verification code!")

    print ("The server successfully signed the message.")
except InvalidSignature:
    print("The server failed our signature verification.")
```

Insert your verification test in place of the raised exception. You can find an example of the function in the documentation (link). The example uses PSS padding, since this is also what my server uses, you should do the same. Padding is used to ensure that the numbers being encrypted are not too small. Imagine if your message was "1". The ciphertext would be 1^e mod n, which is also 1. Not very secure! Padding ensures that whatever size or length the message takes, the actual encrypted value is close to the size of n.

DIGITAL CERTIFICATES

The flaw of RSA is that it only proves that someone controls a private key. Anyone can generate an RSA key pair, it takes a couple of seconds on a modern machine. Try it if you'd like to see:

```
>>> from cryptography.hazmat.backends import default_backend
>>> from cryptography.hazmat.primitives.asymmetric import rsa
>>> private_key = rsa.generate_private_key(
        public_exponent=65537,
        key_size=2048,
        backend=default_backend()
    )
```

This fallibility would cause a problem for Diffie-Hellman. A man-in-the-middle could also generate a false public and private key pair, replacing the true ones. To prevent this, we fix public keys into a certificate that can't be forged. This is done by getting a third party to sign it. These certificate authorities will theoretically only sign the true public key of a server, and so make it impossible to forge messages from that same server. For the last part of this lab, we're going to look at some certificates, and validate a certificate chain from nottingham.ac.uk through to the root certificate of the CA.

When you visit https://nottingham.ac.uk, a TLS handshake is performed that incorporates a digital signature. Nottingham signs the client and server random values, as well as the Diffie-Hellman parameters using its private key. It does this because only the Nottingham server controls this key, and it prevents other servers or attackers from acting as Nottingham. In order to provide trust, Nottingham's server sends over a digital certificate, one which stores its public key. This certificate is signed by a certificate authority, one that we already trust.

The lab files contain Nottingham's current public-key certificate. It is stored as a PEM file, it just contains more information than the RSA public key we used earlier. It's loaded using part of the X.509 Recipe's layer in cryptography. Load the certificate now by adding the following code to your lab3.py file:

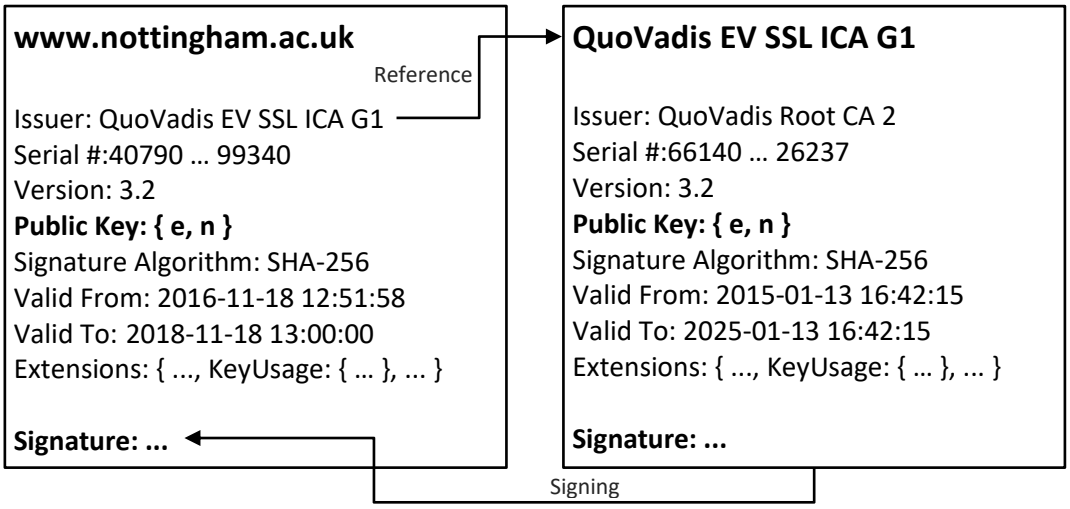
```
def get_bytes(str):
    with open(str, 'rb') as f:
        data = f.read()
    return data

certificate = load_pem_x509_certificate(get_bytes('./data/certs/nottingham.pem'), backend)
```

Using either the terminal, or the print function, have a look at some of the parameters of the certificate – see also the documentation (link). In particular, look at the `.subject` and `.issuer` variables. They're a bit convoluted, with multiple sub-components, you can drill down with code like this:

```
>>> from cryptography.x509 import NameOID
>>> certificate.subject.get_attributes_for_oid(NameOID.COMMON_NAME)[0].value)
'www.nottingham.ac.uk'
>>> certificate.issuer.get_attributes_for_oid(NameOID.COMMON_NAME)[0].value
'QuoVadis EV SSL ICA G1'
```

QuoVadis, whoever that is, signed Nottingham's certificate. The structure of the certificates is therefore a little like this:



One thing worth noting is that the next certificate also contains a signature from another certificate, and a different issuer, which means it isn't a root certificate. This is a certificate chain, or a trust chain. EV SSL ICA G1 is an intermediate certificate, it's not stored in our

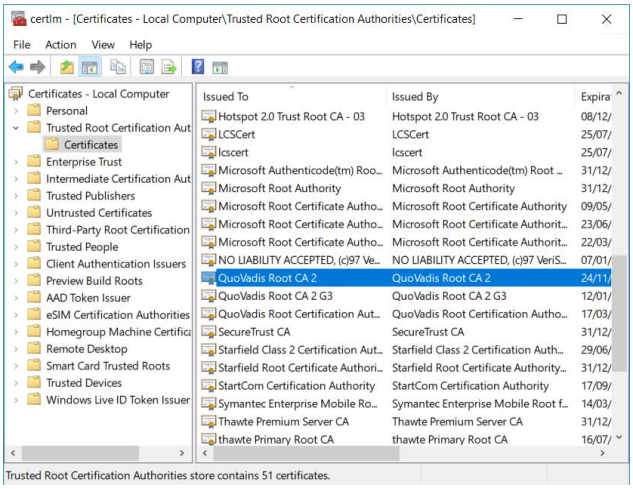
operating system, which would give us the trust we're looking for. We can follow the next issuer for the intermediate signature and obtain another certificate. At some point in the chain we'll reach a root certificate, that is, one that signs itself, and is permitted to sign other certificates. All certificates have KeyUsage information within their extensions. Have a look at the KeyUsage for Nottingham's certificate like this:

```
>>> import cryptography import x509
>>> certificate.extensions.get_extension_for_class(x509.KeyUsage).value.digital_signature
True
>>> certificate.extensions.get_extension_for_class(x509.KeyUsage).value.key_encipherment
True
>>> certificate.extensions.get_extension_for_class(x509.KeyUsage).value.key_cert_sign
False
```

Nottingham's certificate is permitted to sign key data, and produce digital signatures, but not to sign other certificates. Certificates for Nottingham sub-domains are signed by EV SSL ICA G2 as well.

The complexity and additional rules on certificate usage might seem overly controlling, but careful control of what certificates can do is crucial for security. The private key for QuoVadis Root CA 2 will be kept under lock and key, on a machine not connected to the internet. Signing of domains like Nottingham's is delegated to sub-ordinate certificates.

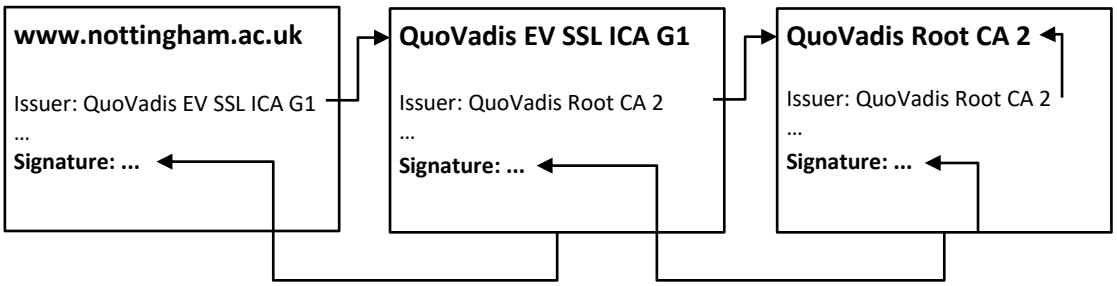
VALIDATING CERTIFICATE CHAINS



Load both of the remaining certificates, intermediate.pem and root.ca.pem in the same way as the nottingham certificate was loaded. Have a look through their parameters if you'd like.

QuoVadis Root CA 2 is in our OS's list of trusted root certificates, so if we can establish a chain of signatures between the three certificates, and if all certificates meet our other criteria, we can trust Nottingham's. Certificate validation is a complex and error prone process (for example, checking certificate revocation lists), we will implement some working initial code, but will leave the full implementation to software libraries. You need to verify the signatures of each certificate, as well as the start and end date.

This is the chain you need to validate:



The code required is similar to that we used to verify a digital signature earlier in the lab. Begin with the Nottingham certificate, verify its signature using the public key on the intermediate certificate. Next, verify the signature on the intermediate certificate using the root CA. Finally, validate the start and end dates of each certificate to ensure they're in date.

Some hints:

- Each time, you'll use the public key of the next certificate in the chain to verify the signature on the previous certificate.
- Most of the information you need is held in the certificate being validate. In particular: `.signature`, `.tbs_certificate_bytes`, and `.signature_hash_algorithm`. The tbs bytes are the certificate's bytes, i.e. the message that was signed.
- The padding used in these signatures is `padding.PKCS1v15()`.
- The root key signs its own certificate – it is self signed.
- Each certificate has `not_valid_before` and `not_valid_after` attributes. These can be compared with the current time, to check the validity of the dates. The current time can be obtained using `current_time = datetime.datetime.utcnow()`.

OPTIONAL

If you'd like to go further with your validation, you can check the KeyUsage attributes for each certificate. Both the intermediate and root certificates should be able to sign others. The Nottingham certificate should be able to sign keys and digital signatures.

CONCLUSION

In this lab session you've gained practical experience of hash functions, digital signatures, and digital certificates. Whenever a TLS handshake is performed, your browser or client will validate the certificate chain to ensure that the server is who they say they are, and they can be trusted. Public key cryptography makes this possible, and makes sure that when we perform Diffie-Hellman, no third party can join in that process. Much of this work is underpinned by hash functions, which provide a useful tool for summarizing longer messages prior to signing.

ENCRYPTIONSERVER FUNCTIONS

Function Definition	Description	Parameters	Returns
Symmetric Cryptography			
`decrypt_aes_ecb(key, ciphertext)`	Decrypts one or more blocks of ciphertext using AES in ECB mode with the provided key	*key*: bytes object of 16, 24 or 32 bytes *ciphertext*: bytes object that has a length that is a multiple of 16 bytes	bytes object containing the decrypted plaintext
`decrypt_aes_ctr(key, message)`	Decrypts any sized message using AES in CTR mode	*key*: bytes object of 16, 24 or 32 bytes *message*: bytes object of len(ciphertext) + 16. The initial 16 bytes must contain the CTR nonce, the remaining bytes the ciphertext.	bytes object containing the decrypted plaintext
Key Exchange			
`get_parameters()`	Returns the shared parameters for use in a Diffie-Hellman key exchange	None	DHParameters object containing shared parameters for Diffie-Hellman.
`get_public_key()`	Generates a secret value for the server and returns the derived public key	None	DHPublicKey object containing the server's public Diffie-Hellman value.

Function Definition	Description	Parameters	Returns
`submit_public_key(pk)`	Takes a third-party public key and performs the last step of the Diffie-Hellman handshake, computing the shared secret. Also uses this secret to derive an AES session key.	*pk*: DHPublicKey object representing the client side of the Diffie-Hellman handshake	None
`get_encrypted_message()`	Encrypts a message encrypted using AES in CTR mode if a shared secret is present (after a successful Diffie-Hellman exchange)	None	bytes object containing a nonce and ciphertext. The initial 16 bytes hold the nonce, the remaining bytes hold the ciphertext
RSA			
`sign_document(message)`	Receives a message and computes a digital signature using the server's private RSA key.	*message*: bytes object from which to calculate a digital signature. The signature is computed using PSS padding.	bytes object containing the digital signature of this message.
`submit_message(ciphertext)`	Receives an encrypted message and attempts to decrypt and print it. This function is not used during these labs, but feel free to try it out.	*ciphertext*: bytes object containing a message encrypted with the server's public key, using OAEP padding and the SHA-256 hash function.	None. Prints the message to the screen.