G53SEC Lab 2: Cryptography Part I

Lab Description

In this lab you'll be making use of a modern cryptography library within Python to encrypt and decrypt messages using a modern symmetric algorithm. You'll make use of the Advanced Encryption Standard, and gain experience with different modes of operation. Since it's unusual to have a secret key to hand, this lab will also use the Diffie-Hellman key exchange protocol to establish a shared secret with a server during an initial handshake, before deriving a symmetric key and decrypting the server's message. If you've never used Python before, don't worry. The ability to program in any other language will be enough for this lab, you can learn the syntax as you go. I'll also provide any more complex code as required.

Introduction

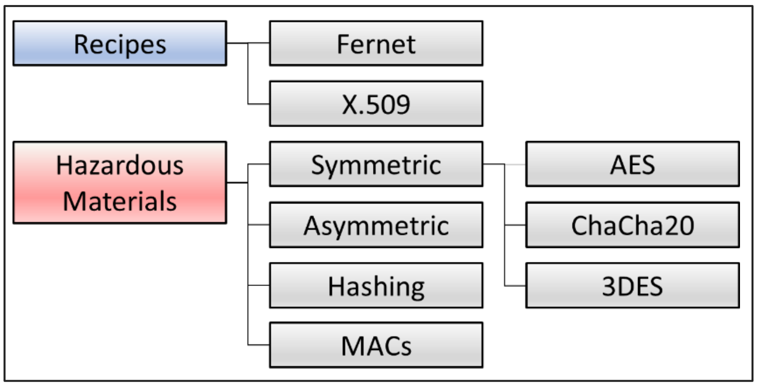
Symmetric encryption lies at the heart of most modern communication and storage technologies. In your future career, it is a near certainty that you'll come across encryption in one way or another. Knowledge of how these algorithms actually work, and practical experience of using them will mean you're much less likely to make fundamental security mistakes in your code.

The Advanced Encryption Standard is the NIST agreed standard for symmetric encryption. The actual algorithm, Rijndael, is a 128-bit block cipher that accepts 128, 192, or 256-bit keys. It is usually used with modes of operation such as counter mode, and more recently AEAD modes such as Galois Counter Mode. The use of symmetric encryption usually assumes we have access to a shared secret between both parties. Diffie-Hellman key exchange is the most common way of achieving this, a mathematical handshake that allows us to establish a shared secret over an insecure channel.

The Cryptography Library

Programming your own encryption algorithms (termed "rolling your own") is strongly discouraged. It's far too easy to make mistakes, and besides there are many excellent implementations available, why invent the wheel? Popular libraries such as libsodium (https://libsodium.org) and NaCL (http://nacl.cr.yp.to/) focus on using so-called Recipe layers.

Recipes are black box algorithms that perform encryption and decryption in a secure way, where you have almost no control over the underlying ciphers, which are used and how they are used. In this way, security decisions such as how to generate random numbers are taken away from the hands of developers, theoretically reducing mistakes and improving security. Remember, most developers using these libraries won't have been to lectures covering cryptographic primitives.



Cryptography Module Layout

The best general advice is to use recipe layers within well-tested libraries as much as possible, using lower level primitives only when necessary (e.g. for compatibility with existing code). However, this wouldn't make for a very interesting lab session! We're using the Cryptography library (https://cryptography.io/), a Python library that contains recipe layers, but also exposes the underlying primitives. These are placed in the wonderfully named "Hazardous Materials" layer, as a warning to those that are in over their heads. Cryptography is open-source, and uses OpenSSL as its backend. It is secure, but also easy to use.

Python and Cryptography

Before we get started on symmetric encryption, it's worth covering a few basics of Python and the Cryptography library. You can run python from the command line by simply typing python, or execute a file using python `file.py`. If you've not used Python before, it might be worth firing it up now and running through this initial section on strings, otherwise a quick read should be fine.

Using Cryptography within Python is straightforward, you can import the module using:

```
>>> import cryptography
```

Usually we use more targeted imports, such as:

```
>>> from cryptography.hazmat.primitives.ciphers import Cipher
```

I will supply these imports for you in most cases, to avoid you spending the entire lab session reading documentation. However, keep a window open on the documentation just in case! (https://cryptography.io/en/latest/)

At the top of the lab files you'll also notice this line of code:

```
>>> backend = default_backend()
```

When Cryptography was first written they envisaged having multiple backends performing the various functions. At the moment, there is only one, OpenSSL. This default backend will be passed as a parameter to most functions.

NOTE
You can code python within a file, or directly at the command line. The python interactive prompt uses three greater than signs (>>>), when you see these in my examples this is just me formatting Python code in the usual way, allowing us to distinguish between input and output. You never actually have to type these!

Cryptography deals with sequences of bytes in most cases, which can lead to a few headaches for those not experienced with Python. We're using Python 3, which unlike Python 2 distinguishes between strings and sequences of bytes. In Python 2 you could pass a string into Cryptography directly as a key, or as some message or ciphertext. This is because in Python 2, the str class is actually just a sequence of bytes, like in C. This is not the case in Python 3, where strings are Unicode objects, and separate to the bytes object that we will be using. Luckily converting between Unicode and bytes is straightforward:

```
>>> s = "Café"
>>> s
'Café'
>>> type(s)
<class 'str'>
>>> b = s.encode("UTF-8")
>>> b
b'Caf\xc3\xa9'
>>> type(b)
<class 'bytes'>
>>> s = b.decode("UTF-8")
>>> s
'Café'
```

By default strings are Unicode, represented by the str class. This explains why the special character é is supported. Unicode stores a series of code points representing characters, these can be 2, 3 or 4 bytes long. Calling `.encode()` on a str will convert it into a sequence of bytes using the given encoding scheme. Bytes can usually be identified by the leading b when they are printed, but they also display in a slightly uglier way, including any escape sequences for the given encoding scheme. Bytes can be converted back into a str using `.decode()`. Bear this all in mind, because only bytes are accepted by Cryptography.

During this lab, we'll be printing bytes such as ciphertext to the screen from time to time. When python prints bytes, it assumes that the bytes represent some meaningful readable text. This isn't always the case, which leads to some confusing output. Have a look below:

```
>>> import os
>>> os.urandom(16)
b'|z+\x19\x9ag\xf59Z\xac?3b\xa4\x10$'
```

What's happened here is that Python is displaying any meaningful ASCII characters as themselves, and anything else as escape characters. Here is the same string with ASCII highlighted in red, and escape characters in blue:

```
b'|z+\x19\x9ag\xf59Z\xac?3b\xa4\x10$'
```

In the same way it is printed, you can declare byte strings directly by prepending a b to a regular string declaration, like this:

```
>>> b = b'A secret message'
>>> b
b'A secret message'
```

This is still a bytes object, but it appears readable because it contains only ASCII characters.

Server Communication

For the remainder of this lab, you'll want to be working within the `~/lab2/` directory. Normally we might use symmetric encryption to communicate over a network, perhaps between a client and a server. Key exchange is also traditionally done between two remote parties. Rather than set up network communications for this lab, we'll use a Python class to emulate one. This class (which I wrote) is called EncryptionServer, and is imported and used like this:

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

This server is stored locally to your file in the `./servers` directory, feel free to have a look through 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.

Symmetric Encryption

Some initial code for this exercise can be found in lab2.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.

Electronic Code Book

Add your code for this section under the appropriate comment in lab2.py. For the first part of the lab, we're going to use AES in Electronic Code Book (ECB) mode to encrypt a two-block message. ECB simply encodes one block at a time, with no connection between them – it's not secure! Nevertheless, it's a fine place to start. The code defines a static 128-bit key, and a message we'd like to encrypt. Normally you wouldn't hard-code a key like this, but this will do until we have a better solution.

Your first task is to encrypt the message using the key. You'll need to make use of the `Cipher` class, as well as `algorithms.AES()` and `modes.ECB()`. As a guide, have a look at the example of CBC mode in the documentation (link). The main differences in your code are that ECB does not require an IV. ECB will also raise an exception if your message is not a multiple of 16 bytes (the block size of AES), but luckily it is. Once you've encrypted the message, you can confirm you were successful by asking our server class to decrypt it again:

```
>>> print(server.decrypt_aes_ecb(key, ciphertext))
```

Add this in, you should see your message back out again.

Counter Mode

The next step is very similar, we're going to use AES in counter mode (CTR) to encrypt a message of an arbitrary size. Counter mode is also much more secure than ECB mode, as it doesn't leak details of identical plaintext blocks. You'll see a new key, message and nonce are set up for you. It's vitally important in CTR mode that you don't use a nonce twice, so change this code to generate a cryptographically secure random number. `os.random()` and `os.urandom()` are suitable for this.

Once done setting up variables, use similar code to part 1 to complete an encryption using CTR. Then test your result using the server's `server.decrypt_aes_ctr()` function. This function expects a key, and a byte sequence containing the nonce prepended to the ciphertext. See the reference at the bottom of this document, or the server's code.

Key Exchange

It's very insecure to store a key within your code, and it's not practical to generate a random one every time – how are we transporting it to the server? Diffie-Hellman allows us to establish a shared secret over an insecure channel. The steps are:

a. Agree mathematical parameters with the server (g,n)
b. Generate a private key (a)
c. Calculate a public key from the private key (g^a mod n)
d. Receive a public key from the server (g^b mod n)
e. Combine the server's public key with our private key into a shared secret (g^{ab} mod n)

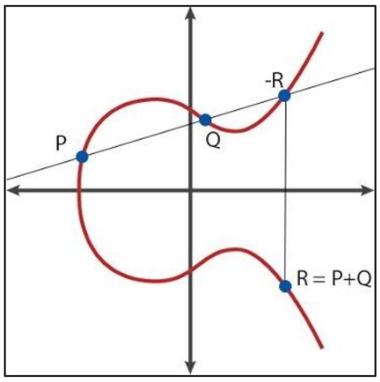
f. Send your public key to the server so they can do the same.

Although this is a complex mathematical process, it can be achieved with only a few lines of code. Write these now, using the example in the documentation (link). Bear in mind this example shows the complete process with both parties, in our case the server will handle the second participant, so you only need to code one side. You can complete steps (a) and (d), to obtain parameters and the other side's public key, using the `server.get_parameters()` and `server.get_public_key()` functions. The final step (f) can be achieved using the `server.submit_public_key()` function.

OPTIONAL – ELLIPTIC CURVES

The above Diffie-Hellman exchange uses modular arithmetic. If you're interested, it's good experience to try the elliptic curve variant.

The server and your own code are easily adapted to elliptic curves. First, change the server class to inherit from ECDHServer instead of DHServer. All the methods are the same, but it now uses the elliptic curve parts of Cryptography. Next you'll have to adapt your own side of the key exchange to use elliptic curves, this isn't too hard, and you can find an example in the documentation (link).



KEY DERIVATION

It's customary to derive a key from the shared secret, rather than use the shared secret directly. The server will do this when you submit a public key for the exchange. You can do it using this code:

```
>>> hkdf = HKDF(algorithm=hashes.SHA256(),
                length=32,
                salt=None,
                info=b'g53sec',
                backend=default_backend())
>>> aes_key = hkdf.derive(pre_master_secret)
>>> print (len(aes_key))
32
```

The hashed key derivation function derives a fixed-length key using an HMAC, based on the shared secret, a salt, and additional info. The salt and info are optional, they're used to provide different derived keys for different circumstances while still using a single shared secret. In this case we'll use no salt, and the same info as the server uses.

The code above is identical to that which the server uses. If your handshake has been successful, and your derived key is the same, you should be able to communicate securely.

Use the `server.get_encrypted_message()` function to obtain a new encrypted message, then write code to decrypt this using AES-CTR. The nonce required will be returned by the server as the leading 16 bytes. See the function definition at the end for further information.

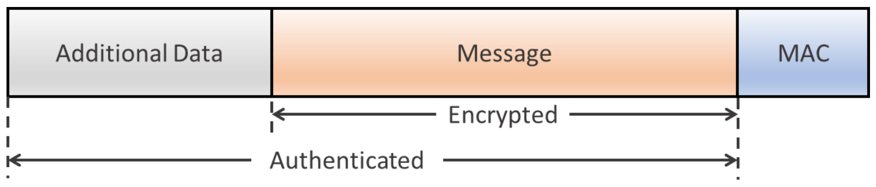
Authenticated Encryption

Recall from the lectures that symmetric encryption alone is usually not enough. A stream or block cipher doesn't preserve message integrity, so we often attach a message authentication code to ensure that a message hasn't been altered. HMAC is the most popular, it's extremely easy to use in Cryptography:

```
>>> from cryptography.hazmat.primitives import hashes, hmac
>>> h = hmac.HMAC(key, hashes.SHA256(), backend=backend)
>>> h.update(b"message to hash")
>>> code = h.finalize()
```

This has become so common that it's been built into modern cipher suites. Notable ones include ChaCha20Poly1305, and AES-GCM. ChaCha20 is a modern stream cipher, Poly1305 a MAC that is calculated modulo 2^{130}-5 (hence the name). GCM is Galois counter mode, commonly seen with AES, this produces a similar mac known as a GMAC, which authenticates the entire message while encryption is performed.

This type of cipher is usually termed Authenticated Encryption with Additional Data (AEAD). This is because the MAC can also authenticate (preserve integrity of) additional unencrypted data as well. It's quite common to want to preserve integrity of a message that isn't encrypted. Database records, for example, or the headers of packets travelling over a network. Often the payload is encrypted, and the header or metadata merely authenticated. This diagram shows a common AEAD message structure:



AEAD Message Structure

For the last part of today's lab, we'll use AEAD to encrypt a fictitious database record. Some of the record will be encrypted, being sensitive information like a date of birth, or address. Other fields must simply be authenticated, to ensure they are not altered, but allow them to be indexed or searchable within the database.

The primitives for AEAD are not kept in the same module as the regular ciphers, the import for AESGCM and ChaCha20 are included already:

```
>>> from cryptography.hazmat.primitives.ciphers.aead import AESGCM
>>> from cryptography.hazmat.primitives.ciphers.aead import ChaCha20Poly1305
```

Both ciphers have the same interface, so either will work. They support varying key and nonce lengths, but a key size of 256-bits and nonce of 96-bits are supported by both. Begin by creating a random key and nonce as appropriate.

In the py file you'll find the code for the fictitious database record, as below:

```
record = {
    "ID": "0054",
    "Surname": "Smith",
    "FirstName": "John",
    "JoinDate": "2016-03-12",
    "LastLogin": "2017-05-19",
    "Address": "5 Mornington Crescent, London, WN1 1DA",
    "Nationality": "UK",
    "DOB": "1963-09-14",
    "NI": "JC123456C",
    "Phone": "01224103232",
    "Data": None,
    "Nonce": None,
}
```

In case you're new to python, `{}` initializes a dictionary of key, value pairs. You can access any value using the key, e.g. `record["FirstName"]`. You'll also find a function definition here that prints the record in a slightly more attractive way than Python's default.

Your final task is to complete the two functions, `encrypt_record()` and `decrypt_record()`. Encrypt record applies AEAD encryption to some fields, while authenticating other fields. The encrypted fields must then be set to None, and the data and nonce fields populated with the ciphertext produced, and nonce used. Decrypt applies the opposite process, restoring the missing fields, and setting the nonce and data fields back to None. Use the following rules:

Encrypted fields: Address, Nationality, DOB, NI, Phone
Authenticated fields: ID, Surname, FirstName, JoinDate, LastLogin

Remember that the encrypted fields are authenticated as part of the ciphertext anyway.

Some hints:

- Use either `AESGCM()` or `ChaCha20Poly1305()`
- Documentation for AEAD is here (link). The encrypt function requires the plaintext, a nonce, and the additional data. These are three bytes objects.
- AEAD works as a single process using the entire data and plaintext at the same time, this means you can't pass in the record fields one by one. You'll need to separately concatenate the encrypted and authenticated fields into two byte strings.
- We need to be able to re-separate the encrypted data back out again during decryption. A reasonable way to concatenate the items together is to use some text delimiter, e.g. `'\x1d'`, which is the ascii unit separator. The string and bytes `.split()` function will reverse this process.

- In python 3 these are Unicode strings, you'll need to encode them into bytes prior to encryption.
- Any change in ordering of field or data will cause the cipher to raise an exception.
- Ask for help if you need it!

As a guide, this is roughly what an encrypted record should look like:

```
>>> print_record(record)
{
  ID : 0054
  Surname : Smith
  FirstName : John               Authenticated fields, can't be changed but still unencrypted.
  JoinDate : 2016-03-12
  LastLogin : 2017-05-19
  Address : None
  Nationality : None
  DOB : None                     Encrypted and authenticated fields.
  NI : None
  Phone : None
  Data : b'4\r\x07\xddG94\ ... \xc2Z<'
  Nonce : b'\x0c&/\x1eu\x99\xc9qo\x18\xc7\x06'   The encrypted data and nonce used
}
```

CONCLUSION

In this lab session you've obtained hands-on experience using a modern cryptography library. It's not hard to use, but it requires a great deal of care not to make a mistake and leave your data vulnerable. If in doubt you should use a recipe layer, but in many cases AEAD cipher suites are also a good option. Diffie-Hellman is a useful tool for sharing a session key, but don't forget from the lectures that it's vulnerable to man-in-the-middle attacks. We'll look at RSA and its ability to solve this problem in the next lab. We'll also look in more detail at hash functions, digital signatures and certificates.

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.