Group 32

INFO3616 Assignment 1

1) Reliability and Security

Security and reliability engineering are both fields dedicated to ensuring the availability of systems despite mischances and errors. In particular, reliability engineering focuses on the robustness of a system. However security engineering also considers the bugs and mishaps in the system that may expose vulnerabilities which an attacker can exploit to do deliberate and malicious things to the system. Threat modelling is a specific aspect in security engineering that considers this point, it involves the security engineers solving problems like what the vulnerabilities are in the system and how the system should defend it, etc. New threats to the system will come if the system environment changes, hence it is a continuous and theoretically never-ending process to completely secure a system.

2) Security Goals in Hospitals

The predecessor was referring to a policy goal - what the system is meant to achieve.

3 security goals related to ‘keeping patient data safe’ are:

1. Data/message integrity: detecting modifications in patient data ensures that tampering (by both good and adversaries) can always be investigated
2. Authorization: determining privilege levels ensures that people who don’t require the data for their roles can’t access it (limits leak potential)
3. Accountability (related to authorization): logs (and protection for logs) allows tracing of user actions throughout a system, can investigate users who are misusing their privileges

Some incentives to defend against can include:

1. Blackmailing (personal/monetary): details of patient data should be kept private, else can be used for blackmailing by attackers if leaked, hence position of power for those with access
2. Disruption: during wartime, availability of patient data necessary for smooth running of hospitals; during political events, attacks can cause havoc and confusion against multiple parties

3. Selling: Attacker might break into the system and steal the patient record, then sell it to research teams secretly because the data is valuable. Therefore user behaviours must be accountable in such situations.

3) A Common Problem

• The Attack

The attacker was able to extract Rosemary Morgan's username and information about her current password (and also will be able to predict her future passwords from the instructions the attacker gave), which compromises authenticity and accountability goals.

The attacker was able to exploit the tendency for humans to respect authority with minimal question, as he/she established themselves as the Information Security department and Rosemary accepted it without hesitation. Also, confirmation bias was another weakness used against Rosemary maliciously, as the attacker acted amicable and helpful towards her based on the fact that she is the new employee of this magazine and might need help, which confirmed her initial impressions of the staff being 'friendlier than she expected'. This led to an easy build up of trust and willingness to accept the attacker's words.

The attacker can call Rosemary again under the guise of another department and exploit the same weaknesses again, by being amicable and friendly. Instead of asking for account details, they can try for familial relations disguised as social services, administration etc. to get her daughter's name and thus break her password by adding the number of the current month.

• Policy, Our Old Friend

One reason why this policy is ineffective is because changing the password doesn't invalidate the fact that if it is random and of an identical length, it will take a negligibly similar amount of time for an attacker to brute force a password.

Another reason is that being forced to continuously change passwords (with policies like adding a number at the end in this scenario) can incite poor habits by lazy employees. The passwords could become more easily 'solvable' by adversaries as people change passwords to more easily memorable strings, because they know it is temporary anyways. Dictionary attacks can become common (and more effective).

• The Aftermath

Rosemary was told by the attacker on the call that their policy is "changing the password every ninety days", which she believed and followed, thus gave the attacker the chance. Therefore she was not aware of the real policy of the company.

Moreover, the real security policy to be read is likely a long and arduous read with potentially difficult to understand/jargon concepts, otherwise known as ‘information deficit’. It places an unnecessary burden on her to have to understand everything (which won’t always apply to her, would be common sense etc.).

One non-technical defense is verifying the user on the other side of the call, by asking to hold and checking with available company information and co-workers either the phone number being used or the person’s details.

One way for Rosemary to choose a better password is using the Letters-from-a-sentence method. This involves choosing a memorable statement (ideally with numbers as well), and then shortening it to the first letters of each word and number. Remembering a statement is much easier, which allows easy conversion towards the shortened string.

4) Privacy Warnings

This is a very effective way because people perceive the warning as an intrusive pop-up that directly stops their website experience, rather than an issue that could be potentially important. Playing on humanity’s laziness, the option ‘I’m ok with that’ sounds more amicable and less tiresome than exploring ‘more options’.

This can definitely be exploited by attackers, humans are generally quick to identify keywords in statements and make instantaneous decisions against these types of decisions. By writing a friendly statement and similarly adding options such as policies and readings, people will identify this as ‘another’ pop-up, act on their laziness and choose the more convenient choice (i.e. ‘yes’) which gives attackers permission to act upon their ‘policies’ and inject malicious code.

5) A Suspected Phish

1. The email targets humans’ respect for authority, as it was ‘sent’ by the vice-chancellor of USYD, Dr. Michael Spence, which of course can easily be verified as a public figure (as well as the email and address).

2. This further leads into another human weakness exploited by attackers, human consistency, as reading through the statements it advises staff of updated policies i.e. ‘this organisation holds itself to the highest ethical standards’, then asks them to ‘recognise their responsibility’ with the provided link. This follows the pattern of humans reaffirming their commitments, which can lead them to trusting the email blindly.

The email client rendered the email such that there was excessive space between the title and the actual start of the letter. Also, the title was not on center. The rendering counteracted the mind's ability to reaffirm logical patterns quickly and move on. To the human mind, this is an illogical format that differs from the norm, which can help conceive some doubt to the legitimacy of the email.

6) Evaluating a Bell-LaPadula-Based Security Policy

- ### Access Control

Table 1 implements discretionary access control (DAC) by listing the extent that subjects can interact with particular objects.

Tables 2 and 3 both implement mandatory access control (MAC) as it lists access rights based on security levels associated with both subjects and objects.

- ### Rule Consistency

Arthur being able to read beetle.exe is an inconsistent rule as beetle.exe is tagged as SECRET, whilst Arthur only has PROTECTED clearance. Another similar case is Kelly, with SECRET clearance, being able to read the missile_codeds.rar file, despite that being tagged under TOP SECRET access. Both these inconsistencies directly contradict the reading-down property of the Bell Lapadula model, where a subject has read-only access only to objects at or below their clearance level.

There is also another violation contradicting the writing-up property of the Bell Lapadula model, where a subject has append access to objects that are at or above their security clearance level. Kelly, of SECRET clearance, has write permissions to transfers.csv, which has been tagged as PROTECTED and is below her security level.

- ### Malicious Operatives

Kelly has write operations to transfers.csv, which as mentioned above directly violates the writing-up property maintained by the Bell Lapadula model. Unless there is some additional form of integrity checking (in which the model emphasizes confidentiality goals rather than integrity first), Kelly can definitely forge transfers with minimal suspicion and the information she has available based on her clearance level.

Dylan could run the time.exe program normally (as Dylan has TOP SECRET clearance for time.exe which is marked SECRET), and use time.exe to run beetle.exe, since time.exe is considered a principal here.

7) Access Control and Operating Systems

- ## OSes in General

OSes provide access control without relying on hardware communication via access control lists, which lists subjects' permissions for all objects (e.g. files, directories). This allows faster authorisation procedures.

OSes detect these violations through segmentation faults (SIGSEGV errors) that are raised when a process attempts to read or write memory outside of its own address space.

- ## Linux

The mount command has read, write and execute permissions for root, read and executable permissions for the root group, and read and executable permissions for others. Note others are run with root permissions because the execute bit is set as 's'.

Code (in case)

```
#!/bin/bash
find / -user $(whoami) -perm /007 2>&1 | grep -v "Permission denied"
```

- ## Windows

The security model implemented here is the User Access Control (UAC) model.

1. OS intercepts attempts of a program if it tries to make changes that exceed normal user rights.
2. Asks users whether they want to give permission to this program to do that

This dialogue is prompted because the firefox installer is attempting to do a task that requires an administrative token (admin access).

Autorun.inf files for CD-ROMS allows manufacturers to determine the initial settings for a new CD being installed. This ranges from specifying the applications to run, to choosing the CD icon.

The CD with an autorun.inf file can be automatically launched as long as the file is located in the root directory of a CD-ROM. However in Figure 3 the program is treated differently (as a program without an administrative token) by having the OS ask the user for confirmation before running it.

Similarly with the above case, while the CD can be automatically run as long as the autorun.inf file is located in the correct directory, mounting something in Linux requires root privileges before being allowed to execute.

- **The USB Key and Virtualised OSes**

No. A virtualised environment is not designed for access control, and attackers can use viruses that are able to discover the VM that it is in and thus operate under different circumstances in virtualised environments rather than the host machine's OS. Because the VM still requires access to hardware functionalities to run, there will always be avenues for creative adversaries to attempt unwanted access into devices.

8) Linux Privilege Escalation

A popular method attackers use to gain escalated privileges involve exploiting kernel vulnerabilities via executing programs with regular user access. These programs are written with manipulating kernel data e.g. process privileges, account updates to gain higher level access. These take advantage of un-updated or unpatched kernels.

A second method is taking advantage of programs run as root. When this happens, any vulnerabilities in that particular program can be easily exploited to gain glean unwanted information or perform actions disguised as the root user. To counteract this, it is important to ensure that no programs (that require consistent usage over a period of time) are run as root. A popular example is SQL injections.

Another vulnerability that can be used is exploiting misconfigured SUDO rights for users and programs. If an attacker finds one such opening, they can escape to the shell and execute privileges as root, e.g. an administrator enabling sudo rights to a command; the attacker can utilize some command parameters to access more commands with root privileges.

9) Nice Images

1. In `encPNG.py`, we open the binary file "`myfile.png.bin`" first, which is generated by the command `./encrypt-prep.sh myfile.png`, where `myfile.png` is what we chose.
2. Then we just read the binary data as bytes, which is already encoded since it's binary
3. We need a key for the encryption, so we use the given string "`INFO3616INFO3616`" and encode it to 16 bytes of data, as the plaintext.

4. We then create an ECB mode block cipher encryption to encrypt the data. Note that ECB mode simply encrypts each block of data separately, therefore identical blocks would encrypt to identical cipher texts. :(
5. We pad out the data before encryption, because the length of the data needs to be a multiple of 16 (block size for AES) to be divided into equal size blocks. The data is then encrypted to bytes of ciphertext.
6. We can then write the binary ciphertext to `"myfile.png.bin.enc.bin"` when we run `encPNG.py`.
7. When we run `./encrypt-finalise.sh myfile.png`, the encrypted image file is generated under the name “myfile.png.enc.png”, we can see that the original image is now encrypted.

Flow: myfile.png ----prep.sh----> myfile.png.bin ------encPNG.py-----> myfile.png.bin.enc.bin -----finalise.sh-----> myfile.png.enc.png

10) Poor Man’s AES

- ### Weakness

He uses ECB mode, which is semantically weak because in this mode every block is encrypted separately rather than correlated together using XOR operation such as CBC mode, therefore identical blocks would result in identical ciphertexts.

In his implementation of the encryption, a line of plaintext is not directly encrypted but simply get XOR’ed with the encrypted nonce to form a line of ciphertext. There are only a certain amount of nonces that he used in a repetition of 10 lines, due to his choice of ECB, we can easily get the 10 encrypted nonces by XOR’ing the ciphertext with the given header and footer plaintext (since together they applied nonces 1 to 10). After that, XOR the ciphertext with these encrypted nonces to get the plaintext back, without knowing his key. This is demonstrated in the pseudocode below:

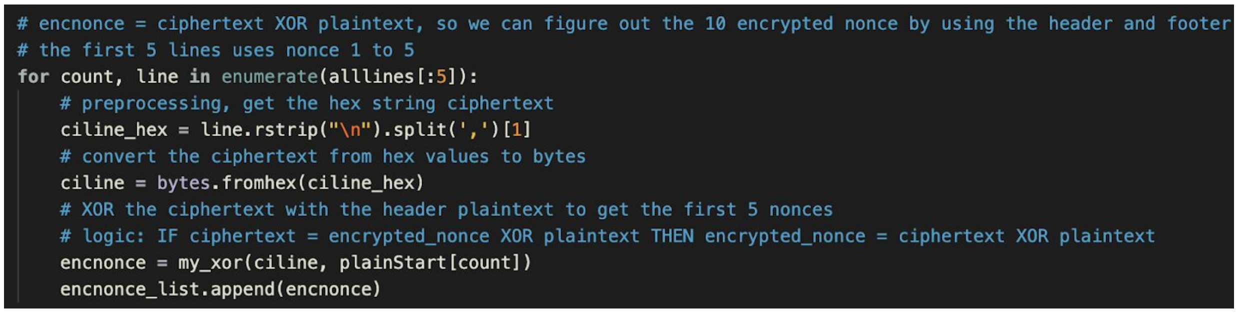
- ### Pseudocode

```
ten_encrypted_nonces = []
for count, line in first_5_lines_ciphertext do:
      encrypted_nonce = XOR(line, plainStart[count])
      ten_encrypted_nonces.append(encrypted_nonce)
endfor
for count, line in last_5_lines_ciphertext do:
      encrypted_nonce = XOR(line, plainEnd[count])
      ten_encrypted_nonces.append(encrypted_nonce)
endfor
nonce_count = 0
```

Group 32

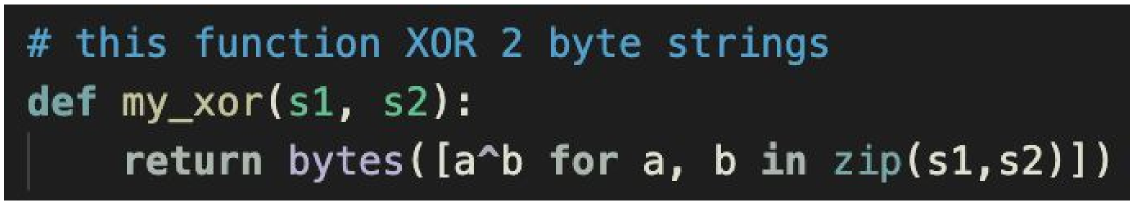
```
for line in ciphertext do:
      plain_line = XOR(line, ten_encrypted_nonces[nonce_count])
      outfile.write(plain_line)
      nonce_count = (nonce_count + 1) mod 10
endfor
```

We first figure out the encrypted nonce 1 to 5 by using the header information given and the encrypted file containing the cipher text. Because Poor Man simply XORed them to get the cipher as we discussed above, we just XOR the ciphertext with the header to get the encrypted nonces back (XOR is powerful, it allows variable substitution in a simple way. If we have a ^ b = c then we have a = c ^ b and a = c ^ a.):



```python
# encnonce = ciphertext XOR plaintext, so we can figure out the 10 encrypted nonce by using the header and footer
# the first 5 lines uses nonce 1 to 5
for count, line in enumerate(alllines[:5]):
    # preprocessing, get the hex string ciphertext
    ciline_hex = line.rstrip("\n").split(',')[1]
    # convert the ciphertext from hex values to bytes
    ciline = bytes.fromhex(ciline_hex)
    # XOR the ciphertext with the header plaintext to get the first 5 nonces
    # logic: IF ciphertext = encrypted_nonce XOR plaintext THEN encrypted_nonce = ciphertext XOR plaintext
    encnonce = my_xor(ciline, plainStart[count])
    encnonce_list.append(encnonce)
```

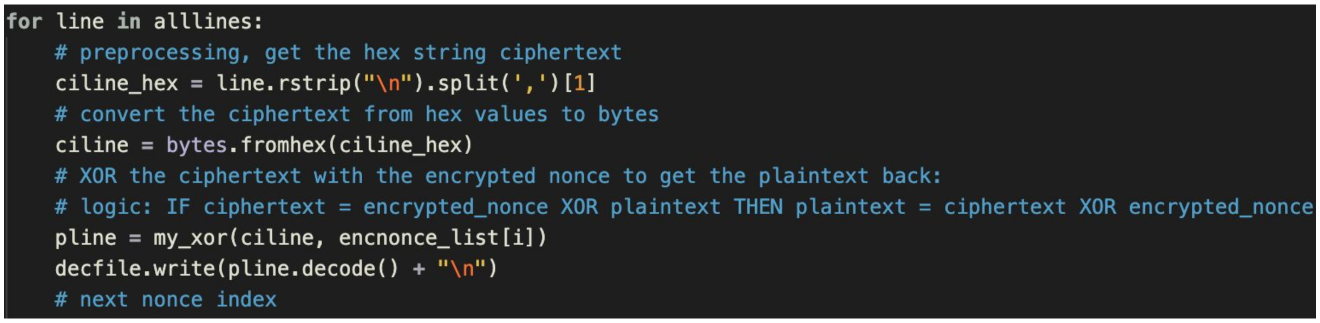
Note that we used our own XOR here, which simply XOR two bytes:



```python
# this function XOR 2 byte strings
def my_xor(s1, s2):
    return bytes([a^b for a, b in zip(s1,s2)])
```

And for nonce 6 to 10 this is done in the same way.

After that we can now XOR the ciphertext with the 10 repeated encrypted nonce we just found to get all the plaintext back:

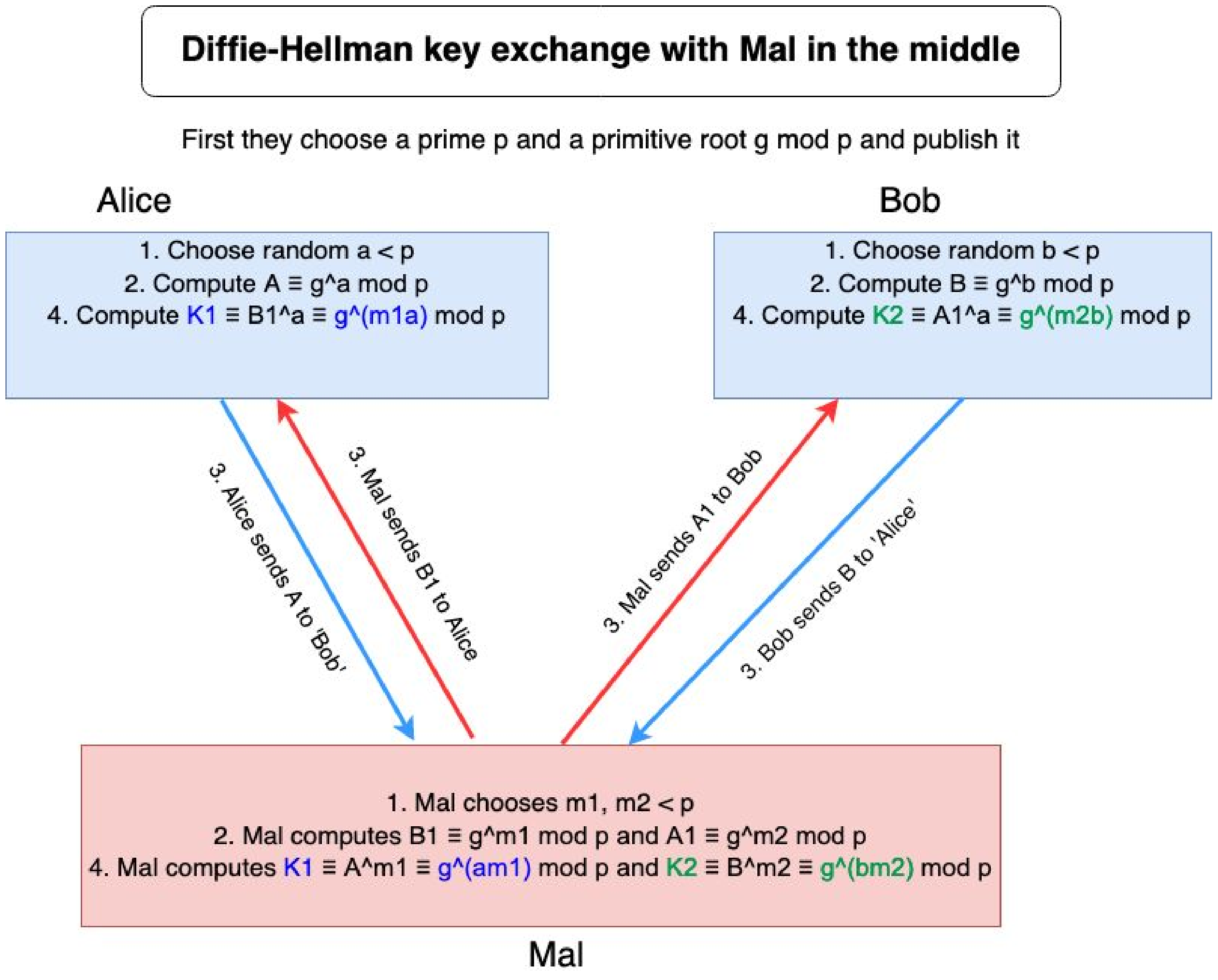


```python
for line in alllines:
    # preprocessing, get the hex string ciphertext
    ciline_hex = line.rstrip("\n").split(',')[1]
    # convert the ciphertext from hex values to bytes
    ciline = bytes.fromhex(ciline_hex)
    # XOR the ciphertext with the encrypted nonce to get the plaintext back:
    # logic: IF ciphertext = encrypted_nonce XOR plaintext THEN plaintext = ciphertext XOR encrypted_nonce
    pline = my_xor(ciline, encnonce_list[i])
    decfile.write(pline.decode() + "\n")
    # next nonce index
```

To run the code simply type `python main.py encrypted.enc`
(argument is the ciphertexts to crack)

11) Breaking Diffie-Hellman

- **Basic Ideas**
  If Mal is able to modify messages intercepted in the middle, during the key exchange, she can substitute Alice's encrypted public value with her own and send it to Bob, and substitute Bob's encrypted public value with her own. Therefore, Alice and Mal will share one key meanwhile Bob and Mal share another. This way, when Alice sends some encrypted message to Bob, she would have used the key that Mal designed for her, and similarly when Bob sends to Alice, and Mal can thus decrypt the messages between them using the keys she modified.
- **Protocol Flow**

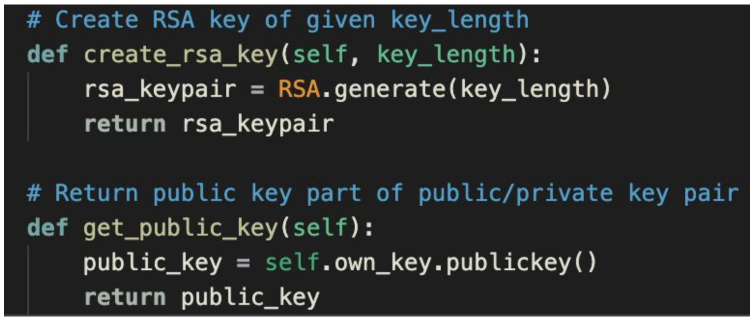

Diffie-Hellman key exchange with Mal in the middle
First they choose a prime p and a primitive root g mod p and publish it
Alice
1. Choose random a < p
2. Compute A ≡ g^a mod p
4. Compute K1 ≡ B1^a ≡ g^(m1a) mod p
Bob
1. Choose random b < p
2. Compute B ≡ g^b mod p
4. Compute K2 ≡ A1^a ≡ g^(m2b) mod p
3. Alice sends A to 'Bob'
3. Mal sends B1 to Alice
3. Mal sends A1 to Bob
3. Bob sends B to 'Alice'
1. Mal chooses m1, m2 < p
2. Mal computes B1 ≡ g^m1 mod p and A1 ≡ g^m2 mod p
4. Mal computes K1 ≡ A^m1 ≡ g^(am1) mod p and K2 ≡ B^m2 ≡ g^(bm2) mod p
Mal

- **Key Usage**
  After the key exchange, Alice will use K1 and Bob will use K2.
- **Explanation of Decryption**
  Mal can thus decrypt the encrypted message **e** that Alice sends to 'Bob' with D(e) ≡ e^K1 mod p, and similarly D(f) ≡ f^K2 mod p for **f** that Bob sends to 'Alice'.

12) Hybrid Cryptography in Python

Alice and Bob use hybrid encryption to communicate, that means:

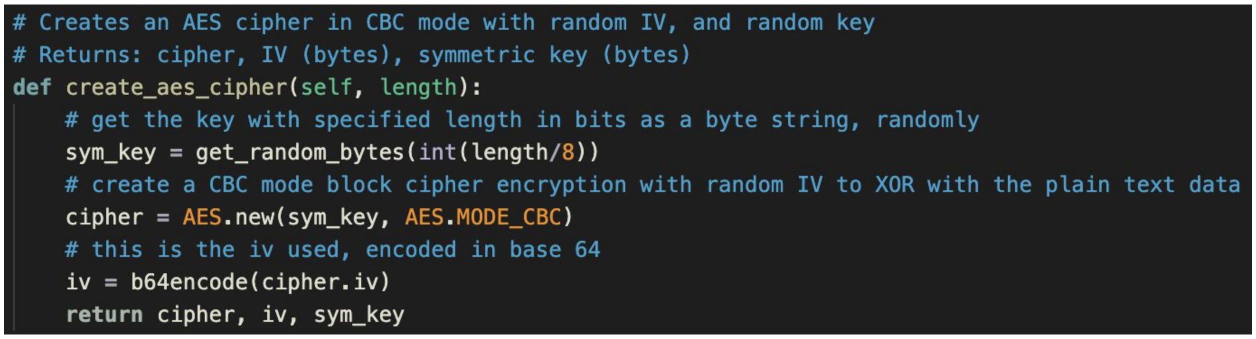
1. Alice and Bob get their own RSA public-private key pair: (pubkey: n, e, privkey: d):



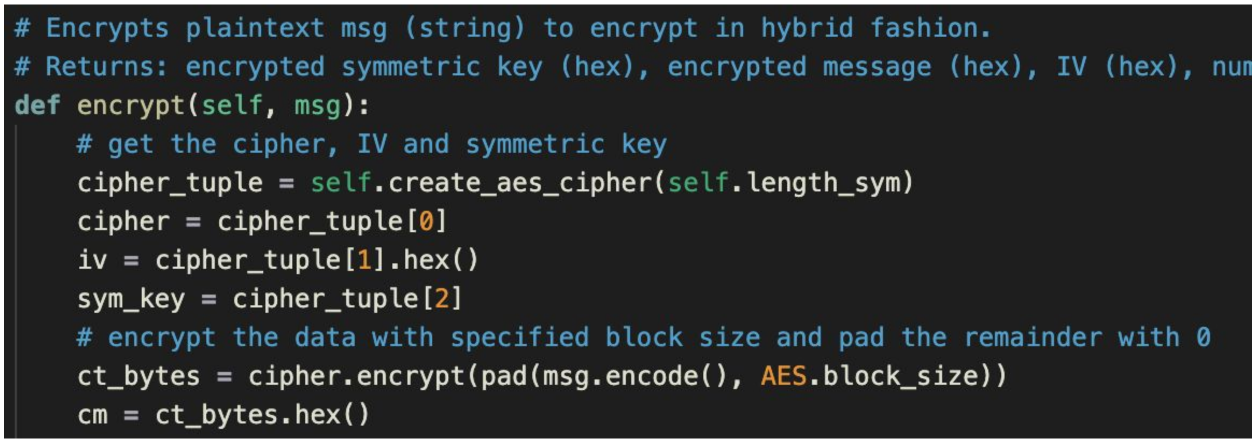
```python
# Create RSA key of given key_length
def create_rsa_key(self, key_length):
    rsa_keypair = RSA.generate(key_length)
    return rsa_keypair

# Return public key part of public/private key pair
def get_public_key(self):
    public_key = self.own_key.publickey()
    return public_key
```

2. Alice is the sender, she creates an AES-CBC cipher with random symmetric key **symkey** and initialisation vector **iv** to encrypt the message to **Em**:



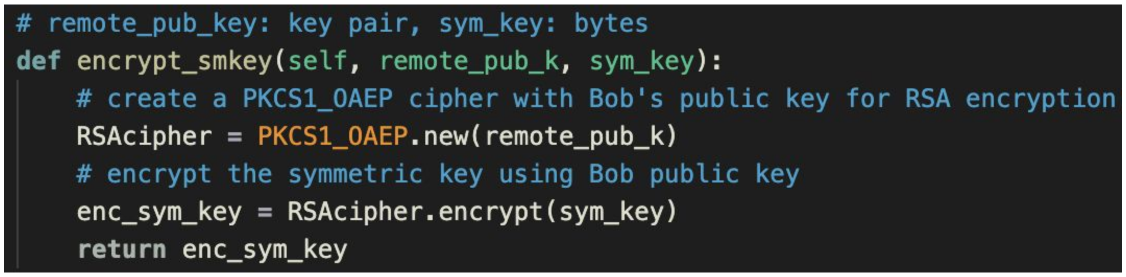
```python
# Creates an AES cipher in CBC mode with random IV, and random key
# Returns: cipher, IV (bytes), symmetric key (bytes)
def create_aes_cipher(self, length):
    # get the key with specified length in bits as a byte string, randomly
    sym_key = get_random_bytes(int(length/8))
    # create a CBC mode block cipher encryption with random IV to XOR with the plain text data
    cipher = AES.new(sym_key, AES.MODE_CBC)
    # this is the iv used, encoded in base 64
    iv = b64encode(cipher.iv)
    return cipher, iv, sym_key
```



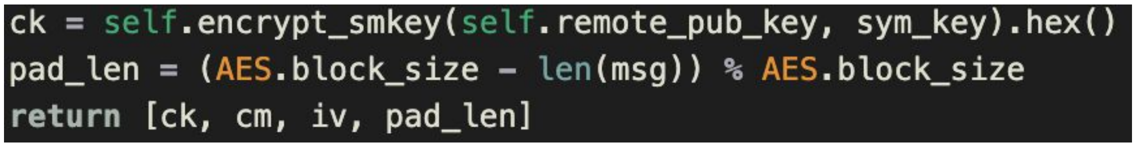
```python
# Encrypts plaintext msg (string) to encrypt in hybrid fashion.
# Returns: encrypted symmetric key (hex), encrypted message (hex), IV (hex), num
def encrypt(self, msg):
    # get the cipher, IV and symmetric key
    cipher_tuple = self.create_aes_cipher(self.length_sym)
    cipher = cipher_tuple[0]
    iv = cipher_tuple[1].hex()
    sym_key = cipher_tuple[2]
    # encrypt the data with specified block size and pad the remainder with 0
    ct_bytes = cipher.encrypt(pad(msg.encode(), AES.block_size))
    cm = ct_bytes.hex()
```

   Note that the message is padded such that it can fit into the blocks without remainder, the pad function from AES.Util.Padding can handle pad and unpad for us so there is no need to manually implement pad and unpad. The ciphertext and iv are converted to hexadecimals for sending.

3. Alice then encrypts the symmetric key to **Ek** using Bob's public key and send the encrypted hybrid message to Bob: **(Em, Ek, iv)**:

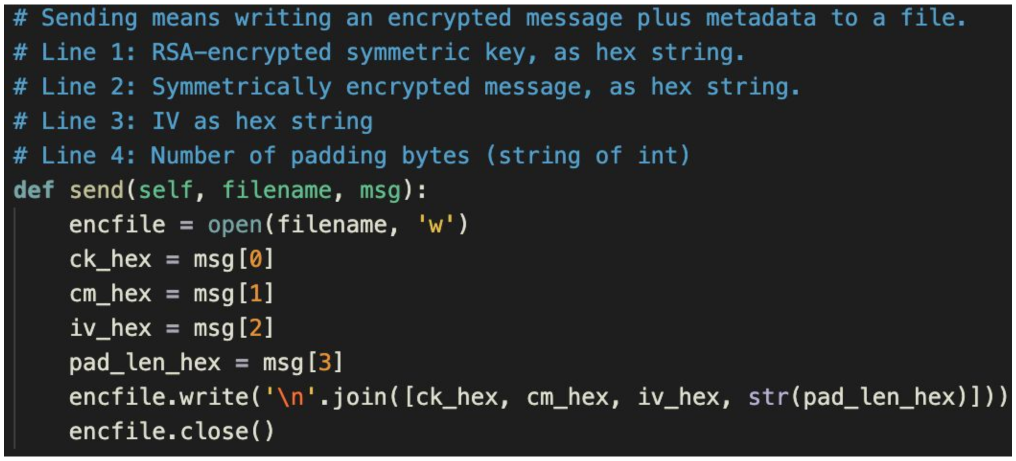


```python
# remote_pub_key: key pair, sym_key: bytes
def encrypt_smkey(self, remote_pub_k, sym_key):
    # create a PKCS1_OAEP cipher with Bob's public key for RSA encryption
    RSAcipher = PKCS1_OAEP.new(remote_pub_k)
    # encrypt the symmetric key using Bob public key
    enc_sym_key = RSAcipher.encrypt(sym_key)
    return enc_sym_key
```



```python
ck = self.encrypt_smkey(self.remote_pub_key, sym_key).hex()
pad_len = (AES.block_size - len(msg)) % AES.block_size
return [ck, cm, iv, pad_len]
```

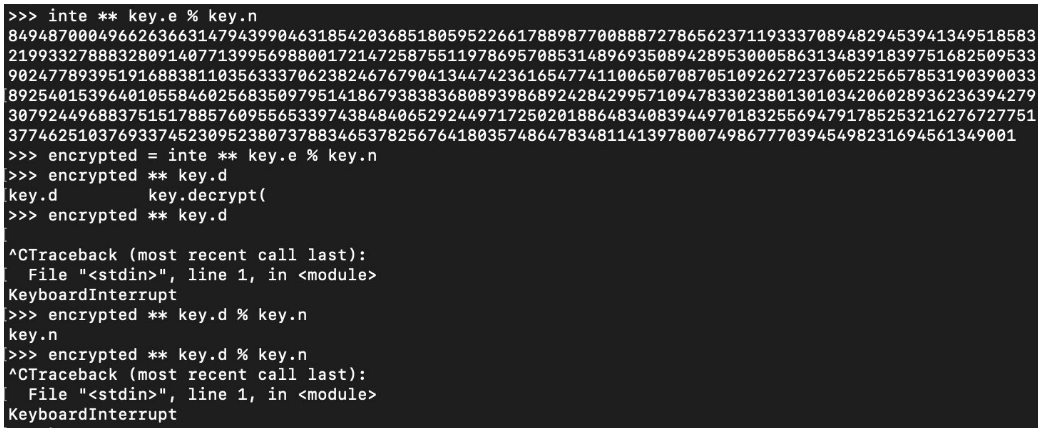
The encrypted stuff in the returned list are all hex strings for convenience, we just pass the list to the `send` function in Alice's `Principal`:



```python
# Sending means writing an encrypted message plus metadata to a file.
# Line 1: RSA-encrypted symmetric key, as hex string.
# Line 2: Symmetrically encrypted message, as hex string.
# Line 3: IV as hex string
# Line 4: Number of padding bytes (string of int)
def send(self, filename, msg):
    encfile = open(filename, 'w')
    ck_hex = msg[0]
    cm_hex = msg[1]
    iv_hex = msg[2]
    pad_len_hex = msg[3]
    encfile.write('\n'.join([ck_hex, cm_hex, iv_hex, str(pad_len_hex)]))
    encfile.close()
```

It is very complex to implement RSA cryptosystem manually whilst the key length and message are large, mainly due to:

a. It is hard to choose distinctive large primes p, q and generate e such that gcd(e, phi(pq)) = 1
b. It is hard to compute modular exponentiation during encryption and decryption **when the base and exponents are very large**, there are different algorithms for that (see MATH2988), python does not know how to reduce the exponent in some smart way, therefore the result computation will just halt if we implement RSA manually. Below is a test, note that we have to control-C:

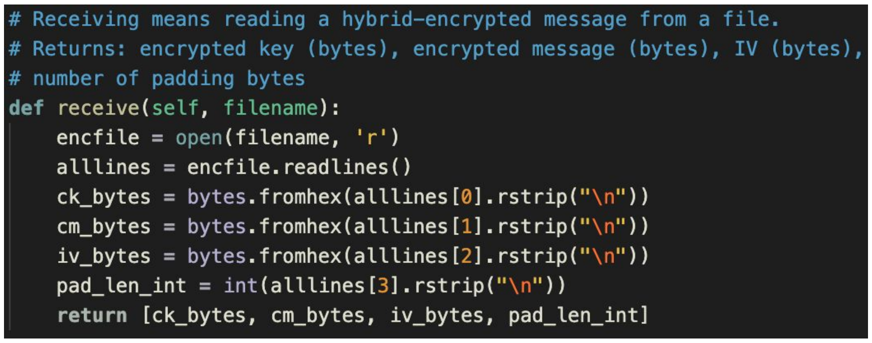


```
>>> inte ** key.e % key.n
84948700049662636631479439904631854203685180595226617889877008887278656237119333708948294539413495185832
19933278883280914077139956988001721472587551197869570853148969350894289530005863134839183975168250953
39024778939519168838110356333706238246767904134474236165477411006507087051092627237605225657853190390033
8925401539640105584602568350979514186793838368089398689242842995710947833023801301034206028936236394279
3079244968837515178857609556533974384840652924497172502018864834083944970183255694791785253216276727751
3774625103769337452309523807378834653782567641803574864783481141397800749867770394549823169456134900 1
>>> encrypted = inte ** key.e % key.n
>>> encrypted ** key.d
key.d          key.decrypt(
>>> encrypted ** key.d

^CTraceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyboardInterrupt
>>> encrypted ** key.d % key.n
key.n
>>> encrypted ** key.d % key.n
^CTraceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyboardInterrupt
```

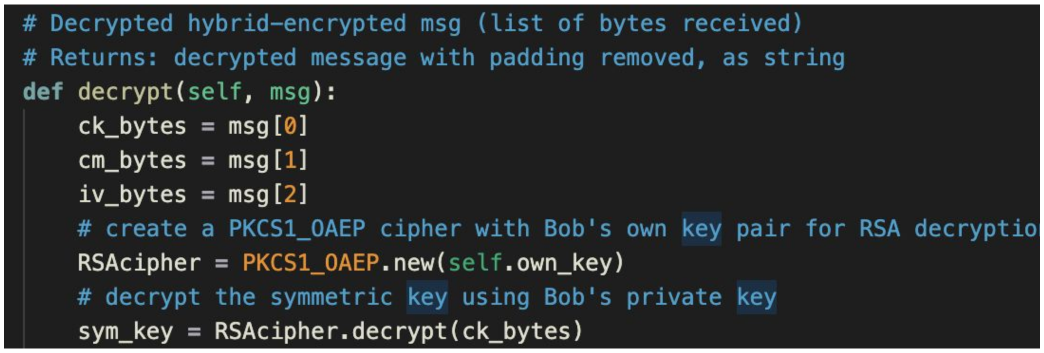
Therefore we choose to use the **PKCS1_OAEP** cipher which implements the RSA cryptosystem so that Alice can safely use Bob's public key for encryption and Bob can safely decrypt using his private key.

4. Bob receives the message, he decrypts the symmetric key **Ek** first with his own private key (since it is encrypted using his public key, therefore Ek^de ≡ k (mod n)):



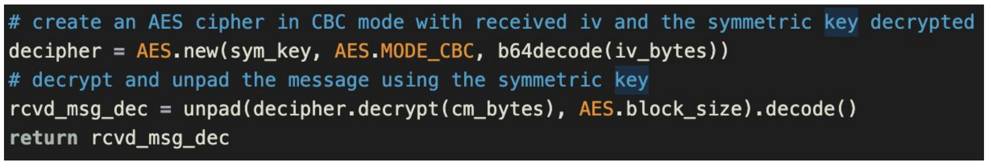
```python
# Receiving means reading a hybrid-encrypted message from a file.
# Returns: encrypted key (bytes), encrypted message (bytes), IV (bytes),
# number of padding bytes
def receive(self, filename):
    encfile = open(filename, 'r')
    alllines = encfile.readlines()
    ck_bytes = bytes.fromhex(alllines[0].rstrip("\n"))
    cm_bytes = bytes.fromhex(alllines[1].rstrip("\n"))
    iv_bytes = bytes.fromhex(alllines[2].rstrip("\n"))
    pad_len_int = int(alllines[3].rstrip("\n"))
    return [ck_bytes, cm_bytes, iv_bytes, pad_len_int]
```

We change the hex strings in the encrypted file to bytes for convenience, and this list gets passed to the `decrypt` function in `HybridCipher`:



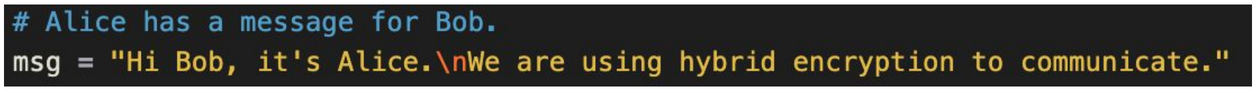
```python
# Decrypted hybrid-encrypted msg (list of bytes received)
# Returns: decrypted message with padding removed, as string
def decrypt(self, msg):
    ck_bytes = msg[0]
    cm_bytes = msg[1]
    iv_bytes = msg[2]
    # create a PKCS1_OAEP cipher with Bob's own key pair for RSA decryptio
    RSAcipher = PKCS1_OAEP.new(self.own_key)
    # decrypt the symmetric key using Bob's private key
    sym_key = RSAcipher.decrypt(ck_bytes)
```

5. Bob then creates an AES-CBC cipher with the symmetric key **k** and **iv** received to decrypt the ciphertext **Em** and get the message **m**:



```
# create an AES cipher in CBC mode with received iv and the symmetric key decrypted
decipher = AES.new(sym_key, AES.MODE_CBC, b64decode(iv_bytes))
# decrypt and unpad the message using the symmetric key
rcvd_msg_dec = unpad(decipher.decrypt(cm_bytes), AES.block_size).decode()
return rcvd_msg_dec
```

Simply type `python hybrid.py` to run, try different messages to send in main():



```
# Alice has a message for Bob.
msg = "Hi Bob, it's Alice.\nWe are using hybrid encryption to communicate."
```