Lab 5: Diffie-Hellman and Public Key

Demo: http://youtu.be/3n2TMpHqE18

1 Diffie-Hellman

No	Description	Result
1	Bob and Alice have agreed on the values: g=2,879, N= 9,929 Bob Select b=6, Alice selects a=9	Now calculate (using the Kali calculator): Bob's B value (g^b mod N): Alice's A value (g^a mod N):
2	Now they exchange the values. Next calculate the shared key:	Bob's value (A^b mod N): Alice's value (B^a mod N): Do they match? [Yes] [No]
3	If you are in the lab, select someone to share a value with. Next agree on two numbers (g and N). You should pick a random number, and so should they. Do not tell them what your random number is. Next calculate your public value, and get them to do the same. Next exchange values.	Numbers for g and N: Your private value: Your public value: The public value you received: Shared key: Do they match: [Yes] [No]

2 Public Key

No	Description	Result
1	With RSA, we have a public modulus (and which is N=p.q, and where p and q are prime numbers). To create this, we need to generate a key pair with: `openssl genrsa -out private.pem 1024` This file contains both the public and the private key.	What is the type of public key method used: How long is the default key: How long are the prime numbers that are used to generate the public key?
2	Use following command to view the output file: `cat private.pem`	What can be observed at the start and end of the file:
3	Next we view the RSA key pair: `openssl rsa -in private.pem -text -noout` You should now see the public exponent (e), the private exponent (d), the two prime numbers (p and q), and the public modulus (N).	Which number format is used to display the information on the attributes: Which are the elements of the key shown: Which are the elements of the public key? Which are the elements of the private key? What does the –noout option do?
4	Let’s now secure the encrypted key with 3-DES: `openssl rsa -in private.pem -des3 -out key3des.pem`	

	You should NEVER share your private key.	
5	Next, we will export the public key: `openssl rsa -in private.pem -out public.pem -outform PEM -pubout`	View the output public key. What does the header and footer of the file identify? Is the public key smaller in size than the private key? [Yes/No]
6	Now we will encrypt with our public key: `openssl rsautl -encrypt -inkey public.pem -pubin -in myfile.txt -out file.bin`	
7	And then decrypt with our private key: `openssl rsautl -decrypt -inkey private.pem -in file.bin -out decrypted.txt`	What are the contents of decrypted.txt:
8	If you are working in the lab, now give your public key to your neighbour, and get them to encrypt a secret message for you.	Did you manage to decrypt their message? [Yes][No]

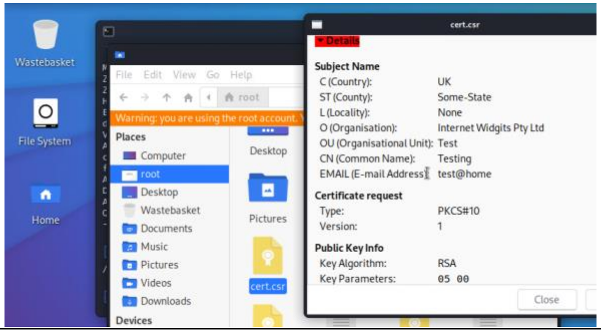
4 Storing keys

We have stored our keys on a key ring file (PEM). Normally we would use a digital certificate to distribute our public key. In this part of the tutorial we will create a crt digital certificate file.

No	Description	Result
1	Next create the crt file with the following:	View the CRT file by double clicking on it from the File Explorer.

<table>
<tr>
<td></td>
<td>

```
openssl req -new -key private.pem -out cert.csr
```

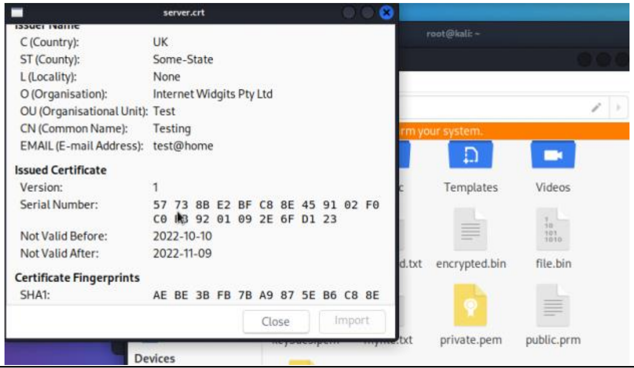


</td>
<td>
Using Google to search, what is PKCS#10, and which is it used for?

What is the type of public key method used:

What is the public key size (in bits):
[512][1024][2048]
</td>
</tr>
<tr>
<td></td>
<td>
We can now take the code signing request, and create a certificate. For this we sign the certificate with a private key, in order to validate it:

```
openssl x509 -req -in cert.csr -signkey private.pem -out server.crt
```



</td>
<td>
From the File System, click on the newly created certificate file (server.crt) and determine:

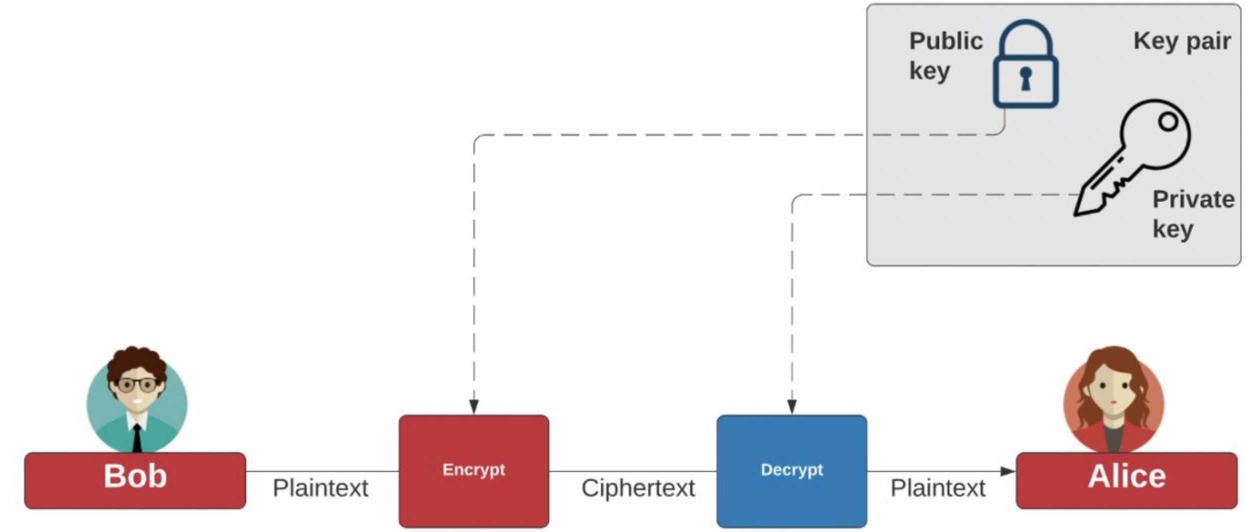
The size of the public key (in bits):
[512][1024][2048]

The public key encryption method:

Which is the hashing method that has been signed to sign the certificate: [MD5][SHA-1][SHA-256]
</td>
</tr>
</table>

5 AWS: Public Key Encryption

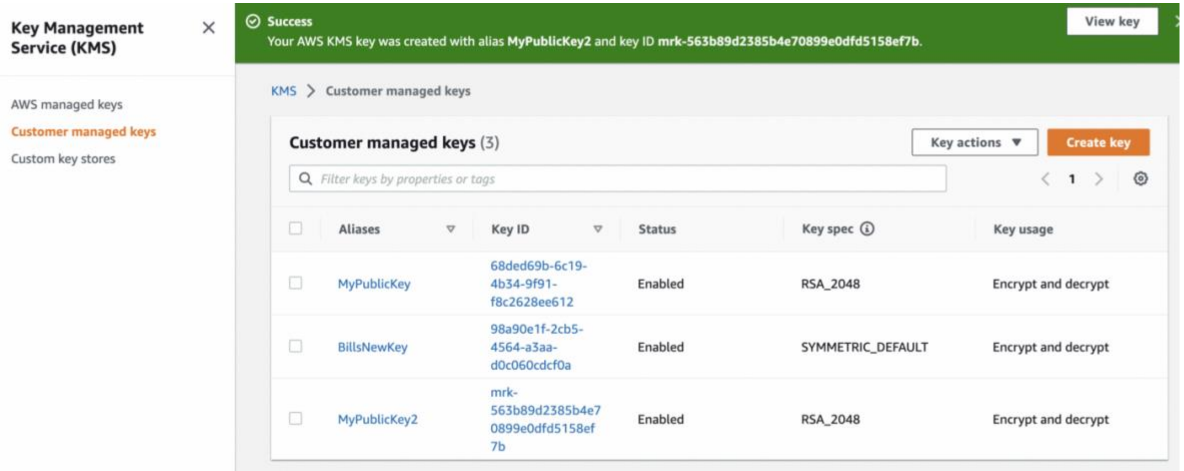
In the following figure, Bob uses Alice's public key to encrypt data, and which creates ciphertext. Alice then decrypts this ciphertext with her private key:



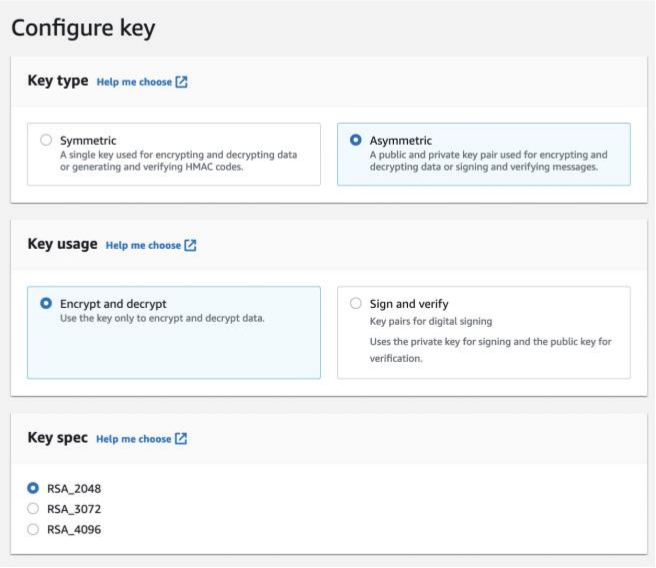
If we use asymmetric keys, we typically just have the choice of using RSA to encrypt and decrypt data. This is because elliptic curve cryptography does not naturally support encryption and decryption, and we must use hybrid methods (such as with ECIES).

Creating an RSA key pair in AWS

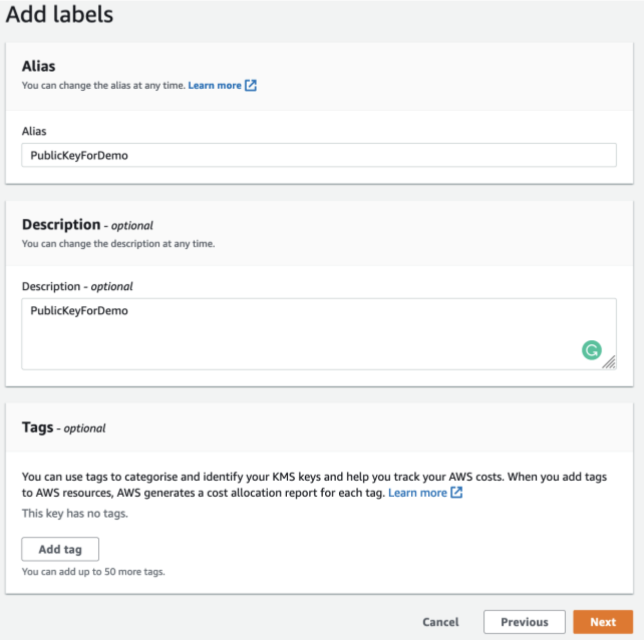
Now, let's create an RSA key pair for encrypting a file. Our keys are contained in the KMS:



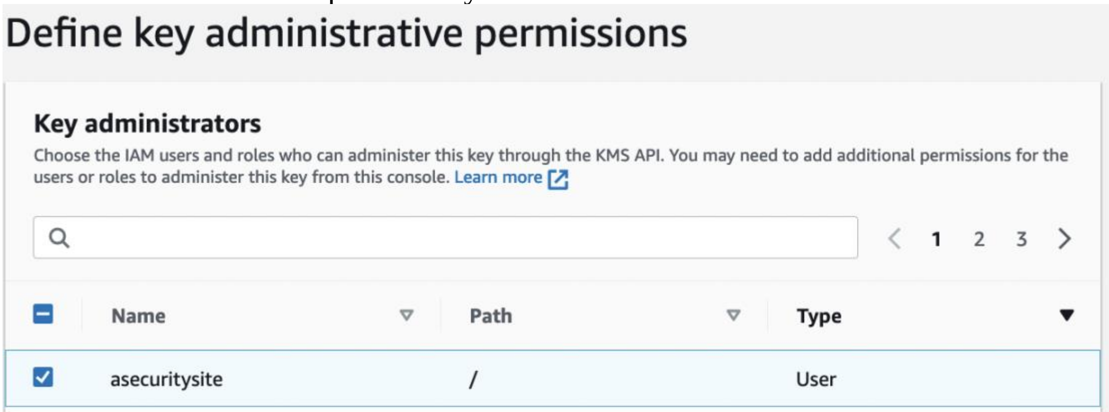
Initially, we can create a Customer-managed key pair with:



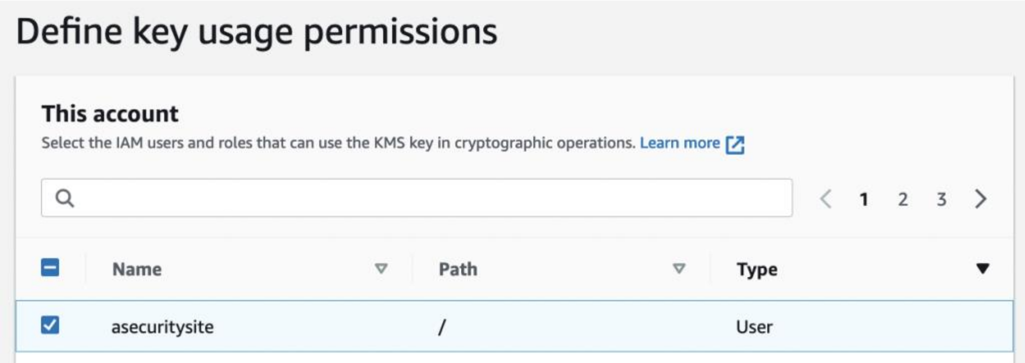
The options are 2K, 3K or 4K RSA key pairs. Next, we can give the key an alias:



Then define the ownership of the keys:



And finally the permissions:

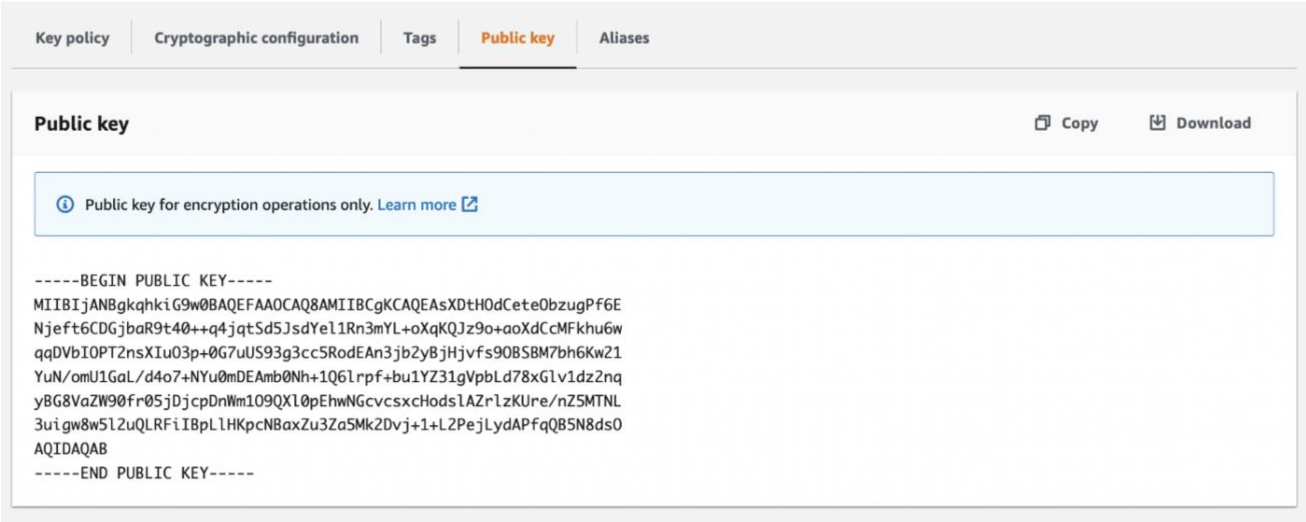


The policy is then:

```
{
    "Id": "key-consolepolicy-3",
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "Enable IAM User Permissions",
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::222222:root"
            },
            "Action": "kms:*",
            "Resource": "*"
        },
        {
            "Sid": "Allow access for Key Administrators",
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::222222:user/asecuritysite"
            },
            "Action": [
                "kms:Create*",
                "kms:Describe*",
                "kms:Enable*",
                "kms:List*",
                "kms:Put*",
```

```
                "kms:Update*",
                "kms:Revoke*",
                "kms:Disable*",
                "kms:Get*",
                "kms:Delete*",
                "kms:TagResource",
                "kms:UntagResource",
                "kms:ScheduleKeyDeletion",
                "kms:CancelKeyDeletion"
            ],
            "Resource": "*"
        },
        {
            "Sid": "Allow use of the key",
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::222222:user/asecuritysite"
            },
            "Action": [
                "kms:Encrypt",
                "kms:Decrypt",
                "kms:ReEncrypt*",
                "kms:DescribeKey",
                "kms:GetPublicKey"
            ],
            "Resource": "*"
        },
        {
            "Sid": "Allow attachment of persistent resources",
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::222222:user/asecuritysite"
            },
            "Action": [
                "kms:CreateGrant",
                "kms:ListGrants",
                "kms:RevokeGrant"
            ],
            "Resource": "*",
            "Condition": {
                "Bool": {
                    "kms:GrantIsForAWSResource": "true"
                }
            }
        }
    ]
}
```

Once created, we cannot access the private key, but will be able to view the public key:

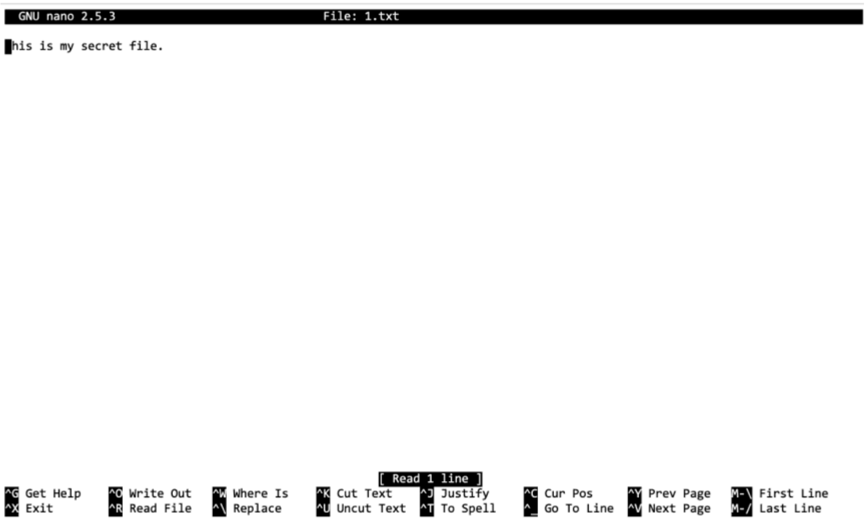


We can download this from the console, or from the command prompt:

```
% aws kms get-public-key --key-id alias/PublicKeyForDemo
{
    "KeyId": "arn:aws:kms:us-east-1:103269750866:key/de30e8e6-c753-4a2c-881a-53c761242644",
    "PublicKey":
"MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAsXDtHOdCeteObzugPf6ENjeft6CDGjbaR9t40++q4jqtSd5JsdYel1Rn3mYL+oXqKQJz9o
+aoXdCcMFkhu6wqqDVbIOPT2nsXIuO3p+0G7uUS93g3cc5RodEAn3jb2yBjHjvfs9OBSBM7bh6Kw21YuN/omU1GaL/d4o7+NYu0mDEAmb0Nh+1Q6lrp
f+bu1YZ31gVpbLd78xGlv1dz2nqyBG8VaZW90fr05jDjcpDnWm1O9QXl0pEhwNGcvcsxcHodslAZrlzKUre/nZ5MTNL3uigw8w5l2uQLRFiIBpLlHKp
cNBaxZu3Za5Mk2Dvj+1+L2PejLydAPfqQB5N8dsOAQIDAQAB",
    "CustomerMasterKeySpec": "RSA_2048",
    "KeySpec": "RSA_2048",
    "KeyUsage": "ENCRYPT_DECRYPT",
    "EncryptionAlgorithms": [
        "RSAES_OAEP_SHA_1",
        "RSAES_OAEP_SHA_256"
    ]
}
```

Encrypting with the public key

We can now create a file (1.txt):



And now encrypt using RSA with OAEP padding (RSAES_OAEP_SHA_1):

```
$ aws kms encrypt  --key-id alias/PublicKeyForDemo   --plaintext fileb://1.txt  --query CiphertextBlob --output
text --encryption-algorithm RSAES_OAEP_SHA_1 > 1.out
This will create a Base64 output of the encrypted file (1.out). We can list the file with:
% cat 1.out
```

```
nORNC8PQotPOpf7R1XlCaz8pQKEn5k6r3VOvLZk9ipzl7mGwV25HVqDc/ocK58eV/3u8IQVZDK81UPxk7D1BSc5LN5lvtxnIx8G7TfePxTDuu2+EM5z
avvU2S/2ZS+DOV2yHthHfNRKSDLB8a9oMzKBNcsfZBLGZEeZxEs/Rt5T7NdwWXnQsXbrgBJnvbfnNTzgyY4lPLjNqS4DPjA4UVI/3ICUjsEdKNvOv3X
ebBFvRaJ1a3flBJM5Bxo73gJSidwEZgTPSvGVdA5KOxoDuFh6gPmr/ztRirrrmkjF6zbdWlRfaNb9pLipvZz4KyDUkkKHOv2iYb+zAWzemuZ47sw==
```


This can be transmitted or stored. But, if we want to decrypt this, we need to convert the Base64 encoded data into binary:

```
$ base64 -i 1.out  --decode > 1.enc
```

Now, if we list 1.enc we see that it has binary data:

```
$ cat 1.enc
M
TYyBk?)@@'NS-=aWnGV
Ū{!Y
5Pd=AIK7oMOo36ي KKWl5
|k
Mrqv̌5^t,]mO82cO.3jKȯT %#GJ6w[hZA$AƎw3ɵe]ƀ>jb1zᵰV5i.*og>
5
```

Decrypting with the private key

Now to decrypt the file (1.enc) with the associated private key. For this, we use:

```
$ aws kms decrypt --key-id alias/PublicKeyForDemo --output text --query Plaintext --ciphertext-blob fileb://1.enc -
-encryption-algorithm RSAES_OAEP_SHA_1 > 2.out
This produces an output file of 2.out. Again, this is in a Base64 format:
$ cat 2.out
VGhpcyBpcyBteSBzZWNyZXQgZmlsZS4K
so we need to decode this with:
$ base64 -i 2.out  --decode
This is my secret file.
```

And, that's it. Note that the two main encryption methods we can use (with padding) are OEAP SHA-1 and OAEP SHA-256:

Using Python

We can use the same type of approach with Python. In the following case we use boto3, select an RSA key pair, and add the option of EncryptionAlgorithm='RSAES_OAEP_SHA_1' for the encryption and decryption:

```
import base64
import binascii
```

```
import boto3

AWS_REGION = 'us-east-1'

def enable_kms_key(key_ID):
    try:
        response = kms_client.enable_key(KeyId=key_ID)

    except ClientError:
        print('KMS Key not working')
        raise
    else:
        return response


def encrypt(secret, alias):
    try:
        ciphertext = kms_client.encrypt(KeyId=alias,EncryptionAlgorithm='RSAES_OAEP_SHA_1',Plaintext=bytes(secret,
encoding='utf8'),
        )
    except ClientError:
        print('Problem with encryption.')
        raise
    else:
        return base64.b64encode(ciphertext["CiphertextBlob"])


def decrypt(ciphertext, alias):
    try:
        plain_text =
kms_client.decrypt(KeyId=alias,EncryptionAlgorithm='RSAES_OAEP_SHA_1',CiphertextBlob=bytes(base64.b64decode(ciphert
ext)))
    except ClientError:
        print('Problem with decryption.')
        raise
    else:
        return plain_text['Plaintext']

kms_client = boto3.client("kms", region_name=AWS_REGION)

KEY_ID = '68ded69b-6c19-4b34-9f91-f8c2628ee612'
kms = enable_kms_key(KEY_ID)
print(f'Public Key KMS ID {KEY_ID} ')
msg='Hello'
print(f"Plaintext: {msg}")
```

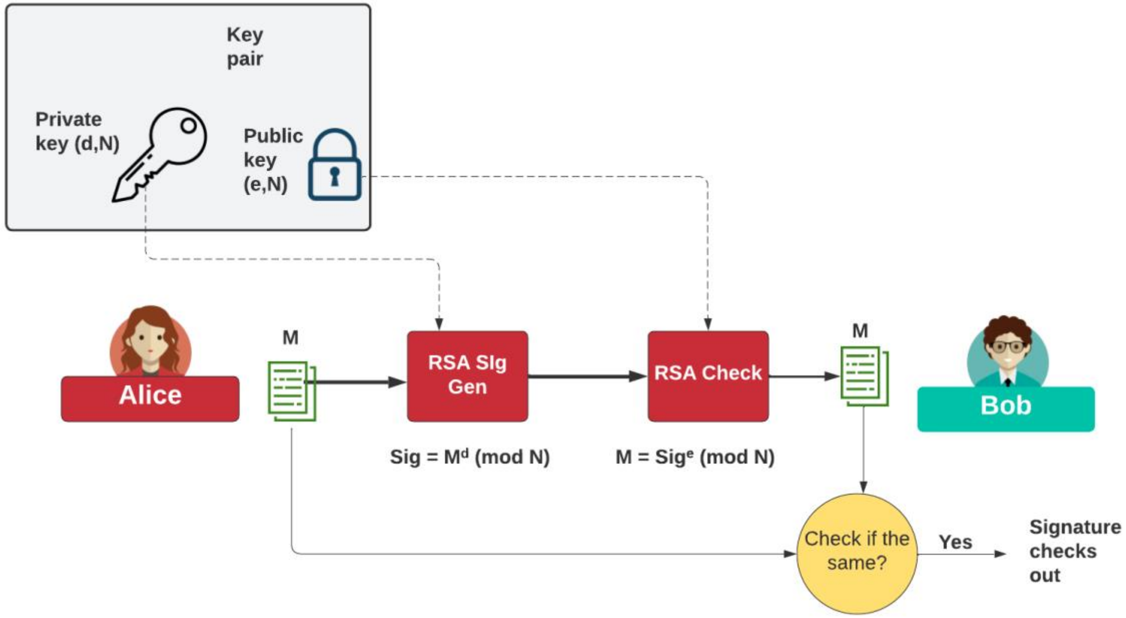
```
cipher=encrypt(msg,KEY_ID)
print(f"Cipher {cipher}")
plaintext=decrypt(cipher,KEY_ID)
print(f"Plain: {plaintext.decode()}")
```

A sample run gives:

```
KMS key ID 68ded69b-6c19-4b34-9f91-f8c2628ee612
Plaintext: Hello
Cipher
b'SvUOFgRLjpekJn1ZDuivW7YP3mCz3dCGwiWzaekrmcKhDyQbAh7wkBlr0ShC5xjJyC+jJ/0SdcXlKkbzWe8W/EfmKgo8zGcHsiil2F1d6fT9veGxO
75ySWz9uwVuoqnsJ0Z32dJG/7nlrGECNU9z984r2cLwiIidgKtqKm2bo48EguVUrU/GuNntxOV0u88r7GShpn6oZV3NPaPOhGEBTpCMGq8nXbv81H6f
MWsG92kbVW8PcOqM7cSw0z+XSaj/ndiKzD3yostib+drVtLPOJJ/idBXtOnKPMPEyiKAhMFUxYn+qk104egf5xn6Swh9nU1sogP4Xg0yBT6TdWQACg=
='
Plain: Hello
```

AWS Digital Signing

With digital signing, we often use RSA. With this, Alices uses her private key (d,N) to encrypt the message and produce a signature (sig). This is then passed to Bob and who takes the signature and Bob's public key (e,N), and then decrypts to determine the message. If the message decrypted is the same of the original message, the signature is valid. Overall we create a public key (e,N) and a private key (d,N). N is known as the public modulus, and has, for security reasons, at least, 2048 bits. e is the public exponent (and typically a value of 65,537) and d is the private exponent. In the following, we create a 2K RSA key pair with:



2 Creating an RSA key pair

In AWS, we use the KMS (Key Management Service) and which integrate a HSM (Hardware Security Module) to create and process with our keys. Within the KMS, we can create and delete keys, along with encrypting and digital signing. It supports both ECDSA and RSA signing. For padding, KMS supports PKCS1 or PSS, and for hashing within the RSA signature, we can either have SHA-256, SHA-384 or SHA-512. In AWS, we can create a key pair with the "aws kms create-key" command:

```
$ aws kms create-key --customer-master-key-spec RSA_2048 --key-usage SIGN_VERIFY --description "My RSA Key Pair"
{
    "KeyMetadata": {
        "AWSAccountId": "960372818084",
        "KeyId": "6545fae6-74d5-40ad-a5a7-cc65a885353d",
        "Arn": "arn:aws:kms:us-east-1:960372818084:key/6545fae6-74d5-40ad-a5a7-cc65a885353d",
        "CreationDate": "2022-12-02T20:55:10.420000-08:00",
        "Enabled": true,
        "Description": "My RSA Key Pair",
```

```
        "KeyUsage": "SIGN_VERIFY",
        "KeyState": "Enabled",
        "Origin": "AWS_KMS",
        "KeyManager": "CUSTOMER",
        "CustomerMasterKeySpec": "RSA_2048",
        "SigningAlgorithms": [
            "RSASSA_PKCS1_V1_5_SHA_256",
            "RSASSA_PKCS1_V1_5_SHA_384",
            "RSASSA_PKCS1_V1_5_SHA_512",
            "RSASSA_PSS_SHA_256",
            "RSASSA_PSS_SHA_384",
            "RSASSA_PSS_SHA_512"
        ]
    }
}
```

3 Creating a signature with AWS

In AWS, we use the HSM (Hardware Security Module) to create and process with our keys. It supports both ECDSA and RSA signing. For padding it supports PKCS1 or PSS, and for hashing we have either SHA-256, SHA-384 or SHA-512. In AWS, we can create a key pair with the "aws kms create-key" command. In the following, we create a 2K key pair with:

```
$ aws kms sign  --key-id 6545fae6-74d5-40ad-a5a7-cc65a885353d --message fileb://1.txt --signing-algorithm
RSASSA_PKCS1_V1_5_SHA_256  --query Signature  --output text  > 1.out
$ base64 -i 1.out -d > 1.sig
The file 1.sig is a binary file, but we can view the 1.out file (as it has a Base64 format):
$ cat 1.out
CG8vukZHOMvtzXas4jAiKCMgNSZHWbT2+HiLB++S2E9cxtmFH8E/jhy34NtQy/2y/ScehrcxcaVFaEyKyqUBsQiFk7QUTi04qm13sCnS0mtEBzpXMUV
WaS41XOM7pAa3j37swzKy+rWOYVgvvUvWL6Zyip6cR4tdvPvW8Bk/CUfq1jds6yLadpRndte+ilVZM6syyvP5d/U1rwpiAWu3BWLLaOZwzWeEd9f40s
1uv1Ag0hYZ3SxVYPQ8OCcqpgV9fjRwKg60uc1tPEPLwjlYSCQrh340E2SxKrMRWP4kbX0vaTKzFGK3fIOonwY8smQB89Fy2wEZhywQ2SCtpU1deA==
```

4 Verifying the signature with AWS

First we can verify the signature with AWS and using the "aws kms verify" command:

```
$ aws kms verify  --key-id 6545fae6-74d5-40ad-a5a7-cc65a885353d --message fileb://1.txt --signature fileb://1.sig -
-signing-algorithm RSASSA_PKCS1_V1_5_SHA_256
            {
            "KeyId": "arn:aws:kms:us-east-1:960372818084:key/6545fae6-74d5-40ad-a5a7-cc65a885353d",
            "SignatureValid": true,
```

```
    "SigningAlgorithm": "RSASSA_PKCS1_V1_5_SHA_256"
}
```

5 Getting the public key

Next we can export the public key from AWS with:

$ aws kms get-public-key --key-id 6545fae6-74d5-40ad-a5a7-cc65a885353d --output text --query PublicKey | base64 --decode > mycert.der

This exports into a binary format for the public key file. In OpenSSL, we can then take this binary file with the public key, and convert it into Base64:

```
$ openssl rsa -pubin -inform DER  -outform PEM -in mycert.der -pubout -out mycert.pem
writing RSA key
We can view the public key now with:
$ cat mycert.pem
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAw8BB3xtJPBgB4jrXCHdE
YhkZWG6nyYVT86C0sGZGSlUtkAgW7hlDN27foXgxLK9A1HlKUkhWaudYaVL42uEc
HihlmK0SnLZlk9j22/N82tGfUwpK9k9F3U/Cf4GoEz99lp97oDTnNTeWtUs0FvfB
iD31FHWhXiHzRU6XFwxh93SQEYBxe4B0j/XaUb5TW1OIhbFwwk/bCZpNvQfozyYP
kj6Yz6qRiNm0KsyBm5/TdWn7yj0D9YZ3kAhV8DtRZZIT4cvJ9yU741PZFiKM5y/5
UB8t89nO4c6yt6sweejQZANCTIhBqSmFtYvXnijofK7WcrW7Liudtvz9N58P6T5q
ZQIDAQAB
-----END PUBLIC KEY-----
```

6 Checking the signature with OpenSSL

Now, we can check the signature with OpenSSL:

```
$ openssl dgst -sha256  -verify mycert.pem -signature 1.sig 1.txt
Verified OK
```