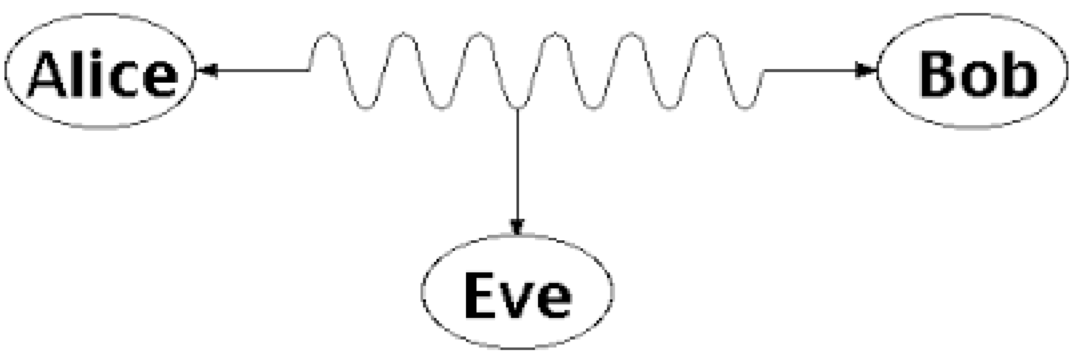
# Public Key Encryption – Week 2 Activities

## Introduction

Last week we learned about the Caesar and Vigenere ciphers. These ciphers allowed us to transform **plaintext** into **cipher-text**, so that people reading the text would not immediately understand the message. Although they were simple, they had flaws, which made them insecure.

To demonstrate why we wanted to use Encryption, we introduced Alice and Bob. Alice and Bob both wanted to engage in a secure conversation, so that a third party, Eve, could not understand what was being said in the conversation.



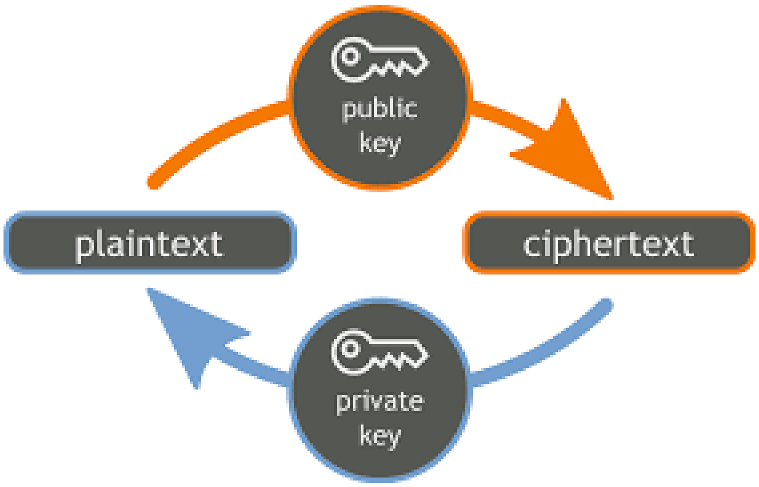

If Alice and Bob agree to use a password to encrypt their conversation, then Eve can still listen to their conversation if she hears Alice and Bob exchange the password to use. **How can Alice and Bob encrypt their conversation without having to agree upon a shared password or key?**

## Public Key Encryption

To solve this problem, let's look at public key encryption (PKE). In this scheme, we use two keys:

**Public Key**: This key is shared with the public and anyone can have it. With your public key, anyone can use it to encrypt a message that only you can decrypt.

**Private Key**: This is the key that only you and nobody else has a copy of. With this key, you are able to decrypt an encrypted message that was encrypted using your public key. This key is also used to sign messages, which we'll learn about later.

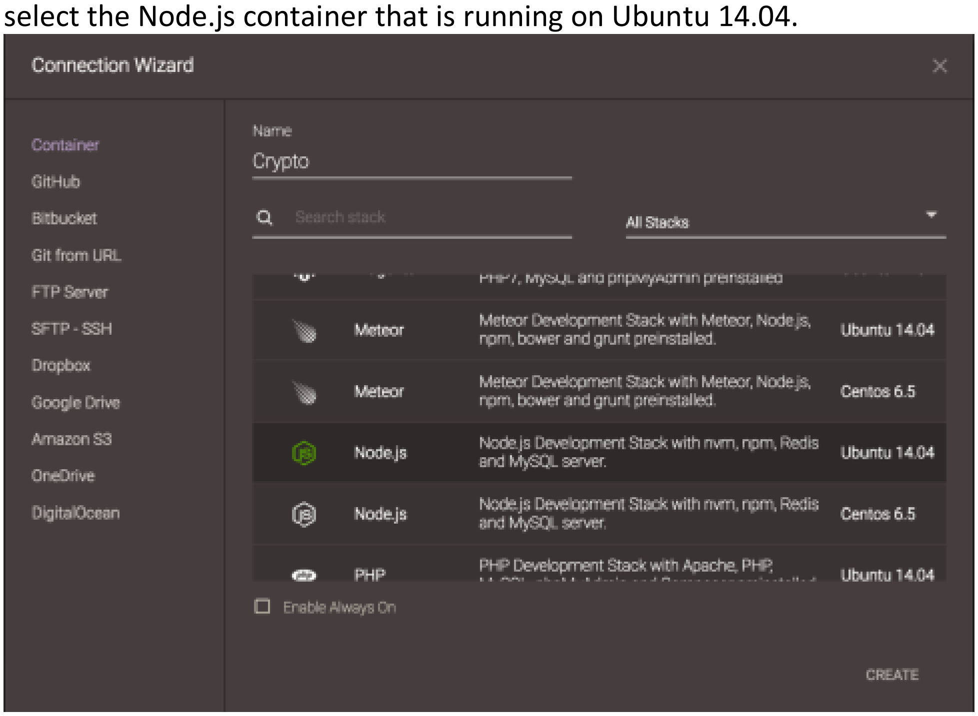



For this week's activities we are going to look at how we can encrypt messages using an implementation of Public Key Encryption called RSA (RSA are the first letters of the last names of the inventors).

## Activity 1: Setup CodeAnywhere Account

CodeAnywhere is a free online code editor, which provides an online terminal and code editor, so we don't need to install any software on our computers.

1. Navigate to [https://codeanywhere.com/](https://codeanywhere.com/) and click on the "Sign Up" button to register an account.
2. After successfully registering an account, click on the "Editor" button.
3. In the "Connection Wizard" window that pops-up, name your project "Crypto" and select the Node.js container that is running on Ubuntu 14.04.



4. Click "Create." If you receive an error about not being able to create an account until you verify your account, then check your email; it should contain a link that will allow you to verify your account.
5. Your project will now take a minute to create itself.

## Activity 2: Setup Node.js

1. Using the CodeAnywhere terminal window you can install a Node.js library for performing encryption and decryption with public and private keys.

```
npm install –g node-rsa
```

2. Run the following command to be able to access any Node.js libraries that you install.

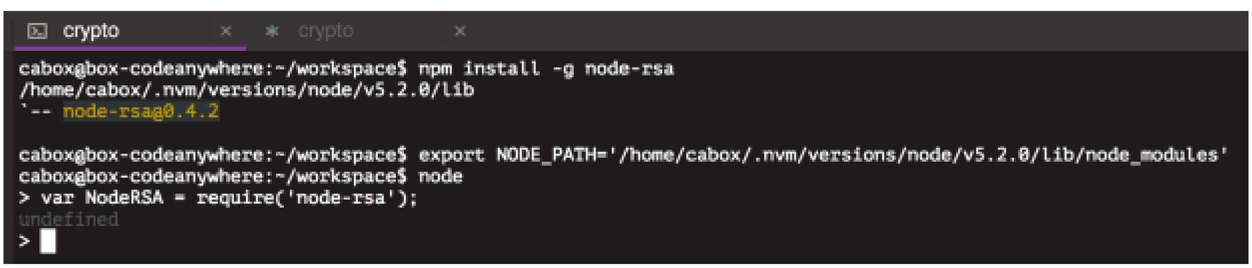
```
export NODE_PATH='/home/cabox/.nvm/versions/node/v5.2.0/lib/node_modules'
```

3. Start Node.js using the following command

```
node
```

4. Verify you have NodeJS setup by running the following command. This will also make the library you installed accessible.

```
var NodeRSA = require('node-rsa');
```




## Activity 3: Encrypt and Decrypt a Message in Node.js

1. The following code will generate a public and private key and save it to a variable, so that you have access to it. The 512 means that it's 512 bits long.

```
var key = new NodeRSA({b: 512});
```

2. To view the public and private key that you generated, use the following code to print them to the screen. It should look like a bunch of numbers and letters.

```
var publicKey = key.exportKey('public');
console.log(publicKey);

var privateKey = key.exportKey('private');
console.log(privateKey);
```

3. Using the key variable, it will encrypt messages with your public key and decrypt messages with your private key. The following code will allow you to encrypt a super secret message:

```
var plainText = "Super secret message";

var encrypted = key.encrypt(plainText, 'base64');
console.log(encrypted);
```

4. The encrypted message will look like a bunch of random numbers of letters, so it should be nearly impossible to figure out what the original message was at this point. To decrypt the message with your private key, use the following code:

```
var decrypted = key.decrypt(encrypted, 'utf8');
console.log(decrypted);
```

5. You should see your original message printed back!
6. You can now exit Node.Js by pressing "CTRL-C" twice.

## Activity 4: Encrypt and Decrypt messages with OpenSSL (Advanced)

Alice and Bob want to communicate, but they know that Eve is listening to their conversation. They both agree to use Public Key Encryption to communicate. Using the general-purpose encryption utility `openssl` in the terminal, help Alice and Bob exchange messages.

1. Using the `wget` command, download Alice's public key, so that Bob can write a message to her. Note that you do not need to type in the dollar sign ($) when typing in these commands.

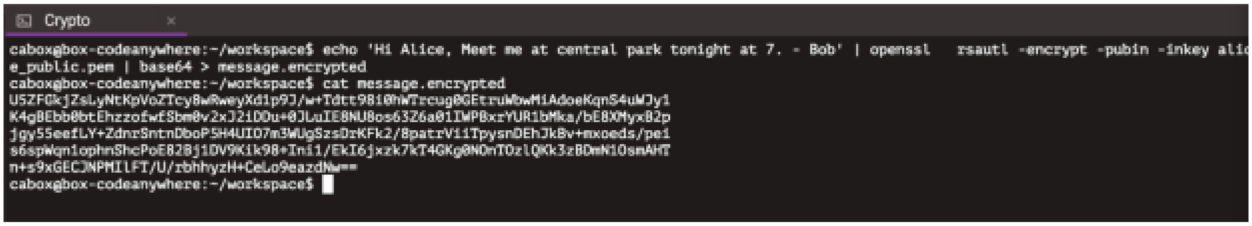
```
$ wget https://raw.githubusercontent.com/strategicpause/coderdojo-crypto/master/alice_public.pem

$ cat alice_public.pem
```

2. Using openssl, help Bob encrypt his message to Alice. The encrypted message will be stored in the file *message.encrypted*. You can use the cat command to view the content of *message.encrypted*.

```
$ echo 'Hi Alice, Meet me at central park tonight at 7. - Bob' | openssl rsautl -encrypt -pubin -inkey alice_public.pem | base64 > message.encrypted

$ cat message.encrypted
```




3. Use the wget command to download Alice's private key, so she can decrypt Bob's message.

```
$ wget https://raw.githubusercontent.com/strategicpause/coderdojo-crypto/master/alice_private.pem

$ cat alice_private.pem
```

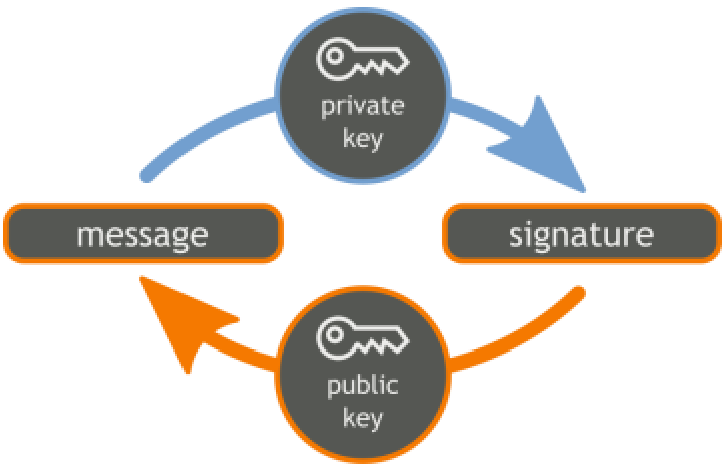
4. As Alice, you can decrypt Bob's message using your private key and the following commands. After running these commands, you should be able to see Bob's message.

```
$ cat message.encrypted | base64 --decode | openssl rsautl –inkey alice_private.pem –decrypt > message.txt

$ cat message.txt
```

## Activity 5: Verify Bob's Message (Advanced)

Since anyone can download your public key and use it to send you a message, you have no way of knowing if the person who said they wrote the message actually wrote it! For example, Eve could pretend to be Bob and send Alice a message using her public key. Digital signatures allow someone to use their private key to let others know that they did indeed write the message. This works because they should be the only one who has a copy of their private key. With a signature, you can determine if Bob did write the message and if the message was tampered with. Help Bob create a digital signature, so that Alice can verify that he actually did send her the message.




1. Using the wget command, download Bob's private key, so you create a signature for your message.

```
$ wget https://raw.githubusercontent.com/strategicpause/coderdojo-crypto/master/bob_private.pem

$ cat bob_private.pem
```

2. Create a signature for the file using the following command. This will save to the signature for *message.txt* to the file *message.txt.sig*.

```
$ openssl dgst -sha256 -sign bob_private.pem message.txt > message.txt.sig
```

3. Download Bob's public key, which Alice can use to verify that Bob signed the message.

```
$ wget https://raw.githubusercontent.com/strategicpause/coderdojo-crypto/master/bob_public.pem

$ cat bob_public.pem
```

4. Verify the signature, stored in *message.txt.sig*, using the following command. You should see "Verified OK" since the message was written by Bob and was not modified.

```
$ openssl dgst -sha256 -verify bob_public.pem -signature message.txt.sig message.txt
```

5. What happens when you modify the message? Try editing the message and verify the message again. What happens?

## Resources

- http://keyase.io - Manage your public and private keys online and share with friends and family.
- https://medium.com/@vrypan/explaining-public-key-cryptography-to-non-geeks-f0994b3c2d5 - Public Key Cryptography Explained
- https://rietta.com/blog/2012/01/27/openssl-generating-rsa-key-from-command/ - Generating RSA Keys