Construindo o próprio Onion Routing

1. Um pouco de história

Este roteiro guiado é orientado à programação do próprio sistema de Onion Routing, base do TOR. Há um outro roteiro guiado já desenvolvido pelo professor Marcos Simplício que ensina a utilizar o navegador TOR bem como a como compreender o seu funcionamento. Este roteiro se diferencia de seu roteiro ao se comprometer em realizar a programação deste sistema de confidencialidade. A linguagem aqui utilizada é python, mas ao se compreender a sua implementação, não haverá grandes dificuldades em se implementar em outra linguagem.
A tecnologia de Onion Routing foi desenvolvida pela marinha norte americana tendo como exposição no paper de divulgação de Goldschlag, Reed e Syverson como uma tecnologia de comunicação em que tanto remetente e destinatário (e por consequência, também um interceptador), não fossem capazes de descobrir as identidades um dos outros ao se utilizar de um sistema de intermediação dos pacotes. Desde que sua pesquisa foi desenvolvida, a tecnologia foi continuada na forma do TOR project, The Onion Routing. Este projeto ajudou a simplificar o seu uso para amplo público ao empacota-lo na forma de navegador.
A base desta tecnologia, o roteamento de cebolas, é o que será implementado ao longo deste roteiro.

2. Relevância de se saber implementar

Os conhecimentos envolvidos nesta implementação passam por conceitos de criptografia simétrica e assimétrica para providenciar o serviço de confidencialidade não apenas da mensagem, mas também da identidade das pontas da comunicação, o que faz com que este roteiro seja um incremento aos estudos de segurança da informação.
Além disso, em posse deste conhecimento de implementação, o leitor será capaz de acompanhar e contribuir com o projeto de Onion Routing no github: https://github.com/TheTorProject.
Finalmente, mesmo que contribuição a software livre não seja a prática buscada pelo leitor, ter posse deste conhecimento pode contribuir com futuras situações em que seja importante para o trabalho implementar um sistema parecido ou melhorado de confidencialidade.

3. Implementação das funções auxiliares de criptografia

Para começar, você pode se utilizar de seu ambiente python ou um notebook para seguir este guia. Antes de tudo, confira se você possui algumas bibliotecas de criptografia do python instalados pelo seu gerenciador de pacotes:

```
!pip install cryptography pycrypto pycryptodome
```

Para facilitar as operações nas seções futuras, vamos encapsular as funções de geração de chaves e operações de cifragem e decifragem em nossas próprias funções:

```
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives.asymmetric import rsa, padding
```

```
from cryptography.hazmat.primitives import hashes

PUBLIC_EXPONENT = 65537
KEY_SIZE = 512
ENCODE_TYPE = 'utf-8'

def generate_private_key():
  private_key = rsa.generate_private_key(
        public_exponent = PUBLIC_EXPONENT,
        key_size = KEY_SIZE,
        backend = default_backend())
  return private_key

def encrypt(public_key, message):
  encrypted_message = public_key.encrypt(message, padding.OAEP(
      mgf=padding.MGF1(algorithm=hashes.SHA1()),
      algorithm=hashes.SHA1(),
      label=None))
  return encrypted_message

def decrypt(crypt, private_key):
    msg = private_key.decrypt(
        crypt,
        padding.OAEP(
            mgf=padding.MGF1(algorithm=hashes.SHA1()),
                             algorithm=hashes.SHA1(),
                             label=None))
    return msg
```

Estas 3 funções se tratam de encapsulamentos de funções de criptografia assimétrica. Os parâmetros de expoente público e tamanho de chave pode ser facilmente modificado nas constantes parametrizadas no começo do código. Além disso, é possível alterar os parâmetros de padding para a cifragem e decifragem. Note que está sendo utilizando padding OAEP com algoritmo de hashing SHA-1. Caso queira alterar estes parâmetros, certifique-se que a mudança ocorreu tanto na função de cifragem quanto na de decifragem.

```
from Crypto.Cipher import AES
from Crypto.Random import get_random_bytes
from Crypto.Util.Padding import pad, unpad
from base64 import b64encode
import json

def generate_shared_secret():
```

```
  key = get_random_bytes(16)
  return key

def encrypt_AES_message(secret, message):
  cipher = AES.new(secret, AES.MODE_CBC)
  padded_message = pad(message, AES.block_size)
  ct_bytes = cipher.encrypt(padded_message)
  iv = cipher.iv
  return ct_bytes, iv

def decrypt_AES_message(secret, ciphertext, iv):
  cipher = AES.new(secret, AES.MODE_CBC, iv)
  pt = unpad(cipher.decrypt(ciphertext), AES.block_size)
  return pt
```

Estas 3 funções se tratam de encapsulamento para criptografia simétrica utilizando AES com modo CBC de criptografia com padding respeitando o tamanho do bloco igual ao tamanho da chave (16).

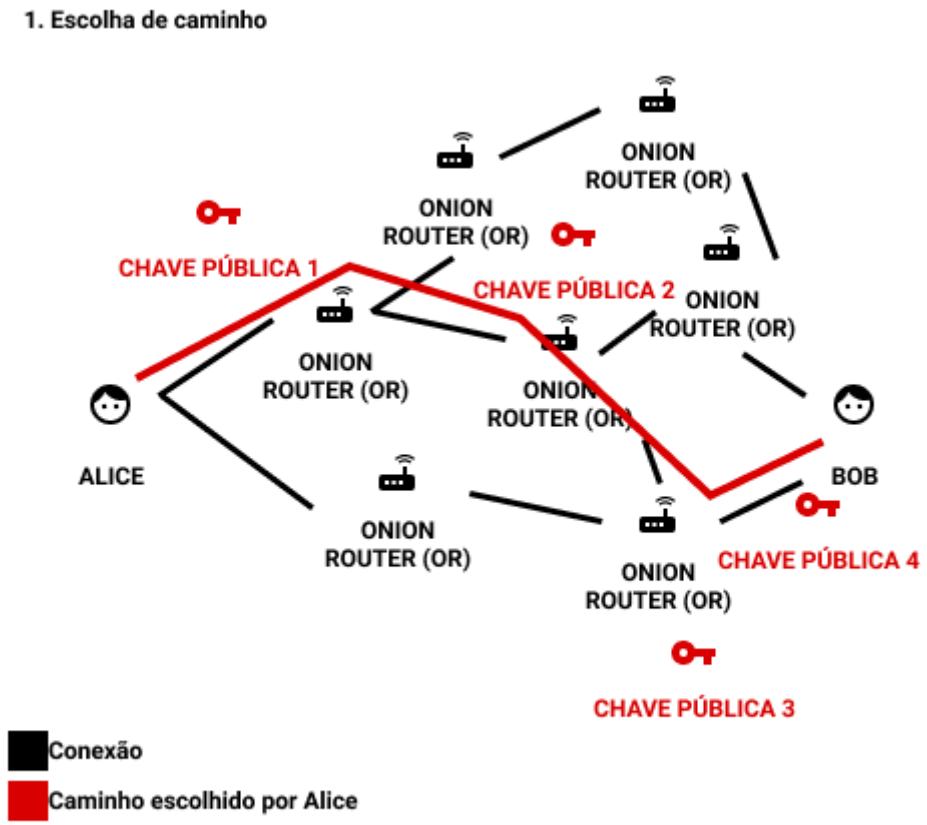
```
from cryptography.hazmat.primitives import serialization

def deserialize_public_key(public_key):
    deserialized_key =
public_key.public_bytes(encoding=serialization.Encoding.PEM,

format=serialization.PublicFormat.SubjectPublicKeyInfo)
    return deserialized_key

def load_public_key(deserialized_public_key):
    public_key =
serialization.load_pem_public_key(deserialized_public_key,
default_backend())
    return public_key
```

Finalmente, estas 2 duas funções se tratam de métodos de serialização e deserialização das chaves. Note que elas são usadas na forma de bytes e não string. Por isso, estamos utilizando estas duas funções para permitir o carregamento de uma chave na forma de string como bytes e vice e versa.

4. Implementação da definição de rota

Para os primeiros passos, vamos falar de um cenário hipotético em que Alice decide enviar uma mensagem para Bob e gostaria de garantir o seu anonimato e de Bob, bem como garantir o sigilo da mensagem se utilizando de uma Rede que se utiliza de Onion Routing. Ao consultar a lista de

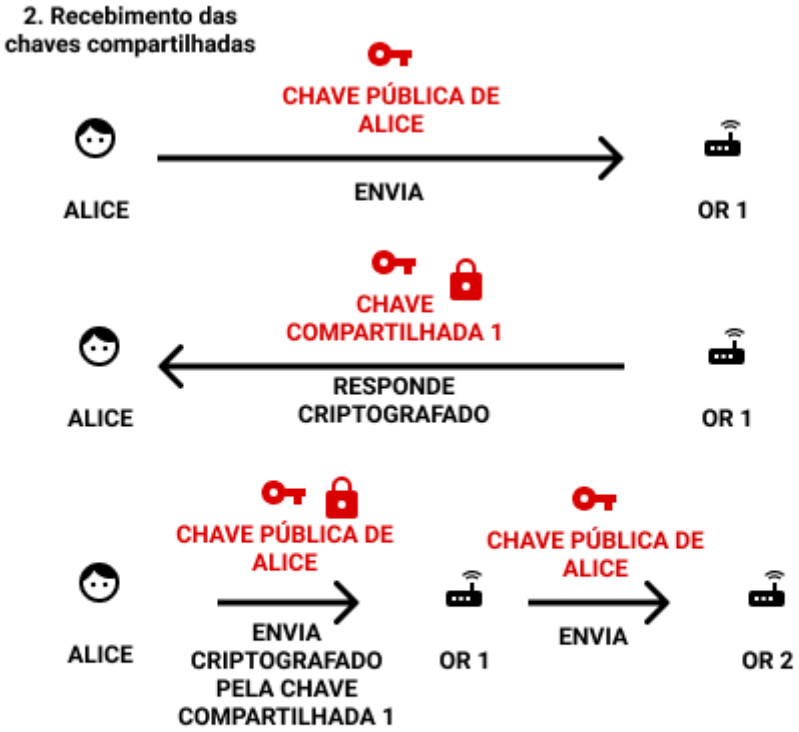
possíveis caminhos, ela decide por um caminho e adquire a chave pública de cada um dos nós intermediários. Vamos gerar 4 chaves privadas e então retornar a chave pública delas para Alice:



Vamos simular este cenário com as seguintes linhas de código:

```
private_keys = []
for i in range(4):
  private_keys.append(generate_private_key())

public_keys = []
for private_key in private_keys:
  public_keys.append(private_key.public_key())
```

O próximo passo é obter as chaves compartilhadas de cada nó.



Este procedimento acima ilustrado pode ser codificado passo a passo com as seguintes operações:

```
alice_private_key = generate_private_key()
alice_public_key = alice_private_key.public_key()

# Alice vai guardar suas chaves compartilhadas nesta lista
shared_secrets = []
# Alice envia sua chave pública para o Onion Router 1 pedindo por um
segredo compartilhado
OR1_shared_secret = generate_shared_secret()
crypted_OR1_ss = encrypt(alice_public_key, OR1_shared_secret)
# OR 1 envia a chave compartilhada para alice. Ela por sua vez
consegue descriptografar
shared_secrets.append(decrypt(crypted_OR1_ss, alice_private_key))

# Alice vai repetir o procedimento, mas cifrando a mensagem com o
segredo compartilhado que OR1 compartilhou com ela:
shared_secret_request_for_OR2 = json.dumps({
    "public_key" :
deserialize_public_key(alice_public_key).decode(ENCODE_TYPE),
```

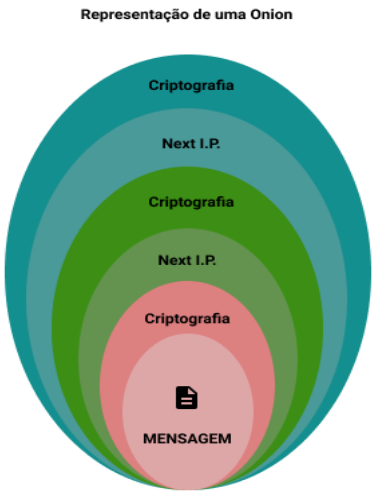
```
    "to" : 1
})

encrypted_request_OR2, iv_OR1 = encrypt_AES_message(shared_secrets[0],
shared_secret_request_for_OR2.encode(ENCODE_TYPE))

# Ao desempacotar a requisição, quando OR1 mandar para OR2, OR2 vai
interpretar como se fosse uma chamada iniciadora de comunicação
qualquer. Pode ser que o pedido estivesse vindo de uma agente como
Alice ou de um roteador.
decrypted_request_OR2 = decrypted_request_OR2 =
decrypt_AES_message(shared_secrets[0], encrypted_request_OR2, iv_OR1)
```

Assim, cada OR vai respondendo com uma chave de criptografia simétrica para Alice enquanto cada OR não sabe se quem enviou a mensagem para si foi um usuário inicial ou apenas mais um OR até chegar no Bob.

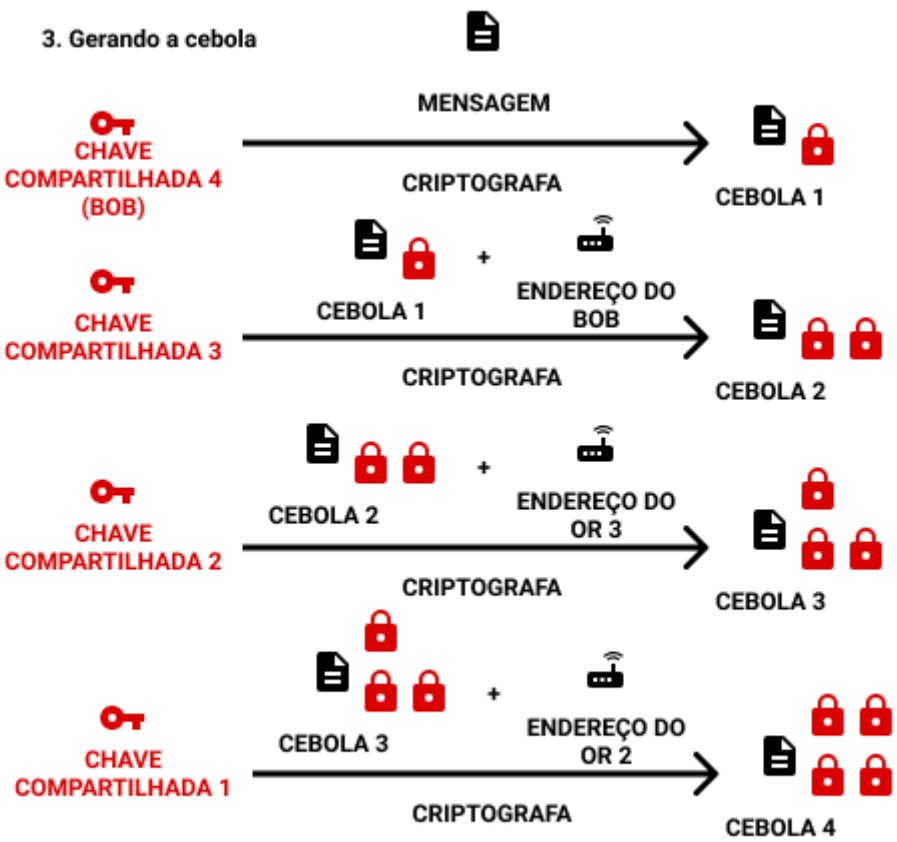
5. Implementação da construção de cebola

Agora que passamos pela etapa de Setup da comunicação em que Alice obtém as chaves compartilhadas para cada nó da comunicação, agora ela é capaz de montar uma cebola.

Cada camada de uma cebola consiste de uma cebola embarcada e um endereço para o próximo usuário, como a imagem a seguir:



A única exceção é o destinatário, pois a cebola que ele descriptografar vai se tratar do dado desejado propriamente. O processo de construção da cebola pode ser ilustrado na seguinte figura:



Para implementar este processo de empacotamento, podemos implementar a seguinte função auxiliar:

```
def add_onion_layer(onion, destiny, shared_secrets):
  new_message = json.dumps({
      "onion": str(onion),
      "destiny": destiny
  })
  new_onion, iv = encrypt_AES_message(shared_secrets,
new_message.encode(ENCODE_TYPE))
  return new_onion, iv
```

A sua função é cifrar uma mensagem junto com o endereço destino do próximo destinatário.
A seguir vamos continuar a nossa simulação com os passos em que a Alice vai ter 4 segredos compartilhados de sua rota definida no passo anterior. Então, ela constrói a cebola com a mensagem “oi bob” como exemplo para atravessar toda a rede até Bob:

```
shared_secrets = []

for i in range(4):
  shared_secrets.append(generate_shared_secret())
```

```
import base64

def encode(message):
  return base64.b64encode(message).decode(ENCODE_TYPE)

message = "oi bob"
onion_to_encrypt, iv = encrypt_AES_message(shared_secrets[3],
message.encode(ENCODE_TYPE))
onion_to_encrypt = json.dumps({
    "encrypted_onion": encode(onion_to_encrypt),
    "iv": encode(iv)
})
print('onion_to_encrypt',onion_to_encrypt)

for i in range(3):
  new_onion, iv = add_onion_layer(onion_to_encrypt, 3 - i,
shared_secrets[2-i])
  onion_to_encrypt = json.dumps({
    "encrypted_onion": encode(new_onion),
    "iv": encode(iv)
  })
  print('onion_to_encrypt',onion_to_encrypt)

print(new_onion)
```

Os prints foram adicionados para que você possa observar o conteúdo de cada camada da cebola. A seguir, há um exemplo desta operação:

```
onion_to_encrypt {"encrypted_onion": "tEQWBeNCUC58vK9RqL6yKQ==", "iv": "dBtC1Kj+10aglYANjNg0pQ=="}
onion_to_encrypt {"encrypted_onion": "X5SUaQt+dyyHNZ8BhFNQeISXboEWlZ3pX4clBoxLV4pfmorfVA/u3roQ5aoEO5Ob+FU/JzFp7R+OyD6GuFNG45OZqmEYMYquS2yx91WgqVM6Zh
onion_to_encrypt {"encrypted_onion": "kHES7UPUxvugSuay9gsugGUCynNulY6beHT9tycWlVhuMzgXhdTxt/BREOWFRlBUaEFJWHMZtG6+9El/mYJTrfhlIoTy0P1b/9TMOc57QoVSGU
onion_to_encrypt {"encrypted_onion": "i8q6b2VBKZy6MlgPq6xlWVjYaLjmcySJ56lFHYUSol83klsQ8ALbjUzDLiHtXvzdehRKHiY7/+TiYWVyX0auPsP7rN314ayqHRcJpn53dwoXOP
b"\x8b\xca\xbaoeA)\x9c\xba2X\x0f\xab\xaceYX\xd8h\xb8\xe6s$\x89\xe7\xa9E\x1d\x85\x12\xa2_7\x92[\x10\xf0\x02\xdb\x8dL\xc3.!\xed^\xfc\xddz\x14J\x1e&;\x
```

Note que a primeira cebola tem um tamanho nitidamente menor do que as seguintes. Isto é um fator de insegurança. Afinal, um atacante consegue inferir quanto mais próximo ele está do emissor da mensagem quanto menor for a cebola enviada. Para evitar esta vulnerabilidade é que se sugere a adição de padding. Com padding, todas as camadas da cebola teriam o mesmo tamanho, assim inviabilizando a análise de mensagens por tamanho. Endereçar este ponto na segurança pode ser um desafio para a sua implementação, mas não será abordado neste guia por questões de simplicidade e focar no escopo de cifragem das mensagens e definição de rota.

Agora, vamos finalizar o nosso caso de uso com uma função de descascar uma camada da cebola:

```
def peel_onion_layer(onion, iv, secret):
  data = decrypt_AES_message(secret, onion, iv)
  return data
```

```
def parse_json_from_encode(encoded_onion):
  json_acceptable_string = encoded_onion.replace("'onion'",
"\"onion\"")
  json_acceptable_string = json_acceptable_string.replace("'destiny'",
"\"destiny\"")
  return json_acceptable_string

def parse_json_from_encoded_encrypted_onion(encrypted_onion):
  json_acceptable_string =
encrypted_onion.replace("'encrypted_onion'", "\"encrypted_onion\"")
  json_acceptable_string = json_acceptable_string.replace("'iv'",
"\"iv\"")
  return json_acceptable_string

def extract_json_from_onion(onion, iv, shared_secret):
  peeled_onion = peel_onion_layer(onion, iv,
shared_secret).decode(ENCODE_TYPE)
  json_acceptable_string = parse_json_from_encode(peeled_onion)
  try:
    json_peeled_onion = json.loads(json_acceptable_string)
    return json_peeled_onion
  except:
    return json_acceptable_string

def decode(message):
  return base64.b64decode(message.encode(ENCODE_TYPE))

def decrypt_onion(onion, shared_secret):
  parsed_onion = parse_json_from_encoded_encrypted_onion(onion)
  parsed_onion_dict = json.loads(parsed_onion)
  encrypted_onion = decode(parsed_onion_dict['encrypted_onion'])
  iv = decode(parsed_onion_dict['iv'])
  json_peeled_onion = extract_json_from_onion(encrypted_onion, iv,
shared_secret)
  return json_peeled_onion
```

A partir destes métodos, podemos prosseguir com a nossa simulação. O primeiro nó a rotear a mensagem irá receber a cebola de Alice, irá descascá-la com a sua chave compartilhada com ela e descobrir o próximo nó para quem deve repassar:

```
new_onion_1 = extract_json_from_onion(new_onion, iv,
shared_secrets[0])
```

```
print('onion', new_onion_1['onion'])
print('destiny', new_onion_1['destiny'])

new_onion_2 = new_onion_1['onion']
```

Resultado:

```
onion {"encrypted_onion": "kHES7UPUxvugSuay9gsugGUCynNulY6beHT9tycW1VhuMz
destiny 1
```

Agora o OR 0 tem o endereço do próximo a quem deve entregar a cebola: o IP 1. Podemos repetir a nossa simulação para o próximo OR:

```
decrypted_onion2 = decrypt_onion(new_onion_2, shared_secrets[1])

print('onion', decrypted_onion2['onion'])
print('destiny', decrypted_onion2['destiny'])
```

Resultado:

```
onion {"encrypted_onion": "X5SUaQt+dyyHNZ8BhFNQeISXboEwlZ3pX4clBoxLV4pfmo
destiny 2
```

OR 1 conseguiu recuperar a segunda parte da informação e agora sabe que deve repassar a cebola para OR 2.

```
decrypted_onion3 = decrypt_onion(decrypted_onion2['onion'],
shared_secrets[2])

print('onion', decrypted_onion3['onion'])
print('destiny', decrypted_onion3['destiny'])
```

Resultado:

```
onion {"encrypted_onion": "tEQWBeNCUC58vK9RqL6yKQ==", "iv": "dBtC1Kj+10a
destiny 3
```

OR 2 conseguiu ter acesso à informação de destino. Agora ele entregará para o endereço de IP 3. Ao enviar a mesnagem para o IP 3, o Bob, será que Bob será capaz de ter acesso à mensagem original de Alice?

```
decrypted_onion4 = decrypt_onion(decrypted_onion3['onion'],
shared_secrets[3])

print(decrypted_onion4)
```

Resultado:

```
oi bob
```

Olá Alice ;). Missão cumprida.

6. Considerações finais

Este roteiro guiado de codificação abordou o escopo de definição de rota e cifragem da mensagem para oferecer o serviço de confidencialidade. Note que a solução apresentada admite melhorias com a adição de padding às mensagens para que as camadas da cebola tenham sempre o mesmo tamanho (MTU 1500 bytes por exemplo).
Esperamos que este roteiro tenha agregado e ajudado na compreensão do TOR bem como ajudado você a pensar com mais facilidade soluções de segurança da informação para contribuir com soluções Open Source.