

# SERVIDOR DE CAIXAS SEGURAS

UNIVERSIDADE DE AVEIRO

Rafael Santos, Paulo Vasconcelos



LABORATÓRIOS DE INFORMÁTICA

# SERVIDOR DE CAIXAS SEGURAS

DETI

UNIVERSIDADE DE AVEIRO

Rafael Santos, Paulo Vasconcelos  
(84951) r.c.santos@ua.pt, (84987) paulobvasconcelos@ua.pt

21 de ABRIL de 2017

## **Resumo**

Este relatório tem como principal objetivo enquadrar o trabalho de implementação da ligação entre o cliente e o servidor (*xcoa*).

# Conteúdo

<b>1</b>	<b>Introdução</b>	<b>1</b>
<b>2</b>	<b>Pesquisa</b>	<b>2</b>
<b>3</b>	<b>Metodologia</b>	<b>3</b>
3.1	Comandos Pre Main . . . . .	3
3.2	Funções . . . . .	4
3.2.1	MAIN . . . . .	5
3.2.2	LIST . . . . .	7
3.2.3	CREATE . . . . .	8
3.2.4	PUT . . . . .	10
3.2.5	GET . . . . .	11
<b>4</b>	<b>Conclusões</b>	<b>13</b>

# Lista de Figuras

3.1	Função <i>gen_key</i> . . . . .	3
3.2	Exceções . . . . .	4
3.3	Função <i>draw_menu</i> . . . . .	4
3.4	Função <i>main</i> . . . . .	6
3.5	Função <i>list_box</i> . . . . .	7
3.6	Função <i>json_list</i> . . . . .	7
3.7	Função <i>box_type_menu</i> . . . . .	8
3.8	Função <i>create_box_public</i> . . . . .	8
3.9	Função <i>create_box_private</i> . . . . .	9
3.10	Função <i>json_create_private</i> . . . . .	10
3.11	Função <i>json_create_public</i> . . . . .	10
3.12	Função <i>put_msg</i> . . . . .	11
3.13	Função <i>json_put</i> . . . . .	11
3.14	Função <i>get_box</i> . . . . .	11
3.15	Função <i>json_get</i> . . . . .	12

# Capítulo 1

## Introdução

Das últimas décadas a esta parte, "globalização" tem sido a palavra-chave. Este processo de aproximação de toda a população foi impulsionado pelo desenvolvimento de tecnologias que permitem uma comunicação rápida e eficaz a longas distancias, como é o caso da Internet. Com o passar do tempo, foi possível transmitir mais informação de maneira mais rápida, o que eventualmente levou ao surgimento de questões relativas à segurança e privacidade dos utilizadores destas tecnologias.

Foi com o intuito de conhecer as dificuldades que são apresentadas no estabelecimento de ligações a longas distancias que este trabalho foi realizado. Neste trabalho, foi realizada uma ligação ao server do (*xcoa*) onde foi implementado um programa de um modelo de caixas seguras, o qual será explicado mais adiante, que irá interagir com o programa desenvolvido.

Este documento está dividido em quatro capítulos. Depois desta introdução, no Capítulo 2 são apresentadas algumas noções básicas de criptografia., no Capítulo 3 é apresentada a metodologia seguida e as funções desenvolvidas juntamente com a utilização de cada uma, no Capítulo 4 são apresentadas as conclusões do trabalho aquando da aplicação do cliente desenvolvido. Finalmente, no Capítulo 4 são explicitadas as contribuições de cada um dos autores para o trabalho e as percentagens atribuídas a cada um conforme os resultados apresentados.

## Capítulo 2

# Pesquisa

Criptografia. Por definição é *o estudo dos princípios e técnicas pelas quais a informação pode ser transformada da sua forma original para outra, ilegível, de forma a que possa ser conhecida apenas pelo seu destinatário, o que torna difícil a sua leitura por alguém não autorizado..* Na prática, é o que permite haver confidencialidade na internet (e não só).

No mundo de hoje, são facilmente identificáveis os dois sistemas básicos de criptografia existentes: sistemas **simétricos** e **assimétricos**.

No sistema simétrico, a chave que realiza a encriptação é a mesma que realiza a descriptação pelo que a sua segurança é fulcral.

Num sistema assimétrico, cada utilizador possui duas chaves. Uma pública e outra privada. A chave pública é utilizada para encriptar todas as mensagens que o utilizador irá receber e a sua chave privada descriptará essas mesmas mensagens. A chave pública é dada a conhecer a todos os utilizadores para que seja possível o envio de mensagens à entidade portadora dessa mesma chave. A chave privada é apenas conhecida pelo próprio utilizador pois qualquer utilizador com acesso à mesma será capaz de descriptar as mensagens obtendo assim o acesso a informação potencialmente privada.

## Capítulo 3

# Metodologia

### 3.1 Comandos Pre Main

Antes de ser executada a função *main*, são executados vários comandos que permitem o bom funcionamento do código. O primeiro comando apresentado, *gen\_key*, verifica se existe um par de chaves *RSA* já gerado, e, caso não exista, gera um novo par de chaves que irá ser usado nas funções *CREATE* (na criação de caixas privadas) e *GET*.

```
def gen_key():
    key = RSA.generate(2048)
    a = open('publicKey.pem', 'w') # Making sure privateKey exists
    f = open('publicKey.pem', 'r')
    g = open('privateKey.pem', 'w')
    if len(f.read()) < 1:
        a.write(key.publickey().exportKey('PEM'))
        g.write(key.exportKey("PEM"))
        a.close()
        f.close()
        g.close()
```

Figura 3.1: Função *gen\_key*

Seguidamente, tem-se a criação dos *sockets* e a ligação ao *server*. Antes de se efetuar uma ligação definitiva ao mesmo, faz-se um teste de ligação ao *server* e, caso não reponda (por o utilizador não estar ligado à Internet ou o *server* estar em baixo), mostra uma mensagem de erro e seguidamente fecha o programa. Caso a ligação seja corretamente realizada, é apresentado um menu com as opções que o utilizador pode escolher através da introdução de texto pelo teclado.



```

gen_key() # Generates a pair of keys in case they don't exist
tcp_s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_addr = ("xcoa.av.it.pt", 8080)
try:
    tcp_s.connect(server_addr)
except:
    print "Couldn't establish a connection with the server. Either the server or you internet connection is down"
    print("Program will now close")
    exit()
main()

```

Figura 3.2: Exceções

## 3.2 Funções

Ir  agora ser apresentada uma descri  o de cada fun  o do programa.

```

def draw_menu(): # Draws a menu
    print '_____ '
    print '      MENU      '
    print ''
    print '      LIST '
    print '      CREATE '
    print '      PUT '
    print '      GET '
    print '      EXIT '
    print '_____ '

```

Figura 3.3: Fun  o *draw\_menu*

### 3.2.1 MAIN

Na função *Main*, é definida a estrutura geral do programa. Começa por definir se o programa vai funcionar apenas com base num menu, ou se cada função deve ser executada diretamente a partir do terminal. Como tal, a função *main* começa por ver se existem argumentos adicionais fornecidos ao programa. Caso existam, executa a função correspondente aos argumentos fornecidos. Caso sejam fornecidos argumentos inválidos, o programa apresenta uma mensagem de erro e encerra. Caso contrário, o programa executa normalmente.

Caso não sejam fornecidos argumentos adicionais ao programa, o programa mostra um menu e várias opções para o utilizador escolher. Caso o utilizador insira um comando inválido no programa, este mostra uma mensagem de erro e volta a mostrar o menu. O programa neste caso só termina se o utilizador inserir o comando *EXIT* ou se forçar o encerramento do programa.

```

def main():
    if len(sys.argv) > 1:
        if sys.argv[1] != "LIST" or "GET" or "PUT" or "GET":
            print "Invalid Arguments"
        else:
            if str(sys.argv[1].upper) == 'LIST':
                list_box()
            elif str(sys.argv[1].upper()) == 'CREATE':
                if len(sys.argv[2]) > 0:
                    if sys.argv[2] == "PUBLIC":
                        create_box_public(raw_input("Insert box name: "))
                    elif sys.argv[2] == 'PRIVATE':
                        create_box_private(raw_input("Insert box name: "))
                    else:
                        print 'Invalid Arguments'
            elif str(sys.argv[1].upper) == 'PUT':
                put_msg()
            elif str(sys.argv[1].upper) == 'GET':
                get_box()
    else:
        while True:
            draw_menu()
            option = str(raw_input().upper())
            if option == 'LIST':
                list_box()

            elif str(option) == 'CREATE':
                box_type_menu()
                command = str(raw_input("What box do you want to create?").upper())
                if command == 'PUBLIC':
                    create_box_public(raw_input("What will the box's name be?"))
                elif command == 'PRIVATE':
                    bname = raw_input("What will the box's name be?")
                    create_box_private(bname)
                else:
                    print "Invalid option"

            elif str(option) == 'PUT':
                put_msg()
            elif str(option) == 'GET':
                get_box()
            elif str(option) == 'EXIT':
                exit()

```

Figura 3.4: Função *main*

### 3.2.2 LIST

No módulo *list\_box*: é enviada para o *server* uma *string* de *JSON* terminada

```
def list_box():
    tcp_s.send(json_list())
    msg = tcp_s.recv(4096)
    if not msg.endswith("\r\n"):
        while not msg.endswith("\r\n"):
            msg = msg + tcp_s.recv(4096)
        dic = dict(json.loads(msg))
        if str(dic['code']) == 'OK':
            for i in range(0, len(dic['payload'])):
                print dic['payload'][i]['name']
        else:
            print 'There has been a problem. Please try again'
```

Figura 3.5: Função *list\_box*

em `\r\n` (isto é necessario para o *server* saber quando é que acabou o envio de informação) a qual é gerada através do módulo *json\_list*:

```
def json_list(): # Creating JSON String for Box Listing
    j_list = {'type': "LIST"} #
    return str(json.dumps(j_list)+'\r\n')
```

Figura 3.6: Função *json\_list*

Apos ser gerada a *string* de *JSON* , é enviada ao server pelo *socket*. Entretanto, o *socket* aguarda por uma resposta por parte do *server* acabada em `\r\n` apresentando depois uma listagem do nome de todas caixas que existem no *server* ao utilizador.

### 3.2.3 CREATE

Nesta função, vão ser criadas as caixas que irão depois ficar armazenadas no *server*. É de destacar que existem 2 tipos de caixas: caixas públicas e caixas privadas, sendo fornecido um menu que permite ao utilizador escolher o tipo de caixa que pretende criar.

```
def box_type_menu():                                     # Draws Another menu
    print '_____'
    print '      Box Type'
    print ' '
    print ' 1: PUBLIC'
    print ' 2: PRIVATE'
    print '_____'
```

Figura 3.7: Função *box\_type\_menu*

#### Caixas Públicas

As caixas públicas são criadas através do método *create\_box\_public* que pede um argumento *box\_name* que é pedido ao utilizador na execução da função *main*.

```
def create_box_public(box_name):
    tcp_s.send(json_create_public(box_name) + '\r\n')
    msg = tcp_s.recv(4096)
    if not msg.endswith("\r\n"):
        while not msg.endswith("\r\n"):
            msg = msg + tcp_s.recv(4096)
    dic = dict(json.loads(msg))
    if str(dic['code']) == 'OK':
        print 'Your box has been created successfully'
    else:
        print 'An error has occurred. Please try again later'
        print 'Details: ' + str(dic['content'])
```

Figura 3.8: Função *create\_box\_public*

É também definida uma string de *JSON* através do método *json\_create\_public(box\_name)*. Após isto, é enviada a string de *JSON*, com os campos relativos ao tipo de função que vai ser executada (*type*), o nome da caixa (*name*) e o tempo em segundos

(*timestamp*) e o socket espera por uma resposta do server que acabe em `\r \n`, que sinaliza o fim de envio de dados por parte do *server*. Recebidos os dados, o programa apresenta uma mensagem de sucesso caso a caixa tenha sido criada. Caso contrário, apresenta uma mensagem de erro com detalhes sobre a razão da criação da caixa ter falhado.

### Caixas Privadas

Nesta função vão ser criadas as caixas privadas, que podem apenas ser acedidas por quem criou a caixa. Para que tal seja possível, é necessário o uso de criptografia (neste caso, segundo um sistema assimétrico).

```
def create_box_private(box_name):
    tcp_s.send(str(json_create_private(box_name) + '\r\n'))
    msg = tcp_s.recv(4096)
    if not msg.endswith("\r\n"):
        while not msg.endswith("\r\n"):
            msg = msg + tcp_s.recv(4096)
        dic = dict(json.loads(msg))
        if str(dic['code']) == 'OK':
            print 'Your box has been created successfully'
        else:
            print 'An error has occurred. Please try again later'
            print 'Details' + str(dic['content'])
```

Figura 3.9: Função *create\_box\_private*

A *string* de *JSON* para ser enviada ao server é produzida pelo módulo *\_create\_private* que leva como argumento a variável *box\_name* e cria um documento em *JSON* com campos relativos ao tipo de função que vai ser executada (*type*), o nome da caixa (*name*), a hora em segundos em que o pedido foi enviado (*timestamp*), a chave pública relativa ao utilizador (*pubk*) e a assinatura do utilizador (*sig*) que é definida pelo método de assinaturas *PKCS1\_PSS*, com assinatura definida pela concatenação da *timestamp* e do nome da caixa com o uso da chave privada do utilizador.

Após isto, o socket vai enviar a *string* de *JSON* acabada em `\r` e vai aguardar por uma resposta do server também acabada em `\r \n`, e após receber a resposta, apresenta ao utilizador uma mensagem de sucesso caso a caixa tenha sido criada, e caso contrário, mostra uma mensagem de erro com detalhes sobre o porquê de não ter sido possível criar a caixa.

```
def json_create_private(box_name):
    secs = str(int(time.time()))
    pubk = str(get_pubk())
    name = str(box_name)
    text = pubk + secs + name
    prv_key = RSA.importKey(open('privateKey.pem', 'r').read())
    digest = SHA.new(text)
    signer = PKCS1_PSS.new(prv_key)
    signature = signer.sign(digest)
    sig = base64.encodestring(signature)

    j_create = {
        'type': 'CREATE',
        'name': name,
        'timestamp': secs,
        'pubk': secs,
        'sig': sig
    }

    return json.dumps(j_create)
```

```
def json_create_public(box_name): # Creating SON String for Box Creation
    j_create = { # then there is no encryption on the message
        'type': "CREATE",
        'name': str(box_name),
        'timestamp': int(time.time()) # Encontrar comando para meter timestamp
    }
    return str(json.dumps(j_create))
```

### 3.2.4 PUT

A função *PUT* envia uma mensagem para um caixa, quer esta seja pública ou privada, tendo a particularidade de que apenas quem criou a caixa pode ver o seu conteúdo. Para isto, tem-se o módulo *put\_msg*.

Este módulo pede o nome da caixa para onde o utilizador quer enviar a mensagem e passa isso como argumento à função *json\_put*, a qual vai criar uma *string* de *JSON*, com campos relativos ao tipo de função a ser executada (*type*), ao nome da caixa para onde vai ser enviada a mensagem (*name*) e a mensagem em si (*content*).

Após isto, a socket vai enviar a *string* de *JSON* acabada em `\r` e vai aguardar por uma resposta do *server* também acabada em `\r \n`, e, após receber a resposta, apresenta ao utilizador uma mensagem de sucesso caso a mensagem tenha sido enviada. Caso contrário, mostra uma mensagem de erro com detalhes sobre o porquê de não ter sido possível enviar a caixa.

```
def put_msg():
    tcp_s.send(json_put(raw_input("Insert Box Name: "), raw_input("Insert Message: ")))
    msg = tcp_s.recv(4096)
    if not msg.endswith("\r\n"):
        while not msg.endswith("\n\n"):
            msg = msg + tcp_s.recv(4096)
    dic = dict(json.loads(msg))
    if str(dic['code']) == 'OK':
        print 'Your message was delivered successfully'
    else:
        print 'An error has occurred while delivering your message. Please Try again'
```

Figura 3.12: Função *put\_msg*

```
def json_put(name, msg):
    j_put = {
        "type": "PUT", # Action : Send a message to a box
        "name": str(name), # Box Name
        "timestamp": int(time.time()), # Box TimeStamp
        "content": str(msg) # Content to be sent to the box
    }
    return str(json.dumps(j_put) + '\r\n') # Returning a JSON String ending in \r\n
```

Figura 3.13: Função *json\_put*

### 3.2.5 GET

Na função *GET* vai ser pedido ao *server* o conteúdo de uma caixa. Para que tal seja possível, criou-se o módulo *get\_box* que pede ao utilizador o nome da caixa a que pretende aceder e usa-a sua resposta como argumento para a função *json\_get*.

```
def get_box():
    tcp_s.send(json_get(raw_input("Insert Box Name: "))) # Method for getting a box from server
    # Sends a JSON string to the server through the socket tcp_s
    msg = tcp_s.recv(4096) # Receiving a message from the server
    if not msg.endswith("\r\n"): # if the message doesn't end with \r\n
        while not msg.endswith("\n\n"): # While it doesn't end with \r\n
            msg = msg + tcp_s.recv(4096) # The socket keeps receiving data
    dic = dict(json.loads(msg)) # Then a dictionary is made from the received JSON
    if str(dic['code']) == 'OK': # If the code from the received string is 'OK'
        print str(dic['content']) # The message is shown
    else: # Otherwise
        print 'An error has occurred while searching for your box.' # An error message is shown
        print 'Details: ' + str(dic['content']) # with details regarding the error
```

Figura 3.14: Função *get\_box*

A partir deste ponto, gera uma string de *JSON* com campos relativos ao tipo de função a ser executada (*type*), nome da caixa (*name*), tempo em segundos (*timestamp*) e a assinatura do utilizador, que é feita com base na concatenação da chave pública, com o *timestamp* e o nome da caixa, sendo depois assinada pelo método *PKCS1\_PSS* com recurso à chave privada do utilizador

Depois disto, o *socket* manda uma mensagem acabada em *\r \n*, à qual o



```

def json_get(name):

    secs = str(int(time.time()))
    name = str(name)
    text = secs + name
    prv_key = RSA.importKey(open('privateKey.pem', 'r').read())
    digest = SHA.new(text)
    signer = PKCS1_PSS.new(prv_key)
    signature = signer.sign(digest)
    sig = base64.encodestring(signature)
    secs = int(time.time()) # Generating the timestamp as an integer
    j_get = {
        "type": "GET", # Action : Request a Box from the Server
        "name": str(name), # Box Name
        "timestamp": int(secs), # TimeStamp
        "sig": sig # Signature with Box Name, Timestamp and Private Key
    }
    return str(json.dumps(j_get)+'\r\n') # Returning a JSON String ending in \r\n

```

Figura 3.15: Função *json\_get*

servidor vai responder enviando também uma mensagem acabada em `\r \n`. Após isto, se a caixa existir, o programa mostra o conteúdo da mesma. Caso a caixa não exista ou esteja vazia, o programa vai mostrar uma mensagem de erro com detalhes sobre esse mesmo erro.

## Capítulo 4

# Conclusões

Nos dias de hoje, para que haja segurança e fiabilidade num programa que faça comunicações pela internet, é preciso que haja muito trabalho por trás. Este trabalho demonstrou as dificuldades que os programadores passam na criação de tais programas. Desde manter o *server* e o cliente a trabalhar segundo os mesmo moldes, a evitar que a privacidade dos clientes seja posta em risco, tendo também especial consideração para que o programa não se comporte de maneira estranha (não prevista) face à passagem de argumentos estranhos (inválidos). Todos estes elementos são essenciais para que o utlizador do programa consiga ter uma boa experiencia aquando da sua utilização, mantendo sempre a sua privacidade.

Concluimos que, para garantir a fiabilidade de um programa é necessário olhar para o mesmo de diversas perspetivas (mesmo as que, provavelmente, seriam consideradas irracionais) para que seja possível tentar prever todos os casos que o programa em causa enfrentará. Cada situação imprevisível levanta problemas e os testes exaustivos tentam ao máximo garantir que essas situação, simplesmente, não existem.

# Contribuições dos autores

RS escreveu a maior parte do código, testes e do relatório

PV focou-se no relatório em  $\text{\LaTeX}$  e revisão textual.

No geral, distribui-se a percentagem de trabalho 70% para RS e 30% para PV.