Trabalho de Aprofundamento 2, Grupo 3

Universidade de Aveiro

Rodrigo Rosmaninho, Eurico Dias

Trabalho de Aprofundamento 2, Grupo 3

DETI

Universidade de Aveiro

Rodrigo Rosmaninho, Eurico Dias
(88802) r.rosmaninho@ua.pt, (72783) dias.eurico@ua.pt

20 de Abril de 2018

Agradecimentos

Gostaríamos de agradecer a:

Professor Auxiliar António José Ribeiro Neves , pelas excelentes aulas sobre Python e Sockets, e esclarecimento de dúvidas.

Professor Auxiliar João Paulo Barraca , regente da U.C., pelo esclarecimento de variadas dúvidas sobre este trabalho de aprofundamento, ajuda com problemas com a conexão à sonda, e extensão do prazo de entrega.

Professor Auxiliar António Manuel Adrego da Rocha , pela aula elucidante que nos foi dada sobre LaTeXno semestre passado, sem a qual não conseguiríamos ter completado este relatório.

Conteúdo

1 Introdução **1**

2 Metodologia **2**
2.1 main.py 2
2.1.1 Objetivos 2
2.1.2 Métodos 2
2.2 net.py 4
2.2.1 Objetivos 4
2.2.2 Métodos 4
2.3 security.py 4
2.3.1 Objetivos 4
2.3.2 Métodos 4

3 Testes **5**
3.1 main.py 5
3.2 security.py 5

4 Conclusões **6**

Lista de Figuras

4.1 *Output* do programa no terminal. 11
4.2 Resultado dos dados presistidos num ficheiro **csv!** (**csv!**). 12
4.3 Resultado dos testes unitários sobre o ficheiro **main.py**. 12
4.4 Resultado dos testes unitários sobre o ficheiro **security.py**. . . . 13

Capítulo 1

Introdução

Este trabalho, desenvolvido no âmbito da **uc!** (**uc!**) de Laboratórios de Informática, constou na criação de um cliente em Python com a capacidade de aceder remotamente a uma sonda de temperatura, humidade, e vento instalada pelos docentes.

O cliente recebe dados em **json!** (**json!**) a cada 10 segundos e regista-os num ficheiro em formato **csv!** (**data.csv**), imprimindo, eventualmente, alguns avisos no terminal conforme as condições climatéricas.

Para além disso, as mensagens entre o cliente e o servidor são encriptadas usando **aes!** (**aes!**)-128 com uma chave de encriptação calculada por ambos os intervenientes através do algoritmo de Diffie-Hellman. No entanto, as mensagens estão sujeitas a sofrer corrupção foi, por isso, necessário acrescentar alguma robustez ao código.

Para simplificar a resolução do trabalho, o grupo decidiu dividir o código-fonte em 4 ficheiros Python (excluíndo os ficheiros utilizados para a elaboração de testes unitários), cada um com a sua função e vários métodos.

Outra decisão do grupo foi a inclusão de cores nas impressões para o terminal, de forma a salientar erros e informações importantes.

O projeto do code.ua associado a este trabalho pode ser consultado em `http://code.ua.pt/projects/labi2018-ap2-g3`

Capítulo 2

Metodologia

Apresentação e descrição da metodologia utilizada para a realização do projeto e obtenção de resultados.

2.1 main.py

2.1.1 Objetivos

O ficheiro **main.py** consiste na lógica principal de comunicação com o servidor, receção e interpretação do **json!**, e escrita de valores num ficheiro em formato **csv!**.

Ou seja, o ficheiro tem as funções:

- Conectar ao servidor por sockets **tcp!** (**tcp!**) usando funções de **net.py**.
- Establecer comunicação encriptada de acordo com o guião (CONNECT/READ) para trocar os parametros necessários para a criptografia por **dh!** (**dh!**). (usar funções de **security.py**.
- Validar e ler respostas do servidor em **json!**.
- Determinar se é necessário avisar o utilizador das condições climatéricas.
- Escrita dos dados gerados em formato **csv!** para o ficheiro **data.csv**.

2.1.2 Métodos

Método main()

Este método contem a lógica principal de todo o projeto. É este que é chamado quando se executa o ficheiro **main.py**. Pode ser dividido em duas partes:

- Establecimento de conexão (encriptada) com o servidor.
- Receção e utilização contínua dos dados recebidos.

Sendo assim, a lógica do método segue a seguinte ordem:

1. Establecer um socket **tcp!** com o porto **8080** de **xcoa.av.it.pt**
2. Chamar a função **DHInitialValues()** do **security.py** para obter os parametros A, p, g enviar "CONNECT <A, p, g>"ao servidor.
3. Receber o parametro B do servidor e apartir dele gerar a chave de encriptação usando **DHGetSecret()** do **security.py**. Também é recebido um token.
4. Enviar a primeira mensagem encriptada com **aes!** ao servidor ("READ <token>"), para que o servidor começe a enviar valores provenientes da sonda.

É ainda necessecário tomar em consideração o facto das transmissões poderem sofrer corrupção e, por isso, não ser possível desencriptar e interpretá-las.

A vasta maioria das mensagens recebidas durante o tempo de vida do programa são atualizações dos valores provenientes da sonda e podem, se ocorrer corrupção, ser ignoradas.

As mensagens recebidas durante a troca de parametros criptográficos são, no entanto, cruciais, visto que se, por exemplo, ocurrer corrupção na mensagem que contém os valores **token** e **B**, o cliente não conseguirá calcular a chave de encriptação usada pelo servidor e não será possivel desencriptar as mensagens que forem recebidas posteriormente.

Desta forma, as partes do código referentes a operações de desencriptação de mensagens foram 'envoltas' em blocos **try/except**, já que a corrupção torna a desencriptação impossível dada a discrepâncias no número de bits.

Como já foi dito, todas as exceções originadas em processos de desencriptação de mensagens não cruciais levam a que a mensagem seja ignorada.

Em contrapartida, erros em mensagens cruciais levam à execução do método **handleFatalError()**, que imprime um erro e reinicia o programa.

Método handleFatalError()

Este método é chamado pelo **main()** quando ocorre corrupção numa mensagem enviada pelo servidor que é crucial ao funcionamento do resto do programa.

Como tal, imprime uma mensagem de 'erro fatal' e chama o método **main()** para que o programa seja 'reiniciado'.

Método getInfo()

O presente método é chamado pelo **main()** para que se averigue as possíveis condições climatéricas existentes no momento de execução do programa, baseado nos valores obtidos pela sonda e enviados pelo servidor, e recomenda

vestuário e/ou ações de acordo. A função imprime uma linha por cada tipo de valor (temperatura, humidade e vento).
Como as condições meteorológicas podem variar drasticamente apenas em períodos de tempo significativametne maiores que 10 segundos (intervalo entre as mensagens), não é necessário imprimir avisos num intervalo de tempo tão curto. Para além disso, os valores que chegam são imediatamente impressos no terminal em forma de tabela, o que tornava o *output* do programa bastante confuso. Com isto, a função informa acerca das condições climatéricas aquando da primeira mensagem que chega do servidor, e só volta a avisar passados 5 minutos.

Método initialExchange()

Este método é chamado pelo **main()** quando é necessário efetuar a troca inicial de parâmetros criptográficos com o servidor, ou seja, enviar **CONNECT A,p,g** e receber os valores **token** e **B**. Estes são devolvidos pelo método, para que no **main()** estes possam ser usados para calcular a chave de encriptação a usar.

Métodos restantes

- **isValidJSON()** -> retorna **True** se o argumento constituir **json!** válido.
- **readJSON()** -> retorna um dicionário equivalente a um objeto **json!** dado como argumento.
- **hasError()** -> retorna **True** se o argumento (mensagem enviada pelo servidor em **json!**) constituir uma mensagem de erro.

2.2 net.py

2.2.1 Objetivos

O ficheiro **net.py** consiste num conjunto de métodos essenciais para a conexão e troca de mensagens com o servidor remoto.

2.2.2 Métodos

Método tcpConnect()

Este método establece a conexão por **tcp!** ao servidor cujo endereço e porto aceita como argumentos, returnando o socket recém-criado.

Para além disso também guarda o socket como variável global para que as restantes funções de **net.py** possam fazer uso dele sem terem de o aceitar como argumento.

Se não for possível establecer a conexão, o programa imprime um erro (que pede ao utilizador que verifique o estado da sua ligação à internet) e sai.

Método tcpSend()

Este método codifica os dados a ser enviados (fornecidos por argumento) em base64 ou **utf8!** (**utf8!**) caso os dados estejam encriptados ou não, respetivamente.
De seguida, os dados recém-codificados são enviados para o servidor.

Método tcpRead()

Este método recebe uma nova mensagem do servidor e retorna-a.

2.3 security.py

2.3.1 Contexto

Troca do segredo partilhado

A geração e troca de chaves partilhadas com o algoritmo de Diffie-Hellmann é um método que possibilita a troca de segredos a partir de duas ou mais entidades desconhecidas uma da outra, por via de um canal de comunicação inseguro.
Em resumo, este método pressupõe a utilização de propriedades e fórmulas matemáticas bem conhecidas, que garantem tanto a consistência do resultado entre ambas as partes negociadoras, assim como a dificultação da aplicação de engenharia reversa ao processo de geração dos valores resultantes da aplicação destas fórmulas.

Encriptação

Após ser gerado o segredo, é possível agora a **cifra das mensagens**, para que assegure um nível relativamente satisfatório de segurança (dada a importância dos dados que são transmitidos neste trabalho) na troca de informação entre o cliente e o servidor. Como algoritmo de cifra, foi nos dado como requisito a utilização do **aes!**, com chaves de comprimento 16 (em bytes).

2.3.2 Objetivos

O ficheiro **security.py** , tal como o nome indica, contém os métodos necessários para garantir algum nível de segurança entre as mensagens **json!** trocadas. Resumidamente, os métodos principais neste ficheiro permitem:

- Gerar valores recorrendo ao algoritmo de Diffie-Hellmann para a negociação da chave a utilizar na cifra da informação
- Computar uma chave dos primeiros 16 carateres hexadecimais a partir dos valores **dh!** calculados pelo servidor, recorrendo a uma síntese **md5!** (**md5!**)

- Encriptar mensagens **aes!** recorrendo ao algoritmo de cifras contínuas **aes!**-128 com a chave gerada anteriormente
- Com a mesma chave, decifrar a informação que chega do servidor.

Os métodos secundários, que completam os métodos principais, possibilitam:

- Descobrir valores que complementam a geração dos parâmetros de Diffie-Hellmann
- Retirar o excipiente de uma mensagem **json!** previamente decifrada
- Verificar a primalidade e paridade de um número

Adicionalmente, este ficheiro pode ser executado por si só para a conexão manual ao servidor. O código no final do ficheiro permite depurar alguns tipos de problemas menos óbvios.

2.3.3 Métodos

Método DHGetInitialValues()

Este método gera os parâmetros iniciais para a primeira conexão ao servidor. O fluxo desta função é a seguinte:

- Gera os parâmetros para o cálculo de A: computa um número primo aleatório p e encontra uma base geradora de campo deste primo respetiva g
- Calcula um inteiro aleatório privado a
- Calcula $A = g^a \pmod{n}$
- Retorna um tuplo da forma (A, p, g)

Antes da inicialização do processo, é necessária a geração dos parâmetros por meio de funções de pseudo-aleatoriedade e do cálculo de valores intermédios. A geração destes parâmetros, tanto do número primo p, como do gerador de campo g, foi realizada a partir de uma adaptação do standard **rfc!** (**rfc!**) (Request for Comments) 2631, do **ietf!** (**ietf!**)[1]/**ansi!** (**ansi!**) X9.42[2]. O parâmetro q é um fator primo de p e é originado realizado operações aritméticas com diferentes valores a partir de sínteses **sha!** (**sha!**)-256 de uma string de bits pseudo-aleatória (chamada de *seed*), seguindo de operações bit a bit. O parâmentro p é gerado a partir de q e também é gerado de forma semelhante, garantindo que p é um primo com fator q.
De notar que a adaptação do standard considerou os requisitos descritos no enunciado e as restrições do servidor, que limita o tamanho do número primo (de 32 a 128 bits) do mínimo de 512 bits descritos neste **rfc!**/**ansi!**, para que os algoritmos recomendados no standard produzam um resultado satisfatório.

Adicionalmente, com o módulo de criptografia para Python3 mais recente (pycryptodome), os métodos da função de síntese para a geração dos parâmetros recomendada (já obsoleta) não é disponibilizada. Com isto, as seguintes alterações aos algoritmos de geração foram realizadas:

- Encurtamento do tamanho da String (denominada seed) para 100 bits;
- Alteração das funções de síntese sobre valores intermédios de **sha!**-1, de 160 bits, para **sha!**256, de 256 bits;
- Alterações a valores incluídos que dependem do ponto anterior (para que as operações bit a bit produzam resultados válidos).

Depois da geração destes valores, é gerado um valor g, que é um gerador do subgrupo (do campo de Galois de p) multiplicativo de ordem q, que satisfaz a equação $g^j \pmod p \neq 1$, sendo j um número inteiro tal que $j = \frac{p-1}{q}$.
De seguida gera-se um número inteiro pesudo-aleatório a, no intervalo $[2^{q-1}, q-2]$ e calcula-se o parâmetro final A tal que $A = g^a \pmod n$, que é retornado pela função juntamente com p e g na forma de um tuplo (A, p, g).

Método _checkPrime(num, seg=50)

Como consequência dos algoritmos estandardizados pelo **rfc!**/**ansi!**, muitos dos números computados são primos (o parâmetro p, por exemplo). Considerando que a maior parte deles possuem um tamanho relativamente considerável (p, neste trabalho, tem 127 bits), é necessário um algoritmo que não despendesse grande parte do tempo de geração, e ainda assim, no entanto, satisfatoriamente robusto, para a verificação de primalidade dos números e parâmetros.
Com isto, utilizou-se o **método de primalidade de Miller-Rabin**[3], explicitado no standard. Este **método probabilístico** permite a verificação da pseudo-primalidade de um inteiro com base no **pequeno teorema de Fermat**[4] e, consequentemente, no **teorema de Euler-Fermat**[5]. Semelhante a todos os métodos probabilísticos, tem sempre uma margem de erro para falsos positivos. Este erro, segundo o standard, não deverá ser menor que $\frac{1}{4^{seg}}$. O parâmetro de segurança, *seg*, consiste no nível de passagens sobre a prova do número, garantindo menor erro.
Resumidamente, este método faz uma aplicação do método probabilístico acima sobre um inteiro(num) e retorna um valor booleano para discernir a provável primalidade de um número.

Método DHGetSecret(*p*, *B*)

Este método, resumidamente, computa o segredo partilhado tal que $key = B^a \pmod p$, sendo p o número primo gerado anteriormente e B um valor gerado pelo servidor, de maneira análoga à do método **DHGetInitialValues()**.
De seguida, retorna uma síntese hexadecimal MD5 do valor obtido (32 carateres, pois a síntese **md5!** retorna 16 bytes, na qual cada byte são 2 nibbles), para posterior utilização em cifras assimétricas.

Método encrypt(chave, msg)

O presente método recebe uma chave e uma mensagem, e aplica o algoritmo de cifras assimétricas **aes!**-128, sendo que a chave tem que possuir exatamente 16 bytes. Como a chave recebida do método **DHGetSecret()** retorna uma representação hexadecimal de 128 bits, como já foi explicado, a chave irá ter, tecnicamente, 32 bytes (em string de carateres). Portanto, como foi requerido pelo enunciado deste trabalho, é necessário forçar o tamanho de 16 bytes selecionando apenas os primeiros 16 da chave dada.
Depois, é feita a encriptação da mensagem em blocos de 16 bytes (o tamanho dos blocos é definido pela chave). O último bloco da mensagem pode não ter o tamanho correto. Para que este fique com o tamanho correto, é necessário adicionar um excipiente. Neste trabalho, foram adicionados espaços em branco. Esta mensagem, quando totalmente encriptada, está pronta para ser codificada em base64 por outra função e, seguidamente, enviada para o servidor.

Método decrypt(chave, msg)

Este método recebe uma chave e uma mensagem **json!** encriptada, e decifra o conteúdo com o mesmo algoritmo de cifra utilizado para a encriptação. Adicionalmente, é necessário retirar-lhe o excipiente para que fique exatamente igual à mensagem de encriptação. O excipiente foi retirado tendo em conta que as mensagens enviadas pelo servidor, quando encriptadas, são sempre mensagens em **json!**. Portanto, a forma mais simplificada de obter a mensagem inicial consiste em percorrer a mensagem do fim para o início e contar os carateres até que se identifique o carater "}". Depois, é só retornar o subconjunto da mensagem até à diferença do tamanho da mensagem com o número de carateres contados. Finalmente, é retornada a mensagem desencriptada, para posterior processamento pelo resto do programa.

Capítulo 3

Testes

3.1 main.py

Para este ficheiro foram feitos testes unitários para os métodos:

- **isValidJSON()**
- **readJSON()**
- **hasError()**

Para o fazer, primeiro foram declaradas diversas variáveis com strings **json!** válidas e inválidas (para passar como argumento a **isValidJSON()**), e, para cada string **json!** válida, um dicionário correspondente (para comparar com o resultado de **readJSON()** com a string **json!** usada como argumento).

Por último, foram criadas variáveis correspondentes a strings **json!** válidas com e sem o formato de uma típica mensagem de erro enviada pelo servidor (para passar como argumento a **hasError()**).

Não foram feitos testes unitários para as restantes funções visto que essas dependiam de comunicação com o servidor.

3.2 security.py

Os testes unitários foram realizados para a totalidade das funções, exceto a função **_checkParity**, por ser bastante simples.
Tentou-se que os testes fossem individualizados para cada função, sem qualquer chamada ou utilização de valores entre elas. No entanto, da forma como foram implementadas, estas agem como complementares umas da outras, e, por esta razão, muitos dos testes utilizam valores e chamadas de outras funções criadas

neste mesmo ficheiro.
Os testes para este ficheiro foram escritos noutro ficheiro, denominado **test_security.py**.
Passando à explicitação dos testes:

- No que toca ao método **DHGetInitialValues()**, o parâmetro A não pode ser negativo, sendo que apenas estamos a lidar com números inteiros positivos. O parâemtro p não pode ser negativo nem menor que dois. Finalmente, o g tem que estar congruente com os algoritmos utilizando, sendo que $g \pmod p < 2 \wedge g \pmod p \neq p - 1$.

- Em relação ao método **DHGetSecret()**, foi averiguado se a chave não era nula e se possui exatamente 32 bytes hexadecimais (como string), ou seja, 16 bytes. Adicionalmente, foi emulado o cálculo (simplificado) dos parâmetros Diffie-Hellmann por parte do servidor servidor, para que se conseguisse testar a igualdade entre as chaves obtidas das "duas"partes.

- Para o método **_checkPrime()**, foi testada a robustez do algoritmo aplicado: testou-se a primalidade de um número negativo, do 0, do 1 e do 2, assim como um número primo óbvio, um número não primo óbvio, entre outros.

- Os testes para o método **_unpadJSON()** foram simples: testou-se a mensagem vazia, a string com apenas o carater "}"(retorna a string vazia, como é suposto), um **json!** vazio com excipiente numérico e um **json!** com uma entrada, também este com um excipiente numérico.

- No que toca às funções **encrypt()** e **decrypt()**, estas foram combinadas numa única "função de testes": tentou-se encriptar e desencriptar logo de seguida um json de uma entrada de teste, e também uma string vazia.

Posto isto, a execução deste *set* de testes unitários é bem sucedida, não encontrando qualquer erro.

Capítulo 4

Conclusões

4.1 Resultados

No que toca aos resultados obtidos pela execução do programa, este funciona corretamente.
Como mostra a **??**, observa-se que os dados que são enviados pelo servidor são corretamente impressos no terminal, juntamente com o aviso inicial das condições climatéricas de momento. Face à corrupção das mensagens enviadas pelo servidor, o programa avisa dessa mesma corrupção, ignora o seu conteúdo e continua a ser executado, como pretendido.

```
Ocorreu um erro fatal na troca de parametros criptográficos com o servidor.
Os dados enviados pelo servidor estavam corrompidos.
O programa irá reiniciar automaticamente.

Conexão ao servidor bem sucedida!
Serão recolhidos novos valores a cada 10 segundos.
Estes serão registados em data.csv

 Temperatura |  Humidade  |   Vento
-------------|------------|-----------
   20.22000  |  71.04000  |   2.03000
Temperatura amena! Leve roupa de acordo com a sua sensação da temperatura.
Vento dentro do normal!
   20.21000  |  71.00000  |   7.11000
   20.22000  |  71.00000  |   4.11000
   20.13000  |  70.96000  |   2.23000
   20.16000  |  71.02000  |   0.99000
   20.17000  |  70.97000  |   5.03000
DECODE ERROR : A última mensagem do servidor estava corrompida.
   20.19000  |  71.01000  |   7.45000
   20.21000  |  71.03000  |  12.47000
   20.28000  |  70.98000  |   1.07000
```

Figura 4.1: *Output* do programa no terminal.

Na figura **??**, pode-se verificar que o ficheiro **csv!** foi preenchido com a informação das mensagens que foram enviadas pelo servidor, com a adição de um campo com a data aquando do seu registo (ou o *timestamp*), que só faz sentido neste caso, pois os dados meteorológicos apenas têm relevância quando são registados temporalmente.

```
TIME,WIND,TEMPERATURE,HUMIDITY
2018-04-21 18:43:44.053412,2.03,20.22,71.04
2018-04-21 18:43:54.077358,7.11,20.21,71.0
2018-04-21 18:44:04.087346,4.11,20.22,71.0
2018-04-21 18:44:14.104983,2.23,20.13,70.96
2018-04-21 18:44:24.118898,0.99,20.16,71.02
2018-04-21 18:44:34.139372,5.03,20.17,70.97
2018-04-21 18:44:54.171911,7.45,20.19,71.01
2018-04-21 18:45:04.186674,12.47,20.21,71.03
2018-04-21 18:45:14.200171,1.07,20.28,70.98
```

Figura 4.2: Resultado dos dados presistidos num ficheiro **csv!**.

Acerca dos testes unitários, foi executado o **pytest** tanto nos testes do ficheiro **test_main.py** como nos do **test_security.py**. Respetivamente, como podemos observar nas figuras **??** e **??**, não foi encontrado qualquer problema na execução destes.

```
============================= test session starts ==============================
platform linux -- Python 3.6.3, pytest-3.1.3, py-1.4.34, pluggy-0.4.0
rootdir: /home/eurico/Desktop/labi2018-ap2-g3, inifile:
collected 3 items

test_main.py ...

=========================== 3 passed in 0.01 seconds ===========================
```

Figura 4.3: Resultado dos testes unitários sobre o ficheiro **main.py**.

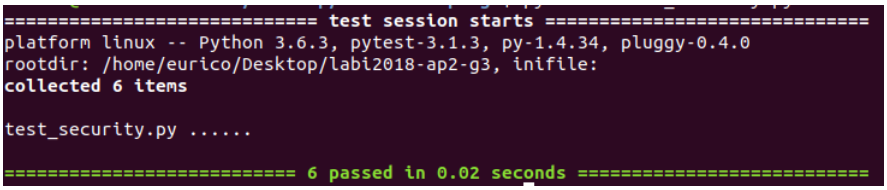

Figura 4.4: Resultado dos testes unitários sobre o ficheiro **security.py**.

4.2 Conclusão

Este trabalho de aprofundamento permitiu aos autores do mesmo consolidar a implementação de sockets, manipulação de estruturas de ficheiros ou mensagens **json!** e **csv!**, e também a aplicação de cifras simétricas por blocos recorrendo a **aes!**, na linguagem de programação Python. Para além disso, forneceu uma ideia geral na negociação de um segredo partilhado pelo método de Diffie-Hellmann, e também na implementação de testes unitários a métodos/funções não triviais, que dependem de outras na sua execução.

Contribuições dos autores

A contribuição individual para o trabalho foi a seguinte:

Rodrigo Rosmaninho

- main.py (exceto método getInfo() e persistência dos dados em ficheiro **csv!**)
- net.py
- colors.py
- test_main.py (Testes unitários)
- Relatório: Estrutura inicial, agradecimentos, introdução, partes respetivas do trabalho

Eurico Dias

- main.py - getInfo() e persistência de dados em ficheiro **csv!**
- security.py
- test_security.py (Testes unitários)
- Relatório: Partes respetivas do trabalho, conclusões, acrónimos e bibliografia

Segue-se a percentagem da contribuição total:

- **Rodrigo Rosmaninho** - 50%
- **Eurico Dias** - 50%

Acrónimos

AES Advanced Encryption Standard

ANSI American National Standards Institute

CSV Comma-Separated Values

DH Diffie-Hellmann

IETF Internet Engineering Task Force

JSON JavaScript Object Notation

RFC Request For Comment

SHA Secure Hash Algorithm

TCP Transmission Control Protocol

UC Unidade Curricular

UTF-8 8-bit Unicode Transformation Format

Bibliografia

[1] I. E. T. Force. (jun. de 1999). RFC 2631 - Diffie-Hellmann Key Agreement Method. [Online. Acedido em Março 2018], URL: https://tools.ietf.org/html/rfc2631.

[2] A. N. S. Institute. (mai. de 1998). Agreement of Symmetric Keys Using Diffie-Hellman and MQV Algorithms. [Online. Acedido em Março 2018], URL: ftp://ftp.iks-jena.de/mitarb/lutz/standards/ansi/X9/x942-05-21-98.pdf.

[3] B. Lynn. (). Number Theory - Primality Tests. [Online. Acedido em Abril 2018], URL: https://crypto.stanford.edu/pbc/notes/numbertheory/millerrabin.html.

[4] Brilliant.org. (jan. de 2018). Fermat's Little Theorem. [Online. Acedido em Abril 2018], URL: https://brilliant.org/wiki/fermats-little-theorem/.

[5] Wikipedia. (abr. de 2018). Euler's Theorem. [Online. Acedido em Abril 2018], URL: https://en.wikipedia.org/wiki/Euler%27s_theorem.