Associação Propagadora Esdeva
Centro Universitário Academia - UniAcademia
Trabalho de Conclusão de Curso

Análise da Importância do Método de Distribuição de Chaves no Processo de Criptografia: o Caso do Protocolo TLS

Guilherme Souza Silva, Romualdo Monteiro de Resende Costa

Curso de Bacharelado em Sistemas de Informação – Centro Universitário Academia (UniAcademia) – Campus Academia
36016-000 – Juiz de Fora– MG– Brasil

{ss.guilherme@hotmail.com, romualdomrc@gmail.com}

Abstract. *Throughout history, the need for cryptographic processes to ensure the security of information has become more recent. Particularly, in the case of the Internet, the TLS protocol is in this context, having evolved over the years, with versions that seek to improve security requirements. However, due mainly to compatibility issues with older versions of browsers, important security requirements may not be used in the communication processes. This is the case, particularly, the key distribution method that can guarantee cryptographic keys used for each transmission session, making the communication process more secure. The TLS protocol and, in particular, the key distribution problem are addressed in this article, which its main goal is to show how the algorithms RSA and DHE works, so then it can demonstrate how the TLS protocol work in a scenario with security problems.*

Resumo. *Ao longo da história, a necessidade dos processos criptográficos para buscar garantir a segurança das informações se tornou mais latente. Particularmente, no caso da Internet, o protocolo TLS se destaca nesse contexto, tendo evoluído ao longo dos anos, com versões que buscaram aprimorar os requisitos de segurança. No entanto, devido à, principalmente, questões de compatibilidade com versões mais antigas de navegadores, importantes requisitos de segurança podem não ser utilizados nos processos de comunicação. Esse é o caso, particularmente, do método de distribuição das chaves que pode garantir chaves criptográficas próprias para cada sessão de transmissão, tornando o processo de comunicação mais seguro. O protocolo TLS e, particularmente, o problema de distribuição das chaves são abordados neste Artigo com o objetivo de demonstrar e analisar o emprego dos algoritmos de criptografia RSA e DHE, abordando, por fim, a utilização do protocolo TLS em um cenário envolvendo problemas na segurança.*

1. Introdução

A segurança computacional é definida em (NIST, 1995) como a proteção aplicada a um sistema de informação a fim de obter as metas de preservar a integridade, a disponibilidade e confidencialidade dos recursos de informação. Essa tríade de requisitos, definidos como os princípios da segurança computacional, podem ser alcançados, ainda que parcialmente, através de métodos de criptografia (STALLINGS, 2011).

Sistemas criptográficos podem ser genericamente definidos como um ambiente onde o conteúdo original é criptografado, isto é, onde o conteúdo é protegido através de algum algoritmo que produz modificações nesse conteúdo original que o tornam inacessível a elementos externos a esse sistema. Normalmente, seguindo o princípio de projetos abertos (CORREIA, 2010), esses algoritmos são parametrizados através de chaves, que modificam o seu funcionamento, de acordo com o valor da chave utilizada. Assim, a segurança do sistema está intimamente ligada à segurança da chave, uma vez que, de posse da chave, o conteúdo original pode ser obtido, independente do algoritmo utilizado.

Associados a um sistema computacional qualquer, algoritmos criptográficos podem ser utilizados para, por exemplo, assinar digitalmente um conteúdo, preservando também a sua integridade, uma vez que modificações nesse conteúdo seriam detectadas pelo processo de assinatura. Adicionalmente, esse processo poderia, também, garantir a origem dos dados, em um processo conhecido como autenticação. No caso da confidencialidade, algoritmos criptográficos são relevantes pois protegem o acesso ao conteúdo, buscando garantir que somente determinados usuários obtenham esse acesso. Por fim, no caso da disponibilidade, a aplicação dos processos criptográficos pode, ainda que indiretamente, ajudar a restringir acessos que, eventualmente, poderiam comprometer o acesso através do comprometimento dos dados.

Devido a abrangência da aplicação dos algoritmos criptográficos, diferentes métodos são associados a esses algoritmos, a fim de serem aplicados em cada item da segurança computacional. Por exemplo, existem algoritmos específicos para aplicações que exigem requisitos de integridade e autenticação

dos dados. Já para a autenticação de equipamentos em um processo de comunicação, podem ser necessários outros métodos, que produzem diferentes algoritmos. Outro exemplo são os algoritmos destinados a preservação da confidencialidade.

Um importante caso onde diferentes métodos de criptografia são empregados, de acordo com o problema em questão, é o protocolo *Transporte Layer Security* (TLS) (IETF, 2015) projetado com o objetivo de oferecer confidencialidade, integridade e autenticação dos dados transferidos através da Internet. Para atingir esses objetivos, os dados são enviados criptografados através da Internet e têm sua origem verificada. Além disso, a chave utilizada no processo de criptografia é modificada a cada sessão e, para isso, é necessário empregar um método criterioso para sua distribuição.

O protocolo TLS utiliza diferentes métodos de criptografia ao longo do seu funcionamento. Para verificar a autenticidade dos servidores e, eventualmente dos clientes, o algoritmo RSA (*Rivest-Shamir-Adleman*) (STALLINGS, 2011) pode ser empregado. Já para a criptografia dos dados, diferentes algoritmos de criptografia podem ser utilizados, com destaque para o AES (*Advanced Encryption Standard*) (STALLINGS, 2011). O AES é um algoritmo de criptografia simétrico, que utiliza, portanto, a mesma chave nos processos de criptografia e descriptografia. Complementarmente, para o processo de distribuição dessa chave, o algoritmo DHE (*Diffie-Hellman*) pode ser empregado. Finalmente, para verificação da integridade das mensagens, o TLS emprega hashs criptográficos (STALLINGS, 2011), como, por exemplo, o SHA256 (STALLINGS, 2011).

Acontece que o processo de distribuição da chave não é obrigatório no protocolo TLS. Nesse caso as chaves de criptografia simétricas são distribuídas usando o próprio RSA em um processo que, apesar de potencialmente seguro, pode levar ao comprometimento da criptografia quando a chave privada é conhecida. Assim, este trabalho tem como objetivo demonstrar a importância do uso do algoritmo de distribuição das chaves no processo de criptografia. Dessa forma, os servidores podem ser configurados a fim de não aceitarem conexões sem o uso do protocolo de distribuição das chaves.

A próxima seção continua a discussão a respeito das diferentes características dos métodos de criptografia e aprofunda sua aplicação com destaque para o uso no TLS. A terceira seção, seguida pela quarta, apresentam em detalhes os algoritmos RSA e DHE, respectivamente. A quinta seção demonstra como é possível descriptografar as mensagens a partir da chave privada quando o algoritmo de distribuição das chaves não é utilizado. Por fim, são apresentadas as conclusões e possibilidades de trabalhos futuros.

2. Princípios Criptográficos aplicados ao TLS

O protocolo TLS é mantido por um grupo de trabalho do IETF (*Internet Engineering Task Force*)[1] a 23 anos sendo, provavelmente, o protocolo mais usado para comunicação segura. A versão mais atual do TLS é a 1.3 (RFC 8446)[2], embora seja usual que a versão 1.2 (RFC 5246)[3] seja encontrada nas implementações existentes na Internet.

Entre as principais diferenças entre as duas versões estão a utilização de algoritmos de criptografia simétricos que suportam autenticação e validação do contexto, impedindo, por exemplo, que trechos de informações criptografadas possam ser extraídas e utilizadas fora de contexto, mesmo que essas informações não possam ser descriptografadas pelo atacante. A nova versão também impede que as chaves utilizadas nos algoritmos RSA e *Diffie-Hellman*, que serão apresentados nas próximas seções, utilizem chaves estáticas, isto é, nessa nova versão as chaves públicas são obrigatoriamente trocadas.

Na verdade, embora a versão 1.3 do TLS traga inovações importantes que possam garantir a segurança do tráfego na Internet para os próximos anos, inúmeras implementações comerciais que utilizam esse protocolo fazem uso da versão 1.2 que, atualmente, é considerada segura. Como exemplo, é possível citar que em 30 de junho de 2018 foi determinado como a data em que as

---

[1] https://www.ietf.org
[2] https://www.ietf.org/rfc/rfc8446.txt
[3] https://www.ietf.org/rfc/rfc5246.txt

empresas de cartão de crédito, em todo o mundo, deveriam terminar a migração da comunicação dos pagamentos para o TLS 1.2[4].

Basicamente, a criptografia no TLS é formada por três principais partes: uma troca de chaves com assinatura e autenticação; um processo de criptografia simétrica para o tráfego das informações e, finalmente, por um algoritmo de hash empregado na autenticação das mensagens. Como diferentes algoritmos podem ser utilizados em cada uma dessas fases, as combinações desses algoritmos são definidas como conjuntos de cifras (*cipher suite*). Como exemplo, a Tabela 1 apresenta parte do conjunto de cifras que pode ser utilizado no Windows 10 a partir da atualização 1809.

Tabela 1. Exemplos de conjuntos de cifras aplicados ao Windows 10 (do autor).

Conjunto de Cifra	Versão TLS
TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384	1.2
TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256	1.2
TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384	1.2
TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256	1.2
TLS_DHE_RSA_WITH_AES_256_GCM_SHA384	1.2
TLS_DHE_RSA_WITH_AES_128_GCM_SHA256	1.2
TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384	1.2
TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256	1.2
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384	1.2
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256	1.2

Os algoritmos do conjunto de cifras escolhido entre os pares de um processo de comunicação são utilizados a fim de manter a comunicação segura. O conjunto de cifras escolhido deve estar disponível nas implementações de ambos os lados da comunicação. Assim, antes do processo de comunicação, propriamente dito, na fase conhecida como SSL handshake[5] o cliente, isto é, o host que solicita os serviços informa ao servidor quais conjuntos de cifras ele

[4] https://www.pcisecuritystandards.org/pdfs/PCI-DSS-3.2.1-Release.pdf

[5] https://www.ssl.com/article/ssl-tls-handshake-overview/

suporta. É o servidor, portanto, que, a partir das possibilidades de conjuntos de cifras informados pelo cliente, decide o conjunto a ser utilizado. Essa escolha, no entanto, é baseada na ordem de preferências enviada pelo cliente.

Como mencionado, cada conjunto de cifras representa diferentes algoritmos utilizados em etapas do protocolo TLS. A Figura 1 apresenta, como exemplo, um conjunto de cifras que usualmente é encontrado em hosts que utilizam o protocolo TLS.

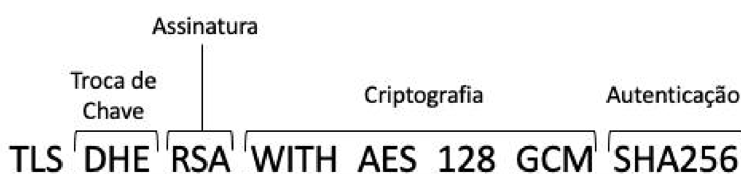

Figura 1. Detalhe dos algoritmos de criptografia em um conjunto de cifra (do autor).

No conjunto de cifra apresentado na Figura 1, o algoritmo DHE, que será explorado na Seção 4 é escolhido para o processo de troca de chaves. O algoritmo RSA, apresentado na Seção 3 é escolhido para autenticar a identidade dos hosts comunicantes (cliente e servidor). O Algoritmo AES é proposto como algoritmo simétrico para criptografia das informações usando uma chave de 128 bits e o algoritmo *Galois Counter Model* – GCM (NIST, 2007) na construção das cifras. Finalmente, o conjunto de cifra apresentado utiliza o algoritmo de hash criptográfico SHA na sua modalidade de 256 bits para auxiliar no processo de autenticação.

O problema é que, dos processos de criptografia mencionados, aquele de distribuição das chaves não é obrigatório. Como exemplo, a Figura 2 apresenta alguns dos conjuntos de cifras aceitos na navegação ao site do Centro Universitário - UniAcademia.

TLS_RSA_WITH_AES_128_GCM_SHA256 (0x9c)	WEAK
TLS_RSA_WITH_AES_256_GCM_SHA384 (0x9d)	WEAK
TLS_RSA_WITH_AES_128_CBC_SHA256 (0x3c)	WEAK
TLS_RSA_WITH_AES_256_CBC_SHA256 (0x3d)	WEAK
TLS_RSA_WITH_AES_128_CBC_SHA (0x2f)	WEAK
TLS_RSA_WITH_AES_256_CBC_SHA (0x35)	WEAK
TLS_RSA_WITH_3DES_EDE_CBC_SHA (0xa)	WEAK

Figura 2. Parte dos conjuntos de cifras aceitos no site www.cesjf.br (do autor).

Como pode ser observado na Figura 2, os conjuntos de cifra apresentados, ao contrário daqueles da Figura 1, não contém previsão de um algoritmo para o processo de distribuição das chaves. As cifras apresentadas na Figura 2 foram coletadas a partir da ferramenta sslabs[6]. Dessa forma, um atacante, de posse das chaves do algoritmo RSA poderia descriptografar todas as mensagens, desde que a comunicação usasse as cifras apresentadas na Figura 2.

3. Processo criptográfico assimétrico, o algoritmo RSA

Com o objetivo de analisar o emprego da criptografia assimétrica esta seção apresenta, inicialmente, uma introdução sobre o algoritmo RSA, seguida por uma proposta de implementação, na subseção a seguir.

3.1. Algoritmo RSA – Teoria

O algoritmo RSA trabalha com criptografia de chave assimétrica, basicamente, utilizando duas ou mais chaves com o propósito de facilitar o processo de distribuição, uma vez que uma das chaves deve permanecer protegida, enquanto a outra pode ser amplamente publicada. Com a chave pública, qualquer pessoa pode criptografar uma mensagem, porém a mensagem só será decifrada com o uso de uma chave privada que deve permanecer protegida.

Um exemplo simples do uso do algoritmo RSA seria com uma pessoa criptografando uma mensagem com uma chave pública e enviando para outra pessoa que, por sua vez, possui a chave privada necessária para decifrar a mensagem criptografada, conseguindo, assim, acessar a mensagem original. A Figura 3 representa esse processo.

---

[6] https://www.sslabs.com

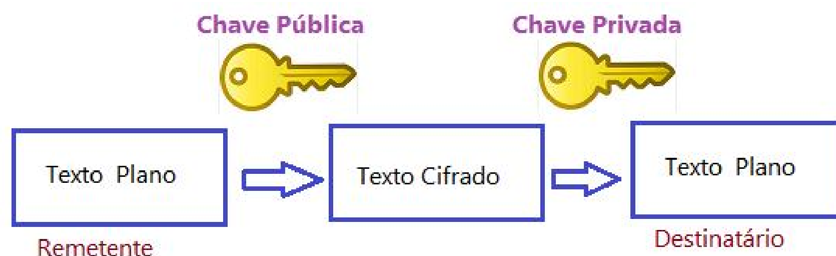

Figura 3. Teoria de funcionamento do algoritmo RSA (do autor).

A ideia do RSA se baseia na teoria dos números, onde se trabalha com números primos e suas fatorações. Como todo número inteiro positivo maior que 1 pode ser decomposto de forma única em um produto de números primos e essa operação não pode ser realizada, até os dias atuais, de forma automática, é difícil executar essa fatoração com números grandes e, portanto, um procedimento dessa natureza levaria muito tempo para ser realizado.

O conceito de função totiente de Euler[7] também é empregado no RSA. Nessa função, dado um número natural x, a função, representada por φ(x) (totiente de Euler), define o número de co-primos desse número x. Esses números são aqueles que não possuem múltiplo divisor comum com x diferente de 1.

No algoritmo RSA as chaves são geradas com base na multiplicação de dois números primos, junto de um valor auxiliar, com o resultado sendo público. Se esse número produzido for grande o suficiente, executar a fatoração da chave pública para descobrir os primos multiplicados para formá-la e, assim, descobrir a chave privada, poderia demorar meses, anos e até mesmo séculos, inviabilizando, portanto, a tentativa de descoberta desses números. Por causa do longo espaço de tempo necessário para descobrir os números necessários, a segurança do algoritmo RSA atualmente é garantida.

---

[7] https://pt.khanacademy.org/computing/computer-science/cryptography/modern-crypt/pi/euler-totient-exploration

3.2. Algoritmo RSA – Prática

Neste trabalho os algoritmos foram implementados e codificados na linguagem de programação Java[8] usando a IDE Netbeans[9] (Versão 8.2). As etapas para seu funcionamento se encontram em duas classes com as funções utilizadas durante o processo. Uma classe define o algoritmo desejado, a outra, uma classe auxiliar, é responsável para recuperar e armazenar os valores utilizados

A Figura 4 apresenta a classe auxiliar *Editor*. Essa classe é responsável por armazenar e recuperar os valores necessários aos algoritmos e que são armazenados em documentos de texto auxiliares. Para implementar essas operações são utilizados quatro métodos. Os métodos *anotar* e *anotarMsg* são responsáveis por armazenar os valores gerados. Já os métodos *recuperarValor* e *recuperarValorMsg* realizam as operações inversas, recuperando do arquivo os valores salvos.

```
1.public class Editor {
2.        public static void anotar(String filename, BigInteger n) throws IOException{
3.            String path = filename+".txt";
4.            BufferedWriter buffWrite = new BufferedWriter(new FileWriter(path));
5.            buffWrite.append(n.toString());
6.            buffWrite.close();
7.        }
8.        public static void anotarMsg(String filename, String msg) throws
IOException{
9.            String path = filename+".txt";
10.            BufferedWriter buffWrite = new BufferedWriter(new FileWriter(path));
11.            buffWrite.append(msg);
12.            buffWrite.close();
13.        }
14.        public static BigInteger recuperarValor(String filename){
15.            BigInteger bigIntegerStr = null;
16.            try {
17.                Scanner scanner = new Scanner(new File(filename+".txt"));
18.                String leitor=null;
19.                while (scanner.hasNext()) {
20.                    leitor = scanner.nextLine();
21.                }
22.                bigIntegerStr=new BigInteger(leitor);
23.                scanner.close();
```

[8] www.java.com

[9] https://netbeans.org/

```
24.                 return bigIntegerStr;
25.                 }catch (IOException e) {
26.                     e.printStackTrace();
27.                 }
28.             return bigIntegerStr;
29.         }
30.         public static String recuperarValorMsg(String filename){
31.            String leitor=null;
32.            try {
33.                 Scanner scanner = new Scanner(new File(filename+".txt"));
34.                 while (scanner.hasNext()) {
35.                     leitor = scanner.nextLine();
36.                 }
37.                 scanner.close();
38.                 return leitor;
39.                 }catch (IOException e) {
40.                     e.printStackTrace();
41.                 }
42.             return leitor;
43.         }
44.}
```

Figura 4. Código da classe auxiliar Editor. (do autor)

A implementação do método principal para execução do algoritmo RSA é apresentada na Figura 5. Nesse método são oferecidas diferentes opções ao usuário, que pode apenas gerar as chaves (opção 1), criptografar e descriptografar mensagens (opções 2 e 3) ou, ainda, através da opção 4, realizar todo o processo.

```
...
1. public class RSA {
2.
3.    public static void main(String[] args) throws IOException {
4.        int opcao = 1;
5.        Scanner sc = new Scanner(System.in);
6.        while(opcao != 0){
7.            System.out.println("Digite a opcão abaixo:\n1 - Gerar chaves\n2 -
Criptografar a mensagem\n3 - Descriptografar a mensagem\n4 - Processo completo\n0-
Finalizar");
8.            opcao = Integer.parseInt(sc.nextLine());
9.                if(opcao == 1){
10.                   gKeys(gPrimo(), gPrimo());
11.               }
12.               if (opcao == 2){
13.                   cripMsg();
14.               }
15.               if (opcao == 3){
16.                   decripMsg();
```

```
17.                 }
18.                 if (opcao == 4){
19.                     gKeys(gPrimo(), gPrimo());
20.                     cripMsg();
21.                     decripMsg();
22.                 }
23.         }
24.     }
...
```

Figura 5. Código inicial para implementação do algoritmo RSA (do autor).

A Figura 6 apresenta o método para geração das chaves. Na 27ª linha, o valor de *n* é determinado pela multiplicação entre os dois primos gerados, nomeados de *p* e *q*, sendo *n* o tamanho do conjunto. O tamanho de um conjunto é necessário para que se tenha um conjunto finito de valores e, com isso, ser possível efetuar o caminho inverso ao realizado para cifrar a mensagem. Os números primos são obtidos a partir da função *gPrimo* (linha 39). Para a função gerar um número primo grande de forma aleatória, o bitlen trabalha como um auxiliar de limite, para evitar que números extremamente grandes sejam gerados e consumam muita memória.

Após o cálculo do valor de *n*, tem-se a implementação da função totiente (φ) definida pela fórmula (n) = (p -1) (q -1). Esta função diz a quantidade de co-primos de um número que são menores que ele. Dois números são co-primos, também chamados de primos entre si, quando o Máximo Divisor Comum (MDC) entre eles é 1.

Após o cálculo da função totiente, deve ser calculada a chave pública, parte dela nomeada de *e*. Deve-se seguir a regra de que $1 < e < \varphi(n)$, onde *e* e φ (n) sejam primos entre si. Basicamente se busca um *e* onde o Máximo Divisor Comum de φ(n) e *e* seja igual a 1, sendo *e* maior do que 1. O número 3 atende aos requisitos necessários e foi utilizado na função, porém, caso seja necessário, pode-se continuar calculando novos números e usar qualquer co-primo que atenda os requisitos.

Assim que se concluem os cálculos dos valores da chave pública e privada, sendo *e* para a chave pública e *d* para a privada, eles são exibidos e, enfim, armazenados através da função *anotar* da classe auxiliar *Editor*.

```
...
25.    public static void gKeys(BigInteger p, BigInteger q) throws IOException{
26.            BigInteger m, n, d, e;
27.            n = p.multiply(q);
28.            m = (p.subtract(BigInteger.ONE).multiply(q.subtract(BigInteger.ONE));
29.            e = new BigInteger("3");
30.            while(m.gcd(e).intValue() > 1) e = e.add(new BigInteger("2"));
31.            d = e.modInverse(m);
32.            System.out.println("p:"+p+"\nq:"+q+"\nn:"+n+"\ne:"+e+"\nd:"+d);
33.            Editor.anotar("p",p);
34.            Editor.anotar("q",q);
35.            Editor.anotar("n",n);
36.            Editor.anotar("e",e);
37.            Editor.anotar("d",d);
38.        }
39.        public static BigInteger gPrimo(){
40.            int bitlen = 2048;
41.            SecureRandom r = new SecureRandom();
42.            return new BigInteger(bitlen / 2, 100, r);
43.        }
...
```

Figura 6. Código para geração das chaves. (do autor)

Com as chaves criadas salvas, a função de cifrar uma mensagem, que se encontra na 44ª linha da Figura 7, pode ser executada. As funções *recuperarValorMsg* e *recuperarValo*r da classe *Editor* leem os respectivos documentos de texto informados no parâmetro e retornam seu valor para se poder trabalhar sob eles, onde a mensagem que se deseja ser criptografada vem do documento de texto *RSA–Mensagem.txt* encontrado na pasta do projeto. Após recuperar os valores, tem início o processo de cifrar uma mensagem, para isso, o RSA utiliza operações modulares, seguindo a fórmula: c = m ^ e mod n. Onde *e* é o valor da chave pública, sendo *n* o tamanho do conjunto, resultante da multiplicação entre as chaves e *m* o valor numérico do conteúdo a ser criptografado. No caso, o valor de *m* são os bytes da mensagem que se está cifrando. Nas linhas 47 e 48 os valores cifrados são exibidos na tela e, com isso, anotados em um novo documento de texto pela função *anotarMsg*.

Indo para o processo responsável por decifrar uma mensagem (linha 50), o programa realiza uma exponenciação modular para o valor de cada byte,

descrito com a seguinte fórmula: m = c ^ d mod n. Onde *d* é a chave privada que foi gerada nas etapas anteriores junta da chave pública, *n* é o tamanho do conjunto e *c* é o valor numérico da letra cifrada. Após a realização, a mensagem cifrada é exibida em seu formato natural, finalizando o processo de decifrar a mensagem.

```
...
44.        public static void cripMsg() throws IOException{
45.            String msg = recuperarValorMsg("Mensagem");
46.            String msgcrip = new
BigInteger(msg.getBytes()).modPow(recuperarValor("e"),
recuperarValor("n")).toString();
47.            System.out.println("Mensagem criptografada: "+ msgcrip);
48.            anotarMsg("RSA - Mensagem criptografada",msgcrip);
49.        }
50.        public static void decripMsg(){
51.            String msgdescrip = new String(new BigInteger(recuperarValorMsg("RSA -
Mensagem criptografada")).modPow(recuperarValor("d"),
recuperarValor("n")).toByteArray());
52.            System.out.println("Mensagem descriptografada: " +msgdescrip);
53.        }
...
```

Figura 7. Processo de criptografia e descriptografia do algoritmo RSA. (do autor)

4. Processo de distribuição de chaves, o algoritmo DHE

Com o objetivo de analisar o emprego da distribuição das chaves esta seção apresenta, inicialmente, uma introdução sobre o algoritmo DHE, seguida por uma proposta de implementação, na subseção a seguir.

4.1 Algoritmo DHE – Teoria

DHE, também chamado de acordo de chaves Diffie-Hellman, apareceu pela primeira vez em um artigo feito por Diffie e Hellman (DIFFIE, 1976), tendo assim originando o nome do algoritmo. O seu objetivo é permitir que dois usuários possam compartilhar e negociar uma chave em segurança, que então será usada para criptografar as mensagens. Para garantir a segurança deste

processo, o DHE depende da dificuldade no cálculo de logaritmos discretos (DIFFIE, 1976).

O funcionamento do algoritmo se baseia através de dois números com conhecimento público, um número primo *q* e um inteiro *a* que é uma raiz primitiva de *q* (DIFFIE, 1976). Em uma negociação de chaves entre dois usuários, o primeiro selecionará um inteiro aleatório de *Xa* < *q* e calculará *Ya = a ^ Xa mod q*. De forma semelhante, o segundo usuário selecionará um inteiro aleatório *Xb* < *q* e calculará *Yb = a ^ Xb mod q*. Cada usuário terá o seu valor *X* privado, tornando apenas o valor *Y* disponível publicamente um ao outro. Após a troca de valores, o primeiro usuário calcula a chave como *K = (Yb) ^ Xa mod q* e o segundo usuário calcula a chave como *K = (Ya) ^ Xb mod q*, ambos os cálculos deverão produzir resultados idênticos.

Ambos os lados negociaram um valor secreto com isto, além de que como Xa e Xb são privados, um possível invasor terá apenas os valores q, a, Ya e Yb para trabalhar em uma invasão, sendo forçado a calcular um algoritmo discreto para determinar a chave, neste ponto entrando uma vantagem em segurança do algoritmo DHE, calcular logaritmo discreto custa muito tempo e se tem uma dificuldade alta na execução, principalmente se os números primos forem grandes (DIFFIE, 1976). A Figura 8 ilustra esse processo.

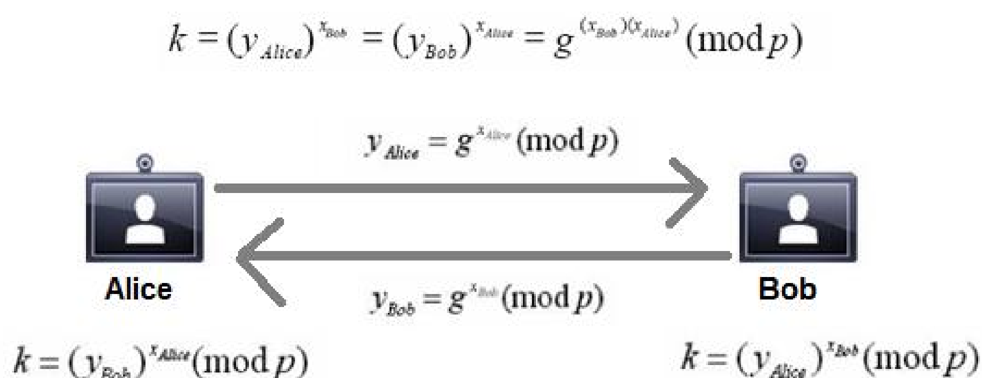

Figura 8. Ilustração da troca de informações entre dois usuários pelo algoritmo DHE, nomeados respectivamente de Alice e Bob para o exemplo comum (do autor).

4.2 Algoritmo DHE – Prática

O código fonte do Algoritmo DHE foi implementado da forma ilustrada na Figura 9 e é apresentado na Figura 10. Inicialmente, entre as linhas 5 e 8, os valores iniciais escolhidos pelos dois usuários são recuperados de seus respectivos documentos de texto, sendo *p* a representação do número primo onde o cálculo de módulo será executado e *g* o número inteiro público auxiliar para a fórmula de cálculo, ambos os números são exibidos após suas escolhas.

```
...
1.        class DHE
2.        {
3.         public static void main(String args[]) throws IOException
4.         {
5.                BigInteger p = Editor.recuperarValor("DHE - p");
6.                System.out.println("Leitura do número primo escolhido para calcular
o módulo: "+ p);
7.                BigInteger g = Editor.recuperarValor("DHE - g");
8.                System.out.println("Leitura do número inteiro público que trabalhará
como root de "+ p +": "+ g);
...
```

Figura 9. Início do código fonte para o Algoritmo DHE. (do autor)

Após os números públicos serem escolhidos pelos usuários, o cálculo do algoritmo pode ser realizado. A Figura 10 ilustra esse cálculo. Na 11ª linha dessa figura é executado *g ^ x mod p* para o primeiro usuário e, na 15ª linha, *g^y mod p* é executado para o segundo usuário, ambos os resultados são armazenados e exibidos como *R1* e *R2* respectivamente. Para finalizar o processo do DHE, após os cálculos serem feitos, as respectivas chaves secretas são calculadas com as informações armazenadas de ambos os usuários, na 17ª linha é feito *R2 ^ x mod p* para o primeiro usuário e, na 19ª linha, a operação *R1^y mod p* é calculada para o segundo usuário. Como medida de controle, caso ambos os usuários tenham chaves secretas idênticas, a mensagem constante na 23ª linha é exibida, por outro lado, caso algum problema tenha ocorrido durante o processo, é a mensagem da 27ª que é exibida.

```
...
9.                BigInteger x = Editor.recuperarValor("DHE - x");
10.                System.out.println("Leitura do número secreto para o usuário 1
```

```
menor do que "+ p +": "+ x);
11.                 BigInteger R1 = g.modPow(x,p);
12.                 System.out.println("R1 = "+ R1);
13.                 BigInteger y = Editor.recuperarValor("DHE - y");
14.                 System.out.println("Leitura do número secreto para o usuário 2
menor do que "+ p +": "+y);
15.                 BigInteger R2 = g.modPow(y,p);
16.                 System.out.println("R2 = "+R2);
17.                 BigInteger k1 = R2.modPow(x,p);
18.                 System.out.println("Chave secreta para o usuário 1: "+ k1);
19.                 BigInteger k2 = R1.modPow(y,p);
20.                 System.out.println("Chave secreta para o usuário 2: "+ k2);
21.                 if(k1.equals(k2))
22.                 {
23.                     System.out.println("Ambos os usuários podem se comunicar entre
si.");
24.                 }
25.                 else
26.                 {
27.                     System.out.println("Houve um erro entre as chaves, os usuários
não podem se comunicar entre si.");
28.                 }
29.             }
30.         }
```

Figura 10. Código fonte para o Algoritmo DHE. (do autor)

5. Utilização do TLS em um cenário envolvendo problemas na segurança

Como mencionado na primeira seção, nem todas as cifras do TLS utilizam um protocolo de distribuição de chaves, como o DHE, apresentado na seção anterior. Esse é o caso, por exemplo, da cifra “AES128-GCM-SHA256”, apresentada na Figura 2 como a primeira cifra fraca aceita pelo site da UniAcademia. Como essa cifra não utiliza um protocolo de distribuição de chaves, a partir da chave privada do certificado é possível realizar toda a descriptografia da comunicação de um site que suporte essa cifra. Para explorar essa falha, a Figura 11 apresenta a arquitetura proposta. Nela, o cliente, a partir de um navegador realiza a chamada do site ao servidor projetado neste trabalho que aceita a cifra mencionada. No meio do caminho, um atacante, usando um

programa de captura de pacotes (sniffer) como o Wireshark[10] tem acesso as informações trafegadas.

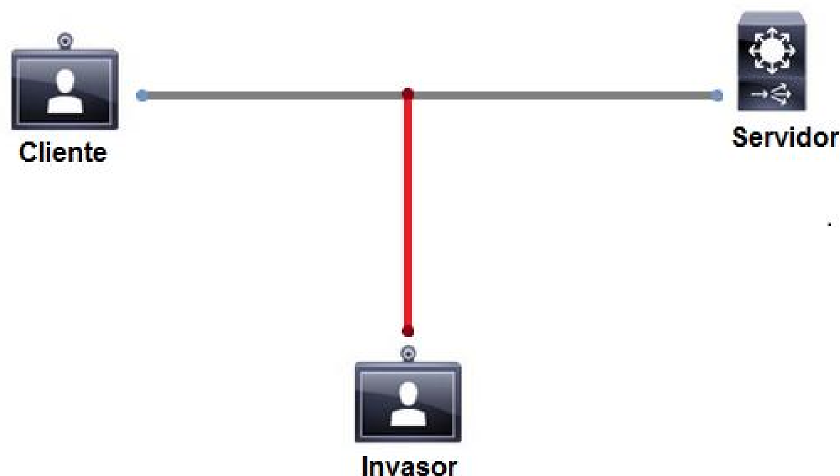

Figura 11. Arquitetura utilizada nos testes de quebra da confidencialidade das mensagens (do autor).

Para implementar a arquitetura proposta na Figura 11, um servidor Web Apache[11] foi instalado e configurado com um certificado, auto assinado, utilizando a cifra “AES128-GCM-SHA256”. A geração do certificado foi realizada utilizando o openssl[12]. Esse procedimento, a princípio, é seguro, uma vez que o tráfego das informações é criptografado. Colocando uma página html simples no servidor e fazendo o acesso, ao capturar o tráfego o processo do TLS, a princípio, parece garantir a confidencialidade das informações. A captura desse acesso é apresentada na Figura 12.

---

[10] www.wireshark.org

[11] httpd.apache.org

[12] www.openssl.org

```
ssl.record.version == 0x0301 && ip.dst==20.44.86.127
No.   Time        Source         Destination    Protocol  Length  Info
3789  185.451616  192.168.0.105  20.44.86.127   TLSv1     186     Client Hello
3798  185.626491  192.168.0.105  20.44.86.127   TLSv1     188     Client Key Exchange, Change Cipher Spec, Encrypted Handshake Message
3805  185.823989  192.168.0.105  20.44.86.127   TLSv1     464     Application Data, Application Data
3808  186.003613  192.168.0.105  20.44.86.127   TLSv1     544     Application Data, Application Data
```

```
Handshake Protocol: Client Hello
    Handshake Type: Client Hello (1)
    Length: 123
    Version: TLS 1.0 (0x0301)
  ▷ Random: 5ee049938614c4612d6617b352a90cf6995e68852f907ebc…
    Session ID Length: 0
    Cipher Suites Length: 28
  ◢ Cipher Suites (14 suites)
      Cipher Suite: TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA (0xc014)
      Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA (0xc013)
      Cipher Suite: TLS_DHE_RSA_WITH_AES_256_CBC_SHA (0x0039)
      Cipher Suite: TLS_DHE_RSA_WITH_AES_128_CBC_SHA (0x0033)
      Cipher Suite: TLS_RSA_WITH_AES_256_CBC_SHA (0x0035)
      Cipher Suite: TLS_RSA_WITH_AES_128_CBC_SHA (0x002f)
      Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA (0xc00a)
      Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA (0xc009)
      Cipher Suite: TLS_DHE_DSS_WITH_AES_256_CBC_SHA (0x0038)
      Cipher Suite: TLS_DHE_DSS_WITH_AES_128_CBC_SHA (0x0032)
      Cipher Suite: TLS_RSA_WITH_3DES_EDE_CBC_SHA (0x000a)
      Cipher Suite: TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA (0x0013)
      Cipher Suite: TLS_RSA_WITH_RC4_128_SHA (0x0005)
      Cipher Suite: TLS_RSA_WITH_RC4_128_MD5 (0x0004)
    Compression Methods Length: 1
  ▷ Compression Methods (1 method)
    Extensions Length: 54
  ▷ Extension: server_name (len=25)
  ▷ Extension: supported_groups (len=6)
  ▷ Extension: ec_point_formats (len=2)
  ▷ Extension: extended_master_secret (len=0)
```

```
ssl.record.version == 0x0301 && ip.dst==192.168.0.105
No.   Time        Source         Destination    Protocol  Length  Info
3795  185.587504  20.44.86.127   192.168.0.105  TLSv1     1188    Server Hello, Certificate, Server Key Exchange, Server Hello Done
3802  185.753341  20.44.86.127   192.168.0.105  TLSv1     113     Change Cipher Spec, Encrypted Handshake Message
3807  185.962118  20.44.86.127   192.168.0.105  TLSv1     224     Application Data, Application Data
3816  186.274090  20.44.86.127   192.168.0.105  TLSv1     272     Application Data, Application Data
```

```
◢ Transport Layer Security
  ◢ TLSv1 Record Layer: Handshake Protocol: Multiple Handshake Messages
      Content Type: Handshake (22)
      Version: TLS 1.0 (0x0301)
      Length: 4009
    ◢ Handshake Protocol: Server Hello
        Handshake Type: Server Hello (2)
        Length: 81
        Version: TLS 1.0 (0x0301)
      ▷ Random: 5ee04ab230b4b11e79da8550d2f56af4437722f2dd571cd5…
        Session ID Length: 32
        Session ID: b92d00007353f8142d577979dbcc7c036fe01289e076fba6…
        Cipher Suite: TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA (0xc014)
        Compression Method: null (0)
        Extensions Length: 9
      ▷ Extension: extended_master_secret (len=0)
      ▷ Extension: renegotiation_info (len=1)
    ◢ Handshake Protocol: Certificate
        Handshake Type: Certificate (11)
        Length: 3585
        Certificates Length: 3582
      ▷ Certificates (3582 bytes)
    ◢ Handshake Protocol: Server Key Exchange
        Handshake Type: Server Key Exchange (12)
        Length: 327
      ▷ EC Diffie-Hellman Server Params
    ◢ Handshake Protocol: Server Hello Done
        Handshake Type: Server Hello Done (14)
        Length: 0
```

Figura 12. Captura de mensagens criptografadas com a cifra "AES128-GCM-SHA256" (do autor).

Na Figura 12 é possível destacar algumas mensagens do TLS. Inicialmente, pelo cliente, várias informações são enviadas por um *client-hello* para o servidor, incluindo, a versão do protocolo e o "Cipher suit list", que consiste em uma lista com todos cifras suportadas. Ao receber as informações do cliente, o servidor pode determinar a melhor forma de trabalhar com o que foi

enviado, respondendo com um *server-hello*, o servidor envia a versão do protocolo escolhida e a cifra escolhida pelo servidor, entre outras informações (STALLINGS, 2011). Após o recebimento das informações vindas do servidor, o cliente envia as informações finais que o servidor necessita, como o Client Key Exchange que contém o material criptográfico do cliente ou seu certificado digital e o Change Cipher Spec que é a sinalização para o servidor que o cliente está pronto para alternar o modo de criptografia para criptografia em massa. Todas essas mensagens podem ser identificadas na Figura 12. No entanto, a partir do início da transmissão dos dados, o Wireshark não consegue descriptografar o tráfego e começa a apresentar mensagens genericamente apresentadas como “application data”.

Neste exemplo, como o servidor possui um certificado autoassinado com uma cifra pré-definida que não possui um protocolo de distribuição de chaves como o DHE a partir da chave privada utilizada na construção do certificado é possível, facilmente, descriptografar todo o tráfego. Para realizar esse processo, o próprio Wireshark pode ser utilizado, a partir das preferências do protocolo TLS nessa ferramenta, a chave pode ser inserida, como apresentado na Figura 13.

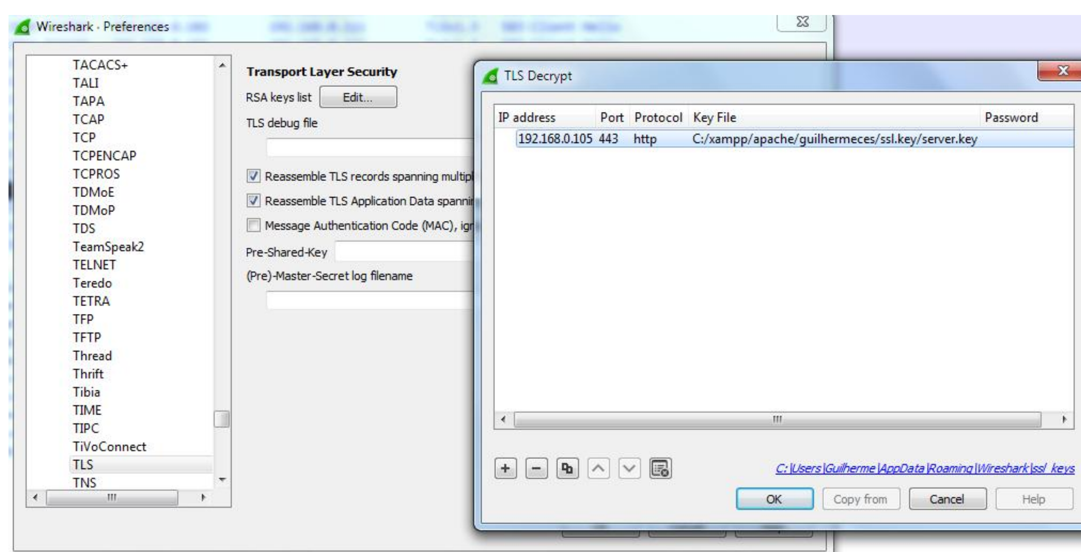

Figura 13. Configuração do Wireshark com a passagem da chave privada (do autor).

Ao ser inserida a chave privada na ferramenta, o conteúdo das mensagens, antes criptografados, são apresentados automaticamente. Por outro

lado, caso o DHE estivesse sendo utilizado, ao invés da chave privada ser utilizada diretamente no processo criptográfico, essa chave seria utilizada na geração de chaves individuais para cada sessão de comunicação, impedindo acessos não autorizados, mesmo que o atacante conseguisse obter a chave privada do certificado digital utilizado no site.

Obviamente, a chave privada, como é esperado, deve ser protegida de acessos não autorizados. Mas apesar das medidas que possam ser implementadas na proteção dessa chave, é possível que, em algum momento ela seja copiada por um atacante e, nesse caso, caso não sejam utilizados mecanismos adicionais de segurança como a utilização de um protocolo de distribuição de chaves como o DHE, o tráfego de informações pode estar vulnerável, como apresentado nesta seção.

Como exemplo, durante o processo de aquisição do certificado digital, muitas empresas recorrem a empresas especializadas como, por exemplo, a Certisign[13] que fornece as chaves pública e privada nesse processo de aquisição. Esse é um momento, por exemplo, onde a chave privada poderia ter sido capturada por um atacante. Além disso, na instalação junto aos servidores, cópias não autorizadas também poderiam ser realizadas o que ressalta a importância da utilização de cifras que contenham um protocolo de distribuição de chaves.

6. Conclusões e Trabalhos Futuros

A criptografia é um procedimento relevante para proteger as informações em diferentes tipos de transações. Com isso, se conclui a utilidade dos respectivos algoritmos estudados que, inclusive, podem ser utilizados em conjunto, como nas etapas do protocolo TLS, ou separados, em atividades menores.

Neste trabalho foram estudados, principalmente, os algoritmos RSA e DHE, o primeiro é um algoritmo assimétrico**,** que permite o uso tanto de uma chave pública quanto de uma chave privada, onde a chave pública ser

[13] www.certisign.com.br

conhecida não causará danos ao projeto. O problema, no entanto, é quando a chave privada se torna conhecida e, no mundo digital, as cópias podem ser feitas sem expor a identidade do atacante. Nesse caso, serviços podem ser atacados sem notarem a violação. Por tudo isso, é altamente recomendável que o DHE, um algoritmo responsável por lidar com troca de chaves entre usuários seja utilizado. Nesse caso, ainda que uma chave seja comprometida ela é válida somente para uma sessão de comunicação e, portanto, a comunicação não permanece comprometida.

Com o objetivo de conhecer melhor os algoritmos mencionados, este trabalho optou por realizar uma implementação individual de cada um antes de realizar testes que comprovassem a necessidade do uso de um algoritmo de distribuição de chaves com o objetivo de tornar o processo de comunicação mais seguro, nesta etapa algumas dificuldades foram encontradas para o emprego correto dos respectivos algoritmos devido a cálculos matemáticos e codificação, onde eventualmente foram finalizados. Essa implementação tem objetivos puramente didáticos, a fim de compreender melhor os procedimentos de cada algoritmo e, assim destacar a importância individual de cada um deles.

Como trabalho futuro, a partir das implementações realizadas, pode ser possível refazer os testes apresentados na seção anterior a partir de comunicações realizadas utilizando essas implementações nos processos de criptografia. Nesse caso, poderia ser construída uma ferramenta didática que mostrasse a importância de cada um dos processos de criptografia mencionados sem a necessidade de montar servidores externos e utilizar clientes como os navegadores, o que ocorreu na seção anterior.

Por fim, cabe ressaltar que a utilização de algoritmos de criptografia de chave pública é atualmente seguro, só sendo possível realizar o processo de descriptografia de posse da chave privada, como apresentado na seção anterior. Esses algoritmos, desde que a chave privada seja preservada de acessos não autorizados são por si só seguros. O problema, como já mencionado, é que cópias não autorizadas dessa chave podem ser realizadas, seja na sua criação ou durante o funcionamento dos serviços.

8. Referências

CORREIA, Miguel Pupo. SOUSA, Paulo Jorge. **Segurança No Software**. FCA, 2010.

DIFFIE, Whitfield. HELLMAN, Martin E**. New Directions in Cryptography**. IEEE Transactions on Information Theory. Vol 22. Novembro, 1976.

GARBE, Galeno. **Diffie-Hellman Na Prática - Completo e Desmistificado**. Youtube, 13 de set. de 2013. Disponível em: <https://www.youtube.com/watch?v=omBdQFxGJs4> Acesso em: 4 de mai. de 2020.

IETF. **Deprecating Secure Sockets Layer Version 3.0.** Internet Engineering Task Force. Request for Comments: 7568. ISSN: 2010-1721. 2015.

NIST. **Introduction to Computer Security: The NIST Handbook**. Special Publication. National Institute of Standards and Technology. 1995.

NIST. **Computer Security: Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC**. Special Publication. National Institute of Standards and Technoology. 2007.

STALLINGS, W. **Network Security Essentials: Applications and Standards**. 4ª ed., Prentice Hall, 2011.

WIKIPÉDIA, **RSA (Sistema Criptográfico)**. Wikipédia. Disponível em: < https://pt.wikipedia.org/wiki/RSA_(sistema_criptográfico)> Acesso em: 20 de abr. de 2020.