

WEBCHAT COM CRIPTOGRAFIA DE PONTA A PONTA



Ronaldo Ávila de Arruda Junior

Leonardo Garcia dos Santos

Santo André – SP

2024

INTRODUÇÃO

Nos últimos tempos, o avanço da tecnologia tem se tornado cada vez mais iminente, esses avanços possibilitam a troca de informações globalmente entre dois dispositivos a milhares de quilômetros em tempo real. Com mais de 3,2 bilhões de usuários que acessam a internet todos os dias e geram centenas de milhares de bytes de informação que trafegam pelo mundo por meio da internet, surge uma pergunta sobre a segurança dessas informações: essas mensagens estão realmente isentas de serem interceptadas e lidas por algum cibercriminoso? A criptografia é fundamental nesse cenário e vem ganhando força nos últimos anos a fim de garantir a confidencialidade dos dados. Existem muitos algoritmos que implementam a criptografia para resolver problemas em relação a segurança, onde cada um utiliza métodos e paradigmas distintos. Nos chats de aplicativos de mensagens, é muito comum utilizarem uma criptografia de ponta a ponta, onde a mensagem é criptografada pelo remetente e só poderá ser descriptografada pelo destinatário, essas estratégias são cada vez mais aprimoradas e disseminadas na atualidade, uma vez que a segurança é fundamental para a harmonia social e política.

VISÃO GERAL

Neste projeto, foi desenvolvido um WebChat com criptografia de ponta a ponta, utilizando o algoritmo RSA e AES. As linguagens de programação utilizadas, juntamente com sua respectiva cobertura, foram: JavaScript, com uma cobertura de 96,9%; Go, com 0,2%; e o restante consistiu em HTML e CSS. A princípio, foi proposta a utilização de uma VPN para criar um servidor seguro entre os usuários. Era definido o uso do Wireguard ou do OpenVPN. No entanto, a implementação desses envolvia a liberação de certas portas bloqueadas no firewall do ponto de acesso. Para evitar conflitos com o provedor de internet, foi decidido utilizar uma instância da AWS para hospedar o sistema.

Como funciona um WebSocket

O websocket é um protocolo que permite a comunicação bidirecional full-duplex entre um único socket TCP. Inicialmente o client solicita conexão com o servidor utilizando o protocolo HTTP, após o servidor abrir a conexão o protocolo HTTP sofre um 'upgrade' para realizar o handshake com o socket

DESCRIÇÃO GERAL

O usuário acessa o sistema através do endereço público da instância AWS usando a porta de onde o servidor está em execução. Após acessar, o usuário deve definir um nome de usuário e um avatar na interface da aplicação para se conectar a um websocket e gerar suas chaves. Caso o outro usuário já esteja presente no socket, o script envia a chave pública deste para o usuário que acabou de entrar. Quando o usuário recém-conectado se conectar, ele também envia sua chave para o outro. Após o handshake do usuário com o servidor, o usuário pode se comunicar com o outro para enviar mensagens criptografadas em tempo real. O usuário criptografa a mensagem com a chave pública do destinatário, e a mensagem é descriptografada apenas com a chave privada do destinatário.

DESCRIÇÃO DO CÓDIGO FONTE

Front-End

Bibliotecas usadas

- Bootstrap 5.3.0 componente front-end;
- JQuery 3.6.0;
- Socket.io 1.4.5;
- Movable Type Scripts AES;
- jsbn

Os recursos utilizados do bootstrap foram classes de estilos já pré-definidos pela biblioteca.

Socket.IO é uma biblioteca que permite a comunicação em tempo real, bidirecional e baseada em eventos entre o navegador e o servidor.

Index.html

Aqui está o entrypoint do sistema, para onde o usuário é direcionado ao acessar a instância. Existem algumas verificações no LocalStorage do cliente assim que o HTML é renderizado, uma vez que as mensagens e as chaves são armazenadas localmente. Caso o usuário tenha acessado anteriormente o sistema,

ele poderá entrar diretamente na sessão de mensagens. Essa verificação é feita no `main.js`.

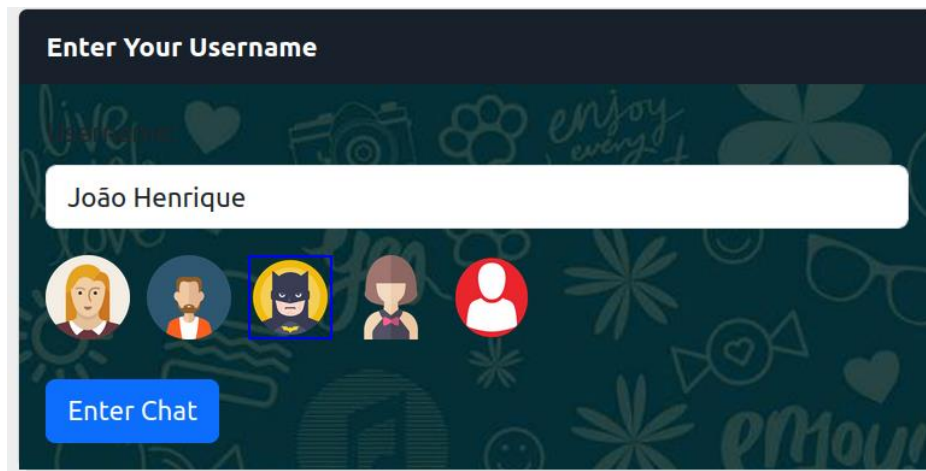


Figura 1: Interface gráfica para registro de nome de usuário e seleção de *avatar*.

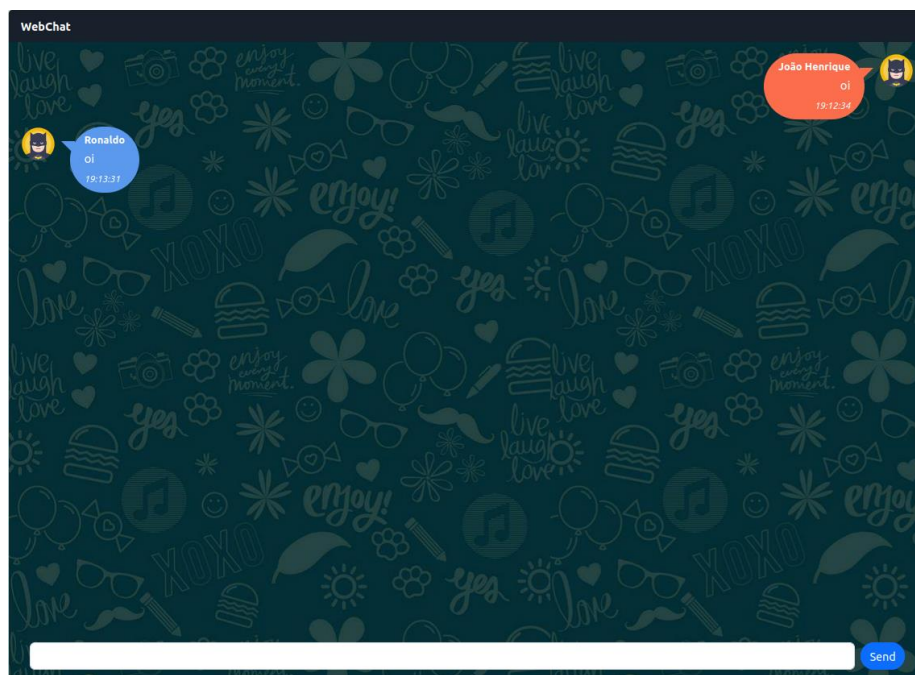


Figura 2: Interface gráfica para o *chat*.

Main.js

O arquivo `main` é aquele que faz toda a lógica estruturada do sistema, o arquivo define algumas constantes, sendo elas o do conteúdo da mensagem e `username`, além de usar a biblioteca `socket.io` para se conectar a um `websocket`:

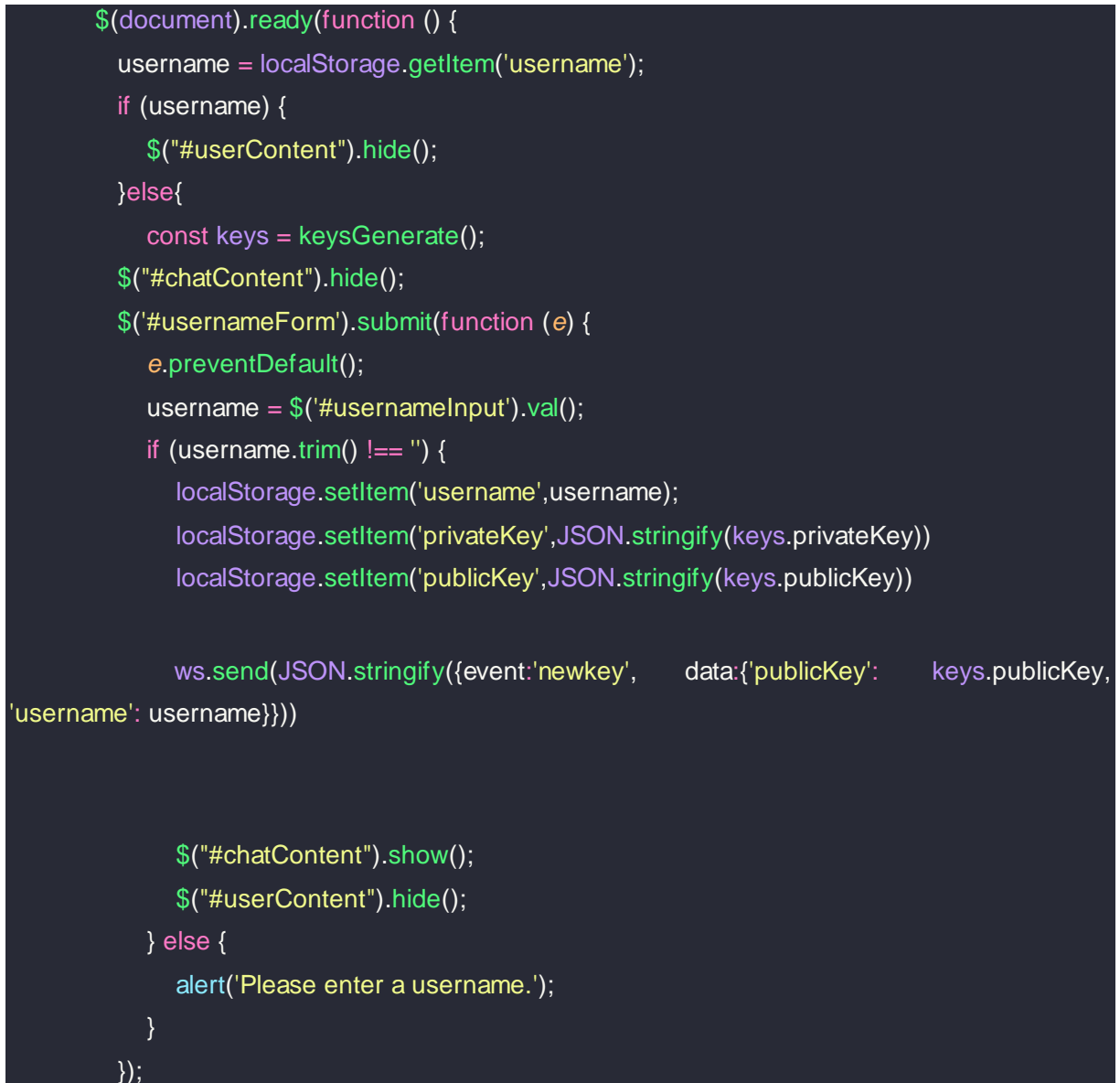
A terminal window with a dark background and three colored window control buttons (red, yellow, green) in the top left corner. It contains a single line of JavaScript code that creates a new WebSocket object.

```
const ws = new WebSocket('ws://localhost:8000/ws');
```

Figura 3: Conexão com o websocket

No local de homologação foi usado o localhost, porém na instância esse endereço se torna o ip público da vm (“virtual machine”) da aws. Foi importado algumas funções do arquivo RSA.js e AES.js para lidar com a criptografia, que serão abordadas na parte do RSA.

Evento inicial:

A terminal window with a dark background showing a large block of JavaScript code. The code is a jQuery event handler for the document's ready state, which checks for a username in local storage, generates keys if none exist, and sends a 'newkey' event to the WebSocket.

```
$(document).ready(function () {  
  username = localStorage.getItem('username');  
  if (username) {  
    $("#userContent").hide();  
  }else{  
    const keys = keysGenerate();  
    $("#chatContent").hide();  
    $('#usernameForm').submit(function (e) {  
      e.preventDefault();  
      username = $('#usernameInput').val();  
      if (username.trim() !== "") {  
        localStorage.setItem('username',username);  
        localStorage.setItem('privateKey',JSON.stringify(keys.privateKey))  
        localStorage.setItem('publicKey',JSON.stringify(keys.publicKey))  
  
        ws.send(JSON.stringify({event:'newkey',    data:{'publicKey':    keys.publicKey,  
'username': username}}))  
  
        $("#chatContent").show();  
        $("#userContent").hide();  
      } else {  
        alert('Please enter a username.');      }  
    })  
  });  
});
```

```
}  
  
});
```

No trecho do código acima, foi utilizado jQuery para facilitar a transição de elementos visuais, além de realizar a verificação se o usuário já se conectou por meio do LocalStorage. Caso o usuário seja novo, a função `keysGenerate` (pertencente ao arquivo `RSA.js`) será chamada. Em seguida, as chaves pública e privada serão armazenadas localmente. Quando as chaves forem geradas, é necessário enviá-las para a outra pessoa conectada. Para isso, é invocado o método `send` do `WebSocket`, um método da biblioteca `Socket.IO` para enviar dados através desse socket.

Evento ao enviar mensagem

```
send.onclick = () => {  
  const message = {  
    username: username,  
    content: input.value,  
    avatar: localStorage.getItem('selectedAvatar'),  
    date: getCurrentDate(),  
  }  
  insertMessage(message, true)  
  
  //CRIPTOGRAFIA  
  message.content = do_encrypt(JSON.parse(localStorage.getItem('targetKey')), message.content);  
  ws.send(JSON.stringify({event: 'message', data: message }));  
  input.value = "";  
};
```

A imagem acima demonstra a função que será chamada após o usuário clicar no botão "send" definido no HTML. Ela gera uma constante "message", que é um JSON dos valores da mensagem. Após isso, é chamada a função `insertMessage(message, true)`. Essa função imprime na tela a mensagem do próprio usuário. O conteúdo da mensagem é criptografado utilizando a função `do_encrypt` (`RSA.js`) para criptografar a mensagem com a chave pública do destinatário. É chamado a função `send` do `websocket` para enviar esse json com o conteúdo da mensagem, com o evento 'message'. O parâmetro "true" da função `insertMessage`

indica que é uma mensagem do usuário, afetando o lado que a mensagem se aloca no chat.

Evento ao receber uma mensagem:

```
ws.onmessage = function (msg) {  
  msg = JSON.parse(msg.data)  
  if (msg.event == "message"){  
  
    if (username != msg.data.username) {  
      msg.data.content = do_decrypt(JSON.parse(localStorage.getItem('privateKey')), msg.data.content)  
      insertMessage(msg.data, false)  
    }  
  }  
  else if (username != msg.data.username){ //newkey  
    localStorage.setItem('targetKey', JSON.stringify(msg.data.publicKey))  
    ws.send(JSON.stringify({event:'newkey', data:{'publicKey': keys.publicKey, 'username': username}}))  
    return  
  }  
  //QUANDO RECEBER A DELE ENVIAR A MINHA  
};
```

A imagem acima ilustra a função onmessage do WebSocket, que é chamada quando a função send é executada, ou seja, quando a mensagem é enviada para o servidor. Foram definidas algumas verificações, uma delas é que o username da mensagem seja diferente em relação ao utilizador. Após isso, a função do_decrypt (RSA.js) é chamada para descriptografar o conteúdo da mensagem. Com o conteúdo decifrado, a função insertMessage é chamada com o segundo parâmetro como "false", o que define que a mensagem seja posicionada à esquerda da tela.

Caso um dado JSON chegue com o "event" diferente de "message" (exemplo: event: "newKey"), significa que o outro usuário enviou sua chave pública (conectou-se à instância). A chave pública é então salva localmente como "targetKey", que como o próprio nome sugere, será a chave alvo usada para criptografar. É necessário enviar a chave do utilizador também, então é invocada a função send para enviar a chave pública do utilizador. A instrução return é usada para quebrar o ciclo, uma vez que a função send é chamada, que por sua vez chama a função onmessage.

```
function insertMessage(messageObj, isOut) {
```

```
const listItem = document.createElement('li');

if (isOut == true) {
  listItem.setAttribute('class', 'out');
}else{
  listItem.setAttribute('class', 'in');
}

const chatBodyDiv = document.createElement('div');
chatBodyDiv.setAttribute('class', 'chat-body');

const chatImgDiv = document.createElement('div');
chatImgDiv.setAttribute('class', 'chat-img');
const img = document.createElement('img');
img.setAttribute('alt', 'Avatar');
img.setAttribute('src', messageObj.avatar);
chatImgDiv.appendChild(img);

const chatMessageDiv = document.createElement('div');
chatMessageDiv.setAttribute('class', 'chat-message');

const heading = document.createElement('h5');
heading.textContent = messageObj.username;

const paragraph = document.createElement('p');
paragraph.textContent = messageObj.content;

const date = document.createElement('div');
date.setAttribute('class', 'hour-format')
date.textContent = messageObj.date;

chatMessageDiv.appendChild(heading);
chatMessageDiv.appendChild(paragraph);
chatMessageDiv.appendChild(date);

chatBodyDiv.appendChild(chatMessageDiv);

listItem.appendChild(chatImgDiv);
```



```
listItem.appendChild(chatBodyDiv);

const messages = document.getElementById('messages');
messages.appendChild(listItem);

localStorage.setItem('chatMessages', messages.innerHTML);
}
```

Acima está a função "insertMessage", que é responsável por inserir mensagens em uma lista no chat. Dependendo do segundo parâmetro, classes diferentes serão dispostas com o método `setAttribute`. Ao longo do código, toda a estrutura do corpo, estilo e conteúdo da mensagem serão definidos. Tanto a estrutura quanto o conteúdo da mensagem serão armazenados localmente, permitindo que os usuários tenham um histórico de mensagens.

RSA.js

Em um primeiro momento, para a criptografia, foi desenvolvida uma implementação do algoritmo RSA sem a utilização de bibliotecas de terceiros. No entanto, não foi possível conseguir desempenho necessário para que aplicação gerasse chaves de tamanho suficiente para conferir segurança à criptografia, com chaves de pelo menos 1024 bits. Devido a isso, posteriormente, passou-se a ser utilizada a biblioteca *jsbn*, escrita por Tom Wu, para a criptografia, em vista de sua simplicidade e desempenho.

A implementação inicial consistia na geração de chaves privadas e públicas. No primeiro passo, é necessário gerar valores P e Q primos da ordem de 10^{100} . No entanto, no projeto, foi utilizado a ordem de 7^{10} . Isso ocorreu porque, para verificar a primalidade dos números, é necessário usar o teste de primalidade de Miller-Rabin, e quanto maior a ordem do número, menor será a probabilidade de o algoritmo encontrar um candidato primo. Utilizando a grandeza recomendada, foi executado um laço de repetição com 10000 iterações e o teste falhou em todas. Por outro lado, com a grandeza menor, todos os testes foram bem-sucedidos.

```
function gerarPQ() {
  while (PQ.length < 2) {
```

```

do {
  let num = Math.floor(Math.random() * 7 ** 10);
  if (num % 2 === 0) {
    num++;
  }
  if (millerRabinTest(num, 10)) {
    PQ.push(num);
    break;
  }
} while (true);
}
return PQ;
}

```

Acima está a função responsável por gerar P e Q. Foi utilizado um array para armazenar P e Q. Para isso, um laço com duas iterações foi utilizado, e dentro de cada iteração, outro laço do while é executado. Este é responsável pela geração e verificação do teste de Miller-Rabin. Um número é gerado com Math.random() de 0 a 1, e em seguida, esse número é multiplicado pela ordem de 7^{10} . Optou-se por números ímpares, uma vez que estão mais presentes no grupo dos primos. Após isso, é chamada a função millerRabinTest (pertencente ao arquivo miller-rabin.js). Os parâmetros dessa função são "num" e "n" (número de vezes a ser executado o teste). Quanto maior o "n", maior a probabilidade do teste acertar o estado de primalidade do número.

Multiplicar "P" e "Q" para encontrar "N":

```

function calcularN() {
  return PQ[0] * PQ[1];
}

```

Calcular Z de forma que $Z = (P - 1) * (Q - 1)$:

```

function calcularZ() {
  return (PQ[0] - 1) * (PQ[1] - 1);
}

```

Encontrar um E aleatório da forma $2 < E \leq Z$ e que não seja primo relativo de Z. Outras implementações optam por utilizar um E fixo e de menor grandeza para todas as chaves, devido ao maior desempenho proporcionado para a criptografia e verificação de assinaturas. No entanto, para maior segurança sem sacrificar performance, é recomendado utilizar o valor de $2^{16}+1$, ou 65337 (BONEH, 1999)

```
function calcularE(z) {  
  let e;  
  do {  
    e = getRandomNumber(2, z);  
  } while (gcd(e, z) !== 1);  
  return e;  
}
```

```
function getRandomNumber(min, max) {  
  return Math.floor(Math.random() * (max - min + 1)) + min;  
}
```

Função para o algoritmo de Euclides para encontrar o máximo divisor comum:

```
function gcd(a, b) {  
  if (b === 0) {  
    return a;  
  }  
  return gcd(b, a % b);  
}
```

Calcular D de forma que $E \cdot D \pmod{Z} = 1$, para isso é necessário o algoritmo de euclides estendido para encontrar o maior divisor comum assim como os coeficientes x e y, tais que $ax + by = \text{MDC}(a,b)$.

```
function calcularD(e, z) {  
  const [d, _] = extendedEuclideanAlgorithm(BigInt(e), BigInt(z));  
  return (d % BigInt(z) + BigInt(z)) % BigInt(z);  
}
```

A mensagem criptografada toma a forma de $C = M^E \bmod N$. Dessa forma, E e N compõem a chave pública.

```
function encrypt(m, publicKey) {
  var result = []
  var arrMsg = textToAscii(m)
  for(let i = 0; i < arrMsg.length; i++){
    result[i] = modPow(BigInt(hexToInt(arrMsg[i])), BigInt(publicKey.e),
    BigInt(publicKey.n));
  }
  return result
}
```

A criptografia consiste em converter os caracteres da mensagem no código da tabela ascii utilizando a função “textToAscii”

```
function textToAscii(message){
  var result = []

  for (let index = 0; index < message.length; index++) {
    result[index] = message.charCodeAt(index).toString(16)
  }
  return result
}
```

A função retorna um array da tabela ascii em hexadecimal e na função “encrypt” é convertido o array de hexadecimal para inteiro a partir da função “hexToInt”:

```
function hexToInt(hex){
  return parseInt(hex,16)
}
```

Função para descriptografar:

```
function decrypt(c, privateKey) {
```

```

var result = []
var resultHex = []
c = convertArrayStringToInt(c)

for(let i=0; i < c.length; i++){
    result[i] = modPow(c[i], BigInt(privateKey.d), BigInt(privateKey.n))
}
for(let i=0; i < result.length; i++){
    resultHex[i] = intToHex(result[i])
}
return asciiToText(resultHex)
}

```

A mensagem será $M = C^D \pmod N$, após descriptografar todo o array do conteúdo da mensagem é necessário converter para hexadecimal e por fim de Ascii para Texto.

Biblioteca jsbn

Devido a impraticabilidade do código anterior para geração de chaves suficientemente grandes para a criptografia segura em tempo real, foi criado um arquivo que gerencia as criptografias e descriptografias utilizando a biblioteca jsbn.

```

class RSAKeyAttributes {
  constructor(e, bits) {
    var rsa = new RSAKey();
    rsa.generate(parseInt(bits), e);
    this.e = e;
    this.n = linebrk(rsa.n.toString(16), 64);
    this.d = linebrk(rsa.d.toString(16), 64);
    this.p = linebrk(rsa.p.toString(16), 64);
    this.q = linebrk(rsa.q.toString(16), 64);
    this.dmp1 = linebrk(rsa.dmp1.toString(16), 64);
    this.dmq1 = linebrk(rsa.dmq1.toString(16), 64);
    this.coeff = linebrk(rsa.coeff.toString(16), 64);
  }
}

```

No código acima foi criado uma classe que instâncias a biblioteca, e no método construtor de RSAKeyAttributes tem como parâmetro o 'e' e 'bits' (número de bits), após, é instanciada a RSAKey e gerada as chaves.

```
function do_encrypt(publicKey, message) {  
  var rsa = new RSAKey();  
  rsa.setPublic(publicKey.n.toString(), publicKey.e.toString());  
  var res = rsa.encrypt(message);  
  if(res) {  
    return linebrk(res, 64);  
  }  
}
```

Acima a função para encriptação é demonstrada, com os parâmetros 'publicKey' e 'message', a classe RSAKey é instanciada e o método setPublic() é chamado para setar a chave pública na classe, após, é chamada a função 'encrypt' para criptografar a mensagem.

```
function do_decrypt(privateKey, message) {  
  var rsa = new RSAKey();  
  rsa.setPrivateEx(privateKey.n, '3', privateKey.d, privateKey.p, privateKey.q, privateKey.dmp1,  
    privateKey.dmq1, privateKey.coeff);  
  if(message.length == 0) {  
    return;  
  }  
  var res = rsa.decrypt(message);  
  return res;  
}
```

Acima a função para a descriptografia das mensagens, foi chamada a função 'setPrivateEx' para definir os parâmetros para os cálculos internos da biblioteca, após, é chamado a função decrypt da classe RSAKey para descriptografar a mensagem.

miller-rabin.js

Esse código foi criado com referência a bibliotecas já existentes que utilizam o teste de Miller-Rabin, em questão foi utilizado como referência a biblioteca disponível no github chamada “number.js”.

Back End

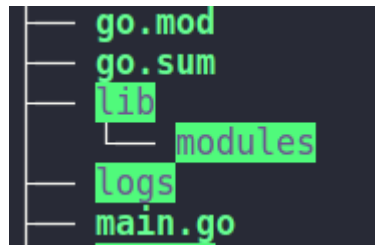


Figura 4: Listagem dos arquivos do back-end

Acima é representada a estrutura dos diretórios que compõem o backend. O arquivo go.mod são os módulos necessários para a aplicação rodar, já o arquivo go.sum é usado para gerar o arquivo de módulos e armazenar no cache, assim não é necessário baixar todos os módulos toda vez que subir o container. Para o servidor go foi usado o módulo “socket” e foram importadas as seguintes bibliotecas:

```
import (  
    static "github.com/gin-contrib/static"  
    "github.com/gin-gonic/gin"  
    "gopkg.in/olahol/melody.v1"  
)
```

Foi escolhido o melody para lidar com o websocket e o gin para criar um servidor web. O arquivo (main.go) que abre conexões websocket contém apenas uma função, que é a main():

```
func main() {  
    r := gin.Default()  
    m := melody.New()  
  
    r.Use(static.Serve("/", static.LocalFile("./public", true)))
```

```

r.GET("/ws", func(c *gin.Context) {
    m.HandleRequest(c.Writer, c.Request)
})

m.HandleMessage(func(s *melody.Session, msg []byte) {
    m.Broadcast(msg)
})

r.Run(":8000")
}

```

"A variável 'r' é uma instância de gin, assim como 'm' é uma instância de melody. O comando gin.default inicializa um roteador HTTP com configurações padrões. Após, é necessário indicar qual diretório o servidor web inicializado em 'r' vai usar como rota "", no caso é o "./public" que contém o index.html que é sempre o primeiro arquivo a ser chamado. É necessário lidar com o websocket e para isso definiu-se um roteamento GET para o caminho "/ws". Quando o servidor receber uma solicitação GET para este caminho, ele chamará a função de retorno de chamada fornecida. Nesta função de retorno de chamada, estamos usando o método HandleRequest do objeto melody.Melody para lidar com a solicitação WebSocket. Isso configura a conexão WebSocket. O melody possui uma função que lida com mensagens recebidas na conexão que é o 'handleMessage', essa função faz o broadcast da mensagem recebida para todas as sessões."

INSTÂNCIA AWS

O sistema foi hospedado em uma instância EC2 da aws, para isso foi necessário realizar algumas configurações, dentre elas a criação de uma chave para o acesso remoto via ssh da instância no computador local, para se conectar a uma instância via ssh é utilizado o comando:

```
“sudo ssh -i key.pem ubuntu@public-ip“
```

É necessário a chave e o endereço público da instância. Também é necessário configurar o ambiente da instância, instalando o docker e algumas dependências. Quando o docker-compose é executado e os container estão em execução,

precisamos configurar os grupos de entrada e saída da instância para permitir a porta do servidor ("8000").

ID da regra do grupo de se...	Intervalo de po...	Protocolo	Origem
sgr-02c8e83862a751821	443	TCP	0.0.0.0/0
sgr-049e554fc37bf752c	80	TCP	0.0.0.0/0
sgr-012b62abe297fa569	22	TCP	0.0.0.0/0
sgr-00e32f5b3cdee9618	8000	TCP	0.0.0.0/0

Figura 5: Regras de entrada da instância do servidor

A imagem acima mostra as regras de entrada onde as portas 443,80,22 e 8000 são liberadas para qualquer endereço de ip.

ID da regra do grupo de se...	Intervalo de po...	Protocolo	Destino
sgr-0a507a92aabaacd2e	Todos	Todos	0.0.0.0/0

Figura 6: Regras de saída da instância do servidor

A imagem acima demonstra a permissão de saída, onde pode enviar tráfego para qualquer ip. Após esses passos podemos acessar a aplicação por meio da seguinte url:

<http://public-ip:8000>.

Quando a instância é interrompida e reiniciada posteriormente ela gera um novo endereço de ip público, dessa forma é necessário alterar no arquivo main a conexão do websocket.

```
const ws = new WebSocket('ws://public-ip:8000/ws');
```

WIRESHARK

O software Wireshark é capaz de capturar todo o tráfego de saída e entrada da máquina, utilizando desse mecanismo foram feitos testes a fim de averiguar a criptografia das mensagens e os seguintes resultados foram coletados:

250	4.344860947	18.229.135.51	192.168.15.8	TCP	68 8000 → 48040
251	4.344861145	18.229.135.51	192.168.15.8	TCP	237 8000 → 48040
253	4.385719793	192.168.15.8	18.229.135.51	TCP	68 48040 → 8000
385	8.705711633	192.168.15.8	18.229.135.51	TCP	68 48026 → 8000

Figura 7: Captura do pacote TCP do servidor utilizando o wireshark

Na imagem acima se refere a uma captura do programa que possibilitou capturar o envio da mensagem e o recebimento. O endereço da instância nesse momento é 18.229.135.51. O conteúdo da mensagem foi devidamente criptografado.

```
|p...+8...x.%...E...
...@.8... '4C...
...@.C. N...
...W... ..u...$
{...u{" event": "
message", "data":
{"userna me": "Ron
aldo", "c ontent":
"7695af5 6159c99e
99350a5d 52f0f434
a1efe3f2 18810186
4401984e 05a4cc03
9\n68e16 b326cf3e
9b8cd28c 8d66b2d5
733b482a 997371ea
80591b42 3948cc24
b4b\n2d8 5be5f23d
48cf254f 23e364fd
805b1b54 32259c62
2c09ac2c 982ec46f
df7e3\nf fa83fd4c
12059ec4 6ba625e3
0ca515fd de832c8c
ab72a610 b774d873
ac6c030", "avatar
": "/Imag es/avata
r3.png", "date": "
19:24:17 "}}
```

Figura 8: Payout do pacote criptografado

CONCLUSÃO

Este trabalho estuda a segurança e a confidencialidade nas trocas de mensagens. Destacam-se a análise do algoritmo de criptografia RSA, um pilar fundamental na estruturação da segurança digital, juntamente com a implementação de firewalls configurados na instância AWS.

Além disso, investigamos o funcionamento dos web chats que empregam criptografia ponta a ponta. Ao longo do projeto, enfrentamos uma série de desafios que exigiram pesquisa detalhada e resolução gradual, muitas vezes buscando soluções em fóruns especializados de programação.

Este estudo reforça a relevância crítica da segurança em web chats, plataformas utilizadas para trocas de mensagens que frequentemente lidam com informações sensíveis, potencialmente sujeitas a riscos políticos, criminais ou pessoais. Destaca-se, portanto, a importância da confidencialidade das mensagens para preservar a integridade e a privacidade dos usuários.

Em suma, este trabalho não apenas amplia nosso entendimento sobre os mecanismos de segurança digital, mas também ressalta a necessidade contínua de aprimoramento e inovação para proteger as comunicações online em um mundo cada vez mais interconectado e vulnerável às ameaças cibernéticas.

REFERÊNCIAS

Number.js, GitHub, 2015. Advanced Mathematics Tools for Node.js and JavaScript. Disponível em: <https://github.com/numbers/numbers.js/blob/master/src/numbers.js>

Bootdey Admin, Bootdey, 2018. Exemple template card-list. Disponível em: <https://www.bootdey.com/snippets/view/card-chat-list>

Melody, GitHub, 2024. websocket framework based on [gorilla/websocket](#). Disponível em: <https://github.com/olahol/melody>

Gin, Github, 2024. Gin is a web framework written in [Go](#). Disponível em: <https://github.com/gin-gonic/gin>

GeekforGeeks, 2023. RSA Algorithm in Cryptography. Disponível em: <https://www.geeksforgeeks.org/rsa-algorithm-cryptography/>

Daniel Rosa, freecodecamp, 2023. Algoritmo de Euclides: MDC. Disponível em: <https://www.freecodecamp.org/portuguese/news/algoritmo-de-euclides-mdc-maximo-divisor-comum-explicado-com-exemplos-em-varias-linguagens/>

MillerRabin, Github, 2023. Miller-Rabin Primality Test. Disponível em: <https://github.com/latonv/MillerRabinPrimality>

Golang, dockerhub, 2024. golang docker image. Disponível em:
https://hub.docker.com/_/golang/

Henrylle Maia, Youtube, 2023. Como lançar uma EC2 com Docker na AWS.
Disponível em:

<https://www.youtube.com/watch?v=rYWjsOWGmxU>

cliffhall, Github, 2019. JSON.stringify() doesn't know how to serialize a BigInt.
Disponível em:

<https://github.com/GoogleChromeLabs/jsbi/issues/30#issuecomment-953187833>

Tom Wu, cs-stanford, 2009. RSA and ECC in JavaScript. Disponível em: <http://www-cs-students.stanford.edu/~tjw/jsbn/>

Boneh, D. Twenty years of attacks on the RSA cryptosystem. N. Am. Math. Soc.
v.46, n.2, p.203–213. fev.1999