Universidade de Brasília
Ciência da computação
Segurança Computacional

Cifra de Bloco e modo operação CTR

Talles Marcelo de Mesquita Caldeira Mat:20/0060295

5 de dezembro de 2023

1 Pré-requisitos

1. Versão do Python 3.x.x >=
2. Acesso ao prompt de comando(cmd)

2 Introdução

A criptografia, uma das pedras angulares da segurança digital, tem evoluído significativamente ao longo dos anos, adaptando-se às crescentes demandas de um mundo cada vez mais conectado. No cerne dessa evolução está a criptografia assimétrica, um método que revolucionou a forma como as informações são protegidas no ciberespaço.

Conceitos Básicos de Criptografia A criptografia, em sua essência, é a arte de codificar mensagens para garantir sua confidencialidade e integridade. Desde tempos antigos, variadas técnicas foram desenvolvidas para proteger informações importantes. No entanto, foi com o advento dos computadores e da internet que a criptografia ganhou uma nova dimensão, evoluindo para métodos mais complexos e seguros.

Criptografia Assimétrica Entre esses métodos, a criptografia assimétrica se destaca. Diferente da criptografia simétrica, que usa a mesma chave para cifrar e decifrar uma mensagem, a criptografia assimétrica utiliza um par de chaves - uma pública e uma privada. Essa abordagem não apenas reforça a segurança, mas também facilita a troca segura de chaves e informações em um ambiente aberto como a internet. A chave pública pode ser compartilhada livremente para cifrar mensagens, enquanto a chave privada, mantida em segredo, é usada para decifrá-las.

2.1 RSA

Um dos pilares dessa tecnologia é o RSA, nome derivado dos sobrenomes de seus inventores - Rivest, Shamir e Adleman. Introduzido em 1977, o RSA permanece como um dos algoritmos mais populares e confiáveis de criptografia de chave pública. Baseado na dificuldade matemática de fatorar grandes números, o RSA é amplamente utilizado para segurança de dados, autenticação e assinaturas digitais.

2.2 OAEP

Junto ao RSA, o Optimal Asymmetric Encryption Padding (OAEP) representa um avanço significativo na segurança de algoritmos de criptografia assimétrica. O OAEP melhora a segurança ao adicionar um elemento de aleatoriedade ao processo de cifragem, mitigando diversos tipos de ataques criptográficos. Essa técnica de padding é essencial para assegurar que o RSA permaneça resiliente contra as ameaças emergentes no mundo digital.

3 Desenvolvimento

3.1 Geração de Chaves RSA

A robustez e a segurança do algoritmo RSA residem fundamentalmente no processo de geração de chaves, um processo que envolve conceitos avançados de matemática e teoria dos números. Este processo pode ser dividido em várias etapas críticas, cada uma desempenhando um papel vital na garantia da segurança do sistema criptográfico.

3.1.1 Seleção de Números Primos

A primeira etapa na geração de chaves RSA é a seleção de dois grandes números primos, geralmente denominados p e q. Estes números devem ser grandes e, idealmente, devem ter comprimentos semelhantes para maximizar a segurança. A seleção de primos grandes e aleatórios é crucial porque a principal vulnerabilidade do RSA é a possibilidade de fatorar o produto desses dois números. Quanto maiores e mais aleatórios forem os primos, mais difícil será para um atacante realizar a fatoração e comprometer o sistema.

O código a seguir ilustra como esses números primos são gerados:

```
def generate_large_prime(keysize=2048):
    while True:
        prime_candidate = random.getrandbits(keysize)
        if is_prime(prime_candidate):
            return prime_candidate
```

3.1.2 Cálculo do Módulo n

Após a escolha dos números primos, o próximo passo é calcular o módulo n, que é simplesmente o produto de p e q. O módulo n é usado como parte da chave pública e da chave privada no RSA. O tamanho de n (em bits) determina o tamanho da chave. A segurança do RSA baseia-se na premissa de que, embora seja fácil calcular n a partir de p e q, é extremamente difícil fazer o caminho inverso sem conhecer os valores originais de p e q.

3.1.3 Função Totiente de Euler

A Função Totiente de Euler, $\phi(n)$, é um conceito crucial na geração da chave privada no RSA. Para o módulo n, que é o produto de dois números primos, $\phi(n)$ é calculado como $(p-1) \times (q-1)$. Esta função é usada para garantir que o expoente escolhido para a chave pública e o cálculo da chave privada sejam realizados de maneira que a cifragem e a decifragem sejam operações inversas uma da outra. A seleção de um expoente público e que seja coprimo com $\phi(n)$ é fundamental para a validade do algoritmo.

3.1.4 Expoente Público "e"e Privado d

O expoente público "e"é escolhido dentro de certos parâmetros. Comumente, o valor 65537 é usado por ser um grande número primo que oferece um bom equilíbrio entre segurança e eficiência computacional. O expoente privado "d"é então calculado como o inverso multiplicativo de "e"módulo $\phi(n)$. O cálculo de "d"deve ser feito de forma que $d \times e$ seja congruente a 1 módulo $\phi(n)$, o que é essencial para a operação de decifragem funcionar corretamente.

A combinação dessas etapas resulta na formação de um par de chaves: a chave pública (e, n) e a chave privada (d, n)

3.2 OAEP e Segurança do RSA

O RSA, embora robusto, enfrenta desafios de segurança, especialmente quando utilizado em sua forma mais básica. É aqui que o Optimal Asymmetric Encryption Padding (OAEP) desempenha um papel crucial, elevando significativamente a segurança do RSA.

3.2.1 Mecanismo de Padding OAEP

O OAEP é uma técnica de padding que adiciona aleatoriedade e complexidade ao processo de criptografia RSA, tornando-o mais resistente a uma série de ataques criptográficos, incluindo ataques de texto cifrado escolhido. O código a seguir mostra a implementação do padding OAEP:

```
def oaep_pad(message, keysize, hash_algo=hashlib.sha256):
    hash_len = hash_algo().digest_size
    k = keysize // 8
    m_len = len(message)

    if m_len > k - 2 * hash_len - 2:
        raise ValueError("Mensagem muito longa")

    padding = os.urandom(hash_len)
    pad_len = k - m_len - 2 * hash_len - 2
    padded_message = b'\x00' + padding + b'\x00' * (pad_len + 1) + message
    return padded_message
```

3.2.2 Processos de Cifração e Decifração

No contexto do RSA com OAEP, o processo de cifração envolve primeiro a aplicação do OAEP na mensagem original, seguido pela cifração usando a chave pública do destinatário. Para a decifragem, o processo é invertido, utilizando a chave privada para decifrar e remover o OAEP, revelando a mensagem original.

3.3 Assinatura Digital RSA

O RSA não se limita apenas à cifração e decifragem de mensagens. Ele também é amplamente utilizado para criar assinaturas digitais, um componente crítico na verificação da integridade e autenticidade das mensagens.

3.3.1 Criação e Verificação de Assinaturas

A assinatura digital no RSA começa com a geração de um hash da mensagem. Esse hash é então cifrado com a chave privada do remetente, criando a assinatura digital. O código para este processo é o seguinte:

```
def sign_message(private_key, message):
    _hash = calculate_sha3_hash(message)
    d, n = private_key
    signature = pow(int.from_bytes(_hash, byteorder='big'), d, n)
    return base64.b64encode(signature.to_bytes(
        (signature.bit_length() + 7) // 8, byteorder='big')
    ).decode()
```

Para verificar a assinatura, o receptor decifra a assinatura usando a chave pública do remetente e compara o hash resultante com um novo hash gerado a partir da mensagem recebida.

4 O Papel da Função 'pow' na Criptografia

4.1 Uso no RSA

Em RSA, usamos a 'pow' para duas coisas principais:

- **Cifração**: Transformamos a mensagem original em uma mensagem cifrada, fazendo algo como "mensagem elevada a um número especial, dividida por outro número e pegamos o resto".
- **Decifração**: Para voltar a mensagem cifrada ao seu formato original, fazemos um processo similar, mas usando um número diferente.

4.2 Por Que É Importante

Usar 'pow' para exponenciação modular é muito mais rápido e prático, especialmente com números enormes, como os que encontramos na criptografia. Além disso, é essencial para a segurança, pois garante que só quem tem a chave certa pode decifrar a mensagem corretamente.

5 Uso de `byteorder='big'` na Criptografia RSA

5.1 Entendendo `byteorder='big'`

No mundo da programação, quando a gente fala de `byteorder='big'`, estamos falando de como os computadores organizam os números grandes. Imagine que você tem um número e precisa dividir em pedacinhos para guardar. `byteorder='big'` é como se você guardasse os pedaços mais importantes primeiro.

5.2 Por que é Importante no RSA

No RSA, a gente lida com números enormes. Às vezes, precisamos transformar esses números em uma sequência de bytes e vice-versa. Usar `byteorder='big'` ajuda a gente a fazer isso direitinho, sem confusão.

5.3 Exemplos no Código

Aqui estão alguns lugares no nosso código onde usamos isso:

5.3.1 Lendo Assinaturas

Quando a gente transforma a assinatura de uma forma que o computador entende (chamamos isso de decodificar) e depois em número grande, usamos `byteorder='big'`. Isso garante que estamos lendo os números da maneira correta.

```
signature = int.from_bytes(base64.b64decode(signature), byteorder='big')
```

Esse trecho é super importante porque transforma a assinatura, que está em bytes, num número grande que a gente pode usar para verificar se a assinatura está certa usando a chave pública RSA.

5.3.2 Transformando Mensagem em Número

Também usamos `byteorder='big'` quando transformamos a mensagem (ou o resumo dela) de bytes para um número grande.

```
message_as_int = int.from_bytes(padded_message, byteorder='big')
```

Isso é importante para manter tudo organizadinho e fazer a cifração e decifração do jeito certo.

6 Conclusão

Este trabalho explora os fundamentos do algoritmo RSA e sua aplicação na criptografia e assinaturas digitais. As implementações em Python demonstram os processos de geração de chaves, cifração e decifração com OAEP, e criação e verificação de assinaturas digitais. Embora a implementação de tais sistemas criptográficos possa ser complexa, o Python fornece uma plataforma acessível para entender esses conceitos avançados, entretanto em algumas tomadas de decisões o Python "esconde"como ele está fazendo algumas manipulações. Assim, levando a problemas grotescos, como por exemplo ao não utilizar o byteorder temos alguns problemas decorentes das escolhas do python.

Referências

[1] David M. Burton. *Elementary Number Theory*. 7th. McGraw-Hill Education, 2010.

[2] Jonathan Katz e Yehuda Lindell. *Introduction to Modern Cryptography*. Chapman e Hall/CRC, 2007.

[3] Alfred J. Menezes, Paul C. van Oorschot e Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[4] *Post-Quantum Cryptography*. Accessed: [your access date here]. 2023.

[5] R.L. Rivest, A. Shamir e L. Adleman. *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems*. Vol. 21. 2. 1978, pp. 120–126.