CryptoSigner — Relatório Técnico

Trabalho I de Segurança Computacional

Autor: Enzo Teles

Sumário

Resumo

CryptoSigner é um protótipo didático de assinatura digital com RSA e um hash simplificado ("SHA-128"). A aplicação oferece uma API HTTP (Flask) e uma interface web para gerar chaves, assinar arquivos e verificar assinaturas. Este relatório descreve a solução, detalha as implementações de RSA e do hash, e relata dificuldades enfrentadas, com foco em conversões de tipos e geração de chaves.

1 Descrição completa da solução

A solução está organizada em camadas simples:

- **Backend (Flask)**: arquivos `src/app.py` e `src/routes.py`. Expõe três rotas: `/api/generate` (gera chaves), `/api/sign` (assina) e `/api/verify` (verifica). Assinaturas em JSON são salvas em `src/data/` (arquivo `.sig`).
- **Core criptográfico**: `src/core/rsa.py` (RSA) e `src/core/sha.py` (hash simplificado).
- **Frontend**: páginas em `src/templates/` com CSS em `src/static/styles.css`. A UI chama a API e apresenta resultados; chaves retornadas são persistidas no `localStorage`.

Fluxo típico: (i) gerar chaves, (ii) enviar arquivo para assinatura, (iii) baixar ou copiar o JSON de assinatura, (iv) verificar enviando o arquivo original, a assinatura, a chave pública e o *salt*. A Figura 1 ilustra telas principais.

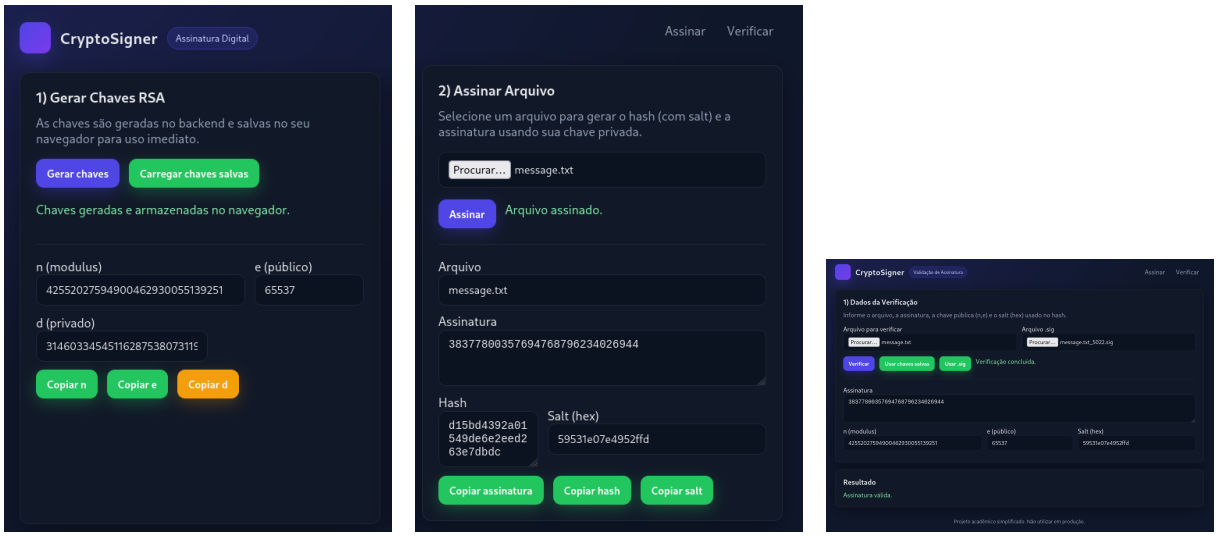

Figura 1: Geração de chaves, assinatura e verificação com sucesso na interface web.

2 RSA: teoria e implementação

2.1 Visão geral do RSA

RSA baseia-se na dificuldade de fatorar o produto de dois primos grandes. Seja p e q primos, $n = pq$ e $\varphi(n) = (p-1)(q-1)$. Escolhe-se e coprimo a $\varphi(n)$, e calcula-se $d \equiv e^{-1} \pmod{\varphi(n)}$. A assinatura de uma mensagem (ou seu resumo) m é $s \equiv m^d \bmod n$; a verificação checa se $m \equiv s^e \bmod n$.

2.2 Como foi implementado (`core/rsa.py`)

- **Primalidade**: `is_prime(n)` usa divisões simples (testes por 2 e 3, e varredura $6k \pm 1$). É adequado para fins didáticos, **não** para produção.
- **Geração de primos**: `gen_prime(bits)` usa `random.Random().getrandbits(bits)` para gerar um ímpar com *bit alto* ligado e testa com `is_prime`. Tamanho padrão: `bits=48` (apenas para demonstração).
- **Chaves**: `gen_keys(bits=48)` fixa $e = 65537$, sorteia p, q, calcula n, φ, verifica $\gcd(e, \varphi) = 1$, e obtém d via `modinv`. Evita $p = q$ e faz várias tentativas até o sucesso.
- **Inverso modular**: `modinv(a,m)` tenta `pow(a,-1,m)` (CPython 3.8+) e faz *fallback* para o algoritmo Euclidiano estendido. Retorna `None` se $\gcd(a, m) \neq 1$.
- **Assinar**: `sign(hash_bytes,d,n)` converte o hash para inteiro grande via `int.from_bytes(..., 'big')` e computa $s = \texttt{pow}(m, d, n)$.
- **Verificar**: `verify(signature,hash_bytes,e,n)` recupera $m' = \texttt{pow}(s, e, n)$ e compara a $m \bmod n$.

Observações de segurança: o uso de primos de 48 bits, RNG não criptográfico e ausência de padding tornam o esquema *inseguro* para uso real. O objetivo é exclusivamente pedagógico.

3 SHA-128: teoria e implementação

3.1 Ideia geral

O módulo `core/sha.py` implementa um *hash* simplificado de 128 bits (saída de 16 bytes), que chamamos "SHA-128 didático". Não segue nenhum padrão de mercado e serve para fins didáticos.

3.2 Etapas da função

1. **Padding** (`pad_message`): concatena 0x80, preenche com zeros até sobrar 8 bytes, e anexa o comprimento original em bits (8 bytes) — formato semelhante ao das famílias SHA reais.
2. **Divisão em blocos**: em `split_blocks`, blocos de 64 bytes (512 bits).
3. **Estado inicial com *salt*** (`initialize_hash_values`): se nenhum *salt* é fornecido, sorteia 8 bytes (`random.randbytes(8)`). Usa esse valor como semente para `random.Random(seed)` e gera dois *words* de 64 bits, (h_1, h_2).
4. **Compressão** (`compression_function`): para cada byte do bloco, faz $h_1 = (h_1 + byte) \bmod 2^{64}$ e $h_2 = (h_2 \oplus byte)$. É propositalmente simples.
5. **Saída**: concatena $h_1 \,||\, h_2$ como 16 bytes: `h1.to_bytes(8,'big') + h2.to_bytes(8,'big')`.

Na assinatura, o servidor devolve o hash em hexadecimal e o *salt* (também em hex). Na verificação, o arquivo é reprocessado com o mesmo *salt* e o resultado comparado via RSA.

Observações: por ser linear e com estado pequeno, esta função de *hash* não é resistente a colisões/preimagens. É apenas uma peça didática.

4 API, formatos e fluxo

4.1 Endpoints

- `GET /api/generate`: retorna n, e e d como *strings* (evita perda de precisão em JavaScript).
- `POST /api/sign`: recebe arquivo e n, d; calcula $(hash, salt)$, gera s, salva um `.sig` em `src/data/` e retorna JSON com {`filename, hash, salt, signature`}.
- `POST /api/verify`: recebe arquivo, s, n, e e *salt*; recomputa o hash com o mesmo *salt* e responde {`filename, hash, signature, valid`}.

4.2 Arquivo .sig

O arquivo JSON salvo contém algo como:

```
{
  "filename": "mensagem.txt",
  "hash": "...hex...",
  "salt": "...hex...",
  "signature": "...inteiro decimal..."
}
```

A UI de verificação pode importar esse JSON (`Usar .sig`).

5 Principais dificuldades e contornos

5.1 Conversões de tipos (int, bytes, str, hex)

Um ponto sensível foi converter corretamente entre tipos:

- Do lado RSA, a mensagem a assinar é o *hash* em `bytes`. Converte-se para inteiro com `int.from_bytes(...,'big')`. Na comparação da verificação, usa-se $(m \bmod n)$ para alinhar o domínio.

- Para transportar dados no JSON, inteiros grandes (`n`, `e`, `d`, assinatura) vão como *strings*; isso evita *overflow*/perda de precisão no JavaScript (`Number` $> 2^{53} - 1$).
- O *salt* e o *hash* trafegam em hexadecimal: usa-se `binascii.hexlify/unhexlify`. Isso simplifica persistência e compatibilidade.

Erros comuns que precisaram de ajuste: espaços em branco em campos numéricos (`strip()`), `ValueError` ao fazer `int()` de entradas inválidas e `binascii.Error` ao decodificar hex — todos tratados nas rotas.

5.2 Geração de chaves e subfunções

Implementar manualmente as sub-rotinas de *número primo*, *inverso modular* e *laço de tentativa* para obter (p, q) adequados exigiu cuidado:

- Evitar $p = q$ e garantir $\gcd(e, \varphi) = 1$.
- *Fallback* robusto em `modinv` quando `pow(a,-1,m)` não está disponível ou quando não há inverso.
- Controlar tentativas e retornar erro claro após muitas falhas (`RuntimeError`).

Essas escolhas deixaram o código claro para fins de ensino, embora **não** sejam suficientes para segurança real (RNG criptográfico, testes de primalidade probabilísticos, tamanhos de chave grandes, padding, etc.).

Conclusão

O CryptoSigner cumpre seu papel didático: demonstra o ciclo de assinatura/verificação com RSA, ilustra um esquema de *hash* com *salt*, e expõe desafios práticos de tipos e geração de chaves. Para uso real, é necessário adotar bibliotecas e padrões consolidados.

Repositório: https://github.com/S3r4ph1el/CryptoSigner