Ciências ULisboa

Faculdade de Ciências da Universidade de Lisboa

Faculdade de Ciências da Universidade de Lisboa

Departamento de Informática

Mestrado em Engenharia Informática/Segurança Informática

RELATÓRIO

Segurança Aplicada

Project - ATM Communication (Phase 1)

David Guilherme (Nº56333)

Rafael Marques (Nº51167)

Kevin dos Santos (Nº64874)

Professor: **Doutor Nuno Ferreira Neves**

2º Semestre Letivo 2024/2025

Abril 2025

Acrónimos

1. **AES** - Advanced Encryption Standard
2. **OAEP** - Optimal Asymmetric Encryption Padding
3. **RSA** - Rivest-Shamir-Adleman
4. **HMAC** - Hash-based Message Authentication Code
5. **Argon2** - Algoritmo de Derivação de Chave
6. **KDF** - Key Derivation Function
7. **DoS** - Denial of Service
8. **PIN** - Personal Identification Number
9. **IV** - Initialization Vector
10. **CBC** - Cipher Block Chaining
11. **MitM** - Man-in-the-Middle
12. **TLS** - Transport Layer Security
13. **SHA-256** - Secure Hash Algorithm 256-bit
14. **TCP/IP** - Transmission Control Protocol/Internet Protocol
15. **Nonce** - Número Único
16. **DoS** - Denial of Service

1. Introdução

O sistema implementa uma comunicação segura, realizada em Python, entre um ou vários clientes (ATM) e um servidor (Bank), simulando transações bancárias. Estas operações são realizadas através de sockets (TCP). É fundamental analisar o contexto em que estamos a trabalhar e compreender o nosso ambiente, ou seja, garantir que as transações são seguras, algo crítico neste tipo de sistema e, ao mesmo tempo, assegurar que não existem falhas que possam causar interrupções no sistema. Para isso, é necessário identificar quais os elementos mais importantes a proteger, tendo sempre em conta as condições do sistema e as capacidades do atacante. Só assim conseguimos alcançar um equilíbrio entre segurança e eficiência, evitando implementar medidas desnecessárias e, acima de tudo, caras

O nosso objetivo acaba por ser dar garantias ao nível da segurança nos três pilares fundamentais: *Confidencialidade, Integridade e Disponibilidade.*
Neste documento são detalhados os protocolos e mecanismos utilizados, nomeadamente o uso de RSA, AES, HMAC e Argon2. Este último vencedor do prémio de melhor algoritmo de *hashing* de 2025. O Argon2 é utilizado tanto para cifrar de forma segura os PINs dos utilizadores como para derivar, a partir de uma chave de sessão, duas chaves distintas: uma para cifragem simétrica (AES) e outra para autenticação (HMAC), garantindo assim um nível adicional de segurança. Todos estes elementos serão explicados em detalhe nas secções seguintes.

Por fim, o relatório também inclui instruções de instalação e execução do sistema, bem como referências às fontes consultadas durante o projeto.

2. Descrição Geral do Sistema

- **Cliente (atm.py)**: Um terminal de atendimento automático que realiza operações bancárias em nome do utilizador.
- **Servidor (bank.py)**: Um servidor bancário que gere contas, processa operações financeiras.

As operações bancárias suportadas são:

- Criação de conta
- Depósito de fundos
- Levantamento de fundos
- Consulta de saldo

Tanto no ATM como no Bank existem um conjunto de validações aplicadas para filtrar os inputs de maneira a alcançar as condições exigidas. Através da função match(), da biblioteca RegEx, verificamos todo o input recebido nos argumentos e garantimos que todos os dados são recebidos conforme o estipulado no enunciado, como por exemplo o *port,* que só é aceite, se estiver entre 1024 e 65535.

3. Protocolo de Comunicação Segura

3.1 Estabelecimento de Sessão Segura

O nosso ponto de partida para garantir uma comunicação segura foi compreender aquilo que o sistema nos assegura desde o início. Tal como descrito no enunciado: *"The auth file will be shared between the bank and the ATM via a trusted channel that is unavailable to the attacker."*

Com base nessa premissa, o nosso objetivo foi encontrar a forma mais segura de trocar as nossas chaves tendo em conta que o auth file iria ser trocado de forma segura. Optamos por utilizar o algoritmo RSA e aí é que começa o nosso processo.

O servidor, através do método start() uma das primeiras coisas que faz é gerar um par de chaves RSA. Utilizou-se uma chave de 2048 bits, considerada segura para este tipo de comunicação, e o expoente público 65537, que é o valor padrão da biblioteca cryptography em Python.

A geração das chaves é feita com o seguinte método:

encryption_algorithm=serialization.NoEncryption()

É de salientar que devido às capacidades do nosso atacante, que não estão totalmente especificadas, optámos por não encriptar localmente a chave privada, por simplicidade. Contudo, essa proteção poderia facilmente ser implementada alterando a seguinte linha:

encryption_algorithm=serialization.NoEncryption()

Por:

encryption_algorithm=serialization.BestAvailableEncryption(b"senha_segura")

A chave privada é guardada no ficheiro <auth_file>.key, enquanto a chave pública é escrita no ficheiro auth_file, que será posteriormente partilhado com os clientes de forma segura.

Toda a documentação dos métodos utilizados nesta fase encontra-se na seguinte biblioteca:

RSA — Cryptography 45.0.0.dev1 documentation.

O passo seguinte consiste em transmitir a chave pública ao ATM. Consideramos este passo seguro, uma vez que é feito fora do alcance do atacante. Com essa chave pública, o ATM cifra uma chave simétrica aleatória,de 32 bytes,um valor que para garantir a sua quebra não seria exequível em tempo útil. Esta vai ser a chave simétrica partilhada

Este processo é realizado na função *encrypt_with_PublicKey*(), que utiliza o algoritmo de cifra asimétrica *RSA* com o esquema de preenchimento *OAEP* e a função de hash *SHA-256*. Esta combinação assegura que a cifragem seja probabilística, não determinística, e resistente a ataques por dicionário (*dictionary attacks*) ou ataques por cifra escolhida (*chosen ciphertext attacks*).

A chave cifrada de seguida vai ser incluída num value de um dicionário deste estilo:

{Session Key: base64 key}

Que então vai ser passada para uma String de binário e de seguida para JSON para ser mais leve e conseguir ser transmitida pela socket, utilizando o método sendall para enviar o pacote.

Durante o desenvolvimento, decidimos substituir o recv(1024) padrão do Python, uma vez que sabíamos que as mensagens transmitidas por TCP não eram entregues de forma consistente. Isto significa que, por vezes, uma mensagem pode chegar completa, dividida ou apenas parcialmente, para o nosso caso não seria muito interessante porque estamos à espera de JSON bem formados. Além disso, considerámos a possibilidade de um atacante enviar pacotes muito grandes, por exemplo num cenário de Man-in-the-Middle.

Devido a isso ajustamos isso a nosso favor, fizemos duas modificações essenciais.

Como a performance não foi algo abordado e não é algo muito crítico neste tipo de sistemas, optamos por passar a ler recv(1) garantido que a mensagem JSON chega completa mas com um buffer para garantir que estamos a ler na RAM para não decrescer muito a performance.

A segunda alteração foi a definição de um limite máximo de 2048 bytes para cada mensagem recebida. Este teto foi definido com base na nossa estimativa do tamanho máximo necessário para qualquer mensagem legítima (tendo em conta o uso de base64, HMAC, IVs, salt, Argon2, pacote máximo possível vindo do cliente). Se este limite for ultrapassado, a função lança uma exceção 63 e a ligação é terminada Esta última exceção foi a pensar num possível ataque de Dos por exemplo

Voltando à nossa chave de sessão, o ficheiro Bank.py recebe os bytes enviados pelo cliente através da nova função recv_all() e converte-os num dicionário JSON. Em seguida, aplica a decifragem com a sua chave privada RSA, recuperando assim a chave de sessão simétrica que o ATM gerou.

Neste momento, podemos afirmar que o ATM e o Bank partilham uma ligação segura.

Contudo, importa salientar que esta troca de chave não autentica a identidade do cliente. Ou seja, o ATM pode ser qualquer pessoa, incluindo um utilizador malicioso, e o servidor não tem, nesta fase, forma de garantir que está a comunicar com um ATM legítimo. O que esta chave de sessão nos assegura é que a comunicação entre os dois nós é segura, no sentido em que ninguém além desses dois consegue aceder ou modificar os dados transmitidos.

3.1 Pacotes

3.1.1 ATM

Após a geração das chaves simétricas, o passo seguinte foi garantir a confidencialidade e integridade das mensagens trocadas entre cliente e servidor e, ao mesmo tempo, assegurar a autenticação do utilizador.

Através do nosso input conseguimos gerar um ficheiro .card associado a cada utilizador,e através do nosso create account ou seja o -n, criamos um card, o nosso mecanismo para este processo foi gerar um PIN aleatório de 8 caracteres.
Optámos por usar 8 dígitos em vez dos habituais 4, pois um PIN curto representa um vetor de ataque fácil e é possível realizar ataques de força bruta com relativa rapidez. Com 8 dígitos, a complexidade sobe de 10^4 para 10^8 combinações possíveis.
Para proteger este PIN, aplicámos o Argon2, vencedor da competição de algoritmos de *password hashing* em 2025 (password-hashing.net). Utilizamos a biblioteca argon2-cffi em Python, que aplica internamente, um salt embutido, o que significa que não temos de passar nada por rede, garantindo, assim, proteção contra rainbow tables. As iterações escolhidas para o algoritmo foram as *default* que após uma pesquisa entendemos que são muito seguras. O link da implementação em python encontra-se aqui: argon2-cffi · PyPI

Ou seja, este ficheiro .card torna-se, assim, o principal fator de autenticação local, e apenas o cliente que o possui consegue gerar operações válidas (desde que o hash do PIN coincida no Bank).

Entendemos adicionar uma camada extra de segurança à chave de sessão simétrica partilhada, pois o nosso próximo passo seria enviar e receber os pacotes sempre com os pacotes cifrados em AES, modo CBC e HMAC (AES+HMAC). O algoritmo AES é responsável por garantir a confidencialidade dos dados, enquanto o HMAC assegura a integridade e a autenticação da origem da mensagem, ou seja, apenas quem conhece a chave de sessão poderá gerar e verificar o HMAC corretamente.

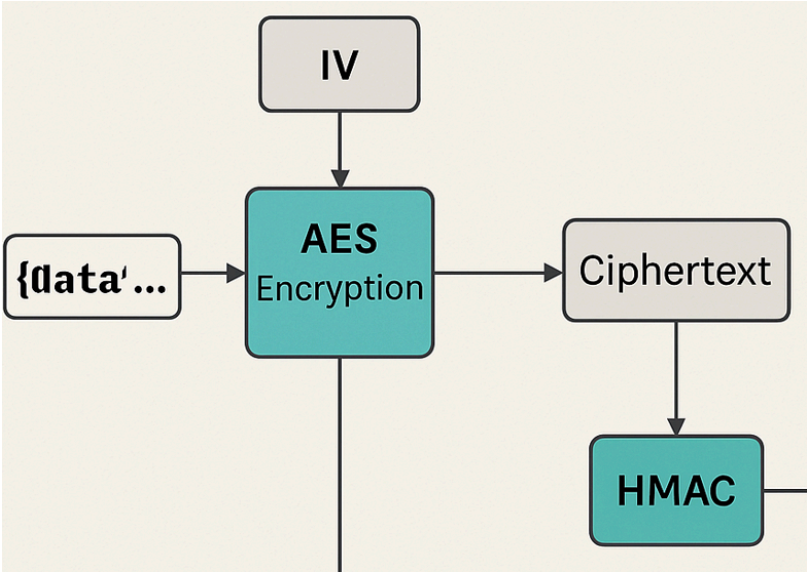
No entanto, ao utilizar uma única chave de sessão para ambas as funções criptográficas, AES e HMAC, identificámos uma vulnerabilidade potencial: se por algum motivo essa chave fosse comprometida, todo o protocolo ficaria exposto, uma vez que a mesma chave estaria a ser usada tanto para cifrar como para autenticar as mensagens. Para resolver esta limitação, utilizamos novamente o algoritmo Argon2, desta vez no seu modo de derivação de chaves, disponível através da biblioteca cryptography em Python.

Este mecanismo permite derivar, a partir da chave de sessão original, duas chaves distintas através da introdução de um salt partilhado entre o cliente e o servidor . O processo gera uma chave para a cifra AES-CBC e outra chave independente para o HMAC, garantindo assim a separação de responsabilidades.

O funcionamento desta derivação está documentado na documentação oficial do Argon2id: KDF: [Key derivation functions — Cryptography 45.0.0.dev1 documentation]

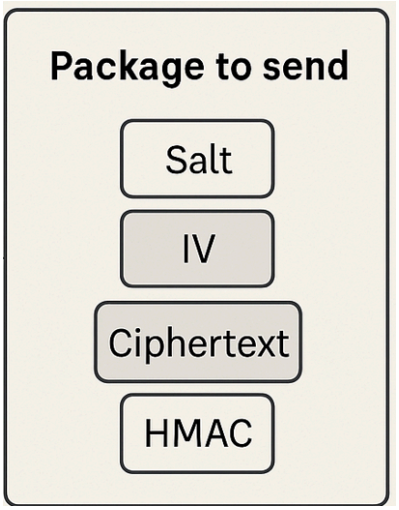
Foi exatamente o que implementámos: a chave derivada de 32 bytes foi dividida em duas partes, sendo os primeiros 16 bytes utilizados para AES e os restantes 16 para HMAC.

Tendo já chaves diferentes para os mecanismos que vamos utilizar fizemos a seguinte imagem para entender-se melhor o processo:



O processo seguido foi o seguinte: primeiro, cifrámos com a chave AES o pacote que queríamos enviar por exemplo, no caso da criação de conta (create account), enviamos o nome da pessoa, o saldo inicial e o hash da sua palavra-passe (armazenado no ficheiro .card). Juntamente com estes dados, também enviamos um nonce aleatório de 16 bytes, que será posteriormente verificado num dicionário no lado do Bank, garantindo a unicidade do pacote e prevenindo ataques de replay.

Como é óbvio os nonces não são 100 por cento aleatórios mas através de uns cálculos conseguimos chegar a conclusão que em 10 000 nonces a probabilidade de colisão anda muito próxima dos 0 , 7.0*10^-4..



Após a cifragem do pacote com AES, foi calculado o HMAC com a chave derivada para esse fim. Este HMAC cobre os três elementos críticos da comunicação: o IV do AES, o salt usado na derivação das chaves e o ciphertext resultante da cifra. Desta forma, garantimos a integridade e autenticidade desses três elementos.

Tanto o IV como o Salt não precisam de ser secretos, apenas únicos e imprevisíveis

Por fim, enviamos um dicionário com o conteúdo da última imagem, obviamente enviado em String de bytes em JSON.

3.1.1 Bank

Quando o Bank recebe um pacote do ATM, o primeiro passo é extrair o campo saltDerivation enviado em claro. Com este salt e a chave de sessão, o servidor utiliza o algoritmo Argon2id, com os mesmos parâmetros utilizados pelo cliente, para derivar duas chaves: uma para AES e outra para HMAC.

De seguida, o Bank verifica a validade do HMAC. Para isso, reconstroi o HMAC, aplicando a chave derivada sobre a concatenação dos campos *saltDerivation, iv* e *cyphertext*. Se o valor obtido for diferente do HMAC recebido, o pacote é considerado inválido

Este processo garante que nenhuma destas três variáveis essenciais foi alterada durante a transmissão. Salt IV e Texto cifrado.

Finalmente deciframos com AES obtendo o nosso pacote, que varia conforme o que o cliente mandou.

No caso da operação -n vamos ter o nosso nounce, o campo action *create* o *account* com o nome associado o *balance* e o *hashPIN*. A partir daí procedemos a inclusão do nome da

pessoa num dicionário como a key e os values o saldo imposto e por fim o HashPIN que valida a password do user.

Nas operações deposit e withdraw, o pacote enviado pelo ATM inclui o valor (amount), o hashPIN e um nonce aleatório. Já na operação get_balance, apenas é enviado o hashPIN e o nonce, uma vez que não há valores a transferir. Todas as validações para garantir que a operação é válida e autêntica são feitas do lado do Bank. Verifica-se, por exemplo, se a conta existe, se o HashPIN corresponde ao registado, se o valor enviado é positivo (quando aplicável) e se o nonce já foi utilizado.

Ainda incluímos locks para quem aceda a conteúdo partilhado, como por exemplo o nosso set() de *nounces* ou o dicionário dos *account*. No caso do set() de nonces, usamos um lock global que protege tanto a verificação como a adição de novos nonces, evitando colisões ou verificações simultâneas incorretas. Para a criação de contas (create), aplicámos também um lock global, uma vez que esta operação pode modificar a estrutura global do dicionário de contas.

Já nas operações deposit, withdraw e get_balance, optámos por uma abordagem mais eficiente, foi aplicado um lock individual por conta, garantindo que apenas a conta em questão fica bloqueada durante a operação. Esta abordagem tira partido do facto de os dicionários em Python permitirem acesso concorrente a chaves diferentes de forma segura, desde que haja um lock

Finalmente, implementámos um mecanismo de rollback para garantir a consistência do sistema caso ocorra uma falha no envio da resposta cifrada ao cliente (sendall). Antes de retornar a resposta, é guardado o estado anterior da conta (rollback_balance) e enviado no dicionário de resposta. A resposta é então preparada e enviada dentro de um bloco try, dentro do try criamos uma cópia e retiramos o rollback não nos interessa enviar para o cliente, mas guardamos esse valor caso falha a comunicação e falhe o sendall().

Se ocorrer uma exceção durante o envio, obtém- se o valor guardado do rollback.

Por fim, o Bank.py fecha com o SIGNALTERM e com SIGNALINT(ctrl+c) por simplicidade, o nosso mecanismo foi guardar as threads geradas no thread.start() em cada conexão para depois dar join quando executar o SIGTERM. Ou seja, espera que as threads todas finalizem e só depois fecha o server,para não haver problemas com as transações.

3.1 .1 AES CBC

Como podemos ver na imagem abaixo (Figura 1), o modo CBC encadeia cada bloco com o anterior, utilizando um vetor de inicialização (IV) aleatório na primeira operação. No nosso sistema, esse IV é gerado aleatoriamente no cliente para cada mensagem cifrada com AES, conforme definido na função aesHMAC encrypt(). Este mecanismo previne padrões repetitivos no ciphertext, mesmo que o conteúdo da mensagem seja idêntico.

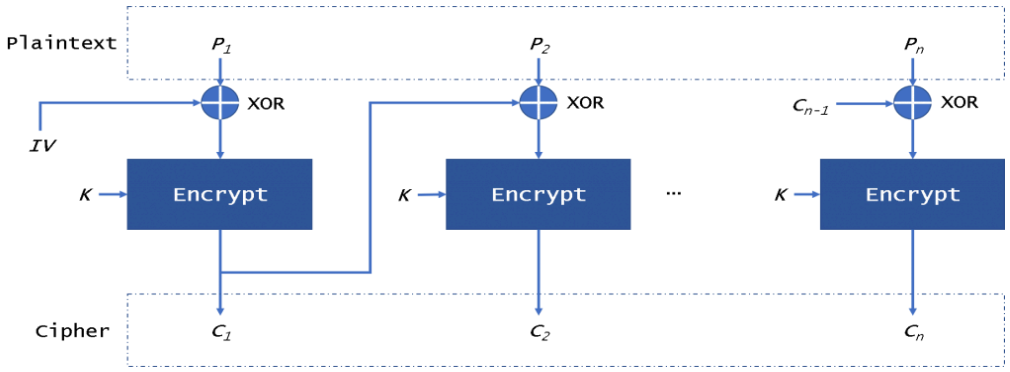


Figura 1- Ilustração do modo de operação CBC usado na cifragem simétrica com AES

O modo CBC revelou-se uma boa escolha para cifrar os nossos pacotes, tanto pela sua eficácia como pela sua ampla documentação e compreensão por parte do grupo. É verdade que existem modos de operação mais modernos, como o AES-GCM, que combinam confidencialidade, integridade e autenticidade num único passo. Mas por maior conhecimento no CBC e por garantias práticas optámos por este.

4. Mecanismos de Defesa Contra Ataques

Neste projeto, foram integradas várias camadas de defesa contra os ataques mais comuns em sistemas de comunicação cliente-servidor de modo a garantir a integridade, confidencialidade e disponibilidade das operações.

4.1 Proteção contra Replay Attacks

Para mitigar ataques do tipo *replay*, foi implementado um mecanismo baseado em **nonces e timestamps**. Cada mensagem trocada entre o cliente e o servidor incorpora um valor único (nonce) (32 bytes) e um timestamp que inclui ano, mês, dia, hora e segundo, dando margem de 10 segundos para a recepção, garantindo que cada operação seja distinta e não repetível. Esta estratégia previne que um atacante intercepte e volte a enviar uma mensagem válida anteriormente, que, de outra forma, poderia causar ações indesejadas (ex. duplicar transações).
Para esta implementação dos nonces foi tomado como máximo, antes de limpar o conjunto de nonces, dez mil nonces.

4.2 Proteção contra Ataques de Negação de Serviço (DoS)

O servidor implementa dois mecanismos principais para proteção contra DoS:

- **Limitação de Tamanho de Mensagens:** Para evitar que um cliente malicioso sobrecarregue o sistema com mensagens excessivamente grandes, criamos um método recv_all que tem como limite máximo de bytes por mensagem 2048.
- **Controlo de Concorrência com Semáforos (thread throttling):** Apesar de utilizarmos listen(20) no nosso socket do Bank, esta instrução apenas define o tamanho da fila de espera para ligações pendentes ou seja, o número máximo de clientes que podem estar à espera de serem aceites.

Funciona como um buffer temporário, evitando que o sistema seja sobrecarregado por pedidos simultâneos, mas não limita o número de threads ativas a processar clientes.

Para controlar efetivamente o número de clientes que estão a ser processados em simultâneo, foi necessário implementar uma restrição adicional ao nível das threads. Para isso, utilizamos um semaphore com o valor máximo de 20, definido como self.semaforo = threading.Semaphore(20).

Este semáforo é utilizado com a instrução "*with*", que funciona como um try/finally implícito: quando uma nova thread entra, faz acquire() automaticamente; quando termina, liberta a vaga com release(). Desta forma, garantimos que nunca existem mais do que 20 threads ativas em simultâneo a aceder aos recursos do Bank.

- **Timeouts:** Criamos dois timeouts um para o server e outro para o Bank que vai ser incluído no início do nossa funcão receive_all(), que se o Bank ou o ATM não receber nada em 10 segundos um recebe protocoll_error e o ATM retorna o erro 63. Previne por exemplo ataques de negação de serviços por estagnação.

4.3 Proteção contra Corrupção de Dados em Ambiente Concorrente

As operações críticas do servidor, como depósitos, levantamentos e verificações de saldo, estão sujeitas a condições de concorrência. Para garantir consistência, o sistema integra **locks** que sincronizam o acesso às contas bancárias e a todos os

Este tipo de proteção é essencial para evitar **race conditions**. No nosso trabalho otimizamos para não serem só locks globais e quando necessário ser só por contas.Tentamos também, limitar ao máximo todas as variáveis globais para haver as menores possíveis, do lado do Bank que são as mais vulneráveis e críticas a acesso múltiplo, trocamos para variáveis locais.

4.4 Criptografia e Autenticidade das Mensagens

Embora este aspeto seja detalhado no ponto 3 do relatório, é importante salientar que a própria utilização de criptografia assimétrica (RSA) para troca da chave de sessão e a posterior utilização de criptografia simétrica (AES em modo CBC), juntamente com **HMACs para validação de integridade**, representam um mecanismo de defesa forte contra:

- **Ataques Man-in-the-Middle (MitM)**;
- **Alterações maliciosas de mensagens em trânsito**;
- **Interseção passiva de dados sensíveis**.

A utilização do *padding OAEP* em RSA, por sua vez, protege contra ataques de texto cifrado escolhido (*chosen ciphertext attacks*), elevando substancialmente o nível de robustez criptográfica do sistema.

5. Instalação e Execução do Sistema:

pip install argon2-cffi

pip install cryptography

8. Conclusão

O sistema desenvolvido cumpre os requisitos definidos para a fase *Build It* do projeto ATM Communication, assegurando os pilares fundamentais da segurança: confidencialidade, integridade e disponibilidade . Através da combinação de técnicas modernas como RSA com OAEP, AES-CBC com HMAC-SHA256, derivação de chaves com Argon2id e autenticação local (.card), garantimos uma comunicação segura entre cliente e servidor, mesmo em cenários com atacante ativo e passivo

Foram também implementados mecanismos de defesa contra ataques comuns como *replay attacks*, *denial-of-service* e concorrência simultânea. O sistema foi desenhado de forma modular e defensiva, com validações extensivas tanto ao nível dos inputs como da comunicação entre processos.

Em suma, acreditamos que a nossa implementação apresenta um equilíbrio sólido entre segurança, eficiência e clareza, estando preparada para resistir a ataques na fase *Break It*..

9. Referências

Porque usar OAEP com RSA:
So How Does Padding Work in RSA?. Basically, PKCS#v1.5 is bad, OAEP is… | by Prof Bill Buchanan OBE FRSE | ASecuritySite: When Bob Met Alice | Medium

OEP COM RSA code:
RSA — Cryptography 45.0.0.dev1 documentation

password Contest:
https://www.password-hashing.net/

Argon use:
argon2-cffi · PyPI

Deriving a password in two: Key derivation functions — Cryptography 45.0.0.dev1 documentation

generate a random safe PIN: secrets — Generate secure random numbers for managing secrets — Python 3.13.2 documentation

AES: Classic modes of operation for symmetric block ciphers — PyCryptodome 3.22.0 documentation

https://stackoverflow.com/questions/37459584/reading-one-byte-vs-multiple-bytes-from-a-socket/37460123#37460123

timeout:
https://docs.python.org/3/library/socket.html#socket.socket.settimeout

Padding process of OAEP:
https://www.researchgate.net/publication/361293348_An_Overview_of_RSA_and_OAEP_Padding