Login Seguro Tkinter — Detalhe das Funcionalidades de Segurança

Objetivo

Este documento descreve, em detalhe, as decisões técnicas adotadas no exemplo de autenticação **segura** em Tkinter, explicando o racional de segurança, boas práticas e armadilhas a evitar.

---

1) Hashing de passwords com PBKDF2-HMAC-SHA256 e *salt* por utilizador

O PBKDF2 (Password-Based Key Derivation Function 2) aplica um número elevado de iterações a uma password combinada com um *salt* único, produzindo um **hash lento** que dificulta ataques por força bruta e tabelas arco-íris.

Porquê SHA-256. Fornece maturidade, suporte alargado e segurança adequada, mas em produção, considera-se frequentemente **Argon2id** (recomendado) ou **bcrypt**.

Parâmetros (exemplo didático).

- **iterations**: 200_000 (ajusta consoante hardware/latência desejada; quanto maior, mais caro para o atacante).
- **dklen**: 32 bytes (256 bits).
- **salt**: 16 bytes (128 bits), único por utilizador.

Boas práticas.

- Nunca guardar passwords em claro nem hashes **sem salt**.
- **Um salt por utilizador** (duas contas com a mesma password geram hashes diferentes).
- Versionar parâmetros (iterations, dklen) e prever **migração de parâmetros** no futuro (ex.: aumentar iterações).

Padrão de armazenamento. Guardar salt e hash em **Base64**, mais iterations, no registo do utilizador.

---

2) Geração de *salt* com secrets.token_bytes

O que é. secrets.token_bytes(n) devolve **n bytes** com entropia criptográfica adequada para *salts*, chaves e *nonces*.

Porquê não usar random. O módulo random não é criptograficamente seguro; é previsível e **inadequado** para proteger segredos.

Tamanho do *salt*. 16 bytes (128 bits) é suficiente para inviabilizar pré-cálculos e colisões práticas; valores superiores são aceitáveis.

Persistência e privacidade. O *salt* **não precisa de ser secreto**; deve ser guardado com o hash para permitir a verificação futura.

---

3) Comparação em tempo constante com hmac.compare_digest

O problema. Comparações triviais (==) podem demorar mais ou menos consoante o ponto de divergência, expondo um **canal lateral de temporização** (*timing side-channel*).

A solução. hmac.compare_digest(a, b) faz a comparação de forma a reduzir a correlação entre **tempo de execução** e **conteúdo**, mitigando ataques de *timing*.

Complemento. Combina-se com **hashing "dummy"** quando o utilizador não existe (ver ponto 6) para manter tempos de resposta semelhantes e **evitar enumeração** de contas.

---

4) *Lockout* simples e *backoff* exponencial

Objetivo. Dificultar **brute force**, limitando o número de tentativas por janela temporal.

Mecanismo (didático). Após MAX_ATTEMPTS falhadas, bloquear a conta por BASE_LOCK_SECONDS. Em reincidência, **duplicar** o tempo de bloqueio (ex.: 30s → 60s → 120s).

Considerações.

- **Persistir** bloqueios (BD/ficheiro) para sobreviver a reinícios.
- **Auditar** eventos (tentativas, IP/host, timestamps) e **monitorizar** alertas.
- Mensagens **genéricas** ("Credenciais inválidas" vs detalhes) para reduzir enumeração.

Experiência do utilizador. Informar de forma clara e sucinta que existe um bloqueio temporário e, **opcionalmente**, o tempo restante.

---

5) Criação de utilizadores via CLI com getpass()

Risco. Passar passwords em argumentos de linha de comandos deixa **rasto** no histórico do shell e em ferramentas de monitorização.

Solução. getpass() lê a password **sem eco** no terminal. Em seguida, **hash** imediato e limpeza de variáveis reduz exposição.

Boas práticas adicionais.

- Aplicar **regras mínimas** (comprimento, classes de caracteres, etc.).
- **Confirmação** da password (dupla introdução).
- Evitar **logs** com dados sensíveis (nunca logar a password ou o hash).

---

6) Persistência atómica do ficheiro users_secure.json

Problema. Escritas interrompidas (quedas de energia, *crash*) podem **corromper** o ficheiro.

Solução. Padrão **atómico**: escrever primeiro para users_secure.json.tmp e depois substituir o ficheiro final com os.replace(...). Isto reduz a probabilidade de corrupção e garante **consistência**.

Boas práticas.

- Definir **permissões restritivas** no ficheiro (apenas leitura/escrita pelo serviço).
- Considerar **encriptação** em repouso (por exemplo, volume cifrado).
- Validar e **tratar erros de I/O**; nunca assumir que a escrita foi bem-sucedida.

---

7) Integração dos mecanismos — *workflow* seguro de autenticação

1. O utilizador introduz credenciais na GUI.
2. A aplicação verifica **lockout** para o username (se ativo, falha e informa tempo restante).
3. Carrega o registo do utilizador; **se não existir**, calcula um hash **dummy** para nivelar tempos.
4. Deriva o hash com **PBKDF2** usando o *salt* e iterações armazenadas.
5. Compara com hmac.compare_digest.
6. Em sucesso, limpa contadores; em falha, incrementa contadores e ativa bloqueio quando aplicável.

---

8) Recomendações de evolução (produção)

- **Argon2id** (ou bcrypt): melhor resistência a GPU/ASIC e parametrização por **memória** e **tempo**.
- **Persistência** de lockouts e **auditoria** centralizada (SIEM).
- **MFA (2FA)**: TOTP/U2F para elevar o nível de segurança.
- **TLS** ponta-a-ponta se a GUI falar com backend remoto; **separar** GUI e serviço de identidade.
- **Proteção do ficheiro** de utilizadores: encriptação em repouso e **permissões** bem definidas.