Tkinter Login Inseguro — Explicação Técnica (PT-PT)

1) Como o código funciona (resumo)

- A interface (Tkinter) tem campos de **Utilizador** e **Palavra-passe** e dois botões (**Login/Sair**).
- Ao clicar em **Login**, a função on_login() lê os valores e chama verificar_credenciais().
- verificar_credenciais() compara o par (username, password) com um **dicionário local** CREDENTIALS.
- Se coincidir, mostra mensagem de sucesso; caso contrário, erro genérico.

2) Principais vulnerabilidades

a) Credenciais "hardcoded" (texto claro)
Passwords escritas diretamente no código-fonte; ficam expostas a quem ler o ficheiro, o executável ou o histórico Git.

b) Verificação sem hashing/salting
Compara-se a password **em claro**; não há PBKDF2/Argon2/Bcrypt. Um compromisso do ficheiro expõe todas as passwords.

c) Sem proteção contra brute force
Não existe limite de tentativas, **lockout** temporário nem **backoff exponencial**. Facilita ataques automatizados.

d) Sem auditoria/logs de segurança
Não há registos de tentativas falhadas, origem (IP/host), horários, etc. Dificulta deteção de ataques e resposta a incidentes.

e) Dados sensíveis em memória sem proteção
A password é lida como string normal e mantida em memória. A máscara show="*" **é apenas visual**, não protege em memória.

f) Ausência de camadas e backend seguro
Toda a autenticação está no **cliente (GUI)**; facilita engenharia reversa e exposição do "segredo" (credenciais).

3) Impacto prático

- **Exposição de credenciais** e acessos não autorizados.
- Reutilização de passwords em outros sistemas (risco ampliado).
- Dificuldade em detetar e investigar incidentes (sem logs/auditoria).
- **Riscos legais** (ex.: RGPD) se envolver dados pessoais.

4) Mitigações e boas práticas (o que fazer corretamente)

1. **Hashing com salt e KDF robusta** (PBKDF2, Argon2, Bcrypt) + comparação em **tempo constante**.
2. **Não guardar passwords em claro**; usar armazenamento seguro **fora do código** (ficheiro/BD com gestão adequada).
3. Implementar **rate limiting/lockout** e **backoff exponencial**.
4. **Auditoria e logs** de segurança com retenção e monitorização.
5. Mensagens de erro **genéricas** (evitar enumeração de utilizadores).
6. Limpar dados sensíveis de memória **quando fizer sentido**.
7. **Separação de responsabilidades**: GUI/cliente + **serviço de autenticação** (backend) sob **TLS**.

5) Caminho para uma versão segura (referência didática)

- Implementar **PBKDF2-HMAC-SHA256** com salt único e iterações elevadas.
- Usar hmac.compare_digest para comparação em tempo constante.
- Adicionar **lockout** após N tentativas e registos de auditoria.
- Criar utilizadores via CLI/GUI própria (sem embutir passwords no código).

6) Conclusão

Em produção, é obrigatório aplicar hashing com salt, armazenar credenciais de forma segura, **limitar tentativas**, **registar eventos** e isolar a autenticação num **serviço seguro** com transporte **cifrado**.