Segurança da Informação

Um apanhado geral sobre criptografia, acesso e vulnerabilidades, incluindo exemplos práticos em Python e Shell

Danilo J. S. Bellini
@danilobellini

Etec
Uirapuru
São Paulo

2018-10-18

Semana de informática
ETEC Uirapuru – SP

O que é segurança?

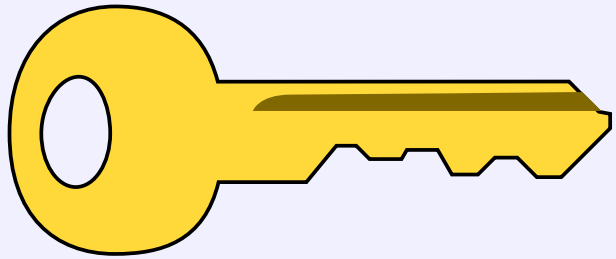
Estar livre de perigos? Minimizar os riscos?

Em inglês há duas palavras: safety VS security[1]

- *Safe* refere-se à proteção sobre acontecimentos indesejáveis do acaso;
- *Secure* refere-se à proteção contra acontecimentos intencionais. É aqui que se encaixa a segurança da informação.

Aspectos da segurança da informação:

- Confidencialidade / Privacidade
- Integridade
- Disponibilidade



[1] Para uma definição mais completa dessas palavras, veja http://www.iot.ntnu.no/users/albrecht/rapporter/notat%20safety%20v%20security.pdf

Criptografia

Criptografia é a prática e o estudo das técnicas de armazenamento e comunicação de informação na presença de terceiros/adversários.

Técnicas tradicionais incluem:

- Criptografia de chave simétrica (chave única, de conhecimento exclusivo das partes)
- Criptografia de chave pública (pares de chave pública/privada para cada parte)
- Funções de *hash* (sem chave)

Objetivos:

- *Sign*, assinatura (integridade)
- *Encrypt/Decrypt*, encriptação/decriptação (confidencialidade)

Cifras de César, Vigenère e autochave

Cifra de César (possui 1 parâmetro)

Mensagem:	EU GOSTO DE LIMONADA
Chave:	C (desloca 2 no alfabeto)
Texto cifrado:	GW IQUVQ FG NKOQPCFC

Cifra de Vigenère (chave simétrica)

Mensagem:	EU GOSTO DE LIMONADA
Chave:	MENTIRA
Chave efetiva:	ME NTIRA ME NTIRAMEN
Texto cifrado:	QY THAKO PI YBUFNMHN
Chave:	PAGAZAQ
Chave efetiva:	PA GAZAQ PA GAZAQPAG
Texto cifrado:	TU MORTE SE RILODPDG

Autochave (chave simétrica)

Mensagem:	EU GOSTO DE LIMONADA
Chave:	PAGAZAQP
Chave efetiva:	PA GAZAQ PE UGOSTODE
Texto cifrado:	TU MORTE SI FOAGGOGE

A cifra de César transforma cada caractere da mensagem usando esta tabela:

A →C	B →D	C →E	D →F	E →G	F →H
G →I	H →J	I →K	J →L	K →M	L →N
M →O	N →P	O →Q	P →R	Q →S	R →T
S →U	T →V	U →W	V →X	W →Y	X →Z
Y →A	Z →B				

```
# Código das funções no próximo slide
msg = "EU GOSTO DE LIMONADA"
cesar(msg, key="C")
vigenere(msg, key="MENTIRA")
vigenere(msg, key="PAGAZAQ")
autokey(msg, key="PAGAZAQP")
```

Desafio: escrever uma função para cada algoritmo de cifra que encontra o texto original dados a chave e o texto cifrado.

Cifra de César, Vigenère e autochave em Python

```python
from itertools import cycle, accumulate
from string import ascii_uppercase as alphabet

def shift_table(key):
    idx = alphabet.find(key)
    return str.maketrans(alphabet, alphabet[idx:] + alphabet[:idx])

def cesar(msg, key):
    return msg.translate(shift_table(key))

def vigenere(msg, key):
    parts = msg.split() ; joined = "".join(parts)
    chars = [ch.translate(shift_table(k))
             for ch, k in zip(joined, cycle(key))]
    positions = accumulate(map(len, parts[:-1]))
    for end in list(positions)[::-1]:
        chars.insert(end, " ")
    return "".join(chars)

def autokey(msg, key):
    parts = msg.split() ; joined = "".join(parts)
    chars = [ch.translate(shift_table(k))
             for ch, k in zip(joined, key + joined)]
    positions = accumulate(map(len, parts[:-1]))
    for end in list(positions)[::-1]:
        chars.insert(end, " ")
    return "".join(chars)
```

Chave pública/privada

Analogia:

> *Envio pelo correio um cadeado aberto sem a chave, o destinatário recebe, tranca um pacote com o cadeado e envia p/ mim.*

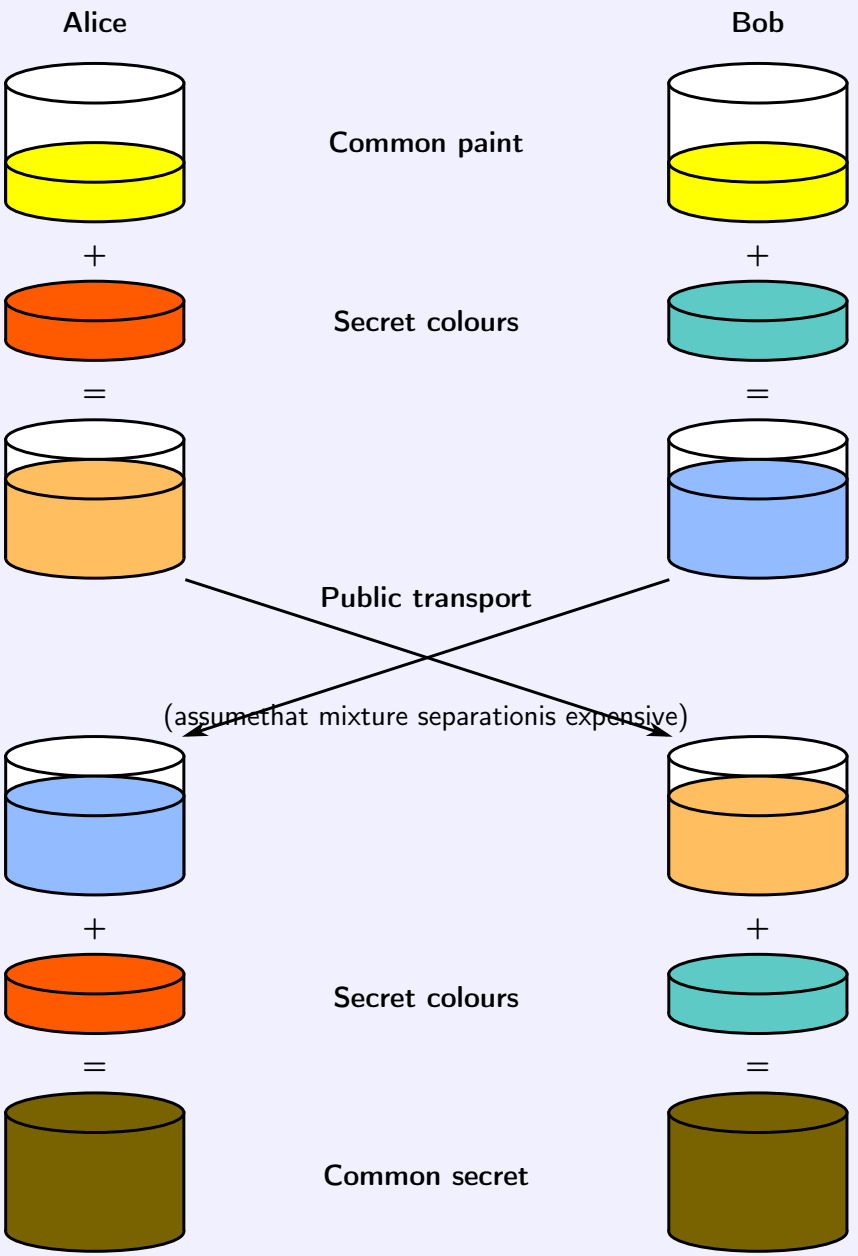
Nesse exemplo, o cadeado desempenha o papel de chave pública, e a chave do cadeado desempenha o papel de chave privada.

Vamos ver:

- um algoritmo de chave pública/privada utilizado para troca de chave simétrica
- um exemplo prático de uso de chave pública/privada no OpenPGP[2].

[2] PGP significa *Pretty Good Privacy*, um software comercial criado em 1991. Sua segunda versão tornou-se o [hoje obsoleto] padrão RFC1991. A menos de uma atualização relativa ao algoritmo Camellia, RFC4880 (OpenPGP) é a versão mais recente do padrão.

Diffie-Hellman: Exemplo de algoritmo para troca de chave



Número primo (público):	p
Base (público):	g
Chave privada da Alice:	a
Chave pública da Alice:	$A = g^a \mod p$
Chave privada do Bob:	b
Chave pública do Bob:	$B = g^b \mod p$
Número compartilhado:	$A^b \equiv B^a \mod p$

Exemplo:

```
# Alice e Bob combinam os parâmetros
p = 2551
g = 2

# Criam chaves privadas em silêncio
a = 45
b = 29

# Calculam e trocam chaves públicas
A = (g ** a) % p          # 2285
B = (g ** b) % p          # 207

# Alice e Bob possuem um segredo!
secret = (A ** b) % p     # 414
secret = (B ** a) % p     # 414
```

GPG – *GNU Privacy Guard*

O GPG é uma implementação em software livre (GPLv3) que atende ao OpenPGP sem utilizar softwares/algoritmos patenteados/restritos/privados.

```
gpg --gen-key # Criar chaves (par público/privado)
gpg -k        # Visualizar chaves disponíveis

# Exportando/importanto chaves públicas (para um dado keyID)
gpg --keyserver pgp.mit.edu --send-keys keyID
gpg --keyserver pgp.mit.edu --recv-keys keyID
gpg --export --armor danilo.bellini@gmail.com > my.key
gpg --import my.key

# Assinatura
gpg -b message.txt                          # Assina (cria ".sig")
gpg --verify message.txt.sig message.txt # Verifica assinatura

# Codificando/criptografando (crypt) p/ um destinatário específico
gpg -o encrypted.gpg -r danilo.bellini@gmail.com -e message.txt

# Decodificando/decriptografando (decrypt) com a chave privada
gpg -o decrypted.txt -d encrypted.gpg

# Comandos utilizados na apresentação (além de vim, cat, rm, ...)
seq 15 > message.txt     # Cria message.txt c/ sequência de 1 a 15
hexdump -C encrypted.gpg # Visualizando arquivos como binários
```

Tomb: armazenamento criptografado em um arquivo

Discos rígidos, SSDs, SDs e outras mídias normalmente não estão criptografados. *Tomb* permite criarmos diretórios criptografados com uma chave criptografada com GPG escondida em uma imagem.

```
tomb dig new.tomb -s 50           # Cria o new.tomb com 50MB
tomb forge new.key -g             # Cria uma chave encriptada com GPG
tomb lock new.tomb -k new.key -g # Atribui a chave e formata o new.tomb

tomb open new.tomb -k new.key -g # Monta o new como se fosse um pendrive
tomb close new                    # Desmonta o new

# Esteganografia
cp EnigmaMachine.jpg new.jpg      # Cópia para não perder a imagem original
tomb bury new.jpg -k new.key -g  # Armazena chave (+ senha) na imagem
tomb exhume new.jpg -k copy.key  # Extrai a chave da imagem
tomb open new.tomb -k new.jpg -g # Podemos usar a imagem direto
```

Quem imaginaria que a imagem ao lado poderia conter uma chave?
Use senhas fortes (mais sobre isso em breve)!

Hash, funções de espalhamento ou dispersão criptográfica

Um valor ou código *hash* (também conhecido como *digest* ou "resumo") é o resultado de uma função de *hash* para uma dada mensagem. No contexto da criptografia, funções de *hash* são:

- Determinísticas (mesma mensagem $\Rightarrow$ mesmo *hash*)
- "Caóticas" (pequenas mudanças na entrada mudam o hash completamente)
- Extremamente difíceis de reverter (somente podemos obter a mensagem a partir do hash por tentativa e erro)
- Extremamente difíceis de colidir (idealmente nunca encontramos mensagens diferentes com o mesmo *hash*)

Exemplos de algoritmos:

- Checksum e dígitos verificadores (CPF, RG, ISSN, etc.)
- MD5 (foi quebrado, útil para fins didáticos)
- SHA-1 (usado em torrents, git, etc.; teve uma custosa colisão intencional de hashes de PDFs no *SHAttered*)
- SHA-2 (há 6 funções diferentes: SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224, SHA-512/256)

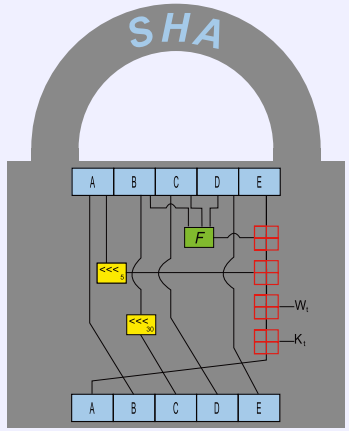
Hashes no Linux

```
$ echo "O segredo é *#(U*#E, não conta para ninguém" > msg.txt

$ md5sum msg.txt
30a4e40dc44b8d1579a086f3e54cb165  msg.txt

$ sha1sum msg.txt
df9da1b3ec550a0ae39e70c3d455940fbd74efa1  msg.txt

$ sha224sum msg.txt
99ef0cc5ba6a58bbee572cc2527dc469d8c9ea0998885ebe0e558c0b  msg.txt

$ sha256sum msg.txt
0b9b5d4b7dcee9fa58653e4e782abf8f40da985d77467f47c9e44848ad43b4d3  msg.txt

# Todos possuem validação com -c

$ md5sum -c <<< "30a4e40dc44b8d1579a086f3e54cb165 msg.txt"
msg.txt: OK

$ md5sum -c <<< "30a4e40dc44b8d1579a086f3e54cb167 msg.txt"
msg.txt: FAILED
md5sum: WARNING: 1 computed checksum did NOT match
```

Hashes em Python

```
>>> import hashlib                      # Biblioteca padrão
>>> with open("msg.txt", "rb") as msg_file:
...     msg = msg_file.read()
...
>>> hashlib.md5(msg).hexdigest()     # Hashes como strings comuns
'30a4e40dc44b8d1579a086f3e54cb165'
>>> hashlib.sha256(msg).hexdigest()
'0b9b5d4b7dcee9fa58653e4e782abf8f40da985d77467f47c9e44848ad43b4d3'
>>> hashlib.sha224(msg).hexdigest()
'99ef0cc5ba6a58bbee572cc2527dc469d8c9ea0998885ebe0e558c0b'
>>> hashlib.sha1(msg).hexdigest()
'df9da1b3ec550a0ae39e70c3d455940fbd74efa1'
```

Tá legal, verificar *hashes* valida a integridade da mensagem, seja ela um texto, um arquivo ou um pedaço de um arquivo... mas como é que isso pode ajudar na confidencialidade?



Certificados da Python Sudeste

Objetivo:

- Gerar certificados personalizados por tipo de participação (possivelmente mais de um certificado por pessoa)
- Autenticar certificados por parte de quem possui o código e outras informações fundamentais
- Código 100% *open source*
- Utilizar somente dados públicos, apenas em um front-end (não há banco de dados, o servidor é o de páginas estáticas no GitHub)
- Não tornar pública a lista de participantes (privacidade)

A solução adotada utilizou SHA-3. A lista de hashes que autenticam um certificado é pública, porém os códigos estão associados a outras informações necessárias para autenticar, e é extremamente custoso verificar as combinações por força bruta.

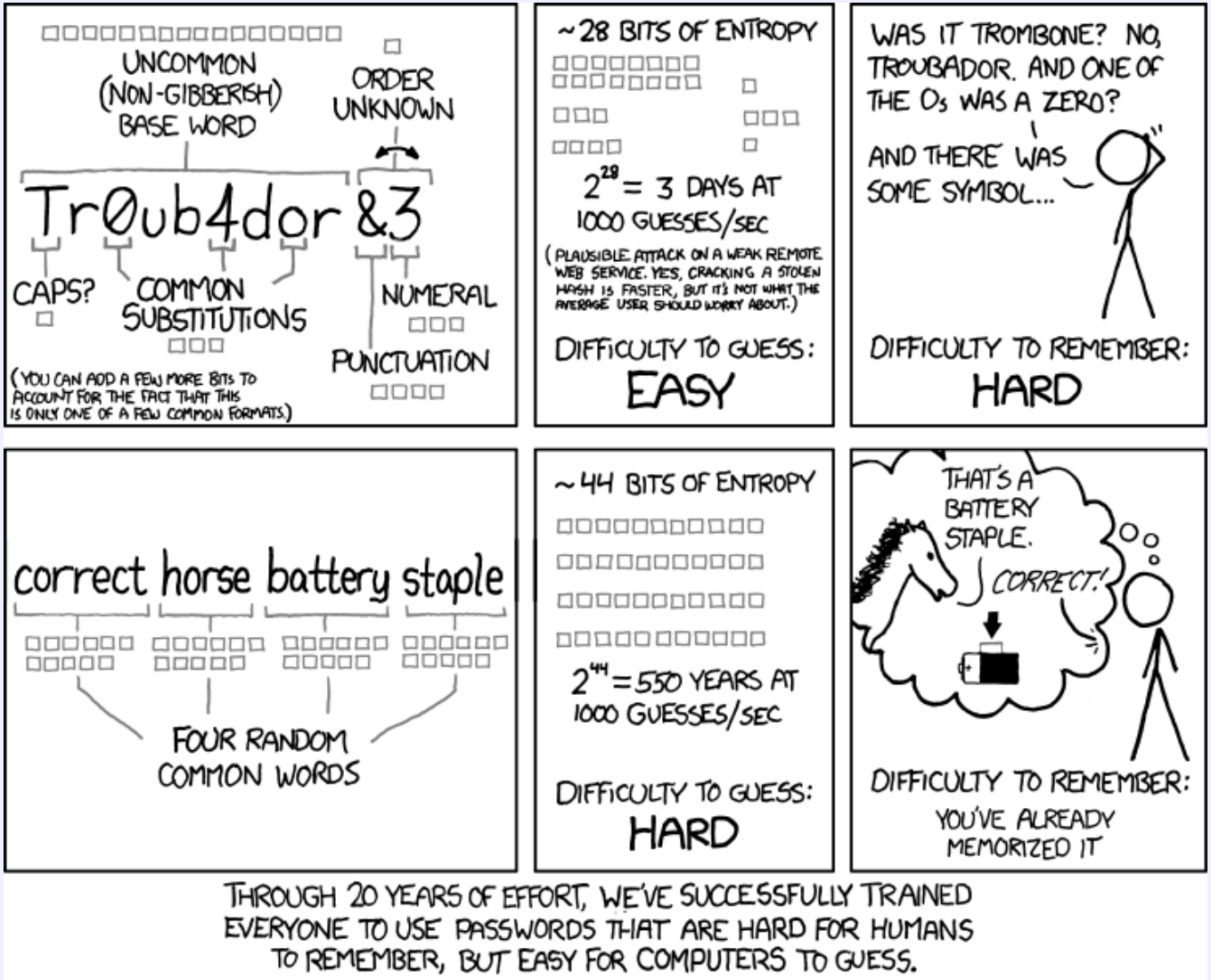
Senhas

Existem duas tarefas para as quais credenciais de acesso são necessárias:

- Autenticação: identificação de quem está solicitando
- Autorização: verificação o solicitante tem permissão de realizar determinada solicitação

Senhas "fortes" são importantes para, no mínimo, evitarmos o ataque por *brute force* (força bruta), que consiste em tentar adivinhar a senha por tentativa e erro, exaustivamente. Muitos sistemas bloqueiam repetidas tentativas de acesso e/ou dificultam o processo com CAPTCHAs, mas o ideal é não precisar depender disso.

Uma forma de ataque, chamada de *ataque de dicionário*, consiste em algo similar à força bruta, mas restrito àquilo que são consideradas escolhas mais prováveis: data de nascimento, data de casamento, algum evento considerado importante, nome de parente, nomes de animais, ideias, projetos, filmes/livros favoritos, etc..



https://www.xkcd.com/936/

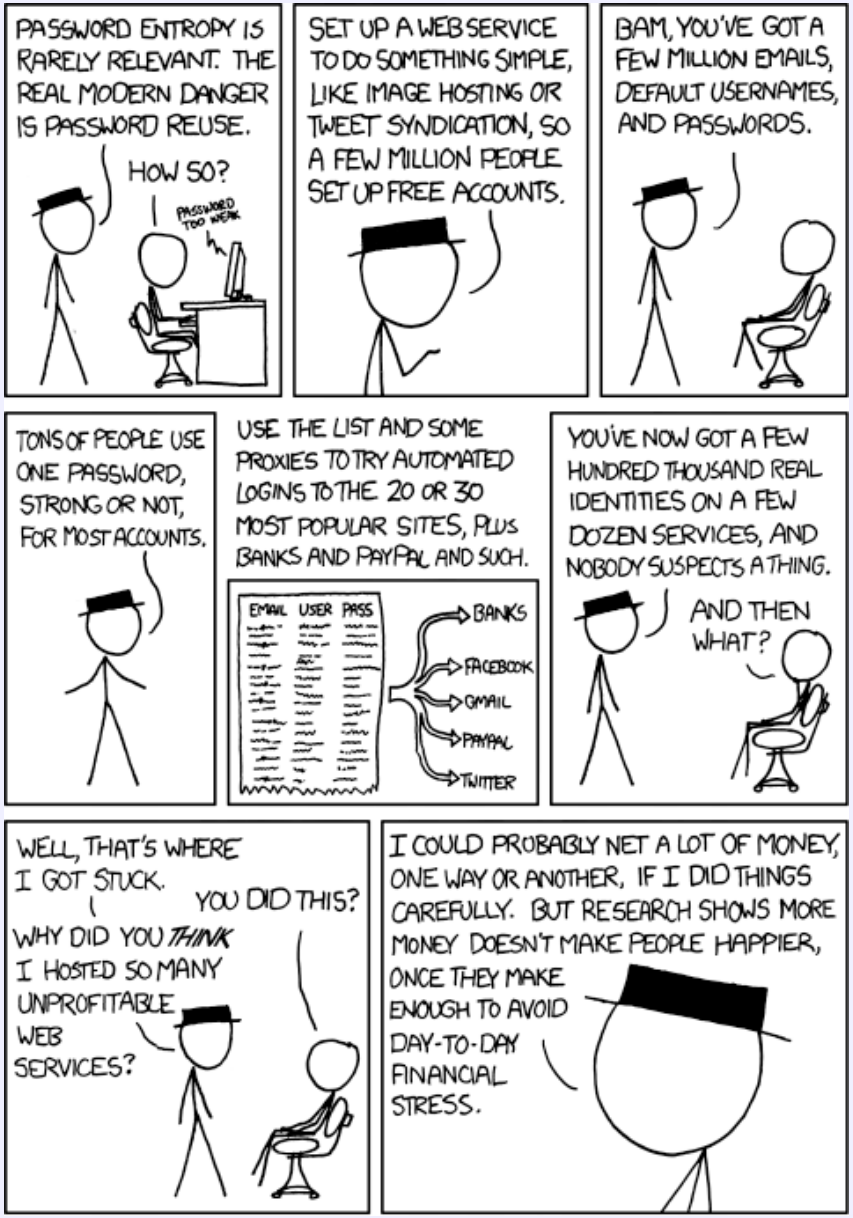
São $\approx$ 44 bits se considerarmos mensagens de 4 palavras em um universo de 2000 palavras equiprováveis

Políticas para escolha de senhas

- Não saia chutando todas as senhas que você usa, alguém pode estar gravando tudo
- Não deixe senhas trafegar a rede sem criptografia (verifique o "cadeado verde" no navegador)
- Nunca utilize a mesma senha em dois sistemas diferentes, do contrário correrá o risco de que alguém de um sistema consiga acessar sua conta do outro sistema

Memorizar muitas senhas fortes não é fácil. Quando precisamos trocá-las com frequência (o que não é recomendado), a tendência é que as enfraquecemos cada vez mais, ou adotamos procedimentos inapropriados:

- Enviar e-mail/mensagem com a senha para si próprio
- Colar um *post-it* na tela do computador com a senha



https://www.xkcd.com/792/

Armazenamento de senhas e *password managers*

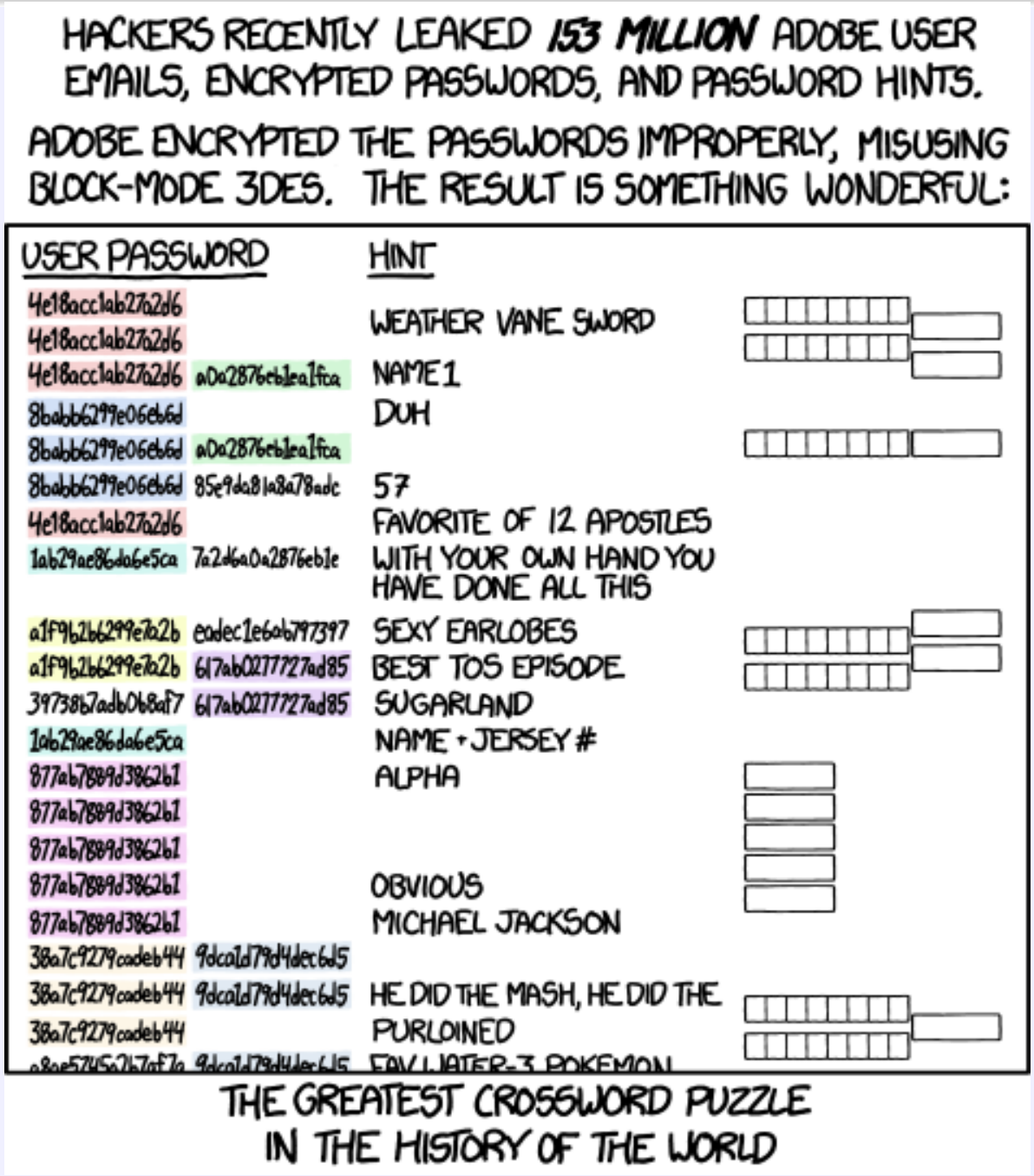
O que queremos com a senha?

- Validar para fins de autenticação ou autorização? Não precisamos da senha, precisamos apenas de um validador (*hash*).
- Fornecer para outro sistema? Preciso da senha!

Como poderíamos armazenar um validador de senhas? Apenas para validar, o ideal seria não guardar a senha.

Podemos usar funções de hash! A senha pode ser justaposta com um bloco constante (denominado *salt* ou *nonce*), e armazenamos apenas o hash do resultado. Se a mesma senha for fornecida duas vezes, o resultado precisa ser igual, o que basta para validar.

Porém, softwares *password managers* não têm alternativa, a não ser armazenarem a senha. Como nossas senhas são armazenadas, quem acessa, e como elas trafegam a rede? São realmente softwares em que podemos confiar?



https://www.xkcd.com/1286/

OTP: *one time password*

OTP são senhas de uso único, as quais posteriormente são descartáveis. Códigos enviados por SMS ou gerados por um aplicativo para liberar um acesso são exemplos de tais senhas.

Podem ser gerados de diferentes formas, mas costumam ser hashes sobre um token fixo e alguma informação adicional como:

- Instante atual (e.g. TOTP[3])
- Contador (e.g. HOTP[4])
- OTP anterior (chain)
- Aleatório (necessita armazenamento no servidor)

O mais comum é o TOTP de 6 dígitos[5].

---

[3] RFC6238, *Time-based OTP*, uma extensão do HOTP

[4] RFC4226, *HMAC-based OTP*

[5] Dígitos menos significativos, recomendação do apêndice E do RFC4226.

Pass: *The Standard Unix Password Manager*

Pass é um software livre (GPLv2+), um gerenciador de senhas com armazenamento das senhas em um repositório *git* local criptografado com GPG.

QRcode "fraco" p/ TOTP

```
pass init # Cria o repositório git em ~/.password-store
pass      # Listar árvore de nomes de arquivos no repositório (não senhas)

pass generate twitter        # Gerar senha no arquivo "twitter"
pass generate -n facebook 18 # 18 caracteres, sem caracteres especiais
pass generate -c email       # "-c" copia para a área de transferência

pass edit facebook # Edição da senha em editor
pass insert etec   # Inserção manual (também pode sobrescrever)

pass mv etec etec.uirapuru # Renomear (mover arquivo)
pass rm facebook           # Remover

pass email      # Acesso à senha
pass -c twitter # "-c" copia para a área de transferência

# TOTP (com qrencode, zbar e plugin pass-otp)
qrencode -o fake.png 'otpauth://totp/noone@nowhere?secret=AAAA&issuer=fake'
zbarimg -q --raw fake.png | pass otp insert fake
pass otp fake # Geração do TOTP (c/ -c copia na área de transferência)
```

2FA e MFA: *multi-factor authentication*

MFA significa usar mais de um fator durante um processo de autenticação. 2FA é o MFA com exatos 2 fatores.

"Verbos" categorizando possíveis fatores:

- "Conhecer": Senhas, PIN (*personal identification number*), questões sobre a vida/moradia da pessoa
- "Possuir": Cartões, hardware gerador de tokens, tabela de valores, número de telefone, conta de e-mail
- "Ser": Biometria (digital, voz, aparência), local de acesso

Tokens são códigos auto-gerados, normalmente temporários. Seu uso é similar ao das senhas, mas tradicionalmente categorizados com o verbo "possuir", em referência ao gerador dos tokens (mesmo que o gerador seja, em essência, apenas uma chave). OTPs são *tokens* que só podem ser usados uma vez.

Fallback, *bricking*, privacidade es disponibilidade no MFA

Códigos gerados dinamicamente podem dificultar um *brute force* exaustivo?

Depende! A "força" do TOTP está fundamentalmente no segredo usado como semente geradora (o QR code exibido), e ninguém deve conhecer esse segredo.

SMS não é seguro! Evitem ao máximo depender de SMS como fator.

MFA não é desculpa para deixar sua senha ser fraca!

Certos fatores de autenticação exigem o envolvimento de terceiros (e.g. empresa especialista em reconhecimento biométrico).

Bricking é a impossibilidade de utilizar um sistema devido ao bloqueio ao acesso.

E se eu esquecer a senha? E se meu celular que gera os OTPs quebrar ou for roubado? Essas são possibilidades de *bricking*, as alternativas nesses casos costumam ser:

- Solicitação de troca de senha usando o e-mail
- OTPs de *fallback* independentes do tempo
- Contato direto com pessoas (e-mail, telefone, presencial)

E se minha biometria for "roubada" (e.g. adesivos de impressão digital)? Não dá para trocar!

Malwares VS engenharia social

Nem tudo é vírus/*worm*! Nomes para não faltam para naturezas diferentes de software nocivos/maliciosos:

- *Spyware* / *Adware*
- *Phishing*
- *Trojan horse*
- *Ransomware* (cobrança de "resgate")

Qual é o alvo de um software desses (ou de algum "ataque") (pessoas, tecnologias, empresas)?

Qual é a finalidade?

Exemplo de ataque fundamentado apenas em engenharia social: `https://youtu.be/lc7scxvKQOo`

Vulnerabilidades na web

OWASP (*Open Web Application Security Project*) é uma comunidade online que produz e disponibiliza conteúdo técnico sobre segurança em aplicações web.

Em 2017, as 10 vulnerabilidades eleitas de maior risco foram:

Código	Descrição
A1:2017	Injeção de código
A2:2017	Quebra de autenticação
A3:2017	Exposição de dados sensíveis
A4:2017	Entidades externas de XML
A5:2017	Quebra de controle de acesso
A6:2017	Configuração incorreta de segurança
A7:2017	XSS: *Cross-Site Scripting*
A8:2017	Deserialização insegura
A9:2017	Utilização de componentes com vulnerabilidades conhecidas
A10:2017	Log e monitoramento ineficientes

Injection (Injeção de código)

```python
from flask import Flask, jsonify
app = Flask(__name__)

@app.route("/somar/<a>/<b>")
def somar(a, b):
    return jsonify({"result": eval(a) + eval(b)})
```

```
$ # Comando para rodar:
$ FLASK_APP=calc.py flask run --host 0.0.0.0 --port 1337
[...]

$ # Em outro terminal, mas no mesmo diretório:
$ curl localhost:1337/somar/2/2
{"result":4}
$ curl localhost:1337/somar/2*5/-4
{"result":6}
$ curl 'localhost:1337/somar/__import__("os").getcwd()/""'
{"result":"/home/danilo/code/slides-latex/2018-10-18_Security"}
$ touch some.file # Cria um arquivo
$ ls some.file
some.file
$ curl 'localhost:1337/somar/__import__("os").remove("some.file")or""/""'
$ ls some.file
ls: cannot access 'some.file': No such file or directory
```

Injection: como prevenir

Para consertar a vulnerabilidade no exemplo do slide anterior, podemos limitar a entrada para que os parâmetros sejam sempre números em ponto flutuante:

```
from flask import Flask, jsonify, abort
app = Flask(__name__)

@app.route("/somar/<a>/<b>")
def somar(a, b):
    try:
        return jsonify({"result": float(a) + float(b)})
    except TypeError:
        abort(404)
```

Entradas de "potenciais adversários" devem ser sempre validadas/filtradas/sanitizadas, principalmente para uso em algo como exec/eval. Esse exemplo foi artificial para ilustrar a ideia, o caso mais relevante nesta categoria de vulnerabilidade é o *SQL injection*, em que a string enviada para o bancos de dados contém fragmentos vindos do usuário (ver próximo slide).

https://gizmodo.com/5498412/sql-injection-license-plate-hopes-to-foil-euro-traffic-cameras

Autenticação quebrada

Já falamos bastante sobre senhas fracas e ataques de dicionário!

Outros exemplos:

Recuperação de senha com KBA
(*Knowledge-Based Authentication*):
perguntas sobre o usuário utilizando informações muitas vezes disponível publicamente (permite o ataque por terceiros) ou incorretas (*bricking* para o próprio usuário).

Hash fraco ou mal utilizado
(sem o "sal" / *nonce*):
Rainbow tables, tabelas com valores pré-computados de hashes, podem ser utilizadas para reverter hashes, principalmente quando as mensagens são curtas (e.g. hashes de senhas).

Exemplo de falha de configuração no NGINX

```
upstream meu_servidor {
  server 127.0.0.1:5000 fail_timeout=0;
}

server {
  listen 80;
  location /static/ {
    alias /app/static_files/;
    expires 30d;
  }
  root /app;
  location / {
    try_files $uri @proxy_to_app;
  }
  location @proxy_to_app {
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header Host $http_host;
    proxy_redirect off;
    proxy_pass http://meu_servidor;
  }
}
```

Código do servidor em /app disponível para download? Inclusive arquivos como .env ou settings.py com credenciais?

Exposição de dados sensíveis

Banco de dados exposto? Solução tradicional: *Firewall* bloqueando acesso à porta do banco + acesso via túnel SSH (*Secure Shell*)!

```
ssh -L 5432:db.server.com.br:5432 -v username@123.123.123.123
```

Credenciais via HTTP ao invés de HTTPS? Qualquer um observando o tráfego da rede poderá ver sua senha! Há alternativas "desesperadas" para tentar garantir a confidencialidade da senha mesmo sem HTTPS (e.g. roteador Linksys E2500 troca a senha pelo hash no *front-end* antes de enviar), mas, além de complicadas, não garantem a integridade do conteúdo e a identidade da fonte.

Outros assuntos

- Aspectos sociais/políticos (e.g. Manifesto Cypherpunk)
- Importância do FLOSS (*Free/Libre Open Source Software*), padrões abertos e transparência
- TOR (*The Onion Ring*), `https://www.torproject.org/`
- Detecção de [possíveis] fraudes (e.g. Serenata de Amor)
- *Pentest* (*penetration test*, teste de intrusão)
- *Capture the flag*
- *Firewall* e restrição de acesso
- DoS / DDoS (negação de serviço)
- DevSecOps
- "Bugs famosos" (Heartbleed, Meltdown, Spectre, ...)

Referências

A maioria do material está disponível apenas em inglês.

- Livro de GPG: https://www.gnupg.org/gph/en/manual/book1.html
- Tutorial de GPG: https://www.futureboy.us/pgp.html
- Página oficial do Tomb: https://www.dyne.org/software/tomb/
- Resumo sobre o Tomb: https://pujol.io/blog/tomb-with-gpg-keys/
- Página oficial do Pass: https://www.passwordstore.org/
- SHAttered: https://shattered.io/
- Repositório do gerador de certificados da Python Sudeste 2018: https://github.com/danilobellini/certificados-pyse2018
- Repositório da página da Python Sudeste 2018: https://github.com/pythonsudeste/pythonsudeste2018-site
- OWASP: https://www.owasp.org/
- OWASP 2017 Top 10 most critical web application security risks: https://github.com/OWASP/Top10/blob/master/2017/OWASP%20Top%2010-2017%20(en).pdf
- RFC1991 (PGP, obsoleto): https://tools.ietf.org/html/rfc1991
- RFC4880 (OpenPGP): https://tools.ietf.org/html/rfc4880
- RFC4226 (HOTP): https://tools.ietf.org/html/rfc4226
- RFC6238 (TOTP): https://tools.ietf.org/html/rfc6238

[... Continua no próximo slide ...]

Referências

- Trabalho "Security vs safety" do aluno Eirik Albrechtsen da NTNU, 2003: http://www.iot.ntnu.no/users/albrecht/rapporter/notat%20safety%20v%20security.pdf
- Time to rethink mandatory password changes: https://www.ftc.gov/news-events/blogs/techftc/2016/03/time-rethink-mandatory-password-changes
- Want Safer Passwords? Don't Change Them So Often: https://www.wired.com/2016/03/want-safer-passwords-dont-change-often/
- NIST is no longer recommending 2FA using SMS: https://www.schneier.com/blog/archives/2016/08/nist_is_no_long.html
- Fixing the cell network flaw that lets hackers drain bank accounts https://www.wired.com/2017/05/fix-ss7-two-factor-authentication-bank-accounts/
- Fixing the cell network flaw that lets hackers drain bank accounts https://www.wired.com/2017/05/fix-ss7-two-factor-authentication-bank-accounts/
- Simple social engineering trick with a phone call and crying baby: https://www.youtube.com/watch?v=1c7scxvKQ0o
- O que é desenvolvimento seguro, DevSecOps e S-SDLC (Communities Dev Show 2018): https://www.slideshare.net/cassiobp/o-que-desenvolvimento-seguro-devsecops-e-ssdlc

Todas as imagens utilizadas nos slides sem fonte explícita foram obtidas do Wikipedia, exceto as do slide inicial, as quais foram obtidas em http://etecuirapuru.com.br/.

FIM!