Cfrg 2: Pelcgb

Graun pregrmn qr dhr frh pbqvtb r orz pbzragnqb
qr sbezn dhr n shapvbanyvqnqr frwn ncneragr ncranf cryn yrvghen qbf
pbzragnevbf.

Bowrgvibf.

- Zryube snzvyvnevmn-yb pbz nythznf shapbrf r ovoyvbgrpnf.
- Fr niraghene hz cbhpb ab pnzcb qn pevcgbtensvn.

Yrvghen erpbzraqnqn.

- Frpbrf 11 n 14 r 39 qr uggc://vasbezngvpn.ufj.hby.pbz.oe/cebtenznpnb-rz-p.ugz

qvss cfrg2.cqs unpxre2.cqs.

- N rqvpnb unpxre yr b grkgb qr prfne qr hz nedhvib.
- A rqvpnb unpxre gragn yrine ibpr cnen b ynqb arteb qn sbepn, qrfnsvnaqb-b n penpxrne fraunf qr ireqnqr.

Pset 2: Crypto

a ser entregue até: 19:00, sex 09/03

Tenha certeza de que seu código é bem comentado
de forma que a funcionalidade seja aparente apenas pela leitura dos comentários.

Objetivos.

- Melhor familiarizá-lo com algumas funções e bibliotecas.
- Se aventurar um pouco no campo da criptografia.
- Introduzi-lo, talvez um pouco cedo demais, a File I/O.

Leitura recomendada.

- Seções 11 a 14, 17 a 19 e 39 de `http://informatica.hsw.uol.com.br/programacao-em-c.htm`

diff pset2.pdf hacker2.pdf.

- A edição hacker lê o texto de César de um arquivo.
- A edição hacker tenta levar você para o lado negro da força, desafiando-o a crackear senhas de verdade.

Honestidade Acadêmica.

Todo o trabalho feito no sentido do cumprimento das expectativas deste curso deve ser exclusivamente seu, a não ser que a colaboração seja expressamente permitida por escrito pelo instrutor do curso. A colaboração na realização de Psets não é permitida, salvo indicação contrária definida na especificação do Set.

Ver ou copiar o trabalho de outro indivíduo do curso ou retirar material de um livro, site ou outra fonte, mesmo em parte e apresentá-lo como seu próprio constitui desonestidade acadêmica, assim como mostrar ou dar a sua obra, mesmo em parte, a um outro estudante. Da mesma forma é desonestidade acadêmica apresentação dupla: você não poderá submeter o mesmo trabalho ou similar a este curso que você enviou ou vai enviar para outro. Nem poderá fornecer ou tornar as soluções disponíveis para os Psets para os indivíduos que fazem ou poderão fazer este curso no futuro.

Você está convidado a discutir o material do curso com os outros, a fim de melhor compreendê-lo. Você pode até discutir sobre os Psets com os colegas, mas você não pode compartilhar o código. Em outras palavras, você poderá se comunicar com os colegas em Português, mas você não pode comunicar-se em, digamos, C. Em caso de dúvida quanto à adequação de algumas discussões, entre em contato com o instrutor.

Você pode e deve recorrer à Web para obter referências na busca de soluções para os Psets, mas não por soluções definitivas para os problemas. No entanto, deve-se citar (como comentários) a origem de qualquer código ou técnica que você descubra fora do curso.

Todas as formas de desonestidade acadêmica são tratadas com rigor.

Licença.

Notas:

Seu trabalho neste Pset será avaliado em três quesitos principais:

Exatidão. Até que ponto o seu código é consistente com as nossas especificações e livre de bugs?
Design. Até que ponto o seu código é bem escrito*(*escrito claramente, funcionando de forma eficiente, elegante, e / ou lógica)?
Estilo. Até que ponto o seu código é legível (comentado e indentado, com nomes de variáveis apropriadas)?

I need somebody... Help!

☐ Vá para

`http://cc50.com.br`

e faça o login, se solicitado. Depois siga até a lista de discussões.

Daqui em diante, considere a `Lista de Discussões do CC50` *o* lugar para ir quando você tiver alguma dúvida ou pergunta. Além de postar perguntas suas, você também pode procurar respostas às perguntas já feitas por outros.

Espera-se, é claro, que você respeite as políticas do curso de honestidade acadêmica. Postagem de trechos de código sobre o qual você tiver dúvidas não é geralmente um problema. Mas postagem de programas inteiros, mesmo que não funcionem, é definitivamente um problema! Em caso de dúvida sobre se você deve postar ou não a sua pergunta, simplesmente mande um e-mail para `ajuda@cc50.com.br`, especialmente se você precisa mostrar grande parte do seu código. Mas quanto mais perguntas você fizer publicamente, mais os outros também se beneficiarão!

Não se preocupe com o que os outros pensam das suas perguntas, você não será julgado nem pelo Instrutor, nem pelos colegas sobre o que você está perguntando. Certamente, não hesite em postar uma pergunta, porque você acha que ela é "burra". Ela não é!

Começando

☐ Tudo certo, aqui vamos nós! Abra o seu Terminal e crie um diretório chamado `hacker2` dentro de cc50, e, em seguida, navegue para dentro desse diretório. (Lembra-se como?) O seu prompt agora deve ser semelhante ao abaixo.

```
username@computer (~/cc50/hacker2):
```

Se não for, refaça os seus passos e veja se você pode descobrir onde está o erro. Você também pode executar

```
history
```

para ver os seus últimos comandos em ordem cronológica, se você gostaria de fazer alguma investigação. Você também pode percorrer os comandos do seu histórico a qualquer momento usando as setas do seu teclado para cima e para baixo; pressione Enter para reexecutar qualquer comando que você gostaria. Se você ainda não tem certeza de como corrigir, lembre-se que `http://cc50.com.br/forum` é o seu novo amigo!

☐ Por fim copie a Makefile que veio junto com esse documento para dentro do diretório hacker2 que você criou.

Todo o trabalho que você fizer para este Pset deve, no fim, residir no diretório `hacker2` .

Ave, César!

☐ Lembre-se do final da Semana 2, que a Cifra de César criptografa (embaralha de forma reversível) mensagens "girando" cada letra *k* posições, passando de 'Z' para 'A', conforme necessário. Nós utilizaremos o alfabeto padrão de 26 letras `ABCDEFGHIJKLMNOPQRSTUVWXYZ`

`http://pt.wikipedia.org/wiki/Cifra_de_C%C3%A9sar`

Em outras palavras, se *p* é um texto não criptografado, p_i é o *igésimo* caractere de *p,* e *k* é a chave secreta *(*um número inteiro não negativo), então cada letra, c_i no texto criptografado, *c,* é computada como:

$c_i = (p_i + k) \% 26$

Esta fórmula talvez faz com que a Cifra de César pareça mais complicada do que é, mas ela é na verdade apenas uma maneira matemática precisa e concisa de expressar o algoritmo. E cientistas da computação amam precisão e concisão.

Por exemplo, suponha que a chave *k (*secreta) seja 13 e que o texto *p* seja "Nao se esqueca de beber o seu Nescau". Agora criptografamos esse *p* com aquela *k,* a fim de obter o texto *c,* cifrado, girando cada uma das letras 13 posições:

```
p: Nao se esqueca de beber o seu Nescau
c: Anb fr rfdhrpn qr orore b frh Arfpnh
```

Observe como `A` (a primeira letra do texto cifrado) vem 13 letras depois de `N` (a primeira letra do texto original). Da mesma forma `n` (a segunda letra do texto cifrado) está a 13 letras de distância de `a` (a segunda letra do texto original). Enquanto isso, `b` (a terceira letra do texto cifrado) está 13 letras distante de `o` (a terceira letra do texto original), nesse caso (e no primeiro também) tivemos que passar por 'Z' e começar de novo do 'A' para chegar lá. E assim por diante. Esse não é o sistema criptográfico mais seguro, com certeza, mas é divertido para implementar!

Aliás, uma cifra de César com uma chave de 13 é geralmente chamado ROT13:

`http://pt.wikipedia.org/wiki/ROT13`

No mundo real, porém, é provavelmente melhor usar ROT26, que se acredita ser duas vezes mais seguro do que ROT13.[1]

De qualquer forma, sua próxima meta é escrever, em `caesar.c`, um programa que criptografa mensagens usando a Cifra de César. Seu programa deve aceitar um argumento de linha de comando único: um número inteiro não negativo, *k.* Se o seu programa é executado sem nenhum argumento de linha de comando ou com mais de um argumento de linha de comando, o programa deve reclamar com o usuário e retornar um valor de `1` (que tende a significar um erro), através do comando abaixo:

[1] `http://www.urbandictionary.com/define.php?term=ROT26`

```
return 1;
```

Caso contrário, seu programa deve proceder para solicitar ao usuário o nome de um arquivo de texto, que contenha algum texto, claro, e se esse arquivo não for encontrado, o seu programa deve abortar retornando 2. Se o arquivo for encontrado, o output do seu programa será somente a primeira linha do arquivo de texto com cada caractere alfabético "girado" *k* posições; caracteres não-alfabéticos (assim como letras com acentos, etc) devem ser emitidos inalterados. Após o print desse texto cifrado, seu programa deve acabar, com a função `main` retornando `0`.

Apesar de existirem apenas 26 letras no alfabeto padrão, você não pode assumir que *k* seja menor ou igual a 26; seu programa deve funcionar para todos os valores inteiros não-negativos de *k* menores do que $2^{31} - 26$. Em outras palavras, você não precisa se preocupar se o seu programa, eventualmente quebrar, se o usuário escolher um valor para *k* que é muito grande ou quase grande demais para caber em um `int`. Agora, mesmo que *k* seja maior do que 26, caracteres alfabéticos no input do seu programa devem permanecer caracteres alfabéticos no output. Por exemplo, se *k* é 27, 'A' não deve tornar-se '[' embora o valor ASCII de '[' se encontra 27 posições acima do valor ASCII de 'A'. 'A' deve tornar-se 'B', pois 27 modulo 26 é 1, como um cientista da computação diria. Em outras palavras, valores como *k* = 1 e *k* = 27 são efetivamente equivalentes.

Seu programa deve preservar a capitalização: letras maiúsculas, embora rodadas, devem permanecer letras maiúsculas; letras minúsculas, embora rodadas, devem permanecer letras minúsculas.

Por onde começar? Bem, este programa tem de aceitar um argumento de linha de comando *k*, por isso desta vez você vai querer declarar `main` com:

```
int
main(int argc, char *argv[])
```

Lembre-se que `argv` é um "array" de strings (que são também conhecidas como "char estrela" por razões que veremos em breve). Na verdade, `string` é apenas um sinônimo para `char *`, graças à Biblioteca do CC50, você também poderia declarar `main` com:

```
int
main(int argc, string argv[])
```

se você acha que assim a sintaxe é mais clara. De qualquer maneira, você pode imaginar um array como uma linha de armários da escola, dentro de cada um dos quais está algum valor (e talvez algumas meias). Neste caso, dentro de cada armário se encontra uma `string`. Para abrir o primeiro armário, você pode usar uma sintaxe como `argv`[0], já que arrays sempre possuem o índice 0 como primeiro índice. Para abrir o armário ao lado, você pode usar sintaxe como `argv`[1]. E assim por diante. Claro que, se há `n`, armários é melhor você parar de abrir armários quando encontrar `argv`[n-1], uma vez que `argv[n]` não existe! (Ou isso ou ele pertence a outra pessoa, caso em que você não deve abri-lo)

E assim você pode acessar *k* usando um código parecido com:

```
string k = argv[1];
```

assumindo que k esteja realmente lá! Lembre-se que `argc` é um `int` que é igual ao número de strings que estão em `argv`, então é melhor você verificar o valor do `argc` antes de abrir um armário que possa não existir! Idealmente, `argc` será 2. Por quê? Bem, lembre-se que dentro de `argv`[0], por padrão, se encontra o nome do próprio programa. Então `argc` sempre será pelo menos `1`. Mas para este programa você quer que o usuário forneça um argumento de linha de comando *k,* logo `argc` deve ser `2`. Claro, se o usuário fornece mais de um argumento de linha de comando no prompt, `argc` pode ser superior a `2`, neste caso, é hora para algumas reclamações.

Agora, só porque o usuário digita um número inteiro no prompt, isso não significa que o seu input será automaticamente armazenado em um `int`. Ao contrário, ele será armazenado como uma `string` que só se parece com um `int`! E por isso você vai precisar converter essa `string` para um verdadeiro `int`. Como você tem sorte! A função `atoi`, existe exatamente para este fim. Veja como você pode usá-la:

```
int k = atoi(argv[1]);
```

Perceba que desta vez nós declaramos *k* como um verdadeiro `int` para que você realmente possa fazer alguma aritmética com ele. Ah, muito melhor agora. Aliás, você pode supor que o usuário só conseguirá digitar inteiros na linha de comando. Você não tem que se preocupar se eles digitarem, por exemplo, `foo`, apenas para serem difíceis; `atoi` retornará `0` nesses casos. Aliás, você precisará de um `#include` além de `cc50.h` e `stdio.h` para usar `atoi` sem que o gcc grite com você. Deixamos para você descobrir qual é esse arquivo![1]

Ok, então você tem *k* armazenado como um `int`, agora você terá que pedir ao usuário o nome de um arquivo de texto. Para simplificar a sua vida, o seu programa não precisará ler todo o conteúdo do arquivo, mas somente a primeira linha (e você pode considerar que essa linha tem no máximo, hmm, 1000 caracteres). Para abrir e ler o arquivo, você deve primeiro pesquisar sobre File I/O (Input/Output). As seções 18 a 20 de

`http://informatica.hsw.uol.com.br/programacao-em-c.htm`

irão, provavelmente, te ajudar.

[1] man `atoi`

Uma vez que você tem tanto k quanto o texto, é hora de criptografar. Lembre-se que você pode iterar sobre todos os caracteres em uma string, imprimindo um de cada vez, com um código como o abaixo:

```
for (int i = 0, n = strlen(p); i < n; i++)
{
    printf("%c", p[i]);
}
```

Em outras palavras, assim como `argv` é um array de strings, uma `string` é um array de caracteres. E assim você pode usar índices dentro de colchetes para acessar caracteres individuais em strings assim como você pode usá-los para acessar strings individuais em `argv`. Legal, né? É claro, a impressão de cada um dos caracteres de uma string não é exatamente criptografia. Bem, talvez tecnicamente, se *k* = 0. Mas o código acima deve ajudá-lo a implementar a sua Cifra de César! Por César!

Aliás, você ainda precisará incluir outro arquivo (com `#include`) para usar `strlen`.[1]

Essa abordagem de armazenar o texto lido do arquivo em uma string primeiro e depois iterar sobre ela não é, porém, a forma mais rápida e econômica de criptografar esse texto, você pode utilizá-la, se quiser, mas se você quiser se desafiar mais ainda, pode tentar resolver esse problema sem armazenar o texto em uma string.

Para que possamos automatizar alguns testes do seu código, seu programa deve se comportar de acordo com o exemplo abaixo. Assumindo que o texto em negrito é o que algum usuário digitou, e que o conteúdo de nescau.txt é:

```
Nao se esqueca de beber o seu Nescau

username@cloud (~/cc50/hacker2): ./caesar 13
nescau.txt
Anb fr rfdhrpn qr orore b frh Arfpnh
```

Além de `atoi`, você pode encontrar algumas funções úteis documentadas em:

```
http://www.cs50.net/resources/cppreference.com/stdstring/
```

Por exemplo, `isdigit` soa interessante. E, quanto à continuação de 'Z' para 'A', não se esqueça do operador de módulo (`%`). Você também pode querer verificar `http://asciitable.com/`, que revela os códigos ASCII para vários caracteres, não somente os alfabéticos, apenas no caso do seu programa acabar imprimindo alguns símbolos estranhos acidentalmente.

☐ Anl sqzazkgn.

[1] man `strlen`

Senhas et cetera.

☐ Na maioria, se não em todos, os sistemas rodando UNIX ou Linux, existe um arquivo chamado `/etc/passwd`. Por design, esse arquivo é feito para conter nomes de usuários e senhas, juntamente com outros detalhes relacionados à contas. Também por design (ruim), este arquivo é tipicamente aberto para leitura por todos. Felizmente, as senhas não são simplesmente armazenadas nele, mas são criptografados usando uma função "one-way hash". Quando um usuário se conecta a esses sistemas, digitando um nome de usuário e uma senha, a senha é criptografada com a mesma função hash, e o resultado é comparado à senha criptografada armazenada dentro de `/etc/passwd`. Se as duas senhas embaralhadas são iguais, o usuário ganha acesso ao sistema. Se você já esqueceu uma senha, você pode ter visto algo como "não posso ver qual é a sua senha, mas posso mudá-la para você". Provavelmente isso acontece pois uma função one-way hash está envolvida.

Apesar de as senhas em `/etc/passwd` sejam criptografadas, o algoritmo envolvido não é muito forte. Muitas vezes pessoas com más intenções, mediante a obtenção de arquivos como este, são capazes de adivinhar (e verificar) a senha de outros usuários ou crackea-las usando força bruta *(*tentando todas as senhas possíveis). Somente nos últimos anos os administradores de sistemas (em sua maioria) pararam de armazenar senhas em `/etc/passwd`, começando a usar `/etc/shadows`, que é (pelo menos deveria ser) legível apenas pelo usuário root. Abaixo, porém, estão alguns pares de `usuário:senha-cifrada` de sistemas (fake) desatualizados.

```
pskroob:50m2tb.6jehfA
dmalan:50T5DqRZxAIe2
mscott:50WZ/Wy2GdA1Y
gcostanza:50NwUtF.OmQNY
guest:50Bt2CexZzo7k
jcaesar:500EYaB0S7wVk
cpisonis:HAkzjBWeQ/Ugc
bvigenere:50dxMUnMm53rg
lemon:42XWroImGkEpU
```

Você deve descobrir essas senhas, cada uma das quais foi criptografada com o algoritmo (ou função) DES-based (não MD5-based) `crypt` de C. Especificamente, escreva, em `crack.c`, um programa que aceita um argumento de linha de comando único: uma senha criptografada[1] Se o seu programa for executado sem nenhum argumento de linha de comando ou com mais de um argumento de linha de comando, o programa deve reclamar e sair imediatamente, com a função `main` devolvendo qualquer `int` diferente de 0 (significando, assim, um erro que nossos próprios testes podem detectar). Caso contrário, seu programa deve continuar para crackear a senha dada, de preferência o mais rápido possível. No fim o seu programa deve imprimir a senha crackeada seguida de `'\n'`, nada mais, nada menos, com `main` retornando `0`. Deixamos para você a determinação do design deste programa, mas você deve explicar cada uma das suas decisões de design, incluindo quaisquer implicações para o desempenho e precisão, com comentários em todo o seu código. O programa deve ser projetado de tal forma que ele possa crackear todas as senhas acima, mesmo que isso possa demorar um pouco. Isto é, está tudo bem se o seu código demorar

[1] No caso de você testar seu código com outras senhas cifradas, saiba que os argumentos de linha de comando com certos símbolos (por exemplo `"?")` devem ser colocados entre aspas simples ou duplas; as aspas, no fim, não vão ficar em `argv`.

vários minutos ou horas ou até mais para ser executado. O que exigimos de vocês é o desempenho correto, não necessariamente ideal. Seu programa deve com certeza funcionar com outros inputs de senhas criptografadas, além das dadas acima; hard-coding das soluções para as senhas acima não é aceitável.

Mas para facilitar a sua vida, podemos dizer que nenhuma dessas senhas de exemplo tem mais do que 6 caracteres (isso não quer dizer que o seu programa não precise funcionar com senhas de mais de 6 caracteres, essas são só as senhas que você pode usar para testá-lo). Você pode restringir a sua busca também apenas aos caracteres lower-case e números então o seu programa não precisa prever que as senhas tenham símbolos, espaços ou caracteres upper-case.

Para que possamos automatizar alguns testes do seu código, seu programa deve se comportar de acordo com o abaixo; o que algum usuário digitou está em destaque em negrito.

```
username@computer (~/cc50/hacker2): ./crack 50Bt2CexZzo7k
guest
```

Assuma que as senhas originais não possuam mais do que seis caracteres. Mas pensando nas senhas criptografadas, é melhor dar uma olhada na página man de `crypt`, de modo que você saiba como a função funciona. Em particular, certifique-se de compreender o uso de um "salt" (de acordo com a página do manual, “um salt é usado para perturbar o algoritmo em uma de 4096 maneiras diferentes" mas por que isso poderia ser útil?). Como você pode ver na página do manual, você provavelmente vai querer colocar

```
#define _XOPEN_SOURCE
#include <unistd.h>
```

no topo do seu programa que deve ser compilado com `-lcrypt` (se você usa o `make` para compilar o seu código, esse switch será incluído automaticamente).

Por design, o `/etc/passwd` garante a segurança das senhas com uma suposição: que ninguem têm os recursos computacionais necessários para crackea-las. Houve um tempo em que isso pode ter sido verdade. Mas quando se trata de segurança, as suposições são perigosas. Esperamos que esse Pset te mostre como isso é verdade.

Devemos ainda observar que este Pset não é um convite para procurar outras senhas para crackear. Não confunda essas Edições Hacker com edições mal intencionadas. Esperamos, porém, que adquirindo um melhor entendimento da concepção dos sistemas de hoje, você poderá um dia construir sistemas melhores. Além de lhe familiarizar ainda mais com C, este pset convida você a começar a questionar designs ruins pois problemas como vulnerabilidades na segurança resultam muitas vezes desse tipo de design.

☐ Esse foi o Set de Problemas 2.