Estrutura de Dados II - 2018/2

Trabalho Prático T3

7 de novembro de 2018

1 *Password Cracking*

O objetivo desse trabalho é escrever um programa que usa uma tabela de símbolos para quebrar um esquema amplamente utilizado de codificação de senhas.

Quando o gerenciador de *login* de um sistema recebe uma senha, ele precisa verificar se a senha corresponde ao nome do usuário. O sistema faz essa verificação olhando nas suas tabelas internas. O método mais ingênuo de uso das tabelas é armazenar as senhas em uma *tabela de símbolos* aonde os nomes de usuários são as chaves e os valores correspondentes são as senhas de cada usuário. Esse método não é seguro porque é vulnerável a um invasor obter um acesso não autorizado à tabela do sistema, expondo as senhas de *todos* os usuários. Para evitar tal situação, a maioria dos sistemas usa um método mais seguro, aonde a tabela de símbolos armazena uma *senha criptografada* para cada usuário. Quando um usuário digita a senha, ela é criptografada e comparada com o valor armazenado. Se a comparação tem sucesso, o usuário é autorizado no sistema.

Para ser efetivo, este esquema exige um método de criptografia com duas propriedades: criptografar uma senha deve ser fácil (uma vez que isso deve ser feito a cada *login* de usuário) e recuperar a senha original a partir da versão criptografada deve ser difícil.

***Subset-sum encryption*.** Um método simples para gerenciar senhas funciona da seguinte forma: o tamanho de todas as senhas é fixado em um número específico de *bits*, por exemplo N. O sistema mantém uma tabela T de N inteiros, aonde *cada* inteiro tem exatamente N *bits* de comprimento. Para criptografar uma senha, o sistema usa a senha para selecionar um subconjunto dos números em T, e a seguir soma os elementos desse subconjunto. A soma *módulo* 2^N é a senha criptografada.

O exemplo pequeno a seguir para chaves de 5-*bits* ilustra o processo. Suponha que a tabela T possui os seguintes números de 5 *bits*:

```
i    T[i]       pwd
0    10110       0
1    01101       0
2    00101       1
3    10001       0
4    01011       1
```

Para a tabela acima, a senha `00101` é criptografada como `10000` já que a senha diz para utilizar a soma das linhas dois e quatro da tabela, e `00101 + 01011 = 10000`. É claro que, na prática, a tabela deve ser muito maior, conforme a discussão abaixo.

Agora, suponha que você consegue obter a tabela T do sistema (por exemplo, inspecionando o código fonte do SO) e você também consegue capturar os nomes de usuários e as senhas criptografadas correspondentes. Isso não é suficiente para invadir uma conta de um usuário: para "*crackear*" a senha, você precisa saber o subconjunto de T cuja soma é a senha criptografada. A segurança do sistema depende da dificuldade desse problema (conhecido como o *problema da soma de subconjunto – subset sum problem*, um problema sabidamente difícil de se resolver computacionalmente). Obviamente, o tamanho da senha deve ser suficientemente longo para te impedir de testar todas as possibilidades, mas também deve ser curto o suficiente para permitir que o usuário lembre a sua

senha. Neste trabalho, você verá que as senhas precisam ser razoavelmente longas. Com N *bits* nós temos 2^N subconjuntos diferentes, e assim pode parecer que 40 ou 50 *bits* devem ser suficientes, mas não é o caso.

Detalhes. Ao invés de usar números diretamente, é comum usarmos alguma tradução conveniente do que os usuários digitam para números. Para esse trabalho, nós vamos usar um *alfabeto* de 32 caracteres (letras minúsculas mais os seis primeiros dígitos) para as senhas, e vamos codificar as senhas em *arrays* de inteiros de *5-bits* (`chars`). Vamos seguir a seguinte codificação: 0 representa `'a'`, 1 representa `'b'` e assim por diante. Portanto, o número de *bits* N é $5 * C$, aonde C é número de caracteres (tamanho) da senha.

Para começar, você pode utilizar o código em `encrypt.c` (fornecido no arquivo `ED2_Trab3_src.zip` no AVA), que é o código que o administrador do sistema utiliza para criptografar a senha do usuário. O código lê a tabela T e a seguir utiliza os *bits* da senha para selecionar as posições da tabela para somar, imprimindo a versão criptografada no final. No arquivo `ED2_Trab3_in.zip` são fornecidas tabelas de codificação para diferentes tamanhos de senha. As tabelas com nome `easy*.txt` são altamente estruturadas e são úteis para depuração do código. Já as tabelas com nome `rand*.txt` foram geradas aleatoriamente.

Por exemplo, com a constante `C` definida como `8` no arquivo `key.h`, você deve obter o seguinte resultado para a senha `password`. O programa imprime a sequência da criptografia, para que você possa entender o processo:

```
$ gcc -Wall key.c encrypt.c -o encrypt
$ ./encrypt password < in/rand8.txt
   password  15  0 18 18 22 14 17  3   01111000001001010010101100111010001000011

 1 gobxmqkt   6 14  1 23 12 16 10 19   00110011100000110111011001000001010100011
 2 qdrvjxwz  16  3 17 21  9 23 22 25   10000000111000110101010011011111011011001
 3 joobqxtz   9 14 14  1 16 23 19 25   01001011100111000001100001011111001111001
 4 xnoixmnk  23 13 14  8 23 12 13 10   10111011010111001000101110110001101010010
10 tcixtvem  19  2  8 23 19 21  4 12   10011000100100010111100111010100100011100
13 lqtsdtca  11 16 19 18  3 19  2  0   01011100001001110010000111001100010000000
15 zlptzlfp  25 11 15 19 25 11  5 15   11001010110111110011110010101100101011111
18 gmjuvyqw   6 12  9 20 21 24 16 22   00110011000100110100101011100010000010110
20 uoqrdhwp  20 14 16 17  3  7 22 15   10100011101000010001000110011111011001111
22 ltdkzndz  11 19  3 10 25 13  3 25   01011100110001101010110010110100011111001
23 btezrznq   1 19  4 25 17 25 13 16   00001100110010011001100011100101101100000
26 bujilqno   1 20  9  8 11 16 13 14   00001101000100101000010111000000110101110
27 qgaicljl  16  6  0  8  2 11  9 11   10000001100000001000000100101101001010011
28 yyefwcld  24 24  4  5 22  2 11  3   11000110000010000101101100001001011000011
30 gnvowyjk   6 13 21 14 22 24  9 10   00110011011010101110101101100000100101010
34 aynzobxh   0 24 13 25 14  1 23  7   00000110000110111001011100000110111001111
38 lxwewfhh  11 23 22  4 22  5  7  7   01011101111011000100101100010100111001111
39 aenipbjd   0  4 13  8 15  1  9  3   00000001000110101000011110000101001000011

   vbskbezp  21  1 18 10  1  4 25 15   10101000011001001010000010010011001011111
```

É possível verificar com uma calculadora que as somas de cada coluna (módulo $R = 32$, o tamanho do alfabeto) correspondem às letras da senha não criptografada. A tabela `easy8.txt` torna essa verificação mais simples.

```
$ ./encrypt password < in/easy8.txt
   password  15  0 18 18 22 14 17  3   01111000001001010010101100111010001000011

 1 aaaaaaac   0  0  0  0  0  0  0  2   00000000000000000000000000000000000000010
 2 aaaaaaae   0  0  0  0  0  0  0  4   00000000000000000000000000000000000000100
 3 aaaaaaai   0  0  0  0  0  0  0  8   00000000000000000000000000000000000001000
 4 aaaaaaaz   0  0  0  0  0  0  0 25   00000000000000000000000000000000000011001
10 aaaaabaa   0  0  0  0  0  1  0  0   00000000000000000000000000000010000000000
13 aaaaaiaa   0  0  0  0  0  8  0  0   00000000000000000000000000010000000000000
15 aaaabaaa   0  0  0  0  1  0  0  0   00000000000000000000000010000000000000000
18 aaaaiaaa   0  0  0  0  8  0  0  0   00000000000000000000010000000000000000000
20 aaabaaaa   0  0  0  1  0  0  0  0   00000000000000000010000000000000000000000
22 aaaeaaaa   0  0  0  4  0  0  0  0   00000000000000000100000000000000000000000
23 aaaiaaaa   0  0  0  8  0  0  0  0   00000000000000001000000000000000000000000
26 aacaaaaa   0  0  2  0  0  0  0  0   00000000000001000000000000000000000000000
27 aaeaaaaa   0  0  4  0  0  0  0  0   00000000000010000000000000000000000000000
28 aaiaaaaa   0  0  8  0  0  0  0  0   00000000000100000000000000000000000000000
30 abaaaaaa   0  1  0  0  0  0  0  0   00000000010000000000000000000000000000000
34 azaaaaaa   0 25  0  0  0  0  0  0   00000110010000000000000000000000000000000
38 iaaaaaaa   8  0  0  0  0  0  0  0   01000000000000000000000000000000000000000
39 zaaaaaaa  25  0  0  0  0  0  0  0   11001000000000000000000000000000000000000

   b0onjjbh   1 26 14 13  9  9  1  7   00001110100111001101010010100100000100111
```

Certifique-se de que você entendeu completamente o funcionamento de `encrypt.c` antes de continuar. Note que o código utiliza `key.h` e `key.c` para realizar a soma módulo $R = 32$ sobre inteiros de C dígitos.

2 Solução força bruta

A sua primeira tarefa é escrever um programa *força bruta* que faz o inverso de `encrypt.c`. Dada uma senha criptografada e a tabela, o seu programa deve descobrir a senha: os C caracteres que, quanto convertidos para um número binário com N*-bits*, especificam o subconjunto cuja soma é a senha criptografada fornecida. Você deve ser capaz de quebrar facilmente senhas com 5 caracteres, uma vez que há apenas 32^5 (cerca de 33 milhões) de possíveis subconjuntos distintos.

Com a constante `C` definida como `5`, você deve obter o seguinte resultado para a senha `passw`:

```
$ ./encrypt passw < in/rand5.txt
   passw  15  0 18 18 22   01111000001001010010101110
   exvx5   4 23 21 23 31   00100101111010110111111111

$ gcc -Wall key.c brute.c -o brute
$ ./brute exvx5 < in/rand5.txt
i0ocs
passw
```

Note que é possível que exista mais de uma solução. Nesse caso o seu programa deve exibir todas as que existirem. Note também que esse método força bruta não vai funcionar para senhas mais longas. Por exemplo, para senhas com 10 caracteres, há 32^{10} (mais de 1000 trilhões) de subconjuntos distintos possíveis.

3 Solução por tabela de símbolos

A sua próxima tarefa é desenvolver um programa de quebra de criptografia mais eficiente que é funcionalmente equivalente ao método força-bruta, mas rápido o suficiente para quebrar senhas de 10 caracteres em um tempo razoável (por exemplo, em menos de uma hora). Para fazer isso, utilize uma tabela de símbolos. A ideia básica é

tomar um subconjunto S de T, computar todas as possíveis somas de subconjuntos que podem ser feitas com S, colocar esses valores em uma tabela de símbolos, e a seguir usar a tabela de símbolos para verificar todas essas possibilidades rapidamente.

Quando consideramos o esquema esboçado acima, várias questões surgem imediatamente: Qual o tamanho máximo de S? Exatamente como o *lookup* na tabela de símbolos vai funcionar? Qual implementação da tabela de símbolos é mais apropriada? Qual o melhor algoritmo para utilizar a tabela? *Responder a todas essas perguntas é o ponto chave do desenvolvimento/avaliação deste trabalho.*

O objetivo maior deste trabalho é verificar o seu entendimento/maturidade sobre como projetar e utilizar adequadamente uma estrutura de dados. Por isso, não serão fornecidos gabaritos de saída nem tempos esperados de execução. Cada trabalho será corrigido isoladamente, sem comparação com a solução do professor ou de outros grupos.

Você deve testar o seu programa para 4 caracteres e ir aumentando o número até 10. O seu objetivo, é claro, é conseguir quebrar qualquer senha criptografada com `encrypt.c`, como por exemplo:

```
$ gcc -Wall key.c decrypt.c -o decrypt
$ ./decrypt vbskbezp < rand8.txt
koaofbmx
password
xvbyofnz
1p1ngsgg
```

Esse método também para de funcionar a medida que o tamanho da senha cresce, mas é suficientemente eficiente para quebrar senhas de sistemas com uma segurança frágil.

4 Entrega do trabalho

Envie pelo AVA um arquivo compactado contendo a sua implementação para os dois programas descritos acima. Envie também um relatório (veja o modelo que deve ser preenchido no AVA) que descreve o método que você usou. Quando for possível, use o seu programa para quebrar as seguintes senhas que foram criptografadas com `encrypt.c`, usando `rand8.txt`, `rand10.txt` e `rand12.txt`, respectivamente.

```
xwtyjjin          h554tkdzti          uz1nuyric5u3
rpb4dnye          oykcetketn          xnsriqenxw5p
kdidqv4i          bkzlquxfnt          4l4dxa3sqwjx
m5wrkdge          wixxliygk1          wuupk1ol3lbq
```

Algumas considerações finais:

- Ao longo do desenvolvimento do trabalho, certifique-se que o seu código não está vazando memória testando-o com o `valgrind`. Não espere terminar o código para usar o `valgrind`, incorpore-o no seu ciclo de desenvolvimento. Ele é uma ferramenta excelente para se detectar erros sutis de acesso à memória que são muito comuns em C. Idealmente o seu programa deve sempre executar sem nenhum erro no `valgrind`.
- Você é livre para tirar quaisquer dúvidas que tiver com o professor, mas em *hipótese alguma* será informado qual tipo de algoritmo e/ou implementação da tabela de símbolos utilizar, uma vez que esse é o ponto fundamental da avaliação deste trabalho.

5 Regras para desenvolvimento e entrega do trabalho

- **Data da Entrega**: O trabalho deve ser entregue até às 23:55 h do dia 07/12/2018 (sexta-feira). Não serão aceitos trabalhos após essa data.

- **Prazo para tirar dúvidas**: Para evitar atropelos de última hora, você deverá tirar todas as suas dúvidas sobre o trabalho até o dia 05/12/2018. A resposta para dúvidas que surgirem após essa data fica a critério do professor.
- **Grupo**: O trabalho deve ser feito em **dupla**.
- **Linguagem de Programação e Ferramentas**: Você deve desenvolver o trabalho na linguagem C, sem usar ferramentas proprietárias (e.g., Visual Studio). O seu trabalho será corrigido no Linux.
- **Como entregar**: Pela atividade criada no AVA. Envie um arquivo compactado com todo o seu trabalho. A sua submissão deve incluir todos os arquivos de código e um `Makefile`. Além disso, inclua um arquivo de relatório (descrito adiante). **Somente uma pessoa da dupla deve enviar o trabalho no AVA. Coloque o nome completo dos dois integrantes da dupla no nome do arquivo do trabalho.**
- **Recomendações**: Modularize o seu código adequadamente. Crie códigos claros e organizados. Utilize um estilo de programação consistente. Comente o seu código extensivamente. Não deixe para começar o trabalho na última hora.

6 Relatório de resultados

Após terminar de implementar e testar o seu programa, você deve avaliar os resultados obtidos para os casos de teste e preencher um relatório que também deve ser entregue junto com o código do trabalho. Veja o arquivo `ED2_Trab3_info.txt` disponibilizado no AVA com as informações que devem ser preenchidas. O relatório deve ser entregue em formato `.txt`.

7 Avaliação

- Trabalhos com erros de compilação receberão nota zero.
- Trabalhos que gerem *segmentation fault* serão severamente penalizados na nota.
- Trabalhos com *memory leak* (vazamento de memória) sofrerão desconto na nota.
- Caso seja detectado plágio (entre alunos ou da internet), todos os envolvidos receberão nota zero.
- A critério do professor, poderão ser realizadas entrevistas com os alunos, sobre o conteúdo do trabalho entregue. Caso algum aluno seja convocado para uma entrevista, a nota do trabalho será dependente do desempenho na entrevista. (Vide item sobre plágio, acima.)
- O trabalho vale 2.0 pontos na média parcial do semestre, distribuídos da seguinte forma:
  - A solução força bruta correta vale 0.5 ponto.
  - O relatório devidamente preenchido com as respostas e textos explicativos solicitados vale 0.5 ponto. *Obs.: Note que um bom relatório não necessariamente depende de uma implementação completa. Por exemplo, se você descrever adequadamente o seu projeto do algoritmo, mesmo sem conseguir implementá-lo totalmente, a pontuação desse item vale.*
  - A implementação do programa de quebra de senha por tabela de símbolos vale 1.0 ponto. Neste item será considerado, além da correção do programa, o projeto/desenvolvimento/uso adequado da estrutura da tabela.