Processo de quebra de senha usando dicionário e força bruta

Renan Domingos Merlin Greca
Universidade Federal do Paraná
renangreca@gmail.com

José Robyson Aggio Molinari
Universidade Federal do Paraná
aggio13@hotmail.com

1. INTRODUÇÃO

Este é o relatório do segundo trabalho da disciplina de Introdução à Segurança Computacional, cursada na Universidade Federal do Paraná no segundo semestre de 2016. O trabalho é dividido em três etapas, cada uma com um objetivo: (1) criar um arquivo de dicionário a partir de um diretório contendo arquivos `txt`; (2) utilizando um algoritmo de força bruta, quebrar uma senha dado seu hash MD5; e (3) utilizar informações obtidas nas etapas (1) e (2) para quebrar um arquivo de senhas.

Cada etapa do trabalho continua seus próprios desafios, que são elaborados adiante.

2. DESENVOLVIMENTO

Conforme mencionado na introdução, este trabalho foi dividido em três partes. Nesta seção, são descritos detalhes sobre os problemas e as soluções de cada uma delas.

2.1 Construção do arquivo dicionário

O primeiro objetivo do trabalho foi desenvolver um programa em C, chamado `wordlist`, para processar arquivos `txt` em um diretório e criar um arquivo dicionário. Para tal, o programa percorre todas as palavras dos arquivos de texto (separadas por espaço ou quebra de linha) e os adiciona a uma lista, evitando palavras duplicadas. A biblioteca tinydir [1] é utilizada para facilitar a iteração sobre os arquivos no diretório. Ao final da execução, esta lista é escrita em um novo arquivo de texto, chamado `dicionario.txt`.

Para compilar o programa, basta executar:

```
make wordlist
```

E, para executá-lo, utiliza-se o seguinte comando:

```
wordlist <dir-path>
```

O argumento `dir-path` é o caminho do diretório contendo os arquivos de texto.

Os Programas 1 e 2 mostram, respectivamente, os laços para iterar sobre arquivos no diretório e sobre palavras em cada arquivo. Juntos, eles permitem a construção do arquivo dicionário.

Programa 1 Laço para iterar sobre os arquivos de texto.

```
tinydir_open_sorted(&dir, argv[1]);

// Skip the first two files, '.' and '..'.
for (i = 2; i < dir.n_files; i++) {
    tinydir_file file;
    tinydir_readfile_n(&dir, &file, i);

    read_words(file.path, words, &count,
        WORD_MAX);
}
```

Programa 2 Laço para iterar as palavras de cada arquivo.

```
while (*count < max) {

    // Read a word from the file
    if (fscanf(f, "%s", temp) != 1) {
        break;
    }

    // If word isn't already in words, add
        it to the array
    if (!exists(temp, words, (*count))) {
        words[(*count)] = strdup(temp);
        (*count)++;
    }
}
```

2.2 Quebra de senha por força bruta

A segunda parte do trabalho foi a mais interessante delas. Seu objetivo era desenvolver um programa, `brutexor`, com duas funcionalidades: primeiro, descobrir uma senha com hash conhecido e, segundo, utilizar essa senha para decifrar uma dica que seria necessária na terceira parte do trabalho.

Para garantir que estávamos no caminho certo, foi utilizada uma *rainbow table* encontrada na Internet para descobrir a senha a partir do hash MD5 e, então, decifrar a dica. *Rainbow tables* são tabelas pré-computadas de hashes, que tornam a quebra desse tipo de cifra trivial. A tabela utilizada contém quase um bilhão e meio de palavras — com

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

ela, encontramos facilmente a palavra que desejávamos:

```
MD5:     8d7b356eae43adcd6ad3ee124c3dcf1e
Palavra: FACIL
```

Com a chave conhecida, o processo de decifrar a dica era simples: bastou efetuar operações de XOR (ou exclusivo, em C representado pelo operador ^) entre a sequência de caracteres detalhada no enunciado do trabalho e a chave descoberta. Isso está descrito no Programa 3. Como a chave já era conhecida, bastou esse algoritmo para decifrar a dica:

```
Cadeia:  0b3430202f270522052921282226193423227 2d
Dica:    MusicaDaLegiaoUrbana
```

Programa 3 O algoritmo que decifra a dica usando a chave.

```
char input[20] = {0x0b, 0x34, 0x30, 0x20, 0
    x2f, 0x27, 0x05, 0x22, 0x05, 0x29, 0x21
    , 0x28, 0x22, 0x26, 0x19, 0x34, 0x23, 0
    x22, 0x27, 0x2d};
char key[5] = "FACIL";
char output[20];

for (i=0; i<20; i++) {
    output[i] = input[i] ^ key[i % strlen(
        key)];
}
```

A seguir, foi necessário criar um programa para descobrir a chave através de exaustão, utilizando o hash MD5 fornecido. Para isso, primeiro foi criado um breve algoritmo que calculava o MD5 de "FACIL" e o comparava com o hash conhecido, conforme o Programa 4. Quando compilado no Linux, o programa utiliza a biblioteca do OpenSSL para calcular o hash, enquanto no macOS é utilizada a biblioteca CommonCrypto.

Programa 4 Algoritmo para testar o hash da chave.

```
void compute_md5(char *str,
        unsigned char digest[17]) {
    MD5_CTX ctx;
    MD5_Init(&ctx);
    MD5_Update(&ctx, str, strlen(str));
    MD5_Final(digest, &ctx);
    digest[16] = '\0';
}

char key[20] = "FACIL";
unsigned char digest[17];

compute_md5(key, digest);
if (!strcmp((char *) hash,
        (char *) digest)) {
    printf("word found: %s\n", key);
}
```

Por último, foi desenvolvido um algoritmo para percorrer todas as palavras possíveis de 1 a 8 caracteres utilizando um alfabeto alfanumérico (letras maiúsculas, minúsculas e algarismos; ou seja, 62 caracteres), calcular os hashes MD5 de cada uma e comparar os resultados ao hash conhecido. Várias iterações desse algoritmo foram escritas, até ser criada uma versão que encontrasse a senha em tempo aceitável.

Para desenvolver o algoritmo, o problema foi tratado como uma busca em árvore, onde cada caracter é um nó e os possíveis caracteres seguintes são filhos do caracter anterior. Devido às condições da palavra descritas no parágrafo anterior, o resultado é uma árvore em que cada nó tem 62 filhos e a profundidade máxima é 8. O número total de nós é 62^8 (aproximadamente 218 trilhões), um valor nada trivial para percorrer. Nenhuma estrutura de dados similar a uma árvore foi desenvolvida; a abstração foi usada apenas para poder aplicar algoritmos conhecidos ao problema.

A versão inicial utilizava uma busca em profundidade. A primeira palavra testada foi "A", seguida por "AA" até chegar em "AAAAAAAA", e então foram percorridas todas as palavras que começam com sete A's e um oitavo caracter. Quando todas as palavras começadas em 'A' fossem percorridas, ele então partiria para aquelas que começam em 'B'. Esta função é mostrada no Programa 5. O problema dessa solução é que percorrer palavras mais longas é muito demorado — o cálculo do MD5 para cada palavra não muda, mas o número de palavras possíveis cresce exponencialmente com cada caracter adicionado. Portanto, se a senha fosse simplesmente a palavra "0", por exemplo, o programa levaria dezenas de horas para encontrá-la, pois precisaria percorrer todas as palavras que iniciam com todas as letras do alfabeto maiúsculas e minúsculas.

Programa 5 Algoritmo de busca em profundidade para encontrar a senha.

```
char *brute_force(char *word,
        int index, int max_len) {
    int i;
    for (i=0; i<62; i++) {
        word[index] = alphabet[i];
        word[index+1] = '\0';

        compute_md5(word, digest);

        if (!strcmp((char *) hash,
                (char *) digest)) {
            return word;
        }
        if (index < max_len) {
            char *res = brute_force(word,
                index+1, max_len);
            if (res != NULL) {
                return res;
            }
        }
    }
    return NULL;
}
```

A próxima versão do algoritmo incorporou paralelização em CPU utilizando a biblioteca OpenMP. Dessa forma, cada thread da CPU era responsável por processar palavras iniciadas com certas letras (um conjunto de letras é atribuído para cada thread). Apesar do processo acelerado pela computação paralela, a busca em profundidade permaneceu extremamente lenta. O Programa 6 mostra o laço e os co-

mandos OpenMP para permitir a paralelização. A variável `thread_alphabet` contém os segmentos do alfabeto que devem ser utilizados como caracteres iniciais para cada thread. A thread 0, por exemplo, é responsável por testar todas as palavras que iniciam em 'A' e outras letras de acordo com o número de threads.

Programa 6 Comandos OpenMP e laço para paralelizar a execução.

```
#pragma omp parallel
{
    #pragma omp single
    for (i=0; i<8; i++) {

        #pragma omp task
        {
            int tid = omp_get_thread_num();

            int j;
            for (j=0; j<strlen(
                thread_alphabet[tid]) && !
                done; j++) {
                char *word = malloc(9 *
                    sizeof(char));
                sprintf(word, "%c",
                    thread_alphabet[tid][j
                    ]);
                brute_force(word,
                    1, MAX_LEN);
            }
        }
    }
    #pragma omp taskwait
}
```

Programa 7 Laço para "forçar" a busca em largura.

```
int k;
for (k=1; (k<=MAX_LEN) && !done; k++) {
    printf("thread %d is testing words of
        length %d\n", tid, k);
    int j;
    for (j=0; j<thread_alphabet_len && !
        done; j++) {
        char *word = malloc(9 * sizeof(char
            ));
        sprintf(word, "%c", thread_alphabet
            [tid][j]);
        bruteForce(word, 1, k);
    }
}
```

Portanto, foi cogitado usar busca em largura ao invés de profundidade. O algoritmo tradicional de busca em largura exige uma fila FIFO (first in, first out) para armazenar as próximas palavras que devem ser testadas. Quando a palavra "A" é testada, o algoritmo coloca todas as palavras com dois caracteres que começam com "A" na fila. Isso permite que todas as palavras curtas sejam testadas antes das palavras mais compridas e, caso a palavra tenha menos que oito caracteres, a execução será notavelmente mais rápida. No entanto, a fila FIFO ocupa espaço demasiado em memória — especialmente quando o algoritmo está paralelizado em várias threads. Nos nossos testes, em um computador com 8GB de memória principal, a RAM era exaustada durante os testes de palavras com três caracteres, tornando essa versão do algoritmo inviável.

A solução final foi desenvolver um processo que "forçava" o algoritmo de busca em profundidade, que utiliza memória desprezível, a testar as palavras em ordem de tamanho. Para tal, há um laço adicional na hora de chamar a função de força bruta em cada thread. Esse laço faz com que a função seja chamada oito vezes, com o tamanho máximo das palavras aumentado a cada iteração. Assim, na primeira iteração a função testará apenas palavras com um caracter, até que na última sejam testadas apenas palavras com oito. Esse laço adicional é mostrado no Programa 7. Vale observar que a variável `done` é utilizada para que todas as threads parem a execução quando uma delas encontrar a senha.

O Programa 8 é a versão final da função de força-bruta, incluindo todas as modificações para a paralelização e a busca em largura. Os condicionais também foram reposicionados para otimizar a execução. A variável global `key` armazena a senha encontrada antes de encerrar o procedimento.

Programa 8 Versão final da função de força bruta

```
void brute_force(char *word,
        int index, int max_len) {

    if (done) return;

    if (index == max_len) {
        // Bottom of recursion, test word
        unsigned char digest[16];
        compute_md5(word, digest);

        if (!strcmp((char *) hash,
                    (char *) digest)) {
            // Word found, end
                execution
            done = true;
            key = strdup(word);
        }
    } else if (index < max_len) {
        // Word not long enough, go further
        int i;
        for (i=0; i<alphabet_len && !done;
            i++) {
            word[index] = alphabet[i];
            word[index+1] = '\0';

            bruteForce(word, index+1,
                max_len);
        }
    }
}
```

Por fim, o programa sofreu pequenas alterações para funcionar com ou sem paralelização e independente do número de threads disponíveis no computador. O Programa 9 mostra o processo para criar blocos do alfabeto que serão alocados

para cada thread disponível. A constante `OPENMP` é definida apenas se o programa foi compilado para ser compatível com multiprocessamento.

Programa 9 Alocação de blocos do alfabeto

```
// If OpenMP is disabled, number of threads
    is 1.
#if defined(OPENMP)
    int num_threads = omp_get_num_threads()
        ;
#else
    int num_threads = 1;
#endif

char *thread_alphabet[num_threads];
int thread_alphabet_len = ((alphabet_len/
    num_threads)+1);

// Each thread gets a portion of the
    alphabet allocated to it.
for (i = 0; i < num_threads; ++i) {
    thread_alphabet[i] = malloc(
        thread_alphabet_len * sizeof(char)
        );
    thread_alphabet[i][0] = '\0';
}

char temp[2];
for (i=0; i<alphabet_len; i++) {
    sprintf(temp, "%c", alphabet[i]);
    strcat(thread_alphabet[i % num_threads
        ], temp);
}
```

2.3 Quebra de arquivo de senha usando arquivo dicionário

O objetivo da terceira e última parte do trabalho era utilizar a dica decifrada na segunda etapa para encontrar arquivos de texto e então utilizar o `wordlist` para gerar um arquivo dicionário. Esse arquivo dicionário deveria então ser utilizado como argumento para o programa John The Ripper, uma solução open-source para quebra de senhas, para quebrar o arquivo `_etc_shadow.txt`.

Com a dica encontrada na segunda parte, foi necessário criar um diretório contendo arquivos de texto com letras de músicas da banda Legião Urbana. Portanto, foi desenvolvido um script em Python, `crawler.py` que utiliza a biblioteca Beautiful Soup [2] para obter as letras das músicas através do site **letras.mus.br**. O script limpa o texto de tags HTML e salva cada letra em um arquivo de texto separado, utilizando o título de cada canção como nome. Ao rodar o `wordlist` com essas letras, o resultado é um arquivo `dicionario.txt` com 5927 palavras distintas.

Após a execução do script Python e do `wordlist`, é possível utilizar esses dados para quebrar a senha com o John The Ripper através do seguinte comando:

```
john --wordlist=dicionario.txt _etc_shadow.txt
```

Dessa forma, foi possível quebrar uma das senhas presentes no arquivo, a do usuário *eva* — "Cavalos-marinhos", palavra presente na letra da canção "Vento no Litoral". Através de conversas com colegas, foi descoberto que a senha do usuário *beto* era "cavalo-marinho", mas essa palavra não está presente em nenhuma letra. Para verificar isso, foi criado um arquivo de texto contendo apenas essas duas palavras; com ele, o John The Ripper foi capaz de quebrar ambas as senhas.

3. RESULTADOS E CONCLUSÕES

Para as execuções dos algoritmos, dois computadores foram utilizados: "SgtPepper", um MacBook Pro, e "Packard", um desktop customizado. O programa `wordlist` roda rapidamente em ambos os computadores, então sua análise não é muito interessante.

O segundo programa, `brutexor`, pode ser extremamente lento ou rápido dependendo de algumas condições, portanto há mais a ser dito sobre ele. Utilizando a primeira versão do algoritmo, com busca em profundidade e sem paralelização, o programa rodou por mais de vinte horas sem sucesso — essa demora motivou o desenvolvimento de versões mais eficientes.

Nome	"SgtPepper"	"Packard"
S.O.	macOS 10.12	Linux Mint 17
CPU	Intel Core i5-4308U	Intel Core i7-2600
Clock base	2.8GHz	3.4GHz
Turbo boost	3.3GHz	3.8GHz
Cache L3	3MB	8MB
Cores/threads	2/4	4/8
RAM	8GB DDR3L	8GB DDR3
Armz.	SSD 512GB	HDD 320GB

O Packard, que dispõe de um processador quad-core com *hyperthreading*, foi capaz de realizar o algoritmo de força bruta em oito threads paralelas. Devido à forma como as letras foram distribuídas, muito rapidamente a thread nº 5 chega à senha "FACIL" (pois ela já começa testando as palavras iniciadas em 'F'). Portanto, o programa `brutexor` nesse computador executa em aproximadamente meio segundo.

Devido a complicações em instalar a biblioteca OpenMP no macOS, o programa foi testado no SgtPepper sem utilizar paralelização. Rodando em apenas uma thread, o `brutexor` levou aproximadamente 38 segundos para encontrar a senha desejada.

Um fator interessante deste problema é que, dependendo da senha a ser encontrada, ele pode executar em décimos de segundos ou dezenas de horas. Em particular, qualquer senha de oito caracteres levaria muito tempo para ser encontrada — e portanto a busca em largura foi adotada. A senha começar com 'F' também ajuda, pois é uma das primeiras letras do alfabeto e será processada rapidamente por uma das threads — se a distribuição de caracteres para as threads fosse diferente, o tempo para encontrar a senha também seria diferente. Sob a presunção que senhas curtas são mais comuns, acreditamos que o caminho adotado é o mais lógico para este tipo de problema.

O tempo de execução do John The Ripper em ambos computadores também foi comparado. No Packard, encontrar a senha "Cavalos-marinhos" (e descobrir que as outras não estavam no dicionário) levou 51 segundos, enquanto, no SgtPepper, o processo levou cerca de 30 segundos. Isso ocorre porque o Core i5 presente no SgtPepper, apesar de ser mais

lento que o Core i7 do Packard, é da família Haswell, que incorporou modificações na arquitetura que favoreceram funções criptográficas, além do armazenamento em SSD que acelera a leitura do arquivo de dicionário.

Por fim, o desenvolvimento do trabalho nos mostrou que o processo para quebrar senhas com oito ou mais caracteres pode ser incrivelmente demorado, caso não se tenha uma *rainbow table* completa do algoritmo de hash utilizado; portanto métodos de engenharia social estão cada vez mais comuns entre atacantes. As fases de otimização do algoritmo de força bruta também foram fascinantes por um ponto de vista computacional, por mostrarem como formas antigas de segurança, que é o caso do MD5, tornam-se obsoletas com o avanço tecnológico dos microprocessadores..

4. REFERENCES

[1] Cong. tinydir. https://github.com/cxong/tinydir.

[2] L. Richardson. Beautiful Soup. https://www.crummy.com/software/BeautifulSoup/.