PONTIFÍCIA UNIVERSIDADE CATÓLICA DE CAMPINAS

CENTRO DE CIÊNCIAS EXATAS, AMBIENTAIS E DE TECNOLOGIA

CHRISTOPHER OLIVEIRA	**RA:18726430**
GIULIANO SANFINS	**RA:17142837**
MATHEUS MORETTI	**RA:18082974**
MURILO ARAUJO	**RA:17747775**
VICTOR REIS	**RA:18726471**

SISTEMAS OPERACIONAIS B – PROJETO 2

CRIPTO SYSTEM CALLS – SOB

CAMPINAS
2020

SUMÁRIO

1. INTRODUÇÃO

Esse projeto teve como principal objetivo familiarizar o aluno com os principais aspectos de implementação de uma chamada de sistema (system calls) em um kernel, contemplando todos os aspectos de sua implementação, compilação, instalação, e testes com o kernel Linux modificado.

Também, um item importante tratado nesse trabalho que não poderia deixar de ser dito foi a parte de manipulação de arquivos, onde é diferente no ambiente windows. Tivemos que estudar como funcionava as principais funções de escrita e leitura e como adaptarmos para a nossa problemática, que tinha como meta realizar duas novas chamadas de sistemas que foram requeridas para o projeto, sendo a primeira de escrita e a segunda de leitura, que foi realizada e confeccionada a nível de kernel, sendo “chamada” a nível de usuário apenas para trabalhar com o que era solicitado.

Um outro objetivo secundário, porém necessário para o caminhar do desenvolvimento do projeto, seria analisar e entender o funcionamento das system calls no sistema, como elas funcionam, onde são implementadas, o que fazem, e o que é preciso para que funcionem no sistema, entre outros aspectos.

2. DETALHES DE PROJETO DAS SYSCALLS

Neste projeto alguns objetivos foram requisitados, implementar duas syscalls em nível de kernel, uma sendo para realizar escrita e a outra leitura em um kernel Linux, onde os conhecimentos relacionados e estabelecidos previamente no projeto anterior, duas novas chamadas de sistema write_crypt e read_crypt. Onde a primeira deveria permitir que programas em espaço de usuário possam armazenar arquivos de forma cifrada, usando algoritmo AES em modo ECB para cifrar os dados, já a segunda deveria permitir que programas em espaço de usuário possam, como o nome indica, ler arquivos cifrados com a chamada write_crypt, usando algoritmo AES em ECB para decifrar os dados lidos.

Nessa implementação, um grande dificuldade que tivemos foi a manipulação de arquivo, onde tivemos que alguns erros de compilações e execuções, no final após testes adaptações, conseguimos chegar num ótimo resultado que permitia a criação de um arquivo assim que o usuário executasse o comando, e escrevesse nesse arquivo a frase criptografada que o usuário inseriu na entrada.

3. DESCRIÇÃO DAS MODIFICAÇÕES NO CÓDIGO FONTE

No Makefile criado dentro do diretório info, foi realizado com a seguinte informação:

obj-y:=listProcessInfo.o

No Makefile principal na parte externa, sendo a principal do código fonte do kernel 4.15.0, foi modificado o seguinte item:

ifeq ($(KBUILD_EXTMOD),)

core-y += kernel/ certs/ mm/ fs/ ipc/ security/ crypto/ block/ **info/**

Inserindo o diretório info que criamos na sequência dos outros diretórios armazenados em core-y.

No arquivo **processinfo.h** criado dentro da pasta info, foi inserido a seguinte informação apenas dentro dele:

asmlinkage ssize_t write_crypt(int fd,const void *buffer, size_t nbytes);

asmlinkage ssize_t read_crypt(int fd,void *buffer, size_t nbytes);

Sendo os escopos das funções que utilizamos em nível de kernel para realizar escrita e leitura de um arquivo.

No arquivo **syscall_64.tbl,** inserimos essas duas linhas que indicam as chamadas das nossas syscalls de escrita e leitura que realizamos no diretório info.

333 common write_crypt write_crypt

334 common read_crypt read_crypt

No arquivo syscalls.h, inserimos essas duas linhas no final do programa, indicando mais uma vez o escopo das nossas funções que realizamos como syscall para o realizamento de escrita e leitura.

asmlinkage ssize_t write_crypt(int fd,const void *buffer, size_t nbytes);

asmlinkage ssize_t read_crypt(int fd,void *buffer, size_t nbytes);

E por fim tem nosso arquivo WriteRead.c e test.c, onde o primeiro foi realizado a nível de kernel, que seria nosso arquivo principal contendo toda a parte de manipulação de arquivo, bibliotecas, criação das syscalls e devidas alterações e a parte de criptografia dos dados.

E o test.c seria o arquivo teste nosso como próprio já diz que se encontra em nível de usuário mesmo, onde é realizado a criação do arquivo, o recebimento da

entrada da frase do usuário desejado, a chamada das syscall de escrita e leitura desse arquivo que por consequência dessas chamadas já realiza a criptografia antes de escrever no arquivo, fizemos a manipulação do arquivo e escrita dessa maneira.

4. DESCRIÇÃO DOS RESULTADOS

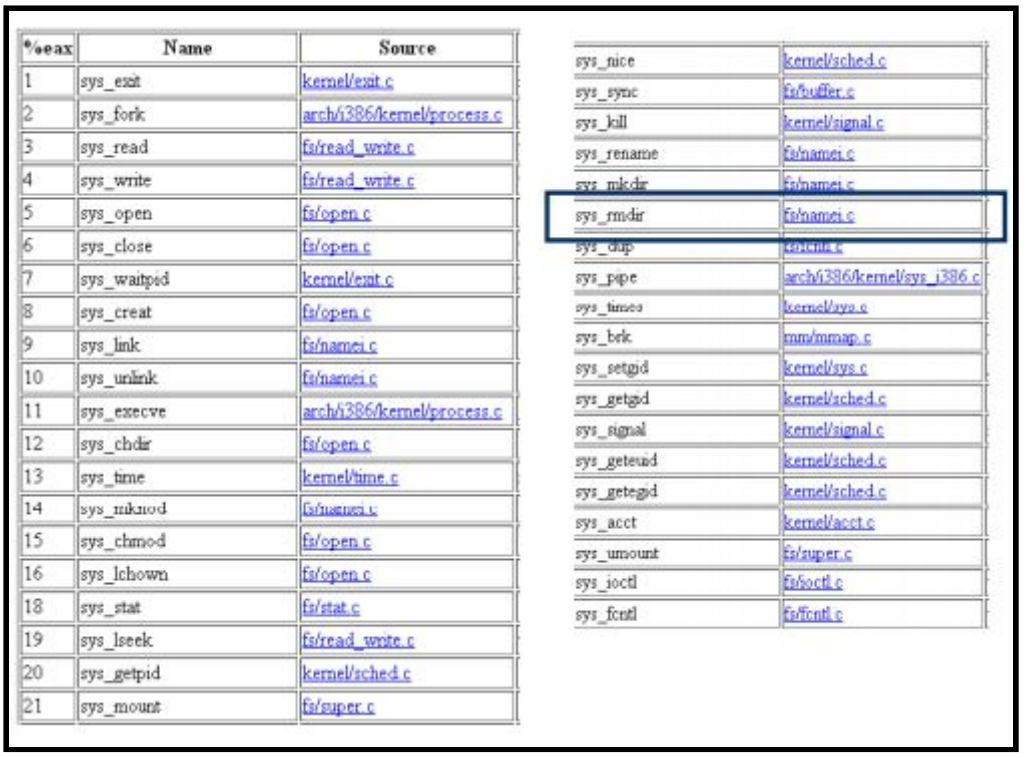
Para entendermos melhor o funcionamento da criptografia, foi nos dado um trabalho onde deveríamos inserir uma frase (*string*), e após isso convertê-la em hexadecimal e analisar o que ocorre com a *string* utilizando e não usando criptografia, para saber a diferença do que ocorre quando é aplicada ou não, podendo observar a partir desse simples teste as metodologias para se obter a *string* original. Após realizarmos a construção da criptografia e toda essa análise através de impressões no terminal do Linux, foi desenvolvido uma segunda função, no qual tinha principal intuito de "*descriptografar*" a frase inserida pelo usuário, e também, realizar uma segunda conversão, que deveria obter a frase que está convertida em hexadecimal, passando-a para a frase original que foi escrita pelo usuário. Essa frase quando continha uma quantidade menor de 16 bytes, era completada com zeros em suas últimas posições após a entrada do usuário.

No começo tivemos algumas dificuldades, pois não saberíamos como realizar essa transformação e quantos "bytes" e posições em um vetor cada letra ou número ocuparia. Então após a realização de alguns testes em um arquivo separado, e analisado como seria a melhor forma de aplicar essas duas conversões, podendo notar que uma letra ou um número poderia ser escrito em hexadecimal com duas posições em um vetor, sendo a letra acompanhada de um número ou até mesmo de outra letra na segunda posição do vetor, onde através disso foi realizado a manipulação de dividir por 16 e trabalhar com o resto da divisão, e depois, o processo inverso, de multiplicar por 16 elevado a 0 ou 1, pois se tratava apenas de duas posições. Podendo dizer que após vários testes e análise foi possível obter a frase de origem exatamente como foi escrita, aplicando depois a criptografia nessa frase como foi dito acima o seu objetivo. Nessa criptografia, tinha que conter sempre uma chave e um *iv* que eram fixo, não tendo a necessidade de inserir o seu valor.

Utilizando o trabalho anterior na parte de criptografia, foi onde realizamos algumas adaptações e aproveitamos ao máximo a parte de criptografia criada onde funciona muito bem. Sendo necessária primeiramente neste trabalho instalar um kernel, no qual optamos pela versão 4.15.0, que após a instalação, alteramos com o comando sudo os diretórios necessários para o funcionamento das duas syscalls, que como já foi dito seria a de write_crypt() e read_crypt().

Detalhando melhor as *Syscalls*, elas são como portas de entrada para se ter acesso às rotinas do SO. Se alguma aplicação desejar chamar uma rotina do S.O um mecanismo de *system call* verificará se a mesma possui os privilégios necessários.

A figura abaixo exibe uma tabela de *system calls* do Linux.



%eax	Name	Source
1	sys_exit	kernel/exit.c
2	sys_fork	arch/i386/kernel/process.c
3	sys_read	fs/read_write.c
4	sys_write	fs/read_write.c
5	sys_open	fs/open.c
6	sys_close	fs/open.c
7	sys_waitpid	kernel/exit.c
8	sys_creat	fs/open.c
9	sys_link	fs/namei.c
10	sys_unlink	fs/namei.c
11	sys_execve	arch/i386/kernel/process.c
12	sys_chdir	fs/open.c
13	sys_time	kernel/time.c
14	sys_mknod	fs/namei.c
15	sys_chmod	fs/open.c
16	sys_lchown	fs/open.c
18	sys_stat	fs/stat.c
19	sys_lseek	fs/read_write.c
20	sys_getpid	kernel/sched.c
21	sys_mount	fs/super.c

sys_nice	kernel/sched.c
sys_sync	fs/buffer.c
sys_kill	kernel/signal.c
sys_rename	fs/namei.c
sys_mkdir	fs/namei.c
sys_rmdir	fs/namei.c
sys_dup	[illegible]
sys_pipe	arch/i386/kernel/sys_i386.c
sys_times	kernel/sys.c
sys_brk	mm/mmap.c
sys_setgid	kernel/sys.c
sys_getgid	kernel/sched.c
sys_signal	kernel/signal.c
sys_geteuid	kernel/sched.c
sys_getegid	kernel/sched.c
sys_acct	kernel/acct.c
sys_umount	fs/super.c
sys_ioctl	fs/ioctl.c
sys_fcntl	fs/fcntl.c

ret_status = syscall (333, fd, buffer, n);
ret_status = syscall (334, fd, bufferReturn, n);

No nosso caso usamos uma *syscall* em nível de usuário para ativar a *syscall* em nível de kernel:
Syscall (333, fd, buffer, n);
333 -> chamada da *syscall* para escrever.
334 -> chamada da *syscall* para ler.
fd -> *file descriptor,* onde é armazenado o arquivo.
n -> seria o tamanho da string que foi escrita pelo *user* (entra com o parâmetro).

Usamos um mecanismo chamado VFS (Virtual File System) que permite que chamadas de sistemas “padrões”, tais como *open( )* e *read( ),* possam ser executadas sem depender do sistema de arquivos usados.

O VFS é um subsistema do Kernel que implementa as funções para que os programas de usuários acessem ao sistema de arquivos.

Todos os sistemas de arquivos existentes levam em consideração o mecanismo de VFS para coexistir e interagir entre si.

Abaixo pode se observar os principais includes utilizados na elaboração desse projeto.

Includes:

```
#include <linux/fs.h>
#include <asm/segment.h>
#include <asm/uaccess.h>
#include <linux/buffer_head.h>
```

Abaixo pode se observar a estrutura da nossa função *write* que utilizamos em nível de kernel.

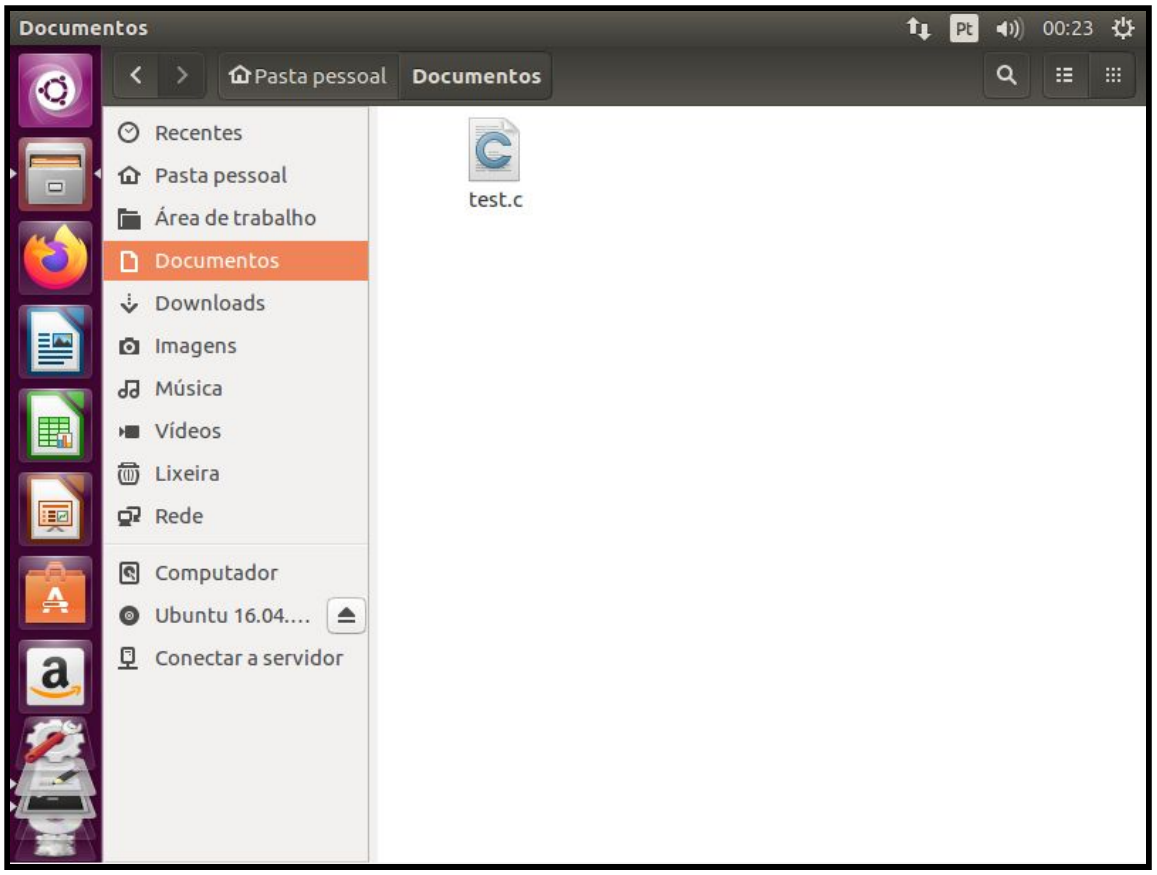
Write:

```
int file_write(struct file *file, unsigned char *data, unsigned int size)
{
   mm_segment_t oldfs;
   int ret;
   oldfs = get_fs();
   set_fs(get_ds());
   ret = sys_write(file, data, size);
   set_fs(oldfs);
   return ret;
}
```

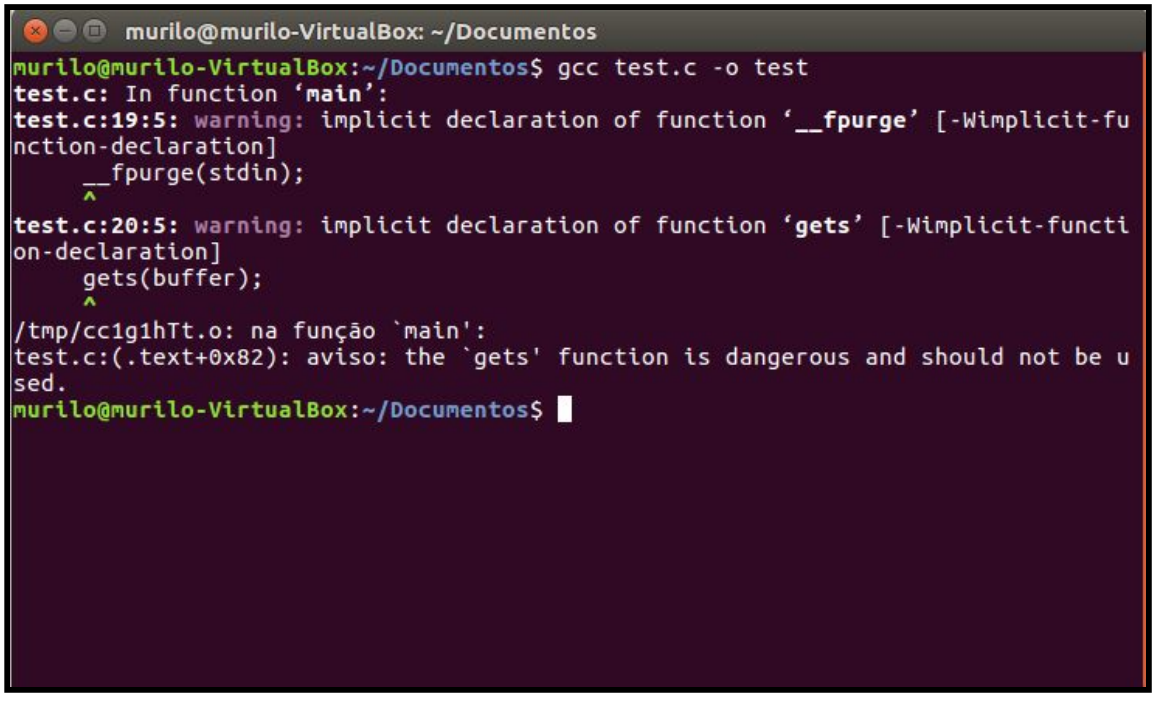
Abaixo pode se observar a estrutura da nossa função *read* que utilizamos em nível de kernel.

```
Read:
int file_read(struct file *file, unsigned char *data, unsigned int size)
{
    mm_segment_t oldfs;
    int ret;

    oldfs = get_fs();
    set_fs(get_ds());

    ret = sys_read(file, data, size);

    set_fs(oldfs);
          printk("TESTANDO DENTRO DA FUNC\n");
          printk("%s",data);
    return ret;
}
```

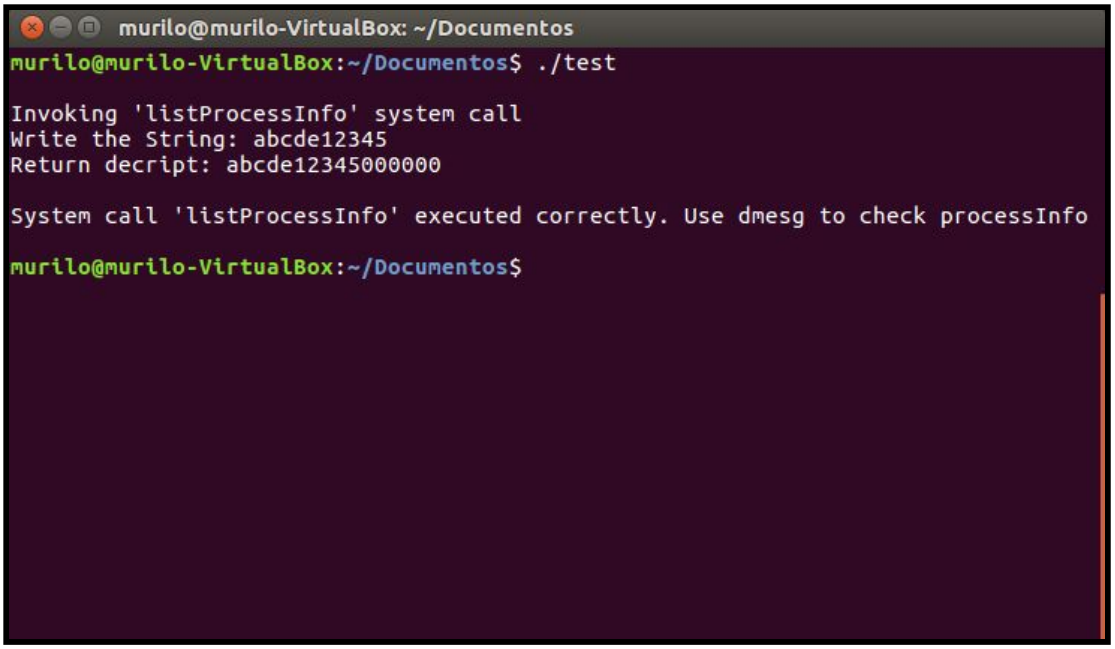
Abaixo pode se observar uma imagem que consta apenas o nosso arquivo teste criado.



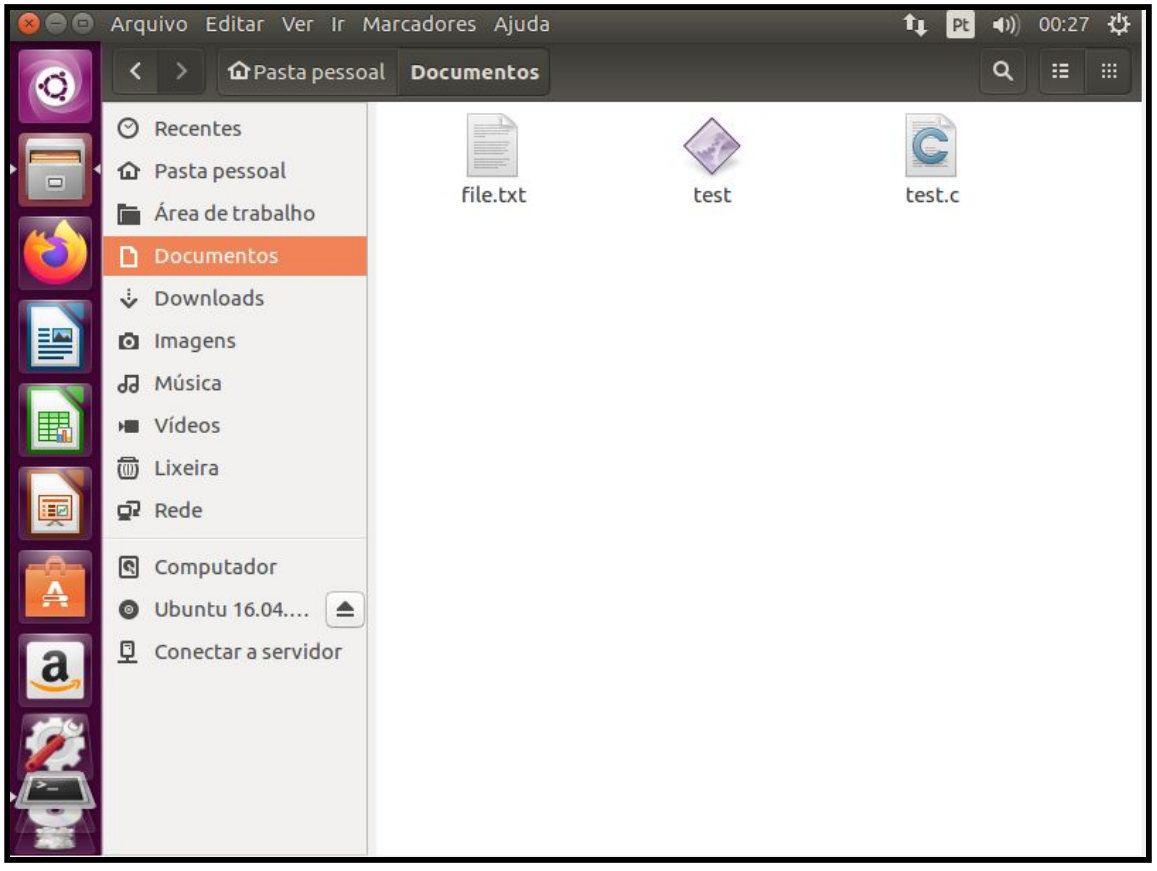
Abrindo o cmd nessa pasta e executando o comando “gcc test.c -o test” podemos observar que a compilação funciona corretamente sem apresentar erros.



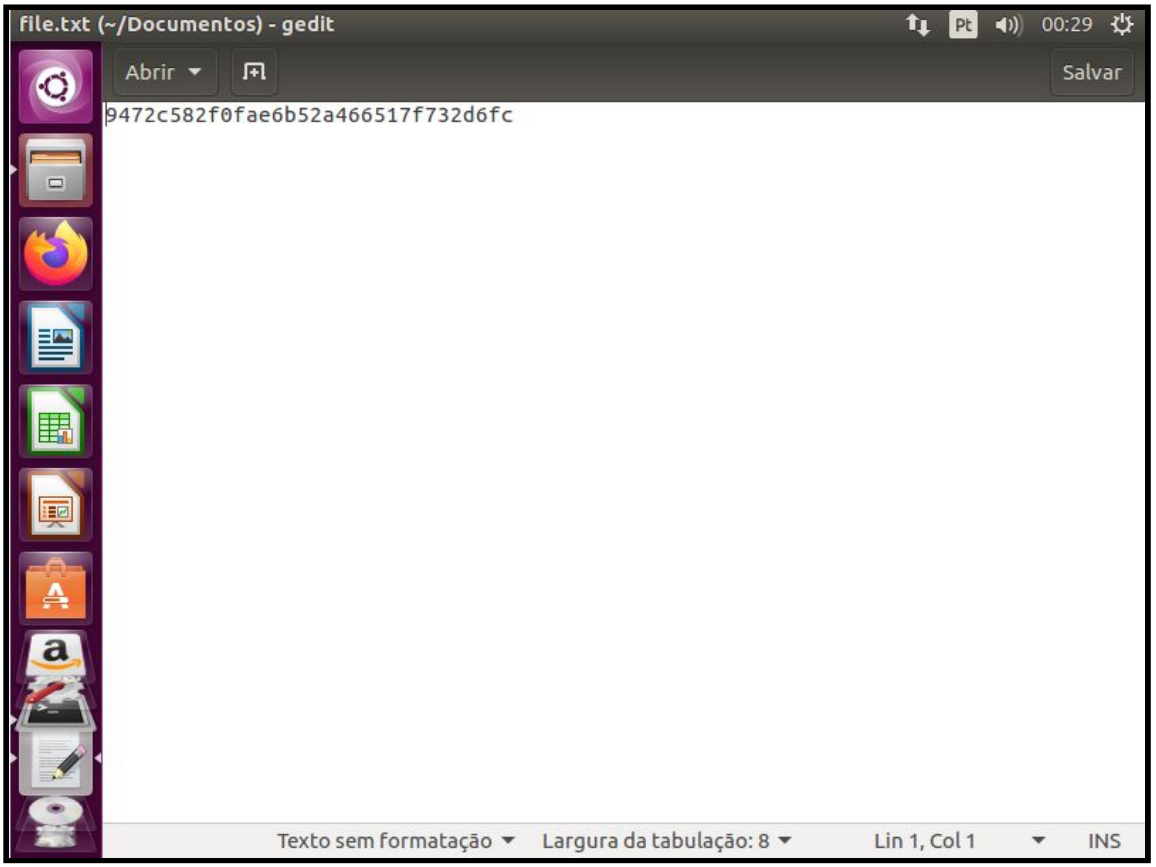
Após a execução do comando “./test” no mesmo cmd executado a compilação podemos observar que o programa solicita para o usuário escrever uma string, que no caso foi escrito a string “abcde12345”.



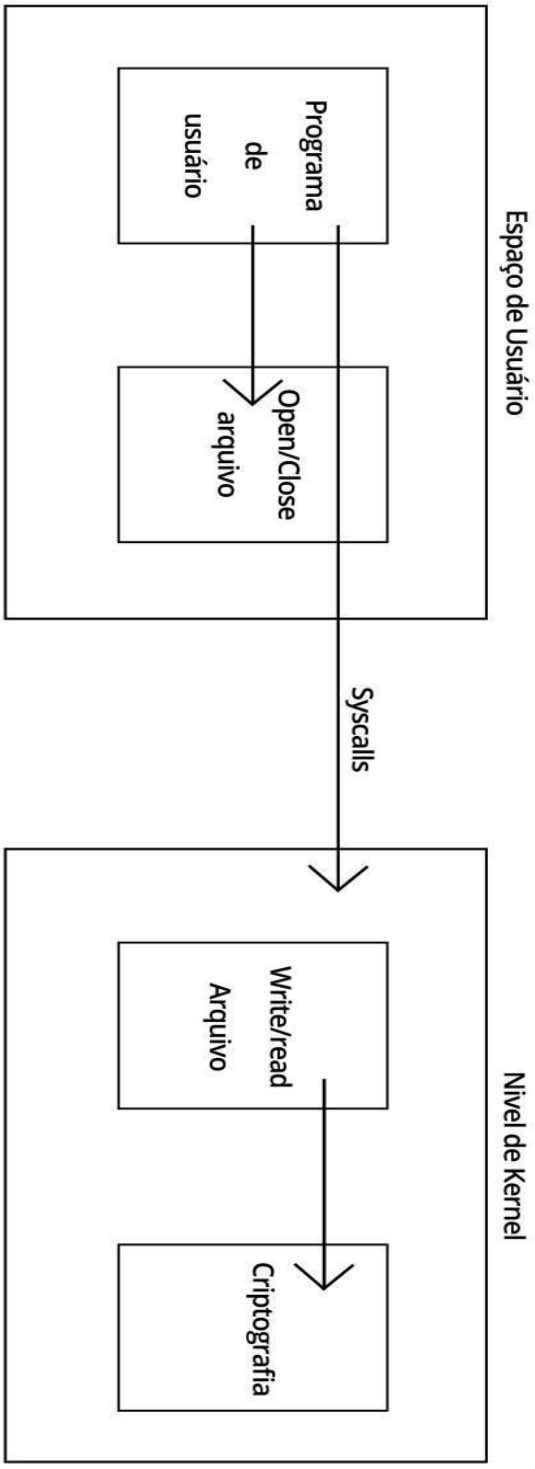
Em simultâneo a isso é possível observar que o próprio programa gera o arquivo test automático após a compilação e o arquivo “file.txt”, que fica armazenada uma string criptografada referente a entrada do *“write the String: ”,* essa string criptografada pode ser observada com mais detalhes na próxima imagem.



Como já dito na descrição da imagem anterior, essa seria a string criptografada, onde foi possível estar comparando o seu valor com o nosso antigo programa e com um site que realiza esse tipo de criptografia, e comprovando que nosso projeto atende aos requisitos solicitados, onde realiza a chamada da syscall de write, leitura comunicando nível de usuário com kernel e o mais importante, criptografando corretamente, onde cria um arquivo automático e salva essa criptografia nele.



Abaixo pode se observar com mais detalhe um diagrama no qual foi construído, para melhor visualização de como o nosso programa funciona e se comporta em relação as devidas interações do usuário que estão sendo indicados nos retângulos do diagrama:



4. CONCLUSÃO

De acordo com os resultados obtidos é possível concluir que o objetivo principal do projeto foi alcançado com êxito, todos os aspectos da implementação das chamadas de sistema foram realizados, testados.

Relatando um pouco da nossa principal problemática na parte de implementação, pois tivemos uma grande dificuldade na manipulação de arquivo, onde se difere um pouco dos programas que já tínhamos feito no sistema operacional windows. E também, tivemos alguns erros de compilações e execuções relacionados ao arquivo e o uso da função *strcpy()*. Entretanto, no final, após a realização de testes, adaptações como por exemplo trocar a função citada por memcpy(), alterar funções e meios de trabalhar com arquivos, conseguimos chegar num ótimo resultado que permitia a criação de um arquivo assim que o usuário executasse o comando, e escrevesse nesse arquivo a frase criptografada corretamente que o usuário inseriu na entrada.

E por fim, com esse experimento pudemos conhecer mais sobre o *Linux* no que se diz respeito as system calls e como implementá-las no módulo de kernel, além de como usá-las, como criar, compilar, instalar no módulo modificado.