Relatório Sistemas Operacionais B
Projeto 2:
System Call Implementation

Gabriel Gonçalves Mattos Santini RA: 18189084,
Luiz Vinícius dos Santos Ruoso RA: 18233486,
Marcelo Germani Olmos RA: 18048298,
Victor Felipe dos Santos RA: 18117820,
Victor Luiz Fraga Soldera RA: 18045674.

Resumo. *Dentro de sistemas desenvolvidos baseados em Linux, projetado por Linus Torvalds, é evidenciado a modularização e a comunicação entre o Kernel e Driver via API. Para esse projeto, foi proposto a implementação de duas chamadas de sistemas (System Calls) em um Kernel Linux, com o objetivo de implementar e compilar um Kernel modificado para permitir que o programa do usuário armazene e leia arquivos de maneira cifrada através da API criptográfica presente neste Kernel.*

1. Introdução

Neste projeto foi proposto que fizéssemos duas *System Calls* (forma com que o programa solicita informações do Kernel): **write_crypt**, que foi feita para cifrar arquivos usando o algoritmo AES (”Advanced Encryption Standard”) com tamanho de bloco de 128 bits. - e **read_crypt**, para permitir que os arquivos encriptados por AES possam ser lidos. Além disso, alteramos o tamanho máximo da string aceita para 256 bytes, ou seja, 256 caracteres.

2. Desenvolvimento

2.1. Criação da Syscall

Para início da implementação foi usado como base para a criação da *syscall*,o tutorial disponível no blog **Medium**, disponibilizado pelo autor: Sreehari. Seguindo os procedimentos, foi implementado um *bash script* como abaixo, para execução dos passos:

```
#!/bin/bash

echo "Copiando com.h e ccom.c para linux-source ----------"

sudo mkdir -p /usr/src/linux-source-4.15.0/info
sudo cp ./com.h /usr/src/linux-source-4.15.0/info

echo "obj-y:=ccom.o">Makefile

echo "É necessário agora editar alguns arquivos--------"
echo "Primeiro, editar Makefile do kernel"
```

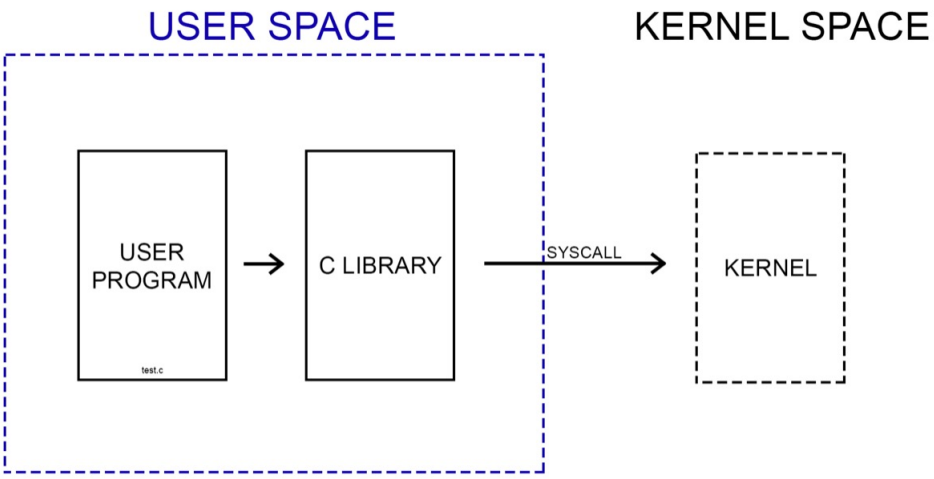
```
echo "Garanta que na linha 882, haja seu /info "

sudo gedit /usr/src/linux-source-4.15.0/Makefile
sudo gedit /usr/src/linux-source-4.15.0/arch/x86/entry/syscalls/syscall_

echo "Adcione ao final do arquivo o prototipo da(s) sua(s) syscalls "
echo "Exemplo: asmlinkage long sys_hello(void)"

sudo gedit /usr/src/linux-source-4.15.0/include/linux/syscalls.h

echo "Compilando Kernel: "
cd /usr/src/linux-source-4.15.0/ && sudo make clean
cd /usr/src/linux-source-4.15.0/ && sudo make -j2

echo "Modules_install"
cd /usr/src/linux-source-4.15.0/ && sudo make modules_install -j2

echo "Make install"

cd /usr/src/linux-source-4.15.0/ && sudo make install -j2

sudo reboot
```

Nesse script, foi deixado explicido os procedimentos realizados, desde a crição da pasta */info* dentro do kernel que seria compilado com a *syscall* nova até mesmo a compilação do mesmo. Dos pontos da implementação que podemos salientar, são a inclusão da nova chamada de sistema dentro de *syscall_64.tbl* e *syscalls.h*, e o modelo de função que deve ser seguido: *asmlinkage long sys_hello(void);*

2.2. Funcionamento da Syscall

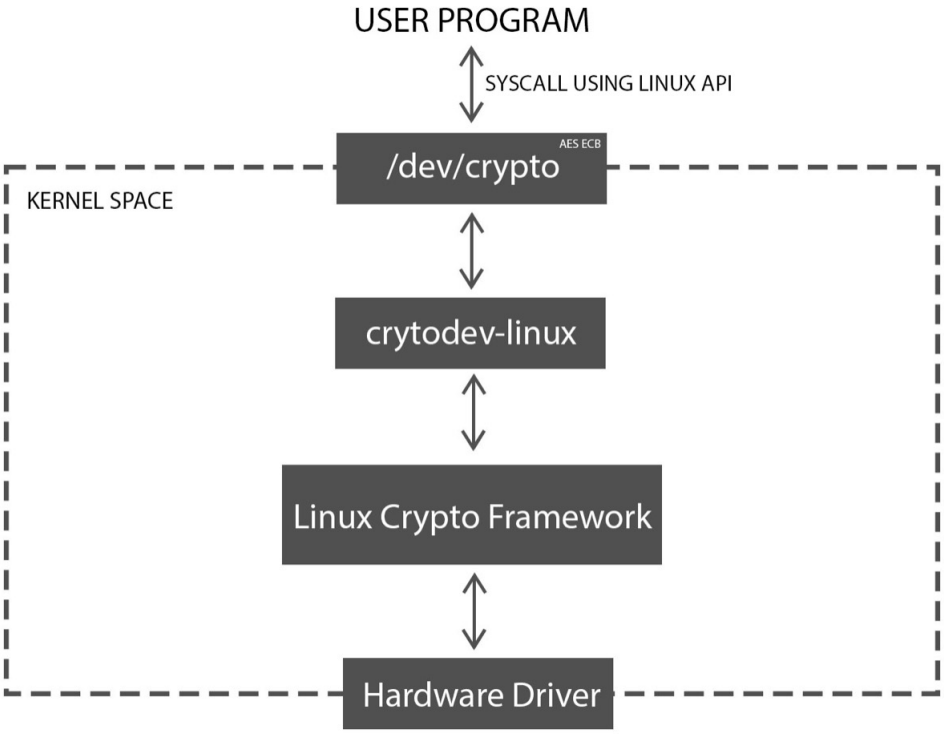
A *syscall*, como seu próprio nome diz, é uma chamada de sistema, que propõem um comunicação entre um software a nível de usuário e o Sistema Operacional, esse pode tratar esse chamada chamando até mesmo um driver que irá realizar operações em hardware. No caso desse experimento, nosso sistema, como na imagem abaixo realiza a *syscall* e aciona uma operação no Kernel.

Figura 1. Syscall



Já com a chamada realizada pela *syscall*, utilizando da API do Linux para criptografia, que se inicia pela seleção do tipo da mesma dentro da */dev/crypto* (um device driver), no caso a **AES EBC**, que aciona a o módulo *cryptodev-linux* que se utiliza do *Linux Crypto Framework* que guarda todos os parâmetros para realização de cálculos em Hardware, esse Framework é a API referenciada nesse relatório.

Figura 2. Chamadas Internas em espaço de Kernel



2.3. Criptografia e Decodificação

Para a realização da criptografia e decodificação, foram reutilizadas as funções já feitas das mesmas, que podem ser encontrado neste **Github** que apresenta o relatório do projeto anterior, com algumas alterações(explicadas abaixo), assim como o código do mesmo.

Tanto para a criptografia quanto para a decodificação, usou-se o método de criptografia *AES-128* em modo ECB, que diferente do modo CBC, não possui IV(vetor de inicialização da criptografia). Como adicional do projeto, foi expandido o tamanho máximo da string que pode ser criptografada, de 16 bytes para 256 bytes (16 caracteres para 256 caracteres), ou seja, aumentou-se de 1 bloco para 16 blocos.

2.4. Resultados

2.5. Biblioteca - com.h

Para o desenvolvimento do projeto, inicialmente, estudamos e compreendemos a maneira de implementar uma chamada de sistema e, após isso, seguimos para desenvolver as chamadas propostas no projeto: *write_crypt* e *read_crypt*. Para isso, criamos uma biblioteca, *com.h*, e o programa *ccom.c*, que realiza as operações utilizando as funções implementadas na biblioteca e já existentes no sistema, deixando o código mais modularizado e organizado.

A biblioteca *com.h* se trata do repositório de funções que realiza, em quase sua totalidade, todas as operações de criptografia, descriptografia e gerenciamento de blocos

das mesmas, devido ao implemento de 256 bytes para as operações, já que por padrão a API aceita apenas 16 bytes para tal.

Dentro do arranjo da biblioteca, podemos salientar esse bloco, que se encontra nas funções: *encode_trigger* e *decode_trigger* .

```
for(i=0;i < getRightTotalBlocksBasedOnStringSize(size_of_string);i++){
    sg_init_one(&sk.sg, scratchpad + (i * 16), 16);
    skcipher_request_set_crypt(req, &sk.sg, &sk.sg, 16, NULL);
    ret = test_skcipher_encdec(&sk, 1);
    if (ret)
        goto out;
    resultdata = sg_virt(&sk.sg);
    memcpy(msgToEncypt + (i * 16), resultdata, 16);
 }
```

O bloco demonstrado, mostra a implementação do suporte para 256 bytes na API de kernel. Nele, andamos com o ponteiro da *string* original de 16 bytes em 16 bytes e em contra partida, esse mesmo conteúdo pode ser descriptografado/criptografado e adicinado com o mesmo parâmetro para a *string* de retorno, no final do *loop*.

Por fim, para o restante da biblioteca, temos a implementação padrão sugerida no site *kernel.org*, com a diferença dessa adaptação, a remoção do suporte para o 'IV' (padrão para AES CBC) e a inclusão em parâmetro de função AES ECB.

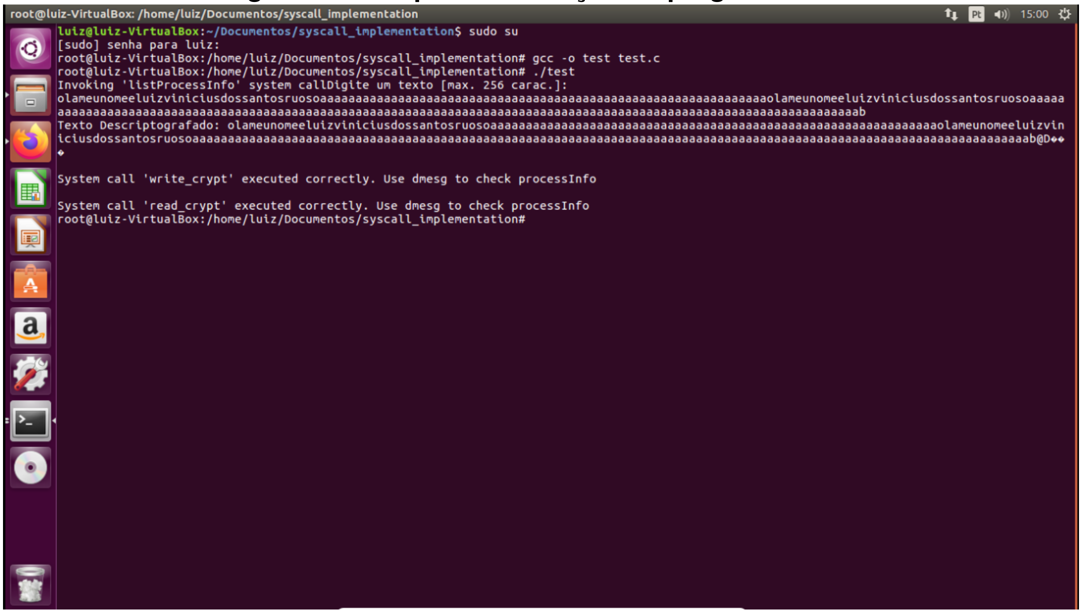
2.6. Problemas encontrados

Inicialmente, para a utilização das chamadas de criptografia, foi necessária a ativação do suporte para criptografia ECB, que estava desativada nas configurações do kernel. Ao testarmos as funções, inicialmente, percebemos que ocorria alguns problemas, por conta de "sujeira"nas variáveis utilizadas, em especifico, nos vetores de caracteres, para solucionar isso, utilizamos a função *mem_set*, de forma a garantirmos que o vetor inteiro seria inicializado com 0. Além disso, ao concluirmos o projeto básico, decidimos tentar aumentar o tamanho máximo da string aceita, mas isso causou um problema na função *str_cpy* quando a string continha mais que 67 bytes. Com o intuito de resolvermos este problema, modificamos o código, substituindo a função *str_cpy* por *mem_cpy*.

3. Resultados

Após a implementação do programa e o posterior solucionamento dos problemas encontrados, o programa foi concluído com sucesso, como possível observar na figura 3.

Figura 3. Exemplo de execução do programa teste



Abaixo, nas figuras 4 e 5, é possível verificar o funcionamento interno do programa.

Figura 4. Funcionamento interno da Syscall

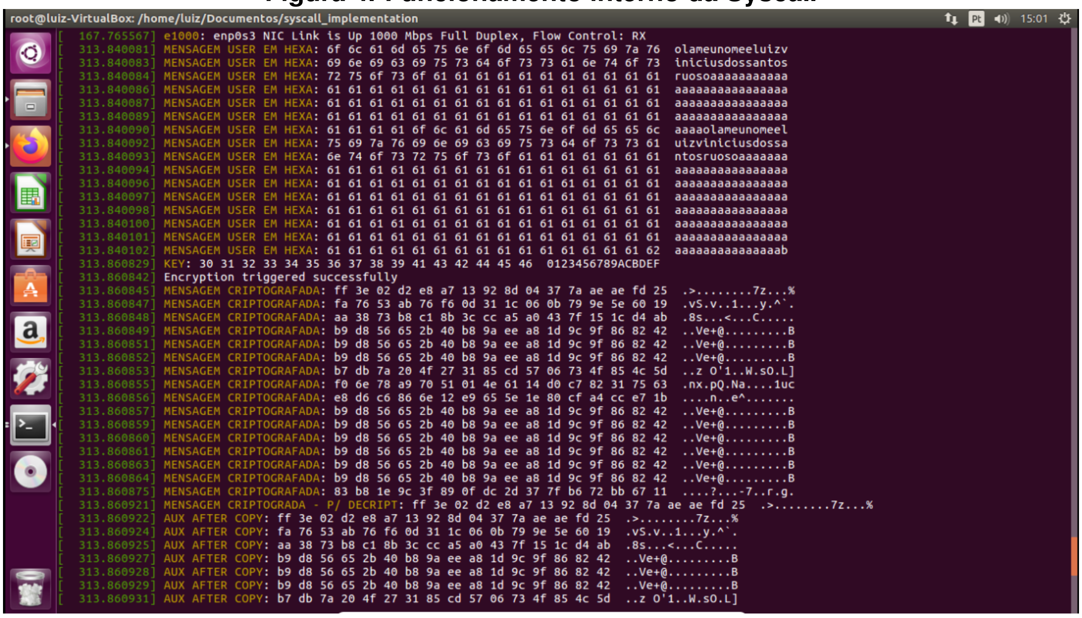
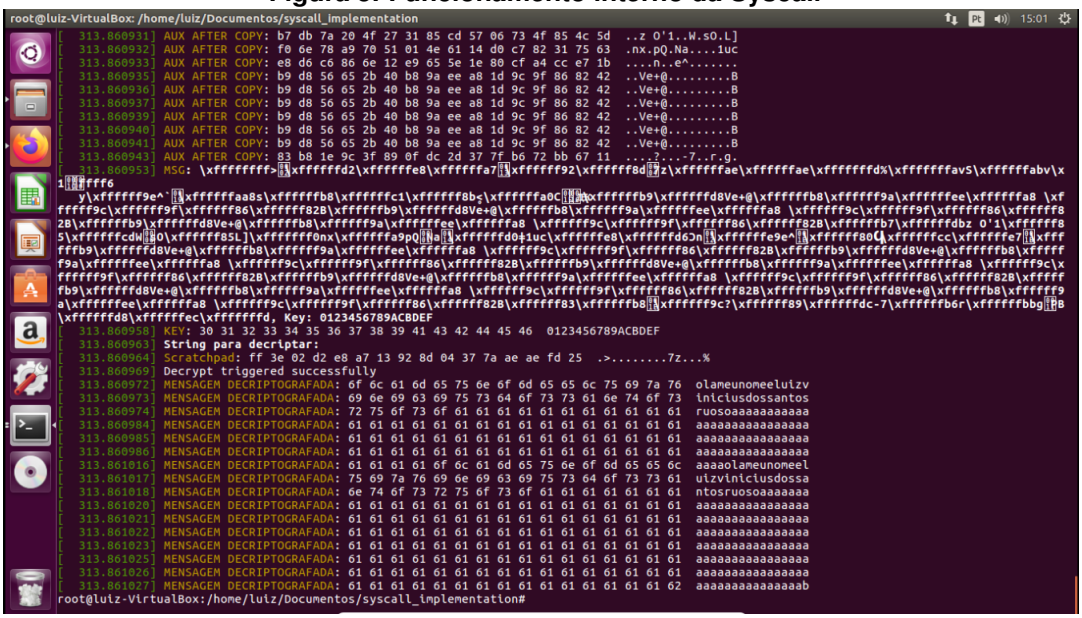


Figura 5. Funcionamento interno da Syscall



4. Conclusão

Durante o trabalho utilizamos majoritariamente informações que foram obtidas através do experimento anterior e também alguns sites recomendados. Como dito anteriormente, vários componentes trabalho anterior foram aproveitados e além disso tivemos que implementar o write e read. Por fim obtivemos sucesso ao utilizar as chamadas de sistema do kernel Linux para realizar o experimento proposto propostas.

5. Referências

System Call in OS: Types and Examples

The Linux Kernel Archives

Implementing a system call in Linux Kernel 4.7.1

Andrew S. Tanenbaum, Modern Operating Systems