Alexandre Tolomeotti Enokida
André Felipe Mireski
Victor Angelo Souza Santos

Projeto 02 - Desenvolver Módulo do Kernel Linux

Relatório técnico de projeto prático solicitado pelo professor Rodrigo Campiolo na disciplina de Sistemas Operacionais do Bacharelado em Ciência da Computação da Universidade Tecnológica Federal do Paraná.

Universidade Tecnológica Federal do Paraná – UTFPR
Departamento Acadêmico de Computação – DACOM
Bacharelado em Ciência da Computação – BCC

Campo Mourão
Setembro / 2024

Resumo

Nesse projeto foi desenvolvido um módulo simples para o kernel linux, cujo objetivo é monitorar a execução de uma `syscall`, configurável via o arquivo `/proc/scm_target_syscall`, e incrementar um contador sempre que ela for chamada, salvando seu valor no arquivo `/proc/scm_syscall_counter`.

Palavras-chave: kernel, linux, syscall, módulo, contador.

Sumário

1 Introdução 4
2 Proposta 4
3 Métodos 5
3.1 Arquivos proc 5
3.1.1 Características principais dos arquivos `/proc` 5
3.1.2 Estrutura do `/proc` 5
3.1.3 Operações de leitura/escrita no `/proc` 6
3.1.4 Vantagens do `/proc` 6
3.1.5 Arquivos `/proc` usados no código 6
3.1.6 Funcionalidade dos arquivos `/proc` 7
3.2 Kprobes 8
3.2.1 Descrição 8
3.2.2 Estrutura 8
3.2.3 Como usar e Funcionamento interno 8
3.2.4 Observações importantes 9
3.3 Kernel 10
3.3.1 Preparar o Ambiente e Trocar para Usuário Root 10
3.3.2 Baixar e Preparar o Código Fonte do Kernel 10
3.3.3 Configurar o Kernel 10
3.3.4 Compilar o Kernel 11
3.3.5 Instalar o Kernel 11
3.3.6 Reiniciar e Verificar o Novo Kernel 12
4 Resultados 12
4.1 Dificuldades encontradas 13
4.1.1 Trocar o símbolo da chamada de sistema à ser monitorada 13
4.1.2 Uso de vsnprintf 15
5 Conclusão 15
6 Referências 15
7 Anexo A - Figuras 17
Figura 1 - Compilação do módulo 17
Figura 2 - Carregamento do módulo 17
Figura 3 - Arquivos `/proc` 17
Figura 4 - Monitorando *sys_open* 17
Figura 5 - Monitorando *sys_read* 17
Figura 6 - Descarregando o módulo 17

1 Introdução

Em um sistema operacional (*UNIX-like*), chamadas de sistemas acontecem a todo instante. Monitorar e analisar a frequência dessas chamadas pode fornecer dados valiosos, que podem ser utilizados posteriormente por aplicações (*e.g*: SystemTap) com outras finalidades, como por exemplo: monitorar problemas funcionais do sistema operacional (REDHAT. . . ,). Neste contexto, decidimos desenvolver um módulo do núcleo Linux para contar o número de ocorrências de uma dada chamada de sistema.

Para a realização do projeto, usamos como ferramentas de estudo e aprendizado: documentações do kernel, repositórios do GitHub, threads em fóruns de desenvolvimento de sistemas (*StackOverflow*, *UnixStackExchange*). Para a implementação do projeto, usamos alguns métodos disponibilizados pela própria interface do kernel para o desenvolvimento de módulos (*e.g*: `simple_read_from_buffer(...)`). Também usamos um módulo específico do kernel que é apresentado na seção 3.2.

Ademais questões de implementações, bem como dificuldades encontradas durante o desenvolvimento, são descritas nas seções a seguir e também no código-fonte do projeto.

2 Proposta

A proposta do grupo consiste em criar um módulo para o kernel linux que realize a contagem do número de vezes que uma certa chamada de sistema foi executada. Para configurar o módulo após a sua inicialização e monitorar o avanço do contador, serão utilizados arquivos `/proc`.

Por padrão, a `syscall` monitorada será a `sys_newuname`, chamada sempre que informações sobre o kernel são requeridas, como quando o comando uname –a é executado.

A chamada monitorada poderá ser alterada via o arquivo `/proc/scm_target_syscall`. Sempre que a *syscall* monitorada mudar, o contador será zerado.

Já o valor do contador poderá ser visualizado acessando o arquivo `/proc/scm_syscall_counter`.

A ideia da equipe era implementar um módulo que explorasse algum conceito interessante e não fosse extremamente trabalhoso de ser desenvolvido. Objetivo alcançado, tendo em vista que foi possível desenvolver um método que intercepta a execução de chamadas de sistema, o que poderia ser expandido para além de incrementar um contador, como obter alguma informação da `syscall`, como quem chamou, ou quando ela foi executada, facilitando o processo de `debugging`.

3 Métodos

3.1 Arquivos proc

Os arquivos no diretório `/proc` fazem parte de um sistema de arquivos especial do Linux chamado `procfs` (Process File System). O sistema de arquivos `/proc` é um sistema de arquivos virtual que permite a comunicação entre o kernel e os programas em espaço de usuário. Ele fornece informações sobre o sistema e processos em execução e permite ajustar certas configurações do kernel. O `/proc` é uma interface muito útil para interação com o kernel, sem a necessidade de recompilação ou reinicialização.

3.1.1 Características principais dos arquivos `/proc`

- **Sistema de arquivos virtual**: O `/proc` não contém dados persistentes em disco, como os arquivos tradicionais. Em vez disso, os arquivos no `/proc` são gerados dinamicamente pelo kernel e representam informações do estado do sistema. Quando um arquivo no `/proc` é lido, o kernel gera o conteúdo em tempo real.
- **Informações sobre o sistema**: O `/proc` fornece uma maneira de visualizar o estado do sistema operacional. Por exemplo, ele contém informações sobre:
  - Processos em execução (`/proc/[pid]` para processos individuais).
  - Configurações de memória e CPU (`/proc/meminfo`, `/proc/cpuinfo`).
  - Estatísticas e status de dispositivos.
- **Interface de configuração e monitoramento**: O `/proc` não só permite que os usuários leiam dados do sistema, mas também possibilita escrever em certos arquivos para modificar parâmetros do kernel. Por exemplo, é possível alterar configurações de rede ou de memória escrevendo em arquivos apropriados dentro do `/proc`.
- **Arquivos pseudo**: Os arquivos em `/proc` são chamados de "pseudo-arquivos"porque eles não contêm dados tradicionais. Eles são gerados dinamicamente pelo kernel em resposta às operações de leitura e escrita.

3.1.2 Estrutura do `/proc`

A árvore de diretórios em `/proc` tem entradas que correspondem a informações e configurações do kernel. Algumas dessas entradas incluem:

- `/proc/[pid]/`: Diretórios que contêm informações sobre processos específicos, onde `[pid]` é o ID do processo. Por exemplo, `/proc/1/` contém informações sobre o processo `init` (PID 1).

- `/proc/cpuinfo`: Exibe informações sobre o processador, como modelo, velocidade e número de núcleos.
- `/proc/meminfo`: Mostra detalhes sobre a utilização da memória.
- `/proc/stat`: Contém estatísticas do sistema, como o uso de CPU.
- `/proc/sys/`: Diretório que permite ajustar parâmetros do kernel em tempo de execução. Por exemplo, você pode alterar parâmetros de rede ou configurar limites de memória.

3.1.3 Operações de leitura/escrita no `/proc`

O módulo define funções de callback que são chamadas sempre que o usuário tenta ler ou escrever nos arquivos criados. Essas funções implementam a lógica do que acontece quando o arquivo é acessado. No projeto, a função `target_write()` é usada para permitir que o usuário defina uma nova `syscall` para monitorar, sendo chamada sempre que algo é escrito no arquivo `/proc/scm_target_syscall`. Já a `counter_read()` permite que o usuário veja o número de vezes que a `syscall` foi chamada, executando sempre que alguém lê o arquivo `/proc/scm_syscall_counter`.

3.1.4 Vantagens do `/proc`

- **Interatividade com o kernel**: Usando `/proc`, os desenvolvedores podem expor dados e configurações do kernel diretamente para o espaço de usuário, sem a necessidade de recompilar o kernel ou criar APIs complexas.
- **Diagnóstico e monitoramento**: Os arquivos `/proc` são amplamente usados para monitorar o estado do sistema e para depuração, já que fornecem uma visão detalhada dos processos e do hardware.
- **Configuração dinâmica**: Com a interface `/proc`, muitos parâmetros do kernel podem ser ajustados em tempo real, como configurações de rede, sem a necessidade de reiniciar o sistema.

3.1.5 Arquivos `/proc` usados no código

No contexto do projeto desenvolvido, os arquivos no diretório `/proc` são utilizados como interfaces para comunicação entre o módulo do kernel e o espaço do usuário, permitindo tanto a leitura de informações quanto a modificação do comportamento do módulo. Essa é uma prática comum para módulos que precisam expor informações e parâmetros ao usuário de uma forma simples e direta, sem a necessidade de uma interface de usuário complexa.

1. `/proc/scm_target_syscall`

    - Esse arquivo é utilizado para definir qual syscall o módulo deve monitorar. O valor inicial referencia a `syscall sys_newuname`, mas ele pode ser alterado pelo usuário. Sempre que o conteúdo desse arquivo é modificado, o módulo do kernel altera a chamada monitorada e zera o contador de ocorrências.
    - **Leitura e Escrita:**
        - O usuário pode ler o arquivo para verificar qual syscall está sendo monitorada.
        - Para alterar a `syscall` monitorada, o usuário pode escrever, no arquivo, o símbolo do `/proc/kallsyms` referente a `syscall` desejada, como mostra a figura 3. Quando o módulo recebe essa nova informação, ele atualiza o ponto de interceptação (`kprobe`) e zera o contador.

2. `/proc/scm_syscall_counter`

    - Esse arquivo expõe o contador de execuções da syscall monitorada. A cada vez que a syscall definida no arquivo `/proc/scm_target_syscall` é invocada, o contador é incrementado.
    - **Leitura:**
        - O usuário pode ler esse arquivo a qualquer momento para verificar quantas vezes a syscall monitorada foi chamada desde o último zeramento (ou desde que o módulo foi carregado).

3.1.6 Funcionalidade dos arquivos `/proc`

A criação de arquivos no `/proc` permite ao módulo fornecer uma interface simples para que o usuário interaja com o kernel. Em vez de manipular variáveis diretamente dentro do código do kernel, o usuário pode alterar parâmetros e visualizar informações diretamente no sistema de arquivos.

No código, os seguintes aspectos foram implementados:

- **Leitura:** O código implementa funções de leitura associadas a esses arquivos, de forma que o conteúdo dos arquivos `/proc/scm_target_syscall` e `/proc/scm_syscall_counter` seja lido pelo usuário com comandos como `cat /proc/scm_target_syscall` ou `cat /proc/scm_syscall_counter`.
- **Escrita:** Para permitir a modificação da syscall monitorada, foi criada uma função de escrita que atualiza o comportamento do módulo toda vez que o arquivo `/proc/scm_target_syscall` é alterado pelo usuário (com o comando `echo 'syscall_name' > /proc/scm_target_syscall`).

3.2 Kprobes

3.2.1 Descrição

Kernel Probes (`Kprobes`) é um módulo do kernel que possui um mecanismo para dinamicamente invadir qualquer rotina do kernel e coletar informações de depuração e desempenho sem interrupções (KENISTON; PANCHAMUKHI; HIRAMATSU,). Em outras palavras, `Kprobes` é um mecanismo de instrumentação do kernel. Um `kprobe` pode virtualmente ser inserido em qualquer função do kernel.

3.2.2 Estrutura

O sistema provê a interface do módulo `Kprobes` com dois métodos de construção e destruição para um `kprobe`.

- **int** register_kprobe(**struct** kprobe *kp); − registra **um** `kprobe`.
- **void** unregister_kprobe(**struct** kprobe *kp); − destrói **um** `kprobe`.

Existem diversos atributos presentes na estrutura de um `kprobe` (LINUX. . . , b), entretanto, neste trabalho, iremos usar somente dois:

- **const char** *symbol_name;.
- kprobe_pre_handler_t pre_handler; − função de pré-manipulação (*pre-handler*).

Na documentação do módulo (KENISTON; PANCHAMUKHI; HIRAMATSU,) também é possível identificar outros métodos interessantes, como por exemplo: habilitar e desabilitar um `kprobe`., entretanto, por falta de tempo, não iremos explorá-los neste trabalho.

3.2.3 Como usar e Funcionamento interno

Para começar a usá-lo é necessário definir um `probepoint` e um *pre-handler*. Um `probepoint` é um endereço onde o `kprobe` será inserido (MAVINAKAYANAHALLI et al., 2006). O probepoint pode ser definido através de um símbolo que é atribuído ao campo `symbol_name` da estrutura do `kprobe`. Este símbolo é resolvido internamente pelo kernel. Símbolos são nomes simbólicos que são mapeados para endereços de memória do kernel (tais como funções) (KALLSYMS,). Símbolos de chamadas de sistema podem ser encontradas através do diretório `/proc/kallsyms` (STACKOVERFLOW. . . , a).

Quando um `kprobe` é registrado, o `Kprobes` copia a instrução sondada e substitui os primeiros bytes da instrução sondada com uma instrução de ponto de interrupção (*breakpoint*). Quando a CPU acessa a instrução sondada, a instrução de *breakpoint* é

invocada, uma interrupção de software (`trap`) é gerada e o controle do sistema passa para o `Kprobes`. Então, o `Kprobes` invoca o *pre-handler*, que recebe como parâmetro o próprio `kprobe` e os valores dos registradores da chamada de sistema (STACKOVERFLOW..., b) antes da instrução sondada ser executada. Após a execução do *pre-handler*, o `Kprobes` executa a instrução sondada da cópia inicial. Caso executasse apenas a instrução original, precisaria remover o *breakpoint*, criando um intervalo de tempo onde outra CPU poderia invocar a mesma chamada de sistema e não capturá-la. Por último, será executada a função de pós-manipulação (*post-handler*) (caso atribuída ao `kprobe`), e a execução continua na instrução sequente ao `probepoint`.

3.2.4 Observações importantes

A primeira aparição do `Kprobes` foi na *release linux-2.6.9-rc2* (MAVINAKAYANAHALLI et al., 2006), portanto, o módulo desenvolvido não irá funcionar para sistemas com núcleo nas versões inferiores à *linux-2.6.9-rc2*. Além disso, é necessário que algumas opções de configuração do kernel estejam habilitadas. São elas:

- CONFIG_KPROBES=y
  - habilita o módulo do `Kprobes` (CATEEE..., a).
  - Pode ser encontrado no seguinte caminho usando `make menuconfig`: *General architecture-dependent options* → *Kprobes*.
- CONFIG_MODULES=y
  - habilita o carregamento de módulos no kernel em execução (CATEEE..., b).
- CONFIG_MODULE_UNLOAD=y
  - habilita o descarregamento de módulos no kernel em execução (CATEEE..., c).
- CONFIG_KALLSYMS=y
  - usado internamente pelo `Kprobes` para a resolução de símbolos em endereços (KENISTON; PANCHAMUKHI; HIRAMATSU,).
- CONFIG_KALLSYMS_ALL=y
  - usado internamente pelo `Kprobes` para a resolução de símbolos em endereços (KENISTON; PANCHAMUKHI; HIRAMATSU,).

3.3 Kernel

O bom funcionamento de um módulo para o kernel linux depende da versão do núcleo que está sendo executada no momento, pois, ao trocar de versão, poderão haver comportamentos, funções e recursos que não funcionarão corretamente, ou que ainda não estarão disponíveis, afetando a execução do projeto.

O módulo deste projeto foi desenvolvido em uma máquina que executava a versão `6.8.1` do kernel linux. Nesse contexto, para garantir melhores chances de replicação, nessa seção será abordado como compilar a versão do kernel em questão.

3.3.1 Preparar o Ambiente e Trocar para Usuário Root

Para realizar a instalação e compilação do kernel, é necessário estar logado como usuário root e garantir que os pacotes necessários estejam instalados.

- Troque para o usuário root:

```
1    su
```

- Instale os pacotes necessários, incluindo o `make` e ferramentas adicionais:

```
1    apt-get update
2    apt-get install build-essential libncurses-dev
         bison flex libssl-dev libelf-dev
```

3.3.2 Baixar e Preparar o Código Fonte do Kernel

1. Faça o download do kernel versão 6.8.1:

```
1 wget https://cdn.kernel.org/pub/linux/kernel/v6.x/linux
    -6.8.1.tar.xz
```

2. Descompacte o arquivo no diretório `/usr/src`:

```
1 tar -xvf linux-6.8.1.tar.xz -C /usr/src
```

3. Entre no diretório do kernel:

```
1 cd /usr/src/linux-6.8.1
```

4. Faça uma cópia de segurança da configuração atual do kernel:

```
1 cp /boot/config-$(uname -r) .config
```

3.3.3 Configurar o Kernel

Para configurar o kernel, utilizamos duas ferramentas: `make localmodconfig` e `make menuconfig`.

- Gerar a configuração inicial com os módulos em uso:

```
1    make localmodconfig
```

  Durante a execução, várias perguntas serão feitas sobre a configuração de módulos específicos. No experimento, foram aceitas as recomendações padrão para simplificar o processo.

- Customizar a configuração com `menuconfig`:

```
1    make menuconfig
```

  Modificações recomendadas:

  – Habilitar o suporte ao sistema de arquivos NTFS:
    * File Systems → DOS/FAT/EXFAT/NT Filesystems → NTFS file system support: Habilitado.
  – Desabilitar o suporte à virtualização:
    * Virtualization: Desabilitado.

  Ademais parâmetros obrigatórios do kernel para o funcionamento do módulo desenvolvido são descritos na seção 3.2.4. Após realizar as modificações, salve as mudanças no arquivo `.config`.

3.3.4 Compilar o Kernel

Com a configuração ajustada, compile o kernel e os módulos. Utilizamos todos os núcleos de CPU disponíveis para otimizar o processo.

- Compile o kernel e instale os módulos:

```
1    make -j$(nproc) && make modules_install -j$(nproc)
```

  O parâmetro `-j`(*nproc*) utiliza o número de núcleos de CPU disponíveis para acelerar a compilação.

3.3.5 Instalar o Kernel

Após a conclusão da compilação, o kernel pode ser instalado e configurado no sistema.

- Instale o kernel:

```
1    make install
```

- Atualize o GRUB (se estiver utilizando o GRUB como gerenciador de boot):

```
1    update-grub
```

3.3.6 Reiniciar e Verificar o Novo Kernel

Depois que a instalação for concluída, reinicie o sistema para carregar o novo kernel.

- Reinicie o sistema:

```
1    reboot
```

- Verifique se o novo kernel está em uso:

```
1    uname -r
```

Após o reinício, o sistema detectará automaticamente o novo kernel (6.8.1) e o configurará como padrão.

4 Resultados

O primeiro passo para executar o novo módulo é compilar o projeto. Para tanto, deve-se executar o comando make para compilar os arquivos de módulo do kernel, como mostra a figura 1.

Em seguida, o novo módulo, `syscall_counter`, deve ser instalado utilizando o comando sudo insmod syscall_counter.ko.

Quando o módulo termina de ser carregado, logs são gerados pelo kernel, indicando que o carregamento foi concluído e que tudo funcionou bem até aquele ponto. Esses logs podem ser vistos via dmesg -wH, que estabelece um `watch` nos logs do kernel.

O carregamento do módulo pode ser visualizado na figura 2.

Após o carregamento, são gerados dois novos arquivos no diretório `/proc`: `scm_target_syscall` e `scm_syscall_counter`. Eles, respectivamente, definem a chamada de sistema monitorada e exibem o valor atual do contador no momento da leitura do arquivo, como mostra a figura 3.

Para detectar a chamada da `syscall` foi utilizado o módulo `Kprobes`, que, basicamente, estabelece um ponto de interrupção que pode ser utilizado para executar alguma ação sempre que ele é acionado. A biblioteca é explicada em mais detalhes na seção 3.2.

Nesse contexto, sempre que uma nova chamada para a `syscall` é detectada, o contador é incrementado em 1, e um `log` é gerado.

Para alterar qual `syscall` monitorará, deve-se editar o arquivo `scm_target_syscall`, informando o símbolo correspondente à `syscall` para o `Kprobes`. Porém, não são aceitos qualquer símbolo, o módulo só resolve símbolos que estão listados no arquivo

`/proc/kallsyms` e as chamadas de sistema geralmente atendem o seguinte padrão de nomenclatura `__x64_{syscall}`.

Sempre que algo é escrito no arquivo `scm_target_syscall`, o `kprobe` vigente é limpo e uma nova instância para a nova `syscall` é criada, zerando o contador. Desde que o símbolo informado no arquivo seja reconhecido pelo módulo, interceptações devem começar a ocorrer para a nova chamada.

O processo para alterar a `syscall` monitorada para `sys_open` pode ser visto na figura 4.

Porém, a `sys_open` não gerou interceptação alguma durante os testes, todavia, demonstra que o contador é zerado na troca.

Por outro lado, a `sys_read` gera um volume muito elevado de interceptações, como mostra a figura 5.

Por fim, o módulo pode ser descarregado, removendo o `kprobe` e os arquivos `/proc`, como mostra a figura 6.

Através dos resultados apresentados, foi possível demonstrar que o módulo está funcionando corretamente e cumpriu todos os seus objetivos.

4.1 Dificuldades encontradas

4.1.1 Trocar o símbolo da chamada de sistema à ser monitorada

Como apresentado na seção 4, é possível efetuar a troca da chamada de sistema à ser monitorada através do arquivo `/proc/scm_target_syscall`, escrevendo um símbolo de chamada de sistema nele.

Toda vez que o arquivo é escrito, ele invoca uma função de `callback` nomeada target_write que:

1. Valida se a entrada é válida (*i.e.* cabe dentro do *buffer* de 64 posições). Se sim, continua com a execução da função.
2. Copia o conteúdo do arquivo para a variável global target_sys_call.
3. Remove o `kprobe` anterior.
4. Copia o símbolo de target_sys_call para o atributo *pre-handler* do `kprobe` global.
5. Registra novamente o `kprobe` global.

Entretanto, durante o desenvolvimento, estávamos com problemas, pois após tentar registrar o `kprobe` novamente, ele falhava no registro e retornava o erro EINVAL −22.

Após algum tempo procurando possíveis causas para o problema, deparamos com a seguinte resposta do fórum (UNIX...,). A partir da resposta do usuário, identificamos que estávamos inicializando o `kprobe` incorretamente:

Antes:

```
1  static struct kprobe kp = {
2      .symbol_name = DEFAULT_SYSCALL,
3  };
4  ...
5  static ssize_t target_write(...)
6  {
7      ...
8      kp.symbol_name = target_syscall;
9      int ret = register_kprobe(&kp); // <— ERROR EINVAL -22
10     ...
11 }
```

Depois:

```
1  static struct kprobe kp = {
2      .symbol_name = DEFAULT_SYSCALL,
3  };
4  ...
5  static ssize_t target_write(...)
6  {
7      ...
8      struct kprobe kp_temp = {
9          .symbol_name = target_syscall,
10         .pre_handler = handler_pre,
11     };
12     kp = kp_temp;
13     int ret = register_kprobe(&kp); // Retorna 0 = OK
14     ...
15 }
```

Quando o módulo é carregado inicialmente com o símbolo padrão `__x64_sys_newuname`, ele irá registrar o `kprobe`, e o `Kprobes` irá resolver o símbolo em um endereço no kernel, sobrescrevendo o atributo `kprobe_opcode_t *addr` da estrutura do `kprobe` global com o endereço da chamada de sistema correspondente ao símbolo `__x64_sys_newuname` de forma implícita. O problema é que não estávamos limpando este campo que nunca definimos de forma explícita na inicialização do módulo, portanto, ao substituirmos o atributo *pre-handler* do `kprobe` global, infringímos o item 3 (KENISTON; PANCHAMUKHI; HIRAMATSU,) que diz que o registro de um `kprobe` irá falhar se o símbolo e o endereço forem especificados simultaneamente. O problema é resolvido como

mostra o trecho de código “Depois”, onde inicializamos um `kprobe` temporário com o campo `*addr` vazio.

4.1.2 Uso de vsnprintf

No código-fonte do módulo desenvolvido, é possível encontrar uma função nomeada `counter_read(...)`. Resumidamente, usamos essa função para realizar a exportação do valor do contador da chamada de sistema vigente. Também vimos que para o desenvolvimento de módulos do núcleo Linux, importamos cabeçalhos de `linux.h` e não de `stdio.h` (LINUX..., a), como tradicionalmente é visto em programas de nível de usuário. Por exemplo: a função `printf(...)` é equivalente à função `printk(...)`, entretanto, a do kernel só pode ser invocada do nível do núcleo.

Sabendo disso, efetuamos buscas na Internet de possíveis funções equivalentes à `sprintf(...)`. Na documentação do kernel, conseguimos encontrar uma semelhante a ela: `vsnprintf(...)` (THE...,). Entretanto, não conseguimos utilizar ela corretamente. O resultado da `string` formatada possuía caractéres especiais ao final dela. Após isso, decidimos manter o uso da `sprintf(...)`. Também encontramos que o uso do `sprintf(...)` em um módulo do kernel é propenso à possíveis falhas tais como *buffer overflow* (STACKOVERFLOW,). Como a função é utilizada para leitura do contador, não identificamos que seria um grande problema para o funcionamento do módulo (por mais que exista falhas de segurança).

5 Conclusão

Desenvolver um módulo do núcleo Linux proporcionou à equipe diversos desafios técnicos e deixou ainda mais evidente para os membros que para realizar a análise e projetar um programa (neste caso, o módulo) que opere sobre um sistema tão complexo quanto o sistema operacional − que visa sempre desempenho −, é necessário que haja grandes esforços, como: obter conhecimentos teóricos sobre a organização do sistema operacional e seus componentes; pensar sempre visando o desempenho; ter bastante foco nas leituras das documentações técnicas (vide 4.1.1);.

6 Referências

CATEEE - $\mathrm{CONFIG}_{K}PROBES.[S.l.].Disponívelem$: $<https : //cateee.net/lkddb/web - lkddb/KPROBES-.html>https : //cateee.net/lkddb/web - lkddb/KPROBES.html.Citadonapágina9.$

CATEEE - $\mathrm{CONFIG}_{M}ODULES.[S.l.].Disponívelem$: $<https : //cateee.net/lkddb/web - lkddb/MODULES-.html>https : //cateee.net/lkddb/web - lkddb/MODULES.html.Citadonapágina9.$

CATEEE - CONFIG$_{M}ODULE_{U}NLOAD$.[S.l.].$Disponívelem$: <$https$:-
$//cateee.net/lkddb/web$ − $lkddb/MODULE\_UNLOAD$-
.html>$https$: $//cateee.net/lkddb/web - lkddb/MODULE\_UNLOAD.html.Citadonapágina9$.

KALLSYMS. [S.l.]. Disponível em: <https://jrgraphix.net/man/K-
/kallsyms>https://jrgraphix.net/man/K/kallsyms. Citado na página 8.

KENISTON, J.; PANCHAMUKHI, P. S.; HIRAMATSU, M. *Kernel Probes (Kprobes)*. [S.l.]. Disponível em: <https://docs.kernel.org/trace/kprobes-
.html>https://docs.kernel.org/trace/kprobes.html. Citado 3 vezes nas páginas 8, 9 e 14.

LINUX Kernel Device Drivers Development Book. [S.l.]. Disponível em: <https://sysplay.github.io/books/LinuxDrivers/book/Content/Part02-
.html>https://sysplay.github.io/books/LinuxDrivers/book/Content/Part02.html. Citado na página 15.

LINUX kprobe struct. [S.l.]. Disponível em: <https:/-
/github.com/torvalds/linux/blob/master/include/linux/kprobes-
.h#L59>https://github.com/torvalds/linux/blob/master/include/linux/kprobes.h#L59. Citado na página 8.

MAVINAKAYANAHALLI, A. et al. Probing the guts of kprobes. In: *Linux Symposium*. [S.l.: s.n.], 2006. v. 6, p. 5. Citado 2 vezes nas páginas 8 e 9.

REDHAT - SystemTap. [S.l.]. Disponível em: <https:/-
/docs.redhat.com/en/documentation/red_hat_enterprise_linux/5-
/html/systemtap_beginners_guide/understanding-how-systemtap-
works>https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/5/html/systemtap_
how-systemtap-works. Citado na página 4.

STACKOVERFLOW. [S.l.]. Disponível em: <https://stackoverflow.com/a-
/12264427>https://stackoverflow.com/a/12264427. Citado na página 15.

STACKOVERFLOW - Does kallsyms have all the symbol of kernel functions? [S.l.]. Disponível em: <https://stackoverflow.com/questions/20196636/does-kallsyms-have-
all-the-symbol-of-kernel-functions>https://stackoverflow.com/questions/20196636/does-
kallsyms-have-all-the-symbol-of-kernel-functions. Citado na página 8.

STACKOVERFLOW - How are system calls stored in pt$_{r}egs$?[$S.l.$].$Disponívelem$: <$https$: $//stackoverflow.com/questions$-
/33104091/$how - are - system - calls - stored - in - pt -$
$regs$>$https$: $//stackoverflow.com/questions$/33104091/$how - are - system - calls - stored - in$

THE Linux Kernel API - The Linux Kernel Documentation. [S.l.]. Disponível em: <https:-
//docs.kernel.org/core-api/kernel-api.html#c.vsprintf>https://docs.kernel.org/core-
api/kernel-api.html#c.vsprintf. Citado na página 15.

UNIX - stackexchange. [S.l.]. Disponível em: <https:/-
/unix.stackexchange.com/questions/647270/unable-to-register-
kprobe>https://unix.stackexchange.com/questions/647270/unable-to-register-kprobe.
Citado na página 14.

7 Anexo A - Figuras

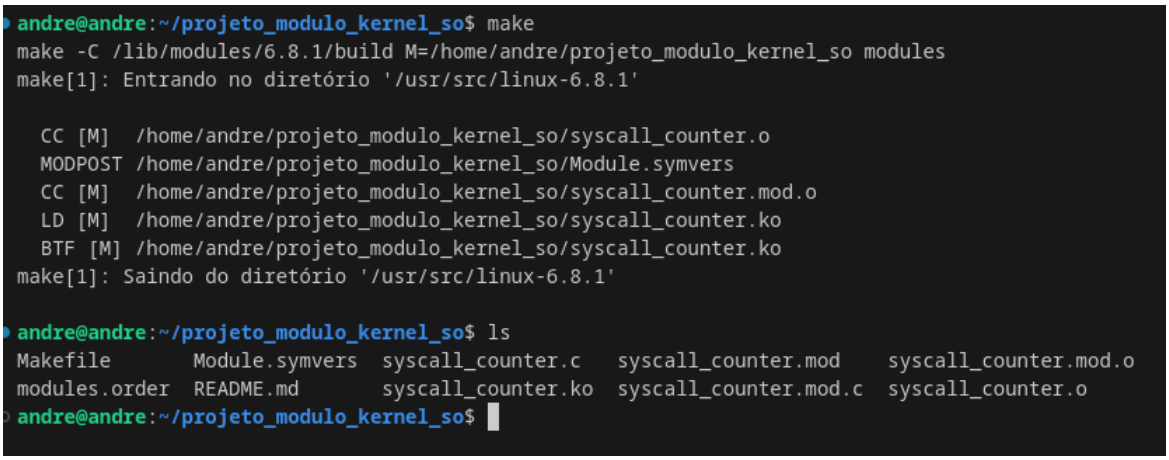

Figura 1 – Compilação do modulo

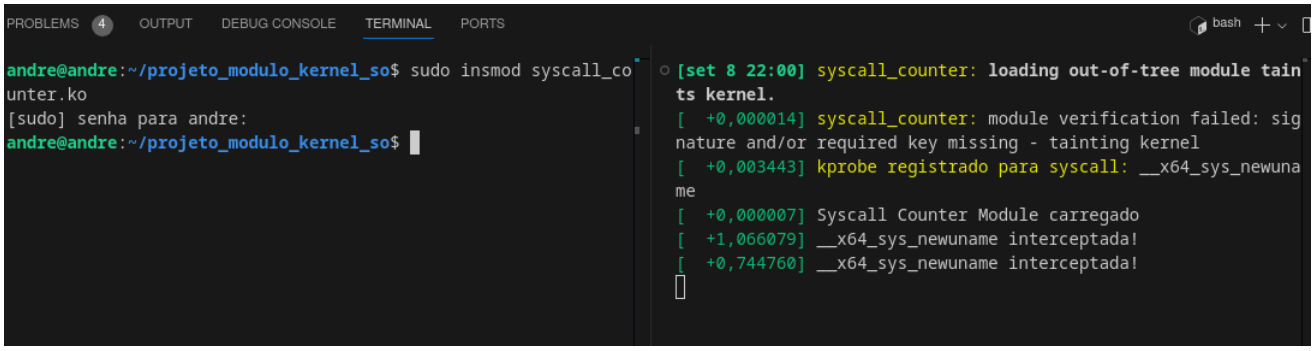

Figura 2 – Carregamento do modulo

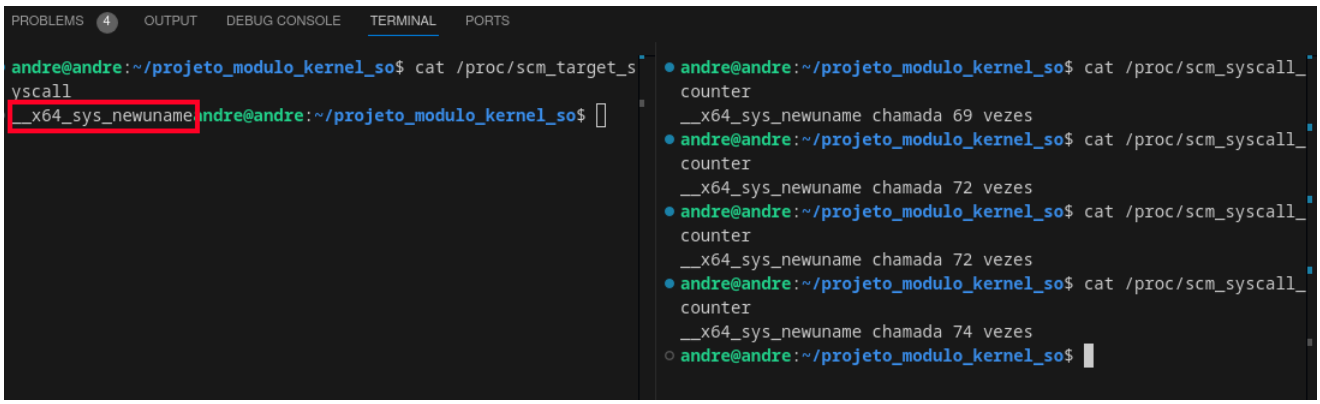

Figura 3 – Arquivos `/proc`

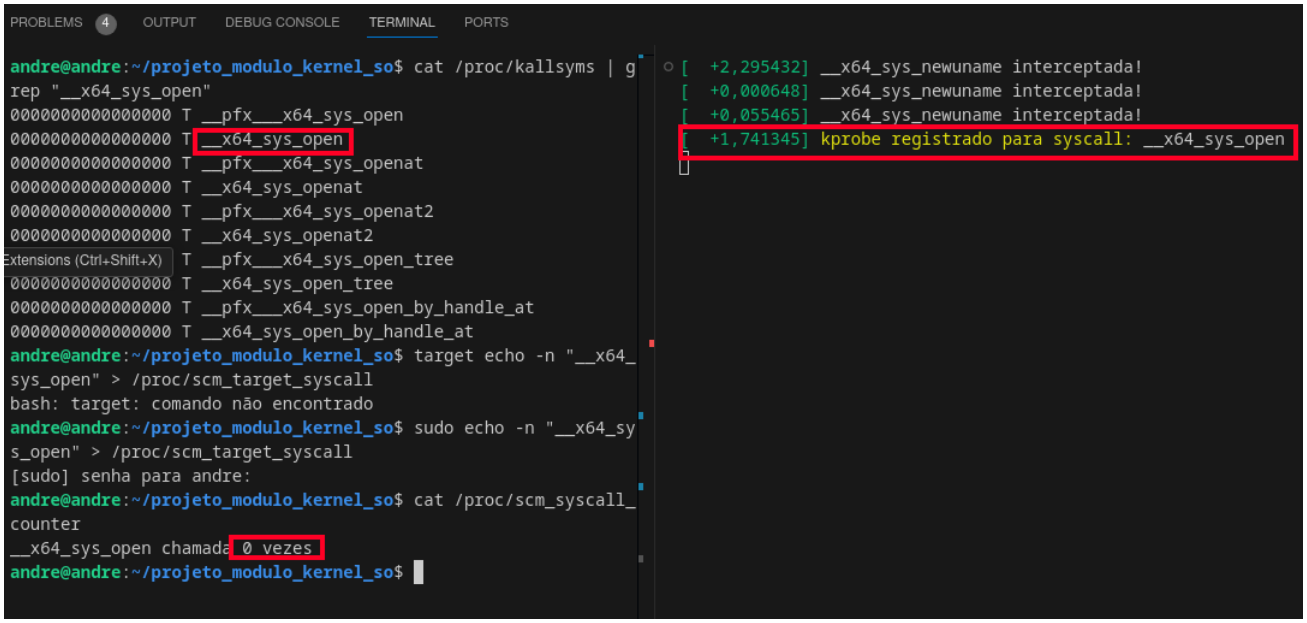

Figura 4 – Monitorando *sys_open*

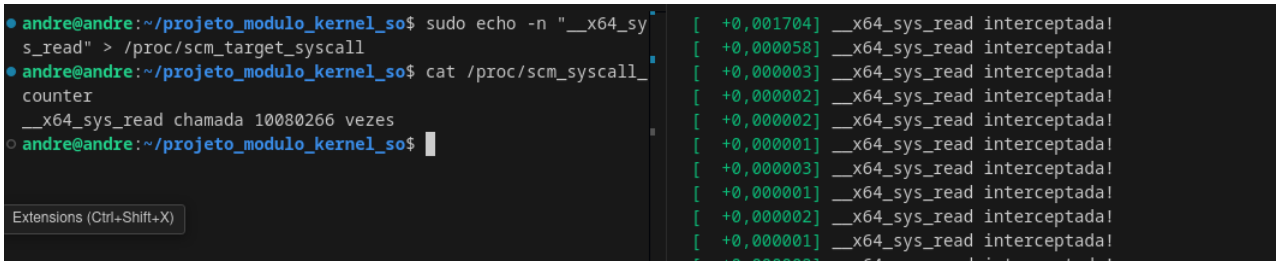

Figura 5 – Monitorando *sys_read*

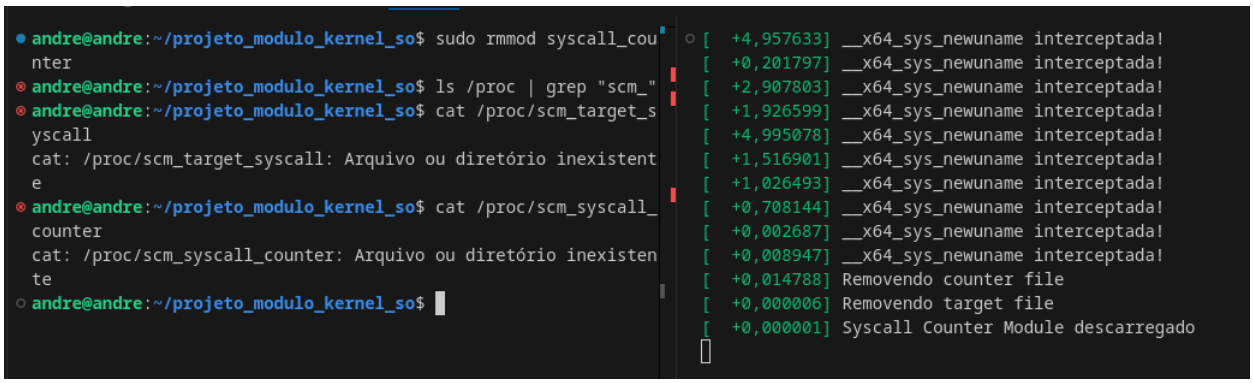

Figura 6 – Descarregando o módulo