Mecanismos de Acesso ao Código do Kernel

Objetivo: detalhar as instruções que transferem execução de user mode para kernel mode e explicar como syscalls são incorporadas nos programas.

1- INT 0X80

https://elixir.bootlin.com/linux/v6.15.6/source/arch/x86/entry/entry_32.S#L933

(Estrutura do pt_regs - 32 bits) - Mostrar o funcionamento

https://elixir.bootlin.com/linux/v6.15.6/source/arch/x86/include/asm/ptrace.h#L12

(Estrutura do handle do_int80_syscall_32) - Mostrar o funcionamento

https://elixir.bootlin.com/linux/v6.15.6/source/arch/x86/entry/syscall_32.c#L246

Contexto Histórico e Uso

Método Tradicional e Lento: Historicamente, em sistemas Linux x86 de 32 bits, a instrução `INT $0x80` era o principal mecanismo para programas solicitarem serviços do kernel. Hoje em dia, em hardware moderno, essa é considerada uma "rota lenta" para chamadas de sistema. A maioria das bibliotecas C (como a `glibc`) em sistemas modernos usa métodos mais rápidos (como `sysenter` ou `syscall` em CPUs mais recentes) para a maioria das chamadas, exceto durante a inicialização do processo.

Fallback e Compatibilidade: Apesar de ser lenta, a `INT $0x80` ainda é crucial para:

- **Programas e bibliotecas legadas:** Muitos programas e bibliotecas compiladas há muito tempo ainda podem ter `INT $0x80` embutido.
- **Fallback do vDSO:** O Virtual Dynamic Shared Object (vDSO) é uma técnica que o kernel usa para expor algumas funções do kernel diretamente no espaço do usuário para chamadas de sistema mais rápidas. Se o hardware não suporta um método mais rápido, o vDSO pode recorrer à `INT $0x80`.
- **Chamadas de sistema reiniciadas:** Se uma chamada de sistema precisar ser reiniciada (por exemplo, após um sinal), ela pode voltar a usar `INT $0x80`, independentemente de como foi iniciada originalmente.
- **Compatibilidade com 64 bits:** Embora esse código seja para 32 bits, programas de 64 bits também podem usar `INT $0x80` (nesse caso, eles são redirecionados para `entry_INT80_compat` em kernels de 64 bits).

Como Funciona: Passos Principais

1. Entrada e Salvamento de Contexto (SYM_FUNC_START(entry_INT80_32))

- **ASM_CLAC**: Limpa o "Alignment Check Flag" (AC flag é um bit no registro EFLAGS que controla a capacidade do kernel de acessar páginas de memória do modo usuário) no registrador de flags, o que é importante para evitar interrupções de alinhamento inesperadas durante a transição do espaço de usuário para o kernel.
- **pushl %eax**: O **número da chamada de sistema** (syscall number) é passado no registrador eax. Este valor é salvo na pilha do kernel, tornando-se parte do pt_regs->orig_ax (original eax antes da chamada, que contém o número da syscall).
- **SAVE_ALL pt_regs_ax=$-ENOSYS switch_stacks=1**: Esta macro é crucial. Ela salva o estado completo do processador do programa que fez a chamada na **pilha do kernel**. Isso inclui todos os registradores de uso geral (ebx, ecx, edx, esi, edi, ebp), o ponteiro de pilha (esp), o ponteiro de instrução (eip), e o registrador de flags (eflags). O objetivo é que o kernel possa restaurar o estado do programa do usuário exatamente como ele estava antes da interrupção. O switch_stacks=1 indica que haverá uma troca para a pilha do kernel, pois as chamadas de sistema sempre são executadas na pilha do kernel, não na pilha do usuário.

2. Execução da Chamada de Sistema

- **movl %esp, %eax**: O endereço do pt_regs (a estrutura na pilha onde o estado do processo foi salvo) é movido para eax.
- **call do_int80_syscall_32**: Esta é a chamada para a função C principal no kernel que realmente lida com a chamada de sistema. A função do_int80_syscall_32 usará o número da chamada de sistema (do pt_regs->orig_ax) e os argumentos (que são passados nos registradores ebx, ecx, edx, esi, edi, ebp de acordo com a convenção de chamada) para encontrar e executar a função do kernel apropriada.

3. Pós-Execução e Restauração (.Lsyscall_32_done:)

- **STACKLEAK_ERASE**: Esta macro (provavelmente relacionada a recursos de segurança para mitigar vazamentos de informações na pilha) pode apagar partes da pilha do kernel que não são mais necessárias para evitar que dados sensíveis sejam expostos.

4. Retorno ao Espaço de Usuário (restore_all_switch_stack:)

- **SWITCH_TO_ENTRY_STACK**: Garante que estamos na pilha correta de entrada do kernel antes de retornar.
- **CHECK_AND_APPLY_ESPFIX**: Lida com possíveis problemas de ESP (ponteiro de pilha) que podem ocorrer devido a certas otimizações ou correções de segurança.

- **SWITCH_TO_USER_CR3 scratch_reg=%eax**: Altera o registrador de controle CR3 para o do espaço de usuário. O CR3 contém o endereço base da tabela de páginas que o processador usa para traduzir endereços virtuais para físicos. Ao trocar o CR3, o kernel garante que as próximas instruções (quando o controle retornar ao programa do usuário) usarão o mapeamento de memória correto do processo do usuário.
- **BUG_IF_WRONG_CR3**: Uma verificação de depuração para garantir que a troca de CR3 foi bem-sucedida.
- **RESTORE_REGS pop=4**: Esta macro restaura todos os registradores do processador que foram salvos anteriormente (incluindo eflags, eip, esp, etc.). O pop=4 indica que 4 bytes (o orig_eax e error_code, se presentes) devem ser ignorados na pilha.
- **CLEAR_CPU_BUFFERS**: Limpa quaisquer buffers internos da CPU, como o TLB (Translation Lookaside Buffer) e o cache de instruções, o que pode ser necessário após uma transição de contexto de segurança.
- **iret**: A instrução **iret** **(Interrupt Return)** é a parte final e mais importante. Ela faz o trabalho pesado de retornar do kernel para o espaço do usuário. A iret retira da pilha o eip, cs (segmento de código), eflags, esp e ss (segmento de pilha), restaurando o ambiente do usuário e permitindo que o programa continue sua execução a partir do ponto onde foi interrompido.

5. Tratamento de Erros (.Lasm_iret_error:)

- Esta seção é um **manipulador de erro** para a instrução iret. Se a iret encontrar um erro (por exemplo, um estado inválido na pilha ao tentar retornar ao usuário), o controle salta para .Lasm_iret_error.
- **pushl $0 e pushl $iret_error**: Empilha um código de erro e um endereço de rotina de tratamento de erro.
- **jmp handle_exception**: Redireciona o controle para uma rotina de tratamento de exceções mais geral no kernel, que pode registrar o erro, tentar recuperar ou até mesmo encerrar o processo.
- **_ASM_EXTABLE(.Lirq_return, .Lasm_iret_error)**: Esta macro é usada para registrar uma "tabela de exceções" que o kernel usa para lidar com falhas na execução de código no kernel. Isso significa que se a instrução iret em .Lirq_return falhar, o kernel sabe para onde pular (.Lasm_iret_error) para tratar o erro, em vez de travar o sistema.

Resumo

1. Um programa de 32 bits no espaço de usuário executa INT $0x80.
2. O processador entra no modo kernel e salta para entry_INT80_32.
3. O estado completo do processador do usuário é salvo na pilha do kernel.
4. O número da chamada de sistema (eax) e os argumentos são preparados.

5. A função do_int80_syscall_32 no kernel é chamada para executar a ação solicitada.
6. Após a conclusão da chamada de sistema, o estado do processo do usuário é restaurado.
7. O registrador CR3 é trocado de volta para o do usuário.
8. A instrução iret é executada, retornando o controle ao programa de usuário no ponto de interrupção.
9. Se a iret falhar, um tratamento de erro é acionado.

Em essência, entry_INT80_32 é a "porta de entrada" segura e controlada para o kernel, permitindo que programas de usuário solicitem serviços do sistema operacional enquanto o kernel gerencia a transição de privilégios e a proteção da memória.

2- SYSENTER/SYSEXIT

https://elixir.bootlin.com/linux/v6.15.6/source/arch/x86/entry/entry_32.S#L786

pt_regs é o mesmo da INT0x80

(do_SYSENTER_32) - Mostrar o funcionamento

https://elixir.bootlin.com/linux/v6.15.6/source/arch/x86/entry/syscall_32.c#L361

Contexto e Importância

- **Método Preferido no 32-bit:** Em sistemas Linux de 32 bits, a instrução SYSENTER é o método **preferencial e mais rápido** para programas solicitarem serviços do kernel, especialmente quando o recurso X86_FEATURE_SEP está disponível. Ela é amplamente utilizada pelo **vDSO** (Virtual Dynamic Shared Object), que é uma área de memória mapeada no espaço do usuário pelo kernel para expor certas funções de forma otimizada.
- **Desvantagens do SYSENTER:** Ao contrário de INT $0x80 que salva automaticamente muitos registradores na pilha, SYSENTER é "minimalista". Ela:
    - Carrega SS (segmento de pilha), ESP (ponteiro de pilha), CS (segmento de código) e EIP (ponteiro de instrução) de **MSRs (Model-Specific Registers)** previamente programados. Isso significa que o kernel precisa configurar esses MSRs antes que o SYSENTER possa ser usado.
    - **Não salva** os registradores EIP, ESP ou EFLAGS na pilha. Isso exige que o código do kernel salve esses valores manualmente, o que é um dos desafios de usar SYSENTER.
    - **Limpa os flags IF (Interrupt Flag) e VM (Virtual 8086 Mode Flag)** em RFLAGS, o que significa que as interrupções são desativadas ao entrar no kernel.

- **Modo VM86:** O texto menciona que SYSENTER é desabilitado explicitamente no modo VM86 para evitar corrupção de estado, reprogramando os MSRs.
- **Single-stepping (TF):** Um ponto importante é que se um programa de usuário tiver o "Trap Flag" (TF) ativado (usado para single-stepping), o SYSENTER **ainda será single-stepped**. Isso significa que cada instrução dentro do bloco SYSENTER (até __end_SYSENTER_singlestep_region) pode gerar uma interrupção de depuração (#DB). O kernel precisa lidar com isso ignorando essas traps geradas nessa região, o que, embora lento, simplifica o código.

Argumentos da Chamada de Sistema

Os argumentos são passados nos mesmos registradores que para INT $0x80:

- eax: número da chamada de sistema
- ebx: arg1
- ecx: arg2
- edx: arg3
- esi: arg4
- edi: arg5
- ebp: ponteiro para a pilha do usuário (e o sexto argumento, arg6, está em 0(%ebp))

Análise Detalhada do Código (entry_SYSENTER_32)

Vamos explorar o fluxo de execução:

1. Entrada e Preparação Inicial (SYM_FUNC_START(entry_SYSENTER_32))

- **pushfl e pushl %eax**: Ao contrário de INT $0x80, SYSENTER não salva EFLAGS nem EAX. Por isso, o código os salva manualmente na pilha imediatamente. EAX é usado como um registrador temporário (scratch_reg) para a troca de CR3.
- **BUG_IF_WRONG_CR3 no_user_check=1**: Uma verificação de depuração para garantir que estamos no CR3 esperado.
- **SWITCH_TO_KERNEL_CR3 scratch_reg=%eax**: Muda o registrador de controle CR3 para o do kernel. Isso é essencial para que o kernel possa acessar sua própria memória e tabelas de páginas.
- **popl %eax e popfl**: EAX e EFLAGS são restaurados após a troca de CR3.

2. Troca para a Pilha da Task (movl TSS_entry2task_stack(%esp), %esp)

- **movl TSS_entry2task_stack(%esp), %esp**: Esta é uma etapa crucial. O SYSENTER entra com o ESP (ponteiro de pilha) apontando para um local que pode ser a pilha do usuário ou uma pilha temporária. Para uma execução segura e organizada no kernel, o código muda o ESP para a **pilha da task (processo)** que está

atualmente em execução. TSS_entry2task_stack(%esp) calcula o endereço correto da pilha do kernel para a task atual.

3. Salvando o Contexto do Usuário (.Lsysenter_past_esp:)

- **Empilhando Registradores Faltantes:** Como SYSENTER não salva SS, ESP, CS, EIP ou EFLAGS, o código os empilha manualmente, criando uma estrutura pt_regs (Process-Task Registers) na pilha do kernel, semelhante à que INT $0x80 faria automaticamente:
  - pushl $__USER_DS: Segmento de dados do usuário.
  - pushl $0: Placeholder para o ponteiro de pilha do usuário (ESP), que será preenchido mais tarde.
  - pushfl: Flags do processador (EFLAGS).
  - pushl $__USER_CS: Segmento de código do usuário.
  - pushl $0: Placeholder para o ponteiro de instrução do usuário (EIP), que também será preenchido.
  - pushl %eax: O número da chamada de sistema (originalmente em eax). Isso se torna pt_regs->orig_ax.
- **SAVE_ALL pt_regs_ax=$-ENOSYS**: Esta macro salva os **demais registradores** (como ebx, ecx, edx, esi, edi, ebp) na pilha, completando a estrutura pt_regs. Note que stack already switched (a pilha já foi trocada), o que significa que essa macro opera na pilha da task.

4. Limpeza de Flags e Execução da Chamada de Sistema

- **testl $X86_EFLAGS_NT|X86_EFLAGS_AC|X86_EFLAGS_TF, PT_EFLAGS(%esp)**: Verifica se os flags **NT (Nested Task)**, **AC (Alignment Check)** ou **TF (Trap Flag)** estão definidos nos EFLAGS salvos.
- **jnz .Lsysenter_fix_flags**: Se qualquer um desses flags estiver definido, o código salta para .Lsysenter_fix_flags para limpá-los. Isso é crucial porque esses flags podem ter efeitos indesejados no modo kernel. A limpeza é feita antes de reabilitar interrupções para evitar preempção com NT definido.
  - **Single-stepping (TF) handling:** O comentário explica que se TF estiver setado, o kernel "single-steppará" (executará instrução por instrução) até este ponto. O manipulador #DB (Debug Exception) do kernel é configurado para ignorar as traps geradas nessa região.
- **.Lsysenter_flags_fixed:**: Ponto de entrada se os flags já estiverem limpos ou depois de serem corrigidos.
- **movl %esp, %eax**: O endereço da estrutura pt_regs é movido para eax.
- **call do_SYSENTER_32**: Chama a função C principal do kernel que implementa a lógica da chamada de sistema, usando o número da syscall e os argumentos.

- **testb %al, %al e jz .Lsyscall_32_done**: Verifica o valor de retorno de do_SYSENTER_32. Se for zero, a chamada de sistema foi concluída e o controle vai para o ponto de retorno (.Lsyscall_32_done).
- **STACKLEAK_ERASE**: Limpeza da pilha para evitar vazamento de informações.

5. Retorno ao Espaço de Usuário (Opportunistic SYSEXIT)

- O código então prepara para usar a instrução **SYSEXIT** para retornar ao espaço do usuário. SYSEXIT é o complemento de SYSENTER e é também uma instrução rápida de retorno.
- **Preparando a Pilha de Entrada:**
  - **movl PER_CPU_VAR(cpu_tss_rw + TSS_sp0), %eax**: Carrega o ponteiro da pilha do kernel (esp0) da TSS (Task State Segment) do CPU atual em eax. Este é o local para onde SYSEXIT retornará o controle.
  - **subl $(2*4), %eax**: Aloca espaço para eflags e eax na pilha de entrada.
  - **movl PT_EFLAGS(%esp), %edi e movl PT_EAX(%esp), %esi**: Copia os eflags e eax salvos da pilha da task.
  - **movl %edi, (%eax) e movl %esi, 4(%eax)**: Move os valores para a pilha de entrada.
- **Restauração de Registradores do Usuário:**
  - **movl PT_EIP(%esp), %edx**: O ponteiro de instrução do usuário (EIP) é movido para edx (pois SYSEXIT espera o EIP em EDX).
  - **movl PT_OLDESP(%esp), %ecx**: O ponteiro de pilha do usuário (ESP) é movido para ecx (pois SYSEXIT espera o ESP em ECX).
  - mov PT_FS(%esp), %fs: Restaura o registrador de segmento FS.
  - popl %ebx, addl $2*4, %esp (para cx e dx), popl %esi, popl %edi, popl %ebp: Restaura os demais registradores de uso geral que foram salvos.
- **movl %eax, %esp**: Troca o ponteiro de pilha para a pilha de entrada.
- **SWITCH_TO_USER_CR3 scratch_reg=%eax**: Troca o CR3 de volta para o do processo de usuário.
- **CLEAR_CPU_BUFFERS**: Limpa buffers da CPU.
- **Restauração Final de Flags e EAX:**
  - **btrl $X86_EFLAGS_IF_BIT, (%esp)**: Limpa o bit IF (Interrupt Flag) dos eflags na pilha. Isso é importante porque sti (Set Interrupt Flag) é usada separadamente para reabilitar as interrupções após um pequeno atraso, garantindo uma janela de uma instrução antes que as interrupções possam ocorrer.
  - **popfl**: Restaura os eflags do usuário (exceto IF).
  - **popl %eax**: Restaura o eax original do usuário.
- **sti**: Reabilita as interrupções.

- **sysexit**: Esta instrução é o complemento de SYSENTER. Ela usa os valores nos MSRs (que o kernel programou com o CS e SS do usuário), e EDX (para EIP) e ECX (para ESP) para retornar ao espaço do usuário, com os privilégios de usuário.

6. Tratamento de Fallback para FS (2: movl $0, PT_FS(%esp))

- O código inclui um bloco com _ASM_EXTABLE para lidar com casos onde o segmento FS pode não ser restaurado corretamente. Se a instrução mov PT_FS(%esp), %fs (no rótulo 1b) falhar, o controle salta para o rótulo 2:, onde FS é explicitamente zerado, e então o fluxo de retorno é retomado (jmp 1b).

7. Correção de Flags (.Lsysenter_fix_flags:)

- **pushl $X86_EFLAGS_FIXED**: Empilha um valor que contém os flags NT, AC e TF limpos (e outros bits fixos).
- **popfl**: Carrega esse valor nos registradores de flags (EFLAGS), efetivamente limpando NT, AC e TF.
- **jmp .Lsysenter_flags_fixed**: Retorna para o fluxo principal após a correção dos flags.

Resumo

1. O kernel salva manualmente **`EFLAGS`** e **`EAX`** na entrada.
2. Troca para o **`CR3`** e a pilha do kernel.
3. **Construção manual** da estrutura `pt_regs` salvando os registradores ausentes (`SS`, `ESP`, `CS`, `EIP`) e os demais.
4. Limpa flags problemáticos (`NT`, `AC`, `TF`).
5. O número da syscall está em `eax`, argumentos em outros registradores. A função `do_SYSENTER_32` é chamada.
6. Após a execução, o kernel restaura o contexto.

3- SYSCALL/SYSRET

https://elixir.bootlin.com/linux/v6.15.6/source/arch/x86/entry/entry_64.S#L87

(pt_regs - 64 bits) - Mostrar o funcionamento

https://elixir.bootlin.com/linux/v6.15.6/source/arch/x86/include/asm/ptrace.h#L103

(do_syscall_64) - Mostar o funcionamento

https://elixir.bootlin.com/linux/v6.15.6/source/arch/x86/entry/syscall_64.c#L87

Contexto e Características da SYSCALL (64-bit)

Design Otimizado: A instrução SYSCALL de 64 bits foi projetada de forma mais eficiente do que suas predecessoras (INT $0x80, SYSENTER). A forma como os argumentos são mapeados para registradores no Linux (64-bit ABI) se encaixa muito bem com os registradores que a SYSCALL disponibiliza.

Uso Generalizado: É a instrução padrão para chamadas de sistema em bibliotecas C (como glibc) e também é usada em algumas partes do vDSO (Virtual Dynamic Shared Object) para funções de tempo, por exemplo.

O que SYSCALL faz (Hardware):

- Salva o endereço de retorno (rip) no registrador rcx.
- Salva o registrador de flags (rflags) no registrador r11.
- Limpa o bit RF (Resume Flag) em rflags.
- Carrega novos valores para ss (segmento de pilha), cs (segmento de código) e rip (ponteiro de instrução) de MSRs (Model-Specific Registers) previamente programados pelo kernel.
- Aplica uma máscara aos rflags de outro MSR (eliminando a necessidade de CLD e CLAC para limpar flags como o Direction Flag ou Alignment Check).
- **Não salva nada na pilha** e **não altera o rsp** (ponteiro de pilha). Isso significa que o kernel é responsável por salvar e restaurar o contexto completo do usuário.

Registradores na Entrada (do Usuário para o Kernel)

Quando a instrução SYSCALL é executada, os registradores contêm os seguintes valores:

- rax: **Número da chamada de sistema**.
- rcx: **Endereço de retorno** no espaço do usuário (rip original).
- r11: **Flags salvos** do espaço do usuário (rflags original).
- rdi: **Argumento 0** (arg0).
- rsi: **Argumento 1** (arg1).
- rdx: **Argumento 2** (arg2).
- r10: **Argumento 3** (arg3 - *Nota: este precisará ser movido para rcx para conformidade com a ABI C no kernel*).
- r8: **Argumento 4**.
- r9: **Argumento 5**.
- r12-r15, rbp, rbx: São registradores preservados na ABI C e não são alterados pela SYSCALL.

Análise Detalhada do Código (entry_SYSCALL_64)

Vamos seguir o fluxo de execução:

1. Entrada e Troca de Contexto Inicial (SYM_CODE_START(entry_SYSCALL_64))

- **UNWIND_HINT_ENTRY**: Dica para depuradores e ferramentas de unwinding de pilha sobre o início da rotina.
- **ENDBR**: Instrução para segurança (CET - Control-flow Enforcement Technology), marcando um alvo válido para chamadas e saltos.
- **swapgs**: Troca o valor do registrador GS (segmento de dados). GS é usado para apontar para a estrutura de dados por-CPU no kernel. No espaço do usuário, GS pode apontar para o Thread Local Storage (TLS). swapgs alterna entre o GS do usuário e o GS do kernel.
- **movq %rsp, PER_CPU_VAR(cpu_tss_rw + TSS_sp2)**: Salva o **ponteiro de pilha do usuário (rsp)** em um espaço reservado na TSS (Task State Segment) por CPU. Isso é necessário porque SYSCALL não altera rsp, e o kernel precisa do rsp do usuário para restaurar o contexto depois.
- **SWITCH_TO_KERNEL_CR3 scratch_reg=%rsp**: Troca o registrador CR3 (que contém o endereço da tabela de páginas) para o do kernel. O rsp é usado como registrador auxiliar.
- **movq PER_CPU_VAR(cpu_current_top_of_stack), %rsp**: Muda o **ponteiro de pilha para a pilha do kernel** da tarefa atual. A execução da syscall agora ocorre na pilha do kernel.
- **ANNOTATE_NOENDBR**: Indica que o código seguinte não deve conter um ENDBR, para otimização ou por não ser um alvo de branch.

2. Construção da Estrutura pt_regs (entry_SYSCALL_64_safe_stack)

O kernel constrói uma estrutura pt_regs na pilha, que é uma representação do estado do processador do usuário. Isso é fundamental para que a chamada de sistema possa acessar os argumentos e para que o contexto possa ser restaurado corretamente:

- **pushq $__USER_DS**: Empilha o seletor do segmento de dados do usuário (pt_regs->ss).
- **pushq PER_CPU_VAR(cpu_tss_rw + TSS_sp2)**: Empilha o rsp do usuário que foi salvo anteriormente (pt_regs->sp).
- **pushq %r11**: Empilha os rflags salvos pela SYSCALL (pt_regs->flags).
- **pushq $__USER_CS**: Empilha o seletor do segmento de código do usuário (pt_regs->cs).
- **pushq %rcx**: Empilha o rip do usuário salvo pela SYSCALL (pt_regs->ip).
- **pushq %rax**: Empilha o **número da chamada de sistema** (pt_regs->orig_ax).

- **PUSH_AND_CLEAR_REGS rax=$-ENOSYS**: Esta macro salva os demais registradores gerais (rbx, rbp, r12-r15, etc.) na pilha. rax=$-ENOSYS define um valor padrão para rax caso a syscall não seja encontrada.

3. Execução da Chamada de Sistema

- **movq %rsp, %rdi**: Move o ponteiro para a estrutura pt_regs (agora na pilha do kernel) para rdi. rdi é o primeiro argumento para a função C do_syscall_64.
- **movslq %eax, %rsi**: Move o número da chamada de sistema (que estava em rax, mas agora está disponível via pt_regs->orig_ax ou ainda pode estar em eax se não foi clobbered) para rsi. movslq realiza uma extensão de sinal, pois os números de syscall são tratados como inteiros de 32 bits. rsi é o segundo argumento para do_syscall_64.
- **IBRS_ENTER, UNTRAIN_RET, CLEAR_BRANCH_HISTORY**: Estas são instruções e macros relacionadas à mitigação de vulnerabilidades de **execução especulativa** (como Spectre e Meltdown). Elas limpam buffers do processador para evitar vazamento de informações através de canais laterais.
- **call do_syscall_64**: Chama a função C principal do kernel que implementa a lógica da chamada de sistema. Esta função irá despachar para a função específica da syscall com base no número da syscall e nos argumentos. Ela retorna com as IRQs (interrupções) desabilitadas.

4. Retorno ao Espaço de Usuário (SYSRET vs. IRET)

Após a do_syscall_64 retornar, o kernel precisa voltar ao processo de usuário. O Linux de 64 bits tenta usar a instrução **SYSRETQ** por ser mais rápida, mas pode precisar usar IRET (a instrução de retorno de interrupção mais geral) em certas condições.

- **ALTERNATIVE "testb %al, %al; jz swapgs_restore_regs_and_return_to_usermode", "jmp swapgs_restore_regs_and_return_to_usermode", X86_FEATURE_XENPV**:
    - Esta é uma otimização condicional. Se o valor de retorno da syscall em %al for zero (indicando sucesso e, presumivelmente, um "contexto limpo"), e se não estivermos no ambiente Xen PV (Xen Paravirtualized), o código tenta o caminho mais rápido (sysretq).
    - Caso contrário (erro ou Xen PV), ele salta para swapgs_restore_regs_and_return_to_usermode que, internamente, usa iret. A documentação observa que SYSRET tem problemas com endereços não canônicos em algumas CPUs AMD e Intel, o que justifica o fallback para IRET em certas situações ou quando o contexto não é "completamente limpo".
- **syscall_return_via_sysret:**: Este rótulo marca o caminho otimizado para SYSRETQ.
- **IBRS_EXIT**: Saída das mitigações de execução especulativa.
- **POP_REGS pop_rdi=0**: Esta macro restaura a maioria dos registradores salvos na pt_regs, exceto rdi (que será restaurado mais tarde) e rsp (que será trocado).

5. Preparação para SYSRETQ

- **movq %rsp, %rdi**: O rsp atual (ponteiro para a estrutura pt_regs na pilha do kernel) é salvo em rdi.
- **movq PER_CPU_VAR(cpu_tss_rw + TSS_sp0), %rsp**: O ponteiro de pilha é trocado para a **pilha de trampoline** ou uma pilha temporária (o sp0 da TSS). Isso é feito para que os últimos pops para rdi e rsp (do usuário) ocorram em uma pilha controlada pelo kernel.
- **UNWIND_HINT_END_OF_STACK**: Dica para ferramentas de unwinding.
- **pushq RSP-RDI(%rdi)**: Empilha o rsp original do usuário (que está na pt_regs apontada por rdi) na pilha temporária.
- **pushq (%rdi)**: Empilha o rdi original do usuário (que também está na pt_regs) na pilha temporária.
- **STACKLEAK_ERASE_NOCLOBBER**: Limpa partes da pilha para segurança.
- **SWITCH_TO_USER_CR3_STACK scratch_reg=%rdi**: Troca o CR3 de volta para o do processo de usuário.
- **popq %rdi e popq %rsp**: Restaura o rdi e o rsp originais do usuário da pilha temporária. Agora, rsp aponta para a pilha do usuário.

6. Execução de SYSRETQ

- **ANNOTATE_NOENDBR**: Novamente, para otimização ou segurança.
- **swapgs**: Troca o registrador GS de volta para o valor do usuário (revertendo a swapgs inicial).
- **CLEAR_CPU_BUFFERS**: Limpa buffers do processador.
- **sysretq**: A instrução **SYSRETQ** é executada. Esta instrução:
    - Carrega rip de rcx (o endereço de retorno salvo pela SYSCALL).
    - Carrega rflags de r11 (os flags salvos pela SYSCALL).
    - Altera o nível de privilégio (de volta para o usuário).
    - Retorna o controle ao programa de usuário.
- **int3**: Uma interrupção de ponto de parada (breakpoint) para depuração. Não é alcançado em execução normal.

Resumo

1. Um programa de 64 bits no espaço de usuário executa SYSCALL.
2. O hardware SYSCALL salva rcx (rip) e r11 (rflags) e salta para entry_SYSCALL_64.
3. O kernel salva o rsp do usuário, troca para o CR3 e a pilha do kernel.
4. É construída uma estrutura pt_regs completa na pilha do kernel, contendo todos os registradores do usuário (incluindo rip e rflags que foram lidos de rcx e r11).
5. Medidas de segurança para execução especulativa são aplicadas.
6. A função C do_syscall_64 é chamada para executar a lógica da chamada de sistema.

7. Após o retorno da do_syscall_64, o kernel decide se pode usar SYSRETQ (caminho rápido) ou precisa de IRET (caminho lento/fallback).
8. No caminho SYSRETQ: registradores são restaurados, rsp e rdi são movidos para uma pilha temporária, o CR3 é trocado de volta para o usuário, e então rdi e rsp são restaurados.
9. Finalmente, swapgs é desfeita, buffers são limpos, e sysretq é executada, retornando o controle ao programa do usuário.