Estrutura Interna syscalls

entry_64.S

Quando um syscall é chamada pelo usuário é necessário que a CPU realize algumas funções, essas funções são descritas no entry.s, ou seja, ele é responsavel pelo oque o hardware deve fazer ao uma syscall ser feita.

Por que salvar essas informações?

Quando o processador executa a instrução syscall, ele troca do modo usuário para o modo kernel, mas não salva automaticamente todo o estado do usuário na pilha do kernel. Para que o kernel possa executar a syscall e depois retornar ao usuário corretamente, ele precisa salvar o contexto do usuário — isto é, os registradores importantes e informações de segmento — em uma estrutura chamada pt_regs.

```
SYM_CODE_START(entry_SYSCALL_64)
    # Metadados para depuração, como um mapa para quem precisar
entender a pilha de chamadas.
    UNWIND_HINT_ENTRY

    # Ponto de verificação para a tecnologia de segurança CET,
garantindo que só se pode entrar aqui por um caminho legítimo.
    ENDBR


    # Troca o acesso a dados do usuário pelos dados do kernel (per-
CPU).
    swapgs


    # Salva o ponteiro da pilha do usuário temporariamente. O TSS é
um local "neutro" para isso.
    movq    %rsp, PER_CPU_VAR(cpu_tss_rw + TSS_sp2)
    # Troca o mapa de memória do usuário pelo mapa de memória
seguro do kernel. Essencial para mitigar o Meltdown.
```

```
    SWITCH_TO_KERNEL_CR3 scratch_reg=%rsp
    # Agora sim, aponta o RSP para a pilha do kernel desta CPU.
Estamos prontos para trabalhar.
    movq    PER_CPU_VAR(cpu_current_top_of_stack), %rsp

SYM_INNER_LABEL(entry_SYSCALL_64_safe_stack, SYM_L_GLOBAL)
    ANNOTATE_NOENDBR



    # Começa a montar a estrutura pt_regs na pilha do kernel,
salvando o estado do usuário.
    pushq   $__USER_DS              /* pt_regs->ss */
    pushq   PER_CPU_VAR(cpu_tss_rw + TSS_sp2)   /* pt_regs->sp */
    # RFLAGS e RIP foram salvos pelo hardware nos registradores
%r11 e %rcx. Agora os salvamos na pilha.
    pushq   %r11                    /* pt_regs->flags */
    pushq   $__USER_CS              /* pt_regs->cs */
    pushq   %rcx                    /* pt_regs->ip */

SYM_INNER_LABEL(entry_SYSCALL_64_after_hwframe, SYM_L_GLOBAL)
    # Salva o número da syscall (que estava em %rax).
    pushq   %rax                    /* pt_regs->orig_ax */
    # Salva todos os outros registradores de propósito geral.
    PUSH_AND_CLEAR_REGS rax=$-ENOSYS



    # Passa um ponteiro para a estrutura pt_regs (%rdi) e o número
da syscall (%rsi) para a função C.
    movq    %rsp, %rdi
    movslq  %eax, %rsi

    # Ativa mitigações contra ataques de execução especulativa
(Spectre v2, BHI).
    IBRS_ENTER
    UNTRAIN_RET
    CLEAR_BRANCH_HISTORY

    # Com tudo pronto e seguro, o controle é passado para a função
C que orquestra a syscall.
    call    do_syscall_64
```

A instrução syscall no processador x86-64 tem um comportamento especial:

```
Ela salva o endereço de retorno (RIP) do usuário em %rcx.

Ela salva o valor dos flags (RFLAGS) do usuário em %r11.
```

Isso é uma convenção do hardware para permitir que o kernel recupere esses valores ao retornar para o usuário via sysret. Portanto, %r11 não é um registrador qualquer; ele é usado para armazenar temporariamente os flags do usuário durante a syscall

```
O kernel constrói um struct pt_regs na pilha do kernel empurrando
os valores dos segmentos, ponteiro de pilha, flags e endereço de
instrução do usuário.

%rcx e %r11 são usados pelo hardware para salvar RIP e RFLAGS do
usuário durante a instrução syscall.

Essa estrutura é essencial para que o kernel possa restaurar o
contexto do usuário corretamente ao finalizar a syscall.
```

Esse mecanismo garante a integridade da troca de contexto entre usuário e kernel, permitindo que o kernel execute chamadas de sistema de forma segura e transparente para o programa usuário.

UNWIND_HINT_ENTRY

Em cada chamada de função, o endereço de retorno e, às vezes, o valor do ponteiro de quadro (frame pointer) são empilhados. O stack unwinding consiste em percorrer esses dados para reconstruir a sequência de chamadas. No passado, o kernel e aplicações dependiam do registrador de frame pointer (RBP), mas hoje, para aumentar o desempenho e reduzir o uso de registradores, o kernel Linux utiliza formatos como o ORC (Omnidirectional Resource Counter), que armazena metadados sobre o layout da pilha em uma seção especial do executável, permitindo desenrolar a pilha sem depender do frame pointer

Aplicação no kernel:
O marcador UNWIND_HINT_ENTRY no seu exemplo serve para instruir o sistema de desenrolamento de pilha (stack unwinder) sobre o ponto de entrada da função, facilitando a análise em caso de falha ou depuração

CET

CET é uma tecnologia de segurança implementada em hardware (principalmente em CPUs Intel) para prevenir ataques de desvio de fluxo de controle, como ROP (Return-Oriented Programming), JOP (Jump-Oriented Programming) e COP (Call-Oriented Programming)

Principais componentes:

```
Shadow Stack:
Uma pilha secundária, protegida e invisível para o programa, que armazena cópias dos endereços de retorno. Quando uma função retorna, o processador compara o endereço de retorno da pilha principal com o da shadow stack. Se forem diferentes, gera uma exceção, impedindo ataques que tentam manipular a pilha para redirecionar o fluxo de execução

Indirect Branch Tracking (IBT):
Usa instruções especiais (como ENDBR) para marcar destinos válidos de saltos indiretos. Antes de realizar um salto indireto, o processador verifica se o destino possui o marcador correto. Se não tiver, o salto é bloqueado, prevenindo ataques que tentam desviar o fluxo para código malicioso
```

Aplicação no kernel:
O uso de ENDBR no código é um exemplo de IBT, onde o kernel marca pontos de entrada válidos para saltos indiretos, aumentando a proteção contra exploração de vulnerabilidades

swapgs

O que o SWAPGS faz?

```
Troca do GS base:
O SWAPGS troca o valor do registrador base do segmento GS (um valor "escondido" usado para calcular endereços quando se usa o prefixo gs:) com o valor armazenado no MSR chamado IA32_KERNEL_GS_BASE (endereço MSR C0000102h)
```

Objetivo:
Isso permite que o kernel, ao entrar via syscall, tenha acesso imediato a uma área de dados específica para a CPU atual, sem precisar salvar registradores ou acessar memória antes de configurar a pilha do kernel

Uso típico:
O kernel usa o prefixo gs: para acessar variáveis per-CPU, como o ponteiro da pilha do kernel, identificador da tarefa atual, etc. Isso é feito logo após a entrada no kernel, antes de configurar a pilha do kernel

Por que é necessário?

```
syscall não configura a pilha do kernel:
Ao contrário de interrupções, a instrução syscall não configura automaticamente a pilha do kernel. O kernel precisa de um meio rápido para acessar dados locais da CPU para saber qual pilha usar.

Sem registro livre:
No momento da entrada no kernel, todos os registradores estão ocupados com valores do espaço do usuário. O SWAPGS permite acessar dados per-CPU sem destruir nenhum registrador
```

TSS

O Task State Segment (TSS) é uma estrutura de dados específica da arquitetura x86 e x86-64, utilizada pelo sistema operacional para armazenar informações essenciais sobre uma tarefa (task) e facilitar a troca de contexto, principalmente durante interrupções, exceções e mudanças de nível de privilégio

Troca de CR3

O que é o CR3?

> O CR3 é um registrador especial usado pela CPU para apontar para a Page Global Directory (PGD), que é a raiz das tabelas de páginas que traduzem endereços virtuais em físicos.
>
> Cada processo (ou contexto) tem sua própria tabela de páginas, que define o mapeamento da memória virtual para física.
>
> Em sistemas modernos, o kernel Linux implementa Page Table Isolation (PTI) para mitigar vulnerabilidades como Meltdown, mantendo duas tabelas de páginas separadas: uma para o espaço do usuário e outra para o espaço do kernel.

Por que trocar o CR3?

> Quando ocorre uma transição do espaço usuário para o kernel (por exemplo, numa syscall), o kernel precisa usar sua própria tabela de páginas para garantir que o acesso à memória seja seguro e que áreas do kernel não fiquem expostas ao usuário.
>
> Para isso, o kernel troca o valor do CR3 para apontar para sua tabela de páginas.
>
> Ao retornar para o usuário, o CR3 é trocado novamente para a tabela de páginas do processo usuário.

IBRS, UNTRAIN_RET, CLEAR_BRANCH_HISTORY

IBRS significa Indirect Branch Restricted Speculation.
É uma mitigação de hardware para a Spectre v2, vulnerabilidade que permite a um atacante explorar a execução especulativa para vazar dados entre processos ou entre usuário e kernel.

Como funciona:

```
Ativa restrições na execução especulativa:
Quando ativado, o IBRS impede que a CPU siga ramificações indiretas
(indirect branches) de forma especulativa após uma troca de
contexto, evitando que um ataque explore o histórico de branches de
outro processo ou modo de privilégio.

Implementação:
No código, IBRS_ENTER é uma macro que executa as instruções
necessárias para ativar o IBRS no processador (por exemplo, usando
a instrução wrmsr para escrever em um MSR específico).
```

UNTRAIN_RET é uma mitigação para ataques de Branch History Injection (BHI).

Como funciona:

```
BHI:
É uma variante de Spectre v2 que explora o histórico de branches
(branch history) para influenciar a execução especulativa.

UNTRAIN_RET:
A macro executa uma sequência de instruções que "limpa" o histórico
de branches do preditor de saltos da CPU, tornando mais difícil
para um atacante influenciar a execução especulativa.

Detalhe técnico:
Normalmente envolve a execução de uma sequência de retornos (ret)
controlados para “confundir” o preditor de saltos.
```

CLEAR_BRANCH_HISTORY é outra mitigação para ataques de especulação, especialmente Spectre v2 e variantes.

Como funciona:

```
Limpa o histórico de branches:
Essa macro executa instruções que forçam a CPU a limpar o histórico
de branches do preditor de saltos (Branch Prediction Buffer).

Objetivo:
Impedir que um ataque explore informações residuais no preditor de
saltos para vazar dados entre contextos de execução.

Detalhe técnico:
Pode envolver a execução de uma sequência de saltos controlados ou
o uso de instruções específicas do processador para limpar o buffer
de predição.
```

do_syscall_64

Despachar: Descobrir qual função do kernel corresponde ao número da syscall e executá-la.

Decidir o Retorno: Após a execução, determinar qual é a maneira mais segura e eficiente de voltar ao modo de usuário: o caminho rápido `SYSRET` ou o caminho completo `IRET`."

```
// retorna um booleano que significa "pode usar SYSRET?"
__visible noinstr bool do_syscall_64(struct pt_regs *regs, int nr)
{
    // 1. Hooks de entrada: notifica subsistemas como seccomp e
ftrace.
    nr = syscall_enter_from_user_mode(regs, nr);

    // 2. O Despacho: Tenta encontrar e executar a syscall nas
tabelas x64 ou x32.
    if (!do_syscall_x64(regs, nr) && !do_syscall_x32(regs, nr) &&
nr != -1) {
        // Se o número não corresponde a nenhuma syscall, executa a
"não implementada".
        regs->ax = __x64_sys_ni_syscall(regs);
    }

    // 3. Hooks de saída: notifica que a syscall terminou.
```

```
    syscall_exit_to_user_mode(regs);

    /*
     * 4. A Decisão Crítica: Uma série de verificações de segurança
para
     * decidir se o retorno rápido via SYSRET é seguro.
     */

    // Em ambientes virtualizados Xen, o IRET é obrigatório.
    if (cpu_feature_enabled(X86_FEATURE_XENPV))
        return false; // -> Use IRET

    // O hardware exige que RCX e R11 contenham RIP e RFLAGS para o
SYSRET.
    if (unlikely(regs->cx != regs->ip || regs->r11 != regs->flags))
        return false; // -> Use IRET

    // Os segmentos de código e pilha devem ser os padrões do
usuário.
    if (unlikely(regs->cs != __USER_CS || regs->ss != __USER_DS))
        return false; // -> Use IRET

    // O endereço de retorno deve estar no espaço de usuário
canônico para evitar falhas no kernel.
    if (unlikely(regs->ip >= TASK_SIZE_MAX))
        return false; // -> Use IRET

    // SYSRET não lida bem com os flags RF e TF, essenciais para
depuração.
    if (unlikely(regs->flags & (X86_EFLAGS_RF | X86_EFLAGS_TF)))
        return false; // -> Use IRET

    // Se todas as verificações passaram, o caminho rápido é
autorizado.
    return true; // -> Pode usar SYSRET!
}
```

A assinatura da função mostra que ela recebe os **registradores do usuário salvos** (`regs`) e o **número da syscall** (`nr`) vindo do registrador `%rax`. Ela retorna um booleano que instrui o código Assembly chamador a usar o caminho de retorno rápido (`true` para `SYSRET`) ou o lento (`false` para `IRET`).

`add_random_kstack_offset()`: Medida de segurança (Kernel Address Space Layout Randomization) que adiciona um pequeno deslocamento aleatório à pilha do kernel para dificultar ataques baseados em corrupção de pilha.

`syscall_enter_from_user_mode()`: Este é o primeiro "gancho" (hook) principal. Ele notifica subsistemas como **ftrace** (para rastreamento) e **seccomp** (para filtragem de segurança) que uma syscall está prestes a ser executada. O Seccomp pode decidir bloquear a syscall, e nesse caso a função retorna um novo valor para `nr` (geralmente -1), efetivamente cancelando a chamada antes mesmo que ela comece.

`do_syscall_x64(regs, nr)`: Tenta executar a syscall usando a tabela para a ABI x86-64 padrão.

`do_syscall_x32(regs, nr)`: Se a primeira falhar, tenta executar usando a tabela para a ABI x32 (usada para compatibilidade).

`if (!... && !... && nr != -1)`: A condição do `if` só é verdadeira se **ambas** as tentativas de despacho falharem (ou seja, o número `nr` é inválido) **e** a chamada não foi previamente cancelada pelo seccomp (`nr != -1`).

`regs->ax = __x64_sys_ni_syscall(regs);`: Se a syscall é inválida, esta função é chamada. `ni_syscall` significa "not implemented" (não implementada) e simplesmente retorna o código de erro `-ENOSYS`. O resultado é colocado em `regs->ax`, que é o campo que corresponde ao registrador `%rax` para o retorno ao usuário.

O restante do código é uma série de **verificações de segurança e sanidade** para decidir se o caminho rápido `SYSRET` pode ser usado. `SYSRET` é mais rápido que `IRET`, mas possui pré-requisitos de hardware muito estritos.

1. **`if (cpu_feature_enabled(X86_FEATURE_XENPV))`**: Em ambientes de paravirtualização Xen, o retorno deve sempre usar `IRET`. Retorna `false`.

2. **`if (unlikely(regs->cx != regs->ip || regs->r11 != regs->flags))`**: `SYSRET` espera que o endereço de retorno esteja em `RCX` e os flags em `R11`. A instrução `syscall` coloca os valores corretos lá. Se eles foram modificados, é inseguro usar `SYSRET`. Retorna `false`.
3. **`if (unlikely(regs->cs != __USER_CS || regs->ss != __USER_DS))`**: `SYSRET` também exige que os seletores de segmento de código (`CS`) e de pilha (`SS`) correspondam a valores fixos definidos pelo kernel no boot. Retorna `false` se estiverem incorretos.
4. **`if (unlikely(regs->ip >= TASK_SIZE_MAX))`**: Verificação de segurança crucial. Em algumas CPUs Intel, executar `SYSRET` com um endereço de retorno "não canônico" (fora do espaço de usuário válido) causa uma falha no modo kernel, o que é uma vulnerabilidade grave. Retorna `false` se o endereço de retorno for inválido.
5. **`if (unlikely(regs->flags & (X86_EFLAGS_RF | X86_EFLAGS_TF)))`**: `SYSRET` não consegue restaurar o `Resume Flag (RF)` e lida com o `Trap Flag (TF)` de forma diferente de `IRET`. Para garantir o comportamento correto de depuradores, `IRET` deve ser usado nesses casos. Retorna `false`.
6. **`return true;`**: Se todas as verificações passarem, a função retorna `true`, autorizando o código Assembly a usar a instrução `SYSRET` para um retorno rápido e eficiente ao modo de usuário.

O `do_syscall_64` trata todas as syscalls da mesma forma. Ele simplesmente pega o ponteiro para a `pt_regs` (que contém *todos* os possíveis argumentos e o estado do usuário) e o passa para a função wrapper correspondente. A chamada é sempre no formato: `wrapper_da_syscall(struct pt_regs *regs)`.

Retorno após chamada da syscall

restaurar o estado do usuário a partir de pt_regs e retornar ao modo de usuário da forma mais rápida e segura possível.

caminho rápido (sysretq) e um caminho mais lento e genérico (iret), além de lidar com a possibilidade de uma troca de contexto.

```
syscall_return_via_sysret:
```

```
    IBRS_EXIT
    POP_REGS pop_rdi=0 // Macro varias instruções popq, restaurando regs prop geral(%rbx, %rcx...) a partir da pilha pt_regs, NÂO restaura o %rdi(reg temp)

    movq    %rsp, %rdi // salva ponteiro da pilha do kernel atual, precisa do endereço para restaurar regs restantes
    movq    PER_CPU_VAR(cpu_tss_rw + TSS_sp0), %rsp // troca pilha atual pela trampoline
    UNWIND_HINT_END_OF_STACK // indica q pilha mudou

    pushq   RSP-RDI(%rdi)// calcula endereço original de rsp dentro de pt_regs(rdi + offsetrsp)
    pushq   (%rdi)      // push valor de rdi ao usuário

    STACKLEAK_ERASE_NOCLOBBER // Apaga rastros da pilha do kernel para evitar vazamento de dados.

    SWITCH_TO_USER_CR3_STACK scratch_reg=%rdi // restaura CR3, contem endereço base das tabelas de paginas do processo de usuário

    popq    %rdi // restaura RDI
    popq    %rsp // aponta para pilha do usuário
SYM_INNER_LABEL(entry_SYSRETQ_unsafe_stack, SYM_L_GLOBAL)
    ANNOTATE_NOENDBR
    swapgs // restaura GS apontando para data structs do espaço de usuário
    CLEAR_CPU_BUFFERS // Limpa buffers internos da CPU (mitigação MDS).
    sysretq // transição atomica de ring0 para ring3 restaurando %rip a partir de %rcx, RFLAGS a partir de %r11, forma mais rapida de retornar para o modo de usuário
```

`syscall_return_via_sysret:`: Um rótulo para ferramentas de profiling como o `perf`, facilitando a análise de desempenho.

`IBRS_EXIT`: **O que faz:** Desativa a mitigação de segurança `Indirect Branch Restricted Speculation` (Spectre v2). **Por que faz:** A proteção foi ativada na entrada (`IBRS_ENTER`) para proteger o kernel. Agora que estamos saindo, ela pode ser desativada para não penalizar o desempenho do código de usuário.

`POP_REGS pop_rdi=0`: **O que faz:** Esta é uma macro que executa uma série de instruções `popq` para restaurar os registradores de propósito geral (`%rbx`, `%rcx`, `%rdx`, `%rsi`, `%rbp`, etc.) a partir da pilha (`pt_regs`). **Por que faz:** Está desfazendo a "fotografia" que tiramos na entrada, restaurando o estado exato dos registradores do programa de usuário. O parâmetro `pop_rdi=0` instrui a macro a **NÃO** restaurar o `%rdi` ainda, pois ele será usado como um registrador temporário.

`movq %rsp, %rdi`: **O que faz:** Salva o ponteiro da pilha do kernel atual (que aponta para a `pt_regs`) em `%rdi`. **Por que faz:** Precisamos do endereço da `pt_regs` para restaurar os últimos registradores, mas vamos trocar de pilha no próximo passo. `%rdi` agora guarda esse endereço vital.

`movq PER_CPU_VAR(cpu_tss_rw + TSS_sp0), %rsp`: **O que faz:** Esta é uma etapa crucial. Ele troca a pilha atual pela **pilha de trampolim** (`TSS_sp0`). **Por que faz:** Por segurança e robustez. A pilha do kernel da tarefa pode conter dados sensíveis. A transição final de volta ao usuário é feita em uma pequena pilha separada, por-CPU, conhecida como "trampoline stack". Isso garante um ambiente limpo e controlado para os últimos passos, mitigando vazamentos de informação (como o `stackleak`).

`UNWIND_HINT_END_OF_STACK`: Metadados para depuradores e `stack unwinders`, indicando que a pilha mudou.

O que faz: Estamos agora na pilha de trampolim. O código empilha os dois últimos valores que precisamos restaurar:

1. `pushq RSP-RDI(%rdi)`: Calcula o endereço original de `%rsp` do usuário (que está salvo dentro da `pt_regs` em `(%rdi) + offset_do_rsp`) e o empilha.
2. `pushq (%rdi)`: Empilha o valor de `%rdi` do usuário (que também estava salvo na `pt_regs`).

Por que faz: Prepara os valores finais para serem restaurados com `popq` após sairmos da pilha de trampolim.

`STACKLEAK_ERASE_NOCLOBBER`: **O que faz:** Apaga qualquer vestígio da pilha do kernel que foi usada durante a syscall. **Por que faz:** Mitigação de segurança para evitar que um programa de usuário possa ler dados deixados na pilha pelo kernel.

`SWITCH_TO_USER_CR3_STACK`: **O que faz:** Restaura o registrador `CR3`, que contém o endereço base das tabelas de página do processo de usuário. **Por que faz:** Na entrada, o kernel trocou para seu próprio mapa de memória (`SWITCH_TO_KERNEL_CR3`). Agora, ele restaura o mapa de memória do usuário, para que o processo enxergue seu próprio espaço de endereçamento virtual novamente.

`swapgs`: **O que faz:** Desfaz o `swapgs` da entrada. Restaura o registrador `GS` para apontar para as estruturas de dados do espaço de usuário (como o Thread-Local Storage).

`CLEAR_CPU_BUFFERS`: Mitigação para `Microarchitectural Data Sampling` (MDS), um tipo de vulnerabilidade de execução especulativa. Limpa buffers internos da CPU antes de devolver o controle.

`sysretq`: **O que faz:** A instrução mágica final. Ela realiza a transição atômica de ring 0 para ring 3, restaurando o ponteiro de instrução (`%rip`) a partir de `%rcx` e os flags de execução (`RFLAGS`) a partir de `%r11`. **Por que faz:** É a forma mais rápida de retornar para o modo de usuário 64-bit, completando a syscall.

SYSCALL_TABLE

```
0   common  read            sys_read
1   common  write           sys_write
2   common  open            sys_open
3   common  close           sys_close
4   common  stat            sys_newstat
5   common  fstat           sys_newfstat
6   common  lstat           sys_newlstat
7   common  poll            sys_poll
8   common  lseek           sys_lseek
9   common  mmap            sys_mmap
10  common  mprotect        sys_mprotect
11  common  munmap          sys_munmap
12  common  brk             sys_brk
```

uma tabela de definição que os scripts de build do kernel usam para gerar automaticamente o código de despacho de syscalls, os cabeçalhos para o espaço de usuário e os stubs de função.

Ele funciona como um mapa de despacho central, conectando o número abstrato de uma syscall à sua implementação concreta no kernel.

`<number>`
É um inteiro único e não negativo que identifica a chamada de sistema. Este é o "ID" universal da syscall naquela arquitetura.
Este é o número que uma aplicação em espaço de usuário (geralmente através da glibc) coloca no registrador %rax antes de executar a instrução syscall. Dentro do kernel, após a transição de modo de usuário para modo de kernel, o dispatcher de syscalls (do_syscall_64) lê este número (a partir da estrutura pt_regs salva na pilha) para saber qual função executar. Ele serve como o índice para a sys_call_table.

`<abi>`
Especifica para qual(is) ABI(s) a chamada de sistema está disponível. A ABI define convenções de chamada, tamanhos de tipo de dados e alinhamento de estruturas.

common: A syscall pode ser usada tanto por aplicações 64-bit padrão (modelo de dados LP64) quanto por aplicações x32 (um ABI que usa ponteiros de 32 bits em modo 64-bit, ILP32). A função do kernel subjacente precisa ser capaz de lidar com argumentos de ambos (read, write)

64: A syscall está disponível apenas para aplicações 64-bit padrão. Não é gerado um ponto de entrada para a ABI x32. Isso é usado quando a syscall depende intrinsecamente de tipos de 64 bits (como long ou ponteiros), exp rt_sigreturn

x32: A syscall é específica para a ABI x32 e não está disponível para aplicações 64-bit padrão.

`<name>`
É o nome canônico e legível por humanos da chamada de sistema.

O script de build cria a macro **NR_ no arquivo de cabeçalho <asm/unistd_64.h>. Por exemplo, a linha 0 ... read ... resulta na definição #define** NR_read 0. É esta macro que a glibc e outras ferramentas usam para obter o número da syscall de forma programática, em vez de codificá-lo diretamente.

O nome é usado para criar o nome do protótipo da função wrapper, como __x64_sys_read.

`<entry point>`
Especifica o nome da função C dentro do código-fonte do kernel que contém a lógica de implementação da syscall.

Este não é apenas um nome, é a referência para o ponteiro de função que será colocado no array sys_call_table. Quando o dispatcher de syscalls usa o número de %rax como índice, o valor que ele obtém da sys_call_table é o endereço desta função.

O ponto de entrada listado (ex: sys_read) é a função de implementação final. O sistema de build geralmente gera uma pequena função "stub" ou "wrapper" (ex: __x64_sys_read). É este stub que é realmente colocado na tabela. O stub tem a assinatura correta para ser chamado pelo dispatcher (recebendo const struct pt_regs *regs) e é responsável por extrair os argumentos dos registradores (salvos em pt_regs) e passá-los para a função sys_read com a convenção de chamada C padrão.

Como kernel utiliza tabela

Para consolidar, aqui está o fluxo completo:

1. **Espaço de Usuário:** A `glibc`, ao executar um `read()`, coloca `0` (`__NR_read`) em `%rax`, o file descriptor em `%rdi`, o ponteiro do buffer em `%rsi`, e a contagem em `%rdx`. Em seguida, executa a instrução `syscall`.
2. **Entrada no Kernel:** A CPU entra em modo kernel e salta para `entry_SYSCALL_64`. Este código assembly salva todos os registradores do usuário na estrutura `pt_regs` na pilha do kernel.
3. **Despacho:** O código assembly chama a função C `do_syscall_64(struct pt_regs *regs)`.
4. **Consulta à Tabela:** `do_syscall_64` pega o número da syscall de `regs->orig_ax` (que é 0).
5. **A Tabela em Ação:** O kernel usa esse número `0` como um índice para o array `sys_call_table`. `const void *entry = sys_call_table[0];`
6. **Execução:** O `entry` conterá o endereço do stub `__x64_sys_read`. O kernel chama esta função.

7. **Execução do Stub:** O stub `__x64_sys_read` extrai os argumentos originais (`%rdi`, `%rsi`, `%rdx`) da estrutura `regs` e finalmente chama a função de implementação real: `sys_read(regs->di, regs->si, regs->dx)`.

Dessa forma, a tabela `syscall_64.tbl` atua como a cola que une o mundo abstrato dos números de syscall com as funções C que fazem o trabalho pesado dentro do kernel.

sys_ni_syscall

`sys_ni_syscall` (de "**Not Implemented**") é uma função do kernel usada como *placeholder* para syscalls que:

- Ainda **não foram implementadas**, ou
- Estão **desativadas** para uma certa arquitetura/configuração.

Quando uma syscall não tem uma função correspondente no kernel, o kernel mapeia o número dela para `sys_ni_syscall`, que simplesmente retorna o erro `-ENOSYS` (Error: No such syscall).

se syscall 456 não existe e usuário chama, a função do_syscall_64 verifica a syscall_table[] e encontra

```
[456] = sys_ni_syscall
```

Syscalls.h

Este header define um conjunto de macros (SYSCALL_DEFINE0, SYSCALL_DEFINE1, etc.) que os desenvolvedores do kernel usam para escrever o código de uma chamada de sistema.

Metadados para Tracing: Arrays de strings com os tipos e nomes dos argumentos, para que ferramentas como o `ftrace` possam exibir `sys_read(fd=3, buf=0x..., count=128)`.

Wrappers de Segurança: Cria uma função (`__se_sys_read`) que garante que os argumentos passados em registradores de 64 bits sejam tratados corretamente, evitando vulnerabilidades.

A Função Real: Cria a função final `static inline` (`__do_sys_read`) onde o código escrito pelo desenvolvedor é inserido.

O Símbolo Público: Cria um alias `sys_read` que aponta para o wrapper de segurança e é colocado na `sys_call_table`.

```
#define __MAP0(m,...)
#define __MAP1(m,t,a,...) m(t,a)
#define __MAP2(m,t,a,...) m(t,a), __MAP1(m,__VA_ARGS__)
// ... até __MAP6
#define __MAP(n,...) __MAP##n(__VA_ARGS__)
```

metaprogramação do arquivo. `__MAP` é uma macro que "aplica" uma outra macro (`m`) a uma lista de pares de argumentos. Essencialmente, é um "loop" do pré-processador.

Exemplo: `__MAP(2, __SC_DECL, int, fd, char *, buf)` se expande para:

1. `__MAP2(__SC_DECL, int, fd, char *, buf)`
2. `__SC_DECL(int, fd), __MAP1(__SC_DECL, char *, buf)`
3. `__SC_DECL(int, fd), __SC_DECL(char *, buf)`
4. Resultado final: `int fd, char * buf`

Por que é necessário: Permite que as macros `SYSCALL_DEFINE` manipulem um número variável de argumentos de forma genérica para gerar listas de parâmetros, declarações de strings, etc.

```
#define SYSCALL_DEFINE1(name, ...) SYSCALL_DEFINEx(1, _##name,
__VA_ARGS__)
// ... até SYSCALL_DEFINE6

#define SYSCALL_DEFINEx(x, sname, ...)              \
    SYSCALL_METADATA(sname, x, __VA_ARGS__)         \
    __SYSCALL_DEFINEx(x, sname, __VA_ARGS__)
```

O que faz: Esta é a API que os desenvolvedores do kernel usam. Para definir uma syscall, em vez de escrever uma função C, você usa a macro. O número no nome (`SYSCALL_DEFINE**1**`) indica a quantidade de argumentos.

Exemplo de uso: `SYSCALL_DEFINE3(open, const char __user *, filename, int, flags, umode_t, mode) { ... }`

Como funciona: `SYSCALL_DEFINE3` é simplesmente um atalho para `SYSCALL_DEFINEx`, que faz duas coisas principais:

1. `SYSCALL_METADATA`: Chama a macro que gera os metadados para o `ftrace`.
2. `__SYSCALL_DEFINEx`: Chama a macro que gera o código da função C real.

```
#define SYSCALL_METADATA(sname, nb, ...)            \
    static const char *types_##sname[] = {          \
        __MAP(nb,__SC_STR_TDECL,__VA_ARGS__)        \
    };                              \
    static const char *args_##sname[] = {           \
        __MAP(nb,__SC_STR_ADECL,__VA_ARGS__)        \
    };                              \
    /* ... preenche uma struct syscall_metadata ... */
```

O que faz: Quando `CONFIG_FTRACE_SYSCALLS` está habilitado, esta macro usa o `__MAP` para criar arrays de strings com os nomes dos tipos e dos argumentos da syscall.

Por que é necessário: Ela popula uma `struct syscall_metadata` que o `ftrace` usa para saber como imprimir os argumentos da syscall de forma legível, sem que o desenvolvedor precise escrever nenhum código de formatação.

```
#define __SYSCALL_DEFINEx(x, name, ...)                 \
    asmlinkage long sys##name(...)                      \
        __attribute__((alias(__stringify(__se_sys##name)))); \
    /* ... */                                           \
    asmlinkage long __se_sys##name(...)                 \
    {                                                   \
        long ret = __do_sys##name(...);                 \
        /* ... */                                       \
        return ret;                                     \
    }                                                   \
    /* ... */                                           \
    static inline long __do_sys##name(...)
```

1. `sys<name>` (ex: `sys_open`):
    - Esta é a função que é exportada e colocada na `sys_call_table`.
    - No entanto, ela não contém código. Ela é apenas um **alias** (usando `__attribute__((alias(...)))`) para a função `__se_sys<name>`. É uma forma de ter um nome de símbolo público e limpo (`sys_open`) apontando para a implementação real com um nome interno.
2. `__se_sys<name>` (ex: `__se_sys_open`):
    - "se" significa "Sign-Extend" (extensão de sinal).
    - Esta é a função wrapper de segurança. Seus argumentos são definidos como `long` (usando a macro `__SC_LONG`).
    - Ela chama a função `__do_sys<name>`, fazendo um cast (`__SC_CAST`) dos argumentos de `long` de volta para seus tipos originais.
    - **Por que é necessário:** Em sistemas de 64 bits, os argumentos da syscall são passados em registradores de 64 bits. Se um usuário mal-intencionado passar um valor de 32 bits com lixo nos 32 bits superiores, isso poderia causar problemas. Esta função garante que todos os argumentos sejam tratados como `long` de 64 bits (com os bits superiores limpos ou com o sinal estendido corretamente), sanitizando a entrada antes de passar para a lógica principal.
3. `__do_sys<name>` (ex: `__do_sys_open`):
    - Esta é a função `static inline` onde o **desenvolvedor do kernel escreve a lógica real da syscall**.

- O corpo da função que o programador escreve após a macro `SYSCALL_DEFINE3(...)` é, na verdade, o corpo desta função `__do_sys<name>`.

Em resumo, quando um desenvolvedor escreve `SYSCALL_DEFINE3(open, ...){ /* código */ }`, o pré-processador C transforma isso em um conjunto completo de funções e metadados que definem, protegem e permitem o rastreamento da syscall `sys_open`.

```
asmlinkage long sys_futex_wake(void __user *uaddr, unsigned long
mask, int nr, unsigned int flags);
```

`asmlinkage`

É uma diretiva para o compilador GCC que instrui a função a buscar seus argumentos diretamente da **pilha (stack)**, e não dos registradores da CPU. Essa é a convenção de chamada padrão para syscalls no kernel, garantindo a portabilidade entre diferentes arquiteturas.

`long`

É o **tipo de retorno** padrão para a maioria das syscalls.

- Se a chamada for bem-sucedida, o valor de retorno é geralmente não-negativo (por exemplo, o número de threads que foram acordadas).
- Se ocorrer um erro, a função retorna um valor negativo que corresponde a um código de erro (ex: `-EINVAL`, `-EFAULT`).

`sys_futex_wake`

Este é o nome da função no kernel que implementa a parte "wake" (acordar) da syscall `futex`.