



UNIVERSIDAD
SANTO TOMÁS

Análisis de Interrupciones en Linux y Seguridad de Bootkits con Simulación Robótica en PyBullet

Jaime Mendez

Universidad Santo Tomás de Aquino

Bogotá, Colombia

jaimemendezj@usantotomas.edu.co

I. INTRODUCCIÓN

Este informe presenta un análisis técnico de dos artículos fundamentales sobre la gestión de interrupciones en sistemas Linux y las amenazas de seguridad mediante bootkits. Además, se desarrolla una simulación sencilla de un brazo robótico usando la biblioteca PyBullet, detallando su implementación en consola y su despliegue en un contenedor Docker. La práctica integra conceptos de bajo nivel del sistema operativo, robótica y virtualización.

II. CONTENIDOS

A. Resumen Extendido del Artículo y *Respuestas a las Preguntas ("Linux Interrupts: The Basic Concepts")*:

B.

C:"BOOTKITS: PAST, PRESENT & FUTURE"

D:Simulación de brazo robótico

III. DESARROLLO

A. Resumen Extendido del Artículo:

El documento "Linux Interrupts: The Basic Concepts" se enfoca en desmitificar la implementación del manejo de interrupciones y excepciones dentro del kernel de Linux, específicamente en la versión 2.4.18-10. Inicia estableciendo una jerarquía clara de los estados en los que puede encontrarse una CPU: sirviendo una interrupción hardware (máxima

prioridad), sirviendo mecanismos de trabajo diferido como softirqs, tasklets o bottom halves, o ejecutando un proceso en modo usuario (mínima prioridad). Se enfatiza que la preemption (interrupción de una tarea por otra de mayor prioridad) sigue estrictamente este orden; por ejemplo, una interrupción hardware puede interrumpir a un softirq en ejecución, pero otro softirq no puede hacerlo en la misma CPU. El texto diferencia conceptualmente entre interrupciones (eventos asíncronos, generalmente de dispositivos de E/S, no ligados a la instrucción actual) y excepciones (eventos síncronos, generados por la CPU al detectar condiciones específicas durante la ejecución de una instrucción). Las interrupciones se subclasifican en emmascarables (pueden ser ignoradas temporalmente) y no emmascarables (NMI) (no pueden ser ignoradas). Las excepciones se clasifican según su causa y comportamiento: faults (faltas, como Page Fault, permiten reejecutar la instrucción problemática), traps (trampas, usadas en depuración, como breakpoints, el flujo continúa tras la instrucción), aborts (abortos, indican errores graves de hardware o sistema, usualmente terminan el proceso) y programmed exceptions (excepciones programadas, generadas por instrucciones como int, int3, into, bound). Se detalla el uso de vectores (números 0-255) en la arquitectura Intel para identificar cada interrupción o excepción. Linux reserva los vectores 0-31 para excepciones y NMIs, 32-47 para interrupciones emmascarables (IRQs provenientes del hardware), y específicamente el vector 128 (0x80) para las llamadas al sistema (una forma de interrupción por software). El manejo de interrupciones hardware (IRQs) se describe conectando dispositivos a un Controlador de Interrupciones

(como el clásico PIC 8259A, que maneja hasta 8 IRQs y puede conectarse en cascada para manejar más, típicamente 15). Se menciona la limitación de líneas IRQ y la necesidad de compartir IRQs, lo que requiere que los manejadores identifiquen qué dispositivo específico generó la señal en la línea compartida. En sistemas más modernos con APIC, las interrupciones pueden llegar por pines o por el bus APIC. Las interrupciones enmascarables pueden deshabilitarse globalmente con el flag IF del registro EFLAGS. Las interrupciones por software, generadas por la instrucción INT n, no son afectadas por el flag IF. La Tabla de Descriptores de Interrupción (IDT) es fundamental, asociando cada vector con un descriptor de puerta que apunta al manejador correspondiente. Linux utiliza principalmente Interrupt Gates (para interrupciones hardware, DPL=0, inaccesibles desde modo usuario) y Trap Gates (para la mayoría de excepciones, DPL=0). Se usan System Gates (un tipo de Trap Gate con DPL=3) para excepciones específicas que deben ser accesibles desde modo usuario, como breakpoint (int3), overflow (into), bounds check (bound) y system call (int 0x80). La IDT se localiza mediante el registro idtr. Se describe minuciosamente el proceso hardware al recibir una interrupción/excepción: determinar vector, leer IDT (usando idtr), leer GDT (usando gdtr) para validar segmento y permisos, comparar CPL vs DPL, cambiar de pila si hay cambio de nivel de privilegio (usando el TSS), guardar estado (EFLAGS, CS, EIP, y código de error si lo hay) en la pila activa, y finalmente cargar CS y EIP del descriptor de puerta para saltar al manejador. El retorno se realiza con la instrucción iret, que restaura EFLAGS, CS, EIP y, si hubo cambio de privilegio, SS y ESP. El concepto de Kernel Control Path (ruta de control del kernel) se define como la secuencia de instrucciones ejecutadas en modo kernel para atender un evento. Linux permite el anidamiento de manejadores de interrupción pero no permite la replanificación (scheduling) durante ellos. Se explica la inicialización cuidadosa de la IDT para prevenir accesos no autorizados desde modo usuario. El manejo de excepciones sigue una estructura: salvar registros, llamar a una función C, y retornar vía ret_from_exception. La función C recibe información contextual y usualmente envía una señal al proceso causante (Tabla 1 lista excepciones, manejadores y señales asociadas). Funciones como die_if_kernel manejan errores fatales ocurridos en modo kernel.

El manejo de interrupciones hardware sigue 5 pasos básicos: salvar IRQ/registros, enviar ACK al PIC, ejecutar ISRs asociados a la IRQ, ejecutar softirqs pendientes, y retornar vía ret_from_intr. Se detallan las estructuras clave: irq_desc_t (indexado por número de IRQ, contiene estado, puntero al hw_irq_controller y lista de irqaction), hw_interrupt_type (define la interfaz de bajo nivel con el controlador de interrupciones específico, como el i8259A), y irqaction (representa un manejador registrado para una IRQ,

con puntero al ISR, flags como SA_SHIRQ para compartición, y puntero next para encadenar manejadores). Se muestra cómo se construyen los puntos de entrada a los manejadores (BUILD_IRQ, BUILD_COMMON_IRQ) y se guarda el contexto (SAVE_ALL). La función central do_IRQ orquesta el proceso: adquiere lock, envía ACK, actualiza estado, llama a handle_IRQ_event (que itera sobre la lista irqaction y llama a los ISRs), llama a end, y finalmente ejecuta do_softirq si hay trabajo diferido pendiente. Se introducen los mecanismos de trabajo diferido para evitar que los manejadores de interrupción (top half) hagan tareas largas: Bottom Halves (BHs) (el más antiguo, 32 estáticos, no paralelizables), Softirqs (reemplazo de BHs, 32 estáticos, pueden correr en paralelo en múltiples CPUs, requieren sincronización explícita) y Tasklets (construidos sobre softirqs, registro dinámico, garantizan ejecución en una sola CPU a la vez, simplificando la concurrencia). Se menciona la gestión dinámica de líneas IRQ con request_irq y free_irq. Finalmente, se aborda el contexto SMP (Symmetrical Multiprocessing), que requiere distribución de interrupciones y comunicación inter-CPU. Esto se logra con el APIC (Advanced Programmable Interrupt Controller), que incluye un I/O APIC (reemplaza al PIC, con tabla de redirección para dirigir IRQs a CPUs específicas en modo fijo o de mínima prioridad) y Local APICs en cada CPU (con registros, temporizador, líneas locales y capacidad de enviar/recibir mensajes). Las Inter-Processor Interrupts (IPIs), enviadas a través del bus APIC (ICC bus), son usadas para mensajes entre CPUs, con vectores específicos para tareas como solicitar replanificación (RESCHEDULE_VECTOR), invalidar TLB (INVALIDATE_TLB_VECTOR), manejar temporizadores locales (LOCAL_TIMER_VECTOR), y ejecutar funciones en otras CPUs (CALL_FUNCTION_VECTOR). El documento concluye resaltando la eficiencia de los mecanismos de Linux para el manejo de interrupciones y excepciones.

1. Respuestas a las Preguntas ("Linux Interrupts: The Basic Concepts"):

¿Qué es una interrupción?

Según el documento, una interrupción es fundamentalmente un evento asincrónico. Esto significa que no ocurre en sincronía con el flujo de ejecución de las instrucciones del procesador. Su origen típico es un dispositivo de Entrada/Salida (E/S). Por ejemplo, cuando un disco duro termina de leer datos o una tarjeta de red recibe un paquete, generan una interrupción para notificar a la CPU que requieren atención.

¿Por qué son necesarias las interrupciones?

El texto no dedica una sección a justificar su necesidad frente a alternativas como el polling. Sin embargo, su descripción exhaustiva del mecanismo y su integración en el kernel subraya su rol indispensable. Permiten que la CPU realice otras tareas mientras los dispositivos de E/S operan independientemente, y solo sea interrumpida cuando un evento externo (llegada de datos, finalización de operación, error de hardware) realmente necesita ser procesado. Esto es crucial para la eficiencia y la capacidad de respuesta de los sistemas operativos modernos, permitiendo la multitarea y el manejo de eventos en tiempo real o casi real. La existencia de manejadores específicos, prioridades implícitas y mecanismos de trabajo diferido evidencia un diseño orientado a manejar eficientemente estos eventos asíncronos.

¿Qué tipos de interrupciones existen?

El documento clasifica los eventos que interrumpen el flujo normal en dos grandes categorías: Interrupciones y Excepciones.

Interrupciones (Interrupts):

Maskable Interrupts (Enmascarables): Son la mayoría de las interrupciones generadas por hardware (IRQs). Pueden ser ignoradas temporalmente por la CPU si el bit IF (Interrupt Flag) en el registro EFLAGS está desactivado. Esto es útil para proteger secciones críticas de código.

Non-Maskable Interrupts (No Enmascarables - NMI):

Interrupciones de alta prioridad (generalmente para errores graves de hardware) que la CPU no puede ignorar, independientemente del estado del flag IF. Linux les asigna vectores en el rango 0-31.

Software Generated Interrupts (Generadas por Software):

Son iniciadas explícitamente por el código mediante la instrucción INT n. No pueden ser enmascaradas por el flag IF. Un uso clave en Linux es la implementación de llamadas al sistema a través de int 0x80 (vector 128).

Excepciones (Exceptions):

Son eventos síncronos generados por la propia CPU al encontrar una condición anómala durante la ejecución de una instrucción. Se subdividen:

Faults (Faltas): Detectadas antes de completar la instrucción causante (ej. Page Fault - vector 14). El eip guardado apunta a la instrucción fallida, permitiendo reintentarla después de corregir la condición (ej. cargar la página faltante).

Traps (Trampas): Detectadas después de ejecutar la instrucción causante (ej. Breakpoint - vector 3). El eip guardado apunta a la siguiente instrucción. Se usan principalmente para depuración.

Aborts (Abortos): Indican errores severos (ej. Double Fault - vector 8), de los que no siempre es posible recuperarse o determinar la instrucción causante. El manejador usualmente termina el proceso afectado.

Programmed Exceptions (Programadas): Son el resultado directo de ejecutar ciertas instrucciones como int (genérica), int3 (breakpoint), into (overflow check) o bound (bounds check).

¿Cuáles son sus características?

Origen y Sincronía: Las interrupciones (hardware) son asíncronas, originadas externamente a la CPU (E/S). Las excepciones son síncronas, originadas internamente por la CPU durante la ejecución. Las interrupciones por software son síncronas respecto a la instrucción INT n.

Enmascaramiento: Las interrupciones hardware pueden ser enmascarables (controladas por el flag IF) o no enmascarables. Las interrupciones por software (INT n) y las excepciones no son enmascarables por el flag IF.

Identificación (Vectores): Cada tipo se asocia a un número único (vector 0-255) que la CPU usa para encontrar el manejador adecuado en la IDT.

Manejo (Handlers/ISRs): Cada interrupción/excepción tiene una rutina específica (manejador o ISR) asociada a su vector a través de un descriptor de puerta en la IDT.

Prioridad y Anidamiento: Linux implementa una jerarquía de prioridad: Interrupciones Hardware > Trabajo Diferido (softirq, tasklet, BH) > Procesos User-Mode. Las interrupciones hardware pueden interrumpir a manejadores de menor prioridad e incluso a otros manejadores de interrupción (anidamiento), aunque Linux intenta minimizar el tiempo con las interrupciones deshabilitadas.

Contexto de Ejecución: Los manejadores se ejecutan en modo kernel. Si la interrupción ocurre en modo usuario, hay un cambio de contexto y de pila.

Compartición (IRQ Sharing): Debido al número limitado de líneas físicas IRQ, Linux permite que múltiples dispositivos compartan la misma línea, requiriendo que los ISRs verifiquen si su dispositivo fue el que generó la interrupción. El flag SA_SHIRQ en irqaction indica esto.

¿Qué son los "exception handler"?

Son las rutinas de software específicas diseñadas y registradas en la IDT para ser ejecutadas por el sistema operativo cuando la CPU detecta una condición de excepción. Cada uno de los aproximadamente 20 tipos de excepciones definidos por Intel tiene su propio manejador en Linux. Su propósito es tratar la condición anómala que causó la excepción. Dependiendo del tipo:

Para Faults (como Page Fault), intentan resolver la causa (ej. cargar la página de memoria faltante) para que la instrucción pueda reejecutarse. Para Traps (como Breakpoint), realizan acciones de depuración y luego continúan la ejecución. Para Aborts (como Double Fault), dado que indican errores graves, suelen terminar el proceso afectado.

La estructura típica de un manejador de excepción en Linux involucra: Código ensamblador inicial que guarda los registros del procesador en la pila del kernel. Llamada a una función C que realiza la lógica principal del manejo. Esta función recibe contexto, como el código de error hardware si existe. La función C a menudo determina qué señal UNIX enviar al proceso que causó la excepción (si ocurrió en modo usuario), almacenando información del error en el descriptor del proceso (tsk->thread.error_code, tsk->thread.trap_no) y usando force_sig(). Ver Tabla 1 para ejemplos de señales.

Retorno de la función C al código ensamblador, que salta a la rutina común de salida ret_from_exception.

En la salida, se verifica si la excepción ocurrió en modo kernel. Si es así y no hay un mecanismo de "fixup" (recuperación conocida), funciones como die_if_kernel o die() pueden imprimir información de depuración (registros) y terminar el proceso actual (do_exit()).

¿Qué son las interrupciones generadas por software?, ¿Para que sirven? y ¿Qué algoritmos/diagramas de flujo son usados para el manejo de interrupciones?

Qué son: Son interrupciones que se originan no por un evento hardware externo, sino por la ejecución explícita de la instrucción INT n dentro de un programa. La n especifica el vector de interrupción a invocar (0-255). A diferencia de las interrupciones hardware enmascarables, estas no pueden ser bloqueadas usando el flag IF del registro EFLAGS.

Para qué sirven: El uso primordial documentado en el texto para Linux es la implementación de llamadas al sistema (system calls). Cuando un proceso en modo usuario necesita solicitar un servicio privilegiado del kernel (como operaciones de archivo, red, gestión de procesos), ejecuta la instrucción int 0x80 (vector 128). Esto causa una interrupción controlada que transfiere la ejecución a la rutina del kernel system_call(), cambiando la CPU a modo kernel para realizar la operación solicitada de forma segura. Aunque no se detalla en el texto, otras interrupciones por software como int 3 (vector 3) se usan comúnmente para breakpoints en depuración.

Algoritmos/Diagramas de Flujo (Manejo General de Interrupciones Hardware): El documento describe el flujo detallado tanto a nivel hardware como software.

Fase Hardware (Iniciada por la CPU/Controlador de Interrupciones):

Detectar Interrupción/Excepción.

Determinar Vector i.

Leer Descriptor de Puerta i desde IDT (localizada por idtr).

Validar Segmento y Permisos usando GDT (localizada por gdtr). Comparar DPL de la puerta con CPL actual; si DPL < CPL (para interrupciones), emitir General Protection Fault (#GP).

Si hay cambio de Nivel de Privilegio (ej. User a Kernel): Obtener nueva SS:ESP del TSS (Task State Segment, localizado por tr), guardar antigua SS:ESP en la nueva pila.

Guardar EFLAGS, CS, EIP en la pila activa. (Si es Fault, EIP apunta a la instrucción causante; si es Trap/Interrupt, a la siguiente).

Si la excepción genera Código de Error Hardware, guardarlo en la pila.

Cargar CS:EIP desde el Descriptor de Puerta (dirección del manejador).

Fase Software (Manejador de Interrupción en Linux - ej. para IRQs):

Punto de Entrada Específico (IRQn_interrupt): Guarda nº IRQ relativo, salta a common_interrupt.

common_interrupt: Llama a SAVE_ALL (guarda registros generales, ajusta segmentos DS, ES).

Llama a do_IRQ(struct pt_regs *regs):

Obtiene irq_desc_t correspondiente al nº IRQ.

Adquiere spinlock_t del descriptor.

Llama a desc->handler->ack(irq) (Ej. envía EOI al PIC).

Actualiza desc->status (ej. pone IRQ_INPROGRESS).

Libera spinlock_t.

Llama a handle_IRQ_event(irq, regs, desc->action):

Itera sobre la lista irqaction (si IRQ compartida).

Llama a cada action->handler (ISR) registrado.

Adquiere spinlock_t.

Llama a desc->handler->end(irq) (Ej. re-habilita IRQ en PIC si es level-triggered o maneja estados post-ISR).

Actualiza desc->status.

Libera spinlock_t.

Llama a do_softirq() si softirq_pending(cpu) es verdadero.

Salta a ret_from_intr.

Fase de Retorno (Común para Interrupts/Exceptions/Syscalls):

Punto de entrada ret_from_intr, ret_from_exception, o ret_from_sys_call.

Obtener current (puntero al task_struct actual).

Verificar si el contexto interrumpido era User Mode o VM86 mode (basado en CS y EFLAGS guardados en pila). Si sí, ir a chequeos adicionales; si no (era Kernel Mode y no anidado), ir directo a restore_all.

(Si era User/VM86) Verificar need_resched flag en current: si es 1, llamar a schedule() y reintentar.

(Si era User/VM86) Verificar sigpending flag en current: si es 1, llamar a do_signal() para manejar señales pendientes. (Manejo especial para VM86 con save_v86_state).

restore_all: Llama a RESTORE_ALL (restaura registros generales desde la pila).

Ejecuta iret (restaura EIP, CS, EFLAGS, y SS:ESP si hubo cambio de privilegio).

¿Qué son las IRQ y las estructuras de datos en relación con las interrupciones?

IRQ (Interrupt ReQuest): Son las líneas o señales físicas utilizadas por los dispositivos hardware para solicitar la atención de la CPU. Indican que un dispositivo necesita servicio. Estas señales llegan a un Controlador de Interrupciones (como el PIC 8259A o el I/O APIC en sistemas SMP), que gestiona estas peticiones, posiblemente las prioriza, y luego notifica a la CPU. El número de líneas IRQ físicas es limitado (ej. 15 en una configuración típica con dos PICs 8259A en cascada), lo que lleva a la necesidad de compartir IRQs entre múltiples dispositivos. Cuando una IRQ compartida se activa, el software (los ISRs registrados) debe determinar cuál de los dispositivos en esa línea necesita atención.

Estructuras de Datos Clave (en Linux Kernel 2.4): El kernel necesita organizar la información sobre cómo manejar cada IRQ. Las estructuras principales descritas son:

irq_desc_t: Es la estructura central. Existe un array global irq_desc de tamaño NR_IRQS (número total de IRQs posibles), donde cada elemento es de tipo irq_desc_t. Cada irq_desc_t representa una línea IRQ y contiene:

status: Un campo de bits que indica el estado actual de la IRQ (ej. IRQ_DISABLED, IRQ_INPROGRESS, IRQ_PENDING, IRQ_SHARED, etc.).

handler: Puntero a una estructura hw_interrupt_type (o hw_irq_controller) que define las funciones de bajo nivel para interactuar con el controlador de hardware específico (PIC, APIC) que gestiona esa IRQ.

action: Puntero a la cabeza de una lista enlazada de estructuras irqaction. Si es NULL, no hay manejador registrado. Si la IRQ es compartida, esta lista contendrá múltiples nodos irqaction.

depth: Contador para gestionar habilitaciones/deshabilitaciones anidadas de la IRQ.

lock: Un spinlock_t para proteger el acceso concurrente a este descriptor en sistemas SMP.

hw_interrupt_type (alias hw_irq_controller): Define una interfaz abstracta para un tipo de controlador de interrupciones. Contiene punteros a funciones específicas del hardware para operaciones como:

startup: Inicializar la IRQ al principio.

shutdown: Desactivar la IRQ.

enable: Habilitar la línea IRQ en el controlador.

disable: Deshabilitar la línea IRQ en el controlador.

ack: Enviar reconocimiento (Acknowledge) al controlador.

end: Tareas post-manejo (ej. re-habilitar tras ACK para IRQs level-triggered).

set_affinity: (Para APIC/SMP) Especificar a qué CPU(s) debe dirigirse la IRQ. El documento muestra un ejemplo (i8259A_irq_type) con funciones para el PIC 8259A.

irqaction: Representa un manejador (ISR) registrado para una IRQ específica. Contiene:

handler: Puntero a la función ISR que implementa la lógica de manejo del dispositivo. La firma es void (*handler)(int irq, void *dev_id, struct pt_regs *regs).

flags: Banderas que modifican el comportamiento. Incluyen SA_SHIRQ (indica que este handler es para una IRQ compartida), SA_INTERRUPT (el sistema deshabilita las interrupciones locales mientras se ejecuta este handler), SA_SAMPLE_RANDOM (usa el timing de esta interrupción como fuente de entropía).

mask: (No descrito en detalle su uso en el texto).

name: Cadena de texto que identifica al dispositivo o driver (útil en /proc/interrupts).

dev_id: Un puntero void que actúa como identificador único para el dispositivo, crucial para free_irq y para distinguir dispositivos en IRQs compartidas. Debe ser único (a menudo, es un puntero a la estructura de datos del propio dispositivo).

next: Puntero al siguiente irqaction en la lista, permitiendo encadenar múltiples manejadores para una IRQ compartida.

B: "BOOTKITS: PAST, PRESENT & FUTURE"

A. Resumen Extendido del Artículo:

El artículo "BOOTKITS: PAST, PRESENT & FUTURE" ofrece una visión detallada de la amenaza que representan los bootkits, catalogándolos como un arma potente para los cibercriminales debido a su capacidad para establecer una presencia persistente y sigilosa en los sistemas comprometidos. El resurgimiento reciente de estos ataques se vinculó a la necesidad de infectar versiones de Windows de 64 bits, que implementaron restricciones en la carga de controladores de kernel no firmados (Kernel-Mode Code Signing policy). Sin embargo, el artículo plantea la cuestión de la seguridad de las plataformas más nuevas basadas en UEFI, ya que las técnicas de bootkit tradicionales no son efectivas contra ellas, y explora si UEFI es inmune o simplemente un nuevo objetivo. El objetivo principal es trazar la evolución de los bootkits y anticipar las amenazas futuras. Primero, se revisan los bootkits conocidos que atacan la plataforma Windows BIOS/MBR, desde los pioneros como Mebroot hasta ejemplos más complejos como TDL4 (también conocido como Olmarik), Rovnix (usado por el troyano bancario Carberp), y Gapz (destacado por sus técnicas de infección VBR extremadamente sigilosas). Se examinan sus métodos de infección (modificación de MBR, VBR, tabla de particiones) y las técnicas empleadas para evadir la detección y resistir la eliminación. Se destaca que la idea de comprometer el sector de arranque no es nueva, remontándose a los virus de sector de arranque de la era MS-DOS como Brain (1987), que ya mostraba características precursoras: uso de área de almacenamiento oculta (sectores marcados como 'bad') y técnicas de sigilo (stealth) al interceptar interrupciones de disco. Incluso antes, virus como Elk Cloner (1982-3) para Apple II ya modificaban el sector de arranque del sistema operativo en disquetes. Se clasifican los bootkits modernos basados en BIOS según el tipo de infección del sector de arranque: los que atacan el MBR (Master Boot Record) y los que atacan el VBR (Volume Boot Record) o IPL (Initial Program Loader). Los bootkits MBR pueden modificar el código del MBR (como TDL4) o la tabla de particiones del

MBR (como Olmasco). Los bootkits VBR, considerados más sofisticados, modifican el VBR o el código IPL que este carga. Se describe el flujo de arranque de TDL4 y sus componentes en el sistema de archivos oculto. Se detalla la técnica de infección de Rovnix, que sobreescribe los sectores IPL después del VBR con su propio código, engancha la interrupción Int 13h para parchear ntldr/bootmgr, y utiliza la IDT (Interrupt Descriptor Table) y los registros de depuración (dr0-dr7) para transferir control entre modos de procesador (real a protegido) y cargar su driver malicioso en modo kernel, evitando la verificación de firmas. Se analiza Gapz, cuya variante VBR es notablemente sigilosa al modificar únicamente el campo "Hidden Sectors" en el Bloque de Parámetros del Volumen (VPB) dentro del VBR, redirigiendo así la carga del IPL hacia el código del bootkit almacenado en otra parte del disco. Gapz también implementa un almacenamiento oculto en un archivo dentro de System Volume Information, formateado como FAT32 y cifrado sector por sector con AES-256 en modo CBC, usando el número de sector como IV para variar el cifrado. Además, posee un stack de red TCP/IP personalizado en modo kernel que interactúa directamente con el driver miniport NDIS, permitiéndole comunicarse con servidores C&C (sobre HTTP cifrado) evadiendo firewalls personales y software de monitoreo de red. En segundo lugar, el artículo se adentra en la seguridad de la plataforma UEFI (Unified Extensible Firmware Interface), que reemplaza al BIOS y modifica sustancialmente el proceso de arranque (sin MBR/VBR, usa tabla de particiones GPT, carga bootloader desde una partición EFI dedicada FAT32 especificada en NVRAM). Se analiza Dreamboot, el primer PoC público de bootkit UEFI para Windows 8. Dreamboot reemplaza el bootloader original (bootmgfw.efi), carga el original, y hookeea rutinas clave (Archpx64TransferTo64BitApplicationAsm en el bootloader y OslArchTransferToKernel en winload.efi) para obtener control antes de la ejecución del kernel y parchearlo, desactivando así mecanismos de seguridad como PatchGuard. Se identifican tres vectores de ataque principales contra UEFI: 1) Reemplazar el gestor de arranque del SO (como Dreamboot); 2) Abusar directamente de drivers UEFI DXE (Driver Execution Environment); 3) Parchear Option ROMs de dispositivos (ej. tarjetas de red, almacenamiento) que contienen drivers DXE. Se discute la tecnología Secure Boot de UEFI como medida de protección, la cual verifica firmas criptográficas de los componentes de arranque. Sin embargo, se señala que los investigadores están explorando activamente vulnerabilidades en las implementaciones de UEFI/BIOS que podrían permitir eludir Secure Boot. Para el análisis y la defensa, se presenta el framework CHIPSEC, una herramienta open-source desarrollada por Intel para evaluar la seguridad de plataformas PC a bajo nivel (hardware, firmware BIOS/UEFI, configuración de componentes). CHIPSEC puede ejecutarse en Windows, Linux y desde el shell UEFI, permitiendo pruebas de seguridad, análisis forense de firmware (incluso offline) y detección heurística básica de bootkits. También se

menciona la herramienta ESET Hidden File System Reader, diseñada para extraer y analizar el contenido de los sistemas de archivos ocultos usados por varios bootkits complejos como TDL3, TDL4, Olmasco, Rovnix, etc., facilitando la respuesta a incidentes y el análisis de malware. Finalmente, el artículo concluye que, aunque Secure Boot representa un avance, no es el fin de la era de los bootkits, sino un cambio en las estrategias de ataque. Los cibercriminales continuarán usando técnicas MBR/VBR contra la gran base de sistemas heredados sin UEFI/Secure Boot para construir botnets. En cambio, para ataques dirigidos contra sistemas modernos con Windows 8/UEFI, los atacantes explotarán vulnerabilidades en el firmware UEFI/BIOS. Esto es particularmente problemático debido al lento y fragmentado ciclo de vida de las actualizaciones de firmware, a diferencia del software o los sistemas operativos; muchos usuarios nunca actualizan su firmware. La protección a nivel de firmware por parte del software de seguridad tradicional es aún incipiente. Los autores predicen un futuro interesante, posiblemente comenzando con ataques dirigidos basados en UEFI, que podrían ya estar ocurriendo.

B. Abstracción Profunda de los Principales Puntos Analizados:

Definición y Amenaza Persistente: Los bootkits son una categoría de malware (rootkits que operan en la fase de arranque) cuyo valor estratégico reside en su capacidad para lograr persistencia extrema (resisten reinstalaciones de SO si infectan firmware o sectores de arranque) y sigilo superior (operan antes que las defensas del SO y el software de seguridad). Son herramientas para obtener control profundo y duradero.

Impulso Evolutivo - Evasión de Defensas: Un motor clave para la evolución de los bootkits modernos (post-Mebroot) fue la introducción de la política de firma obligatoria de controladores de kernel (KMCS) en Windows x64. Al no poder cargar drivers maliciosos directamente, los atacantes recurrieron a infectar el proceso de arranque (MBR/VBR) para cargar su código en modo kernel antes de que la política de firmas entrara en vigor o para parchear el kernel y desactivarla.

Diversificación de Técnicas (Pre-UEFI): Los atacantes desarrollaron múltiples métodos para comprometer el arranque MBR/BIOS:

Infección MBR: Simple sobreescritura del código o manipulación de la tabla de particiones para redirigir el flujo de arranque.

Infección VBR/IPL: Considerada más avanzada. Implica modificar el VBR o los sectores IPL que carga. Técnicas sofisticadas como las de Rovnix (usar la IDT para persistir entre modos real/protegido, usar registros de depuración como breakpoints hardware para ganar ejecución en puntos clave de la inicialización del kernel) y Gapz (modificar solo metadatos - "Hidden Sectors" - para máxima discreción, implementar almacenamiento oculto robusto con cifrado AES, y crear un stack de red propio en kernel para evadir firewalls) demuestran una alta complejidad técnica.

El Desafío y la Oportunidad de UEFI: La adopción de UEFI cambió las reglas del juego. Al eliminar MBR/VBR y introducir mecanismos como Secure Boot, hizo obsoletos los bootkits tradicionales. Sin embargo, UEFI mismo se convirtió en un nuevo y atractivo objetivo. Los vectores de ataque UEFI (reemplazo de bootloader, infección de drivers DXE, parcheo de Option ROMs) ofrecen nuevas vías para lograr el mismo objetivo de compromiso temprano. Dreamboot demostró la viabilidad de estos ataques.

Secure Boot: Defensa Imperfecta: Aunque Secure Boot está diseñado para verificar la integridad del arranque, no es una panacea. Las vulnerabilidades en las implementaciones específicas del firmware UEFI por parte de los fabricantes pueden permitir su elusión. La complejidad de UEFI abre nuevas superficies de ataque.

El Problema del Firmware: El firmware (BIOS/UEFI) tiene un ciclo de vida de seguridad deficiente. Las actualizaciones son poco frecuentes, no están estandarizadas ni automatizadas como en los SO, y muchos usuarios nunca las aplican. Esto crea una ventana de vulnerabilidad persistente que los atacantes pueden explotar, especialmente en ataques dirigidos.

Análisis Forense y Detección: La naturaleza de bajo nivel y el sigilo de los bootkits complican enormemente el análisis forense y la detección. Se requieren herramientas especializadas como CHIPSEC (para auditoría de firmware y hardware) y herramientas de recuperación de datos ocultos (como ESET Hidden File System Reader) para investigar y combatir estas amenazas.

Perspectiva Futura: Se anticipa una dicotomía en las amenazas: persistencia de bootkits "clásicos" (MBR/VBR) contra sistemas heredados para campañas masivas, y un aumento de ataques sofisticados dirigidos contra vulnerabilidades UEFI en sistemas modernos. La seguridad del firmware se perfila como un campo de batalla crítico.

C: Simulación de brazo robótico

Descripción:

Se diseñó un robot con dos articulaciones utilizando el archivo URDF personalizado two_joint_robot_custom.urdf.

Se utilizó PyBullet para renderizar y controlar el robot.

Se emplearon sliders para mover los motores de las articulaciones en tiempo real.

Código base (main.py):

Inicializa el servidor de física.

Carga el modelo URDF.

Usa sliders de GUI para controlar el robot.

Monitorea eventos hasta que el usuario cierre la interfaz.

A. Explicación del código

main.py

1. Importación de librerías

```
python
import pybullet as p
import pybullet_data
import time
import numpy as np
import os
```

Estas librerías sirven para:

pybullet: manejar la simulación física.

pybullet_data: acceder a archivos como el plano (plane.urdf).

time: para hacer pausas en la simulación.

numpy: se usa para manejar valores como pi.

os: para verificar la existencia del archivo URDF.

2. Comprobación del archivo URDF

```
python

robot_urdf_path = "two_joint_robot_custom.urdf"
if not os.path.exists(robot_urdf_path):
    print(f"ERROR: No se encuentra el archivo {robot_urdf_path}")
    ...
```

Aquí se verifica si el archivo two_joint_robot_custom.urdf (que describe el robot) existe en el mismo directorio que el script. Si no, se muestra un error y se detiene el programa.

3. Inicialización de PyBullet

```
python

p.connect(p.GUI)
p.configureDebugVisualizer(p.COV_ENABLE_GUI, 1)
```

Se inicia PyBullet en modo gráfico (GUI), lo que abre una ventana para visualizar la simulación.

4. Configuración del entorno

```
python

p.setGravity(0, 0, -9.8)
p.setAdditionalSearchPath(pybullet_data.getDataPath())
p.resetDebugVisualizerCamera(...)
```

Se aplica gravedad hacia abajo.

Se añade el path donde está el plano (plane.urdf).

Se posiciona la cámara para una mejor vista.

5. Carga del plano y del robot

```
python

planeId = p.loadURDF("plane.urdf")
robotId = p.loadURDF(robot_urdf_path, [0, 0, 0.1], useFixedBase=True)
```

Se carga un plano como suelo.

Se carga el robot definido en el archivo URDF, fijo en su base.

6. Deslizadores de control

```
python

joint1_slider = p.addUserDebugParameter("Joint 1", -np.pi, np.pi, 0)
joint2_slider = p.addUserDebugParameter("Joint 2", -np.pi, np.pi, 0)
speed_control = p.addUserDebugParameter("Velocidad", 0.1, 10.0, 1.0)
pause_button = p.addUserDebugParameter("Pausar/Continuar", 1, 0, 1)
```

Crea interfaces de usuario en PyBullet:

Dos sliders para controlar las posiciones de las articulaciones del robot.

Un slider para la velocidad de la simulación.

Un botón para pausar o continuar.

7. Bucle principal de la simulación

```
python

while True:
    current_pause_value = p.readUserDebugParameter(pause_button)
    ...
```

Se ejecuta continuamente hasta que el usuario interrumpe (Ctrl+C).

Lee los valores de los sliders para actualizar las articulaciones.

Controla las articulaciones con p.setJointMotorControl2.

Usa p.stepSimulation() para avanzar un paso en la simulación.

La velocidad se regula con time.sleep(0.01 / speed).

8. Cierre seguro

```
python

except KeyboardInterrupt:
    ...
finally:
    input("\nSimulación finalizada. Presiona Enter para cerrar completamente...")
    p.disconnect()
```

Si el usuario presiona Ctrl+C, se interrumpe con un mensaje amigable.

Luego espera a que el usuario presione Enter para cerrar la ventana y desconectarse de PyBullet.

two_joint_robot_custom.urdf

1. Declaración del robot

```
xml

<robot name="two_joint_robot">
```

Se define el nombre del robot. Todo lo que sigue dentro de esta etiqueta forma parte del modelo de ese robot.

2. Base del robot: *base_link*

```
xml

<link name="base_link">
```

Esta es la base del robot.

Visual: cómo se ve visualmente la base (un cilindro azul).

Collision: geometría usada para detectar colisiones (mismo cilindro).

Inertial: masa y propiedades inerciales que afectan la dinámica del movimiento.

```

xml

<geometry><cylinder length="0.1" radius="0.2"/></geometry>
<color rgba="0 0 0.8 1"/>
<mass value="1"/>
<inertia ixx="0.1" iyy="0.1" izz="0.1"/>
```

3. Primer eslabón: link1

```

xml

<link name="link1">
```

Es un bloque rectangular (caja) rojo.
 Visual/Collision: caja de $0.1 \times 0.1 \times 0.5$ posicionada verticalmente (origin xyz="0 0 0.25").
 Inertial: centrado en medio del bloque.

```

xml

<box size="0.1 0.1 0.5"/>
<color rgba="0.8 0 0 1"/>
<mass value="1"/>
<origin xyz="0 0 0.25"/>
```

4. Primera articulación: joint1

```

xml

<joint name="joint1" type="revolute">
```

Esta junta conecta la base (base_link) con el primer eslabón (link1).
 Origen: justo sobre la base (xyz="0 0 0.05").
 Eje de rotación: en el eje Z (xyz="0 0 1").
 Tipo: revolute, es decir, permite rotación (como una bisagra).
 Límites: rotación de $-\pi$ a $+\pi$ radianes ($\sim -180^\circ$ a 180°).

5. Segundo eslabón: link2

```

xml

<link name="link2">
```

Otro bloque rectangular (verde), similar a link1, conectado en su parte superior.

Mismo tamaño ($0.1 \times 0.1 \times 0.5$), con origen también a la mitad del bloque.

6. Segunda articulación: joint2

```

xml

<joint name="joint2" type="revolute">
```

Conecta link1 con link2.
 Origen: final del primer eslabón (xyz="0 0 0.5").
 Eje de rotación: en el eje Y (xyz="0 1 0"), rotación perpendicular a la anterior.
 Permite una rotación que genera movimiento 3D combinado al girar ambas juntas.

7. Efector final: end_effector

```

xml

<link name="end_effector">
```

Es una esfera blanca al final del segundo eslabón.
 Radio: 0.05, masa baja (0.1), sirve de punto final o "mano" del robot.

8. Unión fija: joint_end_effector

```

xml

<joint name="joint_end_effector" type="fixed">
```

Une el final de link2 con la esfera end_effector.

Es fija, no tiene movimiento.

La esfera aparece justo encima del segundo eslabón (xyz="0 0 0.5").

Jerarquía del robot

```

text

base_link
  └── joint1 → link1
    └── joint2 → link2
      └── joint_end_effector → end_effector
```

B. Corrimiento en consola

¿Por qué usamos un entorno virtual (venv)?

```
¿Qué comandos usamos para configurarlo?
```

```
bash

# Crear el entorno virtual
python3 -m venv venv

# Activarlo (Linux)
source venv/bin/activate

# Instalar dependencias
pip install pybullet numpy
```

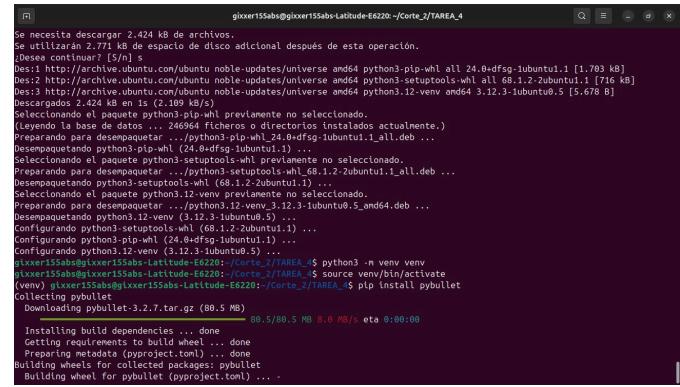
Durante el desarrollo de la simulación con PyBullet, se optó por utilizar un entorno virtual de Python (venv) por las siguientes razones:

Aislamiento de dependencias: Cada proyecto de Python puede requerir diferentes versiones de librerías como pybullet, numpy, o incluso Python mismo. Un entorno virtual asegura que las versiones instaladas para este proyecto no interfieran con otros proyectos ni con el sistema base.

Evitar conflictos con el sistema operativo: En sistemas Linux, instalar paquetes globalmente (con pip install) puede romper dependencias internas del sistema o sobreescribir librerías del sistema. Usar venv evita modificar el entorno de Python global del sistema operativo.

Portabilidad del proyecto: Al usar un venv y documentar las dependencias en un archivo como requirements.txt, es posible reproducir el entorno exactamente igual en otra máquina o en Docker, lo cual es ideal para despliegue y colaboración.

Buenas prácticas en desarrollo Python: El uso de entornos virtuales es una recomendación estándar de la comunidad de Python y del propio equipo de PyBullet para evitar errores como ModuleNotFoundError, versiones incompatibles o corrupción de paquetes.



```
gixer15abs@gixer15abs-Latitude-E6220:~/Corte_2/TAREA_4
Se necesita descargar 2.424 kB de archivos.
Se utilizarán 2.424 kB de espacio de disco adicional después de esta operación.
Descargando...
[ 0%] [ 5%] [ 10%] [ 15%] [ 20%] [ 25%] [ 30%] [ 35%] [ 40%] [ 45%] [ 50%] [ 55%] [ 60%] [ 65%] [ 70%] [ 75%] [ 80%] [ 85%] [ 90%] [ 95%] [ 100%]
Des1 http://archive.ubuntu.com/ubuntu noble-updates/universe amd64 python3-pip-whl all 24.0+dfsg-1ubuntu1.1 [1.793 kB]
Des2 http://archive.ubuntu.com/ubuntu noble-updates/universe amd64 python3-setuptools-whl all 68.1.2-2ubuntu1.1 [716 kB]
Des3 http://archive.ubuntu.com/ubuntu noble-updates/universe amd64 python3.12-venv amd64 3.12.3-1ubuntu0.5 [5.678 kB]
Descargado 2.424 kB en 1s (2.10 kB/s).
Seleccionando el paquete python3-pip-whl previamente no seleccionado.
(Leyendo la base de datos... 246964 Ficheros o directorios instalados actualmente.)
Preparando para desempaquetar .../python3-pip-whl_24.0+dfsg-1ubuntu1.1_all.deb ...
Desempaquetando python3-pip-tools-whl (68.1.2-2ubuntu1.1) ...
Seleccionando el paquete python3.12-venv (3.12.3-1ubuntu0.5) ...
Preparando para desempaquetar .../python3.12-venv_3.12.3-1ubuntu0.5_amd64.deb ...
Desempaquetando python3.12-venv (3.12.3-1ubuntu0.5) ...
Configurando python3-pip-tools-whl (68.1.2-2ubuntu1.1) ...
Configurando python3.12-venv (24.0+dfsg-1ubuntu1.1) ...
Configurando python3.12-venv (3.12.3-1ubuntu0.5) ...
Configurando python3.12-venv (3.12.3-1ubuntu0.5) ...
pixxer15abs@gixer15abs-Latitude-E6220:~/Corte_2/TAREA_4$ python3 -m venv venv
pixxer15abs@gixer15abs-Latitude-E6220:~/Corte_2/TAREA_4$ source venv/bin/activate
(venv) gixer15abs@gixer15abs-Latitude-E6220:~/Corte_2/TAREA_4$ pip install pybullet
Collecting pybullet
  Downloading pybullet-3.2.7.tar.gz (80.5 MB)
    Installing build dependencies ...
      Getting requirements to build wheel ...
        Preparing metadata (pyproject.toml) ...
          done
        Building wheels for collected packages: pybullet
          Building wheel for pybullet (pyproject.toml) ...
            done
```

Aquí estamos instalando venv.

Pasos:

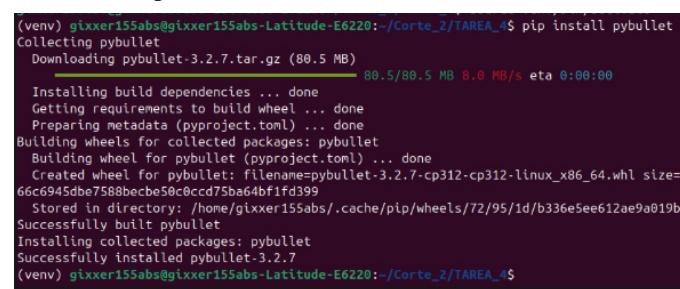
1.Crear entorno virtual:



```
gixer15abs@gixer15abs-Latitude-E6220:~/Corte_2/TAREA_4
Seleccionando el paquete python3-pip-whl previamente no seleccionado.
(Leyendo la base de datos... 246964 ficheros o directorios instalados actualmente.)
Preparando para desempaquetar .../python3-pip-whl_24.0+dfsg-1ubuntu1.1_all.deb ...
Desempaquetando python3-pip-whl (24.0+dfsg-1ubuntu1.1) ...
Seleccionando el paquete python3-setuptools-whl (68.1.2-2ubuntu1.1) ...
Preparando para desempaquetar .../python3-setuptools-whl_68.1.2-2ubuntu1.1_all.deb ...
Desempaquetando python3-setuptools-whl (68.1.2-2ubuntu1.1) ...
Seleccionando el paquete python3.12-venv previamente no seleccionado.
Preparando para desempaquetar .../python3.12-venv_3.12.3-1ubuntu0.5_amd64.deb ...
Desempaquetando python3.12-venv (3.12.3-1ubuntu0.5) ...
Configurando python3-setuptools-whl (68.1.2-2ubuntu1.1) ...
Configurando python3-pip-tools-whl (68.1.2-2ubuntu1.1) ...
Configurando python3.12-venv (3.12.3-1ubuntu0.5) ...
Configurando python3.12-venv (3.12.3-1ubuntu0.5) ...
pixxer15abs@gixer15abs-Latitude-E6220:~/Corte_2/TAREA_4$ python3 -m venv venv
pixxer15abs@gixer15abs-Latitude-E6220:~/Corte_2/TAREA_4$ source venv/bin/activate
```

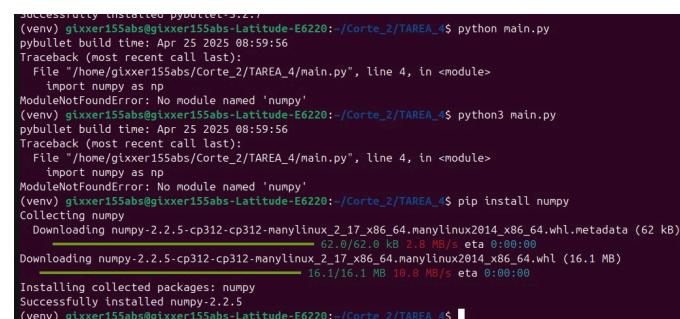
```
python3 -m venv venv
source venv/bin/activate
```

2.Instalar dependencias:



```
(venv) gixer15abs@gixer15abs-Latitude-E6220:~/Corte_2/TAREA_4$ pip install pybullet
Collecting pybullet
  Downloading pybullet-3.2.7.tar.gz (80.5 MB)
    Installing build dependencies ...
      Getting requirements to build wheel ...
        Preparing metadata (pyproject.toml) ...
          done
        Building wheels for collected packages: pybullet
          Building wheel for pybullet (pyproject.toml) ...
            Created wheel for pybullet: filename=pybullet-3.2.7-cp312-cp312-linux_x86_64.whl size=966c6945dbe7588beccbe58c0cccd75ba64bf1fd399
          Stored in directory: /home/gixer15abs/.cache/pip/wheels/72/95/1d/b336e5ee612ae9a019bf
        Successfully built pybullet
      Installing collected packages: pybullet
        Successfully installed pybullet-3.2.7
(venv) gixer15abs@gixer15abs-Latitude-E6220:~/Corte_2/TAREA_4$
```

pib install pybullet



```
Successfully installed pybullet-3.2.7
(venv) gixer15abs@gixer15abs-Latitude-E6220:~/Corte_2/TAREA_4$ python main.py
pybullet build time: Apr 25 2025 08:59:56
Traceback (most recent call last):
  File "/home/gixer15abs/Corte_2/TAREA_4/main.py", line 4, in <module>
    import numpy as np
ModuleNotFoundError: No module named 'numpy'
(venv) gixer15abs@gixer15abs-Latitude-E6220:~/Corte_2/TAREA_4$ python3 main.py
pybullet build time: Apr 25 2025 08:59:56
Traceback (most recent call last):
  File "/home/gixer15abs/Corte_2/TAREA_4/main.py", line 4, in <module>
    import numpy as np
ModuleNotFoundError: No module named 'numpy'
(venv) gixer15abs@gixer15abs-Latitude-E6220:~/Corte_2/TAREA_4$ pip install numpy
Collecting numpy
  Downloading numpy-2.2.5-cp312-cp312-manylinux_2_17_x86_64_manylinux2014.x86_64.whl.metadata (62 kB)
  Downloading numpy-2.2.5-cp312-cp312-manylinux_2_17_x86_64_manylinux2014.x86_64.whl (16.1 MB)
    62.0/62.0 kB 2.5 MB/s eta 0:00:08
  Downloading numpy-2.2.5-cp312-cp312-manylinux_2_17_x86_64_manylinux2014.x86_64.whl (16.1 MB)
    16.1/16.1 kB 10.5 kB/s eta 0:00:08
Installing collected packages: numpy
  Successfully installed numpy-2.2.5
(venv) gixer15abs@gixer15abs-Latitude-E6220:~/Corte_2/TAREA_4$
```

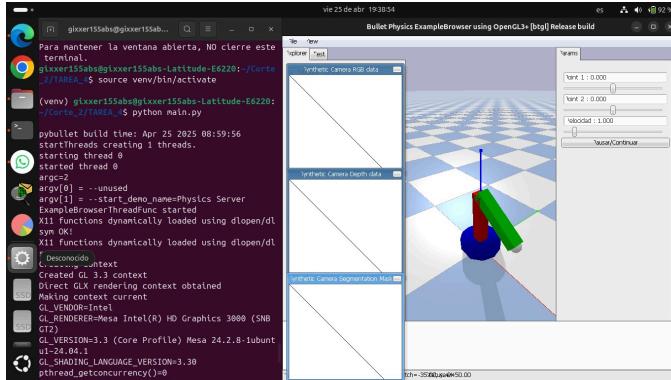
pip install numpy

3.Ejecutar:

```
(venv) gixer15abs@gixer15abs-Latitude-E6220:~/Corte_2/TAREA_4
(gvnc) gixer15abs@gixer15abs-Latitude-E6220:~/Corte_2/TAREA_4$ source venv/bin/activate
(venv) gixer15abs@gixer15abs-Latitude-E6220:~/Corte_2/TAREA_4$ source venv/bin/activate
pybullet build time: Apr 25 2025 08:59:36
startThreads creating 1 threads.
starting thread 0
started thread 0
argc=2
argv[0] = ...unused
argv[1] = ...start_demo_name=Physics Server
ExampleBrowserThreadFunc started
X11 Functions dynamically loaded using dlopen/dlSYM OK!
X11 Functions dynamically loaded using dlopen/dlSYM OK!
Creating context
Created GL 3.3 context
Direct GLX rendering context obtained
Making current
GL_RENDERER=Mesa Intel(R) HD Graphics 3000 (SNB GT2)
GL_VERSION=3.3 (Core Profile) Mesa 24.2.8-ubuntu1-24.04.1
GL_SHADING_LANGUAGE_VERSION=3.30
pthread_getconcurrency()=0
Version = 24.2.8 (Core Profile) Mesa 24.2.8-ubuntu1-24.04.1
Vendor = Intel
Renderer = Mesa Intel(R) HD Graphics 3000 (SNB GT2)
b3PInfo: Selected demo: Physics Server
startThreads creating 1 threads.
starting thread 0
started thread 0
MotionThreadFunc thread started
ven = intel
```

python main.py

Resultado:



C. Ejecución en Docker

```
bashDownExampleBrowser stopping threads
Thread with taskid 0 exiting
Thread TERMINATED
destroy semaphore
semaphore destroyed
destroy main semaphore
main semaphore destroyed
(venv) gixer15abs@gixer15abs-Latitude-E6220:~/Corte_2/TAREA_4$ pip freeze > requirements.txt
(venv) gixer15abs@gixer15abs-Latitude-E6220:~/Corte_2/TAREA_4$ nano Dockerfile
(venv) gixer15abs@gixer15abs-Latitude-E6220:~/Corte_2/TAREA_4$ xhost +local:docker
(venv) gixer15abs@gixer15abs-Latitude-E6220:~/Corte_2/TAREA_4$ docker build -t simulacion-pybullet .
[+] Building 8.7s (3/9)
--> [internal] load build definition from dockerfile
--> transferring dockerfile: 53B
--> [internal] load metadata for docker.io/library/python:3.10-slim
--> [internal] load build context
--> transferring context: 7B
--> [1/7] FROM docker.io/library/python:3.10-slim@sha256:65c843653048a3b2e8bd5083a02f4aaef774974f0f70cbf8ce4e931ac96 6.5s
-->  >> resolve docker.io/library/python:3.10-slim@sha256:65c843653048a3b2e8bd5083a02f4aaef774974f0f70cbf8ce4e931ac96 6.5s
-->  >> sha256:65c843653048a3b2e8bd5083a02f4aaef774974f0f70cbf8ce4e931ac96 digest: sha256:65c843653048a3b2e8bd5083a02f4aaef774974f0f70cbf8ce4e931ac96 6.5s
-->  >> sha256:7c7d077057a1fbfc0e3938edc2724b980753bhbcbew318Scb7a76657fe03 5.31kB 6.5s
-->  >> sha256:9864d8d7cc039985e95888fb7ee240991141176c151ad81f12d643e0986 28.73MB 3.65s
-->  >> sha256:c9661b1e9345ed3064ec93hebo12262429b64eca53158450670rw2231defb3 5.15MB 4.55s
-->  >> sha256:7cc78f5313aaeb0803973dcf97339d7e576e349azc44fc655bf99c93389496 9.44MB 15.65MB 6.5s
-->  >> extracting sha256:8a628cddcc83e90e5a95888fcbb0cc240991141176c151ad81f12d643e0986 2.8s
--> [internal] load build context
--> transferring context: 313.37MB 6.5s
```

1. Creación del Dockerfile

Creamos antes del dockerfile, ejecutamos pip freeze > requirements.txt, donde sera un archivo texto que dice los complementos requeridos para desplegar esta imagen.

Ahora si se crea un archivo llamado Dockerfile en la raíz del proyecto (TAREA_4) con el siguiente contenido:

```
FROM python:3.10-slim

# Instala dependencias necesarias para OpenGL + X11
RUN apt-get update && apt-get install -y \
    python3-tk \
    python3-pip \
    libgl1-mesa-glx \
    libgl2.0-0 \
    x11-apps \
    && rm -rf /var/lib/apt/lists/*

# Crea un directorio de trabajo
WORKDIR /app

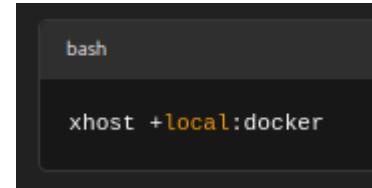
# Copia los archivos del proyecto
COPY . /app

# Instala dependencias Python
RUN pip install --no-cache-dir -r requirements.txt

# Comando por defecto al iniciar el contenedor
CMD ["python", "main.py"]
```

nano dockerfile y agregamos estas lineas de codigo

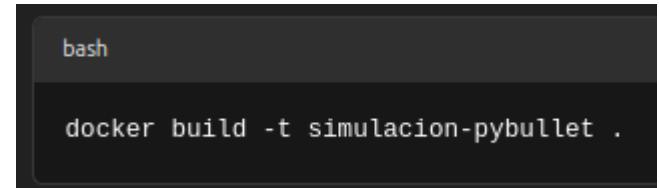
2.Permitir que Docker use X11 para mostrar gráficos



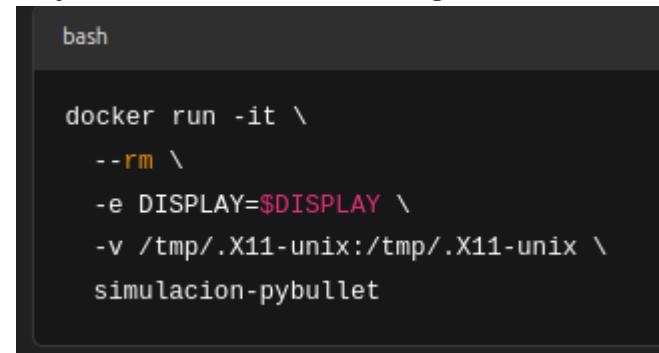
Ejecuta este comando en tu terminal fuera del contenedor para permitir que Docker acceda a tu entorno gráfico

3. Construir la imagen Docker

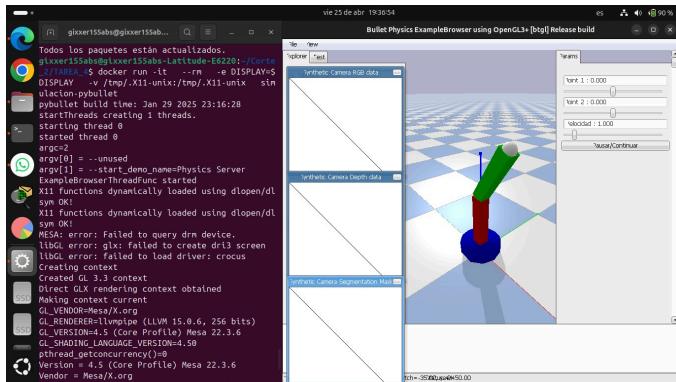
Desde la carpeta ejecuta:



4. Ejecutar el contenedor con acceso gráfico



5.Resultado



```
vie 25 de abr 19:36:54
glicher155abx@glicher155abx-Latitude-E6220i:~/Corte
$ ./TAREA1
Todos los paquetes están actualizados.
glicher155abx@glicher155abx-Latitude-E6220i:~/Corte
$ DISPLAY=: -v /tmp/X11-unix:/tmp/X11-unix sin
dnetc-ping: no se pudo conectar con el destino.
pybullet build time: Jan 29 2025 23:16:28
startThreads time: Jan 29 2025 23:16:28
starting thread 0
started thread 0
args[0] = --unused
args[1] = --start_demo_name=Physics Server
exampleBrowserThreadFunc started
X11 Functions dynamically loaded using dlopen/dl
sym OK
X11 Functions dynamically loaded using dlopen/dl
sym OK
MESA errors: Failed to query drm device.
libGL error: glx: failed to create dri3 screen
Creating context
libGL error: failed to load driver: crocus
Direct GLX rendering context obtained
Making context current
GL_VENDOR=MesaX.org
GL_RENDERER=Mesa DRI (LLW 15.8.6, 256 bits)
GL_VERSION=22.3.6 (Core Profile) Mesa 22.3.6
GL_SHADING_LANGUAGE_VERSION=4.50
pthread_getconcurrency)=0
Version = 4.5 (Core Profile) Mesa 22.3.6
Vendor = MesaX.org
```

IV.CONCLUSIONES

Este ejercicio ha permitido profundizar en conceptos fundamentales de sistemas operativos y seguridad informática, complementando la teoría con aplicación práctica en simulación robótica y tecnologías de despliegue modernas. Se ha logrado una comprensión más integrada de cómo funcionan los mecanismos de bajo nivel del sistema (interrupciones), cómo evolucionan las amenazas de seguridad (bootkits) y cómo se pueden desarrollar y desplegar aplicaciones robóticas simuladas.

Conclusiones Específicas:

Manejo de Interrupciones en Linux:

Se concluye que el subsistema de interrupciones y excepciones en Linux es un mecanismo complejo pero eficiente, esencial para la interacción con el hardware y la gestión de eventos asíncronos y errores.

La diferenciación entre interrupciones (asíncronas, hardware/software) y excepciones (síncronas, CPU) es clave, así como sus respectivos tipos (maskable/NMI, faults/traps/aborts).

El manejo se orquesta a través de la IDT, descriptores de puerta específicos, y un flujo hardware/software bien definido que incluye la gestión de privilegios y el cambio de contexto. Linux utiliza estructuras de datos específicas (`irq_desc_t`, `hw_interrupt_type`, `irqaction`) para gestionar las IRQs, permitir su compartición (SA_SHIRQ), y asociarlas a los manejadores (ISRs) adecuados.

Para optimizar la respuesta, Linux implementa mecanismos de trabajo diferido (Bottom Halves, Softirqs, Tasklets) que permiten ejecutar tareas menos urgentes fuera del contexto inmediato del manejador de interrupción.

En sistemas SMP, el uso de APIC (I/O APIC, Local APIC) y las IPIs son fundamentales para distribuir la carga de interrupciones y facilitar la comunicación entre CPUs.

Amenaza y Evolución de los Bootkits:

Se concluye que los bootkits representan una amenaza de seguridad avanzada y persistente, capaz de comprometer sistemas a un nivel muy bajo (proceso de arranque) para evadir defensas.

Su evolución ha sido constante, desde los primeros virus de sector de arranque hasta sofisticados ataques MBR/VBR (como TDL4, Rovnix, Gapz) y, más recientemente, ataques dirigidos al firmware UEFI.

Las técnicas empleadas son diversas y complejas, incluyendo modificación de código/metadatos de arranque, hooks, uso de características hardware (IDT, registros de depuración), almacenamiento oculto y cifrado, y redes de comunicación personalizadas.

La transición a UEFI, aunque mitiga las amenazas MBR/VBR, presenta nuevos vectores de ataque (reemplazo de bootloader, drivers DXE, Option ROMs) que están siendo activamente explorados. Secure Boot es una defensa importante pero no infalible.

La seguridad del firmware UEFI es crítica pero a menudo descuidada debido a ciclos de actualización lentos y fragmentados, lo que representa un riesgo significativo para ataques futuros, especialmente los dirigidos.

La detección y el análisis forense requieren herramientas especializadas como CHIPSEC y lectores de sistemas de archivos ocultos.

Se logró realizar una simulación funcional de un brazo robótico con dos articulaciones mediante la librería PyBullet, lo cual permitió aplicar conocimientos en cinemática básica, física de cuerpos rígidos, y control de actuadores virtuales, generando una visualización clara del comportamiento robótico.

A pesar de algunos percances técnicos durante la implementación, como la falta de módulos necesarios y problemas con la interfaz gráfica, estos fueron resueltos utilizando un entorno virtual (venv), lo que aseguró la instalación limpia y controlada de dependencias.

La integración de Docker como herramienta de despliegue resultó exitosa. Se demostró que Docker facilita la portabilidad del proyecto, simplifica el proceso de ejecución en diferentes entornos y asegura la reproducibilidad del experimento sin conflictos de dependencias.

En conjunto, el laboratorio proporcionó una experiencia completa, combinando conocimientos de sistemas operativos, ciberseguridad, simulación robótica y herramientas de virtualización modernas, sentando una base sólida para el desarrollo de proyectos más complejos y robustos en el futuro.

REFERENCIAS

- [9] **1. Referencias del Documento "Linux Interrupts: The Basic Concepts"**
 [10] *(Extraídas directamente de la sección 9 del PDF)*
- [11]
- [12] * [1] D. P. Bovet and M. Cesati. Understanding the Linux Kernel: From I/O ports to process management. O'Reilly and Associates, Sebastopol, 2001.
- [13] * [2] Intel. 8259A Programmable Interrupt Controller. Intel Corporation, Dec 1988.
- [14] * [3] Intel. Pentium Pro Family Developers Manual, Volume 2: Programmer's Reference Manual. Intel Corporation, 1995.
- [15] * [4] Intel. 82093AA I/O Advanced Programmable Interrupt Controller (IOAPIC). Intel Corporation, May 1996.
- [16] * [5] Intel. Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture. Intel Corporation, 1997.
- [17] * [6] Intel. Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference. Intel Corporation, 1997.
- [18] * [7] Intel. Intel Architecture Software Developer's Manual, Volume 3: System Programming. Intel Corporation, 1997.
- [19] * [8] Intel. Multiprocessor Specification, Version 1.4. Intel Corporation, May 1997.
- [20] * [9] S. A. Maxwell. Linux Core Kernel Commentary: In-Depth Code Annotation. Coriolis, Arizona, 2001.
- [21]
- [22] **2. Referencias del Documento "BOOTKITS: PAST, PRESENT & FUTURE"**
 [23] *(Extraídas directamente de la sección REFERENCES del PDF)*
- [24]
- [25] * [1] Slade, R. Robert Slade's Guide to Computer Viruses. 2nd Edition, Springer, 1996.
- [26] * [2] Harley, D.; Slade, R.; Gattiker, U. Viruses Revealed. Osborne, 2007.
- [27] * [3] Ször, P. The Art of Computer Virus Research and Defense. Addison Wesley, 2005.
- [28] * [4] Soeder, D.; Permeh, R. eEye BootRoot. BlackHat, 2005.
- [29] * [5] Kumar, N.; Kumar V. Vbootkit. BlackHat 2007.
- [30] * [6] Kleissner, P. Stoned Bootkit. BlackHat 2009.
- [31] * [7] Florio, E.; Kasslin, K. Your Computer is Now Stoned (...Again!): The Rise of MBR Rootkits. Symantec, 2013.
- [32] * [8] Kumar, N.; Kumar V. VBootkit 2.0: Attacking Windows 7 via Boot Sectors. HiTB 2009.
- [33] * [9] Economou, N.; Luksenberg A. Deep Boot. Ekoparty 2011.
- [34] * [10] Ettlinger, W.; Vieböck, S. Evil Core Bootkit: Pwning Multiprocessor Systems. NinjaCon, 2011.
- [35] * [11] Diego, J.; Economou, N.A. VGA Persistent Rootkit. Ekoparty 2012.
- [36] * [12] Rodionov, E.; Matrosov, A. Mind the Gapz: The most complex bootkit ever analyzed? ESET, 2013.
- [37] * [13] Kaczmarek, S. UEFI and Dreamboot. HiTB 2013.
- [38] * [14] Zihang, X. Oldboot: the first bootkit on Android. 360, 2014.
- [39] * [15] Rodionov, E.; Matrosov, A. The Evolution of TDL: Conquering x64. ESET, 2011.
- [40] * [16] Matrosov, A. Olmasco bootkit: next circle of TDL4 evolution (or not?). ESET, 2012.
- [41] * [17] Intel® Platform Innovation Framework for UEFI Specification.
- [42] * [18] UEFI Validation Option ROM Validation Guidance. Microsoft, 2014.
- [43] * [19] Loukas, K. Mac EFI Rootkits. Black Hat 2012.
- [44] * [20] Bulygin, Y.; Furtak, A.; Bazhaniuk, O. A tale of one software bypass of Windows 8 Secure Boot. Black Hat 2013.
- [45] * [21] Kallenberg, C.; Bulygin, Y. All Your Boot Are Belong To Us Intel. MITRE. CanSec West 2014.
- [46] * [22] Intel CHIPSEC. <https://github.com/chipsec/chipsec>.
- [47] * [23] ESET Hidden File System Reader. <https://www.google.com/search?q=http://www.eset.com/int/download//utilities/detail/family/173/>.
- [48] * [24] Matrosov, A.; Rodionov, E.; Harley, D. Rootkits and Bootkits: Advanced Malware Analysis. No Starch, 2015 (in preparation).
- [49]
- [50] **3. Referencias para PyBullet**
 [51] *(Basadas en documentación oficial y recursos relevantes)*
- [52]
- [53] * **PyBullet Quickstart Guide:** (Mencionado en resultados de búsqueda, usualmente encontrado en el repositorio o sitio web oficial)
 - Es el punto de partida recomendado.
- [54] * **PyBullet Planning:** Documentación sobre las funcionalidades de planificación de movimiento.
- [55] * **Bullet Physics SDK Documentation:** PyBullet es la interfaz Python para Bullet Physics. La documentación de Bullet puede ser relevante. (Ver)
- [56] * URL Principal: [Bullet Real-Time Physics Simulation](<https://pybullet.org/wordpress/>)
- [57] * **PyBullet en PyPI:** Información del paquete y metadatos.
- [58] * URL: [pybullet · PyPI](<https://pypi.org/project/pybullet/>)
- [59] * **Repositorio GitHub (Bullet3):** El código fuente de Bullet y PyBullet.
- [60] * URL: [bulletphysics/bullet3 - GitHub](<https://github.com/bulletphysics/bullet3>)
- [61]
- [62] **4. Referencias para Docker**
 [63] *(Basadas en documentación oficial)*
- [64]
- [65] * **Docker Documentation (Sitio Principal):** El portal central para toda la documentación oficial.
- [66] * URL: [Docker Documentation](<https://docs.docker.com/>) (El resultado apunta al repositorio fuente, el sitio en sí es el destino)
- [67] * **Get Started with Docker:** Tutoriales introductorios oficiales.
- [68] * URL: [Get Started | Docker](<https://www.docker.com/get-started/>)
- [69] * **Docker CLI Reference:** Documentación detallada de todos los comandos de Docker.
- [70] * URL: [Use the Docker command line | Docker Docs](<https://www.google.com/search?q=https://docs.docker.com/engine/reference/commandline/cli/>) (Puede variar ligeramente, buscar "docker cli reference" en el sitio oficial)
- [71] * **Dockerfile Reference:** Documentación específica sobre la sintaxis y las instrucciones del Dockerfile.
- [72] * URL: [Dockerfile reference | Docker Docs](<https://docs.docker.com/engine/reference/builder/>)
- [73]
- [74]
- [75] * **Sobre Interrupciones en Linux (Más allá del PDF):**
- [76] * Libro: "Linux Device Drivers" por Jonathan Corbet, Alessandro Rubini, y Greg Kroah-Hartman. (Un clásico que cubre interrupciones en el contexto de drivers).
- [77] * Libro: "Professional Linux Kernel Architecture" por Wolfgang Mauerer. (Muy detallado sobre la arquitectura interna).
- [78] * Artículo/Blog post: "Understanding Interrupts and Tasklets in Linux Kernel Multitasking - Paper-Checker Hub" (Parece un resumen introductorio).
- [79] * Recurso: "[PDF] Analysis of Interrupt Handling Overhead in the Linux Kernel | Semantic Scholar" (Para un análisis más académico del rendimiento).
- [80]
- [81] * **Sobre Bootkits y Seguridad UEFI (Más allá del PDF):**
 * Libro: "Rootkits and Bootkits: Reversing Modern Malware and Next Generation Threats" por Alex Matrosov, Eugene Rodionov, y Sergey Bratus (Nota: A. Matrosov y E. Rodionov son autores del PDF).
- [82] * Recurso: [Papers | Unified Extensible Firmware Interface Forum - UEFI Forum](https://uefi.org/learning_center/papers) (Whitepapers técnicos del foro UEFI).
- [83] * Artículo: "Hiding in the Shadows - Towards Understanding Modern UEFI Bootkits - DUO" (Tesis académica sobre bootkits UEFI).
- [84] * Artículo: "UEFI Bootkit Hunting: In-Depth Search for Unique Code Behavior - BINARLY" (Investigación reciente sobre detección).
- [85] * **Sobre Simulación Robótica (General):**

- [86] * Libro: "Robotics, Vision and Control: Fundamental Algorithms In Python" por Peter Corke. (Cubre conceptos fundamentales implementados en Python).
- [87] * Simulador Alternativo: Gazebo (<http://gazebosim.org/>) - Otro simulador robótico popular, a menudo usado con ROS.
- [88] * **Sobre Docker y Contenedores (General):**
- [89] * Libro: "Docker Deep Dive" por Nigel Poulton. (Una guía popular y práctica).
- [90] * Concepto Relacionado: Kubernetes (<https://kubernetes.io/>) - Plataforma de orquestación de contenedores, a menudo usada junto con Docker.
- [91]
- [92] Espero que esta lista combinada te sea útil para adjuntar a tu documento.