

Tarea 7

Melanie Aponte

Resumen del artículo

El artículo *Linux Interrupts: The Basic Concepts* de Mika J. Järvenpää explica los fundamentos del manejo de interrupciones en el núcleo de Linux (versión 2.4.18-10). Se describe cómo el sistema maneja interrupciones de hardware y software, cómo se implementan las rutas de control del kernel y cómo se utilizan estructuras como la Interrupt Descriptor Table (IDT). También se tratan conceptos como *softirqs*, *tasklets*, y el soporte para sistemas multiprocesadores (SMP) con controladores APIC.

1 ¿Qué es una interrupción y por qué son necesarias?

Una interrupción es un evento asíncrono generado por un dispositivo de entrada/salida (E/S) que solicita la atención del procesador, interrumpiendo el flujo normal de ejecución.

Importancia

- Permiten al sistema responder rápidamente a eventos externos.
- Eliminan la necesidad de consulta constante (*polling*) por parte del CPU.

Tipos y características

- **Interrupciones de hardware:** generadas por dispositivos físicos, como teclados o discos.
- **Interrupciones de software:** generadas mediante instrucciones como `INT n`.
- **Enmascarables y no enmascarables (NMI):** las primeras pueden ser bloqueadas temporalmente.
- **Excepciones:** eventos síncronos que ocurren durante la ejecución de una instrucción, como división por cero.

2 ¿Qué son los “exception handler”?

Los exception handlers son funciones especiales del núcleo diseñadas para gestionar excepciones. Tienen la siguiente estructura:

1. Guardan los registros en la pila del modo kernel.
2. Ejecutan una función C que maneja la excepción.

3. Devuelven el control mediante la instrucción `iret`.

Estas rutinas también pueden enviar señales como `SIGSEGV` o `SIGFPE` al proceso que causó la excepción.

3 ¿Qué son las interrupciones generadas por software y para qué sirven?

Son interrupciones activadas desde el propio código, usando instrucciones como `INT n`. Un ejemplo es `INT 0x80` que invoca una llamada al sistema.

Usos

- Ejecutar llamadas al sistema desde espacio de usuario.
- Forzar condiciones específicas para pruebas o depuración.

Manejo

Se utilizan vectores definidos en la IDT. El flujo de manejo incluye:

1. Guardado del contexto (registros).
2. Ejecución del manejador de interrupciones (`do_IRQ`).
3. Restauración del estado mediante `iret`.

4 ¿Qué son las IRQ y las estructuras de datos relacionadas?

Una **IRQ (Interrupt Request)** es una señal enviada por un dispositivo al CPU para ser atendida.

Estructuras de datos relevantes en Linux

- **`irq_desc_t`:** Describe cada IRQ con campos como `status`, `handler`, `action`.
- **`hw_interrupt_type`:** Define funciones para interactuar con el controlador de interrupciones.
- **`irqaction`:** Lista de funciones a ejecutar cuando ocurre la interrupción (puede estar encadenada para IRQs compartidas).

Estas estructuras permiten compartir IRQs entre dispositivos, diferir el procesamiento con *tasklets* y garantizar la sincronización en sistemas multiprocesadores.

Resumen del segundo artículo

El artículo *Bootkits: Past, Present & Future*, presentado en la conferencia Virus Bulletin 2014 por Eugene Rodionov, Alexander Matrosov y David Harley, analiza la evolución, funcionamiento y futuro de las amenazas conocidas como **bootkits**. Estas herramientas de malware tienen como objetivo infectar y controlar el proceso de arranque del sistema operativo, ocultando su presencia y permitiendo cargar controladores maliciosos de forma sigilosa.

Se presentan casos históricos como Elk Cloner y Brain, y se analizan amenazas modernas como TDL4, Rovnix, Gapz y Dreamboot. Además, se abordan ataques a sistemas UEFI, el uso de cifrado, técnicas de evasión, persistencia y comunicación con servidores de control. Finalmente, se proponen herramientas defensivas como CHIPSEC y sistemas de análisis forense.

5 Puntos clave del documento

5.1 1. Origen y evolución de los bootkits

- Iniciaron con virus de arranque como Elk Cloner y Brain.
- Bootkits modernos surgieron para evadir la política de firma obligatoria de controladores en sistemas Windows de 64 bits.
- PoCs como BootRoot (2005) y Vbootkit (2007) antecedieron a amenazas reales como Mebroot y TDL4.

5.2 2. Técnicas de infección

- **MBR bootkits:** infectan el Master Boot Record (ej. TDL4, Olmasco).
- **VBR bootkits:** infectan el Volume Boot Record (ej. Rovnix, Gapz).
- Utilizan hooking, sectores ocultos y modificación de tablas de partición.

5.3 3. Casos de bootkits avanzados

- **TDL4:** emplea código oculto en el disco para cargar drivers sin firma.
- **Rovnix:** usa hooking de la IDT e interrupciones para interceptar el arranque.
- **Gapz:** modifica ligeramente el VBR, cifra datos con AES y emplea red en modo kernel.

5.4 4. Infección en sistemas UEFI

- Sistemas modernos reemplazan MBR/VBR por UEFI y GPT.
- **Dreamboot:** PoC que reemplaza el cargador `bootmgfw.efi` y desactiva protecciones como PatchGuard mediante hooks.

5.5 5. Futuro y desafíos de seguridad

- Persisten muchos sistemas sin Secure Boot.
- El firmware BIOS/UEFI rara vez se actualiza, creando vectores de ataque duraderos.
- Se esperan más ataques dirigidos directamente al firmware en lugar del SO.

5.6 6. Herramientas de defensa y análisis

- **CHIPSEC:** framework open-source para análisis de seguridad de BIOS/UEFI y forense de firmware.
- **Hidden File System Reader:** herramienta de ESET para analizar almacenamiento oculto de bootkits como TDL4, Rovnix, Flame.

6 Evolución cronológica de los Bootkits

Año	Nombre	Descripción
2005	BootRoot	Primer PoC basado en MBR
2007	Mebroot	Primer bootkit malicioso real
2009	Vbootkit x64	Primer PoC para Windows 7 x64
2010	TDL4	Primer bootkit activo de 64 bits
2011	Rovnix	Primer bootkit basado en VBR
2012	Gapz	Infección VBR extremadamente sigilosa
2013	Dreamboot	Primer PoC UEFI targeting Windows 8
2014	OldBoot	Primer bootkit detectado en Android

Conclusión

Aunque Microsoft introdujo Secure Boot para frenar los bootkits, en la práctica los atacantes han evolucionado su enfoque. Los bootkits seguirán siendo una amenaza en sistemas antiguos y en ataques dirigidos a firmware moderno. El ciclo de seguridad del BIOS/UEFI es más lento que el del software convencional, lo cual representa un riesgo. Es esencial integrar actualizaciones firmes, análisis forense del firmware y defensa activa desde el arranque.

7 simulación sencilla de un brazo robótico con Pybullet y docker.

Objetivo

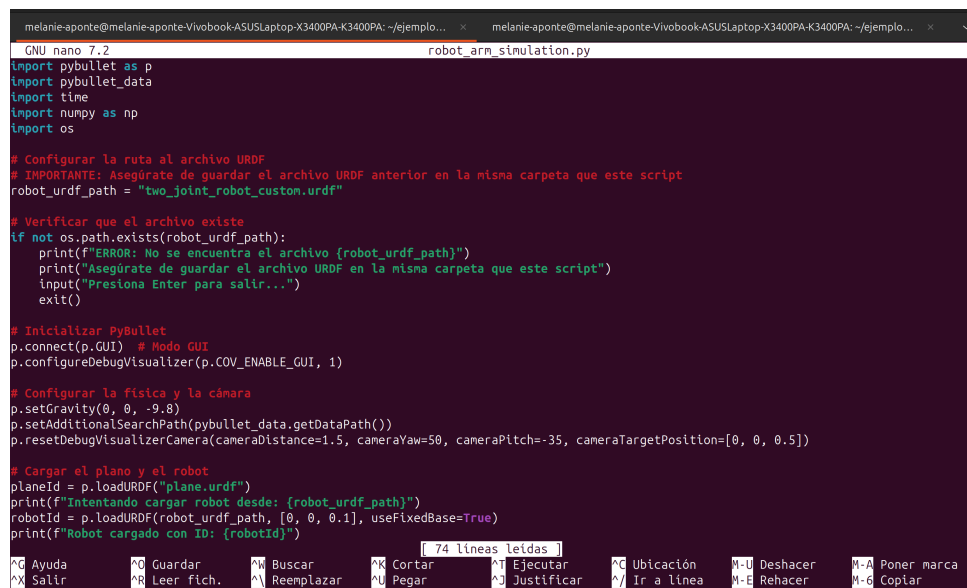
Desplegar una aplicación de simulación de un brazo robótico utilizando la librería PyBullet dentro de un contenedor Docker. La simulación se basa en un archivo URDF personalizado y se ejecuta con visualización en tiempo real.

8 Estructura del Proyecto

El proyecto está compuesto por los siguientes archivos:

```
melanie-aponte@melanie-aponte-Vivobook-ASUSLaptop-X3400PA-K3400PA: ~/ejemplo_iot2$ ls
Dockerfile  robot_arm_simulation.py  two_joint_robot_custom.urdf
```

9 Creación del Archivo de Simulacion .py



```
melanie-aponte@melanie-aponte-Vivobook-ASUSLaptop-X3400PA-K3400PA: ~/ejemplo... x melanie-aponte@melanie-aponte-Vivobook-ASUSLaptop-X3400PA-K3400PA: ~/ejemplo... x
GNU nano 7.2 robot_arm_simulation.py
import pybullet as p
import pybullet_data
import time
import numpy as np
import os

# Configurar la ruta al archivo URDF
# IMPORTANTE: Asegúrate de guardar el archivo URDF anterior en la misma carpeta que este script
robot_urdf_path = "two_joint_robot_custom.urdf"

# Verificar que el archivo existe
if not os.path.exists(robot_urdf_path):
    print(f"ERROR: No se encuentra el archivo {robot_urdf_path}")
    print("Asegúrate de guardar el archivo URDF en la misma carpeta que este script")
    input("Presiona Enter para salir...")
    exit()

# Inicializar PyBullet
p.connect(p.GUI) # Modo GUI
p.configureDebugVisualizer(p.COV_ENABLE_GUI, 1)

# Configurar la física y la cámara
p.setGravity(0, 0, -9.8)
p.setAdditionalSearchPath(pybullet_data.getDataPath())
p.resetDebugVisualizerCamera(cameraDistance=1.5, cameraYaw=50, cameraPitch=-35, cameraTargetPosition=[0, 0, 0.5])

# Cargar el plano y el robot
planeId = p.loadURDF("plane.urdf")
print(f"Intentando cargar robot desde: {robot_urdf_path}")
robotId = p.loadURDF(robot_urdf_path, [0, 0, 0.1], useFixedBase=True)
print(f"Robot cargado con ID: {robotId}")

[ 74 líneas leídas ]
Ctrl+Y Ayuda Ctrl+N Guardar Ctrl+M Buscar Ctrl+K Cortar Ctrl+J Ejecutar Ctrl+U Ubicación Ctrl+L Deshacer Ctrl+P Poner marca
Ctrl+X Salir Ctrl+R Leer fich. Ctrl+O Reemplazar Ctrl+V Pegar Ctrl+I Justificar Ctrl+G Ir a línea Ctrl+E Rehacer Ctrl+C Copiar
```

10 Dockerfile

Este archivo define las dependencias y comandos necesarios para construir el contenedor:

```
melanie-aponte@melanie-aponte-Vivobook-ASUSLaptop-X3400PA-K3400PA: ~/ejemplo... x
GNU nano 7.2 Doc
# Imagen base con Python
FROM python:3.10-slim

# Instalar dependencias del sistema
RUN apt-get update && apt-get install -y \
    python3-tk \
    libgl1-mesa-glx \
    xvfb \
    && rm -rf /var/lib/apt/lists/*

# Instalar PyBullet
RUN pip install pybullet numpy

# Crear directorio de trabajo
WORKDIR /app

# Copiar archivos
COPY robot_arm_simulation.py .
COPY two_joint_robot_custom.urdf /app/two_joint_robot_custom.urdf

# Ejecutar con X virtual framebuffer
CMD python robot_arm_simulation.py
```

11 Construcción de la Imagen

Desde la terminal, en el mismo directorio, se ejecuta:

```
melanie-aponte@melanie-aponte-Vivobook-ASUSLaptop-X3400PA-K3400PA: ~/ejemplo...$ sudo docker build -t pybullet-arm .
[sudo] contraseña para melanie-aponte:
DEPRECATED: The legacy builder is deprecated and will be removed in a future release.
Install the buildx component to build images with BuildKit:
https://docs.docker.com/go/buildx/

Sending build context to Docker daemon  9.728kB
Step 1/7 : FROM python:3.10-slim
--> d7c79db7d957
Step 2/7 : RUN apt-get update && apt-get install -y python3-tk libgl1-mesa-glx xvfb && rm -rf /var/lib/apt/lists/*
--> Using cache
--> 52bc65190d91
Step 3/7 : RUN pip install pybullet numpy
--> Using cache
--> d3e1ace02716
Step 4/7 : WORKDIR /app
--> Using cache
--> e00ce196c31b
Step 5/7 : COPY robot_arm_simulation.py .
--> Using cache
--> e9eea876ed0e
Step 6/7 : COPY two_joint_robot_custom.urdf /app/two_joint_robot_custom.urdf
--> Using cache
--> ed04db38475
Step 7/7 : CMD python robot_arm_simulation.py
--> Running in 5d81f7df67bc
--> Removed intermediate container 5d81f7df67bc
--> d4921f42209d
Successfully built d4921f42209d
Successfully tagged pybullet-arm:latest
```

12 Permisos Gráficos

Para permitir que Docker acceda al entorno gráfico de Ubuntu:

```
sudo docker stop 55110002c000  
melanie-aponte@melanie-aponte-Vivobook-ASUSLaptop-X3400PA-K3400PA:~/ejemplo_iot2$ xhost +local:root  
non-network local connections being added to access control list
```

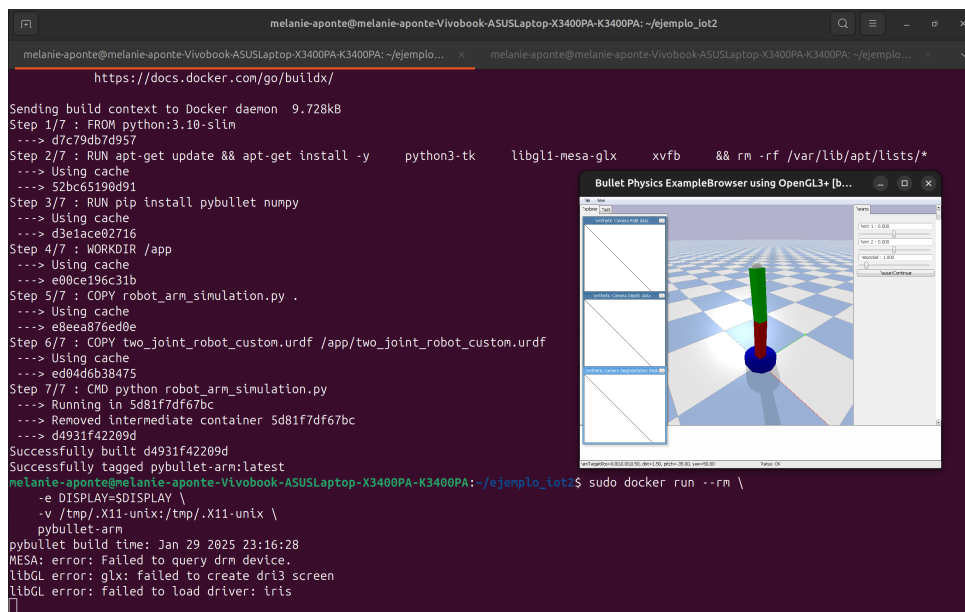
13 Ejecución del Contenedor

Se ejecuta el contenedor permitiendo mostrar la simulación en pantalla:

```
melanie-aponte@melanie-aponte-Vivobook-ASUSLaptop-X3400PA-K3400PA:~/ejemplo_iot2$ sudo docker run --rm \\\n-e DISPLAY=$DISPLAY \\\n-v /tmp/.X11-unix:/tmp/.X11-unix \\\npybullet-arm  
pybullet build time: Jan 29 2025 23:16:28
```

14 Resultado

La simulación se visualiza en una ventana emergente donde el brazo robótico realiza una secuencia de movimientos. Todo se ejecuta dentro de un contenedor, asegurando portabilidad, replicabilidad y separación del entorno local.



- 15 Desarrollar la siguiente simulación de un robot que implementa la tecnología SLAM y LIDAR con Docker, teniendo presente que se debe implementar un TurtleBot3 donde se creará un mapa en tiempo real.

