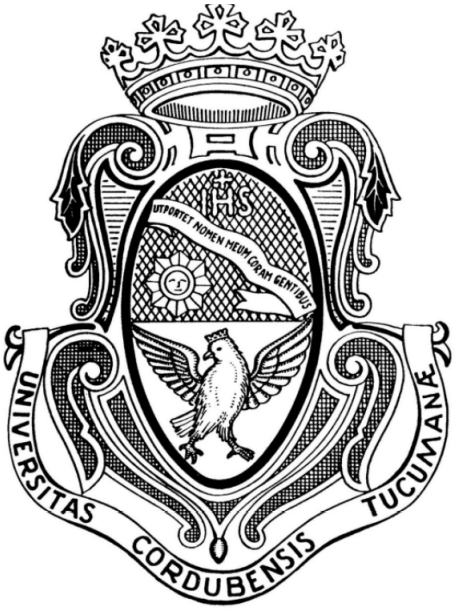
Universidad Nacional de Córdoba



Facultad de Ciencias Exactas, Físicas y Naturales
Escuela de Electrónica y Computacion

Cátedra de Sistemas de Computacion

Trabajo de Laboratorio 3

Profesor Titular: Ing. Javier Alejandro JORGE
Profesor Adjunto: -

Integrantes:
Trucchi, Genaro
Trachtta, Agustin
Rodriguez, Mateo

27 de abril de 2025

Índice

1. Introducción

En este trabajo práctico se aborda la arquitectura UEFI, sus diferencias fundamentales con el antiguo BIOS, la forma de interactuar con ella tanto a nivel usuario como a nivel desarrollador, y se muestra un ejemplo de llamada a una de sus funciones de *Runtime Services*.

2. ¿Qué es UEFI?

UEFI (*Unified Extensible Firmware Interface*) es la especificación que define una interfaz de software moderna entre el firmware de la plataforma (antes BIOS) y el sistema operativo. A diferencia del BIOS tradicional, UEFI:

- Está escrito en C y opera en modo 32 o 64 bits.
- Ofrece controladores propios para hardware básico (red, almacenamiento, gráficos).
- Soporta particiones GPT (superando el límite de 2 TB de MBR).
- Incorpora un entorno seguro (*Secure Boot*) y gestión de clave pública.
- Se organiza en *Boot Services* (solo durante el arranque) y *Runtime Services* (disponibles para el SO tras el `ExitBootServices`).

2.1. ¿Cómo puedo usarlo?

Como usuario final

1. Durante el arranque, pulsar la tecla indicada (DEL, F2, F10, F12, ESC) para entrar en el *Setup* de UEFI.
2. Desde allí se puede:
   - Cambiar el orden de arranque.
   - Habilitar/deshabilitar *Secure Boot*.
   - Ajustar parámetros de hardware (incluido overclocking).
3. En sistemas UNIX-like, utilidades como `efibootmgr` permiten listar y modificar entradas UEFI desde el sistema operativo.

Como desarrollador / cargador de arranque

1. El firmware UEFI pasa al cargador una estructura `EFI_SYSTEM_TABLE`, que incluye punteros a:
   - `BootServices` (servicios disponibles solo durante el arranque).
   - `RuntimeServices` (servicios que perduran tras el `ExitBootServices`).
2. Para invocar una rutina UEFI, se localiza el puntero en la tabla de servicios y se llama en C/C++ correctamente, respetando la ABI UEFI (registro de parámetros, alineamiento, etc.).

2.2. Ejemplo de llamada a función UEFI

Una aplicación UEFI puede acceder a las variables persistentes almacenadas en la NVRAM (Non-Volatile RAM) mediante los *Runtime Services*, que son parte del entorno estándar definido por la especificación UEFI. A continuación, se muestra un ejemplo de cómo utilizar la función `GetVariable` para leer la variable global `BootOrder`, la cual contiene el orden de dispositivos para el arranque del sistema. Esta función está documentada en la sección correspondiente de la especificación oficial UEFI.

```
#include <Uefi.h>
#include <Library/UefiLib.h>
#include <Library/UefiBootServicesTableLib.h>

extern EFI_GUID gEfiGlobalVariableGuid;

EFI_STATUS LeerBootOrder(VOID)
{
    EFI_STATUS status;
    CHAR16 *VariableName = L"BootOrder";
    UINTN DataSize = sizeof(UINT16) * 16;      // espacio para 16 entradas
    UINT16 BootOrder[16];
    UINT32 Attributes;

    status = gST->RuntimeServices->GetVariable(
        VariableName,
        &gEfiGlobalVariableGuid,
        &Attributes,
        &DataSize,
        BootOrder
    );

    if (EFI_ERROR(status)) {
        Print(L"Error al leer BootOrder: %r\n", status);
        return status;
    }
    // Procesar BootOrder...
    return EFI_SUCCESS;
}
```

Listing 1: Lectura de la variable `BootOrder`

Más detalles en la especificación oficial.

3. Casos de bugs de UEFI explotables

Los bugs en UEFI son particularmente peligrosos porque el firmware se ejecuta antes que el sistema operativo y con privilegios muy altos (equivalente a Ring -2 o superior, conceptualmente). Explotarlos puede llevar a ataques muy persistentes y sigilosos (bootkits). Algunos casos notables:

- **LoJax (2018)**: considerado el primer rootkit UEFI in the wild', usado por el grupo APT28 (Fancy Bear). Modificaba directamente la SPI flash del firmware para inyectar un módulo malicioso que persistía incluso tras reinstalar el sistema operativo o cambiar el disco duro. Aprovechaba vulnerabilidades y configuraciones inseguras que permitían escritura no autorizada en la flash de firmware.
- **MosaicRegressor (2020)**: framework de spyware basado en UEFI desarrollado por HackingTeam. Utilizaba imágenes de firmware comprometidas para reinstalar el malware en el sistema operativo cada vez que se eliminaba, asegurando persistencia a nivel de firmware.
- **ThinkPwn (2016)**: vulnerabilidad en ciertos firmwares de Lenovo (y potencialmente otros fabricantes) en la gestión de llamadas SMM (System Management Mode). Permitía a un atacante con privilegios de administrador en el SO escalar a SMM, desde donde podía modificar el firmware UEFI protegido.
- **LogoFAIL (2023)**: conjunto de desbordamientos de búfer en parsers de imágenes de logo personalizados usados por muchos fabricantes de UEFI durante la fase DXE (Driver Execution Environment). Un atacante podía cargar una imagen de logo maliciosa que explotara este bug, obteniendo ejecución de código arbitrario en arranque temprano.
- **Vulnerabilidades en drivers UEFI específicos**: de forma recurrente se descubren bugs (por ejemplo, buffer overflows) en drivers de red, USB, almacenamiento, etc. Si un atacante controla los datos procesados por estos drivers (p. ej. un USB malicioso o un paquete de red manipulado), puede desencadenar la vulnerabilidad y ejecutar código con privilegios de firmware.

4. Converged Security and Management Engine (CSME) e Intel Management Engine BIOS Extension (MEBx)

Converged Security and Management Engine (CSME) / Intel Management Engine (ME): Es un subsistema de microcontrolador autónomo integrado en el chipset (Platform Controller Hub, PCH) de la mayoría de las placas base Intel desde 2006/2008. Funciona de forma independiente de la CPU y del sistema operativo, con su propio firmware (basado en MINIX), memoria y acceso directo a hardware crítico (RAM, red, periféricos). Sus principales funciones son:

- *Intel Active Management Technology (AMT):* administración remota fuera de banda (OOB) — encender/apagar, consola remota, incluso si el SO no responde.
- *Inicialización temprana de hardware:* parte de la configuración del sistema antes de que el BIOS/UEFI principal tome el control.
- *Funciones de seguridad:*
  - Intel Boot Guard: verifica la firma criptográfica del firmware UEFI.
  - Intel Platform Trust Technology (PTT): implementación de TPM en firmware.
  - Gestión de DRM (Protected Audio Video Path, PAVP), etc.

Su naturaleza de "caja negra" propietaria con acceso privilegiado ha generado preocupaciones de seguridad; a lo largo de los años se han descubierto vulnerabilidades críticas en el ME/CSME.

Intel Management Engine BIOS Extension (MEBx): Es el módulo de configuración integrado en el firmware principal (BIOS/UEFI) que permite ajustar las funciones del ME/AMT. Se accede normalmente al presionar `Ctrl+P` (o la combinación específica del fabricante) durante el POST. Desde MEBx se puede:

- Habilitar/deshabilitar AMT.
- Configurar parámetros de red para gestión OOB (dirección IP, VLAN, DNS).
- Establecer contraseñas y políticas de acceso al ME.
- Activar KVM remoto y otras opciones avanzadas de control.

5. coreboot

¿Qué es coreboot? coreboot es un proyecto de firmware libre y minimalista que reemplaza el BIOS/UEFI propietario. Su filosofía es inicializar únicamente el hardware esencial (CPU, RAM, chipset básico) y luego transferir el control a un *payload* especializado.

Payloads comunes:

- `SeaBIOS`: interfaz BIOS tradicional para compatibilidad con SOs antiguos.
- `TianoCore (EDK II)`: implementación de UEFI completa para SOs modernos.
- `GRUB2` / `U-Boot`: cargadores de arranque con gran flexibilidad.
- `LinuxBoot`: utiliza el kernel de Linux como payload para entornos Linux "bare-metal".

Productos que lo incorporan:

- *Chromebooks* de Google (firmware estándar).
- Laptops y desktops de fabricantes enfocados en Linux / privacidad (System76, Purism).
- Diversos sistemas embebidos, routers, servidores y dispositivos de red.
- Comunidad de porting a placas base de escritorio y portátiles de múltiples marcas.

Ventajas de su utilización:

- *Velocidad de arranque* muy superior al firmware propietario.
- *Flexibilidad y personalización*: elección de payloads y fácil adaptación al hardware.
- *Seguridad y transparencia*: código auditables, menor superficie de ataque (aunque a veces requiere *blobs* propietarios para componentes críticos).
- *Control total* sobre el proceso de arranque y eliminación de código heredado innecesario.

6. Linker

6.1. ¿Qué es un linker?

Un linker es una herramienta fundamental en el proceso de desarrollo de software. Actúa como un "ensamblador" de piezas de código y datos provenientes de diferentes fuentes para crear un único archivo de salida.

6.2. ¿Qué hace?

Sus tareas principales son:

- **Combinación de secciones de código y datos:** Los archivos objeto (`.o`) generados por el ensamblador o el compilador contienen secciones como `.text` (código), `.data` (datos inicializados) y `.bss` (datos no inicializados). El linker toma todas las secciones homónimas de los distintos archivos y las une en una sola sección en el archivo de salida, siguiendo un script de linker o las reglas por defecto.
- **Resolución de símbolos:** Cuando el código referencia símbolos (por ejemplo, una función `print_char` o una variable `screen_buffer`), el linker busca su definición entre todos los archivos objeto y bibliotecas de entrada. Luego sustituye cada uso simbólico por la dirección final donde ese símbolo ha sido colocado. Si falta una definición, se produce un error de "referencia no definida"; si hay definiciones duplicadas, un error de "definición múltiple".
- **Relocalización:** Los archivos objeto contienen direcciones provisionales. El linker ajusta (o "parchea") estas direcciones para que apunten a las ubicaciones finales en el ejecutable resultante, teniendo en cuenta la dirección base (por ejemplo, `0x7C00` en un bootloader). Esto incluye tanto saltos internos como referencias a símbolos externos.
- **Generación del archivo de salida:** Finalmente, escribe el ejecutable final en un formato apropiado (ELF, PE, etc.) o, si se solicita (`-oformat binary`), en un binario crudo. Este archivo contiene todas las secciones combinadas, los símbolos resueltos y las direcciones relocalizadas.

7. ¿Qué es la dirección 0x7C00 y por qué es necesaria?

La dirección `0x7C00` es un estándar histórico en la arquitectura x86 que corresponde al punto de memoria donde el BIOS carga el primer sector arrancable (MBR o VBR) de un dispositivo. Su importancia y necesidad en el script del linker se sustenta en varios aspectos:

- **Convención del BIOS:** Cuando una PC compatible arranca, el firmware (BIOS) lee 512 bytes desde el comienzo del disco o del medio seleccionado y los copia en la dirección física `0x7C00`. A partir de ahí, transfiere el control de la CPU a ese bloque de código.
- **Cálculo de direcciones finales:** El código ensamblado incluye referencias a sus propias etiquetas (datos y saltos). Para que esas referencias apunten correctamente durante la ejecución, el linker debe conocer la dirección base donde residirá el bloque completo. Al indicar `. = 0x7C00` en el script, le decimos al linker "trata este desplazamiento como el origen real".
- **Relocalización adecuada:** Gracias a esa directiva, todas las referencias internas (por ejemplo, direcciones de cadenas o puntos de salto) se ajustan en el momento de enlace de modo que, una vez cargado en `0x7C00`, el programa funcione sin errores de dirección.
- **Evitar desajustes catastróficos:** Si el linker no supiera esta dirección, asumiría una base por defecto (normalmente 0). El BIOS cargaría el código en `0x7C00`, pero las direcciones binarias incluidas seguirían apuntando al origen equivocado, haciendo que cualquier acceso a datos o saltos termine en memoria incorrecta y provoque un fallo inmediato.

8. Creación y prueba de un bootloader básico

8.1. Objetivo

El objetivo de esta práctica es comprender el proceso de creación de un **bootloader** sencillo, capaz de ejecutarse directamente al arrancar una computadora, sin la intervención de un sistema operativo.

8.2. Conceptos fundamentales

Un bootloader es el primer fragmento de código ejecutado por el procesador luego del encendido. Cuando la BIOS finaliza sus rutinas de inicialización, busca un sector de arranque (*boot sector*) en los dispositivos conectados. Este sector debe cumplir las siguientes condiciones:

- Tener un tamaño de **512 bytes**.
- Terminar con la **firma mágica** `0xAA55`.
- El código debe asumir que será cargado en la dirección de memoria física **0x7C00**.

8.3. Procedimiento realizado

8.3.1. Creación del código en ensamblador

Se escribió un programa en lenguaje ensamblador NASM que imprime caracteres en pantalla utilizando servicios de la BIOS:

```
[org 0x7C00]

mov ah, 0x0E
mov al, 'H'
int 0x10

mov al, 'E'
int 0x10

mov al, 'L'
int 0x10

mov al, 'L'
int 0x10

mov al, 'O'
int 0x10

hlt

times 510-($-$$) db 0
dw 0xAA55
```

Listing 2: Código del bootloader básico

Explicación:

- `[org 0x7C00]`: indica que el código debe pensar que se ejecuta a partir de la dirección `0x7C00`.
- `mov ah, 0x0E` + `int 0x10`: llama a una interrupción BIOS para imprimir un carácter en pantalla en modo texto.
- `times 510-($-$$) db 0`: rellena con ceros hasta llegar a 510 bytes.
- `dw 0xAA55`: agrega la firma mágica necesaria para que la BIOS reconozca el sector como booteable.

8.3.2. Compilación

Se utilizó NASM para ensamblar el código directamente a un archivo binario plano:

```
nasm -f bin main.asm -o main.img
```

Listing 3: Compilación del bootloader

8.3.3. Prueba en máquina virtual

Antes de grabarlo en hardware real, se utilizó **QEMU** para probar el funcionamiento de la imagen:

```
qemu-system-x86_64 -drive format=raw,file=main.img -display sdl
```

Listing 4: Ejecución en QEMU

Esto lanzó una ventana de emulación donde se pudo observar la impresión de "HELLO" en la pantalla al arrancar.

8.3.4. Opcional: grabación en un pendrive físico

Para probar el bootloader en hardware real, se puede grabar el archivo en un pendrive siguiendo precauciones extremas para no sobrescribir otros dispositivos:

```
sudo dd if=main.img of=/dev/sdX bs=512 count=1 status=progress
```

Listing 5: Grabación en dispositivo USB

donde `sdX` debe ser reemplazado por el identificador correcto del pendrive.

8.4. Resultados

Se logró exitosamente ejecutar un programa de arranque que imprime texto en pantalla, tanto en entorno virtualizado como en condiciones aptas para hardware real.

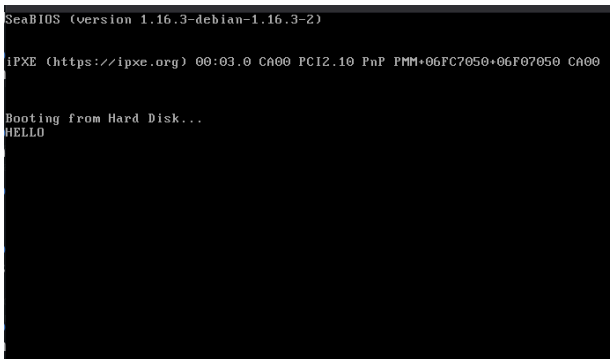


Figura 1: Bootloader mostrando "HELLO" en QEMU.

8.5. Conclusiones

8.5.1. Prueba en hardware real

Una vez comprobado el funcionamiento del bootloader en entorno virtualizado, se procedió a grabarlo en un dispositivo USB físico y probarlo en una computadora real.

Los pasos realizados fueron:

1. Conectar únicamente el pendrive para evitar confusiones con otros discos.
2. Identificar el dispositivo correcto mediante el comando:

```
lsblk
```

3. Desmontar la partición montada del pendrive:

```
sudo umount /dev/sdc1
```

4. Grabar la imagen del bootloader en el dispositivo completo:

```
sudo dd if=main.img of=/dev/sdc bs=512 count=1 status=progress
```

5. Expulsar de forma segura el pendrive:

```
sudo eject /dev/sdc
```

6. Insertar el pendrive en la computadora de prueba, acceder al *Boot Menu* durante el arranque (presionando F12, F10, ESC o la tecla correspondiente) y seleccionar el pendrive como dispositivo de arranque **en modo Legacy (no UEFI)**.

Al bootear desde el pendrive, el sistema ejecutó correctamente el bootloader, mostrando el texto programado.

Figura 2: Bootloader mostrando “HELLO” desde pendrive en hardware real.

Esta práctica permitió:

- Comprender el flujo de arranque de una PC desde nivel BIOS.
- Aplicar conocimientos de lenguaje ensamblador en un entorno de bajo nivel.
- Experimentar con herramientas de emulación de hardware.

Se consolidó así el conocimiento sobre cómo interactuar directamente con el hardware sin sistemas operativos intermedios.

9. Conclusiones

El presente trabajo práctico permitió recorrer desde los conceptos teóricos de la arquitectura **UEFI** hasta la implementación y prueba de un *bootloader* propio, revelando la interacción detallada entre firmware, herramientas de *toolchain* y hardware.

- **UEFI como reemplazo de BIOS**. Comprendimos que UEFI no es una simple evolución sino una plataforma firmware escrita en C, capaz de operar en 32 / 64 bits, con servicios de arranque y ejecución en tiempo de *runtime*, soporte GPT, *Secure Boot* y controladores integrados. Aprender a invocar sus tablas (`BootServices` y `RuntimeServices`) habilita el desarrollo de utilidades de muy bajo nivel que conviven con el sistema operativo.
- **Superficie de ataque y riesgos**. Casos como *LoJax*, *ThinkPwn* o *LogoFAIL* evidencian que vulnerar el firmware permite *bootkits* persistentes aun tras reinstalar el SO. La seguridad del arranque exige tanto configuraciones correctas (protección de la flash) como la aplicación oportuna de parches.
- **CSME / ME y MEBx**. Las funciones de gestión remota (AMT) y arranque verificado (Boot Guard) ofrecen ventajas, pero el motor autónomo de Intel sigue siendo una *caja negra* con fallos críticos publicados. Conocer MEBx resulta clave para equilibrar administración y exposición al riesgo.
- **coreboot: firmware libre**. coreboot demuestra que es posible reemplazar firmware propietario por una solución mínima y auditada, con arranques más veloces y menor superficie de ataque, manteniendo la flexibilidad gracias a los *payloads* (SeaBIOS, TianoCore, LinuxBoot, etc.).
- **Herramientas de *toolchain***. Revisar el `linker` y la convención histórica de 0x7C00 mostró que los scripts de enlace son tan cruciales como el código; si el *linker* desconoce la dirección final, los desplazamientos fallan y el arranque se aborta.
- **Bootloader "HELLO"**. Desde ensamblar 512 bytes y firmarlos con `0xAA55`, hasta probar en QEMU y grabar en un pendrive, cerramos el ciclo completo: escribir, enlazar, testear y ejecutar código que arranca sin sistema operativo.

En síntesis, el TP consolidó cuatro competencias centrales:

1. Comprender el flujo de arranque, desde el encendido hasta la entrega de control al SO.
2. Escribir y depurar código de muy bajo nivel (ASM y C con conciencia de firmware).
3. Reconocer amenazas y aplicar contramedidas en el firmware (UEFI, CSME/ME, Secure Boot).
4. Valorar alternativas *open source* como coreboot para lograr transparencia y control.

Dominar estos pilares habilita tanto el desarrollo cercano al hardware como la auditoría y protección de la primera línea de defensa en sistemas informáticos modernos.