



Sistemas de Computación

Práctico 3:

Modo Protegido.

Alumnos: Cesana Andrés Agustín, Felipe Pillichody,
Sol Agustina Nou.

Profesores: Jorge Javier Alejandro, Solinas Miguel
Ángel.

1.1 UEFI

UEFI significa **Unified Extensible Firmware Interface**. Es una especificación estandarizada que define una interfaz entre el firmware del sistema y el sistema operativo. Fue diseñado como reemplazo del BIOS tradicional en las computadoras modernas. Básicamente, es el software que corre antes de que se inicie el sistema operativo.

Ofrece mejoras con respecto al BIOS como:

- Mayores velocidades de arranque de la computadora.
- Compatibilidad con discos duros grandes (mayores de 2 TB), usando el sistema de partición **GPT**.
- Interfaces gráficas más avanzadas y amigables, reemplazando la de BIOS con sólo texto.
- Seguridad mejoradas como **Secure Boot**, que ayuda a prevenir que se carguen programas maliciosos al arrancar.

1.1.2 ¿Cómo puedo usar UEFI?

Hay dos maneras principales:

A. Como usuario general:

El firmware se ejecuta automáticamente al encender la computadora, sin intervención directa del usuario, es decir se utiliza “sin que el usuario se de cuenta”. Sin embargo, puede acceder a su configuración con las teclas especificadas según cada PC. Desde ese menú el usuario puede:

- Cambiar el orden de arranque (por ejemplo, arrancar desde un USB).
- Configurar opciones como Secure Boot, Virtualización, Overclocking, etc.
- Ver información del sistema, como hardware instalado, temperatura, etc.

B. Como programador o desarrollador:

UEFI permite el desarrollo de aplicaciones y controladores que se ejecutan directamente en el entorno UEFI, antes de cargar el sistema operativo. Esto se hace usando la especificación UEFI y entornos como EDK II (EFI Development Kit II). Se programa típicamente en C, utilizando interfaces definidas por la especificación. Estas aplicaciones pueden ser utilizadas, por ejemplo, para tareas de diagnóstico, mantenimiento del sistema o carga de sistemas operativos personalizados.

Para utilizar UEFI, generalmente se necesita acceder a la configuración del firmware de la computadora durante el arranque. Esto se hace típicamente presionando una tecla específica (como F2, F10, Esc o Del) justo después de encender o reiniciar la misma.

1.2 Vulnerabilidades en UEFI

Las vulnerabilidades en UEFI son **muy graves** porque afectan la **integridad del firmware** y de mecanismos de seguridad como **Secure Boot**. Además, aunque requieren **privilegios de administrador**, una vez explotadas podrían permitir persistencia profunda en el sistema, incluso sobrevivir formateos y reinstalaciones del sistema operativo. Algunos casos fueron:

1.2.1 MoonBounce (2022)

Fue un malware que infecta directamente el firmware UEFI. Alteraba el firmware en la SPI Flash (la memoria donde se guarda el UEFI), permitiendo que el malware se ejecutara antes que cualquier antivirus o el sistema operativo, siendo prácticamente indetectable. Fue descubierto por Kaspersky.

1.2.2 BootHole (2020)

Un fallo en GRUB2 (el gestor de arranque usado en Linux). Permitía modificar la configuración de GRUB y así **desactivar Secure Boot** o **inyectar código malicioso** en el proceso de arranque. Fue descubierto por Eclypsium.

1.2.3 ThinkPwn (2016)

Una vulnerabilidad en el firmware UEFI de algunos modelos de **Lenovo ThinkPad**. Permitía a un atacante obtener acceso completo al System Management Mode (SMM), la parte más privilegiada del procesador (más que el sistema operativo).

1.2.4 LightEater (2015)

Una prueba de concepto presentada en la conferencia Black Hat, donde mostraron que era posible **infectar UEFI** para comprometer todo el sistema operativo.

1.3 Intel Converged Security and Management Engine (Intel CSME)

Intel CSME es un subsistema embebido y dispositivo PCIe que actúa como controlador de seguridad y gestión dentro del PCH. Está diseñado para operar en un entorno aislado del software principal del sistema, como el BIOS, el sistema operativo y las aplicaciones. Accede a interfaces limitadas como GPIO y LAN/WLAN para sus funciones, y su firmware y configuración se almacenan en memoria NVRAM, típicamente en memoria flash en el bus SPI.

Intel CSME está presente en la mayoría de las plataformas de Intel, incluyendo sistemas de consumo y comerciales para clientes, estaciones de trabajo, servidores y productos de IoT (Internet de las Cosas). Para la seguridad basada en hardware, usuarios como proveedores de contenido u organizaciones de TI (Tecnología de la Información) pueden gestionar, por ejemplo, la gestión de derechos digitales (DRM) y la Tecnología de Gestión Activa de Intel (Intel AMT), la cual requiere que la seguridad a nivel de hardware esté disponible cuando el sistema anfitrión no responde o está apagado.

1.3.1 Funciones principales

- Inicialización del Silicio: se encarga de la inicialización básica del PCH, autenticación y carga de firmware en componentes de hardware integrados, y depuración segura del PCH.
- Gestionabilidad: mejora la gestión remota de plataformas a través de Intel AMT, permitiendo características como redirección de consola, redirección de almacenamiento USB, control remoto de teclado, vídeo y ratón, control remoto de energía, gestión de eventos, y diagnóstico independiente del sistema operativo.
- Seguridad: incluye tecnologías como Intel PTT para soporte de TPM, Intel Boot Guard para integridad del arranque, soporte de DRM de hardware, y capacidades de carga y ejecución segura de firmware y applets, proporcionando un alto nivel de seguridad a nivel de hardware.

1.4 Coreboot

Coreboot es un proyecto open source que reemplaza el firmware propietario en las computadoras. Inicializa el hardware y luego transfiere el control a un payload que, generalmente, arranca el sistema operativo. Su diseño flexible permite su uso en aplicaciones especializadas, ejecución de sistemas operativos desde flash, carga de cargadores de arranque personalizados y la implementación de estándares de firmware. Esto reduce la cantidad de código y el espacio en flash necesario, incluyendo solo las funciones esenciales para la aplicación específica.

1.4.1 Productos que lo incorporan

- Chromebooks: muchos Chromebooks, especialmente aquellos fabricados por Google, utilizan Coreboot como su firmware base.
- Servidores: algunos fabricantes de servidores, como Facebook, han adoptado Coreboot en sus centros de datos para una inicialización rápida y eficiente del sistema.
- Dispositivos embebidos: Coreboot se utiliza en una variedad de dispositivos embebidos y sistemas integrados donde se necesita un firmware ligero y flexible.

1.4.2 Ventajas

- **Open source:** Coreboot se basa en los principios del Software de Código Abierto. Muchos de los ingenieros que trabajan en coreboot también han trabajado en el núcleo de Linux. En lugar de mantener las mejoras de un sistema en secreto para todos los demás proveedores, en Coreboot, estas mejoras se comparten en todos los ámbitos, proporcionando a los usuarios finales un firmware mucho mejor y mucho más estable.
- **Flexibilidad:** la principal flexibilidad que ofrece Coreboot es a través del uso de diferentes payloads. Soporta el arranque de sistemas operativos heredados a través de SeaBIOS, el arranque de red con una ROM IPXE integrada o el último payload UEFI. Se pueden crear payloads personalizados utilizando la herramienta lib payload con licencia BSD.
- **Seguridad:** Coreboot viene con una Base de Confianza Mínima que reduce la superficie general de ataque. También soporta un proceso de arranque seguro llamado VBOOT2. Está escrito en el estándar MISRA-C y proporciona otros lenguajes como Ada para la verificación formal de propiedades especiales. Además, el uso de características de plataforma como IOMMU, protecciones de flash y el modo SMM desactivado aumentan la seguridad también.
- **Rendimiento:** los ingenieros de coreboot han trabajado en muchos proyectos de software críticos para la seguridad. La arquitectura de Coreboot está diseñada para tener un proceso de actualización inalterable.
- **Rendimiento:** Coreboot está diseñado para arrancar rápidamente. Para equipos de escritorio y portátiles, puede arrancar frecuentemente al inicio del sistema operativo en menos de un segundo. Para servidores, puede reducir minutos del tiempo de arranque. Algunos proveedores han demostrado una disminución en el tiempo de arranque de más del 70% en comparación con el BIOS OEM.

2.1 Linker

Un linker es una herramienta del proceso de compilación de programas que combina uno o más archivos objeto (resultantes de la compilación de archivos fuente) y bibliotecas en un solo archivo ejecutable, biblioteca compartida u otro formato de archivo de salida. El linker realiza varias funciones esenciales, entre las cuales se incluyen:

- **Resolución de símbolos:** encuentra las definiciones de todas las referencias simbólicas en los archivos objeto, asegurándose de que todas las llamadas a funciones y referencias a variables estén correctamente resueltas.
- **Reubicación:** ajusta las direcciones de los programas y datos en los archivos objeto para reflejar su ubicación final en el archivo ejecutable.

- **Combina segmentos:** junta las distintas secciones de código y datos (como las secciones `.text` para el código y `.data` para los datos) en un formato coherente y continuo.

2.2 Caso práctico: análisis del Código Ensamblador y Script de Linker en el Bootloader.

Pasamos a analizar el código concreto utilizado en este proyecto. Se trata de un programa escrito en lenguaje ensamblador para arquitectura x86 en modo real, diseñado para ser cargado y ejecutado directamente por el BIOS al iniciar el sistema.

link.ld

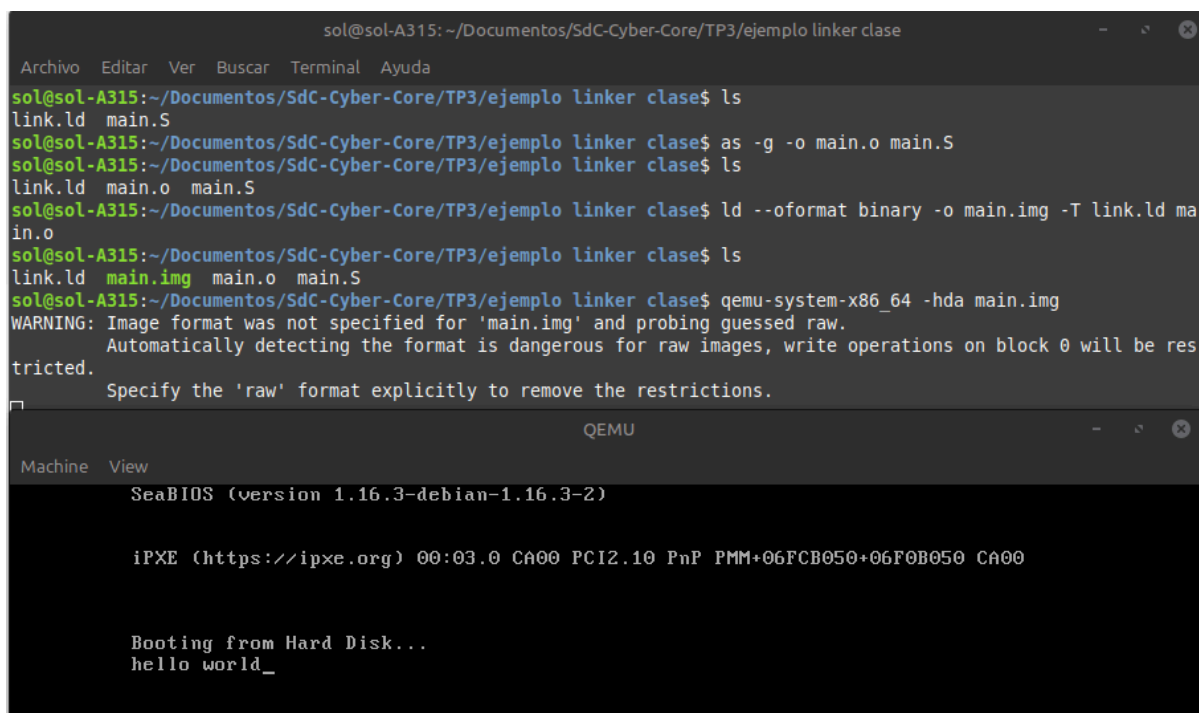
```
SECTIONS
{
    /* The BIOS loads the code from the disk to this location.
    * We must tell that to the linker so that it can properly
    * calculate the addresses of symbols we might jump to. */
    . = 0x7c00;
    .text :
    { __start = .;
      *(.text)
      /* Place the magic boot bytes at the end of the first 512 sector. */
      . = 0x1FE;
      SHORT(0xAA55)
    }
}
```

main.S

```
.code16
    mov $msg, %si
    mov $0x0e, %ah
loop:
    lodsb
    or %al, %al
    jz halt
    int $0x10
    jmp loop
halt:
    hlt
msg:
    .asciz "hello world"
```

Compilamos y corremos la imagen con:

```
as -g -o main.o main.S
ld --oformat binary -o main.img -T link.ld main.o
qemu-system-x86_64 -hda main.img
```



```
sol@sol-A315: ~/Documentos/SdC-Cyber-Core/TP3/ejemplo linker clase
Archivo Editar Ver Buscar Terminal Ayuda
sol@sol-A315:~/Documentos/SdC-Cyber-Core/TP3/ejemplo linker clase$ ls
link.ld main.S
sol@sol-A315:~/Documentos/SdC-Cyber-Core/TP3/ejemplo linker clase$ as -g -o main.o main.S
sol@sol-A315:~/Documentos/SdC-Cyber-Core/TP3/ejemplo linker clase$ ls
link.ld main.o main.S
sol@sol-A315:~/Documentos/SdC-Cyber-Core/TP3/ejemplo linker clase$ ld --oformat binary -o main.img -T link.ld main.o
sol@sol-A315:~/Documentos/SdC-Cyber-Core/TP3/ejemplo linker clase$ ls
link.ld main.img main.o main.S
sol@sol-A315:~/Documentos/SdC-Cyber-Core/TP3/ejemplo linker clase$ qemu-system-x86_64 -hda main.img
WARNING: Image format was not specified for 'main.img' and probing guessed raw.
Automatically detecting the format is dangerous for raw images, write operations on block 0 will be restricted.
Specify the 'raw' format explicitly to remove the restrictions.

QEMU
Machine View
SeaBIOS (version 1.16.3-debian-1.16.3-2)

iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+06FCB050+06F0B050 CA00

Booting from Hard Disk...
hello world_
```

Figura 1: creación y ejecución de la imagen con QEMU.

El archivo *main.S* contiene un pequeño programa en ensamblador de 16 bits que muestra el mensaje "hello world" en pantalla usando la interrupción de BIOS int 0x10, que permite escribir caracteres en modo texto. El mensaje se encuentra en la etiqueta msg. El programa copia el contenido del mensaje carácter por carácter en el registro %al, y mientras no llegue al fin de cadena (0x00), los imprime. Al finalizar, ejecuta hlt para detener el CPU.

El archivo *link.ld* es un script del linker que le indica al mismo cómo organizar el contenido del programa en memoria. Este script es importante para crear la imagen de arranque de bajo nivel porque hay varias consideraciones críticas que el linker debe respetar para que el código funcione como un bootloader:

La dirección 0x7C00

. = 0x7c00; En el script del link encontramos esta dirección. Esta establece la dirección de carga base del código en memoria a 0x7C00, siendo esa la dirección de memoria real donde el BIOS carga el primer sector de arranque (boot sector) del dispositivo de almacenamiento (como un disco duro, disquete o USB) cuando una PC arranca.

La dirección es una convención histórica del diseño original del BIOS de IBM PC. Se eligió porque:

- Está lo suficientemente alto en la memoria para no interferir con interrupciones del BIOS u otras estructuras del sistema.
- Está lo suficientemente bajo como para que todo el código de arranque quepa (512 bytes) antes de llegar a otras áreas usadas por el BIOS o por el sistema operativo.

Master Boot Record

Cuando una PC inicia, el BIOS:

1. Busca un dispositivo de arranque (por ejemplo, disco duro o USB).
2. Carga el primer sector de ese dispositivo, que son los primeros 512 bytes de la RAM en la dirección 0x0000:0x7C00, que es equivalente a la dirección lineal 0x7C00. Vemos que la imagen generada tiene exactamente ese tamaño:

```
sol@sol-A315:~/Documentos/SdC-Cyber-Core/TP3/ejemplo linker clase$ ls -l main.img
-rwxrwxr-x 1 sol sol 512 may  3 19:22 main.img
```

Figura 2: Verificación de tamaño en bytes de main.img.

3. Luego transfiere la ejecución a esa dirección.

Este sector se llama **Master Boot Record (MBR)** o **boot sector**, y debe terminar con la firma 0xAA55 en los últimos dos bytes, como vemos en las líneas `. = 0x1FE; SHORT(0xAA55)`. El código que se coloca en el MBR se llama el **bootloader**, que es responsable de iniciar el sistema operativo o de cargar un cargador de segundo nivel.

2.2.1 Verificación de la Imagen Generada: Comparación entre objdump y hd

Para validar que el código ensamblador ha sido correctamente ubicado dentro del sector de arranque, se realiza una comparación entre la salida de `objdump`, que permite inspeccionar a nivel de instrucciones y direcciones, y `hd` (hexdump), que muestra el contenido binario plano de la imagen. Esta comparación permite confirmar que el código comienza en la dirección 0x7C00 y que la firma 0xAA55 se encuentra en la posición esperada, garantizando que el BIOS lo reconoce como un sector booteable.

Análisis del objdump

Usamos el siguiente comando para desensamblar el archivo binario main.img como si fuera código de una arquitectura Intel 8086:

```
objdump -D -b binary -m i8086 main.img.
```

Este comando interpreta el archivo como un binario plano (-b binary), sin encabezados, y lo traduce a instrucciones máquina del procesador i8086 (-m i8086). Debido a que no tiene cabecera, objdump asume que el código comienza en la dirección 0x0000. Sin embargo, en la práctica, este código se ejecutará en la dirección física 0x7C00, ya que la BIOS carga el primer sector del disco en esa posición de memoria durante el proceso de arranque.

Observamos parte de la salida:

```
main.img:      formato del fichero binary

Desensamblado de la sección .data:

00000000 <.data>:
 0:  be 0f 7c          mov     $0x7c0f,%si
 3:  b4 0e          mov     $0xe,%ah
 5:  ac          lods     %ds:(%si),%al
 6:  08 c0          or      %al,%al
 8:  74 04          je      0xe
 a:  cd 10          int     $0x10
 c:  eb f7          jmp     0x5
 e:  f4          hlt
 f:  68 65 6c       push    $0x6c65
12:  6c          insb     (%dx),%es:(%di)
13:  6f          outsw   %ds:(%si),(%dx)
14:  20 77 6f       and     %dh,0x6f(%bx)
17:  72 6c          jb      0x85
19:  64 00 66 2e    add     %ah,%fs:0x2e(%bp)
1d:  0f 1f 84 00 00 nopw    0x0(%si)
22:  00 00          add     %al,(%bx,%si)
24:  00 66 2e       add     %ah,0x2e(%bp)
27:  0f 1f 84 00 00 nopw    0x0(%si)
2c:  00 00          add     %al,(%bx,%si)
2e:  00 66 2e       add     %ah,0x2e(%bp)
31:  0f 1f 84 00 00 nopw    0x0(%si)
36:  00 00          add     %al,(%bx,%si)
38:  00 66 2e       add     %ah,0x2e(%bp)
3b:  0f 1f 84 00 00 nopw    0x0(%si)
40:  00 00          add     %al,(%bx,%si)
42:  00 66 2e       add     %ah,0x2e(%bp)
45:  0f 1f 84 00 00 nopw    0x0(%si)
```

Figura 3: salida de la herramienta objdump sobre main.img

Línea 0: Los primeros bytes `be 0f 7c` corresponden a la instrucción `mov $0x7c0f, %si`, que establece el registro SI apuntando dentro del segmento 0x7C00. Esto concuerda con la lógica del arranque, donde se apunta a una dirección relativa a 0x7C00 para comenzar a procesar cadenas o datos.^[10]

A continuación se observa un bucle que imprime el string "hello world" carácter por carácter utilizando la interrupción BIOS 0x10 (línea a). También se ve la instrucción hlt para detener el sistema.

Entre las líneas f -19: aparecen los bytes ASCII que representan la cadena "hello world", con un terminador nulo (0x00). Esto se corresponde con la instrucción .asciz "hello world" en el código fuente en ensamblador..

Línea 1fd: podemos ver que ahí contiene los tres bytes 55 AA, que constituyen la firma del sector de arranque. La instrucción mostrada por objdump (add %dl, -0x56(%di)) no es relevante funcionalmente, es solo una interpretación de los bytes que en realidad están cumpliendo la función de indicarle al BIOS que este sector es arrancable.

```
1fd: 00 55 aa                pop     %ds
                add     %dl, -0x56(%di)
```

Figura 4: última línea de la salida de la herramienta objdump sobre main.img.

Análisis del hd (hexdump)

Al usar `hd main.img`, vemos en hexadecimal el contenido binario del archivo. Este comando no intenta interpretar el contenido como código, sino que simplemente muestra los bytes tal como están grabados en el archivo.

```
00000000  be 0f 7c b4 0e ac 08 c0 74 04 cd 10 eb f7 f4 68 |..|.....t.....h|
00000010  65 6c 6c 6f 20 77 6f 72 6c 64 00 66 2e 0f 1f 84 |ello world.f....|
00000020  00 00 00 00 00 66 2e 0f 1f 84 00 00 00 00 66 |....f.....f|
00000030  2e 0f 1f 84 00 00 00 00 00 66 2e 0f 1f 84 00 00 |.....f.....|
00000040  00 00 00 66 2e 0f 1f 84 00 00 00 00 66 2e 0f |...f.....f..|
00000050  1f 84 00 00 00 00 00 66 2e 0f 1f 84 00 00 00 |.....f.....|
00000060  00 66 2e 0f 1f 84 00 00 00 00 66 2e 0f 1f 84 |.f.....f....|
00000070  00 00 00 00 00 66 2e 0f 1f 84 00 00 00 00 66 |....f.....f|
00000080  2e 0f 1f 84 00 00 00 00 00 66 2e 0f 1f 84 00 00 |.....f.....|
00000090  00 00 00 66 2e 0f 1f 84 00 00 00 00 66 2e 0f |...f.....f..|
000000a0  1f 84 00 00 00 00 00 66 2e 0f 1f 84 00 00 00 |.....f.....|
000000b0  00 66 2e 0f 1f 84 00 00 00 00 66 2e 0f 1f 84 |.f.....f....|
000000c0  00 00 00 00 00 66 2e 0f 1f 84 00 00 00 00 66 |....f.....f|
000000d0  2e 0f 1f 84 00 00 00 00 00 66 2e 0f 1f 84 00 00 |.....f.....|
000000e0  00 00 00 66 2e 0f 1f 84 00 00 00 00 66 2e 0f |...f.....f..|
000000f0  1f 84 00 00 00 00 00 66 2e 0f 1f 84 00 00 00 |.....f.....|
00000100  00 66 2e 0f 1f 84 00 00 00 00 66 2e 0f 1f 84 |.f.....f....|
00000110  00 00 00 00 00 66 2e 0f 1f 84 00 00 00 00 66 |....f.....f|
00000120  2e 0f 1f 84 00 00 00 00 00 66 2e 0f 1f 84 00 00 |.....f.....|
00000130  00 00 00 66 2e 0f 1f 84 00 00 00 00 66 2e 0f |...f.....f..|
00000140  1f 84 00 00 00 00 00 66 2e 0f 1f 84 00 00 00 |.....f.....|
00000150  00 66 2e 0f 1f 84 00 00 00 00 66 2e 0f 1f 84 |.f.....f....|
00000160  00 00 00 00 00 66 2e 0f 1f 84 00 00 00 00 66 |....f.....f|
00000170  2e 0f 1f 84 00 00 00 00 00 66 2e 0f 1f 84 00 00 |.....f.....|
00000180  00 00 00 66 2e 0f 1f 84 00 00 00 00 66 2e 0f |...f.....f..|
00000190  1f 84 00 00 00 00 00 66 2e 0f 1f 84 00 00 00 |.....f.....|
000001a0  00 66 2e 0f 1f 84 00 00 00 00 66 2e 0f 1f 84 |.f.....f....|
000001b0  00 00 00 00 00 66 2e 0f 1f 84 00 00 00 00 66 |....f.....f|
000001c0  2e 0f 1f 84 00 00 00 00 00 66 2e 0f 1f 84 00 00 |.....f.....|
000001d0  00 00 00 66 2e 0f 1f 84 00 00 00 00 66 2e 0f |...f.....f..|
000001e0  1f 84 00 00 00 00 00 66 2e 0f 1f 84 00 00 00 |.....f.....|
000001f0  00 66 2e 0f 1f 84 00 00 00 00 0f 1f 00 55 aa |.f.....U.|
00000200
```

Figura 5: salida de la herramienta hd sobre main.img

Nuevamente, los primeros bytes (`be 0f 7c`) coinciden con las instrucciones del código ensamblador. En la salida de la derecha vemos que el mensaje "hello world" empieza al final de la primera línea y continúa en la segunda, así como a la izquierda vemos el mismo string, pero codificado en ASCII (`68 65 6c 6c 6f 20 77 6f 72 6c 64 00`). Al final en la línea 1F0, justo en la posición 0x1FE, aparece `55 aa`, que es la firma que la BIOS busca para considerar válido el sector de arranque.

En ambos dumps, después del código y el mensaje "hello world", se pueden ver muchos bytes adicionales con patrones repetitivos como `66 2e 0f 1f 84`. Estos bytes corresponden a no-op padding patterns conocidos para x86, que se usan para alinear memoria o se insertan automáticamente por el ensamblador o el enlazador para ocupar espacio hasta alcanzar los 512 bytes requeridos.

2.2.2 Generación del binario plano con `--oformat binary`

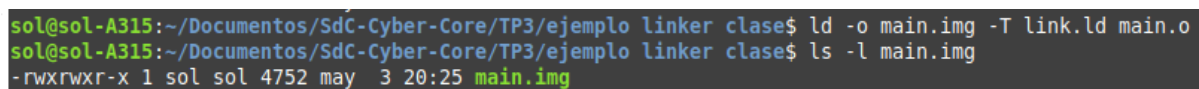
Para generar un archivo binario plano adecuado para ser cargado directamente en memoria por la BIOS, es necesario indicarle al linker que el resultado no debe tener cabecera ni metadatos, sino contener únicamente el código máquina. La opción `--oformat binary` en el linker se utiliza para generar este archivo raw binary..

Un bootloader debe ocupar exactamente los 512 bytes del primer sector del disco, y debe estar en un formato que la BIOS pueda cargar directamente a memoria (en 0x7C00). La BIOS no entiende formatos como ELF o PE, que son comunes para ejecutables en sistemas operativos. Por eso necesitamos un archivo binario puro, y es ahí donde entra `--oformat binary`.

En nuestro caso usamos:

```
ld --oformat binary -o main.img -T link.ld main.o
```

Y como ya vimos anteriormente, el peso del `main.img` generado fue de 512 By. Como verificación, probamos que si no usamos la función de `--oformat binary` tenemos que el peso es 4752 By:



```
sol@sol-A315:~/Documentos/SdC-Cyber-Core/TP3/ejemplo linker clase$ ld -o main.img -T link.ld main.o
sol@sol-A315:~/Documentos/SdC-Cyber-Core/TP3/ejemplo linker clase$ ls -l main.img
-rwxrwxr-x 1 sol sol 4752 may  3 20:25 main.img
```

Figura 6: Verificación de tamaño en bytes de main.img sin -oformat binary.

Además, vemos que el archivo que se genera después de linkear es del tipo ELF (figura 7) por lo que QEMU no lo reconoce como imagen booteable (figura 8).

```
sol@sol-A315:~/Documentos/SdC-Cyber-Core/TP3/ejemplo linker clase$ file main.img
main.img: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), statically linked, with debug_info, not stripped
```

Figura 7: Verificación del tipo de archivo ELF generado sin -oformat binary.

```
sol@sol-A315:~/Documentos/SdC-Cyber-Core/TP3/ejemplo linker clase$ qemu-system-x86_64 -hda main.img
WARNING: Image format was not specified for 'main.img' and probing guessed raw.
Automatically detecting the format is dangerous for raw images, write operations on block 0 will be restricted.
Specify the 'raw' format explicitly to remove the restrictions.

Machine View

Booting from DVD/CD...
Boot failed: Could not read from CDROM (code 0003)
Booting from ROM...
iPXE (PCI 00:03.0) starting execution...ok
iPXE initialising devices...ok

iPXE 1.21.1+git-20220113.fbbdc3926-0ubuntu2 -- Open Source Network Boot Firmware
-- https://ipxe.org
Features: DNS HTTP HTTPS iSCSI NFS TFTP UEFI AUE ELF MBOOT PXE bzImage Menu PXEX
T

net0: 52:54:00:12:34:56 using 82540em on 0000:00:03.0 (Ethernet) [open]
[Link:up, TX:0 TXE:0 RX:0 RXE:0]
Configuring (net0 52:54:00:12:34:56)..... ok
net0: 10.0.2.15/255.255.255.0 gw 10.0.2.2
net0: fec0::5054:ff:fe12:3456/64 gw fe80::2
net0: fe80::5054:ff:fe12:3456/64
Nothing to boot: No such file or directory (https://ipxe.org/2d03e13b)
No more network devices

No bootable device.
-
```

Figura 8: Falla al intentar iniciar la vm con main.img sin -oformat binary.

3.1 Modo protegido

El modo protegido es una característica de los procesadores x86 que permite utilizar segmentación avanzada, paginación y acceso a más de 1 MB de memoria, entre otras funcionalidades propias de sistemas operativos modernos. Para cambiar del modo real al modo protegido, se deben seguir los siguientes pasos:

1. Desactivar las interrupciones mediante la instrucción `cli`, para evitar que una interrupción ocurra durante la transición.
2. Cargar la GDT (Global Descriptor Table) usando `lgdt`, lo cual define los descriptores de segmento que usará el procesador.

3. Setear el bit PE (Protection Enable) en el registro CR0 para habilitar el modo protegido.
4. Realizar un salto lejano (`jmp`) para cargar el nuevo selector de segmento de código y actualizar el valor de CS.

Siguiendo estos requisitos, el código planteado para pasar de modo real a modo seguro fue:

```
[BITS 16]                ; En modo real (16 bits)
[ORG 0x7C00]              ; Dirección de carga estándar del bootloader

start:

    cli                    ; 1. Desactivar interrupciones

    ; 2. Cargar la GDT
    lgdt [gdt_descriptor]

    ; 3. Activar el bit PE en CR0
    mov eax, cr0
    or  eax, 1
    mov cr0, eax

    ; 4. Saltar a modo protegido
    jmp 0x08:protected_mode_start ; salto lejano con selector de código

; -----
; Definimos la GDT
gdt_start:
    dq 0x0000000000000000 ; Descriptor nulo obligatorio
    dq 0x00CF9A000000FFFF ; Descriptor de segmento de código, ejecutable
    dq 0x00CF92000000FFFF ; Descriptor de segmento de datos,
lectura/escritura
gdt_end:

gdt_descriptor:
    dw gdt_end - gdt_start - 1 ; Tamaño de la GDT - 1 byte
    dd gdt_start                ; Dirección base de la GDT

; -----
; Código en modo protegido
[BITS 32]                ; A partir de aca estamos en modo protegido (32 bits)
protected_mode_start:
    mov ax, 0x10          ; Selector del descriptor de datos en la GDT
    mov ds, ax             ; Actualizar segmento de datos
    mov es, ax
    mov fs, ax
    mov gs, ax
    mov ss, ax             ; Segmento de pila

    ; Escribimos una P verde en pantalla para demostrar que estamos en PM
    mov dword [0xB8000], 0x2F50
.halt:
    jmp .halt              ; Bucle infinito para evitar que el código avance

; -----
; Relleno hasta 512 bytes
times 510 - ($ - $$) db 0 ; Rellenar con ceros hasta 510 bytes
dw 0xAA55                 ; Firma de sector de arranque (boot signature)
```

En el código podemos ver como es la forma en la que se realizan los 4 pasos que habíamos definimos necesarios para pasar de modo real a protegido, siendo el final la instrucción `jmp 0x08:protected_mode_start` que realiza el salto lejano usando el selector de código (0x08), cambiando el valor del registro CS y confirma la entrada al modo protegido. Posteriormente, se configuran todos los registros de segmento (ds, es, etc.) con el selector de datos (0x10) y se escribe una letra en la memoria de video para evidenciar visualmente que el salto fue exitoso.

Este código define una GDT con tres descriptores: uno nulo (requerido por convención), uno para código (selector 0x08) y otro para datos (selector 0x10), ambos con base 0x00000000 y límite 0xFFFFF. Esto permite al procesador usar regiones separadas para instrucciones y datos, aprovechando la segmentación del modo protegido para organizar y aislar mejor la memoria.

Si queremos cambiar los bits de acceso al segmento de dato para que sea de solo lectura, debemos cambiar cómo definimos el descriptor de datos, dejando solo lectura (no writable), para esto modificamos la línea :

```
dq 0x00CF92000000FFFF ; Data segment
```

Al valor:

```
dq 0x00CF90000000FFFF ; Data solo lectura
```

Luego, en el modo protegido agregamos un intento de escritura para comprobar el fallo que esto puede generar:

```
mov ax, 0x10
```

```
mov ds, ax
```

```
mov dword [0x00001000], 0x00001234 ; Intento de escribir
```

Analizamos el comportamiento, corriendo qemu en una terminal y gdb en otra, para debugear la .img. Teóricamente, procesador va a generar una excepción general de protección (#GP). Si no hay IDT y no se captura la excepción, el IP salta a una dirección aleatoria y el comportamiento es inesperado. Una vez que entramos en la parte de código en modo protegido podemos ver que efectivamente esto:

```
Output/messages
0x0000e05b in ?? ()
Assembly
0x0000e05b ? add    %al, (%eax)
0x0000e05d ? add    %al, (%eax)
0x0000e05f ? add    %al, (%eax)
0x0000e061 ? add    %al, (%eax)
0x0000e063 ? add    %al, (%eax)
0x0000e065 ? add    %al, (%eax)
0x0000e067 ? add    %al, (%eax)
0x0000e069 ? add    %al, (%eax)
0x0000e06b ? add    %al, (%eax)
0x0000e06d ? add    %al, (%eax)
Breakpoints
[1] break at 0x00007c00 for *0x7c00 hit 1 time
Expressions
```

Figura 9: gdb luego del intento de escritura sin permiso.

Por otro lado, en la ventana de QEMU donde tenemos la imagen corriendo, vemos que cambia constantemente de la imagen que muestra la pantalla, teniendo diferentes salidas inestables, evidenciando el fallo.

3.1.1 Registros de segmento en modo protegido

En modo protegido, los registros de segmento como CS, DS, SS, etc., no contienen direcciones base directamente, sino selectores. Un selector es un valor especial que indica qué descriptor usar dentro de una tabla de descriptors (normalmente la GDT, o Global Descriptor Table).

Cada selector está compuesto por:

- Índice: posición del descriptor dentro de la GDT o LDT.
- Bit TI (Table Indicator): indica si el selector pertenece a la GDT (0) o a la LDT (1).
- Nivel de privilegio (RPL): indica el nivel de privilegio del segmento (de 0 a 3).

```
mov ax, 0x10    ; Selector con índice 2 en la GDT (segmento de datos)
mov ds, ax      ; Carga el segmento de datos usando ese descriptor
```

El procesador utiliza este selector para buscar el descriptor correspondiente en la GDT, y de ahí obtiene la base del segmento, el límite, y los atributos de acceso (como permisos, tipo de segmento, etc.).