TP Sistemas de Computación

Práctico: UEFI

Alumno: Nombre Apellido
Materia: Sistemas de Computación
Comisión: XX

23 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 arranque).

2.1. ¿Cómo puedo usarlo?

2.1.1. 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.

2.1.2. Como desarrollador / cargador de arranque

Al escribir un *bootloader* o sistema operativo *bare-metal* para UEFI:

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>

// GUID estandar para variables globales definidas por UEFI
extern EFI_GUID gEfiGlobalVariableGuid;

EFI_STATUS
LeerBootOrder(VOID)
{
    EFI_STATUS status;
    CHAR16 *VariableName = L"BootOrder";
    UINTN DataSize = sizeof(UINT16) * 16; // espacio para hasta 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;
    }

    // Aqui se puede procesar el contenido de BootOrder...
    return EFI_SUCCESS;
}
```

Listing 1: Lectura de la variable `BootOrder` mediante `GetVariable`

En este ejemplo:

- `GetVariable` es la función que permite acceder al valor de una variable definida por el firmware.
- `gST` es un puntero global a la tabla del sistema UEFI (`EFI_SYSTEM_TABLE`), desde la cual se accede a los *Runtime Services*.
- `gEfiGlobalVariableGuid` identifica el espacio de nombres reservado para variables estándar UEFI.

- `Attributes`, `DataSize` y `BootOrder` son parámetros de salida que reciben respectivamente los atributos, el tamaño en bytes y los datos de la variable.

Para más detalles, se puede consultar la documentación oficial en: UEFI Specification, Runtime Services.

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.