

Sistemas de computación 2022

Tp3:Modo Protegido

Docentes:

Jorge Javier

Solinas Miguel Angel

Grupo: Stack

Sartori Gaston

Tantera Julian

Desafío: UEFI y coreboot

a. ¿Qué es UEFI? ¿cómo puedo usarlo? Mencionar además una función a la que podría llamar usando esa dinámica.

UEFI es una interfaz entre el sistema operativo y el hardware de la plataforma en cuestión. Es la responsable de inicializar el hardware de los dispositivos o equipos antes de dar el control al sistema operativo. De este modo, la interfaz es la responsable de que se cargue un gestor de arranque (bootloader) concreto en la memoria principal, que será el que iniciará las acciones rutinarias de arranque.

Para poder usar la interfaz UEFI, el ordenador necesita disponer de un firmware especial en la placa base. Al encender el ordenador, el firmware utiliza la interfaz UEFI como una capa o layer operativa que actúa de intermediaria entre el mismo firmware y el sistema operativo. Para que el modo UEFI se pueda iniciar antes de que el ordenador en sí haya arrancado realmente, se implementa de forma permanente en la placa base, en un chip de memoria. Así, como parte integral del firmware de la placa base, la programación UEFI se mantiene incluso cuando aún no circula la electricidad.

El acceso a UEFI se realiza habitualmente pulsando una determinada tecla del teclado o una combinación de ellas durante la fase de testeo del hardware o lo que es lo mismo, durante el arranque del equipo que es cuando debemos pulsar para acceder al firmware.

Una función a la que se puede llamar con esta dinámica es dar un orden de arranque de las unidades de almacenamiento. Una gestión imprescindible, por ejemplo, a la hora de instalar/actualizar desde cero el sistema operativo, usar suites de mantenimiento o antivirus que cargamos desde un pendrive o medio óptico. También seleccionar el arranque de varios discos donde podemos tener distintos sistemas instalados.

b. ¿Menciona casos de bugs de UEFI que puedan ser explotados?

Ejemplo: HP UEFI con vulnerabilidades que generan posibles ataques e infecciones de sistemas. Las vulnerabilidades detectadas se dividen en tres grupos según el componente/función que se explota:

1. **SMM Callout (Privilege Escalation):** Llamadas que permiten el escalamiento de nivel de privilegios.
2. **SSM (System Management Module):** heap buffer overflow conduce a la ejecución de código arbitrario, corrupción de memoria conduce a la ejecución de código arbitrario.
3. **DXE (Driver eXecution Environment):** stack buffer overflow que conducen a una ejecución de código arbitrario

c. ¿Qué es Converged Security and Management Engine (CSME), the Intel Management Engine BIOS Extension (Intel MEBx).?

CSME, previamente llamado Intel MEBx, es un subsistema embebido en el chipset de Intel que provee un entorno de ejecución aislado protegido del software host que corre en la CPU principal. Ejecuta el firmware CSME (FW). Cumple tres roles principales:

- Chassis: Arranque seguro de la plataforma, overclocking y carga del micro código en el hardware de la PCH/CPU.
- Seguridad: Ejecución aislada y confiable de servicios de seguridad. (TPM, DRM, DAL)
- Manejabilidad: Administración de la plataforma sobre red fuera de banda. (Intel ATM)

d. ¿Qué es coreboot ? ¿Qué productos lo incorporan ?¿Cuales son las ventajas de su utilización?

Coreboot es un proyecto dirigido a reemplazar el firmware no libre de los BIOS, por un BIOS libre y liviano. Este es diseñado para realizar solamente el mínimo de tareas necesarias para cargar y correr un sistema operativo moderno de 32 bits o de 64 bits para luego pasarle el control a otro programa llamado payload. Es respaldado por la Free Software Foundation (FSF).

Algunos productos que lo incorporan son:

- *Purism*: Purism vende laptops con el foco en seguridad y privacidad del usuario, para lo cual es necesario minimizar la cantidad de código binario y/o propietario.
- *Dispositivos ChromeOS*: Todos los dispositivos ChromeOS lanzados desde el 2012 usan coreboot para el firmware de su sistema principal.
- *Libretrend*: Producen Librebox que contienen firmware coreboot.
- *PC Engines APUs*.

Algunas ventajas que ofrece son:

- Se ha creado con la intención de que realice su cometido en el mínimo de instrucciones posible. Por ejemplo, la versión x86 corre en modo de 32 bits después de ejecutar solamente dieciséis instrucciones.
- Coreboot puede cargar otros núcleos que no sean Linux, o, en lugar de ello, puede pasar el control a un cargador para arrancar un núcleo o imagen
- *Se puede debuggear fácilmente*: hay varias maneras de extraer el log de booteo. Tiene soporte de GDB Stub vía serial.
- *Seguro*: el modo de Recovery está basado en múltiples copias del firmware que pueden ser actualizadas independientemente. Además, la integridad de las etapas y los binarios puede ser verificada fácilmente.

Desafío: Linker

a. ¿Que es un linker? ¿que hace ?

Un linker es un programa que toma los objetos generados en los primeros pasos del proceso de compilación, la información de todos los recursos necesarios (biblioteca), quita aquellos recursos que no necesita, y enlaza el código objeto con su(s) biblioteca(s) con lo que finalmente produce un fichero ejecutable o una biblioteca. En el caso de los programas enlazados dinámicamente, el enlace entre el programa ejecutable y las bibliotecas se realiza en tiempo de carga o ejecución del programa.

La mayoría de las veces se utilizan diversos archivos a la hora de crear un programa. Esto es lo mismo que decir que un programa de ordenador está compuesto por **diferentes módulos**. Cada uno de estos módulos se compila de forma independiente y dispone de una referencia simbólica. Así pues, una vez compilados, el linker podrá crear un «único» archivo, unificando todos los archivos objeto gracias a sus referencias simbólicas.

El enlazador puede tomar la información a través de archivos objeto o de bibliotecas. Además, es capaz de eliminar todos aquellos recursos que no sean necesarios. Finalmente, enlaza toda la información recopilada para poder devolver los ejecutables o las bibliotecas que se quieran crear.

b. ¿Que es la dirección que aparece en el script del linker?¿Porqué es necesaria ?

La dirección "0x7C00" es la dirección de memoria en la que la BIOS carga el MBR (Master Boot Record). El desarrollador del sistema operativo o del cargador de arranque debe asumir que sus códigos de ensamblador están cargados y comienzan desde 0x7C00. De esta manera, es necesario especificar al linker esta dirección de inicio, para que el calculo de las direcciones de los labels se haga de manera correcta en base a esta.

- c. Compare la salida de objdump con hd, verifique donde fue colocado el programa dentro de la imagen.

Utilizando hd, podemos observar el contenido del archivo main.img

```
gaston@gastonnotebook:~/Documents/siscon/tp3/protected-mode-sdc/01HelloWorld$ hd main.img
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 00 |....f.....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 00 |....f.....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 00 |....f.....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 00 |....f.....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 00 |....f.....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.....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 00 |....f.....f...|
000001f0  00 66 2e 0f 1f 84 00 00 00 00 0f 1f 00 55 aa |.f.....U.|
00000200
```

Se puede observar, que dentro del archivo, el programa se encuentra desde la dirección 00. Ya que la especificación de una dirección a la hora de linkear, no refiere a que dentro del archivo el programa comenzará en esa dirección, sino que el cálculo de las referencias a direcciones de memoria se haga a partir de esta.

Con objdump, podemos analizar el archivo main.o:

```
main.o:      file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <loop-0x5>:
  0:  be 00 00 b4 0e          mov     $0xeb40000,%esi

0000000000000005 <loop>:
  5:  ac                    lods    %ds:(%rsi),%al
  6:  08 c0                 or      %al,%al
  8:  74 04                 je      e <halt>
  a:  cd 10                 int     $0x10
  c:  eb f7                 jmp     5 <loop>

000000000000000e <halt>:
  e:  f4                    hlt

000000000000000f <msg>:
  f:  68 65 6c 6c 6f        pushq   $0x6f6c6c65
 14:  20 77 6f              and     %dh,0x6f(%rdi)
 17:  72 6c                 jb      85 <msg+0x76>
 19:  64                    fs
  ...
```

d. Grabar la imagen en un pendrive y probarla en una pc y subir una foto

Se cargó la imagen en un pendrive, y se intentó bootear la pc con esa imagen. El resultado no fue el esperado, ya que pareciera que solo se ejecuta una instrucción halt. Sin embargo, esto puede deberse a que la pc cuenta con un procesador AMD, y no Intel.



e. ¿Para que se utiliza la opción --oformat binary en el linker?

--oformat = formato de salida

ld puede configurarse para admitir más de un tipo de archivo de objeto. Si ld está configurado de esta manera, puede usar la opción --oformat para especificar el formato binario para el archivo de objeto de salida.

Desafío final: Modo protegido

- a. Crear un código assembler que pueda pasar a modo protegido (sin macros).

```
protected_mode.asm
1  [org 0x7c00]
2
3  CODE_SEG equ gdt_code - gdt_start
4  DATA_SEG equ gdt_data - gdt_start
5
6  [bits 16]
7  ;se inicia el proceso de paso a modo protegido
8  cli ;se deshabilitan las interrupciones
9
10 lgdt [gdt_descriptor] ;se carga la GDT, cargando en el registro GDTR, el descriptor de la GDT
11
12 mov eax, cr0           ;se setea el bit 0 del CR0 en 1
13 or eax, 0x1
14 mov cr0, eax
15
16 jmp CODE_SEG : protected_mode ;se salta a la seccion de codigo de 32 bits
17
18 ;GDT
19 gdt_start:
20     gdt_null:           ;descriptor nulo
21         dd 0x0
22         dd 0x0
23
24 gdt_code:               ;descriptor del segmento de codigo
25     ;base: 0x00000000
26     ;limite: 0xffff
27     ;1st double word
28     dw 0xFFFF           ;bits 0-15: parte baja del limite, en este caso, 0xffff
29     dw 0x0               ;bits 16-31: parte baja de la base, en este caso 0x0000
30     ;2nd double word
31     db 0x0               ;bits 0-7: bits 16-23 de la base, en este caso 0x00
32     db 10011010b         ;flags: 10011010
33                           ;bit 8: bit de presencia, lo setea la CPU al acceder, 0
34                           ;bit 9: como es de codigo, setea si es leible o no, 1
35                           ;bit 10: bit conforming, 0
36                           ;bit 11: especifica si es de codigo (1) o data (0), 1
37                           ;bit 12: bit S, seteado para segmentos normales (codigo, dato o pila), 1
38                           ;bit 13-14: bits de nivel de privilegio, nivel 0, el mas alto, 00
39                           ;bit 15: bit de presencia, seteado en 1.
40     db 11001111b         ;flags + limit: 11001111
41                           ;bits 16-19: parte alta del limite, en este caso, 0xf, 1111
42                           ;bits 20: reservado para programador de sistemas, se ignora, 0
43                           ;bits 21: reservado por intel, 0
44                           ;bit 22: bit de size, indica que tendremos codigo de 32 bits, 1
45                           ;bit 23: bit de granularidad, seteado los segmentos tienen limite de 4gb, 1
46     db 0x0               ;bit 24-31: parte alta de la base, en este caso 0x00
```



```

47 gdt_data:                                ;descriptor del segmento de data
48                                         ;base: 0x00000000
49                                         ;limite: 0xfffff
50                                         ;1st double word
51 dw 0xFFFF                                ;bits 0-15: parte baja del limite, en este caso, 0xffff
52 dw 0x0                                    ;bits 16-31: parte baja de la base, en este caso 0x0000
53                                         ;2nd double word
54 db 0x0                                    ;bits 0-7: bits 16-23 de la base, en este caso 0x00
55 db 10010010b                             ;flags: 10010010
56                                         ;bit 8: bit de presencia, lo setea la CPU al acceder, 0
57                                         ;bit 9: como es de data, setea si es escribible o no, 1
58                                         ;bit 10: bit conforming, 0
59                                         ;bit 11: especifica si es de codigo (1) o data (0), 0
60                                         ;bit 12: bit S, seteado para segmentos normales (codigo, dato o pila), 1
61                                         ;bit 13-14: bits de nivel de privilegio, nivel 0, el mas alto, 00
62                                         ;bit 15: bit de presencia, seteado en 1.
63 db 11001111b                             ;flags + limit: 11001111
64                                         ;bits 16-19: parte alta del limite, en este caso, 0xf, 1111
65                                         ;bits 20: reservado para programador de sistemas, se ignora, 0
66                                         ;bits 21: reservado por intel, 0
67                                         ;bit 22: bit de size, indica que tendremos codigo de 32 bits, 1
68                                         ;bit 23: bit de granularidad, seteado los segmentos tienen limite de 4gb, 1
69 db 0x0                                    ;bit 24-31: parte alta de la base, en este caso 0x00
70 gdt_end:
71 ;GDT descriptor, especifica tamaño de la GDT y direccion de inicio
72 gdt_descriptor:
73     dw gdt_end - gdt_start - 1
74     dd gdt_start
75 [bits 32]
76 ;inicializar los registros de segmento para actualizar cache
77 protected_mode:
78     mov ax, DATA_SEG
79     mov ds, ax
80     mov ss, ax
81     mov es, ax
82     mov fs, ax
83     mov gs, ax
84     ;inicilizar stack
85     mov ebp, 0x7000
86     mov esp, ebp
87     ;ya se esta en modo protegido
88     ;probar escribir en una posicion del segmento de dato, desplazamiento 0x1
89     mov ax, 0x4f
90     mov bx, 0x1
91     mov [bx], ax
92 ; Rellenamos el Bootsector
93 times 510 - ( $ - $$ ) db 0
94 dw 0xaa55

```

En este caso, se cuenta con 2 segmentos, uno de codigo y uno de datos, donde ambos tienen:

- Base= 0x00000000
- Limite= 0xfffff

Es decir, que ambos segmentos ocupan toda la memoria.

En ese caso, podemos intentar escribir en algún del segmento de datos, por ejemplo en la posición 0x1 del mismo. Como el segmento es escribible, esto se puede realizar correctamente:

b. ¿Cómo sería un programa que tenga dos descriptors de memoria diferentes, uno para cada segmento (código y datos) en espacios de memoria diferenciados?

Para este caso, los segmentos de código y datos deben estar en parte de memoria diferentes, para esto se puede intentar asignar la mitad de memoria a uno y la mitad de memoria al otro. De esta manera, los segmentos tendrán:

Segmento de código:

- Base= 0x00000000
- Limite= 0x7ffff

Segmento de dato:

- Base= 0x00080000
- Limite= 0x7ffff

```
;GDT
gdt_start:
  gdt_null:                ;descriptor nulo
  dd 0x0
  dd 0x0

  gdt_code:                ;descriptor del segmento de codigo
                           ;base: 0x00000000
                           ;limite: 0x7ffff
  ;1st double word
  dw 0xFFFF                ;bits 0-15: parte baja del limite, en este caso, 0xffff
  dw 0x0                    ;bits 16-31: parte baja de la base, en este caso 0x0000
  ;2nd double word
  db 0x0                    ;bits 0-7: bits 16-23 de la base, en este caso 0x00
  db 10011010b              ;flags: 10011010
                           ;bit 8: bit de presencia, lo setea la CPU al acceder, 0
                           ;bit 9: como es de codigo, setea si es leible o no, 1
                           ;bit 10: bit conforming, 0
                           ;bit 11: especifica si es de codigo (1) o data (0), 1
                           ;bit 12: bit S, seteado para segmentos normales (codigo, dato o pila), 1
                           ;bit 13-14: bits de nivel de privilegio, nivel 0, el mas alto, 00
                           ;bit 15: bit de presencia, seteado en 1.
  db 11000111b              ;flags + limit: 11001111
                           ;bits 16-19: parte alta del limite, en este caso, 0xf, 1111
                           ;bits 20: reservado para programador de sistemas, se ignora, 0
                           ;bits 21: reservado por intel, 0
                           ;bit 22: bit de size, indica que tendremos codigo de 32 bits, 1
                           ;bit 23: bit de granularidad, seteado los segmentos tienen limite de 4gb, 1
                           ;bit 24-31: parte alta de la base, en este caso 0x00
  db 0x0
```

```

gdt_data:                                ;descriptor del segmento de data
                                           ;base: 0x00080000
                                           ;limite: 0x7ffff

                                           ;1st double word
dw 0xFFFF                                ;bits 0-15: parte baja del limite, en este caso, 0xffff
dw 0x0                                    ;bits 16-31: parte baja de la base, en este caso 0x0000

                                           ;2nd double word
db 0x08                                  ;bits 0-7: bits 16-23 de la base, en este caso 0x08
db 10010010b                             ;flags: 10010010
                                           ;bit 8: bit de presencia, lo setea la CPU al acceder, 0
                                           ;bit 9: como es de data, setea si es escribible o no, 1
                                           ;bit 10: bit conforming, 0
                                           ;bit 11: especifica si es de codigo (1) o data (0), 0
                                           ;bit 12: bit S, seteado para segmentos normales (codigo, dato o pila), 1
                                           ;bit 13-14: bits de nivel de privilegio, nivel 0, el mas alto, 00
                                           ;bit 15: bit de presencia, seteado en 1.
db 11000111b                             ;flags + limit: 11001111
                                           ;bits 16-19: parte alta del limite, en este caso, 0xf, 1111
                                           ;bits 20: reservado para programador de sistemas, se ignora, 0
                                           ;bits 21: reservado por intel, 0
                                           ;bit 22: bit de size, indica que tendremos codigo de 32 bits, 1
                                           ;bit 23: bit de granularidad, seteado los segmentos tienen limite de 4gb, 1
                                           ;bit 24-31: parte alta de la base, en este caso 0x00
db 0x0

gdt_end:

;GDT descriptor, especifica tamaño de la GDT y direccion de inicio
gdt_descriptor:
dw gdt_end - gdt_start - 1
dd gdt_start

```

En ese caso, podemos intentar escribir en algún del segmento de datos, por ejemplo en la posición 0x1 del mismo. Como el segmento es escribible, esto se puede realizar correctamente:

```

>>> x/x 0x0080001
0x80001:      0x0000004f

```

Podemos observar como en este caso, escribiendo en la posición 0x1 del segmento de dato, realmente se escribe en la dirección 0x80001, ya que la base del segmento de datos se encuentra en 0x80000, por lo que se realiza un desplazamiento de 0x1 a partir de la misma.

- c. **Cambiar los bits de acceso del segmento de datos para que sea de solo lectura, intentar escribir, ¿Que sucede? ¿Que debería suceder a continuación? (revisar el teórico) Verificarlo con gdb.**
- d. **En modo protegido, ¿Con qué valor se cargan los registros de segmento ? ¿Porque?**

En modo protegido, en los registros de segmento se cargan los selectores de segmento, los cuales apuntan al descriptor del segmento deseado a través de un INDICE, dentro de la GDT o LDT. Es decir, se cargan con el offset dentro de la GDT donde se encuentra la información del segmento deseado.