

Reversing

Tipos de binarios ejecutables

- Binarios de cargador
 - ELF, EXE, EFI, PE, COFF...
- Binarios raw

Ejecutables de cargador

- https://en.wikipedia.org/wiki/Comparison_of_executable_file_formats
- https://en.wikipedia.org/wiki/Executable_and_Linkable_Format
- Son archivos binarios ejecutables
- Contienen tablas con información:
 - Arquitectura
 - Sistema operativo
 - Endianness
 - ...
- Memoria con acceso controlado/gestionado
- Para acceder al hardware hablan con una librería estándar (libc) y el Kernel

Cargador o loader

- Es un programa
- Cuando queremos ejecutar un ejecutable, se llama al loader
- El loader lee el contenido del ejecutable y lo carga en memoria
- El loader mira las tablas de dependencias y se encarga de cargar librerías y otros componentes necesarios en memoria
- El loader etiqueta algunas zonas de memoria como R/W/X
- Cuando todo está listo, el loader cede la ejecución al punto de entrada del ejecutable

Binarios RAW

- Son instrucciones puras
- No hay tablas con información sobre arquitectura, endianness...
- No hay gestión de dependencias externas
- Acceso a memoria directo sin gestión o intervención de un kernel...
- Hablan con el hardware directamente
- Para ejecutarlos, otro trozo de código tiene que saltar a la dirección de memoria donde se encuentre este archivo...

Binarios de cargador vs RAW

- Los binarios de cargador incluyen más información en sus cabeceras
- Esto los suele hacer más fáciles de reversear
- Empezaremos por lo sencillo... :)

Ejercicio 1

- Asegurarnos de que podemos abrir Ghidra
- Asegurarnos de que hemos instalado el plugin Ghidra Findcrypt
- Abrir Ghidra!

Ejercicio 2

- Abrir Ghidra
- Crear un proyecto
- Seleccionar un binario del userspace que queremos analizar
- Importar el binario a Ghidra
- Abrir la herramienta CodeBrowser
- Analizar el ejecutable

Estrategias de reversing

- Buscar el entry point, analizar siguiendo el flujo
 - Funciona bien en binarios pequeños y con poco código
 - Da una visión completa de lo que hace el binario
- Buscar puntos interesantes en base a strings o en base a constantes conocidas como constantes criptográficas
 - Permite un análisis más localizado
 - Visión parcial del binario
 - Podemos no encontrar lo que buscamos si el análisis de Ghidra no ha sido bueno
- Buscar funciones del sistema y etiquetarlas
 - Se pueden identificar porque usan strings de formatos (familia printf) y porque suelen ser de las más referenciadas (memcpy, strlen...)
 - Es tedioso pero da resultados cuando no tenemos muchas otras opciones

Ejercicio 3

- Localizar el punto de entrada del ejecutable
- Localizar la función “main”
- Debatir sobre qué es el código entre el punto de entrada y la función main...
¿Para que sirve?

Ejercicio 4

- Localizar strings interesantes
- Localizar en qué funciones se usan esos strings
- Debatir sobre porque es posible que Ghidra no encuentre todos los strings...

Ejercicio 4

- Localizar constantes criptográficas
- Localizar en qué funciones se usan
- Debatir sobre posibles algoritmos en uso...

Ejercicio 5

- No se permite la ejecución del binario! Debería tratarse como si fuese un trozo de malware!
- Describir el funcionamiento global del ejecutable
- Descubrir la contraseñas del ejecutable
- Descubrir el output final del ejecutable cuando se inserta la contraseña adecuada

Binarios RAW

- Necesitamos recopilar alguna información previa a su estudio...

Ejercicio 6

- Abrir nuestro proyecto de Ghidra
- Tratar de importar el segundo firmware!

Binarios RAW

- El firmware de ejemplo 2 es un binario RAW que compone todo el software que ejecuta un chip nrf52840 de Nordic Semiconductor.
- Ghidra nos pide la arquitectura para poder analizar el binario...
- Estrategias de detección:
 - Búsqueda de información del fabricante
 - Heurísticos:

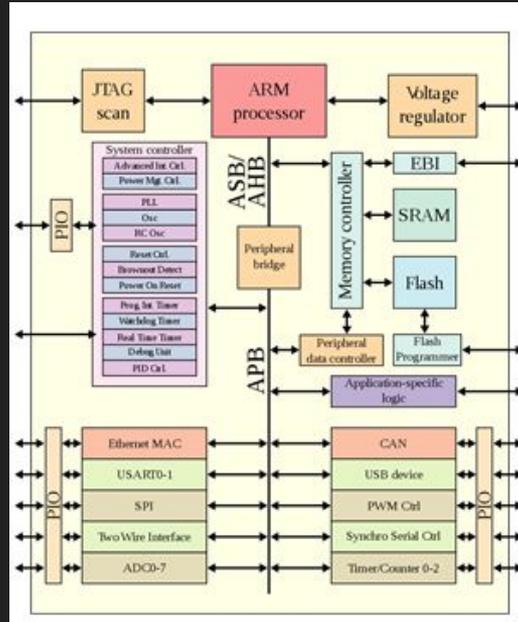
```
binwalk -Y fichero.bin (requiere tener capstone instalado)
```

Ejercicio 6

- Encontrar la arquitectura de nuestro binario
- Ahora sí, importarlo en Ghidra
- Abrir el archivo en el code browser
- Realizar un análisis
- Debatir:
 - ¿Nos sirve la estrategia de análisis del entry point?
 - ¿Y la de los strings?
 - ¿Y la de las constantes criptográficas?
 - ¿Y la de las funciones de sistema?

Arquitectura de una CPU

<https://commons.wikimedia.org/w/index.php?curid=2866881>



Arquitectura de una CPU

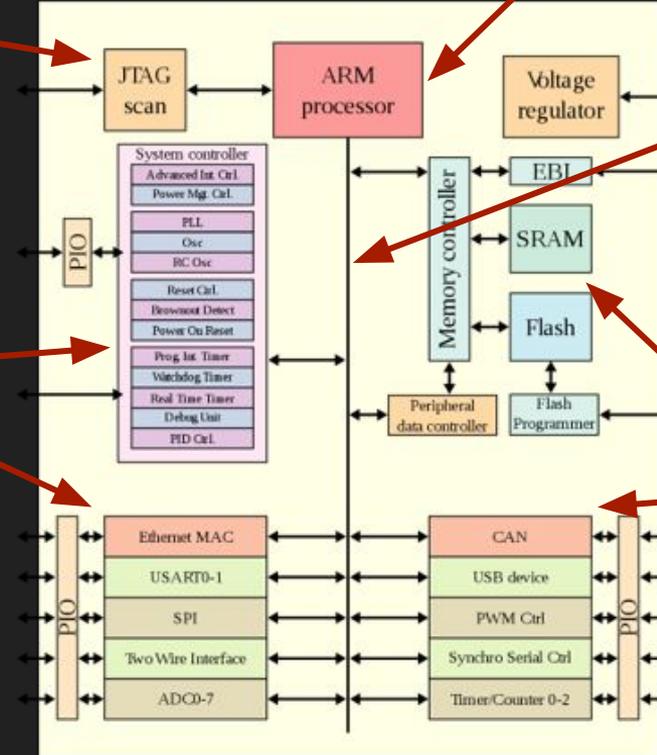
Core

Interfaces de depuración

Bus de memoria

Periféricos

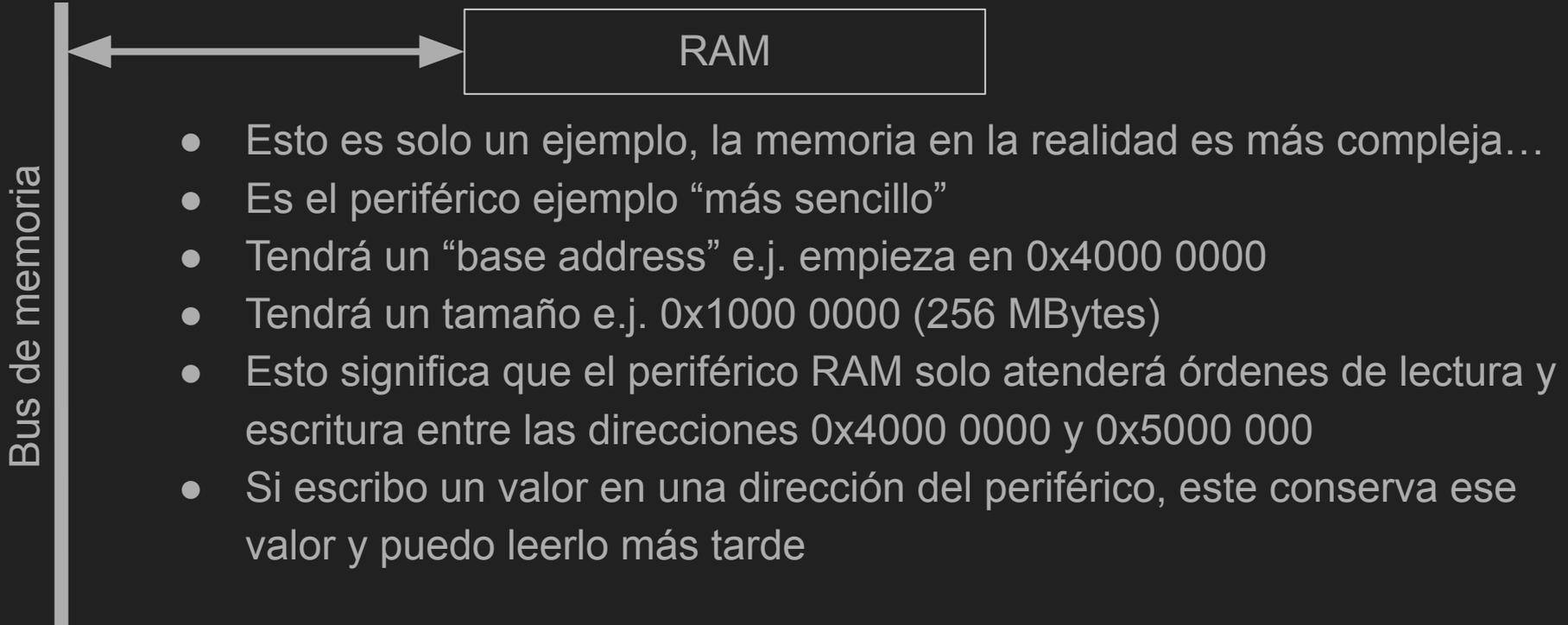
Periféricos



Arquitectura de una CPU

- Tenemos un CORE que realiza operaciones: sumar, restar, leer, escribir...
- Ese core intercambia datos con el exterior a través de un BUS DE MEMORIA...
- En ese bus de memoria se conectan distintos PERIFÉRICOS que se identifican por una dirección...

Arquitectura de una CPU



Arquitectura de una CPU

- Existen periféricos con comportamientos más complicados
- Los más comunes son: Timers, UARTs, GPIO, SPI, System controller
- Cada chip o SoC tendrá unos periféricos diferentes en posiciones de memoria diferentes...

Estrategias de reversing

- Buscar información sobre los periféricos!
 - Muchos fabricantes buscan dar soporte al desarrollador y publican “Datasheets”!

https://infocenter.nordicsemi.com/topic/ps_nrf52840/memory.html?cp=5_0_0_3_1_2#memorymap

Ejercicio 6

- Encontrar el datasheet de nuestro nrf52840
- Localizar el memory map en el datahseet (core components > memory)
- Escoger un periférico de interés para nosotros...
- Crear una sección de memoria en Ghidra para este periférico

Estrategias de reversing

- Meter la info sobre los periféricos a mano es demasiado trabajo...
- Algunos fabricantes publican archivos SVD:
 - Estos son archivos que un debugger puede leer para saber cuales son las secciones de memoria en un microprocesador...
- Si el debugger puede, nosotros también...

Ejercicio 7

- Asegurarse de que tenemos instalado Ghidra SVD
- Localizar el SVD para nuestro Nordic nrf52840
(<https://github.com/NordicSemiconductor/nrfx/blob/master/mdk/nrf52840.svd>)
- BORRAR LA SECCIÓN DE MEMORIA QUE HEMOS CREADO ANTES (o nos dará conflicto con la que va a crear el plugin...)
- File > Load SVD...
- Se recomienda volver a ejecutar un análisis...
- Ir a ver el mapa de memoria :)
- Ir a ver los “namespaces”...

Estrategias de reversing

- Ghidra SVD no es perfecto!
- Cuando hagamos reversing seguiremos teniendo que consultar el “datasheet” si lo tenemos...
- También se dará el caso en el que no tengamos SVD y que incluso los binarios sean más complicados...
- Vamos a por el firmware de ejemplo #3!
- Esto es un bootloader de una cámara ultra low cost con un chip de HiSilicon hi3516ev200

Ejercicio 8

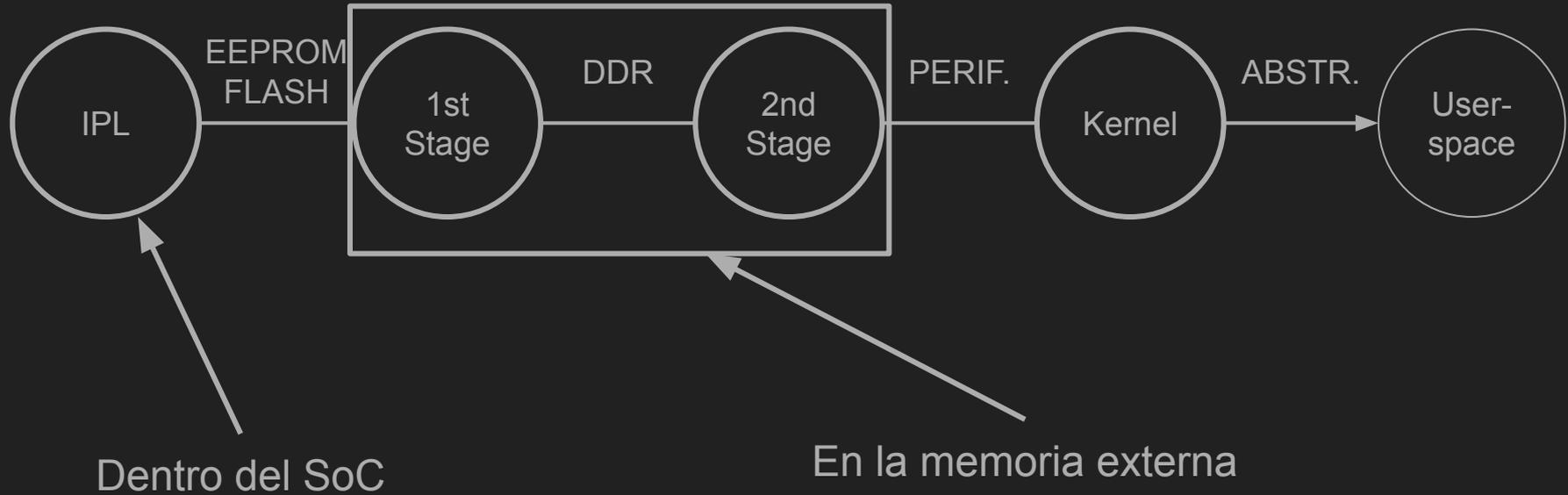
- Importar el bootloader en Ghidra
- Averiguar la arquitectura!
- Abrir el code browser y realizar un primer análisis
- Debate, ¿Qué estrategias tenemos? ¿Cuáles podrían ser las más útiles?:
 - Buscar el entry point
 - Buscar uso de strings
 - Buscar constantes criptográficas
 - Funciones de sistema
 - Archivo SVD
 - Conocer los periféricos y reversear por intuición

Estrategias de reversing

- La estrategia de strings no funciona en este caso porque no hay referencias a los strings! No sabemos donde se usan!
- Además los binarios RAW no tienen una estructura común, pero esto no quiere decir que no exista estructura...
- Hagamos un repaso del proceso de boot...

El proceso de arranque

BOOTLOADER



El proceso de arranque

- Durante el boot, el IPL transfiere la ejecución al bootlader
- Es decir, el SoC cede ejecución a un trozo de código que está fuera del SoC, pero el SoC es complejo!
- ¿Qué hace el SoC cuando alguien divide entre cero? ¿Que hace el SoC cuando un programa lee o escribe en una dirección que no existe?

IVT/ISR table

- Es una pequeña estructura de datos
- Suele estar al principio de todo
- Contiene información acerca de qué hacer cuando se den algunos eventos de sistema
- Es dependiente de la arquitectura! Cada chip tendrá un formato distinto!
- En ARM hay dos tipos de IVT, una para los ARM Cortex M más sencilla y otra para los ARM “tochos” (A y R).
- Ghidra ya conoce la estructura de las IVT en ARM y lo ha etiquetado en rosa en las primeras instrucciones de nuestro bootloader...

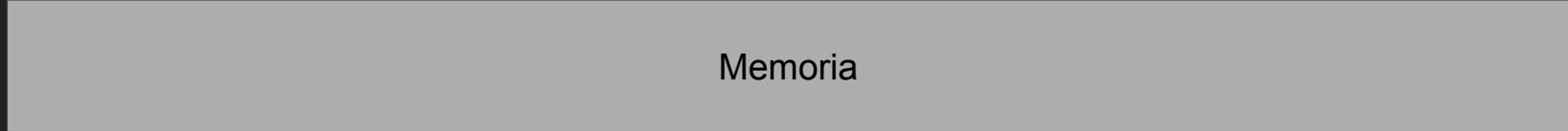
Ejercicio 9

- Analizar la IVT de nuestro bootloader!
- Vamos a tener que leer un poco de ensamblador entre todos...
- ¿Qué está mal en nuestra IVT?

Más sobre el proceso de arranque...

0x0000 0000

0xFFFF FFFF



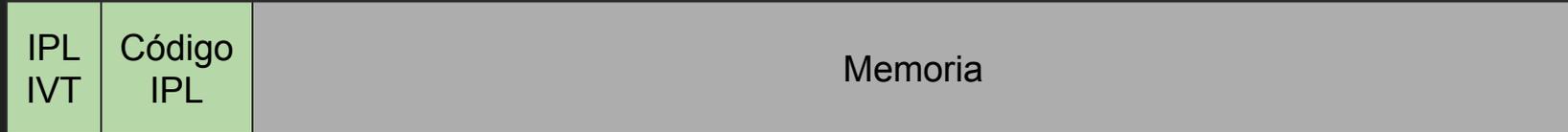
Memoria

Antes del arranque

Más sobre el proceso de arranque...

0x0000 0000

0xFFFF FFFF

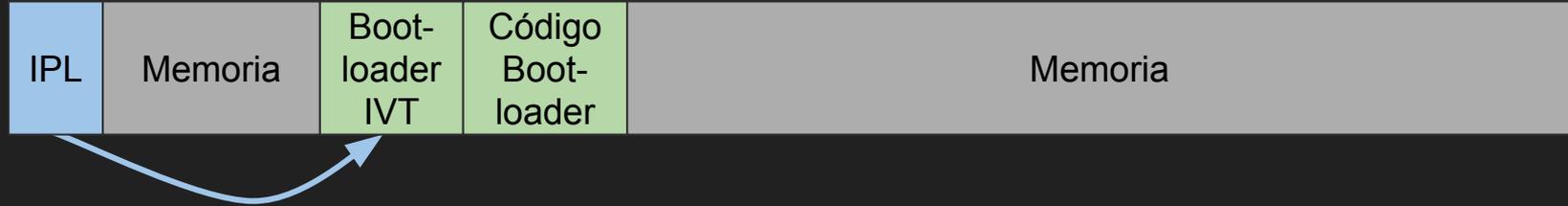


En el momento del arranque se
se ejecuta el código del IPL
dentro del SoC

Más sobre el proceso de arranque...

0x0000 0000

0xFFFF FFFF



El IPL carga parte del bootloader en memoria SRAM. Esto se carga normalmente en una dirección que no es 0x0000 0000

Estrategias de reversing

- Ghidra piensa que nuestro binario está pensado para ser cargado en 0x0000 0000 pero esto no es así...
- Debemos asegurarnos de que nuestro binario está en la dirección correcta!
- Desconocemos la dirección real pero hay pistas en nuestra IVT
- Una vez el binario esté en el lugar correcto podemos verificar que los strings tienen referencias.

Ejercicio 10

- Analizar la IVT de nuestro bootloader y extraer la dirección base real!
- Mover nuestro binario a la dirección correcta.
- Volver a analizar.
- Verificar que los strings ahora si tienen referencias!

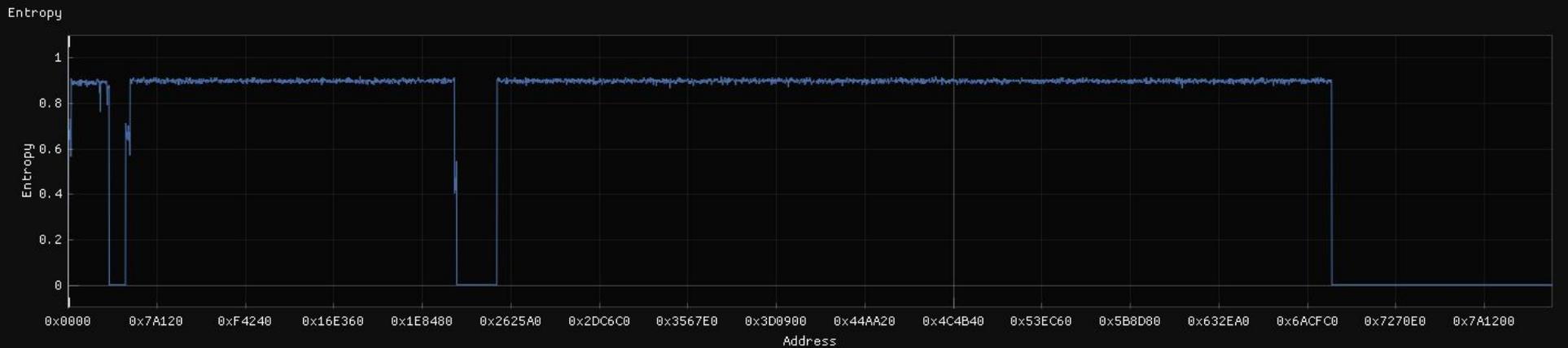
Estrategias de reversing

- Ya podemos usar nuestra estrategia más productiva para este caso: ver referencias a strings.
- A pesar de ello, sigue siendo un trabajo bastante tedioso porque no conocemos la estructura de periféricos...

Sobre el kernel de Linux...

- En sistemas complejos como x86 existe ACPI, que le sirve al Kernel para saber qué dispositivos están presentes en el hardware...
- En sistemas más sencillos, se compila un kernel para un hardware concreto y se le añade un FDT (Flattened Device Tree) o DTB (Device Tree Blob), un archivo binario que le dice que dispositivos hay y dónde están ubicados en memoria...
- Se trata de un archivo binario que podemos convertir a formato legible...

Entropía del firmware hi3516ev200



Extraemos el kernel del firmware

```
% dd if=firmware.bin of=kernel.bin bs=1 skip=327680  
count=1870532
```

```
1870532+0 records in
```

```
1870532+0 records out
```

```
1870532 bytes (1,9 MB, 1,8 MiB) copied, 7,69875 s, 243 kB/s
```

```
dd if=firmware.bin of=kernel.bin bs=1 skip=327680  
count=1870532 0,42s user 7,18s system 98% cpu 7,701 total
```

Binwalk del archivo del kernel

```
% binwalk kernel.bin
```

DECIMAL	HEXADECIMAL	DESCRIPTION
0	0x0	uImage header, header size: 64 bytes, header CRC: 0x7636BDC9, created: 2024-03-02 14:22:00, image size: 1870532 bytes, Data Address: 0x40008000, Entry Point: 0x40008000, data CRC: 0x33756C7, OS: Linux, CPU: ARM, image type: OS Kernel Image, compression type: none, image name: "Linux-4.9.37-hi3516ev200"
64	0x40	Linux kernel ARM boot executable zImage (little-endian)
23503	0x5BCF	xz compressed data
23837	0x5D1D	xz compressed data
1857312	0x1C5720	Flattened device tree, size: 13284 bytes, version: 17

Extraemos el archivo DTB del kernel

```
% dd if=kernel.bin of=kernel.dtb bs=1 skip=1857312  
count=13284
```

```
13220+0 records in
```

```
13220+0 records out
```

```
13220 bytes (13 kB, 13 KiB) copied, 0,0619385 s, 213 kB/s
```

```
% pip install pyfdt
```

```
...
```

```
% dtbdump.py .\kernel.dtb out.txt
```

```
...
```

```
% cat out.txt
```

```
...
```

Estrategias de reversing

- Podemos leer el DTB y crear secciones de memoria en Ghidra para nuestros periféricos...
- Pero como eso es demasiado trabajo, tenemos el plugin Ghidra DTB!

Ejercicio 11

- Cargar el archivo DTB en Ghidra
- File > Load DTB...
- Buscar referencias al periférico UART
- Reversear las funciones principales de la UART!

Estrategias de reversing

- Última estrategia de reversing!
- Intentar buscar código fuente similar al que estamos reverseando.
- U-Boot tiene código fuente disponible
- Buscar la versión de U-Boot en los strings e intentar conseguir el código de la misma versión
- Muchos fabricantes lo customizan por lo que no será igual pero será una buena referencia
- <https://github.com/OpenIPC/u-boot-hi3516ev200/>