Introducción al reversing con Ghidra

Hardware Hacking Day 2024

Ejercicio 1

- Asegurarnos de que podemos abrir Ghidra
  - https://github.com/NationalSecurityAgency/ghidra/releases
  - La última versión es 11.1.1 pero nos vale cualquiera >10
  - Necesitamos JDK >17 y <21!
  - Hay que ejecutar ghidraRun[.bat]

Ghidra

- Ghidra se organiza en proyectos!
- No se puede copiar el proyecto y abrir en otro PC… Tenemos que archivar el proyecto en “File > Archive project…”
- Trabajaremos con un proyecto “no compartido” pero Ghidra soporta un modo “compartido” con un servidor que funciona similar a Git para colaborar…

Ghidra

- Dentro del proyecto podemos importar archivos y organizarlos en carpetas
- Esto es interesante cuando un binario usa una librería y queremos verlos en conjunto, podemos importar ambos y decirle a Ghidra que están relacionados…

Ejercicio 2

- Crear un proyecto “no compartido” llamado “HardwareHackingDay2024”
- Importar un binario que queramos analizar al proyecto

Reversing

- Existen varias estrategias
  - Análisis estático: Miramos el código y tratamos de deducir que hace sin ejecutarlo
  - Análisis dinámico: Se instrumenta el código (con un debugger o un emulador) y se observa como evoluciona durante su ejecución para deducir que hace

Ghidra

- Existen muchas herramientas para trabajar con binarios dentro de Ghidra
  - CodeBrowser - Análisis estático
  - Debugger - Análisis dinámico
  - Emulator - Análisis dinámico
  - …

Ghidra

- CodeBrowser
  - Permite abrir archivos binarios y visualizarlos con múltiples utilidades
  - Hace uso de “analizadores” para clasificar e interpretar los datos de manera automática
  - Todo aquello que los analizadores no analizan, podemos interpretarlo y clasificarlo a mano…

Ejercicio 3

- Abrir el binario con la herramienta CodeBrowser
- Analizar el ejecutable con los analizadores por defecto
- Ver para qué sirve cada una de las utilidades:
  - Bytes
  - Listing
  - Decompile
  - Symbol tree
  - Strings

Estrategias de reversing estático

- Buscar el entry point, analizar siguiendo el flujo de ejecución
  - Funciona bien en binarios pequeños y con poco código
  - Da una visión completa de lo que hace el binario

Ejercicio 4

- 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 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?
  - https://refspecs.linuxbase.org/LSB_3.1.1/LSB-Core-generic/LSB-Core-generic/baselib---libc-start-main-.html

Ejercicio 5

- Realizar un análisis de la función main hasta que la complejidad aumente demasiado

Estrategias de reversing estático

- 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 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
  - Existen plugins con bases de datos que permiten automatizar esto pero por su sencillez recomiendo hacerlo a mano hasta que ganemos confianza…

Ejercicio 6

- En nuestro binario las funciones de sistema ya están etiquetadas. Explicar el porqué de esto.
- Averiguar cuántas funciones de sistema hay en el top 5 de funciones más usadas.

Estrategias de reversing estático

- 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 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
  - Existen plugins con bases de datos que permiten automatizar esto pero por su sencillez recomiendo hacerlo a mano hasta que ganemos confianza…
- 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

Ghidra

- Existen muchos plugins de análisis que nos permiten mejorar mejorar el rendimiento
- Los plugins se gestionan en la ventana de proyecto de Ghidra en “File > Install extensions…”
- En el diálogo de “Install Extensions” tenemos un botón “+” que nos permite cargar zips de extensiones

Ejercicio 7

- Instalar Ghidra Findcrypt
- Volver a lanzar el análisis del binario. Podemos clicar solo en Findcrypt o volver a realizarlo por completo.
- Encontrar constantes criptográficas. ¿A qué algoritmo corresponden? ¿Hay strings que lo confirmen?
- Comentad strings interesantes que hayan aparecido.
- ¿Ha encontrado Ghidra todos los strings?

Ejercicio 8

- Describir el funcionamiento global del ejecutable
- Descubrir la contraseñas del ejecutable
- Descifrar el archivo cifrado
- Descubrir el output final del ejecutable cuando se inserta la contraseña adecuada

Ejercicio 8

- Describir el funcionamiento global del ejecutable
  - Se descifra el fichero binario usando AES con la contraseña 01020304…0f y el iv cafecafe…cafe. Solución en Cyberchef.
  - El programa verifica si se introduce la contraseña “Congratulations!:)”
  - El servicio nos redirige a https://www.youtube.com/watch?v=mCdA4bJAGGk

Ejercicio

- Desc
  - S
    co
    S
  - E
    “C
  - E
    ht

Gk

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
- Los binarios de cargador suelen usar librerías externas, esto reduce su tamaño y da información
- Los binarios RAW interactúan directamente con el hw por lo que suelen incluir drivers o lecturas y escrituras en memoria de la que desconocemos su uso…
- Los binarios de cargador suelen ser más sencillos de reversear…

Ejercicio 8

- Abrir el memory map de nuestro archivo y explorarlo
- Debatir sobre para qué sirve cada sección

Binarios RAW

- El firmware de ejemplo 2 es un binario RAW.
- Este binario que compone todo el software que ejecuta un chip nrf52840 de Nordic Semiconductor.

Ejercicio 9

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

Ejercicio 9

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

Detección de arquitectura

- Estrategias de detección de arquitectura:
  - Búsqueda de información del fabricante
  - Heurísticos:
    - Usamos un conversor de Intel HEX para pasarlo a binario
    - Instalamos binwalk, capstone y python-capstone

```
binwalk -Y fichero.bin
```

Ejercicio 10

- Identificar la arquitectura
- Importar el segundo firmware!
- 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?

Ejercicio 10

- Identificar la arquitectura
  - ARM® Cortex®-M4 Little endian
- Debatir
  - Solo hay un string y no hay constantes criptográficas… Poco que sacar de ahí
  - Funciones de sistema se podría intentar pero requiere un gran esfuerzo y no se usan mucho en este firmware…
  - Podemos analizar el entry point a ver que sucede…

IVT/ISR table

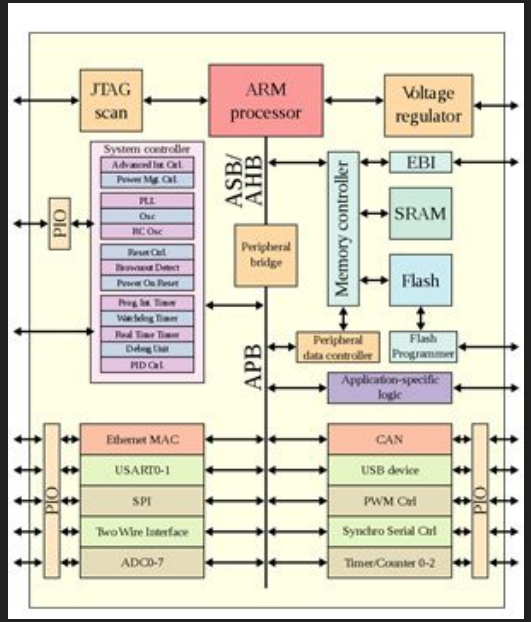
- Es una pequeña estructura de datos
- Suele estár al principio de todo
- Contiene información acerca de qué hacer cuando se den algunos eventos de sistema
- Es dependiente de la arquitectura! Cada chip tendrá un formato distinto!
- En ARM hay dos tipos de IVT, una para los ARM Cortex M más sencilla 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 firmware…

Ejercicio 11

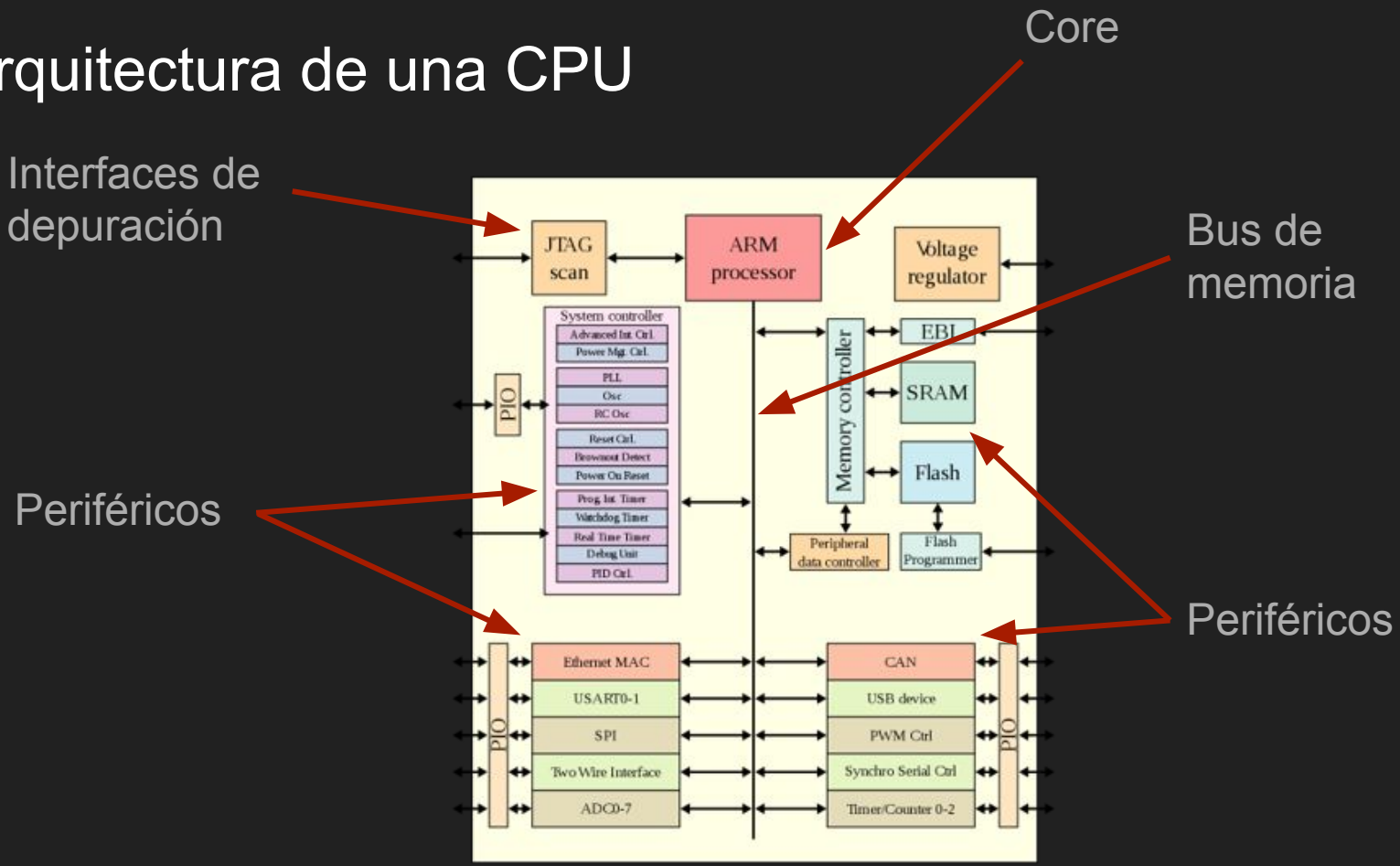
- Repasar el memory map, ver como cambia el código al cambiar sus propiedades
- Revisar el IVT y ver hasta donde podemos llegar…

Arquitectura de una CPU

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



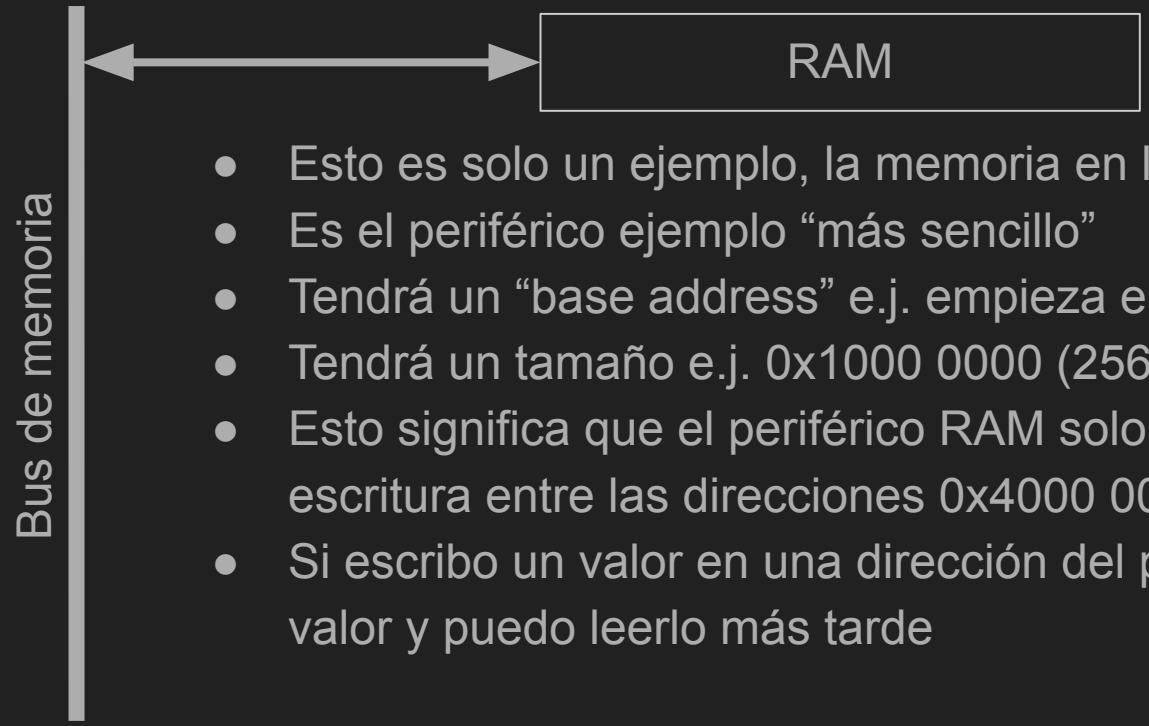
Arquitectura de una CPU



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



- Esto es solo un ejemplo, la memoria en la realidad es más compleja…
- Es el periférico ejemplo “más sencillo”
- Tendrá un “base address” e.j. empieza en 0x4000 0000
- Tendrá un tamaño e.j. 0x1000 0000 (256 MBytes)
- Esto significa que el periférico RAM solo atenderá órdenes de lectura y escritura entre las direcciones 0x4000 0000 y 0x5000 0000
- Si escribo un valor en una dirección del periférico, este conserva ese valor y puedo leerlo más tarde

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”!

Ejercicio 11

- Encontrar el datasheet de nuestro nrf52840: https://infocenter.nordicsemi.com/topic/ps_nrf52840/memory.html?cp=5_0_0_3_1_2#memorymap
- 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 12

- 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”...

Binarios RAW

- 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…

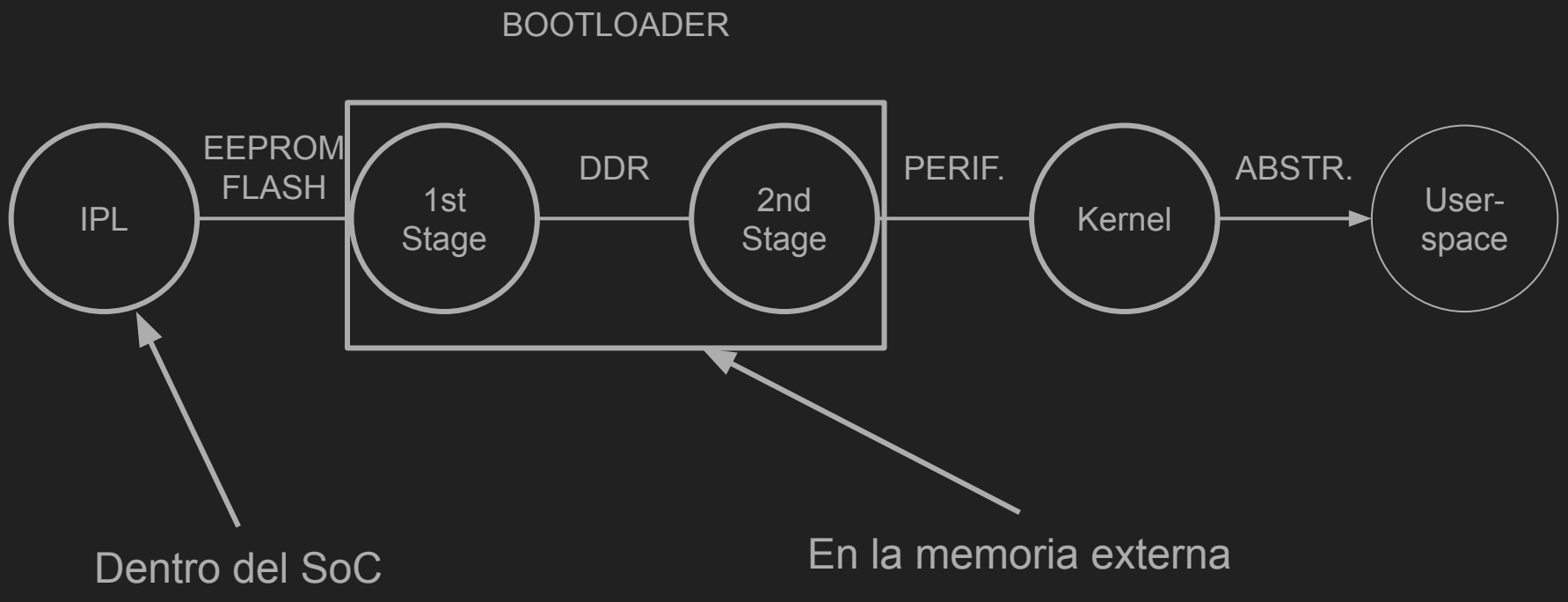
Ejercicio 13

- Analizar el firmware 3. Esto es un bootloader de una cámara ultra low cost con un chip de HiSilicon hi3516ev200
- Averiguar la arquitectura!
- Importar el bootloader en Ghidra
- 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

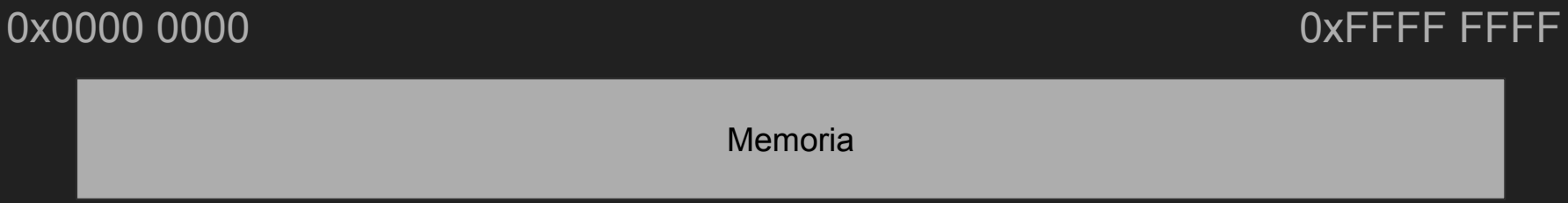
Ejercicio 13

- Analizar el firmware 3. Esto es un bootloader de una cámara ultra low cost con un chip de HiSilicon hi3516ev200
  - ARM v7 Little Endian
- Debate, ¿Qué estrategias tenemos? ¿Cuáles podrían ser las más útiles?:
  - La estrategia de strings no funciona en este caso porque no hay referencias a los strings! No sabemos donde se usan!
  - No tenemos SVD!
  - No tenemos datasheet!
  - El entry point parece tener algo mal reverseado…

El proceso de arranque

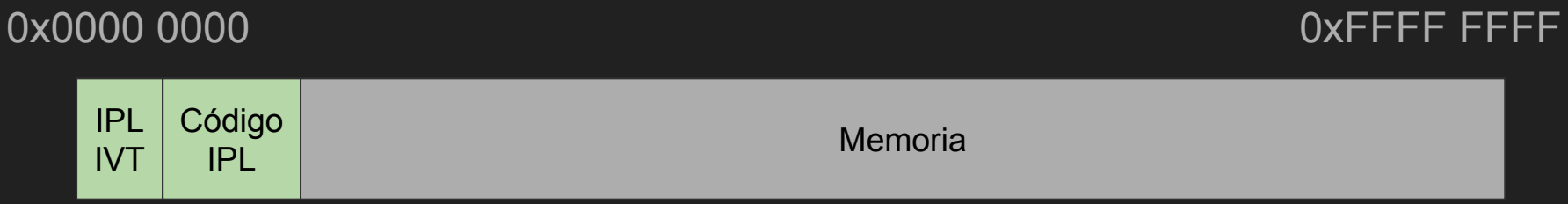


El proceso de arranque



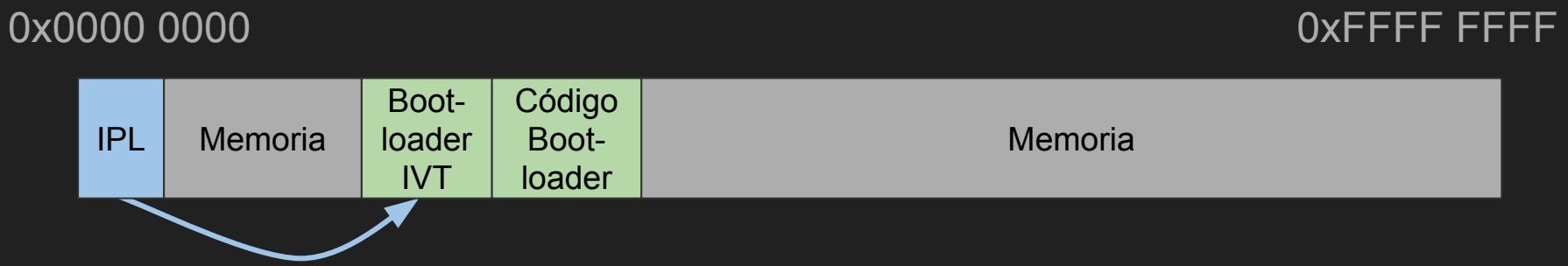
Antes del arranque

El proceso de arranque



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

El proceso de arranque



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

Binarios RAW

- 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 14

- 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!

Ejercicio 14

- Analizar la IVT de nuestro bootloader y extraer la dirección base real!
  - 0x4080 0000

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 y no tenemos datahseet…

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
```

Ejercicio 15

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