Hands on Reversing

Ciberseguridad (CIB)

©Ismael Ripoll

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf
Escola Tècnica Superior d'Enginyeria Informàtica

November 13, 2023

Índice

1. Qué vamos a trabajar
2. Conocimientos de base
3. Reversing
   - Plan de trabajo
   - Conexión serie
   - Bootloader
   - Extraer información
   - Analizar lo extraído
4. Conclusiones

Qué vamos a trabajar

- Breve introducción práctica a la ingeniería inversa.
- Solo se trata de tener una primera toma de contacto (Serían necesarias varias asignaturas para abordar toda la problemática del reversing).
- Te servirá para conocer mejor en qué consiste y valorar si te gustaría aprender más sobre esta área.
- Vamos a utilizar un equipo real, al que le aplicaremos varias técnicas.
- Las herramientas de reversing se trabajarán entorno un caso de uso: el router ZXHN de ZTE, desarrollado en 2013.
- El objetivo es hacer una auditoría de caja negra de seguridad. Por lo que asumiremos que el equipo es nuevo y no existe información en internet o del frabricante.

Por “caja negra” se entiende que partimos de no disponer de ningún tipo de información proporcionada por el fabricante.

Objetivos de la ingeniería inversa (I)

- La ingeniería inversa no es un objetivo en sí misma, sino que solo es un medio (o herramienta intermedia) para el objetivo final.
- Los objetivos finales que pueden necesitar una ingeniería inversa pueden ser:
  - desarrollar un producto compatible con el sistema propietario,
  - hacer una auditoría de seguridad,
  - extender su funcionalidad,
  - buscar malware,
  - o incluso como medio para buscar y/o desarrollar exploits.
- El objetivo del reversing se limita a “tener una noción clara de cómo funciona un sistema”, para luego abordar los objetivos finales con solvencia.

Objetivos de la ingeniería inversa (II)

- La ingeniería inversa se suele limitar a conocer solo aquellos aspectos del sistema que serán necesarios para completar el objetivo final. Si no ponemos un limite a lo que se investiga podemos llegar a analizar el diseño de los transistores en los chips (como por ejemplo, si que hicieron los que hacen reversing en las memorias DRAM, para explotar el RowHammer), o analizar el diseño de las CPUs en el Spectre y Metldown.
- Puesto que nos enfrentamos a un sistema desconocido, es difícil hacer un plan de actuación detallado, ya que las acciones que intentemos dependerán completamente del éxito o fracaso de las acciones anteriores. Por ejemplo, si el sistema no tiene un Linux, entonces no tiene sentido tratar de conseguir tener un shell.

Background

- En general, es necesario tener los mismos conocimientos que los que diseñaron el sistema.
- Si el objetivo es localizar vulnerabilidades, entonces se tiene que tener una visión más global para poder desarrollar un pensamiento lateral.
- En nuestro caso necesitamos saber:
  - La arquitectura y ensamblador MIPS32
  - Compilación cruzada y el ABI.
  - Manejo de librerías de código objeto.
  - Utilizar emuladores: Qemu.
  - Programación avanzada de sistemas operativos.
  - Buses de comunicación: RS232.
  - Nociones de electrónica.

Plan de trabajo

- Los pasos que intentaremos son:
  1. Acceso al target.
     - 1.1 Identificar el hardware.
     - 1.2 Buscar buses de comunicación para conectarse en modo desarrollador.
     - 1.3 Analizar el cargador de arranque (grub, u-boot, etc.)
  2. Extraer el firmware.
  3. Analizar el firmware 1.- reverse engineering dinámico (Ghidra).
     - 3.1 Descompilado.
     - 3.2 Identificación de funciones.
  4. Entorno de desarrollo cruzado
     - 4.1 Desarrollar nuestro código por si hace falta inyectar código.
     - 4.2 Ejecutar código binario en el target.
  5. Analizar el firmware 2.- data forensics (Volatility, Volcado).
     - 5.1 Localización de estructuras de datos.
     - 5.2 Listado de procesos, ficheros, abiertos, etc.

Conexión serie (I)

- Lo primero consiste en tratar de identificar visualmente los componentes más importantes: CPU principal, memoria DRAM, memoria Flash o de almacenamiento, conexiones (tanto con conectores, como las conexiones o las patillas a las que se les pudiera soldar algún cable).
- Excepto en la electrónica de consumo, en todos los demás ámbitos, la interfaz es una “consola” serie. Y en concreto, el puerto RS232, es con mucha diferencia el más utilizado.
- Podemos decir que todos los sistemas tienen un puerto serie que utilizan los desarrolladores, y que suele estar disponible en los equipos en producción.
- Suelen ser 4 terminales, a veces tienen un conector “pin head” o pueden aparecer solo las pistas PCB sin conector, en cuyo caso tendremos que soldar un conector.

Conexión serie (II)

- Utilizando un polímetro se busca el pin de masa comprobando continuidad entre cada uno de los pines y lo que seguramente es la masa del equipo.
- Luego con un osciloscopio se miran las tensiones de todos los pines puesto que el puerto serie puede trabajar con varias tensiones (3.3, 5 o 12 voltios) y no queremos quemar nuestro ordenador cuando lo conectemos.
  El hecho de usar un osciloscopio es porque permite trabajar con un amplio rango de tensiones, mientras que el analizador lógico está limitado a 5 o 3.3.
- Luego utilizando un analizador lógico se busca el terminal TX (transmisión de datos). El programa `pulseview` se puede utilizar para analizar el protocolo. Con este vemos la forma de la señal y con uno de los módulos de análisis de protocolo podemos ver los carácteres transmitidos.

Conexión serie (III)

- Finalmente, con un conversor serie a USB (se pueden comprar en Aliexpress por poco más de un Euro) se puede conectar al PC. El conversor USB-Serie es muy barato pero se rompe con mucha facilidad.
- Y con el programa `screen` (u otro como minicom) se pude tener una consola.

```
# screen -L -Logfile output.screen /dev/ttyUSB0 115200
Booting
Press '1' to enter BOOT console...
Press '2' to enter DEBUG mode......
....
<RTL867X>11
<RTL867X>
```

- Para utilizar el programa `screen` puede que sea necesario ser root, ya que tiene que acceder al fichero especial `/dev/ttyUSB0` y puedes no tener permisos de acceso.

Bootloader (I)

- La consola que hemos obtenido suele mostrar los primeros mensajes de arranque. Tanto del cargador, como del sistema operativo.
- Puede que esa consola también sirva para enviar datos al target. En este paso puede que seamos capaces de interactuar con el cargador y luego posteriormente con el propio sistema operativo.
- Si hay suerte, el sistema operativo nos ofrecerá una consola de root sin password. Pero si no es el caso, entonces necesitaremos más información (esto es, hacer reversing sobre el bootloader).
- Los cargadores de arranque suelen contar con una CLI (Command Line Interface), que es como un shell con comandos especializados para el arranque. Recordemos que un “cargador” no es otra cosa que un miniprograma que inicializa la CPU, carga en memoria la imagen del sistema operativo y acaba dándole el control a este.

Bootloader (II)

- La CLI de los cargadores tienen comandos de muy bajo nivel: lee y escribe memoria RAM y flash, implementan protocolos de red de carga de operativos como bootp o tftp. En nuestro caso, tiene hasta una mini ayuda en línea!.

```
<RTL867X>help
help
info
reboot
run [app addr] [entry addr]
r [addr]
w [addr] [val]
d [addr] <len>
...
```

Extraer información

- Los bootloaders solo suelen hacer una inicialización mínima, entre otras cosas, **no borran memoria RAM**. Por tanto, es posible que la memoria conserve los valores que tenían justo antes del último reboot.
- Utilizando los comandos del CLI podemos volcar por el puerto serie el contenido de la memoria. Eso lleva varias horas (o días).
- Mirando el manual de MIPS32 sabemos que la memoria en modo kernel está situada en la dirección 0x80000000, y de los logs de arranque vemos que tenemos 128Kb, por tanto el comando CLI para extraer la DRAM es:

```
<RTL867X> d 0x80000000 134217728
0x80000000: 10 00 00 FF 00 00 00 00 10 00 01 05 00 00 00 00 ..
...
```

- Para obtener el volcado necesitamos guardar lo que el CLI vuelca por el puerto serie y luego procesarlo con un script (de python) para convertirlo en la secuencia de bytes que hay en la DRAM.

Analizar lo extraído (I)

- Lo que tenemos es un “dump de memoria”. Sobre el que podremos hacer dos tipos de análisis:
  1. Análisis de código: Para lo que utilizaremos herramientas como Ghidra o Ida.
  2. Análisis de datos: Con lo que contamos con los comandos de UNIX como `strings` y `hd` o las herramientas Binwalk y Volatility.
- Todo programa debe contener la cadenas de texto que imprime (puede que estén ofuscadas o cifradas). Por tanto buscar cadenas de carácteres relacionadas con a autenticación pueden que nos lleve a encontrar, en posiciones cercanas, datos interesantes (contraseñas, configuraciones, etc.)
- Recordemos que lo que hemos obtenido es un volcado de la **memoria física**, donde seguramente está el kernel y todos los procesos. También tendremos zonas de memoria con basura o de procesos viejos que ya han terminado.

Analizar lo extraído (II)

- Al tener la memoria física, es posible que muchas páginas de los procesos no estén consecutivas ni ordenadas, y por tanto la información puede estar fragmentada.
- Vamos a trabajar con información incompleta o incluso incorrecta, así que paciencia e imaginación.
- Por suerte, la mayoría de la memoria del kernel sí que está consecutiva y está en direcciones físicas, pero para hacer un análisis preciso de los procesos es necesario encontrar las tabla de páginas de cada uno y luego hacer el mapeado para obtener la memoria virtual.

strings (I)

- El programa más simple, a la vez que efectivo, es `strings`. Que viene como una utilidad más dentro del conjunto de herramientas de desarrollo conocido como “binutils” (entre las que están el compilador, linker, editores de ELF, etc.)
- dado un fichero binarios, `strings` muestra por consola todas aquellas secuencias de bytes que tienen aspecto de cadenas de carácteres. Si la salida la pasamos por el programa `less`, entonces podremos buscar cadenas interesantes.

```
$ strings ZXHN.bin | less
```

- Analizar todos los resultados requiere paciencia.

Ghidra (I)

- Ghidra es un Framework para Reversing, o como ellos se definen “software reverse engineering (SRE) suite”, desarrollado por la NSA (National Security Agency) de EEUU y publicado en 2019. Que rápidamente está substituyendo a IDA Pro, como herramienta preferida por los analistas.
- Fue un desarrollo interno de la NSA, pero se dieron cuenta que les era más práctico hacerlo público, porque así se evitaban la fase de aprendizaje a los que contrataban.
- Ghidra es un desensamblador, descompilador, y recientemente debugger.
- Se puede descargar del `https://ghidra-sre.org/` y es compatible con todos los sistemas operativos.
- Tratar de explicar cómo usarlo está fuera de esta introducción al reversing.

Ejecución en Qemu (I)

- Solo voy a poner los pasos para poder compilar un programa de C para MIPS32 y cómo ejecutarlo y depurarlo utilizando Qemu.
- Como no puede ser de otra forma. Suponiendo que estamos en un Linux Ubuntu, instalamos los siguientes paquetes:

```
# apt install binutils-mips-linux-gnu gcc-mips-linux-gnu \
  libc6-dev-mips-cross gdb-multiarch qemu-user qemu-user-binfmt
```

- Creamos un hello world en C (puedes usar un editor, o hacer está fricada de bash):

Ejecución en Qemu (II)

```
$ cat > hello.c <<EOF
#include <stdio.h>
int main(){
  printf("Hello World\n");
  return 0;
}
EOF
```

- Lo compilamos. <humor>Como se puede observar hacer una compilación cruzada es un tema muy avanzado</humor>:

```
$ mips-linux-gnu-gcc hello.c -o hello
```

- Lo ejecutamos (llamando la interprete/emulador qemu):

Ejecución en Qemu (III)

```
$ qemu-mips -L /usr/mips-linux-gnu/ ./hello
Hello World
```

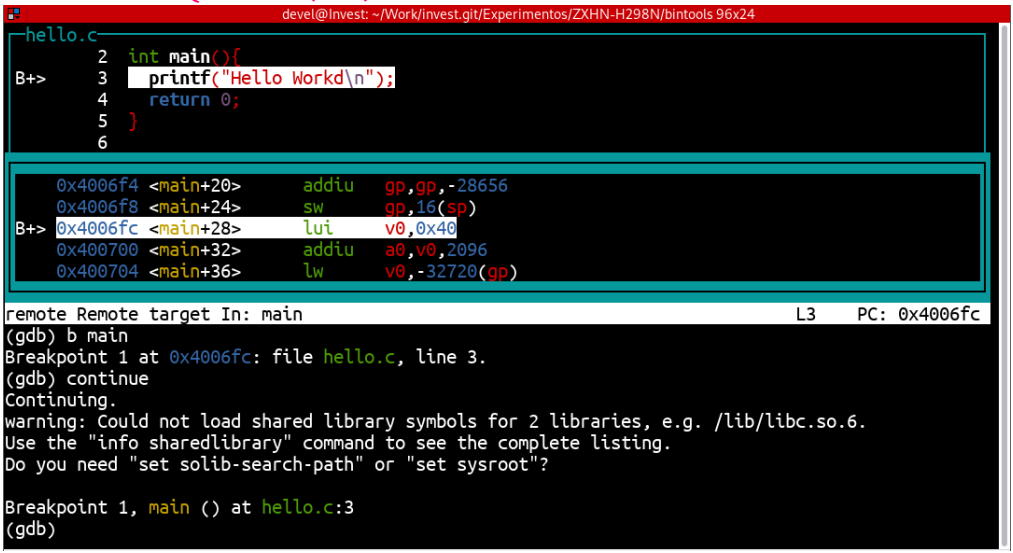
- Para depurar como un pro (compilamos con -ggdb y usamos -g 1234):

```
$ mips-linux-gnu-gcc -ggdb hello.c -o hello
$ qemu-mips -g 1234 -L /usr/mips-linux-gnu/ ./hello
```

y en otra consola:

```
$ gdb-multiarch ./hello
(gdb) target remote :1234
(gdb) b main
(gdb) continue
```

Ejecución en Qemu (IV)



Conclusiones (I)

- Como habéis podido ver, hacer reversing no es más que poner en práctica muchos de los conceptos que YA habíais estudiado (aunque puede que no los tengáis frescos).
- Hay que decir que explicar el ensamblador de MIPS es poco motivante en sí mismo, o ver todas las "cosas" que pasan con las memorias caches, puede parecer bastante marciano... Pero cuando el éxito depende de que un store (`sw`) no se te cuele entre el "`lui`" y el "`jarl`" en el hook:

```
81e07bec: 3c1981df   lui  t9,0x81df
81e07bf0: afbf001c   sw   ra,28(sp) <<-- What the heck!
81e07bf4: 3739f000   ori  t9,t9,0xf000
81e07bf8: 0320f809   jalr t9
....
```

Conclusiones (II)

O que nuestro programa solo funciona si somos capaces de flushear las caches. Pues creo que las cosas se ven de otra manera.

- Otra cosa que he intentado con esta clase es romper la barrera que suele haber entre la programación “clasica” (con Python, java, c++, c...) y el ensamblador o los opcodes. Y que se puede “manipular” opcodes de forma similar a cuando hacemos copypaste de una función en “C” desde github.
- Pero lo que espero que no tengáis la menor duda de que:

Fabrice Bellard es el FMOTU.