INTRODUCCIÓN AL DESARROLLO DE EXPLOIT EN ARM

Bienvenido a esta serie de tutoriales sobre conceptos básicos de ensamblaje de ARM. Esta es la preparación para la serie de tutoriales de seguimiento sobre el desarrollo de exploits ARM . Antes de que podamos sumergirnos en la creación de shellcode ARM y construir cadenas ROP, primero debemos cubrir algunos conceptos básicos de ensamblaje ARM.

Para seguir los ejemplos, necesitará un entorno de laboratorio basado en ARM. Si no tiene un dispositivo ARM (como Raspberry Pi), puede configurar su propio entorno de laboratorio en una máquina virtual usando QEMU y la distribución de Raspberry Pi. En este tutorial, el enfoque estará en ARM de 32 bits, y los ejemplos se compilan en un ARMv6.

**¿POR QUÉ ARM?**

Este tutorial es generalmente para personas que desean aprender los conceptos básicos del ARM Assembly. Especialmente para aquellos de ustedes que están interesados ​​en escribir exploits en la plataforma ARM. Es posible que ya haya notado que los procesadores ARM están en todas partes a su alrededor. Cuando miro a mi alrededor, puedo contar muchos más dispositivos que cuentan con un procesador ARM en mi casa que los procesadores Intel. Esto incluye teléfonos, enrutadores y no olvidar los dispositivos IoT que parecen explotar en las ventas en estosdías. Dicho esto, el procesador ARM se ha convertido en uno de los núcleos de CPU más extendidos del mundo. Lo que nos lleva al hecho de que, al igual que las PC, los dispositivos IoT son susceptibles a un abuso de validación de entrada incorrecto, como desbordamientos de búfer. Dado el uso generalizado de dispositivos basados ​​en ARM y la posibilidad de mal uso, los ataques a estos dispositivos se han vuelto mucho más comunes.

Empresas mas famosas por el uso de ARM

**Qualcomm** Es uno de los principales suministradores de la familia de procesadores para [*teléfonos inteligentes*](https://es.wikipedia.org/wiki/Tel%C3%A9fono_inteligente) [Snapdragon](https://es.wikipedia.org/wiki/Snapdragon), Qualcomm es el Intel de los teléfonos inteligentes

**Apple** no necesita presentación la empresa de la manzana hace poco publico que dejará Intel y se ira con ARM en 2021 para usar ARM en sus macs ya que los iphone ya usaban ARM

Sin embargo, tenemos más expertos especializados en investigación de seguridad x86 que los que tenemos para ARM, aunque el lenguaje ensamblador ARM es quizás el lenguaje ensamblador más fácil de uso generalizado. Entonces, ¿por qué no hay más personas enfocadas en ARM? Quizás porque hay más recursos de aprendizaje que cubren la explotación en Intel que los que hay para ARM. Solo piense en los excelentes tutoriales sobre Intel x86 Exploit escrito por [FuzzySecurity](https://www.fuzzysecurity.com/tutorials/expDev/1.html) o el [Corelan](https://www.corelan.be/index.php/2009/07/19/exploit-writing-tutorial-part-1-stack-based-overflows/) Team- Pautas como estas ayudan a las personas interesadas en esta área específica a obtener conocimientos prácticos y la inspiración para aprender más allá de lo que se cubre en esos tutoriales. Si está interesado en la escritura de exploits x86, los tutoriales de Corelan y Fuzzysec son su punto de partida perfecto. En esta serie de tutoriales aquí, nos centraremos en los conceptos básicos de ensamblaje y aprovecharemos la escritura en ARM.

Procesador ARM vs. Procesador Intel

Hay muchas diferencias entre Intel y ARM, pero la diferencia principal es el conjunto de instrucciones. Intel es un procesador CISC (Computación de conjunto de instrucciones complejas) que tiene un conjunto de instrucciones más grande y más rico en funciones y permite que muchas instrucciones complejas accedan a la memoria. Por lo tanto, tiene más operaciones, modos de direccionamiento, pero menos registros que ARM. Los procesadores CISC se utilizan principalmente en PC, estaciones de trabajo y servidores normales.

**ARM** es un procesador **RISC** (Computación de conjunto de instrucciones reducido) y, por lo tanto, tiene un conjunto de instrucciones simplificado (100 instrucciones o menos) y registros de propósito más general que CISC. A diferencia de Intel, ARM usa instrucciones que operan solo en registros y usa un modelo de memoria Cargar / Almacenar para acceder a la memoria, lo que significa que solo las instrucciones Cargar / Almacenar pueden acceder a la memoria. Esto significa que incrementar un valor de 32 bits en una dirección de memoria particular en ARM requeriría tres tipos de instrucciones (cargar, incrementar y almacenar) para cargar primero el valor en una dirección particular en un registro, incrementarlo dentro del registro y almacenar vuelve a la memoria desde el registro.

El conjunto de instrucciones reducido tiene sus ventajas y desventajas. Una de las ventajas es que las instrucciones pueden ejecutarse más rápidamente, lo que potencialmente permite una mayor velocidad (los sistemas RISC acortan el tiempo de ejecución al reducir los ciclos de reloj por instrucción). La desventaja es que menos instrucciones significan un mayor énfasis en la escritura eficiente de software con las instrucciones limitadas que están disponibles. También es importante tener en cuenta que ARM tiene dos modos, el modo ARM y el modo Pulgar. Las instrucciones en miniatura pueden ser de 2 o 4 bytes (más sobre eso en la [Parte 3: Conjunto de instrucciones ARM](https://azeria-labs.com/arm-instruction-set-part-3/) ).

Más diferencias entre ARM y x86 son:

* En ARM, la mayoría de las instrucciones se pueden usar para la ejecución condicional.
* Las series de procesadores Intel x86 y x86-64 utilizan el  formato **little-endian**
* La arquitectura ARM era little-endian antes de la versión 3. Desde entonces, los procesadores ARM se convirtieron en **BI-endian** y cuentan con una configuración que permite la endianness *conmutable* .

No solo hay diferencias entre Intel y ARM, sino también entre las diferentes versiones de ARM. Esta serie de tutoriales tiene la intención de mantenerla lo más genérica posible para que pueda obtener una comprensión general sobre cómo funciona ARM. Una vez que comprenda los fundamentos, es fácil aprender los matices para la versión ARM de destino elegida. Los ejemplos en este tutorial se crearon en un ARMv6 de 32 bits (Raspberry Pi 1), por lo tanto, las explicaciones están relacionadas con esta versión exacta.

El nombramiento de las [diferentes versiones ARM](https://en.wikipedia.org/wiki/List_of_ARM_microarchitectures) también puede ser confuso:

| **Familia ARM** | **Arquitectura ARM** |
| --- | --- |
| ARM7 | ARM v4 |
| ARM9 | ARM v5 |
| ARM11 | ARM v6 |
| Cortex-A | ARM v7-A |
| Cortex-R | ARM v7-R |
| Cortex-M | ARM v7-M |

**ESCRIBIENDO ENSAMBLADOR**

Antes de que podamos comenzar a sumergirnos en el desarrollo de exploits ARM, primero debemos comprender los conceptos básicos de la programación en lenguaje ensamblador, que requiere un poco de conocimientos previos antes de que pueda comenzar a apreciarlo. Pero, ¿por qué necesitamos ARM Assembly? ¿No es suficiente escribir nuestros exploits en un lenguaje de programación / scripting "normal"? No es así, si queremos poder realizar ingeniería inversa y comprender el flujo del programa de los binarios ARM, construir nuestro propio código shell ARM, crear cadenas ARM ROP y depurar aplicaciones ARM.

No necesita conocer cada pequeño detalle del lenguaje ensamblador para poder realizar ingeniería inversa y explotar el desarrollo, sin embargo, se requiere parte de él para comprender el panorama general. Los fundamentos serán cubiertos en esta serie de tutoriales. Si desea obtener más información, puede visitar los enlaces que figuran al final de este capítulo.

Entonces, ¿qué es exactamente el lenguaje ensamblador? El lenguaje ensamblador es solo una delgada capa de sintaxis en la parte superior del código de la máquina que se compone de instrucciones, que están codificadas en representaciones binarias (código de la máquina), que es lo que nuestra computadora entiende. Entonces, ¿por qué no escribimos código de máquina en su lugar? Bueno, eso sería un fastidio. Por esta razón, escribiremos ensamblaje, ensamblaje ARM, que es mucho más fácil de entender para los humanos. Nuestra computadora no puede ejecutar el código de ensamblaje, porque necesita código de máquina. La herramienta que usaremos para ensamblar el código de ensamblaje en el código de máquina es un ensamblador [GNU del](https://www.gnu.org/software/binutils/) proyecto [GNU Binutils](https://www.gnu.org/software/binutils/) denominado  ***como el*** que funciona con archivos fuente que tienen la extensión \* .s.

Una vez que escribió su archivo de ensamblaje con la extensión \* .s, necesita ensamblarlo con [as](https://sourceware.org/binutils/docs/as/index.html#Top)  y vincularlo con [ld](https://sourceware.org/binutils/docs/ld/) :

$ como programa.s -o programa.o

$ ld program.o -o program

El lenguaje ensamblador es el nivel más bajo utilizado por los humanos para programar una computadora. Los operandos de una instrucción vienen después de los mnemónicos. Aquí hay un ejemplo:

**MOV R2, R1**

Ahora que sabemos que un programa de ensamblaje está compuesto de información textual llamada mnemotecnia, necesitamos convertirlo en código de máquina. Como se mencionó anteriormente, en el caso del ensamblaje ARM, el  proyecto [GNU Binutils](https://www.gnu.org/software/binutils/) nos proporciona una herramienta llamada **as** . El proceso de usar un ensamblador **como** para convertir del lenguaje ensamblador (ARM) al código de máquina (ARM) se llama ensamblaje.

En resumen, aprendimos que las computadoras entienden (responden) la presencia o ausencia de voltajes (señales) y que podemos representar múltiples señales en una secuencia de 0s y 1s (bits). Podemos usar código de máquina (secuencias de señales) para hacer que la computadora responda de una manera bien definida. Como no podemos recordar lo que significan todas estas secuencias, les damos abreviaturas, mnemotécnicos, y las usamos para representar instrucciones. Este conjunto de mnemónicos es el lenguaje ensamblador de la computadora y usamos un programa llamado Assembler para convertir el código de la representación mnemónica al código de máquina legible por computadora, de la misma manera que lo hace un compilador para lenguajes de alto nivel.

Otras lecturas

<https://www.coranac.com/tonc/text/asm.htm>

<http://thinkingeek.com/arm-assembler-raspberry-pi/>

<http://infocenter.arm.com/help/topic/com.arm.doc.dui0068b/index.html>

<http://www.keil.com/support/man/docs/armasm/default.htm>

Cesar Molina 12/05/2020