AZERIA

INTRODUCTION TO ARM ASSEMBLY BASICS

Welcome to this tutorial series on ARM assembly basics. This is the preparation for the followup tutorial series on ARM exploit development. Before we can dive into creating ARM shellcode and build ROP chains, we need to cover some ARM Assembly basics first.

The following topics will be covered step by step:

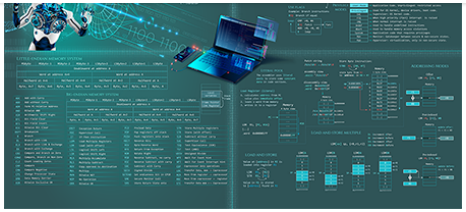
ARM Assembly Basics Tutorial Series:
Part 1: Introduction to ARM Assembly
Part 2: Data Types Registers
Part 3: ARM Instruction Set
Part 4: Memory Instructions: Loading and Storing Data
Part 5: Load and Store Multiple
Part 6: Conditional Execution and Branching
Part 7: Stack and Functions

To follow along with the examples, you will need an ARM based lab environment. If you don't have an ARM device (like Raspberry Pi), you can set up your own lab environment in a Virtual Machine using QEMU and the Raspberry Pi distro by following this tutorial. If you are not familiar with basic debugging with GDB, you can get the basics in this tutorial. In this tutorial, the focus will be on ARM 32-bit, and the examples are compiled on an ARMv6.

Why ARM?

This tutorial is generally for people who want to learn the basics of ARM assembly. Especially for those of you who are interested in exploit writing on the ARM platform. You might have already noticed that ARM processors are everywhere around you. When I look around me, I can count far more devices that feature an ARM processor in my house than Intel processors. This includes phones,

validation abuse such as buffer overflows. Given the widespread usage of ARM based devices and the potential for misuse, attacks on these devices have become much more common.



Yet, we have more experts specialized in x86 security research than we have for ARM, although ARM assembly language is perhaps the easiest assembly language in widespread use. So, why aren't more people focusing on ARM? Perhaps because there are more learning resources out there covering exploitation on Intel than there are for ARM. Just think about the great tutorials on Intel x86 Exploit writing by Fuzzy Security or the Corelan Team – Guidelines like these help people interested in this specific area to get practical knowledge and the inspiration to learn beyond what is covered in those tutorials. If you are interested in x86 exploit writing, the Corelan and Fuzzysec tutorials are your perfect starting point. In this tutorial series here, we will focus on assembly basics and exploit writing on ARM.

ARM PROCESSOR VS. INTEL PROCESSOR

There are many differences between Intel and ARM, but the main difference is the instruction set. Intel is a CISC (Complex Instruction Set Computing) processor that has a larger and more feature-rich instruction set and allows many complex instructions to access memory. It therefore has more operations, addressing modes, but less registers than ARM. CISC processors are mainly used in normal PC's, Workstations, and servers.

ARM is a RISC (Reduced instruction set Computing) processor and therefore has a simplified instruction set (100 instructions or less) and more general purpose registers than CISC. Unlike Intel, ARM uses instructions that operate only on registers and uses a Load/Store memory model for memory access, which means that only Load/Store instructions can access memory. This means that incrementing a 32-bit value at a particular memory address on ARM would require three types of instructions (load, increment and store) to first load the value at a particular address into a register, increment it within the register, and store it back to the memory from the register.

The reduced instruction set has its advantages and disadvantages. One of the advantages is that instructions can be executed more quickly, potentially allowing for greater speed (RISC systems shorten execution time by reducing the clock cycles per instruction). The downside is that less instructions means a greater emphasis on the efficient writing of software with the limited instructions that are available. Also important to note is that ARM has two modes, ARM mode and Thumb mode. Thumb instructions can be either 2 or 4 bytes (more on that in Part 3: ARM Instruction set).

More differences between ARM and x86 are:

- In ARM, most instructions can be used for conditional execution.
- The Intel x86 and x86-64 series of processors use the **little-endian** format

There are not only differences between Intel and ARM, but also between different ARM version themselves. This tutorial series is intended to keep it as generic as possible so that you get a general understanding about how ARM works. Once you understand the fundamentals, it's easy to learn the nuances for your chosen target ARM version. The examples in this tutorial were created on an 32-bit ARMv6 (Raspberry Pi 1), therefore the explanations are related to this exact version.

The naming of the different ARM versions might also be confusing:

ARM family	ARM architecture
ARM7	ARM v4
ARM9	ARM v5
ARM11	ARM v6
Cortex-A	ARM v7-A
Cortex-R	ARM v7-R
Cortex-M	ARM v7-M

WRITING ASSEMBLY

Before we can start diving into ARM exploit development we first need to understand the basics of Assembly language programming, which requires a little background knowledge before you can start to appreciate it. But why do we even need ARM Assembly, isn't it enough to write our exploits in a "normal" programming / scripting language? It is not, if we want to be able to do Reverse Engineering and understand the program flow of ARM binaries, build our own ARM shellcode, craft ARM ROP chains, and debug ARM applications.

You don't need to know every little detail of the Assembly language to be able to do Reverse Engineering and exploit development, yet some of it is required for understanding the bigger picture. The fundamentals will be covered in this tutorial series. If you want to learn more you can visit the links listed at the end of this chapter.

So what exactly is Assembly language? Assembly language is just a thin syntax layer on top of the machine code which is composed of instructions, that are encoded in binary representations (machine code), which is what our computer understands. So why don't we

use to assemble the assembly code into machine code is a GNU Assembler from the GNU Binutils project named ***as*** which works with source files having the *.s extension.

Once you wrote your assembly file with the extension *.s, you need to assemble it with as and link it with ld:

```
$ as program.s -o program.o
$ ld program.o -o program
```

ASSEMBLY UNDER THE HOOD

Let's start at the very bottom and work our way up to the assembly language. At the lowest level, we have our electrical signals on our circuit. Signals are formed by switching the electrical voltage to one of two levels, say 0 volts ('off') or 5 volts ('on'). Because just by looking we can't easily tell what voltage the circuit is at, we choose to write patterns of on/off voltages using visual representations, the digits 0 and 1, to not only represent the idea of an absence or presence of a signal, but also because 0 and 1 are digits of the binary system. We then group the sequence of 0 and 1 to form a machine code instruction which is the smallest working unit of a computer processor. Here is an example of a machine language instruction:

1110 0001 1010 0000 0010 0000 0000 0001

So far so good, but we can't remember what each of these patterns (of 0 and 1) mean. For this reason, we use so called mnemonics, abbreviations to help us remember these binary patterns, where each machine code instruction is given a name. These mnemonics often consist of three letters, but this is not obligatory. We can write a program using these mnemonics as instructions. This program is

operands of an instruction come after the mnemonic(s). Here is an example:

MOV R2, R1

Now that we know that an assembly program is made up of textual information called mnemonics, we need to get it converted into machine code. As mentioned above, in the case of ARM assembly, the GNU Binutils project supplies us with a tool called **as**. The process of using an assembler like **as** to convert from (ARM) assembly language to (ARM) machine code is called assembling.

In summary, we learned that computers understand (respond to) the presence or absence of voltages (signals) and that we can represent multiple signals in a sequence of 0s and 1s (bits). We can use machine code (sequences of signals) to cause the computer to respond in some well-defined way. Because we can't remember what all these sequences mean, we give them abbreviations – mnemonics, and use them to represent instructions. This set of mnemonics is the Assembly language of the computer and we use a program called Assembler to convert code from mnemonic representation to the computer-readable machine code, in the same way a compiler does for high-level languages.

FURTHER READING

1. Whirlwind Tour of ARM Assembly.
https://www.coranac.com/tonc/text/asm.htm

2. ARM assembler in Raspberry Pi.
http://thinkingeek.com/arm-assembler-raspberry-pi/

3. Practical Reverse Engineering: x86, x64, ARM, Windows Kernel, Reversing Tools, and Obfuscation by Bruce Dang, Alexandre Gazet, Elias Bachaalany and Sebastien Josse.

4. ARM Reference Manual.
http://infocenter.arm.com/help/topic/com.arm.doc.dui0068b/index.html

5. Assembler User Guide.
http://www.keil.com/support/man/docs/armasm/default.htm

PART 2: DATA TYPES AND REGISTERS ›