3

SPELL OF REBIRTH: TRANSMUTING GREETINGS INTO SHELLCODE

In this chapter we'll extend our ASM and C knowledge by writing a simple "Hello, world!" program. Then we'll modify that code to create our first hacking-related program: a simple shellcode for gaining command line access to a computer system. Yes, we're already that advanced. Along the way, we'll learn to work with pointers, use the GNU Debugger to inspect a program's allocated memory, and explore different addressing modes for memory access. First, though, we'll cover a few more general concepts about how computers handle information internally, and we'll take a closer look at how memory works.

Representing Information

We're going to start working with values larger than 1 byte in this chapter, and to fully comprehend what's happening we must introduce some basic concepts about how information is represented and stored in memory. This may not sound as exciting as writing a shellcode, but these fundamentals are just as important. Bear with me for a while, and in no time we'll do our very first hacking thing.

Bits, Bytes, Words, and Beyond

A *bit* is the smallest unit of information a computer can manage. A bit can hold one of two values: 0 or 1. These values are sometimes referred as *false* and *true*, especially in the context of logic circuitry.

If you put 8 bits together, you get a *byte*. Therefore, a byte is a sequence of eight 0s or 1s. One byte can represent any number up to 8 binary digits long, or if you're more comfortable counting in decimal, a byte can store 256 different values, typically from 0 to 255. I'll talk more about numeric representation shortly.

Each half of a byte is known as a *nibble*. A byte has two nibbles, the high one and the low one, each being 4 bits long. Therefore, each nibble can represent 16 different values ($2^4 = 16$). When working in hexadecimal (that is, in base 16), we can represent 16 values just with one digit (0 to 9 and then *a* to *f*). This means that a byte can be represented with two hexadecimal digits, one for each nibble.

Going up from bytes, we start doubling the number of bits:

- A *word* is composed of 2 bytes, or 16 bits.
- A *double word* or *dword* is composed of 2 words, 4 bytes, or 32 bits.
- A *quard word* or *qword* is composed of 2 double words, 4 words, 8 bytes, or 64 bits.

We won't go further than this, but feel free to figure out the names of the next sizes in the list.

Note that this is how these terms are usually interpreted, and they mostly derive from the Intel processor definitions. In reality, the definition of a *word* can vary by architecture, based on the processor's native word size, as we'll see in the next section.

Processor Native Word Sizes

A processor's *native word size* is the number of bits the processor is most comfortable working with at one time. In a sense, you encounter native word sizes regularly: Whenever we talk about 32-bit versus 64-bit processors or programs, we're referencing word sizes. Those numbers are more than just the way to choose the right download link for your Linux distribution, however. They also have some low-level implications:

- Do you remember the registers within a processor? Sure you do. The processor's native word size defines the size of those registers. A 32-bit processor has 32-bit registers, and a 64-bit processor has 64-bit registers. This isn't completely accurate, but for now just consider this size to be the size of the processor's registers.
- The native word size is usually also the width of the CPU's data bus.
- The native word size is usually the width of the address bus as well.

We'll talk about what those buses are later in the chapter.

Overall, and without going deep into the electronics within the processor, what you need to know is that each processor is optimized to work with its native word size. For example, a 32-bit processor will perform arithmetic operations or access memory faster when it's working with 32-bit values than when it's working with 16-bit values. I know that sounds a bit counterintuitive and may require an act of faith to believe, but it would be very tedious to go through all the complex details to understand why this is the case. Just to give you a little taste, here's a fragment from the "Data Types" section of the Intel 80386 Programmer's Reference Manual from 1986:

> When used in a configuration with a 32-bit bus, actual transfers of data between processor and memory take place in units of doublewords beginning at addresses evenly divisible by four; however, the processor converts requests for misaligned words or doublewords into the appropriate sequences of requests acceptable to the memory interface. Such misaligned data transfers reduce performance by requiring extra memory cycles. For maximum performance, data structures (including stacks) should be designed in such a way that, whenever possible, word operands are aligned at even addresses and doubleword operands are aligned at addresses evenly divisible by four. Due to instruction prefetching and queuing within the CPU, there is no requirement for instructions to be aligned on word or doubleword boundaries. (However, a slight increase in speed results if the target addresses of control transfers are evenly divisible by four.)

Summing up, try to program your processor using its native word size. It may look like you're wasting some space, but that's the right way to do it.

Decimal, Binary, Hexadecimal, and More

A number is an abstract entity that represents a quantity. Such a quantity is universal, but it can be represented in very different ways. The most obvious deviations comes from human languages. A 1 means one thing, but English speakers say *one*, Dutch speakers say *een*, Spanish speakers say *uno*, and so on. Each language has a different verbal representation for the same abstract quantity.

The same happens when we write numbers symbolically: The same abstract quantity can be represented in many different ways. For instance you can write twenty-four with Western Arabic numerals as 24 or with Roman numerals as XXIV. Setting aside the symbols themselves, the more general way of representing numbers is using *radix-based positional numeral systems*, where the position of each digit is associated with a value.

We normally use radix 10, which means that we use 10 possible symbols (0 to 9) for each digit or position in a number. If we need to represent more than 10 values, we add a new digit, which will be in the second position and will be multiplied by 10. Right? Figure 3-1 shows an example.

```
1234
^^^^
|||+-------- First position times 1     -> 4 x 1    =    4
||+----------Second position times 10   -> 3 x 10   =   30
|+-----------Third poisition times 100  -> 2 x 100  =  200
+------------Fourth Position times 1000 -> 1 x 1000 = 1000
                                            -----------------
                                                         1234
```

Figure 3-1: Parsing a base 10 number

We all know this from school, but perhaps we don't know the terminology. The basis of the system, 10 in this case, is called the *radix* (or *base*), and the value of each digit is multiplied by a power of the radix: The first position is multiplied by $10^0 = 1$, the second position by $10^1 = 10$, the third position by $10^2 = 100$, and so on.

Here's the interesting thing: We can use any number as the radix, and furthermore we can represent any integer number using any radix. Taking into account that computers use digital circuits that can only hold two values, any radix that's a multiple of 2 is interesting for us. Actually, there are three numerical representation systems that are used frequently when working with computers because they make some things much easier:

Binary A binary representation uses base 2, and therefore each digit can take values 0 or 1. Each position is multiplied by a power of 2 (1, 2, 4, 8, 16, . . .). Binary numbers are usually represented with a leading `0b`. For example, `0b101` in binary is 5 in decimal: $(1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) = 5$.

Octal This representation uses powers of 8 and symbols from 0 to 7 for each digit. Each octal digit takes 3 bits to be represented. The octal system was very common in the past, but it's rarely used nowadays. Unix filesystem permissions are one of the few cases where you'll still see it. Octal numbers are usually prefixed with a `0`. For example, `011` in octal is 9 in decimal: $(1 \times 8^1) + (1 \times 8^0) = 9$.

Hexadecimal This representation uses radix 16, so each digit can have 16 values. We only have 10 symbols in our normal numeral systems (0 to 9), so for the 6 missing symbols we use letters: *a* means 10, *b* means 11, and so on up to *f* for 15. Each hexadecimal digit requires 4 bits (a nibble), and this is the reason this system is used with computers. A byte can be represented with two hex digits, a word with four, a dword with eight, and so on. Hex values are usually prefixed with `0x`. For example, `0x21` is hexadecimal for 33: $(2 \times 16^1) + (1 \times 16^0) = 32 + 1 = 33$.

Which representation should you use in your code? It doesn't really matter, since all of them represent the same values. Using one or another depends on what you want to achieve and whether a particular system will make your life easier, but it won't have any impact on the final machine code that a program will generate.

Our focus here has been on whole numbers. Computers also have ways to represent real numbers with fractional components (fixed point and floating point), but we won't worry about such numbers in this book because you rarely need this type of data for hacking/systems programming. No kernel system call expects a real number as a parameter, for example.

We've also sort of been assuming that all numbers are positive, but how does a computer represent negative numbers? It's not hard, but it isn't obvious either.

Negative Numbers

To get a taste for how negative numbers work, let's think about a single byte (eight 1s or 0s). We've said that 8 bits can represent 256 values, from 0 to 255. That's perfect for natural numbers, but things change if we need negative numbers, too. First, we need to store the sign of the number, and that will take up one of the bits from the byte. That leaves us with 7 bits to represent the actual number, meaning 128 values. Figure 3-2 shows one way to represent a few of those numbers.

```
8 => 0 000 1000         -8 => 1 000 1000
7 => 0 000 0111         -7 => 1 000 0111
...                     ...
1 => 0 000 0001         -1 => 1 000 0001
0 => 0 000 0000          0 => 1 000 0000
```

Figure 3-2: A bad way to represent negative numbers

There are a few problems with this representation. The first is that we have two representations for the number zero. That isn't convenient as it can make computations ambiguous, and we're also losing the opportunity to represent one extra number. The second problem is that arithmetic becomes complicated: multiplication is kind of easy, but addition is kind of hell.

Fortunately for us, some smart people long ago came up with a better representation for negative numbers. It also uses the most significant bit to indicate the sign, but the value of the number is encoded in a smarter way. Figure 3-3 shows how it works.

```
8 => 0 000 1000        -8 => 1 111 1000
7 => 0 000 0111        -7 => 1 111 1001
...                    ...
2 => 0 000 0010        -2 => 1 111 1110
1 => 0 000 0001        -1 => 1 111 1111
0 => 0 000 0000
```

Figure 3-3: A good way to represent negative numbers

In addition to the sign, the rest of the number is constructed counting upward as usual for the positive numbers, and counting backward for the negative numbers. Now there's one single representation for zero, and it's actually zero (a 0 in each bit). This has a consequence: Zero is somehow considered a positive number because its most significant bit is 0 (that's our sign bit). This is why a signed char can take values from −128 to +127 (the zero is part of the positives).

With this system, you can change a number's sign—that is, find the positive equivalent of a negative number or vice versa—with two simple steps:

1. Invert all the bits in the number (this is called the *one's complement*).
2. Add 1 to the result.

In all, this sequence of operations is called taking the *two's complement*. Let's use the number 5 as an example and calculate its two's complement to determine the bit representation of −5:

1. The number 5 in binary is `00000101`, so inverting the bits gives us `11111010`.
2. Adding 1 gives us `11111011`, which is indeed −5.

The biggest advantage of this system is that basic arithmetic operations just work. If you add the representations of 5 and −5 that we just figured out, it results in 0. Subtraction and multiplication also work out of the box.

We won't discuss negative number representation further, but the interested reader can see the "Two's Complement" Wikipedia page for more information.

Text Strings

We've been talking about how to represent integer numbers, but there's one more type of data we'll want to work with: text. A *text string* is just a sequence of characters—that is, any message we write in the screen or any text we type on a keyboard to provide as input to our programs.

We can only store numbers in computer memory, so how can we represent the letters in a text string? The answer is to use a code. A code, in it's simplest form, is just a map between symbols, so we can assign each letter and symbol we want to be able to write in the screen a number, and then write code that will convert those numbers into characters printed in the screen.

We have a plethora of options for assigning a number to each character, but some people came up with good codes long ago, with ASCII (American Standard Code for Information Interchange) being the best known. Back in the day, EBCDIC was also popular because early IBM computers used it, and more recently Unicode has become the de-facto standard. UTF-8, one of the encodings for Unicode, uses variable-length encoding and is backward-compatible with ASCII. In other words, 1-byte-long UTF-8 codes are the same as the ASCII codes.

We'll use ASCII string encoding throughout this book. ASCII was originally defined as a 7-bit code in order to be compatible with the teletype machines back in the day. This is also the reason that the first 32 codes (from `0x00` to `0x1f`) are control values that were intended to operate those machines. For example, a code like `0x0a`, also known as *LF* or *line feed*, was intended to move the paper in the teletype machine to the next line. Many of those codes aren't used anymore, but a few of them are: Today you'll find *line feed*, *carriage return*, and *tabulation* represented as `\n`, `\r` and `\t`, respectively.

In ASCII, the value `0x20` represents a space. Numbers are encoded using values `0x30` to `0x39`. This makes it very convenient to print decimal digits in ASM programs: Just add `0x30` to the value. Uppercase letter are encoded using values `0x41` to `0x5a`, and lowercase letters are conveniently stored using values `0x61` to `0x7a`. This is especially interesting because we can convert between uppercase and lowercase just by switching bit 5 of a character on or off. When sorting strings alphabetically, the sort is actually based on these underlying numbers, so a word beginning with a capital letters would be sorted before a word beginning with a lowercase letter.

To use a text string in a program, we need one extra piece of information: We need to tell the program when those numbers in memory are no longer letters. Put another way, we need to define the length of the string. There are two main options:

- Store a number indicating the size of a string together with the sequence of numbers representing the string itself.
- Use a number that would never appear in a text string as a guard value or delimiter at the end of the string.

Languages like Java use option 1. (Historically, Java also used the UTF-16 encoding, an encoding for Unicode that has a minimum unit size of 2 bytes). The C language uses option 2, with `0x00` indicating the end of a string. Figure 3-4 compares how these two languages would store the string `"ABC"` in memory.

```
Java                    C
Size     | 0x00 |       | 0x41 | Char 1
Size     | 0x00 |       | 0x42 | Char 2
Size     | 0x00 |       | 0x43 | Char 3
Size     | 0x04 |       | 0x00 | End of String
Char 1   | 0x00 |
Char 1   | 0x41 |
Char 2   | 0x00 |
Char 2   | 0x42 |
Char 3   | 0x00 |
Char 3   | 0x43 |
```

Figure 3-4: String representation in Java versus C

Java starts with a 32-bit value to store the size of the string, meaning even an empty string will take up at least 4 bytes in memory. Then it uses 16 bits to encode each character. As with UTF-8, UTF-16 also matches the ASCII code for each character, but extending its size to 2 bytes. By contrast, C just uses 1 byte per character, plus an extra byte for the guard value at the end. Our three-character string therefore takes up 4 bytes. Compare this to the the 10 bytes required by Java. Note also that in Java a string is usually an object, which requires even more memory to be stored.

NOTE *Since Java 9, Java's string representation has changed to include what are called* compact strings. *Now, unless the string requires a special character that needs to be encoded using UTF-16, Java uses 1 byte per character. It also adds an extra byte to indicate which type of string it is (1 or 2 bytes per character). In this newer representation, the Java string in our example would take up 8 bytes, still bigger than the C version, but not by as much.*

Why would Java code strings this way when it eats up so much more space? The answer is security. The Java approach is to check the sizes of all strings and make sure that we never write outside of the allotted bounds. The C implementation is lighter and more compact, but it leads to all kinds of security problems when not used properly. We'll see this in Chapter 4 when we discuss buffer overflow exploits.

A Closer Look at Memory

Let's use what we've just learned about how computers represent information—in particular, the processor's native word size—to take a closer look at how memory works. We've said that a 32-bit processor will operate faster on a 32-bit value and will also access a 32-bit value in memory faster than a 16-bit value. An implication of this is that, even if the smallest addressable value in memory is 8 bits (remember the size of our memory drawers from Chapter 1?), a 32-bit processor can read 32 bits (4 bytes) from memory at once.

That simple sentence has some important concepts behind it. Let's go deeper.

A Simplified Hardware Memory Model

The memory system in a PC can be quite complicated, and it's beyond the scope of this book to go into all of the details, but discussing a simplified model of the memory system will be beneficial for us. It'll also make it easier for curious readers to dive deeper into this topic.

In its simplest form, a memory chip provides the following pins:

- A set of *address pins*, designated from `A0` up to `AN`, where `N` is one less than the system's native word size.
- A set of *data pins* designated from `D0` up to `DN`. Again, `N` is one less than the native word size.
- *Control pins* that allow the processor to command the memory. These include the `OE/CS` (output enable, or chip select) pin, which essentially activates the memory chip for input or output, and the $\texttt{WR}/\overline{\texttt{RD}}$ (write, read) pin, which indicates if we want to write into the memory or read values from memory.

NOTE *In digital electronics, a line over a signal name indicates negation, so* $\texttt{WR}/\overline{\texttt{RD}}$ *means that we want to write when the signal is a 1 and read when the signal is negated–that is, when it's a 0.*

The CPU has similar pins that are connected to the memory. Whenever the CPU wants to read a value from the memory, it puts the value of the address to access in its address pins, which are connected to the memory chip's address pins, usually by a bus (see the "Buses" box for more on what this means). Then the right control signal is exercised in the memory chip, in this case enabling the `OE/CS` pin and setting $\texttt{WR}/\overline{\texttt{RD}}$ to 0 in order to select a read operation. This triggers the memory to put the value from the indicated memory address into its data pins so the CPU can read it.

Remember, this is a simplified model. In reality, a lot more goes on when accessing memory.

BUSES

A simple *bus* is just a bunch of wires connecting all the chips in a computer system together. The only requirement is that the chips have some internal circuitry on their pins that allows the chip to produce a third state (in addition to the normal low and high states) in which the pins act as high-impedance gates. In this state, the pins show a very high resistance (impedance), and therefore current won't flow through them. Then you just need somebody to set all pins for all chips into that third state, except for the ones that will be active. That is, in essence, a bus.

What makes one bus different from another are the specific control signals and protocols the bus uses to interchange information. The S-100 bus was one of the first created for microcomputers. The Versa Module Eurocard (VME) bus was another popular bus used on more powerful machines. The Industrial Standard Architecture (ISA) was the selected expansion bus for the first PCs.

> Nowadays, most computers use Peripheral Component Interconnect Express (PCIe) for that purpose.

Figure 3-5 shows a basic diagram of the interface between the CPU and the memory chip, without including the control signals.

```
----+                        +-----------
    +- A0 ---------------  A0 -+
    +- ... -------------- ... -+
CPU +- A31--------------- A31 -+  Memory
    +- ...                     |
    +- D0 ---------------- D0 -+
    +- ... -------------- ... -+
    +- D31 -------------- D31 -+
----+                        +------------
```

Figure 3-5: The connections between the CPU and memory

The address and data pins on the processor and memory are connected by tracks in the motherboard. Activating and deactivating those pins is what makes the computer work. Let's use some concrete numbers to better understand this. Imagine that the `RIP` register (the instruction pointer) is set to `4`, meaning the instruction at address 4 is the next one to execute. The CPU has to read that instruction from memory, so it puts the value `4`, which in binary is `0b100`, on the address bus. Figure 3-6 illustrates the process.

```
----+                              +-----------
    +- A0 -------- 0 ----------  A0 -+  <--- Set address in data bus
    +- A1 -------- 0 ----------  A1 -+
    +- A2 -------- 1 ----------  A2 -+
    +- A3 -------- 0 ----------  A3 -+
    +- ... ------- 0 ---------- ... -+
CPU +- A31 ------- 0 ---------- A31 -+  Memory
    +- WR/RD  ---- 0 -------- WR/RD -+  <--- Read command
    +- D0 -------- 0 ----------  D0 -+
    +- D1 -------- 0 ----------  D1 -+
    +- ...                           |
```

Figure 3-6: Preparing for memory access by setting the address bus and activating the read signal

Only pin `A2` is set, which corresponds to address 4 in binary. Whenever the $\overline{RD}$ control signal is activated, the memory chip will access the value stored at address 4. *Activating* a pin means setting a given voltage on it. In this case, we need a 0, which usually means 0 volts. The voltage for a 1 depends on the technology used. It might be 5 V, 3.3 V, or even less.

Next, the memory chip puts the value from address 4 in its data pins so the CPU can read it. This is illustrated in Figure 3-7.

```
----+                                +-----------
    +- A0 -------- X ----------  A0 -+
    +- A1 -------- X ----------  A1 -+
    +- A2 -------- X ----------  A2 -+
    +- A3 -------- X ----------  A3 -+
    +- ... ------- X ---------- ... -+
CPU +- A31-------- X ---------- A31 -+  Memory
    +- WR/RD  ---- X -------- WR/RD -+
    +- D0 -------- 1 ----------  D0 -+ <--- Data is actually read
    +- D1 -------- 1 ----------  D1 -+
    +- ...                           |
```

Figure 3-7: Retrieving the value from memory through the data bus

As each memory position is 8 bits, we only need 8 physical pins to send the value from address 4 to the CPU. Since there are 32 data pins total, just reading the one address would be a huge waste. As such, the memory typically doesn't put just the 1 byte in the data lines: It puts as much as it can. In this example, that means it will output 4 bytes, starting from address 4, using the 32 data signals in the bus.

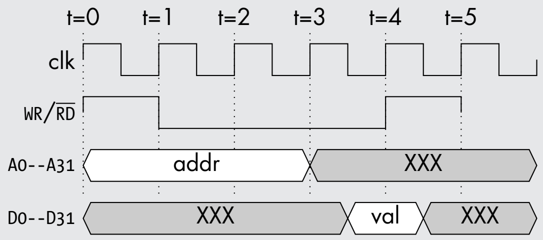
At this stage, the values of the address pins are no longer needed. Some processors take advantage of this and use the same pins to set the memory address and to write or read the values, as the address and values aren't set at the same time. We need to give some time to the memory to read the address and access the memory position.

Once again, note that this is a simplified example. In reality, there are a few more lines involved than we've discussed here. Also, sometimes the memory only has a data bus that's 8 bits long, and multiple memory chips are used to access 16-, 32-, or 64-bit words. In those cases, there's a real physical constraint with regard to memory-aligned accesses. These details can really make a difference between two systems.

For a more complete picture of how memory access works, you'll have to consult the datasheet of the particular processor involved. The memory system also has its own datasheet, and you have to check that too when building your hardware. The datasheets should include chronograms diagramming the exact timing of different memory transactions; see the "Chronograms" box for more on what these look like.

CHRONOGRAMS

The bus interactions between chips are usually represented as *chronograms* in the chips' datasheets. These are graphs that show all the processor lines and how they vary with time. Usually at the top you'll see the system clock signal, which orchestrates the activation and deactivation of each of the processor signals. These chronograms don't just provide information about the sequence of signals to activate but also indicate access time or other delays that may happen. For example, the following chronogram represents the memory read access described in the previous section:



At the top, you can see the system clock. Things happens when this signal changes, but the signals may be in the bus earlier or longer than the clock edges. Below the clock you can see the rest of the lines. The data and address buses are usually represented as a combination of lines, since showing each line individually wouldn't help and would make the diagram huge.

This particular chronogram tell us that:

- The processor puts the address value in the bus at t0.
- At t1, it sets the `WR/`$\overline{\text{RD}}$ signal low. This tells the memory to read the address internally and start to retrieve the data. The time the lines need to be set depends on the memory chip.
- After a while (that's the memory access time), the memory chip outputs the value to the data bus and the CPU can finally read the value.

Real chronograms show many more signals in order to allow the hardware designer to properly interconnect the different chips in the computer.

Native Word Size and Memory Alignment

I've already hinted that there's a relationship between a processor's native word size and the functionality of the rest of the computer. For example, a 32-bit processor will interface to a 32-bit memory system. (It's not that straightforward, but roughly that's what happens; when I say *system*, I may actually mean several chips.)

Imagine that you want to read just 1 byte. You put your desired address in the address lines and ask the memory chip to spit out the content of that address. The memory chip will put the content of the address you request plus the next three addresses on the data lines. The CPU will then read those 4 bytes from the bus but just take the lower 8 bits from that value. In a sense, that requires extra work than just reading the whole 32-bit value into a 32-bit register; the CPU has to consciously discard 3 out of the 4 bytes.

Long story short, a 32-bit processor will access *32-bit-aligned memory*, meaning chunks of memory starting from addresses that are multiples of 4: 0, 4, 8, 12, 16, and so on. Deviating from this alignment creates more work. For example, if we need to access a double word that starts at address 2, the processor will have to read addresses 0 through 3, then addresses 4 through 7, and then reorder the bytes from those two reads to get the single 32-bit value from addresses 2 through 5. This is why memory alignment is so important. Some processors will raise an exception when accessing unaligned memory. Others will just slow down. Still others may do both depending on what you're trying to do. And actually, unless you really try to do otherwise on purpose, the compiler will ensure that you access memory the right way.

What I just described is a very simplified example to illustrate how a processor works in a more efficient way when everything aligns with its native internal word size. The reality is more complex, but unless you need to design your own computer (the motherboard, at least), it doesn't really matter, especially with today's large cache memories. The main takeaway from all this is that it's always better to use the native processor word size and access data aligned to that size.

Little vs. Big Endian

We know that we have to always read data from memory using the native word size of the processor, and we also know that we should access memory in proper alignment with the processor's word size to make our programs efficient. So far, so good, but there's one more thing to take into account: How are the bytes of dwords, qwords, and other multi-byte pieces of data mapped between the memory and a processor's registers?

Let's assume the native word size of a given processor is 4 bytes (a 32-bit processor), and that we have the memory layout shown in Figure 3-8.

```
|  ...   | Drawer ...
+--------+
|  0x44  | Drawer p + 3
+--------+
|  0x33  | Drawer p + 2
+--------+
|  0x22  | Drawer p + 1
+--------+
|  0x11  | Drawer p
+--------+
```

Figure 3-8: A hypothetical memory layout

Let's assume, too, that we want to read the content of addresses `p` through `p + 3` into one of our 32-bit registers. Which value do you think we'll get in the register?

a) `0x44332211`

b) `0x11223344`

The answer is: It depends on the processor. If your processor is *little endian*, you'll get the first value. Otherwise, if your processor is *big endian*, you'll get the second value. Most of the processors we're using in this book are little endian, but ARM can be configured to work as big endian. MIPS processors are usually big endian, but you can find little endian hardware out there too. For the purposes of this book, we'll consider MIPS to be big endian and ARM to be little endian.

In general, you don't have to worry about whether your processor is big or little endian. You just write and read your values to/from memory and the processor will do the right thing. Endianness only becomes important when you have to interchange data between computers, as the sender and receiver may have a different endianness. This happens very often in network programming when using open protocols that have to work with any kind of machine. We'll see examples later in the book. For now, just keep in mind that different processors may read data in different ways.

But enough introduction. Let's see how all these concepts can be put to practical use by writing a traditional "Hello, world!" program.

"Hello, World!" in ASM

I'm pretty sure you're familiar with "Hello, world!" programs, which display the message "Hello, world!" in the console. Let's create a simple "Hello, world!" program in ASM. Then we'll see how the program translates to C.

The way to display a message like "Hello, world!" on Linux is to write it to the *standard output* (usually the console) using the `SYS_write` system call, which expects three parameters:

File descriptor Indicates where we want to write to. Overall a file descriptor represents a file, and in the Unix world, everything is a file. In this case, we'll need file descriptor `1`, which is always associated with `stdout`. We'll discuss file descriptors in more detail in Chapter 6. For now, you just need to know that if you pass `1` as the first parameter to `SYS_write`, you'll be writing to the console.

Buffer The memory address containing the data we want to write to the file descriptor provided in the first parameter.

Length The number of bytes we want to write, starting at the address provided as the second parameter.

Remember that you can always check section 2 of the manual to get all the details about any system call using `man 2 syscall`. Just remove the `SYS_` from the syscall name.

On the x86_64 architecture, the `SYS_write` system call is identified with a `1`. Knowing this, and applying everything we learned in Chapter 2 about invoking system calls, Listing 3-1 shows what our little program looks like.

```
    global _start
_start: mov rax, 1 ; SYS_write = 1
    mov rdi, 1     ; fd = 1
    mov rsi, msg   ; buf = msg
    mov rdx, 14    ; count = 14 (the number of bytes to write)
    syscall       ; (SYS_write = rax(1), fd = rdi, buf = rsi, count = rdx)

    ;;  Exit program
    mov rax, 0x3c  ; SYS_exit = 0x3c
    mov rdi, 0     ; status = 0
    syscall        ; (SYS_exit = rax , status = rdi)

msg:    db 'Hello, world!',0x0a
```

Listing 3-1: A "Hello, world!" program in ASM for Intel x86_64

I hope you can identify the two system calls in there. The first one writes the message, and the second one exits the program with status `0`.

Let's compile the program, following the same steps we used at the start of Chapter 2, then run it:

```
$ nasm -f elf64 -o hello.x86.o x86-hello.asm
$ x86_64-linux-gnu-ld -o hello.x86 hello.x86.o
$ qemu-x86_64-static ./hello.x86
Hello, world!
```

Once you verify that it works, there are some key new elements in our program that we need to discuss.

Labels and Assembler Commands

The first key element in our program is the `msg` label. A *label* is a name used to reference a part of a program (actually a memory address). In this case, `msg` references our "Hello, world!" message. In general, we don't know where in memory our program will be loaded, so using symbolic names like `msg` lets us write programs without caring about the details. Wherever the data representing "Hello, world!" ends up in memory, `msg` will refer to that location. In this way, labels can make programs easier to read and modify.

We've actually already seen labels at work in the past. Can you remember the other label we've been using so far? That's right, it's `_start`.

The second key thing to highlight is the `db` instruction paired with the `msg` label. This is an *assembler directive*, an instruction that's only understood by the assembler and doesn't directly translate into an opcode in the program. We've seen an assembler directive before. Anybody? Correct, it was `global`.

The `db` assembler directive, likely short for *define byte* or *data byte*, allows us to set some memory area with a sequence of bytes. In this case, our `db` instruction has two parts, separated by a comma. The first part is the string `'Hello, world!'`. The assembler will output one byte per character in the string, starting at position `msg`. The second part is an extra byte, expressed in hexadecimal: `0x0a`. Sure, you could just use the decimal value (`10`) instead and everything would stay the same. This is the ASCII code for a line feed, meaning it will move the cursor to the next line in the console. Try removing it and see what happens to your output. With `db`, you could also write your whole string as a list of the ASCII codes for each character, separated by commas, but that isn't very practical.

> **Exercise** Use the `SYS_unlink` system call (87) to write a program named *ghost* that will delete itself every time it's executed. For this exercise assume the name of the program is always *ghost*.

Pointers

There's one more key feature of our "Hello, world!" program that we need to highlight: a pointer.

Okay, don't panic. You may have heard that pointers are the trickiest part of low-level programming, but they're actually a lot simpler than you might think. Furthermore, you can't do much in ASM without using pointers. Just repeat to yourself that you have to understand pointers to create cool things like shellcodes. Take a deep breath, repeat it again, and let's dive in.

A *pointer* is just a position in memory that contains the address of another position in memory. Remember, a memory address is just a number. The contents of a given memory address is also just a number. It's therefore only natural that the number stored at a memory address could itself be interpreted as some other memory address.

One thing you may figure out from this recursive definition is that a pointer's size has to be equal to the number of address pins in the processor. In other words, it has to be the size of the address bus, also known as the processor's native word size. A 32-bit processor with a 32-bit address bus (for example, the Intel 386) will require 32 bits (4 bytes) to store any potential memory address a program could reference. A pointer on a 32-bit machine will therefore take up four consecutive positions in memory. (Remember, each position holds a single byte.) Likewise, for a 64-bit processor with a 64-bit address bus, a pointer will need 64 bits (8 bytes) to reference any possible memory address. A pointer on a 64-bit machine will therefore be stored in eight consecutive memory addresses. The entire range of memory addresses a processor can reference is known as the *addressing space*.

Let's look at an example. Imagine the memory layout shown in Figure 3-9.

```
|  ...   | p + 4 = 0x400004
+--------+
|  0x40  | p + 3 = 0x400003
+--------+
|  0x00  | p + 2 = 0x400002
+--------+
|  0x00  | p + 1 = 0x400001
+--------+
|  0x04  | p     = 0x400000
+--------+
```

Figure 3-9: A pointer in memory

On a 32-bit little-endian machine, the 4 bytes at memory addresses `p` (`0x40000000`) through `p + 3` (`0x40000003`) can be read together as having the value `0x40000004`. If we interpret this as a pointer—a memory address storing another memory address—we can say that address `0x40000000` points at (stores) address `0x40000004`.

What does all this have to do with our "Hello, world!" program? Taking into account that a register can be seen as a special kind of memory, whenever we store a memory address in a register, we can consider that register to be a pointer. With that in mind, have a look at this line from our ASM code:

```
mov rsi, msg
```

Here we store the value of `msg` in the `RSI` register. As we've already discussed, `msg` is just a label for the memory address where our message is stored. In this context, `RSI` contains a pointer to the message: It's a memory location (a register in this case) storing another memory location that we're interested in.

In general, whenever we need to pass some information that's too big to be stored in a register to a system call, we need to store that information somewhere else in memory and give the system call a pointer (which can fit in a register) to the relevant memory address. In our example, the string `"Hello, world!"` requires 13 bytes, plus an extra byte for the line feed, and our registers can only store 8 bytes on a 64-bit platform. In order to make the system call work for all cases and not just for datablocks of 8 bytes, `SYS_write` expects the data to write to be somewhere in memory, and to receive that location (a pointer) as a parameter via a register.

Strictly speaking, `RSI` is the pointer here, but it's more common to refer to the memory address stored in `RSI` as the pointer. The indirection is implicit in the *pointer* definition, and it's really the target address that we're interested in. In other words, even though technically the pointer is the address containing an address, in general, people call the address itself the pointer.

Most of the time, pointers will be true memory addresses and not registers, but the concept is the same. Some registers are actually named after this principle. The instruction *pointer*, the stack *pointer*, and the frame *pointer* are three of the most famous cases.

"Hello, World!" in C

Now let's see how to write the same "Hello, world!" program in C. We'll learn a few more things about the C language and its associated tools in the process. Listing 3-2 shows the code.

```
#include <unistd.h>

int main ()
{
  register void *p = "Hello, world!\n";
  write (1, p, 14);
  _exit (0);
}
```

Listing 3-2: A "Hello, world!" program in C

Again, notice the two system calls in the program, `write` and `_exit`. We already know that the second parameter to `write` has to be a pointer, a memory address containing the address, in memory, of the string to print. We pass the variable `p`, which we declare with the `register` keyword. As mentioned in Chapter 1, you don't typically use this keyword in C programs. Without it, the compiler will simply allocate a regular memory block of 8 bytes (for a 64-bit machine) in which it will store the pointer—the address where our message is located, in this case.

Let's take a look at the assembly generated by `gcc`. We'll create a non-position-independent executable, since that makes it easier to spot the pointer:

```
$ x86_64-linux-gnu-gcc -no-pie -fno-pic -o hello hello.c
$ x86_64-linux-gnu-objdump -d -M intel hello
--snip--
0000000000400544 <main>:
  400544:   55                      push   rbp
  400545:   48 89 e5                mov    rbp,rsp
  400548:   53                      push   rbx
  400549:   48 83 ec 08             sub    rsp,0x8
  40054d:   bb 5c 06 40 00        ❶ mov    ebx,0x40065c
  400552:   ba 0d 00 00 00          mov    edx,0xd
  400557:   48 89 de              ❷ mov    rsi,rbx
  40055a:   bf 01 00 00 00          mov    edi,0x1
  40055f:   e8 dc fe ff ff          call   400440 <write@plt>
  400564:   bf 00 00 00 00          mov    edi,0x0
  400569:   e8 c2 fe ff ff          call   400430 <_exit@plt>
--snip--
```

Skipping over the first four instructions in the resulting ASM (that's stack stuff, which we'll discuss in Chapter 4), try to find the pointer. Have you spotted it? Sure, look at how we set `EBX` (the 32-bit part of `RBX`) to `0x40065c` ❶, then copy `RBX` into `RSI` ❷. And what do you think `0x40065c` is? A memory address, of course! That once again makes `RSI` a pointer: a memory location holding the address of another memory location.

NOTE *The exact memory address you see being assigned to* `EBX` *in your output may vary, depending on your system configuration. Take note of that address, as you'll need it in a minute.*

To find out what our pointer is pointing to, let's introduce a new best friend, the GNU Debugger (`gdb`).

Using the GNU Debugger

A *debugger* like `gdb` is a program that allows us to execute other programs and look inside them. Do you remember in Chapter 2 when I said that the operating system makes sure that a process can't see inside the memory assigned to another process? Well, that's still correct. A debugger uses a special system call to do its work, so technically it's the OS looking into the other process's memory and then reporting what it finds to the debugger.

A debugger is able to load a binary in memory and process all its metadata (the ELF details added during compiling and linking), including the code itself and any data stored in the program. The debugger is then able to examine the contents of the memory associated with the program. Later we'll learn how to figure out the address where some data actually gets loaded, but for now we can just use the debugger to inspect the contents of arbitrary locations.

Technically the debugger launches the program to debug as a new process and after that gets attached to it using the `SYS_ptrace` syscall. With that syscall, the debugger can control the target process and access its memory, as well as the values of its registers. However, this only happens once we actually start *debugging* the program. Before that, the debugger just reads the file and uses the metadata stored in it to let us inspect parts of the program.

Let's use `gdb` to see what's at the memory address our pointer is pointing to (remember that the exact memory address you need to look at may vary):

```
$ gdb ./hello
(gdb) x/s 0x40065c
0x40065c:    "Hello, world!\n"
```

In `gdb`, the `x` command allows us to dump memory content to the console. Here it shows us that the memory location the pointer is referencing holds our "Hello, world!" message.

The `x` command must be followed by the type of data we want to see. In this case, we pass `s` for *string*, but you can, for instance, pass `i` for processor instructions, `a` for addresses, or `t` for binary. The command is a bit more powerful than this, but for now we don't need to know more. See the gdb manual if you want more details.

NOTE *If you aren't working on an x86_64 machine, you may have a problem using gdb directly on an x86_64 binary. See Appendix B for information on how to debug programs for other processors using qemu and gdb-multiarch.*

Declaring Pointers in C

As you may have already guessed looking at the code in Listing 3-2, pointers in C are declared using the * symbol. In our code, *p creates a pointer called p that references the memory location where the "Hello, world!" message is stored:

```
register void *p = "Hello, world!\n";
```

There's a little more to pointers than that, since in C we need to specify data types. In this case, the data type doesn't really make a difference, so we've used void, but in the general case the pointer type is important and useful. In all, a C pointer is therefore declared this way:

```
type *pointer_name;
```

This declares a pointer called *pointer_name* to a memory address containing a value of a certain data type (*type*). Which types does C know? Table 3-1 provides a list.

Table 3-1: C Data Types

C type	Data type	Description
char	Byte	Minimal addressable element (not necessarily 8 bits)
int	Integer	Default integer type
short	Integer	Usually half the size of the default integer
long	Integer	Usually double the size of the default integer
float	Floating point	Single-precision floating point
double	Floating point	Double-precision floating point
void	Nothing	Nothing or anything

C also supports compound types and allows us to define our own new types, but for the time being, Table 3-1 has all we need to know.

CHAR VS. BYTE

In C, a char is defined as a byte, and a byte is defined by the C standard as the size required to store a character in the given platform. That may sound circular, but it opens up the possibility that a byte could be something other than 8 bits. In fact, it was once common to see 6-bit bytes on machines with native word sizes of 12, 18, 24, and so on up to 60, representing 2 to 10 6-bit bytes.

Nowadays it's very strange to find any system where the size of the byte type isn't 8 bits (an *octet*). For the rare exceptions, C defines the `CHAR_BIT` constant, which indicates how many bits define a character for the given platform. Some cases where you may find different byte sizes are digital signal processors (DSPs). For example, the Texas Instruments C54x and TMS320 DSPs define `CHAR_BIT` as 16 bits. Also, when you look into old machines like PDP or CDC machines or even mainframe architectures, you'll find values like 5, 7, or 9.

For your reference, Listing 3-3 shows a simple program to figure out the size of each type in your system and better understand the difference between all those types.

```
#include <stdio.h>
int main ()
{
❶ printf ("Size of void*  : %ld\n", sizeof(void*));
  printf ("Size of short  : %ld\n", sizeof(short));
  printf ("Size of int    : %ld\n", sizeof(int));
  printf ("Size of long   : %ld\n", sizeof(long));
  printf ("Size of float  : %ld\n", sizeof(float));
  printf ("Size of double : %ld\n", sizeof(double));
  return 0;
}
```

Listing 3-3: Showing the sizes of the main C types in bytes

This program uses the `sizeof` operator, which returns the size, in bytes, of a given data type or variable. We display the results with the `printf` function, short for *print formatted* (see the "Format Strings and Type Conversion" box for more on how this works). Notice in particular how we check the size of `void*` ❶. This is a pointer to `void`, meaning a raw pointer, or a pointer to anything. That's what we used in our "Hello, world!" program, and it's the C equivalent of the pointer we used in our original ASM code. If you compile and run the test program in Listing 3-3, it will show you that `void*` has a size of 4 bytes in a 32-bit platform, or 8 bytes in a 64-bit platform.

FORMAT STRINGS AND TYPE CONVERSION

The `printf` function used in Listing 3-3 lets us print messages using *format strings* to compose complex outputs. In this case, we use the `%ld` format string to print the long value returned by `sizeof`. This basically tells `printf`, "I have a number here that I want you to convert into a string. Please do that for me."

The idea of type conversion may be confusing if you're used to high-level programming languages like JavaScript, where many transformations are done automatically, but it's a necessary part of low-level programming. Anything printed to the console, even a number, needs to be provided as a string—that is, a sequence of characters. Internally, though, numbers and characters are represented differently, so some conversion needs to take place prior to printing.

Take the number 123. As an integer, it can be stored in just 1 byte, but to print it out you need to convert it to individual characters: `1`, `2`, and `3`. For our purposes, C represents characters using ASCII codes, each of which requires 1 byte. As a string, the number 123 therefore becomes the byte sequence `0x31, 0x32, 0x33, 0x00`. That's 4 bytes: one for each digit, plus the final `0x00` to indicate the end of the string. Luckily, the `%ld` format string handles all that for us.

There are lots of other `%` format codes you can use with `printf`. Check the `printf` man page for details about the options available. Note also that `printf` is defined in *stdio.h*, which is why we include that file at the start of Listing 3-3.

Writing Your First Shellcode

A *shellcode*, in its simplest form, is a piece of code that fires up a shell. It's usually fed into a vulnerable program using an exploit, effectively enabling the attacker to acquire a shell with the same privileges as the vulnerable program. The attacker typically targets processes running as `root` to get full access to the machine.

We'll leave exploiting a vulnerable program to Chapter 4, but believe it or no you've already learned all the bits and pieces to write a very basic shellcode. Just like a "Hello, world!" program, all it takes is a pointer and the right system call.

Our shellcode won't be usable in the wild, but it's still instructive to learn about. If you're curious, the problem is that shellcodes usually make it into memory through a standard C function that copies strings. As we've discussed, C strings are delimited with the value `0x00`, but that value will also appear as part of our shellcode, which will prevent the string copy function from copying the whole program. Once you understand how the shellcode works, you can read the classic article "Smashing the Stack for Fun and Profit" by Aleph One to learn more about this problem and some tricks for avoiding it.

In the coming sections we'll look at how to construct a shellcode for each of our hardware platforms of interest, starting with x86_64.

x86_64

In Linux, you can execute a process (such as starting a command shell) using the `SYS_exec` system call, which on the x86_64 platform has a number of `59` (`0x3b`). This system call takes three parameters, but for now we need only the first one, a pointer to the name of the program to run. We'll just set the other parameters to `0`. Listing 3-4 shows our basic shellcode.

```
section .text
```

```
    global _start

_start:
    mov rax, 0x3b          ; SYS_exec
    mov rdi, cmd           ; char *cmd
    mov rsi, 0             ; No argv
    mov rdx, 0             ; No env

    syscall

cmd:    db '/bin/sh',0
```

Listing 3-4: A shellcode for Intel x86_64

Notice how similar this is to our "Hello, world!" program. We assign a label (`cmd`) to a string, but this time our string is `'/bin.sh'`, the name of the shell we want to launch, instead of a message to print. Then we create a pointer to this string in a register (`RDI`) and invoke a system call—this time `SYS_exec` instead of `SYS_write`. We no longer need the `SYS_exit` system call, since `SYS_exec` is another one of those system calls that doesn't give control back to the calling process. See its man page (`man 2 exec`) for more on this.

HOW PROCESSES ARE EXECUTED

When we run a program from the command line, we usually specify the name of the binary to run plus a set of arguments. Such a request doesn't go straight into the kernel but rather into the command line interpreter, which somehow has to use the kernel's services to get a new process running. For that, it does two things: It creates a process (we'll see how to do that in Chapter 6), and then it loads the requested binary. This second step is done with the `SYS_exec` system call, which has the following prototype:

```
int execve(const char *pathname, char *const _Nullable argv[],
                char *const _Nullable envp[]);
```

When the kernel runs the `SYS_exec` system call, it loads the binary in memory according to the ELF metadata and sets up a special memory area known as the *stack* (which will become your best friend in Chapter 4) with the two last parameters, which represent the command line arguments and the environment variables. Both of these parameters are in the command line interpreter's process, and the kernel is the only one that can copy those values into the address space of the new process.

When the new process starts execution on its entry point, the command line arguments and environment variables, initially only available in the command line, are now also available to the process via the stack. From this point on, it's up to the process what to do with that information. For C programs, the C runtime code in *crt1.o* will process this information to be provided to the `main` function as three parameters using the proper ABI, so the C program can easily access the information if needed. With that said, do you remember that I told you in Chapter 2 that there's a third way of declaring `main`? Here you go:

```
int main (int argc, char *argv[], char *envp[])
```

This style of declaration gives `main` access to the environment variables as well as the command line arguments. And now you also know where that third parameter comes from. Yep, the `SYS_exec` system call.

You may be wondering why the shell we launch is called */bin/sh*. That doesn't match the name of any of the shells you may have heard of, like bash, dash, zsh, or ksh. And that's just it: There are so many different shells out there that just about any Unix system keeps a symbolic link to the preferred shell, whatever it may be, at */bin/sh*.

The system has to run lots of shell scripts all the time. For example, there are shell scripts that are executed during the boot process, whenever you start or stop a service, and when you launch some applications. If those scripts explicitly referenced a particular shell, imagine what would happen if the user wanted to change their default shell. The system would have to update all those scripts, to say nothing of the ones you wrote on your own that the system knows nothing about. That would break your system badly, as well as force all system users to use the same shell. This is why most Unix systems have a binary at */bin/sh* that runs the user's preferred shell. Shell scripts rely on the existence of that file to work, regardless of the particular shell used.

ARM32

We should be able to port our x86_64 ASM shellcode to ARM32 very easily. Listing 3-5 shows how it looks.

```
.text
.globl _start

_start:
    mov r7, #11  @ SYS_execve

    ldr r0, =cmd @ Command
    mov r1, #0   @ No env
    mov r2, #0   @ No argv

    svc #0
cmd:
    .asciz "/system/bin/sh"
```

Listing 3-5: A shellcode for ARM32

This code is for Android, which deviates from the standard Linux folder structure in that the default shell is no longer at */bin/sh* but rather at */system/bin/sh*. If you're going to test the code on another ARM platform such as a BeagleBone Black, a BananaPi, or an Olinuxino running a standard Linux distro (usually Debian), or if you're going to use `qemu` in your development container, just change the string to the well-known location */bin/sh*. The rest of the code should just work.

We can compile the program like this:

```
$ arm-linux-gnueabi-as -o arm_sc.o arm_sc.s
$ arm-linux-gnueabi-ld -o arm_sc arm_sc.o
$ arm-linux-gnueabi-readelf -S arm_sc | grep ".text"
  [ 1] .text            PROGBITS        00010054 000054 000020 00  AX  0   0  4
```

This produces a shellcode of 32 bytes (`0x20` in hexadecimal).

Let's take a closer look at the code. First, notice how we use `.asciz` to add the shell location to memory instead of `db`. This is a GNU Assembler instruction, and it's available independent of the processor. It's similar to NASM's `db` instruction, but it automatically adds the requisite zero at the end of the string so we don't have to do that explicitly (that's what the `z` in `asciz` means).

Second, notice that we have to use `ldr` to load our pointer into register `r0`. When receiving a 32-bit immediate value (like the memory address referenced by `msg`), `ldr` is an ARM pseudo-instruction. The bottom line is that you can't directly load a 32-bit value into a register in an ARM processor. I won't go into the details, but roughly, ARM produces 32-bit machine code for all instructions (there's another mode for 16 bits, but we won't talk about it for now). This limits the size of the values that can be directly loaded into a register by one of those 32-bit machine code instructions: We would need the whole 32 bits for the value, so there wouldn't be any bits left to indicate which instruction we want to use the value with.

To get around this, the assembler expands the `ldr` pseudo-instruction into the right sequence of actual instructions to load the 32-bit value in a register. You'll find a similar workaround in other RISC processors since, like ARM, they're designed to use a fixed size for instructions. CISC processors like Intel allow a variable number of bytes per instruction, so they don't have this problem.

NOTE *It's possible to directly load small values (up to 12 bits) using `ldr`. The assembler will decide, depending on the value you want to load, whether to use the immediate value or treat `ldr` as a pseudo-instruction and expand it.*

If you're curious, you can use `objdump` to see the actual machine code that `ldr` produces:

```
$ arm-linux-gnueabi-objdump -d arm_sc

arm_sc:     file format elf32-littlearm


Disassembly of section .text:
```

```
00010074 <_start>:
   10074:       e3a0700b        mov     r7, #11
   10078:       e59f0008        ldr     r0, [pc, #8]    ; 10088 <_start+0x14>
   1007c:       e3a01000        mov     r1, #0
   10080:       e3a02000        mov     r2, #0
   10084:       ef000000        svc     0x00000000
   10088:       0002008c        .word   0x0002008c
```

The `ldr` instruction we wrote that loads a label (a 32-bit pointer) directly in the register is changed into an `ldr` instruction that uses indirect PC-relative addressing to get the pointer. The assembler will store the pointer to `msg` near the instruction (at address `0x10088` in this example), and then will load the register `r0` by reading the 32-bit value from that nearby address based on an offset from the current address. Because the address is nearby, the offset value will be a small number (in our example, it's `8`), much smaller than the 32-bit number we want to load into the register, and small enough to fit as part of the 32-bit machine code. You'll see this pattern all over the place in code for the ARM platform.

Exercise Use `gdb` to check that the pointer in the ARM code actually points to the shell path.

A Short Digression on Addressing Modes

Processors provide lots of different methods for accessing values and memory locations. These are known as *addressing modes*. So far we've mostly been using *immediate addressing*, where we just provide literal values in the assembly code. Here are examples for x86_64 and ARM:

```
mov rax, 0   ; x86_64
mov r1,  #0  ; ARM 32 bits
```

This loads the literal value `0` into register `RAX` or `r1`.

Immediate addressing is the most basic addressing mode, but we need more than that. For now, we'll introduce three other modes. Others will come as we need them.

When the second operand to `mov` is a register rather than a literal value, this is called *direct addressing* or *register direct*. In effect, this copies the value from one register into another register.

The second most used addressing mode is *indirect addressing*. This mode is used when we want to access the content of a memory address. In this case, the address is provided between parentheses or square brackets (it depends on the assembly syntax used). This addressing mode is the one that allows us to work with pointers (memory positions that contain an address to another memory position). Here are some examples:

```
                    ; (Intel)
mov rax, (0x600200) ; Loads in RAX the content of address 0x600200
                    ; (ARM)
```

```
mov r0,  [0x20008c]  ; Loads in r0 the content of address 0x2008c
```

In these examples, we specify the memory address literally, but it's more common to reference a register containing the desired memory address (or pointer), like so:

```
                     ; (Intel)
mov rdi, 0x600200
mov rax, (rdi)       ; Loads in RAX the content of address 0x600200
                     ; (ARM)
ldr r1, =0x2008c
mov r0, [r1]         ; Loads in r0 the content of address 0x2008c
```

Finally (for the time being), certain processors support *PC-relative addressing*, where the address we want to access is referenced as an offset from the address stored in the program counter (PC). This is the mode ARM's `ldr` pseudo-instruction uses to load a 32-bit value into a register using a 32-bit opcode, as we discussed in the last section. Let's go back to that example:

```
10078:  e59f0008 ldr     r0, [pc, #8]    ; 10088 <_start+0x14>
--snip--
10088:  0002008c .word   0x0002008c
```

Remember that brackets indicate we're reading the content of a memory address, so the first line says, "Read the value stored at the address located 8 bytes (`#8`) away from where the program counter is right now." For ARM processors, "where the program counter is right now" is always 8 more than the address of the current instruction. In other words, `pc` is pointing two instructions ahead (remember that all ARM32 instructions are 32 bits or 4 bytes). Most processors just point to the next instruction, but for ARM it's different: ARM was originally designed as a three-stage pipeline (fetch-decode-execute), similar to the one we saw in Chapter 1, and the designers decided that `pc` should point to the instruction in the fetch stage. That means that, when executing an instruction, we've already fetched the next two.

In our example, the instruction is at address `0x10078`, so `pc` is at `0x10078 + 8`, or `0x10080`. The value we want is 8 bytes ahead of `pc`, and `0x10080 + 8` is `0x10088`. Therefore, `0x10088` is where the actual pointer we want to load into `r0` is stored (`0x0002008c` in this case).

In ARM jargon, the list of values used together with the `ldr` instruction to load full 32-bit values is known as the *literal pool*, and it's generated automatically by the assembler somewhere close to the code that needs access to those values. You usually won't have to worry about this, as assemblers and compilers will take care of it for you, but it's good to know how it works.

AArch64

The code for AArch64 is basically the same as for ARM32. We just have to use the correct registers and a different syscall number, as shown in Listing 3-6.

```
.text
.globl _start
_start:
    mov w8, #221
    mov x1, #0
    mov x2, #0
    ldr x0, =cmd
    svc #0

cmd:
    .asciz "/bin/bash"
```

Listing 3-6: AArch64 shellcode

No surprises here. You can compile the program the usual way:

```
$ aarch64-linux-gnu-as -o aarch64_sc.o aarch64_sc.s
$ aarch64-linux-gnu-ld -o aarch64_sc aarch64_sc.o
$ aarch64-linux-gnu-readelf -S aarch64_sc | grep -A1 ".text"
  [ 1] .text             PROGBITS         0000000000400078  00000078
       0000000000000028  0000000000000000  AX       0     0     8
```

This produces a shellcode of 40 bytes (`0x28` in hexadecimal).

MIPS32

The shellcode for MIPS32 is pretty similar to the ARM version. Listing 3-7 shows how it looks.

```
.text
.globl __start
__start:
    li $2, 4011

    la   $4, cmd
    move $5, $0
    and  $6, $0, $0

    syscall

cmd:
    .asciz "/bin/bash"
```

Listing 3-7: A shellcode for MIPS32

We already know that MIPS uses different register names and syscall numbers, but there are some other details to highlight. First, as with ARM, MIPS opcodes are all 32 bits long, and therefore you can't directly encode an instruction that loads a 32-bit value. Also like ARM, MIPS provides pseudo-instructions as a workaround: `li` and `la`. They load smaller, more manageable parts of the 32-bit value into different registers, then combine those parts together.

To illustrate, say we have the following `li` pseudo-instruction:

```
li $2, 0x11223344
```

The assembler will convert this to the following two instructions:

```
lui $1, 0x1122
ori $2, $1, 0x3344
```

The `lui` instruction (load upper word immediate) loads the upper word of register `$1` with `0x1122`, so `$1` will hold `0x11220000`. If you look back at Table 2-4, you'll see that register `$1` is reserved for pseudo-instructions. Next, the `ori` instruction (OR immediate) takes the bitwise OR of register `$1` and the value `0x3344`, storing the result in register `$2`. This operation compares the two values bit by bit, yielding a `1` for bits where either value (or both) has a `1`. In the end, we get the 32-bit value we want, `0x11223344`, in register `$2`. The `la` pseudo-instruction is similar to `li`, but instead of using `ori`, it uses `addiu`.

In our shellcode, we use `la` to store our pointer in register `$4`. Then we have to set the other two syscall parameters to `0`. We could just use `li`, but instead we zero registers `$5` and `$6` using MIPS's special `$0` or `$zero` register, which is always set to `0`. I've shown two ways of doing this: First, we simply use `move` to copy register `$zero` into register `$5`, then we perform an AND of register `$zero` and itself, storing the result in register `$6`. (You could also XOR register `$6` with itself.) Remember that the `$zero` register exists when you analyze MIPS code.

Just to get comfortable with the compilation process, here's a reminder of the command line instructions to compile the MIPS version:

```
$ mips-linux-gnu-as -o mips_sc.o mips_sc.s
$ mips-linux-gnu-ld -o mips_sc mips_sc.o
$ mips-linux-gnu-readelf -S mips_sc | grep ".text"
  [ 3] .text         PROGBITS        004000d0 0000d0 000020 00  AX  0   0 16
```

The MIPS shellcode is 32 bytes long (`0x20` in hexadecimal).

MIPS64

The shellcode for MIPS64 is almost identical to the MIPS32 version. In Listing 3-8, instead of using the register numbers we use the register names, which greatly improves readability. The syscall number is also different.

```
.text
.globl __start
```

```
__start:
    dli $v0, 5057
    dla $a0, cmd
    dli $a1, 0
    dli $a2, 0

    syscall
cmd:
    .asciz "/bin/sh"
```

Listing 3-8: A shellcode for MIPS64

We've swapped `li` and `la` for `dli` and `dla` (the `d` stands for *double*) because now our registers can load up to 64 bits. These are also pseudo-instructions working the same way we just described.

You can compile the program using these commands:

```
$ mips64-linux-gnuabi64-as -o mips64_sc.o mips64_sc.s
$ mips64-linux-gnuabi64-ld -o mips64_sc mips64_sc.o
$ mips64-linux-gnuabi64-readelf -S mips64_sc | grep -A1 ".text"
  [ 3] .text             PROGBITS         00000001200000f0  000000f0
       0000000000000030  0000000000000000  AX       0     0     16
```

This produces a slightly bigger file than the 32-bit version: 48 bytes (`0x30` in hexadecimal). Let's take a look at the resulting machine code:

```
$ mips64-linux-gnuabi64-objdump -D mips64_sc
--snip--
Disassembly of section .text:

00000001200000f0 <__start>:
   1200000f0: 240213c1  li v0,5057
   1200000f4: 3c040000  lui a0,0x0
   1200000f8: 3c012000  lui at,0x2000
   1200000fc: 64840001  daddiu a0,a0,1
   120000100: 64210118  daddiu at,at,280
   120000104: 0004203c  dsll32 a0,a0,0x0
   120000108: 0081202d  daddu a0,a0,at
   12000010c: 24050000  li a1,0
   120000110: 24060000  li a2,0
   120000114: 0000000c  syscall

0000000120000118 <cmd>:
   120000118: 2f62696e  sltiu v0,k1,26990
   12000011c: 2f736800  sltiu s3,k1,26624
```

In this case, `dli` is just translated to an `li` because the value is small enough to fit in the instruction. The `dla` pseudo-instruction gets expanded to a few operations, though. Let's figure out what's going on here.

The `cmd` string is stored at address `0x120010118`, as you can see in the `objdump` output. In case you don't believe it, try the following:

```
$ xxd -r -p
2f62696e2f736800
/bin/sh
```

`xxd` allows us to do hex dumps of data, but it can also do the reverse operation using `-r` (for reverse). The `-p` option shows the result immediately. Just type the hexadecimal sequence you want to convert and press ENTER. Then press CTRL-C when you're done.

Here again is everything the `dla` pseudo-instruction actually does:

```
lui a0, 0x00        ; a0 = 0x0
lui at, 0x2000      ; at = 0x0000000020000000
daddiu a0,a0,1      ; a0 = 0x0000000000000001
daddiu at,at, 0x280 ; at = 0x0000000020000118
dsll32 a0,a0,0      ; a0 = 0x0000000100000000
daddu a0, a0, at    ; a0 = 0x0000000120000118
```

We'll break this down step by step:

1. As we already know, `lui` loads an upper word (16 bits). In the second line, it allows us to set bits 31–16 in the lower part of the 64-bit register `at`.
2. The `daddiu` instruction performs an unsigned addition with a given value (the `i` is for *immediate*).
3. The `dsll32` instruction is a little tricky. On MIPS32, `sll` performs a left shift of a given number of bits, but the 64-bit version performs a left shift of 32 plus the value of the third operand. In this case, since the third operand is `0`, we shift left by 32 bits, meaning the lower 32 bits of the 64-bit `a0` register is moved into the higher 32 bits of the register.
4. The `daadu` instruction is similar to `daddiu`, but it uses only registers and no immediate values.

After all that, we get the memory address we want, `0x120010118`, in register `a0`. Jumping through all these hoops is common for a RISC processor like MIPS. The number of available instructions is reduced (that's what RISC means), so it's normal to use several instructions to do relatively simple things. In general, you don't need to care about everything happening under the hood when you use a pseudo-instruction. However, if you're looking to save some bytes here and there, be aware that a single `dla` instruction actually takes up six 32-bit words, or 24 bytes.

As an extra step, try compiling the MIPS 32-bit code with the MIPS 64-bit compiler. You may get some warnings about not using the 64-bit versions of instructions (`daadu`, for example), but you'll get the same result in the end. MIPS was designed that way: MIP32 code can be used directly for MIPS64 machines. The actual difference between the architectures (leaving aside coprocessor and supervised mode) is that registers are 64 bits for MIPS64 and 32 bits for MIPS32. Even instructions are 32 bits long for MIPS64. As such, I won't include any more code for MIPS64 in this book, as it will be almost identical to the MIPS32 version. We'll just talk about MIPS32, which we'll refer to simply as MIPS from this point on.

RISC-V

The RISC-V version of our shellcode is quite straightforward now that we've gone through the ARM and MIPS versions. Listing 3-9 shows the code.

```
.text
.globl _start
_start:
    li a7, 221 # SYS_execve
    la a0, cmd
    li a1, 0
    li a2, 0

    ecall

cmd:
    .asciz "/bin/sh"
```

Listing 3-9: A shellcode for RISC-V 64 bits

You can compile this program using the following instructions:

```
$ riscv64-linux-gnu-as -o riscv64_sc.o riscv64_sc.s
$ riscv64-linux-gnu-ld -o riscv64_sc riscv64_sc.o
$ riscv64-linux-gnu-readelf -S riscv64_sc | grep -A1 ".text"
  [ 1] .text             PROGBITS         00000000000100b0  000000b0
       0000000000000020  0000000000000000  AX       0     0     4
```

As usual, the code is almost identical to MIPS (it even has the same size of 32 bytes), and as with the ARM and MIPS versions, `la` is a pseudo-instruction. Let's take a quick look at the generated assembly:

```
$ riscv64-linux-gnu-objdump -D riscv64_sc

riscv64_sc:     file format elf64-littleriscv


Disassembly of section .text:

00000000000100b0 <_start>:
   100b0: 0dd00893          li a7,221
   100b4: 00000517          auipc a0,0x0
   100b8: 01450513          add a0,a0,20 # 100c8 <cmd>
   100bc: 00000593          li a1,0
   100c0: 00000613          li a2,0
   100c4: 00000073          ecall

00000000000100c8 <cmd>:
   100c8: 6e69622f          .word 0x6e69622f
   100cc: 0068732f          .word 0x0068732f
```

The `li` (load immediate) instructions we wrote in the source code comes through unchanged, but the `la` pseudo-instruction is substituted with a combination of `auipc` and `addi`. The first instruction, `auipc` (add upper immediate to program counter), is the way RISC-V implements PC-relative addressing. It sets the target register to the sum of the current program counter and an immediate value shifted 12 bits left. In other words, it adds the immediate value shifted 3 nibbles (3 hexadecimal digits) left:

```
0x100b4 auipc a0, 0x0   (pc = 0x000100b4)
                         a0 = pc + (0x0 << 12) = pc = 0x000100b4
0x100b8 add a0, a0,20    a0 = 0x000100b4 + 0x14 (20) = 0x000100c8
```

Note that when we reference data that's nearby, `auipc` will use `0x00` as a parameter, in which case it just becomes a way to load the value of `pc` into a register. Then the small offset (`0x14`, or 20, in this case) is handled by the subsequent `add` instruction.

In general, the program counter points to the next instruction to be executed, but for RISC-V's `auipc`, it actually contains the address of the current instruction. That is, we can assume that the instruction hasn't been executed yet when we're about to execute `auipc`, so the program counter hasn't been incremented yet. Usually we wouldn't care about these 4 bytes more or less—the assembler will handle that for us—but it's good to know that there may be small differences between platforms when calculating offsets for different instructions.

Conclusion

In this chapter we had our first encounter with pointers in ASM and C, and we learned how to use them together with a system call. We also had our first contact with addressing modes. Using these concepts, we managed to create a "Hello, world!" program, which coincidentally was almost identical to a very basic shellcode.