C for Exploitation

An Introduction to Low Level C

C is an interesting language because it is the foundation of most operating systems. Many servers and low level systems libraries are written in it, kernels are written in it, and lots of higher-level languages compile down to something compatible with the C ABI. Exploitation often involvs taking advantage of assumptions that are not true at the low level, so it is essential that you have a solid understanding of C at its lowest level.

Data Types

(Assuming 32-bit Linux)

Type	Width (bytes)
bool	1
char	1
short	2
int	4
long	4
long long	8
float	4
double	8
pointer	4
instruction	(variable)

Negative Numbers

How do we represent negative quantities in binary?

There are three primary ways to represent negative numbers:

- Sign + Magnitude: Not used in real hardware
- One's Complement: Invert all the bits
- Two's Complement: Invert all the bits, and add 1

Of these, two's complement is the most common. One's complement negation is sometimes used with boolean values, though.

Why use two's complement?

Because it makes math easy! Let's have a look at -1 + 1 = 0:

- -1 == two's complement of 0001 == 1110 + 1 == 1111
- 1111 + 0001 = 0000 (with overflow), which we expect
- The interested can find a proof here

Signed vs. Unsigned

Usually, when dealing with integers to perform computations on data, we want to be able to represent negative quantities.

However, when negative values do not make sense, we can choose to force the computer to interpret the value as a positive number. To do so we use the unsigned integer type (e.g. unsigned int).

Examples of when to use unsigned types:

- Indexes into an array
- Number of bytes to read
- The size of a buffer
- Representing raw, untyped binary data

A Note on Signed/Unsigned

Security Concerns

Mishandling signed and unsigned data can cause security vulnerabilities because of the differences in range. For a 1 byte integer, there are 256 different values:

- signed char: -128 to +127
- unsigned char: 0 to +255

Consequently, signed values -128 to -1 are represented the same way as unsigned values from +128 to +255.

This can cause problems when programmers treat signed data as unsigned or vice versa.

Endianness

Big vs. Little Endian

There are two ways of ordering the bytes on a computer: big endian and little endian.

- Big Endian: Most Significant Byte (MSB) first
- Little Endian: LSB first

For example, the byte sequence "\x01\x00" represents 0x0100 (256) on a big endian machine and 0x0001 (1) on a little endian machine.

A side effect of little endian is that converting a 32-bit integer to a 16- or 8-bit integer (or 16- to 8-bit) involves ignoring the bytes on the right side, not on the left. This makes the machine code for expressions like short s = *pointer_to_int; simpler, since you don't need to add an offset to the address.

x86 is a Little Endian Architecture!

I promise you that this **will** mess you up at least once when you're writing an exploit!

Shifts and Bitwise Operations

Conceptual

Shifts simply move all of the bits to the right or the left, dropping what falls off the end and filling in with zeros. The exception to this rule is when right-shifting a signed number. In this case, the computer checks if the number is negative by looking at the most significant bit. Positive numbers are filled in with zeros, and negative numbers filled in with ones. This process is called *sign extension*

- Right Shift: >>
- Left Shift: <<

Bitwise operations apply a logical operation (not, and, or, xor) to every bit in order.

- Not: ~
- And: &
- Or: |
- Xor: ^

Shifts and Bitwise Operations

Examples

- 1100 << 1 == 1000
- 1100 (unsigned) >> 1 == 0110
- 1100 (signed) >> 1 == 1110
- ~1100 == 0011
- 1100 & 0000 == 0000
- 1100 & 1111 == 1100
- 1100 | 0000 == 1100
- 1100 | 1111 == 1111
- 1100 ^ 0000 == 1100
- 1100 ^ 1111 == 0011

Floating Point Representation

Floating point numbers are represented according to the standard IEEE-754. That means nothing to you. Since it isn't critically important to you right now, we're going to wave our hands at it. Those interested should go to the Wikipedia page.

Sufficeth to say it is quite different from how integers are represented.

Type Casts

Types of Casts

There are two types of casts:

- Normal Casts: float f = (float)some_int;
- Binary Reinterpretation Casts: float f = *(float*)&some_int;

Needless to say, the two are quite different. The first will do a proper conversion, while the second will copy the raw bit pattern.

In addition, there are multiple types of normal casts:

- Integer<->Integer Casts: Narrowing
- Integer<->Integer Casts: Widening
- Integer<->Float Casts
- Pointer Casts (which result in reinterpret-casts of the data they point to)

Type Casts

Integer<->Integer Casts: Narrowing

Narrowing is casting a larger integer type down to a smaller integer type (e.g. int to short).

To narrow to n bytes, the machine will take the n least significant bytes and store them into the result. Thus, (char)257 == 1.

As shown in the above example, this can cause strange results when narrowing to a type that cannot hold the run-time value of the variable being cast. Narrowing casts are *only* safe when the run-time value can be represented in the target type. Narrowing in other circumstances causes unintuitive results.

Type Casts

Integer<->Integer Casts: Widening

On the other hand, the opposite (widening) is always valid for unsigned->signed casts and casts with no change in signedness. In these cases, the n bytes of the original variable are copied to the least significant bytes of the target. If the target is unsigned, zeros are added to the MSBs, and if it is signed then they are subject to sign extension.