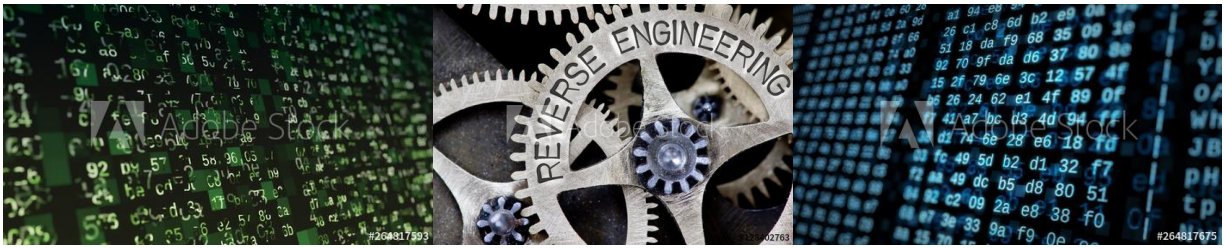

Guide to Software Reverse Engineering

Table of Contents

Section 1: Introduction to Software Reverse Engineering

Are you the type of person who likes to take things apart? Perhaps you have tinkered with an old engine, breaking it down and fixing components before building it again. You may be fascinated by the hidden codes that create form and function. For instance, you may find it eerie or inspiring how DNA provides the blueprint behind life-forms on Earth.

Similarly, with software reverse engineering, you take apart an object or search for hidden code within applications. Software reverse engineering is the process of extracting engineering or designing knowledge from a human-made object or product, often by breaking that product into its formative components or structural codes.

Figure 0.2 shows how **forward engineering** moves from the identification of requirements, to the design of the application, to the coding and testing and, finally, implementation.

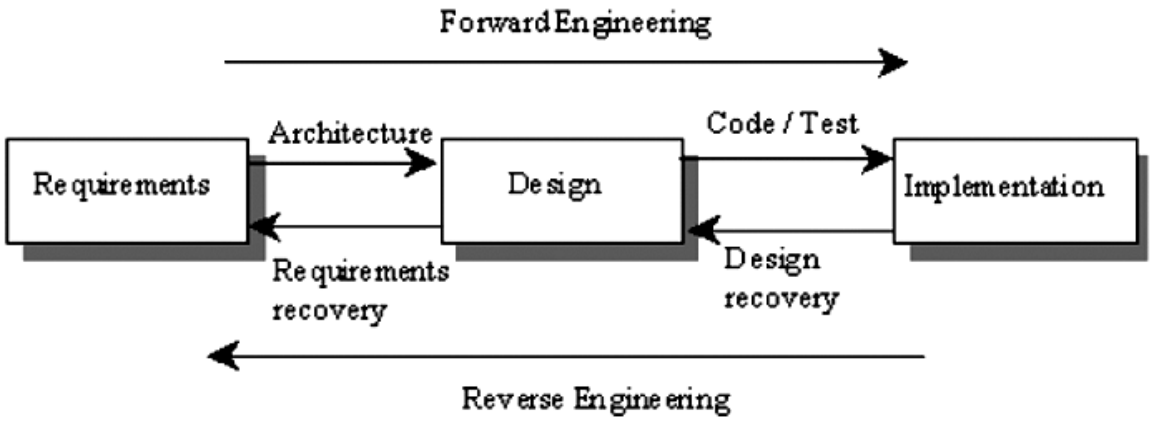

Figure 0.2: Forward and Reverse Engineering
Source: Nilesh Jadav, "How to reverse engineer using advanced ApkTool."
Retrieved from www.c-sharpcorner.com/article/how-to-reverse-engineer-using-advanced-apk-tool/

Software reverse engineering begins with the implemented application and moves "backward" to discover its basis in code; underlying design and architectures, and the original requirements of the application. It involves recovering, recreating, and rewriting the code for applications in a usable form. A common task as a Software Engineer is helping an organization work with a legacy application, for which fundamental information is missing. It's a task that you are likely to perform at some point in your career.

For computer scientists (CS) and information technology (IT) professionals, many definitions of reverse engineering apply, but for our purposes, we define this key methodology as follows:

Software reverse engineering is the act of discovering how an application works by unearthing its architecture and internal structure, as manifested in code.

Software reverse engineering is an opportunity to explore how codes, from the most basic to the more sophisticated, align to create the functionality of any computer application.

Why Software Reverse Engineering?

Software reverse engineering allows one to analyze a product, discover qualities that may be hidden about it, document it, reproduce it, or purposely refine it. Software reverse engineering may also be necessary to recover the code of an application when documentation is missing or incomplete.

Figure 0.1 shows a list of common uses for software reverse engineering among CS and IT professionals.

Researching network communication protocols	**Modernizing legacy code or retrieving lost source code**	**Improving or creating documentation**
Bridging data between different operating systems or databases	Fixing bugs or finding inefficiencies or vulnerabilities	Uncovering and using undocumented features
Enhancing compatibility across applications or platforms	Identifying algorithms for malware such as viruses or worms	Checking for resistance to reverse engineering

Figure 0.1: Uses of Reverse Engineering

No matter the reason behind software reverse engineering, the process requires one quality that CS and IT professionals exhibit and honor: curiosity. When you follow your curiosity to discover how and why an application works, including its basic structures and design principles, you will be better able to use, deploy, and enhance it. In fact, curiosity about code is essential, not only to software reverse engineering but to most of the activities and processes performed by CS and IT professionals.

Software Architecture

Part of the recovery process is accessing a source code file. In software reverse engineering, the source code could be in a variety of formats. You may need to translate machine language to a more workable form in order to begin following the steps to reverse engineering the software.

First, consider how computers and other devices operate through machine language. Machine language is a basic operational language that is numerical and binary in nature. It consists of complex sequences of 1’s and 0’s. While computers require machine language to operate, it is nearly impossible for humans to read or write. Above machine language is assembly language.

Assembly language uses names rather than numbers to present an essential bridge between machine language and the higher-level programming language. For reverse engineering, focusing on assembly is particularly relevant as it's the "layer" of code between the function of the application and the higher-level language. In assembly, an architecture may be hidden but can be unearthed to help explain how an application works.

Higher-level programming languages include C++, C, Java, and Python. These languages are used to produce applications. The following image shows the layers of code.

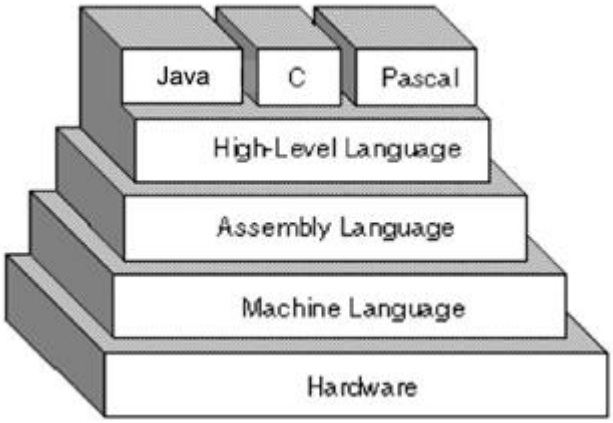

Figure 1.1: Layers of Code
Source: Vangie Beal, "Assembly Language." Retrieved from https://www.webopedia.com/TERM/A/assembly_language.html

Tales of Reverse Engineering

The Dungeon Master's Code

Doug Bell, developer of the Dungeon Master series of computer games, tells the story of Christophe Fontanel, who followed his curiosity to an extreme. Fontanel is a leading player and fan of Dungeon Master who maintains an unofficial encyclopedia devoted to the game. His fascination is also expressed through reverse engineering.

Bell reports that Fontanel has reverse-engineered nearly every version of the game on most of the platforms on which it was released. Through this process, Fontanel has unearthed the complete C and assembly code behind every manifestation of Dungeon Master. He has done this not to pirate the game or for any financial gain, but simply to increase his knowledge of how the game works and to satisfy what must be a compelling curiosity.

Source: Doug Bell, “Is it possible to reverse engineer software without its source code?” Retrieved from https://www.quora.com/Is-it-possible-to-reverse-engineer-software-without-its-source-code

Section 2: Main Components of Assembly Language

Assembly language is the language that earlier programmers worked directly in. When reviewing assembly, we are seeing a part of the history of the CS/IT industry and profession. It is also a component of how information technology works. Assembly language translates machine language to a more workable form.

We begin by examining and practicing reverse engineering by learning assembly language. This section introduces you to the main components of assembly language. It also presents step-by-step instructions on how assembly code "behaves." The goal is to recognize assembly code and it's output so that it can be translated in large blocks.

While each CPU has its own assembly language, all assembly languages have certain characteristics in common. Let's examine the main components of assembly language: the registers. We will discuss the main operations in assembly and the structure of the code, and then walk through code, demonstrating how variables are defined and used in assembly and how operations such as cin and cout are used.

You should be able to understand a complete assembly program that reads input from users, perform operations such as addition or subtraction (or any other operation), and display the results.

Assembly Basics: Registers

Assembly consists of 16 general-purpose registers (GPRs). The table below shows the registers that assembly uses to perform operations.

64-Bit Register	Lowest 32-Bits	Lowest 16-Bits	Lowest 8-Bits
rax	eax	ax	a1
rbx	ebx	bx	b1
rcx	ecx	cx	c1
rdx	edx	dx	d1
rsi	esi	si	si1
rdi	edi	di	di1

rbp	ebp	bp	bp1
rsp	esp	sp	sp1
r8	r8d	r8w	r8b
r9	r9d	r9w	r9b
r10	r10d	r10w	r10b
r11	r11d	r11w	r11b
r12	r12d	r12w	r12b
r13	r13d	r13w	r13b
r14	r14d	r14w	r14b
r15	r15d	r15w	r15b

Figure 1.2: Assembly Language Registers
Source: Ed Jorgensen, "x86-64 Assembly Language Programming With Ubuntu."
Retrieved from http://www.egr.unlv.edu/~ed/assembly64.pdf

Registers are sorted into three groups:

1. General registers
2. Control registers
3. Segment registers

General Registers

General registers are divided into three categories:

- Data registers
- Pointer registers
- Index registers

Data registers are %eax, EBX, ECX, EDX for 32-bit machines and AX, BX, CX, and DX for 16-bit machines. These are shown in Figure 1.3.

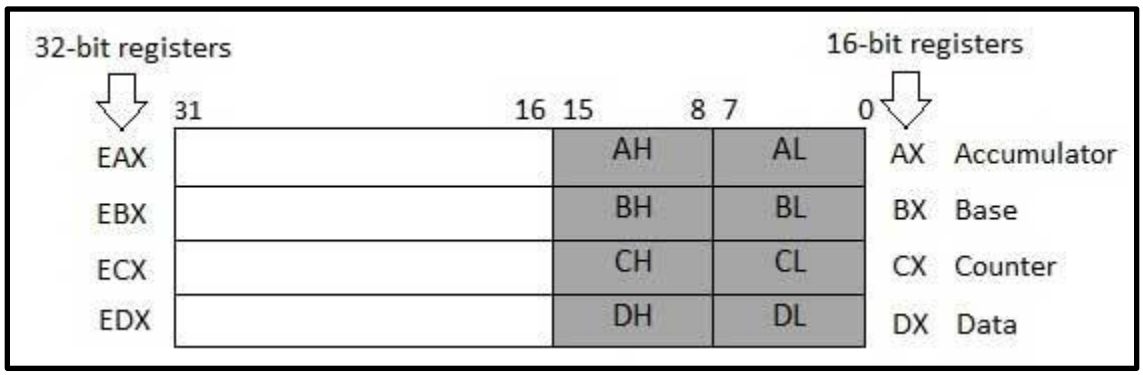

Figure 1.3: Data Registers
Source: https://www.tutorialspoint.com/assembly_programming/assembly_registers.htm

Pointer registers include IP, SP, and BP. These are shown in the figure below.

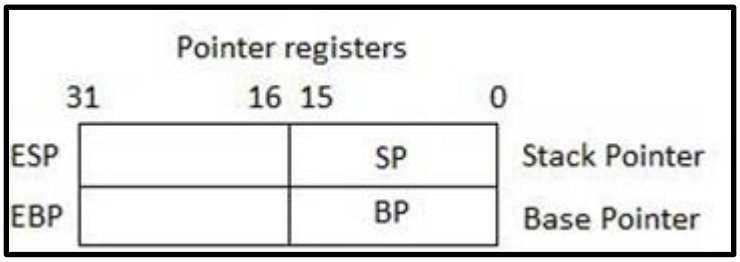

Figure 1.4: Pointer Registers
Source: https://www.tutorialspoint.com/assembly_programming/assembly_registers.htm

Index registers include SI and DI. See Figure 1.5.

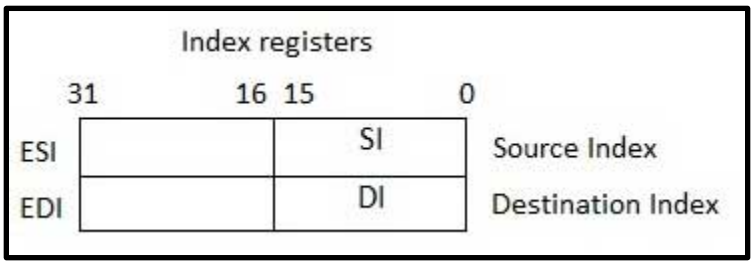

Figure 1.5: Index Registers
Source: https://www.tutorialspoint.com/assembly_programming/assembly_registers.htm

Control Registers

Control registers include the following:

- Overflow flag (OF)
- Direction flag (DF)
- Interrupt flag (IF)
- Trap flag (TF)
- Sign flag (SF)
- Zero flag (ZF)
- Auxiliary carry flag (AF)
- Parity flag (PF)
- Carry flag (CF)

Segment Registers

Segment registers include code, stack, and data registers.

Assembly Basics: Instructions

Assembly has many instructions. In this section, we will focus on these specific instructions: MOV, LEA, ADD, and SUB.

MOV

MOV instruction moves a value from a register or to a register. The main syntax for MOV instruction is as follows:

mov <reg>,<reg>

mov <reg>,<mem>

mov <mem>,<reg>

mov <reg>,<const>

mov <mem>,<const>

There are many uses for MOV, such as these:

```
mov eax, [ebx]          ; Move the 4 bytes in memory at the address contained in EBX
                        into EAX

mov [var], ebx          ; Move the contents of EBX into the 4 bytes at memory
                        address var. (Note that var is a 32-bit constant.)

mov eax, [esi-          ; Move 4 bytes at memory address ESI + (-4) into EAX
4]

mov [esi+eax],          ; Move the contents of CL into the byte at address ESI+EAX
cl

mov edx,                ; Move the 4 bytes of data at address ESI+4*EBX into EDX
[esi+4*ebx]
```

Source: Adam Ferrari, Alan Batson, Mike Lack, Anita Jones, and David Evans, "x86 Assembly Guide." Retrieved from http://www.cs.virginia.edu/~evans/cs216/guides/x86.html

LEA

LEA places the address of the second operand into the register (first operand).

Syntax

lea <reg32>,<mem>

Examples

lea edi, [ebx+4*esi] — the quantity EBX+4*ESI is placed in EDI.

lea eax, [var] — the value in var is placed in EAX.

lea eax, [val] — the value val is placed in EAX.

Source: Adam Ferrari, Alan Batson, Mike Lack, Anita Jones, and David Evans, "x86 Assembly Guide." Retrieved from http://www.cs.virginia.edu/~evans/cs216/guides/x86.html

ADD

The ADD command takes two operands. It adds both operands and places the value in the first operand. ADD syntax is as follows:

```
add <reg>,<reg>
add <reg>,<mem>
add <mem>,<reg>
add <reg>,<con>
add <mem>,<con>
```

SUB

The SUB command takes two operands. It subtracts the second operand from the first operand and stores the value in the first operand. SUB syntax is as follows:

```
sub <reg>,<reg>
sub <reg>,<mem>
sub <mem>,<reg>
sub <reg>,<con>
sub <mem>,<con>
```

Assembly Basics: Syntax

In reverse engineering, it's important to be familiar with the main functions and registers used to perform operations. In this section, we will walk through examples that show how assembly "behaves." This will strengthen your knowledge of assembly logic and show how operations are conducted in assembly.

Assembly Components

Assembly code consists of three main parts:

1. **Data** - The data section referred to as .data, used to declare constant variables and initialize data.
2. **Bss** - The bss section referred to as .bss, used to declare variables.
3. **Text** - The text section referred to as .txt, used to hold the actual code.

In assembly, comments start with the ; character. For example, the following line contains a comment:

```
add %eax, ebx ; this adds one register to the other.
```

Assembly Statements

There are three types of assembly statements:

1. **Executable Instructions**: Each instruction generates one executable machine instruction.
2. **Assembler directives**: These are non-executable instructions used to tell the assembler of various assembly processes.
3. **Macros**: Similar to functions and procedures

An assembly instruction consists of the following:

- Label (optional)
- Mnemonic (mandatory): This is the instruction to be executed (ADD, AND, etc.).
- Operands (optional): These are the operators for the Mnemonic (if any).
- Comment: This is any text that follows the semicolon character (;).

Section 3: Understanding Assembly From a C++ Perspective

In this section, we will examine assembly from the perspective of working in the higher-level general use programming language, C++.

Specifically, we will focus on the basic parts of assembly language used by the GNU compiler collection or g++, which is included in your test environment.

First, let's look at how we can define a variable named "value" and assign it a value of 3. In C++, the code looks like this:

```
#include <iostream>


int main()
{
  int i;
}
```

The assembly code corresponding to the C++ code above is as follows:

```
.text
.globl    main
.type     main, @function
main:
.LFB0:
.cfi_startproc
pushq   %rbp
.cfi_def_cfa_offset 16
.cfi_offset 6, -16
movq    %rsp, %rbp
```

```
.cfi_def_cfa_register 6
movl    $0, %eax
popq    %rbp
.cfi_def_cfa 7, 8
ret
.cfi_endproc
```

Next, we will remove the code that is not necessary for our explanation and focus on purely the assembly code.

So the code above now looks like this:

```
movl    $0, %eax
```

Notice that we removed the stack push and stack pop that happens before we run the code. Think of it this way: Before you run your code, the computer pushes the values of the registers in the stack and uses the registers to run your logic/code; then, when your code is done, you pop the values from the stack back to the registers, returning to the original state.

Now let's define our first variable and assign it a value. In C++, the code would be:

```
int i;


i = 6;
```

In assembly, the code would be as follows:

```
movl    $6, -4(%rbp)
```

For more details on how this is happening, review this video tutorial: x868 Crash Course.

Notice that the value of 6 was moved 4 bytes above our register %rbp.

Now let's declare two variables with values in them and add them together from a third variable.

In C++, the code is as follows:

```
int i, j, k;

i = 2; // notice that this is 4 bytes so this is stored 4 bytes above our starting point
```

```
j = 3; // notice that this is also 4 bytes, now we have 8 bytes allocated for variables

k = i + j; // notice that k also needs to take 4 bytes (total of 12 bytes). Remember this when you see assembly code).
```

In assembly, the code is as follows:

```
movl    $2, -12(%rbp)

movl    $3, -8(%rbp)

movl    -8(%rbp), %eax

movl    -12(%rbp), %edx

addl    %edx, %eax

movl    %eax, -4(%rbp)
```

The first instruction pushes the 2 value 12 bytes above %rbp.

Next, we push the 3 value 8 bytes above the base stack %rbp.

The addl function uses the values in %edx and %eax to perform its operations, so we POP the values from the stack into eax and edx, respectively, then perform the add operation, and then push the result into the stack.

So, to summarize, this is what we have in memory that register rbp points to:

```
%rpb

Value   Location

2       -12

3       -8

5       -4
```

Next, let's look at the same code but with a print statement at the end of it. In C++, the code is as follows:

```
int i, j, k;

i = 2;
```

```
j = 3;

k = i + j;

 cout << k;
```

In assembly, the code is as follows:

```
movl    $2, -12(%rbp)

movl    $3, -8(%rbp)

movl    -8(%rbp), %eax

movl    -12(%rbp), %edx

addl    %edx, %eax

movl    %eax, -4(%rbp)

movl    -4(%rbp), %eax

movl    %eax, %esi

movl    $0, %eax

call     cout
```

Note the code in bold. We move the value stored in rbp (-4) to eax. This is an extended register. Then, to prepare to execute the cout command, which uses register esi, we move the value from eax to esi and issue a call to cout, which prints the value the register esi holds (which is rbp(-4), which is the value 5).

Note: The function cout uses the %esi register to print.

Any number you want to print must be moved to %esi.

Now let's look at code that reads a value from the user then prints it. In C++, the code is as follows:

```
int i;

 cin >> i;

 cout << i;
```

In assembly, the code is as follows:

```
leaq    -4(%rbp), %rax
```

```
movq    %rax, %rsi

call     cin

movl    -4(%rbp), %eax

movl    %eax, %esi

call     cout
```

The assembly instruction LEA is short for Load Effective Address. The LEA instruction acts as a pointer. It places the address of the variable (rbp (-4)) into register rax. Next, the address is placed in register rsi, which is used by the cin function. Then, cin is executed. Note that cin reads a value from the user and stores it in the address in register rsi (which is rbp(-4)).

After cin is executed, the value entered by the user is moved from (rbp(-4)) to eax. Then it is moved from eax to esi, which is used by cout to print on the screen.

Let's merge all operations together. We will try to do this: Add declare 3 variables, read values in two variables, subtract them and store the result in the third variable, then display the value of the third variable. The C++ code looks like this:

```
int i,j,k;
 cin >> i >> j;
k= i - j;
 cout << k;
```

In assembly, the code looks like this; note that we are going to explain each block of code separately this time:

Action Code

Define variables. Make rdx point to one variable and rax point to another variable

```
leaq    -8(%rbp), %rdx
leaq    -12(%rbp), %rax
```

Prepare to read values in the variables. Since cin uses rsi, move address of rax to rsi to the values are read

```
movq    %rax, %rsi
```

Read from user `call     cin`

Now move the values read into edx and eax into rbp (locations -12 and -8)

movl -12(%rbp), %edx

movl -8(%rbp), %eax

Now subtract the values in edx and eax and place result in rbp (-4) subl

%eax, %edx

movl %edx, %eax

movl %eax, -4(%rbp)

Now move the value from rbp(-4), which is the results of the subtraction) into the rsi register so we can print it

movl -4(%rbp), %eax

movl %eax, %esi

Call print function call cout

Now that you have seen how assembly works, including how it relates to C++, you are ready to begin writing assembly.

Section 4: Writing Assembly Code

To get started, we need to look at assembly code as a fixed frame, or skeleton, that allows us to add our own code. In other words, there is code you **should not change or touch,** and there are areas where you can **add whatever code you want**.

Skeleton Code: Defining Variables

The g++ assembler, in general, pushes the values of the registers in the stack and then executes the code and pops the values from the stacks back to the registers.

Let's look at the C++ code for a basic program:

```
#include <iostream>



int main()
{
  int i;
  i = 12;
  return 0;
}
```

The skeleton code that maps to the above simple code is as follows:

```
        .file   "test.cpp"
        .text
        .globl  main
        .type   main, @function
main:
.LFB2:
        .cfi_startproc
        pushq %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq   %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    $12, -4(%rbp)
        movl    $1, %eax
        popq   %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE2:
        .size   main, .-main
        .ident  "g++ : (Ubuntu 4.8.4-2ubuntu1~14.04.3) 4.8.4"
        .section        .note.GNU-stack,"",@progbits
```

Notice that the code in **bold** is created by g++ assembler, which you should not change. The line of code where we assigned the value 12 to a variable is in red and italics: You may add any assembly code you want in that array *only*.

Let's assume we want to create another variable and assign it a value of 13. What would the code be?

The code to move 13 into a variable would be like this:

```
movl    $13, -4(%rbp)
```

Remember that -4(%bp) is already used by our original variable that holds the 12, so our code would need to move 4 bytes, so our code would be:

```
        movl    $13, -8(%rbp)
```

Now let's look at the actual code created by g++ :

```
        .file   "test.cpp"
        .text
        .globl  main
        .type   main, @function
main:
.LFB2:
        .cfi_startproc
        pushq %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq  %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    $13, -8(%rbp)
        movl    $12, -4(%rbp)
        movl    $1, %eax
        popq  %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE2:
        .size   main, .-main
        .ident  "g++ : (Ubuntu 4.8.4-2ubuntu1~14.04.3) 4.8.4"
        .section        .note.GNU-stack,"",@progbits
```

Notice that the value was inserted in the "code" area; everything else was "fixed" and unchanged. This is what is expected from you in this course. Your focus is on generating the assembly code that goes into the "read" area. You should not concern yourself with the g++ "hardcoded" section.

Okay, now you know the "skeleton" for identifying two variables and assigning them values. What about constants? What if you want to print a string; what does the skeleton code for that look like?

Skeleton Code: Dealing With Strings

Strings, as you may recall from previous learning, are series of characters that may function as a constant or variable. Let's see how to include them in skeleton code.

We'll start by looking at a simple program in C++:

```
#include <iostream>


using namespace std;


int main()

{


  cout << "Hello World" << endl;

  return 0;

}
```

This is the assembly code that the above code creates:

```
        .file   "test.cpp"
        .section        .rodata
.LC0:
        .string "Hello World"
        .text
        .globl  main
        .type   main, @function
main:
.LFB2:
        .cfi_startproc
        pushq %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq  %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    $.LC0, %edi
        call    puts
        movl    $1, %eax
        popq  %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE2:
```

```
.size	main, .-main
.ident	"g++ : (Ubuntu 4.8.4-2ubuntu1~14.04.3) 4.8.4"
.section	.note.GNU-stack,"",@progbits
```

Notice that a string was defined at label LC0. That label was later used to move its content to %edi, which is used by the function puts. The function puts is called to print the content of the register %edi, which contains the string defined in $.LC0: “Hello World.” Simple, right?

Let’s print two messages. What would the assembly code look like? See below.

```
        .file   "test.cpp"
        .section        .rodata
.LC0:
        .string "Hello World"
.LC1:
        .string "I am here..."
        .text
        .globl  main
        .type   main, @function
main:
.LFB2:
        .cfi_startproc
        pushq %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq  %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    $.LC0, %edi
        call    puts
        movl    $.LC1, %edi
        call    puts
        movl    $1, %eax
        popq  %rbp
```

```
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE2:
        .size   main, .-main
        .ident  "g++ : (Ubuntu 4.8.4-2ubuntu1~14.04.3) 4.8.4"
        .section        .note.GNU-stack,"",@progbits
```

Notice that because we are printing two strings, two labels were declared: LC0 holds the first string, “Hello World,” and LC1 holds the second string, “I am here.” Looking at the actual assembly code next, notice that the value of LC0 is moved to %edi, which is used by puts, then the value of LC1 to %edi, which is then sent to puts again. Simple again, right?

Skeleton Code: Dealing With Arrays

An array is a variable that can hold multiple values of the same type. The array variable functions as a pointer that points to the address of the first value.

This variable is represented below. The capital “I” should be lowercase for “int”.

To assign values to an array, you would use the index, or value location, or address location. For example, to place a value in the first location, you would write this:

```
numbers[0]=10;
```

This places the value of 10 in the first location, or to the address to which the variable numbers point. The diagram above would now look like this:

The same goes for the rest of the locations. The second location is sub-index 1, and the third location is sub-index 2. So to assign values to our array, we would write this in C++:

```
values[0] = 10;

values[1] = 20;

values[2] = 30;
```

Now let's look at the assembly version of our code.

When we convert the above code to assembly, we get the following:

```
movl    $10, -16(%rbp)
movl    $20, -12(%rbp)
movl    $30, -8(%rbp)
```

Notice that we moved the first value, 10, to location -16 in the register %rbp, the second value was moved to -12, the third value moved to -8.

Address in %rbp	-16	-12	-8
Value	10	20	30

Section 5: Advanced Assembly: Conditionals and Loops

Conditionals and loops add functionality. Let's see how they work in assembly code.

In this section, we will build on this learning by discovering advanced assembly instructions. We will focus on conditional statements and loops. This section also explains the basics of conditional statements and logical operators, and lastly discusses loops.

Relational Operators

Relational operators are conditional statements used to compare two variables. Variables can be of any

type, so you can compare numbers, strings, addresses, and so on. Relational operators can be seen in the table below.

Operator	Description	In C	In Assembly	How It's Read in Assembly
=	Equal	==	je	Jump if Equal
>	Greater than	>	jg	Jump if Greater
<	Less than	<	jl	Jump if Less than
>=	Greater or Equal	>=	jge	Jump if Greater or Equal
<=	Less or Equal	<=	jle	Jump of Less than or Equal
<>	Not Equal	!=	jne	Jump if Not Equal

The concept in assembly is simple: First, we compare the two values, usually located in the registers, and then we implement one of the operators (je, jg, etc.) based on the result.

Logical Operators

Logical operators are used to write Boolean expressions. Boolean expressions are expressions that yield a result of TRUE (1) or FALSE (0). Our focus will be on two logical operators: AND and OR. The most important thing to know about logical operators are the truth tables.

Truth Table for AND operator

Operation	Operation in C	Operation in Assembly	Result
True AND True	**True || True**	**and <dest>, <src>**	**TRUE**
True AND False	**True || False**		**FALSE**
False AND True	**False || True**		**FALSE**
False AND False	**False || False**		**FALSE**

Truth Table for OR operator

Operation	Operation in C	Operation in Assembly	Result
True **OR** True	True || True	or <dest>, <src>	TRUE
True **OR** False	True || False		TRUE
False **OR** True	False || True		TRUE
False **OR** False	False || False		FALSE

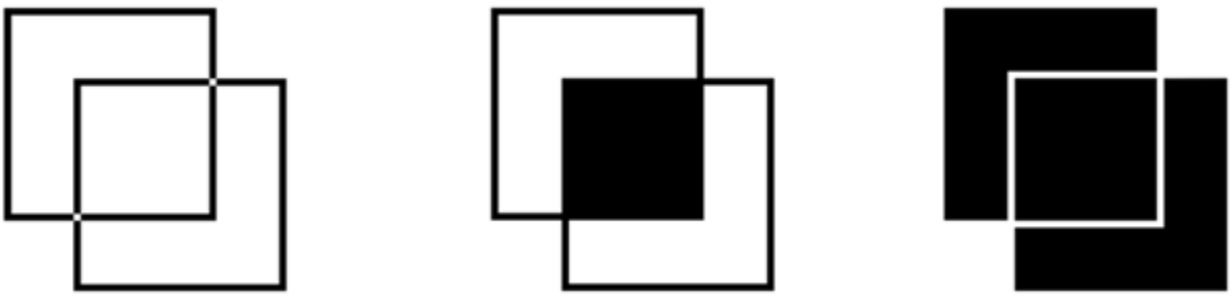

Figure 3.1: Boolean Logic
Source: Noun Project, https://thenounproject.com/

Practicing Logical Operators

Now let's turn our attention to arrays. An array is a variable that can hold multiple values of the same type. The array variable functions as a pointer that points to the address of the first value.

Let's practice the use of logical operators in both C++ and assembly.

Let's look at the following code in C++ language:

```
int a,b;
        cin>> a;

        cin>> b;

if (a > b)
        cout << "First number larger than second number<<endl;
else
        cout << "Second number is larger than first number" << endl;
return0;
```

The code above is very simple; we read two values in two variables, and then we simply compare them. If the first value is greater than the second value, we display this message:

"First number larger than second number"

Otherwise, we display:

"Second number larger than first number"

Now let's look at the assembly part. The translation for the code above is as follows:

```
        .file   "test.cpp"
        .section        .rodata
.LC0:
        .string "%d"
        .align 8
.LC1:
        .string "First Number larger than second number"
        .align 8
.LC2:
        .string "Second number larger than first number"
        .text
        .globl  main
        .type   main, @function
main:
.LFB2:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        subq    $16, %rsp
        leaq    -8(%rbp), %rax
        movq    %rax, %rsi
        movl    $.LC0, %edi
        movl    $0, %eax
        call    __isoc99_cin
        leaq    -4(%rbp), %rax
        movq    %rax, %rsi
        movl    $.LC0, %edi
        movl    $0, %eax
        call    __isoc99_cin
        movl    -8(%rbp), %edx
        movl    -4(%rbp), %eax
```

```
        cmpl    %eax, %edx
        jle     .L2
        movl    $.LC1, %edi
        call    puts
        jmp     .L3
.L2:
        movl    $.LC2, %edi
        call    puts
.L3:
        movl    $1, %eax
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE2:
        .size   main, .-main
        .ident  "g++ : (Ubuntu 4.8.4-2ubuntu1~14.04.3) 4.8.4"
        .section        .note.GNU-stack,"",@progbits
```

Let’s divide the code into blocks and discuss each block.

Block of Code	Translation
.LC1: .string **"First Number larger than second number"** .align 8 .LC2: .string **"Second number larger than first number"**	Declare the two strings we will use in the code.

leaq -8(%rbp), %rax
movq %rax, %rsi
movl $.LC0, %edi
movl $0, %eax
call __isoc99_cin
	Read a value into the first variable using cin. Place value in %rax which points at -8(%rbp).

leaq -4(%rbp), %rax
movq %rax, %rsi
movl $.LC0, %edi
movl $0, %eax
call __isoc99_cin
	Read a value into the first variable using cin. Place value in %rax which points at -4(%rbp).

movl -8(%rbp), %edx movl -4(%rbp), %eax	Move first value from -8(%rbx) to %edx. Move second value from -4(rbx) to %eax.
cmpl %eax, %edx	Compare between %eax and %edx.
jle .L2	Jump to L2 if the result of the comparison is less than or equal. **(Notice that, in C code, we checked for greater than, but here it looked for less than or equal.)**
movl $.LC1, %edi call puts jmp .L3	If the above comp returned false, execute this code, which displays what's in LC1 (declared at the beginning). Once the string is printed, we jump to L3, another label which terminates the code,
.L2: movl $.LC2, %edi call puts	We jump here if the comparison result is false. Notice that it places the string LC2 into %edi and then calls puts (print the string to the screen).
.L3: movl $1, %eax leave	Code done. Terminate program.

Jump to L2 if the result of the comparison is less than or equal. (Notice that, in C++ code, we checked for greater than, but here it looked for less than or equal.)

```
movl    $.LC1, %edi
```

```
call    puts

jmp     .L3
```

If the above comp returned false, execute this code, which displays what's in LC1 (declared at the beginning). Once the string is printed, we jump to L3, another label which terminates the code,

```
.L2:

        movl    $.LC2, %edi

        call    puts
```

We jump here if the comparison result is false. Notice that it places the string LC2 into %edi and then calls puts (print the string to the screen).

```
.L3:

        movl    $1, %eax

        leave   Code done. Terminate program.
```

In the above walkthrough, we learned how to compare between two variables.

Loops: Explanation and Use

Loops are a way to iterate n-number of times to perform certain operations. All high-level programming languages use loops to execute statements a desired number of times.

Each time a loop executes, we call it an iteration. Loops use counters to determine how many iterations happen.

Let's look at an example:

```
for (i = 0; i < 10; i++)

    {

        cout <<

    }
```

The above code iterates 10 times. It prints the value of (i) at every iteration, so the output of the above code is 0123456789.

If we want to explain the above code to a nonprogrammer, we would say the following (notice that this is very similar to how assembly works, which we cover next):

1. Initialize the variable i at 0.
2. Check if i is less than the upper boundary, which is 10.
3. If yes, go inside the loop code.
4. Print the value of i.
5. Increment i by 1.
6. Check the upper boundary: If i is less than 10, jump back to step 3.

Loops in Assembly

Let's examine loops in assembly. We will start with the C++ code:

```
#include <iostream>


using namespace std;


int main()
{
 int i;
 for(i=0;i<10;i++)
   cout << i;
 return 0;
}
```

The assembly translation for the above code is as follows:

```
        .file   "test.cpp"
        .section .rodata
.LC0:
        .string "%d"
        .text
        .globl  main
        .type   main, @function
main:
```

```
.LFB2:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        subq    $16, %rsp
        movl    $0, -4(%rbp)
        jmp     .L2
.L3:
        movl    -4(%rbp), %eax
        movl    %eax, %esi
        movl    $.LC0, %edi
        movl    $0, %eax
        call    cout
        addl    $1, -4(%rbp)
.L2:
        cmpl    $9, -4(%rbp)
        jle     .L3
        movl    $1, %eax
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
```

```
.LFE2:
        .size   main, .-main
        .ident  "g++ : (Ubuntu 4.8.4-2ubuntu1~14.04.3) 4.8.4"
        .section.note.GNU-stack,"",@progbits
```

Remember the steps we discussed on how a loop executes. We will apply the same to our assembly code.

```
movl    $0, -4(%rbp)      Initialize a variable to 0.
```

This is done by placing 0 in -4(%rbx).

```
jmp     .L2
```

Check if we go “inside” the loop or exit it. We do this by jumping to L2, which is the block of code responsible for checking the boundary.

We will explain L2 next, but be careful, as L2 comes AFTER L3 in code.

```
.L2:
        cmpl    $9, -4(%rbp)
        jle     .L3
```

In L2, we check and compare the value in the variable -4(%rbp) to 9. If the comparison result is less than or equal to 9, that means we still need to execute the code inside the loop, so jump to L3.

```
.L3:
        movl    -4(%rbp), %eax
        movl    %eax, %esi
        movl    $.LC0, %edi
        movl    $0, %eax
        call     cout
```

addl $1, -4(%rbp) Move the value of -4(%rbp) to %eax.

Print the value of %eax to the screen.

Add 1 to the value in -4(%rbp).

Refer to Figure 3.2 to see the logical workflow of this code.

Figure 3.2: Loops in Assembly

Section 6: Working With Binary Files

Reverse engineering often involves conversions to or from binaries.

Binary files are any files that are not text files. Binary files are typically accessed and interpreted by programs, processors, or tools preset to read binaries.

In this chapter, you will learn how to understand a binary file, disassemble it, and transform it into assembly code. Then, relying on the skills from earlier chapters, you will see how to convert the assembly code to C++.

To perform these tasks, we will use several Unix commands and tools. Some of the tools and commands you'll be introduced to are as follows:

1. File: A command that gives you a detailed description of the file object (the binary file)

2. Bless: An application that allows us to view the binary file in hex format. The application gives us an easy-to-use interface to search and navigate through the binary file (hex code).

3. Gdb: This is used to disassemble the code into assembly

    a. Load object into gdb.
    b. Identify the functions in the object.

c. Disassemble each function.

So let's get started with working with binary files!

Analyzing an Object File

Our first step in approaching binaries is to create a very simple C program that we can use as an example.

The code is below:

```
#include <iostream>

 using namespace std;

int main()

{

  int i;

  i = 171;

  return 0;

}
```

First, let's compile this code. Assume that the file containing this code is called test.cpp. To compile it, we execute the following command:

G++ -o test.exe test.cpp

This creates an executable file, our binary object that we will be working with. The executable file is called ***test***. To run the program, we simply type this:

./test.exe

Nothing happens, since the program simply declares a variable and places the number 171 in it.

There are many ways to reverse-engineer a file from binary. The following are some suggested steps:

1. Get basic information on the object. This is important because it's important to know what type of executable we have. Was it built using a 32-bit or 64 bit machine? What is the type of processor that this object (executable) was compiled in?
2. Disassemble the code.

Now that we have basic knowledge about this program, we move to our next step: disassembling the code.

Disassembling the Object File

First, let's view our code in hex format. We'll run Bless and send to it our executable; we do this by typing the following:

bless test

Our code is uploaded at 0x00000000 and notice the starting (magic) string at the beginning:

Notice that the beginning of our program is 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00. You can use Bless to navigate through the binary object, which will be needed in the future.

For now, familiarize yourself with the tool. Some of the helpful tutorials you can view to learn more about it are listed below.

Bless Hex Tutorials

Ubuntu 12.04 Forensics: Hex Editor Overview

Bless Hex Editor: Seeing Beyond the Bytes

Now that we have looked at the binary object and uploaded it to a tool that enables us to view it (in HEX format), let's disassemble it. To disassemble the code, we are going to use the gdb command. First, we need to load the object into dgb by typing the following:

gdb test

This will load the object into gdb, enabling us to debug it. The first thing we need to do is to figure out which functions are defined in this object. To do so, we type the following:

Info functions

The following output should be displayed on the screen:

```
All defined functions:

Non-debugging symbols:
0x00000000004003a8 _init
0x00000000004003e0 __libc_start_main@plt
0x00000000004003f0 __gmon_start__@plt
0x0000000000400400 _start
0x0000000000400430 deregister_tm_clones
0x0000000000400460 register_tm_clones
0x00000000004004a0 __do_global_dtors_aux
0x00000000004004c0 frame_dummy
0x00000000004004ed main
0x0000000000400500 __libc_csu_init
0x0000000000400570 __libc_csu_fini
0x0000000000400574 _fini
```

Notice that our start—400400, which we analyzed earlier—is the _start point. Looking below, we notice that there is only one function in our program, the **main()** function. So, let's disassemble the main() function. To do so, we type the following:

disassemble main

The following output will be displayed:

```
0x00000000004004ed <+0>:   push  %rbp
0x00000000004004ee <+1>:   mov  %rsp,%rbp
0x00000000004004f1 <+4>:   movl  $0xab,-0x4(%rbp)
0x00000000004004f8 <+11>:  mov  $0x1,%eax
0x00000000004004fd <+16>:  pop  %rbp
0x00000000004004fe <+17>:  retq
```

Notice that this code pushes the value ab into the stack at -4(%rbp). The value ab is in hex, the decimal value for ab is 171. This can be seen in our original C++ code displayed at the beginning of this chapter.

Tales of Reverse Engineering

Tracking Pokémon

How to track and find rare Pokémon? For the devoted players of Pokémon Go, this was no easy task, especially since the tracker released with the game proved unreliable. Pokévision filled the void by releasing an upgraded tracker produced through reverse engineering and geolocational technology that allowed players to view the Pokémon nearby. Players could also shift locations to track the rare Pokémon in their native settings. While Pokévision is no longer available, other trackers have followed in its wake, perhaps by reverse-engineering the Pokévision tracker.

Source: Paul Tamburro, "Pokémon Go creators take down Pokévision in controversial development."

Retrieved from http://www.craveonline.com/entertainment/1015341-pokemon-go-creators-take-pokevision-controversial-development#8YolJUSXvsGZi7FE.99

Section 7: How to Generate Assembly from C++ Code

Now let's walk through how to convert C++ to assembly.

It's easy to convert C++ code to assembly using g++ . Let's assume you created the Hello.c. To create the assembly code, simply type the following command:

g++ -S Hello.cpp

Note that the S is an uppercase S, not lowercase. Also notice that there is a dash (-) before the letter S. **Also, there MUST be a space between 'g++' and '-S'.** It must read "g++ -S". This tells the compiler to generate an assembly file as an output file. Once you execute that line, you should find a file titled **Hello.s**. If you open the file Hello.s, you should see this code:

```
        .file   "Hello.c"
        .section        .rodata
.LC0:
        .string  "Welcome to my world"
        .text
        .globl   main
        .type    main, @function
main:
.LFB2:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq   %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    $.LC0, %edi
        call    puts
        movl    $1, %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
```

```
.LFE2:
        .size   main, .-main
        .ident  "g++ : (Ubuntu 4.8.4-2ubuntu1~14.04.3) 4.8.4"
        .section        .note.GNU-stack,"",@progbits
```

Congratulations! You have now created your first assembly code!

For more information, review GeeksforGeeks

Compiling with g++.

Section 8: Writing C++ from Assembly

When writing C++ code from assembly code, you may find that there is not a one-to-one correlation between the lines of code. The following table is an example of writing assembly code to C++ code.

Assembly Code	C++ Code	Explanation

movl -8(%rbp), %eax
 	
sall $2, %eax
 	
subl $1, %eax
 	
leal 7(%rax), %edx
 	
testl %eax, %eax
 	
cmovs %edx, %eax
 	
sarl $3, %eax
 	
movl %eax, -4(%rbp)
	int input;
	
	int output = (((input * 4) - 1)/8);
		1. Move contents of -8(%rbp) to %eax
		
		2. Take value of %eax and multiply it by 4 (shift 2 bits to the left)
		
		3. Subtract 1 from contents of %eax
		
		4.Load effective address - put memory address of 7(%rax) into %edx
		
		5.Test %eax to see if it's above zero (AND)
		
		6.Conditional move if negative for %edx to %eda
		
		7. Take value of %eax and divide it by 8 (shift 3 bits to the right)
		
		8. Move contents of %eax into -4(%rbp)

Section 9: Code Security

Enhancing code security is central to software development and reverse engineering.

Identifying security issues and enhancing code security are not only key parts of the software development process but also one of the most common reasons for performing reverse engineering. In this chapter, we will explore the security concerns a programmer needs to take into consideration when writing code. We will also cover some of the code holes that hackers can take advantage of to crash the code, understand its behavior, and eventually hack it.

A program behaves in a manner that is established by the code as it's written. Developers may see only how the program should behave without taking into consideration that a user might enter invalid input that causes it to crash. A crash is unacceptable because it might reveal information about the code itself that gives hackers a chance to hack it.

A characteristic example is SQL injection. SQL injection is one of the common techniques in web hacking ("SQL Injection," 2017). When the user is asked to enter data using a form, they type malicious SQL instructions in the form controls and then submit the page. The code on the server side takes that code, assuming it is regular user input, and passes it to the database. The database management system (DBMS) executes the "bad" code.

Let's look more closely at code security issues.

Common Security Issues

The dynamic nature of coding presents security risks. Every time programmers use a new module, language, or concept, they introduce new opportunities for hackers to break into or alter the code.

In this section, we will review some of the common security concerns that a developer must be aware of, as discussed by Schindler (2006).

Preventing these concerns is an important part of the work of the developer, while identifying and addressing these flaws may be performed through reverse engineering.
Some of these concerns are buffer overflow, not initializing variables, ignoring errors, and dangling pointers.

Let's take a look at each of these concerns.

Buffer Overflow

Buffer overflow occurs when you access an invalid memory location. For example:

```
int numbers[5];
……
cout << numbers[5];
```

Not Initializing Variables

It's very important to initialize a variable before using it. Developers should never assume that by declaring a variable that a 0, blank, or NULL is placed in it. The following is an example of using an uninitialized variable:

```
Char *p;
free(p);
```

Ignoring Errors

Ignoring errors presents security issues. For example: Try

```
{
            cin >> num1 >> num2;
          num1 = num1/num2; // notice that if num2 is a 0, we get divide by error exception which is caught later on in the catch() statement but is then ignored!
}
Catch ()
{
// errors ignored
}
```

Dangling Pointers

```
class CAR
{
…
}
void main()
```

```
{
CAR *c = new CAR();
}
```

Notice that memory was allocated for the class CAR, and that object C was assigned values, but then the code ended without deallocating the memory.

As in the example above and in Figure 5.1, dangling pointers do not point to a valid destination. Commonly, this occurs when an object is deleted or de-allocated but the corresponding pointer is not changed. This constitutes a security risk that may not impact the program immediately and may require careful examination to detect and address.

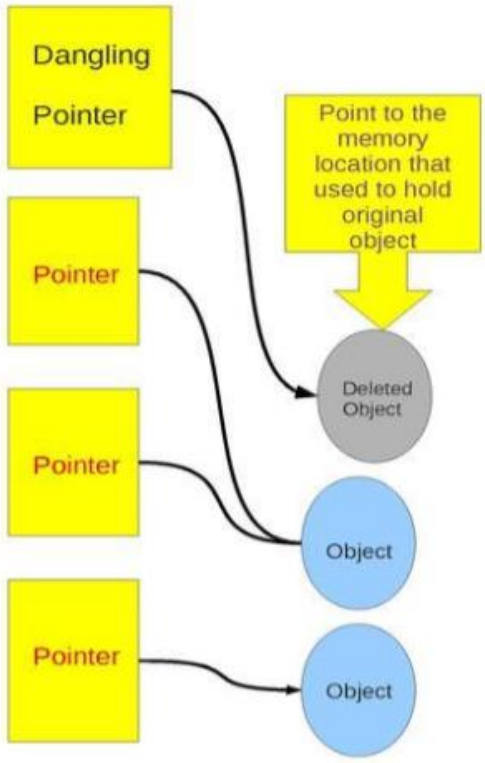

Figure 5.1: Dangling Pointer
Source: Mayur642, Wikimedia Commons. Retrieved from https://en.wikipedia.org/wiki/File:Dangling_Pointer.pdf

Addressing Security Issues

In this section, we'll discuss some recommendations that programmers should take into consideration when writing code.

Encapsulation

When you write code, you should hide as much information from the user as possible.

Check User Input

Do not assume that the user will enter the right type of data. For example, if you ask the user to input a birth date, make sure you check the input to make sure it's valid.

Limit User Input

Limit user input to a fixed format. If you ask the user to input a phone number; for example, limit the input to a fixed format such as (XXX) XXX-XXXX.

Practice Defensive Programming

You should always assume the user is trying to break the code. One way to prevent this is through extensive use of ASSERTS throughout the code to ensure that the data flows from one module to another are valid, and to catch any erroneous data flows.

Test Code Implementation Limits

When dealing with pointers, for example, use assertions when a program "misbehaves" and errors out.

Top Ten List: Secure Coding

Seacord (2011) lists 10 excellent ways to enhance code security:

1. Validate input.
2. Heed compiler warnings.
3. Architect and design security policies.
4. Keep it simple.
5. Default deny.
6. Adhere to the principle of least principle.
7. Sanitize data sent to other systems.
8. Practice defense in depth.
9. Use effective quality assurance techniques.
10. Adopt a secure coding standard.

Avoiding Pointer Trouble

Maioli (2017) lists pointer practices to avoid that can cause serious security issues. The bad practices to avoid include the following:

1. Making pointless optimizations
2. Sweeping things under the rug
3. Ignoring proven best practices
4. Working on your own all the time
5. Not sharing what you've learned with your team
6. Being too slow giving feedback to managers/customers

Tales of Reverse Engineering

Clean Room

As you probably realize, copying proprietary code, especially for commercial reuse, is often illegal and always unethical. However, reverse engineering provides methods to gain similar functionality in an existing product while creating unique code. A classic method for this type of work is known as the Clean Room.

For example, in the 1980s, a company wanted to capture the functionality of the Basic Input/Output System, or BIOS, for PCs made by IBM. BIOS is used to initialize hard-drive files during startup, so a BIOS provides key functionality.

To capture this functionality, the company used a Clean Room approach with two steps:

1. First, a team of engineers studied the IBM BIOS and carefully notated all its functions.
2. Then, a second team of engineers coded from the first team's description to produce similar but distinct BIOS.

The Clean Room effect was produced through the separation of the teams, and the fact that the second team never examined the IBM BIOS directly. They started from scratch to build a program that matched the desired requirement, as in the process of forward engineering. The Clean Room BIOS was used in producing the first IBM-compatible PCs.

Source: Matthew Schwartz, "Reverse engineering." Retrieved from https://www.computerworld.com/article/2585652/app-development/reverse-engineering.html

Section 10: Corporate Security

Security is also an organizational priority.

Security is not just an issue for individual programmers. Organizational and corporate security policies must be put in place to make it more difficult for hackers to successfully attack. A complete system is needed across the organization to ensure that code as written is secure, and that security concerns are systematically addressed.

Code that's properly written, tested, implemented, and deployed is the basis for code security. The "Application Security Architecture Cheat Sheet" from the Open Web Application Security Project (OWASP) contains an excellent template for an application security architecture plan that includes four components:

1. Business requirements
2. Infrastructure requirements
3. Application requirements
4. Security program requirements

Each category is divided into smaller categories with specific questions to address, as shown in Figure 6.1. Implementing this architecture is an excellent way to take a systemic, organization-wide approach to code security.

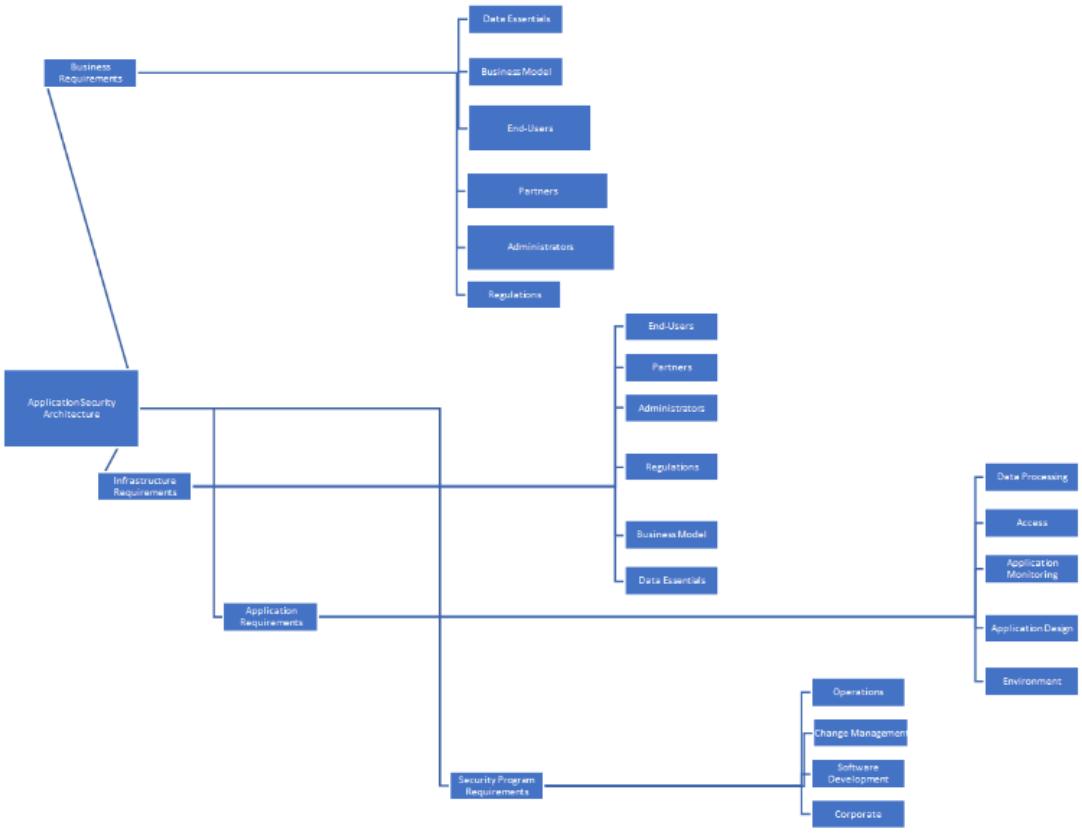

Figure 6.1: OWASP Security Architecture
Source: https://www.owasp.org/index.php/Application_Security_Architecture_Cheat_Sheet
Secure Code: The Big Picture

Developers need to consider security not only at the code level but at the design level as well. In this section, we will examine methods for enhancing security through best practices in design.
The separation of concerns (SoC) is an important starting point for security at the design level.
SoC was first addressed by Edsger (1982). In this approach, developers view each requirement or block of information as a "concern" that they need to address. Separating code based on information handling it makes it easier to debug code, trace issues, and even protect against attacks. A module is created to address each concern. A module, in this case, is a separate block of code that may be a procedure, function, DLL, namespace, or even a class.

Maioli (2017) listed 35 errors programmers make. He divided the errors into four categories: code organization, teamwork, writing code, and testing and maintenance. Code organization and code writing address the issues related to developers that we have discussed earlier in this manual.

In the next section, we'll look more closely at testing and maintenance.

Security by Design

Security by design means designing a product from the ground up to be secure. It begins with the assumption that all code is insecure, and it seeks to achieve security by means such as automated security controls, continuous testing, and streamlined auditing.

Code Testing and Maintenance

Companies need to ensure that a clear policy is in place and consistently used in testing code.
A common error that developers make is writing test scenarios from their own point of view. Instead, test scenarios should also include the point of view of system analysts who know the software business requirements, as well as users.

Additionally, companies must also run automated testing to ensure that code testing is thorough and runs through all possible cases. Testing should also include more than just ensuring that the code runs; testing needs to check for performance as well.

Further, companies must set policies for publishing code. Allowing developers to publish code directly into production is dangerous, and not only because the code may crash, resulting in a loss of business or inconvenience to users at the least. Publication of code by developers also opens the door for hackers to learn more about code details. Some crashes may reveal specific instructions, such as the user ID and password for database connection strings, for example.
Security concerns should also be addressed in nonfunctional activities, such as pushing large changes to production or leaving after pushing a new change.

Pushing

When a new version of the program is ready, it is pushed to production. By pushing, we mean making it available to be used by users. For websites, for example, pushing code means to publish the new version on the server so it can be accessed by users.

Further Reading on Code Security

For more on separation of concerns, see the Application Security Architecture Cheat Sheet: Security Program Requirements from OWASP.

For examples of companies that work on ensuring your code is secured, see Secure Coding from the Computer Emergency Response Team Coordination Center (CERT).

To learn more about how to develop secure code, see Introduction to Secure Coding from the Apple Developers Library.