War Industries Presents:

An Introduction to Programming for Hackers
Part I

By Lovepump, 2004

Visit:
www.warindustries.com

Part I – The Beginning

Intro:

What is programming? Why should I learn to code? What is a hacker?

Many of the questions and discussions on the War Industries Forums ask such things as: "Where can I get X program to 'hack' my friend's computer", or "How can I send a trojan to someone". These questions indicate that the person is only interested in using pre-made programs and possess little understanding of how computer systems really work. They are often referred to as "script kiddies", "skiddies" or "skidiots". They believe that being a hacker only entails downloading a programming and running it.

In my opinion, a hacker is someone who is curious about how something works. Just knowing how to operate a system (a script kiddie) is insufficient to a true hacker. The hacker wants to know the ins and outs, nuts and bolts of the entire thing. If you've ever torn apart a lawnmower, VCR, radio, or any device for that matter, you have a hacker mind. Using duct tape to fix a car is a "hack"!

If you really want to be a hacker, you must want to understand the working guts of a computer. What makes it tick? Why does it crash? What is a buffer overflow? How can I tell the computer what to do?

This series will attempt to do just that.

Conventions:

- The discussion and examples in this series will deal strictly with the x86 architecture. Analogies to other architectures may be made, but there are significant differences between platforms.
- All code examples will be in C and ASM. The intent is to teach C, but some ASM is required for low level explanations of processor functions.
- Code examples will be for Linux platforms, but should work as well under "command line" MS Windows.
- Comments can be made to me (Lovepump) at the WI Forums.
- Watch for bold italics like this: ***address***. It highlights an important term that you need to fully understand. Don't skip over it.

What is a program?

A computer program is a series of statements that provide step by step instructions to a computer. A computer is not a human and cannot divine or interpolate commands nor anticipate steps. It requires *explicit* and *concise* instructions to perform even the most basic task.

To show and example, we use the famous "Hello World" example in C:

```
#include <stdio.h>

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

     printf("Hello World!\n");

}
```

That's it.

We won't get in to the nuts and bolts yet. Just wanted to give you a taste of what a C program looks like.

To understand more, we need to do more of a "deep dive" into the guts of what makes a computer tick. You want to be a hacker don't you?

Memory and Base16

Computer memory is like a series of boxes. Each box can contain a small piece of information. In our case, each box is 1 byte, or 8 bits. That means that each box can contain a number between 0 and 255. That's it. It can't contain anything else.

Each box has a number, or ***address***. The address is used to send and receive information to each piece of memory. The addresses start at 0 and move up sequentially to your maximum installed memory. The numbering system used in referring to addresses is ***hexadecimal, or hex***. Hex is base 16.

Huh? Base 16? Yes. The hex number system goes like this:

0 1 2 3 4 5 6 7 8 9 A B C D E F 10 11 12 13 14 15 16 17 18 19 1A …

So basically instead of "rolling digits" at 10, like the decimal system, we roll digits at 16. So hex 10 = decimal 16. Instead of saying "hex" we will use the

notation: **0x**. So 0x0A would translate to decimal 10. 0xFF would translate to 255. Other notations for hex you might see are: **\x0A** (you'll see this notation in "shellcode") or **0Ah** ('h' for hex).

OK, so it's up to us to make use of a bunch of little boxes that can't hold much and make the computer perform miracles? Let's carry on.

You may be thinking: "If a computer can only store little boxes with numbers between 0 – 255, how can I calculate 100 * 100?" It is true that the computer can only store discreet numbers from 0 – 255 (*hex – FF*), but it can combine two more locations to form larger numbers. This is not done by adding two numbers together, but by combining two to form a larger number. Here's an example:

Here are two memory locations:

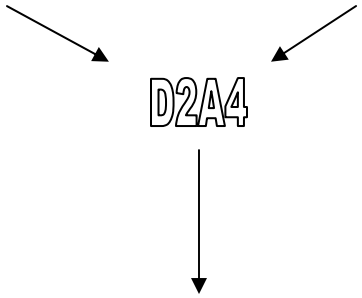
Address	1000	1001
Value (Hex)	3F	2C

We can see that the memory location with address 1000 stores a value of 0x3F (decimal 63) and 1001 holds 0x2C (44 decimal). If we ask the computer to stick the two together to make one number, we get 0x3F2C, or in decimal 16172. By sticking two bytes we can make numbers between 0 – 65535 (0xFFFF). If we stick four bytes together, we can represent numbers from 0 – 4294967295 (0xFFFFFFFF).

This last example is significant. Four bytes put together makes a ***word*** in the x86 architecture. Other processors have different word sizes. Our processor uses a 32 bit or 4 byte word size (hence the 32 bit architecture).

The values stored in memory can serve a number of purposes. You can directly store numeric information. You can also store an address in memory that refers to another memory location! This type of use is called a ***pointer***. A pointer doesn't hold data, but it points to the place that holds the data. **This is an extremely important concept in programming**. The idea of pointing to memory is a base programming concept. Here's an example:

Address	3AF4	3AF5
Value	D2	A4



Address	D2A4
Value	1C

In this example the values at 3AF4 and 3AF5 combine to produce an address D2A4, which holds a value 1C. So we can directly address D2A4, **or** we can tell the computer "the value we want is held in the box ***pointed*** to by 3AF4 & 5".

Let this concept sink in. It's important. In the real world of x86, the pointer is one word in size, but the implementation is the same. When it comes to programming in the later sections, the pointer size is unimportant (and invisible), but the concept is extremely important.

The Processor (CPU)

The processor is the heart of the system. It is designed to do a couple of things:

- Fetch an instruction from memory.
- Execute the instruction.

Computer programs are stored in memory. RAM memory is very slow compared to how fast modern processors can work. The processor needs its own "memory" to save current working information locally. These memory spaces are called ***registers***. The registers are physically located in the processor itself, so it doesn't have to fetch anything from RAM. The processor also uses a space of memory called the stack. More on the stack later.

There are a number of registers on an x86. For now let's concern ourselves with six of the registers:

EAX, EBX, ECX, ESP, EBP, EIP.

Each register starts with an 'E'. This refers to a 32 bit, or 1 word length registers. (Prior to the 386, the registers were typically 16 bit).

The first three registers, EAX, EBX, and ECX are called General Purpose registers. They are used for memory storage and retrieval, addition, subtraction and value passing.

The next three registers, ESP, EBP and EIP are Indexes and Pointers. They help control program flow and memory referencing. These are very important registers to understand for a hacker and are at the heart of a buffer overflow (BOF). More on the buffer overflow later.

The ***stack*** is a FILO (First In Last Out) area of memory used by the processor. Imagine a stack of papers on a desk. You can put papers on the stack and take papers off of the stack. The first paper you put on the stack is the last on off (it's on the bottom). In computer speak, you "push" values on to the stack, and "pop" values off of the stack. The processor has a register that points at the "top" of the stack. This register is ESP. As you may have guessed, SP stands for stack pointer. EBP (Base Pointer) is also used for stack pointing, but it used "locally" by functions.

Physically in memory, the stack starts at a certain point and *builds down*. If the stack started by pointing at address 1000, and 4 bytes are pushed on the stack, the stack pointer would now point at memory address 996 (1000 – 4).

The processor fetches and ***executes*** instructions. Instructions in native machine code are numerical and stored in memory. How does it know which command to execute next? The EIP register (Instruction Pointer) ***points*** to the memory location of the next instruction to execute. This register is read only, so don't jump yet. But you can indirectly change EIP to point to other addresses. This is a key point in buffer overflow exploits.

The native language of the x86 is machine code. The human readable version is assembly. We will not be going into assembly in any great detail here, but here is an example:

```
        .file   "hello.c"
            .section        .rodata
    .LC0:
            .string "Hello World.\n"
            .text
```

```
.globl main
        .type   main,@function
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        andl    $-16, %esp
        movl    $0, %eax
        subl    %eax, %esp
        subl    $12, %esp
        pushl   $.LC0
        call    printf
        addl    $16, %esp
        leave
        ret
.Lfe1:
        .size   main,.Lfe1-main
        .ident  "GCC: (GNU) 3.2.2 20030222 (Red
Hat Linux 3.2.2-5)"
```

This is the assembly version of our "Hello World" program. It was produced by the GNU compiler (more about compilers later). We won't be focusing on assembly, but for interest's sake, we will discuss some highlights of the code above.

The line "main:" is the beginning of our program. Before that there are some other things going on. What's happening? If you look at the section ".LC0" the ASM code is telling the computer to store the string "Hello World\n" in memory and refer to that memory ".LC0". That sounds familiar. LC0 refers to, or ***points***, at the memory where the string starts! It is a symbol that points to the beginning of our text in memory. So now instead of sending the whole string every time you need to access it, you can just send a pointer to the start of the string in memory. This approach is much more efficient.

In "Main" you can see some assembly commands and references to some of the registers we mentioned above. For example:

```
movl            %esp, %ebp
```

This command moves (movl, think of the 'l' to mean 'long' for word. movb moves a 'b' or byte of data) the contents of the register ESP to the register EBP. It moves the entire word (don't forget a word is 4 bytes here) from one to the other.

The command before that one has some interest for us as well.

```
pushl           %ebp
```

This command "pushes" a word (4 bytes) from EBP onto the stack. The pushl command automatically adjusts ESP.

One last interesting one is:

```
subl            $8, %esp
```

subl means subtract. This command subtracts 8 from ESP. Why? More later, but this is the assembly way of creating space for a "variable". It is memory space on the stack that the processor can use later. In this case, the processor created space for 2 words, or 8 bytes.

Please review this section for important words. It is not important that you understand the program above at this point. Just understand the concepts of the ***key words***.

Programming Languages:

Programming languages come in many flavours and shapes., however they all fall in three general categories:

Machine Code – The native language of the processor. Not for human consumption. If you see old pictures of people feeding punch cards in to some of the first computers in the 60's, you'll understand. Programs were made by coding numbers on punch cards and feeding them into a reader one at a time. You have to be a serious grognard to even consider this.

Assembly – The human readable machine code. Assembly takes the numeric commands of machine code and replaces them with text commands (like 'movl' above). It is as close as you get to machine code. You can directly access all registers and memory. There are things (not many though) that can really only be done in assembly. Hand tweaked assembly code can be very fast, however the advantage is not that great over a modern, optimizing compiler. Assembly is difficult to use and can be very difficult to "debug".

High Level Languages – This group is a large family of languages that includes C, Perl, Pascal and Basic. The high level language is most commonly used for programming projects. The advantage of high level languages is that they use the same "language" as the problems we as coders are trying to solve. Review the two previous programs. One in C, one in assembly. You can see that the C

program is “printing” something (printf(“Hello World\n”);) even if you don’t know any C at all. Conversely, the assembly program is not clear at all.

High level languages are generally broken in to two groups: interpreted and compiled. Interpreted languages like Perl, are translated by the computer one instruction at a time. The interpreter makes machine code out of the program one instruction at a time. Troubleshooting is typically easier with interpreted languages. The downside is that the target computer for your program must have the interpreter installed. For example, you cannot run Perl program unless you have Perl installed on your machine.

Compiled languages are not under that restriction. Compiled languages, like C, are passed through a program called a compiler all at once. An executable is generated. This executable can be sent to any computer, without the restriction of having an interpreter installed. The downside to compiled languages is that they are typically stricter in what they will allow.

The language we have chosen to learn here is C. It is a compiled language. To continue on this journey, you will need a compiler. Free compilers are available for most platforms. The examples in this tutorial will be using the GNU C compiler, GCC.

Questions:

- What is a register? Which register tells the processor the address of the next instruction to execute?
- What is 24 in hex? What is 0x2E in decimal?
- What is a pointer?
- How does a computer represent numbers greater than 255?
- What is a ‘word’ and how long is a word in the x86 architecture?

Next – Starting in C.

War Industries Presents:

An Introduction to Programming for Hackers
Part II

By Lovepump, 2004

Visit:
www.warindustries.com

Part II – Programs 101

Goals:

At the end of this part, you should be able to code, compile and execute your first C program. Our programming focus will remain on C, but some ASM will be covered for the sake of being thorough.

Review:

Please ensure you understand the following terms:

- Address
- Hex
- Word
- Pointer
- Registers
- Stack
- Execution

If you are unsure of any of these terms, go back and review Part I now.

Programs:

> **program:**
>
> n. 1. A magic spell cast over a computer allowing it to turn one's input into error messages.
>
> 2. An exercise in experimental epistemology.
>
> 3. A form of art, ostensibly intended for the instruction of computers, which is nevertheless almost inevitably a failure if other programmers can't understand it.
>
> From Jargon File

I prefer to think of a program as a series of *instructions* in a given language and syntax designed to perform a *function*. In our case, the language is C. The function to perform is up to the coder.

C Syntax

Like any language (computer, spoken word, etc) C has a form, or syntax. The syntax includes words, format and punctuation just like a spoken language. Let's review our "Hello World" example from the previous lesson:

```
/* Hello World Program */

#include <stdio.h>

main(){

     printf("Hello World!\n");

}
```

In this example there a number of syntax items we can analyze.

The first line starts with **/***. This character sequence indicates the start of a comment. Everything past this is considered a comment until the 'end comment' symbols are found. The ***/** at the end of the line indicated the end of the comment.

Comments are skipped by the compiler and have absolutely no impact on the final program. They are extremely important for coders however. Well commented code is far easier for others to read. If you want to have some fun and look at some real comments, get the Linux source code, go to the src directory and type:

```
grep –R fuck *.c
```

It will show how many lines of Linux source include the word "fuck" (literally hundreds). Here are some highlights from the 2.6 kernel:

```
arch/sparc/kernel/sunos_ioctl.c:    /* Binary compatibility is good
American knowhow fuckin' up. */

arch/mips/kernel/irixioctl.c: * irixioctl.c: A fucking mess...

arch/sparc64/kernel/traps.c:            /* Why the fuck did they have
to change this? */

drivers/char/watchdog/shwdt.c:   * brain-damage, it's managed to fuck
things up one step further..

drivers/ide/pci/cmd640.c: *  These chips are basically fucked by
design, and getting this driver
```

As you can see, even the kernel programmers get frustrated!

The next line of code is:

#include <stdio.h>

The # character indicates a complier directive. This line will not generate any executable code. It is a special instruction for the compiler. This particular instruction tells the compiler that we need an external library, stdio.h, to be ***included*** in our program. stdio.h contains the 'printf' function that we use in our program. If we didn't 'include' this, the compiler wouldn't know what 'printf' meant. We will see a lot of includes in future code.

The first real program code line is the next one (this has changed from the Part I example for ease of explanation, but more on that later):

main() {

There are two items of interest on this line. The first is the function name ***main***. Main is the entry point of our program. It is the function where the executable will begin when the code is run.

The next item of interest is the curvy bracket, or brace **{** Braces are used in C to enclose sections of code. You will see them everywhere in C programs you read. This particular brace in our program says "the function main starts here". If you look to the last line of code you will see the closed brace. You've probably figured out that it means "main ends here". Braces can be nested inside each other, but more on that when we get to it.

The next line of code is:

printf("Hello World\n");

This is the only thing our program really does. It prints Hello World to the screen. You may wonder what the \n is for. It is a format character meaning 'newline'. When we compile the program later, leave it out as an experiment to see the result.

Notice that this line ends in a semicolon ; The semicolon is required after every statement in C. It declares the 'end of command'. You will see them everywhere in C code.

Notice also that this line is indented. Why? It considered excellent style in C to indent your code. Each time you 'open' a brace, indent one tab further. You will see better examples in later exercises.

Finally, our closing brace meaning 'end of main'. Since main is done, so is our program. Upon reaching the end of main, our program exits.

That's all. Let's try to compile and run this program. First, in your favourite editor, enter the program exactly as it appears above. You may leave out the comments if you wish, but it is good practice to try them out! Save your work as 'test.c' .c is the traditional file extension used for C source code.

To compile the program we use the GNU C compiler, gcc.

```
gcc test.c -o test
```

This invokes the compiler, giving it our code (test.c) and telling it to output the executable (-o) to a file called test. If we omit the –o option, gcc by will save your program to a file called **a.out** by default. We may now execute our code by invoking the command 'test'.

```
./test
```

Here is my result:

```
-sh-2.05b$ gcc test.c -o test
-sh-2.05b$ ./test
Hello World!
-sh-2.05b$
```

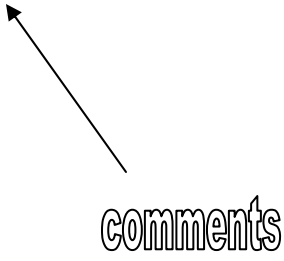
Hello to you too…

To see some of the forms we learned, have a look at some of the source code for a classic ~~trojan~~ Remote Administration Suite: Back Orifice 2k:

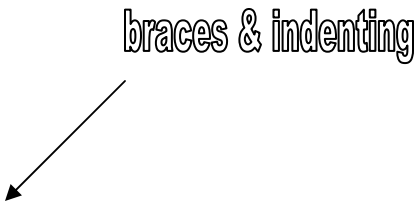
Disclaimer:

The following is C code. The example provided is to demonstrate form only. Don't let your head explode if you don't understand what it does.

```
/*  Back Orifice 2000 - Remote Administration Suite

    Copyright (C) 1999, Cult Of The Dead Cow

The author of this program may be contacted at dildog@l0pht.com. */

// ************************************************
//         BO2K                          cDc
//                 Back Orifice 2000
//         Written By DilDog and Sir Dystic
//    Copyright (C) 1999,  Cult of the Dead Cow
//  Special thanks to L0pht Heavy Industries, Inc.
// ************************************************

#include<windows.h>
#include<main.h>
#include<bo_debug.h>
#include<functions.h>
#include<osversion.h>
#include<bocomreg.h>
#include<commandloop.h>
#include<dll_load.h>
#include<config.h>
#include<pviewer.h>
#include<process_hop.h>

#ifdef NDEBUG
//#define HOOK_PROCESS5684818,5683735,5684300
//#define HIDE_COPY
#endif

HMODULE g_module=NULL;
HANDLE g_hfm=NULL;
DWORD g_dwThreadID=0;

BOOL g_bRestart=FALSE;
char g_svRestartProcess[64];
BOOL g_bEradicate=FALSE;

// --------------- Stealth options ----------------
char g_svSubOptions[]="<**CFG**>BO2k Sub Options\0"

// Back Orifice Thread Entry Point
DWORD WINAPI EntryPoint(LPVOID lpParameter)
{
startofentrypoint:;
        g_bRestart=FALSE;

        g_module=(HMODULE)lpParameter;

        // Load up other DLLs just to make sure we have them (we're acting as a loader
here).

        LoadLibrary("kernel32.dll");
        LoadLibrary("user32.dll");
        LoadLibrary("gdi32.dll");
        LoadLibrary("winspool.dll");
        LoadLibrary("advapi32.dll");
        LoadLibrary("shell32.dll");
        LoadLibrary("ole32.dll");
        LoadLibrary("oleaut32.dll");
        LoadLibrary("wsock32.dll");

        // Create useless window class
        WNDCLASS wndclass;
        wndclass.style = 0;
        wndclass.lpfnWndProc = DefWindowProc;
        wndclass.cbClsExtra = 0;
        wndclass.cbWndExtra = 0;
        wndclass.hInstance = g_module;
```



```
wndclass.hIcon = NULL;
wndclass.hCursor = NULL;
wndclass.hbrBackground = NULL;
wndclass.lpszMenuName = NULL;
wndclass.lpszClassName = "WSCLAS";

RegisterClass(&wndclass);

// Determine OS version
GetOSVersion();

// Use Dynamic Libraries
InitDynamicLibraries();

// Enable permissions on Windows NT
if(g_bIsWinNT) {
        HANDLE tok;
        if(pOpenProcessToken(GetCurrentProcess(),TOKEN_ADJUST_PRIVILEGES,&tok)) {
                LUID luid;
                TOKEN_PRIVILEGES tp;
                pLookupPrivilegeValue(NULL,SE_SHUTDOWN_NAME,&luid);
                tp.PrivilegeCount=1;
                tp.Privileges[0].Attributes=SE_PRIVILEGE_ENABLED;
                tp.Privileges[0].Luid=luid;
                pAdjustTokenPrivileges(tok,FALSE,&tp,NULL,NULL,NULL);

                pLookupPrivilegeValue(NULL,SE_SECURITY_NAME,&luid);
                tp.PrivilegeCount=1;
                tp.Privileges[0].Attributes=SE_PRIVILEGE_ENABLED;
                tp.Privileges[0].Luid=luid;
                pAdjustTokenPrivileges(tok,FALSE,&tp,NULL,NULL,NULL);
                CloseHandle(tok);
        }
}
```



I have edited out some of the code, for ease of read. (If anyone takes issue with this edit, please drop me a line and I will correct it).

There is a lot of stuff there. What we want to see is that this program has a lot of the things we have discussed so far. Look for includes, comments, braces and indenting. The rest will be covered in later exercises.

Proper structure and form is extremely important. I cannot stress that enough. Without these things code can be confusing at best, or unreadable at worst. You may be thinking that "I'm an uber leet haxor who doesn't need all that form and stuff". For an answer to that, let's look at some assembly code.

Disclaimer:

Caution, you are about to look at an assembly program, do not try to understand it yet. Just look at the "form":

```
; ########################################################################

        .486
        .model flat, stdcall
        option casemap :none   ; case sensitive

; ########################################################################

       .nolist
        include kernel32.inc
        include windows.inc
        include user32.inc
        include wsock32.inc
        include ole32.inc
        include shlwapi.inc
        include oaidl.inc
        include wininet.inc
        include advapi32.inc
        include urlmon.inc
        include shell32.inc
        include gdi32.inc

        .list
        includelib kernel32.lib
        includelib user32.lib
        includelib wsock32.lib
        includelib ole32.lib
        includelib shlwapi.lib
        includelib wininet.lib
        includelib advapi32.lib
        includelib urlmon.lib
        includelib shell32.lib
        includelib gdi32.lib

; ########################################################################

        szText MACRO Name, Text:VARARG
          LOCAL lbl
          jmp lbl
            Name db Text,0
          lbl:
        ENDM

        m2m MACRO M1, M2
          push M2
          pop  M1
        ENDM

        mNextListEntry MACRO ML
          cld
          xor     eax, eax
          or      ecx, -1
          repnz scasb
          cmp     byte ptr[edi], 0
          jnz     ML
        ENDM

.data
        EncryptStart2   dw      "$$", "$$"

.code
        EncryptStart    dw      "$$", "$$"

        include Config.inc
        include Src\SrcFile.inc
        include Utils.asm
        include Stream.asm
        include PassGen.asm
        include HashTable.asm
        IFNDEF  DisablePK
                include ProcKiller.asm
```

```
        ENDIF
        include CPLStub.inc
        include CPL.asm
        include VBS.asm
        include HTA.asm
        include ZIP.asm
        include StartUp.asm
        include Network.asm
        IFNDEF  DisableNotify
                include Notify.asm
        ENDIF
        include Admin.asm
        include DNS.asm
        include SMTPClient.asm
        include SMTPThread.asm
        IFNDEF  DisableInfect
                include PVG.asm
                include PEInfector.asm
        ENDIF
        include EmailScanner.asm
        include HDDScanner.asm
        include SMTPMessage.asm

        .data
                ; Do not change order
                szSeDebug               db      "SeDebugPrivilege",0
                szAdvApi                db      "advapi32.dll",0
                                        db      "AdjustTokenPrivileges", 0
                                        db      "InitializeAcl",0
                                        db      "LookupPrivilegeValueA",0
                                        db      "OpenProcessToken",0
                                        db      "SetSecurityInfo",0,0

                szKernel32              db      "kernel32.dll",0
                                        db      "RegisterServiceProcess",0,0    ;
RegisterServiceProcess(GetCurrentProcessID,1);. ..
```

Hey! Includes, comments (look for lines starting with ;) and indenting. The format and structure are just the same as in C. And yes, to those of you who noticed, this is a section of assembly code from the Bagel, (aka Beagle) worm. Even the virus and trojan coders use proper style. You should too.

Let's move on. The first solid piece of programming we need to cover is variables.

Variables:

A ***variable*** is a data storage unit used in your program. Without variables, we would have great difficulty working with pieces of information. Variable types in C are:

```
char, int, float, double
```

To ***declare*** a variable, we use the following syntax:

```
int i;
```

This statement declares 'i' as an integer variable (notice the semicolon). We can now use i to hold integer information.

The int type can hold an integer between -2^{31} and $2^{31} - 1$. The int type is one word (remember, 4 bytes) long.

The char type is one byte long, and can hold on character. If you recall from our previous lesson, one byte can store a number between 0 – 255. That is the extent of character storage.

The first type of operation we need to learn about variables is the assignment:

```
int i;
i = 7;
```

In this example, we create an integer name i, then assign it the value 7. The = operator is the assignment operator in C.

We can perform various arithmetic operations on variables. Arithmetic operations include addition, subtraction and multiplication. For example:

```
int i, j, k;   /* Multiple declaration. */
i = 6;
j = 5;
k = i + j;     /* k now equals 11 (5 + 6) */
j = i * k;     /* j now equals 66 (6 * 11) */
i = j + 1;     /* i equals 67              */
k += 6;        /* Whoa! huh?               */
i++;           /* WTF?                     */
```

The last two examples seem a bit confusing, but are in fact quite real. k += 6 is short form for k = k + 6. This increases the value of k by 6, so in this example it would equal 17. i++ also appears strange. ++ is the increment operator. It increments your variable by one of whatever it is. In the case presented, the statement i++ increments i by one integer, so i would equal 68. We can also use the -- operator to decrement the variable by 1.

We can also use comparison operators on variables. For the example, we'll use something call 'if':

```
int i, j, k;   //Multiple declaration.
i = 6;
j = 5;
```

```
k = 7;
if(i == j);     /* false - note double = */
if(k > j);      /* true                  */
```

Operators to know:

Operator	Meaning
==	Equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
!=	Not equal to

An important distinction is made between = and ==. One is assignment, the other is comparison. Many a coder (me included) have made the mistake:

```
if(a = b) …
```

That statement evaluates to TRUE for any value of b other than zero. Why? When C evaluates expressions, it gives the value of non-zero to TRUE and 0 to FALSE. If b = 17 in the above example, then 'a' would be set to the value of 'b' (17) and the statement would evaluate to true, regardless if 'a' actually did equal 'b'. It would also change the value of 'a' when you don't expect it to. *Yikes*. That little bug can cause no end of head scratching and yelling, so be careful.

It should be:

```
if(a == b) …
```

This version uses the comparison ==, not the assignment =.

Jump Ahead.

OK, you say, it's great we can use some variables, but how am I supposed to display them and take input into them? We'll jump ahead a little and cover two functions that help us out. printf() and scanf(). printf performs formatted printing (PRINT Formatted - printf). You use it as follows:

```
printf("This is text.  This is a variable: %d \n", i);
```

printf displays everything in the quotes verbatim except for special control characters. In the above example, there are two such control sequences: %d and \n.

The %d tells printf to substitute the %d with a decimal number given by a variable. In this case, the value of 'i' is substituted. The next control sequence is \n. This sequence means new line. If the variable 'i' were equal to 1432, our example would print:

```
This is text.  This is a variable: 1432
```

There are other variable types, like %c for char and %f for float. We will learn the rest of these at a later time. The variables are interpreted in order, so if 'c' where the letter **A** and 'x' was **98736**,

```
printf("c: %c, x: %d \n", c, x);
```

Would produce the output:

```
c: A, x: 98736
```

To take input we use the command scanf, for scan formatted. The syntax is very similar to printf, however we need to put a special character in front of our variable for now. It's not important why (yet), we just need to do it. To take an integer input and assign it to 'y', we use:

```
scanf("%d", &y);
```

It's important to note the **&** symbol in front of y. You will learn about this in a later chapter on pointers. scanf expects a ***pointer*** to your variable. The **&** gives scanf the ***address*** of your variable in memory.

Our First (Second?) Program

Grab your trusty editor. Here is the program as promised:

//Our first C Program

```
#include <stdio.h>
main() {
     char c;
     printf("Please type a letter:\n");
     scanf("%c", &c);
     printf("You entered: %c \n", c);
}
```

Enter the program and save it as 'first.c'. What do you think it will do?

To compile our program we enter:

```
gcc first.c -o first
```

Run it.

Here's my output:

```
-sh-2.05b$ gcc first.c -o first
-sh-2.05b$ ./first
Please type a letter:
F
You entered: F
-sh-2.05b$
```

Here's another code example. See if you can figure out what it does before you compile and run it.

```
#include <stdio.h>
main() {
     int a, b, c;
     printf("Please enter an integer: \n");
     scanf("%d", &a);
     printf("Please enter a second integer: \n");
     scanf("%d", &b);
     c = a + b;
     printf("Answer: %d \n", c);
}
```

What does this program do?

The first thing it does after main() is to declare three integer variables:

```
int a, b, c;
```

Next, we prompt the user to enter a number:

```
printf("Please enter an integer: \n");
```

Then, scanf to store the number in 'a'. The code then repeats the same, but stores the next value in 'b'. We then add the two together to produce the value c:

```
c = a + b;
```

Now we display our result:

```
printf("Answer: %d \n", c);
```

So we've created a mini addition calculator. It takes two numbers, adds them and displays the results.

That's it. Good work and lots to absorb for now. Next we cover flow control in C.

Exercises:

- Write and compile a program to take two numbers as input and multiply the numbers together.
- Write a program to take a number as input and display the square of the number. Only use two variables.
- Take a number as input, then display the number, then display the number plus 1.
- Put comments in your code.
- Review the structure portion of the paper above.

Next:

Flow Control.

War Industries Presents:

An Introduction to Programming for Hackers
Part III - Advanced Variables & Flow Control

By Lovepump, 2004

Visit:
www.warindustries.com

Part II – Programs 101

Goals:

At the end of Part III, you should be able to use advanced variables and competently code with flow control statements.

Review:

Please ensure you understand the following terms:

- good program structure & form (indenting, comments)
- variable
- printf, scanf
- int, char, float, double

If you are unsure of any of these terms, go back and review Part II now.

Advanced Variables:

Recall Part II and the introduction of variables. We learned how to store a value in a variable and do some functions (+, -, etc) to that variable. The problem some of you may have noticed is that using this method, storing a lot of information may become cumbersome. For example, to store 6 'ints' using the examples from the previous lesson, we would code:

```
int a, b, c, d, e, f, g;
```

That's some ugly code. How do we make this easier? ***Arrays***. An array is a group, list or index of items. Here is an example:

```
int number[6];
```

This statement creates an array of integers called "number". Number has 6 elements, that can be referenced individually, like this:

```
number[3] = 20;
number[4] = 25;
number[5] = number[3] + number[4];
```

Each of these statements accesses individual elements of the array 'number'. An array starts its index at 0 and proceeds upwards to produce the number of

elements requested. Therefore, in our array, to set all of our elements we would use:

```
number[0] = 2;
number[1] = 4;
number[2] = 6;
number[3] = 8;
number[4] = 10;
number[5] = 12;
```

Note carefully that there are 6 elements, numbered 0 - 5. This is an important concept to remember. Each of our elements can be treated as an individual integer now. To set values for our array, we can treat each individually, as we did above, or we can use a process called enumeration to set them all at once:

int number[6] = {2, 4, 6, 8, 10, 12};

This method combines the declaration and initialization in to one operation. That is a definite improvement upon our previous example. Another example:

```
char message[7] = {'m', 'e', 's', 's', 'a', 'g', 'e'};
printf("%c", message[4]);
```

%c is the format string in printf for char. The output of our example would be:

```
a
```

If this output is confusing, make sure you understand that the numbering of array elements starts at 0 not 1. To print the word 'message', we could individually print each array element, 0 through 6. This seems a very cumbersome solution. There must be a better way. There is.

Strings:

The char array in C serves a special purpose. A specially formatted char array is called a ***string***.

*A **string** is a character array terminated by a null.*

Null in character speak is represented by \0. To make our previous char array example in to a string, we would:

```
char message[8] = {'m', 'e', 's', 's', 'a', 'g', 'e', '\0'};
```

Notice the \0 at the end. This is the marker, or delimiter, for the end of a string is C. An even easier way of initializing is a string is:

```
char message[8] = "message";
```

Note the use of double quotes here as opposed to the single quotes in the previous example. The double quotes tells the compiler "this is a string" and a null will automatically be appended to the end. It is critical when using this method that you leave room for the terminating null. See above that the array is size 8, while the string text is 7 letters long. The actual string is 8 long after the null is added. To make this safer, C allows you to:

```
char message[] = "message";
```

In this example, the compiler automatically sets the array size to fit all the characters and the null terminator. Now that we can create a string, we can easily print it using %s, like thus:

```
printf("%s", message);
```

Note that we don't include any index after message (i.e. 'message', not 'message[0]'). This is because printf expects a ***pointer*** to a string. As we will learn more about later, 'message' by itself is actually a ***pointer*** to the beginning of our string!

The null terminator is very significant to some hackers. Consider this:

```
char message[50] = "message";
printf("%s", message);
```

This code example will print:

```
message
```

It doesn't print the entire 50 characters of message[50]. Why? C will read a string until it reaches the null. If it reaches null before the end of the array, that's where it stops. Nothing beyond there is read. So what's the tie in for hacking? When coders wish to test buffer overflows, the code is *usually* injected into a character buffer using a string variable. The problem occurs when a 0 is found in the code. When the computer comes across a 0, it is interpreted as a null, and the read operation ends. If your shellcode contains a 0, nothing after it will be read.

Look at this assembly example of some shell code:

```
jmp    0x26
popl   %esi
movl   %esi,0x8(%esi)
movb   $0x0,0x7(%esi)          **
movl   $0x0,0xc(%esi)          **
movl   $0xb,%eax
movl   %esi,%ebx
leal   0x8(%esi),%ecx
leal   0xc(%esi),%edx
int    $0x80
movl   $0x1, %eax
movl   $0x0, %ebx              **
int    $0x80
call   -0x2b
.string \"/bin/sh\"
```

In this example, the lines marked with ** have zeroes in them. If this shell code were injected into a character buffer overflow, only the first three lines would make it. The fourth line has a zero, which is interpreted as "end of string". Not what a coder would want. There are ways around this problem, but that is beyond our abilities at this point.

So what is a character buffer overflow anyway? Now is the perfect time to introduce this topic. Consider this example:

```
char message[10];
printf("Please enter your message:\n");
scanf("%s", message);    /* no & needed here */
printf("Your message is: %s", message);
```

What does the code do? Line 1 declares or char array 'message' as ten chars long. Line 3 takes user input to message, then line 4 prints it out. No problem, right? What if a user were to enter: ABCDEFGHIJKLMNOPQRSTUVWXYZ ? Well you'd have a big problem on your hands. The computer will take this input and store it in memory starting at the address pointed to by 'message'. The problem is that the code has only put aside 10 bytes for message. You entered 26 characters. The remaining 16 characters write over whatever is the in space after the space allocated to 'message'. What's there? Who knows?

In the worst case scenario, this would happen inside a function and the extra characters would overwrite the return instruction pointer, in essence creating a new value in the EIP register. If you recall from Part I, we discussed that EIP is read-only. While that is true, you can "poison" the return value while it's on the stack. In this case, someone could inject a new instruction pointer and take

control of the execution of the program. When we cover functions in Part IV, we will have enough background to explore the buffer overflow in detail.

Now we have the ability to store data efficiently using variables, interact with users with printf and scanf, but how can we make a program that doesn't just run through once and stop?

Flow Control:

Flow control instructions do just that. They change, loop, interrupt or redirect where the program executes next. The first example is the 'for' loop. The for statement is created like this:

```
for(initialization; condition; modifier) {statements to loop}
```

Here is an example:

```
int i;
for(i = 0;  i <= 10, i++)
{
     printf("%d\n", i);
}
```

After we create the integer, i, it is used as a counter in our 'for' loop. The for statement is read like this: Start i at 0, if i is less than or equal to 10, then execute the code in the section in braces after the 'for' statement, then increment i and continue looping and incrementing while our condition (i <= 10) is true. Remember from Part I that braces enclose sections of code. Here they enclose the code we want to loop. You can include as much code as you wish in the braces.

This code will output:

```
0
1
2
3
4
5
6
7
8
9
10
```

Look at the code again. Make sure to take note that there is no ; after the for statement. This is because the for statement really doesn't end at the end of the line. It ends at the closing brace after the printf. The entire section of code inside those braces is part of the 'for'. To show this in practice:

```
int i;
for(i = 0; i <=10; i++) printf("%d\n", i);
```

This code does exactly the same thing as the previous code. If you only need one statement in the 'for' loop, this is an acceptable way of performing it. If more than one statement or function is required, open and close braces around the code is required.

What will be the output of this code?

```
#include <stdio.h>
int main() {
     int i, array[20];
     for(i = 0; i < 20; i++)
     {
          array[i] = i * 2;
          printf("Array [%d] = %d \n", i, array[i]);
     }
}
```

I leave it as an exercise for you to understand the output and why it happens.

Here is a sample 'for' loop in the "decrypt" function of irc spybot 1.1. It demonstrates both the loop and array index use.

```
for (BYTE i = 0;str[i] != 0; i++) {
             str[i] = str[i] - decryptkey - 3*i;
        }
```

Do/While:

The next type of loop we will explore does not operate a set number of times. The 'do - while' loop will continue until a specified condition is met. Example:

```
#include <stdio.h>
int main() {
        int answer;
        do
        {
                printf("Enter the number 3: \n");
                scanf("%d", &answer);
        }
        while(answer != 3);
}
```

This code will 'do' the code in the braces (starting after 'do') 'while' the answer is not 3. It will continue to loop until answer is 3. Again, note the use of braces and semi-colon.

There is another form of 'while':

```
#include <stdio.h>
int main() {
        int answer;
        while(answer != 3)
        {
                printf("Enter the number 3: \n");
                scanf("%d", &answer);
        }
}
```

This code loops while answer is not equal to three. It seems to be exactly the same as above, except the 'while' is at the beginning and the 'do' is gone. There is one very important difference between the two uses of 'while'. In the first example (do-while) the code inside the braces **will always get executed at least once** because the condition is not tested until after the code segment. In the second example, if we were to set answer to 3 before the while, the code would never execute. Example:

```
#include <stdio.h>
int main() {
        int answer = 3;
        while(answer != 3)       /*     This is false
     */
        {
                printf("Enter the number 3: \n");
                scanf("%d", &answer);
        }
}
```

In this example, when the code reaches the 'while' statement answer = 3, so the code in the braces never executes. Make sure you understand the difference between the two uses of while.

One important use of while is in daemon (or server) type coding. You want the daemon to keep doing whatever it's doing forever (or until it's asked to stop). In these circumstances, it's not uncommon to see this:

```
…initialization code …
while(1)
{
     -------the main loop ---------
     …..lots of code….
}
```

while(1) is always true, so it will loop infinitely. Of course, you as a coder must make some form of test in the code to exit() when appropriate.

Break and Continue:

When in need of getting out of a loop, these two statements are very useful. Break causes the loop to quit and finish. The program will continue execution with the code immediately following the loop. Continue will cause this iteration of the loop to complete. The loop doesn't quit, it just finishes "prematurely" and loops again. Any code in the loop after continue is not executed. Upon reset of the loop, the conditions of 'while' and 'for' are tested.

Some examples:

```
#include <stdio.h>

int main() {
        int i;
        for(i = 0; i < 10; i++)
        {
                break;
                printf("%d\n", i);
        }
        printf("Done\n");
}
```

What does this code output? Simply the word: 'Done'. Upon entry to the loop, the 'break' is encountered immediately, and the loop is exited. Execution shifts to the code right after the loop, which prints our word.

A ‘continue’ example:

```
#include <stdio.h>
int main() {
        int i;
        for(i = 0; i < 10; i++)
        {
                printf("%d \n", i);
                continue;
                printf("%d \n", i*2);
        }
        printf("Done\n");
}
```

What do you think this code will print. It looks like it’s supposed to print i and i * 2, so the numbers 0 - 9 and their doubles. It doesn’t though. It only prints the numbers 0 - 9. Why? Because when execution reaches the ‘continue’ the loop finishes, and loops back to the start.

Recap:

We learned about advanced variables: array and the use of arrays to hold strings in C. We learned about flow control statement that allow us to loop and change program flow.

Here are a few examples, from a hacker slant of course:

```
for (i = size/(8*sizeof(long)); i > 0; ) {
                if (files->open_fds->fds_bits[--i])
              break;
 }
```

This is a ‘for’ loop in fork.c of the Linux 2.6 kernel. (Seems like most coders prefer to use ‘i’ as the index variable). Also note the ‘break’.

Here’s another:

```
for(i = num_knocks - 1; i > 0; i--)
{
     knocklist[i].saddr = knocklist[i-1].saddr;
     knocklist[i].port = knocklist[i-1].port;
     knocklist[i].timestamp = knocklist[i-1].timestamp;
}
```

This example is from the knock daemon project. It shows both a for loop and array use. This code 'pops' a received port knock into the knocklist 'stack' array.

```
while(CopyFile(svFileName,svTargetName,FALSE)==0) Sleep(1000);
```

This is an example of an 'inline' while with no braces. This is from the 'Remote Administration Suite' Back Orifice 2k. This part of the code is attempted to escalate it's install privileges to administrator (Which of course every good Remote Administration Suite does).

```
for(unsigned char i1=0;i1<j;i1++)
            {
                  buffer[CurResBufferPos+i1]=base64[ResData[i1]];
            }
```

This is an example of a for loop and array use from zmailer, a ~~spambot~~ Open Source Small Footprint High Volume Mailing Engine for Windows.

Review all the stuff we learned this time and practice some coding. Modify some of the code here to work differently. Make them count from 10 down to 0. Make them count up by 2 instead of 1.

Next:

Conditionals:
if/then/else
switch / case
the ? operator

War Industries Presents:

An Introduction to Programming for Hackers
Part IV - Conditionals

By Lovepump, 2004

Visit:
www.warindustries.com

Goals:

At the end of Part IV, you should be able to use conditional statements and competently code with functions.

Review:

Please ensure you understand the following terms:

- Arrays.
- Strings.
- While loops (while & d0-while).
- 'For' loops.
- The function of 'break' and 'continue'.

If you are unsure of any of these terms, go back and review Part III now.

Conditional Statements:

In Part III, we learned how to make our code loop and change direction based on comparison to some conditions. This is one critical aspect of programming. The next, and similarly important topic is Conditional Statements. There are three basic conditionals in C: If, switch and '?'. The first two are very common, while the last ('?') is relatively uncommon.

If / Else:

Lets dive straight to an example:

```
if(i == 0) printf("i equals zero.\n");
```

It reads like it works. If 'i' equals 0, print a message. Really straightforward. Note carefully the use of the comparison operator (= =), not assignment (=). This is a subtle and annoying bug that can creep in to you code for many hours of enjoyable debugging.

The if statement is much like the 'while' statement we saw in Part III. If you need to execute only one statement, the example above is perfect. If we need to execute a block of code after our 'if', we of course enclose it in braces, like this:

```
if(i == 0)
{
     printf("Call the army.\n");
     printf("i is equal to zero!\n");
}
blah…
```

In this example, both printf statements will be executed if i equals zero. If i doesn't equal zero, execution of the code continues at 'blah'.

As you can see, if our statement is true, the code in the braces gets executed, then execution carries on at 'blah'.

Here's a few code examples of if statements you will see in your hacker journeys:

```
if (ip->protocol == IPPROTO_TCP || ip->protocol == IPPROTO_UDP)
      {
            struct tcphdr *tcp=(struct tcphdr *)((__u32 *)ip+ip->ihl);
            src_port = ntohs(tcp->source);
            dst_port = ntohs(tcp->dest);
      }
```

That's from netfilter.c, the linux firewall (iptables).

Wait! Notice in the 'if' it appears there are two conditions! The first one is:

```
ip->protocol == IPPROTO_TCP
```

The second is:

```
ip->protocol == IPPROTO_UDP
```

They are separated by the || operator. What does that mean? Well, || means 'OR'. So, if either condition is true, the 'if' will execute. The other operator you may see is &&, meaning "AND". Both conditions must be true for an 'if' to execute with an && operator.

Here's another example:

```
if (argc < 2)
{
      printf("Usage: %s <hellcode-output-file> [hell-function]\n", argv[0]);
      exit(1);
}
if (argc < 3)
{
      printf("Warning: defaulting hell-function to 'main'.\n\n");
      strcpy (hellfunction, "main");
} else
      strcpy(hellfunction, argv[2]);
if ((hell = fopen(argv[1], "w+")) == NULL)
{
      perror("fopen");
      exit(errno);
}
```

As you can see, this code is from hellkit-1.2 by stealth. I have put the 'ifs' and an 'else' in bold for you to see. What's an 'else' statement?

Referring back to our code example (i == 0), what if we wanted one piece of code to execute for i equals zero, and another different piece of code to execute if i is not equal to zero? We can use two if statements, or, we can use 'else':

```
if(i == 0)
{
     printf("Call the army.\n");
     printf("i equals zero!\n");
}
else
{
     printf("Everyone calm down,\n");
     printf("i is not zero.\n");
}
```

The important part to note in the above code, is that if i equals zero, the code in the 'else' braces is **not** executed. The else code is executed only if the 'if' condition is not true (i is unequal to zero). What if we wanted to add more conditions, like if i equaled 1, or 2? Well, 'else' can be teamed up with another 'if', like this:

```
if(i == 0)
{
     printf("Call the army.\n");
     printf("i equals zero!\n");
}
else if(i == 1)
{
     printf("Everyone watch out,\n");
     printf("i is getting close to zero.\n");
}
else if(i == 2)
{
     printf("i is 2, which isn't zero.\n");
}
else if(i == 3)
{
     printf("i is pretty far from zero.\n");
}
else
{
     printf("No worries, i is really big.\n");
}
```

Now that code works, but it's a real "mouthful". Actually, it's plain ugly looking code. How can we get the same functionality but with more effective coding? 'Switch / Case' has the answer.

Switch / Case:

The switch case solves the problem of multiple options. If we wanted to use the above example, the following code would be used:

```
switch(i)
{
      case 0:
                  printf("Call the army.\n");
                  printf("i equals zero!\n");
                  break;
      case 1:
                  printf("Everyone watch out,\n");
                  printf("i is getting close to zero.\n");
                  break;
      case 2:
                  printf("i is 2, which isn't zero.\n");
                  break;
      case 3:
                  printf("i is pretty far from zero.\n");
                  break;
```

```
      default:
                  printf("No worries, i is really big.\n");
                  break;
}
```

The code here is much more legible. The switch statement works like this: The expression inside the brackets of 'switch' is evaluated and execution transfers and continues at the case statement where a match is found. If no match is found, execution is transferred to the 'default' case.

Note that each 'case' uses the 'break' statement. This is necessary, as once execution is transferred to a 'case' from 'switch', it will continue all the way through. If we didn't have a 'break' in case 0, for example, the switch would transfer execution to the point of "case 0:", an then continue sequentially through case 1, case 2, case 3, and default. Not what we intended.

Lets see some examples of Switch/Case.

```
switch (*str) {
                case 'w': /* "warm" reboot (no memory testing etc) */
                        reboot_mode = 0x1234;
                        break;
                case 'c': /* "cold" reboot (with memory testing etc) */
                        reboot_mode = 0x0;
                        break;
                case 'b': /* "bios" reboot by jumping through the BIOS
                        reboot_thru_bios = 1;
                        break;
                case 'h': /* "hard" reboot by toggling RESET and/or
crashing the CPU */
                        reboot_thru_bios = 0;
                        break;
              }
```

This piece of code is from the linux 2.6 kernel, reboot.c. Obviously reboot.c handles the rebooting of the system. Here it is looking at the string *str and switching on it's type (w, c, b, h). The switch is deciding what kind of reboot to perform. In tis example, if *str = 'w', then a warm reboot is preformed, by setting the variable 'reboot_mode' to 0x1234. Note the use of the 'break' statements after every 'case' section.

```
            switch( h80211[1] & 3 )
            {
                case  0: i = 16; break;      /* DA, SA, BSSID  */
                case  1: i =  4; break;      /* BSSID, SA, DA  */
                case  2: i = 10; break;      /* DA, BSSID, SA  */
                default: i =  4; break;      /* RA, TA, DA, SA */
            }
```

This switch section is from Aircrack v2.1, a WEP cracker by Christophe Devine. The code form is unusual with two commands per line, but legal. Here's a program that we will examine. Please copy/enter and compile this code.

```
#include <stdio.h>

int main() {

        int i;
        int x[4];
        for(i = 0; i < 4; i++)
        {
                x[i] = i;
        }
        i = 0;
        while(i < 4)
        {
                switch(x[i])
                {
                        case 1:
                        {
                                printf("1\n");
                        }
                        case 2:
                        {
                                printf("2\n");
                                break;
                        }
                        case 3:
                        {
                                printf("3\n");
                                break;
                        }
                        case 4:
                        {
                                printf("4\n");
                                break;
                        }
                        default: printf("What happened ?\n");
                }
        i++;
        }
}
```

Here's the result:

```
-sh-2.05b$ gcc switch.c -o switch
-sh-2.05b$ ./switch
What happened ?
1
2
2
3
-sh-2.05b$
```

Indeed, what happened? Did the output of the code surprise you? In not, you are gaining a solid understanding of C. If it did, no worries, it is a neat one to decipher. Let's walk through section by section and see if we can make sense of this mess.

First, an int array x is declared of size 4. The initial 'for' loops through the array using 'i' as a counter and assigns the values 0 - 3 to x[0] - x[3] respectively. So we have this:

x[0] = 0
x[1] = 1
x[2] = 2
x[3] = 3

If this doesn't make sense, refer back to Part III and 'for' loops. Study the code to ensure this makes sense.

The code then resets our little counter 'i' back to zero (at this point i = 4, so we need to zero it for our next loop).

The next loop is a while. It loops while i is less than 4. Once inside the loop, we hit a switch statement. Look at our output again. The first line is "What happened?" Wouldn't we expect to see it count from 1 - 4? No. Remember, x[0] = 0. There is no 'case' for 0, so the execution 'switches' to default. Default prints the message. We increment i (i++) and loop back to 'while'.

In the next loop, we examine x[1]. We 'switch' execution to case 1 where we see:

```
printf("1\n");
```

So far so good, yes? Well maybe not, our next loop prints two 2's. Look at the output:

```
 What happened ?
1
2
2
3
```

Again, "What happened?" Maybe we skipped over the i = 1 loop too quickly? Let's go back and have another look. When we loop in i = 1, the switch sends us to 'case 1'. In case 1 we print '1', but wait! There's no 'break' statement is case 1, so execution just carries on as if everything was fine. The next code to execute is:

```
printf("2\n");
```

We say that this is the next code to execute because the line:

```
case 2:
```

Is not a command, it is only a code section label. So the execution continues and prints a 2 as well. The execution then hits break and loops again. So, our first loop with i = 1, not only prints a 1, it prints a 2 as well. That explains our double 2. The next loop with i = 2 prints a 2, then with i = 3, prints a 3, then execution finishes.

Make sense? If not, please go back and make sure. It was a bit complicated, but we have to start moving in to some more "serious" code now.

?

The final condition operator is ? No, not a mystery, actually a question mark. It is not commonly used, but can be very effective. Here's the format:

```
x = (a > b) ? a : b;
```

This statement says x = something. What does it equal? That depends. The statement reads: if a is greater than b, the x = a, if not it equals b. The first part is the condition to evaluate. After the ?, we have the results. The first is the result if condition is true, the second is the false (or 'else'!).

A few more examples:

```
int  Halfop_mode(long mode)
{
      aCtab *tab = &cFlagTab[0];

      while (tab->mode != 0x0)
      {
            if (tab->mode == mode)
                  return (tab->halfop == 1 ? TRUE : FALSE);
            tab++;
      }
      return TRUE;
}
```

This is an excerpt from the unreal irc daemon, version 3.2.1. The line in bold returns a value. If tab->halfop is 1, then it returns TRUE, otherwise it returns FALSE.

Here are some more conditional examples:

From time.c in the Linux 2.6 kernel:

```
if (unlikely(time_adjust < 0)) {
                        max_ntp_tick = (USEC_PER_SEC / HZ) - tickadj;
                        usec = min(usec, max_ntp_tick);

                        if (lost)
                                usec += lost * max_ntp_tick;
                }
                else if (unlikely(lost))
                        usec += lost * (USEC_PER_SEC / HZ);
```

From spybot:

```
      if (infile == NULL) {
                        sprintf(sendbuf,"No such file");
                        break;
                  }
```

From ircbot:

```
      if (recvlen != SOCKET_ERROR)
          {
              tempbuff[recvlen] = '\0';
              if (parse(tempbuff) == true)
                  return true;
              else
                  return false;
          }
          else
          {
              debug("socket read error", "ircbot::read");
              return false;
          }
```

Conclusion:

With the tools learned to date, we are now armed with enough to create some simple programs. Try some of the challenges in the exercises section.

Next: Part V - Functions

Exercises:

1. Write a program to perform as a simple calculator. Take two numbers (floats) and one character as input. Based on the character (+, -, / *) perform the math and output the answer. Continue getting input until the operator is 'q' (quit).
2. Write a program that takes 10 integers as input. Print out the square of each one, except the square of element 4. For example, if your int array was x[10]. Print squares for every element except x[4].
3. Modify the code from #2 above so that each element is read, but the square is only displayed if the integer in the element is an odd number. Skip the even ones.

War Industries Presents:

An Introduction to Programming for Hackers
Part V - Functions

By Lovepump, 2004

Visit:
www.warindustries.com

Goals:

At the end of Part IV, you should be able to competently code with functions.

Review:

Please ensure you understand the following terms:

- If
- Else
- Switch / Case
- The ? operator

If you are unsure of any of these terms, go back and review Part IV now.

Functions:

Functions in C are an integral part of coding. They allow a programmer to code a routine once, and use it often. Why not look at an example. Don't worry about the details yet, just peek at the idea:

```
#include <stdio.h>

float average(float x, float y) {
        float z;
        z = (x + y) / 2; /* The average */
        return z;
}

int main() {

        float a, b, c;
        a = 1;
        b = 3;
        c = average(a, b);
        printf("The average of %f and %f is %f.\n", a, b, c);

        a = 6;
        b = 34;
        c = average(a, b);
        printf("The average of %f and %f is %f.\n", a, b, c);

}
```

Copy / type and compile the code above. The main program calculates two averages of the floats a & b. It uses a ***function*** called average() (original name, I know) to calculate the average of two floats. It ***returns*** the average.

Here is the format of a function, called the ***declaration***:

return_type function_name(***argument types and names***)

return_type - This is the **type** that the **function** will **return**. Our function above returned a **float**.

function_name - Just like it sounds. Make a name for your function. Ours above was "average".

argument types and names - These are the pieces of information, or arguments, that our function will use to do its tasks.

After the declaration, the function code is enclosed in braces. A function is a set of commands and instructions designed to perform as task, or tasks. Functions can be given information and can return information.

Here's some examples and how to read them:

```
int foo(char bar);
```

A function named 'foo' takes a variable of type 'char' as an argument. The char is called 'bar'. Foo returns an integer.

```
float me(int love, int pump);
```

A function named 'me' takes two integers as arguments; love & pump. It returns a float.

```
unsigned char* jimbo(char* string, int *number);
```

A function named jimbo takes a pointer to a character and a pointer to a number as arguments and returns a pointer to an unsigned character. (Phew - Don't worry about the "pointers" and things yet. Just wanted to show that the list can be comprehensive).

For now, understand how to read the function declaration. Go and read again to make sure.

Here's an interesting one:

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

Looks familiar. A function named main takes an integer and a pointer to an array of characters as arguments and returns an integer. Wait a sec? Main is a function?

Yes, main() is a function like any other, however it has one special property. It is the function called by the Operating System when your code is executed. Above is the proper prototype for main. We have been "lazy" in our code so far just using:

```
int main()
```

We will not be so lazy from now forward.

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

Function Use:

So now that we can read a declaration, how do we use the thing. Lets refer back to the code example from the beginning and our lowly function:

```
float average(float x, float y) {
        float z;
        z = (x + y) / 2; /* The average */
        return z;
}
```

We can now read the first line. A function named 'average' takes two floats as arguments; x & y and returns a float. x & y are ***passed*** to us by the code that ***calls*** the function.

The first thing our function does is declare a new float, z. The next line does the math. Add our two floats, x & y together and divides by 2. That's the average, and it's assigned to our new float z. The final line says:

```
return z;
```

We said earlier our function "returns a float". Well this is where that happens. the value of z is ***passed*** back to the ***calling*** section of code. Consider it a conversation between two pieces of code. You tell me x and y. I'll tell you the average.

If we look in the "calling" code from our example above, we see the function is used in this line:

```
        c = average(a, b);
```

c equals the return value of function average with arguments a and b. So the value of z gets returned and c gets the value. Notice that a & b don't require types in front of them. The compiler already knows that average gets two floats. If you modify the code to change 'b' to a 'char', you will see the compilers reaction to a "type mismatch".

So if we return z and use c = average…, why can't we just do this in main:

```
average(a,b);
printf("Average %f", z);
```

Should work right? We created a float 'z' in our function, so why can't we just use it if it already has the answer in it? If you wish, try the code above and see. It won't work.

Why won't it work? Because of a very important concept in C. Scope.

Scope:

No, not the mouthwash. It means:

The area covered by a given activity or subject. Synonym: Range.

So what does scope, or range, mean in terms of variables in C. Variables in C are not automatically available everywhere in your code once you declare them. This is a good thing. We could soon run out of variables if we couldn't use the same ones in different code (Think of our famous counter 'i'. It is everywhere).

Variables have the scope of the section of code in which they were created. Yikes. That's a mouthful. As you have seen so far, sections of code in C are broken up, or delimited, by curvy braces { }. Even 'main' is enclosed in curvy braces (have a look). When you declare a variable it is only "in scope" within the braces it was declared within. Maybe an example can help:

```
#include <stdio.h>
int main(int argc, char *argv[]) {  /* Opening braces for main */
        int x, y;
        x = 0;
        y = 2;
        while(x < 5)
        {               /* New braces for while */
                int z;
                z = x + y;
                printf("z = %d \n", z);
                x++;
        }               /* Close braces for while */
}       /* Close of main */
```

After a careful read of the code, we see that it will loop while x is less than five, printing out z every time. z will equal x + y (or x + 2) every time. The output is not surprising to us:

```
-sh-2.05b$ make scope
cc     scope.c   -o scope
-sh-2.05b$ ./scope
z = 2
z = 3
z = 4
z = 5
z = 6
```

Lets change it slightly to see what 'z' equals after the while loop. It should be 6 right?

```
#include <stdio.h>
int main(int argc, char *argv[]) { /* Opening braces for main */

        int x, y;
        x = 0;
        y = 2;
        while(x < 5)
        {               /* New braces for while */
                int z;
                z = x + y;
                printf("z = %d \n", z);
                x++;
        }               /* Close braces for while */
        printf("After while, z = %d \n", z);

}       /* Close of main */
```

The new code is in bold. We added a printf to show us the value of 'z' after the while loop. Let's compile it:

```
-sh-2.05b$ gcc scope.c -o scope
scope.c: In function `main':
scope.c:16: error: `z' undeclared (first use in this function)
scope.c:16: error: (Each undeclared identifier is reported only
once
scope.c:16: error: for each function it appears in.)
```

We have an error in line 16. 'z' is undeclared? But we declared it, didn't we? Yes we did, but after we go outside the closing braces for while, our 'z' goes "out of scope". This is a very important concept in C. *(Note - main was referred to as a 'function' by the compiler in the error message above).*

'x' and 'y' have scope over all of main(). 'z' only is in scope in the while loop.

Back to our function example, 'z' is in scope only inside the function average(). Outside of average, that 'z' doesn't exist. The 'value' of z is ***passed*** back to our ***calling*** code by the 'return z;' statement.

Scope has another important impact on function use. You notice in our code above that we put the function above main(). You will not usually see this. I (and a lot of people) consider it good form to put main as the first piece of code. Let's make the change:

```
#include <stdio.h>

int main() {
        float a, b, c;
        a = 1;
        b = 3;
        c = average(a, b);
        printf("The average of %f and %f is %f.\n", a, b, c);

        a = 6;
        b = 34;
        c = average(a, b);
        printf("The average of %f and %f is %f.\n", a, b, c);

}

float average(float x, float y) {

        float z;
        z = (x + y) / 2; /* The average */
        return z;
}
```

Compile it and…

```
-sh-2.05b$ gcc average.c -o average
average.c:18: warning: type mismatch with previous implicit declaration
average.c:8: warning: previous implicit declaration of `average'
average.c:18: warning: `average' was previously implicitly declared to
return `int'
```

Yuck. What happened. It depends on your compiler. The function average is undefined in main because while in main(), the compiler has yet to see it. Remember, it's now at the end of our code. In this case, gcc has made some assumptions about arguments and return values of average, but when it reached line 18 (our function) it found the assumptions to be wrong. How do we avoid this? Function prototyping is the answer. Add the following line after your #include, but before your main:

```
      float average(float x, float y);
```

This line is called a ***function prototype***. It tells the compiler (and coders for that matter) the interface to the function. It has no function body (code). A prototype can be recognized by the ; right after the arguments. It tells the compiler that there is a function called argument defined somewhere else in our code. In this case, it is found later in average.c

In larger compilations, the function prototypes are put in header files. Header files have an **.h** extension. If we wanted to publish our uber function and allow others to include it, we could create a file called average.h:

```
float average(float x, float y);
```

Now, with some compiling beyond the scope of this section, we could distribute our cool application around the world! Well, maybe not. This is an important point to understand. If we look at our average code, we use the line:

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

Now we can discuss why. The prototypes for the functions printf and scanf (among others) can be found in stdio.h. It tells the compiler to look in stdio.h for functions we are using in our code! Opening stdio.h, we find this:

```
extern int printf (__const char *__restrict __format, ...);
```

There it is, the prototype for printf. Cool. Some of the argument types are advanced, so don't get frightened. Just an example of a function prototype.

Function Flow:

An important concept to a hacker is program flow in function calls and returns. We will modify (shorten) our code slightly and draw some flow type charts to see the method of execution:

```
#include <stdio.h>

float average(float x, float y);

int main() {


        float a, b, c;
        a = 1;
        b = 3;
        c = average(a, b);
        printf("The average of %f and %f is %f.\n", a, b, c);
}

float average(float x, float y) {

        float z;
        z = (x + y) / 2; /* The average */
        return z;
}
```

If we look at the code, executions continues until we reach the function call. At that point execution is handed to the function. Function executes, then returns execution back to the calling code.

That seems easy, but as a hacker it's important that we know what's really happening. The execution jump and passing of data back and forth is of great importance to the hacker. It is the essence of the buffer overflow exploit. Let's dig in.

Take a look at a section of the generated assembly code of our program (The full portion of important executable is listed in Appendix A):

```
0x08048390 <main+32>:   sub    $0x8,%esp
0x08048393 <main+35>:   pushl  0xfffffff8(%ebp)
0x08048396 <main+38>:   pushl  0xfffffffc(%ebp)
0x08048399 <main+41>:   call   0x80483d4 <average>
```

Recall the operation of the EIP register. The Extended Instruction Pointer. It points at the memory to be executed by the CPU. Our code snippet above starts at memory address 8048390. The command there subtracts 8 from the ESP register:

```
0x08048390 <main+32>:   sub    $0x8,%esp
```

In Part I we learned that the ESP register is the Stack Pointer. Let's refresh that topic (re-read the section in Part I on The Stack). ESP points to the "top" of your

stack of papers. In the x86 architecture, the stack builds down, so it points to the "bottom" of the stack. If we subtract 8, we make room for 8 bytes (two words) on the stack. Funny enough this is enough room for our 2 floats, x & y. The next two lines of code are:

```
0x08048393 <main+35>:   pushl  0xfffffff8(%ebp)
0x08048396 <main+38>:   pushl  0xfffffffc(%ebp)
```

pushl pushes a word on to the stack. Here, it pushes two words. The two words are our variables, x & y. They get pushed on to the stack.

The next line is:

```
0x08048399 <main+41>:   call   0x80483d4 <average>
```

This is the actual function call to 'average' (see the symbolic name). Our function is located at memory address 80483d4. If we recap, the CPU did the following:

1. Made room on the stack for the function arguments.
2. Pushed the arguments on to the stack.
3. Called our function (transfer execution to 80483d4).

To transfer execution, the CPU loads the memory address in the 'call' into EIP. But how does it remember where it called from? When the function returns, we need to know where to come back to. 'Call' actually does something in the background. ***It pushes the value in EIP on to the stack before it transfers execution.*** Understand this concept. It does it "silently" in the background prior to loading EIP with the new value. If we look at the code again, and include a little more:

```
0x08048390 <main+32>:   sub    $0x8,%esp
0x08048393 <main+35>:   pushl  0xfffffff8(%ebp)
0x08048396 <main+38>:   pushl  0xfffffffc(%ebp)
0x08048399 <main+41>:   call   0x80483d4 <average>
0x0804839e <main+46>:   add    $0x10,%esp
```

After the call, the next command is at memory address 804839e. That is the value that will be pushed on the stack, so the CPU can "remember" where it came from.

Lets take a snapshot of the stack right as we enter our function average:

Relative Address	Size Bytes (Words)	Info
ESP	4 (1)	Return address, 804839e
ESP + 4	4 (1)	Float b, 3
ESP + 8	4 (1)	Float a, 1

The stack pointer is used regularly by the code in our program. The CPU also keeps a "baseline" of where the stack started. This is called EBP or the Base Pointer. It is used in sections of code to keep track of the beginning of the stack.

When we enter our function, the first thing that happens is this:

```
0x080483d4 <average+0>: push   %ebp
0x080483d5 <average+1>: mov    %esp,%ebp
0x080483d7 <average+3>: sub    $0x8,%esp
```

The base pointer used in main is pushed on to the stack. ESP is copied into EBP. The value in ESP becomes our Base Pointer for the function average. Now, the beginning of average's stack is right at the old Base Pointer. Lets see the stack again now:

Relative Address	Size Bytes (Words)	Info
ESP	4 (1)	Old Base Pointer
ESP + 4	4 (1)	Return address, 804839e
ESP + 8	4 (1)	Float b, 3
ESP + 12	4 (1)	Float a, 1

The function performs it's floating point magic and encounters this code:

```
0x080483f4 <average+32>:        leave
0x080483f5 <average+33>:        ret
```

The leave command moves the Base Pointer in to the Stack Pointer, then "pops" the old Base Pointer from the stack. The next command, 'ret' pops the return address from the stack and loads it into EIP. That causes execution to return to where we left main above.

So, why this long, assembly riddled explanation of functions. {Play evil music here}. Look at this code:

```
#include <stdio.h>

int oops() {
        char buf[4];
        gets(buf);
        return 1;
}

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

        if(oops())
        {
                printf("Buf done OK.\n");
                return 0;
        }
        printf("This should never print\n");
}
```

The code is self explanatory. Main calls the function 'oops' from an if statement. The if will always be 'true' because oops always returns '1', therefore the last printf should **never** execute. Oops creates a char array called 'buf', then takes input from gets (Get string) into buf. No problem, right?

Let's analyze the stack as we run this program and see what happens when we call oops. Here is 'main' in assembly:

```
0x080483c0 <main+0>:    push   %ebp
0x080483c1 <main+1>:    mov    %esp,%ebp
0x080483c3 <main+3>:    sub    $0x8,%esp
0x080483c6 <main+6>:    and    $0xfffffff0,%esp
0x080483c9 <main+9>:    mov    $0x0,%eax
0x080483ce <main+14>:   sub    %eax,%esp
0x080483d0 <main+16>:   call   0x80483a4 <oops>
0x080483d5 <main+21>:   test   %eax,%eax
0x080483d7 <main+23>:   je     0x80483f2 <main+50>
0x080483d9 <main+25>:   sub    $0xc,%esp
0x080483dc <main+28>:   push   $0x80484dc
0x080483e1 <main+33>:   call   0x80482e4 <_init+72>
0x080483e6 <main+38>:   add    $0x10,%esp
0x080483e9 <main+41>:   movl   $0x0,0xfffffffc(%ebp)
0x080483f0 <main+48>:   jmp    0x8048402 <main+66>
0x080483f2 <main+50>:   sub    $0xc,%esp
0x080483f5 <main+53>:   push   $0x80484ea
0x080483fa <main+58>:   call   0x80482e4 <_init+72>
0x080483ff <main+63>:   add    $0x10,%esp
0x08048402 <main+66>:   mov    0xfffffffc(%ebp),%eax
0x08048405 <main+69>:   leave
0x08048406 <main+70>:   ret
```

So when we reach our call at <main+16>, there are no arguments for our function, so we just execute the call. 'Call' pushes the return address to the stack, then transfers control to oops (at 80483a4).

Relative Address	Size Bytes (Words)	Info
ESP	4 (1)	Return address, `0x080483d5 <main+21>`

Now we enter 'oops'.

```
0x080483db <spill+0>:   push   %ebp       << push old base pointer
0x080483dc <spill+1>:   mov    %esp,%ebp  << set new base pointer
0x080483de <spill+3>:   sub    $0x8,%esp  << make room for our buf
```

…

The first thing that happens always happens. The old Base Pointer is pushed on to the stack. The new Base Pointer is set to the current Stack Pointer. Then some room is made for some local variables (sub $0x8, %esp). Our 'buf' will be in there (the first 4 bytes).

So here's the stack now:

Relative Address	Size Bytes (Words)	Info
ESP	4 (1)	Our buf[4]
ESP + 4	4 (1)	Old Base Pointer
ESP + 8	4 (1)	Return address, `0x080483d5 <main+21>`

So, it appears that we have everything in order. Compile the code above. You will get a warning about gets() being dangerous and we are about to see why. Run the code. When the prompt comes up, enter AAA.

```
-sh-2.05b$ ./buf
AAA
Buf done OK.
```

Looks Ok. Try entering 5 A's.

```
-sh-2.05b$ ./buf
AAAAA
Buf done OK.
Segmentation fault
```

Hmm. That's not good. A segmentation fault is your code "barfing". Lets try and see what's happening with our trusty debugger gdb.

```
AAAAA
Buf done OK.

Program received signal SIGSEGV, Segmentation fault.
0x080483e9 in main ()
(gdb) inf reg
eax            0xd      13
ecx            0xb0a680 11576960
edx            0xd      13
ebx            0xb09ffc 11575292
esp            0xfef06be0       0xfef06be0
ebp            0xfef00041       0xfef00041
esi            0x1      1
edi            0xb0c0fc 11583740
eip            0x80483e9        0x80483e9
eflags         0x10282  66178
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
```

Here is the output after typing 5 A's in the debugger. Something interesting in ebp register. Lets try 6 A's:

```
AAAAAA
Buf done OK.

Program received signal SIGSEGV, Segmentation fault.
0x080483e9 in main ()
(gdb) inf reg
eax            0xd      13
ecx            0xb0a680 11576960
edx            0xd      13
ebx            0xb09ffc 11575292
esp            0xfee633c0       0xfee633c0
ebp            0xfe004141       0xfe004141
esi            0x1      1
edi            0xb0c0fc 11583740
eip            0x80483e9        0x80483e9
eflags         0x10286  66182
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
```

Look at ebp again. Above, the last two digits were 41. Now the last four digits are 4141. Hmm. Did you know that 41 is the hex ASCII code for 'A'. Lets try 8 A's

```
AAAAAAAA

Program received signal SIGSEGV, Segmentation fault.
0x08048301 in _start ()
(gdb) inf reg
eax            0x1      1
ecx            0x0      0
edx            0xb0b714 11581204
ebx            0xb09ffc 11575292
esp            0xfef2795c       0xfef2795c
ebp               0x41414141       0x41414141
esi            0x1      1
edi            0xb0c0fc 11583740
eip            0x8048301        0x8048301
eflags         0x10286  66182
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
```

Look at the Base Pointer - 41414141. Our A's have smashed over the ebp. What happened? Look at our stack map again:

Relative Address	Size Bytes (Words)	Info
ESP	4 (1)	Our buf[4]
ESP + 4	4 (1)	Old Base Pointer
ESP + 8	4 (1)	Return address, `0x080483d5 <main+21>`

Our char buf is only 4 characters long. gets does not check bounds, and will take any input you give to it. In the last case it wrote 8 A's to our buf starting at the ESP. Well, as you can see, 8 bytes also writes right over top of the saved "Old Base Pointer".

What if we wrote more than 8 bytes into the buf? We could overwrite the next stack entry, which is our saved return address. If we did that, we could return anywhere we wanted! Hmmm. We can't type the address at the prompt. We need something else. What if we made a program to output our desired return address, then redirected the output to our program?

Take a look:

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

        int i = 0;
        char buf[12];
        for (i = 0;i <= 8; i+=4)
        *(long *) &buf[i] = 0x80483f2;
        puts(buf);
}
```

This code creates a buffer and fills it with hex 0x80483f2 repeating. It outputs this string. Compile and run it. The output will be garbage, as these characters cannot be displayed properly. Why are we using 0x80483f2? By looking at the assembly code from main above, I know from experience that the second printf call (The one that will never run) starts there **(Note - this may be different on your machine - Look at the command at 80483f2. Find it in your code and substitute the number)**. If we are successful in our little experiment here, the input to our function oops should not only overwrite the ebp, it should overwrite the return value as well. When the code goes to return, it should return to our "This should never print" printf. Lets do it!

Here's my result of piping the output of one (trick) into buf:

```
-sh-2.05b$ (./trick;cat)|./buf
This should never print

Segmentation fault
```

It should never print, but it did. That is a buffer overflow, and we exploited it.

That's a lot for now. As an exercise, browse the source code you have available and on the net to see function usage. Look at some of the headers in /usr/include for header files.

Next:

Pointers and Structures

Appendix A, Important Sections of "Average" Assembly Code.

```
08048370 <main>:
 8048370:       55                      push   %ebp
 8048371:       89 e5                   mov    %esp,%ebp
 8048373:       83 ec 18                sub    $0x18,%esp
 8048376:       83 e4 f0                and    $0xfffffff0,%esp
 8048379:       b8 00 00 00 00          mov    $0x0,%eax
 804837e:       29 c4                   sub    %eax,%esp
 8048380:       b8 00 00 80 3f          mov    $0x3f800000,%eax
 8048385:       89 45 fc                mov    %eax,0xfffffffc(%ebp)
 8048388:       b8 00 00 40 40          mov    $0x40400000,%eax
 804838d:       89 45 f8                mov    %eax,0xfffffff8(%ebp)
 8048390:       83 ec 08                sub    $0x8,%esp
 8048393:       ff 75 f8                pushl  0xfffffff8(%ebp)
 8048396:       ff 75 fc                pushl  0xfffffffc(%ebp)
 8048399:       e8 36 00 00 00          call   80483d4 <average>
 804839e:       83 c4 10                add    $0x10,%esp
 80483a1:       d9 5d f4                fstps  0xfffffff4(%ebp)
 80483a4:       83 ec 04                sub    $0x4,%esp
 80483a7:       d9 45 f4                flds   0xfffffff4(%ebp)
 80483aa:       8d 64 24 f8             lea    0xfffffff8(%esp),%esp
 80483ae:       dd 1c 24                fstpl  (%esp)
 80483b1:       d9 45 f8                flds   0xfffffff8(%ebp)
 80483b4:       8d 64 24 f8             lea    0xfffffff8(%esp),%esp
 80483b8:       dd 1c 24                fstpl  (%esp)
 80483bb:       d9 45 fc                flds   0xfffffffc(%ebp)
 80483be:       8d 64 24 f8             lea    0xfffffff8(%esp),%esp
 80483c2:       dd 1c 24                fstpl  (%esp)
 80483c5:       68 cc 84 04 08          push   $0x80484cc
 80483ca:       e8 e1 fe ff ff          call   80482b0 <printf@plt>
 80483cf:       83 c4 20                add    $0x20,%esp
 80483d2:       c9                      leave
 80483d3:       c3                      ret

080483d4 <average>:
 80483d4:       55                      push   %ebp
 80483d5:       89 e5                   mov    %esp,%ebp
 80483d7:       83 ec 08                sub    $0x8,%esp
 80483da:       d9 45 08                flds   0x8(%ebp)
 80483dd:       d8 45 0c                fadds  0xc(%ebp)
 80483e0:       d9 05 f0 84 04 08       flds   0x80484f0
 80483e6:       de f9                   fdivrp %st,%st(1)
 80483e8:       d9 5d fc                fstps  0xfffffffc(%ebp)
 80483eb:       8b 45 fc                mov    0xfffffffc(%ebp),%eax
 80483ee:       89 45 f8                mov    %eax,0xfffffff8(%ebp)
 80483f1:       d9 45 f8                flds   0xfffffff8(%ebp)
 80483f4:       c9                      leave
 80483f5:       c3                      ret
 80483f6:       90                      nop
 80483f7:       90                      nop
```