Gathering of Gray Presents:

An Introduction to Programming for Hackers
Part II

By Lovepump, 2004

Visit:
www.gatheringofgray.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>
```

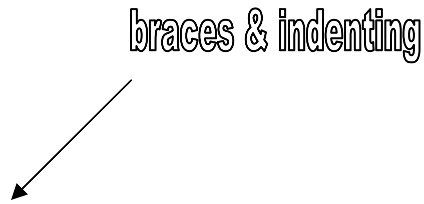
includes

```
#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");
```

comments

```
        // 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.