Gathering of Gray Presents:

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

By Lovepump, 2004

Visit:
www.gatheringofgray.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