

Gathering of Gray Presents:

An Introduction to Programming for Hackers
Part IV - Conditionals

By Lovepump, 2004

Visit:

www.gatheringofgray.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 & do-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:", and 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 in 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 $x = \text{something}$. 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.