Gathering of Gray Presents:

An Introduction to Programming for Hackers Part VIII - Basic File & Sockets Programming, Continued

By Lovepump, 2004

Visit:
www.gatheringofgray.com

Purpose:

In this Part we will complete the server portion of our file server program and learn how to compile together multiple files using ‘make’.

Where we Left Off:

In the last section, we left off at the erroneous function above (how convenient). We have so far created part of a simple file server program. It is able to create a socket, initialize a socket, bind, then listen to a specified port. The function ‘serv_start’ performs the lions share of this work, returning -1 on error.

Let’s Keep Movin’ On!

The next step in our server program is to accept an inbound connection and produce a session socket for this request. Refresh yourself on the ‘accept’ function discussed in Part VII.

To get our session socket, we wait patiently listening for an inbound connection to our bind socket. Let’s make a function to do that now:

```
int serv_conn_wait(char *buffer) {

        char    mess[] = "HELLO.";

        int size_client = sizeof(ft_client);
        sessionsocket = accept(hnd, (struct sockaddr*)&ft_client,
&size_client);

        if(sessionsocket < 0) return -1;

        write(sessionsocket, mess, strlen(mess));

        return sessionsocket;
}
```

Cool. We open up by creating a char array that conatins the string “HELLO.” This will be used as our ‘greeting’ to inbound clients. In the future we could also add a system and version banner here (i.e. - “f-transfer v1.0 HELLO.”).

The hello greeting is given to the new client and the session socket is returned to the calling code. The accept() function mirrors the socket() function discussed previously. The write() function sends our string to the new client.

Now that we have a client and have greeted our client, we need to listen to our client's request. Remember, we are the server and need to know what the client needs us to do. The following code is fairly straight forward:

```
int serv_recv_request(char *req) {

        int     r = 0;

        r = read(sessionsocket, req, MAXCOMMANDLENGTH);

        if(read <= 0) return 0;

        return r;
}
```

serv_recv_request takes a pointer to a char as argument and returns an integer. The read function gets bytes from our session socket. It stores the bytes in the array pointed to by req (the buffer passed to us) up to a maximum of MAXCOMMANDLENGTH in length. MAXCOMMANDLENGTH is a compiler constant (remember IF_ANET from the last part?). We will define it a bit later, but it will provide us a nice common area to change our maximum command length. Why do this? Two reasons:

1. If we define MAXCOMMANDLENGTH in one spot, we can change it in one spot, not have to search through our entire code to make changes to this parameter.
2. It gives us the maximum buffer size we need to 'malloc' to hold our network communications, and avoid buffer overflows. If we always use this constant when allocating memory to network transactions, it will be very unlikely that we will put ourselves in a BOF position.

We now have a bunch of functions that can read from our server socket. Let's try them out, shall we?

First Test (or How to use Make)!

To test our new 'kit', we need a logical method to put them together. Until now we have only created 'single file' programs. This is fine for very small projects, however putting larger projects all in one file is cumbersome. Let's break our program in to three files for now: ***ft.h, ft.c and ftserver.c & ftserver.h***. ft.c will contain our 'main' code. ftserver can contain all of our server code. This makes sense so far, but what are ft.h & ftserver.h? As we learned before, .h is the file extension for a header (.h) file. A header will contain our structure definitions, function prototypes and the like. Here is a demonstration:

```
/*      ftserver.h              */

/* Server Side */
int serv_conn_init();
int serv_start(struct conn *connection);
int serv_conn_wait(char *buffer);
int serv_recv_request(char *req);
```

Recall that to use a function, the complier must know about it first. We discussed the use of the function 'prototype' in Part V. Here, ftserver.h contains all the function prototypes needed to use ftserver.c:

```
/*
        The Simple File Transfer Project

        Server Module

        Part of the Programming for Hackers
        (c)2004, lovepump

*/


#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#include "ft.h"

static int              hnd, connections, sessionsocket;
static struct sockaddr_in       ft_server, ft_client;

int serv_conn_init() {

        hnd = socket(AF_INET, SOCK_STREAM, 0);
        if(hnd < 0) return -1;

        return hnd;
}

int serv_start(struct conn *connection) {

        int     r = 0;

        ft_server.sin_family = AF_INET;
        ft_server.sin_port = connection->port;
        ft_server.sin_addr.s_addr = htonl(INADDR_ANY);
        r = bind(hnd, (struct sockaddr*)&ft_server,
sizeof(ft_server));

        if(r < 0) return -1;

        r = listen(hnd, MAXCONNECT);

        return r;
}

int serv_conn_wait(char *buffer) {

        char    mess[] = "HELLO.";

        if(connections >= MAXCONNECT) return -1;

        int size_client = sizeof(ft_client);
        sessionsocket = accept(hnd, (struct sockaddr*)&ft_client,
&size_client);

        if(sessionsocket < 0) return -1;

        write(sessionsocket, mess, strlen(mess));

        connections++;
        return sessionsocket;
}

int serv_recv_request(char *req) {

        int     r = 0;

        r = read(sessionsocket, req, MAXCOMMANDLENGTH);

        if(read <= 0) return 0;

        return r;
}
```

There's all the functions we have discussed so far in one 'c' file. Let's compile it!

```
-sh-2.05b$ gcc ftserver.c -o ftserver
/usr/lib/gcc-lib/i386-redhat-
linux/3.3.3/../../../crt1.o(.text+0x18): In function `_start':
: undefined reference to `main'
collect2: ld returned 1 exit status
```

What? The key to this error message is in line 3 ": undefined reference to 'main' We don't have a 'main', so there is no place for the linker (part of the compile process) to tell the program where to start. Ok. Let's create a 'main' file called ft.c and compile them together.

```
/*      ft.h    */

#define FILEROOT "ft_share/"
#define MAXCONNECT 1
#define DEFAULTSERVERPORT 150
#define CONNCLIENT 1
#define CONNSERVER 2
#define MAXCOMMANDLENGTH 50

struct conn {
        int     type;
        int     hnd;
        unsigned long   ip;
        int             port;
};
```

There's the header, with some constants we are (or will soon be!) using. See MAXCOMMANDLENGTH. It will be replaced with the number 50 at compile time. This is the only place we have to change it, and everywhere it's used will be updated at the next compile. Cool. Here's and example ft.c:

```
#include <stdio.h>

#include "ft.h"
#include "ftserver.h"

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

        struct conn     connection;
        int             ret;
        char            buffer[MAXCOMMANDLENGTH];

        ret = serv_conn_init();

        if(ret == -1)
        {
                printf("Cannot create socket\n");
                exit(1);
        }

        connection.hnd = ret;

        connection.port = htons(DEFAULTSERVERPORT);

        ret = serv_start(&connection);
        ret = serv_conn_wait(buffer);
        ret = serv_recv_request(buffer);

        printf("%s\n", buffer);

}
```

Let's walk through it. We include ft.h and ftserver.h, but we use quotes instead of the angle brackets: < >. This tells the compiler that these header files are found in the current directory, not in the normal 'include path' (usually /usr/include). The next section declares our variables for use in main, including a 'connection' structure defined in ft.h and a char buffer using MAXCOMMANDLENGTH. A pointer to this buffer will be passed to the 'read' function to get our request from the client.

We next call serv_conn_init from ftserver.c. The compiler doesn't complain because it saw the prototype for this function in ftserver.h. Here we check the return value and see if the function was successful. If we get a -1, the function has failed. This is an example of making code robust. If we had ignored the return here, we would be trying to use a socket that does not exist.

The next function calls call our start, wait and request functions in order. There is no error checking done here. I leave it to the reader to modify the code to be more robust.

In the serv_recv_request function, we pass a pointer to our char buffer. We learned in Part V that by passing a pointer to a variable, we could allow the function to modify that variable. Here, the function will put the received message from the client in to our buffer. The next line, the printf, simply prints the message.

Great. Now we have our code, how do we compile it? There are two different files. To understand this better, we need to dive in to the compile process. Creating an executable with gcc is a three step process:

1. Preprocessor
2. Compiler
3. Assembler
4. Linker

The preprocessor performs macro and constant substitution. For example, our constant MAXCOMMANDLENGTH would be removed and a '50' substituted by the preprocessor. This all happens prior to the compiler. The preprocessor passes its output to the complier.

Compilation, in gcc, is the translation of C code to assembly. The compiler passes it's assembly code to the assembler.

The assembler makes assembly code into machine code, executable on our machine. The executable is given to the linker.

The linker performs all the connections between functions and creates the executable file in the machine native format (like ELF). It points the start of the executable at the start of 'main' in our code. (Our earlier error was given by the linker when it couldn't find 'main').

Does this mean that we can see these thing separately? Absolutely you can. To make you code into assembly, try this with ft.c:

```
-sh-2.05b$ gcc -S ft.c -o ft.s
-sh-2.05b$ cat ft.s
        .file   "ft.c"
        .section        .rodata
.LC0:
        .string "Cannot create socket\n"
.LC1:
        .string "%s\n"
        .text
.globl main
        .type   main, @function
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $104, %esp
        andl    $-16, %esp
        movl    $0, %eax
        subl    %eax, %esp
        call    serv_conn_init
        movl    %eax, -28(%ebp)
        cmpl    $-1, -28(%ebp)
        jne     .L2
        subl    $12, %esp
        pushl   $.LC0
        call    printf
        addl    $16, %esp
        subl    $12, %esp
        pushl   $1
        call    exit
```

Above is an excerpt of my output. Neat. We can also use gcc to produce 'object' code, which is run through the compiler and assembler, but not the linker. The object code can be compiled in to other modules at a later time. Let's try this approach with our ft.c program. To compile object code, we use the -c option:

```
-sh-2.05b$ gcc -c ftserver.c -o ftserver.o
```

Cool, it worked. It produced ftserver.o, the executable functions of our ftserver.c file. Remember, it's not linked, therefore it cannot run in it's present state. Now we can compile ft.c and use ftserver.o as a 'module'. If we try and compile ft.c without it, the linker will not find the functions that are compiled in ftserver.o, like this:

```
-sh-2.05b$ gcc ft.c -o ft
/tmp/ccmuPdBj.o(.text+0x11): In function `main':
: undefined reference to `serv_conn_init'
/tmp/ccmuPdBj.o(.text+0x59): In function `main':
: undefined reference to `serv_start'
/tmp/ccmuPdBj.o(.text+0x6b): In function `main':
: undefined reference to `serv_conn_wait'
/tmp/ccmuPdBj.o(.text+0x7d): In function `main':
: undefined reference to `serv_recv_request'
collect2: ld returned 1 exit status
```

As you can see the linker has complained that it can't find our functions. In your programming learning, watch the error messages that are produced. If you see the 'undefined reference' and 'ld returned' you can be sure that the linker cannot find your function (ld is the linker program).

Lets try and use our new object module:

```
-sh-2.05b$ gcc ft.c ftserver.o -o ft
-sh-2.05b$
```

We did it! We created a multifile executable. This process works, but will become very cumbersome and difficult with more files. Remember, for every change, we have to compile the object files, then compile and link the executable. There has to be a better way. There is.

Make

For linux users, make should be fairly familiar, but it's workings are probably unknown. For those who have installed from source code, the typical usage is:

```
$ make
$ make install
```

'make' is a macro, or batch, utility we can use to automate our compilation process. An in depth study on make is beyond the scope of this paper, but many resources are available on the net for those who wish to learn more.

To use make, we can do two things. invoking make for a single file simplifies things a little for us. If you remember from Part V, we created a program called 'average'. To compile average we typed:

```
-sh-2.05b$ gcc average.c -o average
```

To use make, we can type this:

```
-sh-2.05b$ make average
cc    -c -o average.o average.c
cc   average.o   -o average
```

The second two lines are the output of 'make'. It automatically compiles average and names the executable the same name as the source. Very nice feature.

The power of make we need for our file transfer program comes from what is called the Makefile. If a directory contains a file called 'Makefile' and make is run without a target file, the script found in Makefile will be executed. Lets see the Makefile for ft, then we will discuss it:

```
ft: ft.o ftserver.o
        gcc ft.o ftserver.o -o ft
        rm *.o

ft.o: ft.c ft.h
        gcc -c ft.c -o ft.o

ftserver.o: ftserver.c
        gcc -c ftserver.c -o ftserver.o
```

'make' works on dependencies. Each section of a make file specifies the dependencies of the files in your program. Here, the first line says 'ft is dependent on ft.o and ftserver.o'. If either ft.o or ftserver.o have changed since the last compile, make will run the indented commands below. The commands are:

```
gcc ft.o ftserver.o -o ft
rm *.o
```

This code creates our ft executable from the two object files, ft.o and ftserver.o, then deletes the object files (they serve no purpose after compile & link time).

The next section says: 'ft.o is dependent on ft.c and ft.h". If either of these files has changed, run the indented code:

```
gcc -c ft.c -o ft.o
```

As you can see, make only updates files as necessary. If we execute make with no changes to our code, we get the following message:

```
-sh-2.05b$ make
make: `ft' is up to date.
```

If all files are changed, here is what we get:

```
            -sh-2.05b$ make
            gcc -c ft.c -o ft.o
            gcc -c ftserver.c -o ftserver.o
            gcc ft.o ftserver.o -o ft
            rm *.o
```

As you can see, ft then ftserver were compiled to object files, then ft was linked from ft.o and ftserver.o to create ft, then both object files were deleted. Pretty cool with just one command.

Trial

We have just created our test ‘ft’ above. To try it out, we need to connect from a remote computer. Here’s what happened for me:

```
      -sh-2.05b$ su
      Password:
      [root@d36-36-299 ft]# ./ft
```

I don’t like to run as root, but it is necessary. Linux will not let me bind a port unless I run as root. Here the computer is sitting idle, waiting for a client.

On my other box, I use netcat to connect:

```
      -sh-2.05b$ netcat 34.36.36.299 150
      HELLO.
```

Awesome! As soon as I issued the connect, I got our “HELLO.” message. The server is idle now, waiting for our request. Let’s type one:

```
      -sh-2.05b$ netcat 24.36.29.209 150
      HELLO.Lick my lovepump
```

I sent the command (childish as it may seem) “Lick my Lovepump” to the server. Lo and behold:

```
      [root@d36-36-299 ft]# ./ft
      Lick my lovepump

      [root@d36-36-299 ft]#
```

Woot! It worked! Our server can respond to commands now. Our next task will be to take these commands and do something with them, but that’s for next time…