Joshua Dunne
Assignment 8

Abstract

Using the tools available through the GNU toolchain specifically, and other tools discussed in addition to, today we attempt to understand the function of a given application without relying on an examination of the source code.

Introduction

There were many tools discussed in class, listing those used are *file, strings, ghidra, netcat, strace, gdb, netstat, and ltrace*. Combining these we gain insight and understanding into the function of the program. As one would expect, we use a bash shell and vim to run and parameterize the application and edit text respectively. We rely heavily on our building understanding of system calls and disassembly to reverse engineer the function of the application given.

Summary of Results

We're going to work from the simplest things we can do, then work our way inwards. I'm going to rely heavily on ghidra here because I've never seen it before, and I'm interested.

File

The first and most simple thing I can think of to do is to run file on the application executable that gcc created, the output of that command looks like this

```
jdizzle@jdizzle-vbuntu:~/CS497/CS497/Assignment-8$ file application
application: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked, interp
reter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=0d6cd755931e7d576c001ef1315329c3a48f5904, for G
NU/Linux 3.2.0, with debug_info, not stripped
```

As we can see from the output produced, it is as we'd expect, as we ran gcc and included the *-g* flag we see that debug info was included, that it compiled as 64 bit in ELF, the executable format used by linux.

Strings

The next simplest thing I can think to do is to run strings on the application, this produces a very long list of results, but, upon examination, we find some interesting output that's worth including here.

```
jdizzle@jdizzle-vbuntu:~/CS497/CS497/Assignment-8$ strings -d 5 -f application
application: strcmp
application: fprintf
application: socket
application: atoi
application: close
application: strlen
application: bind
application: htonl
application: htons
application: stderr
application: listen
```

I gave parameters to strings so that we can ignore some of the more pedantic output and be able to focus, upon doing so, I see a few things that are of interest. Particularly, socket, bind, and listen. These three things gave us considerable insight. It suggests that the program uses a socket and listens to a socket.

Netstat

Netstat will provide us with some further information, we give it a few parameters so that our result makes sense, as it is running with a PID of 24719, we can see it listed here.

```
jdizzle@jdizzle-vbuntu:~/CS497/CS497/Assignment-8$ netstat -tulpn
(Not all processes could be identified, non-owned process info
 will not be shown, you would have to be root to see it all.)
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program name
tcp        0      0 127.0.0.1:631           0.0.0.0:*               LISTEN      -
tcp        0      0 0.0.0.0:2024            0.0.0.0:*               LISTEN      24719/./application
```

As we ran with *-l* flag we can tell that it is listening, too, as we see in the output it is listening on a TCP socket.

GDB

Another tool provided as part of the GNU toolchain is GDB, our debugger of choice, using this we can examine the program as it runs to get a deeper understanding. We give it the program, define breakpoints, then step through to see the process in action, step by step.

Firstly is our call to get PID, as the program prints this when starting

```
Catchpoint 1 (call to syscall getpid), getpid () at ../sysdeps/unix/syscall-template.S:93
93      ../sysdeps/unix/syscall-template.S: No such file or directory.
```

It then creates a socket and provides options to it

```
Catchpoint 1 (call to syscall socket), 0x00007ffff7d27d2b in socket () at ../sysdeps/unix/syscall-template.S:120
120     ../sysdeps/unix/syscall-template.S: No such file or directory.
```

```
Catchpoint 1 (call to syscall setsockopt), setsockopt_syscall (len=4, optval=0x7fffffffdb88, optname=2, level=1, fd=3) at ../sysdeps/unix/sysv/linux/setsockopt.c:29
29      ../sysdeps/unix/sysv/linux/setsockopt.c: No such file or directory.
```

We then bind to the socket we've created

```
Catchpoint 1 (call to syscall bind), 0x00007ffff7d276ab in bind () at ../sysdeps/unix/syscall-template.S:120
120     ../sysdeps/unix/syscall-template.S: No such file or directory.
```

And then listen to it

```
Catchpoint 1 (call to syscall listen), 0x00007ffff7d2781b in listen () at ../sysdeps/unix/syscall-template.S:120
120     ../sysdeps/unix/syscall-template.S: No such file or directory.
```

Accept is then called, and will wait until we get some data from the listening socket.

```
Catchpoint 1 (call to syscall accept), 0x00007ffff7d27617 in __libc_accept (fd=3, addr=..., len=0x7fffffffdb84) at ../sysdeps/unix/sysv/linux/accept.c:26
26      ../sysdeps/unix/sysv/linux/accept.c: No such file or directory.
(gdb) continue
Continuing.

Catchpoint 1 (returned from syscall accept), 0x00007ffff7d27617 in __libc_accept (fd=3, addr=..., len=0x7fffffffdb84) at ../sysdeps/unix/sysv/linux/accept.c:26
Show Applications
```

It then reads from that socket

```
Catchpoint 1 (call to syscall read), __GI___read_nocancel (fd=5, buf=0x555555559d30, nbytes=4096) at ../sysdeps/unix/sysv/linux/read_nocancel.c:26
```

And as the text that we sent was *“quit\n”*, the program terminates

```
Catchpoint 1 (call to syscall close), __GI___close_nocancel (fd=5) at ../sysdeps/unix/sysv/linux/close_nocancel.c:26
```

The examination done in this method gives us a clearer understanding, as often strace might give us things in a strange order, we can see what is happening as we step through the program.

Ghidra

The next simplest thing to do is simplicity itself, we run the program, upon doing so we’re presented with a proper usage line to work from.

```
jdizzle@jdizzle-vbuntu:~/CS497/CS497/Assignment-8$ ./application &
[1] 22539
jdizzle@jdizzle-vbuntu:~/CS497/CS497/Assignment-8$ usage: ./application <port>
./application 2024
y pid is 22540
^Z[1]   Exit 1                  ./application
```

We’re told that the program expects to be specified a port, knowing this, we can also better understand the string atoi found above, the program expects to be given a port number, so we do so. In order that the command can continue to run in the background while we send it data, we run it in the background, then use our old friend netcat to send it some input to work with.

```
jdizzle@jdizzle-vbuntu:~/CS497/CS497/Assignment-8$ netcat 127.0.0.1 2024 << ~
this is some traffick
~
server established connection with localhost (127.0.0.1)
server received 22 bytes: this is some traffick
this is some traffick
```

And now we’re getting somewhere, the string that we provided netcat was received by the program and outputted on to the command line. We can further evidence our building understanding by comparing the decompilation through ghidra. We see that connfd is read from here, and compared to a string *“quit\n”*

```
n = read(connfd,buf,0x400);
if (n < 1) {
  error("ERROR reading from socket");
  goto LAB_00101757;
}
printf("server received %d bytes: %s",n,buf);
iVar3 = strcmp(buf,"quit\n");
if (iVar3 == 0) goto LAB_00101757;
```

To, by backtracking through the decompilation a little, we can see that connfd was modified in the following way.

```
connfd = accept(listenfd,&clientaddr,&clientlen);
```

As we suspected, the program attempts to read from a network socket on the port we specified as a command line argument, we can see here that it compares it to string, so, let's netcat it that string and see if the behavior it exhibits is in line with our suspicions.

```
@jdizzle-vbuntu:~/CS497/CS497/Assignment-8$ echo "quit" > aFile
jdizzle@jdizzle-vbuntu:~/CS497/CS497/Assignment-8$ netcat 127.0.0.1 2024 < aFile
server established connection with localhost (127.0.0.1)
server received 5 bytes: quit
[1]+  Done                    ./application 2024
```

Alas, as expected, the application we're examining quits when we pipe it the string *"quit\n"*.

To summarize our findings so far, the application expects an integer parameter that it takes as a port number. It creates a socket on that port and listens for input. If that input is anything other than *"quit\n"* it prints it and continues to listen. If the input it receives on the port it is listening to is *"quit\n"* then it stops listening and exits.

```
 printf("server established connection with %s (%s)\n",hostp->h_name,hostaddrp);
 do {
   puVar6 = (undefined8 *)buf;
   for (lVar5 = 0x80; lVar5 != 0; lVar5 = lVar5 + -1) {
     *puVar6 = 0;
     puVar6 = puVar6 + (ulong)bVar7 * -2 + 1;
   }
   n = read(connfd,buf,0x400);
   if (n < 1) {
     error("ERROR reading from socket");
     goto LAB_00101757;
   }
   printf("server received %d bytes: %s",n,buf);
   iVar3 = strcmp(buf,"quit\n");
   if (iVar3 == 0) goto LAB_00101757;
   uVar4 = strlen(buf);
   n = write(connfd,buf,uVar4);
 } while (-1 < n);
 error("ERROR writing to socket");
AB_00101757:
 close(connfd);
 iVar3 = 0;
 if (lStack_10 != *(long *)(in_FS_OFFSET + 0x28)) {
   iVar3 = __stack_chk_fail();
 }
```

To give credence to what we've come up with so far, we can consult the disassembly again, as it is exhaustive, we can see that there is no further functionality to be examined, it does what we've come to expect it to do, and nothing more.

Let's now go ahead and use the other tools we were introduced to in the class discussion.

Strace

```
jdizzle@jdizzle-vbuntu:~/CS497/CS497/Assignment-8$ strace ./application 2024 2> >(sed $'s,.*,\e[
31m&\e[m,'>&2)
```

I've gone ahead and run strace with a little extra sugar. All it does is highlight stderr, the output from strace, and print that in red.

We read past the various tasks that linux undertakes upon launching the program and read right from the more important section.

```
write(1, "y pid is 23488\n", 15)        = 15
socket(AF_INET, SOCK_STREAM, IPPROTO_IP) = 3
setsockopt(3, SOL_SOCKET, SO_REUSEADDR, [1], 4) = 0
bind(3, {sa_family=AF_INET, sin_port=htons(2024), sin_addr=inet_addr("0.0.0.0")}, 16) = 0
listen(3, 5)                            = 0
```

These 5 system calls are of the utmost importance, beyond writing the PID that was requested earlier, we also set up a socket, bind to that socket, and listen to that socket. The program waits

here as we'd expect, but, we can use another shell to send the program input and examine further system calls

```
server established connection with localhost (127.0.0.1)
newfstatat(AT_FDCWD, "/etc/nsswitch.conf", {st_mode=S_IFREG|0644, st_size=542, ...}, 0) = 0
server received 10 bytes: some text
newfstatat(AT_FDCWD, "/", {st_mode=S_IFDIR|0755, st_size=4096, ...}, 0) = 0
openat(AT_FDCWD, "/etc/nsswitch.conf", O_RDONLY|O_CLOEXEC) = 5
newfstatat(5, "", {st_mode=S_IFREG|0644, st_size=542, ...}, AT_EMPTY_PATH) = 0
read(5, "# /etc/nsswitch.conf\n#\n# Example"..., 4096) = 542
read(5, "", 4096)                       = 0
newfstatat(5, "", {st_mode=S_IFREG|0644, st_size=542, ...}, AT_EMPTY_PATH) = 0
close(5)                                = 0
openat(AT_FDCWD, "/etc/hosts", O_RDONLY|O_CLOEXEC) = 5
newfstatat(5, "", {st_mode=S_IFREG|0644, st_size=229, ...}, AT_EMPTY_PATH) = 0
lseek(5, 0, SEEK_SET)                   = 0
read(5, "127.0.0.1\tlocalhost\n127.0.1.1\tjd"..., 4096) = 229
close(5)                                = 0
write(1, "server established connection wi"..., 57) = 57
read(4, "some text\n", 1024)            = 10
write(1, "server received 10 bytes: some t"..., 36) = 36
write(4, "some text\n", 10)             = 10
```

Before the listen call the program was waiting at, we send some input to the port we requested the program listen on. We can see then that the input we specified was received by the application we're examining. It reads that input from the socket, and prints that to the command line.

Ltrace

With a little troubleshooting, I'm going to change to another version of ltrace, latrace.

```
jdizzle@jdizzle-vbuntu:~/CS497/CS497/Assignment-8$ ltrace ./application 2024
y pid is 23787
server established connection with localhost (127.0.0.1)
server received 10 bytes: some text
```

Having made this change, we see perhaps the clearest and most concise demonstration of the programs function we've found so far. As the program is dependent on system calls, and uses c wrapper libraries to simplify the calling of those calls, we get a very clear explanation.

```
23923      atoi [/lib/x86_64-linux-gnu/libc.so.6]
23923      getpid [/lib/x86_64-linux-gnu/libc.so.6]
23923      printf [/lib/x86_64-linux-gnu/libc.so.6]
23923      socket [/lib/x86_64-linux-gnu/libc.so.6]
23923      setsockopt [/lib/x86_64-linux-gnu/libc.so.6]
23923      htonl [/lib/x86_64-linux-gnu/libc.so.6]
23923      htons [/lib/x86_64-linux-gnu/libc.so.6]
23923      bind [/lib/x86_64-linux-gnu/libc.so.6]
23923      listen [/lib/x86_64-linux-gnu/libc.so.6]
23923      accept [/lib/x86_64-linux-gnu/libc.so.6]
```

Here libc is the standard c library, and we can see the calls very concisely. We call atoi to convert the parameter from a string to an integer to use as the port to listen on. We get the PID and print it. We open a socket, provide parameters, bind to it, and listen for input.

```
server established connection with localhost (127.0.0.1)
server received 10 bytes: some text
23923     gethostbyaddr [/lib/x86_64-linux-gnu/libc.so.6]
23923       calloc [/lib/x86_64-linux-gnu/libc.so.6]
23923     inet_ntoa [/lib/x86_64-linux-gnu/libc.so.6]
23923     printf [/lib/x86_64-linux-gnu/libc.so.6]
23923     read [/lib/x86_64-linux-gnu/libc.so.6]
23923     printf [/lib/x86_64-linux-gnu/libc.so.6]
23923     strcmp [/lib/x86_64-linux-gnu/libc.so.6]
23923     strlen [/lib/x86_64-linux-gnu/libc.so.6]
23923     write [/lib/x86_64-linux-gnu/libc.so.6]
23923     read [/lib/x86_64-linux-gnu/libc.so.6]
```

The program then reads from that socket, prints the output, compares it to *“quit\n”* and then returns to continue listening. If we provide the string it expects to quit, we then see.

```
server established connection with localhost (127.0.0.1)
server received 5 bytes: quit
23940     gethostbyaddr [/lib/x86_64-linux-gnu/libc.so.6]
23940       calloc [/lib/x86_64-linux-gnu/libc.so.6]
23940     inet_ntoa [/lib/x86_64-linux-gnu/libc.so.6]
23940     printf [/lib/x86_64-linux-gnu/libc.so.6]
23940     read [/lib/x86_64-linux-gnu/libc.so.6]
23940     printf [/lib/x86_64-linux-gnu/libc.so.6]
23940     strcmp [/lib/x86_64-linux-gnu/libc.so.6]
23940     close [/lib/x86_64-linux-gnu/libc.so.6]
```

As we’d expect, having sent *“quit\n”* to the port the program is listening to, the comparison returns true, and the program exits cleanly.

Conclusion

Given a program that did something we were not sure of, using various tools and a little bit of thinking, we’ve put together what this program does. With our gained understanding we could now recreate the application, or distribute our understanding to others if necessary. Throughout the examination we combined the function of various tools and combined what they told us to gain further insight.

Additional methods

There are other things we could have used. The tools we used here are quite dated, and as time has progressed, tools like ghidra have since been supplanted by even more powerful and intricate reverse engineering tools. The use of these tools would have eased the more cumbersome parts of this process, and would have allowed us to view much of it simultaneously.

There is another thing to think about, too. Much of what was happening inside of the environment our program ran inside of went unexamined. Were we to integrate other tools into the kernel, they would allow us to intercept much of what the program was trying to do. There exist kernel modules that allow us to do exactly this.