Exercise 4, Buffer overflow and SQL Injection

T-110.4200/6 course staff

deadline Friday 16.10.2009 23:55

1 Buffer overflow

Aim of the assignment: Student understands the anatomy of a simple buffer overflow vulnerability. Student understands why it's bad, bad, bad to read a theoretically unlimited amount of user input into a prelimited memory location.

To pass this excercise you must complete the following script and answer questions at Moodle.

The archive buffer_overflow.tar.gz contains two files to be used in this exercise, buffer_overflow and buffer_overflow_actual and respective .c-files. The first file comes with a complete source code and the other one comes with a source code that leaves some things for you to find out.

Both the files are executables that can be run unix platforms. *Note: There might be some problems with running the files in TKK computing centre computers. So do this exercise in Niksula or use your own computer.*

Let's start with the first file (buffer_overflow.c). If you read the code then you can see,that 2 character tables (a.k.a. strings) are introduced at the beginning. Next, the memory addressess are printed out. As the memory is reserved in consecutive commands, consecutive memory areas will most probably be reserved.

Run the program and examine the memory addresses to determine which one of the strings is first in memory. Now, if you write a string that is longer than the 8 characters of the first table, the extra will "overflow" to the next memory space. If you try to write outside the memory area reserved to the program, unexpected things may happen. Play around a little bit to experiment more until you feel what happens. Try to compile the software from source with

```
gcc -o buffer_overflow buffer_overflow.c
```

and discover that the compiler warns you about unsafe behaviour. Apparently the makers of gcc aren't entirely stupid.

When you feel like hack-hack-hacking a little, move on to see what the file buffer_overflow_actual does. It asks for a password. Actually, it recreates a part of an old Unix security flaw: when the user gives his username, the password hash is read into memory from /etc/passwd. Unfortunately the memory area for the hash is immedately behind the one into which the user-input password is read next. Now, with a correctly formatted string that contains a password and its hash, you can overflow the stored value and the resulting comparison with a passwd and its hash is automatically true.

In buffer_overflow_actual hash is not used, so the password is stored as clear text into that memory area. Use what you learned from playing with the demo program (buffer_overflow) and enter correctly formed string to buffer_overflow_actual.

If you bypass the password correctly, the software will print for you a secret string that you must return to Moodle. We acknowledge the fact that there are other ways to figure out the secret from the binary, but we ask that you try to complete this illustrative assignment instead of figuring out alternative ways to accomplish the same.

2 SQL injection

To pass this excercise you must complete the following script and answer questions at Moodle.

From somewhere inside the TKK network, connect with *ssh(1)* to the server winnie.cs.hut.fi. See the previous excercise for details on usernames and passwords. The machine winnie is supposed to serve as an intermediate X server. On it you can start Firefox (use -X when connecting with ssh).

Go to the website https://honeypot.cs.hut.fi/sql_inj.php. There you will find a small php script that uses a database and has a clear SQL injection vulnerability. There's a table in the database that has been generated with:

```
create table student(
        id INTEGER NOT NULL AUTO_INCREMENT,
        student_id varchar(7) NOT NULL,
        primary key(id)
);
```

Your task is to INSERT a row containing your student number to the table and then *answer the related questions in Moodle.*

Hint: Since table contains AUTO_INCREMENT field that is the key of the table, it is easier to insert value only to that student_id column and let the database handle the key.

Hint: Probably some of you won't follow the abovementioned hint and use a self-picked random id. This may cause collisions in the ID fields. If you receive errors about duplicate keys violating unique constraints, this is the reason. Normally this would be bad worksmanship on the part of the attacker. However, it can be fixed by running the following command through the SQL injection hole:

```
SELECT setval('student_id_seq', (SELECT max(id) FROM student));
```

Which sets the automatic id counter to the highest id value in the table and removes the chance of collision.

For reference you can view the php-code below.

```
<html>
<head>
<title></title>
</head>

<body>
<form name="query_parameters" action="sql_inj.php" method="post">
Select student_from student  <input type="text" name="parameter" />
<input type="submit" value="Submit">
</form>



<?php
$query ="select student_id from
student".stripcslashes($_POST['parameter']);
$connection = pg_connect("dbname=x user=x
password=x");

$results = pg_query($query);

if (!$results) {
$message  = 'Invalid query: ' . pg_last_error() . "\n";
$message .= 'Whole query: ' . $query;
die($message);
```

```
}


echo "<table>";
while ($row = pg_fetch_row($results)){
echo "<tr>";

for ($i = 0; $i < sizeof(row);$i++){
echo "<td>" . $row[$i] . "</td>";
}

echo "</tr>";
}
echo "<table>";

pg_close($connection);
?>
</body>
</html>
```