A BRIEF INTRODUCTION TO SECURE SCRIPTING

Contents

1. Introduction

These days there are a lot of people that release scripts and programs on to the Internet which people will then download and use without a second thought that they may contain the simplest of security holes which can allow malicious people to attack their servers.

In this paper i intend to address some of the common mistakes that are made by the programmers releasing these programs, and how it should be written properly. The paper will mainly focus on Common Gateway Interface (CGI) scripts with examples written in perl (in my opinion the most common language used to write CGI scripts). The paper is, however, relevant to all languages, not just perl.

The main way in which CGI scripts are exploited is by a user presenting it with values that the script does not expect, and therefore does not know how to handle properly.

2. Input taint checking

It is fair to say that most people who start to write perl scripts will rip out a general piece of code from another script they have seen to split the input from a submitted HTML form. These routines, more often than not, just take an input (something like *name=dan&email=dang@dcode.net*), split it firstly by the ampersand in to an array, and then cycle through this array

splitting he values by the equals sign and dumping the keys and values in to a hash. Some processing is done along the way, i.e. turning the codes back in to real characters, but no real taint checking is done.

As a general rule, i refuse to use these routines, i see it as much safer to extract ONLY what you need from all the data inputted, and ONLY in the form it should have been submitted in. For example, if i have a form field called id, which is meant to contain 1 or 2 digits, i would extract it in the following way (if the form method was get):

```
$ENV{'QUERY_STRING'} =~ m/id=([0-9]{1,2})/;
```

This looks for a match in the query string sent to the server by your web browser of *id=<number of 1 or 2 digits>*. When it finds it the value of that number is placed in the read-only variable $1, from which you can use it. A simple way to check if this search has been successful is to place it within and if or unless.

```
unless($ENV{'QUERY_STRING'} =~ m/id=([0-9]{1,2})/) {
     die "Invalid or id supplied";
}
```

Now we will take a look at what this approach affects.

Take a look at the following snippet of code:

```
my $query = "DELETE FROM tablename WHERE id=" . $id;
```

As you can see this will delete all entries from the table named *tablename* that have an id of the value in *$id*. If we had used a generic "input grabber" then someone could have inputted the following:

```
1; DROP TABLE tablename
```

As the semi-colon tells our SQL server to execute the statement preceding it, the server will then execute the query following the semi-colon as well. In this case, as there is no taint checking, the following would have been executed on the SQL server:

```
DELETE FROM tablename WHERE id=1; DROP TABLE tablename
```

This, as you can see, would result in the complete loss of ALL data in the table *tablename*.

If we had used our restrictive "input grabber" as written above, any sign of the input not being a 1 or 2 digit number would result in the program dying a horrible messy death, before it starts executing SQL statements.

Obviously this means that neither the first nor second command would never get executed, and no damage would be done.

This kind of thing does not only affect SQL, as you will see soon.

3. String Manipulation

There are a number of considerations to make when manipulating and creating new strings. Firstly, a lot of people do not take the effort to declare all variables when they are first used. This could allow a hacker to read variables and maybe utilise them in attacks using other parts of the program. All variables should be declared in the appropriate place. This means they should only be defined within the block of code that they will be used in so that they can not be used in any other part of the program.

When joining strings and manipulating existing strings there are a number of things to remember. As all your input should have been checked (as described in the taint checking section) this should not cause major problems. However, string manipulation should be done in the correct way. This means utilising a taint check on the string after it has been composed to check that it is in the right format for its use, so that if a bad input should somehow get through there is another layer of protection before the string can be utilised in a malicious way.

4. File Manipulation

Files, in my experience, are hardly ever treated with enough respect by most programmers. Many people neglect to open file handles with the proper read/write options, which can lead to someone opening a file in the incorrect mode and then destroying your data. There is an example of this below:

Suppose i want to open a file to read it. Most people would use the following bit of code:

```
open(FILE, "/path/to/file");
```

This in itself is not harmful, but imagine if the code was written as such (where *$file* is from user-input):

```
open(FILE,$file);
```

This would be unsafe. Imagine someone came along and entered the following input:

```
>/etc/passwd
```

The program (permissions permitting) would then be able to wipe that file clean as they have successfully opened it as a new file, not in append mode. For this reason all new files should be opened in the proper mode. Further more taint checking should be done on the user input to stop them from using such characters.

Another, often missed, feature used with files is file locking. File locking can stop files being overwritten by your programs while it is still being used by another instance, and therefore reducing the chances of file corruption and race conditions.

5. System Calls

Most languages will have the functionality to make calls to local system commands. There will usually be a number of functions that make this possible, for example in perl we have *system()* and *exec()*, and the more obscure *$result = `command`*. These commands can be dangerous if not executed in the proper way.

There are fundamental differences in the way that these commands execute the external programs. If you pass in shell meta characters to system it will execute the input via a UNIX shell, typically /bin/sh. *exec()* however will execute only through the program itself, bypassing the shell completely.

You must also pass arguments to these calls properly, for the reasons shown

below. If you passed user input to the system call they could easily execute other programs. For example:

```
system("/path/to.program " . $userinput);
```

If someone were to set *$userinput* to *"user && /path/to/program2 arguments && /path/to/program3"* then each program would be executed in turn. When passing arguments to system you should do it in the following way:

```
system("/path/to/program", $argument1, $argument2);
```

system() will then treat each argument as just an argument, and not execute anything in these strings that shouldn't be there.

exec() should also be called in the same way.

6. Variable Declaration

Some languages, such as perl, do not require you to declare all your variables to be able to use them. This can lead to problems as if not declared, these variables are treated as global and could be overwritten by user input even if NOT used in the same subroutine. A way to stop your program executing like this is to use the following:

```
#!/usr/bin/perl -w
use strict;
```

This will stop your program from executing if any variables are not properly declared.

Furthermore, all variables should be declared in the proper places. If they need to be used in more than one subroutine, pass them as arguments. This way it is impossible to over-write them in other subroutines.