

TAINTING

By Nathan Torkington

Introduction

When you think about it, CGI is just a way for anybody on the Internet to run a program on your computer. This makes it the world's most popular security hole. As programmers, our duty is to make sure we don't let the countless hordes of Bad Guys into our pristine Good Guy systems. The same goes for setuid programs, which run with different privileges than the person who ran them.

This CGI program, for instance, would be bad:

```
#!/usr/bin/perl -w
# cgi-bad - bad example of a bad cgi script
...
$file = param("FILE") or die "Must fill out the FILE fieldn";
unlink("/usr/local/public/data/$file") or die "Can't delete $file : $!n";
```

What this does is take a filename as input from a form, and delete the file of that name from /usr/local/public/data/. Wrong. What this does is let any user delete any file on the system that the web server usercode can delete. Watch:

```
% setuid-bad ../../etc/apache/var/userdb
```

Oops! There goes a user database!

What we should have done is check the program's argument to make sure that it was, in fact, a normal ordinary plain boring filename. The general problem is taking data that originated outside your program and using them in system calls like unlink(), open(), and system(). You don't want data originating outside your program to affect the outside world.

Perl has an option you can enable that forces you to check your arguments, your environment, your input, and anything else that might have been poisoned by a malicious user of your program. This option is called "tainting".

If we'd had tainting enabled, the Perl interpreter would have stopped when we tried to unlink(), saying that \$file contained tainted data. The filename was tainted because it came from an untrustworthy source: the user of your program.

ENABLING TAINT CHECKS

To enable taint checking, give Perl the -T option:

```
#!/usr/bin/perl -wT
```

If we ran our previous program with -T, we'd see:

Insecure dependency in unlink while running with -T switch at setuid-bad line 5.

TAINTING

By Nathan Torkington

Perl has kept track of the value in `$file` and knows that it originated outside your program (it is "tainted"). `unlink()` is considered an unsafe operation, because it affects something in the outside world: a file. Attempts to use untrustworthy (tainted) data in an unsafe operation are dangerous because, as we've seen, the data might have something surprising or devious in it. These attempts are caught at runtime by Perl's taint checks, and the program dies.

TAINTED DATA

Tainted data originates from your environment (the `%ENV` hash, from your arguments (`@ARGV`), from files and directories you read, from programs you run, and the results of some system calls (the GECOS field of the password database fetched with `getpw*`). If you do anything with a tainted value (add, concatenate, interpolate), the resulting value is also tainted. It's like a smudge on a value that spreads wherever that value goes.

There are only three ways to get an "untainted" value: it is hard-coded in your program, it comes from a safe function (e.g., `localtime`), or you use a regular expression to pull out part of a tainted string that came from an unsafe function:

```
$a = 4;# untainted
```

```
$file = $ARGV[0]; # tainted  
$file =~ m{^[^/]+$} or die "$file is not a good filename.n";  
$untainted = $1; # untainted
```

By enclosing part of the regular expression in parentheses, the `$1`, `$2`, ... variables are created. These are untainted. By checking the value with a regular expression, you have a chance to ensure it's what you were expecting. If the match fails, you should die. If it succeeds, the `$1` ... variables hold untainted data for you to use.

BAD ACTIONS

Most things you want your Perl program to do will generate errors if the data you use are tainted. Running programs, opening files for writing, and deleting files are all prohibited if the filename or program name is tainted. This section will show you how and what to untaint in these situations.

Consider:

```
system("ls *.h");
```

Perl sees the `*` in your string and decides to call the shell thus:

```
sh -c "ls *.h"
```

TAINTING

By Nathan Torkington

But someone could have run your program with a faked out PATH environment variable (perhaps not in the CGI world, but it's very possible in the setuid world), resulting in the wrong sh or wrong ls being called. So you'd have to untaint your PATH variable, and any others that the SHELL might use to modify its behaviour.

In general, you should take three steps to run other programs:

1. clean your environment to make sure that the program that gets run is the real program.
2. lock out the shell.
3. untaint the arguments to the program.

Cleaning your environment is as simple as:

```
delete @ENV{"IFS", "CDPATH", "ENV", "BASH_ENV"};  
$ENV{PATH} = "/bin:/usr/bin";
```

The first line deletes environment variables that might cause you problems, and the second gives you a PATH that is guaranteed safe. You can add other directories to the path, but always make sure they're hard-coded into your program like this.

Removing the shell is trickier. Perl has its own rules for when it involves the shell in open(), system(), backticks, and exec() calls, and they're not always easy to follow. The best rule is: avoid backticks and pipe open() calls. Instead use system() and exec() and pass them a list.

Most people are used to seeing system work like this:

```
system("someprogram arg1 arg2 arg3");
```

They don't know that you can say:

```
system("someprogram", "arg1", "arg2", "arg3");
```

This tells Perl exactly which arguments are which, and Perl won't call the shell. exec() also has the list-taking no-shell-calling property. There are no guaranteed ways to use piped open() and the backticks without involving the shell.

If you want a piped open or the backticks, you'll have to reimplement them from scratch with a forking open():

```
$pid = open(COMMAND, "-|");  
die "Couldn't fork: $!" unless defined $pid;  
if ($pid) {  
    @lines = <COMMAND>;  
    close(COMMAND);
```

TAINTING

By Nathan Torkington

```
} else {  
    exec("some", "program", "with", "args") or die "execing: $!";  
}
```

This type of `open()` does not run another program, it merely splits your program in two behind the scenes. Each copy of your program follows a different branch of the `if()`, with the new copy replacing itself with the program to be run, and the original program reading the data from the filehandle.

In general it's also a good idea to give the full path to the program you're running, even after securing your `PATH`. This safeguards against accidentally calling `/usr/bin/boom` instead of `/home/user/bin/boom` because `/usr/bin` was in your `PATH` before `/home/usr/bin/boom`.

FILENAMES

Working with filenames, either with `unlink()` or `<*.h>` or just plain `open()`, is tricky. Filenames you read from a directory are tainted. You can open a tainted filename for reading, but you can't open it for writing. Data read from a file, whether the name was tainted or not, is tainted. You can't get a list of files with `<*.h>` because this uses the shell.

To check whether a filename is good, you'll have to write a regular expression that matches valid filenames. In some cases vetting your data can be as simple as:

```
$file = $ARGV[0];  
($file =~ m{^[^/]+$} && $file ne "." && $file ne "..") or die "Bad filename $file";  
$file = $1;
```

We test the file against a regular expression that permits anything that doesn't contain a slash. This rules out subdirectories. We then eliminate `."` (the current directory) and `.."` (the parent directory of the current directory). If all these tests succeed, `$1` holds the filename we can use.

To get a list of files that match some pattern, you can either install one of the globbing modules from CPAN (`File::KGlob` and `File::BSD` are two useful ones) or use the directory reading operations and regular expressions:

```
opendir(DH, "/path/to/directory") or die "opening directory: $!n";  
while (defined ($thing = readdir(DH))) {  
    next unless /^(.*.h)$/;  
    push(@files, $1);  
}  
closedir(DH);  
# @files is the list of untainted *.h filenames
```

TAINTING

By Nathan Torkington

CHECKING FOR TAINTEDNESS

If you need to check for taintedness, there's a little trick you can use:

```
sub is_tainted {  
  return ! eval {  
    join(" @_), kill 0;  
    1;  
  };  
}
```

You need to know two things to understand this: kill 0 doesn't do anything but return "true"; and an expression is tainted if any part of it uses tainted data. So if is_tainted is called with tainted data, the use of @_ with a kill (even though the tainted data aren't arguments to kill) is enough to make Perl die.

SCURRILOUS UNTAINTING

In some situations, but not very many, you will want to blindly untaint your data. That is, you want to untaint a variable's value without caring what it is. This is deliberately and consciously creating a security hole. Tainting is there for a reason. Are you absolutely sure this is what you need to do? If there's another way to solve your problem, then you should use the other way. Some day blindly untainting data may come back to haunt you.

If you MUST untaint everything and not care, you can use this:

```
$var =~ /(.*)/s; # STUPID  
$var = $1;
```

The /s modifier to the regular expression match ensures that the period matches any newlines that might be in the string. We match everything in the string with the .* and use \$1 as an untainted copy of that data.

Like the comment says, this is almost always stupid. Many authors and books don't show you how to do this because it bypasses the security features of tainting. All the tainting in the world does you no good if you blindly untaint.

Besides, you'll be shot by your coworkers at the next code review if you do this.

SUMMARY

-T enables tainting. Data originating outside your program is tainted, and can't be used to affect the outside world. Untaint with regular expressions and the \$1, \$2, ... variables. To run other programs, fix your path, don't use the shell, and untaint the arguments.