Software Assurance Tips

A product of the Software Assurance Tips Team[2]

Kevin Keen

Monday 4th October, 2021

1 Perls of Wisdom

Updated Monday 4th October, 2021

We see far less Perl code than some other languages. Nevertheless, as with any language, there are security concerns that are particular to Perl.

Consider the following short Perl script in Listing 1.

```
my $in_filename;
my $out_filename;
if(scalar @ARGV < 1) {
        print "Usage: $0 <input_file>";
        exit;
}
else {
        $in_filename = $ARGV[0];
        $out_filename = "that_$in_filename";
}
open($in_file, "$in_filename");
open($out_file, ">$out_filename");
while(<$in_file>) {
        $line = $_;
        print "$line";
        if ($line =~ /This is a/) {
                $line =~ s/This is a/That is a/;
                print "First Match!\n";
        }
        elsif($line =~ /This is only/) {
                $line =~ s/This is only/That is only/;
                print "Second Match!\n";
        }
        print $out_file $line;
}
```

Listing 1: Sample Perl Script

This script will read in the input file, perform some substitutions (changing "This" to "That") and write out the results to another file. With Perl's deeply ingrained regex facilities, such string and file processing is very typical of a Perl script. The input file name is passed in as a command line argument. The output filename is derived by prepending `that_`. As with a lot of Perl scripts, this one is written to be run against input files with very specific contents. In this case, it expects the input file to contain the input displayed in Listing 2. The output of the intended use of this script is shown in Listing 3.

```
This is a test
This is only a test
```

Listing 2: Sample Input

```
$ ls
textfile  this_to_that.pl

$ cat textfile
This is a test
This is only a test

$ perl this_to_that.pl textfile
This is a test
```

```
First Match!
This is only a test
Second Match!

$ cat that_textfile
That is a test
That is only a test
```

Listing 3: Sample Execution

The biggest security concern here comes from the use of the two argument form of open, which in some cases, can actually result in shell execution! Contained in the two open statements shown in the above source, you will notice that one filename is prefixed with ">", but the first open has no prefix. The ">" character on the second open tells Perl to open the file for output. The first open is implicitly open for input. This doesn't look bad just looking at the source, but Perl has built-in magic that makes this a security disaster. In this kind of case, if the filename starts with, or ends with a pipe symbol (|), Perl interprets that as requesting execution of a shell command. Note that the location of the pipe symbol also has meaning to Perl. A pipe symbol before the command indicates a shell command that should be written to. A pipe symbol after the command indicates a shell command that should be read from. This means that abusing this form of open is as easy as providing a malicious filename.

```
$ ls
textfile  this_to_that.pl

$ perl this_to_that.pl "touch in_a_shell_command|"

$ ls
 in_a_shell_command   'that_touch in_a_shell_command|'
 textfile              this_to_that.pl
```

Listing 4: Malicious Execution

Listing 4 shows the result of running the script with a malicious filename resulting in the file "im_a_shell_cmd" being created. The other new file, "that_touch im_a_shell_cmd|", is created by the script as part of its normal output. Of course in this case we already had shell access, but consider the case where the filename is coming from an external attack such a socket or web interface.

This security disaster can be partially mitigated by explicitly specifying that the file should be open for input. If a leading "<" is provided (similar to the second open statement), Perl will assume the filename is just a filename even if a pipe symbol appears. If all of that were not enough, this version of open can cause problems when we can't even see it because a number of other constructs use the two argument version of open behind the scenes. The examples in Listing 5 are all implicitly using the two argument form of open.[3] Such constructs should be avoided, and the 3 argument form of open used instead of the two argument version.[1]

```
while(<ARGV>)
while(<>)
perl -n 'print ":: $_\n";' *
perl -p '$_ = ":: $_\n";' *
```

Listing 5: Examples

1.1 Bonus Tip

Using "<" or ">" on the two argument form of open is not enough to solve the woes of using it because Perl has even more magic up its sleeve. A filename of dash is interpreted to mean STDIN. Although explicitly providing the input mode character "<" avoids the command injection, we still have a very

viable denial of service. Often, a string is concatenated before being passed to open. One question that arises is “when will dash be considered to be STDIN”. Both with and without the explicit “<” symbol, both –, and “-“, as a command line argument is interpreted by Perl to mean STDIN. But what of the case where dash is concatenated? It seems backwards, but when the explicit “<” character is **not** provided, “- something” is not interpreted to mean STDIN. However, when “<” **IS** provided the same input will result in an attempt to read from STDIN.

References

[1] Jordan Dimov. Security Issues in Perl Scripts. URL: https://www.cgisecurity.com/lib/sips.html (visited on 10/04/2021).

[2] Jon Hood, ed. SwATips. https://www.SwATips.com/.

[3] David Svoboda. IDS31-PL. Do not use the two-argument form of open(). Nov. 16, 2017. URL: https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88890543 (visited on 10/04/2021).