Global Information Assurance Certification Paper

SECURITY IN SCRIPTING:
SIGNIFICANT THREATS IN INSIGNIFICANT SCRIPTS

GIAC Security Essentials Certification (GSEC)
Practical Assignment, Option 1
June 27, 2004

Fred Chagnon

ABSTRACT

"Security is everybody's business. You may ask yourself, "Why should I take security seriously? I don't have anything on my system that's worth exploiting." Well, that's exactly what the bad guys want you to believe."

-- Randal L. Schwartz

There is a scary reality that's becoming ever more prominent as we continue to see an increasing amount of software vulnerabilities, and the worms and Trojan programs that exploit them. That reality is that most programmers do not possess the necessary awareness to program securely. Security, and security awareness is a mindset that has not been commonly adopted in the world of software development. Having spent the last few years of my life scanning multiple security awareness mailing lists by day (part of a UNIX system administrator's duties) while being schooled in system and application programming at night, this gap in knowledge and education has been made very clear to me. Programmers are not taught rudimentary principles of security surrounding programming at university or college, and many of them may go their entire development careers without so much as a thought on what *really* happens behind that innocent `printf()` statement.

The good news is that a quick search on the Web for information on buffer overflows, format string overflows, adjacent memory attacks, and other stack smashing tactics will return a plethora of information on the subject. This is no surprise when you consider all the hype surrounding these types of attacks lately. However, most of these problems deal specifically with large applications written in C or C++. The purpose of this paper, however, is to look at the security issues surrounding the more overlooked, under-rated and innocent programs that are often the wrappers or glue that keep the large applications running; those humble scripts written in the Bourne shell or Perl.

Script authors (usually software developers, or system administrators) often pride themselves on their unique ability to craft a fancy script to do a simple or complicated task before it's time for lunch. However, more often than not, that undocumented, obscure one or two liner's function has impressed someone in management, and the script ends up as a frequently scheduled task (a cron job) without a further shred of effort to check for potential risks. It is for this reason that secure scripting practices be given the attention they deserve, for the potential damage of exploitation in a small script can be just as severe as that of a largely commercial application.

Below we will discuss the most common security concerns that present themselves in simple scripts, using as examples, both a low-level scripting language (the Bourne Shell) and a high level scripting language (Perl). Comparing these two very common scripting languages side by side allows us to focus on the implications of the security issues at hand, rather than get too wrapped up in the inner workings of a single scripting language. Furthermore,

this comparison should give the reader some ideas to consider when deciding which of the two languages is more appropriate for the task at hand.

The ideal audience should boast a working knowledge of both the Bourne shell and Perl, and knowledge of a basic UNIX environment is assumed.

SECTION A -- INPUT BASED VULNERABILITIES

Input is essentially the user's key to controlling a program in execution. Input to programs can be directly prompted, can be supplied as an argument, read from a configuration file, or even come in more subtle forms such as environment variables. Exploits relating to input vulnerabilities, such as the famous buffer overflow, occur because the program assumed the data to be of a certain type, and the attacker supplied something unexpectedly different. These types of vulnerabilities are the single most common types of exploits being used today and can lead to all kinds of problems, most commonly the execution of other programs in memory at escalated privilege levels (i.e. super user).

While it's enough to say that input data should always be validated before used, let's look at few ways that scripts can be better structured to not fall victim to these types of attacks in the first place.

Avoid 'Shelling Out' unnecessarily

Scripting language like the Bourne shell and Perl are often used to batch a series of other programs together. This is what makes them incredibly useful as *glue languages*. However, in many cases, this involves spawning a shell, which can leave the program open to an unforeseen input vulnerability. Both of these scripting languages have multiple ways in which external programs can be called. In the case of the Bourne shell, where there aren't high level alternatives to shell spawning, we'll explore some of the safety measures that can be put in place to prevent unforeseen behaviour. On the other hand, we'll be less forgiving with Perl and demonstrate the alternatives to 'shelling out'.

Both Bourne shell and Perl support the syntax of launching a program by enclosing the command in backticks. A command enclosed in backticks spawns an separate system shell (usually the Bourne shell, but can vary depending on the underlying platform) to launch its program, and the output can be captured in a variable, a pipe, or simply output to the screen. A savvy user can take advantage of this spawned shell process to gather data the programmer might not have expected. Take this innocent line of code for example.

```
$data=`cat /usr/prog/$datasource`
```

This line of Perl code looks like an innocent way of storing the contents of a file into a variable (the equivalent Bourne shell code would lack the first $). However,

suppose an attacker knew of the quick and dirty way in which this task was done, and managed to supply this as the value to the `$datasource` string:

```
"foo; cat /etc/passwd"
```

This value will be passed verbatim to the shell being spawned. The effect: The attacker has now populated the `$data` variable with the contents of the file `/usr/prog/foo` as well as `/etc/passwd`, the systems' password file, because this is what the spawned shell will have executed:

```
cat /usr/prog/foo; cat /etc/passwd
```

Truthfully, in most cases, gaining access to the password file of a system isn't always serious enough to be considered an attack (account passwords are typically stored in less accessible locations), especially if the user running a shell script on the local machine and has read access to the file anyway. However, suppose this was a CGI script running on a web server? Causing a CGI script to output the contents of the web server's password file in a browser should be serious enough to keep an administrator up at night.

Bourne shell script authors can prevent the unexpected processing of meta-characters (such as the semi-colon illustrated above) when launching commands by suffixing the command with the double-dash switch (`--`). This switch is a way to explicitly divide a program's switches from its arguments. This prevents an attacker from being able to supply the command with a bogus argument, such as a file beginning with a dash, in an attempt to trick the program into using an unexpected switch. Let's look at the implication.

```
rm –i $somefile
```

This command would serve to remove `$somefile` but the `–i` switch tells the command to prompt the user first. What if the value `$somefile` began with a `-f` before supplying a file name, effectively causing our script to launch into an `'rm –i –f filename'`? This force switch would override the interactive switch (expected behaviour described in the `rm` man page) and indeed not prompt the user before deleting any files. This exploit can be averted with this small modification:

```
rm –i -- $somefile
```

Now the shell is aware of the clear delineation between the switch (`–i`) and the argument (`$somefile`) and will not be convinced differently with creatively crafted input. If the above exploit is attempted again, it will correctly expect that '-f' is actually the name of a file being supplied to the `rm` command.

Perl programmers have the alternative of using the `system()` function to launch external programs, rather than resort to backticks, however this method, when not implemented properly, can still spawn a shell and thus be susceptible to all the problems we illustrated above. (with Perl, the shell invoked is that from the $ENV{SHELL} environment variable). However, unlike the backticks, the `system()` function boasts a bit more intelligence. It accepts, as arguments, a list starting with the name of the program, followed by any command line arguments. It will spawn a shell only if its first argument contains any special meta-characters that require a shell for interpretation. If no such character exists, Perl uses the safer, more efficient `execvp()`call to handle the process. Let's look at how the same command could be executed in different ways, and explore the implications.

```
system ("ls -al");
```

Here we're giving Perl the well-known command of listing the contents of the current working directory. However, because, in our first and only argument, we've used meta-characters that only a shell can deal with, Perl will pass the command to a spawned shell to do the work. This can lead to the same problem we explored above using backticks. Instead, we can achieve the same goal by doing this:

```
system ("ls", "-al");
```

Now we've avoided shelling out, because our first argument does not contain any meta-characters, and the rest of the arguments can safely be handled by the `execvp()` library.

There are several other ways in which Perl programmers can avoid using the shell is which mainly demands familiarity with the language's library of built-in functions. It's unnecessary to launch a call to `date` when Perl's `localtime()`function can provide the same data. Similarly, the above example of using `cat` to store the contents of a file in a variable would have been better handled using Perl's built-in file handling functions. Failing Perl's library routines, there's the possibility that a perl module can be used in place of a shell tool. The Comprehensive Perl Archive Network (http://www.cpan.org) contains a plethora of modules whose functionality may serve to replace that of a common UNIX cool. It's safer to build a program dependant on the Net::Whois perl module rather than risk spawning a shell to gather and parse the output of a `whois` command.

Sanitize your Environment

Looking back at our previous example of launching the `ls -al` command, one aspect remains unclear: Just which `ls` are we calling? Is it `/bin/ls` or possibly `/usr/bin/ls`? Since it's not explicitly set, the system looks at the program owner's $PATH environment variable to determine where to find the `ls`

program. If an attacker has altered the $PATH environment to point to a directory containing a rogue `ls` binary, the system can be compromised. This is known as a PATH exploit.

Programmers should never assume where a program being will be executed from. Rather the $PATH environment variable should be explicitly set very early on in the script when external programs are going to be used.

In Bourne Shell, something like this will suffice:

```
PATH="/bin:/usr/bin:/usr/local/bin:/opt/bin"
export $PATH
```

or alternatively in Perl:

```
$ENV{PATH} = "/bin:/usr/bin:/usr/local/bin:/opt/bin"
```

However, one could simply refer to programs by their absolute paths. Alternatively if there are several references to several programs, store the absolute paths in variables at the top of the script and refer the binary by its variable name.

Furthermore, script authors should also take care to explicitly define the $IFS environment variable. This less familiar variable defines the characters considered as *white space* characters when expanding an expression into a list. If an attacker sets this variable to something sneaky, the shell can drastically misinterpret its commands. Even unsetting it as done below will suffice.

```
IFS=""; export $IFS
```

While this is mostly applicable to the Bourne shell, remember that a perl script running out of a crontab will inherit it's user's environment which includes variables like $ENV{PATH}, $ENV{IFS}, and $ENV{SHELL}. Therefore, perl scripts running out of cron should have these variables defined explicitly. In fact, the perlsec man page suggests the following environment cleansing line of code:

```
delete @ENV{qw(IFS CDPATH ENV BASH_ENV)};
```

Perl users need to be weary of the @INC array for reasons similar to those discussed above with PATH exploits. The @INC array contains the list of directories that perl will search through when it encounters a `require` or `use` command; these directories are typically the locations of your perl bases as well as any modules that may be installed. If an attacker alters this list, one can not be sure which modules is being loaded. Manipulation of the @INC array is done with the use of the lib module which ships with is part of the perl base system.

Validate your input

By now the reader should be convinced that any and all externally influenced input should be validated before it is used. However, the science of proper input validation techniques is quite a vast topic, and is could be the focal point of an entirely different research paper. Consider this though: If input validation is even a faint concern for the task at hand, Perl's taint mode may be the answer.

Perl's taint mode is invoked when Perl is called with the –T switch (either from the command shell or in the script's shebang line). It is also invoked automatically when the script's real user or group ID differ from their effective user or group ID. This is covered in greater detail later when we discuss setuid scripts.

The purpose of taint mode is to ensure that foreign data derived (input from the user, the environment or other sources) be considered tainted, and therefore not be permitted to affect anything else outside the program. Essentially, before tainted data can be used, it must be validated, or else an error is generated at compile time. For example, if one still relied on their derived $PATH environment, despite everything covered above, taint mode would provide the appropriate wrist slap by failing to execute the insecure code:

```
Insure $ENV{PATH} while running with –T switch at
./tainted_env.pl line 2, <STDIN> chunk 1.
```

Explicitly setting the $ENV{PATH} would fix this because the variable data would be no longer considered foreign to the program. Validation occurs when a substring is extracted from the tainted data, which could be the result of a regular expression or other validation mechanism.

SECTION B – OTHER SECURITY CONCERNS

While input based attacks are clearly the most common exploits used on programs today we can't neglect the importance of two other glaring security holes often found on our innocent scripts. They are the race condition, and the setuid bit.

While any budding young programmer is most likely able to define a race condition, even veteran programmers overlook these security bugs in their code. Race conditions remain a pretty popular and serious security concern, and unfortunately, are not limited to C/C++ applications.

Though race conditions can manifest themselves in various situations, the type most applicable to our discussion is the Time-Of-Check-Time-Of-Use (TOCTOU) scenario. This type of race condition exists when there is the possibility that a change can occur between a pair of non-atomic operations. The most common occurrence of this is accessing files for input or output.

The setuid bit isn't seen as a vulnerability because the setuid bit is activated explicitly by the programmer or system administrator. However, if a script running setuid is vulnerable to other forms of attack, specifically those mentioned above, a setuid program is in a far worse position to do damage to the system. We discuss methods of avoiding setuid below.

The Symlink Bug

Often dismissed as a non-critical bug, the symlink bug is quite a low-tech that takes advantage of a TOCTOU race condition, or the complete failure to check for the existence of a file before opening it for writing. Consider the following:

```
1. #!/bin/sh
2. if [ -x /tmp/tempfile ]; then
3.   echo "This is a temp file." > /tmp/tempfile
4. fi
```

Although, we can applaud the script author's effort to at least check for the existence of a file, the above code is still susceptible to a symlink bug. Suppose an attacker was able to execute this line on the shell between the time that line 3 and line 4 were executed by the system.

```
ln –s /bin/sh /tmp/tempfile
```

Now the attacker will have linked /tmp/tempfile to /bin/sh. If the script was running with sufficient privileges to overwrite to this file (i.e. as root), the Bourne shell program would be effectively deleted from the system, making shell script execution impossible. Furthermore, any user with this shell in the system password file (the default shell for root on BSD and some Linux systems) would be unable to login at all.

Of course, this is just a theoretical example, and the attacker would have to possess the timing of a modern-day Houdini to exploit it. In practice however, hackers will typically bog the system down with resource hogging processes and Denial of Service attacks before attempting to launch a timing attack like the one above. So it's not as far fetched as one might think.

The theoretical solution to this bug is to check for a file's existence and open it for writing in one system call, so the processor doesn't have time to execute another operation. Unfortunately, this isn't always possible in shell or perl scripts. However, script authors can take precautions with their temporary data by arranging for a safe place to do the work, and by using unpredictable filenames. Best practices for playing with temp files are covered in the next section.

Create Secure Temp Files

Temp files are used very often in scripting practice; perhaps too often. The vast majority of the time, temp files are blindly written to without even so much as a

check to see if the file even exists. We've already seen the dangers that can happen with TOCTOU race conditions even when such a check is made. It should also be evident in the above example that the filename `/tmp/tempfile` isn't exactly what we'd call unpredictable. A symlink trap could have been setup on this predictable filename long before it is ever accessed, just begging to be accessed by the next insecure script. Given all these concerns, what steps can we take to ensure that our tempfiles aren't vulnerable to symlink attacks?

A start would be to avoid using tempfiles in the first place. We've already mentioned that tempfiles are used more frequently then they should be. The programmer is encouraged to consider alternatives to storing data in a file if possible. Data can be stored in a local variable or exported in an environment variable. Output could be left write to STDOUT, which is usually the terminal, but this option also leaves the data able to be sent through a pipe as well. For example, if the purpose is to produce some output in a file, and then print the file, why not just pipe the output right to the `lpr` command, and remove the middle-man?

Consider the /tmp directory on a system The Badlands. Any user can read it, and more importantly, any user can write to it. If the script must use the /tmp directory at all, have the script create a directory first, with secure permissions to do all it's work within. The reason a directory is favourable to a simple file is because the mkdir command will fail if the directory already exists, even as a symlink. Also, both the Bourne shell and Perl use the $$ variable to signify the script's current process ID. We can use this to assist in making the directory name unique by having our script execute the following command:

```
(umask 077 && mkdir /tmp/somename.$$/) ||
    echo "Could not create directory!" 1>&2; exit 1
```

Perl programmers who are considering having their script invoke a shell to perform the above command had better hang their heads in shame.

```
my $tmpdir="/tmp/somename.$$.$^T";
mkdir ($tmpdir, 0700) or
     die "Could not create directory: $!\n";
```

Note in the Perl example, we've also used the BASETIME variable ($^T) as a way to further randomize the directory name.

With the above checks and bounds in place, we've created a safe haven for data manipulation that is not susceptible to sym link bugs or other race conditions. There are however, utilities built specifically for this problem. We'll take a look at them, and compare their features to our own homegrown solution.

The mktemp utility
A good solution for Bourne shell programmers is the use of mktemp. mktemp is a program written by Todd Miller specifically designed for the proper handling of tempfiles and directories. It is available for free at http://www.mktemp.org/. mktemp's main purpose is to add an element of chaos to the user's selected file name by suffixing random characters to the filename (or directory name given the –d option). Our previous exercise could have been emulated by mktemp in the following manner:

```
(umask 077 &&
     directory = `mktemp –d /tmp/somename.XXXXXXX`)
```

mktemp replaces our given X's with it's randomness, thus generating a nicely unpredictable string, which it then outputs to STDOUT.

The only real downside to mktemp is that it doesn't ship with some older versions of Linux or FreeBSD, and isn't currently present at all on a default install of Solaris. If portability is your concern, mktemp may not be a dependency you want to introduce to your script. In this case, the exercise with mkdir should suffice.

The File::Temp perl module
Perl programmers have an elegant way to manage tempfiles securely. File::Temp is a module that ships with a base perl install, thus negating the need to ensure it is installed separately. Among its many features, it can be used to create a single random tempfile (the filename and file descriptor are returned at the same time to prevent further race conditions) or, as we've done above, it can also be used to generate a random directory. Let's have another crack at the above exercise to see how File::Temp would handle it.

```
use File::Temp qw/ tempdir /;

my $dir = "/tmp/somename.XXXXXXXX";
$tempdir = tempdir ($dir);
```

Notice that File::Temp uses a very similar input method to mktemp, in that it uses the X's as a template to determine where to place the randomized characters.

Both mktemp and File::Temp are highly recommended when dealing with the creation of temporary directories. However, in the event that the script will be used in an environment where the presence of these utilities cannot be guaranteed, the fallback method of using mkdir with a suffixed PID tag will suffice.

Avoid setuid scripts like the plague.
A script in execution runs with the privileges of the user who executed it. These privileges are also known as the *real UID* and the *real GID*. CGI scripts running

on a webserver will be owned by the user who owns the webserver process (this is why webservers should never be run as root, but rather by a non-privileged user like nobody). However, scripts can be setuid which imbues them with the ability to exceed the privileges of the real UID. These privileges are also known as the *effective UID* and the effective *GID*. The UNIX `passwd` program is the most commonly referred to example of a setuid program; it allows an unprivileged user to change data reserved only for the root user (the user's password).

While having this kind of access can be handy, it is extremely dangerous. **Shell scripts should never ever be setuid**. In fact, some Linux and BSD systems completely disable the ability for shell scripts to run setuid in the first place. In most cases, making a script run setuid is an ad-hoc workaround put in place to avoid a more complicated solution. Up to this point we've already seen several security problems that could plague a shell or perl script, but operating systems in the past have dealt with bugs at the *kernel* level which exploited setuid scripts.

As we discussed before, a perl script is automatically running in taint mode when it runs setuid. This provides us with a bit of relief as it ensures that malicious input or a poisoned environment has less of a chance of leading to exploitation, but it's not perfect. Unfortunately not all perl function calls acknowledge the presence of taint mode, so the programmer isn't absolved of all secure responsibility.

On the other hand, astute programmers should recognize the benefit in Perl's ability to ensure that a script is *not* running setuid when a critical task is about to be performed (most notably file handling). Since the real and effective UID and GID are stored in special variables, they can (and should) be explicitly defined.

```
$> = $<   # sets the effective UID to the real UID
$) = $(   # sets the effective GID to the real GID
```

Regardless of some of the advantages that Perl may have over the Bourne shell with respects to setuid scripts, making a perl script setuid should be avoided just the same. Script authors should leave setuid business to C/C++ applications.

CONCLUSION

We focused on the scripts in this discussion to show that it doesn't take a large enterprise class application to create a vulnerability serious enough to compromise an entire system. That even a tiny one line script, when not implemented properly, can lead to serious damage.

Shell scripts have their place. They're great in an environment where the users and the data can be trusted. (i.e. startup/shutdown scripts, batch jobs). The shell's design as an interactive language, however, make it inherently flawed for

defending against input based attacks and therefore should be avoided in circumstances where data is being read from untrusted sources. However, even if shell scripts aren't being used for 'secure tasks', they should still be programmed with the techniques discussed here, and elsewhere. Avoid spawning a shell involving user input, watch out for poisoned environment variables, be sure to make secure temp files, never ever make a shell script setuid.

Perl has proved a more capable solution for secure programs with its vast array of input validation techniques and its ability to avoid the shell through the use of high level function calls and imported modules. Nevertheless, all of the advantages can be nullified if the techniques are not implemented properly. Make use of Perl's `strict` module and invoke Perl with the –w flag to provide extra warnings on unsafe errors in the script's code. When dealing with external data, enable taint mode (by invoking Perl with the –T flag) to ensure your program isn't used as a conduit for destruction. Read the *perlsec* man page as well as the plethora of other articles in publication and on the Internet about Perl security.

A quick scan of security awareness mailing lists like BugTraq and CERT will show that computer applications are being exploited just as quickly as they are being patched. Now, more than before, the accountability is being left on the shoulders of the programmers responsible for the vulnerable code. Therefore security awareness should be a priority for all programmers. We won't all be security gurus, but we should all be aware of common threats, and write our programs with security in mind.

REFERENCES.

COLLEY, Shaun. "**Crafting Symlinks for fun and profit.**" April 12, 2004
URL: http://www.infosecwriters.com/texts.php?op=display&id=159 (February 2004)

DIMOV, Jordan. "**Security Issues in Perl Scripts**."
URL: http://www.developer.com/lang/other/article.php/631321 (May 2004)

DIMOV, Jordan. "**Security Issues in Perl Scripts: Perl Taint Mode.**"
URL: http://www.developer.com/open/article.php/631331 (May 2004)

DIMOV, Jordan. "**Introduction to input validation with Perl**."
URL: http://www.developer.com/net/cplus/article.php/861781 (May 2004)

HUSSEIN, Kamran and Robert Breedlove. "**Perl 5 Unleashed**." Sams Publishing, September 1996.

NORDHAUSEN, Stefan. "**Safely Creating Temporary Files in Shell Scripts.**", February 10, 2004.
URL: http://www.linuxsecurity.com/articles/documentation_article-8886.html (March 2004)

SCHWARTZ, Randal L. "**Perl Advisor: Computing Securely**", September 2003.
URL: http://www.samag.com/documents/s=8859/sam0309k/sam0309k.htm (March 2004)

WALL, Larry, Tom Christiansen and Randal L. Schwartz. "**Programming Perl, 2nd Edition**". O'Reilly & Associates, September 1996.

WHEELER, David A. "**Secure Programming for UNIX and Linux HOWTO**", March 3, 2003.
URL: http://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO/index.html (April 2004)

WHEELER, David A. "Secure Programmer: Keep an eye on inputs.", December 19, 2003.
URL: http://www-106.ibm.com/developerworks/linux/library/l-sp3.html (March 2004)