Global Information Assurance Certification Paper

Perl CGI Security

Introduction

Larry Wall developed Perl in 1986 and released it in 1987. It was originally developed as a scripting language for Unix system administrators. Larry said he "developed Perl -- an interpreted data reduction language -- to solve a problem that awk could not handle (Kim)." It was a combination of awk, sed, and sh, with elements of c, csh, Pascal and Basic. According to the original man page, "Perl is a interpreted language optimized for scanning arbitrary text files, extracting information from those text files, and printing reports based on that information. It is also a good language for many system management tasks (Wall)." Perl has become the tool of choice for other tasks as well, such as rapid prototyping and even full fledged application development.

Perl provides the system administrator with an extension of the shell programming interface for UNIX that allows him to accomplish tasks that would otherwise be too time consuming to code in other languages. With the addition of Perl32 for Windows, NT system administrators now have some of the same tools available to UNIX administrators. Typical tasks that are coded in Perl include, adding new users, checking system logs, network maintenance and backup, and file system administration.

Perl was adopted as an early executable language for HTML pages on the web to provide interactivity in web pages, such as data collection and verification. The problem with Perl as a CGI language is similar to other languages that interface with the base operating system in that they open the operating system to uninvited intrusion. Perl in itself is not an insecure language. It is its ease of use by novice programmers and the availability of "canned" scripts on the web that open vulnerabilities to the operating system.

Perl Security

Most beginning Perl programmers and some experienced ones do not realize the damage that an intruder can do using their programs. Hackers have used unsecured Perl programs to steal valuable customer information, credit card numbers and identities. While most programmers assume the operating system will protect their programs, they would be wrong. The ease of locating Perl programs on the server is what lures intruders. Typically these programs have full system privileges. This makes them a prime target for the cracker that wants to gain full access to a server. The SANS Institute lists CGI and Perl vulnerabilities as one of the top ten critical Internet security risks (SANS).

Both Microsoft Windows and UNIX, including all of its variants, have some vulnerability when used as web servers. The unrestricted and ingenuous use of Perl as a CGI language can increase these vulnerabilities many times. Not all operating systems and web host applications have the same vulnerabilities, nor can they be similarly protected.

Security Rules for Perl Programmers

There are some basic tenets that can be used to protect a network that is using Perl as one of its CGI programs. While these will not protect the network from an idiot programmer they will insure that the all-important "due diligence" is accomplished.

Perl Programmers Rule #1: Know your code!

In general Perl programmers should be skeptical of scripts that they download from the web. The power of Perl is its ubiquitous use as a CGI language. This is also its failing from a security standpoint. Perl programmers are efficient opportunists. They will use another programmer's code to solve their problem, if it is available. The Perl programming community makes this very easy to accomplish. Sample code is available from many sources, but notably the Comprehensive Perl Archive Network or CPAN, www.cpan.org.

The programs obtained at these sites, while developed to solve a particular problem, may contain bugs or in extreme cases errors that can expose a network to unscrupulous intruders. It is extremely important that Perl programmers read the source code. They should know exactly what it does and why it does it. This alone will protect a network from most attacks.

Perl Programmers Rule #2: Remove all sample code from the server.

Operating systems and web hosting applications ship with sample programs to get a web site up and running on the server as soon as possible. Most system administrators know nothing about these sample programs and leave them on their servers for their programmers to utilize. This is unwise and very insecure. It is imperative that the system administrator remove all sample programs from the server. Moving them to another server is fine, as long as they do not remain on the system path. At the same time, all documentation should be removed from the web server. This documentation can itself contain sample code.

A Perl savvy administrator will look for sample code in the following locations on an IIS Server: c:\inetpub\iissamples, c:\inetpub\iissamples\sdk,c:\inetpub\AdminScripts. A UNIX administrator will find similar scripts under /bin/perl/lib/samples on some systems, but this depends on who installed it and what version of Perl is installed. He should read the documentation, find out where the sample code is located and remove it from the server, along with the documentation.

Perl Programmers Rule #3: Remove all unused code from the server.

In addition to removing all sample code, the programmer or system administrator should remove all Perl scripts that are no longer used from the server. The system administrator should use the web server log files to determine what programs are run from the server. In UNIX he can look at the *httpd.conf* file to find the path to the cgi directories. Take note that all cgi files do not end in .cgi. They can have other

extensions, such as .pl (for Perl). Look in the path environment statements to make sure that none of the system administration scripts are deleted.

Once again, this goes back to the old saying, "know your system." The programmer should know what scripts are needed to operate the web server. Old scripts should be eliminated, or at least stored on another server, that is inaccessible from the web. The theory here is to have the minimum number of scripts on the system so that it can perform it operation.

Perl Programmers Rule #4: Do not depend on Perl's built-in security to keep your code secure.

The Perl documentation itself lulls a beginning Perl programmer into believing that he cannot write insecure code. An excerpt from the Perl documentation says "Perl is designed to make it easy to program securely even when running with extra privileges, like setuid or setgid programs. Unlike most command line shells, which are based on multiple substitution passes on each line of the script, Perl uses a more conventional evaluation scheme with fewer hidden snags. Additionally, because the language has more built-in functionality, it can rely less upon external (and possibly untrustworthy) programs to accomplish its purposes (CPAN)."

The device that Perl uses to protect the programmer from inadvertently executing insecure code is called "taint mode." This is a built-in function of the Perl interpreter that checks the Perl program for a set of security violations, such as verifying that path directories are not writeable by anyone but the program. In addition, data that is identified as "tainted" may not be used in a command that calls a sub-shell, or in a command that changes a file or directory. But there are exceptions.

The exceptions are what catch the unwary programmers and system administrators. First, "taint mode" has to be explicitly requested on the command line with a –T option or the UNIX permission bit, setuid, set to 04000, or the setgid bit set to 02000. Only these options will cause "taint mode" to be started. Once it is started it continues for the remainder of the script.

After questioning programmers and system administrators from a large web hosting company in my area, I found that the programmers knew nothing of "taint mode" because they did not make the programs executable on the web server, the system administrator did. The system administrator knew nothing about Perl programming so he never set the setuid, or setgid bits to automatically initiate "taint mode." Perl's automatic, built-in security was unknowingly circumvented by two of the most common human traits, lack of communication and failure to read the documentation.

What it also states, later in Perl the document, is that a Perl programmer can purposely execute insecure code, even with "taint mode" initiated. If the Perl program is written so that is passes a list of arguments to either *system* or *exec*, they are not checked for "taintedness." The same applies to *print* and *syswrite*.

The following are examples of tainted and insecure code taken from the Perl documentation:

```
$arg = shift;                    # $arg is tainted
$hid = $arg, 'bar';              # $hid is also tainted
$line = <>;                      # Tainted
```

```
$line = <STDIN>;                        # Also tainted
open FOO, "/home/me/bar" or die $!;
$line = <FOO>;                          # Still tainted
$path = $ENV{'PATH'};                   # Tainted, but see below
$data = 'abc';                          # Not tainted

system "echo $arg";                     # Insecure
system "/bin/echo", $arg;               # Secure (doesn't use sh)
system "echo $hid";                     # Insecure
system "echo $data";                    # Insecure until PATH set

$path = $ENV{'PATH'};                   # $path now tainted

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

$path = $ENV{'PATH'};                   # $path now NOT tainted
system "echo $data";                    # Is secure now!

open(FOO, "< $arg");                    # OK - read-only file
open(FOO, "> $arg");                    # Not OK - trying to write

open(FOO,"echo $arg|");                 # Not OK, but...
open(FOO,"-|")
    or exec 'echo', $arg;               # OK

$shout = `echo $arg`;                   # Insecure, $shout now tainted

unlink $data, $arg;                     # Insecure
umask $arg;                             # Insecure

exec "echo $arg";                       # Insecure
exec "echo", $arg;                      # Secure (doesn't use the
shell)
exec "sh", '-c', $arg;                  # Considered secure, alas!

@files = <*.c>;                         # insecure (uses readdir() or
similar)
@files = glob('*.c');                   # insecure (uses readdir() or
similar)
```

As a last warning, the same Perl documentation cautions this for the programmer and system administrator. "The tainting mechanism is intended to prevent stupid mistakes, not to remove the need for thought. Need I say more?

Perl Programmers Rule #5: Sanitize your data!

The data that is sent to a Perl program from an Internet connection is completely out of the programmer's control. This is true unless the Perl programmer takes the precaution of laundering that data through some simple Perl script. Most programmers think that removal of some well-known meta-characters from the data and replacing them with underscores is a common way of accomplishing this task. They would be wrong.

"The problem with this approach is that it requires the programmer to predict all possible inputs that could possibly be misused. If the user uses input not predicted by the programmer, then there is the possibility that the script may be used in a manner not intended by the programmer (CERT)."

A better way to accomplish this, according to CERT, is to "define a list of acceptable characters and replace those not on the list with underscores. This benefits the programmer by returning control of the received data to him. It allows data, which he specifically defines, as safe. CERT furnishes a sample Perl code snippet to demonstrate the concept.

```
#!/usr/local/bin/perl
$_ = $user_data = $ENV{'QUERY_STRING'};       # Get the data
print "$user_data\n";
$OK_CHARS='-a-zA-Z0-9_.@';     # A restrictive list, which
                               # should be modified to match
                               # an appropriate RFC, for
                               example.
s/[^$OK_CHARS]/_/go;
$user_data = $_;
print "$user_data\n";
exit(0);
```

This concept should not only be applied to data received from the internet, but should include file access which is received from the internet. Perl uses special characters such as > and | which should be check for in file names. As mentioned in #4 above, the *eval* function is especially vulnerable to meta-character attack, since the Perl interpreter does not filter it. CERT is not the only one with warnings about this problem.

CIAC also warns that, "To prevent CGI script compromise, avoid passing remote user input directly to command interpreters such as Unix shells, other interpreters such as Perl and AWK, or programs that allow commands to be embedded in outgoing messages, such as '`/usr/ucb/mail`'. If user input must pass to these types of programs, filter the input for potentially dangerous characters before it is passed along. These characters include the period (.), comma (,), slash (/), semi-colon (;), tilde (~), and exclamation point (!). (CIAC)"

Perl Programmers Rule #6: Know your Web Server Application

Programs that instantiate a web server are notorious for creating security leaks. While the developers of these programs attempt to patch them as fast as the leaks are found, it is up to the system administrators to know which patches to apply. Most system administrators, and I speak from experience; do not know what patches are needed to secure their server, let alone the CGI programs that run on the server. It is obvious that some help is needed to alleviate the confusion.

Help is available for servers that support CGI.A system administrator needs to look at the Network Security Library site, http://secinf.net/info/www/cgi-bugs.htm. He also, needs to be aware of the Microsoft's Internet Information Server 4.0 Security Checklist, found at http://www.microsoft.com/technet/security/iischk.asp.

What items are found in these security sites? Here is a good example. As was noted earlier, intruders want to find the location of the servers Perl scripts, so that they can run sample scripts or run scripts of their own. Early versions of IIS 4 could return the location of the Perl directory if a non-existent file was requested. This can be easily fixed by replacing the perl.exe with the perl.dll (Most Comprehensive List).

Conclusion

As with any network security issue there is no magic bullet. There is only the persistence and determination of the network administrators and programmers, working in concert, to keep the system secure. The list of rules provided above is a good start to securing Perl on a network server. Since the web server and the network are a dynamic, growing things, there must be continued checking and vigilance.

Bibliography

CERT® Coordination Center. "How To Remove Meta-characters From User-Supplied Data In CGI Scripts." 2 May 2001. http://www.cert.org/tech_tips/cgi_metacharacters.html

CIAC, Computer Incident Advisory Capability, U.S. Dept. of Energy. "Securing Internet Information Servers," CIAC-2308 R.3, UCRL-MA-118453, December, 1994 (Revised August, 2000). 9 April 2001. <http://www.ciac.org/ciac/documents/ciac2308.html#6.7>

Herrmann, E. Teach Yourself CGI Programming with Perl 5 in a week. Sams.net Publishing. 1997.

Kim, E.E. "A Conversation with Larry Wall." Dr. Dobb's Journal. February 1988. 12 April 2001. http://www.ddj.com/articles/1998/9802/9802a/9802a.htm,

Microsoft, Microsoft Internet Information Server 4.0 Security Checklist, March 15, 2000. 11 May 2001. http://www.microsoft.com/technet/security/iischk.asp

Most Comprehensive List of CGI & httpd Bugs, The. compiled by lirik@nanko.ru 31.Mar 1999. 14 May 2001. http://secinf.net/info/www/cgi-bugs.htm

Roth, D. Win32 Perl Scripting: The Administrators Handbook. New Riders, November 2000. pg.

SANS Institute, The. "How To Eliminate The Ten Most Critical Internet Security Threats." Version 1.32. January 18, 2001. 12 May 2001. http://www.sans.org/topten.htm

Wall, L. "Perl." Internet Software Consortium. Jan 31 1988. 28 April 2001. http://sources.isc.org/devel/lang/perl.txt,