Global Information Assurance Certification Paper

Securing Perl Scripts

Brett N. DiFrischia
July 3, 2004

GSEC Version 1.4b, Option 1.

Abstract

Perl is an open source, widely used scripting language made available under the GNU General Public License. Its power and flexibility have made it one of the languages of choice for Rapid Application Development. The language is built with many interfaces to the external world, including CGI, Berkeley Sockets, and the Perl Database Interface. These tools give a programmer the ability to perform a wide variety of tasks with only a few lines of source code.

Unfortunately, the same power that makes Perl a great programming language also allows a developer to open security holes to external users. Covered in this paper are examples of key interfaces, which could allow security flaws. Once these security flaws are identified, the mechanisms to assist developers in generating locked down code are presented.

Introduction

Perl[1] is a powerful scripting language created by Larry Wall[2]. As a partially compiled/interpreted language, Perl performs wonders in the domain of Rapid Application Development (RAD)[3]. The primary sources of Perl's power are its extensive capabilities, curt and sometimes difficult syntax, and, most importantly, a widespread user support system.

The great power presented to programmers by the Perl language "make[s] the easy jobs easy, without making the hard jobs impossible."[4] Unfortunately, this same ability also gives programmers great potential for opening security holes in their systems.

The purpose of this research is to make readers aware of potential security holes. First, it gives a small sample of possible ways for a programmer to introduce security risks into their programs and libraries. After these problematic areas are revealed, it will give suggestions on how to protect Perl scripts.

Use with Caution: CGI.pm

Common Gateway Interface (CGI)[5] provides an interface between which data servers and clients can exchange information interactively. This standard is of such use that the Perl source ships with a standard CGI module, CGI.pm[6]. In fact, the widespread use of Perl for creating CGI applications[7] has categorized

Perl as a web application development language. While this is certainly a great use for Perl, it is by no means the only one.

CGI allows a client to send pieces of information to the data server, as stated above. What this does not reveal is that the client has a great potential to control and modify that data. The server-side programmer must realize that any information from an external source is thus potentially dangerous to the server-side system.

Some examples of malicious data include:

- System or Perl commands typed into text input fields
- SQL injection (see below)
- Hidden data and cookie fields, which could contain sensitive or runtime configuration data[8]

The last item sometimes comes as a surprise to programmers. However, these fields are easily viewed through HTML source or client program in an HTTP application. As such, creating an interactive local proxy[9] or using a text processor or homebrewed program can allow the client user to modify this data.

CGI leaves the programmer responsible for many security tasks. As such, a CGI application is often a first target for cracking a server.

Use with Caution: Berkeley Sockets

Perl has an extensive built in Berkeley Sockets API. This functionality allows Perl programmers to instantiate and to connect to Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) based sockets quickly and efficiently[10]. Many modules are available to perform common socket tasks and even generate raw IP packets, the most notable of which is IO::Socket, which allows an object-oriented interface to the socket API. Because of this, the ability to fork() and multithread, and its powerful syntax, Perl is well suited to the needs of network programmers.

Of course, network programming carries with it a tremendous responsibility. Most notably is the need to keep intruders out while allowing clients in. However, once a client (intruder or not) has access to the underlying network application, the programmer must also prevent malicious behavior to the data being used or the underlying system.

Fortunately, Perl provides several built in mechanisms to minimize the security risk involved with external communication. For a more in depth view at network programming security, see Stein[10].

Use with Caution: Executing Code

Perl allows the programmer to execute code and system commands from within a script. This ability allows the programmer not only to interact with the underlying operating system, but also to evaluate Perl code itself. In fact, due to a lack of built in exception handling, the eval{} statement is often used to allow for internal error handling.

Interacting with the system is possible in multiple ways. First and most obvious is the system() routine, which takes its argument(s) and executes the associated system command, returning 0 on success and an error code on failure. Second is the backtick operator (`*string*`), which interprets the given string as a system command, returning the standard output of the operation. The I/O routine, open(), also allows system interaction by preceding or following the filename argument with a pipe(|). A leading pipe indicates that the program will write to the system command, whereas a trailing pipe indicates reading from the command. Finally, the exec() call can be used to replace the current program with the given command. All of these capabilities offer the potential for dangerous interaction with the system. Any command to be executed in this fashion should either be generated directly by the script or thoroughly validated.

Use with Caution: Privileged User Permissions

There are several ways in which a Perl script may be allowed to have privileged access to high security information. The first is simply by being run as root. This tends to be necessary for servers, among other programs, which need to access privileged information or routines on the system.

The other two are through the Setuid and Setgid permission bits. This mechanism, discussed below, can allow the executing user of a file to temporarily act as the owner.

When accessing the system with a privileged user ID, the real user can easily impact the rest of the system, either purposely or accidentally.

Use with Caution: DBI.pm

Database systems are prevalent throughout the computing industry. As such, you will rarely find a programming language that does not have some form of database API built into it. Perl is no exception to the rule. Combined with the aforementioned CGI and Socket capabilities, the Perl Database Interface (DBI)[11] allows for a powerful, database management system (DBMS) independent utility, giving programmers the freedom to concentrate on design over communication[12].

This functionality is extremely useful, but allows for programmer errors when interacting with the underlying DBMS. Not only does the programmer need to be

concerned with protecting impermissible access to the DBMS and the data contained therein, but they must also be wary of the data they use when querying the database. In this manner, the programmer is responsible for both authentication and validation of input data.

Of particular interest in this arena of study is SQL injection. Input from a user can easily imitate the underlying SQL, if not properly validated, typically by inserting single quotes on each end of the input string and inserting valid SQL between them:

```
$id = <>;   # Read from STDIN... 'admin'
$pw = <>;   #  "    "     "  ... "' OR '1'='1"
$query = "SELECT * FROM users " .
         "WHERE id='$id' AND password='$pw'";
```

Given the input values after the ... sequences, $query now contains the value "SELECT * FROM users WHERE id='admin' AND password='' OR '1'='1'"[13]. Upon querying through DBI for this result, the application will receive a one-row result, allowing the cracker to imitate the user 'admin' in this system.

Use with Caution: Public Modules

One of the most useful features of Perl is its user base support. Due to its wide scale use and open source origins, the users of Perl are quite willing to share modules and assist each other with programming challenges. This effort has been organized into the Comprehensive Perl Archive Network (CPAN)[14]. From this site, a Perl programmer can download and install a plethora of useful modules and programs, including Perl itself.

The many of modules available via CPAN are of various origins. A great deal of them are packaged and shipped as standard modules of Perl. Others are Perl interfaces to C, C++, and Java libraries. Most modules available through CPAN also come with self-tests, are reviewed by users, and have installation attempts and failures monitored at their home nodes.

Although a majority of these modules are benign, the programmer may find an occasional module, which, either maliciously or accidentally, damages their software systems.

Incident Prevention: Warnings and `use strict;`

Thankfully, Perl provides several safety mechanisms for its coders. Perhaps the simplest way to prevent incidents is to prevent coding errors. Perl offers several options in this category. The two most prevalent are command line triggered warnings and the strict.pm module.

When Perl is used with the command line flag –w, it will generate warning messages in what Perl considers potentially dangerous situations, including subroutine redefinition, usage of uninitialized variables, and other syntax and usage errors[15]. This flag can save the programmer loads of debugging time, and help him/her prevent unidentifiable security holes due to improper usage of Perl syntax.

strict.pm is a pragmatic module that will cause Perl to raise errors on variables which are undeclared or which it cannot resolve[16]. It also restricts unsafe operations involving references and subroutines. This not only saves the programmer great time and effort when debugging, but will also help identify troublesome areas of Perl code.

Incident Prevention: Taint Mode and Input Validation

The two tools described above, however, can only do so much. They primarily function to prevent syntax problems. When the security issue at hand becomes data, other mechanisms must be used. The primary tool for this purpose is taint mode.

Perl can be put into taint mode by using either of the command line options –t or –T[15]. The difference between the flags is that –t will generate warnings on taint violations, and –T will fail returning an error (clearly desired in production environments). Regardless, these flags interpret how Perl handles data. Taint mode is highly recommended for use with any program that must access untrusted data, including servers, CGI programs, setuid and setgid scripts, etc. Taint mode is highly restrictive. So much so that a –T that appears on a Unix #! line:

```
#!/usr/bin/perl –T
```

will prevent the script from being run directly through the Perl interpreter:

```
% perl myprog.pl
Too late for "-T" option at myprog.pl line 1.
```

Such an error is desirable, since the Perl interpreter has many powerful command line utilities.

Taint mode will taint data retrieved from any external source, including command line arguments, environmental variables, system call input, etc. Tainted data is not allowed to directly interact with the system or executed as or as part of Perl code[17]. These can be significant restrictions, making life more difficult on the programmer, but this is a desirable result. As such, the programmer must examine all data before using it in an insecure manner. This is typically done through the use of regular expressions (regex) since a matching subgroup of the regex can be used to interact as expressed above. This can still lead to errors,

though, as the programmer can simply match all data, and use the resulting sub match:

```
my $data = <>;          # Read from STDIN...  'rm -rf /'
$ENV{PATH} = '/bin';    # must set PATH, since it is
                        # external
#system $data;          # This would raise an error

$data =~ /(.*)/s;
system $1;              # This executes without fail
```

As you can see, this doesn't help avoid security issues.

This leads to the discussion of input validation. Input validation comes in two flavors: permissive and filtering. Permissive validation checks the input for a specific format, failing with the appropriate error message when that form is not followed. Filtering removes any illegal characters or patterns from the input and proceeds as normal with the freshly rearranged input as usual.

Thankfully, Perl has versatile regex syntax. As such, the clever programmer can quickly parse what is desired from the input or delete unnecessary patterns before proceeding to data processing.

Incident Prevention: SQL Injection

As a special case of input validation, SQL injection has potentially cataclysmic consequences for the database driven web that is known and loved (or hated) today. An unwary application programmer can allow a cracker to alter, destroy, or gain illegal access to database information by not validating program input.

For starters, the programmer can use the techniques described above, in "Incident Prevention: Taint Mode and Input Validation." The DBI.pm module, however, offers more help in the way of the quote method, available to the database handle. quote() takes a string and escapes any characters that may have special meaning within an SQL statement. In particular, it escapes quote characters themselves, making SQL injection nearly impossible[12].

Incident Prevention: Graceful Failure

Protecting a Perl script not only involves preventing malicious use of code, but also keeping the program from entering an erroneous state. This can be done in two ways in Perl: catching exceptions and handling signals. Doing so allows either program recovery or at least graceful failure of operations. Such behavior is desirable in applications such as servers, in which case it is necessary to be able to accept new clients, even when another fails.

Exception handling in Perl is done via the eval{} statement, as mentioned above. eval{} can be passed either a string or a block of code, which it will then interpret and execute as Perl code. If the code fails, Perl places the returned error message in the global variable $@ ($EVAL_ERROR when using the English.pm module)[18]. This technique allows the script to catch a fatal error which may occur during an operation, such as attempting to parse a malformed or invalid XML file.

Assigning subroutine references to elements of the global hash %SIG is the method for implementing signal handling in Perl. This method allows a script to intercept system signals such as TERM, INT, HUP, etc. before the interpreter halts, permitting the script to finalize its state before exiting, clean itself up and start from scratch, or do anything else that may be desirable[10].

Incident Prevention: Using Caution With External Libraries

Perl Culture is inherent with good-natured developers, willing to share their solutions to common problems on CPAN[14]. This kind of collaboration is the essence of open source computing, needless to say. However, people can make mistakes, and there are always a few bad eggs lurking out there, so it is necessary to take caution in using external libraries. There are several guidelines that make external libraries a more security friendly resource.

The first thing one should do when downloading any software package is to check the checksums and signatures available for that package. This includes MD5 checksums and PGP signatures, when available. Most unfortunately, CPAN does not provide this information yet.

Build the software package on a non-production, preferably isolated, system. Once the software is built, run any internal tests supplied with it. Finally, before installing it, test it heavily with your own software. This can take time, but it is definitely worth the effort.

Check all Perl libraries for interaction with the variables $< (EREAL_USER_ID or $UID), $> ($EFFECTIVE_USER_ID or $EUID), $((REAL_GROUP_ID or $GID), and $) ($EFFECTIVE_GROUP_ID or $EGID) (the alternative names available through the English.pm module are in parentheses). These variables affect the user and group IDs under which the software will be run. Any modules which read or modify these variables (unless desired) should be checked directly for malicious behavior. Typically, a search for these variables can be done with a simple grep command.

Many Perl modules are built on top of C libraries. Although a useful feature of Perl, this can also allow for any vulnerability present in the C software to directly affect the Perl module. For more information on securing C software, see Ahmed[19].

Finally, keep an eye on the external package distribution. It most likely contains bug fixes, but may also contain security enhancements and repairs. Yet again, due to a lack of monitoring resources, this is not easily done in CPAN without bookmarking every module used.

Incident Prevention: Do not Setuid or Setgid

Unix systems, in particular, allow a great power to the common user. If a specific permission bit is set on an executable file, users with the correct permissions can execute that file with the privileges of the file's owner. This is typically done in programs such as passwd, through which an unprivileged user can change his or her password in a database file owned and writable only by root. Although a powerful technique, this procedure is not recommended and can wreak havoc on a system. However, when necessary, Perl provides some security techniques for the programmer.

Any Setuid or Setgid program written in Perl automatically uses taint mode. This cannot be turned off. Also, Perl must be configured to allow for Setuid and Setgid. As such, if an administrator does not want this capability on his/her system, Perl can simply be built without it.

Finally, as a rule, do as many tasks as possible with an unprivileged user and/or group ID. Limit the amount of interaction between the program and the system when acting as a privileged user[10].

Incident Prevention: chroot

Often a program or module is required to interact with files on a local file system. As such, it is necessary to limit the mobility of that script to prevent damage to critical system files. This is particularly true of Setuid and Setgid scripts, as well as servers requiring root permissions for at least part of their lifetimes.

Perl allows the view of the local file system to be limited through the chroot command[10]. This command takes a single scalar argument, which becomes the new root directory for the process or thread. The process is irreversible, so once the file system is limited to the view chosen, a cracker cannot use the script to access sensitive areas of the file system.

A certain precaution must be used, however, when including external Perl libraries[10]. These libraries must either be found within the new root directory, or loaded before chroot is called.

Incident Prevention: User Authentication

User authentication is problem common to client user programs. A good first step towards securing authentication is to encrypt passwords with the strongest hashing mechanism available for your platform. The built in crypt() function will interface with the C library function of the same name on your platform[20]. Once that is done, the hashed password should be stored as privileged information, either within a root owned file, or, more commonly for web applications, a limited access database table.

A more interesting problem is that of user authentication over an HTTP connection. CPAN offers several modules for basic authentication, most of which are contained in the Library for WWW Programming (LWP) [21]. These include LWP LWP::Authen::Basic, LWP::Authen::DigesT, and LWP::Authen::Ntlm. However, as with any authentication, it is best to perform these operations in a secure environment. This typically involves Secure Socket Layers (SSL); of which LWP include an interface to. If this is not sufficient for your needs, CPAN has a slew of SSL modules available[22].

Incident Prevention: Ask a Guru

Finally, when all else fails, rely on the socially conscious culture of Perl. Perl programmers tend to enjoy solving a challenging Perl problem. As such, most are willing to help a fellow Perl hacker in need. If there is no local guru at your disposal (or they claim to be too busy), there are several places to search for Perl wisdom, including, but not limited to, a large number of Perl mailing lists[23], nntp news lists[24], Perl Monks[25], and your local Perl Mongers user group[26]. When asking for advice from others, remember to use examples, not specific vulnerabilities on your system. You do not want to reveal sensitive information to a potential cracker.

Works Cited

[1] Perl.org, "The Perl Directory at Perl.org." http://www.ruby-lang.org/

[2] wall.org, "Larry Wall's Very Own Home." http://www.wall.org/~larry/

[3] whatis.com, "Rapid Application Development." http://whatis.techtarget.com/definition/0,,sid9_gci214246,00.html

[4] Patwardhan, N., Siever, E., and Spainhour, S. "Perl in a Nutshell, 2nd Edition." O'Reilly, Cambridge: 2002.

[5] w3.org, "CGI – Common Gateway Interface." http://www.w3.org/CGI/

[6] cpan.org, "Lincoln D. Stein / CGI.pm." http://search.cpan.org/~lds/CGI.pm/

[7] Guelich, S. Gundavaram, S., and Gunther, B. "CGI programming With Perl, 2nd Edition." O'Reilly, Cambridge: 2000.

[8] Musciano, C. and Kennedy, B. "HTML and XHTML, The Definitive Guide, 5th Edition." O'Reilly, Cambridge: 2002

[9] Dhanjani, N. "Web App Security Testing with a Custom Proxy Server" http://www.onlamp.com/pub/au/1714

[10] Stein, L. D. "Network Programming with Perl." Addison-Wesley, New York: 2001.

[11] cpan.org, "Tim Bunce / DBI." http://search.cpan.org/~timb/DBI/

[12] Descartes, A. and Bunce, T. "Programming the Perl DBI." O'Reilly, Cambridge: 2000

[13] Jepson, B., "Beware SQL Injection in Web Applications." http://www.oreillynet.com/pub/wlg/1595

[14] CPAN, "Comprehensive Perl Archive Network." http://www.cpan.org/

[15] perldoc.com, "perlrun - how to execute the Perl interpreter." http://www.perldoc.com/perl5.8.4/pod/perlrun.html

[16] perldoc.com, "strict - Perl pragma to restrict unsafe constructs." http://www.perldoc.com/perl5.8.4/lib/strict.html

[17] perldoc.com, "perlsec - Perl security." http://www.perldoc.com/perl5.8.4/pod/perlsec.html

[18] Hall, J. N., with Schwartz, R. L. “Effective Perl Programming: Writing Better Programs with Perl.” Addison-Wesley, New York: 1998.

[19] Ahmed, S. J., “Securely Programming in C.” http://www.sans.org/rr/papers/index.php?id=388

[20] Perldoc.com, “crypt.” http://www.perldoc.com/perl5.8.4/pod/func/crypt.html

[21] cpan.org, “Gisle Aas / libwww-perl.” http://search.cpan.org/~gaas/libwww-perl/

[22] cpan.org “The CPAN Search Site.” http://search.cpan.org/search?query=ssl&mode=all

[23] Perl.org, “The Perl Mailing List Database.” http://lists.perl.org/

[24] Perl.org, “perl.org lists.” http://www.nntp.perl.org/group/

[25] Perl Monks, “Seekers of Perl Wisdom.” http://www.perlmonks.org/index.pl?node=Seekers%20of%20Perl%20Wisdom

[26] Perl Mongers, “Perl Mongers: User groups.” http://www.pm.org/groups/index.html

Last Updated: May 28th, 2017

Upcoming Training

SANS Stockholm 2017	Stockholm, Sweden	May 29, 2017 - Jun 03, 2017	Live Event
Security Operations Center Summit & Training	Washington, DC	Jun 05, 2017 - Jun 12, 2017	Live Event
SANS Houston 2017	Houston, TX	Jun 05, 2017 - Jun 10, 2017	Live Event
Community SANS Ottawa SEC401	Ottawa, ON	Jun 05, 2017 - Jun 10, 2017	Community SANS
SANS San Francisco Summer 2017	San Francisco, CA	Jun 05, 2017 - Jun 10, 2017	Live Event
SANS Charlotte 2017	Charlotte, NC	Jun 12, 2017 - Jun 17, 2017	Live Event
SANS Rocky Mountain 2017 - SEC401: Security Essentials Bootcamp Style	Denver, CO	Jun 12, 2017 - Jun 17, 2017	vLive
Community SANS Portland SEC401	Portland, OR	Jun 12, 2017 - Jun 17, 2017	Community SANS
SANS Secure Europe 2017	Amsterdam, Netherlands	Jun 12, 2017 - Jun 20, 2017	Live Event
SANS Rocky Mountain 2017	Denver, CO	Jun 12, 2017 - Jun 17, 2017	Live Event
SANS Minneapolis 2017	Minneapolis, MN	Jun 19, 2017 - Jun 24, 2017	Live Event
SANS Paris 2017	Paris, France	Jun 26, 2017 - Jul 01, 2017	Live Event
SANS Columbia, MD 2017	Columbia, MD	Jun 26, 2017 - Jul 01, 2017	Live Event
SANS Cyber Defence Canberra 2017	Canberra, Australia	Jun 26, 2017 - Jul 08, 2017	Live Event
SANS London July 2017	London, United Kingdom	Jul 03, 2017 - Jul 08, 2017	Live Event
Cyber Defence Japan 2017	Tokyo, Japan	Jul 05, 2017 - Jul 15, 2017	Live Event
SANS Cyber Defence Singapore 2017	Singapore, Singapore	Jul 10, 2017 - Jul 15, 2017	Live Event
Community SANS Minneapolis SEC401	Minneapolis, MN	Jul 10, 2017 - Jul 15, 2017	Community SANS
SANS Los Angeles - Long Beach 2017	Long Beach, CA	Jul 10, 2017 - Jul 15, 2017	Live Event
Community SANS Phoenix SEC401	Phoenix, AZ	Jul 10, 2017 - Jul 15, 2017	Community SANS
SANS Munich Summer 2017	Munich, Germany	Jul 10, 2017 - Jul 15, 2017	Live Event
Mentor Session - SEC401	Ventura, CA	Jul 12, 2017 - Sep 13, 2017	Mentor
Mentor Session - SEC401	Macon, GA	Jul 12, 2017 - Aug 23, 2017	Mentor
Community SANS Colorado Springs SEC401	Colorado Springs, CO	Jul 17, 2017 - Jul 22, 2017	Community SANS
Community SANS Atlanta SEC401	Atlanta, GA	Jul 17, 2017 - Jul 22, 2017	Community SANS
SANSFIRE 2017	Washington, DC	Jul 22, 2017 - Jul 29, 2017	Live Event
SANSFIRE 2017 - SEC401: Security Essentials Bootcamp Style	Washington, DC	Jul 24, 2017 - Jul 29, 2017	vLive
Community SANS Charleston SEC401	Charleston, SC	Jul 24, 2017 - Jul 29, 2017	Community SANS
Community SANS Fort Lauderdale SEC401	Fort Lauderdale, FL	Jul 31, 2017 - Aug 05, 2017	Community SANS
SANS San Antonio 2017	San Antonio, TX	Aug 06, 2017 - Aug 11, 2017	Live Event
SANS Prague 2017	Prague, Czech Republic	Aug 07, 2017 - Aug 12, 2017	Live Event