Hands-On Ethical Hacking and Network Defense, Edition 4

Chapter 7: Programming for Security Professionals

Module Objectives

- By the end of this module, you should be able to:
  - Explain basic programming concepts
  - Write a simple C program
  - Explain how webpages are created with HTML
  - Describe and create basic Perl programs
  - Explain basic object-oriented programming concepts
  - Describe and create basic Python programs

Introduction to Computer Programming

- Computer programmers:
  - Must understand rules of programming languages
  - Deal with syntax errors
- One minor mistake and the program will not run
  - Or worse, it will produce unpredictable results
- Being a good programmer takes time and patience

Programming Fundamentals

- You can begin writing programs with little knowledge of programming fundamentals
- Fundamental concepts
  - Acronym BLT
    - Branching
    - Looping
    - Testing

Branching, Looping, and Testing (BLT) (1 of 4)

- **Function**
  - Mini program within a main program
    - Carries out a task
- **Branching**
  - Takes you from one program area to another
- **Looping**
  - Performing a task over and over
  - Completes after testing is conducted on a variable and returns a value of true or false
- **Testing**
  - Verifies some condition
    - Returns a value of true or false

Branching, Looping, and Testing (BLT) (2 of 4)

```
#include <stdio.h>
main()
{
    int a = 1 /* Variable initialized as an integer, value 1
    if (a > 2) ; //Testing whether "a" is greater than 2
     printf("a is greater than 2");
    else
     GetOut(); //* Branching: calling a different function
    GetOut() // Do something interesting here
    {
       for(int a=1; a<11; a++) // Loop to display 10 times
       {
         printf("I'm in the GetOut() function");
       }
    }
}
```

Branching, Looping, and Testing (BLT) (3 of 4)

- **Algorithm**
  - Defines the steps for performing a task
    - Keep it as simple as possible
    - Skipping a step can cause problems
- **Bug**
  - An error that causes unpredictable results
- **Pseudocode**
  - English-like language
  - Used to create the structure of your program

Branching, Looping, and Testing (BLT) (4 of 4)

- Documenting your work is essential
  - Add comments to your code
    - Should explain what you are doing
  - Many programmers find it time-consuming and tedious
  - It helps others understand your work
- Industry standard
  - 10 bugs for every 1,000 lines of code
- Windows 10 contains over 50 million lines of code
  - Fewer bugs than the average

Learning the C Language (1 of 3)

- Developed by Dennis Ritchie
  - 1972, Bell Laboratories
  - Powerful and concise language
- UNIX
  - First written in **assembly language**
  - Later rewritten in C language
    - Assembly language uses a combination of hexadecimal numbers and expressions

Learning the C Language (2 of 3)

- C++
  - An enhancement of C language
- **Compiler**
  - Program that converts text-based program, called source code, into executable or binary code
  - Most C compilers can also create executable programs in C++

Learning the C Language (3 of 3)

Compiler	Description
Intel compilers for Windows and Linux	Intel’s C++ compiler is designed for developing applications for Windows servers, desktops, laptops, and mobile devices. The Intel Linux C++ compiler claims to optimize the speed of accessing information from a MySQL database, an open-source database program used by many corporations and e-commerce companies.
Microsoft Visual C++ Compiler	This compiler is widely used by programmers developing C and C++ applications for Windows platforms.
GNU C and C++ compilers (GCC)	These free compilers can be downloaded for Windows and *nix platforms. Most *nix systems include the GNU GCC compiler.

Anatomy of a C Program (1 of 3)

- The first computer program a C student learns:

```
/* The famous "Hello, world!" C program */

 #include <stdio.h>/* Load the standard IO library. The library
contains functions your C program might need to call to perform
various tasks. */

 main()
 {
   printf("Hello, world!\n\n");
 }
```

Anatomy of a C Program (2 of 3)

- Many C programs use the /* and */ symbols to enclose long comments
  - Instead of // for one-line comments
- `#include statement`
  - Loads libraries that hold commands and functions used in your program
- Parentheses in C
  - Mean you are dealing with function
- `main() function`
  - Required by every C program
  - Can also add your own functions to a C program

Anatomy of a C Program (3 of 3)

- Braces
  - Show where a block of code begins and ends
- Functions
  - When a function calls other functions, it uses parameters (known as arguments)
  - Parameters are placed between opening and closing parentheses

Character	Description
\n	New line
\t	Tab

Declaring Variables

- Variable
  - Represents a numeric or string value
  - Can be declared at the beginning of a program
    - To ensure that calculations can be carried out without user intervention
  - Defined as a character or characters
- **Conversion specifiers**
  - Tell the compiler how to convert the values in a function
- Operators
  - Programmers use them to compare values, perform mathematical calculations, and the like
  - Mostly, programs you write will require calculating values based on mathematical operations

Variable Types in C

Variable type	Description
Int	Use this variable type for an integer (positive or negative number).
Float	This variable type is for a real number that includes a decimal point, such as 1.299999.
Double	Use this variable type for a double-precision floating-point number.
Char	This variable type holds the value of a single letter.
String	This variable type holds the value of multiple characters or words.
Const	A constant variable is created to hold a value that doesn’t change for the duration of your program. For example, you can create a constant variable called TAX and give it a specific value: const TAX =.085. If this variable is used in areas of the program that calculate total costs after adding an 8.5% tax, it’s easier to change the constant value to a different number if the tax rate changes, instead of changing every occurrence of 8.5% to 8.6%.

Conversion Specifiers in C

Specifier	Type
%c	Character
%d	Decimal number
%f	Floating decimal or double number
%s	Character string

Mathematical Operators in C

Operator	Description
+ (unary)	Doesn’t change the value of the number. Unary operators use a single argument; binary operators use two arguments. Example: +(2).
- (unary)	Returns the negative value of a single number.
++ (unary)	Increments the unary value by 1. For example, if a is equal to 5, ++a changes the value to 6.
— (unary)	Decrements the unary value by 1. For example, if a is equal to 5, —a changes the value to 4.
+ (binary)	Addition. For example, a + b.
- (binary)	Subtraction. For example, a – b.
* (binary)	Multiplication. For example, a * b.
/ (binary)	Division. For example, a / b.
% (binary)	Modulus. For example, 10 % 3 is equal to 1 because 10 divided by 3 leaves a remainder of 1.

Relational and Logical Operators in C

Operator	Description
==	Equal operator; compares the equality of two variables. In a == b, for example, the condition is true if variable a is equal to variable b.
! =	Not equal; the exclamation mark negates the equal sign. For example, the statement `if a != b` is read as “if a is not equal to b.”
>	Greater than.
<	Less than.
>=	Greater than or equal to.
<=	Less than or equal to.
&&	AND operator; evaluates as true if both sides of the operator are true. For example, `if (( a > 5) && (b > 5)) printf ("Hello, world!");` prints only if both a and b are greater than 5.
| |	OR operator; evaluates as true if either side of the operator is true.
!	NOT operator; the statement `! (a == b)`, for example, evaluates as true if a isn’t equal to b.

Branching, Looping, and Testing in C (1 of 6)

- Branching

```
main()
{
   prompt(); //Call function to prompt user with a question
   display(); //Call function to display graphics onscreen
   calculate(); //Call function to do complicated math
   cleanup(); //Call function to make all variables equal to
              //zero
```

Branching, Looping, and Testing in C (2 of 6)

- Branching (continued)

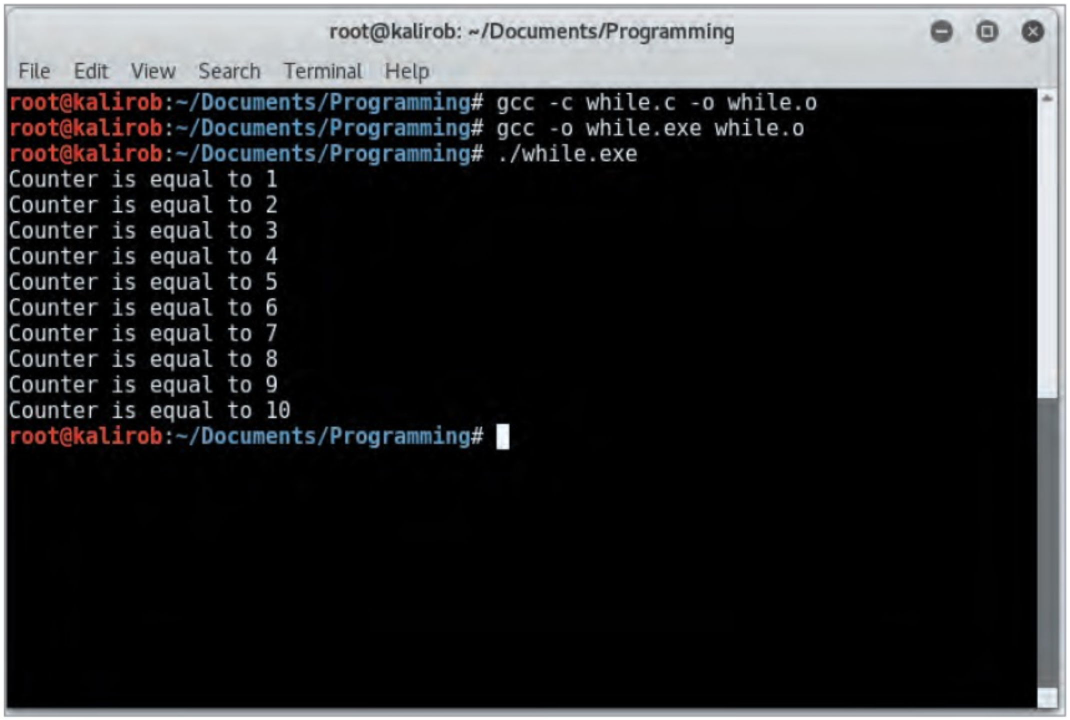
```
  prompt()
  {
    [code for prompt() function goes here]
  }
  display()
  {
    [code for display() function goes here]
  }
  [and so forth]

}
```

Branching, Looping, and Testing in C (3 of 6)

- **While loop**

```
main()
{
   int counter = 1;        //Initialize (assign a value to)
                     //the counter variable
   while (counter <= 10) //Do what's in the brackets until false
   {
      printf("Counter is equal to %d\n", counter);
      ++counter; //Increment counter by 1;
   }
}
```

Branching, Looping, and Testing in C (4 of 6)



root@kalirob: ~/Documents/Programming
File Edit View Search Terminal Help

```
root@kalirob:~/Documents/Programming# gcc -c while.c -o while.o
root@kalirob:~/Documents/Programming# gcc -o while.exe while.o
root@kalirob:~/Documents/Programming# ./while.exe
Counter is equal to 1
Counter is equal to 2
Counter is equal to 3
Counter is equal to 4
Counter is equal to 5
Counter is equal to 6
Counter is equal to 7
Counter is equal to 8
Counter is equal to 9
Counter is equal to 10
root@kalirob:~/Documents/Programming# 
```

Source: Kali Linux

Figure 7-1 A while loop in action

Branching, Looping, and Testing in C (5 of 6)

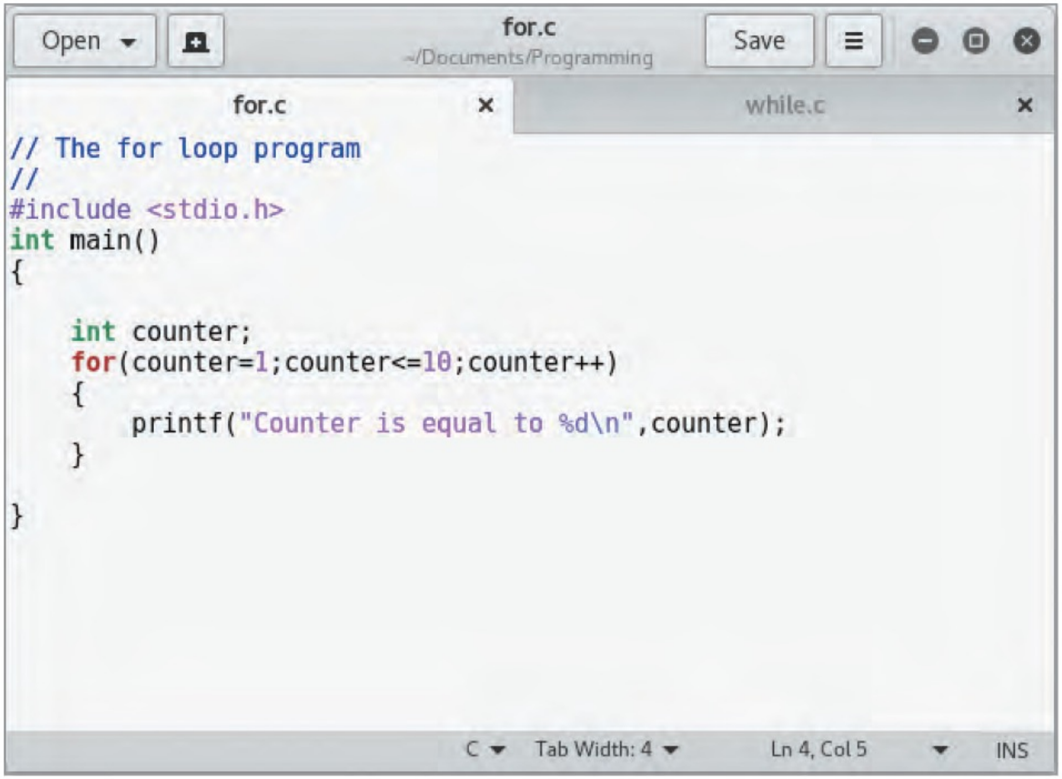
- `do` loop

```
main()
 {
    int counter = 1; //Initialize counter variable
    do
    {
       printf("Counter is equal to %d\n", counter);
       ++counter; //Increment counter by 1
    } while (counter <= 10); //Do what's in the brackets
                              // until false
 }
```

- `for` loop

```
for (counter = 1;counter <= 10;counter++)
```

Branching, Looping, and Testing in C (6 of 6)



```
// The for loop program
//
#include <stdio.h>
int main()
{

    int counter;
    for(counter=1;counter<=10;counter++)
    {
        printf("Counter is equal to %d\n",counter);
    }

}
```

Source: Kali Linux

Figure 7-2 A for loop

Understanding HTML Basics

- HTML
  - Markup language
  - Used mainly for webpage formatting and layout
  - Basic HTML syntax is the basis for web development
- Security professionals often need to:
  - Examine webpages
  - Recognize when something looks suspicious

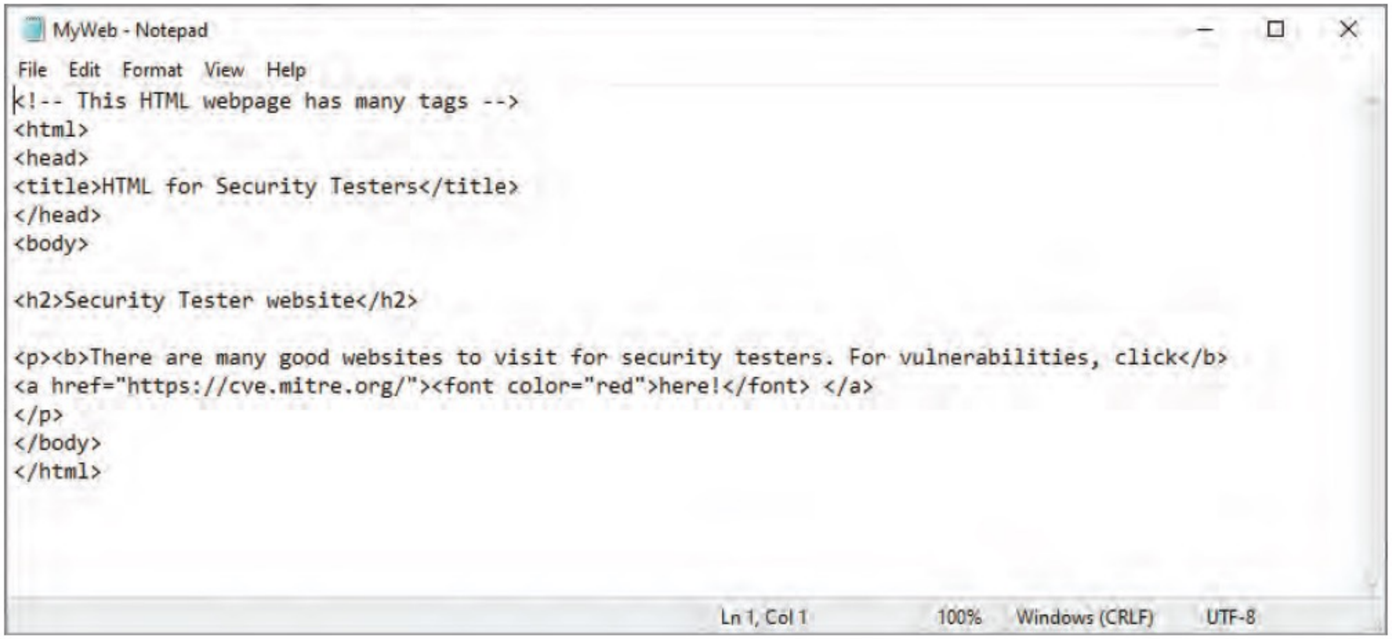
Creating a Webpage with HTML

- You can create an HTML webpage in Notepad
  - View it in a web browser
- HTML
  - Does not use branching, looping, or testing
- The < and > symbols
  - Denote HTML tags
  - Each tag has a matching closing tag that includes a forward slash
    - `<HTML> and </HTML>`

HTML Formatting Tags

Opening tag	Closing tag	Description
<h1>, <h2>, <h3>, <h4>, <h5>, and <h6>	<h1>, <h2>, <h3>, </h4>, </h5>, and </h6>	Formats text as different heading levels. Level 1 is the largest font size, and level 6 is the smallest.
<p>	</p>	Marks the beginning and end of a paragraph.
<b>	</b>	Formats enclosed text in bold.
<i>	</i>	Formats enclosed text in italics.

Creating a Webpage with HTML (1 of 2)

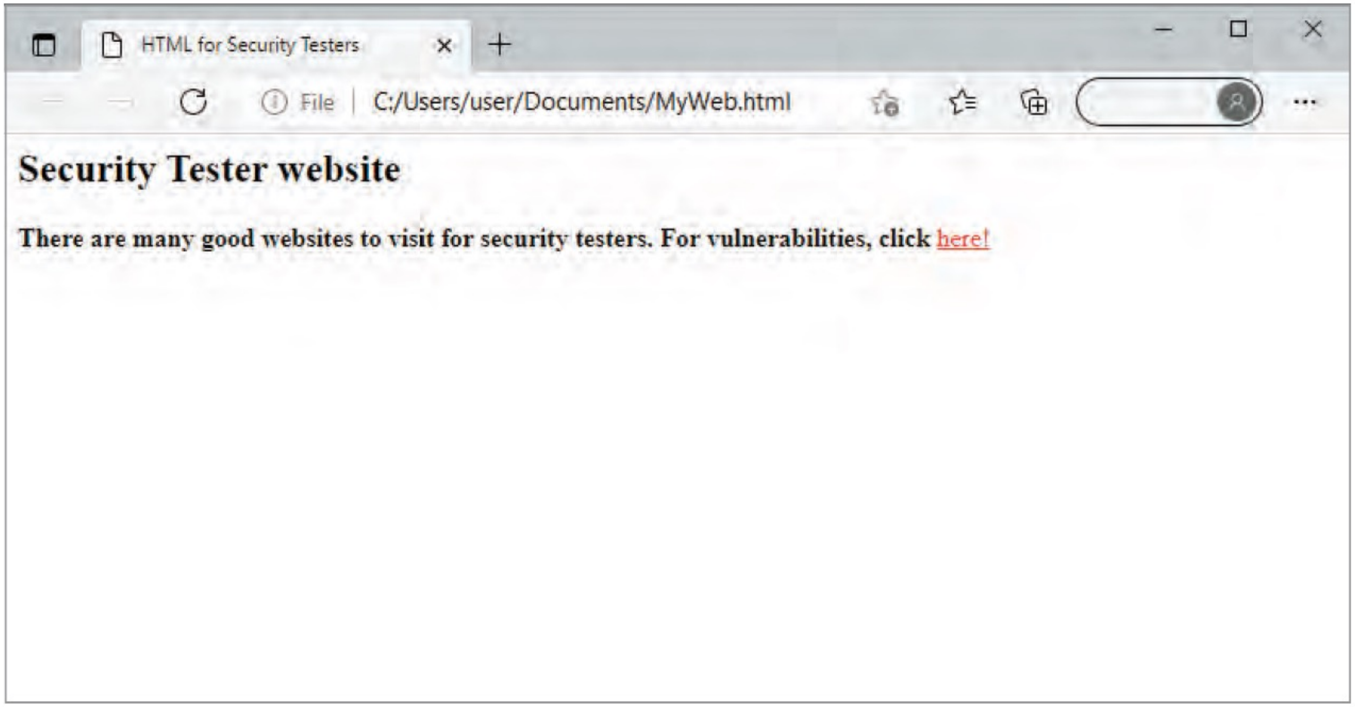


```
<!-- This HTML webpage has many tags -->
<html>
<head>
<title>HTML for Security Testers</title>
</head>
<body>

<h2>Security Tester website</h2>

<p><b>There are many good websites to visit for security testers. For vulnerabilities, click</b>
<a href="https://cve.mitre.org/"><font color="red">here!</font> </a>
</p>
</body>
</html>
```

Source: Microsoft Windows Notepad

Figure 7-4 HTML source code

Creating a Webpage with HTML (2 of 2)



Security Tester website

There are many good websites to visit for security testers. For vulnerabilities, click here!

Source: Microsoft Windows

Figure 7-5 HTML webpage

Understanding Perl

- Practical Extraction and Report Language (Perl)
    - Used to write scripts and programs for security professionals
    - Powerful scripting language
    - Perl and Python are two very popular languages for security professionals

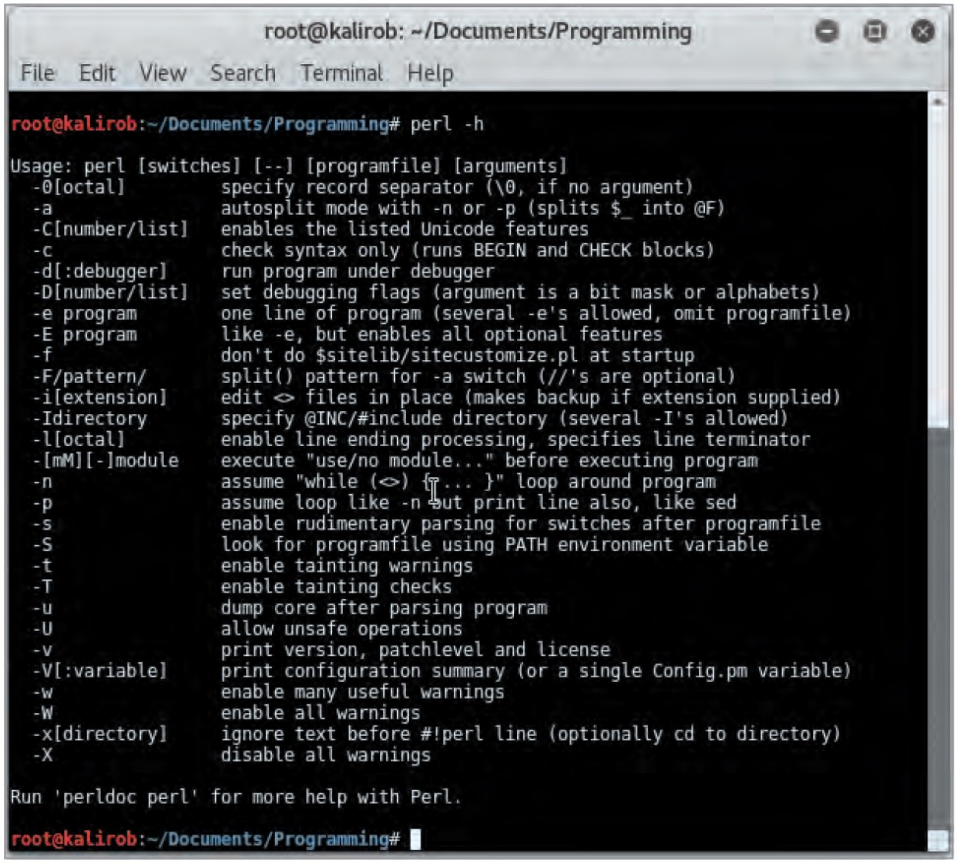
Background on Perl

- Developed by Larry Wall in 1987
- Can run on almost any platform
  - *nix-based OSs already have Perl installed
- Perl syntax is similar to C
- Hackers use Perl to create automated exploits and malicious bots
- Security professionals use Perl to perform repetitive tasks and conduct security monitoring

Understanding the Basics of Perl (1 of 3)

- The `perl -h` command
  - Gives a list of parameters used with `perl` command
- Perl has a `printf` command for formatting complex variables

Understanding the Basics of Perl (2 of 3)


root@kalirob: ~/Documents/Programming
File Edit View Search Terminal Help

```
root@kalirob:~/Documents/Programming# perl -h

Usage: perl [switches] [--] [programfile] [arguments]
  -0[octal]         specify record separator (\0, if no argument)
  -a                autosplit mode with -n or -p (splits $_ into @F)
  -C[number/list]   enables the listed Unicode features
  -c                check syntax only (runs BEGIN and CHECK blocks)
  -d[:debugger]     run program under debugger
  -D[number/list]   set debugging flags (argument is a bit mask or alphabets)
  -e program        one line of program (several -e's allowed, omit programfile)
  -E program        like -e, but enables all optional features
  -f                don't do $sitelib/sitecustomize.pl at startup
  -F/pattern/       split() pattern for -a switch (//'s are optional)
  -i[extension]     edit <> files in place (makes backup if extension supplied)
  -Idirectory       specify @INC/#include directory (several -I's allowed)
  -l[octal]         enable line ending processing, specifies line terminator
  -[mM][-]module    execute "use/no module..." before executing program
  -n                assume "while (<>) { ... }" loop around program
  -p                assume loop like -n but print line also, like sed
  -s                enable rudimentary parsing for switches after programfile
  -S                look for programfile using PATH environment variable
  -t                enable tainting warnings
  -T                enable tainting checks
  -u                dump core after parsing program
  -U                allow unsafe operations
  -v                print version, patchlevel and license
  -V[:variable]     print configuration summary (or a single Config.pm variable)
  -w                enable many useful warnings
  -W                enable all warnings
  -x[directory]     ignore text before #!perl line (optionally cd to directory)
  -X                disable all warnings

Run 'perldoc perl' for more help with Perl.

root@kalirob:~/Documents/Programming#
```

Source: Kali Linux

Figure 7-8 Using the perl –h command

Understanding the Basics of Perl (3 of 3)

Formatting character	Description	Input	Output
%c	Character	printf ‘%c’ , "d"	d
%s	String	printf ‘%s’, "This is fun!"	This is fun!
%d	Signed integer in decimal	printf ‘%+d%d’, 1, 1	+1 1
%u	Unsigned integer in decimal	printf ‘%u’, 2	2
%o	Unsigned integer in octal	printf ‘%o’ ,8	10
%x	Unsigned integer in hexadecimal	printf ‘%x’, 10	a
%e	Floating-point number in scientific notation	printf ‘%e’ ,10;	1.000000e+001 (depending on the OS)
%f	Floating-point number in fixed decimal notation	printf ‘%f’ ,1;	1.000000

Understanding the BLT of Perl

- Some syntax rules to keep in mind:
  - The `sub` keyword is used before function names
  - Variables begin with the $ symbol
  - Comment lines begin with the # symbol
  - The & symbol indicates a function
- Except for these minor differences, Perl’s syntax is much like the C syntax

Branching in Perl

- To go from one function to another, you call the function by entering its name in your source code

```
# Perl program illustrating the branching function
# Documentation is important
# Initialize variables
$first_name = "Jimi";
$last_name = "Hendrix";
&name_best_guitarist;
sub name_best_guitarist
{
 printf "%s %s %s", $first_name, $last_name, "was the best!";
}
```

Looping in Perl

- `for` loop

```
for ($a = 1; $a <= 10; $a++)
{
  print "Hello security testers!\n"
}
```

- `while` loop

```
$a = 1;
while ($a <=10)
{
    print "Hello security testers!\n";
    $a++
}
```

Testing Conditions in Perl (1 of 3)

- Most programs must be able to test the value of a variable or condition
- The two looping examples shown previously use the less than or equal to operator (<=)
- Other operators used for testing in Perl are similar to C operators
  - Often you combine these operators with Perl conditionals, such as the following:
  - if—Checks whether a condition is true

```
if (($age < 12) {
 print "You must be a know-it-all!";
}
```

Testing Conditions in Perl (2 of 3)

- else—Used when there are several conditionals to test

```
elsif ($age > 39)
{
  print "You must lie about your age!";
}
else
{
  print "To be young...";
}
```

Testing Conditions in Perl (3 of 3)

- unless—Executes unless the condition is true

```
unless ($age == 100)
{
  print "Still enough time to get a bachelor's degree.";
}
```

Perl Operators (1 of 3)

Operator	Function	Example
+	Addition	$total = $sal + $commission
–	Subtraction	$profit = $gross sales – $cost of goods
*	Multiplication	$total = $cost * $quantity
/	Division	$GPA = $total_points / $number of classes
%	Modulus	$a % 10 = 1
**	Exponent	$total = $a**10

Perl Operators (2 of 3)

Assignments	Function	Example
=	Assignment	$Last name = "Rivera"
+=	Add, then assignment	$a+ = 10; shorthand for $a=$a+10
-=	Subtract, then assignment	$a–=10; shorthand for $a=$a–10
*=	Multiply, then assignment	$a* = 10; shorthand for $a=$a*10
/=	Divide, then assignment	$a/ = 10; shorthand for $a=$a/10
%=	Modulus, then assignment	$a%=10; shorthand for $a=$a%10
**=	Exponent and assignment	$a**=2; shorthand for $a=$a**2
++	Increment	$a++; increment $a by 1
—	Decrement	$a––; decrement $a by 1

Perl Operators (3 of 3)

Comparisons	Function	Example
==	Equal to	$a==1; compare value of $a with 1
!=	Not equal to	$a!=1; $a is not equal to 1
>	Greater than	$a>10
<	Less than	$a<10
>=	Greater than or equal to	$a<10
<=	Less than or equal to	$a<10

Understanding Object-Oriented Programming Concepts

- Technology
  - Changes frequently
- Object-oriented programming
  - Isn’t new to experienced programmers
  - Might not be familiar to those just learning how to write their first Perl script
  - Takes time and practice to learn

Components of Object-Oriented Programming (1 of 2)

- **Classes**
  - Structures that hold pieces of data and functions
- The `::` symbol
  - Used to separate the name of a class from a member function
  - Example: To access a member function, you use the class name followed by two colons and the member function’s name:
    - `Employee::GetEmp()`

Components of Object-Oriented Programming (2 of 2)

```
// This is a class called Employee created in C++
class Employee
{
public:
   char firstname[25];
   char lastname[25];
   char PlaceOfBirth[30];
   [code continues]
};
void GetEmp()
{
   // Perform tasks to get employee info
   [program code goes here]
}
```

Win32 API Functions (1 of 3)

Function	Description
`GetLastError()`	Returns the last error generated when a call was made to the Win32 API.
`OLELastError()`	Returns the last error generated by the object linking and embedding (OLE) API.
`BuildNumber()`	Returns the Perl build number.
`LoginName()`	Returns the username of the person running Perl.
`NodeName()`	Returns the NetBIOS computer name.
`DomainName()`	Returns the name of the domain the computer is a member of.
`FsType()`	Returns the name of the file system, such as NTFS or FAT.
`GetCwd()`	Returns the current active drive.
`SetCwd(newdir)`	Enables you to change to the drive designated by the newdir variable.
`GetOSName()`	Returns the OS name.
`FormatMessage(error)`	Converts the error message number into a descriptive string.

Win32 API Functions (2 of 3)

Function	Description
`Spawn(command, args, $pid)`	Starts a new process, using arguments supplied by the programmer and the process ID ($pid).
`LookupAccountSID(sys, sid, $acct, $domain, $type)`	Returns the account name, domain name, and security ID (SID) type.
`InitiateSystemShutdown(machine, message, timeout, forceclose, reboot)`	Shuts down a specified computer or server.
`AbortSystemShutdown(machine)`	Aborts the shutdown if it was done in error.
`GetTickCount()`	Returns the Win32 tick count (time elapsed since the system first started).
`ExpandEnvironmentalStrings (envstring)`	Returns the environmental variable strings specified in the envstring variable.
`GetShortPathName(longpathname)`	Returns the 8.3 version of the long pathname. In DOS and older Windows programs, filenames could be only eight characters, with a three-character extension.

Win32 API Functions (3 of 3)

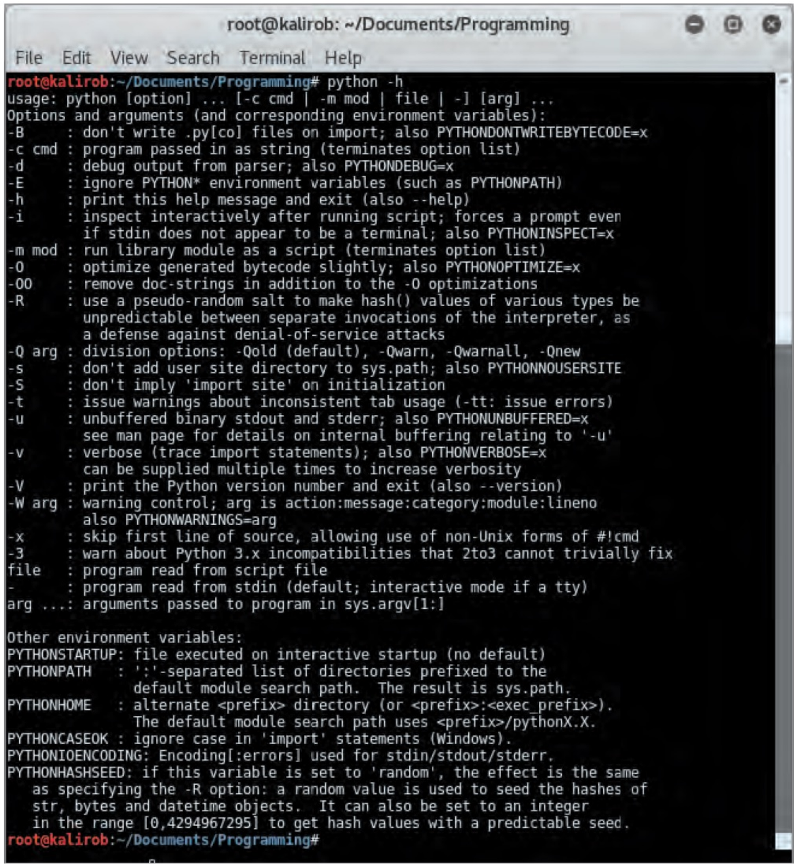
Function	Description
`GetNextAvailableDrive()`	Returns the next available drive letter.
`RegisterServer(libraryname)`	Loads the DLL specified by libraryname and calls the DLLRegisterServer() function.
`UnregisterServer(libraryname)`	Loads the DLL specified by libraryname and calls the DLLUnregisterServer() function.
`Sleep(time)`	Pauses the number of milliseconds specified by the time variable.

Python

- Scripting language with some object-oriented features
- Emphasizes code readability and uses indentation to define blocks of code
- Background
  - Guido van Rossum conceived of Python in the late 1980s
    - Python’s principal author
    - Continuing central figure in decisions regarding the direction of Python’s development
- Runs on almost any platform (including Windows), and *nix-based OSs usually have Python already installed

Understanding the Basics of Python

- Knowing how to get help quickly in any programming language is useful.
- The `python -h` command lists parameters used with the python command



root@kalirob: ~/Documents/Programming
File Edit View Search Terminal Help

```
root@kalirob:~/Documents/Programming# python -h
usage: python [option] ... [-c cmd | -m mod | file | -] [arg] ...
Options and arguments (and corresponding environment variables):
-B     : don't write .py[co] files on import; also PYTHONDONTWRITEBYTECODE=x
-c cmd : program passed in as string (terminates option list)
-d     : debug output from parser; also PYTHONDEBUG=x
-E     : ignore PYTHON* environment variables (such as PYTHONPATH)
-h     : print this help message and exit (also --help)
-i     : inspect interactively after running script; forces a prompt even
         if stdin does not appear to be a terminal; also PYTHONINSPECT=x
-m mod : run library module as a script (terminates option list)
-O     : optimize generated bytecode slightly; also PYTHONOPTIMIZE=x
-OO    : remove doc-strings in addition to the -O optimizations
-R     : use a pseudo-random salt to make hash() values of various types be
         unpredictable between separate invocations of the interpreter, as
         a defense against denial-of-service attacks
-Q arg : division options: -Qold (default), -Qwarn, -Qwarnall, -Qnew
-s     : don't add user site directory to sys.path; also PYTHONNOUSERSITE
-S     : don't imply 'import site' on initialization
-t     : issue warnings about inconsistent tab usage (-tt: issue errors)
-u     : unbuffered binary stdout and stderr; also PYTHONUNBUFFERED=x
         see man page for details on internal buffering relating to '-u'
-v     : verbose (trace import statements); also PYTHONVERBOSE=x
         can be supplied multiple times to increase verbosity
-V     : print the Python version number and exit (also --version)
-W arg : warning control; arg is action:message:category:module:lineno
         also PYTHONWARNINGS=arg
-x     : skip first line of source, allowing use of non-Unix forms of #!cmd
-3     : warn about Python 3.x incompatibilities that 2to3 cannot trivially fix
file   : program read from script file
-      : program read from stdin (default; interactive mode if a tty)
arg ...: arguments passed to program in sys.argv[1:]

Other environment variables:
PYTHONSTARTUP: file executed on interactive startup (no default)
PYTHONPATH   : ':'-separated list of directories prefixed to the
               default module search path.  The result is sys.path.
PYTHONHOME   : alternate <prefix> directory (or <prefix>:<exec_prefix>).
               The default module search path uses <prefix>/pythonX.X.
PYTHONCASEOK : ignore case in 'import' statements (Windows).
PYTHONIOENCODING: Encoding[:errors] used for stdin/stdout/stderr.
PYTHONHASHSEED: if this variable is set to 'random', the effect is the same
   as specifying the -R option: a random value is used to seed the hashes of
   str, bytes and datetime objects.  It can also be set to an integer
   in the range [0,4294967295] to get hash values with a predictable seed.
root@kalirob:~/Documents/Programming#
```

Source: Kali Linux

Understanding the BLT of Python (1 of 4)

- Syntax rules to keep in mind:
  - Spacing is important
  - When creating a function, insert the `def` keyword in front of the function’s name
  - Variables do not begin with any special symbol
  - There are no special characters at the end of lines of code
  - Comment lines begin with the # symbol
- Branching
  - To go from one function to another in a Python program, you call the function by entering its name followed by parentheses

Understanding the BLT of Python (2 of 4)

- The `name_best_guitarist()` line branches the program to the `name_best_guitarist()` function in the following Python program:

```
# Python program illustrating the branching function
# Documentation is important

# Initialize variables
first_name = "Jimi "
last_name = "Hendrix"
```

Understanding the BLT of Python (3 of 4)

(continued)

```
# define the name_best_guitarist function
# a function must be defined before it can be called
def name_best_guitarist():
   print(first_name + last_name + " was the best!")

name_best_guitarist()
```

Understanding the BLT of Python (4 of 4)

- Looping in Python
  - The Python `for` loop repeats until it has gone through each item specified in a list of items

```
names = ["Bob", "Jamal", "Sasha"]
for x in names:
    print(x)
```

- The `while` loop repeats a set of code lines as long as a test condition remains true
  - Do not need `brackets` in a `while` loop

```
i = 1
while i < 6:
   print(i)
   i += 1
```

Python Operators (1 of 2)

Operator	Function	Example
+	Addition	`total = sal + commission`
–	Subtraction	`profit = grossSales – costOfGoods`
*	Multiplication	`total = cost * quantity`
/	Division	`GPA = totalPoints / numberOfClasses`
%	Modulus	`x = a % 2`
**	Exponent	`area = 3.14 * (r**2)`
Assignments	**Function**	**Example**
=	Assignment	`lastName = "Rivera"`
+=	Add, then assignment	`a+ = 10 #shorthand for a=a+10`
-=	Subtract, then assignment	`a-=10 #shorthand for a=a-10`
*=	Multiply, then assignment	`a* = 10 #shorthand for a=a* 10`

Python Operators (2 of 2)

Operator	Function	Example
/=	Divide, then assignment	`a/ = 10 #shorthand for a=a/10`
%=	Modulus, then assignment	`a%=10 #shorthand for a=a%10`
**=	Exponent and assignment	`a**=2 #shorthand for a=a**2`
++	Increment	`GPA = totalPoints / numberOfClasses`
%	Modulus	`a++ #increment a by 1`
—	Decrement	`a-- #decrement a by 1`
Comparisons	**Function**	**Example**
==	Equal to	`a== 1 #compare value of a with 1`
!=	Not equal to	`a!=1 #a is not equal to 1`
>	Greater than	`a>10`
<	Less than	`a<10`
>=	Greater than or equal to	`a>=10`
<=	Less than or equal to	`a<=10`

If Statements and Logical Operators (1 of 3)

- “If statement” combines logical operators with variables and numbers to create conditional checks
  - You can combine an "if statement" with the keywords `else` and `elseif`
- if—Checks whether a condition is true

```
    if (age < 12)
      print("You must be a know-it-all!")
```

- else—Used when there’s only one option to carry out if the condition is not true

```
if (age > 12)
   print("You must be a know-it-all!")
else
   print("Sorry, but I don’t know why the sky is blue.")
```

If Statements and Logical Operators (2 of 3)

- elif—Used when there are several conditionals to test

```
if ( (age > 12) && (age < 20) )
   print("You must be a know-it-all!")
elif (age > 39)
   print("You must lie about your age!")
   else:
      print("To be young...")
```

If Statements and Logical Operators (3 of 3)

- Nested ifs: When you can include if statements inside other if statements

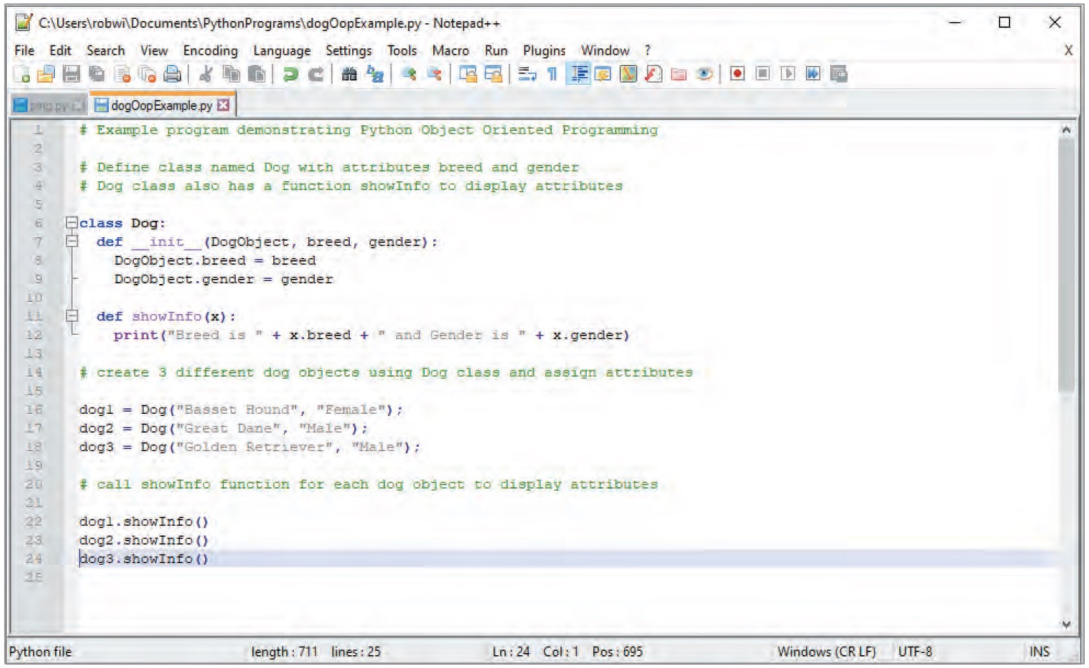
```
y = 69
if y > 10:
    print("Greater than ten")
    if y > 20:
        print("Also greater than 20!")
    else:
        print("But not greater than 20")
```

Python Shell (R E P L)

- An interactive shell where you can enter Python commands and have them immediately executed
- Known as the R E P L
  - Stands for Read, Evaluate, Print, Loop
- Reads a command, evaluates the command, prints the results, and loops back to read more commands
- You can enter the shell by typing `python` and pressing Enter in a terminal or command window

Object-Oriented Programming in Python

- Python supports traditional OOP concepts such as classes, objects, and inheritance



C:\Users\robwi\Documents\PythonPrograms\dogOopExample.py - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
ping.py
dogOopExample.py

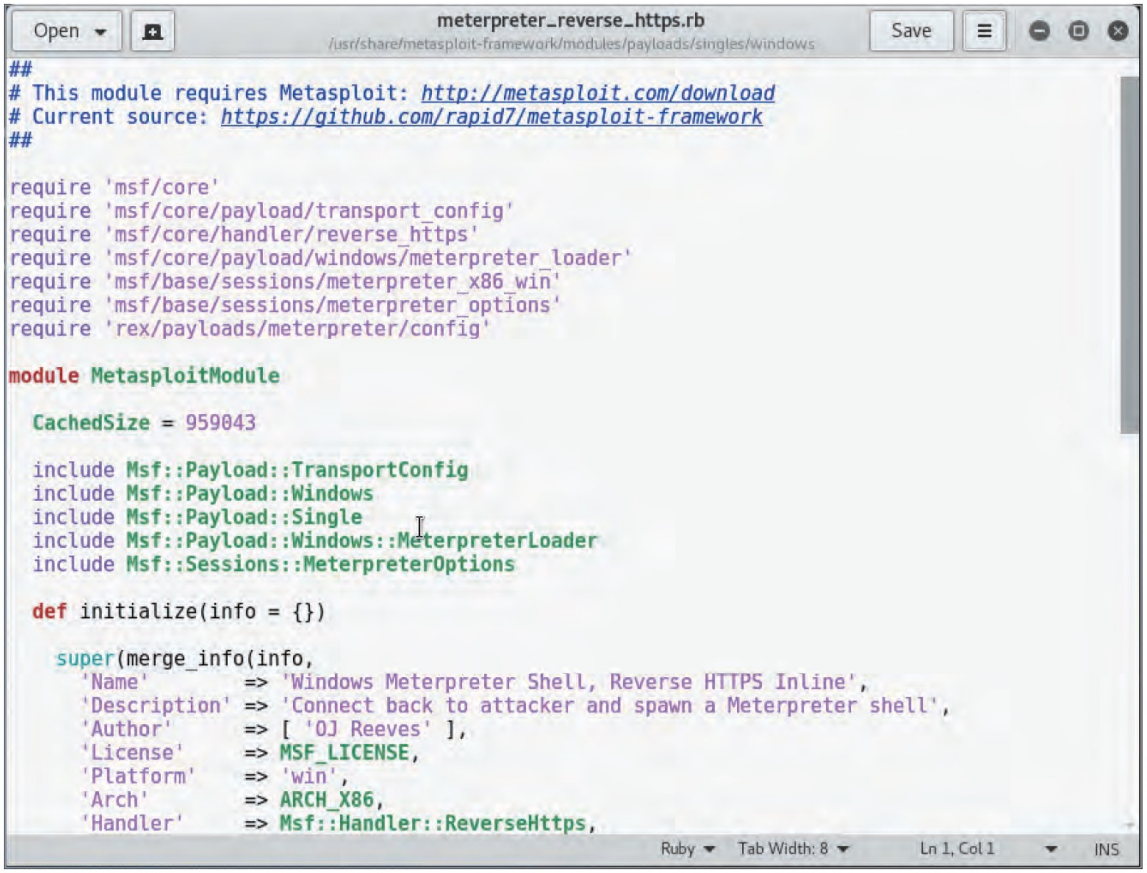
```
# Example program demonstrating Python Object Oriented Programming

# Define class named Dog with attributes breed and gender
# Dog class also has a function showInfo to display attributes

class Dog:
  def __init__(DogObject, breed, gender):
    DogObject.breed = breed
    DogObject.gender = gender

  def showInfo(x):
    print("Breed is " + x.breed + " and Gender is " + x.gender)

# create 3 different dog objects using Dog class and assign attributes

dog1 = Dog("Basset Hound", "Female");
dog2 = Dog("Great Dane", "Male");
dog3 = Dog("Golden Retriever", "Male");

# call showInfo function for each dog object to display attributes

dog1.showInfo()
dog2.showInfo()
dog3.showInfo()
```

Source: Kali Linux

An Overview of Ruby (1 of 3)

- Ruby
  - An object-oriented language used by many security testers
  - Similar to Perl
- Metasploit
  - A Ruby-based program used by security testers
    - To check for vulnerabilities on computer systems
  - Security testers should understand the basics of Ruby
    - Be able to modify Ruby code

An Overview of Ruby (2 of 3)


Open
meterpreter_reverse_https.rb
/usr/share/metasploit-framework/modules/payloads/singles/windows
Save

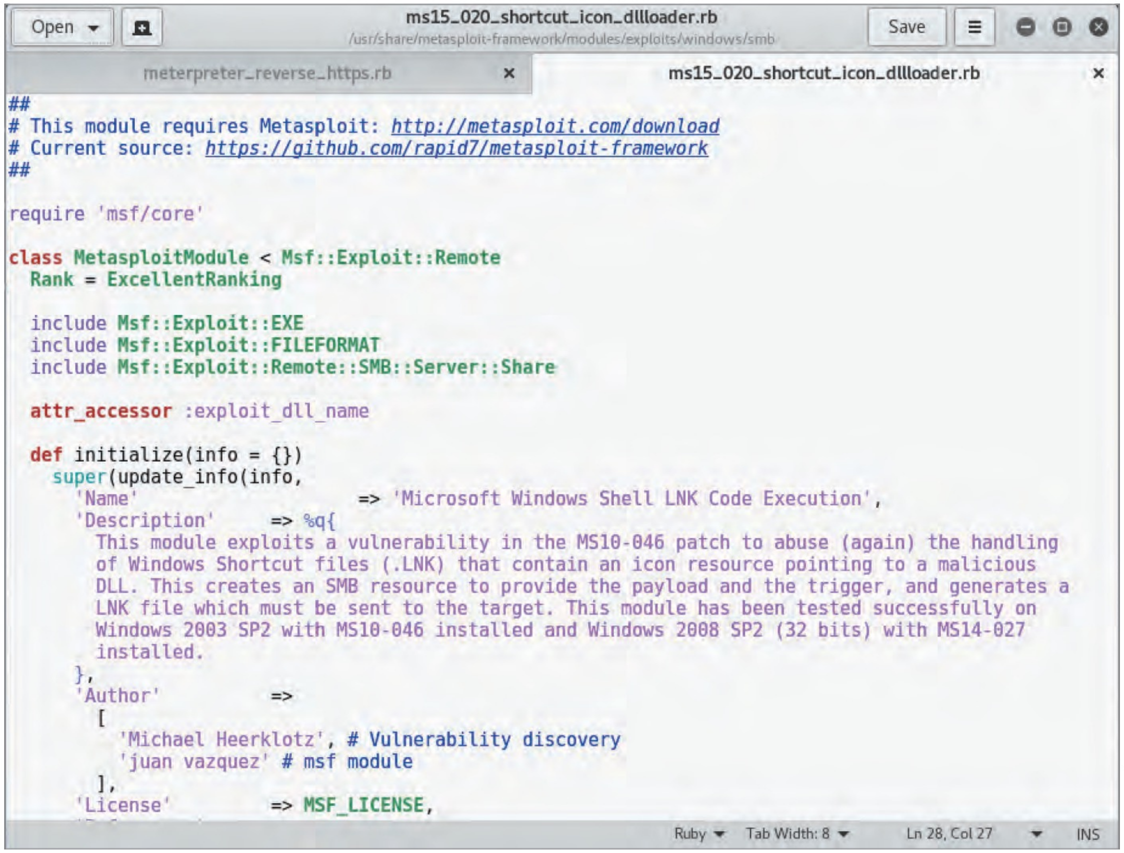
```ruby
##
# This module requires Metasploit: http://metasploit.com/download
# Current source: https://github.com/rapid7/metasploit-framework
##

require 'msf/core'
require 'msf/core/payload/transport_config'
require 'msf/core/handler/reverse_https'
require 'msf/core/payload/windows/meterpreter_loader'
require 'msf/base/sessions/meterpreter_x86_win'
require 'msf/base/sessions/meterpreter_options'
require 'rex/payloads/meterpreter/config'

module MetasploitModule

  CachedSize = 959043

  include Msf::Payload::TransportConfig
  include Msf::Payload::Windows
  include Msf::Payload::Single
  include Msf::Payload::Windows::MeterpreterLoader
  include Msf::Sessions::MeterpreterOptions

  def initialize(info = {})

    super(merge_info(info,
      'Name'          => 'Windows Meterpreter Shell, Reverse HTTPS Inline',
      'Description'   => 'Connect back to attacker and spawn a Meterpreter shell',
      'Author'        => [ 'OJ Reeves' ],
      'License'       => MSF_LICENSE,
      'Platform'      => 'win',
      'Arch'          => ARCH_X86,
      'Handler'       => Msf::Handler::ReverseHttps,
```

Ruby Tab Width: 8 Ln 1, Col 1 INS

Source: Kali Linux

Figure 7-23 Modifying reverse shell payload code in Ruby

An Overview of Ruby (3 of 3)


Open
ms15_020_shortcut_icon_dllloader.rb
/usr/share/metasploit-framework/modules/exploits/windows/smb
Save
meterpreter_reverse_https.rb
ms15_020_shortcut_icon_dllloader.rb
Ruby
Tab Width: 8
Ln 28, Col 27
INS

```ruby
##
# This module requires Metasploit: http://metasploit.com/download
# Current source: https://github.com/rapid7/metasploit-framework
##

require 'msf/core'

class MetasploitModule < Msf::Exploit::Remote
  Rank = ExcellentRanking

  include Msf::Exploit::EXE
  include Msf::Exploit::FILEFORMAT
  include Msf::Exploit::Remote::SMB::Server::Share

  attr_accessor :exploit_dll_name

  def initialize(info = {})
    super(update_info(info,
      'Name'               => 'Microsoft Windows Shell LNK Code Execution',
      'Description'     => %q{
        This module exploits a vulnerability in the MS10-046 patch to abuse (again) the handling
        of Windows Shortcut files (.LNK) that contain an icon resource pointing to a malicious
        DLL. This creates an SMB resource to provide the payload and the trigger, and generates a
        LNK file which must be sent to the target. This module has been tested successfully on
        Windows 2003 SP2 with MS10-046 installed and Windows 2008 SP2 (32 bits) with MS14-027
        installed.
      },
      'Author'          =>
        [
          'Michael Heerklotz', # Vulnerability discovery
          'juan vazquez' # msf module
        ],
      'License'         => MSF_LICENSE,
```

Source: Kali Linux

Figure 7-25 Examining the code of a Metasploit module written in Ruby