

BUILD YOUR PORT SCANNER IN RUBY



Understanding Ruby

Lets now take a look at a programming language instead of software or networking concepts. You may or may not have already heard of Ruby, but it is an extremely popular programming language that is supported on all of the major platforms including Mac OSX, Linux, and Windows.

Furthermore, it has a wide variety of applications on the Internet – especially for developing network penetration tools. Because it is such a flexible and powerful programming language, it can be used to make web-based video games, process text, or be used for network penetration scenarios. Though teaching an entire programming language is well outside the scope of this book, you should at least have a high level understanding of this popular programming language because it is so crucial in the world of penetration testing.

There are countless tools that hackers and penetration testers couldn't do without that are built – at least in part – upon Ruby. When installing penetration testing tools and utilities, it is more than likely that you will see an error message that relates to Ruby if you haven't followed the install procedure correctly or haven't updated your packages – and having an understanding of Ruby will help you overcome these errors.

Utilities that Use Ruby

There a lot of utilities used for penetration testing that are built on the Ruby framework. For example, Metasploit would not be able to run without the right underlying Ruby packages installed first. Metasploit is arguably one of the largest and most popular penetration testing tools that utilizes Ruby, and it relies heavily on Ruby's tools, code libraries, and interfaces.

Furthermore, there are many different Ruby frameworks. For example, Metasm – which is included in Metasploit, is a powerful tool that allows developers to compile, debug, reverse engineer, and disassemble native code. Another good example of a Ruby framework is Ronin, which is well-suited for security, penetration testing, and data exploration. To put it simply, Ruby offers too many security tools and code structures to ignore if a developer wants to create penetration testing software.

What If I Don't Have a Programming Background?

Even if you don't have a programming background, you can still learn Ruby. As opposed to other types of programming languages, Ruby is relatively easier to learn. Some of the larger programming languages use strongly typed syntax that throws compiler errors with the smallest mistakes. It is true, however, that other programming languages give you greater control over the hardware you are programming. These highly technical languages are referred to as low level programming languages.

Ruby, on the other hand, is a high level programming language. What this means is that the code is less cryptic than some of the more complex and complicated languages. In fact, that's not an accident – it was intentionally designed to be more human-readable. The designer of Ruby, Yukihiro Matsumoto, wanted to bridge the gap between machine code and human minds by structuring an emphasis on human needs as opposed to the needs of computers into this programming language. From the perspective of someone who doesn't have a strong programming background, this is fantastic. Ruby doesn't sound so scary now, does it?

In addition to being easier to read, it's easier to write and build an application because it is an interpreted language. This means that you don't need a compiler to process your code. In addition, like the majority of the most popular programming languages today, Ruby is an object oriented programming language.

One of the key goals of object oriented programming languages is to create blocks of code that are easily reused. That way, after you write a section of code you don't need to reinvent the wheel the next time you run into the same problem. Older languages, called procedural languages, lacked this ability and were no more than a long list of tedious commands with difficult looping procedures.

Being that Ruby is not incredibly difficult to learn and its wide variety of applications, it is naturally suited for web penetration testing. It can be used to crawl websites, fingerprint a target, and design applications that gather a wealth of information about a target. However, before we dig into the more exciting uses for Ruby, we first need to understand the concept of RubyGems.

What Are RubyGems?

RubyGems is essentially a package manager for Ruby. It is core to the Ruby programming language because it allows people to share their code in packages and libraries. For example, if I spent hours and hours writing code that solved problems all related to a central concept, it would be very advantageous to other programmers who face the same problems if they could use my code. Fortunately, they can! There are a vast number of code modules and libraries of functions that can be downloaded in the blink of an eye.

But consider what this means for you. Because there are a wealth of code libraries – or Gems – you don't have to waste valuable time coding structures that already exist. In addition, the library structure of Gems means that you don't need to understand how some portions of code work on a lower and more technical level. You just reuse their functions for their intended purposes, and away you go.

You should know that each Gem is given not only a name, but also a version number. This helps people track bug fixes and updates to the library. Also, each Gem has a

platform that it runs on. If the platform is **ruby**, then it can run in any environment that ruby is installed. However, other platforms are hardware specific and don't always have the capability to run on all the major platforms. The determination whether or not a library can run on a platform is influenced by factors such as the processor architecture, operating system type, and the operating system version.

Each Gem follows a similar structure to create consistency in the code libraries, too. Gems are typically made up of the actual code, documentation about the library, and Gem Specifications. These types of files will include valuable information such as the name, version number, licenses, a description, author information, and a link to the library's home page. By following a similar structure, it makes it much easier for developers to find the code that will best their needs for application development. The vast majority if these Ruby Gems are especially suited for developing penetration testing applications for a variety of reasons.

What Makes Ruby So Great for Penetration Testing?

Ruby isn't the only software development package that can develop top-notch network and penetration testing tools. In fact, a lot of people like Python due to the fact that is significantly faster than Ruby. However, Ruby has a lot of benefits that other software packages just don't have. The following are just a few tools, features, and benefits that Ruby contains that provide value to software developers:

- Packages that help to rip apart static binaries, which can be used in algorithms that intercept TCP/UDP streams for inspection
- Ruby uses regular expressions in a way that most other programming languages don't. You don't first need to import other structures, code, objects, or instantiate specific classes.
- Ruby has a smaller library than other programming languages. While this may sound like a disadvantage, it actually makes Ruby more streamlined. Instead of needing to do vast amounts of research on competing code libraries, developers can find the tools they need much quicker and easier than they could if they used another language.
- You can change entire portions of the library with "monkey patches" to bend the library to your will. This makes predefined objects in the library extremely flexible, and you can make tweaks very easily.
- You can even integrate some libraries from the C programming languages into Ruby. On the other hand, you can also integrate Ruby code modules into other programming languages, such as a tool written in C.

In addition, Ruby provides a lot of utilities that help people with reverse engineering, which is a crucial aspect of penetration testing. Reverse engineering allows people to deconstruct code and data after it has been compiled and built. You see, there a lot of

network protocols that aren't open standards – that is, they are completely proprietary. Without knowing how the protocol was written, it would be very difficult to disassemble network protocol data to gather information about different types of hosts and targets.

Regardless if an attacker's goal is to mine data, crack a handshaking algorithm, or deconstruct network protocol headers to gain a clearer picture of network services, Ruby offers a solution. Sometimes people use security tools to obfuscate their data in an effort to enhance privacy or hide their activity, but the Ruby framework has tools that can help developers reverse engineer the protocol to see this data. Ultimately, this aids Internet attackers by allowing them to discover and create backdoors into applications and services.

As another immense benefit, Ruby is extraordinary at providing developers with tools to analyze binaries. Often times penetration testers are faced with data that has been compressed, scrambled, or obfuscated. By editing the code modules with monkey patches, developers are able to tweak these tools and custom tailor them to a certain type of traffic stream of their choosing. By using these techniques, penetration testers and attackers alike can extract data from file systems that has been embedded in compressed data, executables, binary data, and even firmware.

Tools

Though Ruby can be used in many other applications, it is a favorite of developers who want to create penetration testing tools. From the perspective of someone who wants to invest time in learning penetration testing, it would be a good idea in the long term to learn how to code in Ruby. The great attackers and penetration testers of our day have at least a working knowledge of Ruby, and you would be remiss if you didn't try to learn it as well. It isn't one of the most challenging programming languages to learn, and tutorials can be found online for free.

Developing your penetration testing tools using Ruby would take a fair amount of time and focus, but you at least need to know about the various packages, frameworks, and RubyGems. Understanding these fundamentals will aid you in your Metasploit endeavors as well. Remember to check that Ruby has been installed on your operating system before you attempt to install penetration testing tools, because without it you will get a lot of headaches. This couldn't be truer in Linux environments, because you could get an error that says a package or dependency is missing – but the prompt will fail to mention which one. Though there are a lot of great programming languages that can be used to write network and penetration testing tools, Ruby is clearly a favorite because it has so much to offer.

Lets Build our own port scanner

But in order for everyone to understand the code, we're going to crush the basics and do a Ruby crash course! Now we are going to teach you the first half of everything you need to know to write and understand the code behind our port scanner. First, let's make a brief list of what we'll be covering in this article...

- Puts vs. print
- Strings and integers
- Converting to other data types
- Arrays

Alright, now that we have a brief summary of what we'll be covering, let's jump right in!

Puts vs. Print

If you've ever used a scripting language, you should be familiar with print. But for those of you who are completely new to scripting, print simply allows us to display something inside of the terminal. Ruby has two different versions of print, there's the regular print, and then there's *puts*. When we use print, it will print everything in the same line until we tell it to move lines. But if we use puts, it will automatically move to the next line when it's done. Let's quickly demonstrate this difference. I've made two scripts that print "Hello, World!" to the terminal. The first one uses print, and the second one uses puts. Let's go ahead and execute our first script using print:

```
root@kali:~# ruby hello1.rb
Hello World!root@kali:~#
```

Here we can see that using print will stay on the same line as previously stated, now let's see the same script, but we'll use puts instead:

```
root@kali:~# ruby hello2.rb
Hello World!
root@kali:~#
```

Alright! We can see here that puts automatically put a newline after it printed "Hello, World!". Generally, we'll use puts when we want to notify the user of something. For example: If we're beginning a port scan, we'll **put** that the scan has started. We can use print for things such as giving the user a prompt, so input can be done on the same line and is kept clean.

Now that we've covered puts vs. print, let's move on to strings and integers!

Strings and Integers

Remember how when we printed "Hello, World!", there were always quotations around the words? This is because "Hello, World!" is a string. Basically, when you place quotation marks around a set of characters, it signifies them as *words*. For example; 10 is seen as a number, but "10" is seen as a word. Whenever we take input from the user, it will generally be returned as a string, which means we'll need to convert it to a different type. This is because we can't perform math with a string. Try the following equation in your head: "Hello" 1. See? You can't add numbers to a word, and the computer is no different. When we print something, we need to print it as a string. Now that we've covered strings, let's move on to converting data types!

Converting Data Types

As we previously stated, strings and integers don't get along very well. That's why we need to know how to convert between them. When we want to convert between data types, there are multiple ways to do this. We'll be using dot notation. Basically, we list what we want to convert, and then we add a period, followed to "to_" and then the first letter of whatever we want to convert to. I'm going to show you a code snippet, and then we'll dissect it:

```
number1 = gets.chomp
number2 = gets.chomp

total = number1.to_i + number2.to_i

puts 'The total is ' + total.to_s
,
```

Here we can see that we've made two variables (A variable simply stores any value, you make them by typing whatever name you'd like them to have, followed by an equal sign, and then the value you wish to give them) and assigned them the value of *gets.chomp*. Using *gets* will allow the user to enter text into the terminal, and using *.chomp* will strip the newline character from the end of the result. This allows the user to give input as to what they want. Below that, we can see that we are converting what the user entered into integers, and we're adding them together. Let's go ahead and run this script:

```
root@kali:~# ruby convert.rb
9000
1
The total is 9001
root@kali:~#
```

We can see here that it worked! We were able to convert the user's strings into integers and add them together. Now let's move on to our next topic and talk about arrays.

Arrays

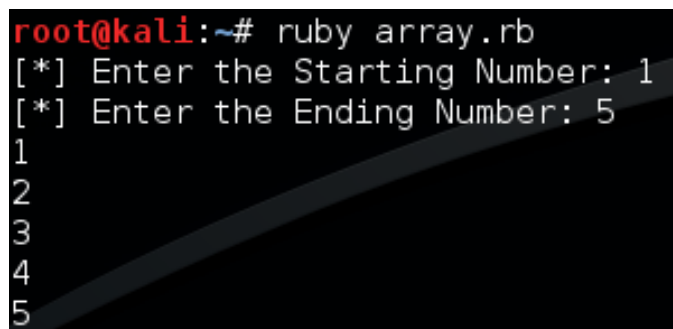
Arrays are fairly simple. An array is just a set of data instead of just one piece. Imagine if you had a box. That box represents a variable that can hold information. Normally, we can only put one thing in the box at a time, but what if we want to put multiple things in the box? We can make it an array. Let's edit the code from our last script and make an array containing every number between the two numbers that the user entered:

```
print '[*] Enter the Starting Number: '
start = gets.chomp
print '[*] Enter the Ending Number: '
ending = gets.chomp

ourArray = ((start.to_i)..(ending.to_i)).to_a

puts ourArray|
```

We can see here that instead of adding the two together, we've converted both of them to integers, and we've added ".." in between them. We placed this whole thing inside of parenthesis and we've ended it with ".to_a". We've already covered the conversions, but the interesting part is the ".." in the middle. Essentially, these two dots represent every number between the first integer and the last. Let's run this script and generate an array of numbers 1 through 5:



```
root@kali:~# ruby array.rb
[*] Enter the Starting Number: 1
[*] Enter the Ending Number: 5
1
2
3
4
5
```

There we go! We were able to successfully convert our strings to integers. Once we did that, we made an array of every number between them!

We're going to quickly cover how to write and execute a Ruby script, so that you can get some practice in. You can use any text editor (I prefer gedit), and once you've finished your script, save the text file with the ".rb" extension. Then open up a terminal and

locate your script. Once you've located your script, enter "ruby" into the terminal, followed by your script. This will run the text file as a Ruby script.

Part 2

Welcome back everyone! In the last article, we went over the first half of the Ruby basics to build our own port scanner. For those of you who didn't read the last article, I highly suggest you do so. In this second (and most likely final) Ruby basics article, we'll be covering some intermediate topics. Let's go ahead and list what we'll be covering here:

- Loops
- Methods
- Begin/Rescue

The first two items on this list are relatively simple, but the third can get rather complicated. But don't worry, we'll do our best to make sense of it. So, let's get to it!

Loops

Loops are just what they sound like, loops. Loops allow us to repeat a set of code over and over until the loop is finished or a certain result is achieved. For example; we may need to loop through an array and connect a port for each element in the array (That's a port scanner).

There are quite a few ways to perform loops in Ruby, but the way that I use most often is the `.each` method. What this means is that we need to use an iterable object and we need call `.each` after it. Once we perform these calls, we follow it with the keyword `do` followed by our variable(s).

Before we explain any further, let's take a look at a script that puts each element of an array individually. After we run it, we should be able to wrap our heads around it. Let's take a look at the code:

```
ourArray = ['My', 'name', 'is', 'Default', 'and', 'I\'m',  
            'an', 'author', 'at', 'howtohackin.com/blog']  
  
ourArray.each do |word|  
    puts word  
end
```

Alright, let's give a quick rundown of this code and then we'll execute it. We've made an array containing many strings, and we've made a `.each` loop. We can see the word "word" encased in pipes (This is a pipe: `|`). Think of this as a temporary variable name. Each element in the list will be housed under this temporary variable when it's their turn to be put through the loops code. Now that we know the in's and out's of this loop, let's execute this script:

```
root@kali:~# gedit loops.rb
root@kali:~# ruby loops.rb
My
name
is
Default
and
I'm
an
author
at
howtohackin.com/blog
root@kali:~#
```

There we go, we were able to print out each element of our array individually. Now that we know how loops work, let's move on to methods.

Methods

Repetition is very important in programming, no matter what language you use. Imagine if we needed to execute the same block of code multiple times in different place, it would be a huge pain to write the chunk of code multiple times, plus it would make the file size of the script rather large. If we want to execute a set of code multiple times in different places, we can use a method. Think of a method as a chunk of code that we've placed under a variable, we only need to call the variable name (and give the arguments) to call the chunk of code. We'll be make two methods, both will perform the same actions, but one will take arguments. Our methods are simply going to say hello to us. Let's write these methods now:

```

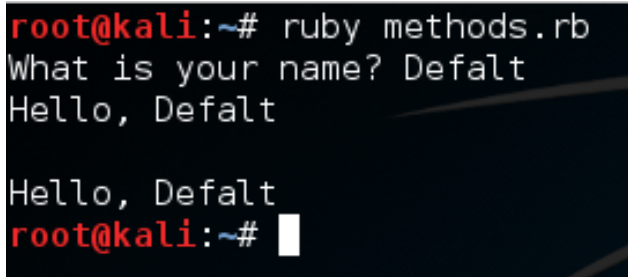
def method1()
  print 'What is your name? '
  name = gets.chomp
  puts "Hello, #{name}\n\n"
end

def method2(name)
  puts "Hello, #{name}"
end

method1()
method2("Default")

```

Alright. Here we've made two methods, both functioning nearly the same, with the exception of the implementation of arguments in the second method. Then we've called our methods. Let's see these methods in action and execute our script:



```

root@kali:~# ruby methods.rb
What is your name? Default
Hello, Default

Hello, Default
root@kali:~#

```

Alright, there we go! We were able to make and properly execute our methods! These will allow us to re-run a section of code whenever and wherever we need. Now that we've learned about methods, let's move on to a *very* important concept, **begin/rescue**.

Begin/Rescue

In the span of these two articles, I'm sure we've run into a few errors. When we make a script, we need to account for any possible errors that can occur. We need to account for these errors so that the error messages don't end up spilling all over the screen. This jumbled output looks extremely unprofessional and can often times devastate the functionality of a script.

We can use **begin/rescue** to account for these errors. It's fairly simple when you think about the words, **begin** this action, and **rescue** it from this error. There are many, *many* errors that can be generated by faulty code, and these errors are organized into a hierarchy. It would be very beneficial to become familiar with this hierarchy, this way you can distinguish between different errors and provide very specific feedback as to what went wrong.

We're going to start this section by making a script that doesn't work. We're going to intentionally make a script that will return an error, and then we're going to use `begin` and `rescue` to fix it. Let's start by making our faulty script:

```
print 'Please Enter a Number: '  
num = gets.chomp  
  
newnum = Integer num + 1  
puts "Your Number Plus 1 Equals #{newnum}"
```

This example script will simply ask the user for a number and tell them what their number plus one is. But as you may have guessed, the user can enter whatever they want, including things that aren't numbers. Let's go ahead and generate an error by entering a word instead of a number:

```
root@kali:~# ruby begres.rb  
Please Enter a Number: howtohackin.com/blog  
begres.rb:4:in `+': no implicit conversion of Fixnum into String (TypeError)  
    from begres.rb:4:in `<main>'  
root@kali:~#
```

Here we can see that we've generated an error. By taking a deeper look at the error message we can see that it was a `TypeError`. Now that we know what kind of error is generated by this bug, we can account for it using `begin` and `rescue`. Let's edit our code to account for this new error:

```
print 'Please Enter a Number: '  
num = gets.chomp  
  
begin  
    newnum = Integer num + 1  
    puts "Your Number Plus 1 Equals #{newnum}"  
rescue TypeError  
    puts "That's not a valid number!"  
end
```

We can see here that the syntax for utilizing `begin` and `rescue` is rather simple. We've merely placed the possibly volatile code under the **`begin`**, and we've placed the rescue code beneath the **`rescue`**. Now that we've modified our code, let's run it again and see what happens:

```
root@kali:~# ruby begres.rb
Please Enter a Number: howtohackin.com/blog
That's not a valid number!
root@kali:~#
```

There we go! We were able to successfully diagnose and account for a possible error! When making real life tools, error detection is *extremely* important. We stand no chance of notifying the users of an error if we don't even know what happened!

We covered some key topics today. Now lets build the port scanner!

Port Scanner

We have a few things we need to discuss first, so let's knock those out. We need to discuss what sockets are and how to use them in Ruby, let's discuss this now.

What are sockets?

Simply put, sockets allow us to make and manage connections over a network interface. This is what allows us to evaluate whether a port is open or closed. If we can successfully connect a socket to the target port, the port is open. Keep in mind that repeated and sequential connection such as this can and will be logged by both the victim and any IDS/IPS devices that may be listening.

Now that we know a little more about sockets, we can actually start making the port scanner. We're going to be breaking the code down into sections and analyzing each section individually, so let's get started!

Step 1: Setting Interpreter Path and Requiring Modules

When we make a Ruby script, it can be a real pain to have to type "ruby [SCRIPT NAME]" in order to execute it every time. So instead we can set the interpreter path. This will allow us to treat the file as a regular executable. We can mark the file as a ruby script from within the file by setting the interpreter path. This must be on the first line of the file, and is preceded by the shebang (!). We also need to import the necessary modules for our port scanner. Modules are chunks of code that are logically grouped into files so we can pick and choose what we need. Now that we've discussed what this section of the script will do, let's take a look at the code, it will seem fairly simple compared to what we just discussed:

```
#!/usr/bin/ruby

require 'socket'
require 'timeout'

$rhost = ARGV[0]
$min_port = ARGV[1]
$max_port = ARGV[2]
```

The last bit of this snippet involves us calling multiple elements out of the "ARGV" array. This is an automatically generated array that contains command line arguments in the order they're given. This means that when we run this script, we need to give the target, the starting port, and the ending port as command line arguments.

Now that we have the first snippet out of the way, we can work with some things we've already covered.

Step 2: Generate an Array of Ports to Scan

Now that we have a start and end port provided by the user, we need to generate an array of numbers to serve as port numbers. We already covered how to convert between different port numbers, and we briefly covered how to generate a range of numbers. We're going to be chaining conversions together here, but don't worry, it'll all come out nice and clean. Let's take a look at this snippet before we dissect any further:

```
begin
  if (Integer $min_port) <= (Integer $max_port)
    $to_scan = ((Integer $min_port)..(Integer $max_port)).to_a
  else
    puts "[!] Error: Invalid Range of Ports"
    exit
  end
rescue ArgumentError
  puts "[!] Error: Invalid Range of Ports"
  exit
end
```

Alright, we can see here that we've placed the whole array generation inside a begin/rescue statement. We start our conversion with an if statement, which tests to see if the start port number is less than or equal to the stop port number. This is to avoid generating an invalid range of ports. We've stored our range of ports in a new variable called \$to_scan. The dollar sign in front of the variable name means this

variable can be accessed from anywhere in the script. Now that we've got our range of ports, we can build the method to scan a given port.

Step 3: Build the Port Scanning Method

When we finally scan the ports of the victim, we're going to need to perform the same action again and again, for every port in our array. In order to use this same piece of code over and over again, we're going to make a method and call that method for every port. Our method will take one argument, a port number, and it will then proceed to connect to that port and return true or false based on the result. Let's take a look at our method and then we'll give a deeper look:

```
def scanport(port)
  s = Socket.new Socket::AF_INET, Socket::SOCK_STREAM
  begin
    sockaddr = Socket.pack_sockaddr_in(port, $rhost)
  rescue
    puts "[!] Error: Failed to Resolve Target"
    exit
  end
  timeout(10) do
    begin
      @result = s.connect(sockaddr)
    rescue
      return false
    end
  end
  if @result == 0
    return true
  else
    return false
  end
end
```

We start by creating a socket. We do this by calling `.new` on the `Socket` module we required earlier. Then we follow it with some attributes we want our socket to have. Next, we make a new variable named `sockaddr`, in this variable we store the result of calling `pack_sockaddr_in` out of the `Socket` module. In order to connect our socket to a remote host, we need to properly pack the addressing information into it. This is the proper way of doing so, we've placed this within a `begin/rescue` just in case the socket fails to resolve the target.

If you remember back to the very beginning, we imported a second module, `timeout`. We can make a `timeout` do loop to attempt a certain action for a certain amount of time. Once the timer runs out it will move on to the next section of code. This `timeout` is

to prevent the script from hanging if a port is unresponsive. It will then assign the resulting value to the result variable. If the connection was successful, it will return 0. We can make a simple if statement to test for 0 result, and return true if it is. We will return false otherwise. Now that we've made a method to scan a given port, we can loop through our array and use it.

Step 4: Iterate Over the Array and Call the Method

Now that we have our method, we can use it on our array. We're going to use a .each loop and give our temporary variable the name "port". This loop is rather simple, so let's take a look before we dissect any deeper:

```
puts "[*] Beginning Scan... \n\n"

$to_scan.each do |port|
  if scanport(port)
    puts "Port " + port.to_s + ": Open"
  end
end

puts "\n[*] Scan Complete!"
```

First, we put that we're beginning the scan, followed by some blank lines for neatness. Then we make a .each loop with our to_scan array. We then make an if statement using our method. Remember how our method returned true or false? Well this is where that comes in handy. Instead of manually evaluating it, we can just place it in an if statement and let it take care of everything. If the result from our method is true, we print that the scanned port is open, anything else we just ignore. Once our scan is complete, we put that the scan is complete. Now that we have our port scanner we can test it out!

Step 5: Test it Out

Since we have a new tool, we need to test it out. First we're going to perform a basic nmap scan against our target and see what results to expect:


```
root@kali:~# nmap -Pn 10.0.0.1 -p1-100

Starting Nmap 7.00 ( https://nmap.org ) at 2016-04-10 23:52 CDT
Nmap scan report for 10.0.0.1
Host is up (0.012s latency).
Not shown: 98 closed ports
PORT      STATE SERVICE
53/tcp    open  domain
80/tcp    open  http
MAC Address: E4:F4:C6:0A:7E:7E (Netgear)

Nmap done: 1 IP address (1 host up) scanned in 0.37 seconds
root@kali:~#
```

Alright, if we scan ports 1 through 100, we can expect that ports 80 and 53 will be open. Let's go ahead and fire up our port scanner! First, we need to make the file executable using the [chmod](#) command. Once we've made it executable, we can fire it up and use it. Let's do both now:

```
root@kali:~# chmod +x portscan.rb
root@kali:~# ./portscan.rb 10.0.0.1 1 100
[*] Beginning Scan...

Port 53: Open
Port 80: Open

[*] Scan Complete!
root@kali:~#
```

There we go! We were able to successfully build our own, functional port scanner! I hope now that we've built our own, we have a bit better understanding about how they work.