Automating Post Exploitation with Metasploit

Ruby Primer

Automating Metasploit Post Exploitation

Disclaimer

The author of this class is not responsible for the use of this information. The information here provided is for the use of security professionals to automate post exploitation task while performing authorized security assessments and tasks.

Not all API calls available will be covered during this class, only those that are considered to be the most useful one based on the instructors experience. The Metasploit Framework is in constant evolution, this course covers the current version of the framework at the time of it’s

RUBY PRIMER

Ruby Primer

- Created in Japan in 1995 by Yukihiro “Matz” Matsumoto
- Interpreter language, not compiled in to machine code
- Object Oriented, everything is an object in Ruby
- The majority of the Metasploit framework is based on Ruby

Ruby Primer

- It is whitespace independent
- Tends to read like English
- Ruby is not Ruby on Rails! RoR is a Framework based on Ruby.
- The current versions of ruby are the 1.8.X and 1.9.X

Ruby Primer

- To run ruby you can run in it in command
  - `ruby -e 'puts hello world'`
- You can run ruby in a file
- Ruby can be used inside the Interactive Ruby Interpreter (IRB)

```
$ irb
irb(main):001:0>
```

- We will use IRB for most of the examples

Ruby Primer

- We will run for the Ruby examples inside the IRB shell provided inside msfconsole since this is a ruby primer for metasploit

```
$ ./msfconsole -q
msf > irb
[*] Starting IRB shell...

>>
```

Documentation

- To get documentation on Ruby core libraries and the language visit http://www.ruby-doc.org/
- For framework documentation http://rapid7.github.io/metasploit-framework/api/
- Google works also :) like “ruby string”
- Another method is using the ri command line tool

Ruby Primer

- There are several ways in the framework to show output

```
>> print "hello world"
hello world=> nil
```

```
>> print_good "hello world"
[+] hello world
=> nil

>> print_status "hello world"
[*] hello world
=> nil

>> print_error "hello world"
[-] hello world
=> nil
```

- Never us `puts` to print out put to the

Ruby Primer

- Everything in Ruby is an object (Except Variables)
- Each object is an instance of an object
- To get the type of object you use the `.class` method
- To get a list of the methods that can be used on an object you can list them with `.methods`

Object Types

Variables

- Variables are not Objects, they are references to object
- Once defined they are treated like objects
- They need to be defined before we can use them
- When naming use all lower case with underscores between words
- Do not use mix case or dashes

Variable Scopes

- Global - $variable (Never use in scripts and modules)
- Class - @@variable (Never use in scripts and modules)
- Instance - @variable
- Local - variable
- Block - variable
- * Constants start with capital letters

Assignment Operators

Operator	Description
=	Simple assignment operator, Assigns values from right side operands to left side operand
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand
**=	Exponent AND assignment operator, Performs exponential (power) calculation on operators and assign value to the left operand

Integers Numbers

- Integers are objects that represent positive numbers.
- There are 2 subclasses of integers Fixnum (machine word -1bit) and Bignum (Any larger number than a word)
- Integers can be positive or negative numbers
- You can get a string representation of a number as an integer with .to_i

Float Numbers

- Float is nothing more that decimal numbers
- Floats can be positive or negative numbers
- Arithmetic of integer numbers will always return an integer number, if any element is a float then float is returned

```
>> 8/2
=> 4

>> 8/3
=> 2

>> 8/3.0
=> 2.6666666666666665
```

Float Rounding

- Turning converting an float object to an integer the rounding varies depending the method used

```
>> 2.45.to_i
=> 2

>> 2.55.to_i
=> 2

>> 2.55.round
=> 3

>> 2.45.round
=> 2
```

Arithmetic Operators

Operator	Description
+	Addition - Adds values on either side of the operator
-	Subtraction - Subtracts right hand operand from left hand operand
*	Multiplication - Multiplies values on either side of the operator
/	Division - Divides left hand operand by right hand operand
%	Modulus - Divides left hand operand by right hand operand and returns remainder
**	Exponent - Performs exponential (power) calculation on operators

Strings

- Strings are a series of characters strung together
- Strings between quotes(“) are evaluated

```
>> "HKLM\System"
=> "HKLMSystem"

>> 'HKLM\System'
=> "HKLM\\System"
```

- Strings can be added together with the +

```
>> "msf" + " " + "Rocks"
=> "msf Rocks"
```

Strings

- You can evaluate Ruby code inside a string with double quotes with #{}

```
>> name = "Carlos"
=> "Carlos"

>> "hello #{name}"
=> "hello Carlos"

>> " 1 + 2 = #{1+2}"
=> " 1 + 2 = 3"
```

- You can also append to a string

```
>> todo_list = "1. Buy groceries\n"
=> "1. Buy groceries\n"

>> todo_list << "2. Take car to get serviced\n"
=> "1. Buy groceries\n2. Take car to get serviced\n"
```

Strings

Escape Sequence	Description
\a	Bell or alert
\b	Backspace
\cx	Control-x
\C-x	Control-x
\e	Escape
\n	Newline
\r	Carriage return
\s	Space
\t	Tab
\v	Vertical tab
\x	Special Character x like backslash or quotes
\xnn	Hexadecimal notation, where n is in the range 0.9, a.f, or A.F

Strings

Some helpful methods are `split`, `chomp` and `strip`

```
>> todo_list.split("\n")
=> ["1. Buy groceries", "2. Take car to get serviced"]
```

```
>> "file.txt  \n".chomp
=> "file.txt  "
```

```
>> "    file.txt  \n".strip
=> "file.txt"
```

Array

- Arrays are an ordered, integer indexed collection of objects.
- Arrays can have a mix of different objects types as it's elements (Numbers, Strings, Arrays ..etc)
- Each element of an array can be called by the integer index value that represent it's position, the index starts at 0

Array

- Requesting specific element of an array

```
>> sampl_array = ["1",2,3.4,nil]
=> ["1", 2, 3.4, nil]

>> sampl_array[3]
=> nil
>> sampl_array[2]
=> 3.4
```

- Adding and Removing from an Array

```
>> array1 = [1,2,3,4,5]
=> [1, 2, 3, 4, 5]
>> array2 = ["a", "b", "c"]
=> ["a", "b", "c"]
>> array3 = array1 + array2
=> [1, 2, 3, 4, 5, "a", "b", "c"]
>> array3 - ["a","b"]
=> [1, 2, 3, 4, 5, "c"]
>> array2 << "d"
=> ["a", "b", "c", "d"]
```

Array

- Checking if an element in an Array is present

```
>> array2 = ["a", "b", "c", "d"]
=> ["a", "b", "c", "d"]

>> array2.include?("f")
=> false

>> array2.include?("a")
=> true
```

- Removing duplicates in an array

```
>> array1 = ["a","b","b",3,3,3,4,5,6,7]
=> ["a", "b", "b", 3, 3, 3, 4, 5, 6, 7]

>> array1.uniq
=> ["a", "b", 3, 4, 5, 6, 7]
```

Hashes

- Hashes are an unordered, object indexed (key) collection of objects (value), known as dictionaries in other languages
- Creating a hash, looking by key and adding a element

```
>> hash1 = { 'path' => "c:\\boot.ini", 'file' => "boot.ini"}
=> {"path"=>"c:\\boot.ini", "file"=>"boot.ini"}

>> hash1['path']
=> "c:\\boot.ini"

>> hash1['description'] = "Windows boot parameter file"
=> "Windows boot parameter file"

>> hash1
=> {"path"=>"c:\\boot.ini", "file"=>"boot.ini", "description"=>"Windows boot parameter file"}
```

Hashes

- Check if it contains a value

```
>> hash1.value?('boot.ini')
=> true

>> hash1.value?('autoexec.ini')
=> false
```

- Merging together 2 hashes

```
>> hash1 = { "key1" => 1, "key2" => 2}
=> {"key1"=>1, "key2"=>2}

>> hash2 = { "key3" => 3, "key4" => 4}
=> {"key3"=>3, "key4"=>4}

>> hash1.merge(hash2)
=> {"key1"=>1, "key2"=>2, "key3"=>3, "key4"=>4}
```

Symbols

- Symbols are labels to identify a pice of data
- A symbol will only be stored in memory one time

```
>> :labe1.object_id
=> 489128

>> :labe1.object_id
=> 489128

>> "label1".object_id
=> 70185273123240

>> "label1".object_id
=> 70185273114580
```

- They are used mostly for Hash keys

```
>> hash3 = { :file => "secretdata.doc", :path => "c:\\secret\\"}
=> {:file=>"secretdata.doc", :path=>"c:\\secret\\"}
```

Booleans

- They are either true or false
- They are use for control structures
- True and False are objects

```
>> true.class
=> TrueClass

>> false.class
=> FalseClass
```

Comparison Operators

Operator	Description
==	Checks if the value of two operands are equal or not, if yes then condition becomes true.
!=	Checks if the value of two operands are equal or not, if values are not equal then condition becomes true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.
<=>	Combined comparison operator. Returns 0 if first operand equals second, 1 if first operand is greater than the second and -1 if first operand is less than the second.
===	Used to test equality within a when clause of a case statement.
.eql?	True if the receiver and argument have both the same type and equal values.
equal?	True if the receiver and argument have the same object id.

Booleans

- Using comparison Operators

```
>> 1 == "1"
=> false

>> 1 == "1".to_i
=> true
```

```
>> 1 > 2
=> false
```

Logical Operators

Operator	Description
and	Called Logical AND operator. If both the operands are true then then condition becomes true.
or	Called Logical OR Operator. If any of the two operands are non zero then then condition becomes true.
&&	Called Logical AND operator. If both the operands are non zero then then condition becomes true.
||	Called Logical OR Operator. If any of the two operands are non zero then then condition becomes true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.
not	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.

Booleans

- Logical Operators

```
>> 1 > 2 && 1 == 1
=> false

>> 1 > 2 || 1 == 1
=> true
```

```
>> 1 > 2 or 1 == 1
=> true

>> 1 > 2 and 1 == 1
=> false
```

Ranges

- Short notation object for a range of characters or numbers
- There are 2 types of ranges inclusive and exclusive, the exclusive excludes the last value

```
>> r = 1..10
=> 1..10

>> [*r]
=> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

>> r = "a".."f"
=> "a".."f"

>> [*r]
=> ["a", "b", "c", "d", "e", "f"]
```

Control Structures

If Conditional

- If statement will execute a block of code is the condition is returns a true object

```
x, y = 1, 2
# Execute if condition is true
if x > y
    puts "x is more than y ... WTF!"
end
```

```
# if condition is not true execute one action, if not execute alternate action
if x > y
    puts "x is more than y ... WTF!"
else
    puts "y us more than x"
end

# if we want to check more than one condition
if condition
    action
elsif condition
    action
end
```

If Conditional

- The “if” conditional can be used as a modifier

```
print_status("Verbose mode is enabled") if datastore['VERBOSE']
```

- Nil object is treated as false

```
>> nil_value = nil
=> nil

>> puts "variable is empty" if not nil_value
variable is empty
=> nil

# Recommended method
>> puts "variably is empty" if nil_value.nil?
variably is empty
=> nil
```

- For empty variables use **`.empty?`**

Unless Conditional

- Works just like if but the condition is inverted

```
>> target_compat = false
=> false

>> print_error("Target is not compatible") unless target_compat
[-] Target is not compatible
=> nil
```

- It can use else just like **`if`** and you can mix it with **`elsif`** - NOT Recommended for readability

Case Conditional

- Used when multiple conditionals are needed. recommended when more than 2 multiple conditionals are needed.

```
case name
    when "Path"
        print_status "\tPath: #{val}"
        path = val
    when "Type"
        print_status "\tType: #{val}"
        stype = share_type(val.to_i)
    when "Remark"
        remark = val
end
```

- Regular Expressions can be used for the

Loops

- They are used to run a pice of code over an over until it is told to stop

```
x = 0
loop do
    x = x + 1
    print_status("x is #{x}")
    break if x == 100
end
```

- You control the loop with Control Calls

Control Call	Description
break	Terminate the whole loop
next	Jump to the next loop
redo	Restart the loop
retry	Start the whole loop over

Boolean Loops

- This are loops that execute while a boolean conditional is meet
- While Boolean Loop

```
?>   x = 0
=> 0
>> while x < 5
>>   print_status("x is #{x}")
>>   x = x + 1
>>   end
[*] x is 0
[*] x is 1
[*] x is 2
[*] x is 3
[*] x is 4
=> nil
```

Boolean Loops

- Until Boolean Loop

```
>> x = 0
=> 0
>> until x > 5
>>   print_status("x is #{x}")
>>   x = x + 1
>>   end
[*] x is 0
[*] x is 1
[*] x is 2
[*] x is 3
[*] x is 4
[*] x is 5
=> nil
```

Iterators

- They are like loop, they traverse a set of data
- There are 2 iterator for Integers and floats named **`upto()`** and **`downto()`**

```
>> 1.upto(6) do |i|
?>     print_status(i)
>>   end
[*] 1
[*] 2
[*] 3
[*] 4
[*] 5
[*] 6
=> 1
```

```
>> 6.downto(0) do |i|
?>     print_status(i)
>>   end
[*] 6
[*] 5
[*] 4
[*] 3
[*] 2
[*] 1
[*] 0
=> 6
```

Iterators

- For range the iterators available are **`range.each`** and **`range.step(n)`**

```
>> (1..5).each do |e|
?>     print "#{e} "
>>   end
1 2 3 4 5 => 1..5
```

```
>> (1..6).step(2) do |e|
?>     print e.to_s + " "
>>   end
1 3 5 => 1..6
```

- **For** can be used and an iterator like in other languages

```
>> for n in (1..6)
>>   print n.to_s + " "
>>   end
1 2 3 4 5 6 => 1..6
```

Iterators

- For strings the iterators available are **`str.each_line`**, **`str.each_char`** and **`str.each_byte`**

```
>> multi_line_str = "line1\nline2\nline3\n"
=> "line1\nline2\nline3\n"
>> multi_line_str.each_line do |l|
?>     print_status(l.strip)
>>   end
[*] line1
[*] line2
[*] line3
=> "line1\nline2\nline3\n"
```

- The **`for`** iterator can be used, behave like **`each_line`**

Iterators

- For Array the iterators available are **`arr.each`**, **`arr.each_index`** and **`arr.each_with_index`**

```
>> arr = [1,2,3,4,5,6]
=> [1, 2, 3, 4, 5, 6]
>> arr.each do |i|
?>     print_status(i)
>> end
[*] 1
[*] 2
[*] 3
[*] 4
[*] 5
[*] 6
=> [1, 2, 3, 4, 5, 6]
```

```
>> arr.each_with_index do |i,d|
?>     print_status("value:#{i} index:#{d}")
>> end
[*] value:1 index:0
[*] value:2 index:1
[*] value:3 index:2
[*] value:4 index:3
[*] value:5 index:4
[*] value:6 index:5
=> [1, 2, 3, 4, 5, 6]
```

- The **`for`** iterator can be used, behave like

Iterators

- For Hash the iterators available are **`hsh.each`**, **`hsh.each_key`**, **`hsh.each_value`** and **`hsh.each_pair`**

```
>> hsh = {:company => "Rapid7", :name => "HD", :project => "Metasploit"}
=> {:company=>"Rapid7", :name=>"HD", :project=>"Metasploit"}
>> hsh.each_pair do |k,v|
?>     print_status("#{k} = #{v}")
>> end
[*] company = Rapid7
[*] name = HD
[*] project = Metasploit
=> {:company=>"Rapid7", :name=>"HD", :project=>"Metasploit"}
```

- The **`for`** iterator can be used, behave like **`each`**

Methods

- Methods are also known as functions in other scripting and programing languages
- They tend to be blocks of code that server a specific function
- Variable declared inside methods are local to the methods
- The last assignment in a method is returned by default in a methods, do not rely on this

Methods

- Methods are defined with the keyword of **`def`** and the block of code is closed with the keyword of **`end`**

```
def get_base_info
    sys_data = session.sys.config.sysinfo
    sys_data['User'] = session.sys.config.getuid
    sys_data['PID']  = session.sys.process.getpid
    sys_data["PWD"]  = session.fs.dir.getwd
    return sys_data
end
```

- It is recommended that you use **`return`** to specify the data returned by a method instead of relying of the last assignment behavior

Methods

- You can pass values to methods thru a list of comma separated variables know as arguments

```
def eventlog_clear(evt = nil)
    evntlog = []
    if evt.nil?
        evntlog = eventloglist
    else
        evntlog << evt
    end
    evntlog.each do |e|
        log = session.sys.eventlog.open(e)
        log.clear
    end
    return evntlog
end
```

- You can set default values to arguments by declaring them in the method definition

Methods

- The number of arguments and order are of importance they should be specified in the same order they are declared
- Place default variables at the end of the list to make calling methods simpler
- If the argument is optional place it at the end with a value of nil or an empty variable (“”,[], {})
- A return statement will also exit a method

Methods

- Instance variables (Variables that start with @) can be used to have an object that all methods inside the class or script can access.