Writing Your Own Scanner a11y.text Writing Your Own Scanner Using your own Metasploit Auxiliary Module a11y.text Using your own Metasploit Auxiliary Module There are times where you may need a specific networkÂ security scanner, or having scan activity conducted within Metasploit would be easier for scripting purposes than using an external program. Metasploit has a lot of features that can come in handy for this purpose, like access to all of the exploit classes and methods, built in support for proxies, SSL, reporting, and built in threading. Think of instances where you may need to find every instance of a password on a system, or scan for a custom service. Not to mention, it is fairly quick and easy to write up your own custom scanner. Some of the many Metasploit scanner features are: It provides access to all exploit classes and methods Support is provided for proxies, SSL, and reporting Built-in threading and range scanning Easy to write and run quickly Writing your own scanner module can also be extremely useful during security audits by allowing you to locate every instance of a bad password or you can scan in-house for a vulnerable service that needs to be patched. Using the Metasploit Framework will allow you to store this information in the database for organization and later reporting needs. We will use this very simple TCP scanner that will connect to a host on a default port of 12345 which can be changed via the scanner module options at run time. Upon connecting to the server, it sends â€˜HELLO SERVERâ€™, receives the response and prints it out along with the IP address of the remote host. require 'msf/core'

class Metasploit3 < Msf::Auxiliary include Msf::Exploit::Remote::Tcp include Msf::Auxiliary::Scanner

def initialize super('Name' => 'My custom TCP scan',

```
          'Version'        => '$Revision: 1
```

Saving and Testing our Auxiliary Module

We save the file into our **./modules/auxiliary/scanner/** directory as **simple_tcp.rb** and load up msfconsole. Itâ€™s important to note two things here. First, modules are loaded at run time, so our new module will not show up unless we restart our interface of choice. The second being that the folder structure is very important, if we would have saved our scanner under **./modules/auxiliary/scanner/http/** it would show up in the modules list as **scanner/http/simple_tcp**.

[![Metasploit Auxiliary Modules - modules/auxiliary/scanner](https://www.offsec.com/wp-content/uploads/2015/04/metasploit-framework-modules-auxiliary-scanner.png)](https://www.offsec.com/wp-content/uploads/2015/04/metasploit-framework-modules-auxiliary-scanner.png)

Metasploit Auxiliary Modules â€“ modules/auxiliary/scanner path | Metasploit Unleashed

To test our security scanner, set up a netcat listener on port 12345 and pipe in a text file to act as the server response. root@kali:~# nc -lnvp 12345 < response.txt
listening on [any] 12345 â€¦ Next, you select your new scanner module, set its parameters, and run it to see the results. msf > use scanner/simple_tcp
msf auxiliary(simple_tcp) > set RHOSTS 192.168.1.100
RHOSTS => 192.168.1.100
msf auxiliary(simple_tcp) > run [ ] Received: hello metasploit from 192.168.1.100
[ ] Auxiliary module execution completed As you can tell from this simple example, this level of versatility can be of great help when you need some custom code in the middle of a penetration test. The power of the framework and reusable code really shines through here.

### Reporting Results from our Security Scanner

The _report_ mixin provides _report_*()_. These methods depend on a database in order to operate:

* Check for a live database connection
* Check for a duplicate record
* Write a record into the table

The database drivers are now autoloaded. db_driver postgres (or sqlite3, mysql) Use the **Auxiliary::Report** mixin in your scanner code. include Msf::Auxiliary::Report Then, call the report_note() method. report_note(
:host => rhost,
:type => â€œmyscanner_passwordâ€•,
:data => data
) Learning to write your own network security scanners may seem like a daunting task, but as weâ€™ve just shown, the benefits of creating our own [auxiliary module](https://www.offsec.com/metasploit-unleashed/creating-auxiliary-module/) to house and run our security scanner will help us in storing and organizing our data, not to mention help with our report writing during our pentests.,

```
                        'Description'    => 'My quick scanner',
                        'Author'         => 'Your name here',
                        'License'        => MSF_LICENSE
                )
                register_options(
                        [
                                Opt::RPORT(12345)
                        ], self.class)
        end
```

```
    def run_host(ip)
        connect()
 greeting = "HELLO SERVER"
 sock.puts(greeting)
        data = sock.recv(1024)
        print_status("Received: #{data} from #{ip}")
        disconnect()
    end
```

end Saving and Testing ourÂ Auxiliary Module a11y.text Saving and Testing ourÂ Auxiliary Module We save the file into our ./modules/auxiliary/scanner/ directory as simple_tcp.rb and load up msfconsole. Itâ€™s important to note two things here. First, modules are loaded at run time, so our new module will not show up unless we restart our interface of choice. The second being that the folder structure is very important, if we would have saved our scanner under ./modules/auxiliary/scanner/http/ it would show up in the modules list as scanner/http/simple_tcp . Metasploit Auxiliary Modules â€“ modules/auxiliary/scanner path | Metasploit Unleashed To test our securityÂ scanner, set up a netcat listener on port 12345 and pipe in a text file to act as the server response. urltomarkdowncodeblockplaceholder10.9428923116238732 Next, you select your new scanner module, set its parameters, and run it to see the results. urltomarkdowncodeblockplaceholder20.9067942391682755 As you can tell from this simple example, this level of versatility can be of great help when you need some custom code in the middle of a penetration test. The power of the framework and reusable code really shines through here. Reporting Results from our Security Scanner a11y.text Reporting Results from our Security Scanner The report mixin provides report_*() . These methods depend on a database in order to operate: Check for a live database connection Check for a duplicate record Write a record into the table The database drivers are now autoloaded.

urltomarkdowncodeblockplaceholder30.1327780749406582 Use the Auxiliary::Report mixin in your scanner code. urltomarkdowncodeblockplaceholder40.871109615486563 Then, call the report_note() method. urltomarkdowncodeblockplaceholder50.6872090478721424 Learning to write your own network security scanners may seem like a daunting task, but as weâ€™ve just shown, the benefits of creating our own auxiliary module to house and run our security scanner will help us in storing and organizing our data, not to mention help with our report writing during our pentests.

Next Windows Patch Enumeration Prev SNMP Sweeping