**MINISTÉRIO DA DEFESA**

**EXÉRCITO BRASILEIRO**

**DEPARTAMENTO DE CIÊNCIA E TECNOLOGIA**

**INSTITUTO MILITAR DE ENGENHARIA**

**(Real Academia de Artilharia, Fortificação e Desenho, 1792)**

# ANÁLISE DE SOFTWARE MALICIOSO (*MALWARE*) PARA O COMBATE DE VÍRUS, *WORMS* E VARIANTES

**Alu KIZZY FERNANDA TERRA FERREIRA DOS REIS**

**Rio de Janeiro**

**2012**

**INSTITUTO MILITAR DE ENGENHARIA**

**Alu KIZY FERNANDA TERRA FERREIRA DOS REIS**

# ANÁLISE DE SOFTWARE MALICIOSO (*MALWARE*) PARA O COMBATE DE VÍRUS, *WORMS* E VARIANTES

Relatório Final do Programa Institucional de Bolsas de Iniciação em Desenvolvimento Tecnológico e Inovação do CNPq/Instituto Militar de Engenharia

Orientador: Claudio Gomes de Mello, DSc, IME

Rio de Janeiro

2012

**SUMÁRIO**

## LISTA DE FIGURAS

## RESUMO

Este trabalho tem por objetivo a pesquisa e o desenvolvimento de uma metodologia para análise de vírus, *worms* e outros agentes de software maliciosos (chamados de *malware – malicious software*). Este enfoque foi adotado devido ao crescimento das ameaças virtuais, em virtude, principalmente, da globalização e conseqüente popularização dos serviços eletrônicos. Dessa forma, faz-se necessário criar um conhecimento avançado sobre o funcionamento dos *malwares*, de tal forma a permitir o melhor entendimento de suas técnicas de replicação e infecção, assim como o desenvolvimento de defesas proprietárias, ou seja, software antivírus, contra esses agentes. Como prova de conceito deste trabalho foram desenvolvidos códigos-fonte que possibilitam a leitura de um arquivo no formato *Portable Executable(PE)* e a verificação de certos perfis maliciosos, de tal forma a identificar possíveis *malwares*.

**ABSTRACT**

*This paper aims to research and develop a methodology for analysis of viruses, worms and other malicious software agents. This approach was adopted due to the growth of cyber threats, due mainly to the globalization and the consequent popularity of electronic services thus it is necessary to create an advanced knowledge on the functioning of malware so as to enable a better understanding of their techniques of replication and infection as well as the development of proprietary defenses, i.e., antivirus software against these agents. As a result of this research were developed source codes that allow reading a file in Portable Executable (PE) and verification of certain malicious profiles in such a way as to identify possible malware.*

# 1 INTRODUÇÃO

A facilidade de utilização dos diversos serviços eletrônicos disponíveis nos mais variados web sites (e-mail, games, chats), impulsionada, principalmente, pelo processo de democratização do acesso às tecnologias de informação, incentiva o crescimento das ameaças dos agentes de softwares maliciosos como os vírus, cavalos de tróia e worms (chamados de *malwares – malicous software*), por exemplo. Torna-se, portanto, muito importante desenvolver conhecimentos avançados sobre o comportamento desses agentes a fim de que se possa evitar sua ação e prognosticar possíveis danos.

Nesse contexto, insere-se a necessidade de analisar arquivos binários maliciosos. Contudo para que isso seja possível é necessário dominar as peculiaridades do formato de cada arquivo suspeito que esteja sendo estudado para que se possa, então, extrair e analisar todas as informações que indiquem se o arquivo é realmente malicioso.

A geração de um software de engenharia reversa pode ser dividida em três fases: a primeira fase é a varredura (*parser)* do arquivo; a segunda, é a extração e organização do maior número de informações possível, ao passo que na terceira fase deve-se comparar os dados extraídos com assinaturas de vírus conhecidos com o intuito de se detectar agentes maliciosos.

O objetivo geral deste trabalho, por sua vez, é o desenvolvimento de códigos-fonte de análise de agentes maliciosos, para as fases de varredura, extração e detecção de *malwares*.

No capítulo 2, apresenta-se o formato *Portable Executable*(PE), formato escolhido como objeto principal de estudo dessa pesquisa, devido ao fato de que este é o formato de arquivo compatível com Windows e por essa razão é alvo de um maior número de ameaças virtuais, isto é a maioria dos *worms* é desenvolvido para Windows.

No capítulo 3, faz-se uma breve apresentação sobre as características do formato ELF o qual é o padrão de arquivo binário para os sistemas *Unix*. Os capítulos 4 e 5, tratam das principais ferramentas *debugger* e *disassembler*, respectivamente.

No capítulo 6, é descrita a etapa de implementação do programa que faz a leitura de um arquivo PE, a qual ainda está em andamento e que corresponde a etapa de criação dos códigos-fonte. Apresenta-se um diagrama das classes até então implementadas.

O capítulo 7, trata da análise das informações extraídas do arquivo executável e o capítulo 8 é feita a conclusão do trabalho.

# 2 O FORMATO PORTABLE EXECUTABLE (PE)

O formato de arquivo *Portable Executable* (PE) foi projetado pela Microsoft e padronizado pelo Comitê do *Tool Interface Standart* (TIS) - Microsoft, Intel, Borland, Watcom, IBM e outros - em 1993. Foi baseado no *Common Object File Format* (COFF), usado originalmente para arquivos de objetos e executáveis nos vários sistemas UNIX e no VMS. Sua denominação deve-se a portabilidade dos arquivos desse tipo, que possibilitam a implementação em diversas plataformas (x86, MIPS®, Alpha, entre outros) sem que seja necessário reescrever as ferramentas para desenvolvimento. (DUMMER, DANIEL, 2006)

Os arquivos obedecem a um padrão de armazenamento de dados onde estão encapsuladas todas as informações necessárias ao carregamento (*loader)* do sistema operacional, essas informações são a representação binária das instruções de máquina para o processador.

## 2.1 ANALISANDO A ESTRUTURA DE UM ARQUIVO PORTABLE EXECUTABLE (PE)

O formato PE é constituído pelas seguintes estruturas de dados: cabeçalho MZ do DOS, fragmento (*stub*) do DOS, cabeçalho de arquivo PE, cabeçalho de imagem opcional, tabela de seções (que possui uma lista de cabeçalhos de seção), diretórios de dados (que contém os ponteiros para as seções) e por último as seções propriamente ditas conforme a FIG. 2.1.

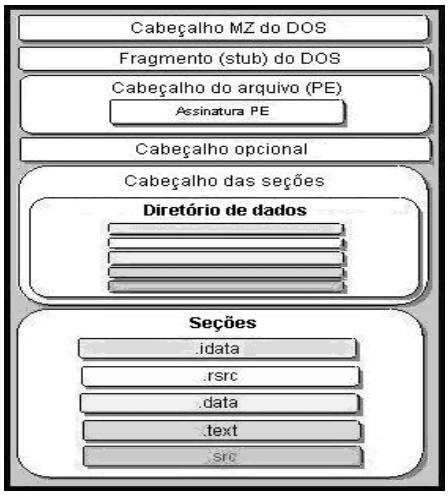


**FIG. 2.1 Estrutura do formato *Portable Executable***

### 2.1.1 CABEÇALHO MZ DO DOS

Os primeiros 64 bytes de qualquer arquivo PE constituem o cabeçalho do DOS. Os primeiros dois bytes deste cabeçalho constituem a assinatura do DOS. O *word* (conjunto de dois bytes) da assinatura sempre é a sequência "MZ", ou seja, "4D5A" em hexadecimal, por essa razão através dessa assinatura é possível fazer a validação do cabeçalho DOS. O cabeçalho DOS armazena outra informação importante do arquivo executável, seus últimos quatro bytes indicam o *offset* do cabeçalho de arquivo PE.

### 2.1.2 FRAGMENTO (STUB) DO DOS

Para o formato PE, o fragmento do DOS é um executável compatível com o MS-DOS 2.0, constituído por aproximadamente 100 bytes, cuja função é exibir instruções de erro. Uma parte dos valores tem o correspondente em ASCII de *"This program cannot be run in DOS mode"*, que é a string mostrada caso se tente executar este programa a partir do DOS.

### 2.1.3 CABEÇALHO DE ARQUIVO PE

O cabeçalho do arquivo ocupa 24 bytes e seus componentes são: *Signature*, *Machine* (tipo de máquina previsto para rodar o executável), *NumberOfSections*, *TimeDateStamp* (carimbo de data e hora),

*PointerToSymbolTable* (ponteiro para tabela de símbolos e número de símbolos), *SizeOfOptionalHeader* (tamanho do cabeçalho opcional), *Characteristics* (características).

O *Signature* é a assinatura do cabeçalho PE indica o início deste cabeçalho e sempre é a sequência "PE" seguida de dois zeros, correspondente o *dword "*50450000" em hexadecimal. Outro componente importante do cabeçalho do arquivo PE é o *NumberOfSections* que indica o número de seções após o cabeçalho

### 2.1.4 CABEÇALHO OPCIONAL

Imediatamente após o cabeçalho do arquivo vem o cabeçalho opcional que contém informações de como o arquivo PE deve ser tratado. Os componentes desse cabeçalho são: *Magic*, *MajorLinkerVersion*, *MinorLinkerVersion*, *SizeOfCode, SizeOfInitializedData, SizeOfUninitializedData, AdressOfEntryPoint, BaseOfCode, BaseOfData, ImageBase, SectionAlignment, FileAlignment, MajorOperatingSystemVersion, MinorOperatingSystemVersion, MajorImageVersion, MinorImageVersion, MajorSubsystemVersion, MinorSubsystemVersion, Win32VersionValue, SizeOfImage, SizeOfHeaders, CheckSum, Subsystem, DllCharacteristics, SizeOfStackReserve, SizeOfStackCommit, SizeOfHeapReserve, SizeOfHeapCommit, LoaderFlags, NumberOfRvaAndSizes, DataDirectory.*

Alguns dos componentes mais importantes deste cabeçalho são o *AdressOfEntryPoint*, que indica a posição relativa de memória (RVA- *Relative Virtual Adress*) do ponto de entrada do código do executável; e o *DataDirectory* armazenado nos 128 últimos bytes do cabeçalho opcional.

O *DataDirectory* compreende 16 diretórios de dados que descrevem a posição relativa de memória (um RVA de 32 bits) e o tamanho (também de 32 bits, chamado *Size*) de cada um das diferentes seções que seguem as entradas de diretório.

### 2.1.5 CABEÇALHO DAS SEÇÕES

Imediatamente após o cabeçalho do arquivo PE, está o cabeçalho das seções. Este cabeçalho é um *array* de estruturas, que armazena informações de cada uma das seções. Cada estrutura tem o 40 bytes, e contém as seguintes informações: *Name1* (um *array* de nomes das seções), *PhysicalAdress, VirtualSize, SizeOfRawData, PointerToRawData, PointerToRelocations, PointerToLinenumers, Characteristics*.

### 2.1.6 SEÇÕES

Após os cabeçalhos das seções seguem as seções propriamente ditas. Dentro do arquivo, elas estão alinhadas em *FileAlignment* bytes, ou seja, após o cabeçalho opcional e após cada uma das seções haverá bytes zerados de preenchimento para que seja atingido o tamanho *FileAlignment*. Por outro lado, quando carregadas na memória RAM, as seções ficam alinhadas segundo o tamanho designado pelo *SectionAlignment.* Além disto, as seções estão ordenadas pelos seus RVA. Também é possível encontrar o início das seções através do *PointerToRawData* (ponteiro de dados) ou através do *VirtualAddress* de maneira que a informação dos alinhamentos passa a ter menos relevância.

É interessante notar que existe um cabeçalho de seção para cada seção e cada diretório de dados apontará para uma das seções. Entretanto, vários diretórios de dados podem apontar para uma mesma seção e podem existir seções que não sejam apontadas pelo diretório de dados.

# 3 O FORMATO ELF

O formato ELF *(Executable and Linkable Format)* foi escolhido em 1999 como padrão de arquivo binário para os sistemas *Unix*. Atualmente, o formato ELF já substituiu outros formatos de execução mais antigos, tais como a.out e COFF nos sistemas operacionais *Linux*, *Solaris*, IRIX, *FreeBSD, NetBSD, e OpenBSD*. Existem três formas principais de emprego do formato ELF: relocável, executável e objeto compartilhado. Os arquivos relocáveis são criados pelos compiladores e devem ser processados pelo *linker* antes de executar; os arquivos executáveis são arquivos que podem ser executados pelo sistema operacional; e os arquivos-objeto compartilhados são as bibliotecas dos sistemas. (HEXSEL, ROBERTO A., 2009)

Os arquivos do formato ELF podem ser interpretados de duas maneiras distintas, comumente denominadas *Linking View* (ou visão de ligação), em que os compiladores e *linkers* tratam o arquivo como um conjunto de seções; e *Execution View* (ou visão de execução), em que o *loader* trata o arquivo como um conjunto de segmentos. Um segmento normalmente contém várias seções. As seções serão processadas pelo *linker,* ao passo que os segmentos serão mapeados na memória pelo *loader.*

A figura FIG. 3.1 a seguir ilustra as duas visões distintas acima descritas:

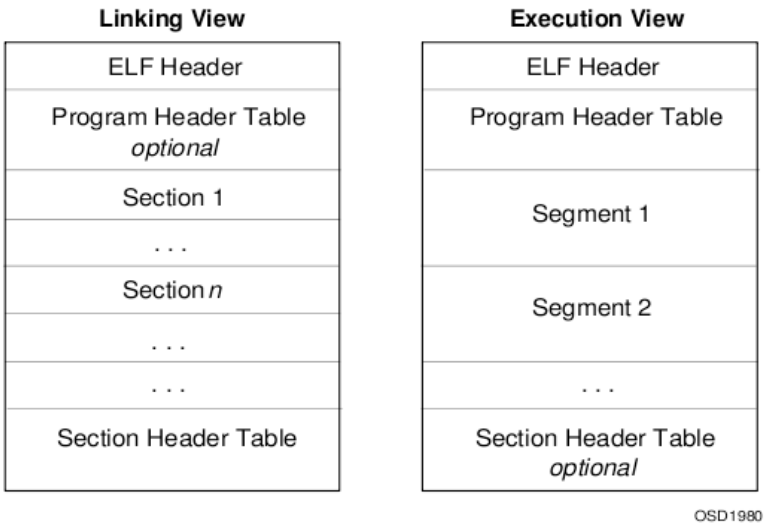


**FIG. 3.1 *Linking View* e *Execution View***

De maneira geral, o binário ELF apresenta um cabeçalho, que armazena informações gerais sobre o arquivo, tais como o tipo de arquivo (objeto compartilhado, biblioteca ou executável), arquitetura (MIPS, x86 68K etc.), versão, endereços de memória, etc; e uma área de dados que contém a tabela de cabeçalhos do programa, a qual armazena a informação do *offset* em que um segmento inicia e termina; a tabela de cabeçalho das seções, que especifica as seções do arquivo propriamente ditas (.text, .data ou .bss, por exemplo), bem como as seções dentro de cada segmento, quando for o caso. As seções armazenam bytes que podem ser códigos, dados ou comentários. Podem existir, entretanto, bytes que não pertencem a nenhuma seção, quando isso acontece esses bytes são intitulados bytes órfãos. A tabela de cabeçalhos do programa é utilizada pelo SO, mais especificamente, para carregar o programa executável em memória.

# 4 FERRAMENTAS DEBUGGER

Uma ferramenta *debugger* é um programa de computador que é utilizado para testar e depurar aplicativos de software. Depurar uma aplicação consiste em encontrar ou reduzir defeitos de execução dessa aplicação.

Esse tipo de ferramenta permite a interrupção da execução de instruções do programa que está sendo depurado em qualquer ponto, possibilitando a análise das variáveis. Além disso, comumente, os depuradores também oferecem funcionalidades como a suspensão do programa em pontos predefinidos, denominados pontos de parada, para que se possa examinar seu estado atual.

Um programa é abortado pelo sistema operacional quando tenta acessar, indevidamente, recursos indisponíveis para ele, como um espaço de memória inexistente ou bloqueado pelo mecanismo de proteção. Quando o programa aborta, o depurador mostra a posição correspondente no código-fonte original.

É importante ressaltar que as funcionalidades de um *debugger*, que o tornam útil e permitem eliminar defeitos, podem ser usadas por usuários mal-intencionados que queiram quebrar a proteção contra cópia de uma aplicação ou fazer um *crack* de software com alguma limitação de uso.

As principais ferramentas *debugger* são DBG, GNU Debugger (GDB), OllyDbg, Visual Studio Debugger. A ferramenta escolhida nesse trabalho como objeto de estudo, um pouco mais aprofundado, foi o OllyDbg.

## 4.1 OllyDbg

O OllyDbg é uma ferramenta *debugger* para arquivos binários já compilados, por exemplo os arquivos dos formatos PE e ELF descritos nos capítulos 2 e 3, respectivamente; sua utilidade se torna maior, portanto, quando da análise de aplicações cujo código-fonte não está disponível. O software traça registradores, reconhece procedimentos, chamadas de API, switches, tabelas, constantes e strings, bem como localiza rotinas de arquivos-objeto e bibliotecas. Além disso, o OllyDbg possui um

*disassembler* embutido a fim de decodificar as instruções. As ferramentas *disassembler* são utilizadas para transformar linguagem de máquina em linguagem *assembly* e serão discutidas, mais detalhadamente, no capítulo 5.

O software é gratuito, exigindo apenas que os usuários se registrem com o autor. Encontra-se na versão 2.0, recentemente lançada, na qual o OllyDbg foi reescrito do zero. Atualmente, a funcionalidade *disassembler* não está disponível para binários compilados para processadores de 64 bits.

A figura FIG. 4.1 a seguir mostra a interface do programa:

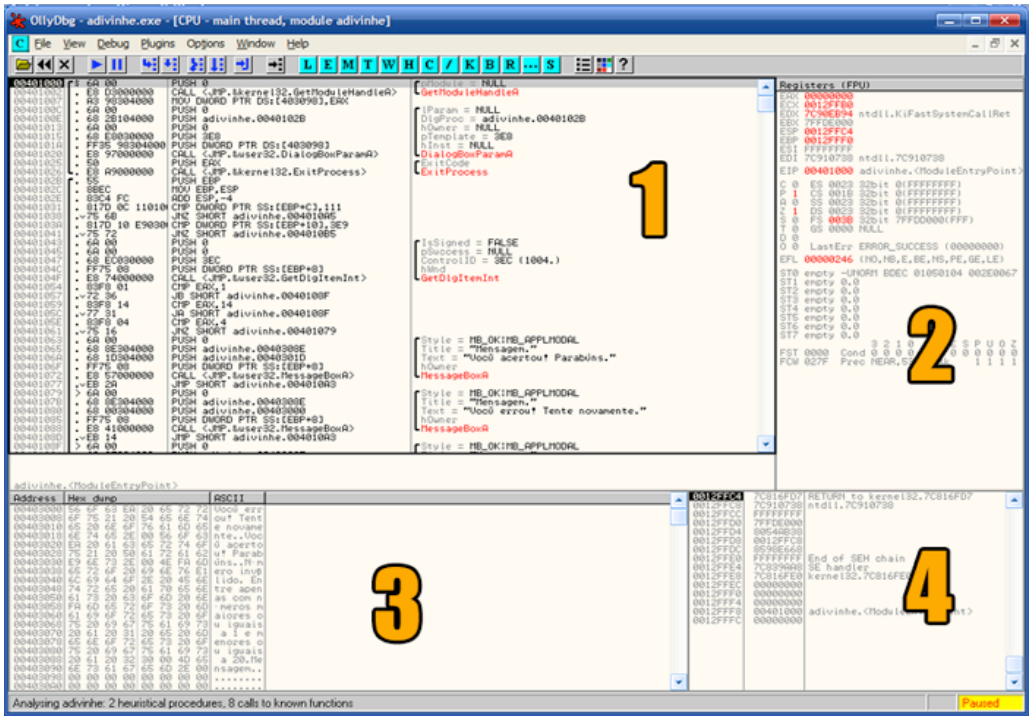


**FIG. 4.1 OllyDbg**

As regiões destacadas na figura FIG. 4.1 correspondem a:

1- Tela principal do OllyDbg, em que é apresentado o código do programa em linguagem de máquina;

2- Tela de exibição dos *flags* e registradores;

3- Memória física destinada ao aplicativo;

4- Estado atual da pilha.

# 5 FERRAMENTAS DISASSEMBLER

A principal funcionalidade de uma ferramenta *disassembler* é transformar linguagem de máquina em linguagem *assembly*. Alguns *disassemblers* utilizam as informações de depuração simbólica presentes em arquivos objetos, tais como o ELF.

Os *disassemblers* podem ser auto-seuficientes ou interativos. No primeiro caso, quando executado, o software gera um arquivo de linguagem *assembly* que pode ser examinado; no segundo caso, o usuário pode fazer alterações ao longo da execução e o software exibe os efeitos dessas modificações, permitindo que o usuário analise essas alterações e vá realizando outras ações enquanto a aplicação ainda está sendo executada.

A maioria dos *debuggers* (vide capítulo 4) possui um *disassembler* embutido para que possam interpretar as instruções resultantes da depuração.

As principais ferramentas *disassembler* são PE Explorer, WinGDB e IDA Pro, tendo sido esta última escolhida, neste trabalho, como objeto de um estudo mais detalhado. Cabe ressaltar que a versão estudada foi a *versão 5.0*, visto que esta é disponibilizada gratuitamente.

## 5.1 IDA Pro

Criado como um aplicativo shareware por Ilfak Guilfanov, o anteriormente denominado IDA foi posteriormente vendido como um produto comercial pela empresa belga DataRescue, que o aperfeiçoou e passou a vendê-lo com o nome IDA Pro. Entretanto, uma versão anterior (versão 5.0), com menos funcionalidades ainda é distribuída gratuitamente.

O software é um disassembler interativo para aplicações, que suporta diversos formatos de executáveis para diferentes processadores e sistemas operacionais. Pode ser utilizado, também, como debugger para PE, Mac OS X, Mach-O e ELF.

O IDA Pro disponibiliza, ainda, um *plugin* decompilador para programas compilados em compiladores C/C++. Além disso, o software executa a análise de

código, utilizando referências cruzadas entre seções de código, conhecimento de parâmetros de chamadas de API, entre outras informações. O usuário pode realizar operações para consertar códigos, inserir comentários, criar ou remover funções, estruturas e tipos, além de renomear objetos.

A figura FIG. 5.1 apresenta a tela inicial de análise de um executável do IDA Pro:

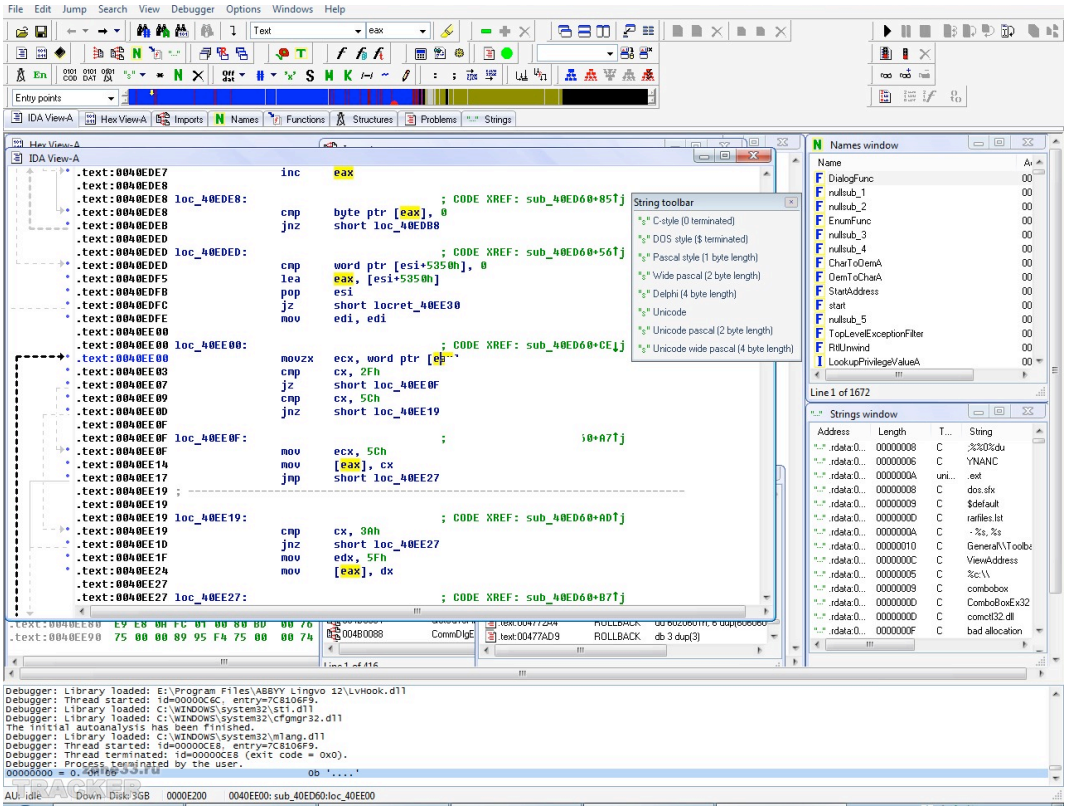


**FIG. 5.1 IDA Pro**

# 6 IMPLEMENTAÇÃO DO LEITOR PE

A implementação de um programa que faça a leitura de um arquivo PE está entre as principais etapas do trabalho, visto que conseguir extrair as informações de um arquivo binário é o primeiro passo da análise de um possível *malware*.

Torna-se fundamental, portanto, a total compreensão da estrutura de um arquivo PE a fim de que seja possível extrair todas as informações de maneira adequada e que facilite posterior análise.

O programa que faz a leitura de um arquivo do formato PE foi implementado na linguagem C#, utilizando a ferramenta gratuita Microsoft Visual C# 2010 Express para implementação do algoritmo e elaboração de uma interface para a saída dos dados para o usuário.

O programa consiste basicamente na leitura byte a byte de um arquivo binário *Portable Executable*, para isso foram utilizadas, até a etapa atual, 8 (oito) classes que correspondem as diferentes estruturas de um PE, mencionadas no capítulo 2. Outra classe (PEread) foi criada para, efetivamente, fazer a leitura do arquivo que estiver sendo objeto de análise. Criou-se também uma classe abstrata (EscreveArray) para formatar a saída das informações que são mostradas na interface do programa para o usuário.

A figura FIG. 6.1 na página a seguir apresenta o diagrama de classes do projeto do LeitorPE. Nas páginas a seguir são apresentadas as figuras FIG. 6.2, FIG. 6.3, FIG. 6.4, FIG. 6.5, FIG. 6.6, FIG. 6.7 e FIG. 6.8 que correspondem às diferentes janelas geradas pelo programa implementado a partir da leitura do arquivo no formato PE “notepad.exe”.

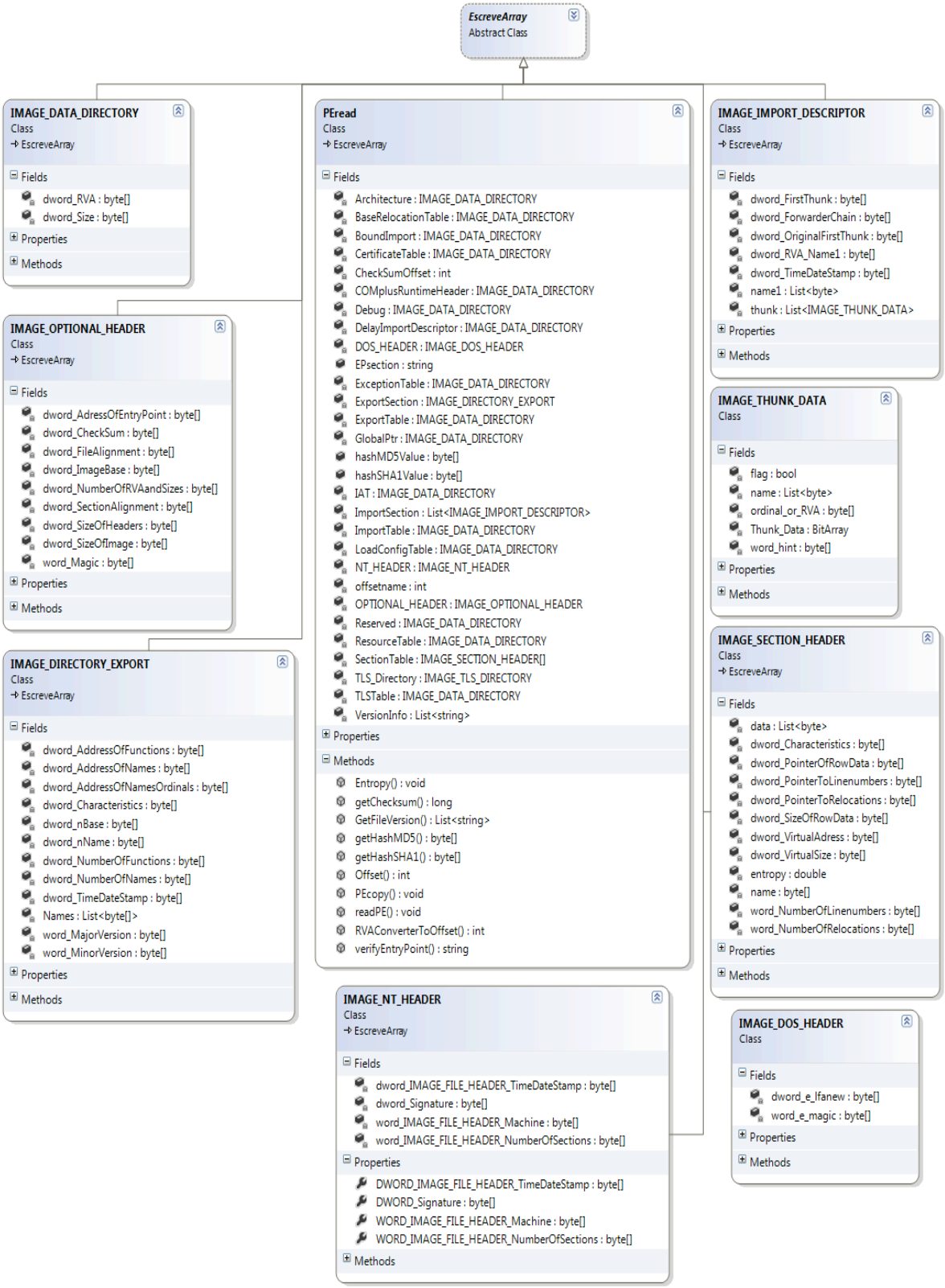


**FIG. 6.1: DIAGRAMA DE CLASSES DO LeitorPE**

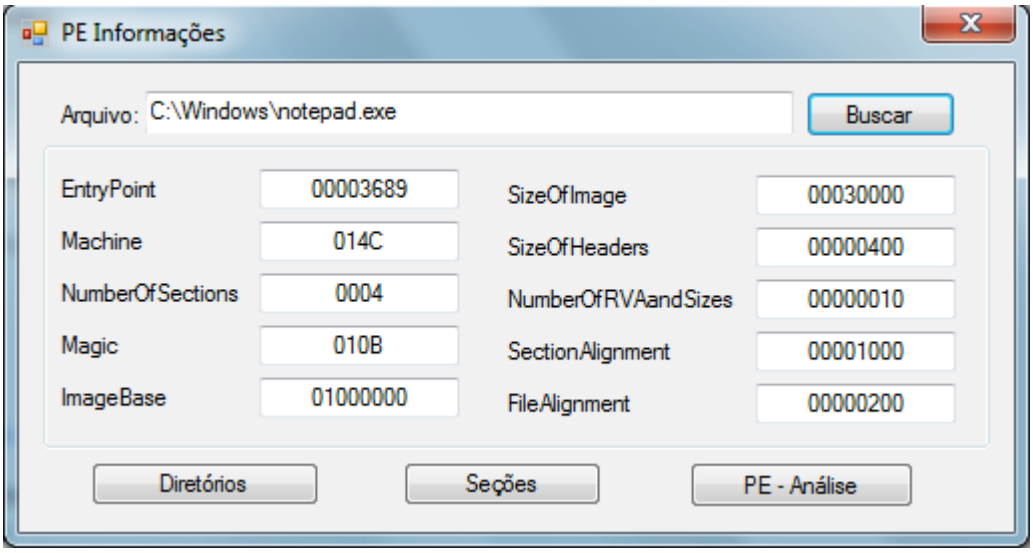


**FIG. 6.2 Tela inicial – Leitura do arquivo "notepad.exe"**

A figura FIG. 6.2 corresponde à tela inicial do programa, onde são exibidas as principais informações do arquivo binário analisado.

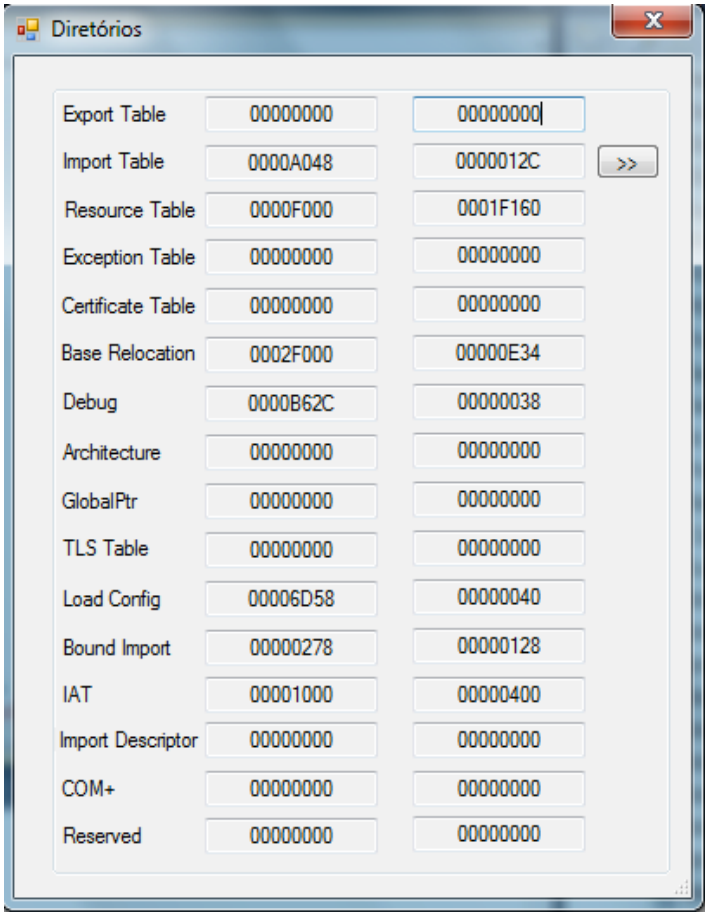


**FIG. 6.3 Diretórios**

A figura FIG. 6.3 corresponde à janela obtida a partir das informações dos diretórios do arquivo. Se um arquivo importa alguma *DLL (Dynamic Link Library)*, sua informação estará armazenada no diretório de importação (*Import Table* na figura anterior). A figura FIG. 6.4 mostra a janela do LeitorPE que lista as *DLL* importadas pelo “notepad.exe”.

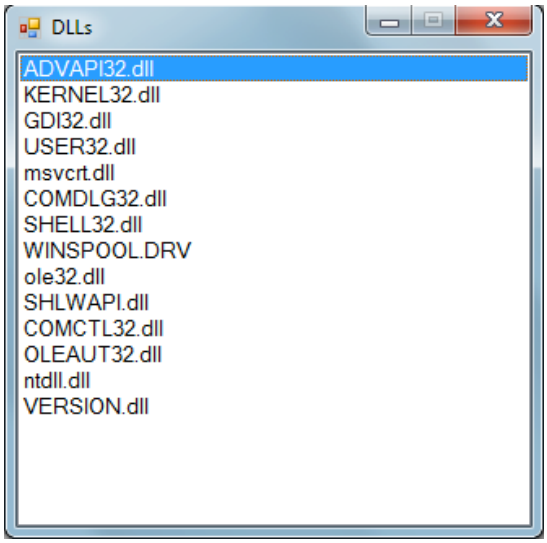


**FIG. 6.4 dll’s importadas pelo programa “notepad.exe”**

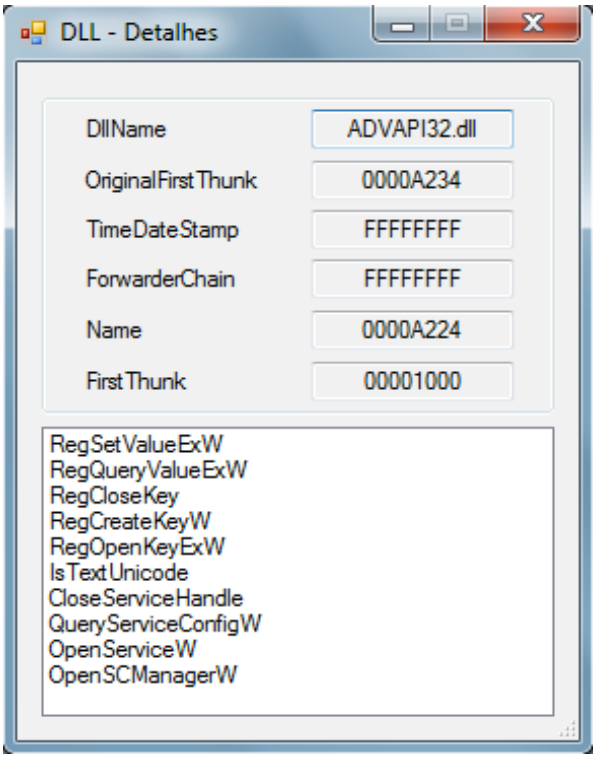


**FIG. 6.5 ADVAPI32.dll**

Cada *DLL*, por sua vez, possui informações próprias entre as quais destacam-se as *API (Application Programming Interface)* que são as funções chamadas pela respectiva *DLL*. Na figura FIG 6.5 acima, mostra-se a janela do LeitorPE que detalha as informações da "ADVAPI32.dll" e lista suas *API*.

A figura FIG. 6.6 mostra a lista de seções do arquivo "notepad.exe", de tal forma que ao posicionar o cursor em cada um dos nomes de seção são abertas as janelas correspondentes, como é mostrado na FIG. 6.7 para a seção *.text* .

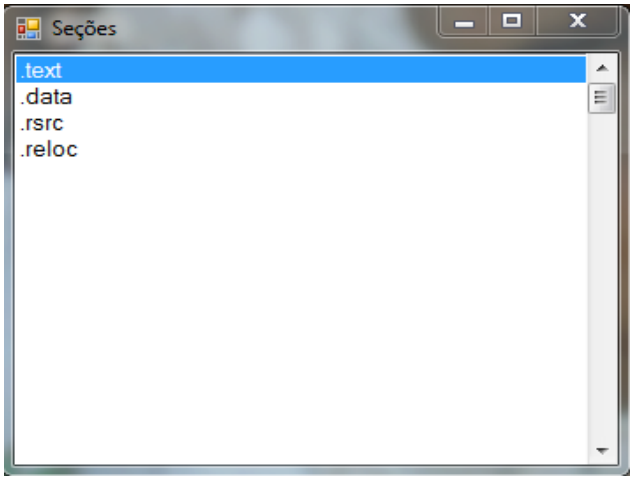


**FIG. 6.6 Lista de seções**

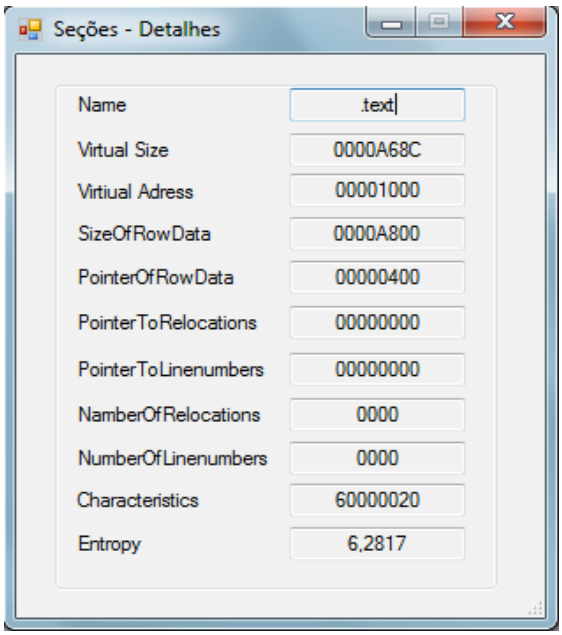


**FIG. 6.7 Seção .text**

# 7 ANÁLISE DAS INFORMAÇÕES EXTRAÍDAS DO ARQUIVO PE

A análise das informações extraídas do arquivo executável é a principal funcionalidade do LeitorPE, em virtude da grande importância da análise estática de *malware* já citada nos capítulos anteriores. Por essa razão dedicou-se um capítulo para tratar exclusivamente da análise.

Na tabela a seguir, são apresentados os principais atributos do que devem ser analisados em um arquivo executável para que se possa classificá-lo como malicioso, bem como as condições que determinam um comportamento suspeito do arquivo para cada um dos atributos.

| |
|---|
| **Ano < 1992 ou Ano > 2012** |
| **NumberOfSections<1 ou NumberOfSections>9** |
| **AdressOfEntryPoint (section)** |
| **CheckSum** |
| **Version Info** |
| **APIs suspeitas (Import Directory)** |
| **NumberOfRvaAndSizes !=16** |
| **Raw Size=0** |
| **Virtual Size/Raw Size>10** |
| **(Section Entropy >0 e Section Entropy<1) ou (Section Entropy>7)** |

**TAB. 7.1 Atributos analisados pelo LeitorPE (YONTS, 2012)**

Na figura FIG. 7.1 mostra-se a janela do programa que imprime para o usuário esses atributos. A figura FIG. 7.2 mostra o arquivo de texto que é gerado pelo programa como resultado da análise dos atributos.

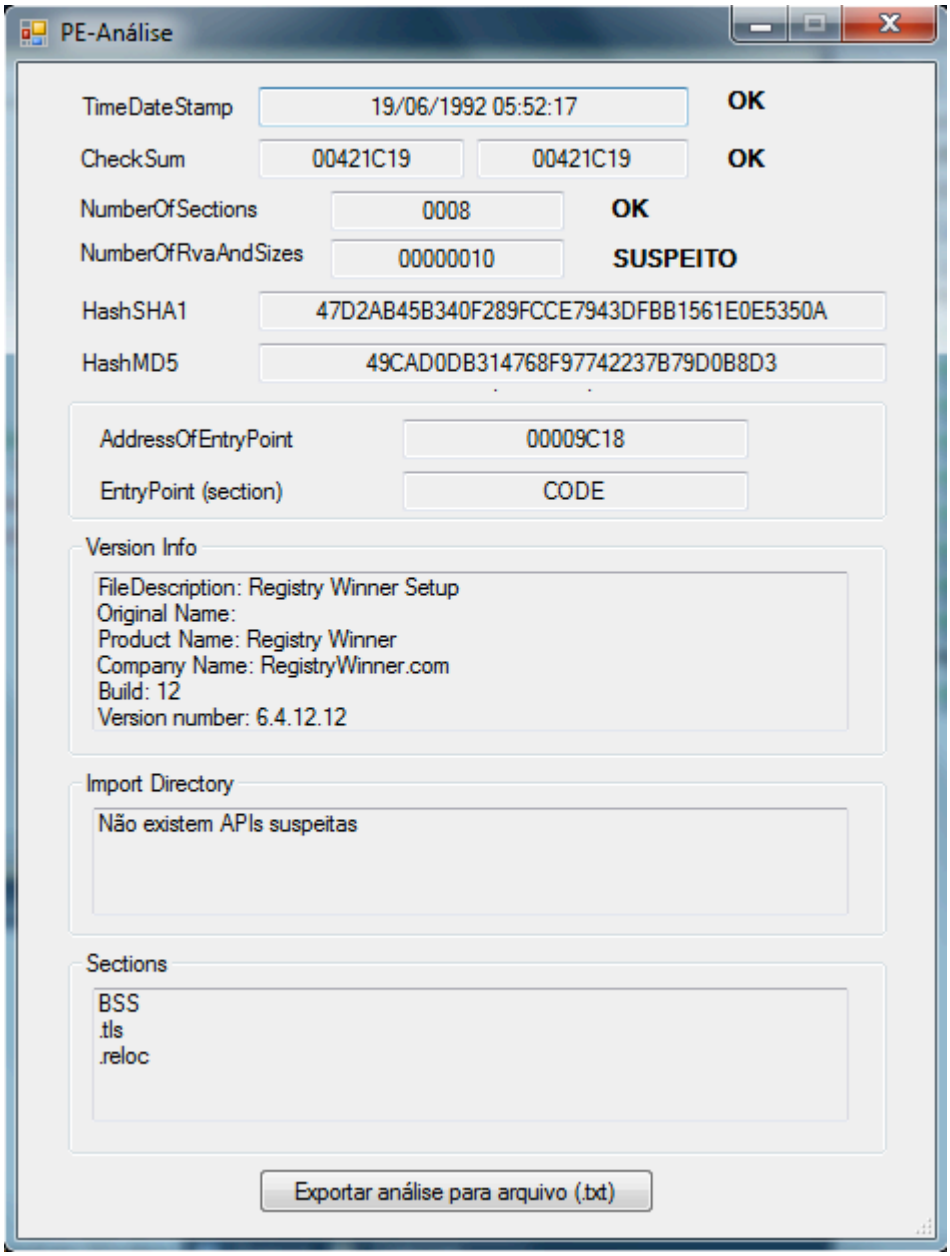


**FIG. 7.1 PE-Análise para um *malware***

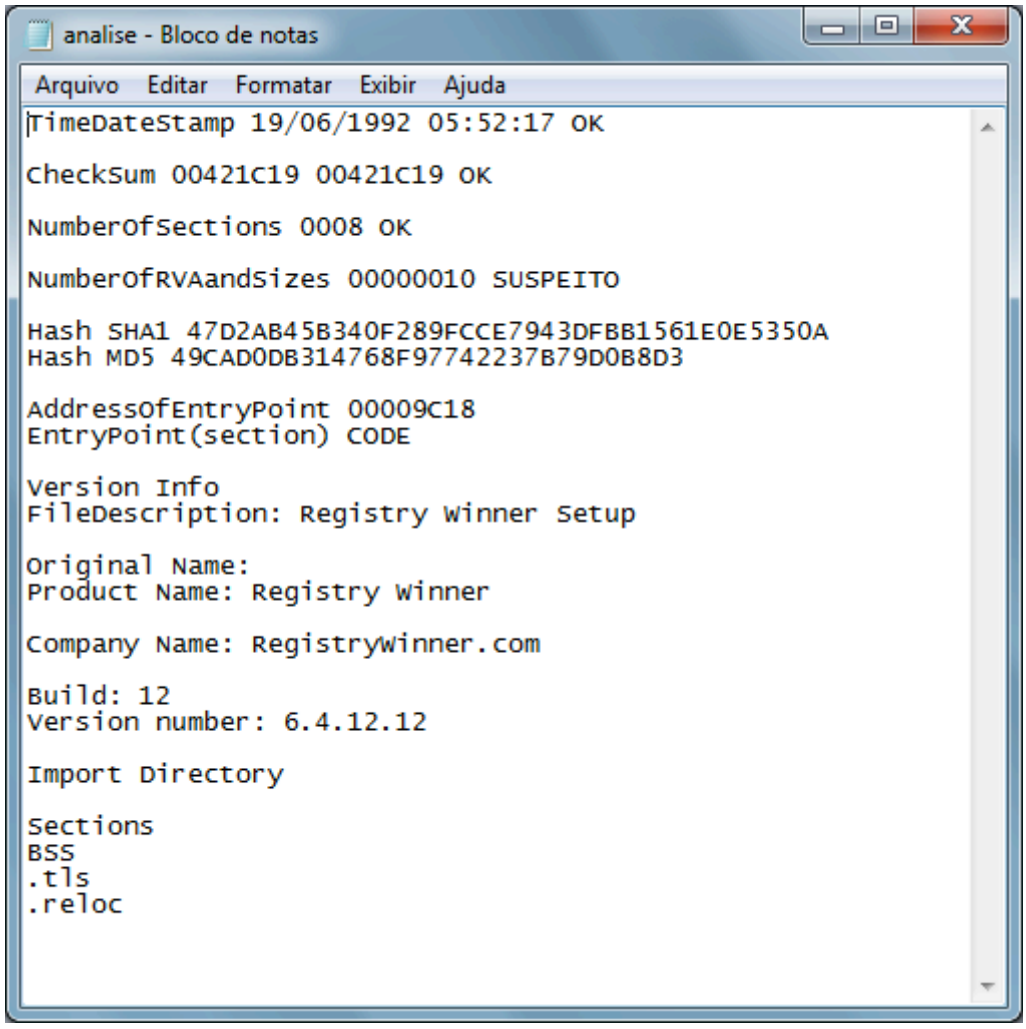


**FIG. 7.2 Arquivo de Texto com a Análise de um malware**

## 7.1 ENTROPIA

A entropia encontra-se entre os principais atributos de um arquivo executável para que devem ser analisados para classificá-lo como malicioso ou não-malicioso. Assim como as funções hash MD5 e hash SHA1 – também utilizadas no LeitorPE -, a entropia não pertence à estrutura padrão deste tipo de arquivo e deve ser calculada. (LYDA, 2007)

Esse atributo é uma medida da incerteza em séries de números ou bytes. De outra maneira, a entropia mede o nível de dificuldade ou a probabilidade de prever cada número, independentemente, em uma série e é utilizada para verificar se um dado foi

comprimido ou encriptado, isto é, permite a verificação de como o dado foi gerado. Podemos calcular a entropia de um evento aleatório discreto ‘x’ utilizando a fórmula a seguir:

$$H(x) = -\sum_{i=1}^{n} p(i) \log_2 p(i)$$

# 8 CONCLUSÃO

Nos últimos anos, a democratização das tecnologias de comunicação tais como emails, redes socias e a internet de uma maneira geral, propiciou a inclusão digital de milhares de habitantes do mundo inteiro e intensificou o processo de globalização. Entretanto, esta popularização dos serviços eletrônicos potencializou, também, o crescimento das ameaças virtuais como vírus, *worms* e outros agentes de software maliciosos.

Por essa razão, esse trabalho de pesquisa destinou-se ao estudo da estrutura de arquivos do formato *Portable Executable,* bem como à implementação de códigos-fonte que possibilitassem a verificação de alguns atributos desse tipo de arquivo, averiguando, dessa forma, a possibilidade de serem maliciosos.

O produto resultante dessa pesquisa é, portanto, uma ferramenta de engenharia reversa, capaz de fazer a análise estática de arquivos do formato .exe e .dll e gerar um arquivo de texto com os atributos analisados. Os atributos escolhidos para análise, neste trabalho, foram: *TimeDateStamp, NumberofSections, NumberOfRvaAndSizes, Checksum, AdressOfEntryPoint, EntryPoint Section, Sections(Entropy, RawSize, VirtualSize), APIs suspeitas (Import Directory) e Version Info*.

Sugere-se, dessa forma, que em trabalhos futuros uma quantidade ainda maior de atributos seja analisada, tais como as estruturas de recursos (*IMAGE_RESOURCE*) e de *callbacks (IMAGE_TLS_DIRECTORY)*. Além disso, é possível fazer uma busca, no executável, por assinaturas a fim de verificar se o arquivo foi empacotado por um *packer* conhecido. Outra sugestão interessante, é aliar a análise estática de *malware* gerada pelo LeitorPE a alguma técnica de análise dinâmica, aumentando, assim, a confiabilidade dos resultados a serem obtidos.

**Referências**


KATH, RANDY, **The Portable Executable File Format from Top to Bottom**, Microsoft Developer Network

DE ABREU, LUIZ E REZENDE, ROBERTO, **Protótipo de um Laboratório de Análise de Malware e Metodologia para Identificação de Códigos Hostis**, Rio de Janeiro, 2012.

MICROSOFT CORPORATION, **Microsoft Portable Executable and Common Object File Format Specification**, Revision 6.0, Fevereiro de 1999.

GOPPIT, **Portable Executable File Format – A Reverse Engineer View.** Code Breakers Journal, Security Analysis. VOL. 2, NO.3 2005, 15 de agosto de 2005.

YONTS, JOEL. ***Attributes of Malicious Files***, 30 de junho de 2012, disponibilizado por *Global Information Assurance Certification Paper.*

LYDA, ROBERT. **Using Entropy Analysisto Find Encrypted and Packed Malware**. Publicado por IEEE Computer Society, 2007.

PIETREK, MATT, **Peering Inside the PE: A Tour of the Win32 Portable Executable File Format**, Março de 1994, disponível em <http://msdn.microsoft.com/en-us/library/ms809762>, acessado em agosto de 2011.

DUMMER, DANIEL, **Windows Portable Executable** , Setembro de 2006, disponível em <http://www.devmedia.com.br/articles/viewcomp.asp?comp=857>, acessado em setembro de 2011.

YOUNGDALE, ERIC, **The ELF File Format by Dissection,** 1º de maio de 1995, disponível em: <http://www.linuxjournal.com/node/1060/print>, acessado em setembro de 2011.

O'LEARY, MICHAEL J., THE PORTABLE EXECUTABLE FORMAT, diponível em <http://www.nikse.dk/petxt.html#OVERVIEW> , acessado em setembro de 2011.

McKINLEY, DAN, A Word for OllyDbg, 11 de setembro de 2005, disponível em <http://mcfunley.com/181/a-word-for-ollydbg>, acessado em novembro de 2011.

TUTORIAL SOBRE ENGENHARIA REVERSA, disponível em <http://fergonez.net/tutoriais/fergo/reveng/tut1/tut_engrev1.html/>, acessado em dezembro de 2011. THE CODE PROJECT, **Reverse Disassembly**, 25 de agosto de 2004, disponível em <http://www.codeproject.com/Articles/4210/C-Reverse-Disassembly>, acessado em dezembro de 2011.

INFORMÁTICA DA ALDEIA, **O formato PE**, 11 de abril de 2008, disponível em <http://www.numaboa.com.br/informatica/oraculo/230-formatos/1096-formato-pe>, acessado em janeiro de 2012.

ICZELION'S WIN32 ASSEMBLY HOMEPAGE, disponível em <http://win32assembly.online.fr/pe-tut1.html>, acessado em fevereiro de 2012.

MOGRE, GAURAV, **The PE Format,** 8 de dezembro de 2008, disponível em <http://www.thehackerslibrary.com/?p=377>, acessado em março de 2012.

FREEBSD HANDBOOK, **Capítulo 3: Unix Básico – Formatos Binários**, 2012, disponível em <http://doc.fug.com.br/handbook/binary-formats.html>, acessado em março de 2012.

HEXSEL, ROBERTO A, **Formatos de Arquivos Objeto**, 2009, disponível em <http://www.inf.ufpr.br/roberto/ci064/ci064a18.pdf>, acessado em março de 2012.

PRADO, SERGIO, **Padrão ELF para arquivos-objeto**, 24 de julho de 2010, disponível em <http://sergioprado.org/2010/07/24/padrao-elf-para-arquivos-objeto/>, acessado em março de 2012.

COMUNIDADE HARDWARE.COM.BR, **Utilizando um debugger – OllyDbg**, disponível em <http://www.hardware.com.br/comunidade/utilizando-debugger/785195/>, acessado em março de 2012.

CARRERA, E., ***pefile.py,*** 2012, disponível em <http://code.google.com/p/pefile/w/list>, acessado em junho de 2012.

CARRERA, E., ***pescanner.py,*** 2012, disponível em <http://code.google.com/p/pefile/w/list>, acessado em junho de 2012.