CENTRO UNIVERSITÁRIO FEEVALE

DANIEL ROBERTO DUMMER

ENGENHARIA REVERSA APLICADA À EXPLORAÇÃO E PROTEÇÃO DE SOFTWARE

Novo Hamburgo, junho de 2007.

DANIEL ROBERTO DUMMER

ENGENHARIA REVERSA APLICADA À EXPLORAÇÃO E PROTEÇÃO DE SOFTWARE

Centro Universitário Feevale
Instituto de Ciências Exatas e Tecnológicas
Curso de Ciência da Computação
Trabalho de Conclusão de Curso

Professor Orientador: Carlos Sérgio Schneider

Novo Hamburgo, junho de 2007.

RESUMO

Com a total adesão da informática tanto nas esferas industrial e comercial quanto na pessoal, cresce diária e exponencialmente a quantidade de dados, sigilosos ou não, armazenados nos computadores ao redor do mundo. Junto a esse crescimento acompanha também o de *hackers* mal-intencionados que se utilizam de profundos conhecimentos de computadores e sistemas para roubar estes dados. Somadas a essa espionagem digital, todos os dias novas brechas de *software* e novos vírus aparecem, *spywares* e *malwares* são criados, ameaçando a privacidade e afetando a segurança destes. Para tanto, é confiado em *patches* e atualizações diárias de sistema, *Firewalls*, Anti-vírus e *IDS*, dentre vários recursos, sem muitas vezes saber para que servem, nem o que protegem e/ou solucionam. No entanto se, durante o desenvolvimento destes sistemas, fosse destinada uma atenção maior referente à segurança, não seria necessário investir tanto tempo, dinheiro e trabalho na correção de *bugs*. Uma simples analogia ao ditado popular reflete bem essa situação: "É melhor prevenir a remediar". A criação de aplicativos de qualidade superior evitaria a maior parte destas ferramentas reparatórias, de filtragem ou detectoras de acesso indevido ou mal-intencionado aos nossos dados. A engenharia reversa de *software* auxilia o desenvolvedor a entender como o *software* funciona internamente permitindo, a partir do produto final, estudar e aprender sua estrutura e lógica, além de auxiliar a efetuar testes de *softwares* mais profundos, obter o conhecimento perdido em casos de indisponibilidade ou perda do código-fonte, incluir e alterar funções de um *software* pronto e melhorar a qualidade do *software* tanto na questão segurança quanto desempenho. Assim sendo, este trabalho tem por objetivo estudar a engenharia reversa de *software* demonstrando, a partir de suas técnicas de monitoramento e análise, o funcionamento de *softwares* de formato *Win32/Portable Executable* sob o sistema operacional *Windows*, além de apresentar os tipos mais comuns de ataques à *software* e suas respectivas proteções, além de discutir os aspectos legais e éticos na aplicação da engenharia reversa.

Palavras-chave: Engenharia de Software; Segurança de Computador; Proteção de Software; Exploração de Software.

ABSTRACT

The entire adoption of information systems such in industrial, commercial and personal way's, rises every day exponentially the amount of information, confidential or not, stored in the computers around the world. Besides these, grow up the number of bad-intentioned hackers who'll use all their deep computer knowledge's to steal this information. Added to this digital espionage, every day new software's vulnerabilities and viruses appear, spywares and malwares are created, haunting these data and affecting the confidence of its proprietors about its security. So, to protect theses data, are trusted in system's patches and daily updates, Firewalls, Anti-virus and IDS, amongst some other tools and resources, without not knowing for what that they serve, nor what they protect or solve. However if, during the development of these systems, was destined bigger attention to the security, would not be necessary to invest so much time, money and work, in these bugs correction's. A simple analogy to the popular dictated one reflects well this situation: "It is better to prevent then to cure". The creation of superior level applications would prevent the most of these fixing or filtering tools, also system of intrusion or bad-intentioned user detection, who try to access illegally our data. The Software's Reverse Engineering helps the developer to understand how work internally software, allowing to study and learn the software's structure and logic analyzing only the final product, also assist to run deeper tests of software, get lost knowledge in non-availability cases or losses of source code, include and modify functions to the software and improve software's quality such in security like in performance ways. So, this work has for objective to study the software's reverse engineering, analyzing its functioning and, in specific, the Win32/Portable Executable file format, under the operational system Windows, assisting in the understanding and agreement of its structure and functioning. To demonstrate the techniques of reverse engineering for extraction, monitoring and information's data analysis, also demonstrate the most common types of attacks and techniques of protection, and arguing the legal and ethical aspects of this technique.

Key-words: Software Engineering; Computer Security; Software Protection; Software's Exploitation.

LISTA DE FIGURAS

Figura 1.1 – Esquema engenharia 15

Figura 1.2 – Esquema engenharia reversa 15

Figura 2.1 – Definição e disposição dos registradores 31

Figura 2.2 – Análise da pilha após inserção de 3 valores 33

Figura 2.3 – Análise da pilha após exclusão de 3 valores 34

Figura 2.4 – Estrutura do formato PE 40

Figura 2.5 – Cabeçalho DOS 42

Figura 2.6 – Cabeçalho do arquivo 44

Figura 2.7 – Cabeçalho opcional 48

LISTA DE QUADROS

Quadro 2.1 – Descrição dos registradores e suas relativas funções ____________________ 32

Quadro 2.2 – Exemplos de referências de operandos de instruções e seus significados ____ 35

Quadro 2.3 – Instruções para as operações aritméticas básicas _______________________ 36

LISTA DE TABELAS

Tabela 2.1 – Instruções por função____ 27

LISTA DE ABREVIATURAS E SIGLAS

COFF	*Common Object File Format*
CPU	*Central Processing Unit*
DLL	*Dynamic Link Library*
DoS	*Denial of Service*
ER	Engenharia Reversa
GUI	*Graphical User Interface*
IA	Inteligência Artificial
IA-32	*Intel's 32-bit architecture*
IDS	*Intrusion Detection System*
JVM	*Java Virtual Machine*
LIFO	*Last In, First Out*
LOC	*Lines Of Code*
MSIL	*Microsoft Intermediate Language*
OEM	*Original Equipment Manufacturer*
PE	*Portable Executable*
RAM	*Random Access Memory*
RVA	*Relative Virtual Address*
TCP	*Transmission Control Protocol*
UDP	*User Datagram Protocol*

Win32 *Windows 32 bits*

SUMÁRIO

INTRODUÇÃO **12**
1 ENGENHARIA REVERSA **15**
1.1 Importância da aplicação da Engenharia Reversa 16
1.2 História 17
1.3 A Origem dos Problemas 18
1.3.1 Complexidade 18
1.3.2 Extensibilidade 19
1.3.3 Conectividade 20
1.4 Análise de *Software* 21
1.4.1 Análise de Caixa-Branca 21
1.4.2 Análise de Caixa-Preta 22
1.4.3 Análise de Caixa Cinza 23
1.4.4 Qualidade na análise 23
1.5 Segurança de *Software* 23
1.5.1 Terminologia 24
1.6 Engenharia Reversa versus *Cracking* 24
2 SISTEMAS E FORMATOS **26**
2.1 Linguagens de Programação 26
2.2 *Assembly* 28
2.2.1 Gerenciamento de dados em baixo nível 29
2.2.2 Registradores 30
2.2.3 Identificadores (*Flags*) 32
2.2.4 Pilha (*Stack*) 32
2.2.5 Interrupções 34
2.2.6 Instruções 34

2.2.6.1 Movimentação de dados 35
2.2.6.2 Manipulando a pilha 35
2.2.6.3 Operações aritméticas 36
2.2.6.4 Comparação de dados 37
2.2.6.5 Execução condicional 37
2.2.6.6 Chamada de Funções 37
2.2.6.7 Outras funções 38
2.3 Formato *Win32/Portable Executable* 38
2.3.1 História 38
2.3.2 Formato *Portable Executable* 39
2.3.2.1 Cabeçalho *DOS* 40
2.3.2.2 Fragmento (*Stub*) do *DOS* 43
2.3.2.3 Cabeçalho do Arquivo 43
2.3.2.4 Cabeçalho Opcional 45
2.3.2.5 Cabeçalho de Seções 49
2.3.2.6 Seções 49
2.3.3 Considerações Gerais 49
3 FERRAMENTAS 51
3.1 Depuradores (*Debuggers*) 52
3.2 *Disassemblers* 53
3.3 Descompiladores ou compiladores reversos 53
3.4 Ferramentas de Monitoramento de Sistema 53
3.5 *Dumping* 55
3.6 Ferramentas de Patching - Editores Hexadecimais 55
3.7 Outras ferramentas - PEVIEW 56
CONCLUSÃO 57
REFERÊNCIAS BIBLIOGRÁFICAS 58

INTRODUÇÃO

Cresce diariamente a importância do quesito segurança nos *softwares*. As estatísticas comprovam: no segundo semestre de 2006, a *Symantec* identificou 2.526 novas vulnerabilidades, o maior índice desde o início das medições, iniciadas em janeiro de 2002. Este número superou em 12% o índice do semestre anterior. Deste total de vulnerabilidades, 1.667 (66%) afetam aplicativos *Web*, sendo 1.150 (69%) deles de fácil exploração e 1.081 (64%) de fácil exploração e exploráveis remotamente. Doze vulnerabilidades dia-zero[1], um enorme salto comparado a apenas uma do semestre anterior. Porém não são somente aplicativos *Web* que possuem vulnerabilidades. Como exemplo, neste mesmo período foram identificadas 168 vulnerabilidades no banco de dados *Oracle*. (Symantec Internet Security Threat Report XI, 2007)

"Quase todos os sistemas modernos têm o mesmo calcanhar-de-aquiles, que é o *software*". (HOGLUND; McGRAW, 2006, p.1)

Além disso, o custo, trabalho e dinheiro gastos nessas correções também impressionam. No mesmo período pesquisado anteriormente, todas as empresas desenvolvedoras analisadas pela *Symantec* informaram que foi gasto muito mais tempo desenvolvendo patches de correção que no semestre anterior. Algumas delas, como a *Sun*, por exemplo, com seu sistema operacional *Solaris*, teve um tempo médio de 122 dias para desenvolvimento de seus *patches* - o maior dentre a categoria sistemas operacionais -, contra os 21 dias necessários pela *Microsoft*, o menor analisado na categoria. (Symantec Internet Security Threat Report XI, 2007)

[1] Vulnerabilidade divulgada antes que haja correção disponível para a mesma.

“Não seria necessário empregar tanto tempo, dinheiro e trabalho em segurança de rede se não tivéssemos uma segurança de *software* tão ruim”. (VIEGA; McGRAW apud HOGLUND; McGRAW, 2006, p. 1)

Diante desses índices, para proteção dos dados, faz-se necessário apelar para programas “repressivos” como anti-vírus, *firewall* e outros, quando se poderia, simplesmente, utilizar *softwares* mais seguros.

A engenharia reversa apresenta técnicas para estudar o funcionamento interno do sistema e como ele realmente interage junto à máquina via monitoramento de instruções e *flags* (indicadores). Para isto será estudado o seu conceito e as técnicas de exploração do *software*, assim como o funcionamento interno do sistema *Windows*, memória e processamento, além do formato de arquivo *Win32/Portable Executable*.

Serão apresentadas as ferramentas disponíveis no mercado para auxílio na extração, estudo e monitoramento de *software*, assim como os tipos mais comuns de ataque e técnicas de proteção e prevenções. Serão discutidas as questões éticas incidentes na reversão de *software* e como as leis de diversos países lidam com o assunto.

A demonstração dos tipos mais comuns de ataques tem por finalidade alertar sobre erros comumente implementados pelos desenvolvedores. Serão demonstrados os problemas de implementação, sintaxe e tratamento de erros, e ainda sugestões para correção dos mesmos.

A engenharia reversa é útil também na aquisição de conhecimento perdido nos casos onde ocorre a perda ou indisponibilidade do código-fonte de um *software*. Utiliza-se das técnicas de engenharia reversa para se estudar o funcionamento do mesmo a partir do produto final, ou seja, do *software* pronto.

A melhor compreensão e conhecimento interno do *software* capacitam o desenvolvedor na escrita de códigos mais seguros e otimizados, desenvolvendo *softwares* de maior qualidade, já que a aplicação e utilização das técnicas exigem um nível considerável de conhecimento sobre o funcionamento do sistema e da estrutura dos arquivos. Isto qualifica tanto o *software* quanto o profissional.

Devido à rara bibliografia nacional especializada e à escassez de material de consulta referente ao assunto, será desenvolvido e apresentado um trabalho de revisão sobre

engenharia reversa aplicada a *software*, com objetivo de divulgação do assunto. A disseminação deste tende a criar o interesse de um maior número de pesquisadores e desenvolvedores, incentivando o desenvolvimento de novas técnicas e auxiliando na descoberta de novas vulnerabilidades nos sistemas existentes, além de possibilitar um compartilhamento de experiências e, por conseguinte, um aprimoramento geral na qualidade dos *softwares*.

Dada a importância destes aspectos, este trabalho tem como objetivo geral apresentar o conceito de engenharia reversa, demonstrando as ferramentas de auxílio de extração, realizando um estudo da estrutura, funcionamento e monitoramento de *software*, além de uma apresentação dos tipos mais comuns de ataque e técnicas de proteção e prevenção, analisados todos sob funcionamento no sistema operacional *Windows* e arquivos executáveis no formato *Win32/Portable Executable*.

Como objetivos específicos, podem ser destacados:

- Apresentar o conceito, história, origem, importância e utilidade da engenharia reversa;
- Analisar o sistema operacional *Windows* e o formato de arquivos *Win32/Portable Executable*;
- Apresentar as ferramentas de auxílio de extração de informações, funcionamento e estrutura do *software*;
- Apresentar os tipos de ataques mais comuns;
- Apresentar os cuidados e técnicas de proteção anti-reversão;
- Discutir a questão ética e aspectos legais da aplicação de engenharia reversa.

Este trabalho será composto de 3 capítulos. No primeiro capítulo será apresentado o conceito e importância da Engenharia Reversa de *Software*, demonstrando o conceito, a origem e sua história, além de discutir o que é segurança de *software*, quais os tipos de análise e as origens dos problemas de *software*. No segundo capítulo será apresentado o formato de arquivo *Win32/Portable Executable*, sua estrutura e funcionamento no ambiente *Windows 32 bits*, além de debater sobre os tipos e níveis das linguagens de programação atuais, em especial a linguagem *Assembly*. Por fim, no terceiro capítulo serão relacionadas as ferramentas da Engenharia Reversa, definindo os tipos e apresentando alguns utilitários como exemplos.

1 ENGENHARIA REVERSA

Segundo Aurélio (2004), Engenharia é a "arte de aplicar conhecimentos científicos e empíricos e certas habilitações específicas à criação de estruturas, dispositivos e processos que se utilizam para converter recursos naturais em formas adequadas ao atendimento das necessidades humanas", ou seja, aplicar métodos e conhecimentos sobre a matéria-prima, de qualquer espécie, com a finalidade de elaboração de produtos mais sofisticados (manufatura).

Figura 1.1 – Esquema engenharia

O conceito de engenharia reversa é o mesmo, porém logicamente, inverso. Partindo da manufatura, esta é decomposta e detalhadamente analisada na finalidade de conhecer, decifrar e entender seus componentes, funcionamento e organização.

Figura 1.2 – Esquema engenharia reversa

Na informática, a Engenharia Reversa é a obtenção de conhecimento sobre o funcionamento e estrutura do *software* a partir de sua exploração e descompilação, resultados estes expressados em forma de código-fonte. Sobre este código-fonte, o engenheiro tem a possibilidade de criar novos programas baseados no conhecimento adquirido ou modificar o programa existente, tanto para incremento de funcionalidade quanto correção de *bugs*.

Algumas outras razões para utilização da Engenharia Reversa são:

- Aquisição de conhecimento;
- Correção de defeitos de *software* e produtos de terceiros;

- Incremento de funcionalidade em *software* e produtos de terceiros;
- Adaptação de sistemas para compatibilidade de comunicação e compartilhamento de informações;
- Verificação e detecção de roubo ou reutilização de código-fonte próprio por outras empresas em seus aplicativos;
- Recuperação de dados comerciais ou pessoais codificados em arquivos de formato próprio de arquivos;
- Análise de vírus, *trojans* e *malwares* em geral.

Além de outras razões, a Engenharia Reversa normalmente é utilizada para obtenção de conhecimento, idéias e design perdidos ou indisponíveis. Em alguns outros casos, o engenheiro pode aplicar seus conhecimentos para edição de *software* de terceiros quando estes não estão dispostos a fornecer atualizações ou o código-fonte dos mesmos.

1.1 Importância da aplicação da Engenharia Reversa

Conforme Hoglund e McGraw (2006), a segurança de *software* é considerada somente como um problema de Internet, porém isso está longe de ser verdade. Boa parte das vulnerabilidades de sistemas de Internet provém de falhas de *software*. A Internet acaba sendo apenas uma facilitadora, devido à conectividade que ela proporciona.

O relatório semestral da *Symantec*, *Internet Security Threat Report XI* (2007), que analisou as ameaças e ataques na Internet durante o segundo semestre de 2006, demonstrou os dados estatísticos referentes à segurança de alguns *softwares* como, por exemplo, navegadores de Internet. Neste relatório foram documentadas 54 vulnerabilidades que atingiram o *Microsoft Internet Explorer* e 40 vulnerabilidades que atingiram o *Mozilla Firefox*. A janela média para correção destes *bugs* foi de 47 dias, ou seja, cada *bug* teve 47 dias para ser explorado, colocando em risco a estabilidade e segurança dos seus usuários.

Mas não apenas sistemas conectados apresentam vulnerabilidades. De acordo com Hoglund e McGraw (2006), um bom exemplo ocorreu com uma das primeiras edições do *Windows Server*. Um *hacker* analisou, via Engenharia Reversa, a biblioteca *run32.dll* e descobriu que a variável que controlava o número e seqüência das janelas abertas do *Prompt* de Comando MS-DOS era um *byte* (variável de 1 *byte* - 8 *bits* - que pode armazenar valores que variam de 0 a 255). Assim, foi realizado um teste excedendo-se esta faixa e pôde

comprovar-se que, após abertas 257 janelas do *Prompt* de Comando, o sistema travar-se-ia. Isto ocorrera porque este valor ficara fora da faixa de armazenamento da variável, fato não previsto na construção e modelagem do sistema.

Além de comprometer a estabilidade dos sistemas, os gastos com desenvolvimento, correções e os períodos de inatividade ou indisponibilidade destes sistemas podem ser exorbitantes. Uma simples falha de *software* custou aos contribuintes americanos cerca de US$ 165 milhões quando o *Mars Lander* da NASA colidiu com a superfície de Marte. Conforme destaca o próprio relatório da NASA (1999), o problema foi um simples erro de tradução computacional entre as unidades de medida do sistema inglês e o sistema métrico. Como resultado do *bug*, houve um erro significativo na trajetória da espaçonave conforme esta se aproximava de Marte, fazendo com que fosse desativado os motores de descida antes da hora, resultando em um acidente.

Outro importante ganho com o estudo e familiarização das técnicas de Engenharia Reversa é o próprio conhecimento proporcionado, que promove o engenheiro auxiliando-o numa melhor compreensão não só do sistema, mas tornando-o apto para o destrincho de aplicativos e componentes, estudo e aprendizagem de processos pela via inversa, partindo de obras prontas. Esse diferencial pode ser um fator muito interessante na vida profissional de um analista ou programador de sistemas, abrindo substancialmente seu leque de aptidões.

1.2 História

Espionagem industrial? Espionagem militar? Denning (1998) afirma que estas e muitas outras razões impulsionaram a prática através dos tempos, destacando ainda que o conceito de Engenharia Reversa é conhecido muito antes do advento da informatização, datado ao período da Revolução Industrial onde eram dissecadas máquinas para estudo minucioso de suas peças, compreendendo seu funcionamento e permitindo assim a clonagem de equipamentos.

No entanto Denning (1998) recorda que antes disso, porém, a guerra de informações já promovia técnicas e atitudes similares para aquisição de conhecimento e informação. A guerra, mesmo sendo a segunda profissão mais velha do mundo, tem seu representante moderno e cibernético:

> "A guerra de informações é essencial a cada nação e corporação que pretende prosperar (e sobreviver) no mundo moderno. Mesmo se uma nação não estiver construindo a potencialidade da Guerra de Informação, pode-se assegurar que seus inimigos estão, e que a nação estará em uma desvantagem distinta para guerras futuras." (DENNING, 1998, p. 42) (Tradução nossa)

Nações e instituições monitoravam e espionavam uma as outras, no intuito de adquirir (ou roubar) sua tecnologia, ressalta Denning (1998). De um certo ponto de vista, isto não difere da Engenharia Reversa, já que esta pode ser aplicada com a mesma finalidade.

Já Szor (2005) lembra que, na informática, a engenharia reversa entrou em cena no início dos anos 80 com a quebra de proteção de jogos de computadores, na época escritos para o *Apple II*. Embora esta técnica se transformasse rapidamente em uma maneira de distribuir *softwares* de computador pirateados, um núcleo dos programadores continuou pesquisando e desenvolvendo novas técnicas e ferramentas na área puramente por questões acadêmicas.

1.3 A Origem dos Problemas

Robustez, produtividade e confiança deveriam ser sinônimos de *software*, pois são estes alguns dos benefícios almejados na compra e escolha de um *software*. Porém, nem sempre isto acontece. Passado o deslumbre das qualidades prometidas na compra, muitas vezes são encontrados *softwares* mal-projetados e mal-implementados que, além de não cumprirem seus propósitos, comprometem o sistema afetando a estabilidade e expondo recursos antes não vulneráveis.

Conforme Hoglund e McGraw (2006), as raízes dos problemas de segurança de *software* derivam de três fatores:

- Complexidade;
- Extensibilidade;
- Conectividade.

1.3.1 Complexidade

À medida que avança a tecnologia, avança também o grau de complexidade e inteligência artificial dentro dos sistemas. As atribuições do *software* vão muito além de processamento de fórmulas, envio de e-mails ou navegação pela Internet. Um *software*

moderno é complexo e repleto de condições e regras a serem seguidas. E, na medida da evolução natural dos *softwares*, cada vez maior será a complexidade dos mesmos.

Um breve exemplo demonstrado por BRAND (1995) relata que, em 1983, o *Microsoft Word* possuía somente 27 mil de *LOC* (*Lines Of Code* – Linhas de código). Já em 1995, essa contagem chegava a 2 milhões de *LOC*.

No entanto BRAND sugere:

> "Peça para alguém escolher entre a versão de 1982, com índice zero de defeitos, ou a nova versão, repleta de novos recursos, porém com diversos *bugs*? Com certeza todos escolherão a nova. Só espero que estes nunca me convidem para viajar em seus aviões, controlados pela mais recente versão do *software* de controle aéreo, pilotado por algum deles."
>
> (BRAND, 1995) (Tradução nossa)

E essa relação *LOC* versus *bugs*, justifica a afirmação de que, quanto maior a complexidade, maior a probabilidade de ocorrência de *bugs*. Estima-se que o número de *bugs* de um sistema varia de 5 a 50 *bugs* por *KLOC* (mil *LOC*).

Porém imagine isto em sistemas operacionais como *Windows* 95 que possuía 5 milhões de *LOC*, o *Linux* que possuía 1,5 milhões de *LOC* ou o *Windows XP* que possui cerca de 40 milhões de *LOC*?

1.3.2 Extensibilidade

Os sistemas extensíveis são aqueles construídos baseados em máquinas virtuais que preservam a segurança e executam verificações de segurança de acesso em tempo de execução. Dois exemplos deste modelo são o *Java* e o *.NET*.

A maioria dos sistemas operacionais dá suporte à extensibilidade por meio de *drivers* de dispositivos e módulos dinamicamente carregáveis, aceitando atualizações ou extensões, também chamadas de códigos móveis. Como exemplo temos o *JVM* (*Java Virtual Machine*), que permite instanciar uma classe em um *namespace* e pode permitir que outras classes interajam com ela.

Porém esta própria característica dos sistemas extensíveis dificulta a segurança, pois é difícil impedir que um código malicioso penetre como uma extensão indesejável. Assim, estes sistemas devem ser projetados levando em conta a segurança na agregação de novos

recursos (como mecanismo de carregamento de classes do *Java*), bloqueando assim a execução de código móvel não confiável.

1.3.3 Conectividade

A crescente conectividade dos computadores por meio da Internet aumentou, significativamente, as vias para exploração de falhas e assaltos de informações assim como facilitou o trabalho para os invasores, observam Hoglund e McGraw (2006). Estas conexões variam desde PCs domésticos a servidores que controlam infra-estruturas críticas. O alto grau de conectividade torna possível que pequenas falhas se propaguem e causem grandes danos.

Como o acesso na rede não requer obrigatoriamente intervenção humana - podendo este ser programado e/ou utilizando redes zumbis[2] - é possível programar e acionar ataques automatizados à sites, empresas e intuições, alertam Hoglund e McGraw (2006). Esta possibilidade muda definitivamente o panorama da segurança, visto o poder da ação e a relativa facilidade de elaboração e execução. O escopo é mundial, potencializando o alastramento e o grau de atividade na rede.

Redes altamente conectadas estão particularmente vulneráveis a estes ataques. É o paradoxo da conectividade: Alta conectividade é o mecanismo clássico para incremento de disponibilidade e compartilhamento de informações, porém é também o mecanismo que mais propicia a distribuição e sustentação de *worms*.

Diante dessa interconexão de empresas e sistemas, onde bilhões de dólares circulam por segundo, o aspecto econômico entra em pauta. Por exemplo, a rede SWIFT, que conecta 8.100 financeiras internacionais em 207 países (SWIFT, 2007), movimenta trilhões de dólares por dia. Uma falha nesse sistema poderia causar uma catástrofe instantânea, desestabilizando economias inteiras, alertam Hoglund e McGraw (2006).

Logicamente trata-se de um risco moderno que tem que de ser enfrentado. A conectividade e compartilhamento de informações não devem ser banidos, estagnados ou reduzidos, já que esses trazem diversos benefícios às instituições, acelerando o crescimento, a produção e a rentabilidade.

---

[2] Redes seqüestradas e controladas à distância por crackers, também chamadas de Botnets.

1.4 Análise de *Software*

Existem diversas maneiras de avaliar *softwares* via Engenharia Reversa. Os métodos variam de acordo com a disponibilidade de acesso ao código-fonte e ao código binário do aplicativo, define McGraw (2006). Apesar de distintos pelas abordagens diferentes, todos os métodos procuram entender o *software* examinando algumas áreas fundamentais para localização de vulnerabilidades:

- Funções que verificam os limites suportados das variáveis (*range*) incorretamente ou simplesmente não os verificam;
- Funções que passam por dados fornecidos pelo usuário ou os utilizam em uma *string* de formato (*format string*);
- Rotinas que obtém entradas de usuários utilizando *loop*;
- Operações de baixo nível de cópia de *bytes*;
- Rotinas que utilizam aritmética de ponteiro em *buffers* fornecidos pelo usuário;
- Chamadas de sistema confiáveis que aceitam entradas dinâmicas.

Dentre os pontos comuns de verificação de vulnerabilidade, estes são os mais freqüentes, devido a maior possibilidade de exploração e também por serem pontos que não recebem a atenção devida por parte do programador ao implementá-las, já que muitos deles pressupõem (equivocadamente) que o sistema operacional irá gerenciá-las.

Os métodos empregados para estes exames são as análises de Caixa-Branca (*White Box*), Caixa-Preta (*Black Box*) e Caixa-Cinza (*Gray Box*).

1.4.1 Análise de Caixa-Branca

A análise de Caixa-Branca se refere ao estudo e compreensão do código-fonte, sendo muito eficiente para localização de erros de programação e de implementação no *software*. Esta análise pode ser auxiliada e automatizada por um analisador estático, efetuando testes de comparação de padrões, no entanto, uma das desvantagens desse tipo de teste é o alto índice de obtenção de falso-positivos[3]. (McGRAW, 2006)

---

[3] Vulnerabilidades encontradas equivocadamente, não existentes.

Muitas vezes problemas descobertos em análises Caixa-Branca podem não ser exploráveis em um sistema real e distribuído. Outras ferramentas de segurança podem, por meio de filtros e detecções de intrusos como *Firewalls* e *IDS*, bloquear estes ataques. Entretanto, análises Caixa-Branca são úteis para testar como um sistema irá se comportar em distintos ambientes e condições.

Das ferramentas existentes, elas se distinguem em 2 tipos: analisadoras de código-fonte e aquelas que automaticamente convertem o arquivo binário em código-fonte para posterior análise. Como no segundo tipo, em casos de indisponibilidade de código-fonte, pode ser feita a reversão do *software* e estudado o código-fonte obtido, ainda assim será caracterizado como análise de Caixa-Branca.

1.4.2 Análise de Caixa-Preta

A análise de Caixa-Preta examina um programa em execução sondando-o com várias entradas, sem a necessidade de acesso e, conseqüentemente, estudo do código-fonte ou código binário do aplicativo. McGraw (2006) destaca que este tipo de análise torna-se interessante justamente por este fator, já que abre a possibilidade de exame e exploração remota do aplicativo. E, como nem sempre é possível ter acesso ao código-fonte ou binário do aplicativo, este tipo de análise faz com que invasores reais freqüentemente adotem-na.

Neste método, entradas maliciosas são fornecidas a um sistema em funcionamento no intuito de quebrá-lo, sendo esta uma maneira de validar e avaliar problemas de negação de serviço (*DoS*),. Caso o programa quebre, um problema de segurança pode ter sido descoberto. Esta análise ainda serve para determinar se uma possível área vulnerável é realmente explorável, pois como citado anteriormente, nem sempre é possível explorar todas as vulnerabilidades devido as proteções externas fornecidas por outros dispositivos, como *Firewalls* e *IDS*.

Apesar de muito mais fácil e de requerer menos habilidade que a análise de Caixa-Branca, a análise de Caixa-Preta, normalmente, não é tão eficiente quanto ao reconhecimento de comportamento e funcionamento, afirma McGraw (2006).

1.4.3 Análise de Caixa Cinza

A análise de Caixa-Cinza combina técnicas de Caixa-Branca e teste de entradas de Caixa-Preta, requerendo normalmente a utilização de diversas ferramentas em conjunto. Um exemplo seria a execução e fornecimento de entradas de um programa-alvo diretamente dentro de um depurador, analisando e detectando quaisquer possíveis falhas ou comportamentos defeituosos. (McGRAW, 2006)

1.4.4 Qualidade na análise

Hoglund e McGraw (2006) consideram que um dos maiores problemas de segurança de *software* é a negligência das *software houses* na etapa de testes. Devido às restrições de tempo e de orçamento, a maioria das fabricantes se restringe ao teste funcional, empregando pouco tempo para procurar e entender os riscos à segurança.

Indiferente do método de análise utilizado, maior ênfase deve ser - e ultimamente até tem sido – dada ao gerenciamento da qualidade de *software*, identificando e gerenciando seus riscos desde o ponto mais inicial de seu ciclo de vida e desenvolvimento. (HOGLUND e McGRAW, 2006)

1.5 Segurança de *Software*

Um aspecto central e crítico dos problemas referentes à segurança de *software* é justamente a existência de *bugs*, sejam eles aplicados à segurança ou meros *bugs* de rotinas de *software*. Defeitos de *software*, desde *bugs* de implementação (como estouros de *buffer*) a falhas de projetos (como manipulação inconsistentes de erros), são comuns em qualquer tipo de *software* tornando-os, na maioria das vezes, sistemas exploráveis. (McGRAW, 2006) A alta conectividade proporcionada pela Internet também aumenta este risco.

Estima-se que o número de vulnerabilidades tende a aumentar cada vez mais. Pesquisadores e acadêmicos de segurança afirmam que mais da metade das vulnerabilidades atuais provêm de estouros de *buffer*. Porém, outros problemas mais sutis podem ser igualmente perigosos, mesmo sendo considerados apenas meros *bugs*. (McGRAW, 2006) Sobre estes dados, nota-se claramente a necessidade de mudança no modo de tratamento do quesito segurança e na maneira de desenvolver disciplinadamente *softwares*.

Segurança de *software* é compreender como funciona o *software*, entender seus riscos e saber como gerenciá-los. Boas práticas de segurança de *software* alavancam boas práticas de engenharia de sistemas, levando esta preocupação para os níveis iniciais do ciclo de vida de *software*, conhecendo e entendendo problemas comuns e considerando, já nesta etapa, todas as possíveis situações que possam implicar risco ao projeto.

1.5.1 Terminologia

A terminologia para classificação dos riscos de *software* basicamente se resume aos termos abaixo. McGraw (2006) considera que esta categorização pode auxiliar na compreensão:

- Defeito: Tanto os problemas de implementação quanto os problemas de projeto são considerados defeitos. É um desvio de comportamento do sistema que pode provocar falhas.
- Falha: Operação incorreta de um sistema ou de um de seus componentes. Uma falha não implica necessariamente na produção de um erro, porém a ocorrência de uma falha pode acarretar um erro.
- *Bug* (Erro): É um problema de implementação ao nível de *software*. É um resultado incorreto, fora de suas especificações.
- Risco: É a probabilidade de que uma falha ou *bug* cause impacto à um *software*.

1.6 Engenharia Reversa versus *Cracking*

Muitas vezes é confundido o conceito de Engenharia Reversa com o conceito de *Cracking*. Tipicamente, a aplicação da Engenharia Reversa visa à melhoria, correção ou desenvolvimento de aplicativos de maior qualidade e robustez. Como em qualquer técnica, existe a aplicação ilegal ou fora-da-lei desta técnica, denominada *Cracking*. (SCHNEIER, 2000)

Segundo Schneier (2000), o *cracking* surgiu junto com a programação de *softwares* em si e basicamente se resume no método de rastrear, identificar e burlar métodos de proteção e validação de *softwares* comerciais, ativando-os e permitindo a execução integral de seus recursos.

Programas *Shareware* (distribuídos livremente, porém com restrições de recursos) requerem algum tipo de validação, seja ela uma chave de registro ou uma senha codificada. Nestes casos, um *cracker* pode ser apropriar dos métodos de Engenharia Reversa com a finalidade de burlar essas proteções. O *cracker* pode estudar o *software* até adquirir conhecimento do *software* a tal nível que se possível sobrepor, redirecionar ou nulificar as restrições e validações do *software*, provendo, de maneira ilegal, acesso irrestrito a todo *software* mesmo sem este ser registrado ou validado legitimamente.

2 SISTEMAS E FORMATOS

2.1 Linguagens de Programação

As linguagens de programação são dividas em níveis de linguagens – baixo, alto e muito alto nível –, além de serem divididas em 5 gerações. Os termos “baixo”, “alto” e “muito alto” nível não se referem à inferioridade ou superioridade das linguagens, mas sim ao nível de abstração da linguagem à linguagem de máquina. (FOTOPOULOS, 2001)

Conforme define EILAM (2005), linguagens de baixo nível são aquelas que fornecem pequena ou nenhuma abstração às instruções do processador, ou seja, linguagens próximas ao *hardware*.

Linguagens de alto nível são mais abstratas em comparação às linguagens de baixo nível, sendo assim, de mais fácil uso e entendimento. Nestas, ao invés de tratarmos diretamente com registradores, endereços de memórias e pilhas de chamadas como é feito em linguagens baixo nível, é trabalhado com expressões, variáveis, vetores e fórmulas aritméticas. Isto facilita e acelera em muito o processo de implementação. (EILAM, 2005)

A programação de muito alto nível possui, assim como está definido em sua classificação, um alto nível de abstração, sendo basicamente utilizada como uma ferramenta de produtividade. Normalmente estas linguagens são limitadas a um propósito específico e, na maioria das vezes, são encapsuladas, internas e próprias de *softwares*. (WIKIPEDIA, 2007g)

Quanto às gerações das linguagens, existem:

- 1ª geração ou 1GL: é a linguagem de máquina, baixo nível, constituída apenas de “1” (um) e “0” (zero), dispensando o uso de compiladores; (WIKIPEDIA, 2007c)

- 2ª geração ou 2GL: é a linguagem *Assembly*. Linguagem de baixo nível "convertida" por um simples mapeamento do código *Assembly* em linguagem de máquina (*opcode*); (WIKIPEDIA, 2007e)
- 3ª geração ou 3GL: linguagens de alto nível, estruturadas e projetadas para um mais fácil entendimento e utilização. Suporta programação orientada a objetos (POO) tornando-a extremamente mais flexível e poderosa; (WIKIPEDIA, 2007f)
- 4ª geração ou 4GL: estas linguagens possuem sintaxe similar à fala humana, sendo geralmente utilizadas em banco de dados via *scripts* de consulta e programação. Objetivam a redução de custo e de tempo de desenvolvimento; (WIKIPEDIA, 2007d)
- 5ª geração ou 5GL: ao invés de programadas por algoritmos, são linguagens baseadas na solução de problemas através de inteligência artificial. São lançados problemas ao *software* que os analisa e processa via IA. (WIKIPEDIA, 2007b)

Conforme Fotopoulos (2001), na medida em que se sobe o nível (geração) da linguagem de programação, maior a abstração e, conseqüentemente, menor o número de *LOC* para execução de um programa. Esta tendência faz com que, com o avanço das linguagens de programação, cada nova linguagem tenha funções mais genéricas e operacionais, facilitando a programação. A tabela 2.1 exemplifica a afirmação, considerando, no caso analisado, um programa com 320 *LOC* de *Assembly*.

Tabela 2.1 – Instruções por função

Linguagem	Linhas de código (*LOC*)
Assembly	320
C	128
Pascal	80
Lisp	65
C++	50
Delphi	30
Perl	25

Fonte: http://www.site.uottawa.ca/~shervin/courses/seg4100/project/FunctionPointEstimating.pdf

Conforme a tabela 2.1, pode-se perceber uma notável diferença na quantidade de instruções necessárias para a criação de um programa de acordo com o nível da linguagem. Essa diminuição de *LOC* facilita e acelera a programação. Comparando os números de *LOC*,

pode-se verificar que um programa escrito em *Assembly* tem 10 vezes mais *LOC* do que quando escrito no *Delphi*, por exemplo.

Esta tabela pode ainda servir como parâmetro na comparação da dificuldade de aprendizagem de novas linguagens de programação, lembra Fotopoulos (2001). Quanto maior o *LOC*, maior a dificuldade de aprendizado da mesma.

2.2 *Assembly*

Apesar da linguagem *Assembly* ser uma linguagem primitiva – segunda geração –, Eilam (2005) afirma que ela não pode ser considerada obsoleta. O *Assembly* existirá enquanto processadores existirem, pois, por mais que subamos o nível de linguagem, em algum ponto o código desta será convertido ou transformado em código de máquina. Assim, indiferente da linguagem que for utilizada para criação do *software*, será sempre possível converter o *software* em código-fonte *Assembly* – exceto nos casos de código pré-compilados como o *byte-code* do *Java* e o *Common Intermediate Language* do *Microsoft .NET*, que podem ser revertidos para seus códigos-fonte próprios.

“A linguagem *Assembly* é a linguagem da Engenharia Reversa” (EILAM, 2005, p. 44) (Tradução nossa). É essencial o seu domínio para compreensão do funcionamento interno dos sistemas e das técnicas de Engenharia Reversa. Esta aprendizagem geralmente é muito difícil, variando com as experiências de cada programador, costumando exigir muito tempo e prática.

Além disso, o *Assembly* é dependente de *hardware*. Cada modelo de processador possui instruções próprias, obrigando o programador a estudar e familiarizar-se com os conceitos e funcionamento separadamente.

Como este trabalho tem o intuito de meramente introduzir a linguagem, será apresentando apenas os princípios e instruções básicas, num breve objetivo de entendimento e extração de informação de curtos códigos-fonte *Assembly*, assim como será analisada apenas a arquitetura IA-32 (*Intel’s 32-bit architecture*), por ser este o modelo mais utilizado pelas máquinas atuais (x86). No entanto, esse conteúdo deve ser profundamente explorado, pois é o cerne das atividades de Engenharia Reversa.

2.2.1 Gerenciamento de dados em baixo nível

Para Eilam (2005), umas das maiores dificuldades no estudo da linguagem *Assembly* é justamente a diferente perspectiva da programação de baixo nível em comparação à programação de alto nível. A organização, os métodos, os passos para chamada de instruções, as variáveis, enfim, todo gerenciamento de dados é tratado distintamente.

Na programação de alto nível, os detalhes internos referentes à maneira com que o programa se comunica internamente com o *hardware* são ocultos. Considerando a função escrita em código C a seguir, pode-se fazer uma analogia entre o baixo e alto nível das linguagens.

```
int Multiply(int x, int y)
{
int z;
z = x * y;
return z;
}
```

Em uma linguagem de baixo nível, a representação desta função dar-se-ia basicamente pelos seguintes passos:

1) Armazenar o estado atual da máquina antes de iniciar a execução da função;

2) Alocar memória para a variável Z;

3) Carregar os parâmetros X e Y da memória para os registradores internos, que é a memória interna do processador;

4) Multiplicar X por Y, gravando o resultado num registrador;

5) Copiar o resultado – alocado num registrador – para o endereço de memória da variável Z;

6) Restaurar o estado anterior da máquina;

7) Retornar ao *caller* (instrução que chamou esta função) enviando o valor da variável Z como resultado.

Como pode perceber-se, muito da complexidade agregada às linguagens de baixo nível é justamente o modo de tratamento e gerenciamento de dados. Para tanto, serão

demonstradas, sinteticamente, as instruções comumente utilizadas, assim como os tipos e funções dos registradores do processador, realizando um comparativo entre as instruções de uma linguagem de baixo nível com as instruções de uma linguagem de alto nível..

2.2.2 Registradores

Na intenção de evitar o acesso à *RAM* a cada instrução – fato que atrasaria substancialmente o processamento das máquinas –, os microprocessadores possuem uma memória interna de alta performance, explica Eilam (2005). Esses pequenos blocos de memória interna do processador são chamados de registradores. São de fácil e simples acesso e, devido à sua localização – dentro do próprio processador –, são extremamente rápidos quanto à leitura e gravação de dados.

Os processadores *IA-32* possuem 8 registradores genéricos de 32 *bits*: `EAX`, `EBX`, `ECX`, `EDX`, `ESI`, `EDI`, `EBP` e `ESP`. Ainda existem outros tipos de registradores internos, como registradores de pilhas de ponto flutuante e outra variedade de registradores de controle, mas, a maioria deles é utilizada para recursos internos ou não são utilizáveis pelo *software*.

Alguns dos registradores podem ser lidos ou subdivididos de acordo com o número de *bits*, destaca Eilam (2005). Todos registradores 32 *bits* iniciam com a letra E - que vem da palavra *extended* em inglês (estendido) – e alguns deles, como os registradores `EAX`, `EBX`, `ECX` e `EDX`, podem ser lidos como registradores 16 *bits*: `AX`, `BX`, `CX e DX`, respectivamente. Nestes casos são considerados apenas os 16 *bits* mais baixos do registrador. Subseqüentemente, cada registrador 16 *bits* pode ser subdividido em 2 registradores de 8 *bits*. A figura 2.1 ilustra esta disposição.

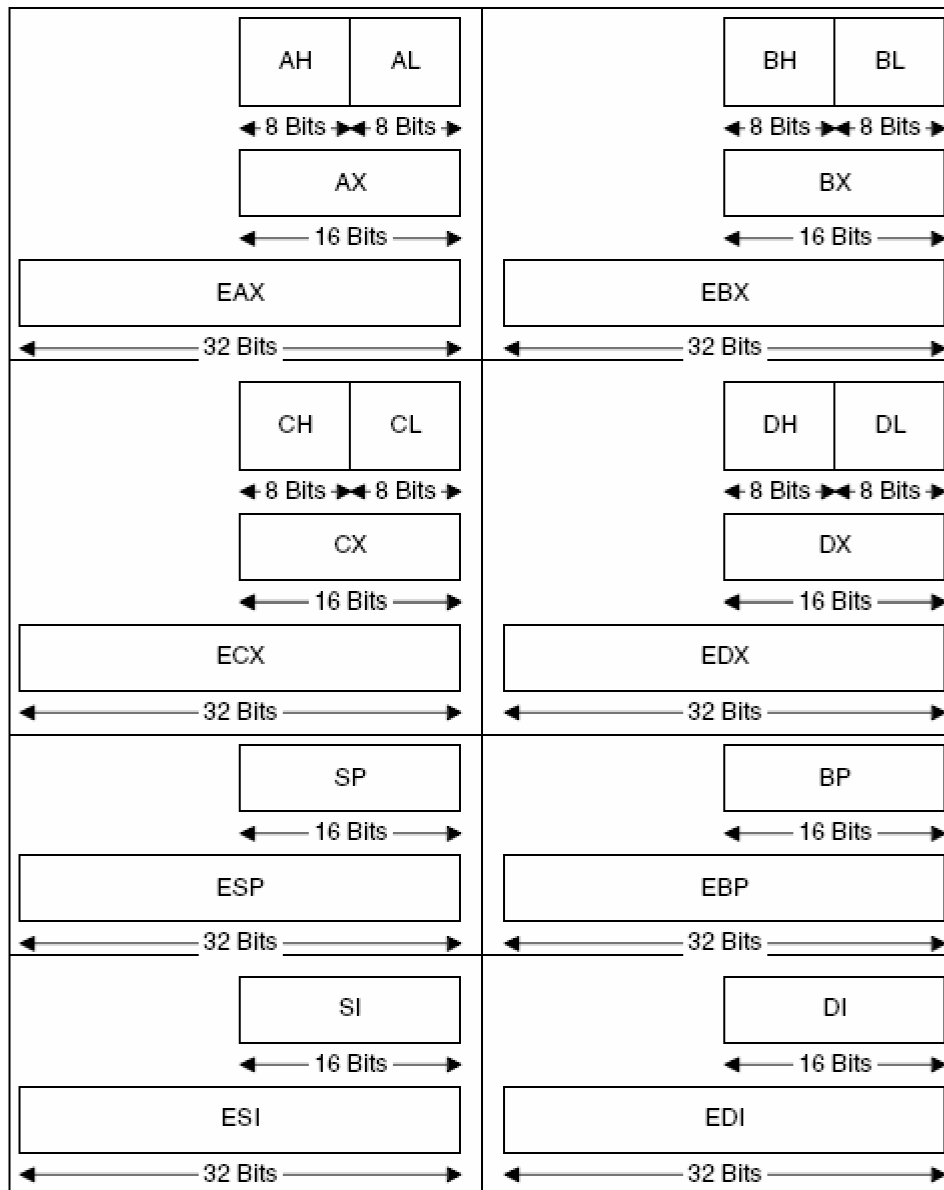

Figura 2.1 – Definição e disposição dos registradores

Imagem do autor, adaptado de EILAM, 2005, p. 46

Como visto anteriormente, os registradores são utilizados para armazenamento de informações quanto ao funcionamento, dados e variáveis da aplicação em execução. O quadro 2.1 detalha mais especificadamente cada registrador.

Quadro 2.1 – Descrição dos registradores e suas relativas funções

Registrador	Função
EAX, EBX, EDX	Registradores genéricos usados para armazenamento de variáveis como inteiros, lógicas (*booleanas*) ou endereços de memória.
ECX	Registrador genérico normalmente utilizado como um contador de repetição (em instruções que requerem contagem).
ESI/EDI	Registradores genéricos normalmente utilizados como ponteiros de origem e destino (respectivamente) para instruções de cópia de memória.
EBP	Pode ser utilizado como um registrador genérico porém normalmente é utilizado como ponteiro base da pilha de instruções. Se usado em conjunto com o ponteiro da pilha, serve para definir o frame da função.
ESP	É o ponteiro da pilha de instruções, local onde é armazenada a posição atual da pilha. Cada inserção e exclusão de dado na pilha atualiza automaticamente o registrador para o novo ponteiro.

2.2.3 Identificadores (*Flags*)

Eilam (2005) ressalta que, além dos registradores demonstrados, os processadores *IA-32* possuem um registrador interno especial chamado de *EFLAGS*, que contém uma diversidade de estados e identificadores (*flags*) lógicos do sistema, utilizados principalmente para detecção de condições relativas às instruções recém executadas.

Em certas operações, as instruções podem testar os operandos e marcar as *EFLAGS* de acordo com seu resultado, tornando possível que a instrução subseqüente leia esses indicadores e defina suas operações de acordo com elas. (INTEL, 1999)

Um caso típico acontece no salto condicional, que processa uma instrução de comparação, testando os operandos e marcando o resultado no *EFLAGS*. Então o *bit flag ZF* (*Zero Flag* – Indicador de zero) da *EFLAGS* é lido para definir se há a necessidade ou não do salto.

2.2.4 Pilha (*Stack*)

Considerando novamente o exemplo anterior de multiplicação, mais especificamente no passo 2 (aonde é alocado memória para a variável Z), é importante destacar o conceito de pilhas. Sempre que houver um carregamento de um valor para memória, este poderá ficar

armazenado em um registrador ou ir para a pilha, dependendo da disponibilidade dos registradores ou por diversas outras razões que variam conforme as regras do compilador.

A pilha é uma área da memória do programa utilizada para armazenamento de informações do programa e da máquina e, principalmente, responsável pela passagem de parâmetros às funções, define Eilam (2005). Após os registradores, a pilha pode ser vista como uma segunda área de armazenamento de informações. Fisicamente é apenas uma área da *RAM* alocada como qualquer outro tipo de informação, porém com este propósito definido. Internamente elas funcionam como pilhas *LIFO top down*, onde o último item a entrar é o primeiro a sair, alocando os dados, continuamente, dos maiores para os menores endereços de memória. As figuras 2.2 e 2.3 demonstram como a pilha controla a entrada e saída dos valores.

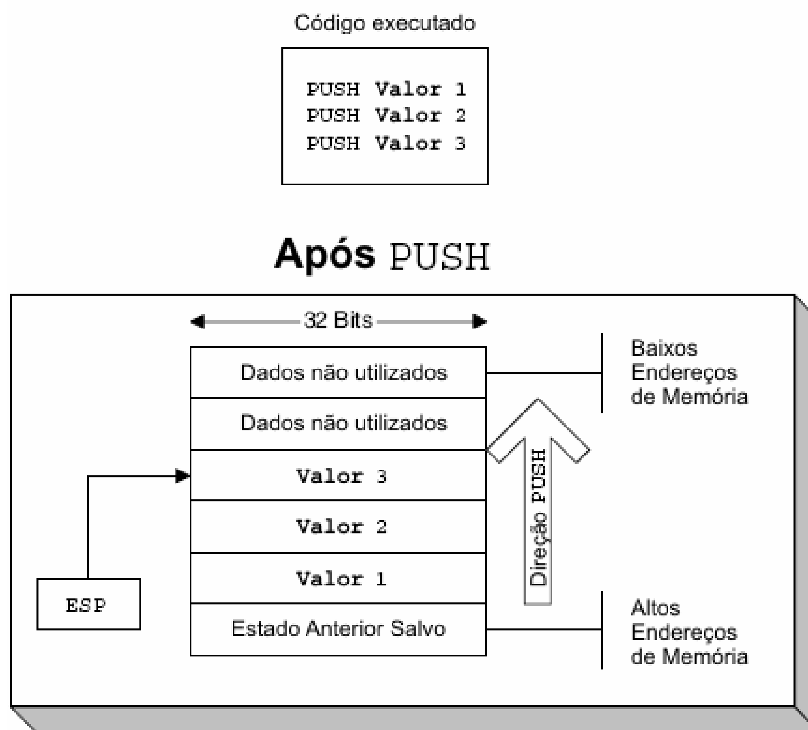

Figura 2.2 – Análise da pilha após inserção de 3 valores

Imagem do autor, adaptado de EILAM, 2005, p. 41

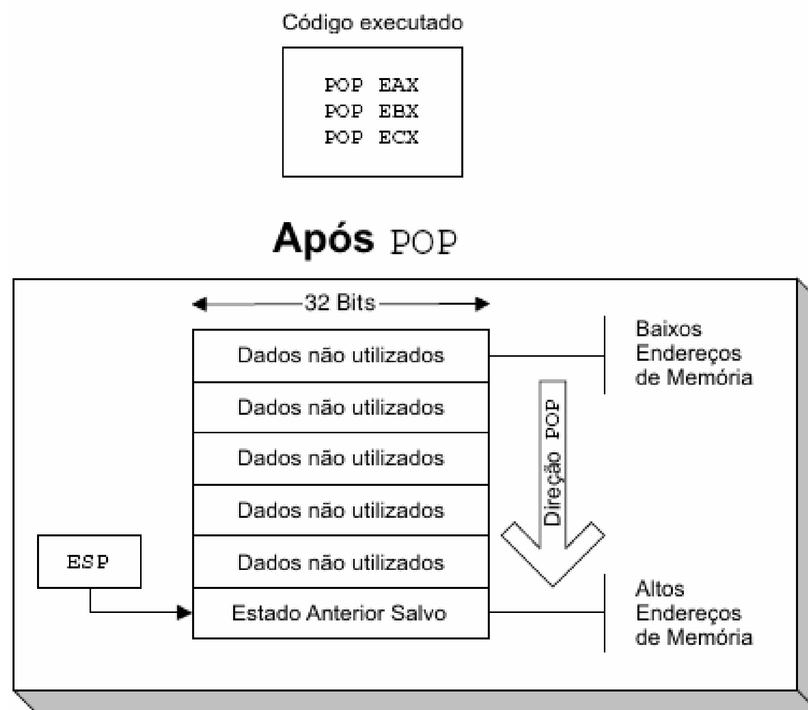

Figura 2.3 – Análise da pilha após exclusão de 3 valores

Imagem do autor, adaptado de EILAM, 2005, p. 41

2.2.5 Interrupções

Interrupções são cruciais a qualquer sistema computacional, pois é maneira utilizada pelo *hardware* para efetuar as comunicações de entrada e saída com a CPU. As interrupções funcionam basicamente paralisando a execução do processador e chamando um bloco de código, denominado de rotina de interrupção. Após esta rotina, o processador restaura os estados salvos e resume a execução da aplicação. (FOTOPOULOS, 2001)

2.2.6 Instruções

Instruções são códigos de operação (*opcode*) utilizados para efetuar a comunicação com o processador. Cada instrução *Assembly* possui seu correspondente em *opcode*, como, por exemplo, a instrução `POP EAX` que corresponde aos *bytes* `66 58` (em hexadecimal). (INTEL, 1999) A lista completa de instruções *IA-32* e seus respectivos *opcodes* pode ser encontrada no site da *Intel*, pelo endereço <http://www.intel.com/design/intarch/manuals/ 243191.htm>.

As instruções podem ser sucedidas de um, dois ou nenhum parâmetro, que são denominados operandos. O número de operandos varia conforme a instrução e servem para

fornecer os dados necessários para execução de cada instrução. Os operandos podem ser utilizados de 3 formas básicas, conforme mostra o quadro 2.2.

Quadro 2.2 – Exemplos de referências de operandos de instruções e seus significados

Operando	Tipo	Descrição
EAX	Registrador	Referência ao registrador EAX, utilizado tanto para escrita quanto para leitura.
0x30004040	Constante	Refere-se a um valor, uma constante codificada implicitamente no código do programa
[0x4000349e] ou [EAX]	Endereço de Memória	Cercado por colchetes, o valor pode referenciar uma posição de memória direta ou um registrador que armazene um endereço de memória.

Quanto à sintaxe, as instruções são normalmente dispostas conforme abaixo:

```
Instrução Operando1, Operando2
```

Dentre a vasta lista de instruções suportadas pelos processadores *IA-32*, serão apresentadas as instruções mais freqüentes, que reproduzem as ações mais comuns dentro de uma aplicação.

2.2.6.1 Movimentação de dados

A instrução MOV é provavelmente a instrução *IA-32* mais utilizada e serve para mover os dados de um registrador ou endereço de memória a outro, explica Eilam (2001). A instrução é composta de 2 operandos, sendo eles respectivamente o operando de destino e o operando de origem. A sintaxe da instrução é a seguinte:

```
MOV Operando_de_destino, Operando_de_origem
```

O operando de destino pode ser um registrador ou um endereço de memória, enquanto o operando de origem pode ser de qualquer tipo. Estes, porém, devem seguir a regra geral das instruções *IA-32* que exige que apenas um dos operandos seja um endereço de memória.

2.2.6.2 Manipulando a pilha

Conforme citado, quando há no *software* a necessidade de passagem de parâmetros para outras funções ou bloco de código, ou que seja necessário o armazenamento de

informações que não são alocáveis nos registradores, o *software* trabalha com uma pilha *LIFO top down* para esse armazenamento. Para isto existem 2 funções: uma para armazenar e outra para remover os valores.

A instrução `PUSH` insere um valor no topo da pilha, enquanto a instrução `POP` move o valor do topo da pilha para um registrador ou endereço de memória, retirando-o da pilha. Ambas instruções atualizam automaticamente o ponteiro da pilha, gerenciado pelo registrador `ESP`. (EILAM, 2001; INTEL, 1999)

A sintaxe de ambas instruções é similar, conforme demonstrado abaixo:

```
PUSH Operando
POP  Operando
```

2.2.6.3 Operações aritméticas

Eilam (2001) relaciona a existência de 6 instruções *IA-32* responsáveis pelas operações aritméticas básicas: soma, subtração, multiplicação e divisão. Cada instrução possui sintaxe própria, variando no tipo e número de operandos, conforme exemplifica o quadro 2.3.

Quadro 2.3 – Instruções para as operações aritméticas básicas

Instrução	Descrição
`ADD Operando1, Operando2`	Instrução que soma dois inteiros (com ou sem sinal), armazenando o resultado no operando 1. O operando 1 pode ser do tipo registrador ou endereço de memória, enquanto o operando 2 pode ser de qualquer tipo (lembrando sempre a condição que diz que não é possível que ambos operadores sejam endereços de memória).
`SUB Operando1, Operando2`	Instrução que subtrai o valor do operando 2 do operando 1, armazenado o resultado no operando 1. Os valores podem ser com ou sem sinal. O operando 2 pode ser do tipo registrador ou endereço de memória, enquanto o operando 1 pode ser de qualquer tipo, seguindo a mesma regra de impossibilidade dois endereços de memória.
`MUL Operando`	Instrução que multiplica o operando (sem sinal) pelo valor do registrador `EAX`, armazenando o seu resultado (64 *bits*) em `EDX:EAX`. `EDX:EAX` é um arranjamento comum onde os 32 *bits* menos significativos (baixos) são salvos em `EAX` e os 32 *bits* mais significativos (altos) são salvos em `EDX`.
`DIV Operando`	Instrução que divide o valor 64 *bits* (sem sinal) armazenado em `EDX:EAX` pelo operando, armazenando o quociente da divisão em `EAX` e o resto em `EDX`.

Instrução	Descrição
`IMUL Operando`	Instrução que multiplica o operando (com sinal) pelo valor do registrador `EAX`, armazenando o seu resultado (64 bits) em `EDX:EAX`.
`IDIV Operando`	Instrução que divide o valor 64 *bits* (com sinal) armazenado em `EDX:EAX` pelo operando, armazenando o quociente da divisão em `EAX` e o resto em `EDX`.

2.2.6.4 Comparação de dados

A comparação de operando se dá pela utilização da instrução `CMP`, que compara os operandos salvando o resultado na *EFLAGS*. Obviamente, a instrução requer 2 operandos que podem ser de quaisquer tipo, porém obedecendo a regra que inibe a utilização de 2 endereços de memória. (EILAM, 2001; INTEL, 1999)

```
CMP Operando1, Operando2
```

Internamente, esta instrução de comparação simplesmente executa uma subtração do segundo operando no primeiro, descartando seu resultado e marcando as *flags* da *EFLAGS* conforme suas respectivas finalidades.

2.2.6.5 Execução condicional

As instruções de execução condicional formam um grupo de instruções que são utilizadas para saltos e desvios na execução do *software*. Cada instrução deste grupo possui sua peculiaridade, verifica sua(s) respectiva(s) *flag*(s) no *EFLAGS*, e no caso de coincidência, transfere a execução do *software* para o endereço de memória informado pelo operando. Se a condição não é atendida, a instrução é ignorada e é dada a seqüência normal ao *software*. (EILAM, 2001; INTEL, 1999)

```
Jcc Novo_Endereço_de_Código
```

2.2.6.6 Chamada de Funções

A chamada de funções ocorre pela instrução `CALL`, que desvia a execução do *software* para o endereço de memória informado pelo operando. Assim, é desviada a execução do programa e processadas todas instruções até o encontro de uma instrução `RET`, que identifica o final da função. A função `RET` termina a chamada, retornando a execução ao endereço subseqüente ao de origem da chamada. (EILAM, 2001; INTEL, 1999)

```
CALL Endereço_da_Função
```

2.2.6.7 Outras funções

Conforme citado, o rol de instruções do processador *IA-32* é bastante vasto e seria inconveniente listar todas. Foram apenas apresentadas as instruções mais comuns com o objetivo de familiarizar as instruções, sintaxes e tipos, além de demonstrar internamente o funcionamento do *software* junto ao processador.

2.3 Formato *Win32/Portable Executable*

Conforme define Luevelsmeyer (1999), o formato de arquivo *Win32/Portable Executable* ou simplesmente *PE*, é a estrutura de arquivo utilizada nos arquivos executáveis no ambiente *Windows 32 bits*, e é basicamente uma estrutura padrão de armazenamento de dados onde estão encapsuladas todas as informações necessárias – como os blocos de códigos, definições internas de estrutura, variáveis, dados de inicializados e não inicializados, e todas outras informações necessárias – ao sistema operacional (*system loader*) para sua leitura e execução.

Costumeiramente pouco estudado por desenvolvedores, devido até pela não obrigatoriedade de conhecimento específico e aprofundado do mesmo no desenvolvimento de aplicativos, este assunto dificilmente é tópico de discussão ou debate na comunidade, porém este estudo pode agregar em muito àqueles que buscam conhecer o mecanismo interno de funcionamento de um *software*.

No caso de proteção de *software*, é de extrema importância seu conhecimento, pois é nele que se encontram as informações básicas e fundamentais para seu funcionamento, facilitando assim a criação de métodos, armadilhas e bloqueios para prevenção de descarregadores de memória (*memory dumpers*), depuradores (*debuggers*) e outros tipos de utensílios e ferramentas de *cracking*, afirma Cerven (2002).

2.3.1 História

O formato *Portable Executable* foi desenvolvido pela *Microsoft* e padronizado em 1993 pelo Comitê de Padrões de Interfaces de Ferramentas (*Tool Interface Standard*

Committee), formado pela *Microsoft*, *Intel*, *Borland*, *Watcom*, *IBM*, entre outras. (LUEVELSMEYER, 1999)

O modelo *PE* foi desenvolvido com base no modelo *COFF*, isto porque a maioria de seus criadores eram os mesmos que desenvolveram e codificaram o *COFF*. Como muitos recursos foram reaproveitados, até porque funcionava muito bem e já havia sido testado exaustivamente, seria necessário apenas fazer as adaptações necessárias e exigidas pelas novas plataformas. (MICROSOFT, 2006)

O nome *Portable* é devido à sua portabilidade, que possibilita sua implementação em diversas plataformas (x86, *MIPS®*, *Alpha*, entre outros) sem que sejam necessárias alterações em seu formato, lembra Luevelsmeyer (1999). É lógico que diversas alterações (como codificação binária de instruções de *CPU*, etc.) são necessárias para o funcionamento nestas plataformas, porém o detalhe interessante, é que não foi necessário reescrever os carregadores dos sistemas operacionais (*system loader*) e outras ferramentas para desenvolvimento.

2.3.2 Formato *Portable Executable*

O formato *PE* possui as seguintes estruturas de dados: cabeçalho *MZ* do *DOS*, fragmento (*stub*) *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, ultimamente, as seções propriamente ditas, conforme figura 2.4:

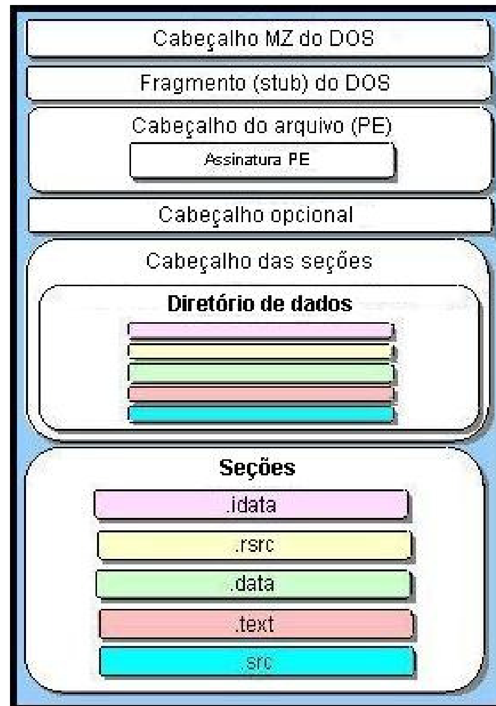

Figura 2.4 – Estrutura do formato PE

Fonte: Imagem do autor, adaptado de PIETREK, 1994

Os arquivos *PE*, quando carregados na memória, são bastante similares aos arquivos no disco. Com isto, o carregador (*system loader*) pode executar essa operação mais rapidamente, apenas tendo que mapear os endereços do arquivo para endereços de memória (*RVA*).

Um *RVA* é a sigla da palavra inglesa *Relative Virtual Address*, que se refere a um *offset* de memória relativo à memória base onde ele se encontra. Por exemplo, considerando que um aplicativo seja mapeado na memória a partir do endereço `0x10000` e que o *RVA* de um objeto nesta memória seja `0x00464`, seu endereço de memória real será `0x10464`. (PIETREK, 1994)

2.3.2.1 Cabeçalho *DOS*

Todos arquivos no formato *PE* começam com o cabeçalho *DOS* (*IMAGE_DOS_ HEADER*), oriundo dos antigos formatos *COFF* e mantido basicamente por compatibilidade. A maioria das informações deste cabeçalho não é considerada pelos aplicativos 32 *bits*, já que este serve basicamente como repositório de informações dos aplicativos 16 *bits*. Para os aplicativos 32 *bits*, existe um cabeçalho específico, denominado de cabeçalho *PE*, discutido a seguir.

Apenas dois dos campos do cabeçalho *DOS* tem valor significativo aos aplicativos 32 bits: a assinatura *MZ* e o *offset* do cabeçalho *PE*.

Os primeiros 2 *bytes* do cabeçalho DOS – e conseqüentemente de um arquivo *PE* – constituem a assinatura *MZ*. Esta assinatura é obrigatória para identificação do formato *PE* e, conforme o próprio nome sugere, é composta pelos *bytes* "*MZ*" (`4D 5A` em hexadecimal), derivadas das iniciais do nome de um de seus arquitetos: Mark Zbikowski. (PIETREK, 2002)

Os 4 *bytes* localizados a partir do *offset* `0x3C` são igualmente importantes, pois eles indicam o *offset* do cabeçalho *PE*, cabeçalho que fornece os dados necessários para o carregamento do *software* 32 *bits*.

O cabeçalho *DOS* é composto de 64 *bytes*, dispostos da seguinte maneira: (CERVEN, 2002; PIETREK, 1994; LUEVELSMEYER, 1999)

1) Assinatura "*MZ*" (2 *bytes*);
2) Tamanho em *bytes* da última página do arquivo (2 *bytes*);
3) Número total de páginas no arquivo (2 *bytes*);
4) Itens de relocação (2 *bytes*);
5) Tamanho do cabeçalho *DOS* (2 *bytes*): Estes *bytes* indicam a quantidade de seqüências de 16 *bytes* contidas no cabeçalho, ou seja, para determinar o tamanho do cabeçalho basta multiplicar o valor encontrado neste campo por 16;
6) Tamanho mínimo da memória (2 *bytes*): sempre encontrado "`00 00`" em hexadecimal;
7) Tamanho máximo da memória (2 *bytes*): sempre encontrado "`FF FF`" em hexadecimal;
8) Valor inicial do registrador `SS` (*Stack Segment*) (2 *bytes*);
9) Valor inicial do registrador `SP` (*Stack Pointer*) (2 *bytes*);
10) *Checksum* do cabeçalho *DOS* (2 *bytes*);

11) Valor inicial do registrador `IP` (*Instruction Pointer*) (2 *bytes*);

12) Valor inicial do registrador `CS` (*Code Segment*) (2 *bytes*);

13) *Offset* do fragmento (*stub*) *DOS* (2 *bytes*);

14) *Overlay* (2 *bytes*);

15) Identificador *OEM* (2 *bytes*);

16) Informações *OEM* (2 *bytes*);

17) *Bytes* reservados (24 *bytes*);

18) *Betov's CheckSum* (4 *bytes*);

19) *Offset* do cabeçalho de arquivo *PE* (4 *bytes*).

Para melhor exemplificar, será usado, como base de dados para as figuras, o arquivo *notepad.exe* do *Windows XP SP 2*.

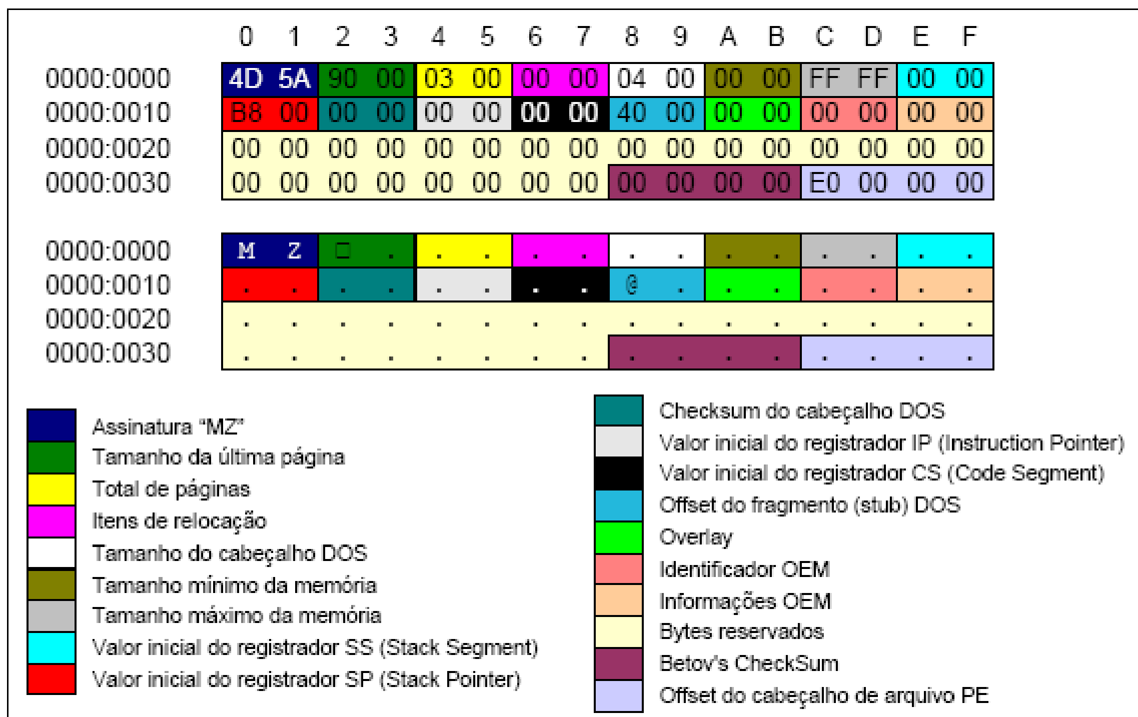

Figura 2.5 – Cabeçalho DOS

2.3.2.2 Fragmento (*Stub*) do *DOS*

Conforme se pôde perceber no cabeçalho *DOS*, é encontrado no *offset* `0x18` (item 13) o *offset* do fragmento (*stub*) do *DOS*. Este fragmento, na verdade, é um pequeno bloco de código executável embutido no cabeçalho *PE*, que é chamado quando o arquivo *PE* não puder ser executado. (CERVEN, 2002)

Formado por num número muito pequeno de *bytes* (até por que o tamanho padrão do fragmento *DOS* é de 64 bytes), este bloco é dividido em instruções de máquina e num texto, que reproduzem uma mensagem de erro ao usuário em caso de falhas no carregamento do arquivo. Assim, ao carregar o programa na memória, é verificado se o aplicativo é compatível com o sistema e, caso não for, é desviada a execução para este bloco.

2.3.2.3 Cabeçalho do Arquivo

Conforme detalhado anteriormente no cabeçalho *DOS*, é encontrado no *offset* `0x3C` (item 19) o cabeçalho *PE* do arquivo (*IMAGE_NT_HEADER*).

Os componentes do cabeçalho do arquivo são os seguintes: (CERVEN, 2002; PIETREK, 1994; LUEVELSMEYER, 1999)

1) Assinatura do cabeçalho *PE* (4 *bytes*): é sempre a constante "`PE00`" ("`50 45 00 00`" em hexadecimal)

2) Tipo mínino de processador requerido para execução do *software* (2 *bytes*):

- `0x014c:` *Intel 80386* (padrão);
- `0x014d:` *Intel 80486*;
- `0x014e:` *Intel Pentium*;
- `0x0160:` *MIPS R3000, big endian*;
- `0x0162:` *MIPS R3000, little endian*;
- `0x0166:` *MIPS R4000, little endian*;
- `0x0168:` *MIPS R10000, little endian*;
- `0x0184:` *DEC Alpha AXP*;
- `0x01F0:` *IBM Power PC*.

3) Número de seções na tabela de seções (2 *bytes*);

4) Data e hora da criação ou modificação do arquivo (*TimeDateStamp*), armazenados em forma de segundos calculados desde 31 de Dezembro de 1969, 16:00h. (4 *bytes*);

5) Ponteiro para Tabela de Símbolos: valor para uso exclusivo dos depuradores (4 *bytes*);

6) Número de Símbolos: valor para uso exclusivo dos depuradores (4 *bytes*);

7) Tamanho do Cabeçalho Opcional: é valor deve ser idêntico ao tamanho da estrutura *IMAGE_OPTIONAL_HEADER* (2 *bytes*);

8) Características do arquivo (definidas por *bit flags*) (2 *bytes*): As mais importantes são:

- *Bit* 2: Indica se o arquivo é um executável.
- *Bit* 10: Indica se o executável pode ser executado diretamente de uma unidade removível, como o disquete, CD ou DVD. Caso não, o sistema operacional ira copiar o arquivo para um diretório temporário, executando-o dali.
- *Bit* 11: Indica se o executável pode ser executado diretamente da rede. Caso não, o sistema operacional ira copiar o arquivo para um diretório temporário, executando-o dali.
- *Bit* 13: Indica se o arquivo é uma *DLL*.
- *Bit* 14: Indica se o executável pode ser executado em sistemas com mais de um processador.

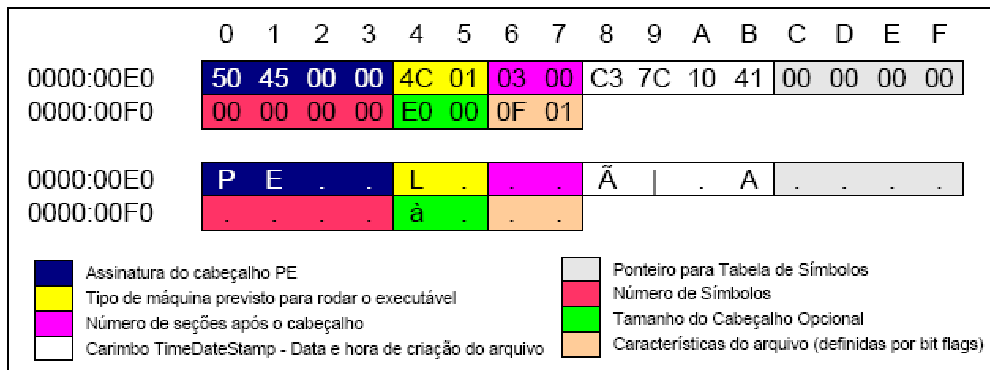

Figura 2.6 – Cabeçalho do arquivo

2.3.2.4 Cabeçalho Opcional

Imediatamente após o cabeçalho do arquivo vem o cabeçalho opcional que, apesar do nome, está sempre presente. A denominação de cabeçalho opcional é devida ao modelo *COFF*, que utiliza cabeçalhos apenas para bibliotecas, não para objetos.

Este cabeçalho fornece mais detalhes de como o *software* deve ser carregado, pois nele estão detalhados dados como endereço base da memória, além de conter informações de como o arquivo *PE* deve ser tratado pelo sistema operacional.

Os componentes do cabeçalho opcional são os seguintes: (CERVEN, 2002; PIETREK, 1994; LUEVELSMEYER, 1999)

1) Valor *Magic* (2 *bytes*): assinatura `0x010B` (em hexadecimal):

2) Byte que indica a versão do lincador (*Major Linker Version*) (1 *byte*). Em conjunto com a subversão, forma a versão do lincador. Meramente informativo, sem utilidade para o sistema operacional;

3) Byte que indica a subversão do lincador (*Minor Linker Version*) (1 *byte*);

4) Tamanho dos blocos de código executável (*SizeOfCode*) (4 *bytes*);

5) Tamanho dos blocos de dados de inicialização (segmento de dados) (4 *bytes*);

6) Tamanho dos blocos de dados de não-inicialização (segmento *BSS*) (4 *bytes*);

7) *RVA* do código do executável (4 *bytes*): indica o *RVA* do ponto de entrada para execução do *software*;

8) *RVA* da base do código (4 *bytes*);

9) *RVA* da base dos dados (4 *bytes*);

10) Endereço base da memória para remapeamento (4 *bytes*): Este endereço indica a posição inicial da memória do *software*. Assim, quando um aplicativo é carregado pelo sistema, é feita remapeamento dos endereços *RVA* (relocation), determinando assim seus endereços reais dentro da memória alocada;

11) Alinhamento da seção na *RAM* (4 *bytes*);

12) Alinhamento do arquivo em disco (4 *bytes*);

13) Indica a versão mínima necessária do sistema operacional (*Major Operating System Version*) (2 *bytes*). Em conjunto com a subversão, forma a versão mínima do sistema operacional exigido;

14) Indica a subversão mínima necessária do sistema operacional (*Minor Operating System Version*) (2 *bytes*);

15) Indica a versão do arquivo *PE* (*Major Image Version*) (2 *bytes*). Em conjunto com a subversão, forma a versão do arquivo;

16) Indica a subversão do arquivo *PE* (*Minor Image Version*) (2 *bytes*);

17) Indica a versão mínima necessária do subsistema (*Major Subsystem Version*) (2 *bytes*). Em conjunto com a subversão, forma a versão mínima do subsistema operacional exigido;

18) Indica a subversão mínima necessária do subsistema (*Minor Subsystem Version*) (2 *bytes*);

19) Versão do *Win32* (4 *bytes*): sempre zerado;

20) Tamanho da imagem (4 *bytes*): é a soma dos tamanhos dos cabeçalhos e seções. Este campo serve como dica para o carregador do sistema (*system loader*) saber quanto de memória alocar para carregar o programa;

21) Tamanho dos cabeçalhos (4 *bytes*): é a soma dos tamanhos dos cabeçalhos, incluindo diretórios de dados e cabeçalhos de seções;

22) *CheckSum* do arquivo *PE* (4 *bytes*);

23) Indica o tipo de subsistema requerido (2 *bytes*):

- `0x0002h:` *Windows GUI* (padrão);
- `0x0003h:` *Windows* console;
- `0x0005h:` *OS/2* console;
- `0x0007h:` *POSIX* console.

24) Características de *DLL* (2 *bytes*): configuração por *bit-flags*;

25) Tamanho de reserva de pilha (4 *bytes*);

26) Tamanho inicial da pilha salva (4 *bytes*)

27) Tamanho da reserva de *heap* (4 *bytes*);

28) Tamanho inicial *heap* salvo (4 *bytes*);

29) *Flags* para o carregador do sistema operacional (4 *bytes*);

30) Número e tamanho de *RVAs* (4 *bytes*);

31) Diretório de Dados: vetor com 16 descritores de diretórios, com localização (*RVA*) e o tamanho de cada peça de informação.

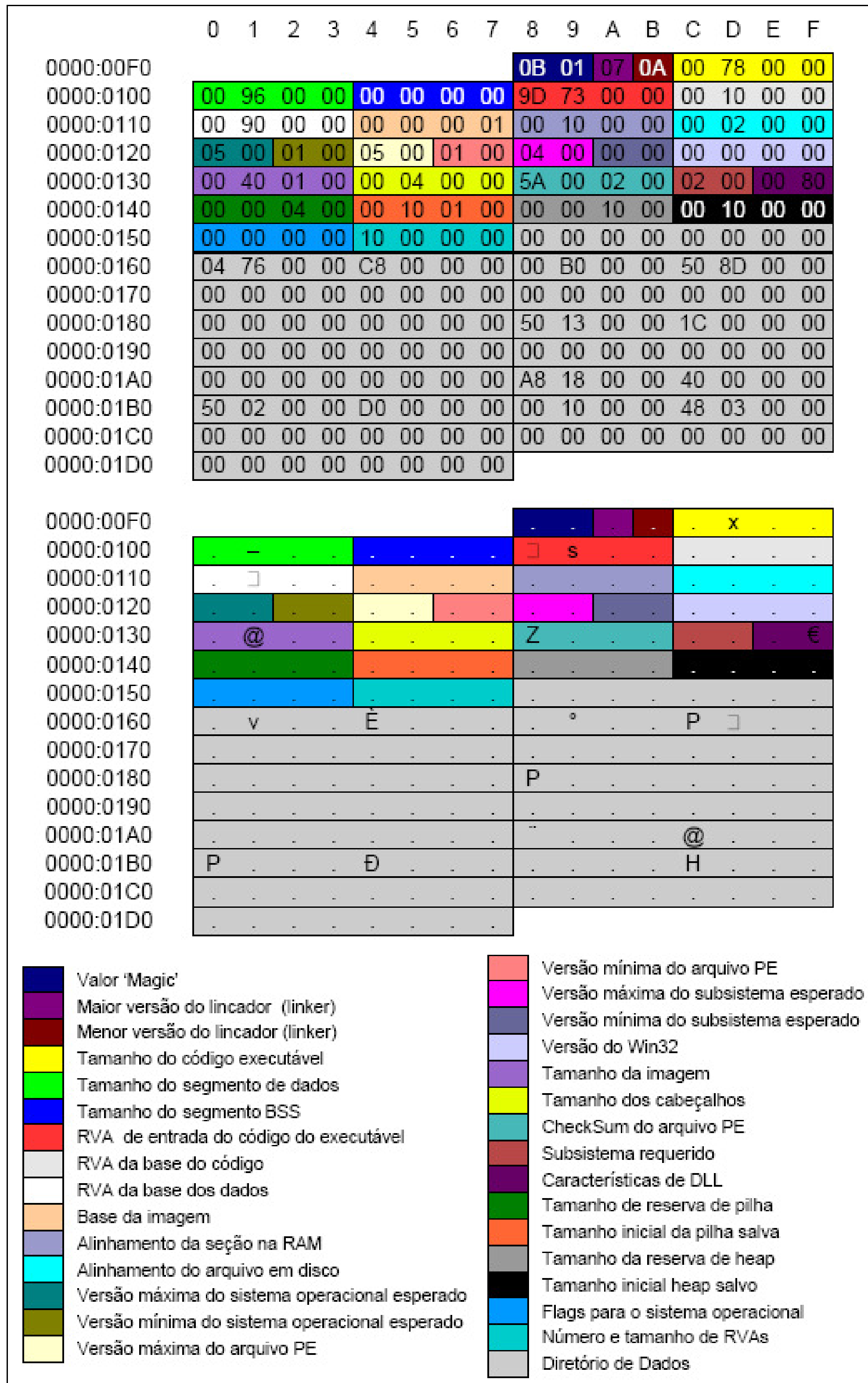

Figura 2.7 – Cabeçalho opcional

2.3.2.5 Cabeçalho de Seções

Os cabeçalhos de seção são as descrições que antecedem a seção propriamente dita. Um cabeçalho de seção contém: (CERVEN, 2002; PIETREK, 1994; LUEVELSMEYER, 1999)

1) Um *array* (8 *bytes*) com o nome da seção;
2) Tamanho virtual da seção (4 *bytes*);
3) Endereço físico da seção (4 *bytes*);
4) Tamanho alinhado (4 *bytes*) - tamanho dos dados da seção arredondado para cima para o próximo múltiplo do alinhamento de arquivo;
5) *Offset* do início do arquivo em disco até os dados da seção (4 *bytes*);
6) Ponteiro para remanejamento (4 *bytes*) – apenas para arquivos-objeto;
7) Ponteiro para números de linha (4 *bytes*) – apenas para arquivos-objeto;
8) Número de remanejamentos (2 *bytes*) – apenas para arquivos-objeto;
9) Quantidade de números de linha (2 *bytes*) – apenas para arquivos-objeto;
10) Características que descrevem como a memória da seção deve ser tratada (4 *bytes*) (por *bit flag*);

2.3.2.6 Seções

Após os cabeçalhos, seguem as seções em si. Existem vários tipos de seções, que se diferem de acordo com seu conteúdo. Nestas seções ficam salvas as instruções, recursos (*resources*) e todos os dados e rotinas do programa. Cada seção possui algumas *flags* sobre alinhamento, o tipo de dados contidos, etc. Em resumo, são nestas seções onde ficam salvas os dados do programa propriamente dito.

2.3.3 Considerações Gerais

Um bom conhecimento e compreensão de como é e como funciona o formato *PE* leva a um bom conhecimento e compreensão do sistema operacional em um todo. Conhecer

como funcionam internamente suas bibliotecas e executáveis faz com que o desenvolvedor saiba não somente sobre programação, mas também auxilia na aprendizagem e compreensão de tudo que ocorre nas aplicações e toda interação com o sistema operacional.

3 FERRAMENTAS

Conforme Eilam (2005), a expressão “um homem é tão bom quanto as suas ferramentas” se encaixa muito bem quando o assunto é Engenharia Reversa. O fato é que, sem as ferramentas adequadas, muitas vezes é impossível a reversão. O entendimento das diferenças entre cada ferramenta e sua escolha correta são pontos cruciais para o sucesso. Eilam (2005) ainda reforça que, como não existe uma ferramenta ‘faz tudo’, é importante que cada reversor selecione e crie seu arsenal.

Existem diversas ferramentas para engenharia reversa. A escolha da ferramenta certa depende de diversos fatores como: qual é o tipo de *software* a ser analisado, qual é a plataforma em que ele roda, em qual linguagem ele foi desenvolvido, qual é o tipo de informação que se deseja extrair dele, entre outros. No entanto existem 2 metodologias de análise para reversão: análise *offline* e análise em tempo real.

A análise *offline* refere-se ao exame de código a partir da desassemblagem[4] do arquivo binário do executável, interpretação e análise de cada parte extraída. Este método é muito poderoso, pois fornece muitas informações que facilitam a compreensão do funcionamento do programa. (EILAM, 2005)

Porém, uma desvantagem deste tipo de análise é que, como o estudo se baseia sobre o código do *software*, não se consegue ter um bom entendimento de como são tratados e gerenciados os dados, devido à dificuldade na reconstrução da lógica e do fluxo dos dados no sistema. Além disso, em alguns casos pode ser impossível a aplicação desta técnica, pois algum programas, protegidos por técnicas anti-reversão, podem possuir um mecanismo que mantêm o arquivo *offline* compactado, descompactando-o apenas no momento da execução.

---

[4] Neologismo que se refere ao processo de converter o código binário em código assembler.

Já na análise em tempo real, explica Eilam (2005), a conversão do *software* em código humanamente legível também ocorre, porém, ao invés de ser feito um estudo estático, o *software* é executado e analisado dentro de um depurador. Esta ferramenta permite observar todo estado da execução do *software*, fornecendo dados referentes à memória interna, fluxo dos dados e do programa, entre diversos outros recursos.

Devido à essas funcionalidades extras, a análise em tempo real é sugerida a principiantes, pois facilita relativamente o trabalho e entendimento do fluxo e gerenciamento dos dados.

3.1 Depuradores (*Debuggers*)

Conforme definem Hoglund e McGraw (2006), depurador é um programa que se conecta e controla outros programas, permitindo sua análise durante a própria execução, auxiliando na determinação do fluxo lógico do programa. Um depurador permite a execução passo-a-passo do código, inclusão de pontos de interrupção (*breakpoints*) ativados de acordo com suas configurações, rastreamento de código e dados, além de permitir a visualização das variáveis e do estado da memória do *software*, tudo em tempo real.

Os depuradores dividem-se em duas categorias:

- Depuradores em modo de usuário: Funcionam como programas normais sob o sistema operacional, permitindo e depurando unicamente o *software* em foco. Um bom exemplo de depurador em modo de usuário (*freeware*) é o OllyDbg, encontrando no endereço <http://www.ollydbg.de>.
- Depuradores em modo de *kernel*: Instalam-se no sistema, conseguindo assim depurar programas, dispositivos e até o próprio sistema operacional. Um dos depuradores em modo *kernel* mais conhecidos é o *SoftIce* (*shareware*), encontrado no endereço <http://www.compuware.com/products/driverstudio/ds/softice.htm>.

3.2 *Disassemblers*

Um *disassembler* é uma ferramenta que converte código de máquina (binário) em código *assembly*, traduzindo quais instruções de máquina estão sendo utilizadas no código, explicam Hoglund e McGraw (2006).

A tradução do código de máquina para *assembly*, assim como as instruções de máquina, são dependentes de plataforma, ou seja, para cada modelo de processador existe um *disassembler*.

O *disassembler* mais conhecido para processadores *IA-32* é o *IDA Pro* (*shareware*), encontrado no endereço <http://www.datarescue.com>. Para *.NET* existe um *disassembler* chamado *ILDasm*, que converte a partir do código *MSIL*, que é encontrado no endereço <http://msdn2.microsoft.com/en-us/library/f7dy01k1(vs.80).aspx>.

3.3 Descompiladores ou compiladores reversos

Um descompilador é uma ferramenta que converte um código *assembly* ou código de máquina em um código-fonte de uma linguagem de nível mais alto como C, explanam Hoglund e McGraw (2006).

Essas ferramentas são extremamente úteis para determinar a lógica de nível mais alto, como *loops*, *switches* e instruções *if-then*. Os descompiladores são muito parecidos com os *disassemblers*, porém, leva o processo um passo à diante, melhorando e facilitando a compreensão.

Existem diversos descompiladores disponíveis para os diversos tipos de linguagem alto nível. Alguns bons exemplos de descompiladores para a linguagem C são: o *DCC* (*freeware*) e o *Bommerang* (*open source*), encontrados nos endereços <http://www.itee.uq.edu.au/~cristina/dcc.html> e <http://boomerang.sourceforge.net>, respectivamente.

3.4 Ferramentas de Monitoramento de Sistema

Outro tipo importante de ferramenta para análise de programas são as ferramentas de monitoramento de sistema, lembra Eilam (2005). Estas, são essenciais nos casos em que há a necessidade de monitoramento de acesso à arquivos e registro do sistema, protocolos e portas

de rede, visualização de processos, entre diversos outros processos e dispositivos ativos no sistema.

Devido à grande variedade de dispositivos e recursos monitoráveis, existem diversas ferramentas disponíveis. Alguns exemplos são:

- *FileMon* (*freeware*): utilitário que monitora e exibe a atividade do sistema de arquivos em um sistema em tempo real. Com avançados recursos é uma ferramenta eficaz para explorar o modo como o Windows funciona, observando de que maneira os aplicativos utilizam os arquivos e as DLLs, ou rastreando problemas nas configurações do arquivo do sistema ou do aplicativo. Encontrado no endereço <http://www.microsoft.com/brasil/technet/sysinternals/utilities/ filemon.mspx>.
- *RegMon* (*freeware*): utilitário de monitoramento do Registro que mostra os aplicativos que estão acessando o seu Registro, as chaves que estão acessando e os dados do Registro que eles estão lendo e gravando, tudo em tempo real. Encontrado no endereço <http://www.microsoft.com/brasil/technet/sysinternals/ utilities/regmon.mspx>.
- *Process Monitor* (*freeware*): utilitário que combina os recursos dos utilitários *FileMon* e *RegMon*, já que estes foram descontinuados para as versões *XP SP2* e *Vista* do *Windows*. Encontrado no endereço <http://www.microsoft.com/brasil/ technet/sysinternals/utilities/processmonitor.mspx>.
- *TCPView* (*freeware*): utilitário que exibe listas detalhadas de todos os pontos de extremidade *TCP* e *UDP* do sistema, incluindo endereços locais e remotos e o estado das conexões *TCP*. Encontrado no endereço <http://www.microsoft.com/ brasil/technet/sysinternals/utilities/tcpview.mspx>.
- *Ethereal* (*open source*): utilitário que monitora toda atividade de rede, possibilitando a visualização de pacotes que nela trafegam (como um *sniffer*), detalhando os pacotes recebidos e enviados de acordo com o protocolo. Encontrado no endereço <http://www.ethereal.com>.
- *PortMon* (*freeware*): utilitário que monitora e exibe toda a atividade das portas seriais e paralelas em um sistema. Os recursos avançados de filtragem e pesquisa são ferramentas eficazes para explorar o modo como o *Windows* funciona, observando de que maneira os aplicativos utilizam as portas, ou rastreando

problemas nas configurações do arquivo do sistema ou do aplicativo. Encontrado no endereço <http://www.microsoft.com/brasil/technet/sysinternals/utilities/portmon.mspx>.

- *WinObj* (*freeware*): utilitário para acesso e exibição de informações sobre o *namespace* do Gerenciador de Objetos, possibilitando o rastreamento de problemas em objetos. Encontrado no endereço <http://www.microsoft.com/brasil/technet/sysinternals/systeminformation/winobj.mspx>
- *Process Explorer* (*freeware*): utilitário que exibe uma enorme quantidade de informações sobre os processos em execução no sistema, possibilitando a visualização de todos os recursos por ele abertos ou carregados, como *handles* de arquivo, serviços de sistema vinculados, *debug* de memória, entre outras valiosas informações. Encontrado no endereço <http://www.microsoft.com/brasil/technet/sysinternals/Security/ProcessExplorer.mspx>.

3.5 *Dumping*

As ferramentas de *dumping*, também conhecidas por *core dump*, servem para criar uma imagem (binária, em disco) do estado atual da memória de um programa em execução. Inicialmente, estas ferramentas foram desenvolvidas para facilitar o processo de depuração de erros, já que ali estariam todos os estados internos do *software* no momento da falha. (EILAM, 2005; WIKIPEDIA, 2007a).

Porém esta ferramenta pode servir também para estudo na quebra de proteções dinâmicas de *softwares*, ou seja, proteção que compactam e encriptam dados tornando-os ‘legíveis’ apenas durante sua execução.

As ferramentas mais conhecidas para *dumping* são: o *DumpBin*, fornecido junto ao *Microsoft Visual C++*, e o *ProcDump*, encontrado no endereço <http://www.fortunecity.com/millenium/firemansam/962/html/procdump.html>

3.6 Ferramentas de *Patching*

Este grupo de ferramentas é formado por editores hexadecimais, que são utilizados para edição de todo e qualquer tipo de arquivo em formato binário, explica Eilam (2005).

Estas ferramentas não são exclusividades da Engenharia Reversa, porém, são itens obrigatórios no pacote de ferramentas de um engenheiro.

Estes editores servem para, depois de detectado o erro num *software*, ajustar os *bytes* (instruções de máquina) relativos a este, de forma a ajustar o fluxo do programa e corrigir seu processamento. É também muito utilizado por *crackers* para desativação de métodos de proteção ou validação de um *software*.

Dois ótimos editores hexadecimais são o *Hiew* (*shareware*) – que possui uma funcionalidade muito interessante que é a possibilidade de edição do arquivo a partir da digitação do código *Assembly* – encontrado no endereço <http://www.hiew.ru> e o *Hex Workshop* (*shareware*) encontrado no endereço <http://www.bpsoft.com>.

3.7 Outras ferramentas

Apesar de demonstradas diversas ferramentas para auxílio na análise de *software* e seu comportamento, Eilam (2005) lembra que, além destas, existe um número incontável de outras ferramentas para os mais diversos fins.

Um tipo interessante de ferramenta que ainda merece destaque é a de visualizadores de arquivos no formato *PE*. Nelas é possível ver detalhada e identificadamente cada *byte* de um arquivo *PE*, como os cabeçalhos e seções do mesmo. Um bom exemplo deste tipo de ferramenta é o *PEView* (*freeware*), encontrado em <http://www.magma.ca/~wjr>,.

No entanto, a criação e seleção dos utilitários que comporão o pacote de ferramentas de um reversor variarão de acordo com seu gosto, intimidade e conhecimento junto ao processo.

CONCLUSÃO

Esta fase do trabalho de conclusão de curso teve como objetivo apresentar e introduzir a Engenharia Reversa aplicada à exploração e extração de conhecimento de arquivos *Portable Executable* no ambiente *Windows*.

Foram encontradas dificuldades nas pesquisas, principalmente no que diz respeito à bibliografia. Muitos dos livros obtidos não refletiam as necessidades do trabalho, tornando um tanto escassas as referências bibliográficas. No entanto, isto já era esperado. Tanto, que este foi um dos maiores motivadores na redação deste trabalho.

A disseminação deste assunto e a importância de seu domínio no desenvolvimento de aplicativos podem fazer com que programadores previnam futuros inconvenientes que exponham a qualidade ou segurança de seus sistemas.

Como continuação deste trabalho, pretende-se estudar mais a fundo o funcionamento dos aplicativos no ambiente *Windows*, demonstrando os erros mais comuns na implementação de *software* relacionando-os aos tipos mais freqüentes de ataques, além de demonstrar as práticas e métodos para construção segura de *software*, aliadas às técnicas de proteção anti-reversão.

REFERÊNCIAS BIBLIOGRÁFICAS

AURÉLIO, Novo Dicionário Eletrônico. Rio de Janeiro: Editora Positivo, [2004]. CD-ROM. Windows XP.

BRAND, Stewart. **The Physicist**. Disponível em <http://www.wired.com/wired/archive/3.09/myhrvold.html>. Acesso em: 15 mai 2007.

CERVEN, Pavol. **Crackproof Your Software**. São Francisco, Estados Unidos da América: No Starch Press, 2002. 170p.

DENNING, Dorothy. **Information Warfare & Security**. Massachusetts, Estados Unidos da América: Addison-Wesley Professional, 1999. 544p.

EILAM, Eldad. **Reversing**: Secrets of Reverse Engineering. Indianápolis, Estados Unidos da América: Wiley Publishing, Inc., 2005. 589p.

FOTOPOULOS, Fotis. **Reverse Engineering**: In Computers Applications. Estados Unidos da América: MIT Lecture Notes, 2001.119p.

HOGLUND, Greg; McGRAW, Gary. **Como quebrar códigos**: A Arte de Explorar (e Proteger) Software. São Paulo: Pearson Education do Brasil, 2006. 424p.

INTEL. **Intel Architecture Software Developer's Manual**: Volume 2: Instruction Set Reference. Estados Unidos da América. 1999. 854p.

LUEVELSMEYER, Bernd. **The PE file format**, 1999. Disponível em <http://webster.cs.ucr.edu/Page_TechDocs/pe.txt>. Acesso em 16 out 2005.

McGRAW, Gary. **Software Security**: Building Security In. Estados Unidos da América: Addison-Wesley Professional, 2006. 448p.

MICROSOFT, Corporation. **Microsoft Portable Executable and Common Object File Format Specification**. Revision 8.0. Estados Unidos da América. 2006. 65p.

NASA. **Mars Climate Orbiter Mishap Investigation Board**: Phase I Report. 1999. Disponível em <ftp://ftp.hq.nasa.gov/pub/pao/reports/1999/MCO_report.pdf>. Acesso em: 10 jun 2007. 44p.

PIETREK, Matt. **An In-Depth Look into the Win32 Portable Executable File Format**. 2002. Disponível em <http://msdn.microsoft.com/msdnmag/issues/02/02/PE/>. Acesso em: 15 mai 2007.

PIETREK, Matt. **Peering Inside the PE**: A Tour of the Win32 Portable Executable File Format. 1994. Disponível em <http://msdn2.microsoft.com/en-US/library/ms809762.aspx>. Acesso em: 15 mai 2007.

PRODANOV, Cleber. **Manual de Metodologia Científica**. 3ª ed. Novo Hamburgo: FEEVALE, 2003. 79p.

SCHNEIER, Bruce. **Secrets and Lies**: Digital Security in a Networked World. Estados Unidos da América: John Wiley & Sons, 2000.

SWIFT. **Annual Report 2006**, 2007. Disponível em <http://www.swift.com/index.cfm?item_ id=61861>. Acesso em: 12 jun 2007. 68p.

SYMANTEC. **Symantec Internet Security Threat Report**: Trends for July–December 06 - Volume XI. Disponível em <http://eval.symantec.com/mktginfo/enterprise/white_papers/ent-whitepaper_internet_security_threat_report_xi_03_2007.en-us.pdf>. Acesso em: 03 abr 2007. 104p.

SZOR, Peter. The Art of **Computer Virus Research and Defense**. Estados Unidos da América: Addison-Wesley Professional, 2005. 744p.

WIKIPEDIA. **Core dump**, 2007a. Disponível em: <http://en.wikipedia.org/wiki/ Core_dump>. Acesso em: 10 jun. 2007.

WIKIPEDIA. **Fifth-generation programming language**, 2007b. Disponível em: <http://en.wikipedia.org/wiki/Fifth-generation_programming_language>. Acesso em: 08 jun. 2007.

WIKIPEDIA. **First-generation programming language**, 2007c. Disponível em: <http://en.wikipedia.org/wiki/First-generation_programming_language>. Acesso em: 08 jun. 2007.

WIKIPEDIA. **Fourth-generation programming language**, 2007d. Disponível em: <http://en.wikipedia.org/wiki/Fourth-generation_programming_language>. Acesso em: 08 jun. 2007.

WIKIPEDIA. **Second-generation programming language**, 2007e. Disponível em: <http://en.wikipedia.org/wiki/Second-generation_programming_language>. Acesso em: 08 jun. 2007.

WIKIPEDIA. **Third-generation programming language**, 2007f. Disponível em: <http://en.wikipedia.org/wiki/Third-generation_programming_language>. Acesso em: 08 jun. 2007.

WIKIPEDIA. **Very high-level programming language**, 2007g. Disponível em: <http://en.wikipedia.org/wiki/Very_high-level_programming_language>. Acesso em: 08 jun. 2007.