CENTRO UNIVERSITÁRIO FEEVALE

DANIEL ROBERTO DUMMER

ENGENHARIA REVERSA APLICADA À EXPLORAÇÃO E PROTEÇÃO DE SOFTWARE

Novo Hamburgo, novembro 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, novembro 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, confia-se em *patches* e atualizações diárias de sistema, *Firewalls*, Anti-vírus e *IDS*, dentre vários recursos, sem saber, muitas vezes, 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*. Um simples adágio popular reflete bem essa situação: "É melhor prevenir a remediar". A criação de aplicativos de qualidade superior evitaria a maior parte destas ferramentas reparatórias, como as 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 permitir efetuar testes 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 sua qualidade, tanto na questão segurança quanto no desempenho. Nesta ótica, este trabalho tem por objetivo estudar a engenharia reversa de *software* demonstrando, a partir de suas técnicas de monitoramento e análise, o funcionamento de aplicativos 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, analisados diante os aspectos legais e éticos da aplicação da engenharia reversa.

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

ABSTRACT

The complete adoption of information systems such in industrial, commercial and personal spheres, each day the amount of data, confidential or not, grows exponentially, stored in the computers around the world. This growth is followed by an increase in the number of bad-intentioned hackers who take advantage of their deep computer and systems knowledge to steal these data. Added to this digital espionage, everyday new software breaches and viruses appear, spywares and malwares are created, threatening the privacy and affecting their security. To fight this, there's general trust in system patches and daily updates, Firewalls, Anti-viruses and IDS, amongst other tools and resources, even if without knowing what they're for nor what they protect or solve. However, if during the development of these systems, higher attention be destined to security issues, it wouldn't be necessary to invest so much time, money and work in bug corrections. A simple analogy with a popular saying reflects well this situation: "It is better to prevent than to remedy". The creation of superior quality applications would avoid most part of these fixing or filtering tools, and of tools for detecting unauthorized or bad-intentioned access to our data. The Software's Reverse Engineering helps the developer to understand how the software works internally, making it possible to study and learn the software's structure and logic by analyzing only the final product, and also to facilitate thorough tests, restore lost information in non-availability cases or losses of source code, include and modify functions to the software after it is finished and improve the software's quality in security as well as in performance. Therefore, this work has aims at studying software reverse engineering by demonstrating, through monitoring and analyzing, the functioning of applications of format Win32/Portable Executable file format, under the operational system Windows, and also to present the most common types of attacks to software and techniques of protection, besides arguing the legal and ethical aspects of this technique.

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

LISTA DE FIGURAS

Figura 1.1 – Esquema engenharia ____ 18

Figura 1.2 – Esquema engenharia reversa ____ 18

Figura 2.1 – Função em alto nível (código em C) ____ 32

Figura 2.2 – Definição e disposição dos registradores ____ 34

Figura 2.3 – Registrador *EFLAGS* ____ 35

Figura 2.4 – Análise da pilha após inserção de 3 valores ____ 37

Figura 2.5 – Análise da pilha após exclusão de 3 valores ____ 37

Figura 2.6 – Modelo de instrução *Assembly* ____ 38

Figura 2.7 – Sintaxe da instrução *Assembly* MOV ____ 39

Figura 2.8 – Sintaxe da instrução *Assembly* PUSH ____ 40

Figura 2.9 – Sintaxe da instrução *Assembly* POP ____ 40

Figura 2.10 – Sintaxe da instrução *Assembly* CMP ____ 41

Figura 2.11 – Sintaxe das instruções de salto em *Assembly* ____ 41

Figura 2.12 – Sintaxe da instrução *Assembly* CALL ____ 42

Figura 2.13 – Estrutura do formato *PE* ____ 44

Figura 2.14 – Cabeçalho DOS ____ 46

Figura 2.15 – Cabeçalho do arquivo ____ 48

Figura 2.16 – Cabeçalho opcional ____ 52

Figura 4.1 – Regiões da memória de um processo ____ 64

Figura 4.2 – Exemplo ilustrativo para análise do prólogo e epílogo ___________________ 65

Figura 4.3 – Código desassemblado do exemplo de prólogo e epílogo _________________ 65

Figura 4.4 – Código exemplo de prólogo ____ 66

Figura 4.5 – Estado da memória da função *function* após processo de prólogo ___________ 66

Figura 4.6 – Exemplo de um *buffer overflow* ______________________________________ 67

Figura 4.7 – Estado da memória após um estouro de *buffer* ________________________ 67

Figura 4.8 – Exploração da vulnerabilidade de erro de ponto num servidor *IIS* __________ 69

Figura 4.9 – Exploração da vulnerabilidade por caminhos relativos num servidor *IIS* _____ 69

Figura 4.10 – Código desassemblado de inicialização do programa Splish.exe __________ 71

Figura 4.11 – Instruções da função de exibição da propaganda do aplicativo Splish.exe ___ 72

Figura 4.12 – Rotina de cálculo de senhas sobre o nome do usuário para um buffer interno 75

Figura 4.13 – Rotina de cálculo transcrita para uma linguagem alto nível (*Delphi*) _______ 75

Figura 4.14 – Rotina de cálculo de senhas da senha informada para outro buffer interno __ 76

Figura 4.15 – Rotina de cálculo transcrita para uma linguagem alto nível (*Delphi*) (2) ____ 76

Figura 4.16 – Rotina de validação dos *buffers* e conferência das senhas ________________ 78

Figura 5.1 – Código binário da instrução *API IsDebuggerPresent* ____________________ 82

Figura 5.2 – Código para detecção de um depurador via *Trap Flag* ___________________ 84

Figura 5.3 – Código para detecção de um depurador via busca da instrução *Int 3h* _______ 87

Figura 5.4 – Código para detecção do depurador *SoftICE* via busca em memória ________ 90

Figura 5.5 – Código para detecção do depurador *SoftICE* via abertura de seus *drivers* ____ 92

Figura 5.6 – Código para inserção de um *byte* inválido ao código ____________________ 94

Figura 5.7 – Resultado da desassemblagem do código pelo depurador *SoftICE* __________ 94

Figura 5.8 – Resultado da desassemblagem do código pelo depurador *OllyDbg* _________ 95

Figura 5.9 – Predicado opaco falso ____ 96

Figura 5.10 – Predicado opaco verdadeiro _______________________________________ 96

Figura 5.11 – Código exemplo utilizado para testes de desassemblagem _______________ 97

Figura 5.12 – Resultado da desassemblagem do código exemplo pelo depurador *IDA PRO* 97

Figura 5.13 – Resultado da desassemblagem do código exemplo pelo depurador *OllyDbg* _ 97

Figura 5.14 – Código exemplo anti-desassemblagem utilizando saltos indiretos _________ 98

Figura 5.15 – Ofuscação de código via transformações no tratamento de dados_________ 102

Figura 5.16 – Código para proteção anti-descarregamento de memória via *ProcDump* ___ 104

Figura 5.17 – Cabeçalho opcional exemplo de proteção via edição de cabeçalho *PE*_____ 105

LISTA DE QUADROS

Quadro 2.1 – Descrição dos registradores e suas relativas funções ____________________ 34

Quadro 2.2 – Exemplos de referências de operandos de instruções e seus significados ____ 38

Quadro 2.3 – Instruções para as operações aritméticas básicas _______________________ 40

Quadro 5.1 – Depuradores e seus métodos de desassemblagem ______________________ 93

LISTA DE TABELAS

Tabela 2.1 – Instruções por função ____ 30

LISTA DE ABREVIATURAS E SIGLAS

API	*Application Programming Interface*
ASP	*Active Server Pages*
COFF	*Common Object File Format*
CPU	*Central Processing Unit*
CRC	*Cyclic Redundancy Check*
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*
IAT	*Import Address Table*
IIS	*Internet Information Services*
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*
SEH	*Structured Exception Handling*
TCP	*Transmission Control Protocol*
UDP	*User Datagram Protocol*
Win32	*Windows 32 bits*

SUMÁRIO

INTRODUÇÃO ______ **15**

1 ENGENHARIA REVERSA ______ **18**

1.1 Importância da aplicação da Engenharia Reversa ______ 19

1.2 História ______ 20

1.3 A Origem dos Problemas ______ 21

1.3.1 Complexidade ______ 22

1.3.2 Extensibilidade ______ 22

1.3.3 Conectividade ______ 23

1.4 Análise de *Software* ______ 24

1.4.1 Análise de Caixa-Branca ______ 25

1.4.2 Análise de Caixa-Preta ______ 25

1.4.3 Análise de Caixa Cinza ______ 26

1.4.4 Qualidade na análise ______ 26

1.5 Segurança de *Software* ______ 26

1.5.1 Terminologia ______ 27

1.6 Engenharia Reversa versus *Cracking* ______ 28

2 SISTEMAS E FORMATOS ______ **29**

2.1 Linguagens de Programação ______ 29

2.2 *Assembly* ______ 31

2.2.1 Gerenciamento de dados em baixo nível ______ 32

2.2.2 Registradores ______ 33

2.2.3 Identificadores (*Flags*) ______ 35

2.2.4 Pilha (*Stack*) ______ 36

2.2.5 Interrupções ______ 38

2.2.6 Instruções ______ 38

2.2.6.1 Movimentação de dados ____ 39
2.2.6.2 Manipulando a pilha ____ 39
2.2.6.3 Operações aritméticas ____ 40
2.2.6.4 Comparação de dados ____ 41
2.2.6.5 Execução condicional ____ 41
2.2.6.6 Chamada de Funções ____ 41
2.2.6.7 Outras funções ____ 42
2.3 Formato *Win32/Portable Executable* ____ 42
2.3.1 História ____ 43
2.3.2 Formato *Portable Executable* ____ 43
2.3.2.1 Cabeçalho *DOS* ____ 44
2.3.2.2 Fragmento (*Stub*) do *DOS* ____ 47
2.3.2.3 Cabeçalho do Arquivo ____ 47
2.3.2.4 Cabeçalho Opcional ____ 49
2.3.2.5 Cabeçalho de Seções ____ 53
2.3.2.6 Seções ____ 53
2.3.3 Considerações Gerais ____ 53
3 FERRAMENTAS ____ 55
3.1 Depuradores (*Debuggers*) ____ 56
3.2 Desassembladores *(Disassemblers)* ____ 57
3.3 Descompiladores ou compiladores reversos ____ 58
3.4 Ferramentas de Monitoramento de Sistema ____ 58
3.5 *Dumping* ____ 60
3.6 Ferramentas de *Patching* ____ 60
3.7 Outras ferramentas ____ 61
4 ATAQUES COMUNS ____ 62
4.1 Buffer Overflow ou estouro de *buffer* ____ 62
4.1.1 Organização da memória de um processo ____ 63
4.1.2 Estourando o *buffer* ____ 66
4.2 Entradas maliciosas ____ 68
4.3 *Patching* ____ 70
4.4 *Keygenning* ____ 73
5 PROTEÇÕES E TÉCNICAS ANTI-REVERSÃO ____ 79
5.1 Anti-depuradores ____ 81
5.1.1 Detecção via *API IsDebuggerPresent* ____ 81

5.1.2 Detecção via estado lógico da *Trap Flag* 83
5.1.3 Detecção via busca da instrução *Int 3h* 84
5.1.4 Detectando o depurador *NuMega® SoftICE* / *Compuware® DriverStudio* 87
5.1.4.1 Detectando por busca em memória 88
5.1.4.2 Detectando pela abertura dos *drivers SICE* e *NTICE* 90
5.1.4.3 Detecção via registro do *Windows* e pasta de instalação 92
5.2 Anti-desassemblagem 93
5.2.1 Conversão Linear (*Linear Sweep*) 93
5.2.2 Conversão Recursiva (*Recursive Traversal*) 95
5.3 Criptografia de código 99
5.4 Ofuscação de código 100
5.4.1 Transformações no Controle de Fluxo 100
5.4.2 Transformações no Tratamento de Dados 101
5.5 Anti-patching 102
5.6 Anti-descarregamento de memória 103
5.7 Edição do Cabeçalho *PE* 105
5.7.1 Alteração do endereço base do aplicativo 106
5.7.2 Anti-*OllyDbg* e Anti-*SoftICE* 106
5.7.3 Alteração da *flag* de características da seção *.code* 106
5.8 Aspectos Legais e Ética na aplicação da Engenharia Reversa 106
CONCLUSÃO **108**
REFERÊNCIAS BIBLIOGRÁFICAS **110**

INTRODUÇÃO

Cresce diariamente a importância do quesito segurança nos *softwares*. As estatísticas comprovam: no segundo semestre de 2006, a Symantec (2007) 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) Do contrário, conforme definem Hoglund e McGraw (2006, p. 1) "quase todos os sistemas modernos têm o mesmo calcanhar-de-aquiles, que é o *software*". (HOGLUND; McGRAW, 2006, p.1)

Associado a isto, o custo, o trabalho e o 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 de sistemas operacionais -, contra os 21 dias necessários pela *Microsoft*, o menor analisado na categoria. (Symantec Internet Security Threat Report XI, 2007) Viega e McGraw, apud Hoglund e McGraw (2006, p. 1), lamentavelmente, destacam que "não seria necessário empregar tanto tempo, dinheiro e trabalho em segurança de rede se não tivéssemos uma segurança de *software* tão ruim".

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 analisadas 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 evidenciados os problemas de implementação, sintaxe e tratamento de erros, e apresentadas 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*. As técnicas de engenharia reversa são utilizadas para 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

---

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

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 e 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, serão destacadas a apresentação do conceito, a história, a origem, a importância e a utilidade da engenharia reversa, analisados sob o sistema operacional *Windows*, no formato de arquivos *Win32/Portable Executable*. Da mesma forma, serão apresentadas as ferramentas de auxílio de extração de informações, funcionamento e estrutura do *software*, juntamente com os tipos de ataques mais comuns, técnicas e métodos de proteção anti-reversão, finalizando com uma discussão sobre a ética e os aspectos legais da aplicação de engenharia reversa de *software*.

Este trabalho será composto de 5 capítulos. No primeiro capítulo serão apresentados o conceito e a importância da engenharia reversa de *software*, a origem e sua história, além de discutir o que é segurança de *software*, quais os tipos de análise e as raízes de seus problemas.

No segundo capítulo será apresentado o formato de arquivo *Win32/Portable Executable*, sua estrutura e funcionamento no ambiente *Windows 32 bits*, além de abordar os tipos e níveis das linguagens de programação atuais, em especial a linguagem *Assembly*.

No terceiro capítulo serão relacionadas as ferramentas da engenharia reversa, definindo os tipos e apresentando alguns utilitários como exemplos.

No quarto capítulo serão abordados os tipos mais freqüentes de ataques, como se constituem e como são executados. Em defesa à estes ataques, no quinto capítulo serão exemplificadas técnicas e métodos de proteção. Finalizando o quinto capítulo e o trabalho, serão levantadas e discutidas as questões éticas da aplicação da engenharia reversa.

1 ENGENHARIA REVERSA

Segundo Paula Filho (2003, p. 1), 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

Imagem do autor

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

Imagem do autor

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 às 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, p.) (Tradução nossa)

E essa relação *LOC* versus *bugs*, justifica a afirmação de que, quanto maior a complexidade, maior a probabilidade de ocorrência de *bugs*.

De acordo com Brand (1995), estima-se que o número de *bugs* de um sistema varia de 5 a 50 *bugs* por *KLOC* (mil *LOC*). Imagine isto em sistemas operacionais como *Windows* 95 que possuía 5 milhões de *LOC*, o *Linux* que possuía 1,5 milhões de *LOC* e o *Windows XP* que possui cerca de 40 milhões de *LOC*. Se fossem levando ao pé da letra, a resistência na migração de um sistema, com certeza, seria muito maior.

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, bloqueando dessa forma 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, bem 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 controlado por redes zumbis[2] - é possível programar e acionar ataques automatizados à sites, empresas e instituições, 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, visto que esta 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*.

Essa interconexão e compartilhamento de informações entre 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

---

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

catástrofe instantânea, desestabilizando economias inteiras, alertam Hoglund e McGraw (2006).

Logicamente, trata-se de um risco moderno que tem 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.

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)

Muitas vezes problemas descobertos em análises de 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 de 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

---

[3] Vulnerabilidades encontradas equivocadamente, não existentes.

explorável, pois, como citado anteriormente, nem sempre é possível explorar todas as vulnerabilidades devido às 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) Por 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 a um *software*.

1.6 Engenharia Reversa versus *Cracking*

O conceito de engenharia reversa muitas vezes é confundido o 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 se apropriar dos métodos de engenharia reversa com a finalidade de burlar essas proteções. O *cracker* pode estudar o *software* até adquirir conhecimento a tal nível que seja 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 divididas 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 entendimento e utilização mais fáceis. 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-exemplo 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 deste programa-exemplo 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 este 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, serão apresentandos 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 da figura 2.1, escrita em código C, pode-se fazer uma analogia entre o baixo e o alto nível das linguagens.

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

Figura 2.1 – Função em alto nível (código em C)

Fonte: Eilam, 2005, p. 37

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 se pode perceber, 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 é utilizável 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 2.2 ilustra esta disposição.

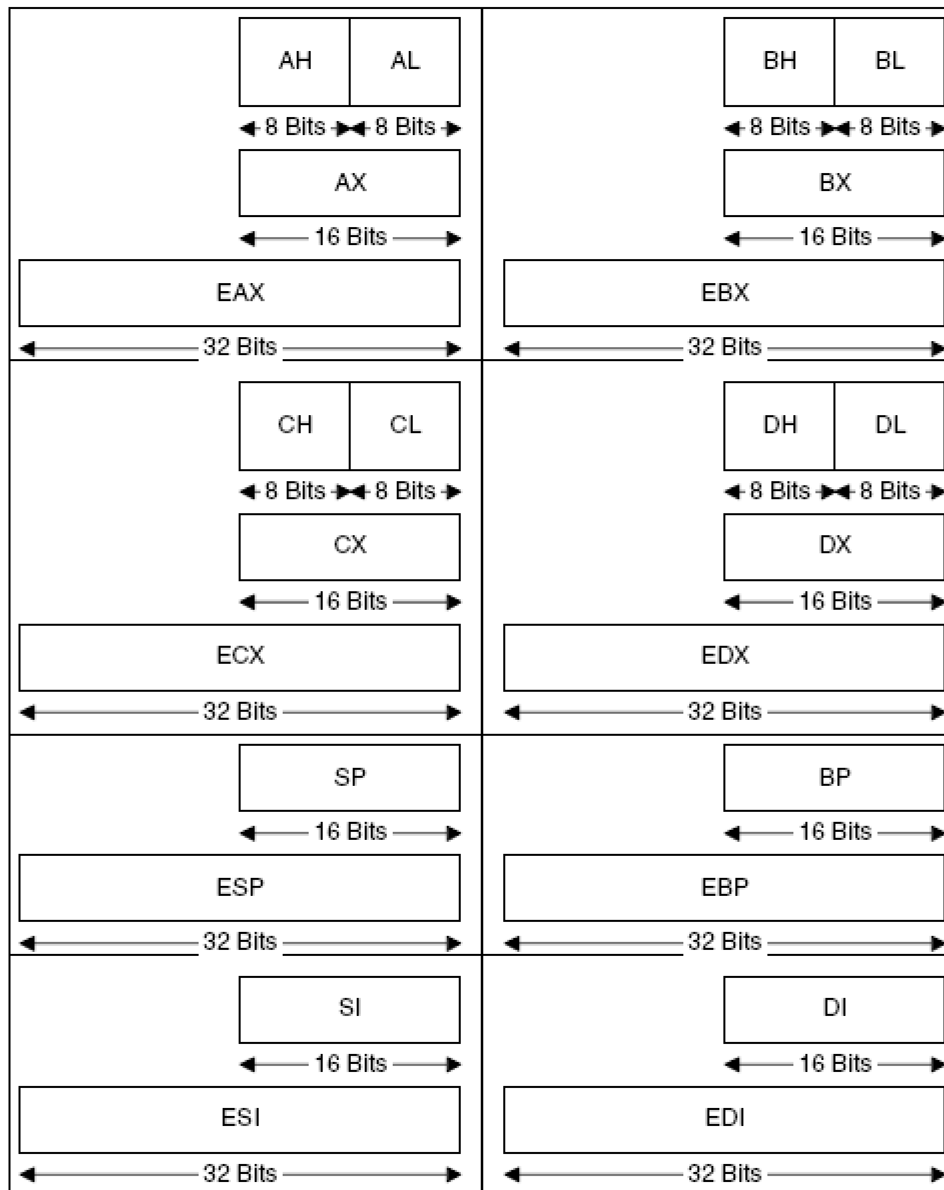


Figura 2.2 – Definição e disposição dos registradores

Fonte: 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).

Registrador	Função
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 de pilha base de funções, pois quando utilizado em conjunto com o ponteiro da pilha, serve para definir o *frame* das variáveis 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.

Fonte: Adaptado de Eilam, 2005, p. 45

2.2.3 Identificadores (*Flags*)

Eilam (2005) ressalta que, além dos registradores demonstrados, os processadores *IA-32* possuem um registrador interno (32 *bits*) 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.

Os identificadores lógicos são divididos em 3 categorias: de sistema, de controle e de estado (INTEL, 1999a); dispostos conforme a figura 2.3:

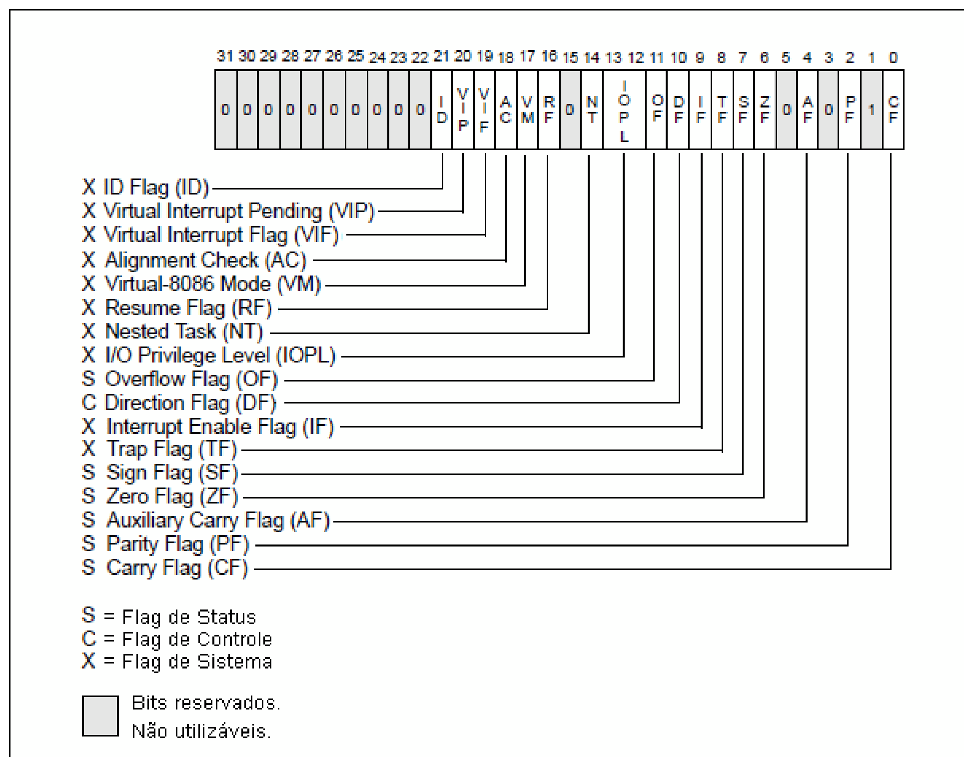


Figura 2.3 – Registrador *EFLAGS*

Fonte: Imagem do autor, adaptado de INTEL, 1999a, p. 57

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, 1999a)

Um caso típico acontece no salto condicional, que processa uma instrução de comparação, testando os operandos e marcando o resultado na *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, ela funciona pelo sistema *LIFO* (*Last In, First Out*) *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 2.4 e 2.5 demonstram como a pilha controla a entrada e saída dos valores.

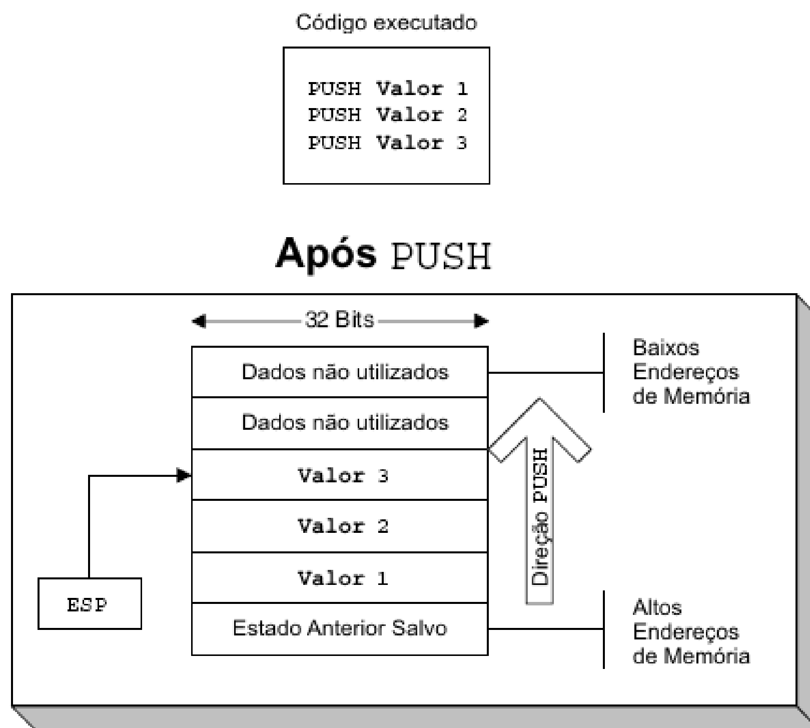


Figura 2.4 – Análise da pilha após inserção de 3 valores

Fonte: Imagem do autor, adaptado de Eilam, 2005, p. 41

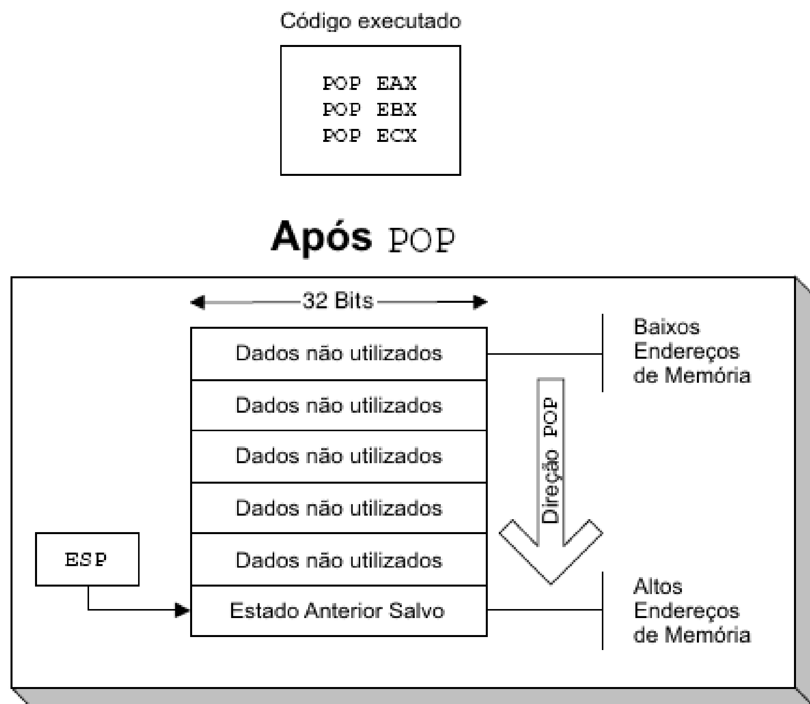


Figura 2.5 – Análise da pilha após exclusão de 3 valores

Fonte: 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, 1999b) 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 serve 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.

Fonte: Adaptado de Eilam, 2005, p. 48

Quanto à sintaxe, as instruções são normalmente dispostas conforme a figura 2.6:

```
Instrução Operando1, Operando2
```

Figura 2.6 – Modelo de instrução *Assembly*

Fonte: Imagem do Autor

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 (2005). 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
```

Figura 2.7 – Sintaxe da instrução *Assembly* MOV

Fonte: Imagem do autor, adaptado de INTEL, 1999b, p. 442

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, 2005; INTEL, 1999b)

A sintaxe de ambas instruções é similar, conforme demonstrado nas figuras 2.8 e 2.9:

```
PUSH Operando
```

Figura 2.8 – Sintaxe da instrução *Assembly* PUSH

Fonte: Imagem do autor, adaptado de INTEL, 1999b, p. 621

```
POP  Operando
```

Figura 2.9 – Sintaxe da instrução *Assembly* POP

Fonte: Imagem do autor, adaptado de INTEL, 1999b, p. 571

2.2.6.3 Operações aritméticas

Eilam (2005) 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`.
`IMUL Operando`	Instrução que multiplica o operando (com sinal) pelo valor do registrador `EAX`, armazenando o seu resultado (64 bits) em `EDX:EAX`.

Instrução	Descrição
`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`.

Fonte: Adaptado de Eilam, 2005, p. 50

2.2.6.4 Comparação de dados

A comparação de operandos se dá pela utilização da instrução `CMP`, que os compara 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, 2005; INTEL, 1999b)

```
CMP Operando1, Operando2
```

Figura 2.10 – Sintaxe da instrução *Assembly* CMP

Fonte: Imagem do autor, adaptado de INTEL, 1999b, p. 116

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) na *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, 2005; INTEL, 1999b)

```
Jcc Novo_Endereço_de_Código
```

Figura 2.11 – Sintaxe das instruções de salto em *Assembly*

Fonte: Imagem do autor, adaptado de INTEL, 1999b, p. 369

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, 2005; INTEL, 1999b)

```
CALL Endereço_da_Função
```

Figura 2.12 – Sintaxe da instrução *Assembly* CALL

Fonte: Imagem do autor, adaptado de INTEL, 1999b, p. 93

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 seu 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 era a mesma que desenvolveu e codificou 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 2.13:

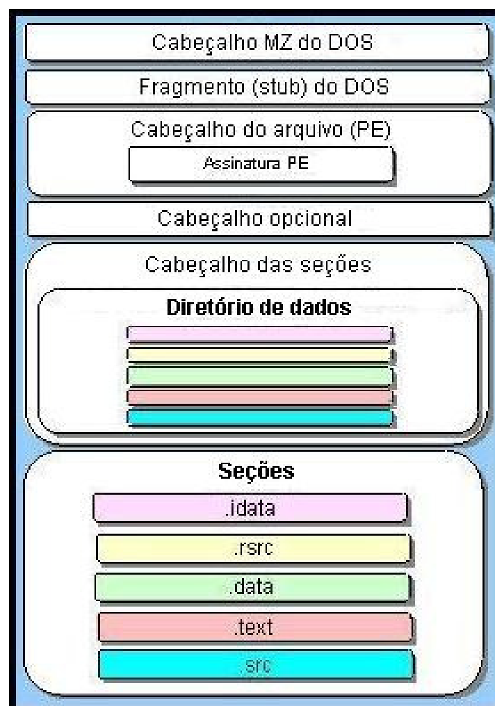


Figura 2.13 – Estrutura do formato *PE*

Fonte: Dummer, 2005, p. 2

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 os 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* têm 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), derivados 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*.

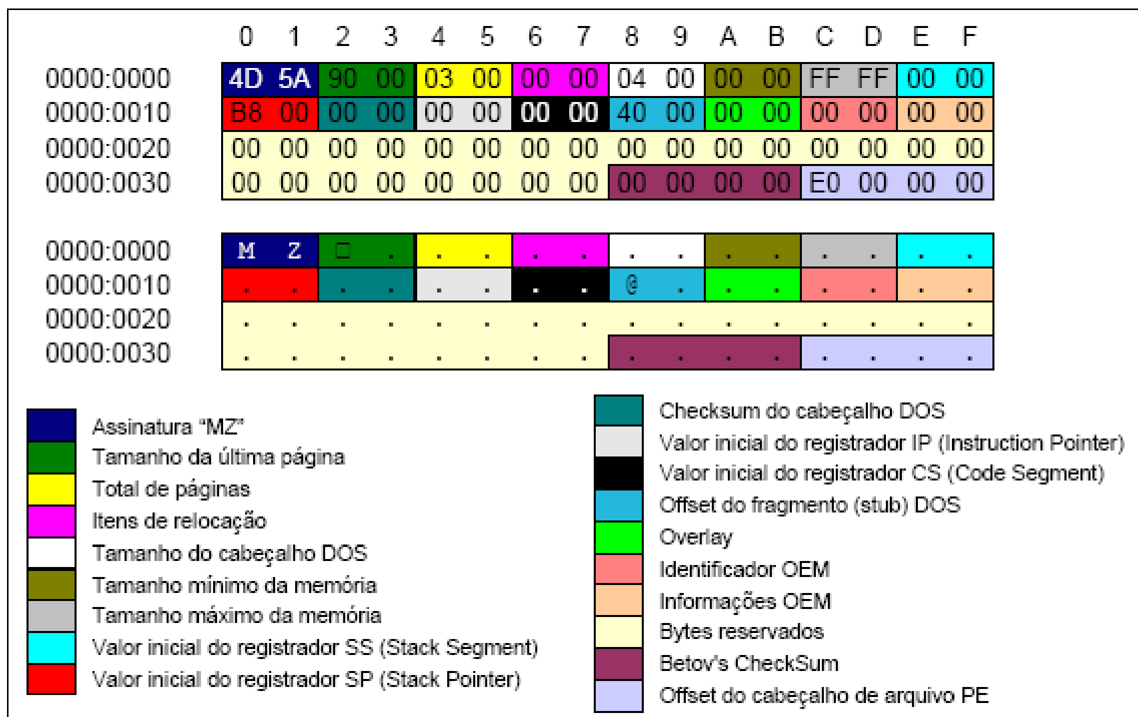


Figura 2.14 – Cabeçalho DOS

Fonte: Dummer, 2005, p. 4

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 um 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: o 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 irá 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 irá 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.

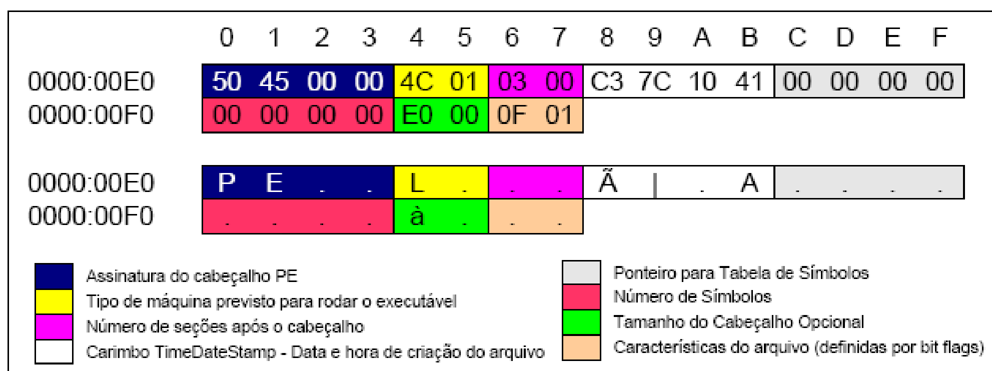


Figura 2.15 – Cabeçalho do arquivo

Fonte: Dummer, 2005, p. 5

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 devido 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 o 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 feito o 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*)

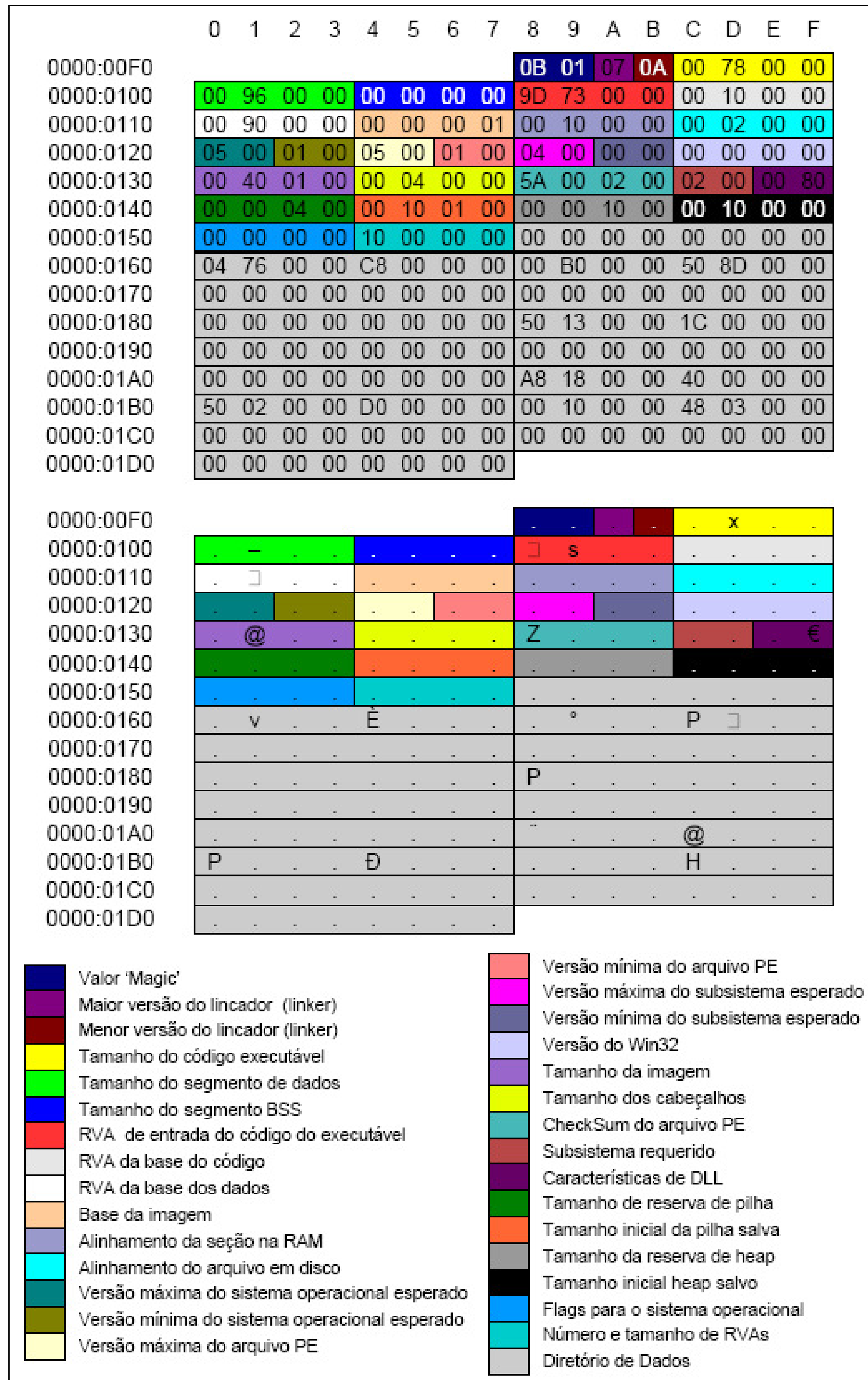
27) Tamanho da reserva de *heap* (4 *bytes*);

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.



	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0000:00F0									0B	01	07	0A	00	78	00	00
0000:0100	00	96	00	00	00	00	00	00	9D	73	00	00	00	10	00	00
0000:0110	00	90	00	00	00	00	00	01	00	10	00	00	00	02	00	00
0000:0120	05	00	01	00	05	00	01	00	04	00	00	00	00	00	00	00
0000:0130	00	40	01	00	00	04	00	00	5A	00	02	00	02	00	00	80
0000:0140	00	00	04	00	00	10	01	00	00	00	10	00	00	10	00	00
0000:0150	00	00	00	00	10	00	00	00	00	00	00	00	00	00	00	00
0000:0160	04	76	00	00	C8	00	00	00	00	B0	00	00	50	8D	00	00
0000:0170	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0000:0180	00	00	00	00	00	00	00	00	50	13	00	00	1C	00	00	00
0000:0190	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0000:01A0	00	00	00	00	00	00	00	00	A8	18	00	00	40	00	00	00
0000:01B0	50	02	00	00	D0	00	00	00	00	10	00	00	48	03	00	00
0000:01C0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0000:01D0	00	00	00	00	00	00	00	00								

Figura 2.16 – Cabeçalho opcional

Fonte: Dummer, 2005, p. 7

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 salvos os dados do programa propriamente dito.

2.3.3 Considerações Gerais

Um bom conhecimento e uma boa compreensão de como é e como funciona o formato *PE* leva a um bom conhecimento e uma boa 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. (DUMMER, 2005)

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 e à 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 no 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 alguns 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, encontrado 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>.

O funcionamento de um depurador dá-se, basicamente, de 2 maneiras: por *software* e por *hardware*.

Na depuração por *software*, quando o usuário define um ponto de interrupção em uma instrução, o depurador substitui esta instrução por uma chamada à instrução de

interrupção `int 3h`, que é uma instrução especial que retorna ao depurador informando que o tal ponto de interrupção foi alcançado. Assim que alcançado, o depurador substitui a instrução de interrupção de volta à instrução original e congela a execução do mesmo, permitindo a análise completa do seu estado na memória. (EILAM, 2005; KASPERSKY, 2003)

Já na depuração por *hardware*, é utilizado um recurso dos processadores *IA32* denominado *Single-Stepping*. Para tanto, basta configurar o *bit* 9 da *EFLAGS* – o *Trap Flag*, fazendo assim com que o processador chame a interrupção `int 1h` automaticamente após o processamento de cada instrução, transferindo o controle de execução ao depurador. Com isto, o depurador poderá rastrear e analisar dados e estados lógicos (*flags*), examinar registradores e emular instruções, tudo de acordo com a necessidade do usuário. (EILAM, 2005; KASPERSKY, 2003)

3.2 Desassembladores *(Disassemblers)*

Um desassemblador 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 deste código de máquina para *Assembly*, assim como as instruções de máquina, são dependentes de plataforma, ou seja, para cada modelo de processador existe um desassemblador.

Os algoritmos de conversão utilizados pelos desassembladores estão, basicamente, divididos em 2 categorias: lineares e recursivos.

A conversão linear (*linear sweep*) é a trivial e seqüencial conversão dos *bytes* de um bloco em instruções de máquina. Já a conversão recursiva (*recursive traversal*) é muito mais ‘inteligente’, pois analisa as instruções de acordo com o fluxo das mesmas, detectando e descartando blocos inválidos de dados, sendo este o algoritmo mais eficaz em programas protegidos por técnicas anti-desassemblagem. (EILAM, 2005)

O desassemblador mais conhecido para processadores *IA-32* é o *IDA Pro* (*shareware*), encontrado no endereço <http://www.datarescue.com>. Para *.NET* existe um desassemblador 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 desassembladores, porém, leva o processo um passo a 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 para os casos em é necessário o monitoramento de acesso a arquivos e registro do sistema, protocolos e portas de rede, visualização de processos, e quaisquer outros 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, proteções que compactam e criptografam 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 reversor.

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*, e para incorporação de novos recursos a aplicativos prontos, já compilados.

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.

4 ATAQUES COMUNS

Comumente, aplicativos comerciais possuem métodos de validação de registro, averiguando se a cópia em execução provém de fonte legal. Cálculos de senhas, chaves de registro e validações online são alguns dos métodos utilizados para verificação de originalidade de um produto.

Qualquer tipo de ataque que vise alterar, burlar ou eliminar alguns destes métodos de validação, ou que caracterize um ataque à integridade de um *software*, é denominado de *cracking*. No entanto, o estudo do que são e como se constituem os ataques à *software* não serve unicamente para o aprendizado do *cracking*.

O entendimento avançado sobre o funcionamento destas técnicas ajudará, com certeza, programadores a anteverem riscos e avaliar possíveis vulnerabilidades já existentes, no entanto, ainda desconhecidas. Conforme Eilam (2005), criar métodos de proteção de *software*, sem sequer conhecer as vias utilizadas pelos invasores (*crackers*), é totalmente ineficaz.

A variedade dos modos e das formas de ataques é incomensurável, até porque, normalmente, cada tipo de ataque explora apenas uma específica vulnerabilidade. Devido à esta extensa diversidade e também à especificidade de muitas destas técnicas, este capítulo abordará apenas os tipos de ataques das vulnerabilidades mais freqüentes, com o objetivo de esclarecimento e aprendizado, visando a qualificação do *software* e maior proteção anti-reversão a futuros projetos.

4.1 Buffer Overflow ou estouro de *buffer*

Conforme mencionado anteriormente, estima-se que mais da metade das vulnerabilidades atuais provêm de estouros de *buffer*. (McGRAW, 2006) O *buffer overflow* é,

atualmente, o principal método utilizado de exploração *software* e tende a permanecer como tal por vários anos futuros, definem Hoglound e McGraw (2006).

Devido à capacidade de gerenciamento de memória desatualizada de linguagens como o C e o C++, este problema não está unicamente ligado à erros de implementação. A própria linguagem é parte responsável deste problema, fazendo com que os *buffers overflows* acabem tornando-se mais comuns do que deveriam ser. (HOGLOUND e McGRAW, 2006)

Antes de tratar, discutir e diagnosticar este problema cabe ratificar alguns conceitos. Um *buffer* nada mais é que um espaço ou bloco contíguo de memória utilizado para armazenamento de dados dentro de um sistema, ou seja, qualquer variável utilizada num sistema pode, de certo ponto de vista, ser considerado um *buffer*. Um vetor de caracteres seria um bom – e pertinente – exemplo.

O estouro de um *buffer* se dá por uma atribuição de dados que exceda o seu tamanho. Um bom exemplo seria a atribuição do texto ‘EMPRESA’ em uma variável de tamanho 5. Como o tamanho do texto excede o tamanho do texto (*buffer*), ocorre o *buffer overflow*.

Outro importante conceito a ser abordado é o da pilha (*stack*), pois esta é item chave para compreensão e exploração da técnica. No entanto, este conceito já está explicado no item 2.2.4 deste trabalho.

4.1.1 Organização da memória de um processo

Abordada anteriormente no capítulo 2.2.1, a compreensão de como ocorre o gerenciamento, execução e estrutura interna da memória é essencial para o entendimento desta técnica de exploração. Conforme explica One (1996), a memória de um processo dispõe-se em 3 regiões, de acordo com a figura 4.1:

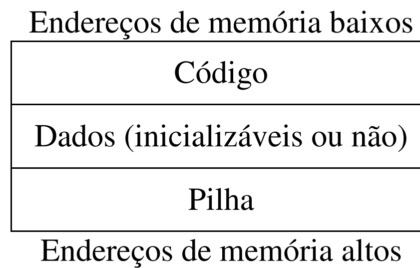


Figura 4.1 – Regiões da memória de um processo

Fonte: Imagem do autor, adaptado de One (1996), sn

Conforme as próprias denominações sugerem, a região de código armazena os dados estáticos do programa, como, por exemplo, as instruções do programa. Igualmente, a região de dados abriga os dados inicializados e não inicializados do programa, incluindo variáveis estáticas, entre outros.

A pilha, no entanto, é um bloco de memória utilizado principalmente para passagem de valores à funções, dentre diversos outros recursos. Para controle, existe um registrador chamado `ESP`, responsável por guardar o ponteiro do topo da pilha. Ambas as definições anteriores foram abordadas previamente neste trabalho, no entanto, para demonstração desta técnica, faz-se necessário um estudo um pouco mais aprofundado.

One (1996) lembra que, além da *CPU* atualizar automaticamente o ponteiro `ESP` nas instruções `PUSH` (inserção na pilha) ou `POP` (extração da pilha), a pilha tem outras atribuições importantes. Uma pilha é constituída de uma série de quadros lógicos (*frames*), que são empilhados de tal maneira que possam ser repassados à outras funções. Cada um destes *frames* contém suas variáveis locais juntamente com os dados necessários para retorno da chamada, incluindo o ponteiro de retorno.

A princípio, estas variáveis locais podem ser referenciadas por seus respectivos *offsets* dentro da pilha, no entanto, se houverem alterações na pilha, tanto por inclusão quanto por exclusão de valores, estes *offsets*, conseqüentemente, serão mudados. Assim, para evitar confusões entre o ponteiro da pilha e do *frame*, a maioria dos compiladores utiliza um registrador secundário (`EBP`) para salvar o ponteiro do *frame*.

Desta maneira, sempre que ocorrer uma instrução de chamada de função, a primeira instrução realizada é a cópia do ponteiro do *frame* anterior (`SPF`) para pilha – permitindo assim a sua restauração no final do procedimento. Feito isto, é copiado o ponteiro da pilha

para o ponteiro do *frame*, informando – e, virtualmente, criando – assim um novo *frame* na memória. Esta operação é chamada de prólogo. Finalizada a execução desta chamada, a pilha deve ser limpa, operação chamada de epílogo. (ONE, 1996)

A figura 4.2, disponibilizada por One (1996), demonstra este funcionamento:

```
void function(int a, int b, int c) {

   char buffer1[5];
   char buffer2[10];

}

void main() {

  function(1,2,3);

}
```

Figura 4.2 – Exemplo ilustrativo para análise do prólogo e epílogo

Fonte: One (1996), sn

Desassemblando o programa acima, é obtido o seguinte código:

```
        push $3
        push $2
        push $1
        call function
```

Figura 4.3 – Código desassemblado do exemplo de prólogo e epílogo

Fonte: One (1996), sn

Analisando o exemplo da figura 4.3, nota-se o carregamento, de forma inversa, dos 3 parâmetros da função seguidos da chamada da função *function*. Cada vez que uma chamada `CALL` é realizada, a CPU grava, intrínseca e automaticamente, o ponteiro `EIP` (ponteiro que indica a posição atual de execução) na pilha. Este endereço, que é este responsável pela indicação de endereço de retorno, é utilizado pela função `RET`, que é a instrução de encerramento de chamada.

Logo após estes passos, acontece o processo de prólogo, conforme demonstrado na figura 4.4:

```
        push %ebp
        mov  %esp,%ebp
        sub  $20,%esp
```

Figura 4.4 – Código exemplo de prólogo

Fonte: One (1996), sn

Neste exemplo, pôde-se observar o processo de prólogo e a criação de um novo *frame*, perceptível pela instrução que subtrai `20h` do registrador `ESP`, valor este que representa a soma dos tamanhos *buffers*.

A figura 4.5 demonstra o estado interno da memória após a chamada da função *function* e execução do prólogo:

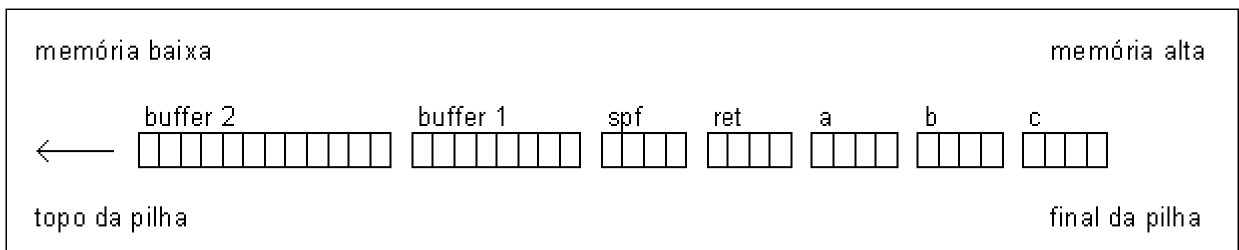


Figura 4.5 – Estado da memória da função *function* após processo de prólogo

Fonte: One (1996), sn

4.1.2 Estourando o *buffer*

O *buffer overflow* ocorre quando são armazenados, em uma variável, mais dados do que a mesma pode suportar. Este erro, por si só, pode comprometer todo um sistema, do mais simples ao mais complexo. A figura 4.6 demonstra um exemplo de estouro de *buffer*:

```
void function(char *str) {

   char buffer[16];

   strcpy(buffer,str);

}

void main() {

  char large_string[256];
  int i;

  for( i = 0; i < 255; i++)
        large_string[i] = 'A';
```

```
  function(large_string);
}
```

Figura 4.6 – Exemplo de um *buffer overflow*

Fonte: One (1996), sn

Este programa demonstra um erro clássico de *buffer overflow*. Inicialmente, é alocado um vetor (*large_string*) de 256 caracteres, sendo estes preenchidos pelo caractere 'A'. Após, este vetor é passado como parâmetro para a função *function*, que lerá este vetor e transferirá seu conteúdo para um vetor de 16 caracteres (*buffer*).

O erro ocorre justamente na função de cópia *strcpy*(). Esta função copia um *buffer* de texto para outro *buffer*, ajustando e controlando o número de elementos a serem copiados de acordo com o tamanho do primeiro *buffer*. No entanto, se o segundo *buffer* for menor que o primeiro, este será estourado, causando o erro de violação de segmento. A figura 4.7 exemplifica o erro:

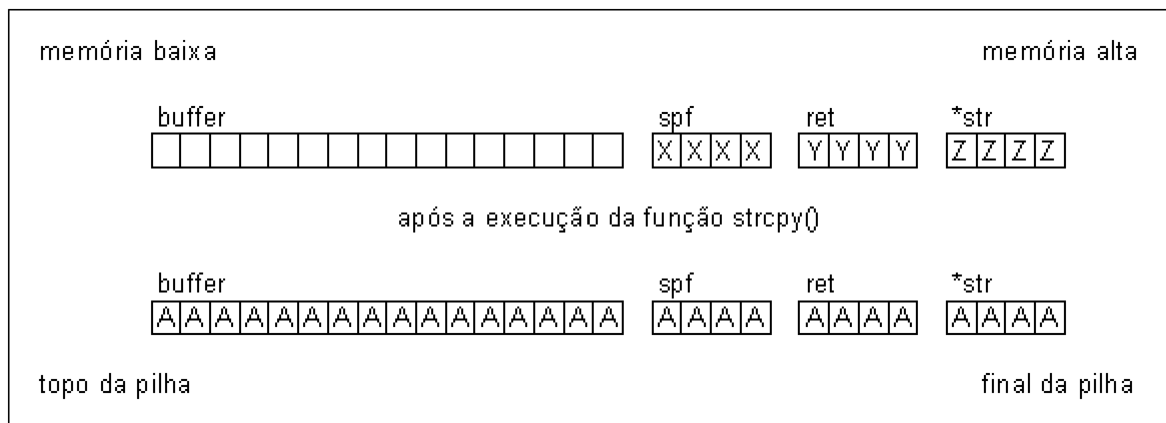


Figura 4.7 – Estado da memória após um estouro de *buffer*

Fonte: Imagem do autor

O exemplo demonstra claramente que, além da cópia dos, teoricamente corretos, 16 caracteres iniciais, houve também a sobreposição dos 250 *bytes* seguintes da memória, sendo alguns deles o ponteiro salvo do *frame* anterior e o (importantíssimo) ponteiro de retorno.

A sobreposição deste ponteiro de retorno pelos caracteres 'A' alterou o endereço de retorno para `0x41414141`, visto que o caractere 'A' é representado pelo valor hexadecimal `41`, comprometendo todo funcionamento do sistema, causando, neste caso em específico, a finalização da aplicação. No entanto, este é o menor dos problemas.

Apenas este exemplo justificou toda importância dada ao *buffer overflow* e demonstrou o quão poderosa pode ser sua exploração. A possibilidade de injeção de código e alteração do endereço de retorno via exploração de um estouro de *buffers* – de modo que faça com que o ponteiro de retorno aponte para o código injetado, executando-o –, permite ao invasor a execução de qualquer código arbitrário na máquina, pondo em risco total um sistema ou até o ambiente inteiro de trabalho.

Como solução deste tipo de problema, sugere-se a não utilização de funções como *strcpy*(), que não possuem verificadores de faixa. Como estas funções não possuem parâmetros que informem a quantidade de *bytes* a serem copiados ou trabalhados, estas funções procuram e se executam até encontrarem o caractere *NULL* (zero), responsável pela identificação de término do texto. Do ponto de vista da segurança do software, isto é péssimo, pois, desta maneira, é impossível validar o *buffer* destino, deixando brecha e favorecendo o estouro.

Ao invés de se utilizar estas funções, deve-se utilizar outras como a *strncpy*(), que exige, como parâmetro, o número de caracteres a serem copiados. Isto limitará o número de *bytes* copiados para o *buffer* de destino, protegendo este do estouro.

4.2 Entradas maliciosas

A principal razão da existência deste método de ataque a *software* é a suposição, por parte dos desenvolvedores, de que os usuários de seus aplicativos nunca serão hostis. De fato, esta afirmação não é verdadeira, principalmente se este *software* aceitar entradas diretamente da Internet, alertam Hoglund e McGraw (2006). Um usuário, por exemplo, pode afetar profundamente o estado de um programa em execução elaborando cuidadosamente uma entrada.

Hoglund e McGraw (2006) destacam ainda outro equívoco comum no desenvolvimento de aplicativos, no qual desenvolvedores pressupõe que se a interface do programa não permitir uma determinada entrada, esta não ocorrerá. Este é outro erro, pois invasores não têm nenhuma necessidade de utilizar um código especifico de cliente para gerar entrada a um servidor ou aplicação.

Há tantas vulnerabilidades justamente porque a entrada do usuário é explicitamente confiada. Se meticulosamente elaborada, uma entrada pode permitir ao invasor abrir arquivos arbitrários, efetue consultas a bancos de dados e até mesmo desligue sistemas ou partes dele.

Como exemplo, Scambray, McClure e Kurtz (2001) destacam as vulnerabilidades do servidor *IIS* 3.0 (*Internet Information Services*) da Microsoft ao executarem certos códigos ASP (*Active Server Pages*).

Um dos ataques constituía-se basicamente na substituição do ponto do arquivo *asp* por sua representação hexadecimal (`2Eh`), fazendo com que o arquivo *asp* fosse baixado ao invés de executado no servidor. O exemplo da figura 4.8 apresenta a exploração desta vulnerabilidade:

```
http://192.168.51.101/code/exemplo%2easp
```

Figura 4.8 – Exploração da vulnerabilidade de erro de ponto num servidor *IIS*

Fonte: Scambray, McClure e Kurtz (2001, p. 579)

Scambray, McClure e Kurtz (2001) relacionam ainda outra vulnerabilidade que afetava os servidores *IIS* 4.0. A vulnerabilidade a seguir não é um erro do servidor por si só, mas sim um exemplo de programação ruim aliada a uma brecha.

```
http://192.168.51.101/code/showpage.asp?source=/../../../../../boot.ini
```

Figura 4.9 – Exploração da vulnerabilidade por caminhos relativos num servidor *IIS*

Fonte: Scambray, McClure e Kurtz (2001, p. 580)

Este tipo de ataque consiste em encontrar algum arquivo *asp* que forneça acesso a arquivos via parâmetros – muito comuns ainda hoje em dia –, liberando assim acesso irrestrito à qualquer arquivo do servidor. Esta exploração se dá pela utilização da seqüência '..' no caminho do arquivo, permitindo a navegação para diretório adjacentes ao qual está se executando o servidor.

Para evitar este tipo de exploração, é sugerida a validação de todos e quaisquer dados de entrada utilizados num *software*. Em programas servidores, deve-se fazer com que estes dados sejam criticados pelo próprio servidor, ao invés de delegar esta função aos aplicativos clientes, ajustados com uma configuração de segurança adequada.

4.3 *Patching*

O *patching* não é especificamente uma técnica nem um tipo de ataque direcionado à *software*. Aplicar um *patch* em um arquivo nada mais é que editá-lo em modo binário, alterando assim dados e instruções do sistema, de modo a corrigir, alterar ou desativar o conteúdo e/ou o fluxo de execução deste. (HOGLOUND e MCGRAW, 2006)

No entanto, apesar de não pertencer ao grupo de técnicas de exploração, este processo é um dos principais, se não o principal, artifício utilizado pelos *crackers*, pois, através deste processo, é que se realizam efetivamente os ajustes necessários em um *software* já compilado.

Os *patches* podem ser utilizados para inserir instruções de desvio, novos blocos de código e até para sobrescrever dados estáticos. A aplicação de patches é uma das formas pelas quais os piratas de *software* quebram mecanismos de direitos autorais digitais.

Para melhor compreensão, será utilizado um programa exemplo chamado *Splish.exe*, que era utilizado para certificação e permissão de ingresso na extinta Universidade (virtual) de Engenharia Reversa (*Reverse Engineering Academy - REA*).

Este programa-teste era disponibilizado livremente na *Internet*, e os candidatos tinham um mês para quebrar as 3 rotinas embutidas neste *software*: remoção da tela de propaganda na abertura do aplicativo, quebra de uma senha interna fixa e quebra de uma senha cujo valor era calculado de acordo com o nome do candidato. Para exemplificação de um *patching*, será utilizado este programa-teste e será demonstrada a remoção da tela de abertura.

Este tipo de procedimento tende a ser de simples localização, devido tanto à própria organização interna de um *software* – que costumeiramente separa este tipo de rotina do fluxo principal de execução do programa, separando e chamando-o através de uma instrução `CALL` –, quanto à possibilidade de localização através de uma depuração *Single-Step*, que é uma depuração linha a linha do programa, possibilitando a observação passo a passo do *software*.

Depurando o bloco de código inicial, é detectável a abertura da tela de propaganda no *offset* `00401079`. Ainda é possível notar que as instruções que a precedem são meramente instruções de inicialização do aplicativo.

```
00401000 >/$ 6A 00          PUSH 0
00401002  |. E8 83070000    CALL <&KERNEL32.GetModuleHandleA>

00401007  |. A3 80344000    MOV DWORD PTR DS:[403480],EAX
0040100C  |. E8 73070000    CALL <&KERNEL32.GetCommandLineA>

00401011  |. 6A 0A          PUSH 0A                          ; Arg4 = 0000000A
00401013  |. FF35 84344000  PUSH DWORD PTR DS:[403484]       ; Arg3 = 00000000
00401019  |. 6A 00          PUSH 0                           ; Arg2 = 00000000
0040101B  |. FF35 80344000  PUSH DWORD PTR DS:[403480]       ; Arg1 = 00400000
00401021  |. E8 06000000    CALL 0040102C

00401026  |. 50             PUSH EAX                         ; /ExitCode
00401027  \. E8 52070000    CALL <&KERNEL32.ExitProcess> ; \ExitProcess

0040102C  /$ 55             PUSH EBP
0040102D  |. 8BEC           MOV EBP,ESP
0040102F  |. 83C4 B0        ADD ESP,-50
00401032  |. C705 6F344000 >MOV DWORD PTR DS:[40346F],15E
0040103C  |. C705 73344000 >MOV DWORD PTR DS:[403473],0C8

00401046  |. 6A 00          PUSH 0                       ; /Index = SM_CXSCREEN
00401048  |. E8 D7060000    CALL <&USER32.GetSystemMetrics>

0040104D  |. 50             PUSH EAX
0040104E  |. FF35 6F344000  PUSH DWORD PTR DS:[40346F]
00401054  |. E8 08040000    CALL 00401461

00401059  |. A3 77344000    MOV DWORD PTR DS:[403477],EAX
0040105E  |. 6A 01          PUSH 1                       ; /Index = SM_CYSCREEN
00401060  |. E8 BF060000    CALL <&USER32.GetSystemMetrics>

00401065  |. 50             PUSH EAX
00401066  |. FF35 73344000  PUSH DWORD PTR DS:[403473]
0040106C  |. E8 F0030000    CALL 00401461

00401071  |. A3 7B344000    MOV DWORD PTR DS:[40347B],EAX
00401076  |. FF75 08        PUSH DWORD PTR SS:[EBP+8]       ; /Arg1
00401079  |. E8 F9030000    CALL 00401477
```

Figura 4.10 – Código desassemblado de inicialização do programa Splish.exe

Fonte: Imagem do autor

Percebe-se que neste *offset* encontra-se a chamada para a tal rotina de propaganda, localizada no *offset* `00401477`. A figura 4.11 demonstra o código da rotina referenciada:

```
00401477  /$ 55             PUSH EBP
00401478  |. 8BEC           MOV EBP,ESP

...

;Após toda uma seqüência de instruções de inicialização de tela,
;encontra-se claramente uma rotina de contagem de tempo,
;o que pode ser a grande responsável pela contagem de tempo de exibição da
;tela de propaganda:
```

```
...

00401556  |. E8 35020000    CALL <&KERNEL32.GetTickCount>
                           ;esta função que da API do Windows retorna o
                           ;tempo que o Windows está ligado, normalmente
                           ;utilizada para cronometragem de algum proceso

0040155B  |. 8945 CC        MOV DWORD PTR SS:[EBP-34],EAX
                           ;move o valor lido para a posicao EBP - 34

0040155E  |. 8145 CC D00700>ADD DWORD PTR SS:[EBP-34],7D0
                           ;acresce 7D0h (2000 em numeral) ao valor lido,
                           ;indicando o horário de termino da exibição

                           ;assim, nota-se nas instruções abaixo o loop
                           ;responsável pela verificação do tempo,
                           ;mantendo a exibição enquanto TickCount for
                           ;menor que o tempo inicial - 2000.

00401565  |> E8 26020000    CALL <&KERNEL32.GetTickCount> ;inicio loop

0040156A  |. 3945 CC        CMP DWORD PTR SS:[EBP-34],EAX ;compara valores

0040156D  |. 76 02          JBE SHORT 00401571            ;se excedido,
                                                          ;efetua salto
                                                          ;finalizando

0040156F  |.^EB F4          JMP SHORT 00401565            ;se não, loop


                           ;prepara e envia Message de fechamento da tela
00401571  |> 6A 00          PUSH 0                ; lParam = 0
00401573  |. 68 60F00000    PUSH 0F060            ; wParam = F060
00401578  |. 68 12010000    PUSH 112              ; Message = WM_SYSCOMMAND
0040157D  |. FF35 11324000  PUSH DWORD PTR DS:[403211] ; hWnd = 270122
00401583  |. E8 D2010000    CALL <JMP.&USER32.SendMessageA>

00401588  |. C9             LEAVE
00401589  \. C2 0400        RETN 4  ; retorna ao endereço de chamada
```

Figura 4.11 – Instruções da função de exibição da propaganda do aplicativo Splish.exe

Fonte: Imagem do autor

De acordo com a explicação da rotina, nota-se claramente que o aplicativo registra a hora em que o computador inicia a exibição da tela (instrução do endereço `00401556`), acrescendo a este `7D0h` ou, mais legivelmente, 2000 milisegundos, determinado assim o horário de término de exibição da referida tela. Em seguida, a rotina é finalizada com um *loop* que será realizado enquanto o horário atual do computador for menor ao valor computado.

De posse destes dados, a remoção torna-se simples e duplamente solucionável. Tanto a desabilitação da instrução de chamada no *offset* `00401079` quanto a alteração de tempo

referência de 2000 para 0, da instrução do endereço `0040155E`, podem ser considedas como soluções do problema. Para facilitação na demonstração, será aplicado o *patch*, alterando o tempo referência conforme a segunda solução sugerida.

Para tanto, basta, a partir de qualquer editor hexadecimal, editar os *bytes* nos *offset* `00401561` e `00401562` para zerar o valor de tempo de referência, fazendo com que esta tela nem chegue a ser exibida.

4.4 *Keygenning*

Assim como o *patching*, o *keygenning* não é uma técnica que explora alguma vulnerabilidade ou um recurso falho de um aplicativo. Esta técnica consiste na análise do funcionamento de um programa protegido por senha, encontrando e destrinchando todo cálculo para obtenção da chave, e a partir deste conhecimento, produzir programas que calculem as senhas de acordo com as entradas do usuário.

O *keygenning* é o processo de criação de programas que imita o algoritmo de geração das chaves utilizado por aplicativos protegidos, cujos cálculos forneçam um número ilimitado de chaves válidas, define Eilam (2005).

Deste processo, o mais difícil e exaustivo passo é o de localização e entendimento do algoritmo calculador, visto que este é muitas vezes ocultado ou ofuscado dentro do sistema. Como alternativas para localização deste código, normalmente são utilizados *breakpoints* dentro de depurados sobre as chamadas *APIs* de saída como *MessageBoxA* ou *MessageBoxW,* ou sobre funções que extraem os valores dos campos *edit* como *GetDlgItemTextA* ou *GetDlgItemTextW*. Quando alcançados estes *breakpoints*, é verificado o *call stack* (lista das últimas instruções executadas) ou é feita uma depuração *single-step* a partir do endereço focado.

Considerando novamente o programa-exemplo anterior e procedendo a busca da maneira sugerida, pode-se encontrar o bloco de cálculo de chave. Analisando as instruções, decifra-se o código:

```
0040161B  |. 33C9          XOR ECX,ECX  ;zera registrador ECX
0040161D  |. 33DB          XOR EBX,EBX  ;zera registrador EBX
0040161F  |. 33D2          XOR EDX,EDX  ;zera registrador EDX

00401621  |. 8D35 36324000  LEA ESI,DWORD PTR DS:[403236]
```

```
                        ;carrega em ESI o ponteiro para o nome do
                        ;usuário

00401627  |. 8D3D 58324000  LEA EDI,DWORD PTR DS:[403258]
                        ;carrega em EDI o ponteiro do buffer utilizado
                        ;para salvamento de alguns bytes necessários
                        ;para o cálculo

0040162D  |. B9 0A000000    MOV ECX,0A
                        ;move 0A = 10 para o registrador ECX

                        ;inicio do loop

00401632  |> 0FBE041E       /MOVSX EAX,BYTE PTR DS:[ESI+EBX]
                        ;como EBX é inicializado em zero e incrementado
                        ;a cada loop do for, percebe-se que esta função
                        ;copia o caractere da posição EBX do nome do
                        ;usuário

00401636  |. 99             |CDQ   ;converte o valor pra quad

00401637  |. F7F9           |IDIV ECX
                        ;divide o valor ordinal do caractere em EAX por
                        ;ECX (ECX = 10), jogando o quociente em EAX e o
                        ;resto em EDX

00401639  |. 33D3           |XOR EDX,EBX
                        ;efetua um XOR do resto (EDX) sobre o contador
                        ;crescente do loop (EBX), salvando o
                        ;resultado em EDX

0040163B  |. 83C2 02        |ADD EDX,2
                        ;adiciona +2 ao registrador EDX

0040163E  |. 80FA 0A        |CMP DL,0A
                        ;verifica se esse o valor em DL (low byte de
                        ;EDX) é maior que 10

00401641  |. 7C 03          |JL SHORT 00401646
                        ;se não for, pula pra instrução 00401646

00401643  |. 80EA 0A        |SUB DL,0A
                        ;se for, subtrai -10 de EDX

00401646  |> 88141F         |MOV BYTE PTR DS:[EDI+EBX],DL
                        ;move o valor calculado para um buffer para
                        ;posterior verificação

00401649  |. 43             |INC EBX
                        ;incrementa o contador

0040164A  |. 3B1D 63344000  |CMP EBX,DWORD PTR DS:[403463]
                        ;anteriormente a este código, o programa moveu
                        ;para a posição de memória 403463 o tamanho da
                        ;string do nome do usuário
                        ;aqui então é verificada se o loop já percorreu
                        ;todos os caracteres

00401650  |.^75 E0          \JNZ SHORT 00401632
                        ;se não, loop
```

Figura 4.12 – Rotina de cálculo de senhas sobre o nome do usuário para um buffer interno

Fonte: Imagem do autor

Nesta primeira parte de código, percebe-se que foram percorridos os 10 primeiros caracteres do nome de usuário fornecido, dividindo seu valor ordinal por 10. O resto desta divisão é pego e é aplicado a operação XOR (ou exclusivamente) ao valor do contador menos 1. Após é incrementado 2 ao valor do *buffer* e, caso maior de 10, é subtraído pelo mesmo. Para mais fácil compreensão, o código em alto nível (*Delphi*) abaixo auxilia no entendimento:

```
X := Length(Nome);
If X > 10 then X := 10;

For I:=1 to X do
begin
  Buffer[I] := (Ord(Nome[I]) mod 10) xor (I-1);

  Inc(Buffer[I], 2);
  If (Buffer[I] >= 10) then Dec(Buffer[I], 10);
end;
```

Figura 4.13 – Rotina de cálculo transcrita para uma linguagem alto nível (*Delphi*)

Fonte: Imagem do autor

A criação deste *buffer* de valores será utilizado para conferência e validação da chave. É importante guardar e lembrar destes dados, pois o algoritmo calculará a seguir outro *buffer*, desta vez para a chave fornecida, que será utilizado para a validação. Para a chave ser válida, o número de itens em cada *buffer* deve ser igual, assim como o valor de cada um destes item. A figura 4.14 demonstra como é calculada o *buffer* para a chave fornecida:

```
00401652  |. 33C9           XOR ECX,ECX  ;zera novamente o registrador ECX
00401654  |. 33DB           XOR EBX,EBX  ;zera novamente o registrador EBX
00401656  |. 33D2           XOR EDX,EDX  ;zera novamente o registrador EDX

00401658  |. 8D35 42324000  LEA ESI,DWORD PTR DS:[403242]
                            ;carrega em ESI o ponteiro para a senha
                            ;informada pelo usuário

0040165E  |. 8D3D 4D324000  LEA EDI,DWORD PTR DS:[40324D]
                            ;carrega em EDI o ponteiro de outro buffer
                            ;utilizado para salvamento de outros bytes
                            ;necessários para o cálculo

00401664  |. B9 0A000000    MOV ECX,0A
                            ;move 0A = 10 para o registrador ECX
00401669  |> 0FBE041E       /MOVSX EAX,BYTE PTR DS:[ESI+EBX]
                            ;como EBX é inicializado em zero e incrementado
                            ;a cada loop do for, percebe-se que esta função
                            ;copia o caractere da posição EBX da senha do
                            ;usuário
```

```
0040166D  |. 99            |CDQ   ;converte o valor pra quad

0040166E  |. F7F9          |IDIV ECX
                           ;divide o valor ordinal do caractere em EAX por
                           ;ECX (ECX = 10), jogando o quociente em EAX e o
                           ;resto em EDX

00401670  |. 88141F        |MOV BYTE PTR DS:[EDI+EBX],DL
                           ;move o low byte do resto para o buffer, para
                           ;posterior verificação

00401673  |. 43            |INC EBX
                           ;incrementa o contador

00401674  |. 3B1D 67344000 |CMP EBX,DWORD PTR DS:[403467]
                           ; verificada se o loop já percorreu todos os
                           ;caracteres

0040167A  |.^75 ED         \JNZ SHORT 00401669
                           ;se não, loop

0040167C  |. EB 2A         JMP SHORT 004016A8
                           ;salta para instrução 004016A8
```

Figura 4.14 – Rotina de cálculo de senhas da senha informada para outro buffer interno

Fonte: Imagem do autor

O trecho de código acima demonstra então como é calculado o *buffer* para a chave fornecido. Desta vez, a fórmula é um pouco mais simples. São lidos os caracteres de acordo com o tamanho do nome do usuário, e para cada um deles é aplicada um divisão de seu valor ordinal por 10. O resto é separado e jogado ao *buffer*. Novamente, para melhor compreensão, o código foi transcrito em uma linguagem de alto nível (*Delphi*):

```
X := Length(Usuario);
If X > 10 then X := 10;

For I:=1 to X do Buffer2[I] := Ord(Senha[I]) mod 10;
```

Figura 4.15 – Rotina de cálculo transcrita para uma linguagem alto nível (*Delphi*) (2)

Fonte: Imagem do autor

Consideravelmente mais simples que o primeiro algoritmo, estas poucas instruções geram o segundo *buffer* para validação da senha. Com base nestes *buffers*, o programa segue e compara-os entre si. O bloco de código da figura 4.16 demonstra o funcionamento do sistema:

```
004016A8  |> 8D35 4D324000  LEA ESI,DWORD PTR DS:[40324D]
                            ;carrega no registrador ESI o ponteiro do
                            ;buffer calculado a partir da chave informada
```

```
004016AE  |. 8D3D 58324000  LEA EDI,DWORD PTR DS:[403258]
                           ;carrega no registrador EDI o ponteiro do
                           ;buffer calculado a partir do nome do usuario

004016B4  |. 33DB           XOR EBX,EBX
                           ;zera o registrador EBX, que neste caso será o
                           ;contador do loop


004016B6  |> 3B1D 63344000  /CMP EBX,DWORD PTR DS:[403463]
                           ;verificada se o loop já não percorreu todos
                           ;os caracteres

004016BC  |. 74 0F          |JE SHORT 004016CD
                           ;se já percorreu, significa que todos os
                           ;caracteres estavam OK e que a chave é válida.
                           ;salta para mensagem de sucesso

004016BE  |. 0FBE041F       |MOVSX EAX,BYTE PTR DS:[EDI+EBX]
                           ;copia o caractere da posição EBX do buffer do
                           ;usuário

004016C2  |. 0FBE0C1E       |MOVSX ECX,BYTE PTR DS:[ESI+EBX]
                           ;copia o caractere da posição EBX do buffer da
                           ;chave

004016C6  |. 3BC1           |CMP EAX,ECX
                           ;compara se os itens dos buffers são iguais

004016C8  |. 75 18          |JNZ SHORT 004016E2
                           ;se não forem, pula para a mensagem de erro

004016CA  |. 43             |INC EBX
004016CB  |.^EB E9          \JMP SHORT 004016B6
                           ;se forem, incrementa loop e testa próximo item


                           ;mensagem de sucesso
004016CD  |> 6A 00          PUSH 0            ; /Style = MB_OK|MB_APPLMODAL
004016CF  |. 68 0A304000    PUSH 0040300A     ; |Title = "Splish, Splash"
004016D4  |. 68 42304000    PUSH 00403042     ; |Text  = "Good job, now
                                              ;          keygen it."
004016D9  |. 6A 00          PUSH 0            ; |hOwner = NULL
004016DB  |. E8 68000000    CALL <JMP.&USER32.MessageBoxA>
004016E0  |. EB 13          JMP SHORT 004016F5


                           ;mensagem de erro
004016E2  |> 6A 00          PUSH 0            ; /Style = MB_OK|MB_APPLMODAL
004016E4  |. 68 0A304000    PUSH 0040300A     ; |Title = "Splish, Splash"
004016E9  |. 68 67304000    PUSH 00403067     ; |Text  = "Sorry, please try
                                              ;          again."
004016EE  |. 6A 00          PUSH 0            ; |hOwner = NULL
004016F0  |. E8 53000000    CALL <JMP.&USER32.MessageBoxA>



                           ;finalização da rotina
004016F5  |> C9             LEAVE
004016F6  \. C2 0800        RETN 8
```

Figura 4.16 – Rotina de validação dos *buffers* e conferência das senhas

Fonte: Imagem do autor

Depois desta análise e descoberto o funcionamento do cálculo, basta criar um sistema que calcule, de forma inversa, senhas de modo à possibilitar a descoberta de chaves a quaisquer usuários.

5 PROTEÇÕES E TÉCNICAS ANTI-REVERSÃO

Proteger os dados e todo conhecimento implementado num sistema é de vital importância para o sucesso de um *software*. Deixá-lo desprotegido é como deixar um livro aberto sobre a mesa. Efetivamente, poucos programadores sabem o quão importante é prevenir seus códigos de desassembladores e depuradores. (EILAM, 2005; CERVEN, 2002) "É impossível prevenir completamente um sistema da reversão". (EILAM, 2005, p. 327) (Tradução nossa) No entanto, qualquer artifício de proteção será sempre válido.

As técnicas anti-reversão tentam evitar o acesso a esses dados, bloqueando, rastreando ou, simplesmente, dificultando a sua leitura e compreensão. Mesmo uma simples proteção anti-reversão pode dificultar, em muito, o trabalho de um reversor.

Eilam (2005) destaca que, o processo de estudo e de *cracking* de um sistema pode, muita vezes, ser abortado dependendo do nível na relação complexidade da proteção versus motivação do reversor. Aplicativos simples podem, muitas vezes, ser mais facilmente reescritos do que *crackeados*. Esta relação é que definirá a necessidade e deverá ser analisada pelos criadores de *software*.

Existem diversos contextos dentro das técnicas anti-reversão, cada um com seus prós e contras. Eilam (2005) os classifica, básica e simplificadamente, em 3 grupos:

- Eliminação de informações simbólicas: o primeiro e mais óbvio passo para afastar reversores é eliminar qualquer informação textual disponível no programa compilado, como nomes de funções exportadas pelo *software* ou nomes de seções. Estas informações podem ser extremamente úteis e devem sempre ser renomeadas e substituídas por seqüências de caracteres que desvinculem o nome de seu significado;

- Técnicas de ofuscação de código: estas técnicas se baseiam numa escrita de código que, do ponto de vista funcional, não altera em nada o programa, porém tornam a compreensão e leitura muito mais complexa. Seria reescrever, reestruturar e alterar o fluxo de operações de tal maneira a confundir o reversor;
- Códigos anti-reversão embutidos: esta técnica consiste em inserir códigos na aplicação no intuito de identificar, desabilitar e/ou danificar quaisquer ferramentas ou técnicas de reversão. É essencialmente uma medida de contra-ataque.

Como citado anteriormente, o entendimento da linguagem *Assembly* é muito importante neste processo, já que a grande maioria dos exemplos das técnicas é obtida nesta forma. No entanto, nada impede que sejam escritos mecanismos em linguagens alto nível. O *Assembly* é apenas sugerido, pois nele pode-se definir diretamente as instruções da maneira desejada, permitindo assim criar as armadilhas e obstáculos diretos e que permanecerão explícitos no código final, já que nessa linguagem não há compilação nem otimização de código, apenas lincagem.

Outra importante avaliação a ser feita é considerar que muitas das técnicas são dependentes de plataformas, podendo funcionar apenas em sistemas baseados no *Windows NT* (*NT, 2000, 2003, XP*), outras apenas nos *Windows 9x*. Todavia, existem diversas técnicas livres dessas dependências, permitindo ao programador selecionar a melhor técnica para cada cenário.

Cerven (2002) ressalta que não é necessária a proteção de todo o código de um aplicativo, pois esta tarefa pode ser excessivamente maçante ao programador e desnecessária, já que não será de interesse do reversor. Deve-se, apenas, preocupar com os pontos críticos e vitais do *software*, visto que estes serão os processos-alvo do sistema. Além disto, a maneira de como detectar e responder aos testes devem ser cuidados. Deve-se evitar ao máximo exibir mensagens de erro ou falhas, pois este tipo de operação é extremamente fácil de ser rastreado.

Como simples modelo, Cerven (2002) sugere apenas a utilização de técnicas anti-desassemblagem e alguns testes de verificação de depuradores ativos na memória, sendo uma vez na inicialização do *software* e outra durante a execução. Só estas proteções já seriam suficientes para dar, por menor que seja, uma boa proteção ao *software*.

5.1 Anti-depuradores

Como a grande parte dos processos da engenharia reversa de *software* ocorre dentro ou com auxílio de um depurador, muitas vezes é importante incorporar ao sistema em desenvolvimento, um mecanismo de proteção a esse tipo de ferramenta.

Complicar a depuração criando mecanismos de detecção de *breakpoints*, detectores de depuradores ativos na memória ou mesmo procurando no registro do *Windows* por chaves de instalação, são alguns dos métodos comumente utilizados.

Cerven (2002) ainda sugere a combinação de técnicas anti-depuradores com as técnicas de criptografia de dados, visto que o efeito protetor gerado pela combinação dessas técnicas pode dificultar significativamente o processo de reversão já que, nesta situação, o programa não poderá ser depurado nem descarregado (*core dumped*) da memória do sistema operacional.

No entanto, Cerven (2002) ressalta um fator adverso no uso específico da técnica de anti-depuradores dentre as técnicas anti-reversão: a possibilidade de falso-positivos. Diferentemente das outras técnicas – como a ofuscação ou criptografia de código, que são apenas penalizadas no desempenho do aplicativo –, a implementação de um destes métodos pode acarretar na detecção errônea de um depurador, assim comprometendo a funcionalidade do sistema.

Além disto, como é passível em qualquer outra técnica, alguns desses métodos servirão apenas para certas versões de sistemas operacionais e/ou depuradores específicos. Outros métodos podem tornar-se obsoletos com o passar dos anos, pois se baseiam em versões específicas de depuradores.

Nesse intuito, serão tratadas a seguir algumas das diversas técnicas existentes para proteção anti-depuradores demonstrando, principalmente, aquelas com menor dependência possível de plataforma.

5.1.1 Detecção via *API IsDebuggerPresent*

O *Windows* disponibiliza uma função *API* chamada *IsDebuggerPresent*, que serve para informar se uma aplicação está rodando sob um depurador em modo de usuário. Esta

função acessa, basicamente, o *Process Environment Block* (*PEB*) de um processo e, verificando em seu estado, determina a existência ou não de um depurador. (EILAM, 2005)

No entanto, este método não é muito eficaz na proteção de um sistema, pois como qualquer chamada *API*, os nomes dessas funções estão extremamente expostos e são muito facilmente identificáveis dentro do código binário, facilitando o processo de quebra. Seria unicamente necessária a identificação do ponto de chamada desta *API* e a aplicação de um *patch* sobre esta chamada.

Nestas situações, Eilam (2005) sugere a implementação intrínseca da função, que, neste caso, é código do bloco abaixo – proveniente da extração binária da função *IsDebuggerPresent* da biblioteca Kernel32.dll –, tornando assim dispensável a chamada *API* e, conseqüentemente, eliminando todos os contras recém listados e dificultando seu rastreamento.

```
mov    eax, fs:[00000018]
mov    eax, [eax+0x30]
cmp    byte ptr [eax+0x2], 0
je     ExecutaProg
jmp    TerminaProg
```

Figura 5.1 – Código binário da instrução *API IsDebuggerPresent*

Fonte: Imagem do autor, adaptado de Eilam, 2005, p. 332

O código *Assembly* demonstrado na figura 5.1 acessa o *Thread Environment Block* (*TEB*) do programa em execução, lendo a estrutura *Process Environment Block* (*PEB*) no *offset* +30h. A seguir é lido o *byte* no *offset* +2h, cujo *byte* indica se há ou não a presença do depurador. Caso o valor deste *byte* for 0 (zero), nenhum depurador foi encontrado e assim é rodado o programa. Do contrário, a aplicação é finalizada.

No entanto, uma grande desvantagem da utilização deste código intrínseco é o fato deste assumir, internamente, o *offset* de duas estruturas de dados (*TEB* e *PEB*) do *Windows*. Apesar destas estruturas permanecerem inalteradas desde a versão 4.0 do *Windows NT* (1996) até a versão 2003 do *Windows Server*, graves problemas podem ocorrer caso versões futuras as modifiquem. Uma mudança deste âmbito poderia travar ou derrubar o programa, mesmo que nenhum depurador exista.

5.1.2 Detecção via estado lógico da *Trap Flag*

Como destacado anteriormente, um dos métodos de execução de depuradores dá-se por *hardware*, utilizando-se da configuração da TF (*Trap Flag*) da *EFLAGS*. Esta técnica de detecção consiste em ler da memória do aplicativo os parâmetros da *EFLAGS* e verificar a existência ou não de um depurador ativo.

Quando a *Trap Flag* estiver ativa, o processador executará a instrução `int 1h` após cada instrução do programa. Isto fará com que sejam gerados erros de violação de acesso (*STATUS_ACCESS_VIOLATION*), sendo assim capturados pelos depuradores e silenciados, já que estes são gerados unicamente para rastreamento de execução. (EILAM, 2005)

Compreendido o funcionamento, pode-se utilizar este ataque como contra-ataque configurando, em modo de execução, a *Trap Flag* e verificando se a tal violação de acesso será gerada. Se não for gerada, significa que um depurador está ativo em memória e capturou a exceção antes da própria aplicação em execução.

```
BOOL bTemDebugger = TRUE;

__try
{
       _asm                             // Inicio do bloco Assembly
       {
            pushfd                      // Instrução utilizada para ler os
                                        // valores da EFLAGS para o
                                        // registrador ESP

            or dword ptr [esp], 0x100   // Configura o bit da Trap Flag no
                                        // registrador ESP

            popfd                       // Salva o valor do registro ESP
                                        // no registrador EFLAGS

            nop                         // Gera uma instrução qualquer
                                        // (NOP = No Operation)
                                        // forçando com isso a geração do
                                        // erro de violação
       }
}
__except(EXCEPTION_EXECUTE_HANDLER)     // captura as exceções do bloco
{
       bTemDebugger = FALSE;            // Se a exceção de fato ocorrer,
                                        // não sendo capturada, significa
                                        // que nenhum depurador esta
                                        // ativo ou em execução.
}

if (bTemDebugger == TRUE)
```

```
        printf ("Existe um depurador ativo!\n");
```

Figura 5.2 – Código para detecção de um depurador via *Trap Flag*

Fonte: Imagem do autor, adaptado de Eilam, 2005, p. 335

Devido ao fato desta técnica usufruir do mesmo mecanismo utilizado para habilitar um depurador, aliado ao fato de independer de plataforma, é possível notar uma grande vantagem na utilização desta técnica. Cerven (2002) ressalta ainda que, se bem camuflada dentro do código, esta técnica torna o *software* quase impossível à quebra, devido especialmente à dificuldade que *crackers* têm de combater e emular essas instruções.

Já Eilam (2005) sugere, complementarmente, que sejam evitadas quaisquer mensagens de avisos ou erros após a identificação do depurador devido à facilidade no rastreamento destes códigos, sugerindo, preferencialmente, términos abruptos dos aplicativos.

5.1.3 Detecção via busca da instrução *Int 3h*

Outro método de localizar depuradores ativos na memória é procurando pela presença de pontos de interrupção (*breakpoints*), sob a forma da instrução `int 3h`, na memória do aplicativo. Como explicado anteriormente, quando definido um breakpoint sobre uma instrução, os depuradores de *software* permutam a instrução original pela instrução `int 3h`. Assim, a idéia é simplesmente procurar pela seqüência de caracteres `0CCh`, que é a representação binária da instrução `int 3h`.

Logicamente, esta técnica funcionará apenas para detecção de depuradores que procederem desta forma. Se o depurador não efetuar nenhuma alteração no código no momento da execução, não será possível detectar sua presença.

Cerven (2002) descreve no programa-exemplo abaixo como é feita a localização desta seqüência, procurando todas as funções importadas na tabela IAT e verificando, uma a uma, a presença da seqüência `0CCh` nos primeiros caracteres da função. Entretanto, Cerven alerta também da possibilidade de haverem problemas na utilização desta técnica.

Grande parte das aplicações possui esta seqüência internamente em seu código, tanto como parte integrante de um ponteiro, de um valor constante ou de qualquer outra forma distinta, não necessariamente representando a instrução `int 3h`. Assim, caso o programa em

desenvolvimento já possua intrinsecamente esta seqüência, os ponteiros destas deverão ser previamente salvos em algum lugar dentro do aplicativo, permitindo assim a futura diferenciação destas seqüências de códigos *breakpoints*. (CERVEN, 2002)

```
.386
.MODEL FLAT,STDCALL
locals
jumps
UNICODE=0
include w32.inc

Extrn SetUnhandledExceptionFilter : PROC

.DATA
message1 db "Detecção de breakpoint pela procura Int 3h",0
message3 db "Breakpoint encontrado",0
message2 db "Breakpoint não encontrado",0

delayESP dd 0           ;salva o registrador ESP aqui
previous dd 0           ;o registrador ESP salvará o endereço antigo do
                        ;serviço SEH aqui

current_import dd 0     ;aqui será salvo o endereço da função que estará
                        ;sendo testada

.code

Start:

;----------------------------------------------------------------------------
;Configura o SEH - em caso de erro
;----------------------------------------------------------------------------
       mov  [delayESP], esp
       push offset error
       call SetUnhandledExceptionFilter
       mov  [previous], eax
;----------------------------------------------------------------------------

       lea edi, Start        ;carrega no registrador EDI o endereço inicial
                             ;do bloco de código do programa

       mov ecx, End-Start+1  ;move para o registrador ECX o tamanho do
                             ;programa, definindo assim a faixa de memória
                             ;que será vasculhada

       mov eax, 0BCh
       add eax, 10h          ;carrega no registrador EAX o valor 0CCh,
                             ;porém não diretamente, senão seria
                             ;identificado como um breakpoint

       repnz scasb           ;procura pela seqüência 0CCh
       test ecx,ecx          ;verifica se ECX = 0

                             ;se diferente, breakpoint encontrado
       jne found             ;e executa o salto

       lea eax, Imports+2    ;carrega a primeira função do IAT no
                             ;registrador EAX
```

```
        mov dword ptr [current_import], eax     ;salva o ponteiro da função
                                                ;em EAX no ponteiro
                                                ;current_import
start_searching:

        mov eax, dword ptr [current_import]     ;copia o valor do ponteiro
                                                ;current_import para EAX

        cmp dword ptr [eax],0 ;verifica se EAX = 0

                              ;se for, então significa que este é o final da
                              ;da tabela IAT, e
        je notfound           ;pula para finalização não encontrada

        mov ecx, [eax]        ;lê o endereço indicado em EAX, que é o
                              ;endereço inicial da função, para o
                              ;registrador ECX

        mov ecx, [ecx]        ;lê o conteúdo do endereço inicial da função,
                              ;(definido em ECX) para o próprio registrador
                              ;ECX

        mov eax, 0BCh
        add eax, 10h          ;move novamente o valor 0CCh à EAX

        cmp byte ptr [ecx],al ;verifica então se o primeiro caractere da
                              ;função é então a int 3h (0CCh)

        je found              ;se for, executa salto para finalização found

        add [current_import],6;se não, atualiza o valor de current_import,
                              ;apontando para a próxima função da IAT

        jmp start_searching   ;efetua o salto de volta ao inicio da busca

found:

        mov eax, 1            ;salva o valor 1 no registrador EAX, indicando
                              ;que foi encontrado um depurador

        jmp farther           ;salta para o bloco de finalização farther

notfound:

        xor eax, eax          ;zera o valor de EAX, indicando que não foi
                              ;encontrado nenhum depurador
farther:

        push eax              ;salva o resultado da procura que esta EAX no
                              ;stack

;---------------------------------------------------------------------------
;Restaura o serviço SEH antigo
;---------------------------------------------------------------------------
        push dword ptr [previous]
        call SetUnhandledExceptionFilter
;---------------------------------------------------------------------------

        pop eax               ;carrega do stack o resultado da procura,
                              ;salvando-o em EAX
```

```
       test eax, eax         ;verifica se EAX = 0

       jnz jump              ;se EAX diferente de zero, salta para o bloco
                             ;de finalização jump, indicando que a
                             ;seqüência foi encontrada

continue:                    ;exibe mensagem encontrado

       call MessageBoxA, 0, offset message2, offset message1, 0
       call ExitProcess, -1

jump:                        ;exibe mensagem não encontrado

       call MessageBoxA, 0, offset message3, offset message1, 0
       call ExitProcess, -1

error:                       ;restaura o serviço SEH antigo em caso de erro

       mov esp, [delayESP]
       push offset continue
       ret

End:

Imports:                     ;este label necessita estar no fim do programa
                             ;pois a IAT começa apos esta seção
end Start
```

Figura 5.3 – Código para detecção de um depurador via busca da instrução *Int 3h*

Fonte: Imagem do autor, adaptado de Cerven, 2002, p. 121

5.1.4 Detectando o depurador *NuMega® SoftICE / Compuware® DriverStudio*

Conforme abordado, algumas técnicas de proteção são específicas, protegendo ou detectando apenas um tipo específico de ferramenta ou até mesmo uma versão específica de um determinado produto.

O depurador *NuMega SoftICE* – que, mesmo depois de ser comprado pela empresa *Compuware* (www.compuware.com), ter seu nome mudado para *DriverStudio* e ter sido oficialmente descontinuado em Abril de 2006 (PIETREK, 2006) – é, ainda hoje, uma das ferramentas mais conhecidas no universo *cracker*.

Premiado e reconhecido pelas principais autoridades em engenharia reversa de *software*, este produto é um poderoso depurador em modo *kernel*, sendo utilizado pela grande maioria de *crackers*. (CERVEN, 2002)

Devido à sua vasta utilização, nada melhor que proteger os *software*s desenvolvidos deste depurador. Assim, diferentemente dos métodos anteriormente abordados, serão analisadas formas e técnicas de proteção específicas ao *SoftICE*.

5.1.4.1 Detectando por busca em memória

Apesar de um tanto obsoleta, visto que esta técnica funciona apenas nos *Windows* 9x, este método será utilizado para demonstrar outra das várias maneiras possíveis de se detectar depuradores num sistema.

Esta técnica lembra bastante a técnica de detecção de depuradores via procura da instrução `int 3h`, no entanto é buscado o texto "WINICE.BR" na memória do aplicativo.

Embora tenha o revés de funcionar apenas nos *Windows* 9x, esta técnica é muito interessante, pois pode ser facilmente camuflada dentro de um aplicativo, já que esta não se utiliza de nenhuma chamada *API* nem de instruções de interrupção. Esta vantagem torna quase impossível a detecção, sendo apenas detectável numa análise minuciosa do código binário/desassemblado. (CERVEN, 2002)

```
.386
.MODEL FLAT,STDCALL
locals
jumps
UNICODE=0
include w32.inc
Extrn SetUnhandledExceptionFilter : PROC

.data

message1 db "Detecção SoftICE por busca em memória",0
message2 db "SoftICE não encontrado",0
message3 db "SoftICE encontrado",0

delayESP dd 0           ;salva o registrador ESP aqui
previous dd 0           ;o registrador ESP salvará o endereço antigo do
                        ;serviço SEH aqui

.code

Start:

;-----------------------------------------------------------------------
;Configura o SEH - em caso de erro
;-----------------------------------------------------------------------
       mov [delayESP],esp
       push offset error
       call SetUnhandledExceptionFilter
       mov [previous], eax
;------------------------------------------------------------------------
```

```
        mov al, "W"             ;move o caractere W para o registrador AL,
                                ;já que, por motivos de desempenho, esta
                                ;rotina iniciará procurando o caractere W e,
                                ;quando encontrado, comparará os caracteres
                                ;seguintes, ate encontrar o texto "WINICE.BR"
                                ;completo

        lea edi, Start          ;carrega no registrador EDI o endereço inicial
                                ;do bloco de código do programa

        mov ecx, End-Start+1    ;move para o registrador ECX o tamanho do
                                ;programa, definindo assim a faixa de memória
                                ;que será vasculhada

more:

        repnz SCASB             ;procura pelo caractere "W" na memória

        jecxz notfound          ;caso o caractere W não tenha sido encontrado,
                                ;efetua o salto para a finalização notfound,
                                ;indicando que o SoftICE não foi localizado

        cmp dword ptr [edi], "INIC"  ;caso encontrado, verifica se os quatro
                                     ;caracteres seguintes são a seqüência
                                     ;"INIC", sub-parte do texto "WINICE.BR"

        jz found1               ;se coincidir, salta para o bloco de procura
                                ;dos caracteres seguintes

        jmp more                ;se não, procura pelo próximo caractere W

found1:

        add edi, 4              ;move o ponteiro em EDI em +4 bytes, indicando
                                ;que a seqüência "INIC" foi encontrada,
                                ;posicionando o ponteiro para os caracteres
                                ;seguintes

        cmp dword ptr [edi], "RB.E"  ;compara os caracteres seguintes,
                                     ;verificando se o conteúdo é a sub-parte
                                     ;final "E.RB"

        jnz more                ;se não coincidir, reinicia todo procura,
                                ;localizando novamente o caractere W a partir
                                ;do endereço de memória atual

        push word ptr 1         ;se coincidir, salva no stack o caractere 1,
                                ;indicando que a busca foi bem sucedida e a
                                ;seqüência "WINICE.BR" foi encontrada

        jmp short found         ;salta para o bloco de finalização found

notfound:

        push word ptr 0         ;se não encontrado, salva no stack o caractere
                                ;0, indicando que a seqüência "WINICE.BR" não
                                ;foi encontrada

found:
```

```
;---------------------------------------------------------------------------
;Restaura o serviço SEH antigo
;---------------------------------------------------------------------------
       push dword ptr [previous]
       call SetUnhandledExceptionFilter
;---------------------------------------------------------------------------

       pop ax                 ;carrega do stack o resultado da procura,
                              ;salvando-o no registrador AX

       jnz jump               ;se diferente, exibe mensagem não encontrado

       test ax,ax             ;verifica se AX = 0

       jnz jump               ;se AX diferente de zero, salta para o bloco
                              ;de finalização jump, indicando que o SoftICE
                              ;foi encontrado

continue:                     ;exibe mensagem encontrado

       call MessageBoxA, 0, offset message2, offset message1, 0
       call ExitProcess, -1

jump:                         ;exibe mensagem não encontrado

       call MessageBoxA, 0, offset message3, offset message1, 0
       call ExitProcess, -1

error:                        ;restaura o serviço SEH antigo em caso de erro

       mov esp, [delayESP]
       push offset continue
       ret

End:

end Start
```

Figura 5.4 – Código para detecção do depurador *SoftICE* via busca em memória

Fonte: Imagem do autor, adaptado de Cerven, 2002, p. 78

5.1.4.2 Detectando pela abertura dos *drivers SICE* e *NTICE*

De acordo com Cerven (2002), esta é a técnica mais utilizada na detecção do depurador *SoftICE*, devido à sua eficiência, simplicidade e facilidade na implementação, mesmo em linguagens de alto nível.

O princípio deste método de detecção é simples: o sistema deverá tentar abrir um arquivo – para suposta edição – com os mesmos nomes dos arquivos do *SoftICE*. Se os *drivers* do *SoftICE* estiverem ativos na memória, o valor de retorno da função (*handle*) será diferente de *INVALID_HANDLE_VALUE*, indicando a presença do depurador. (CERVEN, 2002)

Apenas deverá ser observada a presença do parâmetro *OPEN_EXISTING* (parâmetro que define e permite a abertura de arquivos mesmo quando estes já estiverem abertos ou em edição) na função de abertura do arquivo (*CreateFileA* da biblioteca *kernel32.dll* do *Windows*). Do contrário, esta técnica não funcionará corretamente. Cerven (2002) destaca o exemplo abaixo para melhor compreensão.

```
#include <stdio.h>
#define WIN32_LEAN_AND_MEAN
#include <windows.h>

BOOL IsSoftIce95Loaded( )
{
       HANDLE hFile;

       // Arquivo "\\.\SICE" para o Windows 9x
       hFile = CreateFile(    "\\\\.\\SICE",
                              GENERIC_READ | GENERIC_WRITE,
                              FILE_SHARE_READ | FILE_SHARE_WRITE.
                              NULL,
                              OPEN_EXISTING,
                              FILE_ATTRIBUTE_NORMAL,
                              NULL);

       if( hFile != INVALID_HANDLE_VALUE )
       {
            CloseHandle(hFile);
            return TRUE;
       }

       return FALSE;
}


BOOL IsSoftIceNTLoaded()
{
       HANDLE hFile;

       // Arquivo "\\.\NTICE" para o Windows NT
       hFile = CreateFile(    "\\\\.\\NTICE",
                              GENERIC_READ | GENERIC_WRITE,
                              FILE_SHARE_READ | FILE_SHARE_WRITE,
                              NULL,
                              OPEN_EXISTING,
                              FILE_ATTRIBUTE_NORMAL,
                              NULL);

       if( hFile != INVALID_HANDLE_VALUE )
       {
            CloseHandle(hFile);
            return TRUE;
       }


       return FALSE;
}
```

```
int main(void)
{

       if( IsSoftIce95Loaded() )
            printf("SoftICE para Windows 9x detectado na memória.\n");

       else if( IsSoftIceNTLoaded() )
            printf("SoftICE para Windows NT detectado na memória.\n");

       else
            printf("SoftICE não encontrado.\n");

       return 0;

}
```

Figura 5.5 – Código para detecção do depurador *SoftICE* via abertura de seus *drivers*

Fonte: Imagem do autor, adaptado de Cerven, 2002, p. 79

5.1.4.3 Detecção via registro do *Windows* e pasta de instalação

A técnica a seguir se utiliza de consultas à chaves de registro e localização de pastas no *Windows* para identificar a possível existência do depurador *SoftICE* num sistema.

Esta técnica serve mais para demonstrar as diversas possibilidades existentes na identificação de depuradores do que propriamente proteger um aplicativo. Mesmo sendo facilmente rastreável – tanto por monitores de arquivos quanto por monitores de registro –, e de fácil desabilitação, sua aplicação é extremamente segura e, em casos de baixa importância, pode ser, seguramente, utilizada. (CERVEN, 2002)

Como a grande maioria dos aplicativos desenvolvidos para a plataforma *Windows*, o *SoftICE* utiliza-se do registro do *Windows* para gravar diversas informações, como parâmetros de execução, dados de versão e informações de configuração e instalação, estes últimos úteis para definição da existência ou não do depurador na máquina.

Por padrão, o *SoftICE* grava estas informações dentro das chaves de registro (CERVEN, 2002):

- *HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Uninstall\SoftICE*
- *HKEY_LOCAL_MACHINE\Software\NuMega\SoftICE*

Tendo em mãos essas informações, a detecção é bastante simples: basta verificar a existência de uma destas chaves no registro do *Windows*, procedimento bastante simples em uma linguagem de alto nível. Além disto, tende a funcionar em todas as versões do *Windows*, devendo apenas ser observadas possíveis modificações em versões futuras do *SoftICE*.

5.2 Anti-desassemblagem

Enganar desassembladores no intuito de prevenir ou inibir reversores não é considerado particularmente como uma maneira robusta de evitar reversão, porém é um recurso vastamente utilizado, e de fácil aplicação. (EILAM, 2005)

Em arquiteturas de processadores que suportam instruções de tamanho variável, como acontece nos processadores *IA-32*, é possível ludibriar os desassembladores de maneira a tratar erroneamente um bloco de dados inválido, sendo assim interpretados como instruções do programa, explica Eilam (2005). Isto causará um erro de sincronia na desassemblagem, comprometendo todo código até o final do arquivo, sendo apenas 'consertado' após a resincronização das instruções.

Conforme citado anteriormente, os desassembladores utilizam dois métodos para conversão do código binário para a linguagem *Assembly*: linear e recursiva. O quadro 5.1 demonstra os métodos utilizados pelos depuradores e desassembladores mais conhecidos.

Quadro 5.1 – Depuradores e seus métodos de desassemblagem

Depurador/Desassemblador	Método de desassemblagem
OllyDbg	Conversão recursive
NuMega SoftICE	Conversão linear
Microsoft WinDbg	Conversão linear
IDA Pro	Conversão recursive
PEBrowse Professional	Conversão recursive

Fonte: Adaptado de Eilam, 2005, p. 337

Para melhor entendimento, serão demonstrados mais detalhadamente cada algoritmo e como evitá-los.

5.2.1 Conversão Linear (*Linear Sweep*)

Os desassembladores que traduzem os programas via conversão linear podem ser facilmente trapaceados. Como cada instrução é traduzida seqüencialmente, basta inserir *bytes* inválidos em partes inacessíveis (do ponto de vista fluxo-gráfico) para comprometer toda tradução do código, explica Eilam (2005).

A figura 5.6 demonstra um programa onde é gerado um *byte* inválido no meio do código, com a finalidade de evitar a desassemblagem:

```
_asm
{
       jmp Seguinte     ;salto para o bloco seguinte

       _emit 0x0f       ;instrução para geração do byte 0F no programa.
                        ;esta instrução, no entanto, jamais será executada,
                        ;pois a instrução anterior é um salto incondicional
Seguinte:

       mov eax, [Variavel1]
       push eax
       call AFunction
}
```

Figura 5.6 – Código para inserção de um *byte* inválido ao código

Fonte: Imagem do autor, adaptado de Eilam, 2005, p. 337

Compilando este código e desassemblando-o no depurador *NuMega SoftICE*, pode-se notar que houve erros de interpretação. O *byte* `0F` foi considerado parte de uma instrução, comprometendo todo resto da tradução:

```
001B:0040101D JMP 00401020
001B:0040101F JNP E8910C6A
001B:00401025 XLAT
001B:00401026 INVALID
001B:00401028 JMP FAR [EAX-24]
001B:0040102B PUSHAD
001B:0040102C INC EAX
```

Figura 5.7 – Resultado da desassemblagem do código pelo depurador *SoftICE*

Fonte: Eilam, 2005, p. 338

Em contraste, quando analisado por um desassemblador recursivo, como o *OllyDbg*, pode-se verificar a tradução correta do código. O código `0F` não é desassemblado e a instrução que o segue é perfeitamente convertida:

```
0040101D EB 01          JMP SHORT disasmtest.00401020
0040101F 0F             DB 0F
```

```
00401020 8B45FC        MOV EAX,DWORD PTR SS:[EBP-4]
00401023 50            PUSH EAX
00401024 E8D7FFFFFF    CALL disasmtest.401000
```

Figura 5.8 – Resultado da desassemblagem do código pelo depurador *OllyDbg*

Fonte: Eilam, 2005, p. 337

Como se pôde notar, o *SoftICE*, como qualquer desassemblador linear, é confundido pela inserção de blocos de *bytes* inválidos, mesmo que inalcançáveis pelo fluxo do programa.

5.2.2 Conversão Recursiva (*Recursive Traversal*)

Esta técnica pode ser muito eficaz para evitar reversores, porém não pode ser considerada como solução definitiva, afirma Eilam (2005). Depuradores ou desassembladores que utilizam da conversão recursiva, normalmente não se limitam a proceder conforme rege o algoritmo, visto que, normalmente, estas ferramentas possuem uma gama extra de mecanismos para combater estas técnicas.

De qualquer maneira, serão examinadas técnicas que confundem e tornam quase impossível a detecção antecipada destes blocos de *bytes* inválidos. Para isto, ao invés de saltos incondicionais explícitos serão utilizados predicados opacos, que são ramificações condicionais que aparentam considerar todas as possibilidades do sistema, no entanto sendo, também, essencialmente incondicionais. (EILAM, 2005)

Como em qualquer ramificação condicional dentro de um sistema, o fluxo do programa sempre se divide em dois caminhos. Neste caso, um levará ao código real e o outro ao bloco de *bytes* inválidos. A figura 5.9 ilustra um predicado opaco onde a condição será sempre falsa, enquanto a figura 5.10 ilustra um predicado opaco onde a condição será sempre verdadeira.

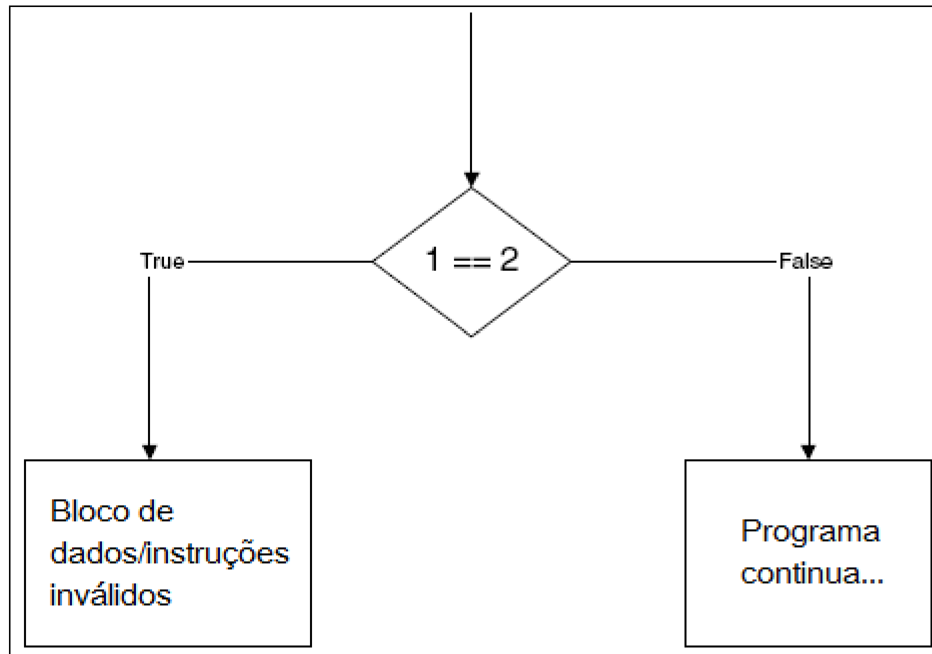


Figura 5.9 – Predicado opaco falso

Fonte: Imagem do autor, adaptado de Eilam, 2005, p. 339

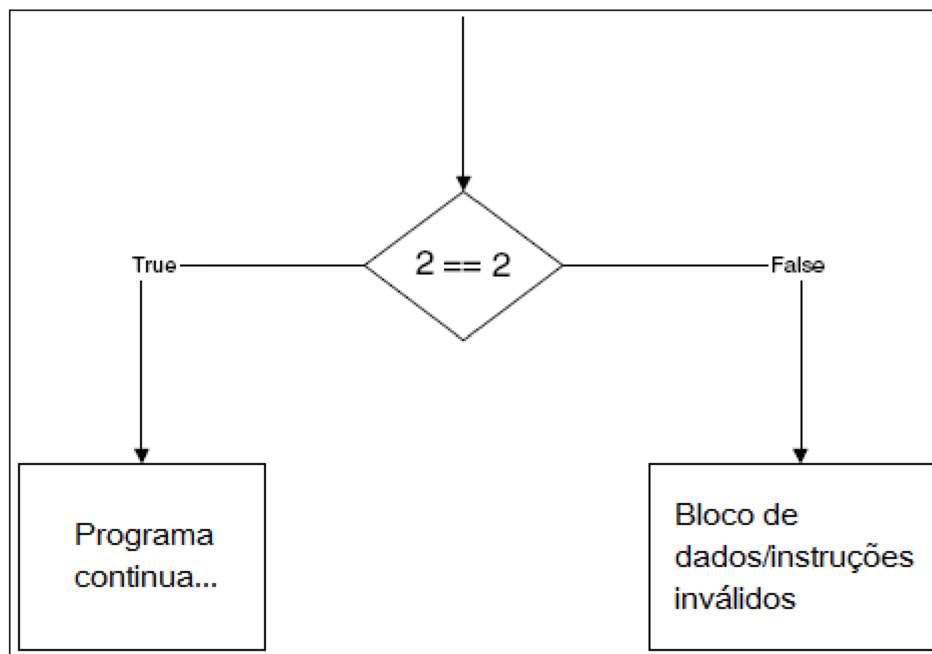


Figura 5.10 – Predicado opaco verdadeiro

Fonte: Imagem do autor, adaptado de Eilam, 2005, p. 339

Considerando o exemplo da figura 5.11 – similar ao utilizado no exemplo da conversão linear, exceto pela troca da instrução do salto incondicional pelas instruções de comparação e salto condicional –, será analisado a conversão de alguns desassembladores e depuradores:

```
_asm
{
       mov eax, 2
       cmp eax, 2
       je Seguinte
       _emit 0xf
Seguinte:
       mov eax, [Variavel1]
       push eax
       call AFuncao
}
```

Figura 5.11 – Código exemplo utilizado para testes de desassemblagem

Fonte: Imagem do autor, adaptado de Eilam, 2005, p. 339

O predicado opaco do exemplo da figura 5.11, simplesmente move o valor 2 para o registrador `EAX`, comparando-o e verificando se este corresponde a 2. Neste caso é possível, logicamente, notar que esta condição será sempre verdadeira, porém assim estar-se-á evitando o salto incondicional explícito.

Desassemblando este código pelo depurador *IDA Pro*, temos o seguinte resultado:

```
.text:00401031 mov eax, 2
.text:00401036 cmp eax, 2
.text:00401039 jz short near ptr loc_40103B+1
.text:0040103B
.text:0040103B loc_40103B: ; CODE XREF: .text:00401039 _j
.text:0040103B jnp near ptr 0E8910886h
.text:00401041 mov ebx, 68FFFFFFh
.text:00401046 fsub qword ptr [eax+40h]
.text:00401049 add al, ch
.text:0040104B add eax, [eax]
```

Figura 5.12 – Resultado da desassemblagem do código exemplo pelo depurador *IDA PRO*

Fonte: Eilam, 2005, p. 339

Conforme demonstrado, pôde perceber-se que *IDA Pro* não conseguiu interpretar o código corretamente, caindo na armadilha. Em contrapartida, o depurador *OllyDbg* reproduziu o código da seguinte maneira:

```
00401031 . B802000000   MOV EAX,2
00401036 . 83F802       CMP EAX,2
00401039 . 7401         JE SHORT compiler.0040103C
0040103B   0F           DB 0F
0040103C > 8B45F8       MOV EAX,DWORD PTR SS:[EBP-8]
0040103F . 50           PUSH EAX
00401040   E8BBFFFFFF   CALL compiler.main
```

Figura 5.13 – Resultado da desassemblagem do código exemplo pelo depurador *OllyDbg*

Fonte: Eilam, 2005, p. 339

Nota-se como o *OllyDbg* claramente ignorou o *byte* `0F` inválido, utilizando-se do salto condicional para efetuar a conversão, trazendo assim a conversão correta do programa. Visto este recurso, pode-se concluir que o *OllyDbg* possui artimanhas internas de prevenção de técnicas anti-desassemblagem. No entanto, ainda analisando o código acima, nota-se a visível possibilidade de tratamento deste bloco inválido, já que, se consideradas as instruções que precedem o salto, pode-se afirmar que o salto será sempre executado. (EILAM, 2005)

Aprimorando esta idéia, pode-se reescrever o código acima de maneira menos legível ou interpretável pela máquina. Ao invés de se utilizar um salto para o endereço do bloco real de código, deve-se utilizar um salto indireto, cujo valor (endereço) seja proveniente de um registrador. O exemplo da figura 5.14 demonstra a idéia:

```
_asm
{
       mov eax, 2
       cmp eax, 3
       je Lixo
       mov eax, Seguinte
       jmp eax
Lixo:
       _emit 0xf
Seguinte:
       mov eax, [Variavel1]
       push eax
       call AFuncao
}
```

Figura 5.14 – Código exemplo anti-desassemblagem utilizando saltos indiretos

Fonte: Imagem do Autor, adaptado de Eilam, 2005, p. 342

Conforme pesquisa realizada por Eilam (2005), o código acima confunde todos os desassembladores, lineares ou recursivos. A razão é simples: neste caso, o desassemblador não tem como interpretar que a seqüência “`mov eax, Seguinte`” e “`jmp eax`” é equivalente ao salto “`jmp Seguinte`”, fazendo assim com que nem seja tentada a desassemblagem a partir dos ponteiros dos saltos.

No entanto, Eilam (2005) comenta que a desvantagem da utilização destas técnicas é justamente o pressuposto de que a quase totalidade dos desassembladores para o *Windows* são, relativamente, burros o suficiente para permitir que sejam burlados.

Uma simples análise, um pouco mais aprofundada deste caso, seria suficiente para verificar que o salto realizado é um mero predicado opaco falso. A aplicação desta técnica é uma medida temporal, à mercê de novos desassembladores e suas contra-medidas.

5.3 Criptografia de código

A criptografia de código é um método muito comum na prevenção da análise estática de código. Este método consiste essencialmente na criptografia de todo bloco de código de um programa pronto (já compilado), adicionado a uma rotina de descriptografia (esta descompilada) para restauração deste código ao seu formato original. (EILAM, 2005)

O *software* pronto para distribuição é pego, e sobre ele, é aplicado este processo, criptografando o código de tal maneira que suas instruções não sejam diretamente legíveis por um reversor. Assim, na inicialização deste programa, todo bloco de código criptografado é carregado em memória e, após, é chamada a rotina de descriptografia, que tornará todas as instruções do código de volta ao seu estado original.

No entanto, Eilam (2005) lamenta que esta técnica, assim como todas as técnicas de proteção, é bastante frágil quando utilizada para combater reversores especializados, criando nada além de uma inconveniência no processo de quebra, já que toda informação necessária, incluindo a lógica de descriptografia e, principalmente, a chave de criptografia, está embutida no aplicativo.

De qualquer maneira esta técnica é uma boa opção, pois, para reversores não tão experientes, a impossibilidade de se fazer a análise estática torna o processo extremamente complicado, permitindo este estudo apenas em tempo de execução.

Existem diversas ferramentas comerciais para criptografia de *software* – a mais conhecida é a *ASProtect*, encontrada no endereço <http://www.aspack.com>–, para facilitar a vida do programador. No entanto, na maioria destas ferramentas, existe também o programa *cracker*, chamado de *unpacker*, para descriptografia automática do aplicativo protegido.

Este acaba sendo outro problema, já que estes *unpackers* são programados de tal maneira a se familiarizarem com os métodos de criptografia, sabendo onde e como adquirir as chaves de descriptografia, automatizando todo processo.

Assim, Eilam (2005) ressalta a importância do combate destes *unpackers* quando utilizada esta técnica. Métodos metamórficos, ocultação da chave de criptografia e geradores de chave automatizados (*key generators*) são alguns dos mecanismos sugeridos para aprimoramento da técnica e proteção contra *unpackers*.

5.4 Ofuscação de código

Como se pôde notar, as técnicas anti-reversão servem para impor obstáculos no processo de quebra e dificultar a aquisição, leitura e interpretação do código de um programa e suas instruções.

A ofuscação de código aplica literalmente este conceito, transformando o código de um aplicativo e dificultando-o de tal maneira que torne sua interpretação humana quase impossível, porém, mantendo todas suas funcionalidades, define Szor (2005).

Esta transformação não envolve nenhuma técnica de exploração específica, dando-se apenas pelo rearranjamento de instruções, quebra de paradigmas de programação, inserção de instruções irrelevantes, além de diversos outros recursos deste gênero, todos no intuito de tornar o funcionamento do *software* muito mais complexo. (EILAM, 2005)

No entanto, apesar de toda complexidade introduzida por esta enxurrada de instruções adicionais, o mais importante é fazer com que estas funções sejam de difícil remoção. Como a maioria destas transformações adiciona instruções que não produzem dados significantes ao programa, é possível, através de uma intensa análise algorítmica, detectar e eliminar estas instruções, criando assim ferramentas desofuscadoras.

A aplicação de qualquer tipo de método desta técnica, até mesmo na reorganização de código, tipicamente implica num custo. Este pode ser em forma de grande amontoamento de dados extras ao código, execução mais lenta em certos trechos do programa ou aumento do consumo de memória. É importante considerar cada fator e cada cenário, visando sempre a qualidade do sistema.

5.4.1 Transformações no Controle de Fluxo

Eilam (2005) categoriza as transformações no controle de fluxo em:

- Transformações computacionais;
- Transformações conjunturais;
- Transformações de ordenação.

Transformações computacionais almejam a redução de legibilidade do código modificando a estrutura do controle de fluxo original do programa na finalidade de criar códigos funcionalmente iguais, porém muito mais difíceis de serem traduzidos de volta à linguagem de alto nível. Isto pode ser conseguido pela remoção de informações de controle de fluxo ou pela adição de novas instruções.

Nas transformações conjunturais são destruídas as estruturas de alto nível via quebra das abstrações de alto nível utilizadas pelos programadores, tornando a organização destas menos interpretáveis.

A Transformação de ordenação é a menos impactante dos três métodos, pois esta meramente randomiza, o maior número de vezes possível, a seqüência das instruções para diminuir, também, a legibilidade.

5.4.2 Transformações no Tratamento de Dados

Diferentemente da ofuscação de controle de fluxo, as transformações de dados focam os dados e suas estruturas ao invés do funcionamento e fluxograma de um aplicativo. Isto é definitivamente importante, pois o conhecimento do *layout* das estruturas de dados é essencial para o entendimento do programa e como o mesmo funciona.

Um dos exemplos deste método é a modificação da codificação de uma variável. Uma destas aplicações pode ser, por exemplo, a alteração na contagem de um contador inteiro, deslocando seu valor um *bit* à esquerda. Isto fará com que os valores sejam dobrados, e para cada passo de um *loop* `for`, seja incrementado 2 ao invés de 1. A figura 5.15 ilustra esta idéia:

```
//instrução original
for (int i=1; i < 100; i++)

//instrução transformada
for (int i=2; i < 200; i += 2)
```

Figura 5.15 – Ofuscação de código via transformações no tratamento de dados

Fonte: Imagem do Autor, adaptado de Eilam, 2005, p. 356

Nota-se que ambas as instruções exercerão o mesmo papel, no entanto, de maneiras diferentes. Analisando este código, do ponto de vista alto nível, não se repara diferença, porém, em baixo nível, essa escrita fora de padrão pode confundir reversores.

Logicamente este é um exemplo trivial, meramente ilustrativo. Serve apenas para criar a idéia de como se pode utilizar esta técnica no desenvolvimento de aplicações, tanto é que, se utilizada em alto nível, é bem provável que não surta nenhum efeito, devido a otimização de código realizada pelos compiladores.

Outro exemplo é a reestruturação de estruturas como vetores ou *records/structs*, de modo que preservem suas funções originais, porém confundindo reversores. O agrupamento de vetores ou *record/struct* clássicos para geração de um terceiro vetor maior, a quebra de um vetor em elementos menores ou a alteração do formato e/ou tamanho de um elementos de um *record/struct* já são o suficiente para desvincular o conhecimento do reversor com sua funcionalidade nominal.

5.5 Anti-patching

Um dos objetivos de um reversor é a aquisição ilícita do direito de execução de um *software*, usufruindo ilegalmente das funções que este proporciona; é mais que um excelente motivo para se proteger um programa de *patching*, técnica necessária para se burlar qualquer proteção anti-cópia de um *software*.

Como não pode ser criado um mapa completo das instruções e estados internos originais de um *software*, até porque esta lógica é recursiva e infinita, não se pode ter um guia de consulta para verificar quais partes do programa foram alteradas.

Para isto existe o cálculo de *CRC*, que é uma fórmula matemática que calcula, de acordo com cada *byte* de um fornecido bloco de dados, um valor identificador para o tal. Este valor considera cada *byte* do bloco, retornando sempre o mesmo valor identificador enquanto nenhum *byte* for alterado. (EILAM, 2005)

Com esta funcionalidade, pode-se utilizar este método para geração de identificadores internos de blocos de instrução, para assim detectar possíveis alterações nos aplicativos.

5.6 Anti-descarregamento de memória

O descarregamento de memória é uma técnica utilizada para burlar programas protegidos por criptografia ou *packers*, criando uma imagem binária (em disco) do conteúdo carregado pela memória do *software*.

Existem diversas ferramentas para isto, a mais conhecida delas chamada de *ProcDump*. Esta ferramenta se destaca pela fácil utilização e pela extensa gama de pré-configurações de *unpackers*, combatendo assim as principais proteções comerciais.

Como não foi encontrado nenhum método genérico para combate anti-descarregamento de memória, será demonstrado uma maneira de combater a mais famosa destas ferramentas: o *ProcDump*.

A implementação deste método consiste em alterar, durante o carregamento do aplicativo, o tamanho do cabeçalho *PE* na memória. Isto fará com que o *ProcDump* tente ler seções inexistentes e não alocadas, causando com isto o erro. (CERVEN, 2002) A figura 5.16 exemplifica o processo:

```
.386
.MODEL FLAT,STDCALL
locals
jumps
UNICODE=0
include w32.inc

Extrn SetUnhandledExceptionFilter : PROC

.data
message1 db "Técnica Anti-ProcDump",0
message2 db "Tamanho do cabeçalho PE aumentado",0

delayESP dd 0           ;salva o registrador ESP aqui
previous dd 0           ;o registrador ESP salvará o endereço antigo do
                        ;serviço SEH aqui

.code

Start:

;--------------------------------------------------------------------------
;Configura o SEH - em caso de erro
```

```
;------------------------------------------------------------------------
       mov [delayESP],esp
       push offset error
       call SetUnhandledExceptionFilter
       mov [previous], eax
;-------------------------------------------------------------------------

       mov eax, fs:[30h]      ;lê a versão do Windows
       test eax,eax           ;testa versão
       js found_win9x         ;efetua o salto se um Windows 9x encontrado

found_winNT:                  ;procedimento para o Windows NT

       mov eax,[eax+0ch]
       mov eax,[eax+0ch]
       add dword ptr [eax+20h], 3000h    ;aumenta o tamanho do cabeçalho
       jmp end                           ;salta para o final

found_win9x:                  ;procedimento para o Windows 9x

       push 0
       call GetModuleHandleA       ;localiza o handle do módulo
       test edx,edx
       jns end                     ;em caso de erro, salta para o final

       cmp dword ptr [edx+08], -1  ;testa se o valor é igual a -1

       jne end                     ;se não for, um erro encontrado e
                                   ;salta para o final

       mov edx, [edx+4]            ;localiza o endereço do cabeçalho PE

       add dword ptr [edx+50h],3000 ;aumenta o tamanho do cabeçalho PE,
                                    ;alterando o item SizeOfImage

end:

;-------------------------------------------------------------------------
;Restaura o serviço SEH antigo
;-------------------------------------------------------------------------
       push dword ptr [previous]
       call SetUnhandledExceptionFilter
;-------------------------------------------------------------------------

continue:

       call MessageBoxA,0, offset message2, offset message1,0
       call ExitProcess, -1

error:      ;restaura o serviço SEH antigo em caso de erro

       mov esp, [delayESP]
       push offset continue
       ret

ends

end Start
```

Figura 5.16 – Código para proteção anti-descarregamento de memória via *ProcDump*

Fonte: Imagem do Autor, adaptado de Cerven, 2002, p. 132

5.7 Edição do Cabeçalho *PE*

Complementando este capítulo, serão demonstrados métodos que não consistem na escrita de armadilhas ou quaisquer adição de código ao programa. Serão abordados apenas métodos que alteram o cabeçalho de um aplicativo *Win32/PE*, fazendo com que depuradores, desassembladores ou descarregadores de memória se percam e não consigam obter e/ou extrair o conteúdo para qual foram projetados.

Para facilitar o entendimento, Brulez (2005) disponibilizou o exemplo da figura 5.17, contendo o cabeçalho opcional de um aplicativo fictício já editado para combater algumas ferramentas:

```
Cabeçalho Opcional

       Magic:                        0x010B (HDR32_MAGIC)
       MajorLinkerVersion:           0x02
       MinorLinkerVersion:           0x19 -> 2.25
       SizeOfCode:                   0x00000200
       SizeOfInitializedData:        0x00045400
       SizeOfUninitializedData:      0x00000000
       AddressOfEntryPoint:          0x00002000
       BaseOfCode:                   0x00001000
       BaseOfData:                   0x00002000
       ImageBase:                    0x00DE0000 <--- Endereço base alterado
       SectionAlignment:             0x00001000
       FileAlignment:                0x00001000
       MajorOperatingSystemVersion: 0x0001
       MinorOperatingSystemVersion: 0x0000 -> 1.00
       MajorImageVersion:            0x0000
       MinorImageVersion:            0x0000 -> 0.00
       MajorSubsystemVersion:        0x0004
       MinorSubsystemVersion:        0x0000 -> 4.00
       Win32VersionValue:            0x00000000
       SizeOfImage:                  0x00049000
       SizeOfHeaders:                0x00001000
       CheckSum:                     0x00000000
       Subsystem:                    0x0003 (WINDOWS_GUI)
       DllCharacteristics:           0x0000
       SizeOfStackReserve:           0x00100000
       SizeOfStackCommit:            0x00002000
       SizeOfHeapReserve:            0x00100000
       SizeOfHeapCommit:             0x00001000
       LoaderFlags:                  0xABDBFFDE <--- Valor inválido
       NumberOfRvaAndSizes:          0xDFFFDDDE <--- Valor inválido
```

Figura 5.17 – Cabeçalho opcional exemplo de proteção via edição de cabeçalho *PE*

Fonte: Imagem do Autor, adaptado de Brulez, 2005, p. 2

5.7.1 Alteração do endereço base do aplicativo

Programas *Win32* têm por padrão (estabelecido pela maioria dos compiladores) o endereço base dos aplicativos igual a `400000h`. Este valor base é utilizado para cálculo de *offset* das instruções, e devido à sua quase universal utilização, a mera alteração desta indexação poderá confundir reversores, já acostumados com este padrão.

Esta não é especificamente uma armadilha anti-reversão, no entanto pode ser considerada mais uma boa maneira de complicar o processo de estudo e de reversão.

5.7.2 Anti-*OllyDbg* e Anti-*SoftICE*

Analisando ainda o cabeçalho acima, pode-se ver que os itens *LoaderFlags* e *NumberOfRvaAndSizes* foram também modificados.

Estas duas alterações fazem com que o *OllyDbg* creia que o arquivo esteja corrompido, fazendo assim com que ele seja rodado sem a instalação de *breakpoints*, fato extremamente perigoso especialmente quando deseja-se depurar *malwares* em seu computador. Já no SoftICE, a alteração do valor contido no *NumberOfRvaAndSizes* faz com que a máquina seja reiniciada. (BRULEZ, 2005)

5.7.3 Alteração da *flag* de características da seção *.code*

Cerven (2002) destaca outro método de edição de cabeçalho *PE*. Se a *flag* de características da seção *.code* for alterada para o valor `C0000040h`, qualquer tentativa de desassemblagem via o depurador *WinDASM* resultará na obtenção de um código incorreto.

Este método também surte efeito quando executado no *SoftICE*, pois o *Symbol Loader* não encontrará o ponto de entrada do aplicativo, rodando-o livremente na memória.

5.8 Aspectos Legais e Ética na aplicação da Engenharia Reversa

A engenharia reversa, como qualquer outra habilidade, pode ser utilizada tanto para o bem quanto para o mal. Assim como ela pode ser utilizada para reconstrução de um código fonte, ela pode ser utilizada para burlar proteções anti-cópias. Isto acaba complicando a discussão dos aspectos legais e éticos de sua aplicação.

Vários contratos de licenciamento de *software* proíbem expressamente a engenharia reversa, numa tentativa de evitar que reversores quebrem as suas rotinas de registro ou até mesmo para proteção do conhecimento empregado e aplicado ao *software*, apesar destas serem judicialmente discutíveis.

Citando a analogia utilizada por Hoglund e McGraw (2006, p. 67), "a proibição total da engenharia reversa seria como uma lei que tornasse ilegal a abertura do capô de um carro para consertá-lo". E como os objetivos de sua aplicação podem ser totalmente distintos, cria-se esta confusão referente à sua legalidade.

Hoglund e McGraw (2006) afirmam não existir atualmente nenhuma lei geral contra a engenharia reversa em especifico. Apesar das leis americanas serem diferentes das leis brasileiras, esta afirmação também se faz válida aqui.

As leis brasileiras não destacam diretamente a engenharia reversa. O que existe são leis de propriedade intelectual de *software* e de direitos autorais, regidas no Brasil pelas leis 9.609/98 e 9.610/98, respectivamente.

Estas leis tratam basicamente dos direitos autorais sobre os programas desenvolvidos, porém não tratando a hipótese do estudo via engenharia reversa. O que existe é abordagem sobre a proteção de uma obra, transferindo e assegurando o crédito do desenvolvimento e direito adquirido do produto aos seus desenvolvedores.

CONCLUSÃO

O estudo do formato e estruturação de aplicativos *Portable Executable* num ambiente *Windows 32 bits* é de grande importância quando se deseja qualificar um *software*. Esse conhecimento permite ao programador aprimorar o desempenho, a eficiência, a estabilidade e aumentar a segurança dos seus sistemas.

A abordagem deste trabalho, ordenado da maneira pela qual foi disposto, visa, à quaisquer interessados que busquem material de iniciação, o ensinamento e compreensão de como realmente funciona um *software*, analisando, via engenharia reversa, como são executadas as instruções de maneira tão profunda que seja possível a percepção de toda a interação que ocorre entre um aplicativo e o processador da máquina.

A conceituação da engenharia reversa junto da análise detalhada dos arquivos de formato *Windows Portable Executable* sob o ambiente *Windows 32 bits*, permitiu elucidar os mecanismos internos e expor a organização de um aplicativo na memória, possibilitando, a partir das ferramentas correta, inspecionar, avaliar e até mesmo sua alterar seus dados em memória.

A escassa base bibliográfica e a dificuldade de obtenção de material de uma fonte centralizada impulsionaram a pesquisa e a compilação dos tópicos deste trabalho, no intuito de ensinar e auxiliar, da maneira mais clara e compreensível, o processo de aprendizagem deste assunto que, por si só, é bastante árduo.

O estudo aprofundado e a dissecação do formato *PE*, além de essencial a compreensão de um todo, permite a visualização da maneira com que são armazenados os dados e como estão dispostos os blocos de código no formato binário de um arquivo. Juntamente disto, a enumeração e descrição das ferramentas de reversão são essenciais para extração destas informações.

Agregados a estes, o trabalho prezou pela demonstração de técnicas que objetivam a blindagem de um *software* via artimanhas anti-reversão, analisados juntamente com diversas técnicas de ataque. A construção linear destes tópicos permite o vislumbre da aplicabilidade de cada um dos conceitos, pois apenas o conhecimento de como são constituídos estes ataques permitirá a elaboração de métodos e armadilhas de segurança re e pró-ativas.

Para trabalhos futuros, sugere-se, com base no conhecimento adquirido a partir deste trabalho, desenvolver uma ferramenta que possa aplicar, sob um *software* compilado no formato *PE*, diversas técnicas de proteção, blindando-o e evitando que sejam necessárias as implementações das mesmas diretamente em seus códigos. Isto facilitará e dispensará a inserção dos códigos de cada uma das técnicas em cada um dos trabalhos realizados pelo desenvolvedor, permitindo ainda a proteção de *softwares* mais antigos, inclusive daqueles que não se possui mais o código-fonte.

REFERÊNCIAS BIBLIOGRÁFICAS

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

BRULEZ, Nicolas. **Anti Reverse Engineering Uncovered**. Code Breakers Journal, v. 2, n. 1, mar. 2005, 27p. Disponível em <http://www.codebreakers-journal.com>. Acesso em 19 out 2005.

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.

DUMMER, Daniel R. **Analisando o Formato de Arquivo Windows Portable Executable (PE)**. In: CONGRESSO ESTADUAL DE INFORMÁTICA E TELECOMUNICAÇÕES, 10., 2005, Cuiabá. **Anais...** Cuiabá: SUCESU-MT, 2005. p. 2-8.

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 1: Basic Architecture. Estados Unidos da América. 1999a. 369p.

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

KASPERSKY, Kris. **Hacker Disassembling Uncovered**, 2003. Estados Unidos da América: A-LIST Publishing, 2003.

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.

ONE, Aleph. **Smashing The Stack For Fun And Profit**. Phrack Magazine, v. 7, n. 49, 11 ago. 1996. Disponível em <http://www.phrack.org/issues.html?issue=49&id=14>. Acesso em 30 out 2006.

PAULA FILHO, Wilson de Pádua. **Engenharia de Software**: Fundamentos, Métodos e Padrões. 2.ed. Rio de Janeiro: Livros Técnicos e Científicos Editora SA, 2003. 602p.

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.

PIETREK, Matt. **Under The Hood**: R.I.P. SoftICE. 2006. Disponível em <http://blogs.msdn.com/matt_pietrek/archive/2006/04/07/570927.aspx>. Acesso em: 13 out 2007.

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

SCAMBRAY, Joel; McCLURE, Stuart; KURTZ, George. **Hackers Expostos**: Segredos e Soluções para Segurança de Redes. 2.ed. São Paulo: Makron Books, 2001. 694p.

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. 2007. 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.